

CS4414 Recitation 14

Final review

12/03/2021

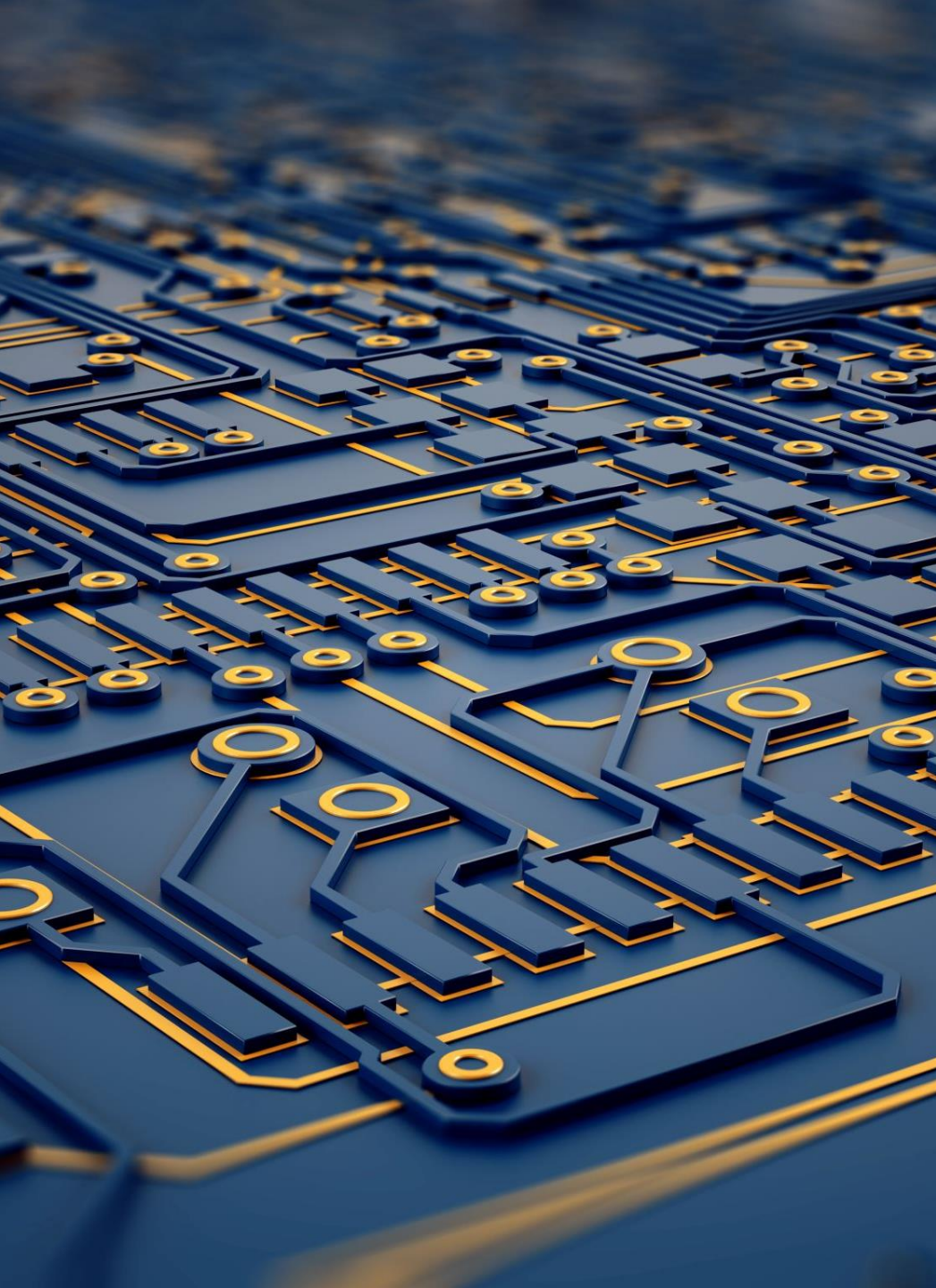
Sagar Jha

What does it mean to write optimal systems code?

