

Parallel programming on multicore architecture

Julien 'Lta' BALLEET



June 23, 2016

- ▶ Graduated from EPITECH, class of 2009
- ▶ Teacher and head of the “OpenSource” lab
 - ▶ Linux “multimedia” specialty
- ▶ Freelance developer and trainer
 - ▶ A lot of different things
- ▶ Production Engineer @ Facebook
 - ▶ *Operating Systems*: Core team
 - ▶ Linux system programming trainer
- ▶ Linux Audio Developer

- ▶ High proficiency in C
- ▶ Knowledge of C++
- ▶ GNU/Linux environment
 - ▶ The shell
 - ▶ General usage
 - ▶ The more, the better
- ▶ Linux development toolchain
 - ▶ g++/clang
 - ▶ gdb/lldb
 - ▶ make
 - ▶ man
- ▶ Be ready to invite quantum physics into computer science

- ▶ A low level approach
 - ▶ Hardware's architecture
 - ▶ Kernel's internals
 - ▶ Kernel's API and tools
- ▶ A practical approach
 - ▶ Examples
 - ▶ Exercices / Workshops

Introduction

System APIs

Designing parallel algorithms

Hardware and Kernel

Tools and Libraries

Resources

Summary

Introduction

System concepts

Parallel computing

The Unix/Linux model

System APIs

Designing parallel algorithms

Hardware and Kernel

Tools and Libraries

Resources

System concepts

What is a CPU ?

- ▶ Execute sequences of instruction
- ▶ Fetched from the memory through caches
- ▶ Operates on
 - ▶ Registers
 - ▶ Memory (through caches)
- ▶ Registers:
 - ▶ General purposes
 - ▶ PC
 - ▶ Stack pointer (BP/SP)
 - ▶ Various statuses

CPU Execution pipeline

Instr No.	Pipeline Stage						
	IF	ID	EX	MEM	WB		
1							
2							
3							
4							
5							
Clock Cycle	1	2	3	4	5	6	7

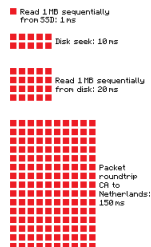
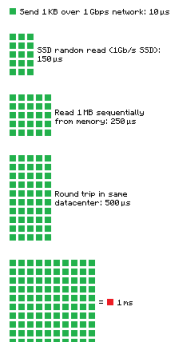
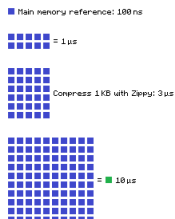
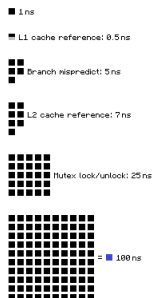
- ▶ IF = Instruction fetch
- ▶ ID = Instruction decode
- ▶ EX = Execute
- ▶ MEM = Memory access
- ▶ WB = Write back

Memory

- ▶ Arrays of bytes for the CPU to work on
- ▶ Organized in Hierarchy:
 - ▶ register > caches > main memory > disk / networks
- ▶ A few numbers¹:
 - ▶ Register: no latency (0-2 cycles)
 - ▶ Caches:
 - ▶ L1: 0.5 ns
 - ▶ L2: 7 ns
 - ▶ L3: 15 ns
 - ▶ Memory: 100 ns
 - ▶ SSD: 150 us
 - ▶ Disk read: 1ms
 - ▶ US ping: 150 ms

¹<https://gist.github.com/jboner/2841832>

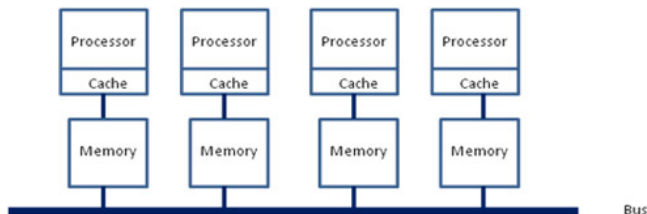
Latency Numbers Every Programmer Should Know



Source: <https://gist.github.com/2041832>

Non-Uniform Memory Access

- ▶ NUMA for short
- ▶ Memory is split in nodes
- ▶ Each core has direct access to only one node
- ▶ Access to other nodes is shared (slower)
- ▶ Different NUMA layouts
 - ▶ 1:1
 - ▶ 1:n



Virtual memory

- ▶ Real/Direct mode
 - ▶ Programs access memory directly
 - ▶ 0x00 → first byte of memory
 - ▶ Unsafe!
- ▶ Virtual/Protected mode
 - ▶ Programs addresses are mapped to different addresses
 - ▶ 0x00 → unknown
 - ▶ Safe !
 - ▶ Kernel maintains mapping (page directory)
 - ▶ MMU / TLB

Context switching

- ▶ Stopping the execution of a program to run another
- ▶ What is a context ?
 - ▶ Registers (pc, stack, ...)
 - ▶ Memory mappings (pagedir, TLB)
- ▶ Save the current context, load another
- ▶ Costly !

Multitasking

- ▶ Multiple program
- ▶ in their “own memory”
- ▶ running “at the same time”
- ▶ but actually sharing Memory and CPU Time
- ▶ Understanding the kernel is **vital**

Parallel computing

What is parallel computing ?

Parallel computing is a type of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved at the same time. There are several different forms of parallel computing: bit-level, instruction-level, data, and task parallelism. Parallelism has been employed for many years [...] but interest in it has grown lately due to the physical constraints preventing frequency scaling.

Wikipedia

Instruction-level parallelism

- ▶ Instruction pipelining
- ▶ Branch prediction
 - ▶ Predict *if-then-else* to prefetch code
- ▶ Hyper-threading
 - ▶ *Logical* core
 - ▶ Fill the bubbles in the instruction pipeline
- ▶ ...

