Multi-threading

Oct. 30 & 31st, 2019

Olivier Coudert

CONFIDENTIAL INFORMATION

The information contained in this presentation is the confidential and proprietary information of Synopsys. You are not permitted to disseminate or use any of the information provided to you in this presentation outside of Synopsys without prior written authorization.

IMPORTANT NOTICE

In the event information in this presentation reflects Synopsys' future plans, such plans are as of the date of this presentation and are subject to change. Synopsys is not obligated to update this presentation or develop the products with the features and functionality discussed in this presentation. Additionally, Synopsys' services and products may only be offered and purchased pursuant to an authorized quote and purchase order or a mutually agreed upon written contract with Synopsys.

Summary

- Parallelism
- pthread vs. tbb
- Basic concepts of tbb
- Thread synchronization
- Multi-threading and memory allocation
- Thread-local storage
- Conclusion

Parallelism

Parallelism

- Use multiple processing agents
  - Goal is to decrease wall time (total CPU time may increase)
- Flavors (Single/Multiple/Instruction/Data)
  - SIMD
    - Several processing units perform same operation on multiple data
  - MISD
    - Several processing units perform different operations on the same data
  - MIMD
    - Several processing units perform different operations on multiple data

SIMD is often the one of interest: same code to apply on a lot of data

Parallelism: distributed system

- Multiple nodes on a farm process data
  - Inter-node communication via network (slow)
  - Data exchange is the bottleneck
  - But can have 100-1000’s of nodes on a farm

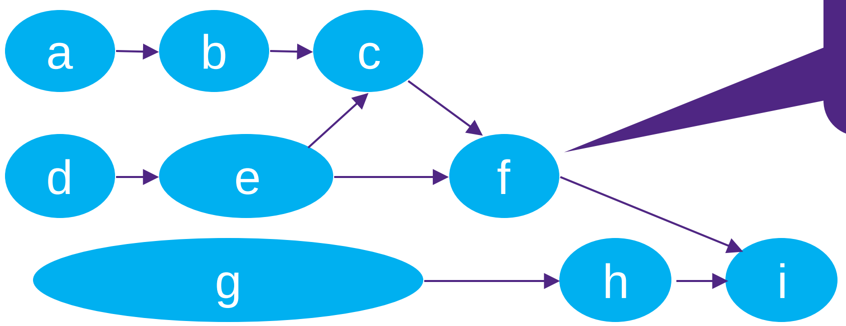
- Paradigm
  - Because inter-node communication is expensive, dataset is usually partitioned so that each job is completely independent
    - E.g.: map-reduce
  - At join point, a pure sequential post-processing of each node’s output may be required
  - Good scalability can be achieve for sufficiently large datasets

Parallelism: MT’ed system

- Multiple cores process data
  - Can share large dataset in RAM
  - Inter-thread communication
    - L1/L2 cache consistency
    - Via RAM
    - Via thread ID
    - Via interruption
  - RAM much faster than network but still much slower that cores
  - Thread synchronization and RAM sharing can become the bottleneck

Basics

- Must identify the task dependency graph



Convergence point is a “join”: need all the incoming tasks to complete before we can move forward

- A join on multiple tasks of uneven workloads is bad
  - Wall time is the longest path in the dependency graph
  - Non-critical threads are idle until the join happens
- For complex dependency graphs
  - Algorithm may need to be redesigned so that dependencies can be removed (possibly by adding redundancies)

Divide-and-conquer

- One large data set processed with multiple threads
  - E.g., parallel sort

- Need to partition the input dataset
  - Does it require inter-partition communication?
  - Can you do an upfront partition?...
  - …or is the dataset generated on the fly?
  - Can you partition so as to produce even workloads?
  - Do you have a random access iterator on your dataset?

- Need to limit communication between threads

pthread vs. tbb

pthread

- Explicit thread management (pthread, boost, std)
- Basic unit of computation is a thread
- Developer has to create her threads and manage their synchronization and termination

thread::thread(callable& fun)	build thread and calls fun::operator()()
thread::get_id()	unique ID associated with the thread
thread::join()	make parent thread waits until child thread completes
thread::detach()	detach child thread from parent thread
thread::joinable()	return true iff neither join() and detach() has been called, i.e., it is a point of synchronization

tbb (Threading Building Blocks)

- Set of templates to hide the thread management
- Basic unit of computation is a “task”, not a thread
  - Usually there is a 1-2-1 mapping between threads and cores of the machine
  - The assignment of tasks to threads is done by tbb
  - A task might be executed by a single or multiple threads
- Provide simple templates, for example:

Focus on synchronization and parallelism strategies, not thread management

tbb::parallel_foreach	applies functor on each element of collection in parallel
tbb::parallel_for	applies functor on range of elements in parallel
tbb::parallel_do	as above, and can dynamically add new elements to process
tbb::parallel_invoke	calls multiple functors in parallel, join all

Basic concepts of tbb

Basics of tbb

- tbb does not expose threads directly
  - Instead its basic computational unit is a tbb::task
  - Tasks are created by the user or by tbb's templates
  - Tasks are nodes in a dependency graph
  - Tasks are then assigned to physical threads to run tbb::task::execute() by a scheduler
  - The assignment of tasks to threads is random

- tbb vs.pthread
  - Plus
    - Powerful templates to automatically create and manage tasks
    - Built-in scheduler
  - Cons
    - The scheduler may have a larger overhead than a hand-made pthread-based scheduler

Parallelization of loops

- Loop

```
for (auto it = begin; it != end; ++it) fun(*it);
```

- Instead apply a functor on the [begin, end[interval

```
std::for_each(begin, end, fun);
```

- Same, but in a multi-threaded manner

```
tbb::parallel_for_each(begin, end, fun);
```

