Expressing parallelism with C++ and tbb

Felice Pantaleo

CERN Experimental Physics Department

felice@cern.ch

You will learn...

- std::threads
- locks/mutual execution
- atomics
- Threading Building Blocks

2

A quick recap

Threads

- A thread is an execution context, a set of register values
- Defines the instructions to be executed and their order
- A CPU core fetches this execution context and starts running the instructions: the thread is running
- When the CPU needs to execute another thread, it *switches the context* , *i.e.* saving the previous context and loading the new one
  - Context switching is expensive
  - Avoid threads jumping from a CPU core to another

4

count number of 5s

```
array[N]
numberOf5 = 0
for i in [0,N[:
    if array[i] == 5
        numberOf5++
return numberOf5
```

```
numberOf5 = 0

nThreads = 4

count5(array, tId):

  beg = tId*N/nThreads

  end = beg + N/nThreads

  for i in [beg,end[:
         if array[i] == 5:
            numberOf5++
```

Data Race

- A race condition occurs when multiple tasks read from and write to the same memory without proper synchronization.
- The "race" may finish correctly sometimes and therefore complete without errors, and at other times it may finish incorrectly.
- If a data race occurs, the behavior of the program is undefined.

count number of 5s

```
array[N]
numberOf5 = 0
for i in [0,N[:
    if array[i] == 5
         numberOf5++
return numberOf5
```

```
numberOf5 = 0

nThreads = 4

count5(array, tId):

  beg = tId*N/nThreads

  end = beg + N/nThreads

  for i in [beg,end[:
         if array[i] == 5:
            lock()
            numberOf5++
            unlock()
```

count number of 5s

```
array[N]
numberOf5 = 0
for i in [0,N[:
    if array[i] == 5
         numberOf5++
return numberOf5
```

```
numberOf5 = 0

nThreads = 4

count5(array, tId):

  privateResult = 0

  beg = tId*N/nThreads

  end = beg + N/nThreads

  for i in [beg,end[:

      if array[i] == 5:
            privateResult++

  lock()
  numberOf5 += privateResult
  unlock()
```

std::threads – Hello World

```
#include <thread>
#include <iostream>
int main()
{

}
```

- compile with
- `g++ std_threads.cpp -pthread -o std_threads -std=c++20`

std::threads – Hello World

```
#include <thread>
#include <iostream>
int main()
{


}
```

Define a function that prints Hello world

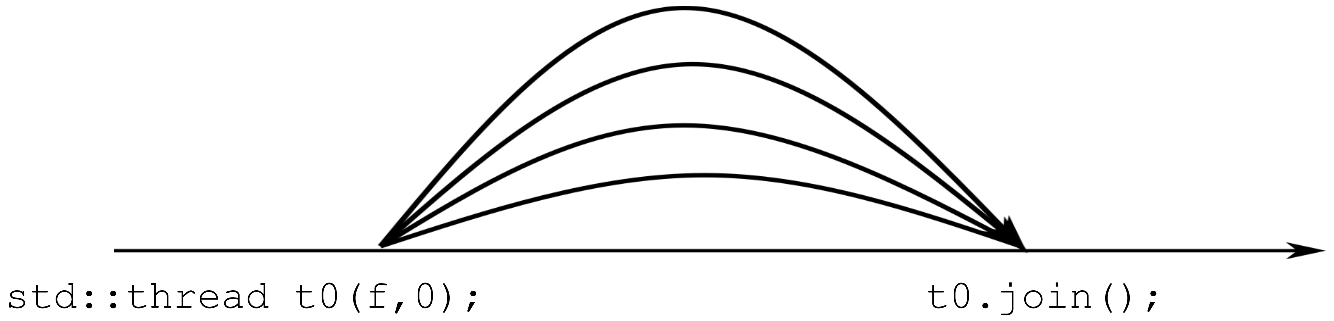
```
void f(int i){
  std::cout << "Hello world from thread " << i << std::endl;
}
```

std::threads – Hello World

```
#include <thread>
#include <iostream>
int main()
{
   auto f = [](int i){
   std::cout << "hello world from thread " << i << std::endl;
  };
//Construct a thread which runs the function f
  std::thread t0(f,0);

//and then destroy it by joining it
  t0.join();
}
```

Fork-join

- The construction of a thread is asynchronous, fork
- Threads execute independently
- join is the synchronization point with the main thread



Need more threads?

```
auto n = std::thread::hardware_concurrency();
std::vector<std::thread> v;
for (auto i = 0; i < n; ++i) {
     v.emplace_back(f,i);
}
for (auto& t : v) {
    t.join();
}
```

std::mutex

Avoiding that multiple threads access a shared variable

Use it together with a scoped lock

When using two or more mutexes, pay attention to deadlocks

```
#include <mutex>
std::mutex myMutex;
...
{
  std::scoped_lock myLock(myMutex);
  //critical section begins here
  std::cout << "Only one thread at a time" << std::endl;
} // ends at the end of the scope of myLock
```

Before we move on, measuring time

```
#include <chrono>

...

auto start = std::chrono::steady_clock::now();

  f(i);

auto stop = std::chrono::steady_clock::now();

std::chrono::duration<double> dur= stop - start;

std::cout << dur.count() << " seconds" << std::endl;
```

- `f()` is the function that you want to measure
- **Be careful, asynchronous functions return immediately: remember to synchronize before stopping the timer.**

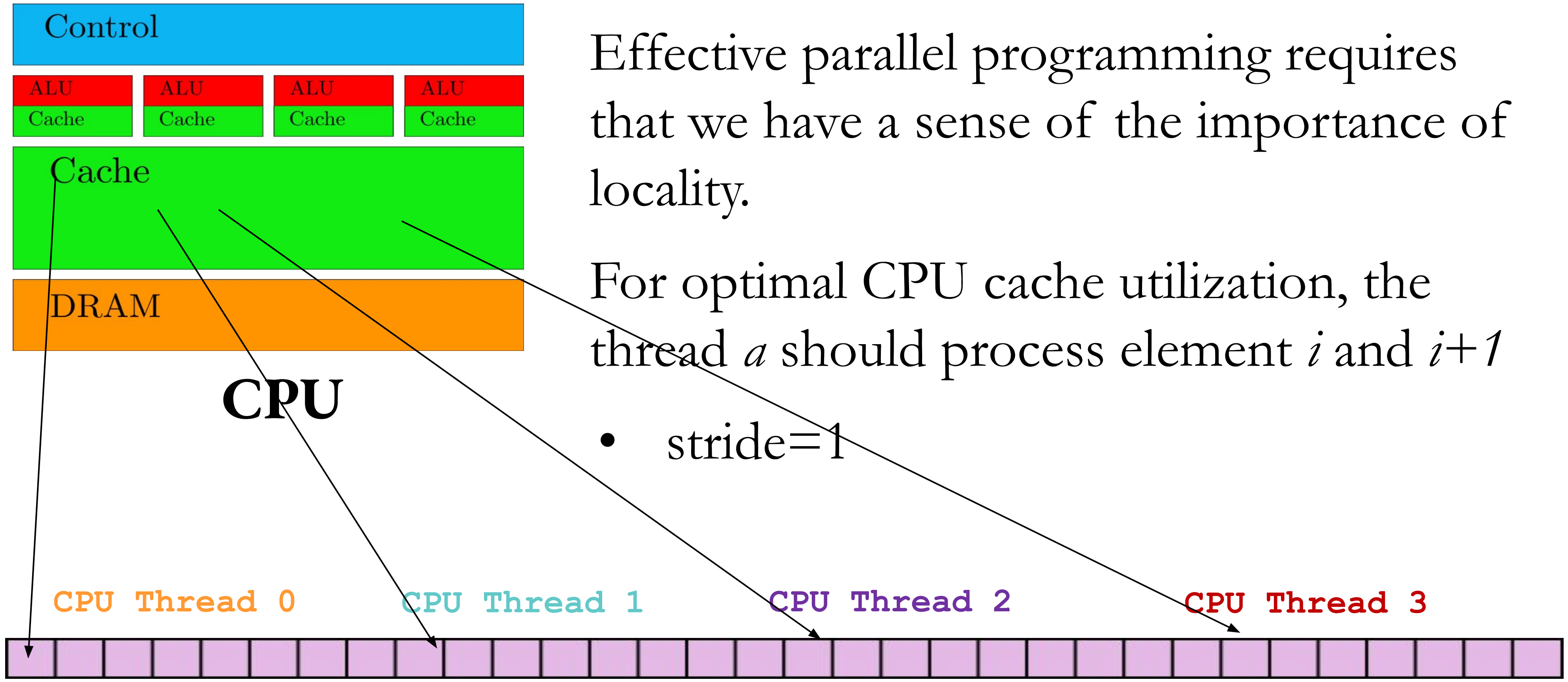
Exercise 1

- You want to sum the elements of a vector in parallel using 4 threads
- Accumulate the sum in the variable sum
- Let's start by creating a thread
- Brainstorming time!