Task/Data-level parallelism

- ▶ Task Level
 - ▶ Do X and Y in parallel
 - ▶ Pipeline pattern
- ▶ Data-level
 - ▶ Split X and split the parts in parallel
 - ▶ SIMD
 - ▶ Producer/consumer pattern
- ▶ Some overlap between the two
- ▶ More on this topic later

Why parallelism matters ?

- ▶ Frequency scaling vs multi-core
- ▶ Operating systems are multitask
- ▶ Latency control
- ▶ Important even for non-parallel programs

Some history

- ▶ Context switching for batches (*Leo III, UK, 1961*)
- ▶ Cooperative multitasking Time-sharing (*CTSS, MIT, 1961*)
 - ▶ IBM, MacOS, Win 3.1/9.x
- ▶ Preemptive multitasking (*Multics, Cambridge, 1964*)
 - ▶ Multics, very influential OS.
 - ▶ Linux, OSX, Win NT+
 - ▶ Most OSes today
- ▶ Virtual memory (*Atlas/Burroughs, 1961/1962*)
- ▶ Threads (*OS/360, IBM, 1967*)

The Unix/Linux model

POSIX

- ▶ *Many* UNIXes (Linux, BSDs, HP-UX, AIX, Solaris, ...)
- ▶ Common features and semantic
 - ▶ Virtual Memory
 - ▶ Preemptive multitasking
 - ▶ File system
 - ▶ Shell
 - ▶ Written in C
- ▶ Standardization effort
 - ▶ Portable Operating System Interface (IEEE, 1988-2008)
 - ▶ Single Unix Specification (Austin Group, 1997-2008)

Processes

- ▶ A running program
- ▶ Container of resources
 - ▶ Memory
 - ▶ File descriptors
 - ▶ Execution path
 - ▶ ...
- ▶ Coordinated by the kernel
 - ▶ Scheduler
 - ▶ Communication (pipes, shared memory)
 - ▶ Synchronization (locks, semaphores)
- ▶ Unix parallelism is process based

fork()

- ▶ The Unix process creation model:
 - ▶ Process duplication: *fork()*
 - ▶ Process image replacement: *exec()* family
 - ▶ Take care of lifecycle management !
- ▶ Creating processes is *cheap*
 - ▶ unlike Windows
 - ▶ Many optimizations (COW, ...)

fork() example

```
pid_t pid = fork();
if (pid) {
    // Parent (original process)
    int status;
    wait(&status); // Ask if you don't know
} else {
    // Child (new process)
    char *newenviron[] = { NULL };
    char *newargv[] = { NULL };

    // Replace the current process by "/bin/ls"
    execve("bin/ls", newargv, newenviron);
}
```

Threads

- ▶ a.k.a *Lightweight processes*
- ▶ *Relatively recent* in UNIX history
- ▶ Task with shared resources
 - ▶ Memory space
 - ▶ File descriptors
 - ▶ ...
- ▶ Multiple execution path within a process
 - ▶ 1 process: 1-n threads
- ▶ Separate stack and registers

clone()

- ▶ In Linux, *fork()* wraps *clone()*
- ▶ Duplicate tasks
 - ▶ Task: kernel name for execution context + resources
 - ▶ Something *schedulable*
- ▶ Choose what you share or copy/reset:
 - ▶ Memory
 - ▶ File descriptors
 - ▶ Network namespace, ...
- ▶ Share everything: You're a thread !
 - ▶ *and a hippie :)*

Summary

Introduction

System APIs

POSIX Threads

C++ 11

A quick tour of process based
parallelism

Designing parallel algorithms

Hardware and Kernel

Tools and Libraries

Resources

Threads API

- ▶ Lifecycle management
 - ▶ Creation
 - ▶ Destruction
- ▶ Synchronization
 - ▶ Lifecycle synchronization
 - ▶ Share resources
 - ▶ Share work
- ▶ Tuning
 - ▶ Scheduling
 - ▶ Stack
 - ▶ Signals
 - ▶ ...

POSIX Threads

Introducing the POSIX Threading API

- ▶ *pthread*s for short
 - ▶ *pthread_*xxx functions
 - ▶ Man pages: **man 7 pthreads**
- ▶ Primitives
 - ▶ Lifecycle management
 - ▶ Synchronization
 - ▶ Tuning

Why the low-level API ?

- ▶ The more you know
 - ▶ Used in a lot of code
 - ▶ Precise documentation
 - ▶ System semantic and behavior
- ▶ More control
- ▶ Not everything is wrapped
- ▶ Access to non-portable features

Lifecycle management

- ▶ Creation
 - ▶ Process creation
 - ▶ *pthread_create()* with a function pointer
- ▶ Destruction
 - ▶ The thread function returns
 - ▶ *pthread_exit()* exits the current thread
 - ▶ *pthread_cancel()* ask for another thread to exit

Cancellation

- ▶ *pthread_cancel()* sends cancellation request
- ▶ Thread can configure their policy (*pthread_setcanceltype()*)
 - ▶ Asynchronous: Exit ASAP
 - ▶ Deferred: Exit on a cancellation point
- ▶ Can be disabled with *pthread_setcancelstate()*
- ▶ Cleanup handlers with *pthread_cleanup_push*
- ▶ Cancellation points:
 - ▶ *pthread_testcancel()*
 - ▶ Certain syscalls: see “man 7 pthreads”

Synchronization

- ▶ MUTual EXclusions (aka mutex or lock)
 - ▶ Thread can acquire/release it
 - ▶ Only 1 thread can acquire it at the time
 - ▶ Helps protect shared resources
- ▶ RW Locks
 - ▶ Acquired for Read OR Write
 - ▶ Multiple reader OR one writer at the time
- ▶ Conditions / Signals
 - ▶ Wait for a condition to happen
 - ▶ Signal (awake threads) when a condition happen
- ▶ Barriers
 - ▶ Wait for N other threads to reach a certain point

