Introduction to multi-threading and vectorization

Matti Kortelainen

LArSoft Workshop 2019

25 June 2019

Outline

Broad introductory overview:

- Why multithread?
- What is a thread?
- Some threading models
  - std::thread
  - OpenMP (fork-join)
  - Intel Threading Building Blocks (TBB) (tasks)
- Race condition, critical region, mutual exclusion, deadlock
- Vectorization (SIMD)

Motivations for multithreading

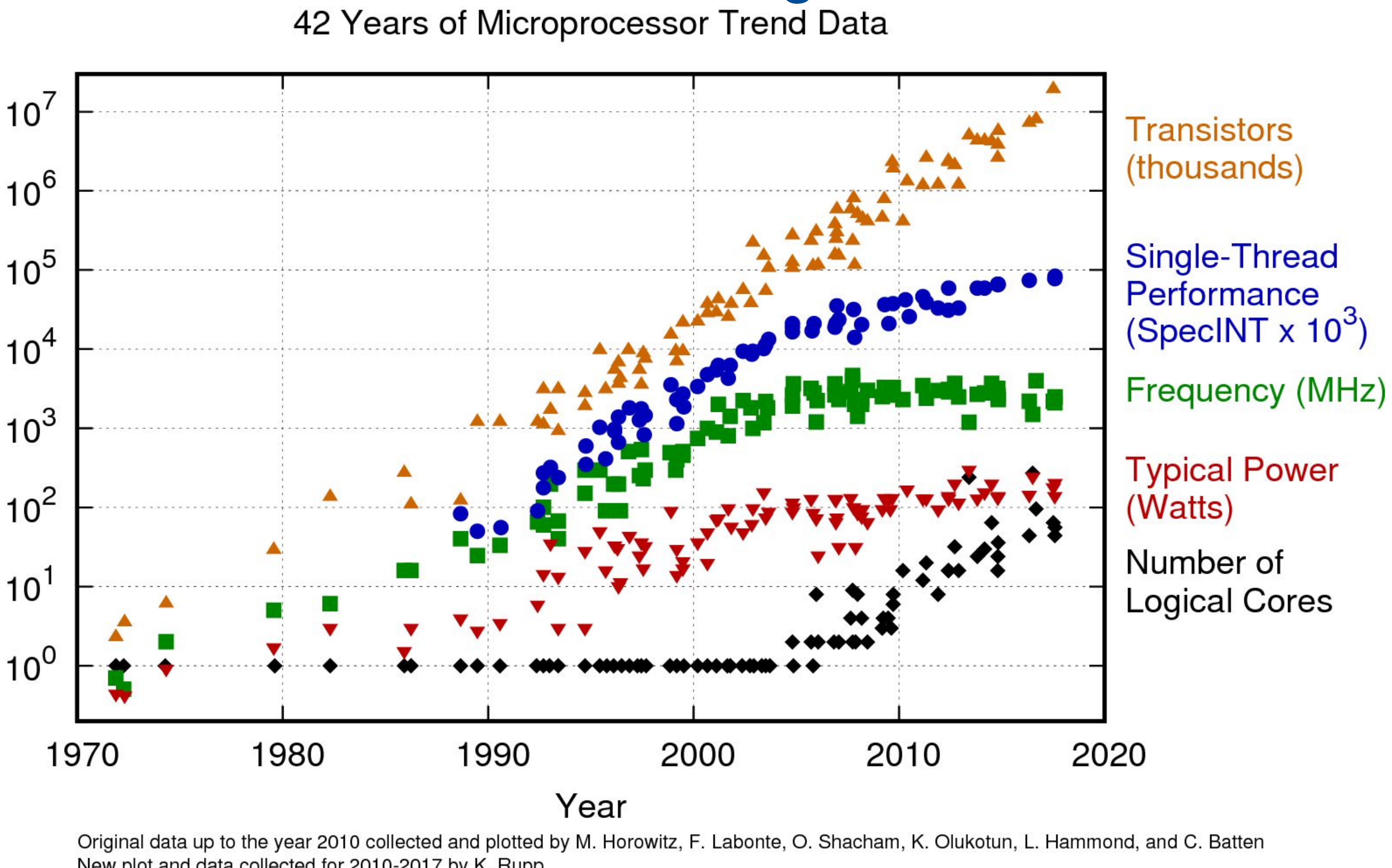


Image courtesy of [K. Rupp]

Motivations for multithreading

- One process on a node: speedups from parallelizing parts of the programs
  - Any problem can get speedup if the threads can cooperate on
    - same core (sharing L1 cache)
    - L2 cache (may be shared among small number of cores)
- Fully loaded node: save memory and other resources
  - Threads can share objects -> N threads can use significantly less memory than N processes
- If smallest chunk of data is so big that only one fits in memory at a time, is there any other option?

