Writing Parallel software

Sébastien Ponce
`sebastien.ponce@cern.ch`

CERN

Thematic CERN School of Computing 2024

Outline

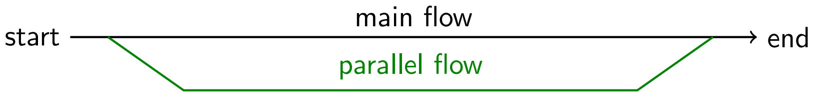
Credits to Danilo Piparo for the original talk in previous tCSCs
... and all the content I shamelessly stole from it

Introduction and expectations

Parallelism

Definition

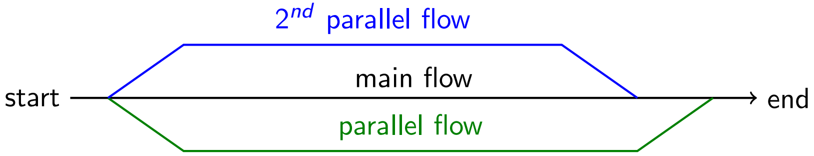
- the ability to make 2 or more things concurrently



Parallelism

Definition

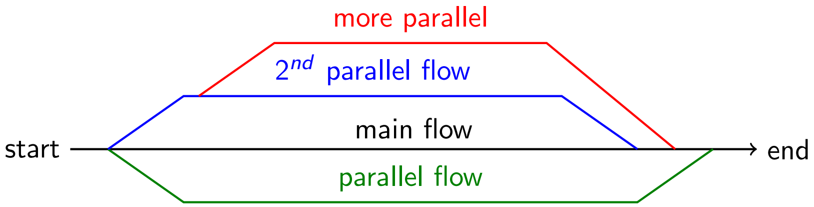
- the ability to make 2 or more things concurrently



Parallelism

Definition

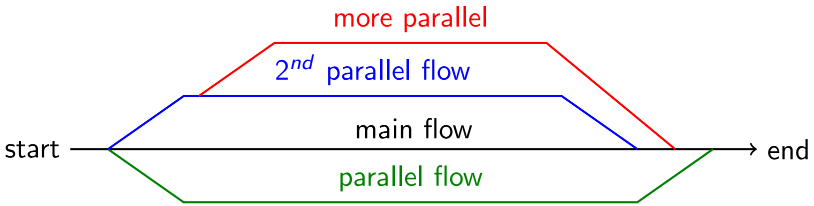
- the ability to make 2 or more things concurrently



Parallelism

Definition

- the ability to make 2 or more things concurrently



concepts

- **concurrency** running 2 things in parallel
- **asynchronicity** disentangle launching a task from getting its result

Different flavors along the history

Multiple levels of parallelism

- machines (forever) / processors (for 40 years) / cores (for 25 years)
- so not really a new topic !

Matching different programming techniques

- RPC, MPI, map/reduce, .../ multi-processing / multi-threading

Why is it trendy now ?

- on many core processors you cannot avoid it
  - individual cores are too slow
- multi-core/multi-threading is tricky
  - and a major source of nasty bugs

Why would you use parallelism ?

From the software point of view

2 main targets

- multiplexing different tasks on the same machine
- making one heavy task faster
    - by running different subparts of it in parallel

From the hardware point of view

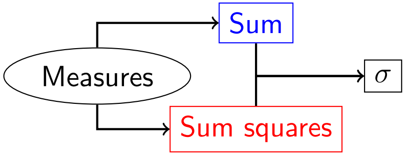
- do not waste available resources
- make best value for money

Net consequence : we need to find things to be done in parallel

Finding parallelism

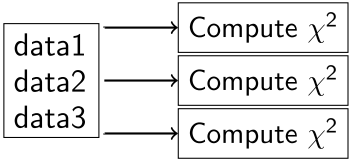
Task parallelism

- act on the work flow
- break the work in tasks
- run several concurrently



Data parallelism

- split data in pieces
- run on several concurrently



Task parallelism

Concept

parallelism achieved through the partition of load into “baskets of work” consumed by a pool of resources.

