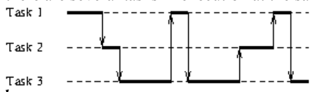
Lecture 1 - Intro

What?

concurrent
there are several tasks in execution at the same moment, that is, task 2 is started before task 1 ends:



parallel
(implies at least a degree of concurrency) there are several processing units that work simultaneaously, executing parts of several tasks at the same time.

distributed
(implies parallelism) the processing units are spatially distributed.

Why?

optimize resource utilization
Historically, the first. While a task was performing an I/O operation, the CPU was free to process another task. Also, when a user thinks about what to type next, the CPU is free to handle input from another user.

increase computing power
Single-processor systems reach their physical limits, given by the speed of light (300mm/ns) and the minimum size of components. Even single processors have been using parallelism between phases of execution (pipeline).

integrating local systems
A user may need to access mostly local resources (data), but may need also some data from other users. For performance (time to access, latency) and security reasons, it is best to have local data local, but we need a mechanism for easily (transparently) accessing remota data also.

redundancy
We have multiple units able to perform the same task; when one fails, the others take over. Note that, for software faults (bugs), it is possible that the backup unit has the same bug and fails, too.

Why not (difficulties)

increased complexity

race conditions
What happens if, while executing an operation, some of the state that is relevant for it, is changed by a concurrent operation?

deadlocks
Task A waits for task B to do something, while task B waits for task A to do someother thing.

non-determinism
The result of a computation depends on the order of completion of concurrent tasks, which in turn may depend on external factors.

lack of global state; lack of universal chronology (distributed systems only)
A process can read a local variable, but cannot read a remote variable (that resides in the memory of another processor); it can only request the value to be sent, and, by the time the value arrives, the original value may have changed

Clasification

Flynn taxonomy

SISD (single instruction, single data)
SIMD (single instruction, multiple data)
MISD (multiple instruction, single data)
MIMD (multiple instruction, multiple data)

Shared memory vs message passing

Shared memory

SMP (symmetrical multi-processing)
identical processors (cores) accessing the same main memory

AMP (asymmetrical multi-processing)
like SMP, but processors have different capabilities (for example, only one can request I/O)

NUMA (non-uniform memory access)
each processor has a local memory but can also access a main memory and/or the local memory of the other processors.

Message passing

cluster
many computers packed together, maybe linked in a special topology (star, ring, tree, hyper-cube, etc)

grid
multiple computers, maybe of different types and characteristics, networked together and with a middle-ware that allows treating them as a single system.

Lecture 2 - Handling concurrency

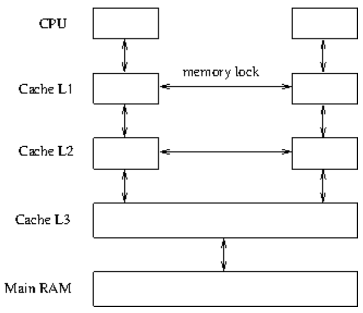
Multi-processors with shared memory

Idea: several CPUs sharing the same RAM. (Note: by CPU, we understand the circuits that process the instruction flow and execute the arithmetic operations; a PC CPU chip contains additional circuits, as well, such as the memory cache, memory controller, etc.)

Memory caches

Problems: high memory latency; memory bottleneck

Solution: use per-processor cache



New problem: ensure cache consistency (consider that one CPU modifies a memory location and, immediately afterwards, another CPU reads the same location).

Solution: cache-to-cache protocol for ensuring consistency. (Locking, cache invalidation, direct transfer between caches.) However, this means that:

- if multiple CPU access for both read and write the same memory location, the access is serialized and no speed-up results from multiple cores (moreover, there is a penalty to be paied for the cache ping-pong);
- the same happens if two variables are placed in distinct memory locations, but in the same cache line (false sharing).

Note: see false-sharing.cpp and play with the *alignof* argument.

Instruction re-ordering

In the beginning, the CPU executed instructions purely sequentially, that is, it started one instruction only after the previous one was fully completed.

However, each instruction consists in several steps (fetch the instruction from memory, decode it, compute the addresses of the operands, get the operands from memory, executing any arithmetic operation, etc) and sub-steps (a multiplication, for instance, is a complex operation and takes many clock cycles to complete). Thus, the execution of an instruction takes several clock cycles to complete.

It is possible, however, to parallelize the stages in instruction execution, for instance, to fetch the next instrunction while the previous one is being decoded. The result is a processing *pipeline*, and thus, at each moment, there are several instruction in various stages of their execution. The advantage is that the average execution time per instruction is reduced, but there is a problem if an instruction needs some results from the previous instruction before those results are ready. To solve this problem, the solution is to add wait states or to re-order instructions (so that there is enough time between dependent instructions). Both waits and re-orderings can be done either by the compiler or by the CPU itself.

The result for the programmer is that instructions can be re-ordered without the programmer knowing about that. The reordering is never allowed to change the behavior of a single thread, but can change the behavior in multi-threading contexts. Consider the following code:

```
  bool ready = false;
  int result;

Thread 1:
  result = <some expression>
  ready = true

Thread 2:
  while(!ready) {}
  use(result)
```

Because of re-ordering, the above code may not be correct. The compiler or the CPU can re-order the instructions in Thread 1 because the behavior of thread 1 is not change by that. However, this makes thread 2 belive the result is ready before actually being so.

Processes and threads

A *thread* has a current instruction and a calling stack. In more details, it has the following attributes:

- The pointer to the current instruction (IP register);
- The stack of nested function calls (with the return address from each function to its caller);
- The local variables and temporaries for each active function.

At each moment, a thread can:

- *run* on one of the CPUs,
- *be suspended*, waiting for a CPU to become available,
- *sleep*, waiting for some external operation to complete.

This means that each CPU executes instructions from one thread until it either launches a blocking operation (read from a file or from network), its time slice expires, or some higher piority thread becomes runable. At that point, the operation system is invoked (by the read syscall, by the timer interrupt, or by the device driver interrupt), saves the registers of the current thread, including the *instruction pointer* (IP) and the *stack pointer* (SP), and loads the registers for the next scheduled thread. The last operation effectively restores the context of that thread and jumps to it.

It should be noted that a *context switch* (jumping from one thread to another) is quite an expensive operation, because it consists in some hundreds of instrunctions, and may invalidate a lot of the CPU cache.

Creation and termination of a thread is also expensive.

A *process* can have one or more threads executing for it. The memory and the opened files are per process.

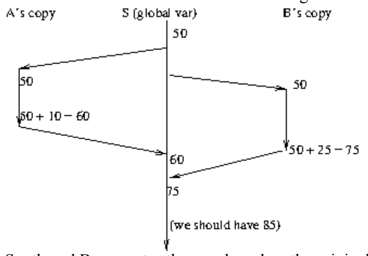
Mutual exclusion problem

The problem

Two threads walk into a bar. The bartender says:
Go I don't away! want a race to get condition last like I time had.

Consider several threads, where each of them adds a value to a shared sum. For instance, each thread processes a sale at a supermarket, and each adds the sale value to the total amount of money the supermarket has.

Since the addition itself is done in some register of the CPU, it is possible to have the following timeline:



So, thread B computes the sum based on the original value of S, not the one computed by thread A, and overwrites the value computed by A. What we should have is to execute the addition either fully by A and then fully by B, or viceversa; but not overlapped.

Atomic operations

These are simple operations, on simple types (integers, booleans, pointers), that are guaranteed to execute atomically. They have hardware support in the CPU. They need to be coupled with *memory fences* — directives to the compiler and CPU to refrain from performing some re-orderings.