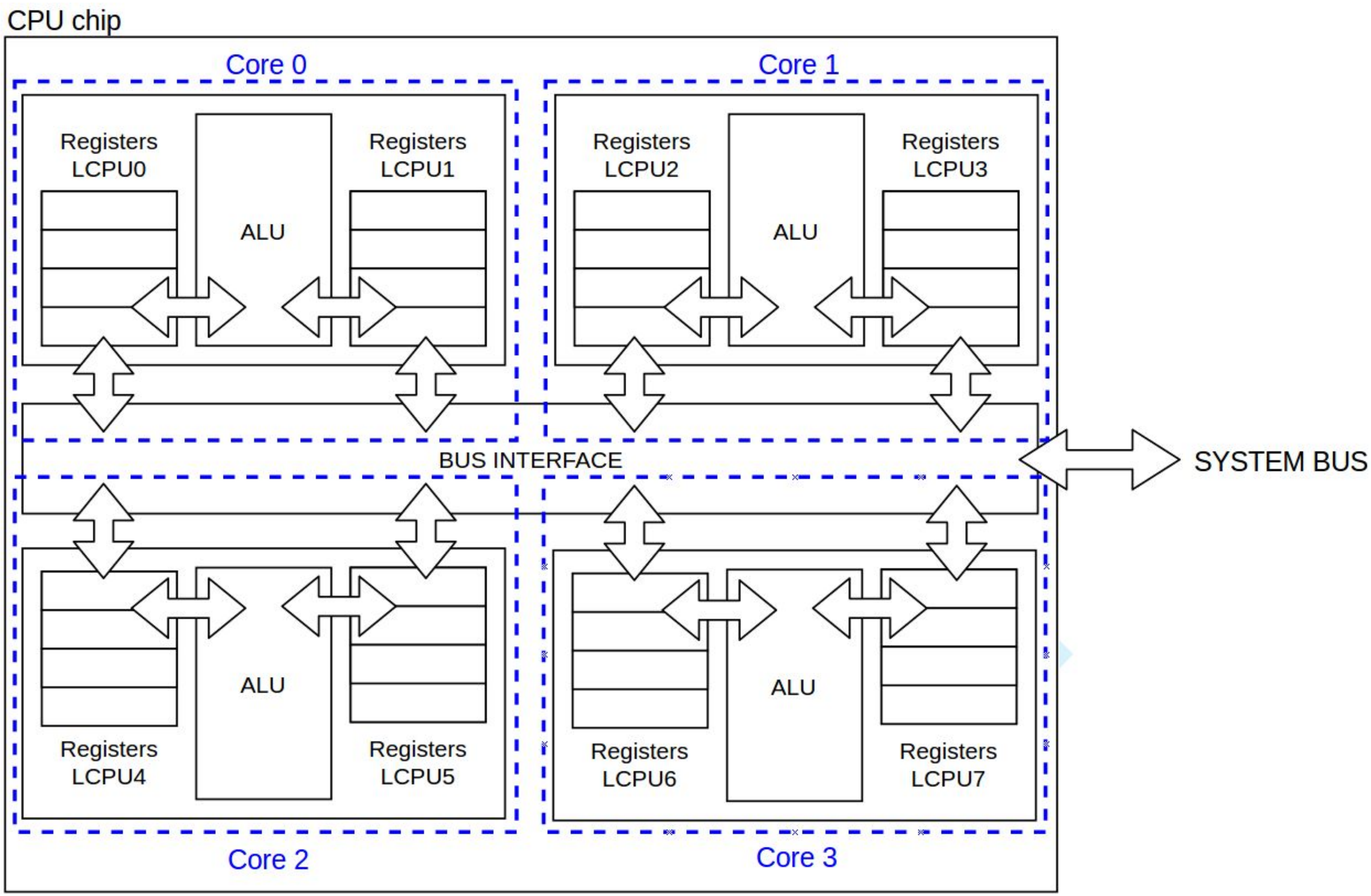
What is a (software) thread? (in POSIX/Linux)

- “Smallest sequence of programmed instructions that can be managed independently by a scheduler” [Wikipedia]
- A thread has its own
  - Program counter
  - Registers
  - Stack
  - Thread-local memory (better to avoid in general)
- Threads of a process share everything else, e.g.
  - Program code, constants
  - Heap memory
  - Network connections
  - File handles

What is a hardware thread?

- Processor core has
  - Registers to hold the inputs+outputs of computations
  - Computation units
- Core with multiple HW threads
  - Each HW thread has its own registers
  - The HW threads of a core share the computation units

Machine model



Quad-core hyperthreading CPU

Image courtesy of [Daniel López Azaña]

What is a hardware thread?

