Plain Threads are the GOTO of Today's Computing

Plain Threads Considered Harmful

Hartmut Kaiser (Hartmut.Kaiser@gmail.com)

GOTO Considered Harmful

- Edsger Dijkstra (1968):

> Since a number of years I am familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. Later I discovered why the use of the go to statement has such disastrous effects and did I become convinced that the go to statement should be abolished from all "higher level"

Plain Threads Considered Harmful

- A large fraction of the flaws in software development are due to programmers not fully understanding all the possible states their code may execute in. In a multithreaded environment, the lack of understanding and the resulting problems are greatly amplified, almost to the point of panic if you are paying attention.

  Programming in a functional style makes the state presented to your code explicit, which makes it much easier to reason about, and, in a completely pure system, makes thread race conditions impossible.

John Carmack: In-depth: Functional programming in C++
http://gamasutra.com/view/news/169296/Indepth_Functional_programming_in_C.php

STE||AR GROUP

Plain Threads Considered Harmful

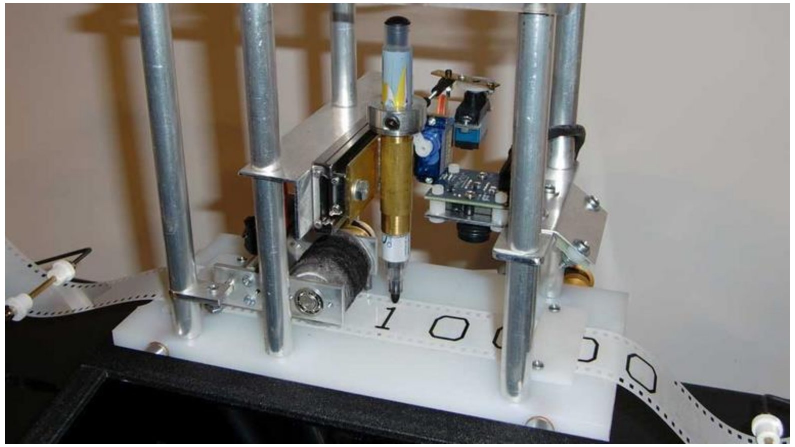
What’s a ‘Thread’?

- C++ Standard defines it as follows:

  - **Thread of execution**: “single flow of control within a program” (§1.10p1)
  - **`std::thread`**: is specified as a component “that can be used to create and manage threads" (see §30.1p1 and §30.3p1), where “threads" explicitly refers to the definition of ”threads of execution” (see §30.1p1)
    - Note: not defined to be an “OS-thread”
  - **Execution agent**: In §30.2.5.1p1, an execution agent is defined as “an entity such as a thread that may perform work in parallel with other execution agents”.

N4231: Torvald Riegel: Terms and definitions related to threads

What's a 'Thread'?



- An entity which exposes 4 properties:
  - A single flow of control (sequence of op-codes)
  - A program counter marking what's currently being executed
  - An associated execution context (stack, register set, static and dynamic memory, thread local variables, etc.)
  - A state (initialized, pending, suspended, terminated, etc.)

STE||AR GROUP

Plain Threads Considered Harmful

For instance, mentioned in Edward E. Lee's paper (in 2006): 'The Problem With Threads', others have talked about this

Main problems:

1. Threads are not composable
   - It's impossible to tell whether a library function creates threads itself
   - Prone to massive oversubscription
2. Parallelism can't be 'disabled'
   - Program logic is closely tied to parallelism even if two threads don't necessarily run concurrently
   - Difficult to reason about the whole and the parts independently
3. Difficult to ensure balanced load manually
   - Even smallest differences in runtime of parallel tasks influence overall performance

STE||AR GROUP

Plain Threads Considered Harmful

Other issues

- No ‘standard’ way of ‘returning’ values from threads
  - But all require explicit synchronization
- Working with threads makes concurrency explicit
  - Difficult to get right with large number of concurrent threads
  - Difficult to reason about programs using threads
- Threads are SLOW
- We need explicit parallelism, well integrated into C++ instead!
  - We need higher level parallelism constructs
  - We need new programming models helping to express parallelism


Getty

Threads are SLOW

Technology Demands new Response

Tianhe-2's projected theoretical peak performance: 54.9 PetaFLOPs

16,000 nodes, ~3,200,000 computing cores (32,000 Intel Ivy Bridge Xeons, 48,000 Xeon Phi Accelerators)

Amdahl's Law (Strong Scaling)

