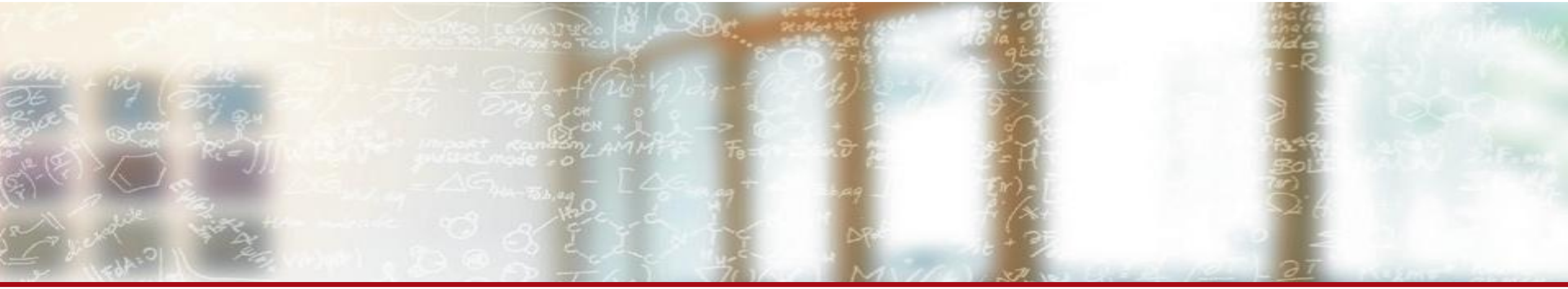
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



HPX : High performance computing in C++ with concurrency, parallelism and futures

Italian C++ day, Pavia, November 24 2018

John Biddiscombe + material from Mikael Simberg, Raffaele Solcà, Shoshana Jakobovits, Joost VandeVondele, Hartmut Kaiser, Thomas Heller, Agustín Bergé and many other HPX contributors

Topics of discussion (not in order)

- Quick introduction to CSCS/SSL
  - Task based programming & How we are using HPX
- HPX, what is it and how is it different (or the same) to other libraries
  - Futures, Executors, Schedulers, Thread Pools
- Standards conformance and proposals
  - Some of the more relevant ones (not exhaustive)
- Parallel Algorithms
  - API, Implemented using the same internals as the public API
  - Execution policies
- Future directions
  - Mixing HPX with other libraries
  - Fork-Join & Asynchronous
- Distributed HPX
  - Unified syntax and semantics for local and remote operations.

Three things you should learn today

- HPX aims to be a C++ standards conforming implementation of as *many as possible* of the Parallelism and Concurrency (and Heterogeneous) proposals for C++ 17/20/23/…
    - Keep stuff that works … Fix stuff that doesn't (if possible)

- We want to give C++ programmers the power to optimize their code for the hardware they are using, whilst making it as easy as possible to do the basics.

- Task Based Programming with `futures<>` unlocks the power of multicore processors using a straightforward and easy to use API
    - Future syntax might change, but the principles of AMT won't

CSCS - Swiss National Supercomputing Centre

- CSCS is autonomous sub-dept of ETHZ
  - Swiss Universities & Collaborators world-wide use our machines
  - Piz-Daint is #5 as of Nov 2018 (SC18 list announced)
    - Piz Daint - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , Cray Inc.
    - Cores = 387,872, Peak = 21PFlops, scratch filesystem = 8.8 PB
    - 5704 GPU nodes (Pascal P100) (Haswell 12 core CPU)
    - 1813 Multicore nodes (2 x Broadwell 18 core CPU)

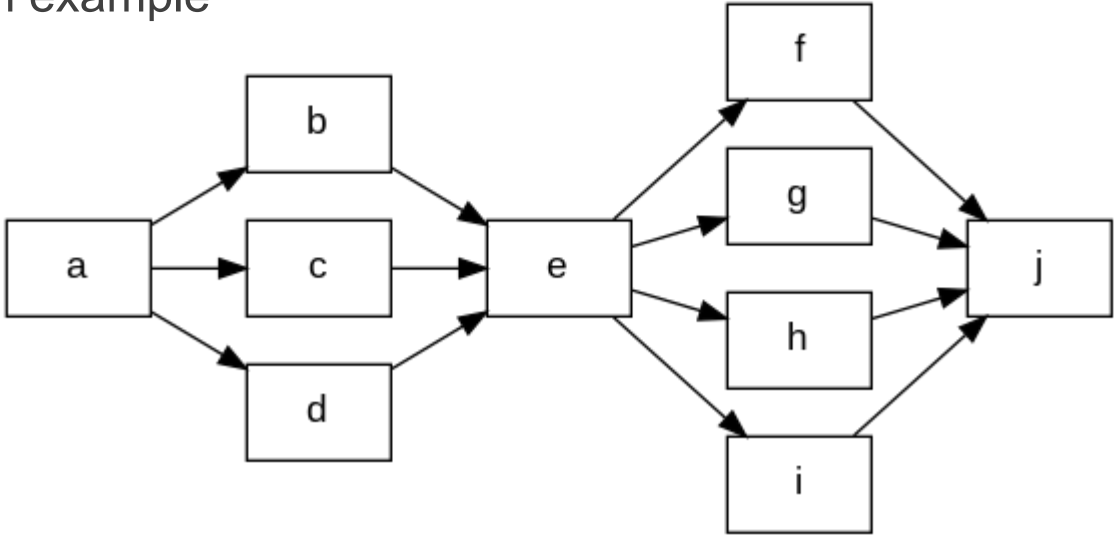
CSCS and SSL

- SSL - Scientific Software & Libraries
  - Developing libraries/tools to optimize codes used by researchers
  - R&D on new techniques to squeeze performance out of
    - Algorithms
      - Better ways of solving existing problems
    - Hardware
      - Cache, vector instructions, threads, messaging
    - Programming Languages
      - Adopting and testing new standards
    - Approaches like “Task Based Programming”
      - Can we improve codes by using it
      - The subject of this talk

Task Based programming #1

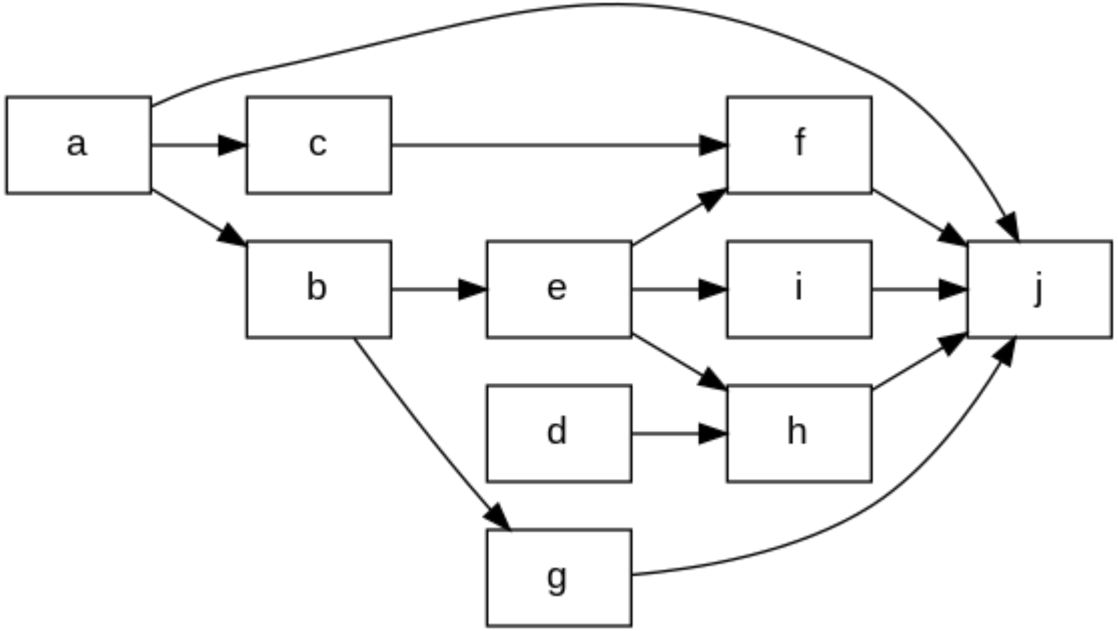
- Breaking program/problem into asynchronous “Tasks”
  - And defining flow of execution in terms of those tasks
  - By building Directed Acyclic Graphs (DAGs)

- Fork-join example



Task Based programming #2

- Using a Scheduler (runtime) to coordinate execution of tasks
  - Maximize throughput of the cores
  - Prioritize work according to need
  - Minimize wait time



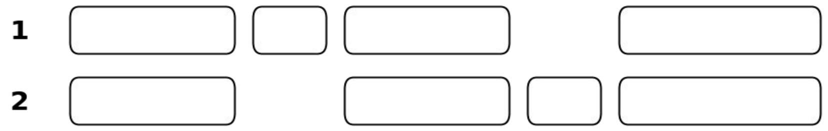
Task Based programming #3

- Hiding the finer details of multi-threading behind the task API
    - Create and connect tasks, not threads
    - Synchronization implicit in graph structure

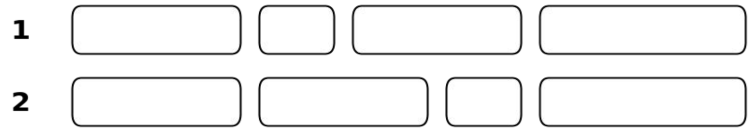
- Encouraging a functional programming approach
    - Dataflow
        - Tasks take inputs
        - Tasks produce outputs
        - Continuation Passing Style (CPS)
    - All state should ideally pass in/out via parameters and returned values
    - Avoid global data
    - Less Locking/Races/Waiting in user code

Task size helps scheduling in the runtime

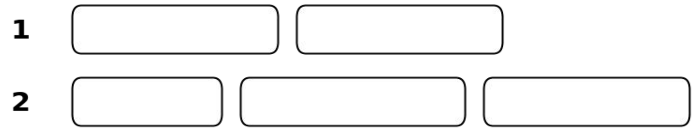
- Fork-join



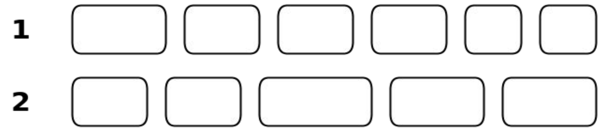
- No fork-join (optimal)



- Tasks too large



- Tasks just right
    - Minimize dependencies
    - Keep queues full



Q: Why HPX? A: Standards conformance

- Task based parallelism is here to stay
  - A significant rewrite might/will be required for many HPC codes
  - But only once
    - Do not want to port to library X this year
    - Then library Y a few years later
  - Future-proof against major code rewrites

- Core language supported features
  - C++ 17/20/23 and beyond
  - Not reliant on vendor or compiler extensions
  - Platform portable syntax
  - Architecture friendly (+ Heterogeneous computing)

Why not OpenMP (or one of many other runtimes)

- Directives (#pragma ...) are not as flexible as core language
  - OpenMP is great for adding parallelism to existing code
  - We want run-time as well as compile time parallelism/features
    - Functions returning tasks
    - Dynamic task trees (‘if’ based branches/continuations)
  - Nested parallelism

- Maintenance of code in 5-10-more? years timescale
  - As features are added to C++
    - Less need for (#pragma) extensions
- Other task based languages/runtimes
  - No preprocessing or other tools/dependencies
  - No symbolic/intermediate representation needed

HPX : Objective : unified/consistent ‘futurized’ API

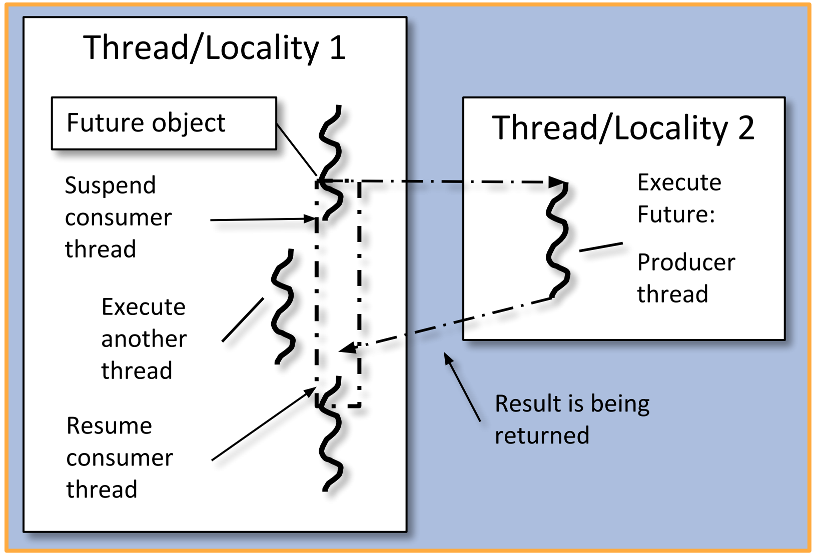
- Concurrency
  - Several tasks doing work using shared resources
- Parallelism
  - Several tasks working on the same core algorithm

- Heterogeneous computing
  - CPUs and GPUs etc
- Distributed computing
  - Tasks distributed on a multi-node machine

- Eventually replace OpenMP, OpenAcc, MPI?
  - Absorb them all into pure `std::C++`

What is a `future<T>`?

- A future is an object representing a result which has not been calculated yet



- Transparent synchronization with producer
  - Thread A sets value, thread B gets it
- Hides notion of dealing with threads
  - Shared futures can be accessed by threads B,C,D…
  - Makes asynchrony manageable
- Allows for composition of several asynchronous operations
  - Continuations : Building DAGs from many futures
  - Express parallel algorithms using the language of concurrency

Tasks on fine grained lightweight threading model

- Reimplement C++ `future<>` on lightweight threads
  - Tasks / asynchronous computing
  - Dataflow / continuation based (DAG)
    - Dependencies between arbitrary tasks
    - Fork-join semantics
    - Nested parallelism
- STL algorithms implemented
  - Using same framework as higher level task/future API
- Distributed tasks
  - Same API for executing remotely
  - `future<>` returned from remote tasks
- Accelerator integration
  - `future<>` returned from GPU tasks

What about Lightweight Threads?