- Processor core has
  - Registers to hold the inputs+outputs of computations
  - Computation units
- Core with multiple HW threads
  - Each HW thread has its own registers
  - The HW threads of a core share the computation units
- Helps for workloads waiting a lot in memory accesses
- Examples
  - Intel higher-end desktop CPUs and Xeons have 2 HW threads
    - Hyperthreading
  - Intel Xeon Phi has 4 HW threads / core
  - IBM POWER8 has 8 HW threads / core
    - POWER9 has also 4-thread variant

Parallelization models

- Data parallelism: distribute data across “nodes”, which then operate on the data in parallel
- Task parallelism: distribute tasks across “nodes”, which then run the tasks in parallel

Data parallelism	Task parallelism
Same operations are performed on different subsets of same data.	Different operations are performed on the same or different data.
Synchronous computation	Asynchronous computation
Speedup is more as there is only one execution thread operating on all sets of data.	Speedup is less as each processor will execute a different thread or process on the same or different set of data.
Amount of parallelization is proportional to the input data size.	Amount of parallelization is proportional to the number of independent tasks to be performed.

Table courtesy of Wikipedia

Threading models

- Under the hoods ~everything is based on POSIX threads and POSIX primitives
  - But higher level abstractions are nicer and safer to deal with
- std::thread
  - Complete freedom
- OpenMP
  - Traditionally fork-join (data parallelism)
  - Supports also tasks
- Intel Threading Building Blocks (TBB)
  - Task-based

- Not an exhaustive list...

std::thread

- Executes a given function with given parameters concurrently wrt the launching thread

```
void f(int n) {
  std::cout << "n " << n << std::endl;
}

int main() {
  std::thread t1{f, 1};
  return 0;
}
```

- What happens?

std::thread

- Executes a given function with given parameters concurrently wrt the launching thread

```
void f(int n) {
  std::cout << "n " << n << std::endl;
}

int main() {
  std::thread t1{f, 1};
  return 0;
}
```

- What happens?
  - Likely prints n 1

std::thread

- Executes a given function with given parameters concurrently wrt the launching thread

```
void f(int n) {
  std::cout << "n " << n << std::endl;
}

int main() {
  std::thread t1{f, 1};
  return 0;
}
```

- What happens?
  - Likely prints n 1
  - Aborts
- Why?

Fermilab

std::thread

- Executes a given function with given parameters concurrently wrt the launching thread

```
void f(int n) {
  std::cout << "n " << n << std::endl;
}

int main() {
  std::thread t1{f, 1};
  return 0;
}
```

- What happens?
  - Likely prints n 1
  - Aborts
- Why? Threads have to be explicitly joined (or detached)

std::thread (fixed)

- Executes a given function with given parameters concurrently wrt the launching thread

```
void f(int n) {
  std::cout << "n " << n << std::endl;
}

int main() {
  std::thread t1{f, 1};
  t1.join();
  return 0;
}
```

- What happens?
  - Prints n 1

std::thread: two threads

```cpp
void f(int n) {
  std::cout << "n " << n << std::endl;
}

int main() {
  std::thread t1{f, 1};
  std::thread t2{f, 2};
  t2.join();
  t1.join();
  return 0;
}
```

- What happens?

std::thread: two threads

```cpp
void f(int n) {
  std::cout << "n " << n << std::endl;
}

int main() {
  std::thread t1{f, 1};
  std::thread t2{f, 2};
  t2.join();
  t1.join();
  return 0;
}
```

- What happens?

```
n 1
n 2
```

std::thread: two threads

```
void f(int n) {
  std::cout << "n " << n << std::endl;
}

int main() {
  std::thread t1{f, 1};
  std::thread t2{f, 2};
  t2.join();
  t1.join();
  return 0;
}
```

- What happens?

```
n 1
n 2
```

```
n 2
n 1
```

std::thread: two threads

```
void f(int n) {
  std::cout << "n " << n << std::endl;
}

int main() {
  std::thread t1{f, 1};
  std::thread t2{f, 2};
  t2.join();
  t1.join();
  return 0;
}
```

- What happens?

```
n 1
n 2
```

```
n 2
n 1
```

```
n 1n 2

```

std::thread: two threads

```cpp
void f(int n) {
  std::cout << "n " << n << std::endl;
}

int main() {
  std::thread t1{f, 1};
  std::thread t2{f, 2};
  t2.join();
  t1.join();
  return 0;
}
```

- What happens?
  - etc

```
n 1
n 2
```

```
n 2
n 1
```

```
n 1n 2
```

- Why?

std::thread: two threads

```
void f(int n) {
  std::cout << "n " << n << std::endl;
}

int main() {
  std::thread t1{f, 1};
  std::thread t2{f, 2};
  t2.join();
  t1.join();
  return 0;
}
```

- What happens?
  - etc

```
n 1
n 2
```

```
n 2
n 1
```

```
n 1n 2
```

- Why? `std::cout` is not thread safe

Fermilab

OpenMP: fork-join

The strength of OpenMP is to easily parallelize series of loops

```
void simple(int n, float *a, float *b) {
  int i;

#pragma omp parallel for
  for(i=0; i<n; ++i) {
    b[i] = std::sin(a[i] * M_PI);
  }
}
```