Transferring ownership of a thread

```
void some_function();
void some_other_function();
std::thread t1(some_function);
std::thread t2=std::move(t1);
t1=std::thread(some_other_function);
```

Memory access patterns: cached



Effective parallel programming requires that we have a sense of the importance of locality.

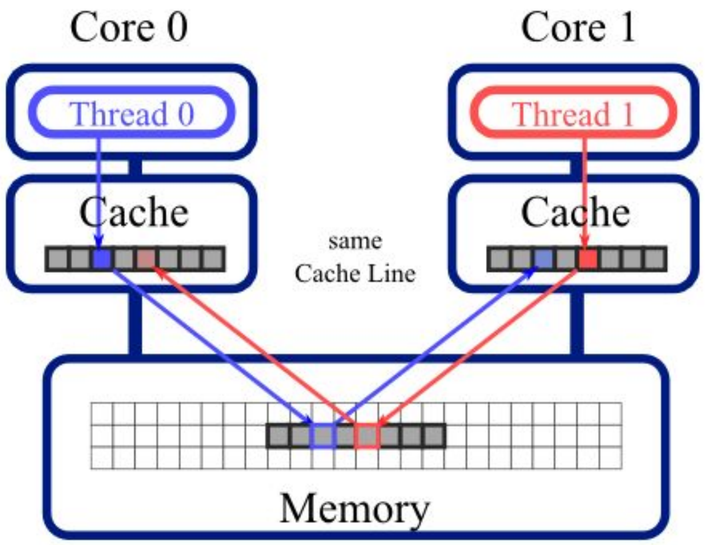
For optimal CPU cache utilization, the thread *a* should process element *i* and *i+1*

- stride=1

False Sharing

- Problems of sharing arise when two threads access different words that share the same cache line.
- The problem is that a cache line is the unit of information interchange between processor caches.
- If one processor modifies a cache line and another processor reads the same cache line, the line must be moved from one processor to the other, even if the two processors are dealing with different words within the line.
- False sharing can hurt performance because cache lines can take hundreds of clocks to move (going through higher level caches or main memory)

False Sharing



- Suppose that:
  - a cache line is 64bytes
  - two threads (x and y) run on processors that share their cache
  - we have two arrays `int A[500], B[500]`
  - the end of `A` and the beginning of `B` are in the same cache line
  - thread x modifies `A[499]`, and loads the corresponding cache-line in cache
  - thread y modifies `B[0]`
- The processor needs to flush the cache lines, reloading the cache for thread x and invalidating the cache for thread y
- A possible solution is padding

std::atomic

- Atomic types:
  - encapsulate a value whose access is guaranteed to not cause data races
  - other threads will see the state of the system before the operation
  - started or after it finished, but cannot see any intermediate state
  - can be used to synchronize memory accesses among different threads
  - at the low level, atomic operations are special hardware instructions
  - (hardware guarantees atomicity)
- The primary std::atomic template may be instantiated with any TriviallyCopyable type T
- Common architectures have atomic fetch-and-add instructions for integers

```
#include <atomic>
std::atomic<int> x = 0; int a = x.fetch_add(42);
```

- reads from a shared variable, adds 42 to it, and writes the result back: **all in one indivisible step**

Trivially Copyable

- Trivially copyable
- The primary std::atomic template may be instantiated
- with any `TriviallyCopyable` type T
  - Continuous chunk of memory
  - Copying the object means copying all bits (memcpy)
  - No virtual functions, noexcept constructor

```
std::atomic<int> i; // OK
std::atomic<double> x; // OK
struct S { long x; long y; };
std::atomic<S> s; // OK!
```

`std::atomic<T>`

- read and write operations are always atomic
- `std::atomic<T>` provides operator overloads only for atomic operations (incorrect code does not compile)

```
std::atomic<int> x{0}
++x;
x++;
x += 1;
x |= 2;
x *= 2; //this is not atomic and will not compile
int y = x * 2; // atomic read of x
x = y + 1; // atomic write of x
x = x + 1; // atomic read and then atomic write
x = x * 2; // atomic read and then atomic write
int z = x.exchange(y); // Atomically: z = x; x = y;
```

Atomic references

- In real life, we usually want to perform atomic operations when the object is shared among different threads, forgetting about its atomicity in portion of the code where it is not contented
- The std::atomic_ref class template applies atomic operations to the object it references
- For the lifetime of the atomic_ref object, the object it references is considered an atomic object

```
int x = 0;
std::atomic_ref<int> atomic_x(x);
…
// later inside another thread
atomic_x+= 5;
```

Expressing Parallelism with Threading Building Blocks

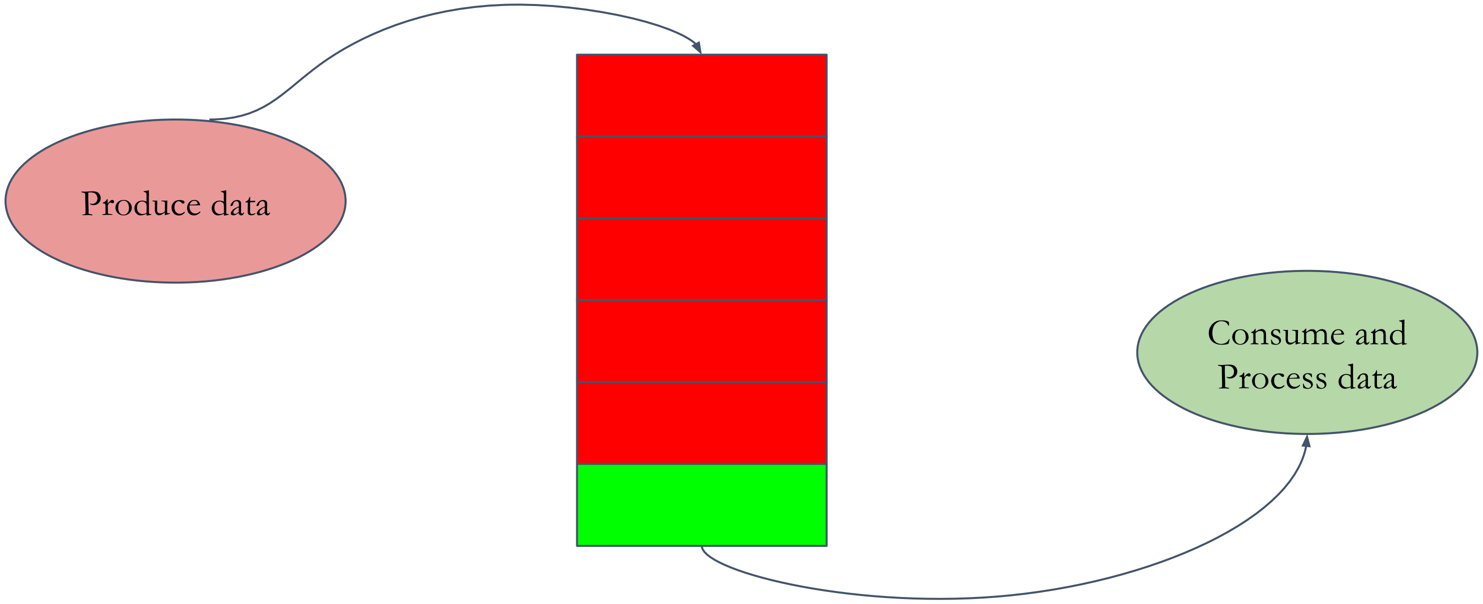
Why TBB?