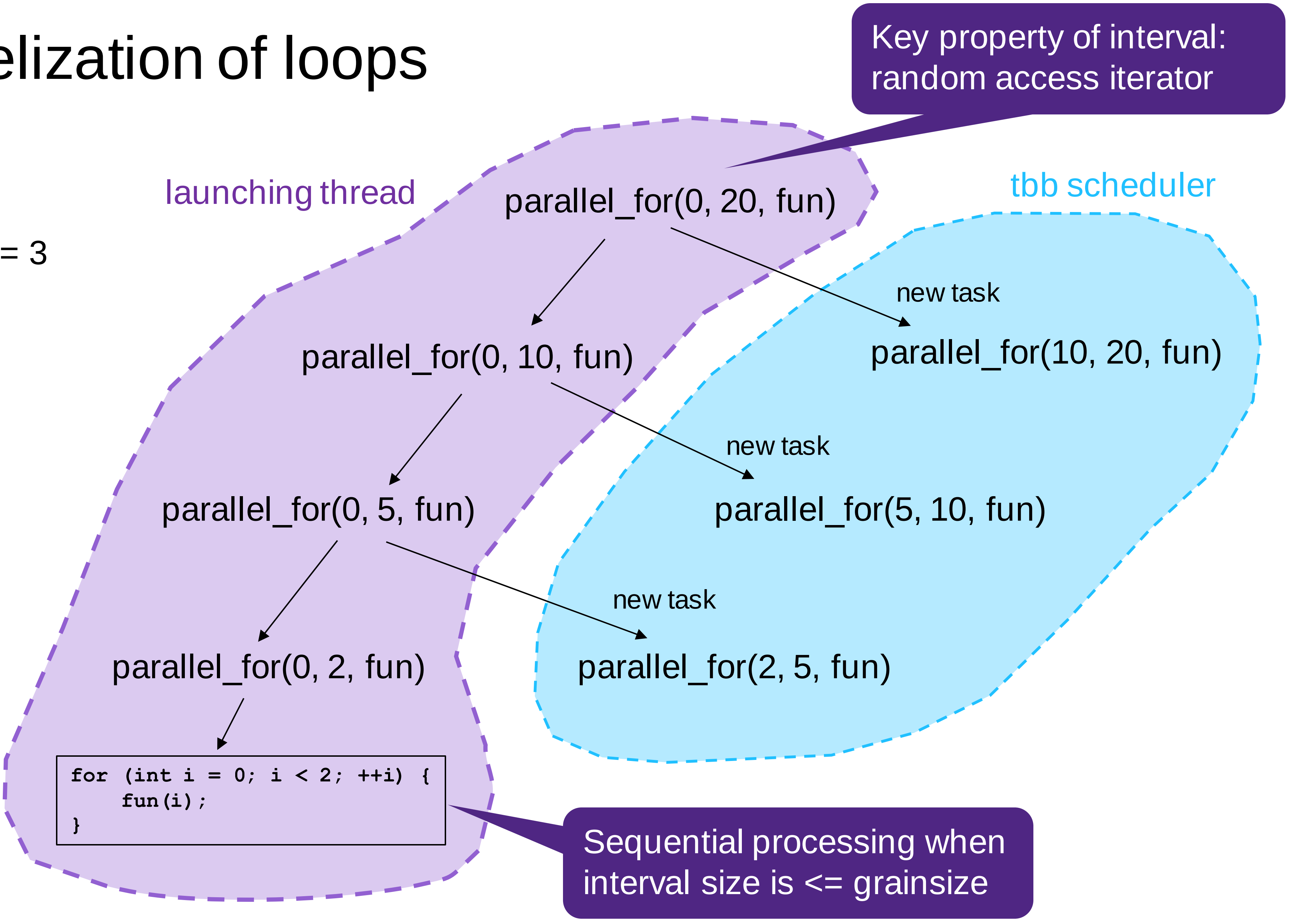
Small enough = size of the interval is less than grainsize

- Basic flow
  - If the interval is small enough, apply the functor sequentially
  - Otherwise divide the interval in two and have two threads working on each sub-intervals

Thus we MUST have a random-access iterator

Parallelization of loops

Grainsize = 3

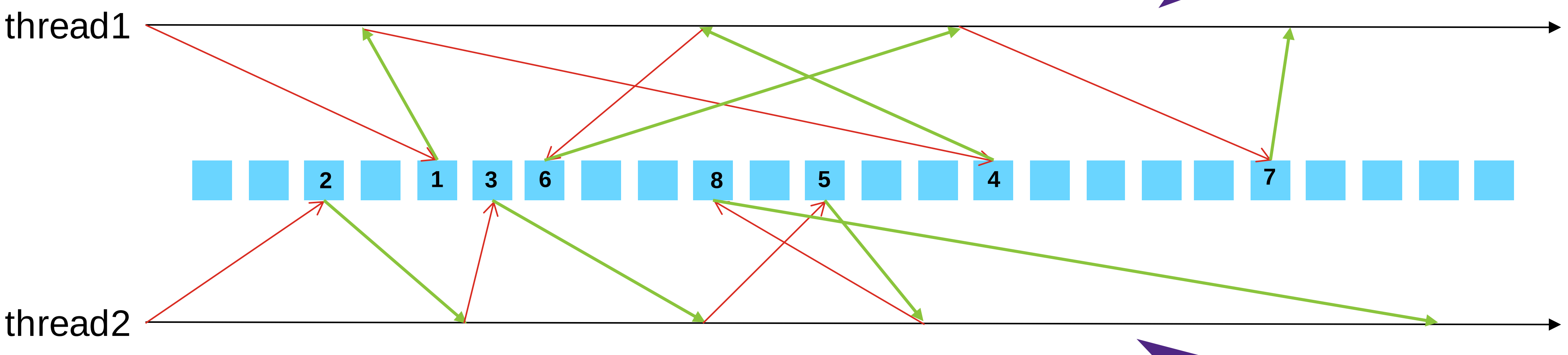


parallel_for_each(begin, end, fun)