Operations:

- *increment / decrement*;
- *add / substract* a given value;
- *compare-and-exchange*: compare the atomic variable with a given value and, if equal, set it to another value; also, return the old value.

See:

- C++ 11: atomic<T>
- C#: Interlocked (static class)
- Java: AtomicInteger

Uses:

- Simple atomic add/substract values;
- Basic block to implement spinlocks;
- Compare and exchange: compute new value, then replace current value with new value if the current value is still the old one.

Mutexes

A mutex can be hold by a single thread at a time. If a second thread tries to get the mutex, it waits until the mutex is released by the first thread.

A mutex can be implemented as a *spin-lock* (via atomic operations), or by going through the operating system (which puts the thread to sleep until the mutex is freed).

Mutexes are used to prevent simultaneous access to the same data.

Each mutex should have an associate invariant that holds as long as nobody holds that mutex:

- some variables are guaranteed not to change;
- some consistency conditions are guaranteed to hold.

Mutexes in various languages:

- C++ 11: mutex: lock(), unlock(), try_lock; also unique_lock<mutex>
- C#: Monitor (static class) Enter(), Exit(), or lock(){}
- Java: Semaphore or synchronized(){}

Invariants in single-threaded applications

In a single threaded program, when a function begins execution, it assumes some *pre-conditions* are met. For instance:

```
// preconditions:
//   - a, b and result are valid vectors
//   - vectors a and b are sorted in increasing order
//   - vector result is not an alias to either a or b
// post-conditions:
//   - a, b and result are valid vectors
//   - vector result contains all the elements in a and b, each having
//       the multiplicity equal to the sum of its multiplicities in a and b
//   - vector result is sorted
//   - vectors a and b are not modified
//   - no other program state is modified
void merge(vector const& a, vector const& b, vector&result);
```

If the pre-conditions are met, the function promises to deliver the specified *post-conditions*.

If the pre-conditions are not met, the behavior of the function is undefined (anything may happen, including crashing, corrupting other data, infinite loops, etc).

In conjunction with classes, we have the concept of a *class invariant*: a condition that is satisfied by the member data of the class, whenever no member function is in execution.

Any public member function assumes, among its pre-conditions, that the invariant of its class is satisfied. Also, any public member function promises, among its post-conditions, to satisfy the class invariant.

At a larger scale, there are various invariants satisfied by subsets of the application variables. Consider the case of a bank accounts application: an invariant would be that the account balances and history all reflect the same set of processed transactions (the balance of an account an is the sum of all transactions in the account history, and if a transaction appears on the debited account, it appears also on the credited account, and viceversa).

At the beginning of certain functions (for instance, those performing a money transfer, as above), we assume some invariant is satisfied; the same invariant shall be satisifed in the end. Then, sub-functions are invoked, concerned with sub-aspects of the computation to be done; the precise pre- and post-conditions for those functions should be part of their design; however, many bugs arise from a misunderstanding regarding those per- and post-conditions (in other words, the exact responsability of each function).

Note that sometimes the history is not kept as a physical variable in the system; nevertheless, we could think of it as if it were really there.

An implicit assumption in a single-threaded program is that nobody changes a variable unless explicit instructions for that exist in the currently executing function.

Invariants in multi-threaded applications

In multi-threaded applications, it is hard to know when it is safe to assume a certain invariant and when it is safe to assume that a certain variable is not modified.

This is the role of mutexes: a mutex protects certain invariants involving certain variables. When a function aquires a mutex, it can rely that:

- the invariants are satisfied at the point the mutex is aquired;
- no other thread will modify the variables before the mutex is released.

The function must re-establish the invariant before releasing the mutex.

The above also implies that, in order to modify a variable, a function must make sure it (or its callers) hold all mutexes that protect invariants concerning that variable.

Read-write (shared) mutexes

There are two use cases concerning the invariants:

1. a function changes some variables, it needs to ensure that the invariant holds when it begins, promises to re-establish the invariant, but it will violate the invariant during its execution. Therefore, during the execution, nobody else can be allowed to see the variables involved in the invariant.
2. a function needs to ensure that some invariant is satisfied during its execution, but it does not change any variable involving in that invariant.

A thread doing case 1 above is incompatible with any other thread accessing any of the variables involved in the invariant. A thread doing case 2 above, however, is compatible with any number of threads doing case 2 (but not with one doing 1).

For this reason, we have *read-write mutexes*, also called *shared mutexes*. Such a mutex can be locked in 2 modes:

1. *exclusive lock* or *write lock*, which is incompatible with any other thread locking the mutex;
2. *shared lock* or *read lock*, which is incompatible with any other thread locking in *exclusive* mode the same mutex, but is compatible with any number of threads holding the mutex in *shared* mode.

Caveat: the implementation of a shared mutex must deal with the following dilemma: Suppose several readers hold the mutex in shared mode, and a new (writer) thread attempts to lock it in exclusive mode. What to do if, before all the readers finish, a new reader comes in? If we allow the reader, we run the risk of starving the writer (if we have enough readers to keep at least one active one for a long time). If we deny the reader, we miss a parallelizing opportunity.

On recursive mutexes

A recursive mutex allows a lock operation to succeed if the mutex is already locked by the same thread. The mutex must be unlocked the same number of times it was locked.

The problem with recursive mutexes is that, if a function attempting to aquire a mutex cannot determine if the mutex is alreay locked or not, will not be able to determine if the invariant protected by the mutex holds or not immediately after the mutex is aquired. On the other hand, if the function can determine if the mutex is already locked, it has no need for a recursive mutex.

Radu-Lucian LUPŞA
2020-10-06

Lecture 3 - More concurrency issues

Finer granularity locks

Fighting the bottlenecks

A mutex is a bottleneck in the program. If multiple threads try to acquire a mutex, their actions will be serialized, thus negating the parallelism.

Reducing the contention over mutexes can be done the following ways:

- making as much of the data read-only; this can be very effective in conjunction with functional-style programming. Functional programming means (mong other things): Use pure functions - functions that depend only on the explicit parameters (not on some global state) and do not modify them. Use variables assigned only once (at the definition time) and only read afterwards.
- reducing the granularity of each mutex;
- using shared mutexes whenever reasonable.

Caveats:

- with many mutexes, it is harder to know exactly what operation requires which mutex;
- with many mutexes, it is harder to avoid deadlocks;
- with many mutexes, it the overhead of aquiring and releasing mutexes can negate the gain with increased parallelism;
- a shared lock is more expensive than an exclusive lock.

Consistency issues

Consider the bank accounts problem (lab 1 pb 2):

```
struct Account {
    unsigned balance;
    mutex mtx;
};

bool transfer(Account* a, Account* b, unsigned amount) {
    a.mtx.lock();
    if(a.balance < amount) {
        a.mtx.unlock();
        return false;
    }
    a.balance -= amount;
    a.mtx.unlock();

    // ---> what if the audit() occurs here?

    b.mtx.lock();
    b.balance += amount;
    b.mtx.unlock();
    return true;
}
```

An operation running on some other thread between updating the two accounts will see an inconsistent state, with money no longer in the first account and not yet in the second one; so, the audit would fail.

With using locks, the solution is to lock the second mutex before freeing the first one.