- OneAPI Threading Building Blocks is an open source library which allows to express parallelism on CPUs in a C++ program https://github.com/oneapi-src/oneTBB/
- Parallelizing for loops can be tedious with std::threads
- One wants to achieve *scalable* parallelism, easily
- To use the TBB library, you specify tasks, not threads, and let the library map tasks onto threads in an efficient manner

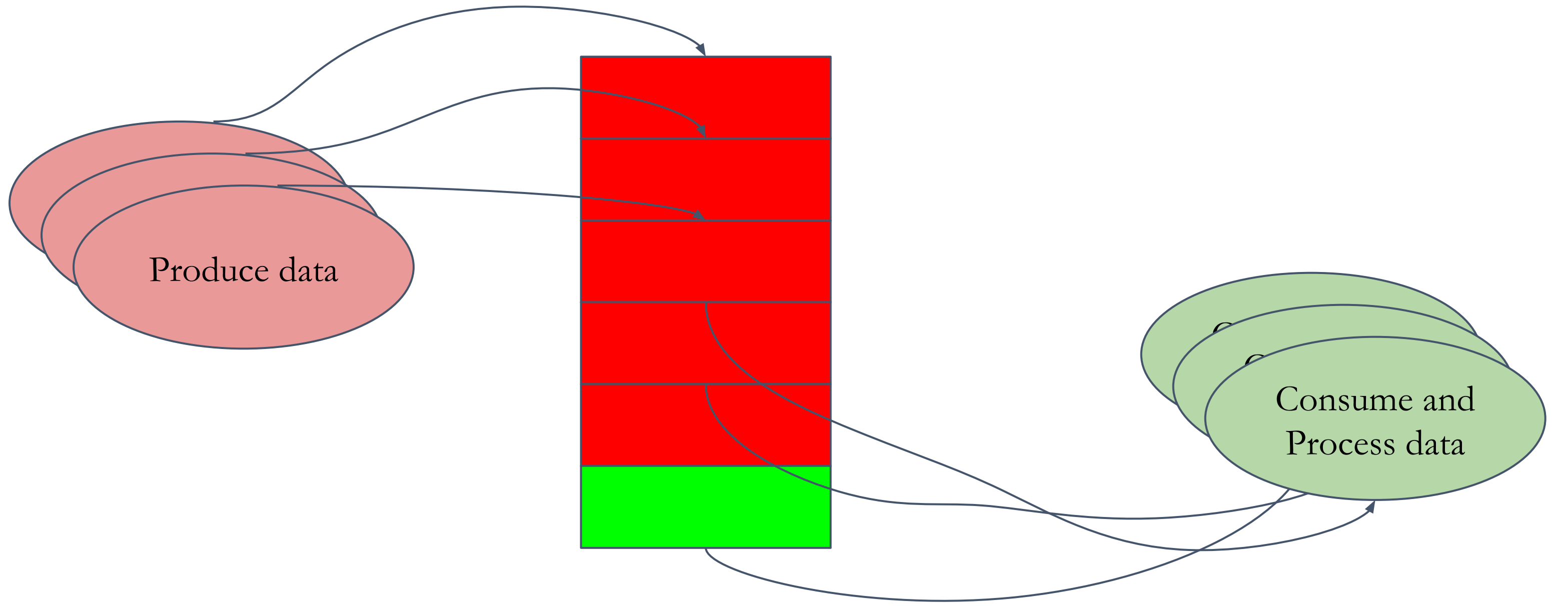
Why TBB?

- Direct programming with threads forces you to do the work to efficiently map logical tasks onto threads
- TBB Runtime library maps tasks onto threads allowing them to steal tasks when idle to maximize load balancing and squeezing performance out of the processor
  - Better portability
  - Easier programming
  - More understandable source code
  - Better performance and scalability
  - Integration with C++ exceptions

Produce-Consume



Produce-Consume



The CMS dependency graph

http://fpantale.web.cern.ch/fpantale/dependency.png

One graph per event

Many events reconstructed at the same time

Each box is a C++ module

tbb and High-Throughput Computing in HEP: a success story

- Perfect load balance up to 128 threads/node
  - then I/O overhead
- Huge reduction of resident memory
- Non uniform time per module

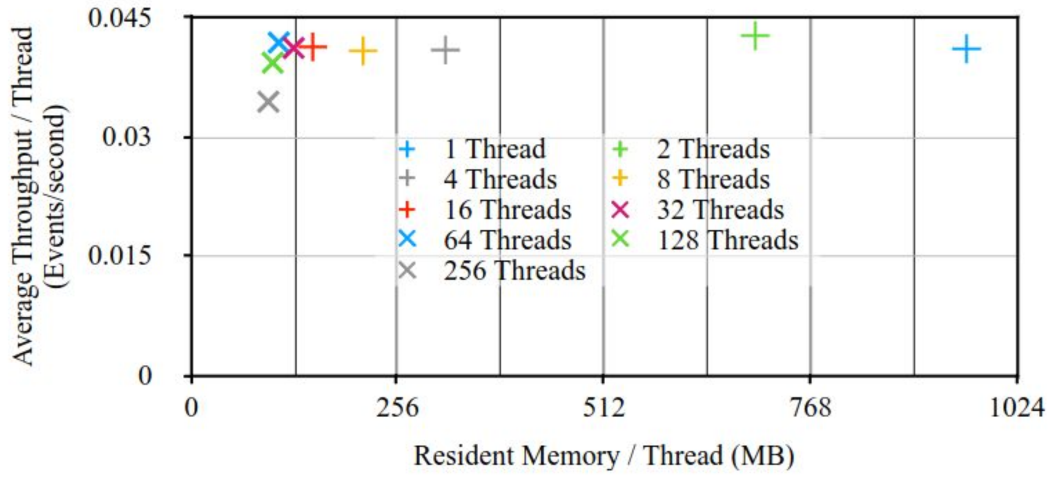


Fig 2. Average event throughput per thread as a function of average resident memory used per thread for simulation process. Each point represents the measurement of a process using that number of threads.

CMSSW Scaling Limits on Many-Core Machines

TBB Threads

Open `hands-on/stdthreads_tbb/hello_world_tbb.cpp`

Compile:

```
g++ hello_world_tbb.cpp -ltbb -o hello_world_tbb -std=c++20
```

Thread pool

A number of threads will be reused throughout your application to avoid the overhead of spawning them (or spawning too many)

```
// analogous to hardware_concurrency, number of hw threads:
int num_threads = oneapi::tbb::info::default_concurrency();

// or if you wish to force a number of threads:
auto t = 10; //running with 10 threads
oneapi::tbb::task_arena arena(t);

// And query an arena for the number of threads used:
auto max = oneapi::tbb::this_task_arena::max_concurrency();
// Limit the number of threads to two for all oneTBB parallel interfaces
oneapi::tbb::global_control global_limit(oneapi::tbb::global_control::max_allowed_parallelism, 2);
```

Parallelizing for loops with tbb

```
for(int i =0; i<N; ++i)  x[i]++;
```

```
oneapi::tbb::parallel_for(oneapi::tbb::blocked_range(0, N, <G>), [&](const
auto& range) {
    for (auto i=range.begin(); i!= range.end(); ++i) {
        x[i]++;
    }
}, <partitioner>);
```

