HIGH PERFORMANCE COMPUTING FOR SCIENCE AND ENGINEERING

LECTURE 4

Fabian Wermelinger
Harvard University

CS205

Thursday, February 2nd 2023

LAST TIME

- Why do we need caches?
- Principle of locality
- Strided memory accesses
- High-level approach of how caches work
- Cache lines/blocks
- The different types of cache misses

TODAY

Main topic: ***Introduction to shared memory***

Details:

- *Quiz 1*
- Extension to multicore architectures
- Kinds of parallelism and how the hardware exploits them
- Differences between processes and threads
- Careful sharing: race conditions
- Creation of threads: fork/join model
- Shared memory, distributed memory? The big picture

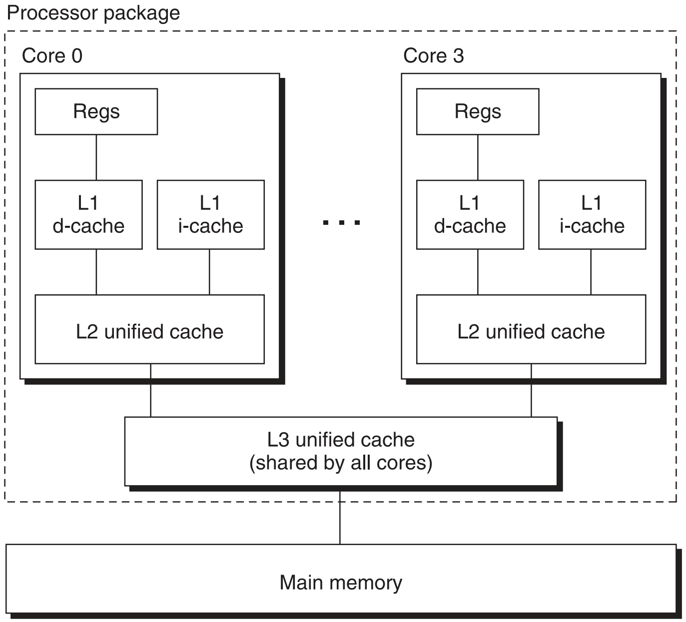
WHAT IS SHARED MEMORY PARALLELISM

- We have seen that Moore's Law still holds in the *multicore* era → doubling the number of *core* on a CPU every two years while cores operates at power wall *(die area increases and 2.5D or 3D semiconductor designs)*
- Up to this point we have not talked about *how* cores cooperate with each other when working on a task
- In a nutshell, when cores cooperate they need to share and/or exchange information (i.e. *data*). Exchange of information is done by *sharing a memory address space* where multiple cores have *read* and *write* privileges.
- This may all sound very easy but is actually quite complicated. ***Assume you have two tasks each running on a core with shared memory. Think about the following situations:***
  - Is it safe if both tasks read from the same memory location?
  - Is it safe if one task reads from and the other writes to the same memory location?
  - Is it safe if both tasks write to the same memory location?

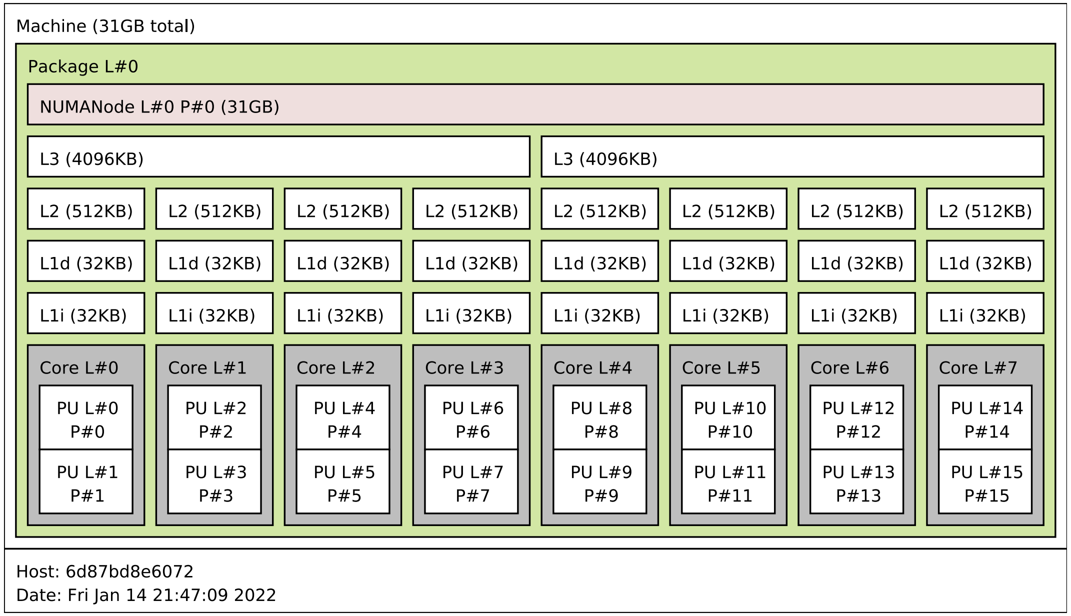
→ *Does any of the three items above expose more parallelism than the other?*