The general idea is:

1. There will be, in the code, a reference point where the transaction can be postulated to occur;
2. For every variable involved, there is a mutex that protects it between the time the ideal transaction occurs and the point(s) it is read and/or modified

```
struct Account {
    unsigned balance;
    mutex mtx;
};

bool transfer(Account* a, Account* b, unsigned amount) {
    a.mtx.lock();
    if(a.balance < amount) {
        a.mtx.unlock();
        return false;
    }
    a.balance -= amount;
    // ---> the reference time for the transaction is later, but nobody can see a's balance until then

    b.mtx.lock();
    // ---> we can consider the whole transaction occurs here
    a.mtx.unlock();

    // ---> nobody can see that the balance of account b is not yet modified
    b.balance += amount;
    b.mtx.unlock();
    // ---> as we unlock the mutex, b's balance has the right value
    return true;
}
```

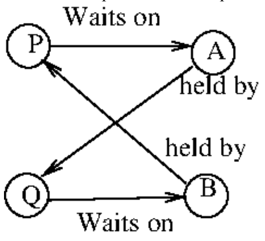
However, we've just traded one problem for another...

Deadlocks

A deadlock occurs when there is a cycle in the wait dependency between threads (a thread waits on another thread if the first thread tries to aquire a mutex that is already locked by the second thread).

For the account transfer example above, consider thread P executing a transfer from account A to B has aquired the mutex on A and is attempting to aquire the mutex on B. Meanwhile, thread Q executes a transfer from B to A, aquires the mutex for B, and tries to aquire mutex for A. Thread P is put on hold and cannot advance (and release A) until thread Q releases mutex B. Meanwhile, Q cannot advance (and release B) until P releases mutex A.

This can be represented as a dependency graph. Any cycle in such a dependency graph means a deadlock:



A guaranteed solution to avoid deadlocks is to have a total order relation between mutexes and to allow threads to aquire mutexes only in that order (it is ok to skip mutexes, but not to go back). An special (extreme) case of this is to aquire at most one mutex at any time. This solution is, however, hard to implement if the set of mutexes to be aquired is known only after having aquired other mutexes.

For the bank accounts problem above, the solution would be to have each account have an ID, and lock first the mutex of the account with the lowest ID.

Another solution is to detect deadlocks and to refuse an aquiring attempt when a deadlock would result. This leads to more complex code (the code must threat the case when a lock operation fails) and it is possible to result in a *livelock* - when two or more threads attempt to aquire locks, give up because of a potential deadlock, restart their operations and repeat the same scenario over again.

Exercices

1. Design a thread-safe doubly-linked list with per-node mutex.

2. Bank transfer when accounts can be created and deleted/disabled dynamically.

Producer-consumer communication

The problem

We have a computation that depends on the result of some other computation or on some external data (from the disk, from the network, etc).

A simple solution is to use an atomic variable like in the following example:

```
  atomic_bool ready = false;
  int result;

Producer thread:
  result = <some expression>
  ready = true

Consumer thread:
  while(!ready) {}
  use(result)
```

Note that the default semantic of the atomic variables prevent the compiler and the CPUs from re-ordering the instructions around the atomic variable operations.

However, the problem is that, if the consumer thread gets to the waiting loop long before the producer manages to produce the data, the consumer will busy-wait for the result. This wastes CPU time, which would be much more useful for the producer thread; the operating system cannot know that the consumer does nothing useful and schedule something else.

Condition variables and their usage

A *condition variable* is a mechanism, involving the operating system if needed, allowing the consumer to wait until signaled by the producer thread. Waiting consumes no CPU.

Simplifying a bit, it has 2 operations

wait()
: puts the calling thread on wait, until the condition variable gets signaled

notify()
: wakes up (ends waiting) any thread waiting in a wait() call at that time.

Actually, *condition variable* is a misnomer: it is simply a communication channel, rather than a variable: it has no state, and the signal passed is transient: it wakes up any consumer waiting at that time, but leaves no trace for

The previous example, re-written using condition variables would be (not entirely correct, yet):

```
  std::mutex mtx;
  bool ready = false;
  int result;
  std::condition_variable cv;

Producer thread:
  int local_result = <some expression>
  {
    unique_lock<mutex> lck(mtx);
    result = local_result;
    ready = true;
    cv.notify();
  }

Consumer thread:
  int local_result;
  {
    unique_lock<mutex> lck(mtx);
    while(!ready) {
      cv.wait();
    }
    local_result = result;
  }
  use(local_result);
```

There is a problem with the above code:

- The lock is needed because otherwise, if the producer sets *ready* to true and signals the condition variable just between the time the consumer checks the value of *ready* and the time it enters the *wait()*, then the consumer misses the wake-up call by *signal()* and will wait forever.
- On the other hand, with the lock, we get a deadlock because the consumer will sleep with the mutex locked, therefore not allowing the producer to aquire it in order to set the result and signal the condition variable.

To solve this, the mutex needs to be released for the duration of the sleep, but the release must be atomic with the sleep. The code becomes, on the consumer side:

```
  int local_result;
  {
    unique_lock<mutex> lck(mtx);
    while(!ready) {
      cv.wait(lck);
    }
    local_result = result;
  }
  use(local_result);
```

That is, the *wait()* function gets the mutex holder as an argument. The mutex is release upon entering the sleep, and re-aquired before returning from *wait()*.

Radu-Lucian LUPŞA
2020-10-11

Lecture 4 - Higher level multithreading concepts

Thread creation issues

Creating many threads has two issues:

1. creating threads, and swithcing between threads, is expensive;
2. there is an OS-dependent upper limit on the number of threads.

This can be illustrated by running the program vector_sum_multithread.cpp. It attempts to compute a vector sum, creating one thread for each element (or for every specified number of consecutive elements). Sample runs:

To avoid the OS-imposed limit on the number of threads, we can suspend creating new threads when some pre-configured maximum is reached, and to resume when some of them terminate. The resulting program is neither efficient nor maintanable: vector_sum_limited_thread.cpp.

Thread pools

The idea behind a thread pool is the following: instead of creating a new thread when we have some work to do and finish it when the work is done, we do the following:

- we pre-create a number of threads and have them wait (on a condition variable, so the SO doesn't give them CPU time yet;
- when some work comes in, we give it to a free thread;
- when the work is done, the thread returns to the waiting state;
- if work comes in, but all the threads are busy, there are two possibilities: either we temporarily increase the number of threads, or we just block waiting for a thread to finish its work. Note that increasing the number of threads runs the risk of reaching the OS limit, while blocking waiting for a thread to finish its work runs the risk of a deadlock.

An example, with a fixed size thread pool, is given at vector_sum_thread_pool.cpp.

Producer-consumer communication

Futures and promises

This is an easier mechanism to work with, compared to the condition variables. Essentially, we have an object that exposes two interfaces:

- On the *promise* interface, a thread can, a single time, set a value. This actions also marks the completion of an activity (a *task*), and the value is the result of that computation.
- On the *future* interface, a thread can wait for the value to become available, and retrieve that value. Thus, a future is a result of some future computation.
- It is also possible to set a *continuation* on a future (see next lecture).

Examples:

futures-demo1.cpp
    using futures to get the result from asynchronous tasks
futures-demo1-with-impl.cpp
    as above, but with a possible implementation for futures and the async() call

Producer-consumer queue

See the examples:

producer-consumer.cpp
    threads communicating through producer-consumer queues
ProducerConsumer.java
    same, but in Java
producer-consumer2.cpp
    same, but the queues have limited length and enqueueing blocks if the queue is full

Lecture 5 - Futures with continuations

This is an easier mechanism to work with, compared to the condition variables. Essentially, we have an object that exposes two interfaces:

- On the *promise* interface, a thread can, a single time, set a value. This actions also marks the completion of an activity (a *task*), and the value is the result of that computation.
- On the *future* interface, a thread can wait for the value to become available, and retrieve that value. Thus, a future is a result of some future computation.
- It is also possible to set a *continuation* on a future. This is done in the C# TPL (*Task Parallel Library*). The continuation is some computation that will be executed when the future will get a value (and the computation can use that value). It is also possible to set a continuation when all the futures in some set get values, or when any of the futures in some set gets a value.