- Designing for the hardware
- Picking the right language. Understanding it deeply.
- Profiling code to understand program performance characteristics
- Software multithreading and efficient synchronization

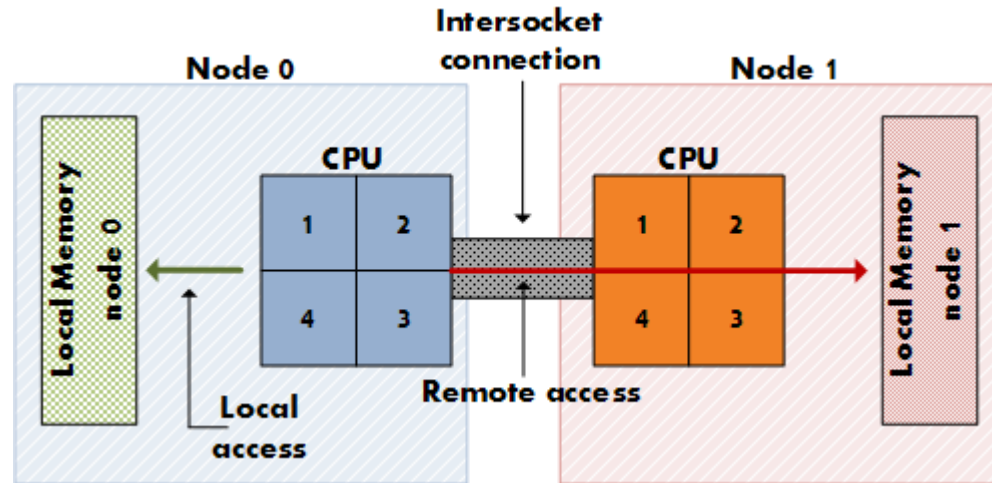
What does it mean to write optimal systems code?

- Designing for the hardware
 - Understanding multicore parallelism
 - Understanding NUMA architecture
 - Understanding any other relevant aspects e.g., disk/memory characteristics, network performance
- Picking the right language. Understanding it deeply.
 - Use containers efficiently – `std::vector<T>`, `std::map<K, V>`, `std::unordered_map<K,V>`
 - Learn C++ idioms: reference vs object vs pointer, copying costs, memory behavior, ownership issues
- Profiling code to understand program performance characteristics
 - Using gprof to understand program time distribution
- Learning software multithreading and efficient synchronization
 - Learning how to parallelize your code and coordinate efficiently among threads



I. Designing for the hardware

Non-uniform memory access (NUMA)



- In multi-core processor architectures, a processor can access local memory faster than non-local memory (shared memory or memory local to other processors)
- In uniform memory architectures, the cost of accessing memory is higher, and increases with the number of processors

How can you find the CPU architecture information on Linux?

- Command `lscpu` gives you the information
- `lscpu` output on `compute16` on Fractus

field	value
Architecture	x86_64
CPU(s), On-line CPU(s) list	32, 0-31
Thread(s) per core	2
Core(s) per socket	8
Socket(s)	2
NUMA node(s)	2
NUMA node0 CPU(s)	0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30
NUMA node1 CPU(s)	1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31

How does NUMA affect performance?

```
int counter = 0; std::function<void()>
repeated_increment = [&counter]() {
for(uint32_t i = 0; i < one_billion; ++i) {
counter++; } };

std::thread t1(repeated_increment);
std::thread t2(repeated_increment);
t1.join(); t2.join();
```

Assignment of threads to cores	Total time taken (avg. over 10 runs)
taskset 0x3	9.04s
taskset 0x5	5.63s