MULTICORE ARCHITECTURE

- A more general sketch of a recent multicore architecture (simply extended by multiple cores and more caches):

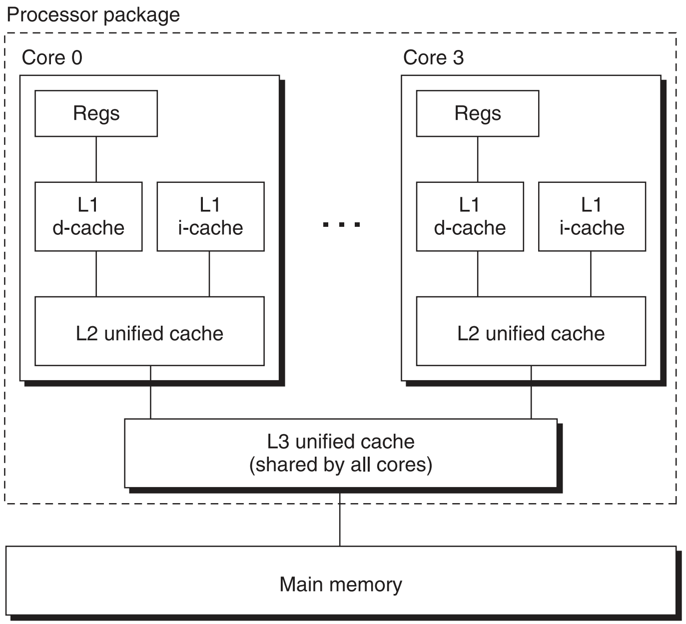


Abstraction of a multicore architecture you can find in textbooks.



How reality looks like: AMD CPU on my laptop. Different vendors produce different architectures. *Conceptually* they are all the same.

PARALLELISM IN HARDWARE



- Multiple physical cores (obviously)
- Multiple physical memory levels (on some architectures caches may be shared among cores, on others they may be exclusive)
- To have something "exclusive" is always in favor for parallelism (being able to work on something *independently* motivates parallelism)
- *CPU architecture level:* pipelining, issuing multiple instructions and vector registers (third extension to von Neumann architecture, does not directly address the bottleneck but makes processor faster by exploiting low-level parallelism)

PARALLELISM IN APPLICATIONS

```c
int a[1024];

int main(void)
{
    // task 1: set first half of the array to 1
    for (int i = 0; i < 512; ++i) {
        a[i] = 1;
    }

    // task 2: set second half of the array to 2
    for (int i = 512; i < 1024; ++i) {
        a[i] = 2;
    }
    return 0;
}
```

Forms of parallelism on the software level:

Task-level parallelism (TLP)

This form of parallelism identifies tasks that can be executed in parallel. Shared memory programming belongs mainly here.

Data-level parallelism (DLP)

This form of parallelism arises because there are many data items that can be operated on at the same time. Vectorization and GPUs belong here. We will discuss this later in the lecture.

How are these kinds of parallelism exploited in hardware:

Instruction-level parallelism (ILP)

Exploits *data-level parallelism* with the help of a compiler using ideas like *pipelining* at the low-level or speculative execution at intermediate levels

Vector architectures, GPUs and vector instructions

Exploit *data-level parallelism* by applying a single instruction to a collection of data

Thread-level parallelism (also often abbreviated with TLP)

Exploits either *data-level parallelism* or *task-level parallelism* in a tightly coupled hardware model that allows for interaction between parallel *threads* (shared memory programming)

PROCESSES AND THREADS

Processes and threads are not the same:

Processes:

- Is an execution sequence within the operating system (a *running* program with a process) ID
- Owns a *virtual address space*
- Has an *execution state* (program counter (PC), register values)
- Is managed by the OS