- It's the job of the operating system to manage 'hardware level' or 'kernel level' OS threads.
  - OS threads are expensive to create/destroy
  - OS threads take a time slice of CPU time
  - Creating too many of them can hamper performance

  "*A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system*" Wikipedia

- On Startup HPX creates one 'worker' thread per core
  - On each hardware thread, HPX runs its own Task Scheduler.
- HPX (Threads)/Tasks are executed on the HPX worker (OS) thread
  - You must be careful not to block the underlying thread
  - Each HPX 'task' is referred to as a *lightweight thread*

Side track : Threaded code is hard to get right

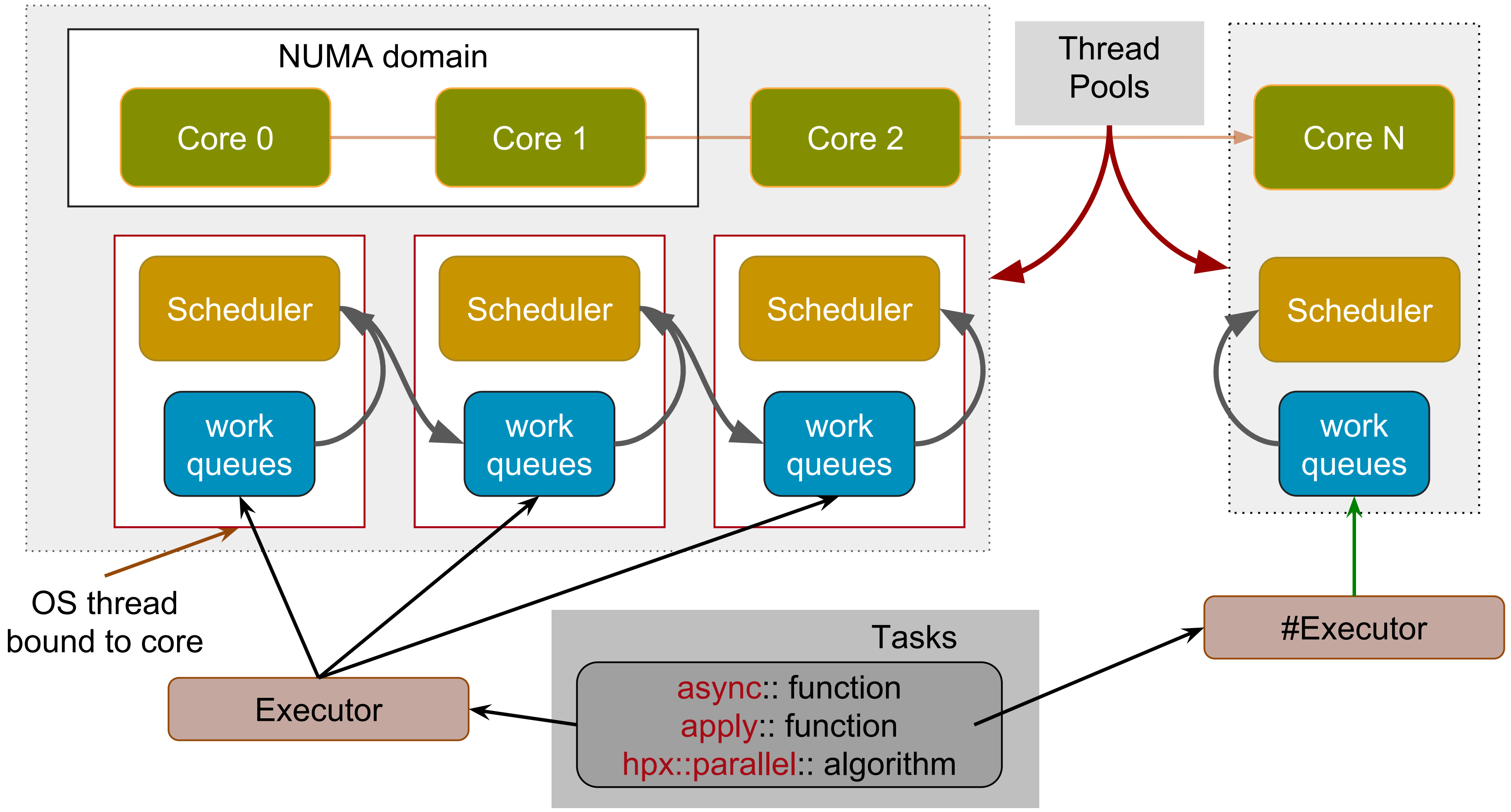
- Threaded programming is hard to get right:

```
// infiniband pseudo example
my_map[tag] = send(dest, data, ...)
```

- OS suspended thread after send had gone, but before assignment to map
- Return message came back before thread resumed
  - Another thread handled incoming message
  - Map entry was not there
  - How do you get a reply to a message you (think you) haven’t sent yet?

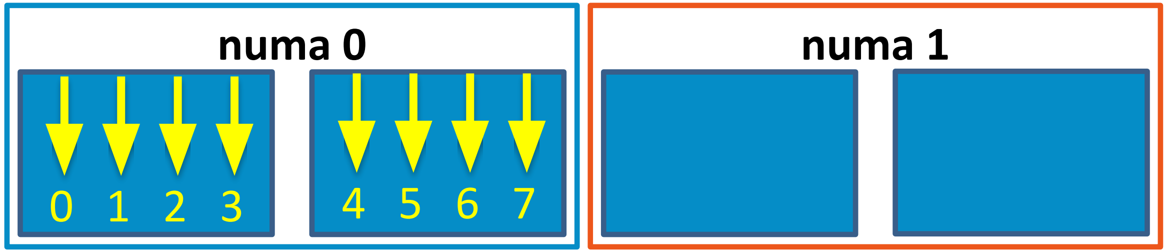
Anything that can go wrong - will go wrong - in multithreaded code

Runtime: Threads, Schedulers, Executors & Tasks

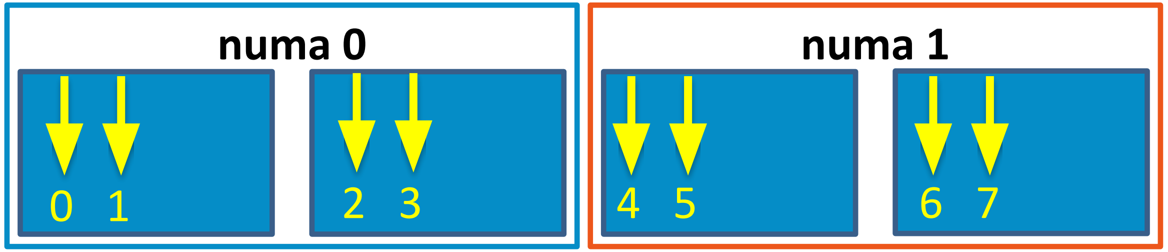


Affinities

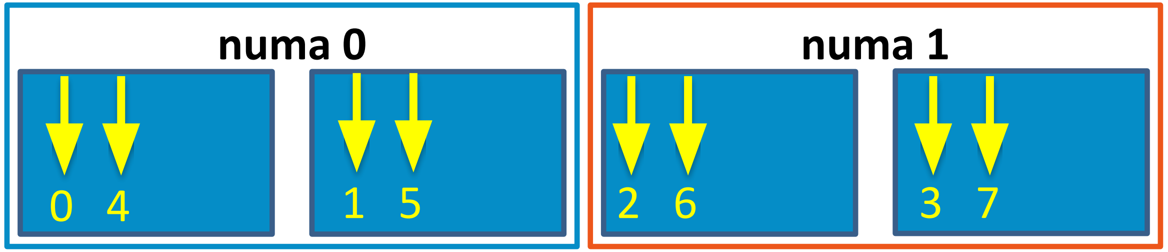
compact



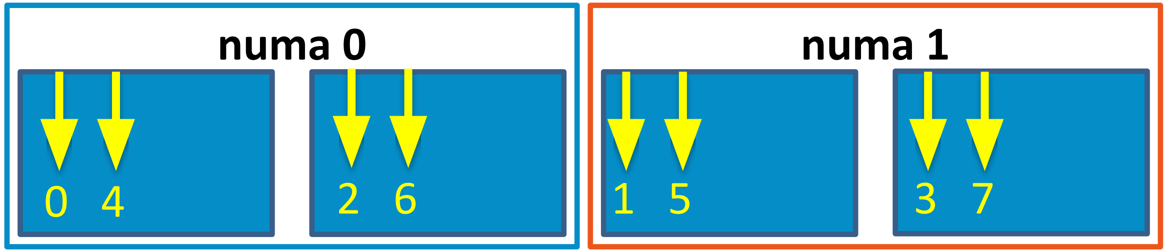
scatter



balanced



numa-balanced



Advantage of Lightweight Threads

- The philosophy behind lightweight threads is to switch from one task to another as quickly as possible as soon as anything
  - finishes / needs to wait (suspend)

- Suspended tasks
  - Many tasks can run on the same worker thread (one after another)
  - or intermixed as one task suspends and another resumes

- HPX does not ever interrupt your task directly
  - the OS worker thread may be suspended as it loses its time slice
  - the HPX task running at the time is therefore suspended
  - other (dependent) tasks might in turn suspend

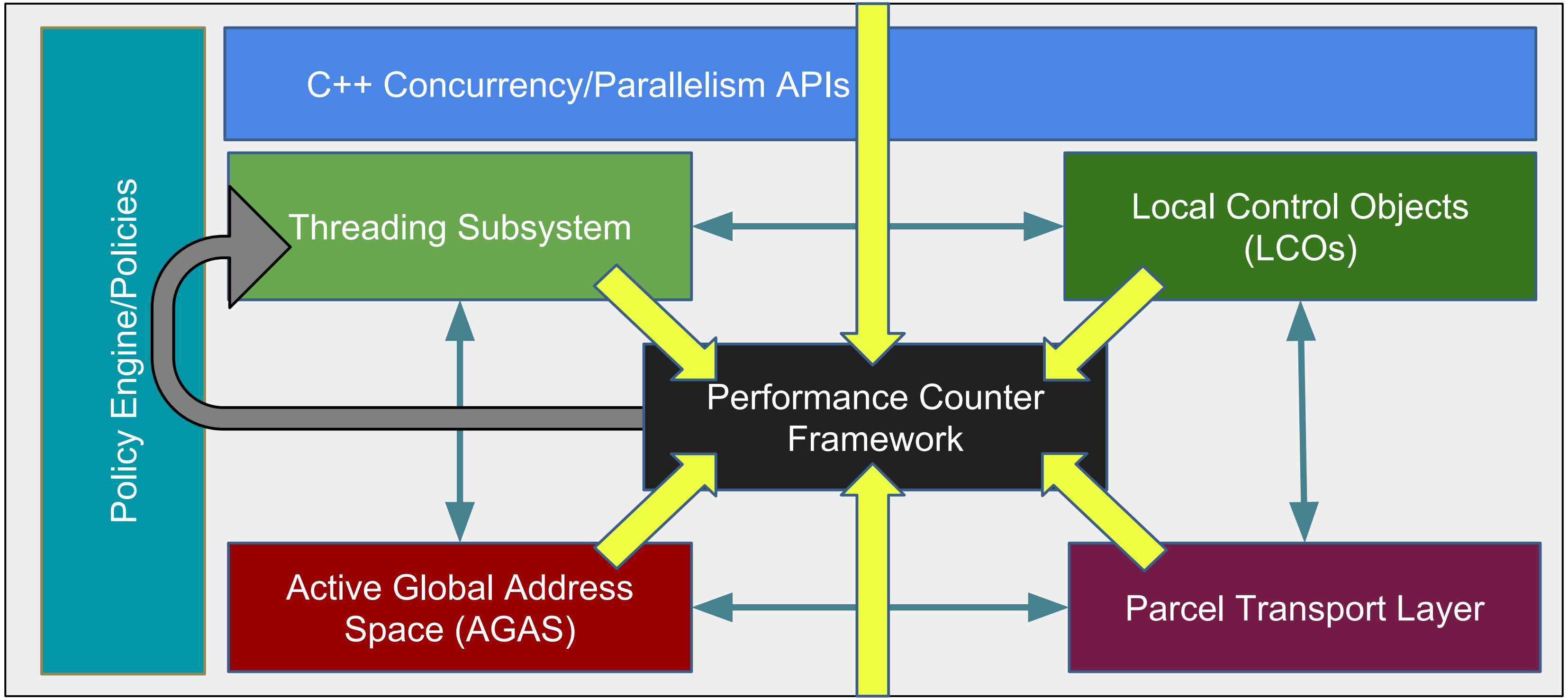
HPX Runtime

- Similar to OpenMP/TBB, but ...
  - In OpenMP has parallel regions a thread pool executes your loops/tasks
  - outside those regions, code runs as usual (on normal 'OS thread')
  - TBB does not allow for your tasks to be suspended

- With HPX, the runtime is always active
  - the runtime is started on program startup
  - it stays active until program termination
  - there are no parallel regions
  - everything is running on an HPX thread
  - everything is part of a task

- Disclaimer : you can manually start/stop the runtime

HPX system components


C++ Concurrency/Parallelism APIs
Policy Engine/Policies
Threading Subsystem
Local Control Objects (LCOs)
Performance Counter Framework
Active Global Address Space (AGAS)
Parcel Transport Layer

API replacements necessary for runtime

- As close as possible to C++17/20/... standard library, where appropriate, for instance
  - std::thread hpx::thread (e.g. sleep)
  - std::mutex hpx::mutex
  - std::future hpx::future (including N4538, ‘Concurrency TS’)
  - std::async hpx::async (including N3632)
    - std::bind hpx::bind
    - std::function hpx::function
    - std::tuple hpx::tuple
    - std::any hpx::any (N3508)
  - std::cout hpx::cout
  - std::parallel::for_each, etc. hpx::parallel::for_each (‘Parallelism TS’)

HPX API Basics

Obtaining a Future via async

- Many ways to get hold of a future, simplest way is to use (std) async:

```
int universal_answer() { return 42; }

void deep_thought()
{
    future<int> promised_answer = async(&universal_answer);

    // do other things for 7.5 million years

    cout << promised_answer.get() << endl;   // prints 42
}
```

Ways to Create a future

- Standard defines 3 possible ways to create a future,

- 3 different '*asynchronous providers*'

  - **`std::async`**
    - See previous example, **`std::async`** has caveats (blocking destructor of future in **`std::`** **not honoured** in **`hpx::`**)

  - **`std::packaged_task`**

  - **`std::promise`**

Packaging a Future

- `std::packaged_task` is a function object
  - It gives away a future representing the result of its invocation

- Can be used as a synchronization primitive
  - Pass to `std::thread`

- Converting a callback into a future
  - Observer pattern, allows to wait for a callback to happen
  - (callbacks run on the thread they are invoked on)

Packaging a Future

```
template <typename F, typename ...Arg>
future<typename std::result_of<F(Arg...)>::type>
simple_async(F func, Arg arg...)
{
    packaged_task<F> (func);
    auto f = pt.get_future();

    thread t(std::move(pt), arg...);
    t.detach();

    return f;
}
```

Promising a Future

- `std::promise` is also an *asynchronous provider*
  - ("an object that provides a result to a shared state")

- The promise is the thing that you *set* a result on, so that you can *get* it from the associated future.

- The promise initially creates the shared state

- The future created by the promise shares the state with it

- The shared state stores the value

Promising Futures

```
hpx::lcos::local::promise<int> p;
hpx::future<int> f = p.get_future();
// f.is_ready() == false, f.get(); would lead to a deadlock

p.set_value(42);

// Print 42
std::cout << f.get() << std::endl;
```

Producing Futures (supporting remote async)

```
hpx::lcos::promise<int> p;
hpx::future<int> f = p.get_future();
// f.is_ready() == false, f.get(); would lead to a deadlock

hpx::async(
    [](hpx::id_type promise_id)
    {
        hpx::set_lco_value(promise_id, 42);
    },
    p.get_id());

// Print 42
std::cout << f.get() << std::endl;
```

Promising a packaged task

```
template <typename F> class simple_packaged_task;

template <typename R, typename ...Args>
class simple_packaged_task<R(Args...)> // must be move-only
{
    std::function<R(Args...)> fn;
    promise<R> p;                      // the promise for the result
public:
    template <typename F>  explicit simple_packaged_task(F && f) :
        fn(std::forward<F>(f)) {}
    template <typename ...T>
    void operator()(T &&... t) { p.set_value(fn(std::forward<T>(t)...)); }
    std::future<R> get_future() { return p.get_future(); }
};
```

Make a ready Future

- Create a future which is ready at construction (N3857)

```
future<int> compute(int x)
{
    if (x < 0) return make_ready_future<int>(-1);
    if (x == 0) return make_ready_future<int>(0);

    return async([](int par) { return do_work(par); }, x);
}
```

- Wondering when you might need one?