- Grainsize = 1
- The functor is shared between threads

The functor CANNOT have mutable data members unless we sync the threads

Each thread jump randomly from one data to the next



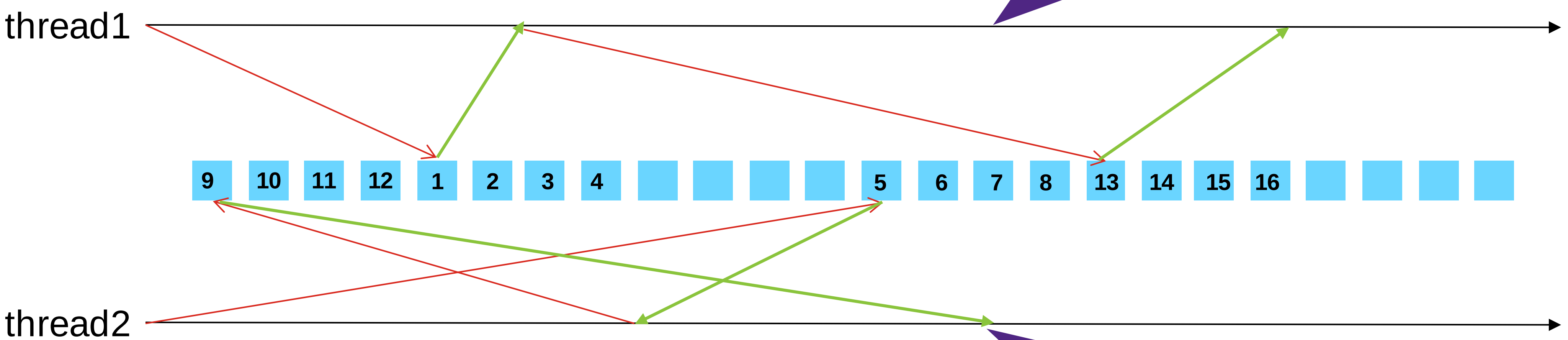
Data is processed in random order

parallel_for(tbb::blocked_range<>(begin, end, grainsize), fun)

- Grainsize = k
- The functor is copied and applied on each range

The functor's data members is local to the thread

Each thread jump randomly from one range to the next



Data is processed in quasi random order

parallel_for vs. parallel_for_each

- Use parallel_for when:
  - Processing one single data is very fast
    - Because the overhead of tbb for grainsize=1 will be impacting performance
  - We want to cluster data together for better load balancing
  - Data is contiguous in memory
    - Processing a range produces less cache miss
  - We want to have data members local to the thread

- Use parallel_for_each when:
  - We want quick development
  - We want to create as many tasks as there are threads

```
const size_t numThreads = getNumThreads();
vector<T> data(numThreads);

class Fun {
  void operator()(size_t i) const { data[i].process(); }
};

tbb::parallel_for_each(0, numThreads, Fun());
```

parallel_do(begin, end, fun)

- The interval end of [begin, end[might not be known (e.g., list)
- New items might be dynamically added to the interval
- The functor has signature:
  - Fun::operator()(T& o, parallel_do_feeder<T>& feeder);
  - parallel_do_feeder<T>::add(const T&) allows thread to add items

```
void
FunVisitTFI::operator(Node& n, parallel_do_feeder<Node>& feeder) {
    vector<Node> queue;
    queue.push_back(n);

    while (!queue.empty()) {
        Node n = queue.back();
        queue.pop_back();
        uint_8 old = _bytes[n.idx()].fetch_and_store(0x1);
        if (old == 0x1) {
            // Another thread took ownership of the node
            continue;
        }
        queue.insert(queue.end(), n.beginFanin(), n.endFanin());

        while (queue.size() > TOO_LARGE) {
            Node n = queue.back();
            queue.pop_back();
            feeder.add(n);
        }
    }
}

tbb::parallel_for_each(begin, end, Fun());
```

The workload of this thread has become too large: delegate the processing of some items to other threads

Achieve even workload distribution

Thread synchronization

Synchronization

- Needed when
  - Threads compete for the same resource
  - Threads must wait for another thread to complete before then can move on (e.g., join)
- Lock
  - Used to take ownership of a resource (e.g., to read/write)
  - This can be exclusive
    - no other thread can access the resource
  - This can be non-exclusive but with restricted actions
    - E.g., multiple threads can access the same resource read-only. If a thread comes to write, he will have to wait for all reading threads to complete. A thread can promote a lock-to-read to a lock-to-write access.
  - This can be non-blocking
    - mutex::try_lock()

Synchronization is anything but free

- A lock takes memory
- Acquiring a lock has some overhead
- Failure to acquire a blocking lock results in
  - Threads wasting CPU (e.g, spin_lock)
  - Threads doing context switching (yield), which is expensive

- Scalability of multi-treads applications is hard
  - Not only it can be difficult to design the algorithm for multiple threads…
  - …but we must account for the cost of thread synchronization

Example

```
class Fun {
  virtual ~Fun();
  virtual void operator()();
};
class A : public Fun {
  virtual ~A();
  virtual void operator()();
};
class B : public Fun {
  virtual ~B();
  virtual void operator()();
};

vector<thread<Fun>* > tasks;
tasks.push_back(new std::thread(Fun()));
tasks.push_back(new std::thread(A()));
tasks.push_back(new std::thread(B()));

// Here...

for (int i = 0; i < 3; ++i) {
   tasks[i].join();
}

// There...
```

operator()() is the code executed by the thread

Once the functor is created, operator()() is immediately called

At this point we have 3 threads running in parallel

This forces the main thread to wait for the 3 children threads to complete

At this point the 3 threads have completed, main thread resumes

Example

```
class Fun {
   Fun(std::mutex& mutex) : _mutex(mutex) {}
   void operator()();
   std::mutex& _mutex;
}

void Fun::operator()() {
   std::map<Key, Value>& table = GetTable();
   {
      _mutex.lock();
      doSomethingWithThatTable(table);
      _mutex.unlock();
   }
};

std::mutex m;
vector<thread<Fun>* > tasks;
for (int i = 0; i < 100; ++i) {
   tasks.push_back(new std::thread(Fun(m)));
}
for (int i = 0; i < 100; ++i) {
   tasks[i].join();
}
```

Share the mutex so that threads can share common resources

Critical region

Mutex flavors

- mutex
  - medium slow, fat, blocking
- spin_mutex
  - very fast, lean, blocking and non-blocking
  - To use for light contention access
- queuing_mutex
  - medium fast, lean, non-blocking, fair
  - To use when deadlock happens on blocking

Can produce deadlock

Can produce uneven load distribution

tbb mutex flavors

Use blocking for long wait

Good for very short wait

Typical usage is r/w in non-contentious container

Mutex	Scalable	Fair	Recursive	Long Wait	Size
mutex	OS dependent	OS dependent	no	blocks	≥ 3 words
recursive_mutex	OS dependent	OS dependent	√	blocks	≥ 3 words
spin_mutex	no	no	no	yields	1 byte
speculative_spin_mutex	HW dependent	no	no	yields	2 cache lines
queuing_mutex	√	√	no	yields	1 word
spin_rw_mutex	no	no	no	yields	1 word
speculative_spin_rw_mutex	HW dependent	no	no	yields	3 cache lines
queuing_rw_mutex	√	√	no	yields	1 word
null_mutex	moot	√	√	never	empty
null_rw_mutex	moot	√	√	never	empty

Atomic operations (1/2)

An operation acting on shared memory is atomic if it completes in a single step relative to other threads

- This means that when multiple threads read/write a data in an atomic way, the data is always consistent (as in “not corrupted”)
- But the read value MAY depend on race conditions
- Modern processors allows native (i.e., lock-free) atomic read/write on scalar types with 1, 2, 4, and 8 bytes.