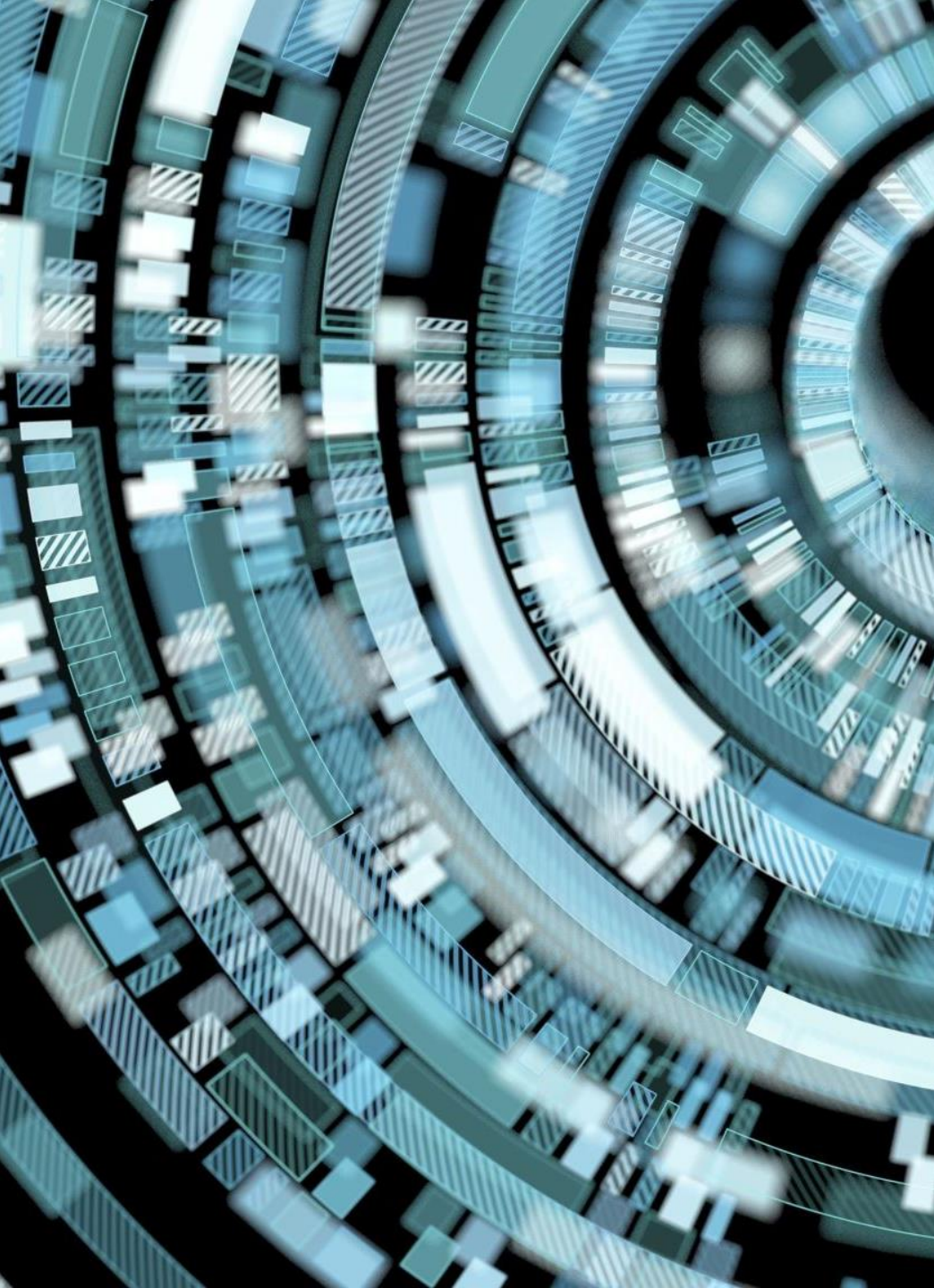
Question 1: Why such a big discrepancy in performance?

How does NUMA affect performance?

- We know accessing remote memory is slower. In what cases, will our program be forced into doing that?

How does NUMA affect performance?

- We know accessing remote memory is slower. In what cases, will our program be forced into doing that?
- If we run out of memory on the same NUMA node, the OS will allocate more memory on a remote node
- Two threads running on different NUMA nodes sharing data
- Modern systems have caches, but the behavior remains the same
 - need to access remote memory on a cache miss
 - overhead of cache coherence with memory on the remote node
- **NUMA-aware design:** Designing applications to take full advantage of the NUMA architecture



II. Picking the right language.
Understanding it deeply.

Why use C++ for systems programming?

Why use C++ for systems programming?