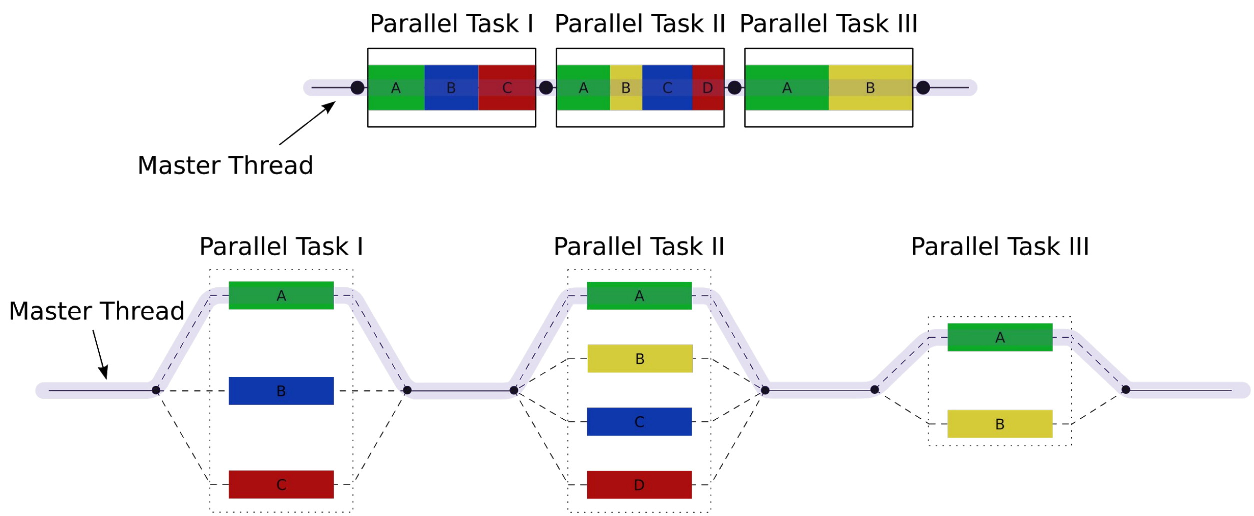


Image courtesy of [Wikipedia]

OpenMP: fork-join (2)

- Works fine if the workload is a chain of loops
- If workload is something else, well …
  - Each join is a synchronization point (barrier)
    - those lead to inefficiencies
- OpenMP supports tasks
  - Less advanced in some respects than TBB
- OpenMP is a specification, implementation depends on the compiler
  - E.g. tasking appears to be implemented very differently between GCC and clang

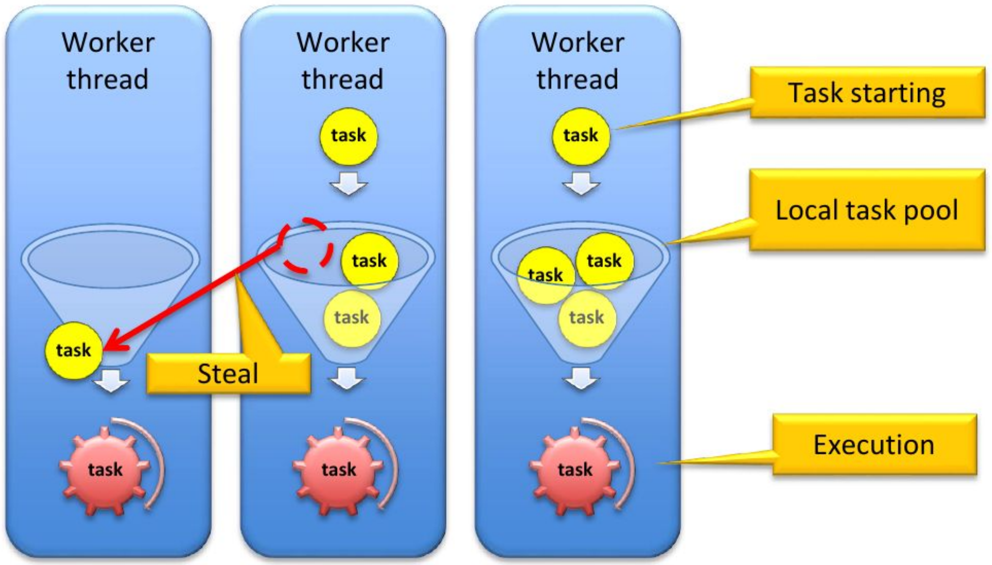
Intel Threading Building Blocks (TBB)

- C++ template library where computations are broken into tasks that can be run in parallel
- Basic unit is a *task* that can have dependencies (1:N)
  - TBB scheduler then executes the task graph
  - New tasks can be added at any time
- Higher-level algorithms implemented in terms of tasks
  - E.g. parallel_for with fork-join model

```
void simple(int n, float *a, float *b) {
  tbb::parallel_for(0, n, [=](int i) {
    b[i] = std::sin(a[i] * M_PI);
  }
}
```

TBB (2)

- Applications often contain multiple levels of parallelism
  - E.g. task-parallelism for scheduling algorithms, fork-join within algorithm
- The work is described at higher level than threads
  - Work is described as tasks
  - Threads are used to execute the tasks
- Automatic load balancing by work stealing



Abstract version of the scheduler

Race condition

```
int sum;

void add(int n) {
  sum += n;
}
```

```
int main() {
  sum = 0;
  std::thread t1{add, 1};
  std::thread t2{add, 2};
  t2.join();
  t1.join();

  std::cout << sum << std::endl;

  return 0;
}
```

- What gets printed?