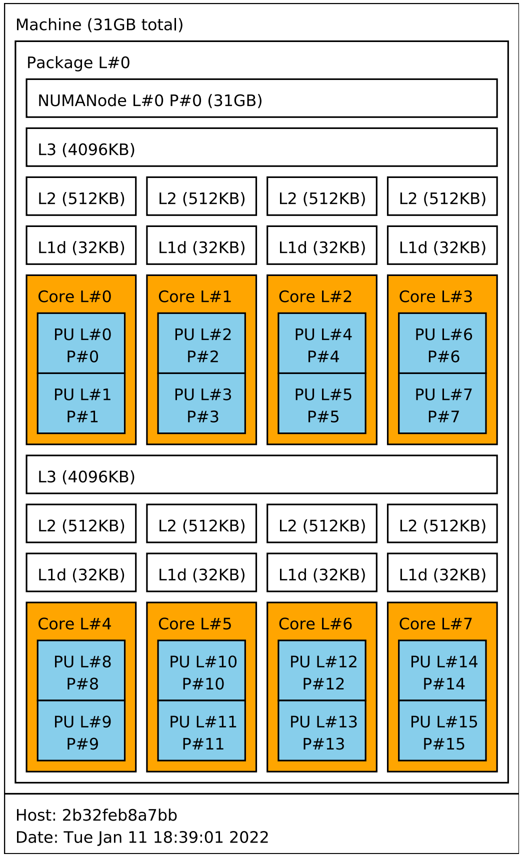
Software threads:

- Can execute a *sequence of instructions* within a running process
- All threads associated to the same process *share* the application data in main memory
- Threads are *cheaper to create* than processes
- The programmer can *control* the creation of threads (*logical* entities in software)

PROCESSES AND THREADS

Hardware threads: Intel Hyper-Threading technology (HT)
(AMD has a similar technology)

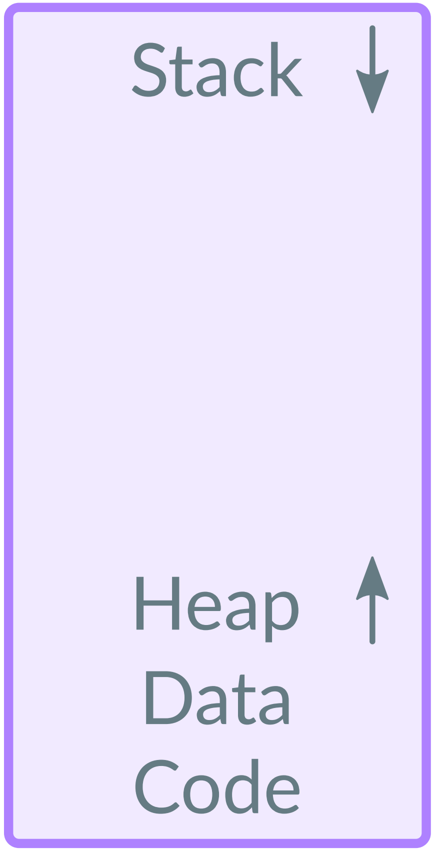
- Hyper-threads are *hardware threads* to help *amortize* idle time (hide latency)
- They split one physical core into two logical cores (a.k.a. hardware threads). *Only one can run at a time*.
- Software threads (previous slide) are scheduled for execution on such logical cores (can be disabled in BIOS)
- *Context switches are cheap on hardware threads*: if a logical threads waits for data, another is run in the meantime for which data is ready (hide latency)



8-core AMD Ryzen 7 PRO 4750U with 16 hardware threads

PROCESSES

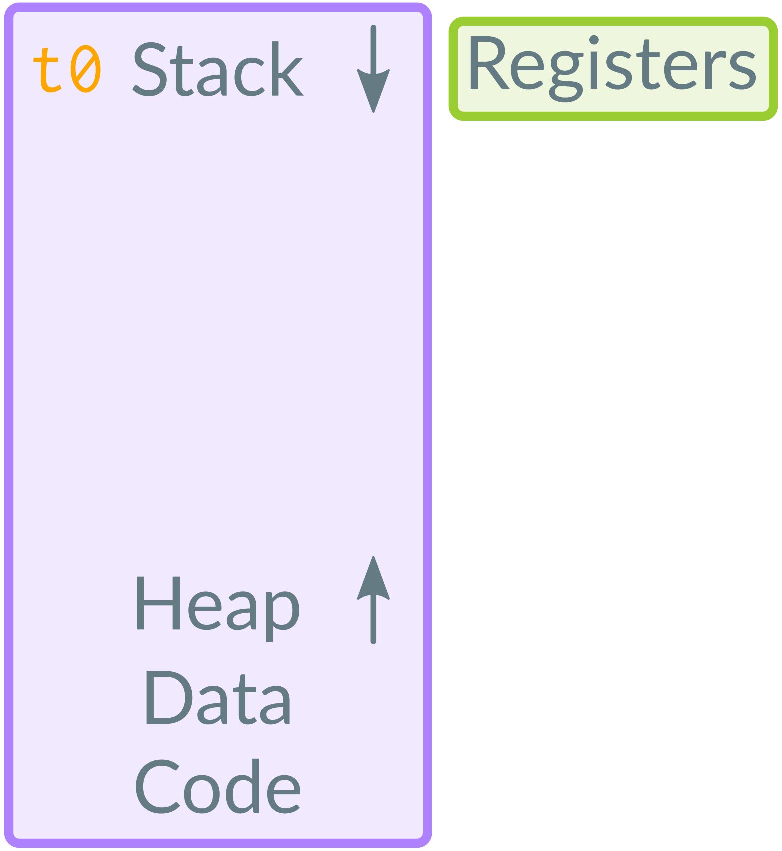
- No process can access one others virtual memory (and therefore physical memory) → a basic safety requirement in multi-user environments such as Linux
- Communication between processes is important when they cooperate on a problem (e.g. MPI)
- Inter-process communication is implemented by the OS with several mechanisms: signals, files, pipes, sockets or shared memory