Mutex

- ▶ Helps protect a shared resource from concurrent access.
- ▶ Lifecycle
 - ▶ *pthread_mutex_init()*
 - ▶ *pthread_mutex_destroy()*
 - ▶ `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
- ▶ Lock
 - ▶ *pthread_mutex_lock()*
 - ▶ *pthread_mutex_timedlock()*
 - ▶ *pthread_mutex_trylock()*
- ▶ Unlock: *pthread_mutex_unlock()*

RW Locks

- ▶ Helps protect a shared resource from concurrent access.
- ▶ For *read-mostly* data
- ▶ Lifecycle
 - ▶ `pthread_rwlock_init()`
 - ▶ `pthread_rwlock_destroy()`
 - ▶ `pthread_rwlock_t rwl = PTHREAD_RWLOCK_INITIALIZER;`
- ▶ Lock as a writer:
 - ▶ `pthread_rwlock_wrlock()`
 - ▶ `_trywrlock()`, `_timedwrlock()`
- ▶ Lock as a reader:
 - ▶ `pthread_rwlock_rdlock()`
 - ▶ ...
- ▶ Unlock: `pthread_rwlock_unlock()`

Conditions

- ▶ 1-n thread(s) wait(s) on a condition
- ▶ Another thread wakes one or all waiting thread(s)
- ▶ Requires a mutex
- ▶ API calls:
 - ▶ *pthread_cond_init()* and *pthread_cond_destroy()*
 - ▶ Wait: *_cond_wait()* and *_cond_timedwait()*
 - ▶ Wake one: *_cond_signal()*
 - ▶ Wake all: *_cond_broadcast()*

Barriers

- ▶ A barrier is initialized with an int “N”.
- ▶ Threads may *wait* on the barrier.
- ▶ When “N” threads are waiting, they’re all resumed.
- ▶ One thread get a special value returned, the other get 0.
- ▶ API calls:
 - ▶ *pthread_barrier_init()* and *_destroy()*
 - ▶ *pthread_barrier_wait()*
- ▶ != Memory barriers

Spinlocks

- ▶ Busy-wait based locks
 - ▶ Avoid context switches
 - ▶ Waste CPU Time
- ▶ Good for short lived locks
- ▶ API Calls:
 - ▶ *pthread_spin_init* and *_destroy*
 - ▶ *_lock()* and *_unlock()*

Utils

- ▶ Wait for a thread to terminate: *pthread_join()*
- ▶ Get called before/after fork: *pthread_atfork()*
- ▶ Get the current thread: *pthread_self()*
- ▶ Compare threads: *pthread_equals()*
- ▶ Call a function once per process: *pthread_once()*
- ▶ Relinquish the CPU: *pthread_yield()* or *sched_yield()*

Thread Local Storage

- ▶ Thread specific pointers
- ▶ Non thread-safe library (codec, interpreter, ...)
- ▶ API Calls:
 - ▶ Create a key: *pthread_key_create()*
 - ▶ *pthread_key_destroy()*
 - ▶ Store a pointer for key/thread: *pthread_setspecific()*
 - ▶ Get the pointer: *pthread_getspecific()*

TLS Example

```
static pthread_key_t key;
static pthread_once_t key_once = PTHREAD_ONCE_INIT;

static void make_key() {
    (void) pthread_key_create(&key, NULL);
}

func() {
    void *ptr;

    (void) pthread_once(&key_once, make_key);
    if ((ptr = pthread_getspecific(key)) == NULL) {
        ptr = malloc(OBJECT_SIZE);
        // ...
        (void) pthread_setspecific(key, ptr);
    }
    // ...
}
```

C++ 11

threading in std::

- ▶ C++11 adds threading support in the stdlib
- ▶ Clean C++
- ▶ Support portable behaviors and primitives
- ▶ Supported by most compilers/stdlib:
 - ▶ gcc/libstd++
 - ▶ clang/libc++
 - ▶ MSVC 2012+
 - ▶ icc: meh !
- ▶ boost::thread is close-enough

Features

- ▶ `std::thread`
- ▶ `std::mutex`
- ▶ `std::atomic`
- ▶ `std::condition_variable`
- ▶ `std::future`
- ▶ `thread_local` keyword

std::thread

- ▶ Spawn a thread at construction
 - ▶ No return value
 - ▶ No way to terminate externally
- ▶ Movable but non-copyable
- ▶ `::join()`
- ▶ `::get_id()`
- ▶ `::native_handle()`
- ▶ `::hardware_concurrency()`

std::thread example

```
void call_from_thread(int tid) {
    std::cout << "Launched by thread " << tid << std::endl;
}

int main() {
    std::thread t[num_threads];

    //Launch a group of threads
    for (int i = 0; i < num_threads; ++i) {
        t[i] = std::thread(call_from_thread, i); //C++11 magic !
    }

    // Join the threads
    for (int i = 0; i < num_threads; ++i) {
        t[i].join();
    }
}
```

std::mutex

- ▶ Basic locking: *std::mutex*
 - ▶ *::lock()*
 - ▶ *::try_lock()*
 - ▶ *::unlock()*
- ▶ Locking with timeout: *std::timed_mutex*
 - ▶ *::try_lock_for()*
 - ▶ *::try_lock_until()*
- ▶ Recursive locking: *std::recursive_mutex()*
 - ▶ Caution

