LINZ INSTITUTE OF TECHNOLOGY

PARALLEL COMPUTING
Parallel Computing with Modern C++

Univ.-Prof. Dr. Alois Zoitl
LIT | Cyber-Physical Systems Lab
Johannes Kepler University Linz

Parallelism is Everywhere

- Servers
- Computers
- Smartphones

Why Parallel Computing?

- High performance (e.g., low execution time, high throughput, low latency)
- Scalability
- Quality of services
- Reduce the energy consumption?
  - Less cost
  - More sustainable

→ **How is this possible?**

Why Parallel Computing?

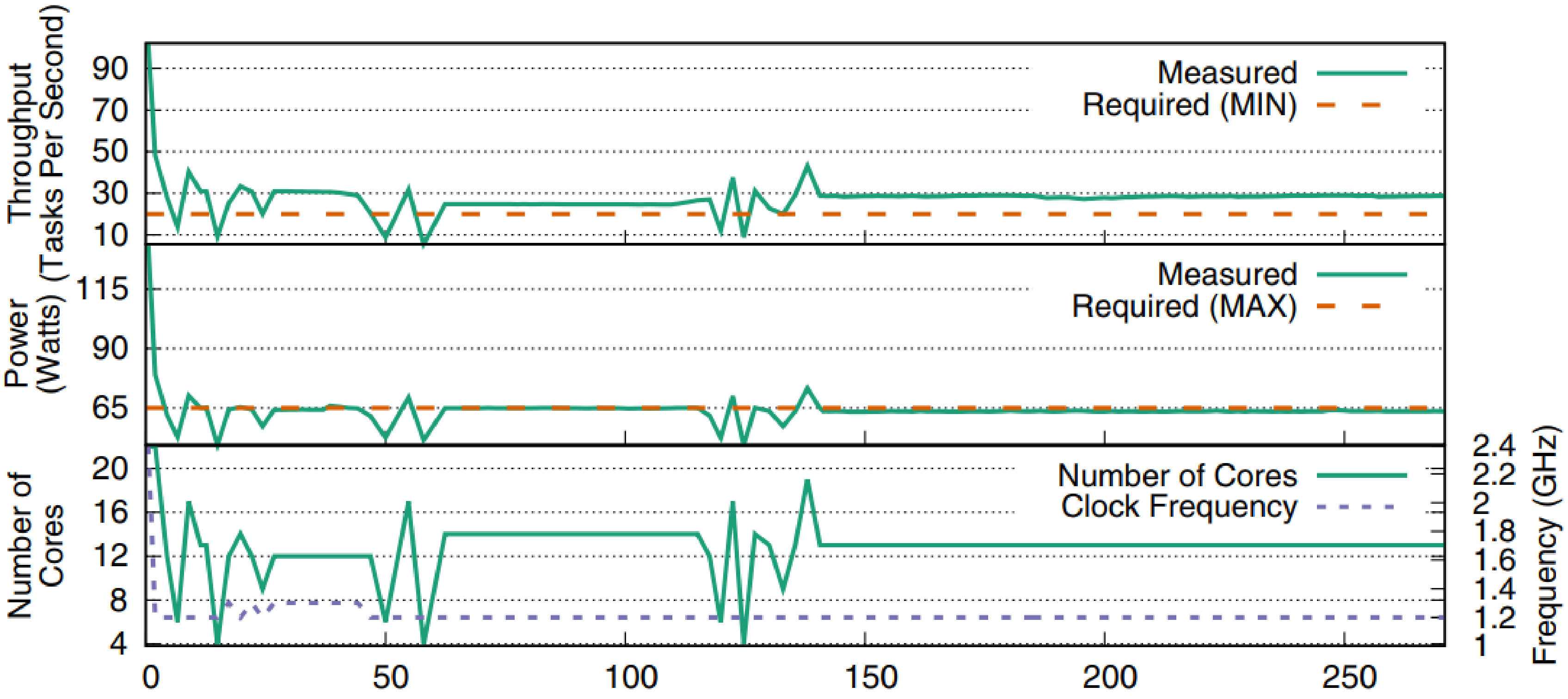
- Power consumption reduction obtained with parallel execution compared to the sequential ones [3]

	Pbzip2	Lane detection	Person recognition
Power consumption reduction (%)	– 9.43%	– 10.37%	– 7.39%

How is it possible?

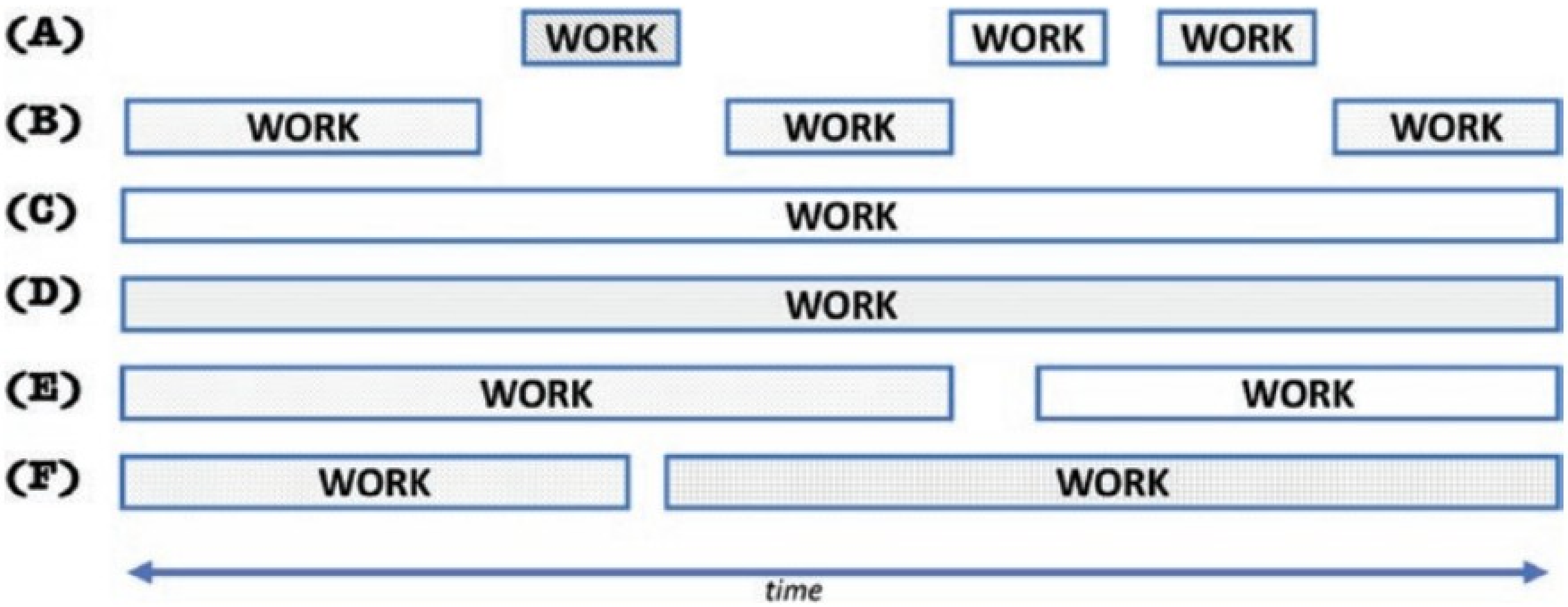
Why Parallel Computing?

- Power consumption reduction obtained with parallel execution compared to the sequential ones



Source [3]

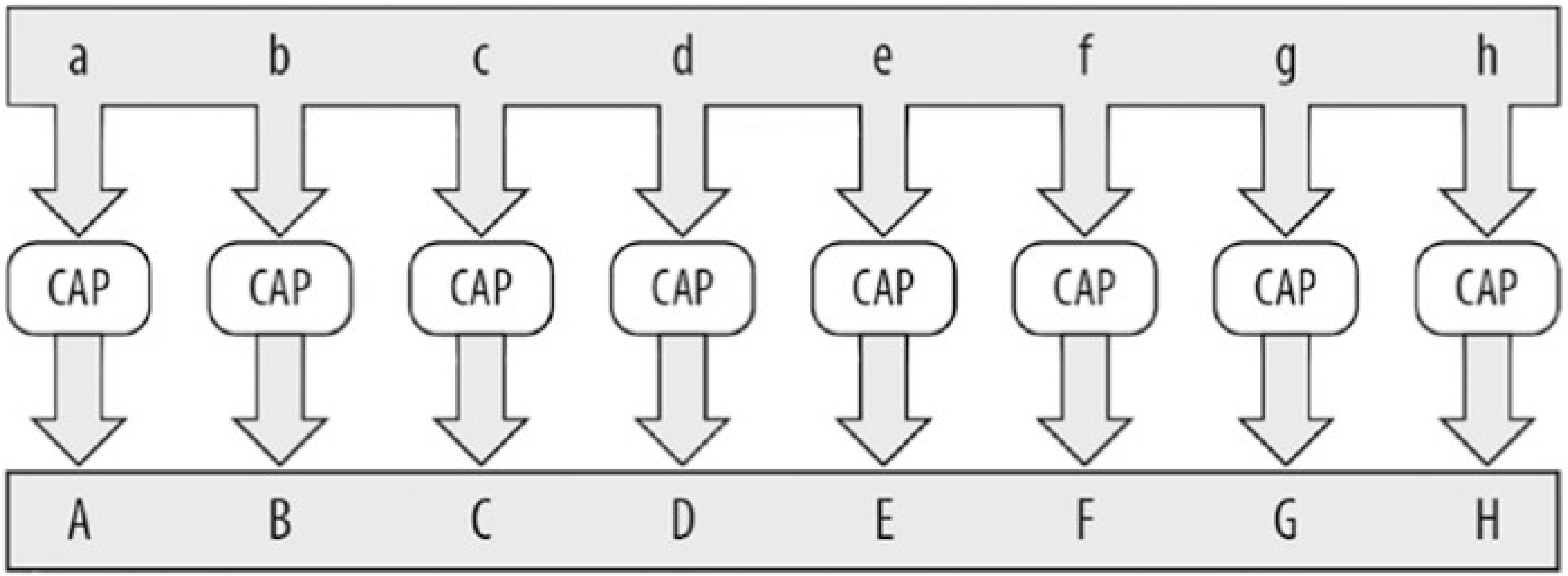
Concurrent vs. Parallel



- Tasks (A) and (B) are only concurrent
- The others are concurrent and parallel

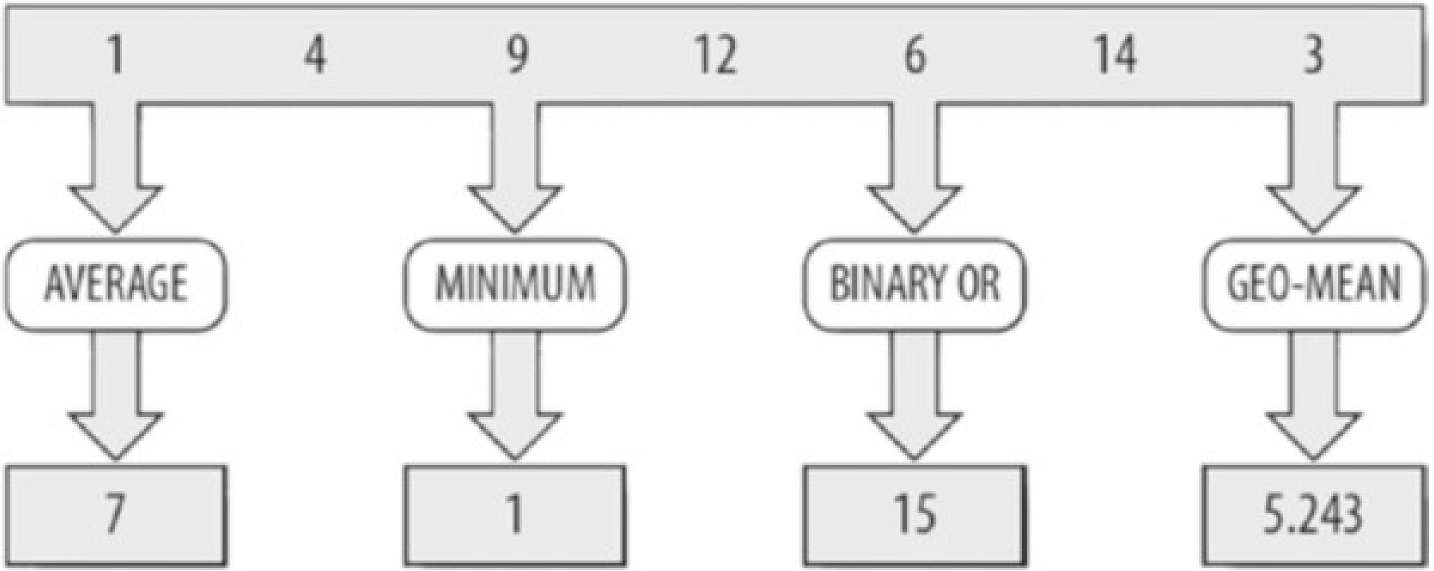
Source [1]