Futures can change the way you write code

- You might end up writing your code backwards
- Modify a function to spawn tasks
- Return a future instead of a value
- Modify the function that calls it
- Return a future from that
- Rinse and repeat … until you are back at the start
- Somewhere you'll need a `make_ready_future`
- NB `future<future<T>>` can decay to `future<T>`

Extending futures (hpx style)

- Several proposals (draft technical specifications) for (next?) C++ Standard

  - Extension for `future<>`
    - Compositional facilities
    - Parallel composition
    - Sequential composition

- Parallel Algorithms
- Parallel Task Regions

- Extended async semantics: dataflow

Compositional Facilities .then() [.submit()]

- Sequential composition of futures (see old Concurrency TS)

```
string make_string()
{
    future<int> f1 = async([]() -> int { return 123; });
    future<string> f2 = f1.then(
        [](future<int> f) -> string {
            return to_string(f.get());    // here .get() won’t block
        });
}
```

- **The continuation is only called when the first future completes**
- The syntax of continuations is now uncertain since P0443 has been reduced
- P1054 adds `FutureContinuation`, `SemiFuture`, `ContinuableFuture`

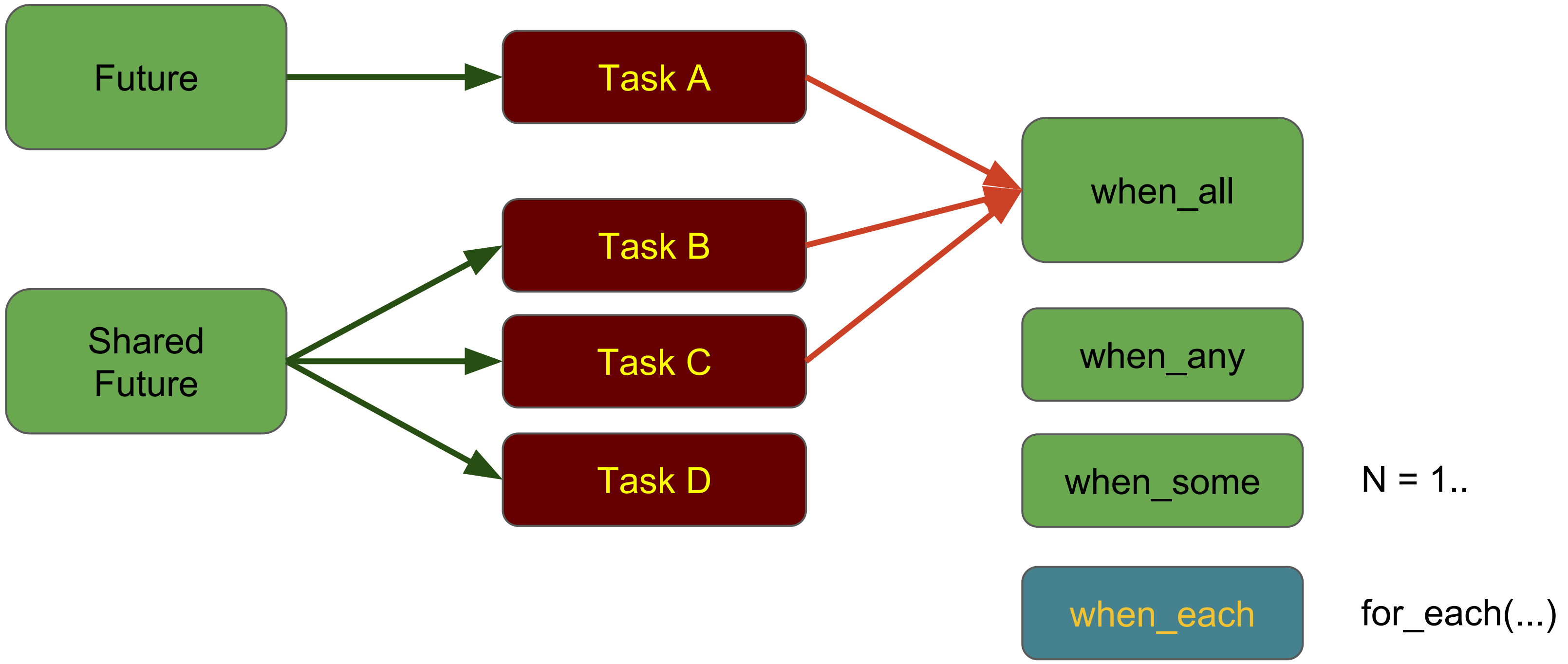
Compositional facilities and `shared_future<T>`

- Parallel composition of futures (see N3857)

```
void test_when_all() {
    shared_future<int> shared_future1 = async([]() -> int { return 125; });
    future<string> future2 = async([]() -> string { return string("hi"); });

    future<tuple<shared_future<int>, future<string>>> all_f =
        when_all(shared_future1, future2); // also: when_any, when_some, etc.

    future<int> result = all_f.then(
        [](future<tuple<shared_future<int>, future<string>>> f) -> int {
            return do_work(f.get());
        });
}
```

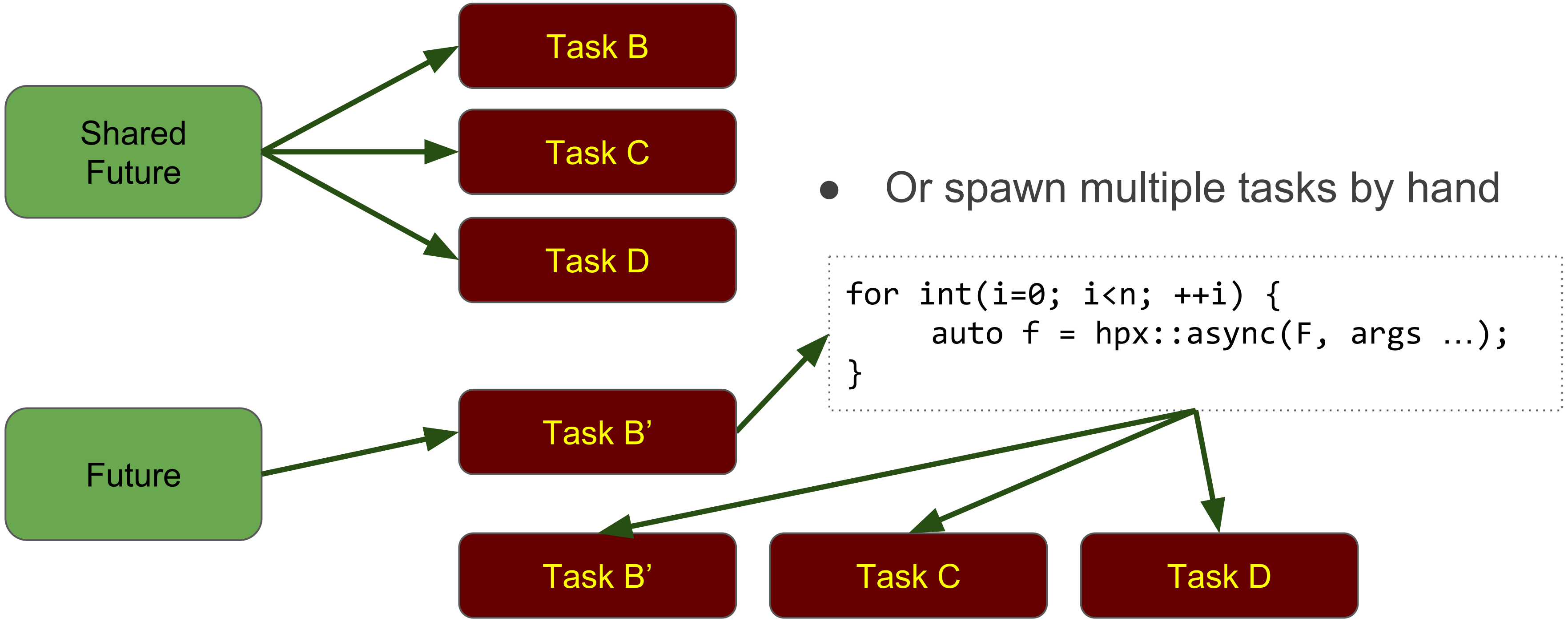
DAGs from `shared_/future<T>`

- Continuations `.then()` and `.when_xxx()` give us composability
- Futures are non-copyable, so `.then()` consumes its input



DAGs from `shared_/future<T>`

- Spawn multiple tasks from a continuation using a shared future



- Or spawn multiple tasks by hand

```
for int(i=0; i<n; ++i) {
     auto f = hpx::async(F, args ...);
}
```

- It depends where you need the `future` to be

Extending async: dataflow

- What if one or more arguments to ‘async’ are futures themselves?
- Normal behavior: pass futures through to function
- Extended behavior: wait for futures to become ready before invoking the function:

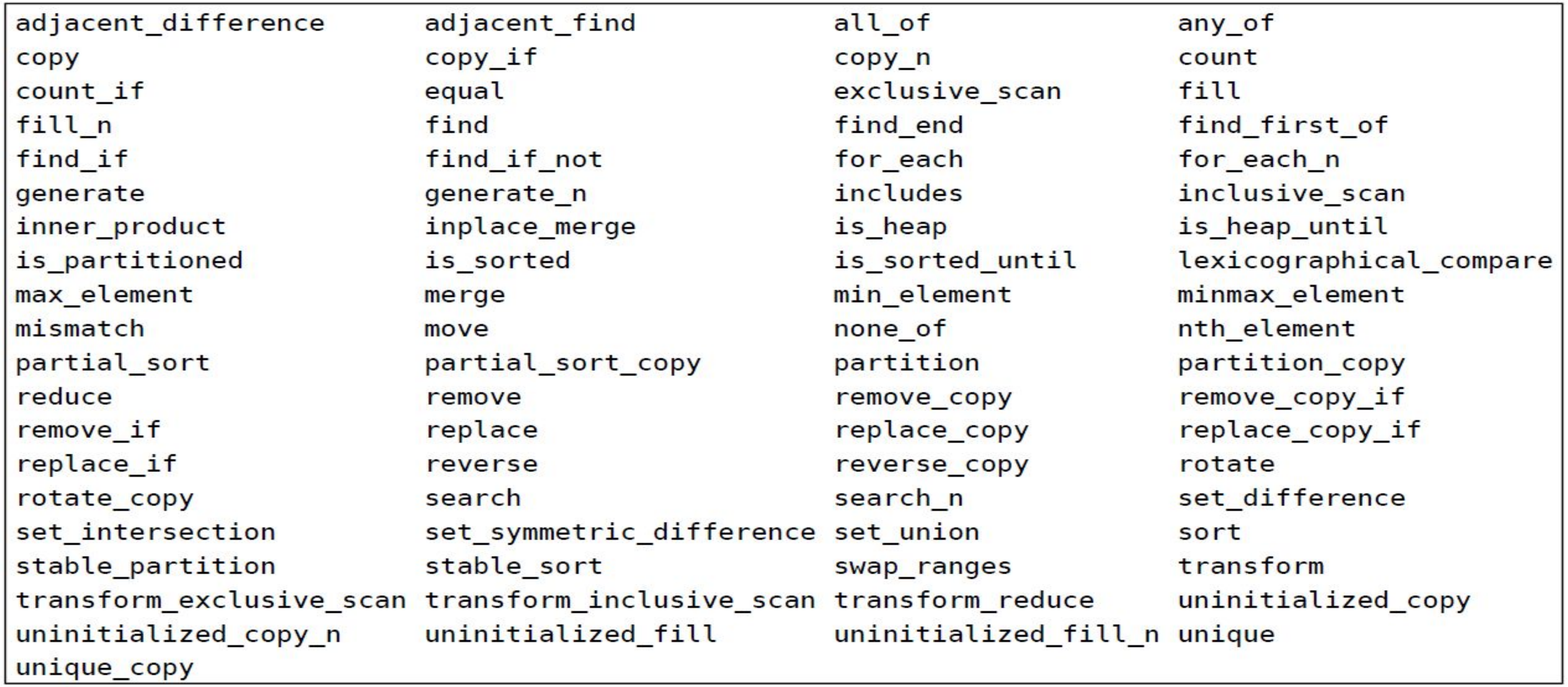
```
template <typename F, typename… Arg>
future<typename result_of<F(Args…)>::type> dataflow(F&& f, Arg&&… arg);
```

- If ArgN is a future, then the invocation of F will be delayed
- Non-future arguments are passed through
- `when_all(...).then(...)` can be replaced by `dataflow(...)`
  - Saves 1 future (similar to P1194 change to continuations)
- Dataflow has become a very useful tool in the hpx toolbox

HPX async/actions summary / extensions

R f(p...)	Synchronous (return R)	Asynchronous (return future<R>)	Fire & Forget (return void)
Functions (direct)	`f(p…)` C++	`async(f, p…)`	`apply(f, p…)`
Functions (lazy)	`bind(f, p…)(…)`	`async(bind(f, p…), …)` C++ Library	`apply(bind(f, p…), …)`
Actions (direct)	`HPX_ACTION(f, a)` `a(id, p…)`	`HPX_ACTION(f, a)` `async(a, id, p…)`	`HPX_ACTION(f, a)` `apply(a, id, p…)`
Actions (lazy)	`HPX_ACTION(f, a)` `bind(a, id, p…)(…)`	`HPX_ACTION(f, a)` `async(bind(a, id, p…), …)`	`HPX_ACTION(f, a)` `apply(bind(a, id, p…), …)` HPX

Parallel algorithms



adjacent_difference	adjacent_find	all_of	any_of
copy	copy_if	copy_n	count
count_if	equal	exclusive_scan	fill
fill_n	find	find_end	find_first_of
find_if	find_if_not	for_each	for_each_n
generate	generate_n	includes	inclusive_scan
inner_product	inplace_merge	is_heap	is_heap_until
is_partitioned	is_sorted	is_sorted_until	lexicographical_compare
max_element	merge	min_element	minmax_element
mismatch	move	none_of	nth_element
partial_sort	partial_sort_copy	partition	partition_copy
reduce	remove	remove_copy	remove_copy_if
remove_if	replace	replace_copy	replace_copy_if
replace_if	reverse	reverse_copy	rotate
rotate_copy	search	search_n	set_difference
set_intersection	set_symmetric_difference	set_union	sort
stable_partition	stable_sort	swap_ranges	transform
transform_exclusive_scan	transform_inclusive_scan	transform_reduce	uninitialized_copy
uninitialized_copy_n	uninitialized_fill	uninitialized_fill_n	unique
unique_copy			