Scope-based locking

- ▶ Nice and safe.
- ▶ `std::lock_guard`
 - ▶ Locks at construction
 - ▶ Unlocks at destruction
- ▶ `std::unique_lock`
 - ▶ Mutex ownership proxy.
 - ▶ Optionally locks the mutex at construction
 - ▶ Movable, non-copyable
 - ▶ `::lock()`, `::try_lock()`, `::unlock()`

std::mutex example

```
static std::mutex mtx;

void call_from_thread(int tid) {
    // Access to stdout is synchronized
    std::lock_guard<std::mutex> lock;
    std::cout << "Launched by thread " << tid << std::endl;
}

int main() {
    std::thread t[num_threads];

    for (int i = 0; i < num_threads; ++i) {
        t[i] = std::thread(call_from_thread, i); //C++11 magic !
    }
    for (int i = 0; i < num_threads; ++i) {
        t[i].join();
    }
}
```

std::condition_variable

- ▶ Requires a *std::unique_lock*
- ▶ *::notify_one()*
- ▶ *::notify_all()*
- ▶ *::wait()*, *::wait_for()* and *::wait_until()*

std::atomic

- ▶ Various integer types
- ▶ Memory synchronization
- ▶ Standard atomic operations
 - ▶ Add, Subtract, Store
 - ▶ Compare and Swap
 - ▶ And, Or, Xor
- ▶ Base for lock-free algorithms

std::atomic example

```
std::atomic<bool> running = true;

void thread() {
    while (running) {
        std::this_thread::yield()
    }
}

void stop_thread() {
    // Do stuff 1
    running = false;
    // Do stuff 2
}
```

std::future

- ▶ Framework for deferred execution
 - ▶ Lazy evaluation
 - ▶ Thread pool
 - ▶ Manual scheduling
- ▶ std::promise: Stores the data
- ▶ std::future: Wait for the data
- ▶ std::async: Run a method asynchronously, returns a future

std::async example

```
template <typename ITER>
int parallel_sum(ITER beg, ITER end)
{
    auto len = end - beg;
    if(len < 420)
        return std::accumulate(beg, end, 0);

    ITER mid = beg + len/2;
    auto handle = std::async(std::launch::async,
                             parallel_sum<ITER>, mid, end);
    int sum = parallel_sum(beg, mid);
    return sum + handle.get();
}

int main()
{
    std::vector<int> v(42000, 1);
    std::cout << "The sum is " << parallel_sum(v.begin(), v.end()) << '\n';
    // The sum is 42000
}
```

thread_local

- ▶ Storage specifier keyword
- ▶ Can be combined with
 - ▶ *static*
 - ▶ *extern*
- ▶ Use Thread local storage

A quick tour of process based parallelism

Why care about processes ?

- ▶ Original UNIX-way
- ▶ Very good support
- ▶ Safer and more robust:
 - ▶ Isolate segfaults
 - ▶ Less coupling
- ▶ Isolate incompatible licenses
- ▶ Complementary with threads

Two APIS

- ▶ System V
 - ▶ Older
 - ▶ Akward API
 - ▶ Widely supported
- ▶ POSIX
 - ▶ Not fully supported everywhere
 - ▶ More traditional semantic (open, mmap, ...)
- ▶ We'll talk about SysV APIs here

Shared memory

- ▶ Pages of memory shared between 2 or more processes
- ▶ API Calls:
 - ▶ Open/Create a segment: *shmget()*
 - ▶ Map the segment in memory: *shmat()*
 - ▶ Unmap the segment: *shmdt()*
 - ▶ Delete the segment: *shmctl()* with the *IPC_RMID* flag
- ▶ spinlocks and barriers can be shared via shared memory

Semaphores

- ▶ Interprocess synchronization primitive
- ▶ Positive or null integer with 3 operations:
 - ▶ Add another integer
 - ▶ Wait until value is null
 - ▶ Wait until value is bigger than N , then subtract N

Semaphores API

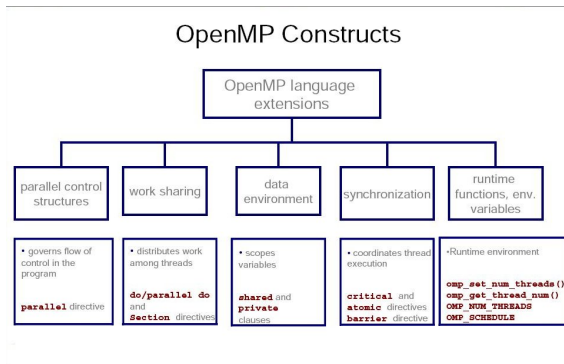
- ▶ Manipulated as sets
- ▶ Create/Open a semaphore set: *semopen()*
- ▶ Operates on the semaphore set values: *semop()*
- ▶ Deletes a semaphore set: *semctl()* with *IPC_RMID* operation

Message Queues

- ▶ Exchange arbitrary messages between processes
- ▶ API Calls:
 - ▶ Create/Open a queue: *msgget()*
 - ▶ Post a message on the queue: *msgsnd()*
 - ▶ Receive a message from a queue: *msgrcv()*
 - ▶ The message is removed from the queue
 - ▶ Delete a queue: *msgctl()* with *IPC_RMID* operation

OpenMP

- ▶ Open Multi Processing
- ▶ Higher-level alternative to pthreads
- ▶ Integrated into compiler



MPI

- ▶ Message Passing Interface
- ▶ Process based parallel computing API
- ▶ Targeted to clusters
- ▶ Feature rich
 - ▶ Message passing
 - ▶ Synchronization
 - ▶ Data sharing
 - ▶ I/O

Summary

Introduction

System APIs

Designing parallel algorithms

Know the problem

Partitioning

Task communication

Load balancing and Tuning

Common patterns

Hardware and Kernel

Tools and Libraries

Resources

- ▶ Define a repeatable process
- ▶ Lay out some terminology
- ▶ Identify classical patterns
- ▶ Observe some real world examples

Know the problem

Serial problem

Premature optimization is the root of all evil -Knuth

- ▶ Solve the business domain problem first
- ▶ Best done as a serial problem
 - ▶ Existing code ?
 - ▶ Naive implementation ?
- ▶ Easier to grasp
- ▶ Easier to get data

- └ Designing parallel algorithms
 - └ Know the problem

Get data !

Use the data

- ▶ or do an algorithmic complexity evaluation
- ▶ Identify the time consuming parts
- ▶ Where are the hotspots ?
 - ▶ 80% of the time in 20% of the code
- ▶ Where are the bottlenecks ?
 - ▶ IO
 - ▶ I'm looking at you hard drive

Evaluate other options

- ▶ Parallel algorithms are hard
- ▶ Are there other algorithms ?
- ▶ A good-enough heuristic ?
- ▶ Already parallel libraries available ?
- ▶ Fear the threads and avoid them if possible

Is it parallelizable ?