Implications

- requires splitting the processing into blocks
- and dealing with dependencies between them
- well adapted to data processing frameworks

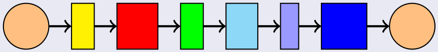
Thread/process pools usually useful

- a pool a workers is created at start
- and reused during all processing
- tasks are mapped to available workers

Task parallelism example

Event reconstruction case

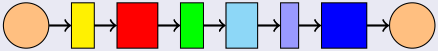
- processing consists in running a bunch of algorithms
- originally as a sequence



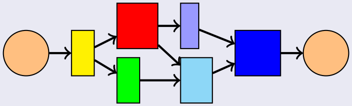
Task parallelism example

Event reconstruction case

- processing consists in running a bunch of algorithms
- originally as a sequence



- now as a dependency graph
- using a thread pool and a scheduler
  - mapping what can be done to available threads.



Data parallelism

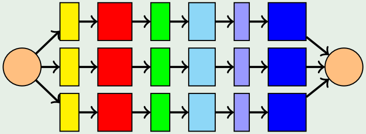
Concept

parallelism achieved through the application of the same transformation to multiple pieces of data

Implications

- requires independent pieces of data
- may have an impact on data structures
- and potentially on memory

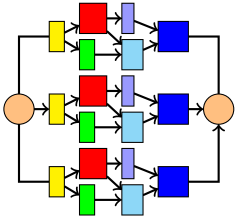
Practically for our event processing example



Overall Strategy

Data or task based parallelism ?

- **Definitely both concurrently**
- Use as much parallelism as you can
- allows to hide dependencies/contention
- and to keep the hardware busy



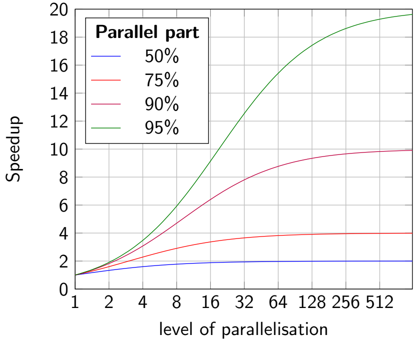
Is it worth it ? Amdahl's law

Applies to a fixed size problem, speedup in terms of time

Maximum achievable speedup

$$Speedup_{max} = \frac{1}{(1-p)+\frac{p}{n}}$$

- p the parallel portion, $p \in [0,1]$
- n the number of workers



"... the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude." - Gene Amdahl - 1967

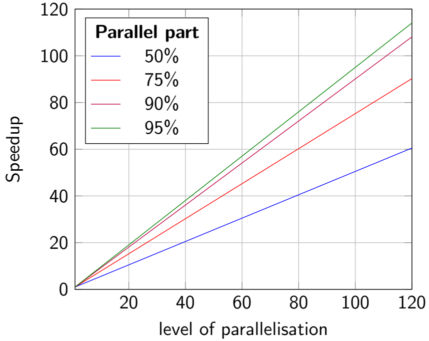
Is it worth it ? Gustafson's law

Applies to a fixed time interval, with more or less work done

Maximum achievable speedup

$$Speedup_{max} = (1 - p) + np$$

- p the parallel portion, $p \in [0, 1]$
- n the number of workers



Conclusion

- there is no limit to the achievable speedup !
- you only need enough work items
    - providing they can run in parallel

Consequences

Amdahl's limitations

- the speedup of a given task has a limit
- in other terms, **latency improvements are constrained**
    - the limit being inversely proportional to the size of the serial part
- **so it is vital to reduce the non parallel part**
    - and even more with large parallelism (manycores)

Consequences

Amdahl's limitations

- the speedup of a given task has a limit
- in other terms, **latency improvements are constrained**
  - the limit being inversely proportional to the size of the serial part
- **so it is vital to reduce the non parallel part**
  - and even more with large parallelism (manycores)

Gustafson's promises

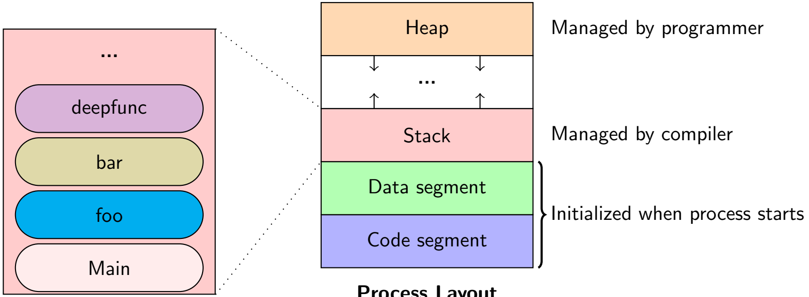
- doing more processing at once will drop limitations
  - by reducing the percentage of serial work overall
  - as it's usually a fixed amount of time
- in other terms, **throughput is not limited**
  - providing you have enough independent items
- **think throughput rather than latency**