Examples:

futures-demo1.cpp
  using futures to get the result from asynchronous tasks
futures-demo1-with-impl.cpp
  as above, but with a possible implementation for futures and the async() call
futures-demo1.cs
  as above, but in C#
futures-demo2-cascade1.cs
  cascading tasks through the ContinueWith() mechanism
futures-demo2-cascade2.cs
  same as above, but showing that the actual execution is de-coupled from the setup
futures-demo3-when-all.cs
  using the WhenAll() mechanism to start a task only after all its input data is computed (by other tasks)

Handling operations that depend on external events

Blocking calls

```
recv(sd, data, len); // blocks until data is available
... // process
send(sd, result, len);
...
```

- the current thread gets blocked in recv() / send() calls until the data is received / sent from / to the connection;
- the processing is easy to understand, since the current instruction and the execution stack contains the current state of the interaction with the single client;
- needs one thread for each client (or for each blocking operation to be executed in parallel)

Event-driven, `select()`

```
while(true) {
    select(nr, readFds, writeFds, exceptFds, nullptr);
    if(FD_ISSET(sd, readFds) {
        recv(sd, data, len); // a single read is guaranteed not to block
        ...// process
    }
}
...
```

- the only point where it blocks is in select();
- hard to combine differnet libraries, since everything that may have to wait for external events needs to be dispatched from the same central point;
- a single thread servers all clients;
- the state is harder to manage (event-driven);

Event-driven, based on callbacks:

There is a `begin...()` call that initiates the operation and sets a callback that will be executed on completion. The `begin...()` operation returns immediatey an identifier of the asynchronous operation. The callback is called by the library when the operation completes. The callback (or someother thread) needs to call the corresponding `end...()` operation, that returns the results of the operation and frees the associated resources in the library.

```
class Receiver {
    void Callback(IAsyncResult ar) {
        int receivedBytes = sd.EndReceive(ar);
        // process data
        if(expectMoreData) {
            sd.BeginReceive(m_buf, m_offset, m_bufSize, 0, Callback, null);
        }
    }

    void Start() {
        // ...
        sd.BeginReceive(m_buf, m_offset, m_bufSize, 0, Callback, null);
    }
}
...
```

A complete server implementation (for a very simple server) is given in srv-begin-end.cs.

Features:

- no need to explicitly create threads;
- easier to work with, compared to `select()` (no single central event loop and dispatcher), but still harder compared to one thread per client (still event-driven);
- need to know exactly what operations can be executed on which callbacks.

Combining callbacks with futures and continuations

The idea is that the `begin...()` call is inside a function that returns a future, and the callback completes that future.

The previous server re-implemented using futures: srv-task.cs.

The `async-await` mechanism

- The programmer writes the code (almost) like code-driven (not event-driven), with blocking like calls;
- The functions with pseudo-blocking calls are declared `async` and need to return futures; the called pseudo-blocking functions must be functions returning futures and the calls are marked with `await`;
- The compiler will generate, in place of a single `async` function, multiple functions that maintaing a state machine. The initial call goes up to the first `await`; at that point, it returns the control to the caller, returning a future as the result. Just before that, however, it calls the `await` function and enqueues a continuation to the resulting future. That continuation will execute the next part of the `async` calling function, up to either a `return` - that would complete the future - or to another `await` call - that would call that function and enqueue the next part of the caller as continuation.

The previous server re-implemented using *async-await*: srv-await.cs.

Radu-Lucian LUPŞA
2020-10-31

Lecture 7 - Parallel explore and other algorithms

Parallel explore

Explore a tree, in recursive decomposition fashion.

Example: Given a set of integers, find, if possible, a partition into two subsets of equal sum.

The solution is a backtracking algorithm. To parallelize it, the backtracking search space is divided between the threads. Solution: subset_sum_async.cpp.

Pipeline pattern

Processing is divided in stages; the output of one stage is fed as input in the next stage.

Pipeline can be generalized to any DAG.

Lecture 7 - Simple parallel algorithms

Data-parallel vs task-parallel:

- data-parallel — same operation is performed, independently, on distinct subsets of the processing data;
- task-parallel — distinct operations, that do not depend on each other, are performed in parallel

The most general approach: consider the dependency graph between computed quantities. It is a DAG (directed acyclic graph). Notes:

- parallelizable activities can be found, for instance, by splitting the graph into levels;
- the *critical path* (the longest path, or the maximum cost if distinct processing steps take different times) gives a lower limit for the execution time even on an infinite number of CPUs.
- sometimes, the critical path can be shorteden at the expense of increasing the amount of computation to be performed.
- the graph may not be known from the beginning...

Simple data decomposition

Processing of an array of data can often be split into independent blocks.

The easiest case is when each input produces one output — the *map* pattern. See the example for computing the sum of two vectors: vector_sum_split_work.cpp.

Simple way of computing the boundary index: beginIdx = (threadIdx * nrElements) div nrThreads

However, beware of cache effects! Processing consecutive elements is significantly faster than processing every *k*-th element. Compare the previous program with the one at vector_sum_split_work_bad.cpp.

A more complex case arises when each output depends on a group of inputs, around the input at the same position — the *stencil* pattern. See vector_average_stencil.cpp.

It is preferrable to split on output than on inputs — so that each output is computed by exactly one worker (thread, task) and so no mutexes are necessary.

Recursive decomposition

The initial worker splits the data into two or more fragments, gives the fragments as inputs to subordinate workers, and finally it combines the results.

Example 1: Compute the sum of a vector. Create a binary tree of adders. The depth is $O(\log(n))$. Source code: recursive_decomposition_sum.cpp.

Example 2: Merge sort. The basic (non-parallel algorith) is to divide the input vector into two parts, merge-sort each part, then merge the resulting two sorted vectors into one. For parallelizing, merge-sorting the two parts can easily be done in parallel. However, the final merge is a bit harder. It can be done as follows:

- take the middle element in the first sorted vector;
- find its position in the second sorted vector, by a binary search;
- divide the two vectors by the two positions found above;
- merge independently the two pairs of sub-vectors;
- concatenate the results (this is a no-op actually).