X’s value at any time depends on race between threads

```
class Fun {
  Fun(atomic<int>& x) : _x(x) {}
  void operator()() { sleep(10); ++x; cout << x; }
  atomic<int>& _x;
};

atomic<int> x;
x = 0;
vector<thread<Fun>* > tasks;
for (int i = 0; i < 100; ++i) {
   tasks.push_back(new std::thread(Fun(x)));
}
for (int i = 0; i < 100; ++i) { tasks[i].join(); }
cout << x << endl;
```

Final value of x is known (i.e., 100), but output sequence is non-deterministic

Atomic operations (2/2)

- read-modify-write
  - Atomic operations that consist of reading a value and writing a new value at the same memory location in a single step
- They enable lock-free thread synchronization
  - Extremely efficient –assuming we don't re-implement a lock…
  - Can lead to quite complicated synchronization mechanism
  - Correctness can be hard to prove
- Most common:

x.fetch_and_store(y)	do x = y, and return the old value of x
x.fetch_and_add(y)	do x += y, and return the old value of x
x.compare_and_swap(y, z)	if x equals z, do x = y. Always return old value of x.

Example

```
bool
TraceTFI::FunTrace::needToProcessWire(UnfoldedWire fw)
{
    // Atomic reference
    tbb::atomic<BYTE>& aref = awb().atomicGetRef(fw.getUnfoldedIndex());

    // Atomic read
    BYTE oldStatus = aref;

    if (oldStatus == PROCESSED) {
        // Already processed.
        ++_stats[e_numHits];
        return false;
    }

    const BYTE newStatus = PROCESSED;

    // Atomic swap.
    oldStatus = aref.fetch_and_store(newStatus);

    if (oldStatus == PROCESSED) {
        // Another thread just marked that wire since
        // the last test on PROCESSED.
        ++_stats[e_numOverlaps];
        return false;
    }

    return true;
}
```

Between these two points another thread may have changed the value of aref.

Current thread checks status of object

Current thread computes the status it wants to write

Atomic swap: current thread writes its data and return the old data

Overlapping threads

Current thread took ownership of its data

How costly is a mutex?

- Acquiring a mutex is attempted with an atomic compare_and_swap

```
bool
tryToAcquireMutex(tbb::mutex& mutex) {
    // Atomic operation: if _byte == 0x0, assign it to 0x1.
    uint8_t oldValue = mutex._byte.compare_and_swap(0x0, 0x1);
    // If oldValue is 0x0, this thread has just written 0x1,
    // and therefore has acquired the mutex.
    return (oldValue == 0x0);
}
```

Successfully acquiring a mutex on first try is very cheap

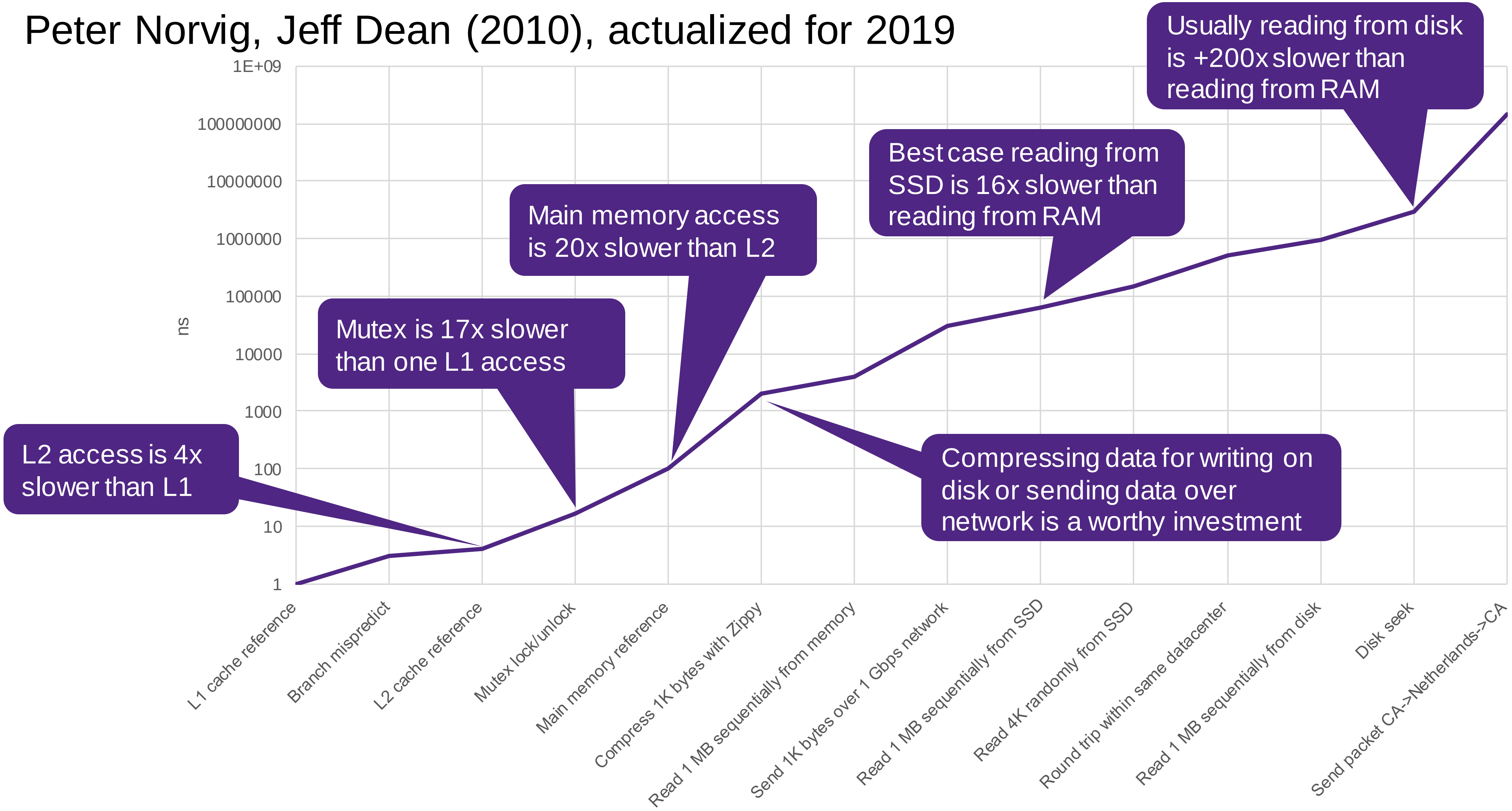
- If we fail to acquire the mutex
  - The thread must wait until the mutex is unlocked
  - The task is sent back to the system scheduling queue…
    - Costly system call
  - …and the core is assigned to another task
    - Costly context switch