Data Parallelism



Source [1]

Task Parallelism



Source [1]

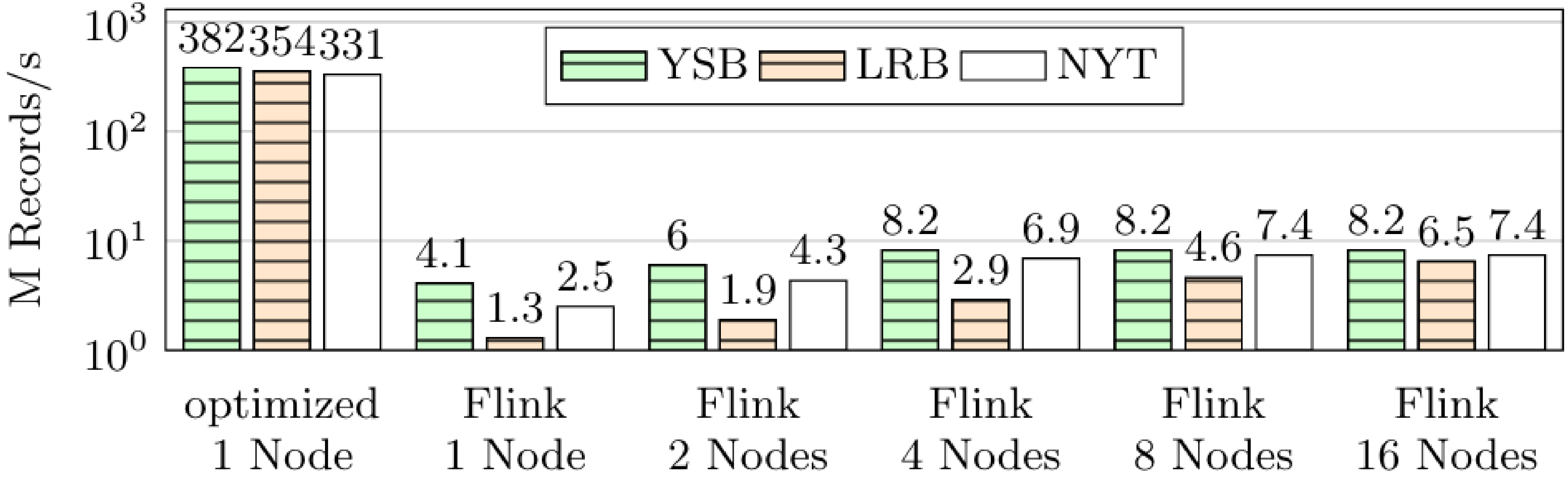
Parallel Computing

- How do we achieve parallelism in computing applications?
- We (still) need to model and program our applications to execute in parallel (in the vast majority of cases).
- Software must be designed to run in parallel: “The free lunch is over.” [Ref 5]
- Different ways were already presented in this course.
- Today we will see how to parallel computing works in **modern C++** using the **standard C++** threads
- Requirements:
  Familiarity with modern C++ features and access to C++17 compiler

Why C++?

Why C++?

It is efficient!



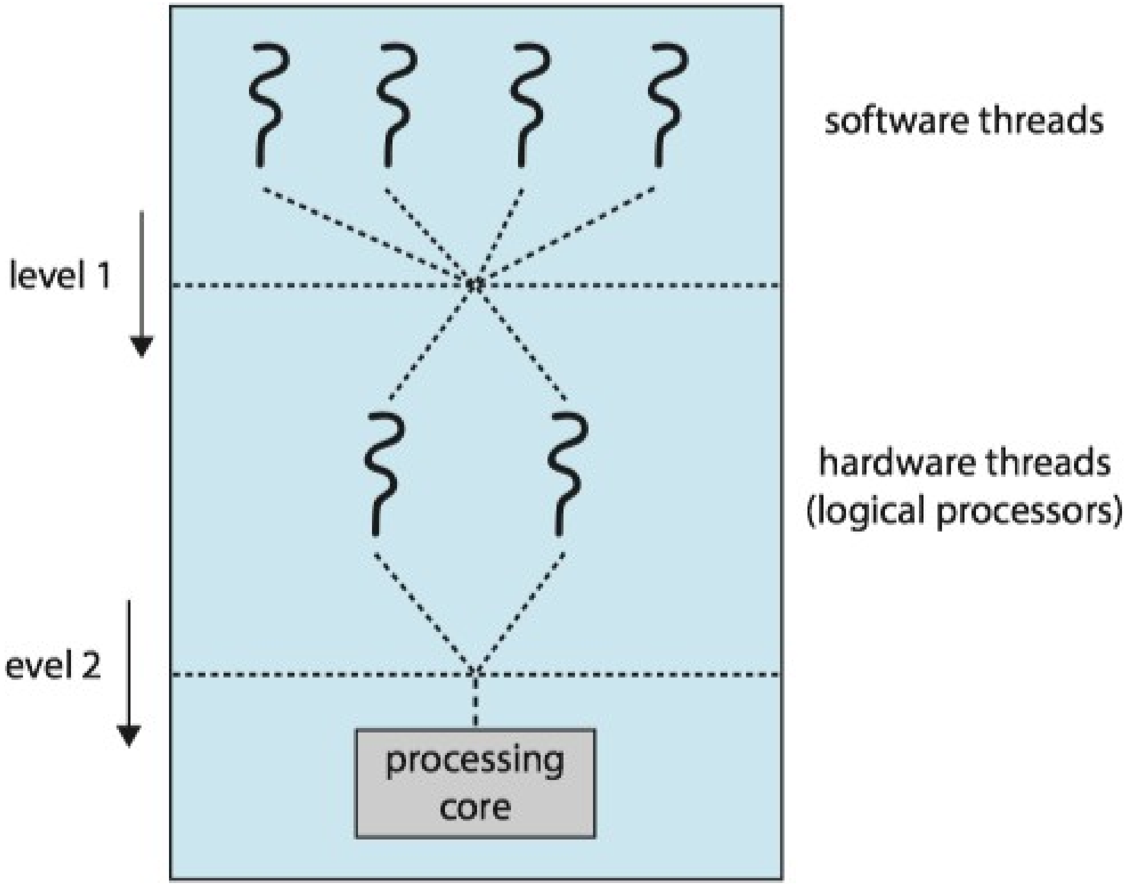
Read more on Zeuch *et al.* **Analyzing Efficient Stream Processing on Modern Hardware**

Concurrency in C++

- C++11 standard provided support for concurrency through multithreading: Standard C++ Thread Library
- Improved support with C++17 and C++20
- No major updates seen (until now) in C++23

What are threads?

- Hardware threads
- Software threads
- std::threads



Source: https://techlarry.github.io/OS

What if we create more threads than available hardware threads?

Standard C++ Threads

- Code examples using C++ thread class
- Implemented with RAII: Resource Acquisition Is Initialization

```cpp
#include <iostream>         // std::cout
#include <thread>           // std::thread
void foo() {
  // do stuff...
}
void bar(int x) {
  // do stuff...
}
int main() {
  std::thread first (foo);     // spawn new thread that calls foo()
  std::thread second (bar,0);  // spawn new thread that calls bar(0)
  std::cout << "main, foo and bar now execute concurrently...\n";
  // synchronize threads:
  first.join();                // pauses until first finishes
  second.join();               // pauses until second finishes
  std::cout << "foo and bar completed.\n";
  return 0;
}
```

Standard C++ Threads

- Many other features: https://cplusplus.com/reference/thread/thread/
  - Arguments
  - Change of ownership
  - Running in background
  - Identifying threads
  - System thread interface
    - Pause threads (this_thread::sleep_for(time))
    - Thread priority
    - Thread affinity “pinning”

Data Shared Between Threads

- There's mostly no problem if all shared data is read-only
- Modifying the shared data can cause problems
- Be careful when sharing data:
  - Problematic race conditions (the threads execution order affects the correctness) data races occur when the threads access the same memory location and one updates it.
  - We need to serialize to guarantee consistency and defined behavior.

Protecting Shared Data

- Critical sections
- Mutex
- Locks
- Deadlock

Why is this topic so relevant?

Parallelism Challenges

- Thinking in parallel
- Locks and mutexes
- Shared mutable state

```
timed_mutex the_mutex;
void task1() {
    cout << "Task1 trying to get lock" << endl;
    the_mutex.lock();
    cout << "Task1 has lock" << endl;
    this_thread::sleep_for(500ms);
    cout << "Task1 releasing lock" << endl;
    the_mutex.unlock();
}
```

Parallelism Challenges

- Locks and Mutexes
- “Locks, can’t live with them, can’t live without them.” [Ref 1]

Why locks are so problematic?

Thread Synchronization
Condition Variables

- From CPP reference:
  “*A condition variable is a synchronization primitive that allows multiple threads to communicate with each other. It allows some number of threads to wait (possibly with a timeout) for notification from another thread that they may proceed. A condition variable is always associated with a mutex.*”

Threads Synchronization
Condition variable example from cplusplus.com

```cpp
#include <iostream>
#include <string>
#include <thread>
#include <mutex>
#include <condition_variable>
std::mutex m;
std::condition_variable cv;
std::string data;
bool ready = false;
bool processed = false;
void worker_thread() {
    std::unique_lock lk(m);
    cv.wait(lk, []{return ready;}); // Wait until main() sends data,
                                       then we own the lock.
    std::cout << "Worker thread is processing data\n";
    data += " after processing";
    processed = true; // Send data back to main()
    std::cout << "Worker thread signals data processing completed\n";
    lk.unlock(); //Manual unlocking is done before notifying
    cv.notify_one();
}
```

```cpp
int main() {
    std::thread worker(worker_thread);
    data = "Example data";
    {
        std::lock_guard lk(m);
        ready = true;
        std::cout << "main() signals data ready\n";
    }
    cv.notify_one();
    {
        std::unique_lock lk(m);
        cv.wait(lk, []{return processed;});// wait for the
                                              worker
    }
    std::cout << "Back in main(), data = " << data << '\n';
    worker.join();
}
```

```
main() signals data ready
Worker thread is processing data
Worker thread signals data processing completed
Back in main(), data = Example data after processing
```

Threads Synchronization

Futures

Facility to obtain values that are returned and to catch exceptions that are thrown by asynchronous tasks

```cpp
#include <iostream>
#include <future>
int task() {
    std::cout << "Task started" << std::endl;
    std::cout << "Task completed" << std::endl;
    return 1;
}
int main() {
    //future that launches a task
    std::future<int> fut1 = std::async(std::launch::async, task);

    // Wait for the result of task
    int result = fut1.get();
    std::cout << "The result is : " << result << std::endl;
    return 0;
}
```

Standard C++ Tasks

- AKA Asynchronous programming
- Contrary of blocking and waiting, tasks can run in background
- Threads vs. tasks

Threads Synchronization
Promises

- std::promise provides means to set a value that can later be read with a std::future object:
  - The waiting thread could block on the future
  - The thread providing the data could use the promise to set the associated value and make the future ready [6].
- Promise: producer/writer
- Future: consumer/reader

Threads Synchronization

Promise Example (from https://cplusplus.com/reference/future/promise/)