See the C++ implementations:

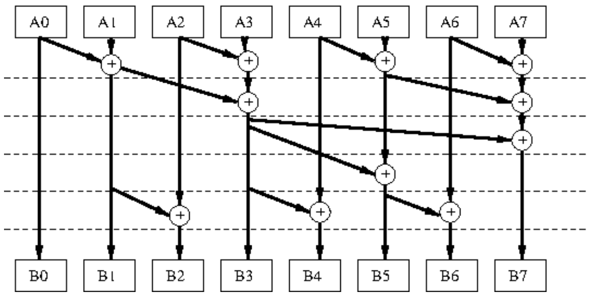
- mergesort.cpp — serial implementation;
- mergesort-par1.cpp — parallelized, but with non-parallel merge;
- mergesort-par2.cpp — fully parallelized, including parallel merge;

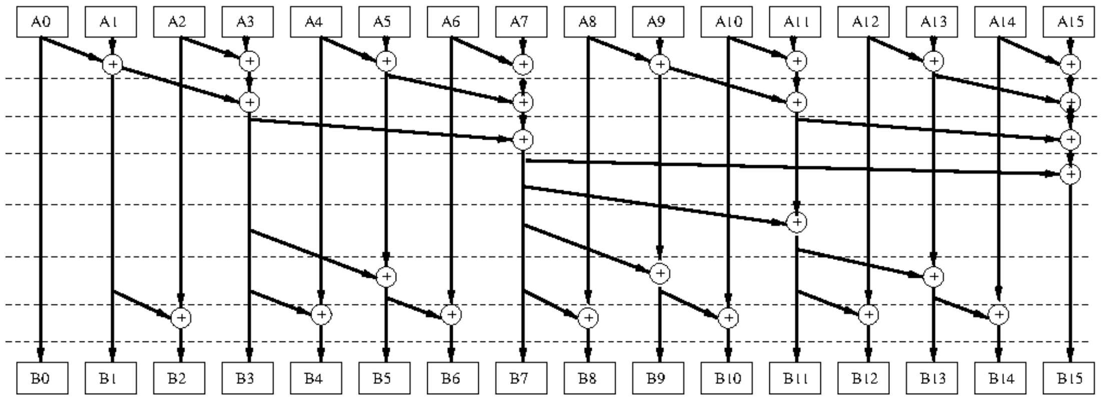
Example 3: Compute the sequence of sums of prefixes. Given $a_0, a_1, ..., a_{n-1}$, compute $b_0 = a_0, b_1 = a_0+a_1, b_2 = a_0+a_1+a_2, ..., b_{n-1} = a_0+a_1+a_2+...+a_{n-1}$.

Solution: start with a binary tree computing the sum of all numbers in the sequence. Then, compute each prefix sum from the largest parts already computed.

```
// First, compute the sums of 2^j consecutive numbers;
// b[i*2^j - 1] = a[(i-1)*2^j] + ... + a[(i-1)*2^j + 2^j - 1]
b = a
for(size_t k=1 ; k<n ; k = k*2) {
    for(size_t i=2*k-1 ; i<n ; i+=2*k) { // in parallel
        b[i] += b[i-k];
    }
}
// Then, compute each partial sum as a sum of 2^j groups:
k = k/4
for( ; k>0 ; k = k/2) {
    for(size_t i=3*k-1 ; i<n ; i+=2*k) { // in parallel
        b[i] += b[i-k];
    }
}
```

Examples:





Radu-Lucian LUPŞA
2020-11-09

Lecture 8 - Advanced parallel algorithms

Advanced recursive decomposition

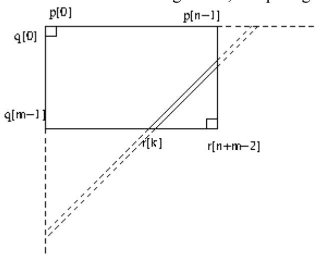
Sometimes, recursive decomposition can be done more efficiently in a way in which the parts are not straightforward, but reduce the number of operations.

Basic example of such decomposition: compute a complex product by using only 3 (instead of 4) real multiplications).

Solution: (a+bi)*(c+di) = (ac-bd) + (ad+bc)i = (ac-bd) + (ac+ad+bc+bd-ac-bd)i = (ac-bd) + ((a+b)*(c+d) - ac - bd)i, which can be computed using only 3 real multiplications (but more additions/substractions).

Polynomial multiplication using Karatsuba algorithm

Note: for the classical algorithm, computing the coefficients leads to computing the products and then add diagonals in the following table:



Idea:

- Split each of the input polynomials in half;
- Instead of multiplying each of the 4 pairs from the step above, use a similar trick to make only 3 multiplications

Assume input polynomials P(X) and Q(X) of degree 2*n-1.

Write them as P(X) = P_1(X)*X^n+P_2(X) and Q(X) = Q_1(X)*X^n+Q_2(X).

Now P(X)*Q(X) = (P_1(X)*X^n+P_2(X)) * (Q_1(X)*X^n+Q_2(X)) =
= P_1(X)* Q_1(X)*X^2n + (P_1(X)*Q_2(X)+P_2(X)*Q_1(X))*X^n + P_2(X)*Q_2(X)

But the second term can be written as
(P_1(X)+P_2(X)) * (Q_1(X)+Q_2(X)) - $_1$(X)* Q_1(X) - P_2(X)*Q_2(X)

Radu-Lucian LUPȘA
2020-11-09

Lecture 8-9 - MPI programming

Basic concepts

Launching MPI applications

A MPI application can be launched via the `mpirun` command. The MPI application is launched on several specified nodes (hosts) and in a specified number of instances on each node.

Each instance starts the execution from the `main()` function and has the same command-line arguments (specified in the `mpirun` command). Each instance can retrieve (via `MPI_Comm_size()` and `MPI_Comm_rank()`) the number of instances launched (in the same `mpirun` command) and the own ID.

The MPI setup

The `mpirun` command normally uses `ssh` to connect to the remote nodes in order to launch there instances of the MPI application. For that purpose, it is best to prepare a public key authentication setup for SSH.

In addition, `mpirun` (actually, the remote daemon) must be able to find an executable with the same name and in the same location on each of the nodes. This can be achieved either by setting up a NFS or other networked sharing of files. It is enough, however, simply to copy the executable on all of the nodes before invoking `mpirun`.

The nodes can be of distinct architecture. In that case, the executable must, of course, be distinct: each node must have the executable of the appropriate architecture.

Basic API

An MPI program must call `MPI_Init()` in the beginning and `MPI_Finalize()` in the end.

Finding out the number of launched instances is done via `MPI_Comm_size()`, with `MPI_COMM_WORLD` as the identifier. Finding out one's own ID is done via `MPI_Comm_rank()`.

Basic communication operations consist in sending and receiving an array of elements of the same type (array of integers, array of doubles, etc).

There are two kinds of communications:

- buffered communication, where the sender puts the data in a buffer and continues execution. If the receiver comes after the sender, then it takes the data from the buffer; if it comes before the sender, it has to wait for the sender.
- Synchronous communication, where the sender waits for the receiver to be ready. Whichever of the processes (the sender of the receiver) comes first, it is blocked until the other gets to the communication operation. Then, the communication is done and each process continues execution.

The receive operation is `MPI_Recv()`. The send operations are: `MPI_Ssend()` (synchronous), `MPI_Bsend()` (buffered) and `MPI_Send()` (unspecified, implementation-defined).

See the example mpi1.cpp, where a first process sends a number to a second one, the second adds 1 and sends the result to the third one, and so on, until the last process adds 1 and prints the result.

Other operations include:

- Broadcast — `MPI_Bcast()` - see example bcast-mpi.cpp
- Launching a send or receive without waiting — `MPI_Isend()` and `MPI_Irecv()`. These calls return immediately; to actually wait for the completion of the operation, call `MPI_Wait()` or `MPI_Waitany`.
- Scatter-gather: `MPI_Scatter()`, `MPI_Gather()`, `MPI_Allgather()`

Other simple example

Add all numbers in a vector. See solution sum-mpi.cpp:

- how to distribute input data;
- how each slave process gets its part as well as the meta-data (chunk size, etc)
- how the result gets gathered back at the root process

Another solution is given in sum-scatter-mpi.cpp. This one uses MPI_Scatter() and MPI_Gather() to distribute input data to workers and to gather the results.

Algorithm design isses

Divide and conquer algorithms

- Divide operations should stop when all the processes received work items;
- The parent process should not idle; so, when dividing the work, one part remains on the parent and only the other are sent to child processes (in the hierarchy).