Threading, theory and practice

Anatomy of a process

4 main areas

the code segment	for the code of the executable
the data segment	for global variables
the heap	for dynamically allocated variables
the stack	for parameters of functions and local variables a collection of stack frames



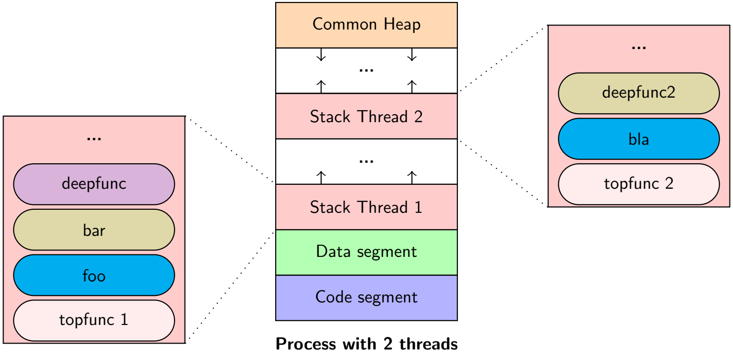
Process Layout

Threads

Definition

A lightweight process

- with its own stack
- sharing heap with other threads in the process



Process with 2 threads

Pros and cons of threads

Why should you use threads

- lightweight
- fits well with multi/many cores architecture
- fast and easy inter-thread communication
    - through memory in the heap

Pros and cons of threads

Why should you use threads

- lightweight
- fits well with multi/many cores architecture
- fast and easy inter-thread communication
  - through memory in the heap

The price to pay

- access to heap may have to be synchronized
- same for I/O and any shared resource
- the code has to be "thread safe"
  - not trivial at all
  - particularly challenging for legacy code

Threads in python

threading module

- the Thread object allows to create a thread
- call start to start the processing
- join waits until the thread ends

Example code

```
def doSth(i): ...
t = threading.Thread(target=doSth, args=(3,))
t.start()
... do sth concurrently ...
t.join()
```

Single core used most of the time

- due to the Global Interpreter Lock (GIL)
- but useful to multiplex and hide latency (e.g. I/O)

Easy alternative : `multiprocessing`

`multiprocessing` module

- the `Process` object allows to create a process
- call `start` to start the processing
- `join` waits until the process ends

Example code

```python
def doSth(i): ...
p = multiprocessing.Process(target=doSth, args=(3,))
p.start()
... do sth concurrently ...
p.join()
```

No GIL limitation anymore

- but more heavy on start (process creation)
- and memory may explode (times nb of processes)

Multi-processing in C^{++}

A complicated task

- not supported by the language itself
- external libraries are available
    - in particular Boost Process
- but inter-process communication is complex
- thus very seldom used in my experience

Example code - from Boost documentation

```
#include <boost/process.hpp>
using namespace boost::process;
ipstream pipe_stream;
child c("gcc --version", std_out > pipe_stream);
...
std::string line;
while (pipe_stream && std::getline(pipe_stream, line) && !line.empty())
  std::cerr << line << std::endl;
...
c.wait();
```

Threads in C^{++}

std::thread

- new (C^{++}11) object std::thread in <thread> header
- takes a callable as argument of its constructor
- must be detached or joined before the main thread terminates
- C^{++}20: std::jthread automatically joins at destruction

Example code

```
void doSth(int i) {...}
int main() {
  std::thread t1(doSth, 3);
  std::thread t2([](){ ... lambda code ...});
  for (auto t: {&t1,&t2}) t->join();
}
```

Asynchronicity

Concept

- separation of the specification of what should be done and the retrieval of the results
- “start working on this, and ping me when it’s ready”

Asynchronicity

Concept

- separation of the specification of what should be done and the retrieval of the results
- “start working on this, and ping me when it’s ready”

Practically

- std::async function launches an asynchronous task
- std::future template allows to handle the result

Asynchronicity

Concept

- separation of the specification of what should be done and the retrieval of the results
- “start working on this, and ping me when it’s ready”

Practically

- std::async function launches an asynchronous task
- std::future template allows to handle the result

Example code

```
int computeSth() {...}
std::future<int> res = std::async(computeSth);
std::cout << res->get() << std::endl;
```

Mixing asynchronicity and threading

Is async running concurrent code ?