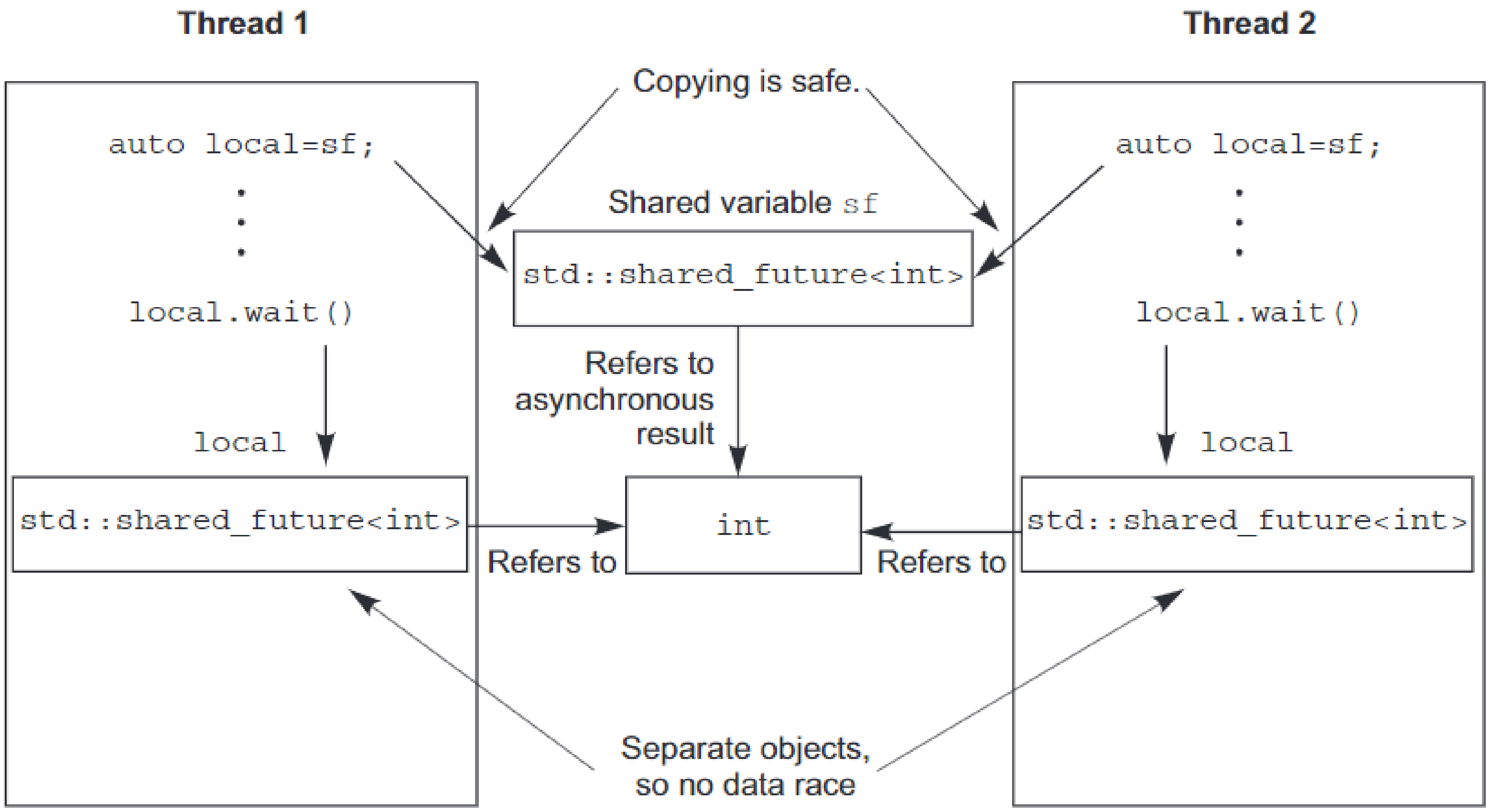
```
#include <iostream>        // std::cout
#include <functional>      // std::ref
#include <thread>          // std::thread
#include <future>          // std::promise, std::future
void print_int (std::future<int>& fut) {
  int x = fut.get();
  std::cout << "value: " << x << '\n';
}
int main () {
  std::promise<int> prom;                       // create promise
  std::future<int> fut = prom.get_future();     // engagement with future
  std::thread th1 (print_int, std::ref(fut));   // send future to new thread
  prom.set_value (10);                          // fulfill promise
                                                // (synchronizes with getting the future)
  th1.join();
  return 0;
}
```

Threads Synchronization

- A problem with futures:
  Data race and undefined behavior when accessing a std::future object from multiple threads (without additional synchronization)

Threads Synchronization

- Solution: shared_future
- Single producer multiple consumers
- Several threads can receive a “value”

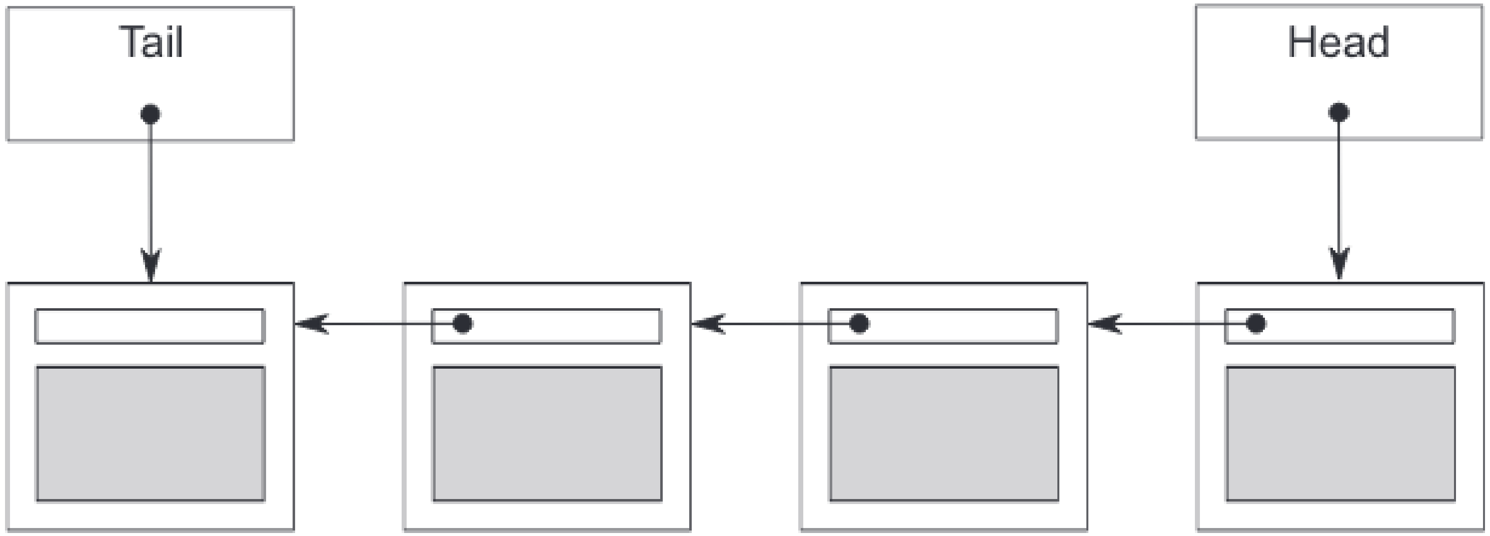


Source [6]

Communication Between Threads

- Thread safe concurrent data structures, such as:
  - Stacks
  - Queues
  - Lists
- (Potentially) Safe and (potentially) efficient threads communication

Queue: Represented as a Single-Linked List [6]



Source [6]

Requirement: Thread Safe Queues

- std::queue FIFO:
  - New data is pushed to end and the oldest data is popped at the “beginning”
  - front() return a reference to the value at the “beginning”
  - pop() no return, removes the element at the “beginning” (C++ constraint for exception safety)
- std::queue is not suitable to be used as a concurrent queue:
  - race conditions in concurrent function call
  - undefined behavior

C++ concurrent data structures

- Needed to share data and synchronize messages
- A queue between producers and consumer threads
- But, C++ does not provide a standard concurrent queue (why?)

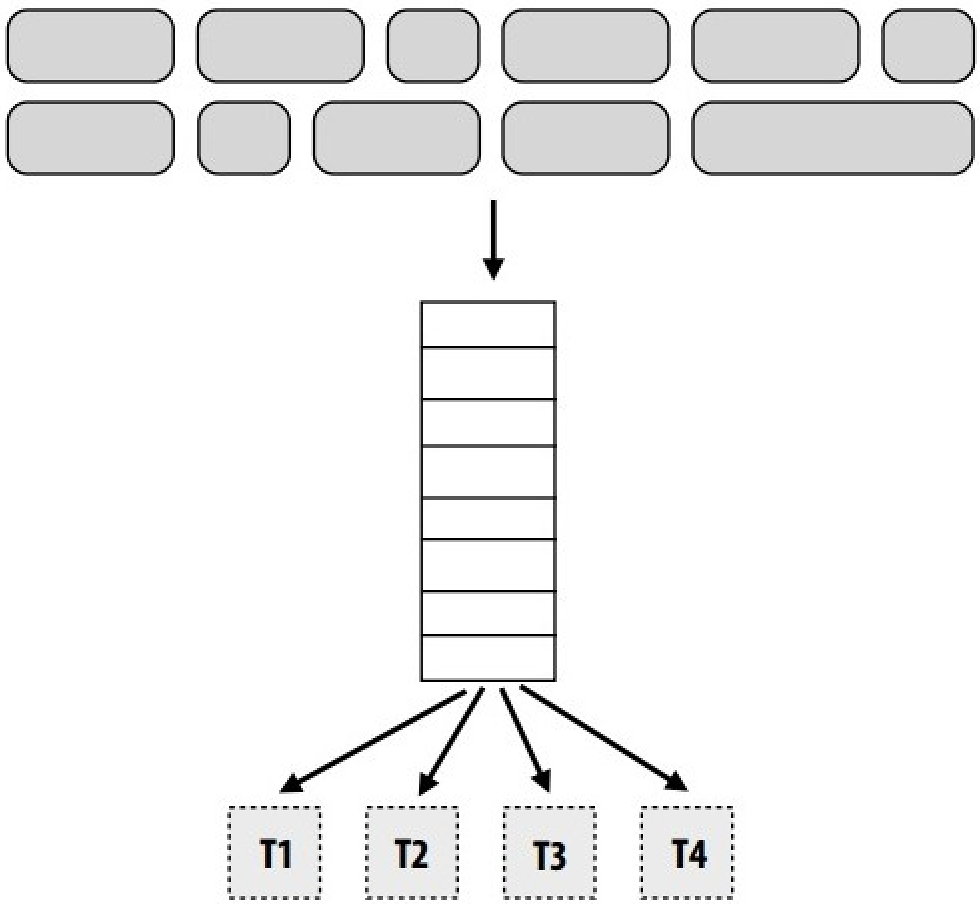
Thread Safe Concurrent Queues

- Simplest solution: Use a wrapper class that protects shared data with member instances:
  - std::queue
  - std::mutex
- Locking a mutex before calling a std::queue member function, then unlocks.
- Only one thread per time can access a given queue member function.

Thread Safe Concurrent Queues

```cpp
template <class T>
class threadSafeQueue {
    std::mutex m;
    std::queue<T> q;
    std::condition_variable cv;
public:
    threadSafeQueue() = default;
    void push(T value) {
        std::lock_guard<std::mutex> lg(m);
        q.push(value);
        cv.notify_one();
    }

    void pop(T& value) {
        std::unique_lock<std::mutex> lg(m);
        cv.wait(lg, [this] {return !q.empty();});
        value = q.front();
        q.pop();
    }
};
```



Code example from [7]
Representation from [8]

Using the Thread Safe Concurrent Queues

```
threadSafeQueue<int> myQueue;
void consumer() {
    int data;
    std::cout << "The consumer is running" << std::endl;
    myQueue.pop(data);                              // Get a value from the queue
    std::cout << "Consumer received: " << data << std::endl;
}
void producer() {
    std::cout << "The producer is running..." << std::endl;
    myQueue.push(10);                          // Push the data into the queue
    std::cout << "The producer has pushed some data" << std::endl;
}
int main() {
    auto cons = async(std::launch::async, consumer); //starting consumer
    auto prod = async(std::launch::async, producer); //starting producer
    cons.wait();
    prod.wait();
}
```

```
The consumer is running
The producer is running...
The producer has pushed some data
Consumer received: 10
```

```
The producer is running...
The consumer is runningThe producer has
pushed some data

Consumer received: 10
```

Standard C++ Parallelism

- Is it enough to achieve scalability?
- Not for the many use-cases!
- Why?