Parallelizing for loops with tbb

```
for(int i =0; i<N; ++i)  x[i]++;
```

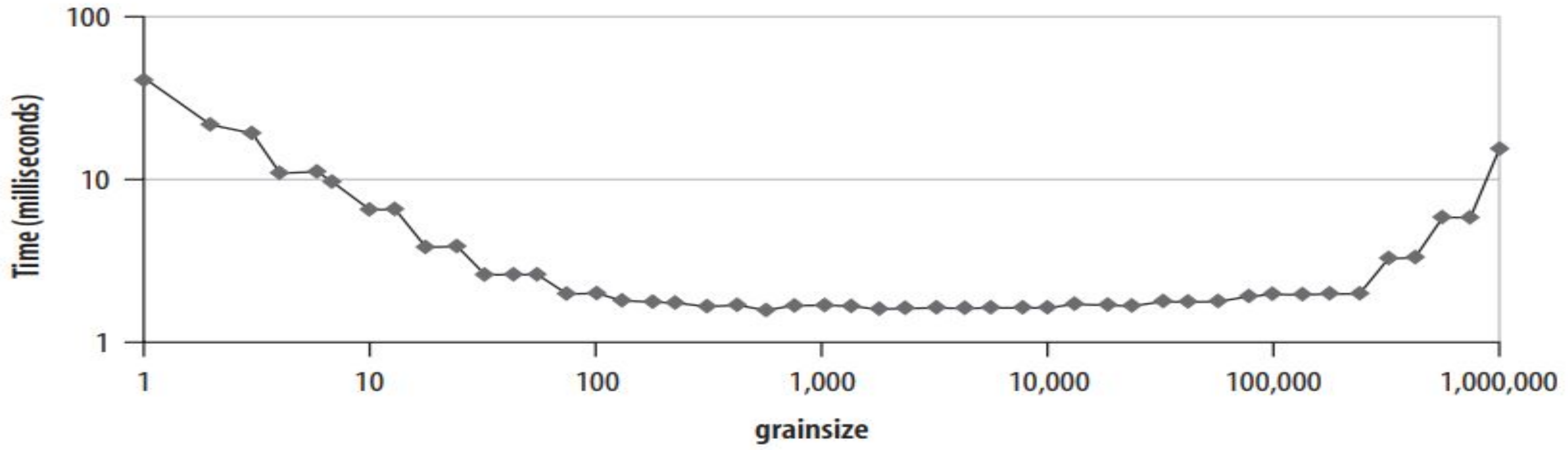
```
oneapi::tbb::parallel_for(oneapi::tbb::blocked_range(x.begin(), x.end(), <G>),
[&](const auto& range) {
    for (auto& element : range) {
        element++;
    }
}, <partitioner>);
```

Scalability

- A loop needs to last for at least 1M clock cycles for parallel_for to become worth it
- If the performance of your application improves by increasing the number of cores, the application is said to *scale strongly*. There is usually a limit to the scaling.
- Usually, adding more cores than the limit does not only result in performance improvements, but performance falls.
  - Overhead in scheduling and synchronizing many small tasks starts dominating
- TBB uses the concept of *Grain Size* to keep data splitting to a reasonable level

Grain Size

- If GrainSize is 1000 and the loop iterates over 2000 elements, the scheduler can distribute the work at most to 2 processors
- With a GrainSize of 1, most of the time is spent in packaging



Automatic Partitioner

- The automatic partitioner is often more than enough to have good performance
- Heuristics that:
  - Limits overhead coming from small grain size
  - Creates opportunities for load balancing given by not choosing a grain size which is too large
- Sometimes controlling the grainSize can lead to performance improvements

Partitioners

- affinity_partitioner can improve performance when:
  - data in a loop fits in cache
  - the ratio between computations and memory accesses is low
- simple_partitioner enables the manual ninja mode
  - You need to specify manually the grain size G
  - The default is 1, in units of loop iterations per chunk
  - Rule of thumb: G iterations should take at least 100k clock cycles

Mutex Flavors

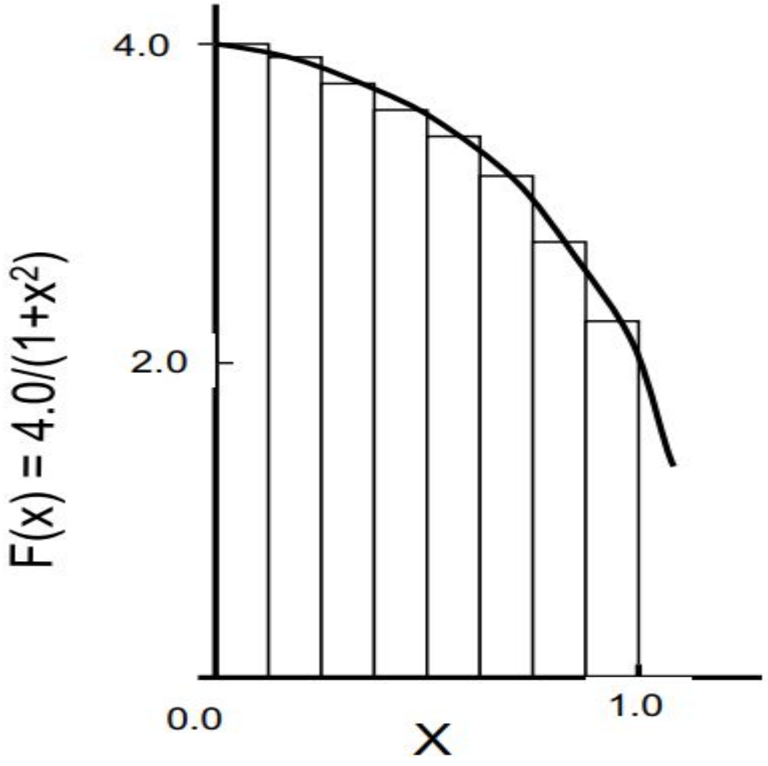
- Scalability
  - Not scalable if the waiting threads consume excessive processor cycles and memory bandwidth, reducing the speed of threads trying to do real work
- Fairness
  - Serves threads in the order they arrived (queuing_mutex)
  - Fair mutexes prevent thread starvation
- Yielding or Blocking
  - Yield: repeatedly poll, if no work allowed temporarily yield the processor
  - Block: yield the processor until the mutex permits progress

Mutex

- Header: `#include <oneapi/tbb/mutex.h>`
- Wrapper around OS calls:
  - Portable across all operating systems supported by TBB
  - Releases the lock if an exception is thrown from the protected region of code

```
oneapi::tbb::mutex  myMutex;
…
{
   oneapi::tbb::mutex::scoped_lock myLock( myMutex );
   //critical section here
   …
}
```

Exercise 2 - Numerical Integration



We know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\, dx = \pi$$

The integral can be approximated as the sum of the rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Numerical integration

- Try to parallelize it
- Measure time vs number of threads, vs number of steps, play with parameters and check precision
- Try privatization
- What happens if one thread runs over more steps than the others?

```
int num_steps = 1<<20;
double pi = 0.;
double step = 1.0/(double) num_steps;
double sum = 0.;

for (int i=0; i< num_steps; i++){
  auto x = (i+0.5)*step;
  sum = sum + 4.0/(1.0+x*x);
}
pi = step * sum;
std::cout << "result: " << std::setprecision (15) << pi << std::endl;
```

Concurrent containers

Concurrent containers allow concurrent thread-safe read-write access by multiple threads

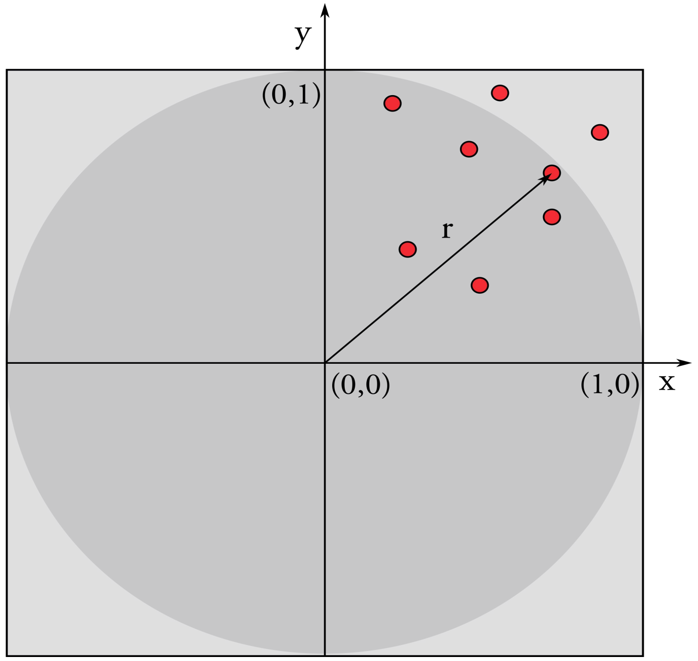
- `oneapi::tbb::concurrent_vector<T>`
- `oneapi::tbb::concurrent_queue<T>`
- `oneapi::tbb::concurrent_hashmap<Key,T,HashCompare>`