Failing to acquire a mutex is expensive

Multi-threading and memory allocation

“Latency Numbers Every Programmer Should Know”

Peter Norvig, Jeff Dean (2010), actualized for 2019



Memory allocators

- malloc
  - MT'ed allocation requires a lock
- Hand-made block allocator
  - Manage dedicated pages and free lists
  - May beat malloc if we don't need a generic allocation
  - Prone to fragmentation if dealing with uneven size objects
- tcmalloc
  - Designed upfront for MT'ed applications
  - Block allocator with binned sizes
  - Can dynamically rebalance free lists among threads

Hand-made allocators don't bring a lot of value and can be counter-productive

Avoid fragmentation

tcmalloc should be the default all-purpose allocator

MT'ing and memory allocation

- Allocation on the stack
  - No lock needed
  - But stack has a fixed static size when using threads
  - Recursive algorithms can quickly exhaust the stack
- Allocation on the heap (new/delete)
  - Must synchronize between multiple threads doing new/delete
  - Lock is used for exclusive access to complex resources (page manager, free lists, etc)
  - Too many concurrent new/delete will hurt scalability

MT'ed algorithm must be non-recursive

MT'ed algorithm must avoid dynamic heap allocation

Reduce the amount of new/delete

- NEVER use lists to accumulate objects
  - Use vector instead
    - #new/delete in is O(log_2(n)) for a vector, as opposed to O(n) for a list
    - Amortized cost is always better
- Pre-allocate memory dedicated for each thread when possible
  - E.g., oversize the containers to reduce dynamic allocation
- Accumulate results in memory local to a thread, wait for the join to communicate it to the shared memory
  - This may create redundant results, but it’s always more scalable to do so than locking during processing
- Use specialized containers
  - E.g., open addressing hash table: it manages collisions w/o linked list, i.e., no dynamic memory allocation is required.

Memory access and granularity do matter

- Simple application: scan a large collection and collect statistics
  - E.g., how many in this collection satisfy predicate P?

- Sequential implementation

```
size_t _Num = 0;

for (size_t idx = begin(); idx < end(); ++idx) {
  _Num += isPredicateTrue(idx);
}
```

Memory access and granularity do matter

- Implementation 1

```
tbb::atomic<size_t> _Num;

class Fun {
  void operator()(size_t idx) const { _Num += isPredicateTrue(idx); }
};

_Num = 0;
tbb::parallel_for_each(begin(), end(), Fun());
```

Wall: 640s
User: 988s
Sys: 96s

- Implementation 2

```
tbb::atomic<size_t> _Num;

class Fun {
  Fun() : _num(0) {}
  Fun(const Fun&) : _num(0) {}
  typedef tbb::blocked_range<size_t> Range;
  void operator()(const Range& r) const {
    for (Range::const_iterator it = r.begin(); it != r.end(); ++it) {
      _num += isPredicateTrue(*it);
    }
  ~Fun() { _Num += _num; }
  size_t _num;
};

_Num = 0;
size_t grainsize = std::distance(begin(), end()) / (numThreads() + 20);
tbb::parallel_for(Fun::Range(begin(), end(), grainsize), fun());
```

Wall: 68s
User: 1153s
Sys: 8s

Accumulate result local to the functor

Aggregate functor-local result to global result

Don't aggregate too often

- #2's user time higher due to the overhead to generate the intervals
- #1's sys time higher due to too much contention on _Num
- #2's wall time lower because better thread utilization

Memory layout does matter

- Use local memory associated to each thread
  - Caching local to the thread is good
  - But avoid allocating memory!
  - Watch the functor's destructor overhead!
- Keep shared memory as contiguous as possible
  - Cache miss is expensive
    - L2 is 4x slower than L1
    - RAM access is 25x slower than L2

- Alloc on the stack is limited
- Alloc on the heap locks

Need to reuse threads

- E.g., same simulation algorithm (874M leaf cells, 640 cycles):
- # step SMZ_SIMUL : Done in elapsed:225 s user:1227 s system:4 s vm:103472 m
- # step SMZ_SIMUL : Done in elapsed:173 s user: 770 s system:3 s vm:129414 m

Contiguous allocation with random order

Levelized contiguous allocation

User time 1.6x faster
Wall time 1.3x faster

Thread-local storage

Example: aggregate a feature in a collection

Naïve version 1

```
tbb::atomic<size_t> count;

class Fun {
  void operator()(const T& o) const { count += o.getCount(); }
};

const vector<T>& data = getData();
tbb::parallel_for_each(data.begin(), data.end(), Fun());
```

Problem: leads to contention on the global atomic

Not so naïve version 2

```
tbb::atomic<size_t> count;

class Fun {
  typedef tbb::blocked_range<vector<T>::const_iterator> Range;
  void operator()(const Range& r) const {
    for (auto it = r.begin; it != r.end(); ++it) {
      _localCount += it->getCount();
    }
  }
  ~Fun() { count += _localCount; }
  size_t _localCount = 0;
};

const vector<T>& data = getData();
Fun::Range r(data.begin(), data.end(), grainsize());
tbb::parallel_for(r, Fun());
```

Better: locally count on a range, then aggregate

But what if += is costly?

But what if creating/deleting the local variable is costly?

Example: aggregate a feature in a collection

Better version 3

```
typedef tbb::combinable<size_t> TLS;
TLS tls;

class Fun {
  typedef tbb::blocked_range<vector<T>::const_iterator> Range;
  void operator()(const Range& r) const {
    size_t& count = tls.local();
    for (auto it = r.begin; it != r.end(); ++it) {
      count += it->getCount();
    }
  }
};

const vector<T>& data = getData();
Fun::Range r(data.begin(), data.end(), grainsize());
tbb::parallel_for(r, Fun());

size_t res = 0;
tls.combine_each([&](size_t count){
        res += count;
     });
```

local() returns a reference to a thread-local counter

Aggregation is done at the end, sequentially

No atomic, no synchronization!