Simplified representation of the virtual address space of a Linux process (see lecture 2)

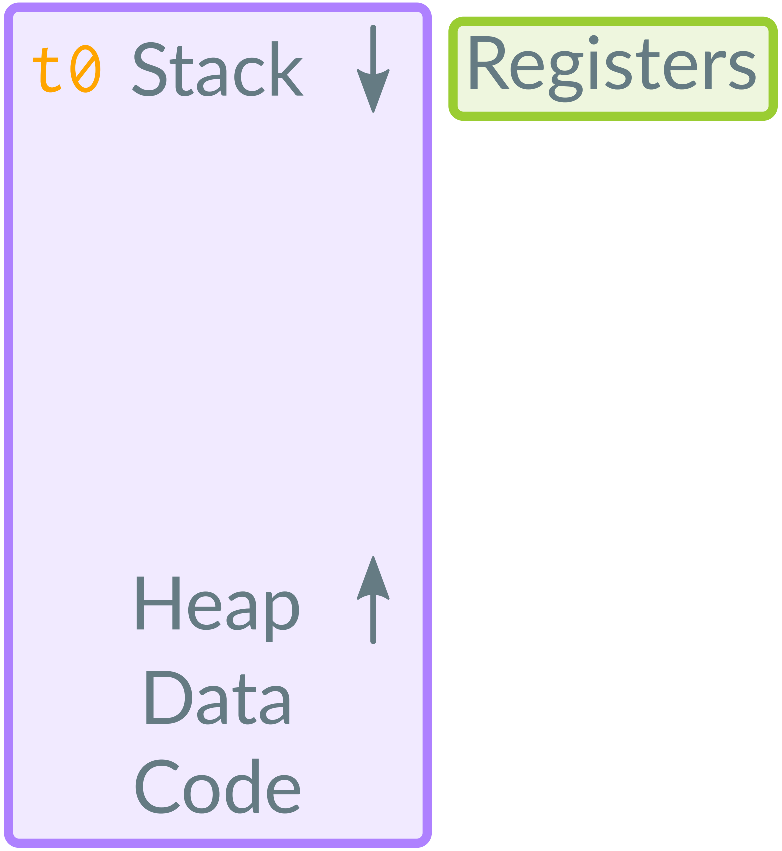
SOFTWARE THREADS

- An *independent stream of instructions* scheduled to run by the OS (runs on a CPU core or possibly hardware thread)
- *A process can create many threads* (more than there are physical cores)
- Each thread:
  - *shares the memory* of the process they belong to with other threads
  - has its own state and some private memory on the stack
  - is free to run on *any* available processor (core or hardware thread)
  - *is cheap to create*

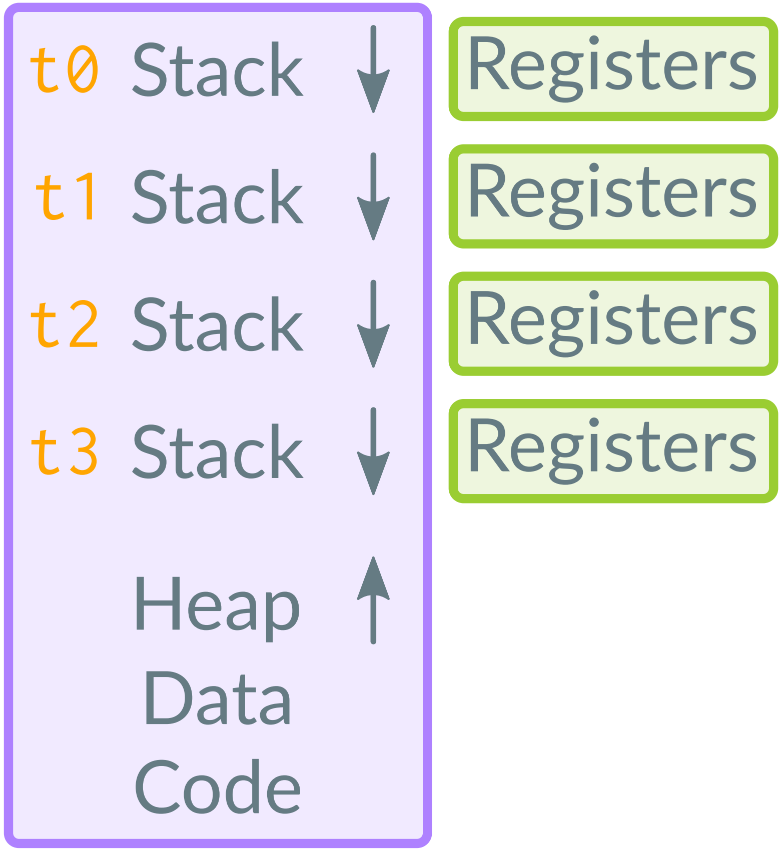


Sequential process

SOFTWARE THREADS



Sequential process



Multithreaded process (concurrent execution). Communication between threads `t0`, `t1`, `t2` and `t3` is through *shared main memory*