For example:

```
#include <oneapi/tbb/concurrent_vector.h>

…

oneapi::tbb::concurrent_vector<int> myVector;

… // later in a parallel section

myVector.push_back(x);
```

Exercise 3 - π with Monte Carlo

- The area of the circle is π
- The area of the square is 4
- Generate N random x and y between -1 and 1:
  - if r < 1: the point is inside the circle and increase N_{in}
  - The ratio between N_{in} and N converges to the ratio between the areas



Exercise 4 - Parallel Histogram

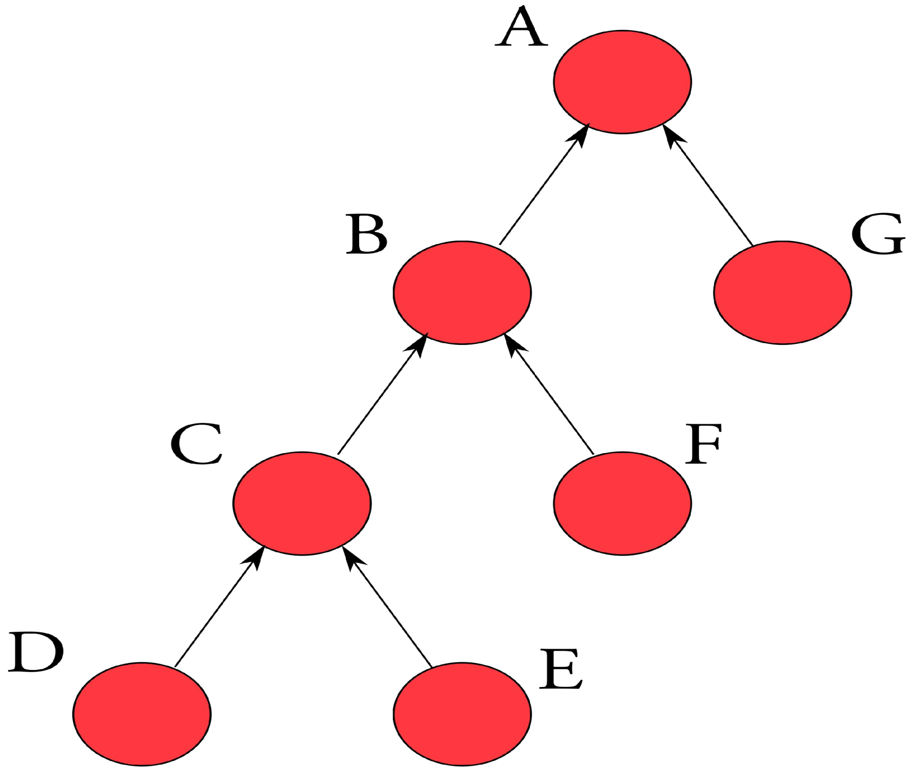
- Generate 2M floats normally distributed with average 0 and sigma 50
- Create a thread-safe histogram class with 100 bins of width 2 (first and last bins contain overflow)
- Use parallel for to push these numbers in the histogram
- Measure strong scaling
- Measure how performance changes, when modifying the number of bins
- Can you think of another pattern for mitigating high contention cases?

Parallel Scheduler

- Efficient load balancing by work stealing
- Reduce context switching
- Preserve data locality
- Keep CPUs busy
- Start/terminating tasks is up to 2 orders of magnitude faster than spawning/joining threads

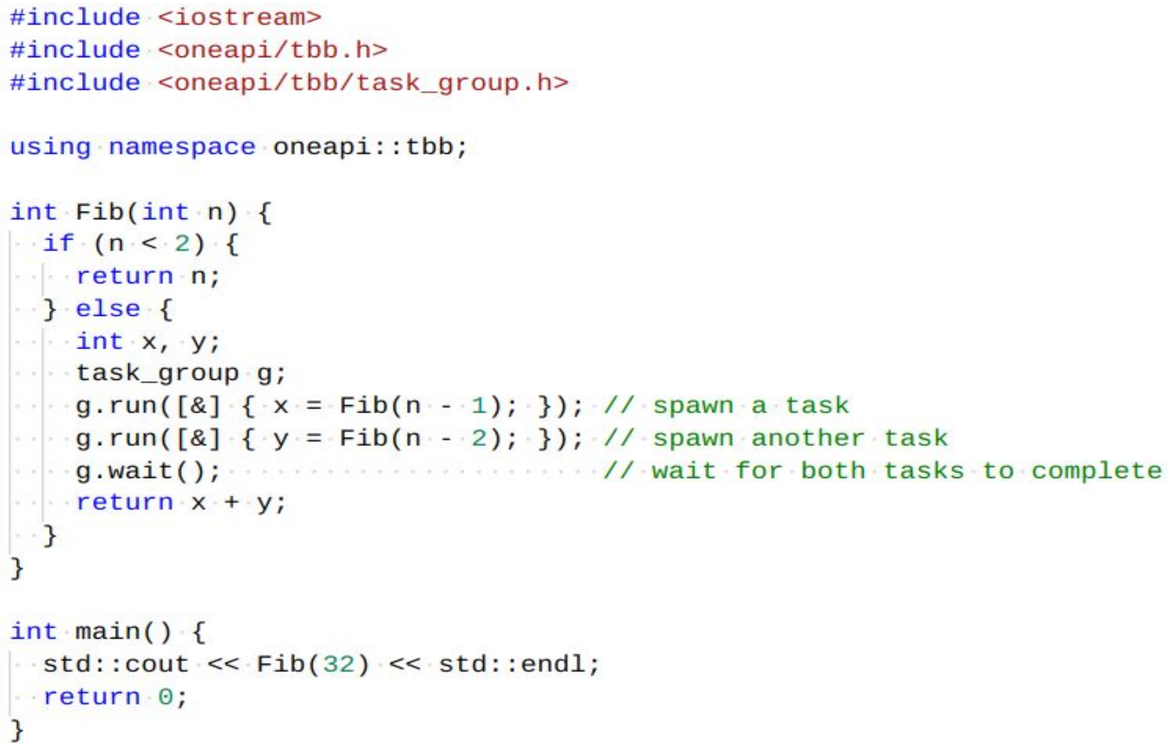
Depth-first execute, breadth-first theft

- Strike when the cache is hot
  - The deepest tasks are the most recently created tasks and, therefore, the hottest in the cache



Task Parallelism with TBB

- A task_group is a container of potentially concurrent and independent tasks
- A task can be created from a lambda or a functor
- A very stupid way to compute the Fibonacci sequence (a lot of duplicate calculations)