- it depends!
- you can control this with a launch policy argument

  std::launch::async spawns a thread for immediate execution

  std::launch::deferred causes lazy execution in current thread

  - execution starts when get() is called
- default is not specified!

Mixing asynchronicity and threading

Is async running concurrent code ?

- it depends!
- you can control this with a launch policy argument

  std::launch::async spawns a thread for immediate execution
  std::launch::deferred causes lazy execution in current thread

  - execution starts when get() is called
- default is not specified!

Usage

```
int computeSth() {...}
auto res = std::async(std::launch::async,
                      computeSth);
auto res2 = std::async(std::launch::deferred,
                       computeSth);
```

Fine grained control

std::packaged_task template

- creates an asynchronous version of any callable
  - identical arguments
  - returns a std::future
- provides access to the returned future
- associated with threads, gives full control on execution

Fine grained control

std::packaged_task template

- creates an asynchronous version of any callable
  - identical arguments
  - returns a std::future
- provides access to the returned future
- associated with threads, gives full control on execution

Usage

```
int task() { return 42; }
std::packaged_task<int()> pckd_task(task);
auto future = pckd_task.get_future();
pckd_task();
std::cout << future.get() << std::endl;
```

Thread-safety issues

Data races

Example code - godbolt

```cpp
int a = 0;
void inc100() {
  for (int i=0; i < 100; i++) a++;
};
int main() {
  std::thread t1(inc100);
  std::thread t2(inc100);
  for (auto t: {&t1,&t2}) t->join();
  std::cout << a << std::endl;
}
```

Data races

Example code - godbolt

```cpp
int a = 0;
void inc100() {
  for (int i=0; i < 100; i++) a++;
};
int main() {
  std::thread t1(inc100);
  std::thread t2(inc100);
  for (auto t: {&t1,&t2}) t->join();
  std::cout << a << std::endl;
}
```

What result do you expect ?

Data races

Example code - godbolt

```cpp
int a = 0;
void inc100() {
  for (int i=0; i < 100; i++) a++;
};
int main() {
  std::thread t1(inc100);
  std::thread t2(inc100);
  for (auto t: {&t1,&t2}) t->join();
  std::cout << a << std::endl;
}
```

What result do you expect ?

Anything between 100 and 200 !!!

Atomicity

Definition (Wikipedia)

- an operation (or set of operations) is atomic if it appears to the rest of the system to occur instantaneously

Practically

- an operation that won't run concurrently to another one
- an operation that will have a stable environment during execution

Atomicity

Definition (Wikipedia)

- an operation (or set of operations) is atomic if it appears to the rest of the system to occur instantaneously

Practically

- an operation that won't run concurrently to another one
- an operation that will have a stable environment during execution

Is ++ operator atomic ?

Atomicity

Definition (Wikipedia)

- an operation (or set of operations) is atomic if it appears to the rest of the system to occur instantaneously

Practically

- an operation that won't run concurrently to another one
- an operation that will have a stable environment during execution

Is ++ operator atomic ?

Usually not. It behaves like :

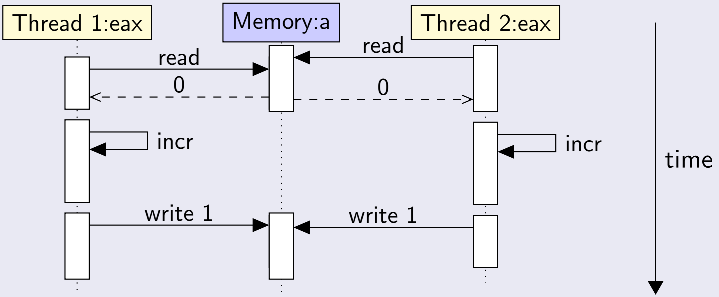
```
eax = a          // memory to register copy
increase eax     // increase (atomic CPU instruction)
a = eax          // copy back to memory
```

Timing of ++ operation

Code

```
eax = a           // memory to register copy
increase eax      // increase (atomic CPU instruction)
a = eax           // copy back to memory
```

For 2 threads



Thread safe code

Definition (Wikipedia)

Thread-safe code only manipulates shared data structures in a manner that ensures that all threads behave properly without unintended interaction

Practically, in most cases

- no data races
- any shared data has to be protected
- usage of heap is dangerous
- stack should be used when possible

What is not thread/safe ?