C++ Thread pools

- Scalability
- Use properly the CPU resources
- Manage the overhead of thread creation

```
#include <iostream>
#include <chrono>
#include <functional>
#include "concurrentQueue.h"
using namespace std;
// Example of a computation
void processTask(int taskId) {
  cout << "Processing task " << taskId << " in thread "
       << this_thread::get_id() << endl;
  this_thread::sleep_for (chrono::seconds(1)); // task processing
}
int main() {
  const int numTasks = 10;
  const int numThreads = 3 ;//std::thread::hardware_concurrency();
  cout << "Executing " << numTasks << " tasks in a thread pool of: "
       << numThreads << " threads" << endl;
  ThreadPool threadPool(numThreads);
  for (int i = 0; i < numTasks; ++i) {
     threadPool.enqueue(processTask, i);
  }
  return 0;
}
```

ThreadPool Class

```
class ThreadPool {
  public:
      ThreadPool(size_t num_threads) {
          for (size_t i = 0; i < num_threads; ++i) {
              threads_.emplace_back([this] {
                  while (true) {
                      std::function<void()> task;
                      {
                          std::unique_lock<std::mutex> lock(mutex_);
                          condition_.wait(lock, [this] {
                              return stop_ || !tasks_.empty();
                          });
                          if (stop_ && tasks_.empty()) {
                              return;
                          }
                          task = std::move(tasks_.front());
                          tasks_.pop();
                      }
                      task();
                  }
              });
          }
      }
```

```
      ~ThreadPool() {
          {
              std::unique_lock<std::mutex> lock(mutex_);
              stop_ = true;
          }
          condition_.notify_all();
          for (std::thread& thread : threads_) {
              thread.join();
          }
      }
      template<typename F, typename... Args>
      auto enqueue(F&& f, Args&&... args) -> std::future<typename
                std::result_of<F(Args...)>::type> {
          using return_type = typename std::result_of<F(Args...)>::type;
          auto task = std::make_shared<std::packaged_task<return_type()>>(
                  std::bind(std::forward<F>(f), std::forward<Args>(args)...)
          );
          std::future<return_type> result = task->get_future();
          {
              std::unique_lock<std::mutex> lock(mutex_);
              tasks_.emplace([task]() {
                  (*task)();
              });
          }
          condition_.notify_one();
          return result;
      }

  private:
      std::vector<std::thread> threads_;
      std::queue<std::function<void()>> tasks_;
      std::mutex mutex_;
      std::condition_variable condition_;
      bool stop_ = false;
};
```

Executing Thread Pools Example

```cpp
#include <iostream>
#include <chrono>
#include <functional>
#include "concurrentQueue.h"
using namespace std;
// Example of a computation
void processTask(int taskId) {
  cout << "Processing task " << taskId << " in thread "
       << this_thread::get_id() << endl;
  this_thread::sleep_for (chrono::seconds(1)); // task processing
}
int main() {
  const int numTasks = 10;
  const int numThreads = 3 ;//std::thread::hardware_concurrency();
  cout << "Executing " << numTasks << " tasks in a thread pool of: "
       << numThreads << " threads" << endl;
  ThreadPool threadPool(numThreads);
  for (int i = 0; i < numTasks; ++i) {
     threadPool.enqueue(processTask, i);
  }
  return 0;
}
```

```
Executing 10 tasks in a thread pool of: 3 threads
Processing task 0 in thread 140446390413056
Processing task 1 in thread 140446373627648
Processing task 2 in thread 140446382020352
Processing task 3 in thread 140446373627648
Processing task 4 in thread 140446390413056
Processing task 5 in thread 140446382020352
Processing task 6 in thread 140446373627648
Processing task 7 in thread Processing task
1404463904130568 in thread
140446382020352
Processing task 9 in thread 140446373627648
```

Practical Example

Parallelize the prime number calculation with C++ threads

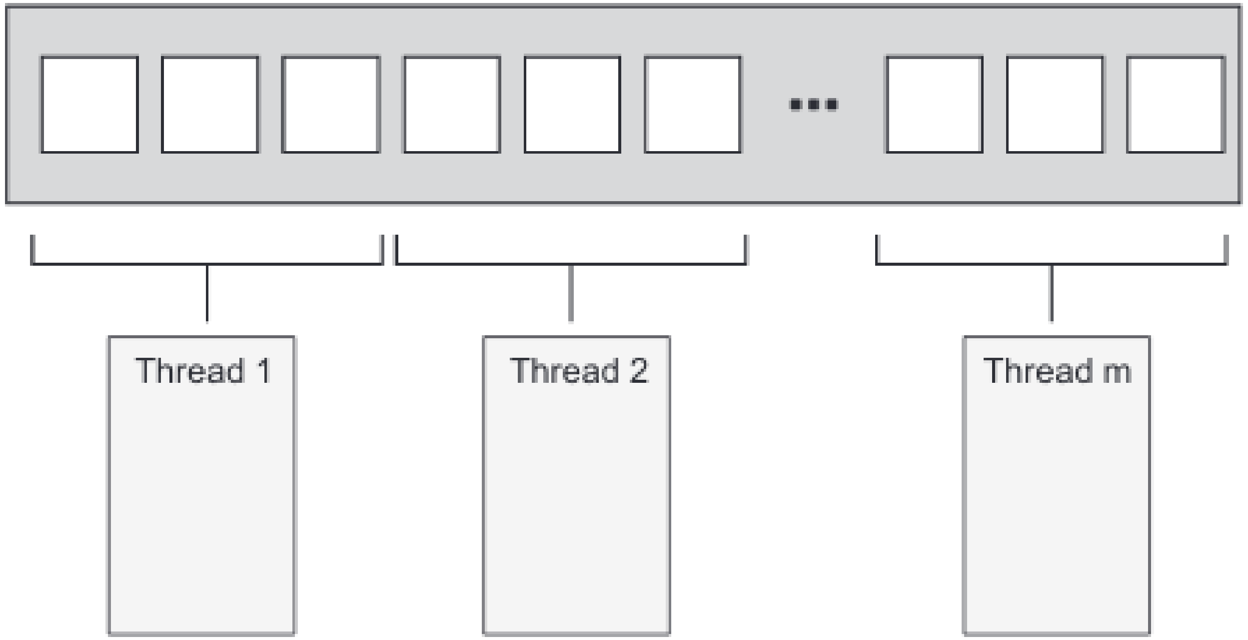
```cpp
// Function that checks if a number is prime
bool isPrime(int num) {
    if (num <= 1)
        return false;

    for (int i = 2; i < num; ++i) {
        if (num % i == 0)
            return false;
    }
    return true;
}
```

Practical Example

Naive thread pool with fixed chunks (AKA static assignment)

- Very low runtime overhead
- Works very well when the workload is fairly divided between the worker threads (balanced workload)



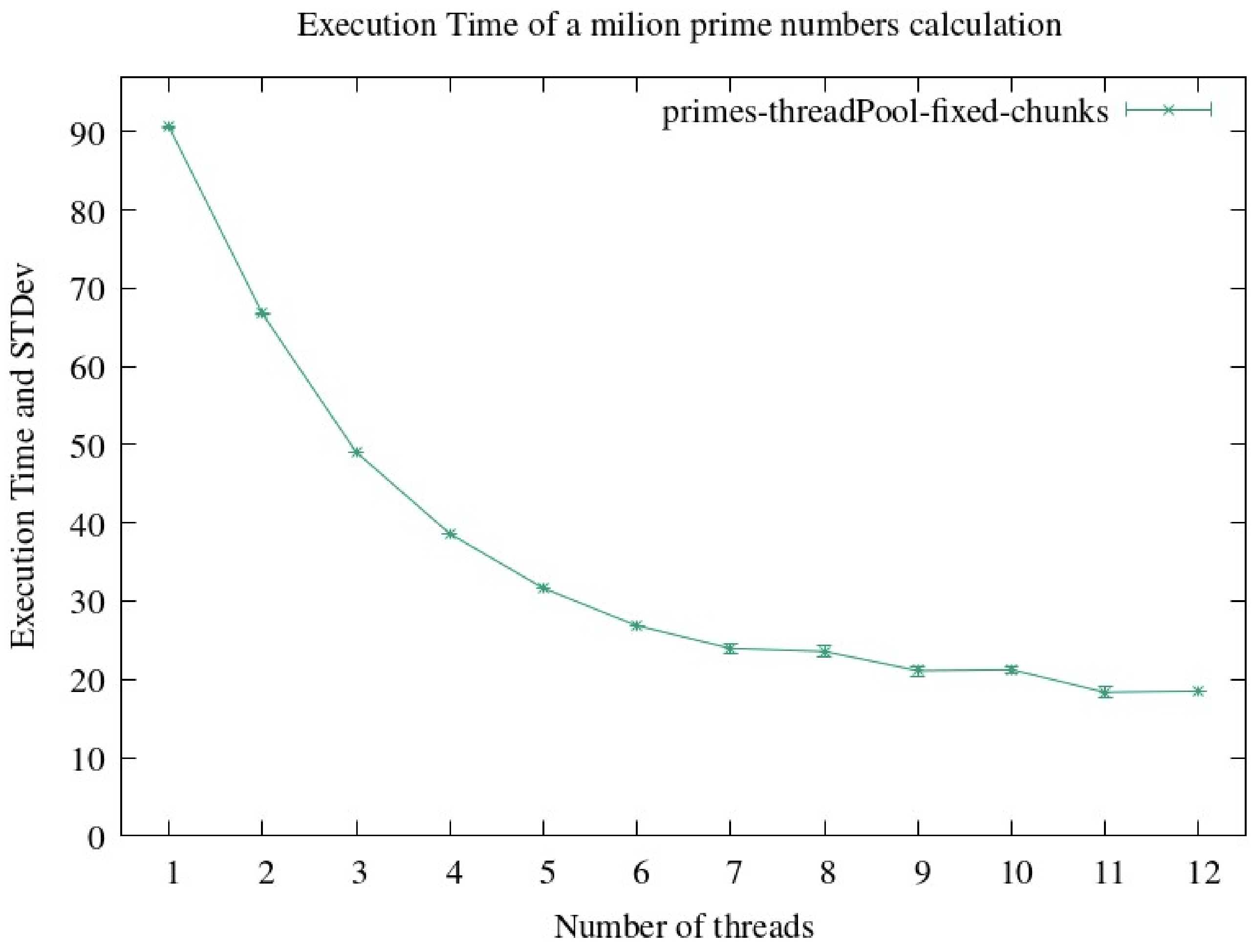
Source [6]

Parallel Prime Numbers

```
int main(int argc, char *argv[]){
   int interval=0, threadPoolSize=0;
   /* interval and threadPoolSize are argos code here and removed for visual clarity */
   const int rangeStart = 1;
   const int rangeEnd = interval;
   std::vector<std::thread> threads;
   std::vector<int> threadResults(threadPoolSize, 0);
   int chunkSize = (rangeEnd - rangeStart + 1) / threadPoolSize;
   int remaining = (rangeEnd - rangeStart + 1) % threadPoolSize;
   int start = rangeStart;
   for (int i = 0; i < threadPoolSize; ++i) {
       int end = start + chunkSize - 1;
       if (i < remaining)
           ++end;
       threads.emplace_back([start, end, i, &threadResults]() {
           threadResults[i] = countPrimesInRange(start, end);
       });
       start = end + 1;
   }
   for (auto& thread : threads) {
       thread.join();
   }
   int totalPrimes = 0;
   for (int result : threadResults) {
       totalPrimes += result;
   }
   /* Here we calculate the exec time */
   return 0;
}
```