- C++ is designed for systems programming
- Static type checking – memory layout of each object is known beforehand
- Code compiles down to the architecture
- C++ compilers spend significant time during compilation to improve performance at runtime. g++ is the best compiler at optimizing code
- Reduced runtime checking for maximum performance
- C++ gives you many options for each programming feature – pick and choose based on the exact need and performance requirements
- Punishes you every time you make a mistake – develops good programming habits in the long run

Learn effective usage of C++ containers

- When to use a vector, when to use a list, when to use a map...
- **std::vector** is the most important C++ data structure
 - It's heavily optimized for good performance
- Learning the `emplace` commands in order to avoid copying

Using std::vector<T> effectively

```
std::vector<uint32_t> random_elements;  
uint64_t running_sum = 0;  
for(uint32_t i = 0; i < one_million; ++i) {  
    random_elements.emplace(random_elements.begin(),  
                            (running_sum + get_random_number()) % 1000);  
    running_sum += random_elements.front();  
}
```

Question 2: What's wrong with the code above?

std::vector<T>::emplace is expensive

```
for(uint32_t i = 0; i < one_million; ++i) {  
    random_elements.push_back(  
        (running_sum + get_random_number()) % 1000);  
    running_sum += random_elements.back();  
}  
std::reverse(random_elements.begin(), random_elements.end());
```

Two improvements:

- Using push_back to insert elements
- Reversing the vector at the end

Question 3: Can we do even better?

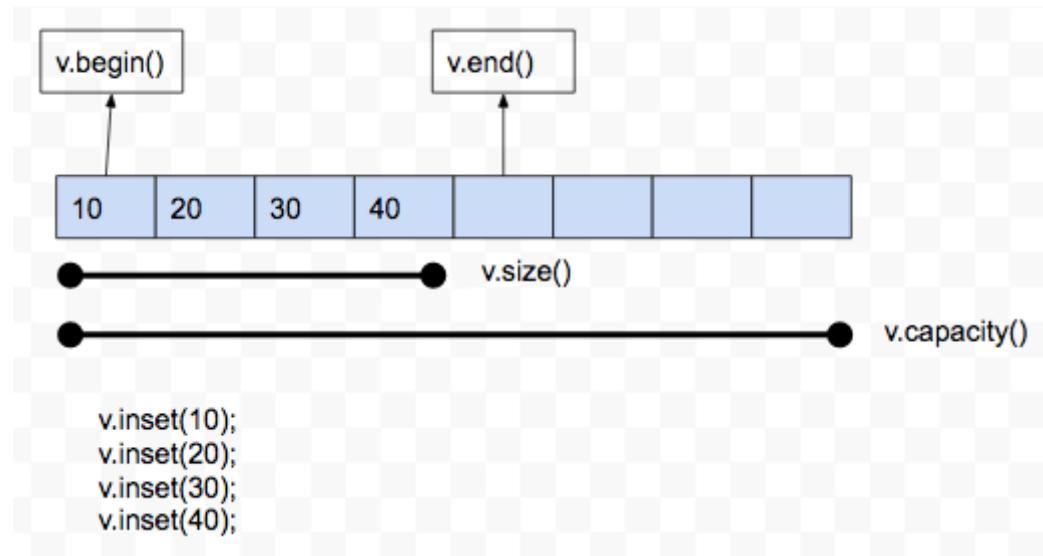
Resize the vector with the known size upfront

```
std::vector<uint32_t> random_elements(one_million);  
for(int32_t i = one_million - 1; i >= 0; --i) {  
    random_elements[i] = (running_sum +  
                          get_random_number()) % 1000;  
    running_sum += random_elements[i];  
}
```


Looking at the numbers

Approach	Runtime
emplace	55 seconds
push_back and reverse	60 milliseconds
resize upfront	30 milliseconds

Theory: How does `std::vector<T>` work?



- `std::vector` is a collection of elements stored **contiguously** in memory
- Contiguous storage provides random access in $O(1)$ time
- `emplace` is linear because elements after the inserted position need to be moved right
- `push_back` is amortized $O(1)$
- when size equals capacity and a new element is inserted at the end, the vector needs to be reallocated and moved to a new memory location

When to use other containers?

- Use `std::list<T>` if you need to insert in the middle given an iterator
- Traversing an `std::list<T>` is costlier because non-contiguous storage of elements leads to worse cache behavior
- `std::map<K, V>` vs. `std::unordered_map<K, V>`
 - `std::map<K, V>` provides deterministic $\log n$ insert and find, while `std::unordered_map<K, V>` provides amortized $O(1)$
 - `std::map<K, V>` provides additional features such as traversing in sorted key order, finding upper/lower bounds, etc.
 - What happens when the key type is `std::string`?