Note the lambda-capture by reference to accumulate in variable “res”

Example: collect objects satisfying a predicate

```
typedef tbb::combinable<set<T>> TLS;
TLS tls;

class Fun {
  typedef tbb::blocked_range<vector<T>::const_iterator> Range;
  void operator()(const Range& r) const {
    set<T>& s = tls.local();
    for (auto it = r.begin(); it != r.end(); ++it) {
      if (satisfyPredicate(*it)) {
         s.insert(*it);
      }
    }
  }
};

const vector<T>& data = getData();
Fun::Range r(data.begin(), data.end(), grainsize());
tbb::parallel_for(r, Fun());

set<T> res;
tls.combine_each([&](const set<T>& s){
        res.insert(s.begin, s.end());
    });
```

Thread-local storage of set<T>

A set<T> is created only once per thread

Aggregation is done at the end, sequentially

No atomic, no synchronization!

Memory access

- A thread works with four types of memory:
  1. The data it must process
  2. The thread-local data
  3. The data members of the functor
  4. The data on the stack

Make it contiguous in memory so that processing a range results in lesser cache miss

Created/deleted only once per thread. The function local() provides a very fast access

Created/deleted every time we create/copy/delete the functor

Created/deleted every time we call operator()(const Range&)

- Type (3) is subsumed by (2)
  - Can be justified only to increase data locality

Combinable

Why the name “combinable”?

- Apply-combine flow
  - Apply a functor on chunks (=ranges) of data, using k workers (=threads) in parallel
  - A worker produces some local result (=thread-local storage)
  - Combine (=reduce) the local results to produce the final result

- Combining

```
template<T> class tbb::combinable {

  template<BinaryFun> T    combine(BinaryFun fun);

  template<UnaryFun>  void combine_each(UnaryFun fun);
};
```

Why the name “combinable”?

- Apply-combine flow
  – Apply a functor on chunks (=ranges) of data, using k workers (=threads) in parallel
  – A worker produces some local result (=thread-local storage)
  – Combine (=reduce) the local results to produce the final result

- Combining

Fun is the reduction function

```
template<T> class tbb::combinable {

  template<BinaryFun> T    combine(BinaryFun fun);

  template<UnaryFun>  void combine_each(UnaryFun fun);
};

// Example using tbb::combinable<size_t>
size_t res = tls.combine([](size_t n1, size_t n2){
                              return (n1 + n2);
                           });

// Produces same result as above
size_t res = 0;
tls.combine_each([&](size_t n){
                     res += n;
                 });
```

Fun’s signature must be T(T, T) or T(const T&, const T&)

Fun’s signature must be void(T), void(T&), or void(const T&)

Combinable

- Methods combine and combine_each are very generic
  - They may do nothing, e.g., when the thread-local storage is a scratchpad
  - They MUST use an associate and commutative reduction function
    - Otherwise the end result depends on the thread ID and the assignment of tasks to the threads

Non-trivial examples

Example: backward tracing (1/3)

```
auto fun = [&](CellID cid) {
    if (!isStartPointForTraceTFI(cid)) { return; }

    vector<CellID> queue;

    while (!queue.empty()) {
        CellID cid = queue.back();
        queue.pop_back();

        // Thread sync
        tbb::atomic<uint8_t>& ref = cid2mark.atomicGetRef(cid);
        // Atomic read
        uint8_t newMark = (ref | VISITED);
        // Atomic swap
        uint8_t oldMark = ref.fetch_and_store(newMark);
        if (oldMark & VISITED) {
            // Another thread already went there
            continue;
        }

        const Cell& cell = *nl.getCell(cid);
        queue.insert(queue.end(), cell.beginFanin(), cell.endFanin());
    }
}
```

```
void Main::run() {
    typedef CellAnnotation<uint8_t> CID2Mark;

    NL::Netplus& nl = getNetlist();
    CID2Mark cid2mark(nl);

    _nl.applyFun(fun);
}
```

Note: this guarantees only one thread will back trace a CellID

It is not mandatory: we can allow redundant computations

Example: backward tracing (2/3)

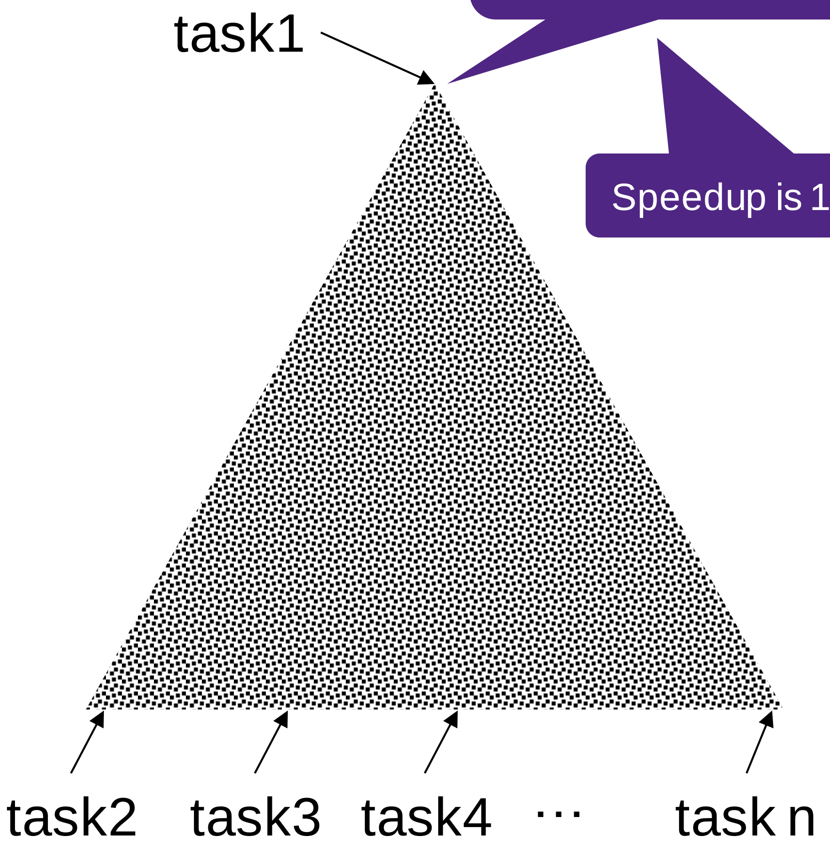
Does that algorithm scale?

```
auto fun = [&](CellID cid) {
    if (!isStartPointForTraceTFI(cid)) { return; }

    vector<CellID> queue;

    while (!queue.empty()) {
        CellID cid = queue.back();
        queue.pop_back();

        // Thread sync
        tbb::atomic<uint8_t>& ref = cid2mark.atomicGetRef(cid);
        // Atomic read
        uint8_t newMark = (ref | VISITED);
        // Atomic swap
        uint8_t oldMark = ref.fetch_and_store(newMark);
        if (oldMark & VISITED) {
            // Another thread already went there
            continue;
        }

        const Cell& cell = *nl.getCell(cid);
        queue.insert(queue.end(), cell.beginFanin(), cell.endFanin());
    }
}
```

We should launch a new tbb task if the queue is too large (using a feeder)

The thread assigned to task1 does all the work.