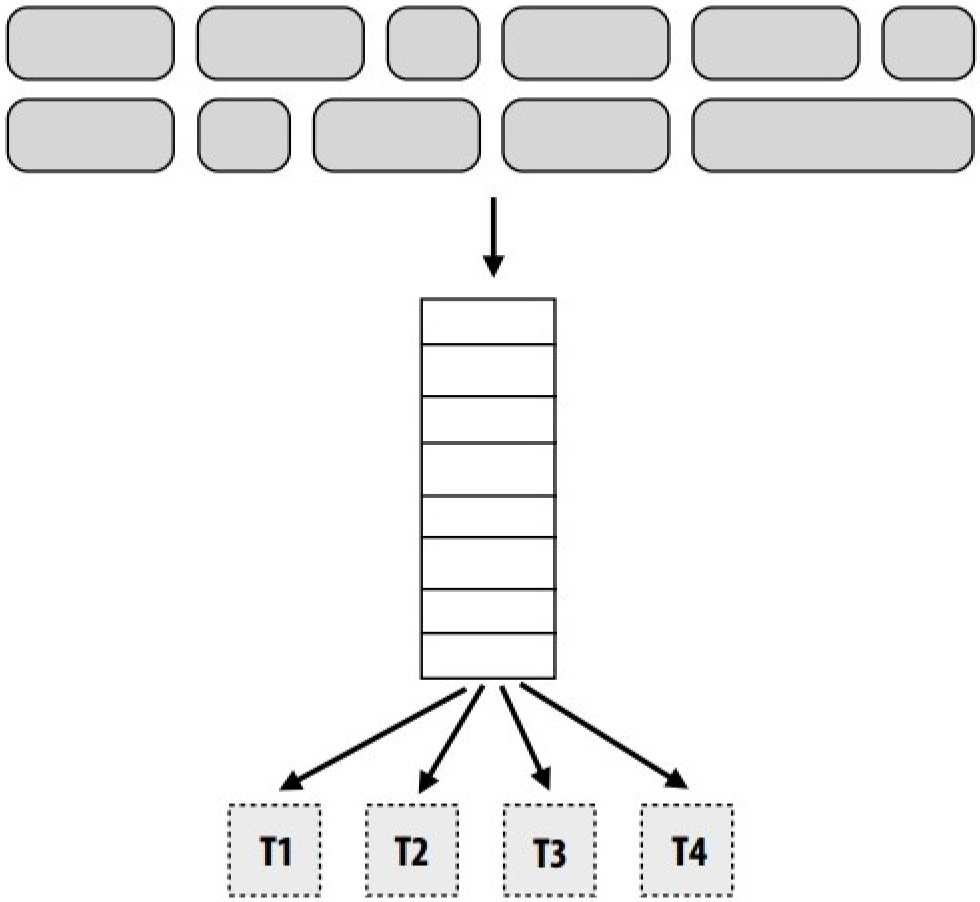
Parallel Prime Numbers

- Evaluation in a machine with 6 cores and 12 Hyperthreads
- Why this performance?
- Is it optimal?
- Remember: it works very well when the workload is fairly divided between the worker threads (balanced workload)



How can the performance be further improved?

What about using a concurrent queue?



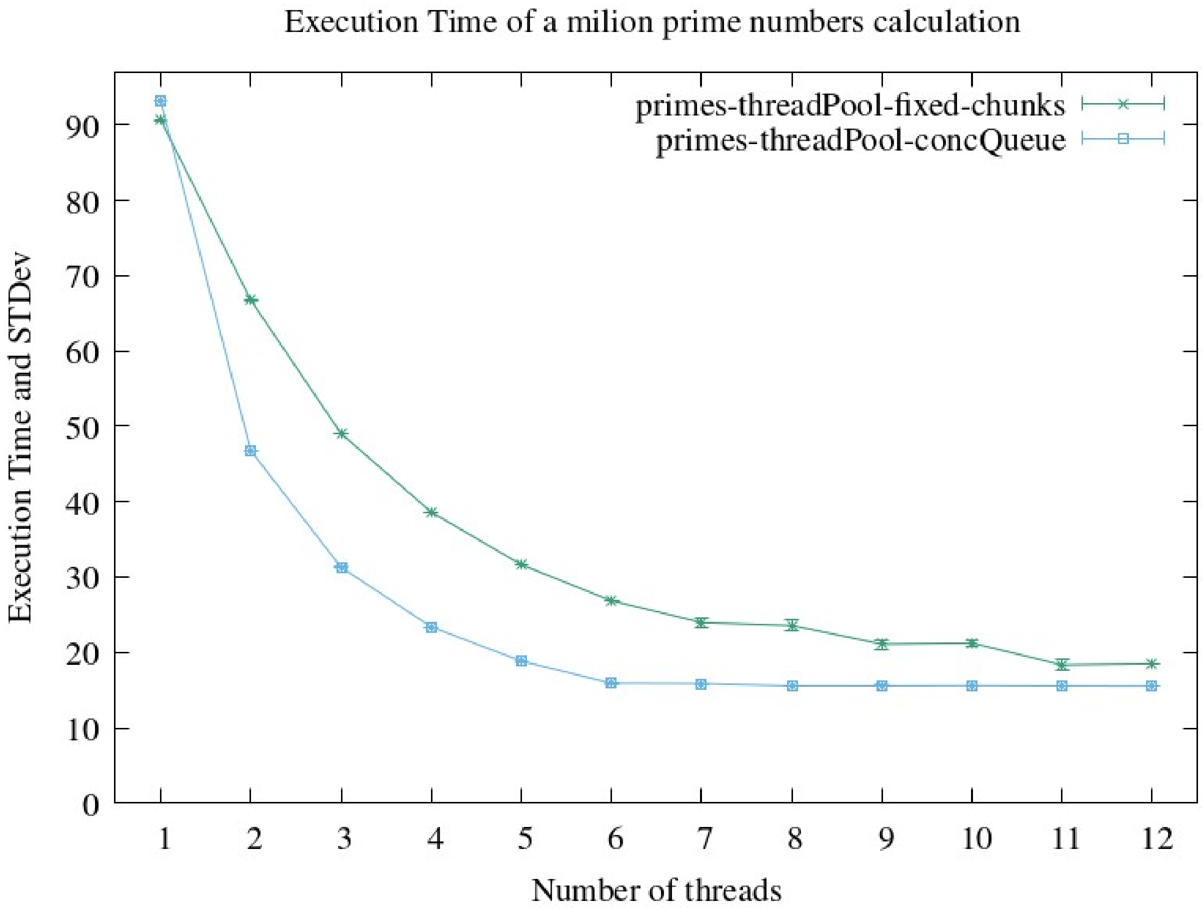
Parallel Prime Numbers with a Concurrent Queue

```
#include <iostream>
#include <vector>
#include <chrono>
#include "concurrentQueue.h"
int main(int argc, char *argv[]) {
   /* interval and threadPoolSize are argos code here and removed for visual clarity */
   ThreadPool pool(threadPoolSize);
   std::vector<std::future<bool>> results;
   for (int i = 0; i <  interval; ++i) {
       results.emplace_back(pool.enqueue([](int value) {
           if (value <= 1)
           return false;
           // Check from 2 to n-1
           for (int i = 2; i < value; i++){
           if (value % i == 0)
               return false;
           }
           return true;
       }, i));
   }
   int primerCount = 0;
   for (auto& result : results) {
       bool isPrime =  result.get();
       if (isPrime)
       {
          primerCount++;
       }
   }
   /* Here we calculate the exec time */
   return 0;
}
```

What performance can we expect?

Parallel Prime Numbers

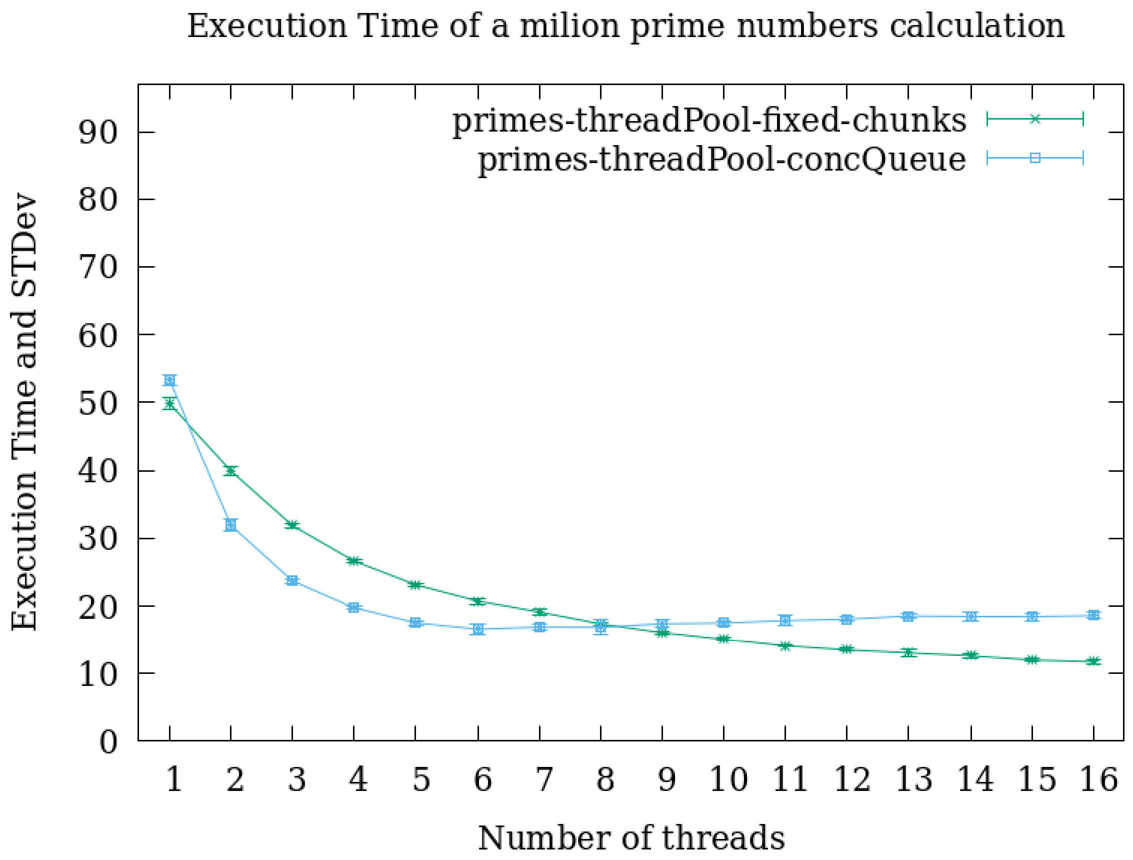
- Evaluation in a machine with 6 cores and 12 Hyperthreads
- Why this performance?



What if we run in a more powerful machine?