What's wrong with the following code?

```
std::vector<double> recursively_process(  
    std::vector<double>& numbers, uint32_t num_times = hundred_thousand) {  
    if(num_times == 0) {  
        return numbers;  
    }  
    // modify the numbers using some math operations... finally  
    return recursively_process(numbers, num_times - 1);  
}  
  
int main() {  
    std::vector<double> numbers(ten_thousand);  
    // initialize the elements somehow...  
    recursively_process(numbers);  
}
```

Question 4: Can you guess which optimization brought the runtime from 5 seconds down to 2?

Pass arguments by reference when possible

- Not applicable for very small data types (int, double, bool etc.)
- Fun fact: In my undergrad, I helped a friend in ECE with this exact problem
- What's the larger idea with passing arguments by reference?

Larger idea: Avoid copying objects

- Pass arguments by reference
- Share an object across multiple entities using `std::shared_ptr`
- Pass ownership of an object using `std::move`
- C++ incorporates RVO (return value optimization)
- Disable copying explicitly in the class definition. E.g.,

```
myClass(const myClass&) = delete;  
myClass& operator=(const myClass&) = delete;
```

Learn to use lambda functions

Question 5: Partition students based on their score – Below 75 and on or above 75

Given

- `template< class ForwardIt, class UnaryPredicate >`
`ForwardIt partition(ForwardIt first, ForwardIt last, UnaryPredicate p);`
 - Reorders elements between first and last such that elements for which **p** is true are to the left of the elements for which **p** is false
 - Returns iterator to the first element of the second partition
- `std::vector<Student> students;`
- `int Student::get_score() const;`
 - returns a number between 0 and 100

Solution: encode the partition condition into a lambda function

```
std::partition(students.begin(), students.end(),  
              [] (const Student& s) {  
                  return s.get_score() < 75;  
              }) ;
```


Metaprogramming using templates

- Generic programming: How can we write code that works with different types?
- We write a template for the code. The compiler generates the code based on the template. All types are completely defined at compile-time!
- Variadic templates: Using this, a function or class that can take variable number of template arguments

Variadic templates question

Question 6: Define a class `print_all` that takes any number of arguments of a type and calls `print` on each one of those arguments

- For example,

```
print_all (student1, ta1);
```

should work as well as

```
print_all (student1, ta1, student2, ta2);
```

assuming `void Student::print() const` and `TA::print() const` is defined

Solution: Use recursion!

```
void print_all() {  
}
```

```
template <typename T, typename... Ts>  
void print_all(const T& t, const Ts&... ts) {  
    t.print();  
    print_all(ts...);  
}
```

C++'s RAI technique

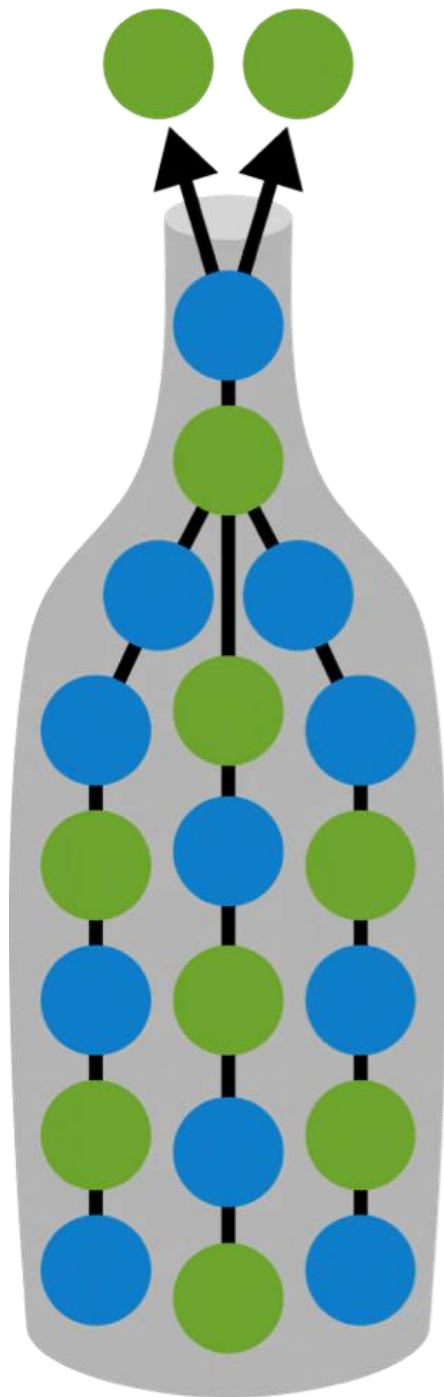
```
std::mutex m;  
std::function<void()> process =  
    [&m] () {  
        m.lock();  
        somefunction();  
        m.unlock();  
    };
```

