Haifa::C++ presents:

Parallel Programming Models

By Eran Gilad

28/1/20, 17:30 @ yahoo! research, Matam, Haifa

Opening notes

- Core C++ 2020 conference: 25-27/5
  - Call for submissions, sponsors, early bird
- C++20 - who's up to the challenge?
  - Coroutines, modules, concepts, ranges
- Raffle at the end - thanks, Jetbrains!

About me

IDEA Information Systems

הפקולטה למדעי המחשב בטכניון
TECHNION COMPUTER SCIENCE DEPARTMENT

yahoo! research

Haifa::C++

Fun fact - me & 42:

איגוד האתלטיקה בישראל
ISRAELI ATHLETICS ASSOCIATION

רשימת הטובים גברים
קפיצה לגובה

דר`	תוצאה	מיקום	אתלט	ש.ל	אגודה	עיר	תאריך
42	2.01	(6)	ערן גלעד	75	הפועל מגידו	תל-אביב	21.06.92

42!

About yahoo! research

- → Part of Verizon Media:
  - → Yahoo, AOL, TechCrunch, HaffPost and many other brands
  - → #4 in the US in terms of unique visitors, #3 in video views
  - → 10% of global internet traffic
  - → 180 engineers & scientists in Tel Aviv and Haifa
- → Yahoo research:
  - → Haifa office
  - → Data science, NLP, AI, scalable systems
  - → Products, open source, patents, academic papers
- → Hey, we're hiring! see verizonmedia.com/careers

Why parallel programming?

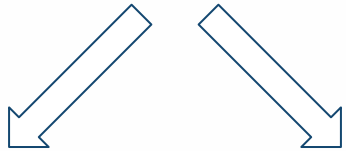
1. Performance
2. Hide I/O
3. Responsiveness

Why is it hard?
non-deterministic execution, data races, deadlocks

Having *structure* makes things easier

Programming model

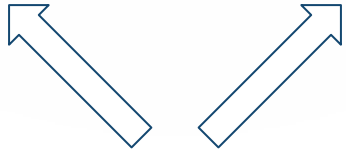
Models provide structure



User can reason about correctness

Library can provide useful features

Runtime can optimize scheduling



Crucial in parallel programming!

C++ parallel programming facilities

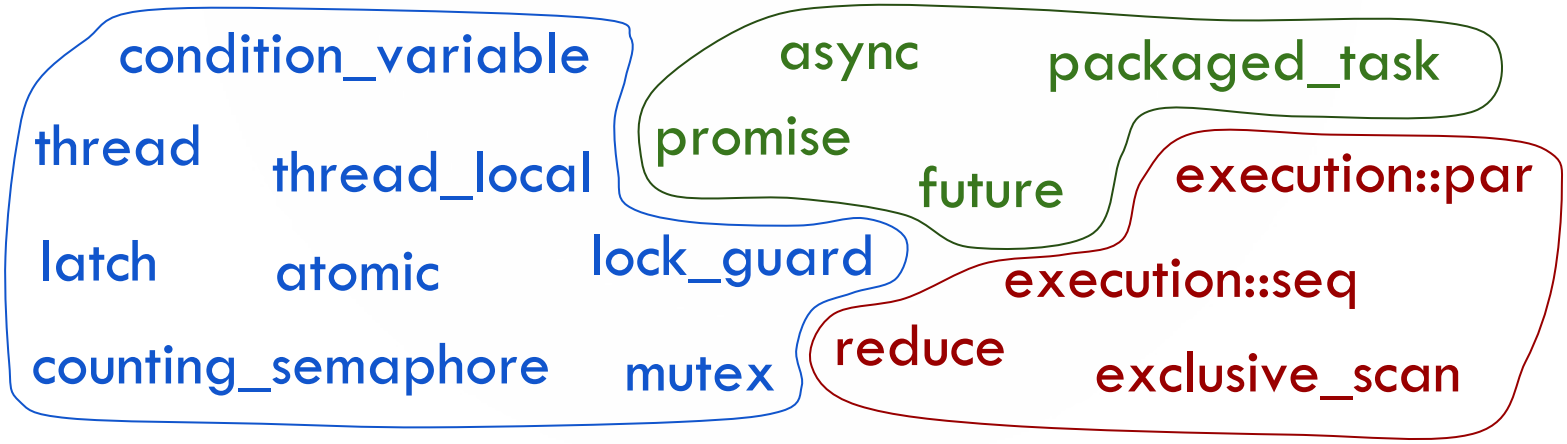
condition_variable async packaged_task

thread thread_local promise future execution::par

latch atomic lock_guard execution::seq

counting_semaphore mutex reduce exclusive_scan

C++ parallel programming facilities



Not random facilities, not a single API:
3 parallel programming models

C++ parallel programming models

→ Modern C++ offers 3 parallel programming APIs:

1. **Unstructured** – thread, mutex, atomics etc. (C++11/14)
2. **Tasks** – async, promises and futures (C++11)
3. **Data parallelism** – concurrent algorithms (C++17)

→ Not different *abstraction levels* - different program structuring

→ Each model is appropriate to different use cases

Talk outline

→ Intro
→ Unstructured parallelism
→ Task-based parallelism
→ Data parallelism
→ Mixing models
→ Conclusions

This talk is about the forest, not about the trees

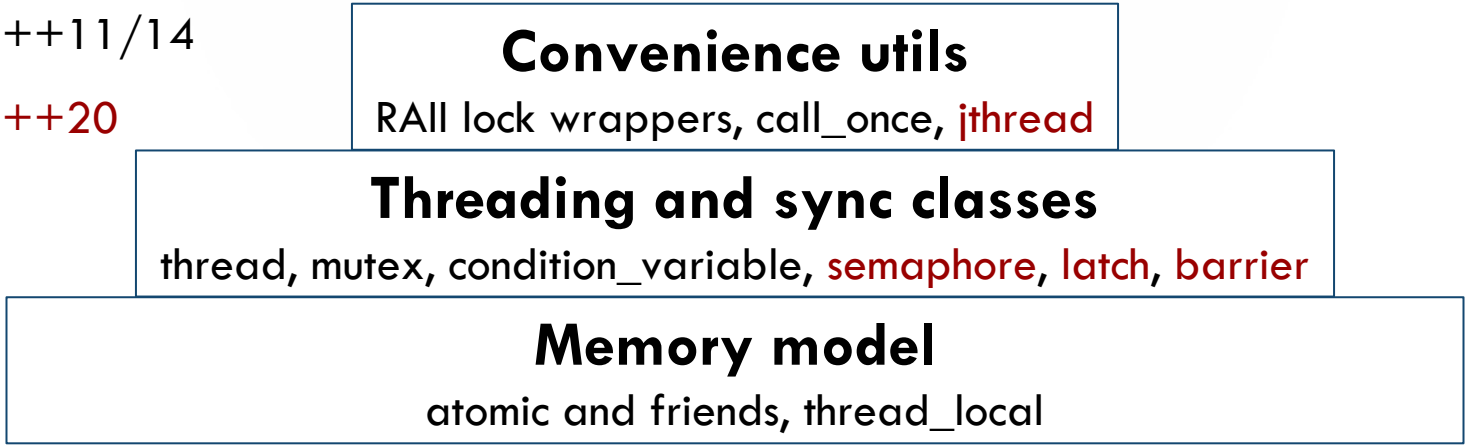
Unstructured parallelism

- → Ad hoc use of parallelization facilities; no particular structure
- → Relies on lower-level abstractions
- → Minimal overhead and maximal hardware utilization
- → Minimal safety, so requires maximal proficiency
- → Introduced in C++11, central language feature

Major components (partial list)

C++11/14

C++20



Exclusive use cases

- → Construct higher level facilities
  - → E.g., thread pool, spin lock etc.
- → Concurrent data structures
  - → At least thread safe, usually better if lock free
- → Long running services

Missing part - safe shared state

- → Threads communicate via shared state
  - → atomics are too fine-grained
  - → locking complete containers is too coarse grained
- → We need *concurrent data structures!*
  - → Being considered: queues, hash maps, RCU etc.
  - → No consensus, hopefully some for C++23

Pros and cons

✔ Maximal control

✔ Usually maximal performance (when done right)

✗ Complicated memory model, hard to make ideal use

✗ Data races, deadlocks, non-determinism

Thread-level API shortcomings

- → Thread ~= processor
  - → Gets an execution entry point
  - → No functional semantics
  - → Synchronization, communication, scheduling interleaved in code

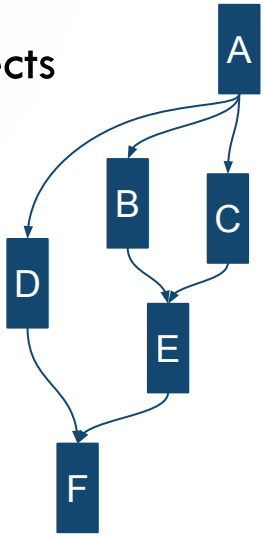
Higher abstraction level + clever runtime