Everything ! unless explicitly stated

What is not thread/safe ?

Everything ! unless explicitly stated

Non thread-safe code

- heap
- STL containers
  - they use the heap !
  - some functions may be thread safe though
- global (non const) variables
- any member of a C^{++}class modified in a non const method
  - the method may be called concurrently by two threads
- `constcast`, singletons, caches, ...
- many C library calls (e.g. strtok, ctime, ...)
- external libraries
  - random number generators (most of them)

Consequences of lack of thread safety

Best case : crashes

- segfault, core dumps, ...
- visible and easy to analyze

Not so good case : non reproducible behavior

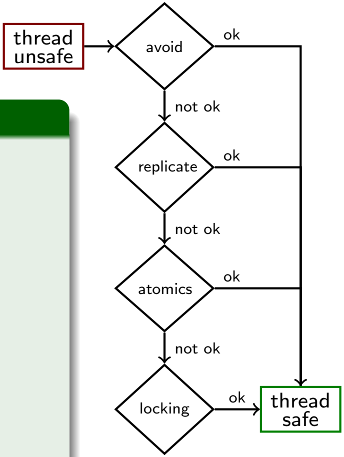
- e.g. crash once every 1000 runs
- intermittent wrong results

Bad case : silent data corruption

- wrong output (think physics results)
- invisible systematic biases

Thread-safety solutions

Overall strategy



To be tried in this order

- Avoid the problem
  - remove states in your code
- Remove problems via replication
  - remove shared states in your code
- Use atomics
  - reduce shared state to one atomic
- Use locking
  - and pay the price
  - slowness, dead locks, contention, ...

Ideal case : avoid the problem

Good practices in that direction

- use the stack
- avoid global variables
- use method constness
- pass context explicitly as function parameters

Problematic code

```cpp
struct Example {
  Context *ctx;
  std::vector<int>* tmp;
  auto foo() {
    tmp->clear();
    ...use ctx, fill tmp...
    return tmp;
  }
};
```

Thread-safe code

```cpp
struct Example {
  auto foo(Context const& ctx) const {
    std::vector<int> tmp;
    ...use ctx, fill tmp...
    return tmp; // copy elision
  }
};
// or even drop class Example
```

Functional programming

Definition (Wikipedia) and rules

a programming paradigm where programs are constructed by applying and composing functions. Functions have

- no side effects
- no modification of the input
- new return values

Consequences

- no state, no globals, no members
- guaranteed thread-safety !

Usage

- dedicated languages : Haskell, Erlang, Lisp, ...
- modern C^{++}(lambdas, move, copy elision, ...)

Data replication

Idea

- drop shared states by having one state per thread
  - potentially all identical
- only when the state is not a synchronization point

Typical cases

- external state
  - internal state of the random number generator
  - current geometry / event / working item
- counters
  - count per thread instead of globally
  - potentially merge at the end
  - there synchronization will be needed

Thread local storage in C^{++}

Principles

- have a private memory block per thread
  - some kind of private heap
- so called `thread_local` storage (new C^{++}keyword)
- and automatically allocate one instance of each variable per thread

Practical usage in C^{++}

```cpp
void counter() {
  thread_local int count{0};
  std::cout << "called " << ++count << " times in this thread";
}
```

- count is initialized on first call in each thread
- it's a kind of `static` variable
- it's destroyed when thread terminates
- low cost but not for free

Atomic types

Definition (cplusplus.com)

types that encapsulate a value whose access is guaranteed to not cause data races and can be used to synchronize memory accesses

Usage in C++

- `#include <atomic>`
- wrap your type with `std::atomic<>`
- resulting type is optimized
    - uses hardware atomics when available, locks otherwise

```cpp
std::atomic<int> a{0};
std::thread t1([&](){ a++; });
std::thread t2([&](){ a++; });
a += 2;
t1.join(); t2.join();
assert( a == 4 ); // Guaranteed to succeed
```