Race condition

```
int sum;

void add(int n) {
  sum += n;
}
```

```
int main() {
  sum = 0;
  std::thread t1{add, 1};
  std::thread t2{add, 2};
  t2.join();
  t1.join();

  std::cout << sum << std::endl;

  return 0;
}
```

- What gets printed?
  - 3

Race condition

```
int sum;

void add(int n) {
  sum += n;
}
```

```
int main() {
  sum = 0;
  std::thread t1{add, 1};
  std::thread t2{add, 2};
  t2.join();
  t1.join();

  std::cout << sum << std::endl;

  return 0;
}
```

- What gets printed?
  - 3
  - 2
  - 1

Race condition

```
int sum;

void add(int n) {
  sum += n;
}
```

```
int main() {
  sum = 0;
  std::thread t1{add, 1};
  std::thread t2{add, 2};
  t2.join();
  t1.join();

  std::cout << sum << std::endl;

  return 0;
}
```

- What gets printed?
  - 3
  - 2
  - 1
  - Anything, data race is undefined behavior

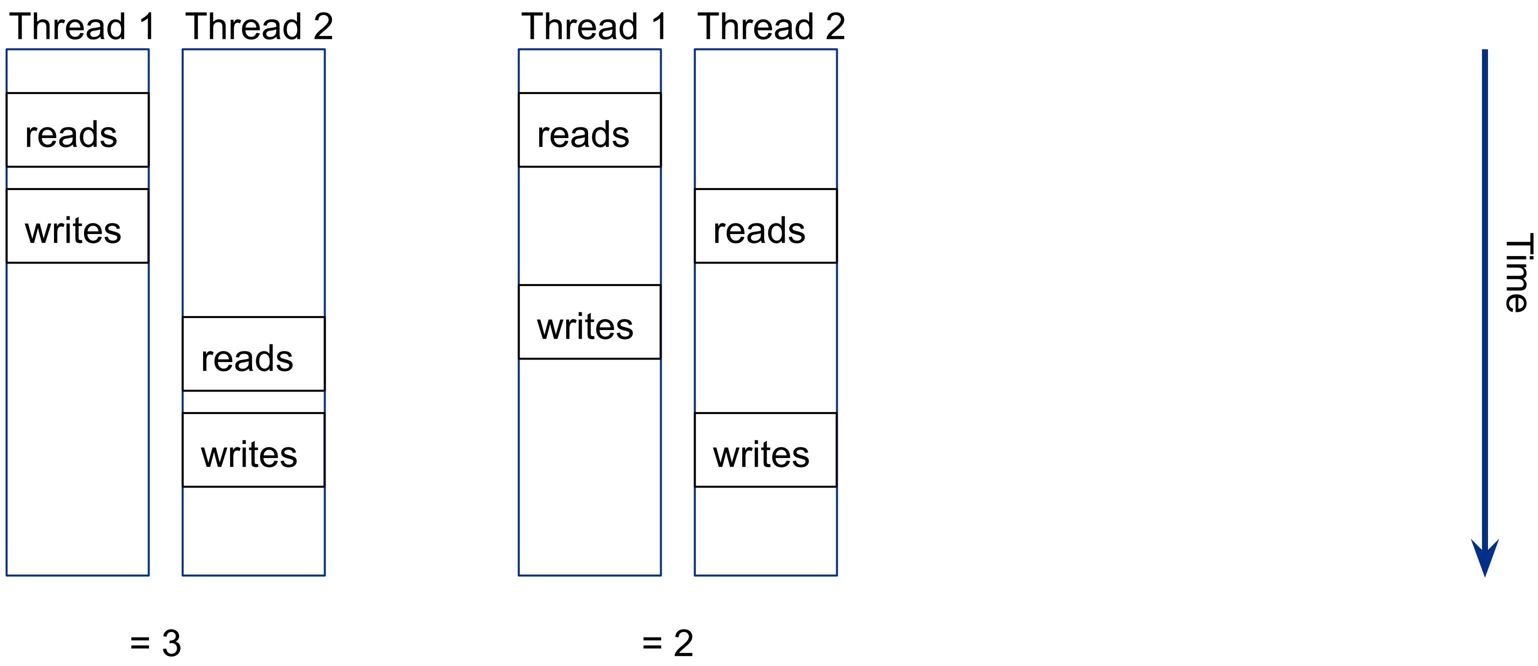
Race condition: explanation

- Two threads “race” to read and write `sum`
- Many variations on what can happen