Parallel Prime Numbers

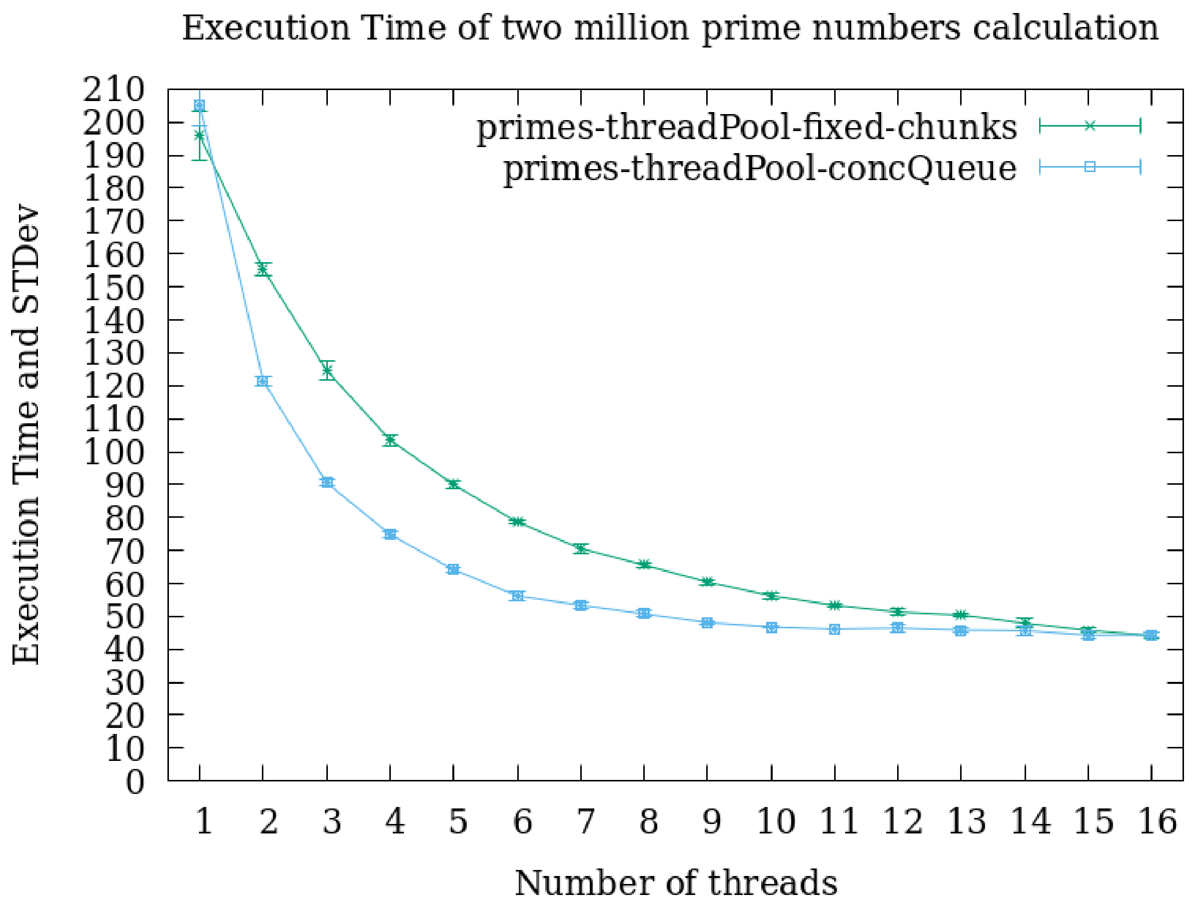
- Evaluation in a machine with 8 cores and 16 Hyperthreads
- Why this performance?



What if we increase the workload?

Parallel Prime Numbers

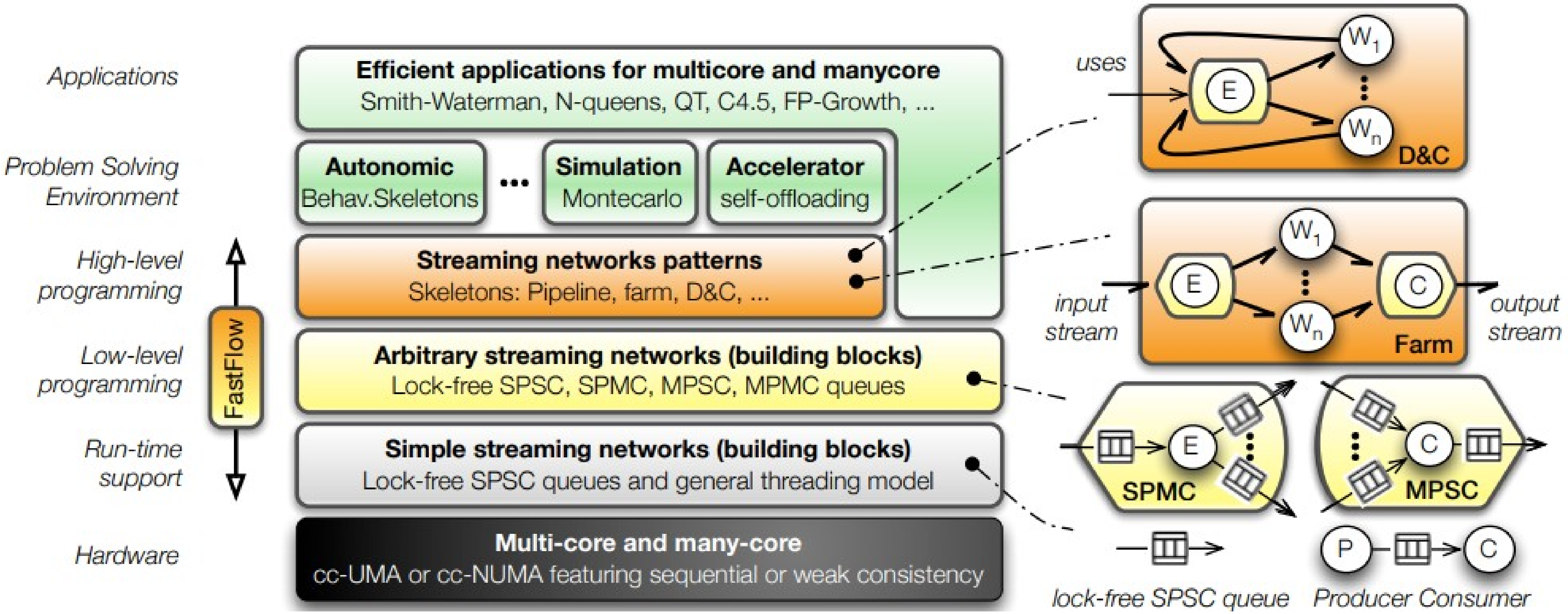
- Evaluation in a machine with 8 cores and 16 Hyperthreads
- Why this performance?



C++ concurrent data structures

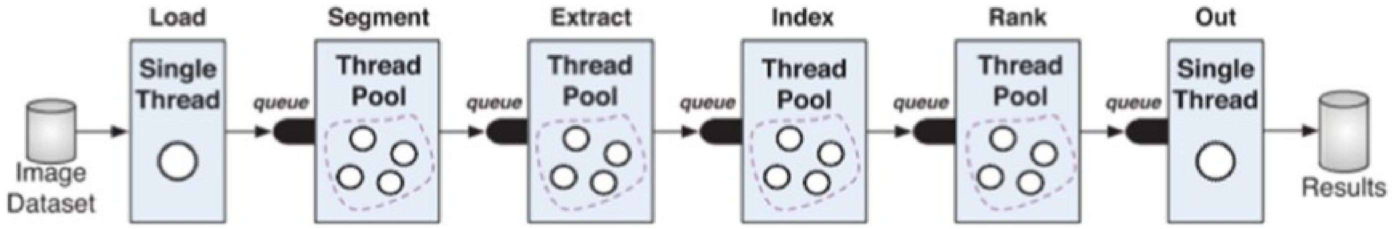
- Lock-free concurrent data structures?
- **A data structure where more than one thread can access the data structure concurrently**
- *“a lock-free queue might allow one thread to push and one to pop but break if two threads try to push new items at the same time”*
- *“A wait-free data structure is a lock-free data structure with the additional property that every thread accessing the data structure can complete its operation within a bounded number of steps, regardless of the behavior of other threads”*
- *“Writing wait-free data structures correctly is extremely hard”*
  memory **ordering** constraints, **atomic** operations, making **changes visible** to other threads in a **exact order**.
- Quotes from Williams [6]

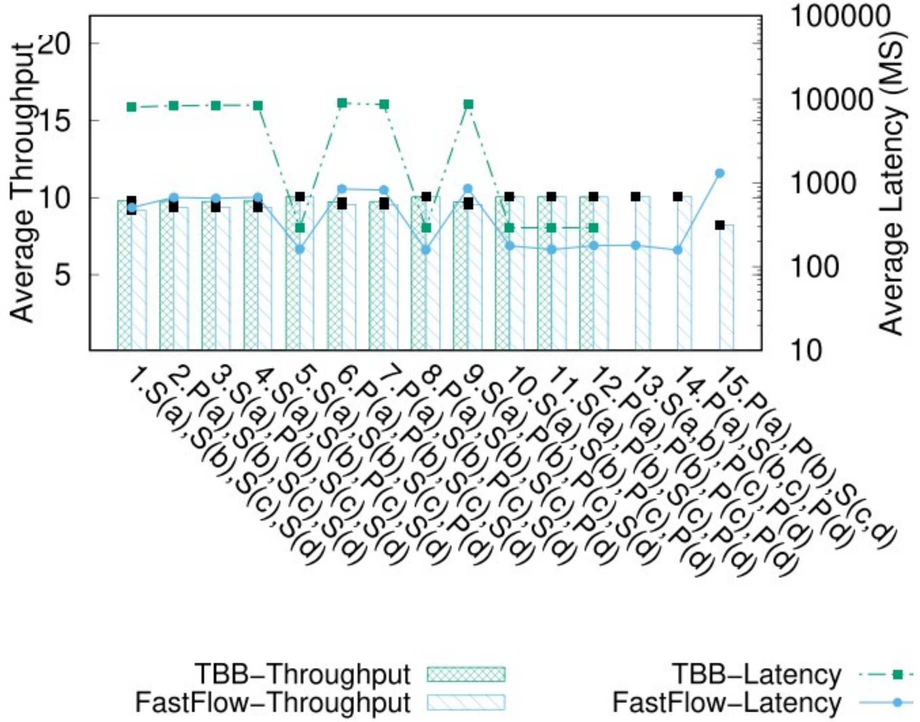
Lock-Free single producer single consumer (SPSC) Queues



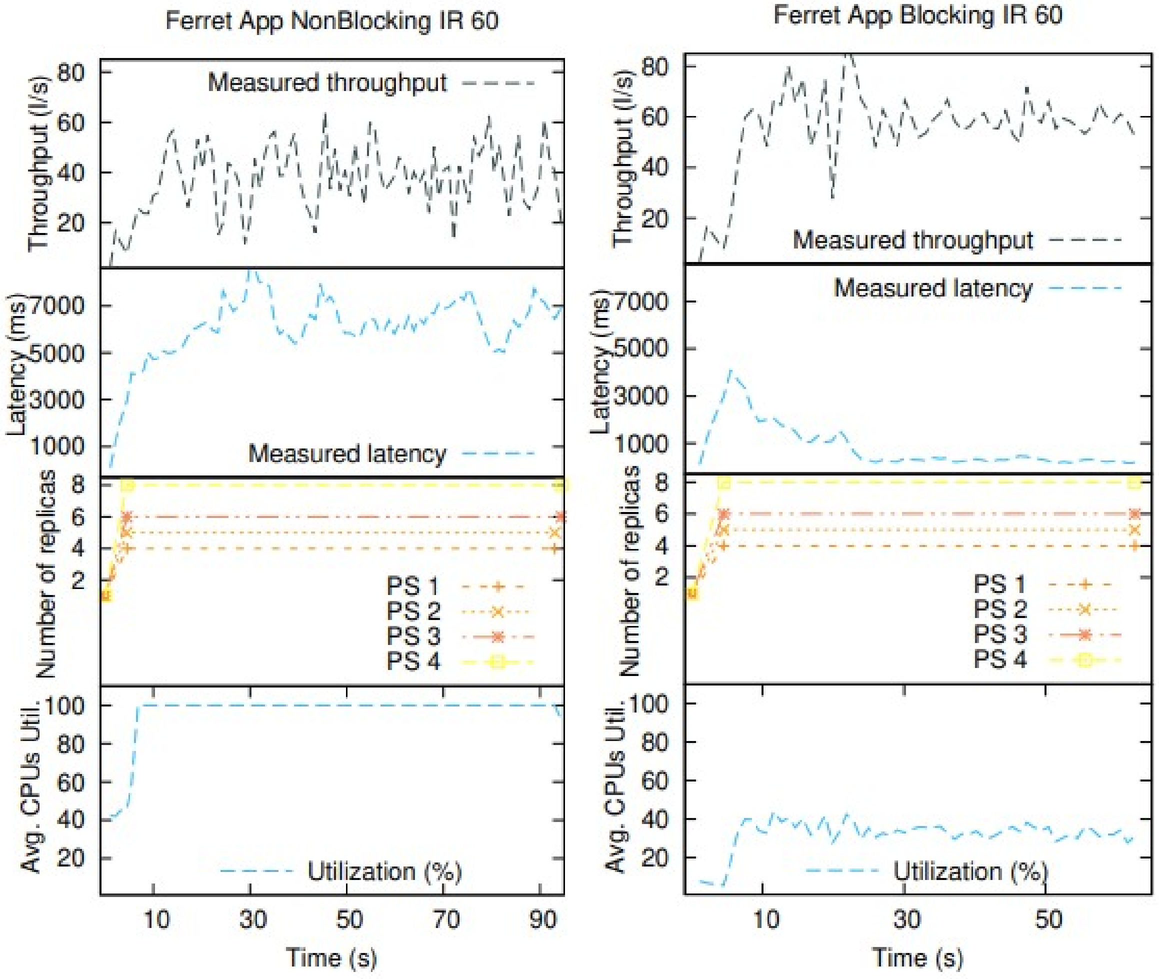
Read more about FastFlow in: https://doi.org/10.1007/978-3-642-32820-6_65