Question 7: Function process is supposed to be called from different threads in different parts of the code. It calls `std::mutex::lock` and `std::mutex::unlock` explicitly. In testing, it is found that the code deadlocks. What could possibly be happening in `somefunction`?

C++'s RAII technique

- Tie resource allocation and release to the lifetime of an object
- Works because C++ destroys an object right when it goes out of scope
- Use `std::unique_ptr` instead of raw pointers
- Use `std::unique_lock` (or variants) to lock or unlock mutexes

```
std::scoped_lock<std::mutex> lock(m) ;  
somefunction() ;
```



III: Profiling code to understand performance characteristics

Bottleneck analysis: Targeted algorithmic improvements

- Profiling with gprof or by performance tests can guide the optimization process
- We made several informed decisions about what to optimize:
 - We found that calls to `std::stoi` and `std::stod` were expensive. We tried to optimize by caching the results
 - We implemented a vector-based event processing system to get rid of possible inefficiencies of the `priority_queue`-based approach
 - I decided to not worry about pipelining opening files and processing them in my word count program since opening files took an insignificant amount of time.
- Understanding the workload can similarly provide great insight into systems design
 - There are entire systems dedicated to optimizing read throughput since read requests can make up to 95% of application workloads

To optimize, or not to optimize, that is the question

Question 8: A multithreaded program consists of a sequential step S followed by a perfectly parallel step P. Step S takes 5% and P takes 95% of the total runtime when the program is run with a single thread. There is an option to optimize either of the two steps by 25%. Under what conditions (number of parallel threads) would it be more beneficial to optimize S than P?



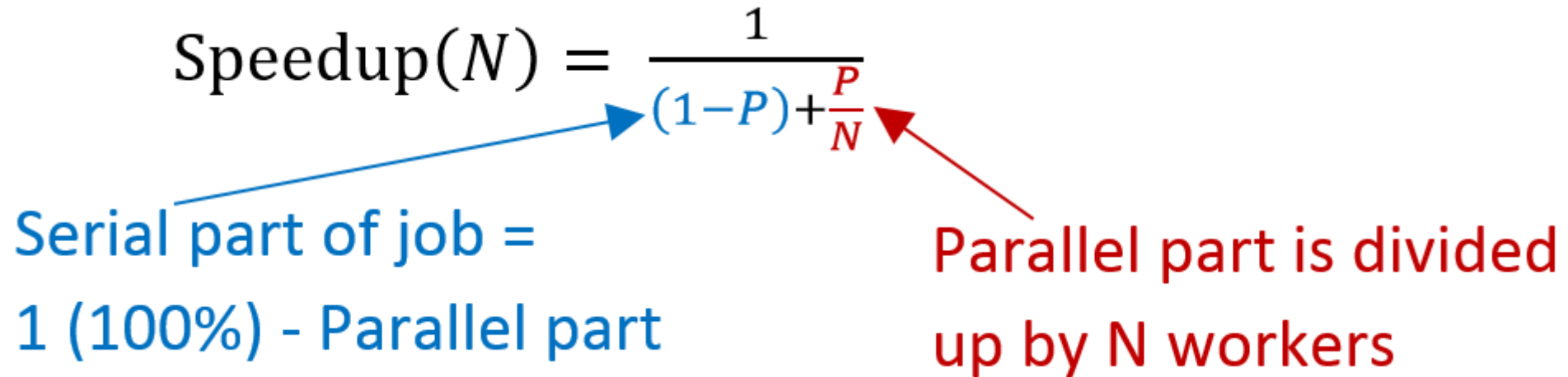
IV. Learning software multithreading and efficient synchronization