Be cautious with atomics

Expressions using an atomic type are *not always* atomic!

```
std::atomic<int> a{0};
std::thread t1([&]{ a = a + 2; });
```

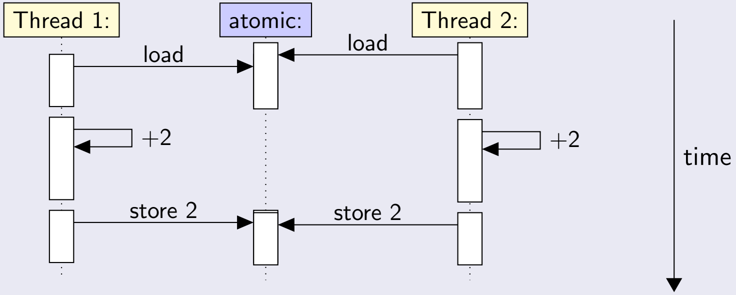
Be cautious with atomics

Expressions using an atomic type are *not always* atomic!

```
std::atomic<int> a{0};
std::thread t1([&]{ a = a + 2; }); // a += 2 would be fine !
```

- 3 steps : Atomic load / `value += 2` / atomic store

Sequence diagram



Mutexes and Locks

Concept

- Use “locks” to serialize access to a non-atomic piece of code
- The basic item is a `std::mutex`
- It can only be locked by one thread at a time
  - others trying will wait until the lock is freed

Mutexes and Locks

Concept

- Use "locks" to serialize access to a non-atomic piece of code
- The basic item is a `std::mutex`
- It can only be locked by one thread at a time
    - others trying will wait until the lock is freed

Practically

```
int a = 0; // please use atomics in such a case !
std::mutex m;
void inc() {
  std::scoped_lock lock{m}; // serialization point
  a++;
} // lock released
```

And now comes the bill...

Locks are relatively slow

- previous example slows down by factor 15...
- but usually fine for reasonable cases
    - essentially not taking locks too often

Locks bring a range of new issues

- contention : threads idle waiting for a lock
    - and the holding thread may be on hold by the OS
    - may prevent to reach expected speedup
- dead locks : threads fighting for locks

Dead lock

Scenario

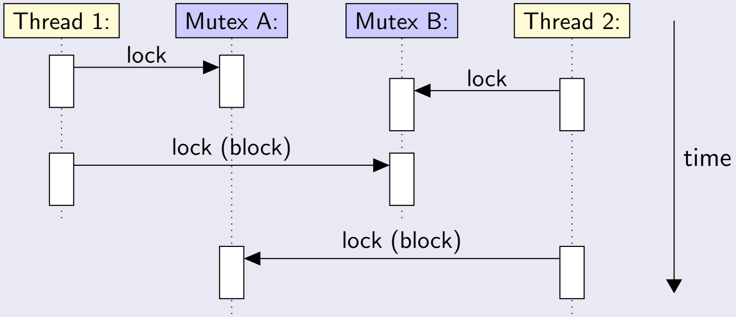
- 2 mutexes, 2 threads
- locking order different in the 2 threads

Dead lock

Scenario

- 2 mutexes, 2 threads
- locking order different in the 2 threads

Sequence diagram



How to avoid dead locks

Possible solutions

- C++17: `std::scoped_lock lock{m1, m2};` comes with deadlock-avoidance algorithm
- Never take several locks
  - Or add master lock protecting the locking phase
- Respect a strict order in the locking across all threads
- Do not use locks
  - Use other techniques, e.g. queues

Lock summary

Good practice

- **Always avoid locks when possible**
- Take as few locks as possible
- Hold a lock for the smallest amount of time possible
  - wrap critical section within "{ }"
- Use `scoped_lock` to avoid deadlock and missing release

```cpp
void function(...) {
  // ...
  {
    std::scoped_lock myLocks{mutex, ...};
    // critical section
  }
}
```

Conclusion

Key messages of the day

- parallelism is ubiquitous today
  - required by modern hardware
  - usually through cores and threading
- multi-threading is the key on modern processors
  - easy to use with python or modern C^{++}
  - brings new problems of synchronization
- many solutions
  - avoiding, replication, atomics, locks, ...
  - respect this order and pay for your needs