- reduce_by_key, sort_by_key

Parallel algorithms

- Similar to standard library facilities known for years
  - Add execution policy as first argument
- Execution policies have associated default executor + parameters
  - `par` → `parallel_execution_policy` parallel executor, static chunk size
  - `seq` → `parallel_execution_policy` sequential executor, no chunking
  - `datapar / par_unseq` → vectorized/SIMD

    (Requires external library: currently Vc in N3759 + many more)

```
// Simplest case: parallel execution policy
std::vector<double> d(1000);
parallel::fill(
    par,
    begin(d), end(d), 0.0);
```

Execution Policies (HPX Extensions)

- Extensions: asynchronous execution policies
  - parallel_task_execution_policy (asynchronous), generated with
    `par(task)`
  - sequenced_task_execution_policy (asynchronous), generated with
    `seq(task)`
  - parallel_unsequenced_task_policy (asynchronous), generated with
    `datapar(task) / par_unseq(task)`

- In all cases the formerly synchronous functions return a future<>
- Instruct the parallel construct to be executed asynchronously
- Allows integration with asynchronous control flow

Parallel algorithms : asynchronous

```
auto f = parallel::for_loop(par(task), begin(d), end(d),
     [](iterator it){ … });
auto f = parallel::for_loop(par(task), begin(c), end(c),
     parallel::reduction_plus(sum),
     [](iterator it, std::size_t& sum) {sum += *it;});
auto f = parallel::for_loop(par(task), begin(c), end(c),
     hpx::parallel::induction(0), hpx::parallel::induction(0,2),
     [&d](iterator it, std::size_t i, std::size_t j)
          { *it = 42; d[i] = 42; TEST_EQ(2*i, j);});
auto f = parallel::for_loop_n(par(task), begin(c), c.size(),
     [](iterator it) { … });
auto f = parallel::for_loop_strided(
     par(task), begin(c), end(c), stride, [](iterator it) { … });
```

Executors

- Executors must implement one function: `async_execute(F && f)`
- Invocation of executors happens through executor_traits which exposes (emulates) additional functionality:

```
    executor_traits<my_executor_type>::execute(
            my_executor,
            [](...){ // perform task },
            ...);
```

- Four modes of invocation: single async, single sync, bulk async and bulk sync
- The async calls return a future (two way executor)

Types of Executor

- `sequential_executor`, `parallel_executor`:
  - Default executors corresponding to par, seq
- `pool_executor`
  - Runs on a “named” thread pool (“default” executor in hpx)
- `this_thread_executor`
  - Runs on the current thread
- `distribution_policy_executor`
  - Use one of HPX’s (distributed) distribution policies, specify node(s)
- `cuda::default_executor`
  - Use for running things on GPU
- `host::parallel_executor`
  - Specify core(s) to run on (NUMA aware) (to be deprecated?)

Executor parameters (HPX)

- Same scheme as for executor/executor_traits:
    - parameter/executor_parameter_traits
- Various execution parameters, possibly executor specific
- For instance:
    - Allow control of grain size of work
        - i.e. amount of iterations of a parallel `for_each` run on each thread
        - Similar to OpenMP scheduling policies: static, guided, dynamic
            - `auto_chunk_size`, `static_chunk_size`, `dynamic_chunk_size`
        - Used by parallel algorithms to adjust chunk size
    - Stack size to use for tasks
    - Pool name
- Numerous other possibilities for affinity, prefetching, control

Executor rebinding

```
// Simplest case: parallel execution policy
std::vector<double> d(1000);
parallel::fill(
     par,
     begin(d), end(d), 0.0);

// rebind execution policy
//   .on():    executor object, ‘where and when’
//   .with():  parameter object(s), possibly executor specific parameters
std::vector<double> d(1000);
parallel::fill(
     par.on(exec).with(par1, par2, ...),
     begin(d), end(d), 0.0);
```

Executor rebinding

```
// uses default execution policy: par
std::vector<double> d = { ... };
parallel::fill(par, begin(d), end(d), 0.0);

// rebind par to user-defined executor
my_executor my_exec = ...;
parallel::fill(par.on(my_exec), begin(d), end(d), 0.0);

// rebind par to user-defined executor and user defined executor parameters
my_params my_par = ...
parallel::fill(par.on(my_exec).with(my_par), begin(d), end(d), 0.0);
```

- Algorithms use traits to determine policy specifics
- Executor params may balloon : Requires xxx

Executor rebinding

```
hpx::threads::executors::pool_executor
     pool_exec("pool");
hpx::parallel::execution::guided_chunk_size
     gcs(100);

hpx::parallel::transform(
     hpx::parallel::execution::par(hpx::parallel::execution::task)
          .on(pool_exec)
          .with(gcs), ...);
```

- `guided/dynamic/static/auto` chunk sizes can be used

Summary - Where, How, When

- Where
  - The executor controls placement of a task
- How
  - Execution policy and executor parameters controls how a task is to be executed (sequential, parallel, chunk size, …)
- When
  - The user controls when explicitly using DAG dependencies
  - … and implicitly using priorities/parameters in executor

Different *types* of future

- Current status

```
    new_future = old_future.then(executor, ....)
```

- Proposed

```
    new_future = executor.then(old_future)
```

- Where new and old futures can be different types of future

```
    future<T>, cuda_future<T>, mpi_future<T>
```

- The responsibility shifts from future to executor

```
    cuda_future = cuda_exec.then(cuda_result, new_task,...)
```

Real world use

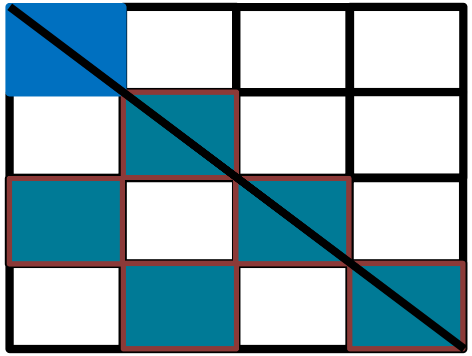
An example of use in Linear Algebra

- Cholesky decomposition used widely in HPC
  - Forms backbone of solver of linear systems

- Frequently see very large matrices
  - Tiled representation used for memory access efficiency
    - and convenience/compatibility with BLAS/LAPACK routines

  - Tiles arranged in block cyclic fashion on a node

  - Matrices distributed across ranks using MPI

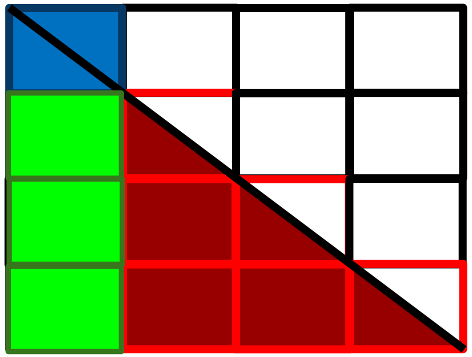


Cholesky Decomposition : patterns

- Iterate over each column
  - Factorize the diagonal
  - Update the sub-panel (column)
  - Rank update the submatrix

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T = \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} L_{11} & L_{21} & L_{31} \\ 0 & L_{22} & L_{32} \\ 0 & 0 & L_{33} \end{pmatrix}$$

$$= \begin{pmatrix} L_{11}^2 & & (\text{symmetric}) \\ L_{21}L_{11} & L_{21}^2 + L_{22}^2 & \\ L_{31}L_{11} & L_{31}L_{21} + L_{32}L_{22} & L_{31}^2 + L_{32}^2 + L_{33}^2 \end{pmatrix},$$



$$\mathbf{L} = \begin{pmatrix} \sqrt{A_{11}} & 0 & 0 \\ A_{21}/L_{11} & \sqrt{A_{22} - L_{21}^2} & 0 \\ A_{31}/L_{11} & \left(A_{32} - L_{31}L_{21}\right)/L_{22} & \sqrt{A_{33} - L_{31}^2 - L_{32}^2} \end{pmatrix}$$

Image copyright wikipedia

Cholesky with communication

1) Factorize diagonal block,

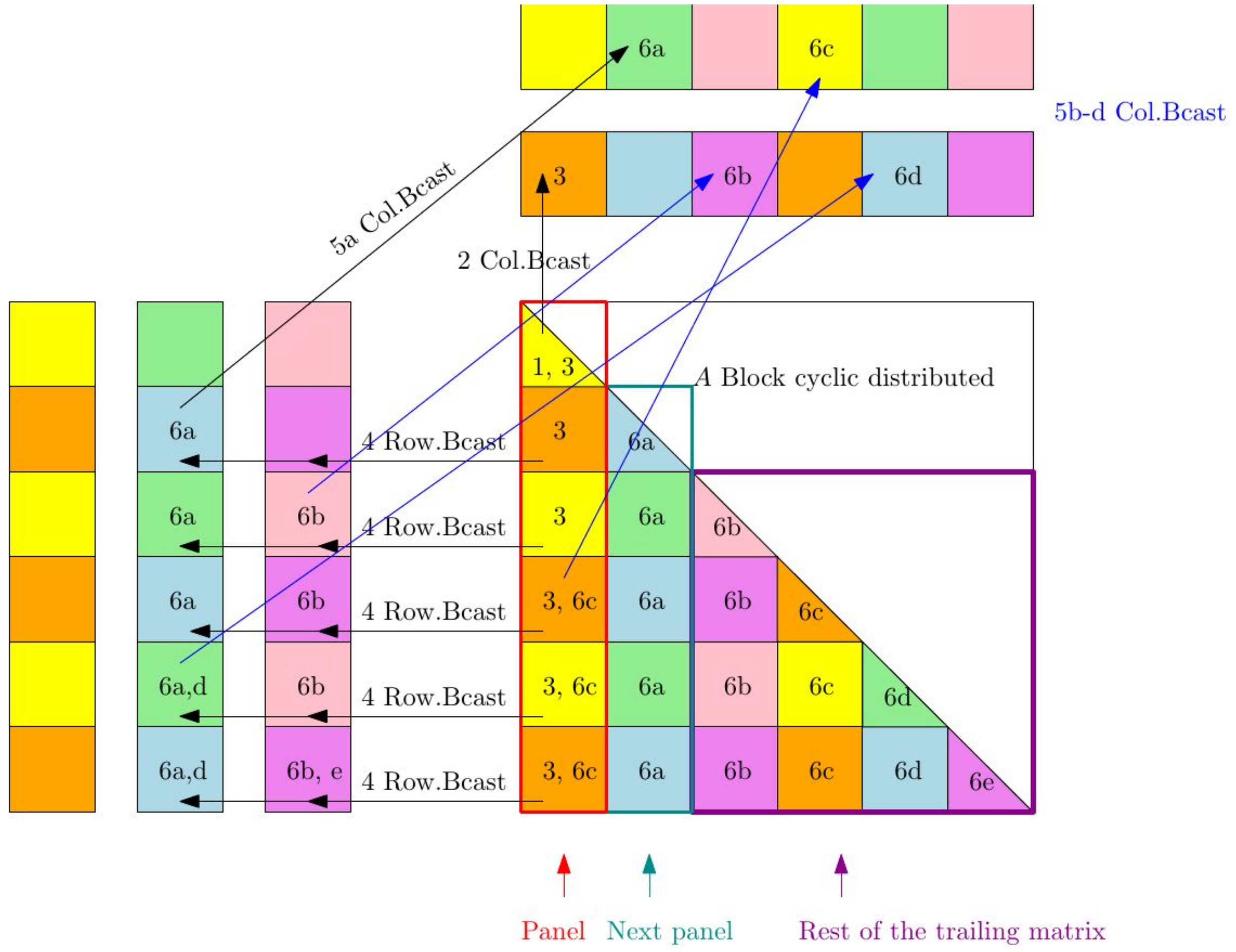
2) Columnwise **broadcast** diagonal block,

3) Triangular solve local part of panel with the diagonal block,

4) Rowwise **broadcast** the panel,

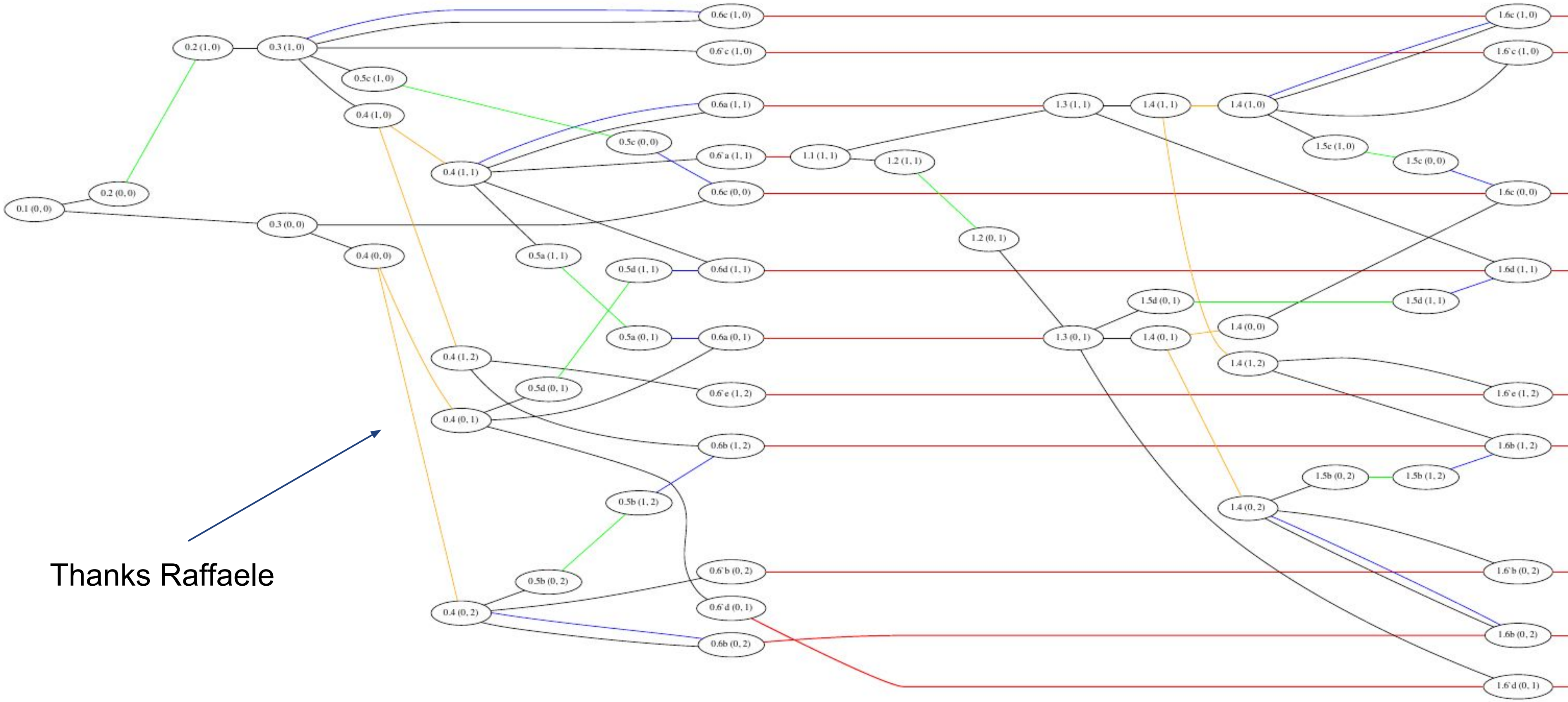
5) Columnwise **broadcast** some part of the panel,

6) Update trailing matrix.