Example: distributed merge-sorting:

- mergesort-simplified-mpi.cpp (simplified version) or
- mergesort-mpi.cpp (optimizing copying from one buffer to the other, and having children nodes deduce the parent, process pool size and vector size).
- quicksort-mpi.cpp a quicksort implementation

Interesting algorithms — Cannon's matrix multiplication

Idea (consider, for simplicity, the product of two square matrices *N*x*N*, to be computed on *K*x*K* processors, where *K* divides *N*):

- divide all 3 matrices (the two inputs and the output) into *K*x*K* equal blocks;
- each process is responsible of computing one block;
- in the beginning, each process gets one block of each input matrix and computes their product;
- then, each process passes the block from the first matrix to the rigth and the block from the second matrix down, computes their product and adds it to the result block (actually, both are circular permutations);
- After *K* such circular permutations, each process has its result and the algorithm ends.

Note that the initial distribution of blocks must be done in such a way that each process has two blocks that it has to multiply together. This involves some circular permutations of rows or columns of blocks.

See the implementation in matrix-mpi.cpp

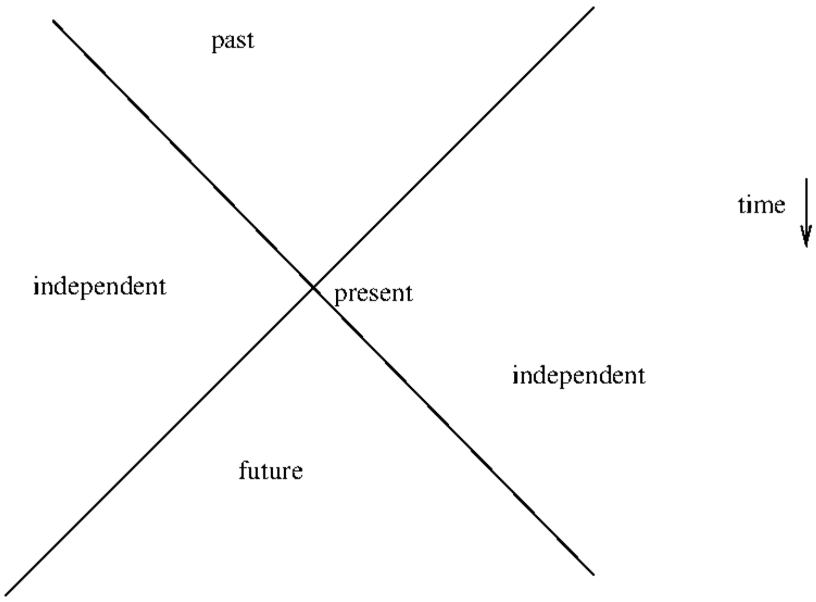
.

Lecture 10 - Basic distributed protocols

Issues

- no global state (no process has a complete knowledge of the current state of everyone);
- no well-defined time (infinitely accurate clock synchronization is not even theoretically possible).

Causal relations



a happens-before b if either:

- *a* and *b* take place on the same node and *a* occurs before *b* in the program there;
- *a* is the event of sending a message on the sender node and *b* is the event of receiving the same message on the destination node;
- there is a transitive chain from *a* to *b* based on the two cases above;

Note: non causally related events are called *concurrent*.

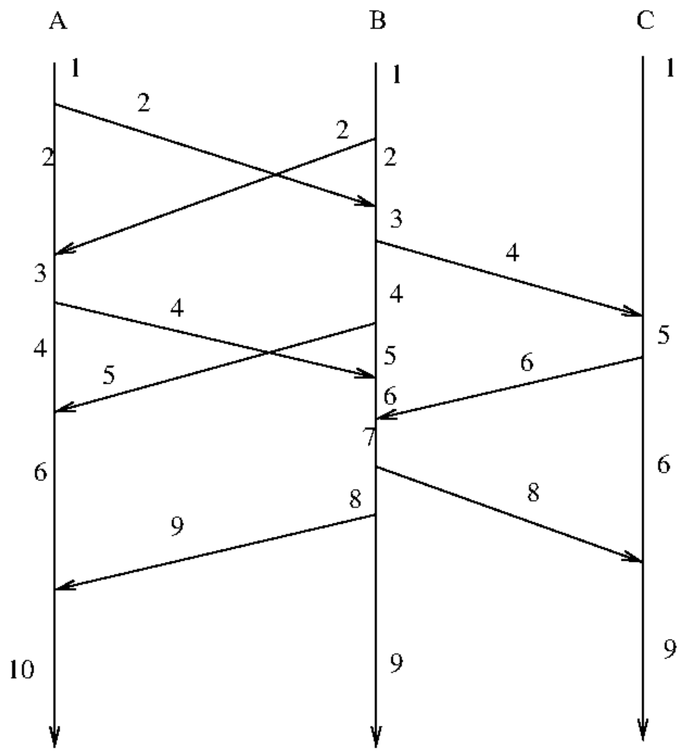
Consistent chronology problem

Problem

We want that all nodes agree on the order some events happen.

We want to assing to each event a distinct timestamp, such that all timestamps are totally ordered and, furthermore, the order is compatible with causal relations (if event A *happens-before* event B, then the timestamp of A must be smaller than the timestamp of B).

Lamport clocks



Each node keeps the current time as an integer (node: this integer has no relation with the real, physical time; it is just a number that increases as time goes by).

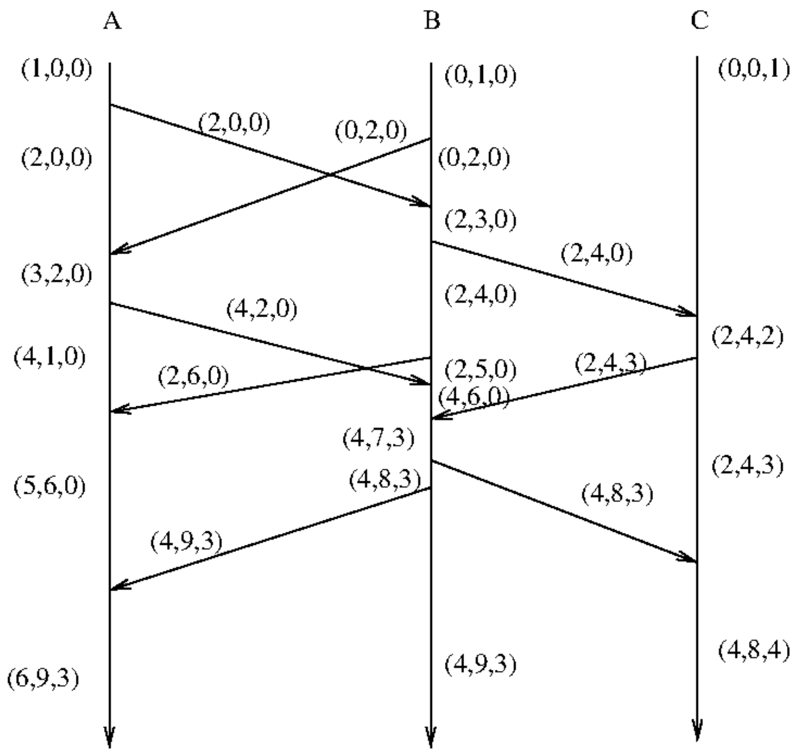
Each time a local event happens on a node, that node increments its clock;

Each message carries the sending timestamp;

When a message is received, the destination sets the clock forward if needed, so that the local clock is larger than the message timestamp;

If we need a strict order of the events, we use the node ID as a tie-break to distinguish events that otherwise have the same timestamp (note that such events can only be concurrent).