= 3

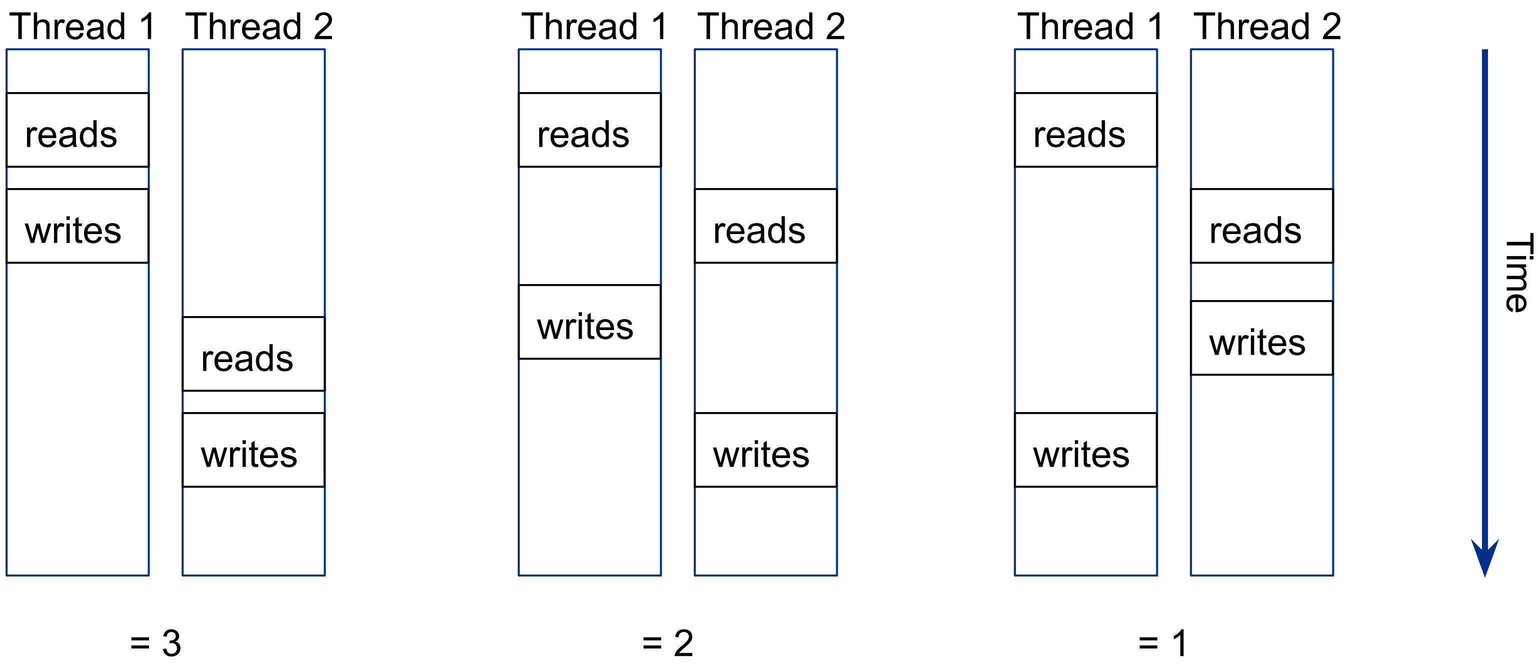
Race condition: explanation

- Two threads “race” to read and write `sum`
- Many variations on what can happen