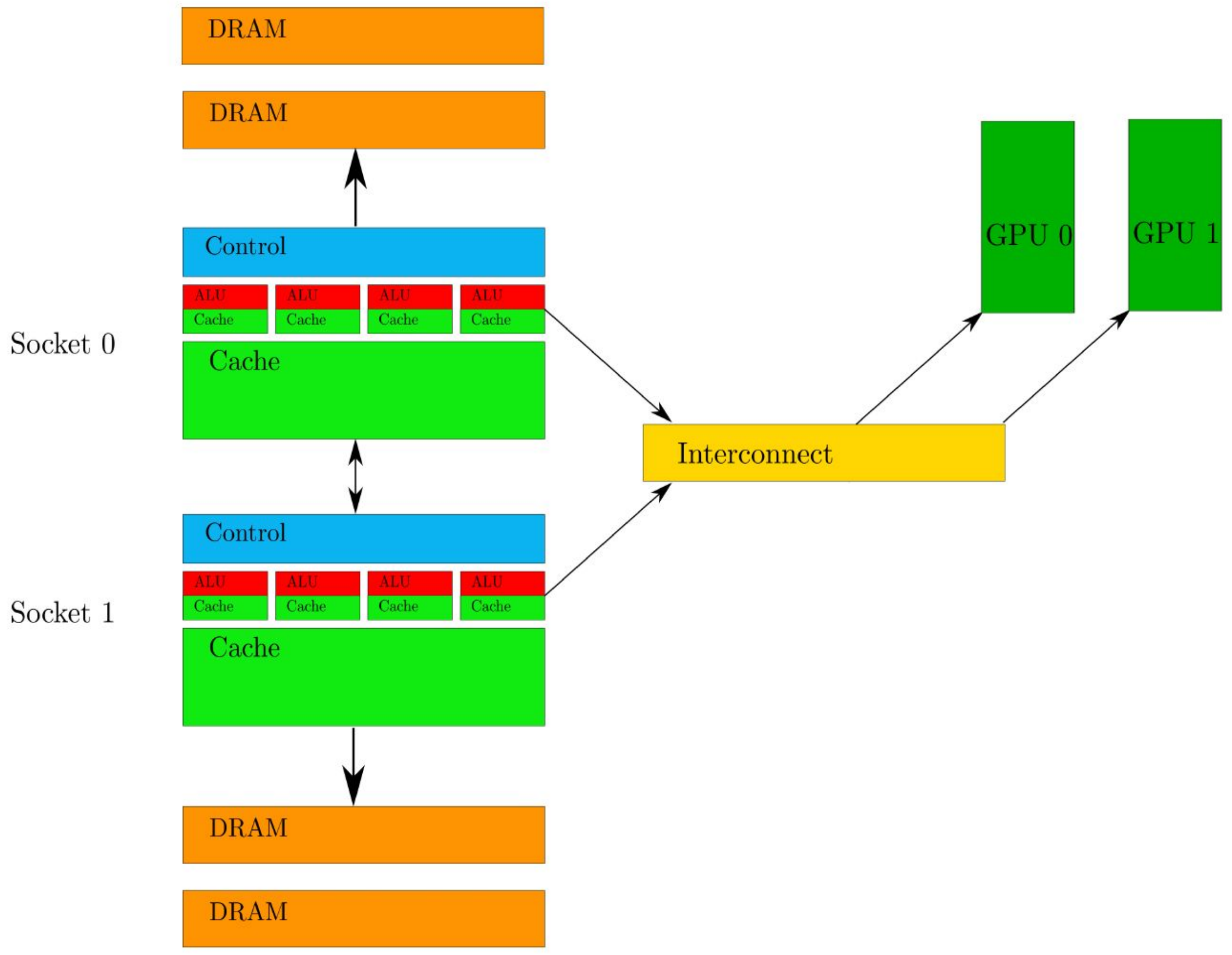
```cpp
#include <iostream>
#include <oneapi/tbb.h>
#include <oneapi/tbb/task_group.h>

using namespace oneapi::tbb;

int Fib(int n) {
  if (n < 2) {
    return n;
  } else {
    int x, y;
    task_group g;
    g.run([&] { x = Fib(n - 1); }); // spawn a task
    g.run([&] { y = Fib(n - 2); }); // spawn another task
    g.wait();                       // wait for both tasks to complete
    return x + y;
  }
}

int main() {
  std::cout << Fib(32) << std::endl;
  return 0;
}
```

Conclusion

TBB owes its success in HEP to being flexible, and cache friendly in non-well-balanced workloads

- deeply nested and recursive parallelism (the nature of tbb)
- complex parallel patterns (non-loops)
- complex flow graph structures
- exception safety or C++ friendly constructions
- no compiler support is required

backup


DRAM
DRAM
Control
ALU
Cache
ALU
Cache
ALU
Cache
ALU
Cache
Socket 0
Cache
Interconnect
GPU 0
GPU 1
Control
ALU
Cache
ALU
Cache
ALU
Cache
ALU
Cache
Socket 1
Cache
DRAM
DRAM

Parallel algorithms in C++

- Starting from C++17, parallel/vectorized versions of standard algorithms started to appear
- You mostly don't have to think about what kind of parallel implementation is hidden under the hood
- You can control the behavior by changing the execution policy

Execution Policies (since C++17)

- `std::execution::seq`: a parallel algorithm's execution may not be parallelized.
- `std::execution::par` : indicate that a parallel algorithm's execution may be executed in an unordered fashion in unspecified threads, and sequenced with respect to one another within each thread.
- `std::execution::par_unseq`: indicate that a parallel algorithm's execution may be executed in an unordered fashion in unspecified threads, and unsequenced with respect to one another within each thread.

Parallel Algorithms

- std::accumulate
- std::adjacent_difference
- std::inner_product
- std::partial_sum
- std::adjacent_find
- std::count
- std::count_if
- std::equal
- std::search_n
- std::transform
- std::replace
- std::replace_if
- std::max_element
- std::merge
- std::min_element
- std::nth_element
- std::partial_sort

Examples of what is possible

- #include <execution>
- ...
- std::vector<int> v;
- // fill the vector
- ...
- // sort it in parallel
- std::sort(std::execution::par, v.begin(), v.end());

- // apply a function foo to each element
- std::for_each(std::execution::par_unseq, v.begin(), v.end(), foo);

Unordered algorithms

- std::vector<int> v;
- // fill the vector
- ...
- // reduce it in parallel
- // reduction_binary_op has to be commutative and associative   // operation
- auto y = std::reduce(std::par_unseq, v.begin(), v.end(), [initialvalue], [reduction_binary_op]);

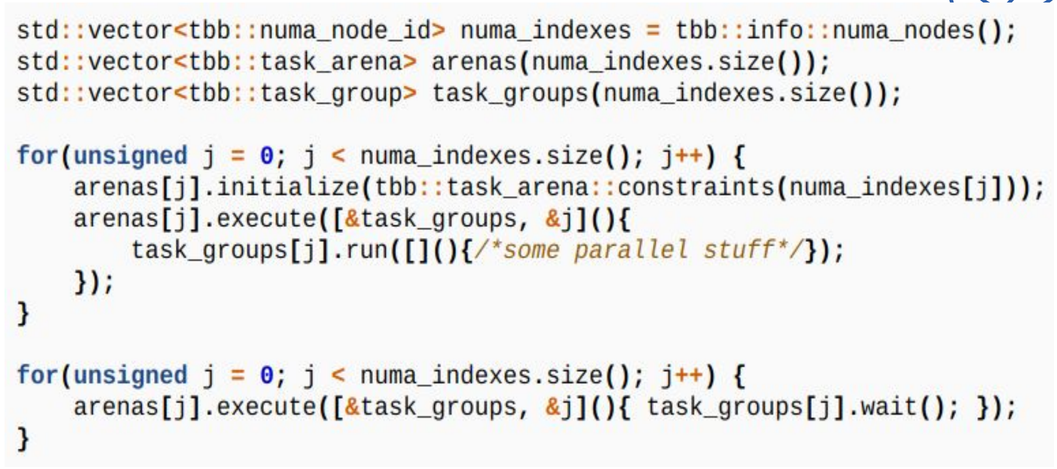
std::transform_reduce, aka the parallel C++ swiss knife

- Takes a container of elements of type T
- Produces an object of type R
- Requires a transformation function
  - R foo(const T&)
- Requires a requires a binary operation:
  - R bar(const R&,const R&)
- Requires an initial value for the reduction

example

- The norm of a vector is:
- sqrt(x[0]*x[0] + x[1]*x[1] + … + x[N-1]*x[N-1])
- std::vector<double> v; // fill it
- double result2 = std::transform_reduce(std::par_unseq,
  - v.begin(), v.end(),
  - // transform
  - [](double elt) { return elt*elt; },
  - // initial value
  - 0.0,
  - // reduction
  - [](double x, double y) {return x+y;}
  -);

task_arena::constraints



```
std::vector<tbb::numa_node_id> numa_indexes = tbb::info::numa_nodes();
std::vector<tbb::task_arena> arenas(numa_indexes.size());
std::vector<tbb::task_group> task_groups(numa_indexes.size());

for(unsigned j = 0; j < numa_indexes.size(); j++) {
    arenas[j].initialize(tbb::task_arena::constraints(numa_indexes[j]));
    arenas[j].execute([&task_groups, &j](){
        task_groups[j].run([](){/*some parallel stuff*/});
    });
}

for(unsigned j = 0; j < numa_indexes.size(); j++) {
    arenas[j].execute([&task_groups, &j](){ task_groups[j].wait(); });
}
```