CSCS

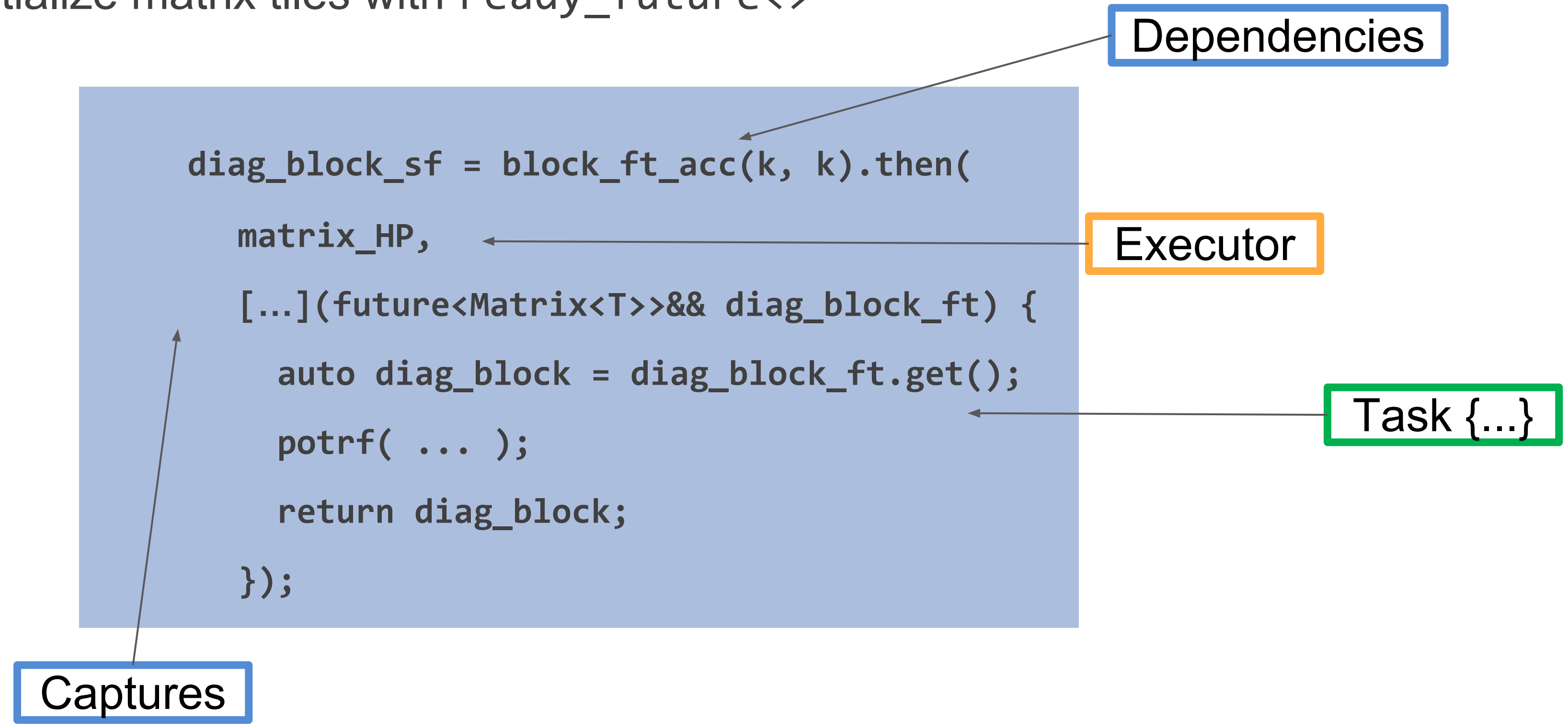
Decomposition forms a DAG of operations



Thanks Raffaele

Cholesky : Designing software with CPS

- Initialize matrix tiles with `ready_future<>`



```
diag_block_sf = block_ft_acc(k, k).then(
  matrix_HP,
  [...](future<Matrix<T>>&& diag_block_ft) {
    auto diag_block = diag_block_ft.get();
    potrf( ... );
    return diag_block;
  });
```

Trigger N tasks from diagonal

```
for (int i = k + nb; i < n; i += nb) {
  panel_sf[i] = hpx::dataflow(
    matrix_HP,
    [...](shared_future<Matrix<T>> &&diag_block_sf,
    future<Matrix<T>> &&panel_ft)
    {
      auto panel = panel_ft.get();
      const auto& diag_block = diag_block_sf.get();
      trsm( ... );
      return panel;
    }, ... );
}
```

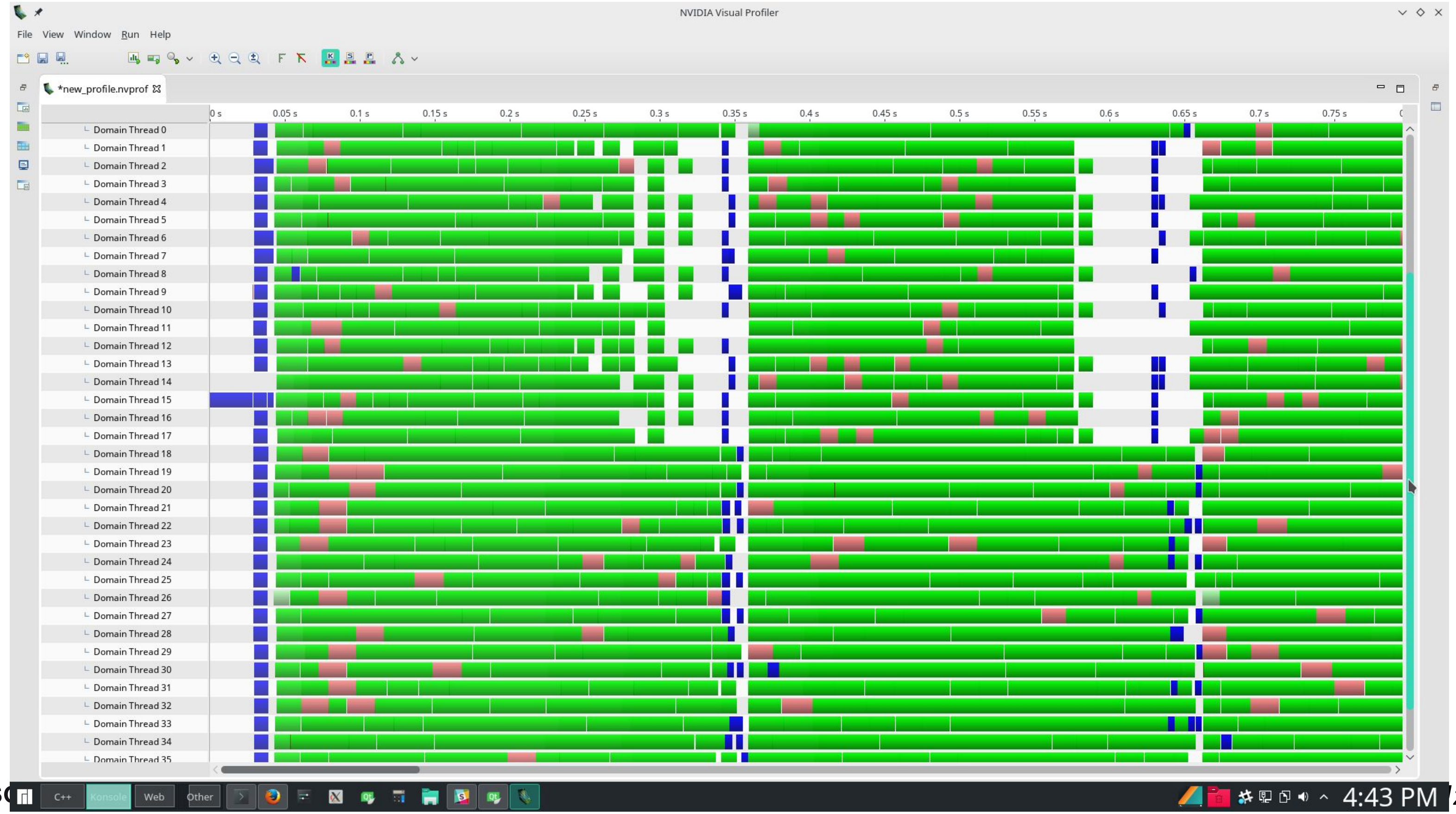
- Note the use of `f = f.then();`
- Replace current tile future with new one

Broadcast panel along row

```
if (panel_block.is_mine()) {
  // broadcast to all ranks receiving on the row
  comm2D_ft = hpx::dataflow(
    mpi_executor,
    [...](future<CommType> &&comm2D_ft,
    shared_future<Matrix<T>> &&panel_sf) {
      const auto& panel = panel_sf.get();
      auto comm2D = comm2D_ft.get();
      comm2D->rowBcast( ... );
      return comm2D;
    }, ...);
  // else receive a block broadcast
```

- Special executor for MPI tasks
- This goes onto it's own pool so there is no interference from any other work
- When you have 36 cores, losing one is not a big deal

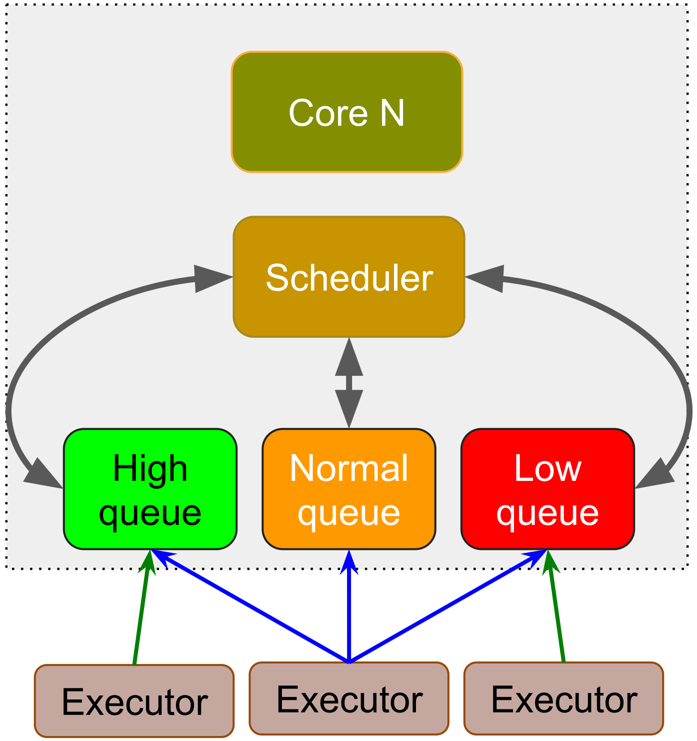
Task priorities and stealing not quite right



Schedulers, Executors + Priority

- Multiple schedulers exist
  - Stealing strategies (local queues first?)
  - Numa awareness
  - Number of queues

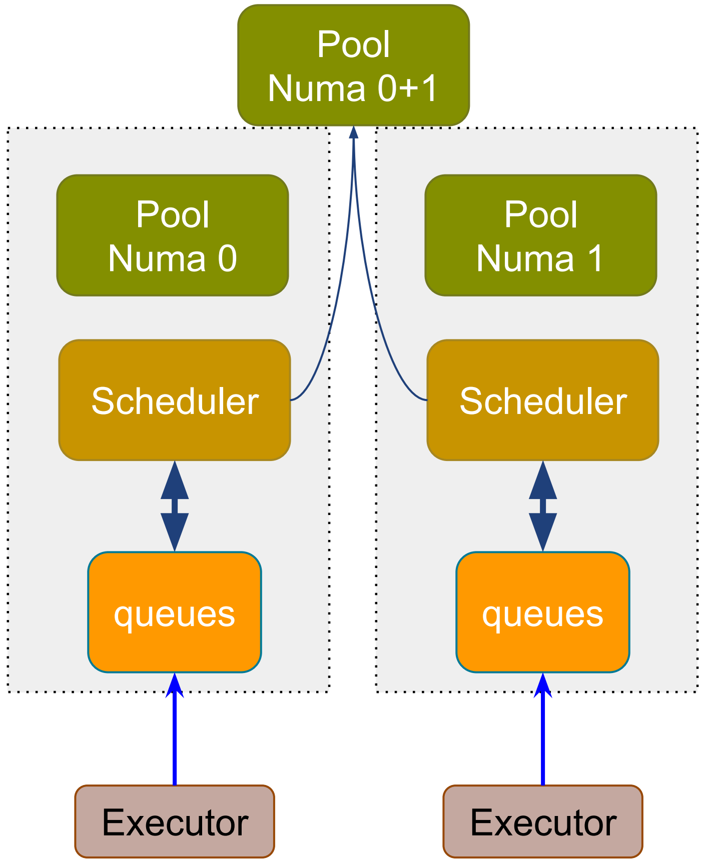
```
matrix_HP = pool_executor("default",
   hpx::threads::thread_priority_high);

matrix_normal = pool_executor("default",
   hpx::threads::thread_priority_normal);

mpi_executor = pool_executor("mpi");
```



Pool and Executor decisions

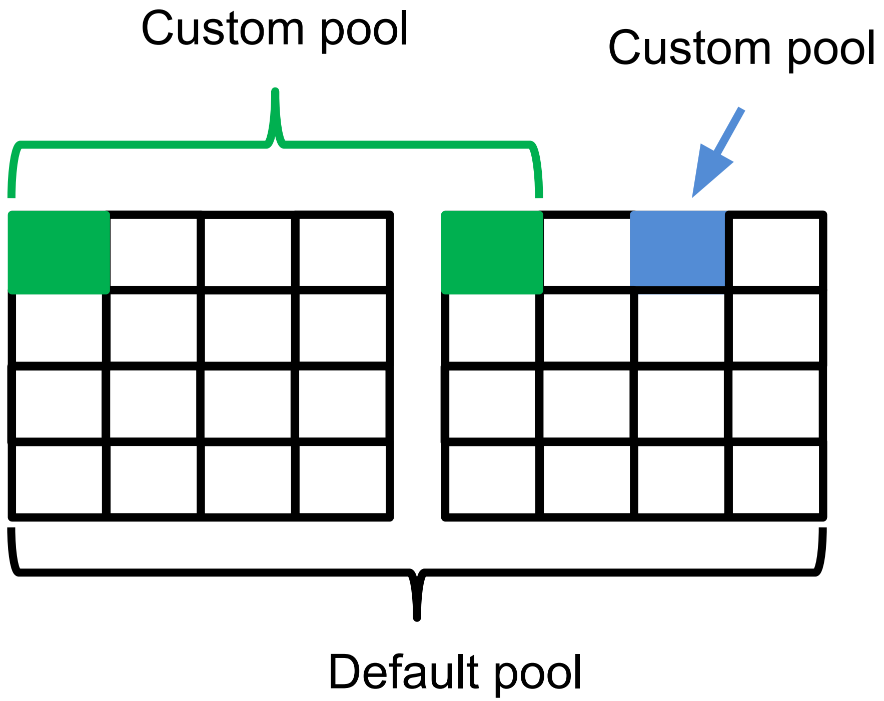
- Create 1 pool on each numa domains
    - No stealing across domains
    - = MPI + OpenMP = good
- Create 1 pool across numa domains
    - Potential x-numa traffic
    - Might reduce performance

- 1 Pool requires much more work from programmer (c.f. MPI messaging)
    - Have to decide at compile time which executor to use
    - Child tasks spawned on parent executor



Topology / Resources / Thread pools

- Topology class wraps hwloc
  - NUMA domains
  - Sockets / Cores PU
  - (GPUs/NICs)
- Resource Partitioner
  - Iterators for access
  - Every Nth PU on Mth domain
  - Create thread pool
- Planned Pool Features
  - Oversubscription
  - Auto sleep?