Finding causal relations: vector clocks



Each node keeps a local clock, plus, for each of the other nodes, the most recent value of the clock of that node.

Each timestamp is a vector with the clocks of all processes.

A local event leads to incrementing the local clock, but keeping fixed all other clocks (on the node the event happens).

Each message carries its sending timestamp (the vector of all clock values on the source node).

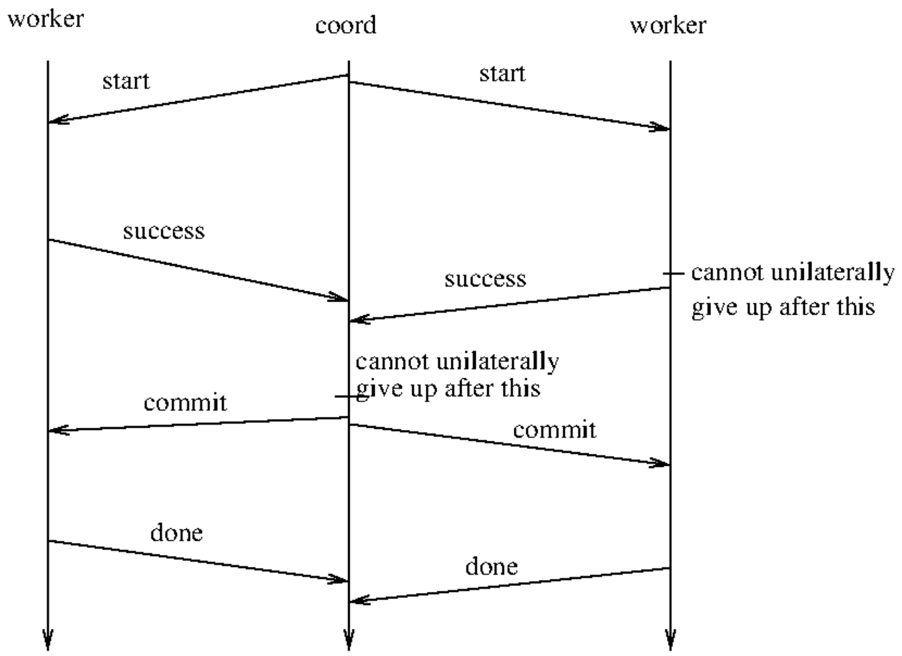
When receiving a message, the local clock is incremented and then each clock of the receiving node is set to the maximum of the current value and the value from the message timestamp.

Causal relation = all components are in larger than or equal to relation.

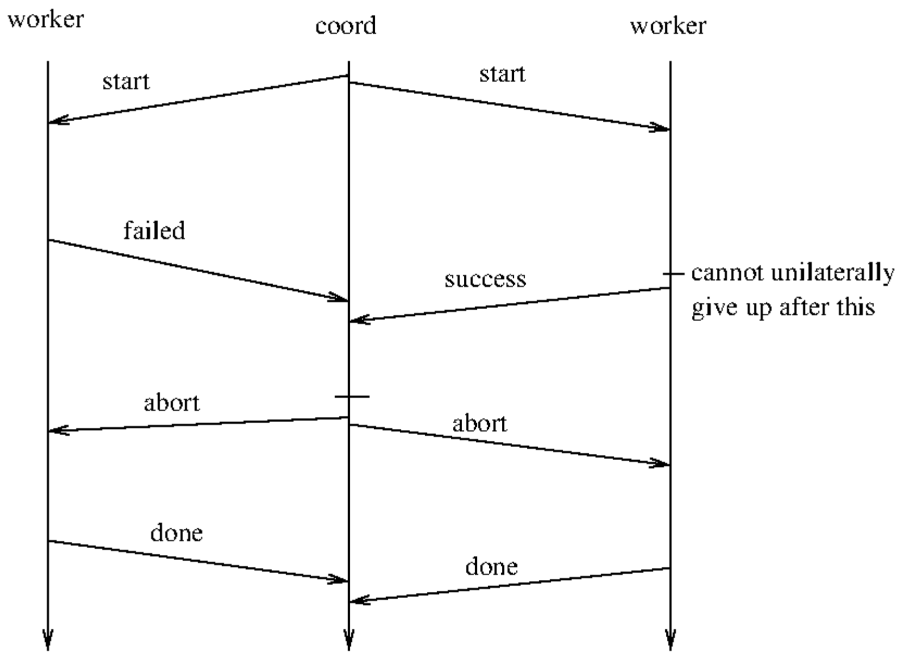
Consistency models (for shared memory and for broadcast of events)

- Sequential consistency: all events appear to happen in the same order for all processes;
- Causal consistency: two events that are causally related appear in the causal order for all processes; concurrent events may appear in different orders for different processes.
- Aquire/release consistency

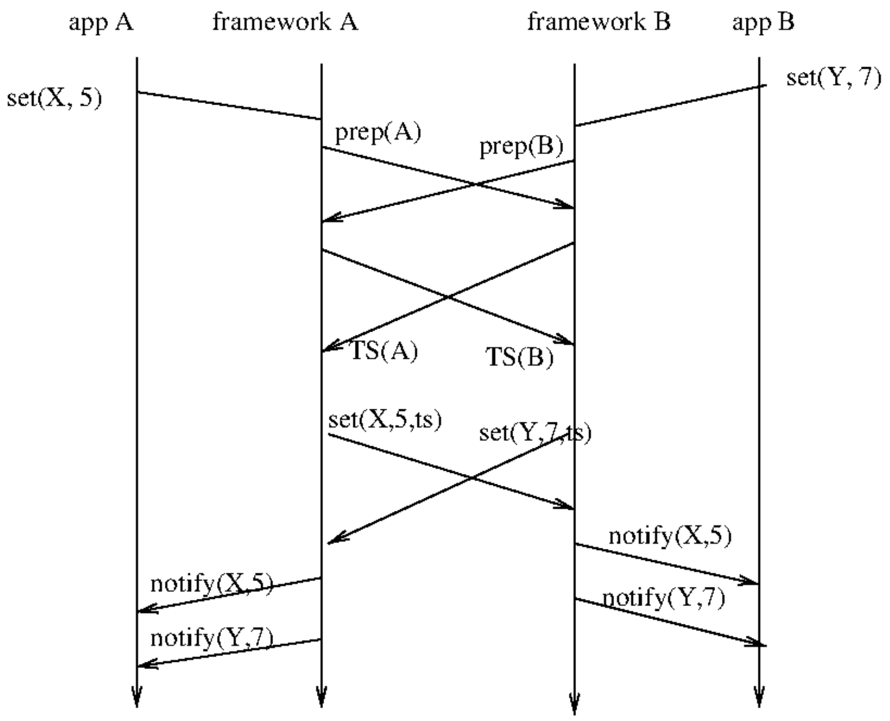
Distributed transactions


worker
coord
worker
start
start
success
success
cannot unilaterally give up after this
cannot unilaterally give up after this
commit
commit
done
done

- 2 phases;
- Phase 1: at the end, each sends a message "ready" or "failed" to the coordinator; if the message is ready, the process must be able to commit;
- Phase 2: the coordinator sends "commit" or "abort"



Distributed shared memory


app A
framework A
framework B
app B
set(X, 5)
set(Y, 7)
prep(A)
prep(B)
TS(A)
TS(B)
set(X,5,ts)
set(Y,7,ts)
notify(X,5)
notify(Y,7)
notify(X,5)
notify(Y,7)

Lecture 11 - Fault tolerance

Failure types

- Crash failure — the process executes normally until some point, then it stops completely (it doesn't send any message afterwards).
- Byzantine failure — any behavior is possible on the part of the failing process.
- Communication failure — some messages are lost (however, they are not altered, duplicated, or created). Furthermore, it is usually assumed that there is no point in time from which all the messages are lost; this is often phrased as *if infinitely many messages are sent, then infinitely many messages are delivered*.