- ▶ Real-time constraints
 - ▶ Scheduling isn't really deterministic
- ▶ Control plane latency
- ▶ Communication-intensive ²
- ▶ **Data dependencies**

²https://en.wikipedia.org/wiki/Amdahl's_law

Is it parallelizable ?

- ▶ 3D rendering ?
- ▶ Game of Life ?
- ▶ Sorting algorithms ?
- ▶ Search algorithms ?
- ▶ Data compression ?
- ▶ Google Page rank :) ?

Partitioning

Drawing the dependency graph

- ▶ Input \rightarrow Program \rightarrow Output
- ▶ Moving data through a graph of algorithms
- ▶ Let's draw this graph
 - ▶ All the inputs
 - ▶ All the intermediary data
 - ▶ All outputs
- ▶ Edge = Data dependency

Data partitioning

- ▶ Which data can be split ?
- ▶ How do we split them ?
 - ▶ http requests ?
 - ▶ images ?
 - ▶ financial data analysis ?

Functional partitioning

- ▶ What are the functional blocks within the program ?
- ▶ Within the algorithms ?

Partition

- ▶ Using the partitioning/graph above
- ▶ and the gathered data
- ▶ Group/split items into tasks

Examples

- ▶ Genetic algorithm
- ▶ Search algorithm
- ▶ Webpage rendering
- ▶ Video transcoder

Task communication

Communication

- ▶ Data dependency means communication
- ▶ Don't forget the control plane
- ▶ How do they communicate ?
 - ▶ Message passing ?
 - ▶ Synchronous ?
 - ▶ Asynchronous ?
 - ▶ Data sharing/copy ?

Models

- ▶ 1-1
- ▶ 1-n
 - ▶ Broadcast
 - ▶ Scatter
 - ▶ Gather
 - ▶ Reduce
- ▶ n-n (hard to parallelize)

Cost

- ▶ What's the cost of communicating ?
 - ▶ Always an overhead
 - ▶ **Synchronization**
 - ▶ Data formatting ?
 - ▶ Data copy ?
 - ▶ The hidden cost: coupling
- ▶ **Trade-off:** *Latency vs Bandwidth*
- ▶ Overhead and diminishing returns: what's the balance?

Food for thought

- ▶ What is the cost of
 - ▶ Synchronous communication ? (CPU Time, code complexity, ...)
 - ▶ Asynchronous communication ?
 - ▶ Data copying ?
 - ▶ Data sharing ?

Load balancing and Tuning

Back on the real world

- ▶ A fancy graph ? Neat !
- ▶ Ideas of the communications models ? Good !
- ▶ How does it map to an actual program ?
- ▶ Running on an actual machine ?
- ▶ Let's group thing into tasks

- ▶ The goal:
 - ▶ Use 100% of the CPUs
 - ▶ Doing actual work
- ▶ You need actual data or you're just guessing
 - ▶ Pre-existing code
 - ▶ Naive implementation
 - ▶ Individual algorithms benchmarks
 - ▶ Algorithmic complexity (:-/)

- ▶ Split large task
 - ▶ Data partitioning
 - ▶ Functional partitioning
- ▶ Aggregate smaller tasks
- ▶ Measure the overhead
- ▶ Preserve data locality
- ▶ Load balance
 - ▶ Static ?
 - ▶ Dynamic ?

Tuning

- ▶ No magic here. Only experiments and intuition
 - ▶ And you should avoid depending on intuition
- ▶ Experiment with different strategies
 - ▶ Load-balancing
 - ▶ Communication
 - ▶ Synchronization
- ▶ Measure
- ▶ *Rinse*
- ▶ *Repeat*

Examples

- ▶ Genetic algorithm
- ▶ Search algorithm
- ▶ Webpage rendering
- ▶ Video transcoder

Common patterns

- ▶ Super-scalar sequences
- ▶ Pipelining
- ▶ Speculative selection
- ▶ Map/Reduce
- ▶ (Gather: Map + Read. Allow for some optimizations)
- ▶ Partition
- ▶ Scan
- ▶ Pack

Summary

Introduction

System APIs

Designing parallel algorithms

Hardware and Kernel

Some hardware considerations

Scheduling

Syscall and real-time

Memory management

Errors and Performance

Tools and Libraries

Resources

Some hardware considerations

Processing architectures

- ▶ Mono-processor (a.k.a The good ol' days)
 - ▶ One CPU, One memory bus
 - ▶ Time-slicing
- ▶ Multi-processor
 - ▶ SMP, Symmetric Multi-Processing
 - ▶ Many CPUs, One Memory bus
- ▶ NUMA
 - ▶ Memory bus is the bottleneck
 - ▶ Many CPUs, Many memory bus
 - ▶ One shared memory bus


Caches

- ▶ No direct memory accesses
- ▶ Always go through caches
- ▶ *Write-through vs Write-back*
- ▶ *Inclusive vs exclusive*
- ▶ Different levels
 - ▶ L1: 2 per-core (data/instructions)
 - ▶ L2: shared between some cores
 - ▶ L3: Shared between all cores

Cache lines

- ▶ Memory caching granularity
 - ▶ Usually 64 bytes ($8 * int64_t$)
- ▶ Cache associativity
 - ▶ Direct-mapping: 1 possible line for every memory address
 - ▶ n-way associative: n possible lines
 - ▶ Fully associative: Any line can cache anything
 - ▶ Modern CPUs: n-way associative (L1: 2/4, L2: 8/16)
- ▶ Line sharing and MESI Protocol
 - ▶ Modified, Exclusive, Shared, Invalid