Lock-Free single producer single consumer (SPSC) Queues





Lock-Free single producer single consumer (SPSC) Queues



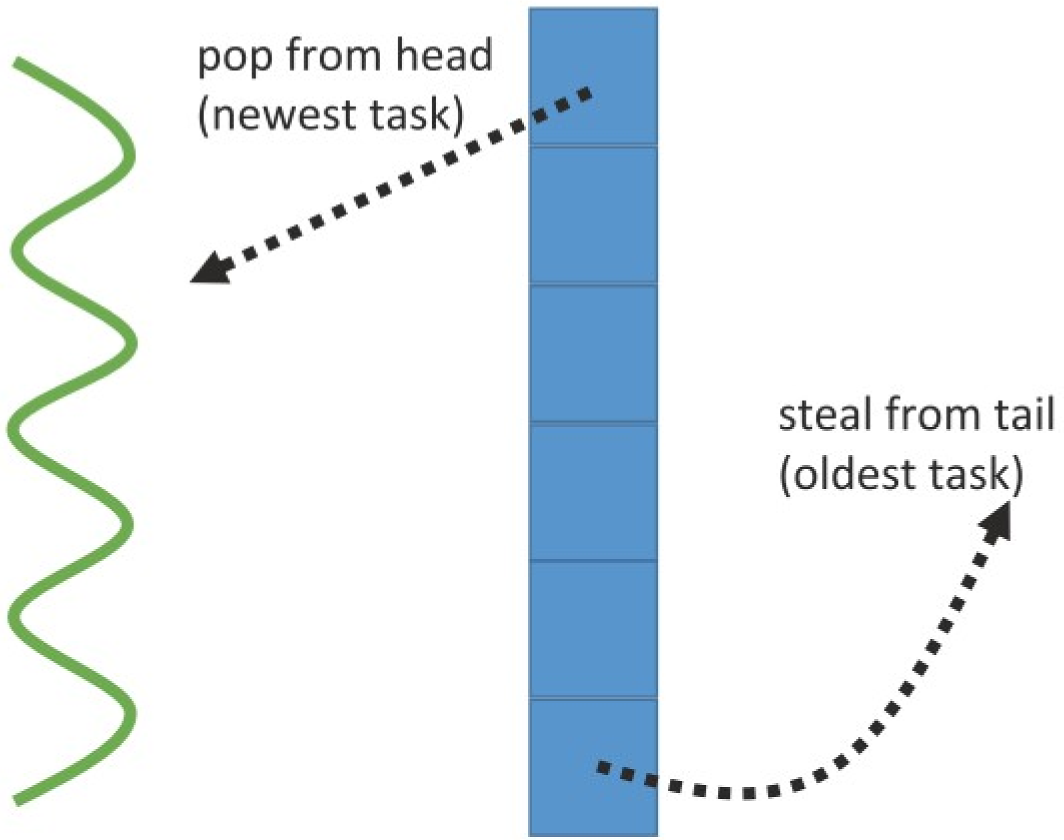
(a) Non-blocking Mode. (b) Blocking Mode.

Lock-free Concurrent Data Structures

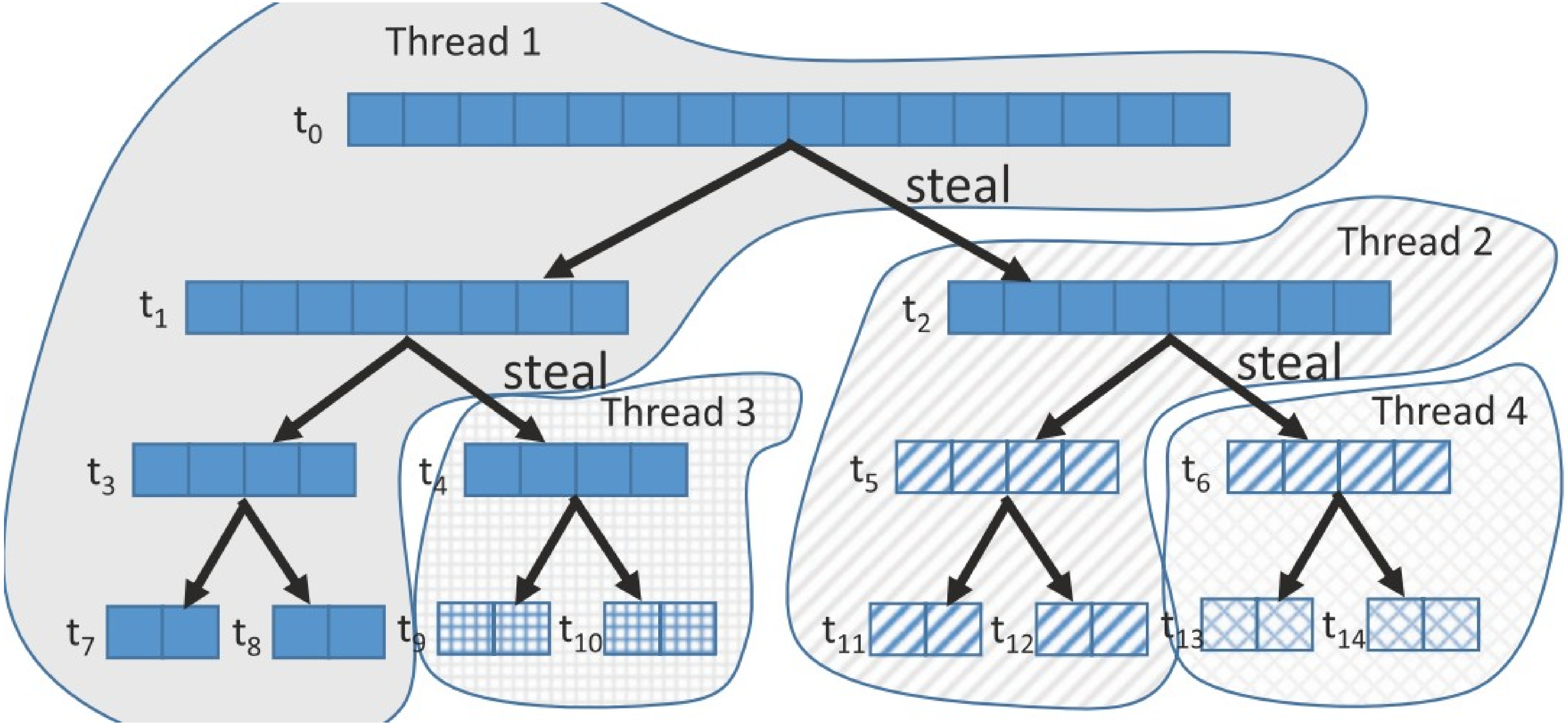
- Very strong reasons are needed to write one
- The benefits have to outweighs the costs
- **Advantages**
  - Every thread can progress no matter the status of others
  - Robustness: if a thread fails only its data is lost
- **Challenges**
  - "Although it can increase the potential for concurrency of operations on a data structure and reduce the time an individual thread spends waiting, **it may well decrease overall performance**" [6]
  - The needed atomic operations can be much slower than the non-atomic ones

Work-stealing?

- “work stealing is a rare event” [6]
- Work-stealing with intel Threading Building Blocks (One API) [1]



Work-stealing with intel Threading Building Blocks (One API) [1]



Standard C++ Parallel Algorithms

- C++17 added parallel algorithms to the standard library, with only a new first parameter for the execution policy. Example [6]:

```
std::vector<int> my_data;
std::sort(std::execution::par,my_data.begin(),my_data.end());
```

- Parallel algorithms require at least C++17 and ltbb (install libtbb-dev)

Standard C++ Parallel Algorithms

Parallel For

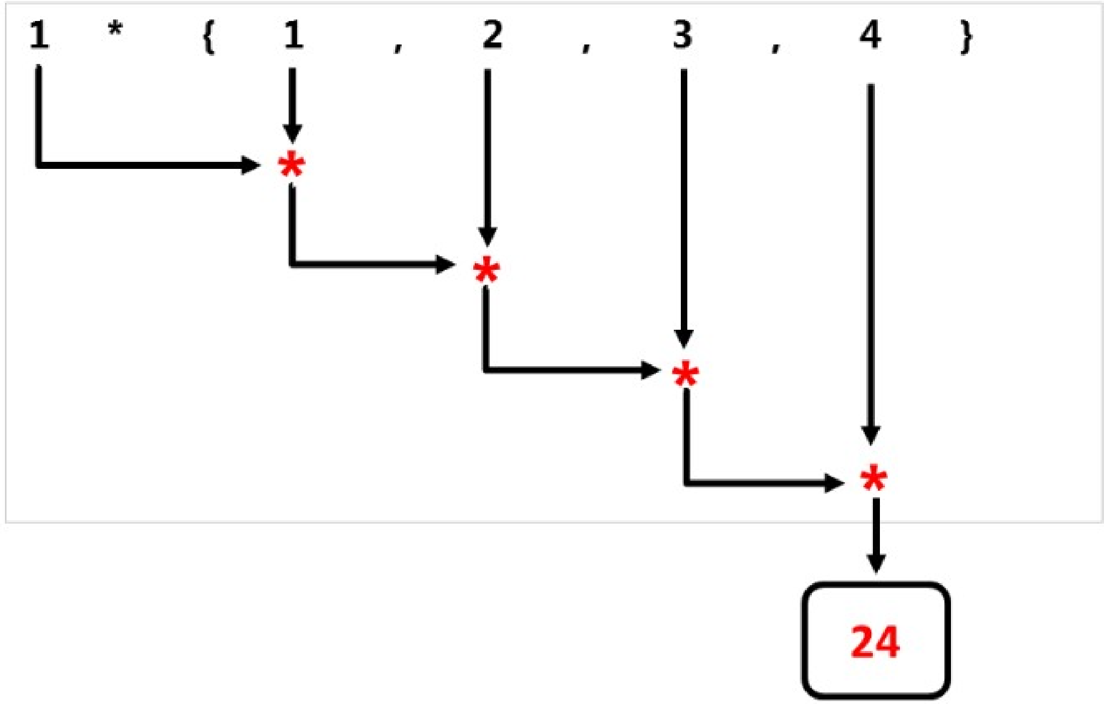
```
#include <future>
#pragma omp parallel for
for (unsigned i = 0; i < v.size(); ++i){
   do_stuff(v[i]);
}
```

```
std::sort(std::execution::par,
          v.begin(), v.end(), do_stuff);
```

Standard C++ Parallel Algorithms

std::accumulate (from left successively applying the operator)

```
std::vector<int> v{1, 2, 3, 4};
std::accumulate(v.begin(), v.end(), 1, [](int a, int b){ return a * b; });
```