Resource partitioner pool creation

- Create pools by traversing topology
- Interfaces to hwloc to create `hpx::topology` object
- Needs to be extended for GPUs/NICs/etc

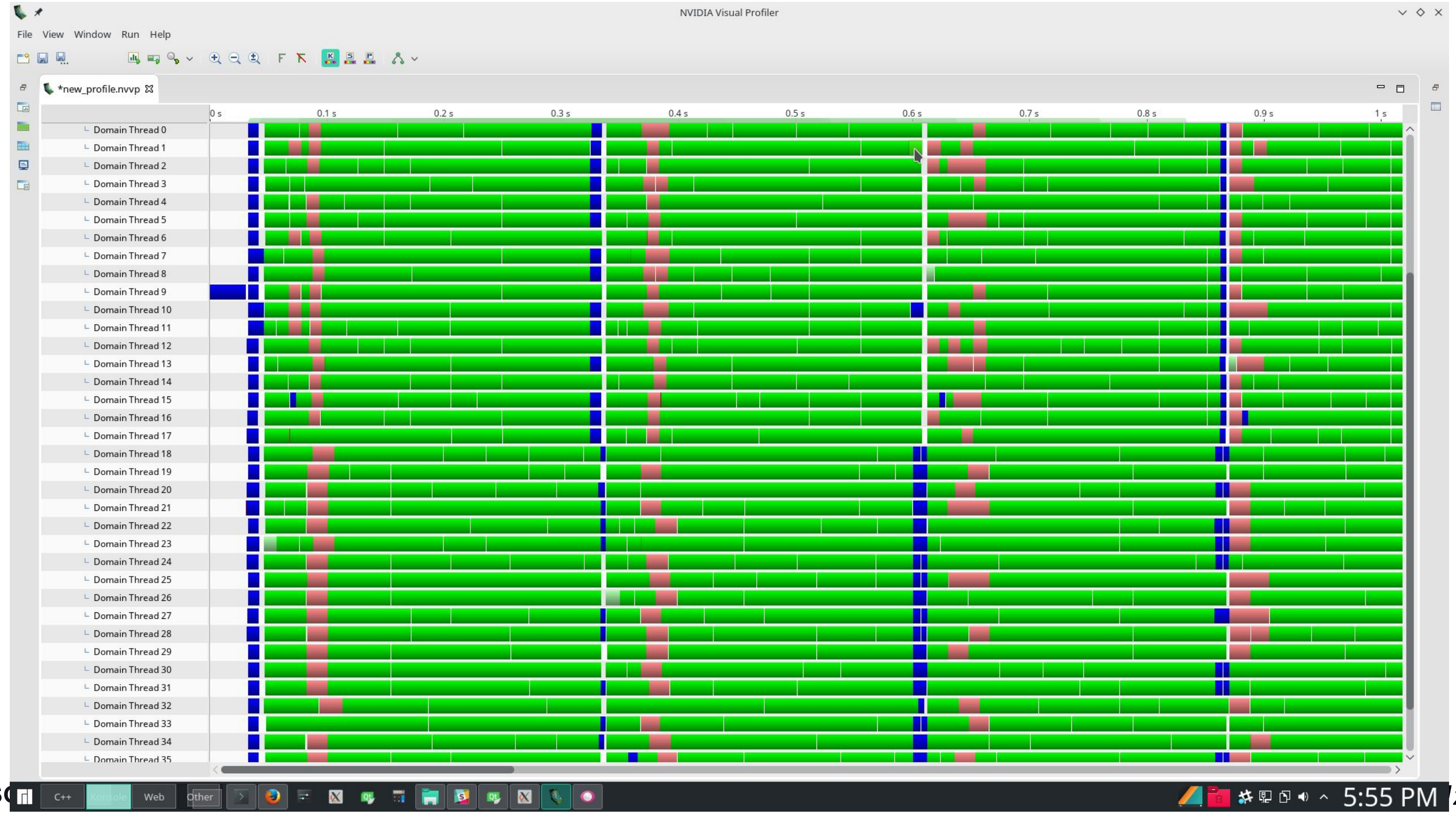
```
// add N cores to mpi pool
int count = 0;
for (const hpx::resource::numa_domain& d : rp.numa_domains()) {
     for (const hpx::resource::core& c : d.cores()) {
          for (const hpx::resource::pu& p : c.pus()) {
               if (count < mpi_threads) {
                    std::cout << "Added pu " << count++ << " to mpi pool\n";
                    rp.add_resource(p, "mpi");
               }
          }
     }
}
```

Note hwloc 2 changes numa parent/child relationship

Custom pools

- For MPI communication
  - A separate pool to handle send/recv/etc
- For GPU management
  - Manage stream events on a dedicated pool
- For anything that requires isolation
  - GUI threads (Qt)
  - Real time sensor data
  - IO stream activity (or anything with blocking system calls)
  - The ***C++ community*** is much bigger than the HPC one
- Numa domains (todo)
  - Numa allocator needs one (or more) tasks per numa domain
  - No stealing across numa domains (between pools)
  - Want a pool that mostly sleeps but wakes when given work

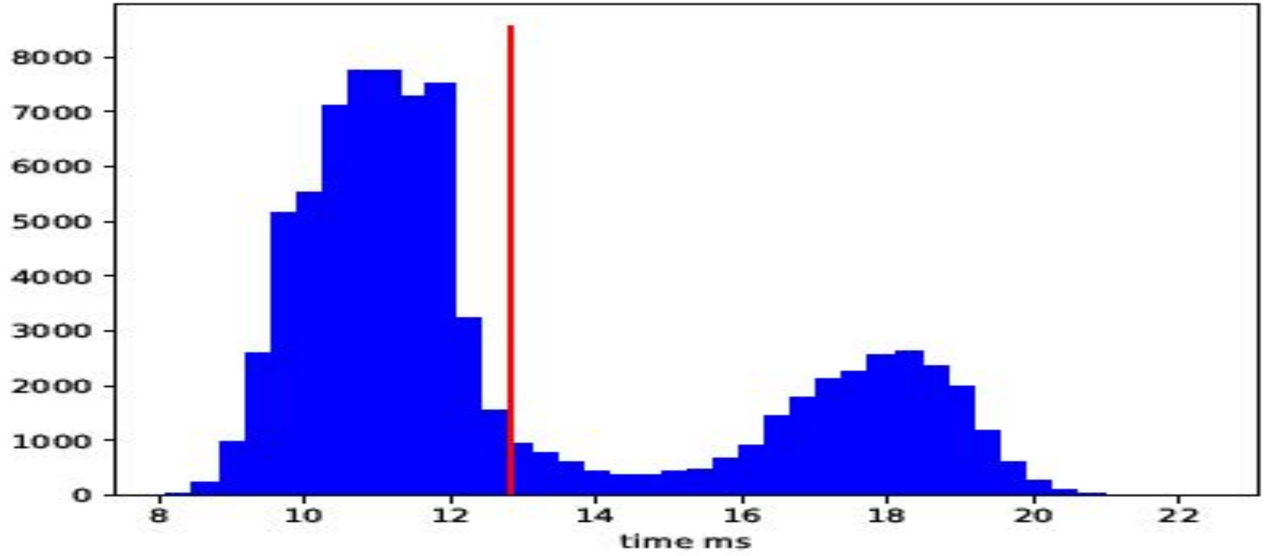
Cleaning up the task stealing



Bad use of memory channels

- Matrix allocated and initialized on thread
- Only using memory from domain 0
  - default policy is *first-touch*
- X-numa traffic quite visible on socket 1

- Allocators!



Numa `allocator<T>` and placement

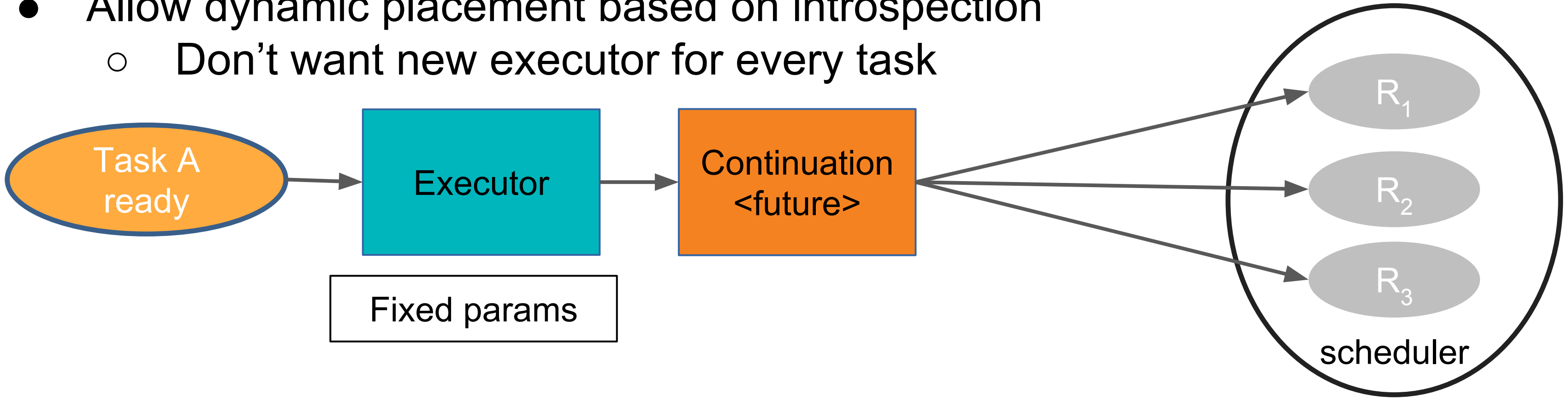
- Use `hwloc` memory allocation
  - mem_policy *interleaved*
  - pages alternate domain
  - use both memory channels
  - huge speed improvement
  - but for N>32 tile>page
    - column major order for our matrices
- Assign pages using a pattern
  - mem_policy *bind*
    - used up all the OS virtual memory page tables
  - mem_policy *first-touch*
    - launch 1 task per domain - ‘touch’ memory according to pattern

```
tiles/domain : 1
0101010101010101
1010101010101010
0101010101010101
1010101010101010
0101010101010101
1010101010101010
0101010101010101
1010101010101010
................
```

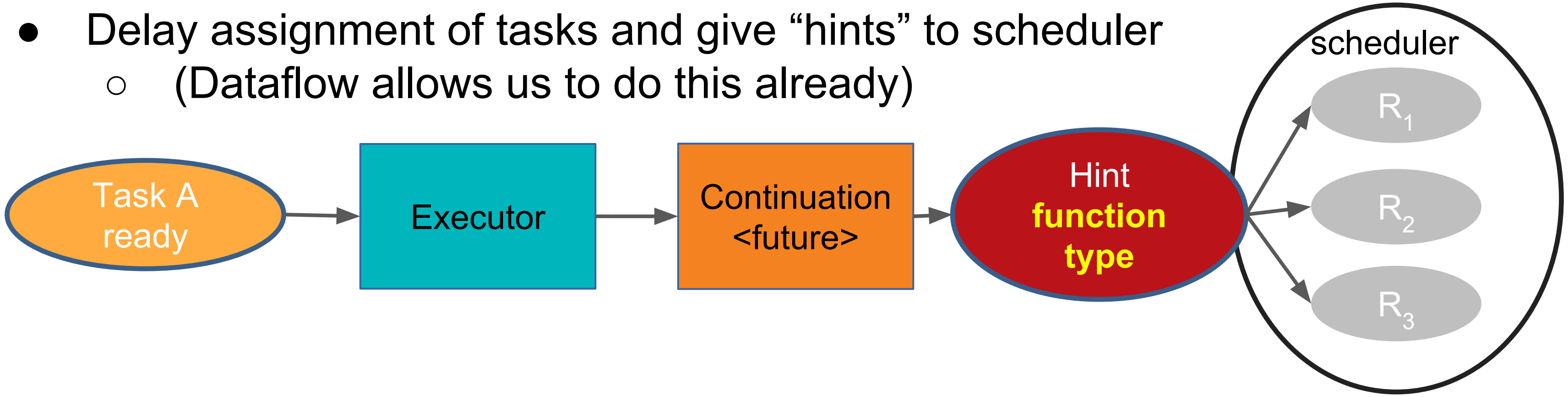
```
tiles/domain : 4
0000111100001111
0000111100001111
0000111100001111
0000111100001111
1111000011110000
1111000011110000
1111000011110000
1111000011110000
................
```

Need an executor can (dynamically) introspect tasks

- Allow dynamic placement based on introspection
  - Don't want new executor for every task



- Delay assignment of tasks and give "hints" to scheduler
  - (Dataflow allows us to do this already)



Executor hints that are passed to scheduler

- Provide a callback function signature
  - Executor calls it before scheduling
  - Use it on any algorithm –not just cholesky
  - Type safe compiler support

```
struct hint_type { ...
  template <typename M>
  int operator()(M arg1, M arg2, M arg3) const {
    return get_numa(args);
  } // as many more overloads as you want
}
...
guided_pool_executor<hint_type> matrix_HP(...);
```

Numa hints for matrices

```
template <> struct pool_numa_hint<cholesky_tag> {
    int operator()(const MatrixType& matrix) const {
        int dom = matrix.get_numa();
        return dom;
    }
    int operator()(const MatrixType& matrix1, const MatrixType& matrix2) const {
        int dom = matrix2.get_numa();
        return dom;
    }
    int operator()(const MatrixType& matrix1, const MatrixType& matrix2,
                   const MatrixType& matrix3) const {
        return matrix3.get_numa();
    }
};
```

Guided executor is prototype for

- A way to decorate executors with scheduler information
  - Schedulers may have arbitrary params to tweak
  - They are (may be) non standardized

- Currently, executor proposals don’t handle shedulers
  - Executor traits/parameters can be used to get certain functionalities

- guided_pool_executor can be extended
  - allow types exposed by schedulers to be passed back through executors
  - user code can dynamically generate

```
scheduler::hint_type
```