Memory barriers

- ▶ Cache invalidation is “*slow*”
- ▶ → Done asynchronously
 - ▶ Store buffer
 - ▶ Invalidation queue
- ▶ Operation on a CPU takes time to propagate
- ▶ 
- ▶ Memory barriers are “instructions” which synchronously flush the buffers
- ▶ Also prevent some instruction reordering
- ▶ Very important with atomic operations

Pre-fetching and branch prediction

- ▶ Access to memory is slow
- ▶ Instruction execution is fast
- ▶ Let's preload the program instructions
- ▶ Lots of jump
 - ▶ Control statements
 - ▶ Function calls
- ▶ The compiler and CPU tries to predict jumps

Out of order execution

- ▶ What you see is NOT what you get
 - ▶ Compiler optimizations
 - ▶ Processor optimizations !
- ▶ Instructions are reorganized
- ▶ *Out of order execution or dynamic instruction scheduling*
 - ▶ Instruction queue
 - ▶ Executed based on data availability
- ▶ Superscalar architecture
 - ▶ Instruction level parallelism
 - ▶ Use of multiple execution unit

SMT

- ▶ Simultaneous Multi Threading
- ▶ A.k.a *HyperThreading*
- ▶ 2 (or more) threads on one core.
- ▶ Increase usage of
 - ▶ Dynamic instruction scheduling
 - ▶ Super-scalar
- ▶ 2 logical core, only one real core
- ▶ 0-25% performance gain
- ▶ Can reduce performance !

Scheduling

Tasks, threads and processes

- ▶ In the kernel, everything is a *task*³
 - ▶ Execution context that can be allocated CPU time
 - ▶ Something *schedulable*
 - ▶ With associated resources: Memory⁴, Files⁵, ...
- ▶ Used for userland and kernel
 - ▶ Userland task → thread
 - ▶ Threads w/ shared memory and fds → process
- ▶ One of the 3 kernel's core concepts

³task_struct

⁴mm_struct

⁵files_struct

What is scheduling ?

- ▶ The art of sharing the CPU(s)
- ▶ NP-complete problem
- ▶ Kernel's role
- ▶ What is running ? Where ? For how long ?
- ▶ Are all the tasks equals ?
- ▶ What is the cost ?

The cost of scheduling

- ▶ Context switching
 - ▶ To the kernel
 - ▶ To another process
- ▶ Latency vs throughput
- ▶ When to schedule is important
 - ▶ Suspend a task holding mutexes ?
 - ▶ Before an I/O operation ?

When does scheduling happen ?

- ▶ When a thread waits (I/O, mutex, ...)
- ▶ When returning to user-space from a syscalls
- ▶ When returning from an interrupt
 - ▶ Device event (keyboard, network, ...)
 - ▶ Timer event

Scheduling classes

- ▶ What are scheduling classes:
 - ▶ Different algorithms
 - ▶ Different priority
- ▶ The 'normal' classes:
 - ▶ *SCHED_NORMAL* (a.k.a *SCHED_OTHER* in userland)
 - ▶ *SCHED_BATCH*. *SCHED_NORMAL* for CPU-intensive tasks
 - ▶ *SCHED_IDLE*. Very low priority. Only run when CPU would idle.
- ▶ The 'real-time' classes:
 - ▶ *SCHED_FIFO*
 - ▶ *SCHED_RR*
 - ▶ *SCHED_DEADLINE*

CFS

- ▶ The Completely Fair Scheduler⁶
- ▶ Current scheduler for *SCHED_NORMAL*
- ▶ Models an *ideal* CPU, perfectly shared
- ▶ CPU usage is tracked with nano-second precision
- ▶ rbtree / $O(\log n)$
- ▶ *Sleeper fairness*

⁶Since 2.6.23

Other features

- ▶ Process priorities (aka *niceness*)
 - ▶ between process of the same class
- ▶ Processor affinity

Spinlock, mutex, futex

- ▶ The basic locking primitive is the spinlock
 - ▶ Waste CPU cycles until ready
 - ▶ Atomic variable
- ▶ Mutex = spinlock + scheduler cooperation
- ▶ Futex⁷ stands for Fast Userland muTEX
 - ▶ Library and syscalls to implement *pthread_mutex_XXX()*
 - ▶ No syscall when no contention

⁷man 2/7 futex

Syscall and real-time

What's a syscall ?

- ▶ The kernel API
 - ▶ i.e: The only way for the userland to talk to the kernel
- ▶ Use processor's software interrupt mechanism (*int 80*)
 - ▶ Context-switch to a predefined kernel handler
 - ▶ Back to ring 0
 - ▶ The kernel does its thing
 - ▶ Context-switch back to the calling userland thread
- ▶ No way for the user thread to resume until completion
- ▶ Preemption point

Real time programming

- ▶ Deterministic programming
- ▶ Response time guarantees (deadlines)
- ▶ Different types:
 - ▶ Hard: Airplanes, nuclear reactor, car control
 - ▶ Firm: Sound on your computer
 - ▶ Soft: Audio/Video streaming

Worst case scenario

- ▶ RT system is defined by it's worst-case
- ▶ Investigate worst-case execution time.
 - ▶ Is it known ?
 - ▶ Is it good enough ?
 - ▶ Can you control it ?
- ▶ Almost everything is optimized for the average-case

Practically ?

- ▶ No (*blocking*) syscalls
 - ▶ Goodbye *read()* and *write()*
 - ▶ You need to know about the kernel's internals
- ▶ Or libc functions which are doing syscalls
 - ▶ Goodbye *malloc()* and *free()*
 - ▶ ...
- ▶ Synchronization is critical
 - ▶ Goodbye *mutex_lock()*
 - ▶ Hello lock-free data structures
- ▶ Takes the rest of the system into account
- ▶ Evaluate the worst case complexity

Why do we care about real-time ?