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

- S: Speedup
- P: Proportion of parallel code
- N: Number of processors

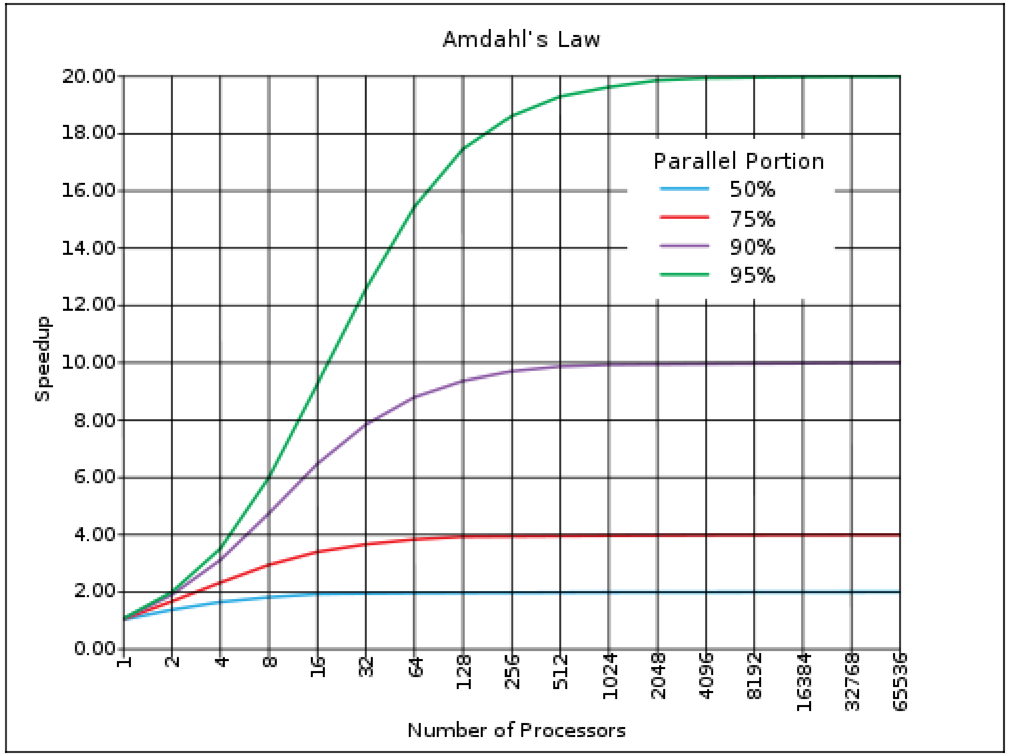


Figure courtesy of Wikipedia (http://en.wikipedia.org/wiki/Amdahl's_law)

The 4 Horsemen of the Apocalypse: SLOW

- **Starvation**
  - Insufficient concurrent work to maintain high utilization of resources
- **Latencies**
  - Time-distance delay of remote resource access and services
- **Overheads**
  - Work for management of parallel actions and resources on critical path which are not necessary in sequential variant
- **Waiting** for Contention resolution
  - Delays due to lack of availability of oversubscribed shared resources