⇓

Less work, less bugs, probably better performance

Task-based parallelism

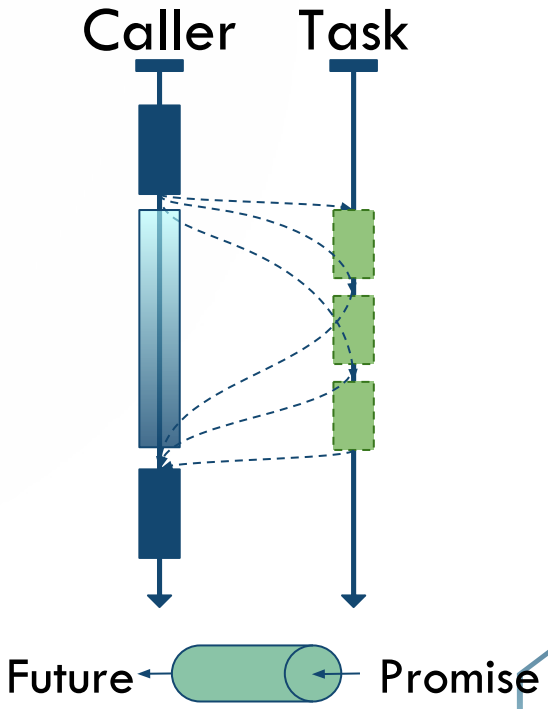
- Task: a limited computation providing a single result
  - Function, lambda, loop iteration etc.
  - Ideally, has inputs and/or output but no side effects
- Decouples functionality from execution
  - One thread can run many tasks
  - One task can be run by many threads



Tasks are asynchronous

→ Executed somewhere, sometime
→ Caller proceeds till outcome needed

→ *Future*: A handle to the task outcome
→ *Promise*: the task's end



C++ tasks

- → Spawn tasks using std::async
- → Obtain results using std::future
- → The runtime assigns tasks to worker threads
  - → Yes, the C++ runtime can create and manage threads without user intervention!

async

```
future<int> ans = async(when, []{ return 42; });
pretend_to_work();
cout << "found the answer: " << ans.get() << endl;
```

Task execution determined by *when*:

1. launch::async - on another thread ("as if"!)
2. launch::deferred - right before the results are needed
3. async | deferred - implementation dependent (default)

future and promise

- future
  - Returned by a call to async
  - Provides access to the task's result
  - The "pull" end of a communication channel
- promise
  - The "push" end of the communication channel
  - Encapsulated by async

packaged_task

```
packaged_task<int()> task([]{ return 42; });
future<int> ans = task.get_future();
thread task_td(move(task)); // ~= async(...)
pretend_to_work();
task_td.join();
cout << "found the answer: " << ans.get() << endl;
```

→ async minus execution

→ Can be used by custom task management libraries

What's C++ tasks model like?

Current C++ model

Pool size
Composition
Cooperative
scheduling

Well designed task model

Experimental future features

- `then()` - chain continuations to one task
- `when_all()`, `when_any()` - chain to a set of tasks
- Parallelism TS1

```
future<int> ans = async(launch::async, []{...});
pretend_to_work();
ans.then(print_answer);
```

- On hold due to...

Coroutines and executors

coroutines

(C++20)

executors

(C++23?)

Threads vs. tasks

Threads	Tasks
Background services	Many short and independent ops
Long complex parallel operations	“local”/ad-hoc (async I/O)
Equal amount of per-thread work	Dynamic load balancing
Thread pool building block…	Non-trivial runtime

async vs. thread

- async (and tasks) provide
  - Better resource utilization (when called often)
  - Clean return value mechanism
  - Exception safety
  - Timed waiting (unlike strict thread joins)
  - Safe multiple accesses to futures (unlike recalling join)
- Threads provide deterministic execution context
  - A separate thread…

Data parallelism

A programming model in which parallelism stems from the individual computation associated with every element in a collection.