A SIMPLE CLASS FOR COUNTING

```cpp
#include <iostream>

class Counter
{
public:
    Counter() : count_(0) {}

    void increment() { ++count_; }
    unsigned int get_count() const
    {
        return count_;
    }

private:
    unsigned int count_;
};

int main(void)
{
    Counter counter;
    for (int i = 0; i < 100; ++i) {
        counter.increment();
    }
    std::cout << counter.get_count() << '\n';
    return 0;
}
```

What the code does:

- The `Counter` class defines `increment` and `get_count` in the public interface
- Any new instance of `Counter` is initialized to `0`
- Incrementing the counter is done with the ++ prefix increment operator
- The `main` function increments the counter 100 times and prints the final count to `stdout`

A SIMPLE CLASS FOR COUNTING

```cpp
#include <iostream>

class Counter
{
public:
    Counter() : count_(0) {}

    void increment() { ++count_; }
    unsigned int get_count() const
    {
        return count_;
    }

private:
    unsigned int count_;
};

int main(void)
{
    Counter counter;
    for (int i = 0; i < 100; ++i) {
        counter.increment();
    }
    std::cout << counter.get_count() << '\n';
    return 0;
}
```

Sequential execution:

- A single thread (called the primary thread) calls the `increment()` method arbitrarily to increment the counter
- The code executes *sequentially* and the instructions in the compiled binary execute in the *correct (program) order*.
- We expect the program to print `100` every time we call it:

```
$ g++ -o counter counter.cpp
$ ./counter
100
$ ./counter
100
$ ./counter
100
$ ./counter
100
```

A SIMPLE CLASS FOR COUNTING

```cpp
1  #include <iostream>
2
3  class Counter
4  {
5  public:
6      Counter() : count_(0) {}
7
8      void increment() { ++count_; }
9      unsigned int get_count() const
10     {
11         return count_;
12     }
13
14 private:
15     unsigned int count_;
16 };
17
18 int main(void)
19 {
20     Counter counter;
21 #pragma omp parallel for
22     for (int i = 0; i < 100; ++i) {
23         counter.increment();
24     }
25     std::cout << counter.get_count() << '\n';
26     return 0;
27 }
```

Parallelize the code:

- We can easily parallelize the code by adding a `#pragma` preprocessor macro for OpenMP *(more next week)*
- → line 21 is useful for homework 1

Multithreaded execution:

- Now multiple threads in a team may call the `increment()` method in the loop body at arbitrary time *in parallel*
- The single `Counter` instance `counter` (line 20) is *shared* by all threads in the team (hence "shared memory")
- What memory operations can threads perform on the shared `counter` variable?

A SIMPLE CLASS FOR COUNTING

```cpp
1  #include <iostream>
2
3  class Counter
4  {
5  public:
6      Counter() : count_(0) {}
7
8      void increment() { ++count_; }
9      unsigned int get_count() const
10     {
11         return count_;
12     }
13
14 private:
15     unsigned int count_;
16 };
17
18 int main(void)
19 {
20     Counter counter;
21 #pragma omp parallel for
22     for (int i = 0; i < 100; ++i) {
23         counter.increment();
24     }
25     std::cout << counter.get_count() << '\n';
26     return 0;
27 }
```

Multithreaded execution:

- Now multiple threads in a team may call the `increment()` method in the loop body at arbitrary time *in parallel*
- The single `Counter` instance `counter` (line 20) is *shared* by all threads in the team (hence "shared memory")
- Compile the code, run with 4 threads:

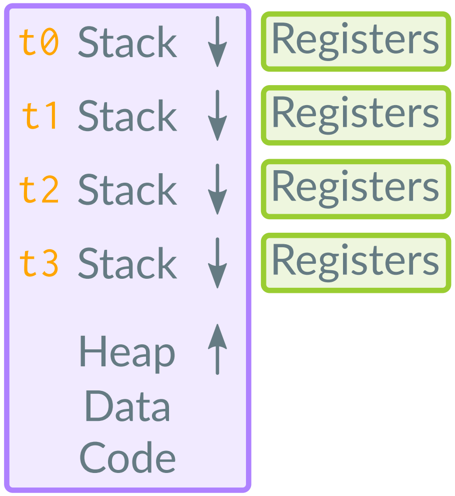
```
$ g++ -fopenmp -o counter counter.cpp
$ OMP_NUM_THREADS=4 ./counter
87
$ OMP_NUM_THREADS=4 ./counter
65
$ OMP_NUM_THREADS=4 ./counter
94
$ OMP_NUM_THREADS=4 ./counter
91
```

- Any ideas why the result is not consistent and different from 100?