- It allows to specify the following restrictions:
- Preferred NUMA node
- Preferred core type
- The maximum number of logical threads scheduled per single core simultaneously

```
int concurrency_one_thread = oneapi::tbb::info::default_concurrency(
    oneapi::tbb::task_arena::constraints{}.set_max_threads_per_core(1)
);
oneapi::tbb::task_arena arena( concurrency_one_thread );
arena.execute([] { parallel_foo();});
```

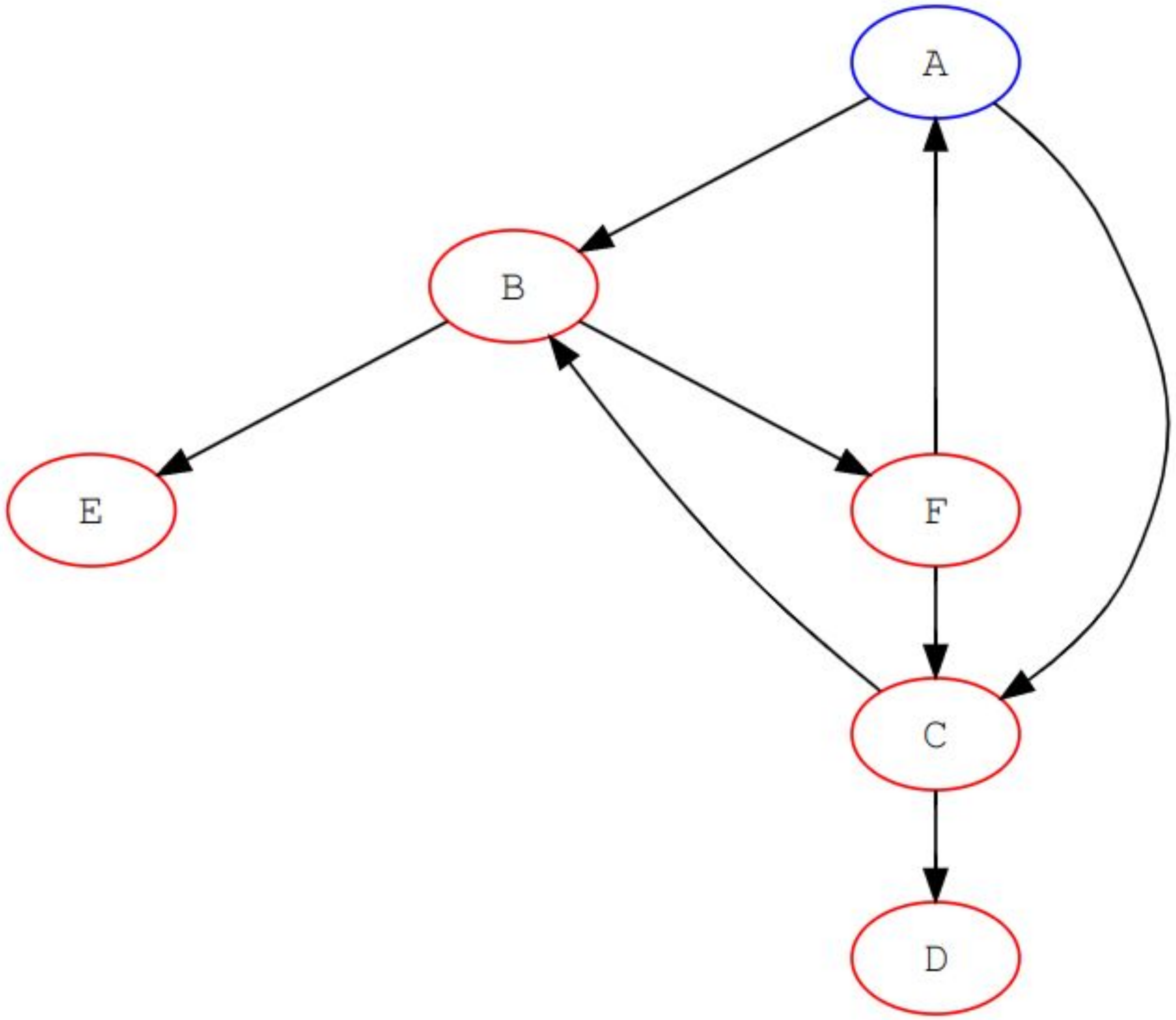
- The level of task_arena concurrency

Your turn: BFS

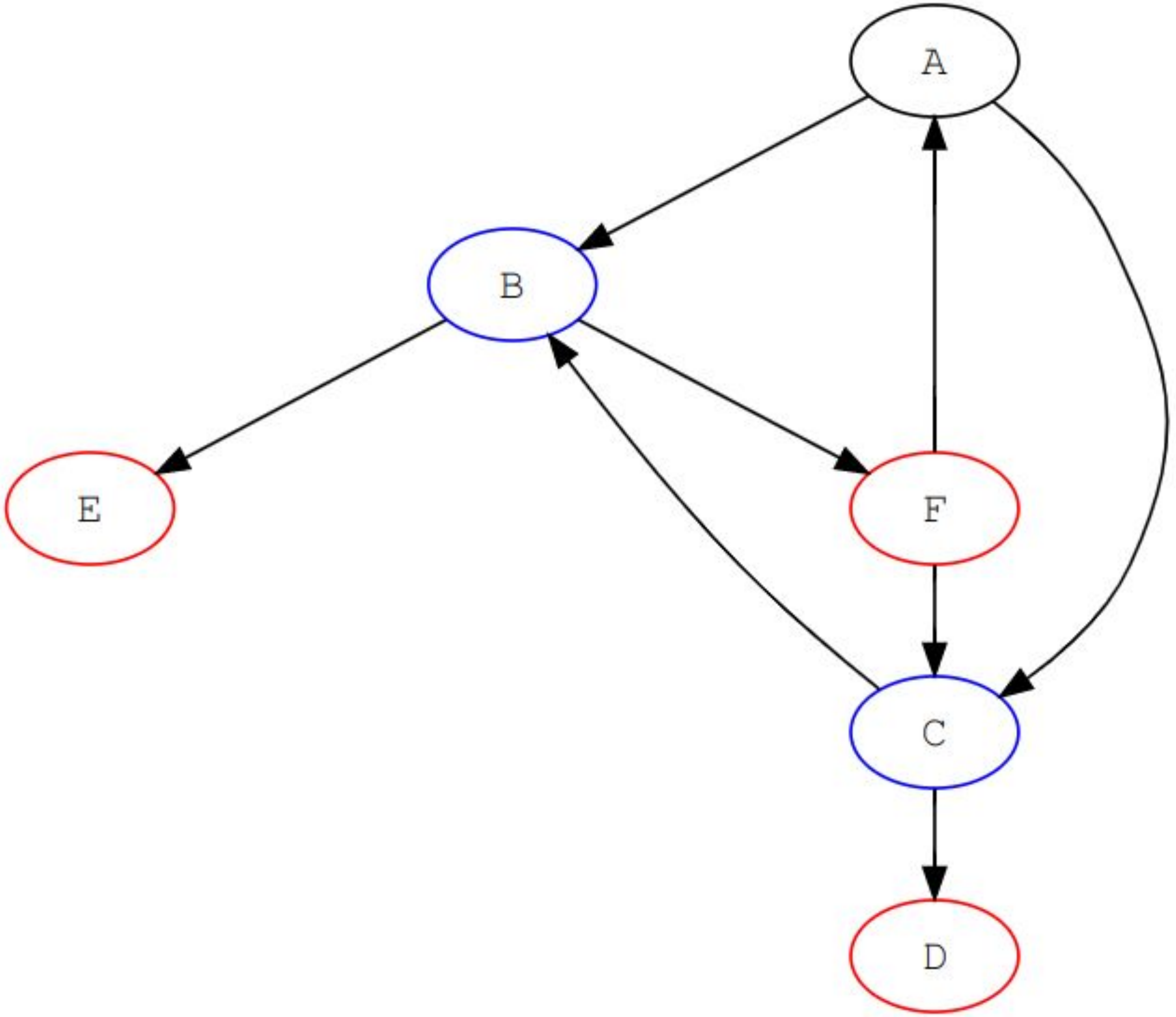
- Calculate the shortest distance to travel from a vertex to all the other vertices in a graph
- You can find sequential iterative and recursive implementations in BFS.cpp
- Implement it with parallel_for and tasks

$$\begin{tikzcd}
 & & A \arrow[dl] \arrow[ddd, bend left] \\
 & B \arrow[dl] \arrow[dr] & \\
E & & F \arrow[uu] \arrow[d] \\
 & & C \arrow[uul] \arrow[d] \\
 & & D
\end{tikzcd}$$