Speedup is 1x: no scaling

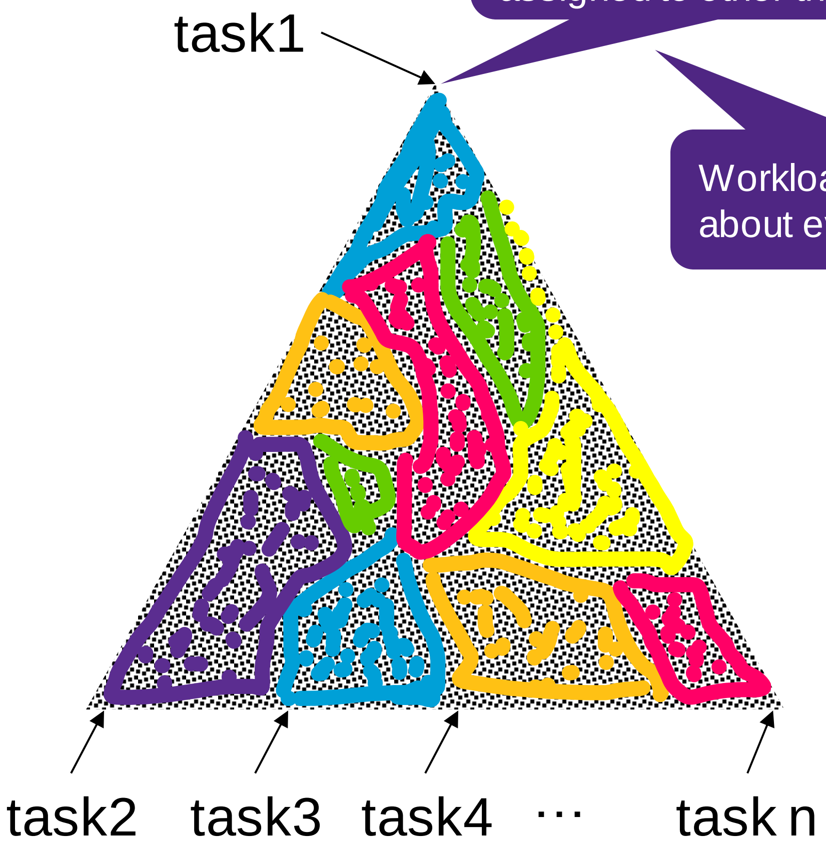


Example: backward tracing (3/3)

```
auto fun = [&](CellID cid, Feeder& feeder) {
    if (!isStartPointForTraceTFI(cid)) { return; }

    vector<CellID> queue;

    while (!queue.empty()) {
        CellID cid = queue.back();
        queue.pop_back();

        // Thread sync
        tbb::atomic<uint8_t>& ref = cid2mark.atomicGetRef(cid);
        // Atomic read
        uint8_t newMark = (ref | VISITED);
        // Atomic swap
        uint8_t oldMark = ref.fetch_and_store(newMark);
        if (oldMark & VISITED) {
            // Another thread already went there
            continue;
        }

        const Cell& cell = *nl.getCell(cid);
        auto it = cell.beginFanin();
        for (; it != cell.endFanin() && queue.size() < MAX_SIZE; ++it) {
            queue.push_back(it->cid());
        }
        for (; it != cell.endFanin(); ++it) {
            feeder.add(it->cid());
        }
    }
}
```

Spin off new tasks for the excess cells

Once task1's queue is too large, it starts new tasks, which are assigned to other threads.



Workloads become about even, this scales

Example: constant propagation

```
enum {
    VAL_VOID = 0x0,
    VAL_ZERO = 0x1,
    VAL_ONE  = 0x2
} e_Mark;

class Fun {
public:
    Fun(Main& main) : _main(main) {}
    Fun(const Fun& o) : _main(o._main) {}
    void operator()(CellID cid) const { run(cid); }
private:
    const Netplus& nl() { return _main._nl; }
    CID2Mark& cid2mark() { return *_main._cid2mark; }
    void run(CellID cid);

    struct Task {
        Task() : _cid(NullCellID), _val(VAL_VOID) {}
        Task(CellID cid, e_Mark val)
            : _cid(cid), _val(val) {}
        CellID _cid;
        e_Mark _val;
    };
private:
    Main&   _main;
};

void Main::run() {
    _cid2mark.reset(new CID2Mark(_nl));

    Fun fun(*this);
    _nl.applyFun(fun);
}
```

```
void
Fun::run(CellID cid) {
    e_Mark val = getConstantValue(cid);
    if (val == VAL_VOID) return;

    _taskQueue.push_back(Task(cid, val));

    while (!_taskQueue.empty()) {
        Task task = _taskQueue.back();
        _taskQueue.pop_back();

        CellID cid = task._cid;
        e_Mark val = task._val;

        // Thread sync
        uint8_t newMark = val;
        uint8_t oldMark = main().cid2mark().atomicGetRef(cid).fetch_and_store(newMark);
        if (oldMark == VAL_ZERO || oldMark == VAL_ONE) {
             // Another thread propagated that constant.
             continue;
        }

        auto wire = cid2wire().getWire(cid);
        for (auto it = wire.beginReader(), itEnd = wire.endReader(); it != itEnd; ++it) {
            // Get the cell driven by wire.
            CellID cid = it->cid();
            Cell& cell = *nl().getCell(cid);
            // Try to push the constant through cell.
            e_Mark val = propagateCstThroughCell(cell);
            if (val != VAL_VOID) {
                _taskQueue.push_back(Task(cid, val));
            }
        }
    }
}
```