- → Applicable when there's lots of non-dependent data
- → Procedure:
  - → Divide processed elements among processors
  - → Perform fairly short and simple operation on each element

C++ Parallel algorithms

- → Accelerate processing of large amount of data
- → Sequential execution context
- → Declarative API
  - → Few customization points for user
  - → Library + runtime can be very efficient
  - → When used properly: no data races, deadlocks etc.
- → High level abstraction
  - → No control over parallelism, scheduling, work distribution etc.
  - → Parallel execution possible but not guaranteed

Parallel algorithms availability

- → Introduced in C++17
- → Only recently supported:
  - → GCC 9
  - → VS 19 (partial support earlier)
- → Available as library implementation (Intel, SYCL)
- → Possible delay reason: new skill set for library authors
  - → A portable efficient implementation isn't easy

Using parallel algorithms

```
vector<int> v = {1, 2, 3, 5, 11, 20};
int res = reduce(execution::par, v.begin(), v.end());
assert(res == 42);
```

⟶ Most standard algorithms have a parallel overload

- → First parameter: *ExecutionPolicy*
- → *ForwardIterator* instead of InputIterator / OutputIterator

⟶ Complexity requirements more lax

⟶ Implementation isn't specified – in theory, can use GPUs

sequenced_policy

→ Forces execution to take place on the calling thread

→ Differs from no-policy call in three ways:

1. exceptions invoke std::terminate
2. order of execution is unspecified; operations not interleaved
3. uses ForwardIterator instead InputIterator and OutputIteration

→ execution::seq is an *instance of* sequence_policy

→ Allows overloading

```
reduce(execution::seq, ..., ...);
```

parallel_policy

→ Execution on caller or another thread (runtime pool)

→ Per thread, semantics are similar to sequenced_policy - unspecified order, no interleaving

→ Data races are now possible if multiple operations write to unprotected data

```
reduce(execution::par, ..., ...);
```

parallel_unsequenced_policy

- → Operations can now be interleaved and moved from thread to thread during execution
  - → Operations must not use any locks
  - → Cannot assume a thread executes a single operation at a time
- → More user restrictions => more library options
  - → Vectorization can now be used
  - → finer grained scheduling

```
reduce(execution::par_unseq, ..., ...);
```

unsequenced_policy?

- → Expected for C++20
- → Operations can be interleaved on a single thread
  - → Not a multithreaded context
  - → But vectorization can still be used

```
reduce(execution::unseq, ..., ...);
```

Non-standard policies

- ⟶ Vendor-specific
- ⟶ Can allow the use of accelerators
  - → GPU
  - → FPGA
  - → ASIC

Parallel algorithms != parallel containers

→ Parallel algorithms rarely modify containers *structure*

- → E.g., back_inserter not allowed as output iterator
- → Safely done by the algorithm on a few cases (e.g., reduce)

→ User threads can cause races if accessing a container participating in a parallel algorithm

```
vector<int> v = {1, 2, 3, 4};
set<int> u;
transform(v.begin(), v.end(), inserter(u, ...), ...);
transform(execution::par, v.begin(), v.end(), inserter(u, ...), ...);
```

Models comparison

	Unstructured	Tasks	Data parallel
Parallelism	Custom	Functional	Data
State	Shared, global	Promise → future	Container
Context	Custom	Spawning task	Sequential
Scheduling	Custom	Task queue	Max fan-out
Status	Established	To be redesigned?	Slow adoption

Mixing models 1 - unstructured context

- Reminder: runtime schedulers unaware of user threads
- Case 1: most cores are used (e.g., server)
  - Can use tasks for async I/O
  - No resources for parallel algorithms (till GPUs supported)
- Case 2: most cores are unused (background services)
  - Tasks and parallel algorithms can freely be used
  - Careful with shared state! (as always)

Mixing models 2 - tasks context

- Case 1: most cores are used (many tasks created)
  - No point in creating threads or parallelising algorithms
- Case 2: most cores are unused (async I/O, UI worker)
  - Can use parallel algorithms
  - Creating threads doesn't make sense - spawn tasks instead
  - Using sync mechanism (mutex etc.) doesn't fit the model

Mixing models 3 - parallel algorithms

- → No point in spawning async tasks or creating threads within a parallel algorithm
- → Using sync mechanisms to access external state possible when needed, but might kill concurrency
- → No memory model requirements for the container (specified at the algorithm level)

Summary

parallelism facilities -- E.G.

"The nice thing about ~~standards~~ is that there are so many of them to choose from." -- Andrew S. Tanenbaum

→ Parallelism is hard, prefer high-level models

→ Parallel algorithms - simpler mental model; slow support; great potential with accelerator support

→ Task based - underutilized today; big future improvements; key for asynchronous programming