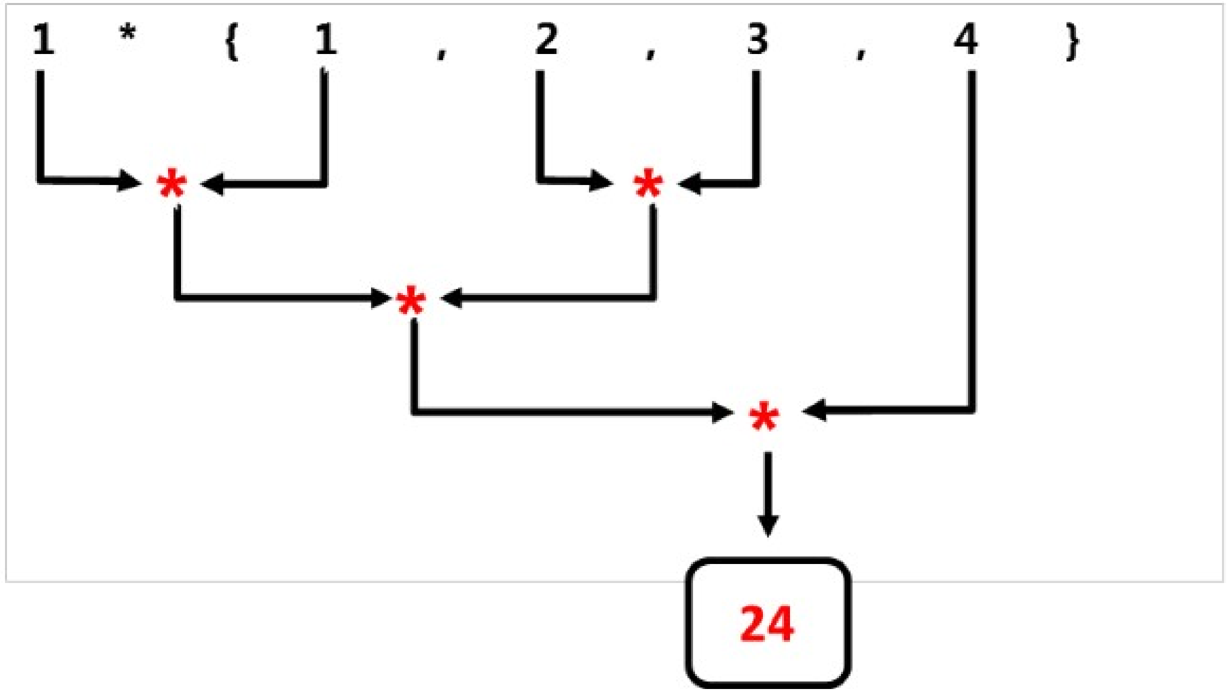


From www.modernescpp.com

Standard C++ Parallel Algorithms

std::reduce (applying the operator in a non-deterministic way)

```
std::vector<int> v{1, 2, 3, 4};
std::reduce(std::execution::par, v.begin(), v.end(), 1 , [](int a, int b){ return a * b; });
```



From www.modernescpp.com

Standard C++ Parallel Algorithms

std::transform_reduce

`first, last`	-	the range of elements to apply the algorithm to
`init`	-	the initial value of the generalized sum
`reduce`	-	binary *FunctionObject* that will be applied in unspecified order to the results of `transform`, the results of other `reduce` and `init`.
`transform`	-	unary or binary *FunctionObject* that will be applied to each element of the input range(s). The return type must be acceptable as input to `reduce`.

Standard C++ Parallel Algorithms

std::transform_reduce

```cpp
// Example modified from https://dev.to/sandordargo/the-big-stl-algorithms-tutorial-reduce-operations-3f1m
#include <iostream>
#include <numeric>
#include <vector>
int main() {
    std::vector v {1, 2, 3, 4, 5};
    int calc = std::transform_reduce(v.begin(), v.end(), 0,
                   [](int l, int r) {return l+r;},
                   [](int i) {return i*i;});
    std::cout << "The calculated result is: " << calc << std::endl;
}
```

std::transform_reduce

```
#include <iostream>
#include <numeric>
#include <vector>
int main() {
    using namespace std;
    std::vector v {1, 2, 3, 4, 5};
    int calc = std::transform_reduce(
        v.begin(),
        v.end(),
        0, //beginning of the vector
        [](int l, int r) {
            cout << "Reduce - L: " << l << " & R: " << r << " local: " << l+r << endl;
            return l+r;
            }, //reduce (sum transformed values)
        [](int i) {
            cout << "Transform - i:  " << i << " local: (" << i << "*" << i << "): " << i*i << endl;
            return i*i;
        } //transform: multiplies the values
    );
    std::cout << "The calculated result is: " << calc << std::endl;
}
```

```
Transform - i:  2 local: (2*2): 4
Transform - i:  1 local: (1*1): 1
Reduce - L: 1 & R: 4 local: 5
Transform - i:  4 local: (4*4): 16
Transform - i:  3 local: (3*3): 9
Reduce - L: 9 & R: 16 local: 25
Reduce - L: 5 & R: 25 local: 30
Reduce - L: 0 & R: 30 local: 30
Transform - i:  5 local: (5*5): 25
Reduce - L: 30 & R: 25 local: 55
The calculated result is: 55
```

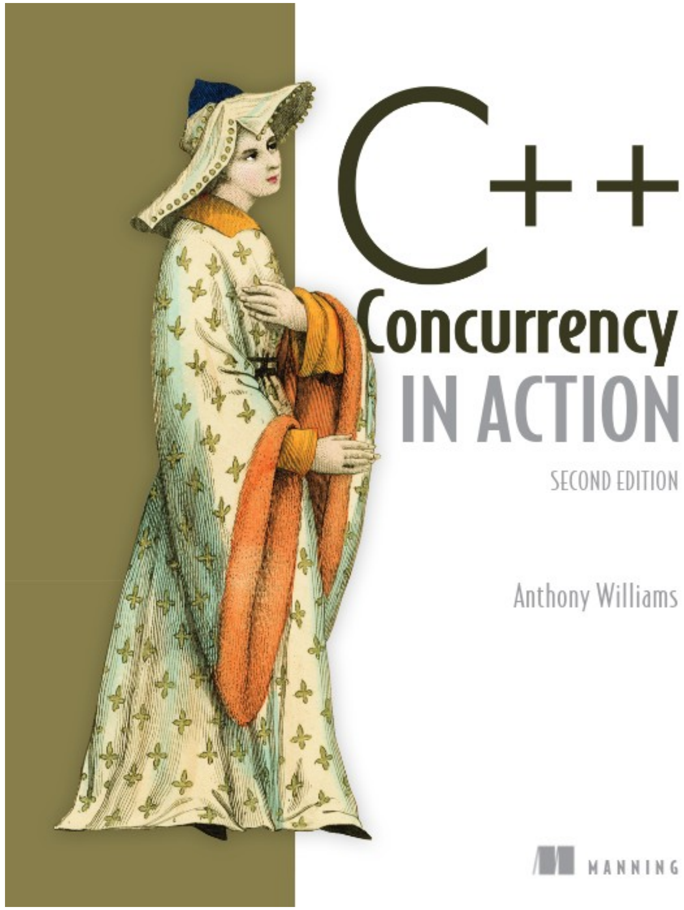
Standard C++ Parallel Algorithms

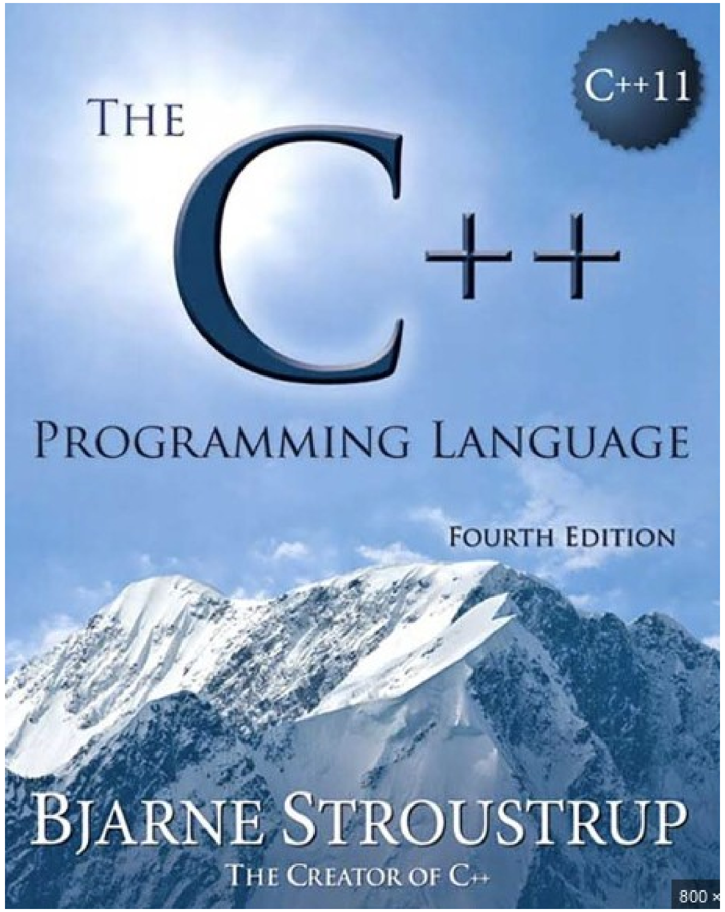
- See the list of parallelized algorithms: https://en.cppreference.com/w/cpp/algorithm

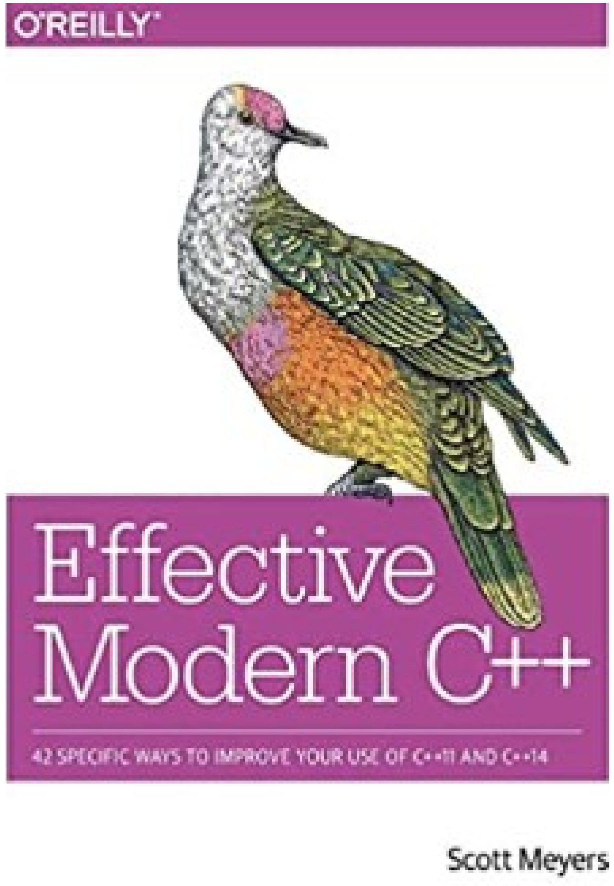
Standard C++ Parallel Algorithms

- **Is C++ STL scalable enough for all use cases?**
  - Probably not. That is why it can be extended to run in accelerators (e.g., GPGPUs, FPGAs) or multiple machines (distributed computing).
  - Other programming languages have a better support for distributed computing than C++

Further Resources







References

1. Voss, Michael, Rafael Asenjo, and James Reinders. Pro TBB: C++ parallel programming with threading building blocks. Vol. 295. New York: Apress, 2019.
2. Meyers, Scott. Effective modern C++: 42 specific ways to improve your use of C++ 11 and C++ 14. " O'Reilly Media, Inc.", 2014.
3. Griebler, Dalvan, Adriano Vogel, Daniele De Sensi, Marco Danelutto, and Luiz G. Fernandes. "Simplifying and implementing service level objectives for stream parallelism." The Journal of Supercomputing 76 (2020): 4603-4628.
4. Aldinucci, Marco, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. "Fastflow: High-Level and Efficient Streaming on Multicore." Programming multi-core and many-core computing systems (2017): 261-280.
5. Sutter, Herb. "The free lunch is over: A fundamental turn toward concurrency in software." Dr. Dobb’s journal 30, no. 3 (2005): 202-210.
6. Williams, Anthony. C++ concurrency in action. Simon and Schuster, 2019.
7. Raynard, James. Learn Multithreading with Modern C++. Independently published, 2022
8. http://15418.courses.cs.cmu.edu/spring2013/article/13

LINZ INSTITUTE
OF TECHNOLOGY

Thank you!

Univ.-Prof. Dr. Alois Zoitl, alois.zoitl@jku.at
LIT | Cyber-Physical Systems Lab
Johannes Kepler University Linz