Amdahl's law

$$\text{Speedup}(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

Serial part of job =
1 (100%) - Parallel part

Parallel part is divided
up by N workers



- Infinite parallelism will not give infinite speedup
- Performance is limited by the cost of the sequential parts
- We saw it firsthand in word count: While the parallel file processing was dominating runtime for smaller #threads, the sequential merge step became a bottleneck with large #threads (~64)

Main takeaway

- **It helps to make your program more parallel (get rid of the sequential steps) than to optimize the parallel steps**
- That's why efficient thread synchronization is a big deal – whenever a thread holds a lock that prevents other threads from progressing, your program loses performance

Race conditions and deadlocks

- Race condition occurs when two threads access a critical section simultaneously.
- A critical section is part of the code that must be run exclusively at a given time by a single thread. It is the part where variables shared across threads are accessed safely. An `std::mutex` can be used to implement mutual exclusion.
- Deadlock occurs when the system cannot make progress.
- What is the most salient feature of a deadlock?
 - Threads (or processes in a distributed system) are waiting on each other

std::atomic vs std::mutex

Question 9: Which of the following is definitely going to be more efficient in the word-count program:

- 1. We maintain an std::set of unprocessed files. Each thread acquires a mutex and removes an element from the set. That is the next file it processes.**
- 2. We maintain a vector of all files to process. Files are processed from left to right, thus we maintain an atomic integer that stores the position of the first unprocessed file. Each thread atomically reads and increments the integer and process the file corresponding to the position read.**

When to use an `std::atomic`?

- `std::mutex` is general purpose. It provides a lot more features than an `std::atomic`. E.g., `std::lock` can lock multiple mutexes simultaneously atomically. When the mutex is already locked, the thread will be blocked instead of hogging the CPU.
- When they both meet the synchronization requirements, `std::atomic` is going to be much more efficient.
- `std::atomic` only wraps a single variable, which is often a primitive type. Must use mutex if atomicity is needed over multiple state variables.

Purpose of a condition variable

- When a thread needs to wait for a condition that can only be enabled by another thread, we use an `std::condition_variable`. This is combined with acquiring/releasing the mutex to guarantee mutual exclusion.
- Synchronization is on the condition, not on just waking up from the wait. There are spurious wake-ups, there may be multiple threads waiting when only few of them can proceed.
- A condition variable works only with a mutex. Use an `std::condition_variable_any` object to work with a shared mutex

std::promise<T> and std::future<T>

- A mechanism by which a thread can pass the result of a computation on to another thread
- The thread holding the future object simply needs to call the get function
- The thread that produces the result satisfies the promise by calling set_value on the promise object

Example scenario for using promise and future

- Suppose I implement a thread pool class
- The user submits tasks in the form of lambda functions along with arguments that are stored in an object of the class and asynchronously executed
- Whenever a thread from the pool becomes available, it processes the next task that is on the list
- The user is returned a future object upon a successful submission of a task. The thread that completes the task satisfies the promise object with the result of the function call

ThreadPool: Public functions

(ref: <https://github.com/progschj/ThreadPool>)

- `ThreadPool::ThreadPool(size_t threads);`
 - Create a thread pool with *threads* number of threads
- `template<class F, class... Args>`
`auto ThreadPool::enqueue(F&& f, Args&&... args)`
`-> std::future<typename std::result_of<F(Args...)>::type>`
 - Add a new task to the pool
- `ThreadPool::~~ThreadPool()`
 - Non-trivial destructor since we are working with threads

ThreadPool: Data members

(ref: <https://github.com/progschj/ThreadPool>)

- `std::vector< std::thread > workers;`
 - collection of threads in the pool
- `std::queue< std::function<void()> > tasks;`
 - collection of tasks that need to be completed
- `std::mutex queue_mutex;`
`std::condition_variable condition;`
`bool stop;`
 - For synchronization

Solve the C++ puzzle I posted on Ed discussions

Thanks for taking the course!