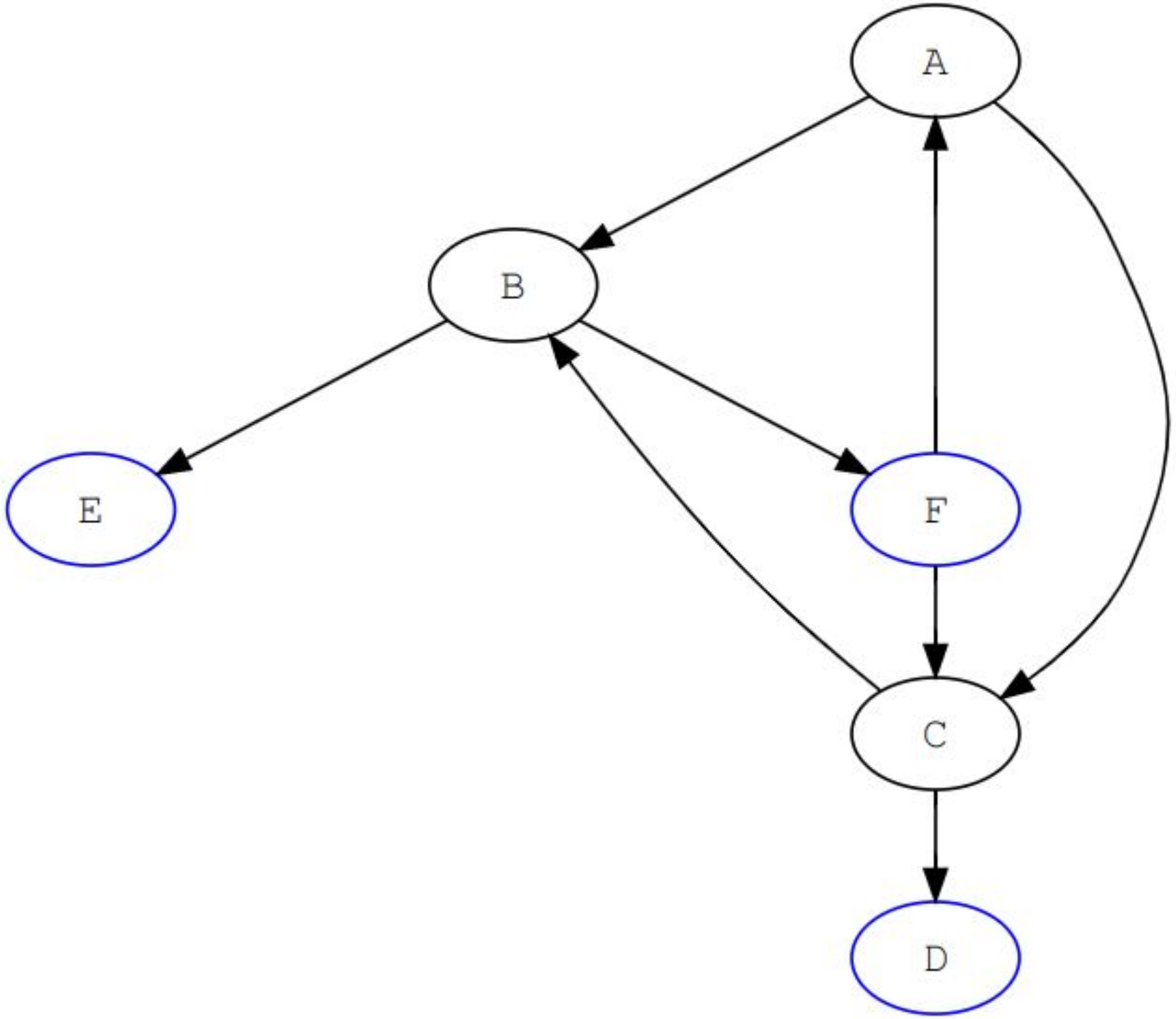
d=0



d=1



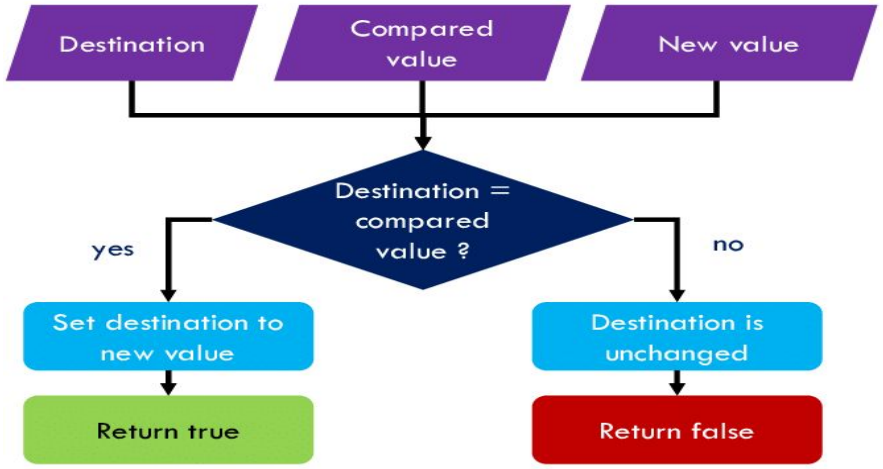
d=2

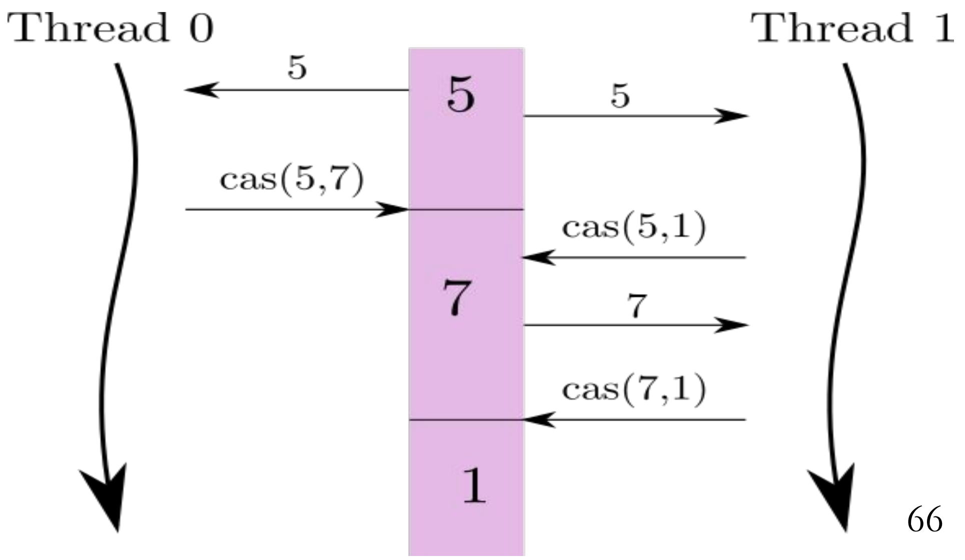


Compare-and-swap (CAS)

- Allows to create more complex lockless data structures

```
bool success = x.compare_exchange_weak(expected, new_value_if_success);
```


Destination
Compared value
New value
Destination = compared value ?
yes
no
Set destination to new value
Destination is unchanged
Return true
Return false


Thread 0
Thread 1
5
5
5
cas(5,7)
7
cas(5,1)
7
cas(7,1)
1