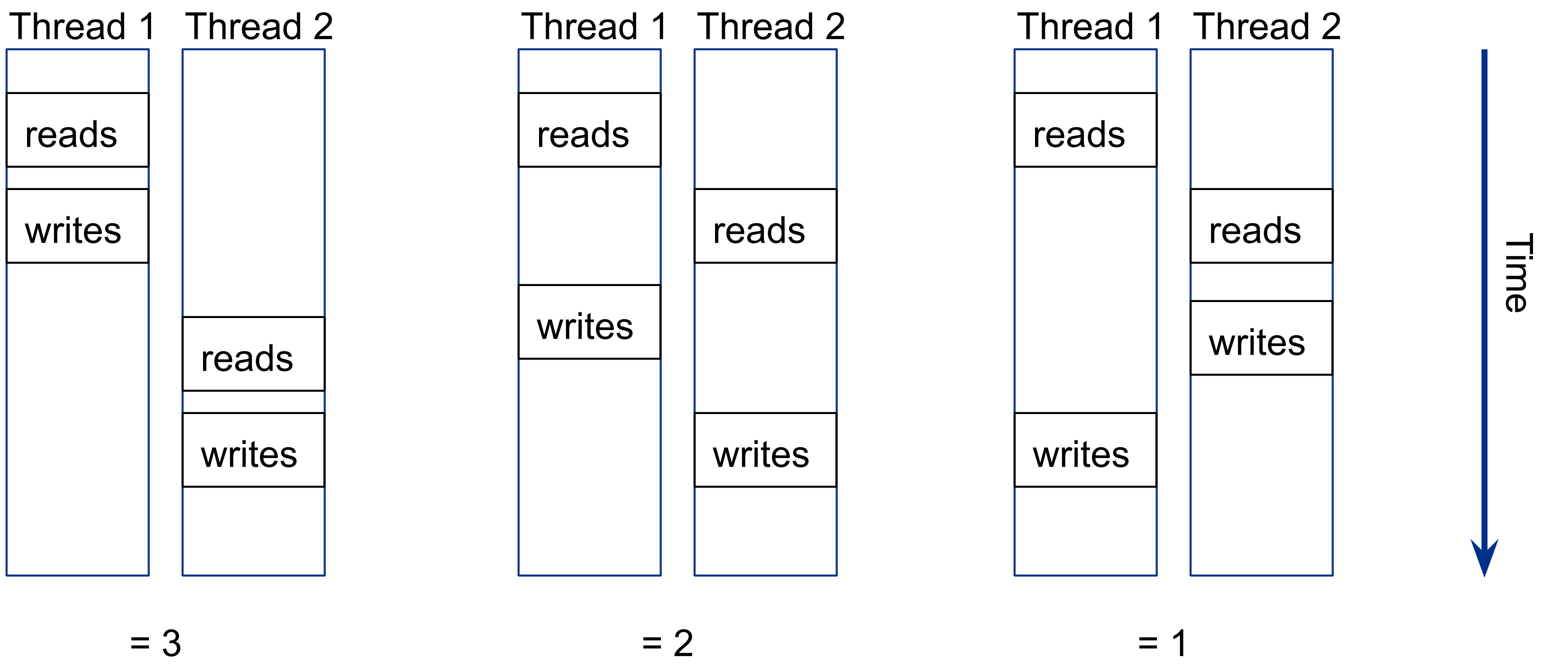
Race condition: explanation

- Two threads "race" to read and write `sum`
- Many variations on what can happen



Race condition: explanation

- Two threads “race” to read and write `sum`
- Many variations on what can happen



- How to solve this problem?

Critical section

```cpp
int sum;

void add(int n) {
  sum += n;
}
```

```cpp
int main() {
  sum = 0;
  std::thread t1{add, 1};
  std::thread t2{add, 2};
  t2.join();
  t1.join();

  std::cout << sum << std::endl;

  return 0;
}
```

- Region of program where shared resource(s) are accessed
  - Needs to be protected

Mutual exclusion

- "is the requirement that one thread of execution never enters its critical section at the same time that another concurrent thread of execution enters its own critical section" [Wikipedia]
- Can be achieved in many ways, a simple way is `std::mutex` and locks
- Some other synchronization mechanisms:
  - Condition variable, semaphore, monitor, barrier
    - Blocking if implemented with mutexes
  - Memory fences with atomics (non-blocking)
    - Needs to be careful

std::mutex and locks

```
int sum;
std::mutex mut;

void add(int n) {
  std::lock_guard<std::mutex> lock{mut};
  sum += n;
}

int main() {
  sum = 0;
  std::thread t1{add, 1};
  std::thread t2{add, 2};
  t2.join();
  t1.join();

  std::cout << sum << std::endl;

  return 0;
}
```

Mutex offers exclusive, non-recursive ownership semantics

lock_guard provides RAII-style mechanism for owning a mutex for the duration of a scoped block

Now the program always prints 3

Deadlock

- "is a state in which each member of a group is waiting for another member, including itself, to take action" [Wikipedia]

```
std::mutex mut1;
std::mutex mut2;

void f1() {
  std::lock_guard<std::mutex> lock1{mut1};
  std::lock_guard<std::mutex> lock2{mut2};
}
void f2() {
  std::lock_guard<std::mutex> lock2{mut2};
  std::lock_guard<std::mutex> lock1{mut1};
}
```

```
int main() {
  std::thread t1{f1};
  std::thread t2{f2};
  t2.join();
  t1.join();
  return 0;
}
```

Very easy to do, rather difficult to find

Atomics

- Primitive types whose operations are atomic
- Additions, subtractions etc, **compare-and-exchange**

```
std::atomic<int> sum;

void add(int n) {
  sum += n;
}

int main() {
  sum = 0;
  std::thread t1{add, 1};
  std::thread t2{add, 2};
  t2.join();
  t1.join();

  std::cout << sum << std::endl;

  return 0;
}
```

Threading guarantees

- Thread friendly
  - E.g. independent non-thread-safe objects for each thread
- Thread safe
  - An operation can be called simultaneously from multiple threads
  - C++11 expects operations on `const` objects to be thread safe
    - either bitwise-`const`, or internally synchronized
- Thread efficient
  - A single mutex for all functions is safe, but not efficient
  - Most performant is if each thread operates on different regions of memory
    - Threads modify the same cache line -> “false sharing”
      - Huge performance hit