Synchronous vs asynchronous system

- Asynchronous — there is no upper limit on the processing time or on the message transit time. However, note that any computation and any message delivery takes a finite time.
- Synchronous — there is a known upper limit on the processing times and on the message transit times. Consequently, it is possible to arrange the computations in *synchronous rounds*: in each round, each process performs some computations based on the inputs and on the messages received from peers during the previous rounds, and then it sends messages to peer processes. If, during a round, a process is supposed (according to the protocol) to receive a message from some other process, but it doesn't, it reliably detects this as a missing message (and deduce that either the message was lost or the source process has failed).

Consensus and other closely related problems

Background: consider a system consisting of *n* functionally identical process computers, each having a complete set of sensors. Each computer evaluates the readings from the sensors and decides what to do to controll the process. If they command the same action to tha actuators, the action is performed; otherwise, the action is based on the command from the majority of the process computers and an alarm is raised. To do this, however, all the computers must have all the input data from the sensors and must agree on the content of that data. This leads to the following *consensus problem*.

Consensus problem

We assume we have *n* processes, and each one has an input value x_i. Those values should be the same but, occasionally, they can differ. All processes must agree on a value to be used as input for further processing. If all inputs are equal, the agreed value must be equal to this input; if the inputs are not all equal, the agreed value can be the input of any process, but it must be the same for all processes.

The requirements are:

- Each correct process must decide an output value y_i in a finite time;
- All correct processes must decide the same output value;
- The decided value for the correct processes must be the input of some process; consequently, if all processes have the same input value, then all correct processes must decide that value as output.

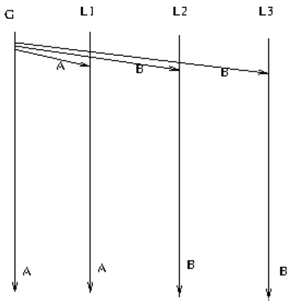
Interactive consensus

Like the consensus, but the processes are supposed to find a vector containing the inputs of all processes. The requirements are:

- Each correct process must decide an output vector in a finite time;
- All correct processes must decide the same output vector;
- The decided value for the position of a correct processes must be the input of that process.

General's problem (reliable broadcast)

A source process (the general) must send a message to other processes (its lieutenants). However, the source process may be faulty; even in that case, all the (correct) recipients must agree on the transmitted message. Essentially, this is the interactive consensus problem, but we need only the component corresponding to one process (the source).



Asynchronous case

The problem is unsolvable, even if there is at most one process that can fail and the only failure is crash-failure.

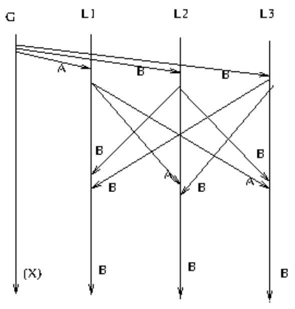
Intuitively, the problem is that we cannot distinguish between a failed process and a slow process. However, the formal proof relies on assuming the existence of a solution, examining the decision tree and finding a contradiction.

Synchronous case, byzantine failures

The solution for interactive agreement is the following:

For at most one failed process, we need $n \geq 4$. In the first round, each process sends its input to everybody else. In the second round, each process sends everybody else the values received in the first round. In the third round, each process *p* computes, for each other process *q*, the corresponding value in the output vector as follows: it looks at the reported values for *q*'s input (the one received directly in round 1, and the two reported by the others in round 2). If at least 2 of them are equal, that value is set in the output vector; otherwise, a default (*null*) value is set.

The algorithm generalizes to any number *n* of processes, as long as the number *t* of faulty processes is strictly less than one third ($3t < n$). This limit is proven necessary.



Two generals problem

Two processes must reach an agreement (like in the consensus problem). The two processes are supposed correct, but the communication may lose messages. However, it is supposed that the communication does not become permanently faulty. This means that, at any point, if one process sends enough messages, one of them will be delivered (however, the source cannot know how many messages to send).

It is proven that no solution exists. The idea is to suppose that a protocol exists, to remove all unnecessary messages, and then to notice that for the last message, while it is necessary for the receiver in order to decide, the sender cannot know if it was delivered or not.

Radu-Lucian LUPŞA
2016-12-20

Lecture 12 - OpenCL programming

What is OpenCL?

OpenCL is a standard API for accessing, especially, the GPU for doing general purpose computing in a program (not necessarily graphics related). However, the OpenCL can be used for accessing other specialized co-processors and even the CPU itself.

OpenCL is a standard developped by the Khronos group. It is similar to the CUDA, developped by NVIDIA for similar purposes; however, OpenCL is implemented on a wider variety of hardware.

Concepts

Components

The executable is linked against a (device-independent) library (OpenCL library).

At run-time, the OpenCL library must be able to find the driver(s) for the hardware on the execution machine.

Concepts

There is code that executes on the host machine, and code that executes on the OpenCL device (normally the GPU).

The code executing on the host can be written in any language; OpenCL provides a C API, but a C++ wrapper is available, and also bindings for other languages.

The code executing on the OpenCL device must be written in C, with some extensions and with some restrictions.

The main code (the code executing on the host) normally does the following things:

1. Queries the device capabilities;
2. Gives the code to be executed on the OpenCL device as a string to the OpenCL API and asks the OpenCL to compile the code;
3. Sends the data to the device;
4. Requests the execution of the device code;
5. Retrieves back the results.

Work groups, work items, and memory types

The code on the device is launched in multiple instances (somewhat like distinct processes or threads). The instances are called *work items*. Each work item can retrieve its work item ID and use it to decide which part of the work it is supposed to perform.

Work items are grouped in work groups. Within a work group, the work items can synchronize and can access the so-called *local* memory. No synchronization is possible among work items in different work groups.

The memory available to the code on the device is divided into:

- *global memory* — this is available to all the work items. It can also be copied from and to the host.
- *local memory* — there is an instance of it within each work group, and it is available to all work items in the same group.
- *private memory* — there is an instance of it within each work item, and it is available only to the owner work item.

The host API has an operation allowing to launch a specified number of work groups, each consisting in a specified number of work items.

Execution queue and events

The operations requested by the host code — memory transfer and work item executions — are placed in a queue. The host API allows to request enqueueing of such an operation.

To synchronize operations, each operation has an associated *event*, that gets signalled when the operation completes. It is possible:

- for an enqueued operation, to specify a list of events to be completed before the operation is allowed to start;
- in the host code, it is possible to wait for an event to be signaled.

Simple examples

- opencl-sample1.cpp
- opencl-sample2.cpp

More complex cases — divide-and-conquer algorithms

There are important limitations on what a kernel can do. Most importantly:

- A kernel cannot spawn another kernel; therefore, the host code must know beforehand how many instances of a kernel to launch.
- GPU code cannot be recursive.

Problem: how can we do the recursive split and combine steps?

Solution: do the splits and combines in stages, and make the host code aware of the number of necessary instances and make it drive the split operations and combine operations. See opencl-bin-sum.cpp and opencl-mergesort.cpp