Side note 1: Flow control with sliding_semaphore

With N iterations 'in flight' at a time, scheduler queues get very full

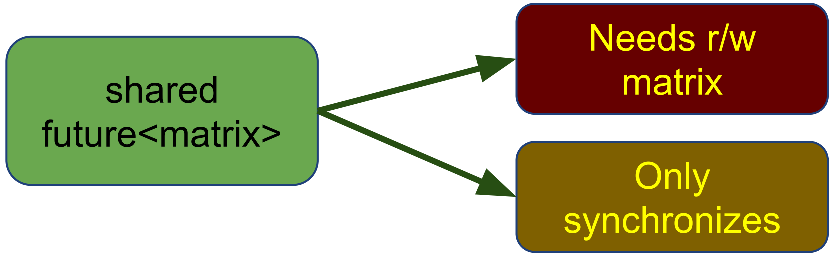
```
hpx::lcos::local::sliding_semaphore sem(N);
for (int j=0; j<max_cols) {
    int f_index1 = complex_indexing_scheme1(i,j);
    int f_index2 = complex_indexing_scheme2(i,j);
    future[f_index1] = future[f_index2].then(hpx::launch::sync,
        [](auto &&f){
            if (j==max_cols-1) sem.signal(i);
            return temp;
        }
    );
}
sem.wait(i);
```

CSCS

ETHzürich

Side note 2: Split future

- Cholesky DAG contains synchronization points and data flow



- Shared future passes const ref to data
    - Complicates code unnecessarily
    - Return a pair <bool,matrix> instead

```
split_future<future<pair<T1, T2>> -> pair<future<T1>, future<T2>>
```

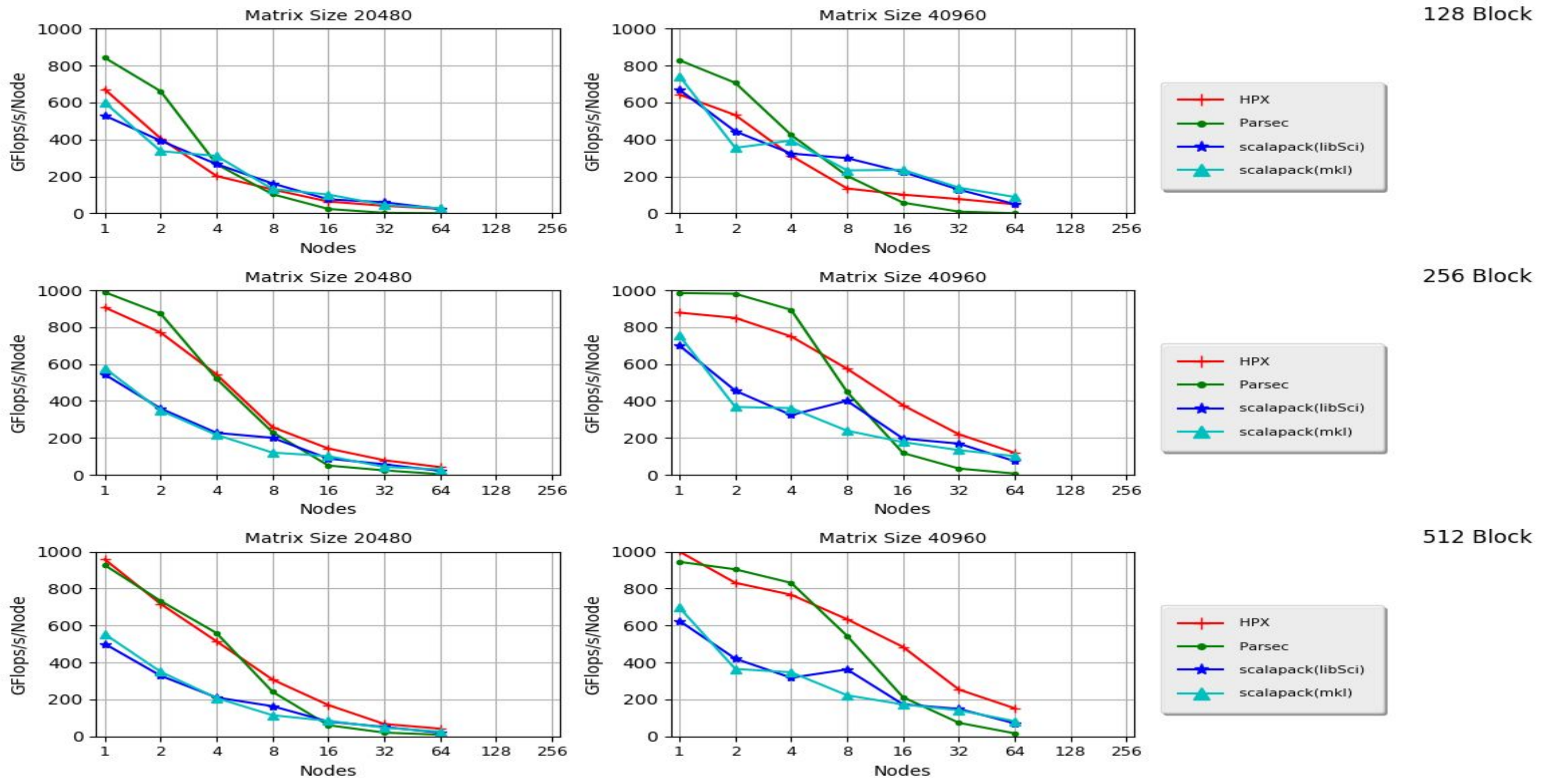
Final version recap

- Scheduler cleanup
  - Task priorities, stealing
  - Some primitive flow control
- MPI pool
  - Communication not in queue behind other tasks
- Custom allocator
  - For our numa placement of memory
- Specialized executor
  - for task placement with our numa allocated matrices

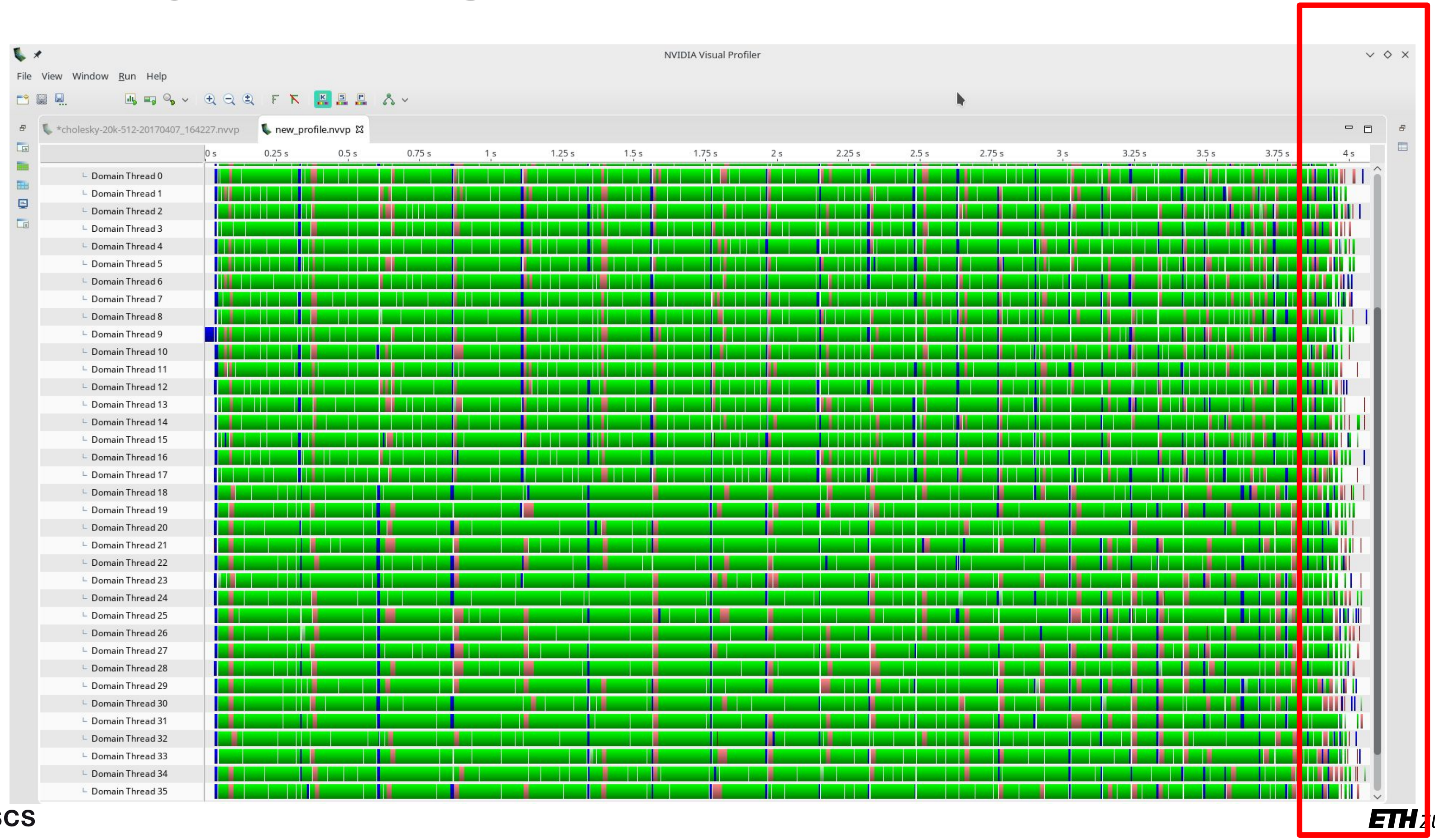
How does the algorithm perform overall?

Cholesky results HPX, Parsec, scalapack

- Parsec is 'gold standard'

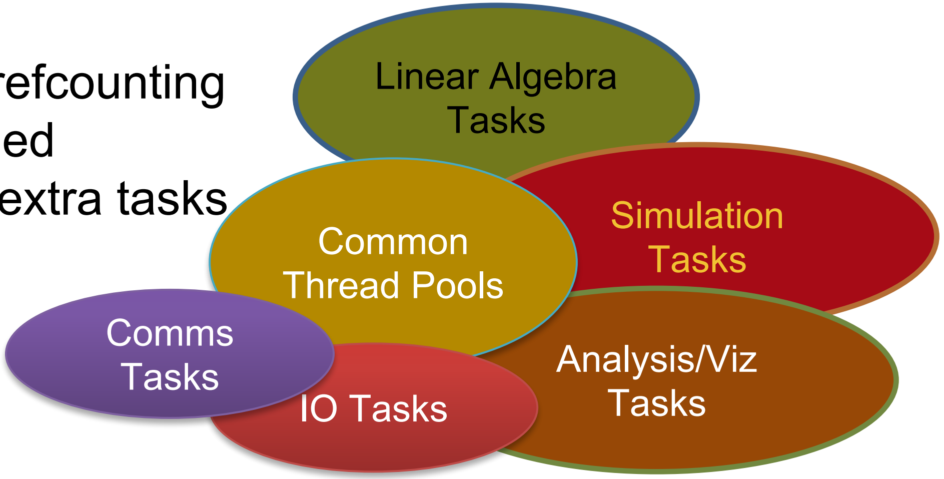


Efficiency dropping at the end



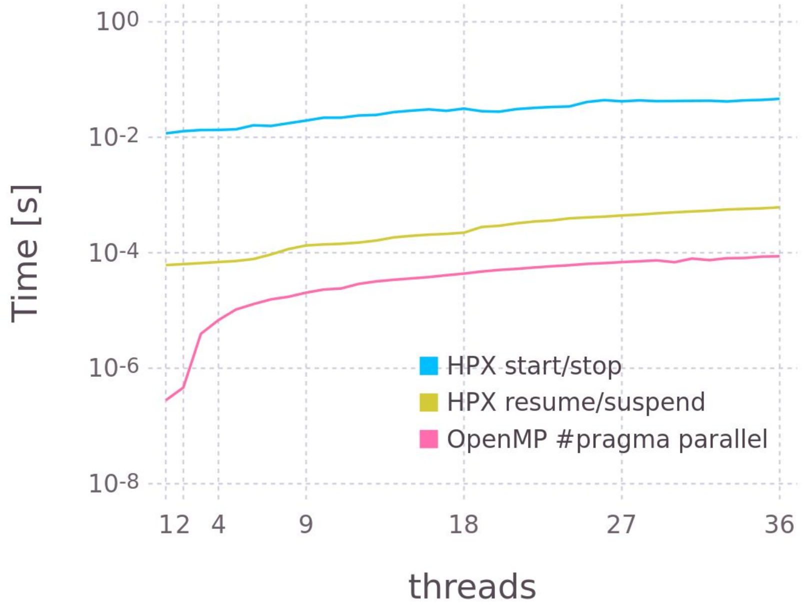
Opportunities with shared/common runtimes

- Linear algebra as an HPX library
  - Overlap matrix tasks with others
  - Start 2nd calculation before 1st completes
- Couple simulation/analysis/visualization
  - Don’t pause simulation whilst doing viz
    - (as most in-situ libs do)
  - Use shared data structures and refcounting
    - Delete when no longer needed
  - Maximize CPU/GPU usage with extra tasks
- IO
- Communication



Interoperability with non HPX code

- HPX runtime binds worker threads to cores
  - Other libraries also bind/create threads
- Combine HPX Application with 3rd party library
  - (or vice versa)
  - OpenMP : most common
  - plasma/parsec : runtime
  - OpenMP / TBB / other
- Things we *could* do
  - 1 socket HPX
  - 1 socket OpenMP
  - Dynamically reassign
  - … and lots more
  - Changes in setup ...



Work in progress

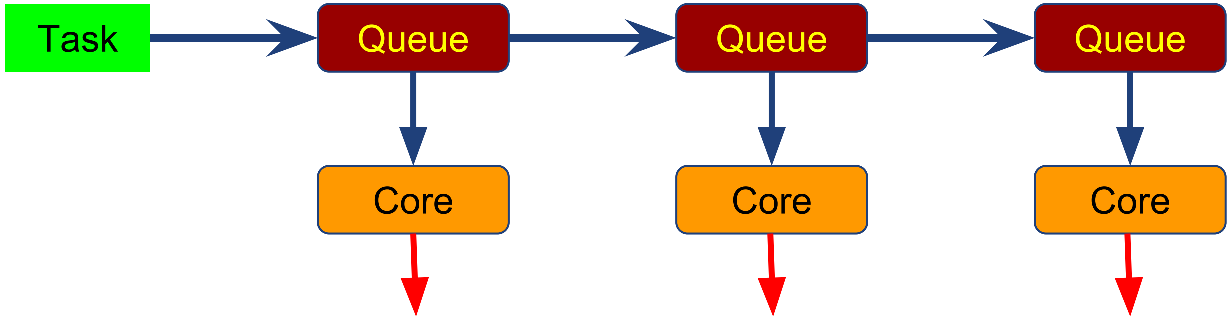
- What is stopping us using HPX for everything?

  Fine grained small tasks are current bottleneck

- Executor design
- Executor parameters
- Interaction with schedulers
- Affinity
- Topology
- Distributed collective algorithms
- Potential integration of other libraries (kokkos)

What is wrong with HPX?