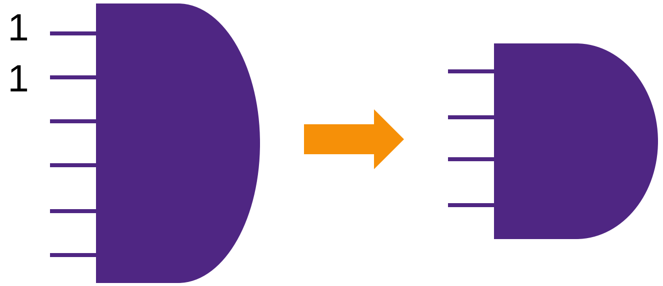
Multithreading: key observations

- Memory layout
  - In a tbb::parallel_for, a thread goes sequential after the range is small enough
    - Make sure the data the thread reads/writes is in the same cache line
  - Design memory layout properly –linear and/or vectorized layout
  - Optimize memory footprint –the smaller, the more data in the cache line
  - Use functor-local or thread-local data
- Caching
  - It can be very counterproductive, as it requires thread sync
  - It is then often much better to allow redundant computations
- Determinism
  - Forcing a total order on any sequence of objects (or actions) is a performance killer
  - Rethink algorithms and flow
    - To avoid dependency on object order
    - Or to introduce normalization steps

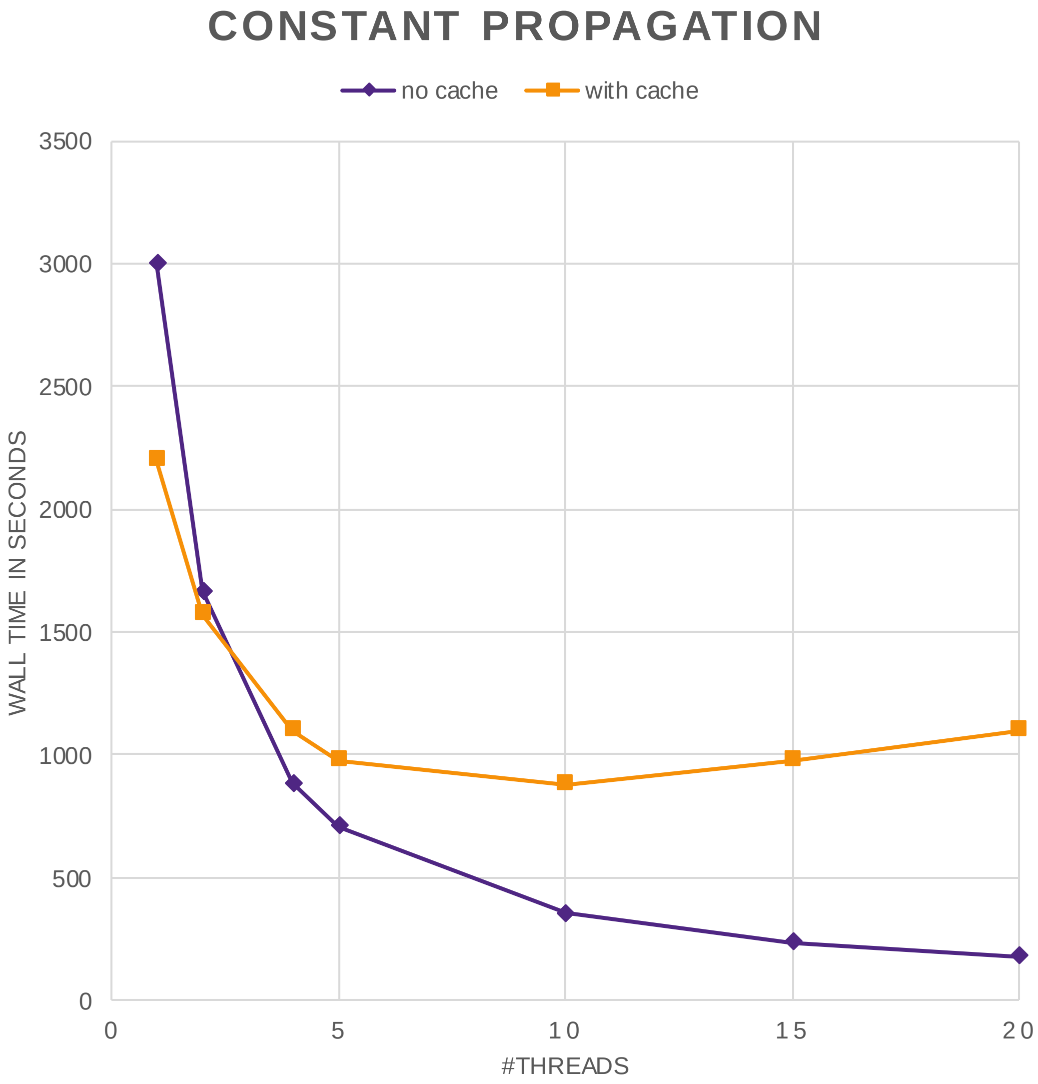
The two most overlooked algorithmic choices that prevent free-running threads

The less synchronization, the more scalable

- LUT is a lookup table that can be simplified whenever an input is constant



- When a constant value reaches an input:
  - Check if the cell is constant given the values (0, 1, unknown) at its inputs
- We can cache simplified lookup tables to avoid redundant LUT simplifications
  - Need mutex to read/write the cache
  - Too much contention: not scalable



Conclusion

Conclusion

- Design
  - Identify independent tasks
  - Consider redesigning the algorithm to improve scalability
  - Implement concurrency at the highest possible level
  - Never assume a particular order of execution
  - Atomic-based synchronization is harder to write but scales much better
  - Avoid high frequency dynamic memory requests (new/delete)
  - Use combinable whenever it makes sense
  - Watch the memory layout
- Test for correctness
  - Always have a sequential version using the same functor Fun::operator()(…)
  - Define determinism and check for it
- Test for scalability
  - Performance with 1 master thread should be the same as std::for_each
  - Check *absolute* speedup: (wall time with 1 thread) / (wall time with n threads)
  - Check *relative* speedup: (user time with n threads) / (wall time with n threads)
  - System time should be low (< 1-6% of user time)

Add redundancy to increase independency of tasks

Fine-grain scalable MT'ing is much more difficult to achieve

Threads WILL overlap

Determinism can be extremely challenging