RACE CONDITIONS

- ***The biggest difficulty in shared memory programming is to ensure correctness of the code***
- From sequential programming, you are accustomed to assume that your program will take a particular trajectory through the execution state space to yield a correct result (program order)
- In shared memory programming multiple such trajectories are possible, ***all of them must yield the correct result***
- The problem in the previous example is a ***race condition*** on the variable `count_` in shared memory. If multiple threads *read* from this variable everything is fine (`get_count()` is *thread-safe*) but if multiple threads ***write*** to this variable we encounter inconsistent and wrong results. We write to `count_` in the `increment()` method which is not *thread-safe*.

RACE CONDITION IN COUNTER CLASS



Each thread maintains its *own state* of registers and program counter.

- The `increment()` method in the `Counter` class

```
void increment() { ++count_; }
```

increments `count_` by one and then *writes* back the result to memory.

- When we disassemble this code and inspect the machine instructions that are executed for `++count_` we find essentially *3 instructions*:

```
mov edx, DWORD PTR [rax] ; load value at address rax into register edx (count_)
add edx, 1 ; add 1 to value in register edx, store result in edx
mov DWORD PTR [rax], edx ; write the value in edx to memory location rax
```

- The data in registers `rax` and `edx` *is private to each thread*. If two threads execute this code, a ***race condition*** might look like this:

```
mov edx, DWORD PTR [rax] ; thread 0 loads current value of count_
mov edx, DWORD PTR [rax] ; thread 1 loads current value of count_
add edx, 1 ; thread 1 increments count_ and stores result in edx
add edx, 1 ; thread 0 increments count_ and stores result in edx
mov DWORD PTR [rax], edx ; thread 0 writes register edx to memory
mov DWORD PTR [rax], edx ; thread 1 writes register edx to memory (same value!)
```

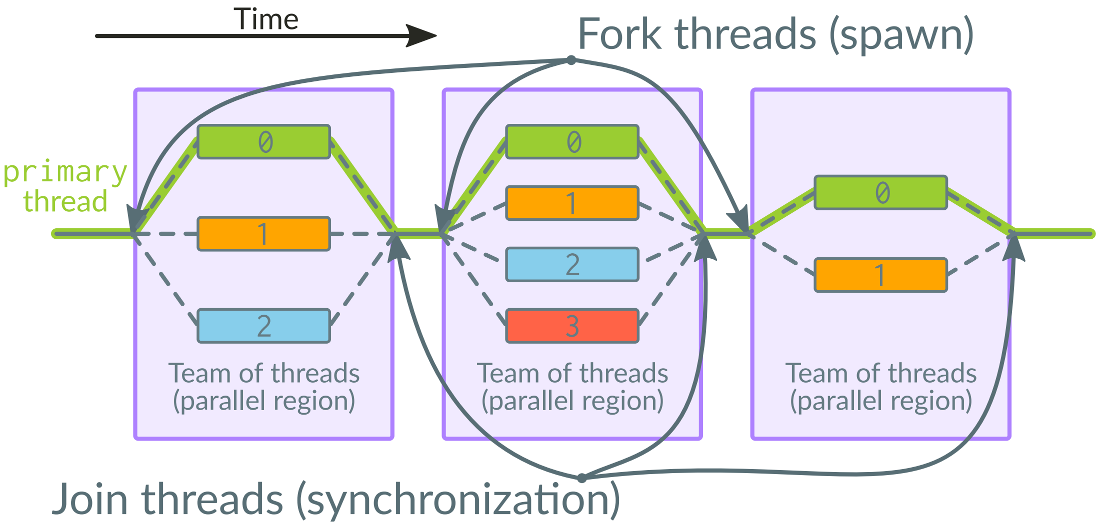
ATOMIC OPERATIONS

- Note that the increment `++count_` is composed of *3 instructions* at the machine level
- What appears to us as a *single line of source code* is effectively "3 lines" of machine code
- This is the root of race conditions in multithreaded execution flow. Be aware that race conditions are ***hard to debug*** and one of the reasons why shared memory programming is more difficult than sequential programming.
- The statement `++count_;` ***is not atomic!***