- ▶ Make us think about
 - ▶ Latency
 - ▶ p99
- ▶ The field pioneered:
 - ▶ Asynchronous I/Os
 - ▶ Lock-free programming
 - ▶ ...
- ▶ What would you do if you **had** to have a strict latency

Memory management

Virtual memory (bis)

- ▶ Controls the MMU
 - ▶ Page directory (mapping between real/virtual addresses)
 - ▶ Translation Lookaside Buffer
- ▶ Swap
- ▶ Pretty complex system
 - ▶ Kernel's own needs
 - ▶ User memory
 - ▶ Lots of sharing
 - ▶ Threads: heap vs stack
 - ▶ SYSV/POSIX shared memory
 - ▶ *mmap()*
 - ▶ DMA
- ▶ Memory sharing (SYSV, threads, mmap, ...)

Memory pressure

- ▶ Balance between different memory users
 - ▶ Userland programs
 - ▶ Drivers
 - ▶ Various caches (fs, network, ...)
- ▶ What happens if there are no memory:
 - ▶ Disk write ?
 - ▶ User request ?
 - ▶ Interrupt handler ?

NUMA

- ▶ Memory access from a different NUMA node is slow
- ▶ Each NUMA-node is managed separately
- ▶ Scheduler is NUMA-aware
 - ▶ It tries to keep a task on the same CPU
- ▶ Memory allocation is NUMA-aware
 - ▶ It tries to allocate on the local NUMA-node
- ▶ User can help:
 - ▶ *libnuma* and *numactl*
 - ▶ *set_mempolicy()*, ...
 - ▶ *sched_setaffinity()*

Swapping

- ▶ Use disk as memory
- ▶ **Very** slow
 - ▶ Memory access: 100 ns
 - ▶ Disk access: 1 ms = 1 000 000 ns
- ▶ Should never happen...
- ▶ ... but it's better than a failed allocation
- ▶ `echo -15 > /proc/2592/oom_adj`
- ▶ `man 2 mlock()`

Errors and Performance

Classical problems

- ▶ Deadlock
- ▶ Live lock
- ▶ Priority inversion
 - ▶ Priority inheritance: `pthread_mutexattr_setprotocol()`
- ▶ Convoying
- ▶ Signal handling
 - ▶ No mutex allowed

Lock contention

- ▶ Time spent waiting on lock ?
- ▶ Measure
- ▶ Re-design
- ▶ Removes locks
 - ▶ Split data (differently)
 - ▶ DCLP: Double Check Locking Pattern
 - ▶ Lock-free

Cache-fighting

- ▶ Don't forget about cache
 - ▶ Lines
 - ▶ Associativity
 - ▶ Sharing
 - ▶ Coherence
- ▶ Caches are fighting when accessing the same data
- ▶ Split the data
- ▶ Duplicate if necessary

False sharing

- ▶ Cache-fighting for different data
- ▶ on the same cache lines !
- ▶ Split/duplicate the data
- ▶ Add some padding
- ▶ Use a cache-aware memory allocator

Summary

Hardware and Kernel

Introduction

Tools and Libraries

System APIs

Debug tools

Performance tools

Designing parallel algorithms

Intel TBB

Resources

I hope you like CLI tools

- ▶ We're **engineers**, we don't care about GUI
- ▶ We care about
 - ▶ Features
 - ▶ Performance
 - ▶ Efficiency
 - ▶ Automation
- ▶ CLI tools are **much** better for that
- ▶ You can find a GUI later

Open Source only

- ▶ Widely available
 - ▶ Your knowledge have more value
- ▶ Independence from vendor
 - ▶ Remember Borland C++ ? ⁸
- ▶ Usually better maintained and documented
 - ▶ Source is the best documentation
- ▶ Adaptable to your needs

⁸lol

Debug tools

`gdb`

- ▶ The venerable debugger ⁹
- ▶ Linux/Unix standard
- ▶ Loads of features
- ▶ Very stable
- ▶ Part of the gcc project
- ▶ Large official documentation
- ▶ Lots of resources on the web

⁹<http://www.gnu.org/music/gdb-song.html>

lldb

- ▶ The newcomer
- ▶ Part of the clang project
 - ▶ More modern codebase
 - ▶ Very active community
- ▶ Pretty mature for its age
- ▶ Better C++ support
- ▶ Documentation is not so good

QuickStack

- ▶ Get a running process stacktrace
- ▶ Very low-overhead
- ▶ Handy for production
 - ▶ Is it deadlocked ?
 - ▶ Where ?
 - ▶ Quick and dirty profiler
- ▶ requires `-fno-omit-frame-pointer`

Valgrind's memcheck

- ▶ Not exactly related to threading but...
- ▶ It's **Amazing**. It saves lives, kitten and the queen
- ▶ Very slow (that's the price of greatness)
- ▶ Check your code for any memory related bug
 - ▶ Out of bound access
 - ▶ Uninitialized access
 - ▶ Double free
 - ▶ Segfaults
 - ▶ Leaks

Valgrind's Helgrind

- ▶ Detects errors in your multithreaded program:
 - ▶ pthread API errors
 - ▶ Uninitialized mutexes/conditions/...
 - ▶ Recursive locking on non-recursive mutexes
 - ▶ ...
 - ▶ *Potential* deadlocks
 - ▶ Possible data races
 - ▶ Data accessed by two threads with no mutex

Valgrind's DRD

- ▶ Another threading error detector
- ▶ memcheck should pass
- ▶ Mostly the same errors as Helgrind, plus:
 - ▶ False-sharing
 - ▶ Lock contention
- ▶ Different engine than Helgrind
 - ▶ None is better

userspace lockdep

- ▶ lockdep: Kernel deadlock detector / lock validator
- ▶ liblockdep: userland version
- ▶ Similar to Helgrind
 - ▶ Doesn't check for data races
 - ▶ Much lighter and faster
 - ▶ For production / embedded

clang's sanitizers

- ▶ A suite of sanitizer (memory, thread, ...)
- ▶ Based on instrumentation and runtime library
- ▶ Usually much faster than Valgrind's
- ▶ Less precise detection
- ▶ More precise and readable output
- ▶ Need to recompile all the dependencies