Vectorization (SIMD): basic idea

- SIMD = Single Instruction Multiple Data [Wikipedia]
- Same operation on multiple data points simultaneously
- Intel SIMD instruction sets
  - SSE: 128 bits = 4 floats
  - AVX(-2): 256 bits = 8 floats, Fused Multiply-Add (FMA)
  - AVX-512: 512 bits = 16 floats

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

+

21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36

=

22	24	26	28	30	32	34	36	38	40	42	44	46	48	50	52

Vectorization (SIMD): hardware support

- SIMD = Single Instruction Multiple Data [Wikipedia]
- Intel SIMD instruction sets
  - SSE: 128 bits = 4 floats
    - SSE2 is the minimum of x86-64 (first Pentium 4, 2000)
    - SSE3 introduced in Prescott Pentium 4, 2004
    - SSE4 introduced in Core, 2006
  - AVX(-2): 256 bits = 8 floats
    - AVX: Sandy Bridge, 2011
    - AVX-2 added FMA Haswell, 2013
  - AVX-512: 512 bits = 16 floats
    - Xeon Phi KNL, 2013
    - Skylake (Xeon), 2015

Autovectorization

- Let the compiler to do all the work

```
void add(const float *a, const float *b, float *c, int size) {
  for(int i=0; i<size; ++i)
    c[i] = a[i]+b[i];
}
```

- How to know if the compiler actually vectorized?
  - Diagnostic messages
  - Check assembly
    - `movss`/`addss`, `xmm` imply SSE

```
add(float const*, float const*, float*, int):
test ecx, ecx
jle .L1
lea r8d, [rcx-1]
xor eax, eax
.L3:
movss xmm0, DWORD PTR [rdi+rax*4]
addss xmm0, DWORD PTR [rsi+rax*4]
mov rcx, rax
movss DWORD PTR [rdx+rax*4], xmm0
add rax, 1
cmp rcx, r8
jne .L3
.L1:
ret
```

- Different compilers and versions may generate different code
- Small changes to code may lead to non-vectorized code

Pragmas

- Next simplest option
  - E.g. tell the compiler that there are no loop-carried dependencies
- Compiler-specific pragmas (#ivdep)

```
void ignore_vec_dep(int *a, int k, int c, int m) {
#pragma GCC ivdep
  for (int i = 0; i < m; i++)
    a[i] = a[i + k] * c;
}
```

- OpenMP
  - Offers lots of knobs for tuning

```
void ignore_vec_dep(int *a, int k, int c, int m) {
#pragma omp simd
  for (int i = 0; i < m; i++)
    a[i] = a[i + k] * c;
}
```

Libraries

- Program directly with “vector types”
  - E.g. Vc (base for std::experimental::simd)

```
using Vc::float_v
using Vec3D = std::array<float_v, 3>;
float_v scalar_product(Vec3D a, Vec3D b) {
  return a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
}
```

- Scales to 1/4/8/16/… scalar products calculated in parallel, depending on the target hardware

Fermilab

Intrinsics

- Lowest level just a bit above assembly
- Full control, full responsibility

```
float CalcDotProductSse(__m128 x, __m128 y) {
    __m128 mulRes, shufReg, sumsReg;
    mulRes = _mm_mul_ps(x, y);

    shufReg = _mm_movehdup_ps(mulRes);
    sumsReg = _mm_add_ps(mulRes, shufReg);
    shufReg = _mm_movehl_ps(shufReg, sumsReg);
    sumsReg = _mm_add_ss(sumsReg, shufReg);
    return  _mm_cvtss_f32(sumsReg);
}
```

From Stack Overflow

Practical experience

- Vector operations operate on vector registers, want contiguous data
  - Usually Structure-of-Arrays is more performant than Array-of-Structures

```
struct Vec {
  float x, y, x;
};
std::vector<Vec> vecAOS;
```

```
struct VecSOA {
  std::vector<float> x, y, z;
};
VecSOA vecSOA;
```

- Array programming! (numpy, Matlab)
- Most efficient if allocated memory is aligned by 64 bytes
- AVX comes with CPU frequency throttling
- GeantV: only 15-30% improvement from vectorization
- MkFit achieves 2x improvement from vectorization in CMS tracking pattern recognition

Conclusions

- Multi-threading is here to stay
  - A lot of potential pitfalls when going to details
    - There are many high-level abstractions that help
- Most of the time our data processing frameworks abstract away most of the details
  - Enough to write thread friendly/safe/efficient code
- Also some simple guidelines
  - Avoid mutable shared state as much as you can
  - Use `const` properly everywhere you can
- Vectorization works well for math-heavy problems with large arrays/matrices/tensors of data
  - Not so well for arbitrary data and algorithms
  - Keep in mind the CPU frequency scaling