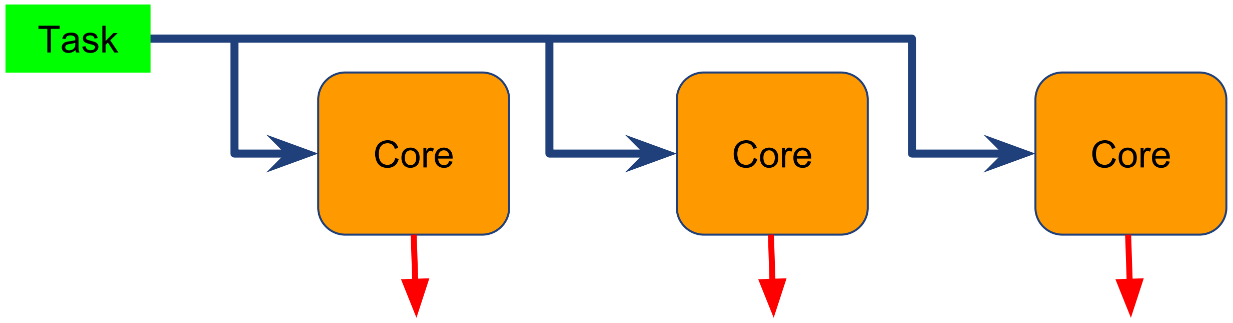
- Fine grained fork-join parallelism is too expensive
-
- Simple parallel::for_loop(chunk_size, N, ...)
    - Creates T=N/chunk_size tasks (or a multiple of them)
    - Puts them into scheduling queues
    - Allows the scheduler to take them and execute them on threads in pool
    - Returns a synchronization future for each of T
    - Waits on T futures (when fork-join)
    - (Cleans up tasks on completion of each)



What is right with OpenMP

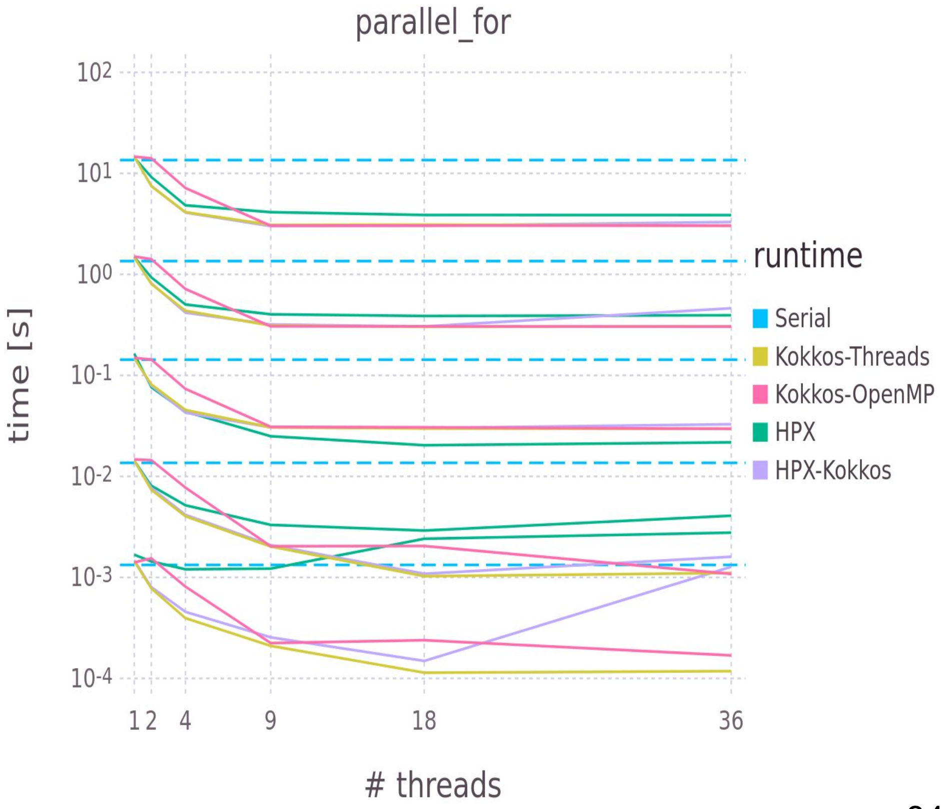
- Fine grained fork-join parallelism is reasonably cheap

- Simple parallel::for_loop(chunk_size, ...)
  - Passes function pointer to each thread
    - Threads begin work immediately
    - No stacks required
  - Wait for (join on) completion of all threads



kokkos: overheads comparison

- Low overheads of task launch in OpenMP approach
- HPX fine on larger workloads but falling behind on small ones
- Redesign of `parallel::for()` is planned
    - Bypass queues and task creation steps
    - Careful task managment



Task blocks (see P0155R0)

- Canonic fork-join parallelism of independent non-homogeneous code paths

```
template <typename Func>
int traverse(node const& n, Func compute) {
    int left = 0, right = 0;

    define_task_block(
        [&](auto& tb)
        {
            if (n.left)  tb.run([&] { left = traverse(*n.left, compute); });
            if (n.right) tb.run([&] { right = traverse(*n.right, compute); });
        });
    return compute(n) + left + right;
}
```

Task blocks (HPX)

- Added optional execution policy argument
  - Allows to make task block execute asynchronously

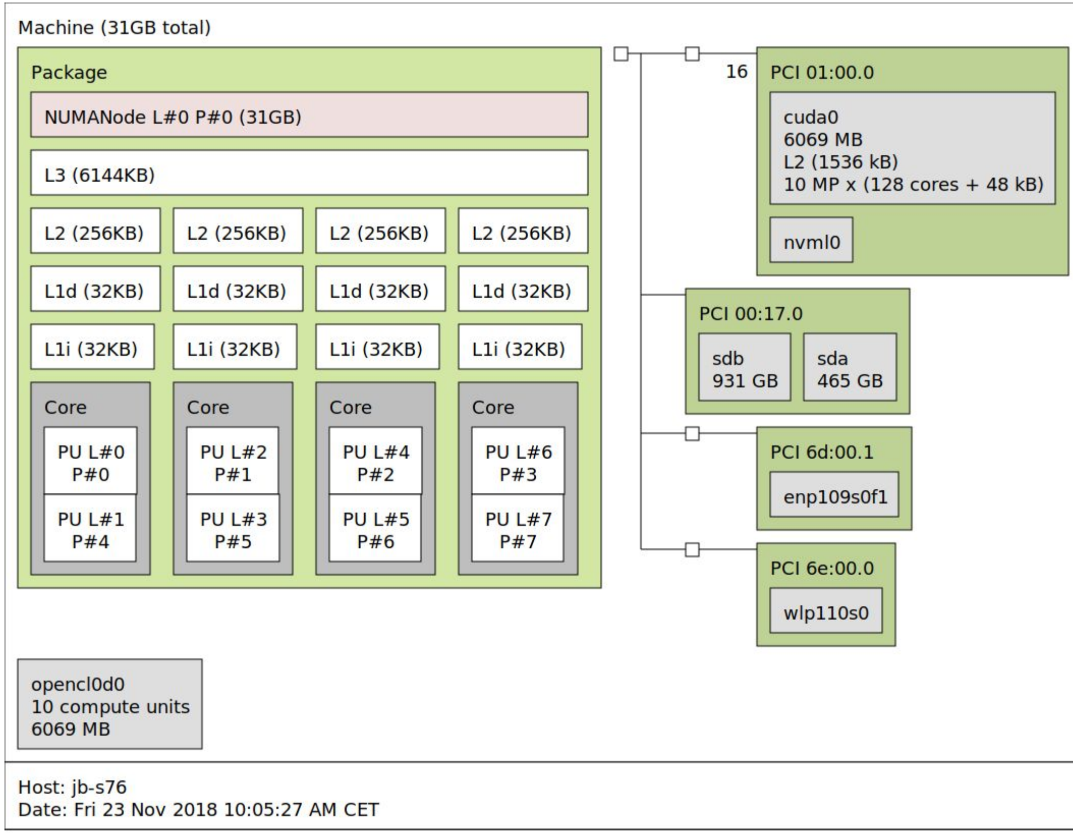
- Added optional executor argument to task_block::run()
  - Allows for fine control of execution of various tasks run inside the task block

```
define_task_block(
     policy,
     [&](auto& tb)
     {
          if (n.left)  tb.run(exec, [&] { left = traverse(*n.left, compute); });
          if (n.right) tb.run(exec, [&] { right = traverse(*n.right, compute);});
     });
```

Topology - Affinity - P0796

- Proposal for querying hardware to get information about locality/memory cache/ speeds/resources

- Integrate this with
  - Allocators
  - Executors
  - Schedulers
  - Thread pools
  - Accelerator use



cscs

Distributed HPX

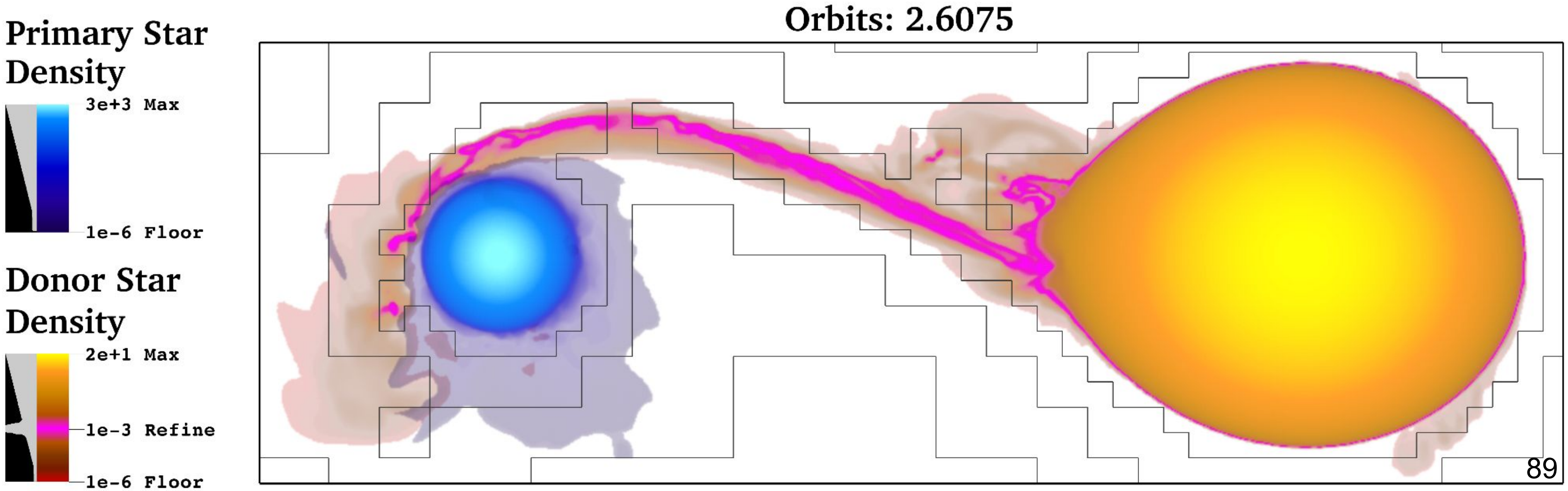
- Same API but use localities in async calls
  - Makes very easy transition from 1 node to N

- Not yet developed good communicators with collectives
  - Communicators could be a type of executor
    - Do this AllGather on communicator

- AGAS acts as distributed in memory key/value store
  - Handles to remote objects
  - Equivalent to `this->` pointers

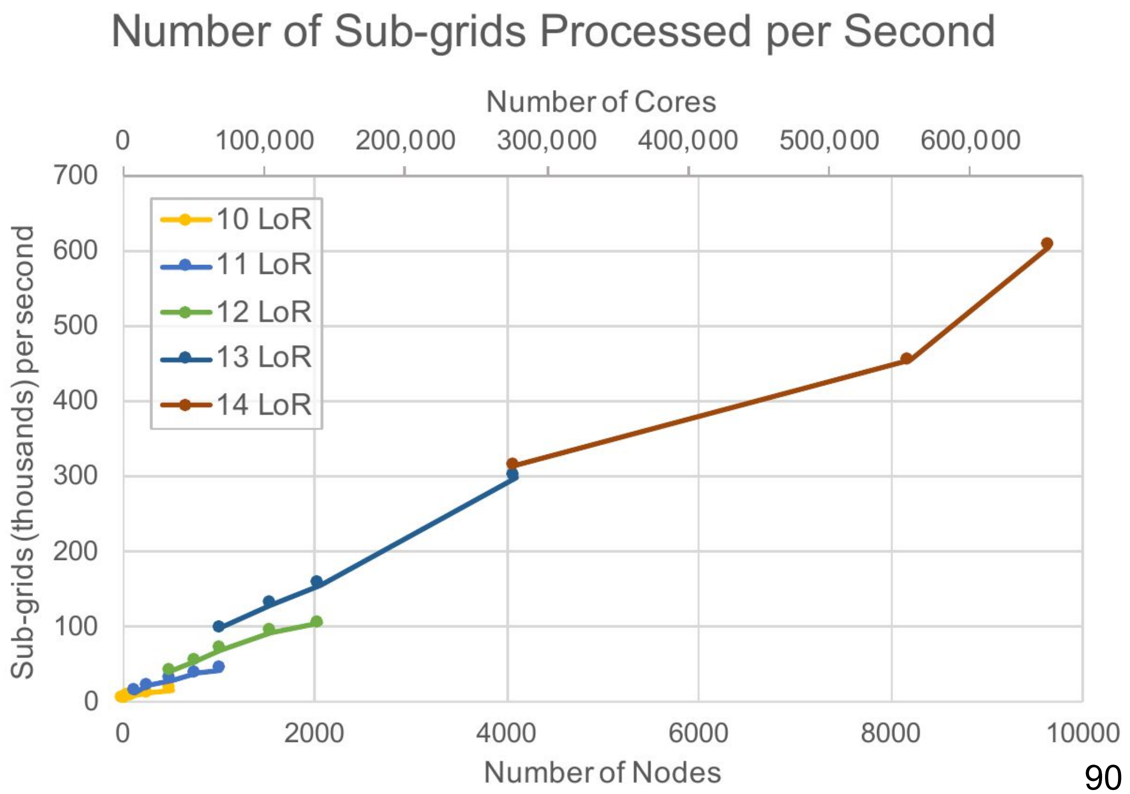
OctoTiger

- Astrophysics AMR code (for binary systems) using HPX on and across nodes
    - Developed at LSU
- Adaptive on the fly remeshing as material is transferred
- Scaled to whole of Cori machine at NERSC



Octotiger on 650,000 cores

- Mix of Strong and Weak scaling
- Levels of refinement adjusted as problem size increases
- Cannot run big jobs on small node counts



CSCS

Conclusion

- End of prepared slides

HPX Summary

- Giving the programmer a high level API
  - C++ programmers like it
- Allowing access to low level features
  - Task placement etc.
- High performance runtime
  - Still a few places that can be improved (small tasks)
- Future proof
  - Evolving with (and influencing) the C++ standard
  - Absorbing heterogeneous API developments (GPUs etc)

“The best library I’ve ever used, ~~that doesn’t work~~” (JB, 2015)
that‘s not bad at all (JB 2018)

HPX needs you and your ideas/help

- Please check

- Github
  - https://github.com/STEllAR-GROUP/hpx

- IRC
  - #ste||ar on Freenode

- Slack
  - HPX on cpplang.slack.com

- Mailing list
  - hpx-users@stellar.cct.lsu.edu

Spare

-