ASAN

- ▶ Address SANitizer
- ▶ Much faster than Valgrind
- ▶ Detects:
 - ▶ Out-of-bounds access
 - ▶ Use after free
 - ▶ Invalid free
 - ▶ Memory leak (experimental)

Other sanitizers

- ▶ Thread Sanitizer: Detects data races
- ▶ Memory Sanitizer: Detects uninitialized reads
- ▶ Various undefined behavior sanitizers
 - ▶ Alignment
 - ▶ Numeric casts and overflows
 - ▶ Object casts and dynamic typing
 - ▶ ...¹⁰
- ▶ A few other advanced stuff

¹⁰<http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html#ubsan-checks>

clang's Thread Safety Analysis

- ▶ Compile-time checker...
- ▶ ...using your instrumentation
 - ▶ Variable is *GUARDED_BY(mutex)*
 - ▶ Method *REQUIRES(mutex)* to be locked
 - ▶ Powerful *CAPABILITIES* system
- ▶ Time consuming
- ▶ Enforce code quality and safety

Performance tools

Optimizations always bust things, because all optimizations are, in the long haul, a form of cheating, and cheaters eventually get caught.

- Larry Wall

Selection of tool

- ▶ Open source and freely available
- ▶ Stable enough
- ▶ Nothing really multi-threading specific
- ▶ *but* amazing options available on Linux

gprof

- ▶ gcc's profiler
- ▶ Based on code instrumentation
- ▶ Linux historical profiler
- ▶ Stable and well supported
- ▶ Doesn't work well with threads :-/

oprofile

- ▶ The first Linux system sampling profiler
- ▶ No code instrumentation
- ▶ Can profile your app, or the whole system
- ▶ Very low overhead compared to gprof (1-10%).
- ▶ Is being superseded by *perf*
 - ▶ But reports lower level data
 - ▶ Also a matter of taste

Valgrind's Cachegrind

- ▶ Based on a simulation
- ▶ Cache profiling
 - ▶ Find cache misses
 - ▶ **Line granularity !!!**
- ▶ Branch prediction
 - ▶ Find branch misprediction
 - ▶ Helps you place hints
 - ▶ Get those cycles back

Valgrind's Callgrind

- ▶ *Classical* profiler
- ▶ Valgrind → pretty slow
- ▶ Pretty precise
- ▶ Includes Cachegrind features
- ▶ Good threading support *-separate-thread=yes*

System monitoring tool

- ▶ On performance critical application, system is important
- ▶ vmstat: Various metrics of the system
 - ▶ dstat: Same with nicer UI
 - ▶ sar: Very detailed
- ▶ iostat: I/O monitoring
- ▶ mpstat: Scheduling monitoring
- ▶ strace: Trace syscalls made by process/thread
- ▶ slaptop: Monitor kernel memory usage

mutrace

- ▶ mutrace is a mutex profile
- ▶ Very fast
- ▶ Mutex locking stats
- ▶ Lock contention

perf

- ▶ One perf tool to **rule them all**
- ▶ System sampling profiler
 - ▶ Your app
 - ▶ The kernel
- ▶ Access all the kernel tracepoints and data
- ▶ *Extremely* powerful
- ▶ Pretty complex
- ▶ Flame Graphs

SystemTap

- ▶ Another level above perf
- ▶ Script based
- ▶ Debugger features
- ▶ Dynamic tracer compilation and insertion
- ▶ Comparison with perf:
<https://sourceware.org/systemtap/wiki/SystemtapDtraceComparison>

Intel TBB

What is it ?

- ▶ C++ template library for parallelism
- ▶ Nice license (same as libstdc++)
- ▶ Nice features
 - ▶ Thread safe / Lock free containers
 - ▶ Parallelized loops
 - ▶ Memory allocators
 - ▶ Design patterns
- ▶ Parts are being obsoleted by C++11

Summary

Introduction	Hardware and Kernel
System APIs	Tools and Libraries
Designing parallel algorithms	Resources

- ▶ I still need to search through my notes to complete this section
- ▶ More to come...

Parallel computing

- ▶ Structured Parallel Programming. McCool, Robinson, Reinders.
- ▶ Patterns for Parallel Programming. Mattson, Sanders, and Massingill.
- ▶ Dr.Dobb's series on Parallel Pattern:
<http://www.drdobbs.com/database/parallel-pattern-1-superscalar-sequences/223101511>

Hardware

- ▶ Intel 64 and IA-32 Architectures Optimization Reference Manual: <http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html>
- ▶ Memory Barriers: a Hardware View for Software Hackers <http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.07.23a.pdf>
- ▶ Wikipedia on CPU Cache: https://en.wikipedia.org/wiki/CPU_cache
- ▶ HyperThreading: <https://www.cs.uaf.edu/2011/spring/cs641/proj1/rltorgerson/>

Performance

- ▶ Brendan Gregg Performance's gold mine:
<http://www.brendangregg.com/>
- ▶ CppCon 2015: Fedor Pikes "Live Lock-Free or Deadlock (Practical Lock-free Programming)"
 - ▶ Part 1: <https://www.youtube.com/watch?v=1VBvHbJsg5Y>
 - ▶ Part 2: <https://www.youtube.com/watch?v=1obZeHnAwz4>
- ▶ A gallery of cache effects:
<http://igoro.com/archive/gallery-of-processor-cache-effects>

Kernel

- ▶ Optimizing preemption:
<https://lwn.net/Articles/563185/>
- ▶ Real-time Linux wiki:
https://rt.wiki.kernel.org/index.php/Main_Page
- ▶ The definitive guide to system calls:
<http://blog.packagecloud.io/eng/2016/04/05/the-definitive-guide-to-linux-system-calls/>
- ▶ Linux Cross Reference (LXR). THE way to read the kernel's code:
<http://lxr.free-electrons.com/>
- ▶ Kernel doc page: <https://www.kernel.org/doc/>