  An atomic operation is an operation that will always be executed without any other thread being able to read or change state that is read or changed during the operation. It is effectively executed as a single atomic step (smallest building block).
- We will later see that the correct (*thread-safe*) `increment()` method is to add this pragma as well: (using OpenMP)

```
void increment()
{
#pragma omp atomic
    ++count_; // now this line of code is atomic (and thread-safe)
}
```

CREATION OF THREADS – FORK/JOIN MODEL



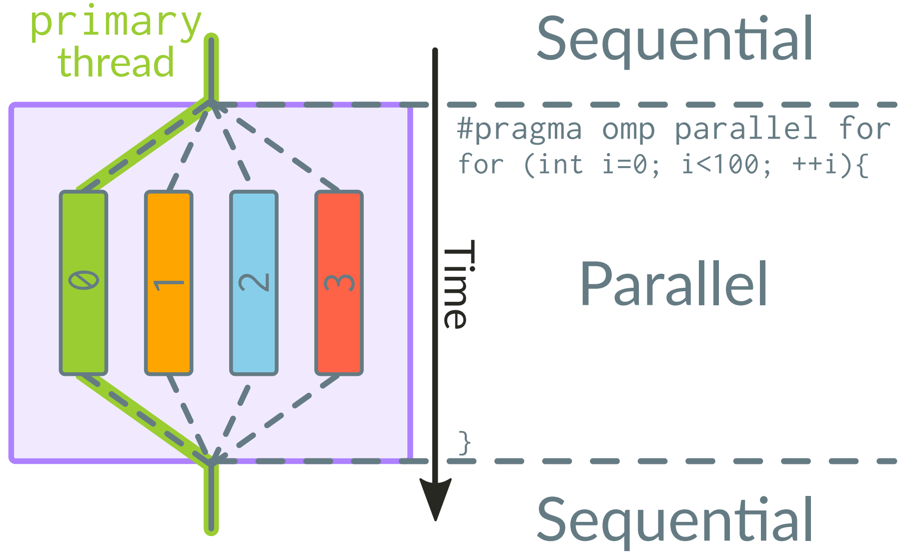
The longer a team of threads can execute in parallel, the higher the *parallel fraction* of your code *(coarse grained granularity, embarrassingly parallel)*

- The `primary` thread is the main thread that enters the `main` function in your program. This thread also exists if you execute sequentially.
- It controls the *creation of child threads* with a fork/join operation
- Child threads within a *team of threads (parallel region)* may create sub-teams of threads (nested parallelism)
- Threads are *joined* together at *synchronization* points (barrier, memory fence)
- After synchronization, *only* the `primary` thread continues execution

CREATION OF THREADS – FORK/JOIN MODEL

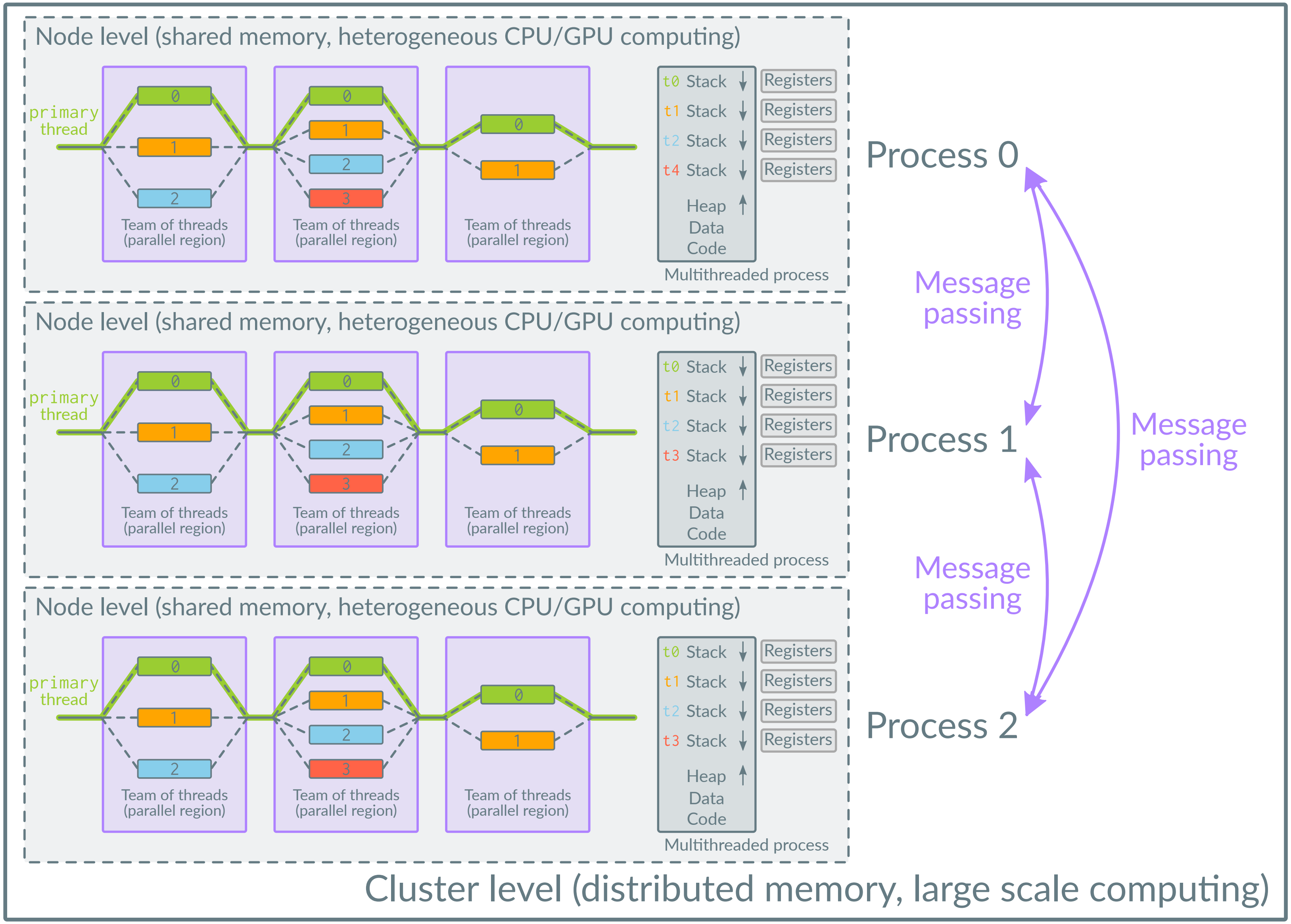
The `Counter` example from before:

```
1  #include <iostream>
2
3  int main(void)
4  {
5      Counter counter;
6  #pragma omp parallel for
7      for (int i = 0; i < 100; ++i) {
8          counter.increment();
9      }
10     std::cout << counter.get_count() << '\n';
11     return 0;
12 }
```

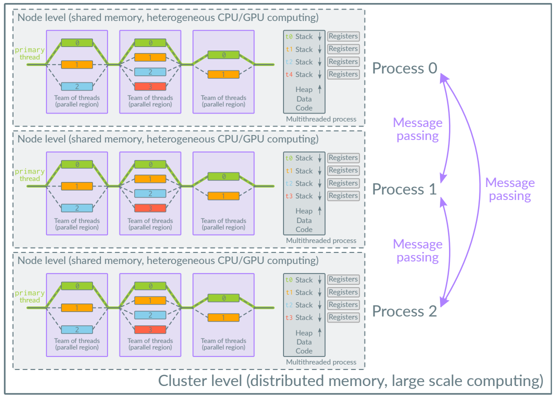


- The highlighted code *spawns* a parallel region using the OpenMP directive `#pragma omp parallel`
- The body of the loop is executed in parallel where the work contained in the 100 iterations is divided among the threads (*trying* to exploit task-level parallelism)
- At the end of the loop (line 9) there is an *implicit synchronization*, the threads are joined together again and the `primary` thread continues sequential execution

WHERE WE ARE GOING (THE BIG PICTURE)



WHERE WE ARE GOING (THE BIG PICTURE)



- The target of a shared memory application is a *compute node*. Examples of compute nodes are your laptop, cell phone or a node on the academic cluster of Harvard.
- Main memory on a node is limited → to *scale* a parallel application we must expand to multiple nodes *(distributed memory)*.
- Communicating with multiple nodes requires *message passing* between *processes*. We learned that the virtual memory of a process cannot be accessed by other processes but we can communicate with signals, pipes or sockets for example. The Message Passing Interface (MPI) takes care of this complexity for us.

RECAP

- Shared memory programming belongs to the MIMD category of parallel computers. We can exploit data-level parallelism and/or task-level parallelism using shared memory
- Software threads (often we just call them "threads") are lightweight logical entities that have their own state (register values, program counter) and can run on any hardware core (scheduled by the OS)
- A race condition happens when two (or more) threads try to *write* to the same memory location at the same time. The program behavior is not deterministic if you have race conditions (why?)
- Threads are forked (or spawned) into teams of threads that operate in *parallel regions*. The longer they are spawned, the larger the available parallelism in your application. Threads are joined at *synchronization* points
- We employ shared memory on the *node-level*

Further reading:

- Section 2.4 in Pacheco, *An Introduction to Parallel Programming*, Morgan Kaufmann, 2011
- Sections 2.3, 2.4, 2.5 and 2.6 in Eijkhout, *Introduction to High Performance Scientific Computing*, free PDF
- Section 4.2 and Chapter 6 in Hager and Wellein, *"Introduction to High Performance Computing for Scientists and Engineers"*, CRC Press, 2011
- Chapter 5 in Hennessy and Patterson, *"Computer Architecture"*, Morgan Kaufmann, 2019