courtesy of www.albrecht-durer.org

STE||AR GROUP

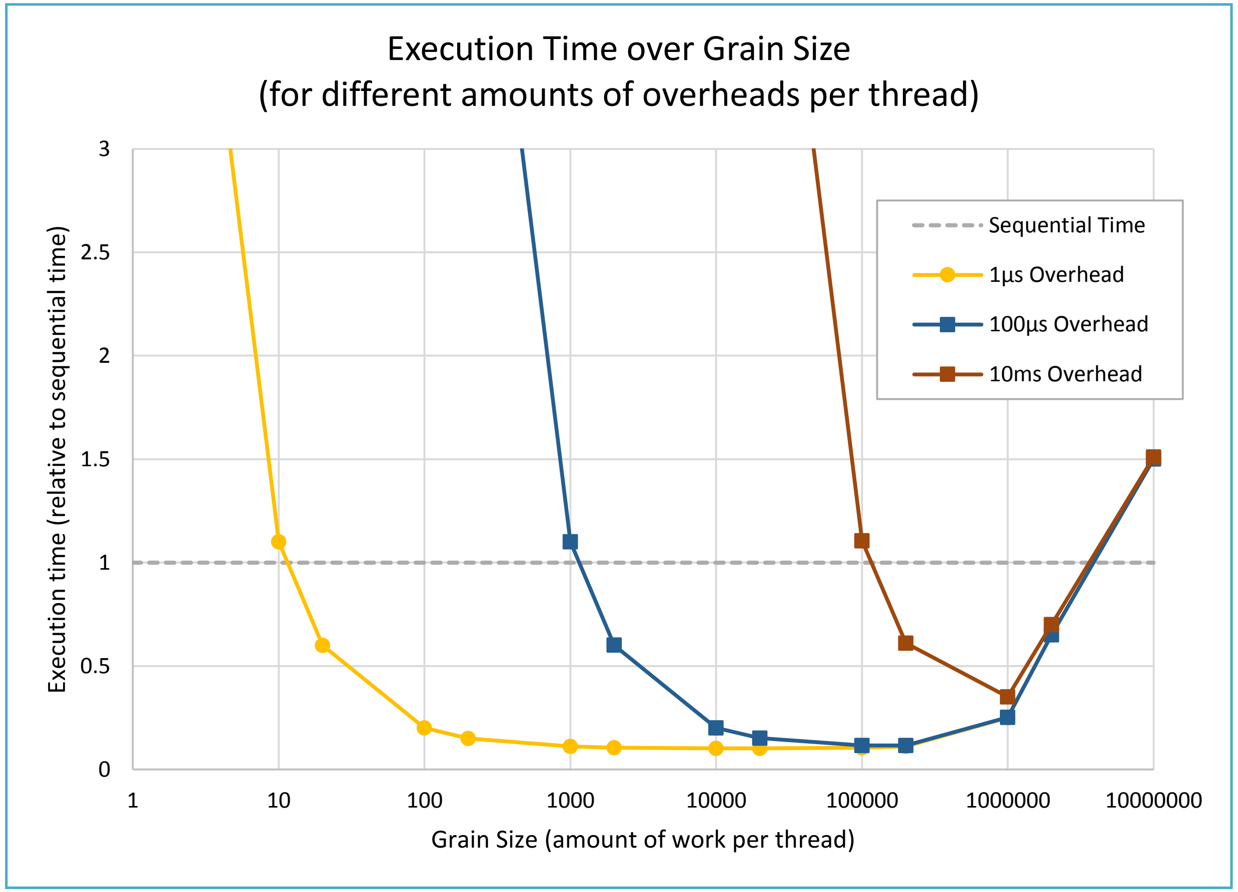
The 4 Horsemen of the Apocalypse: SLOW

- Starvation
  - Insufficient concurrent work to maintain high utilization of resources
- Latencies
  - Time-distance delay of remote access and services
- Overhe...
  - W... are ...ant
- W... Contention resolution
  - ...ue to lack of availability of o...rsubscribed shared resources

Impose upper bound on both, weak and strong scaling

courtesy of www.albrecht-durer.org

Overheads: The Worst of All?



STE||AR GROUP

Overheads: The Worst of All?

- Even relatively small amounts of work can benefit from being split into smaller tasks
  - Possibly huge amount of 'threads'
    - In the previous gedankenexperiment we ended up considering up to 10 million threads
    - Best possible scaling is predicted to be reached when using 10000 threads (for 10 seconds worth of work)
- Several problems
  - Impossible to work with that many kernel threads (p-threads)
  - Impossible to reason about this amount of tasks
  - Requires abstraction mechanism

STE||AR GROUP

The Challenges

- We need to find a usable way to fully parallelize the applications
- Goals are
  - Defeat The Four Horsemen
  - Provide manageable paradigms and APIs for handling parallelism
  - Expose asynchrony and parallelism to the programmer without exposing concurrency
  - Make data dependencies explicit, hide notion of 'thread', 'communication', and 'data distribution' as much as possible

Stepping Aside

HPX – A General Purpose Runtime System for Applications of Any Scale

HPX – A General Purpose Runtime System

- Solidly based on a theoretical foundation – a well defined, new execution model (ParalleX)
- Exposes an uniform, standards-oriented API for ease of programming parallel and distributed applications.
  - Enables to write fully asynchronous code using hundreds of millions of threads.
  - Provides unified syntax and semantics for local and remote operations.
- Enables writing applications which out-perform and out-scale existing ones
  - A general purpose parallel runtime system for applications of any scale
    - http://stellar-group.org/libraries/hpx
    - https://github.com/STEllAR-GROUP/hpx/
- Is published under Boost license and has an open, active, and thriving developer community.
- Can be used as a platform for research and experimentation

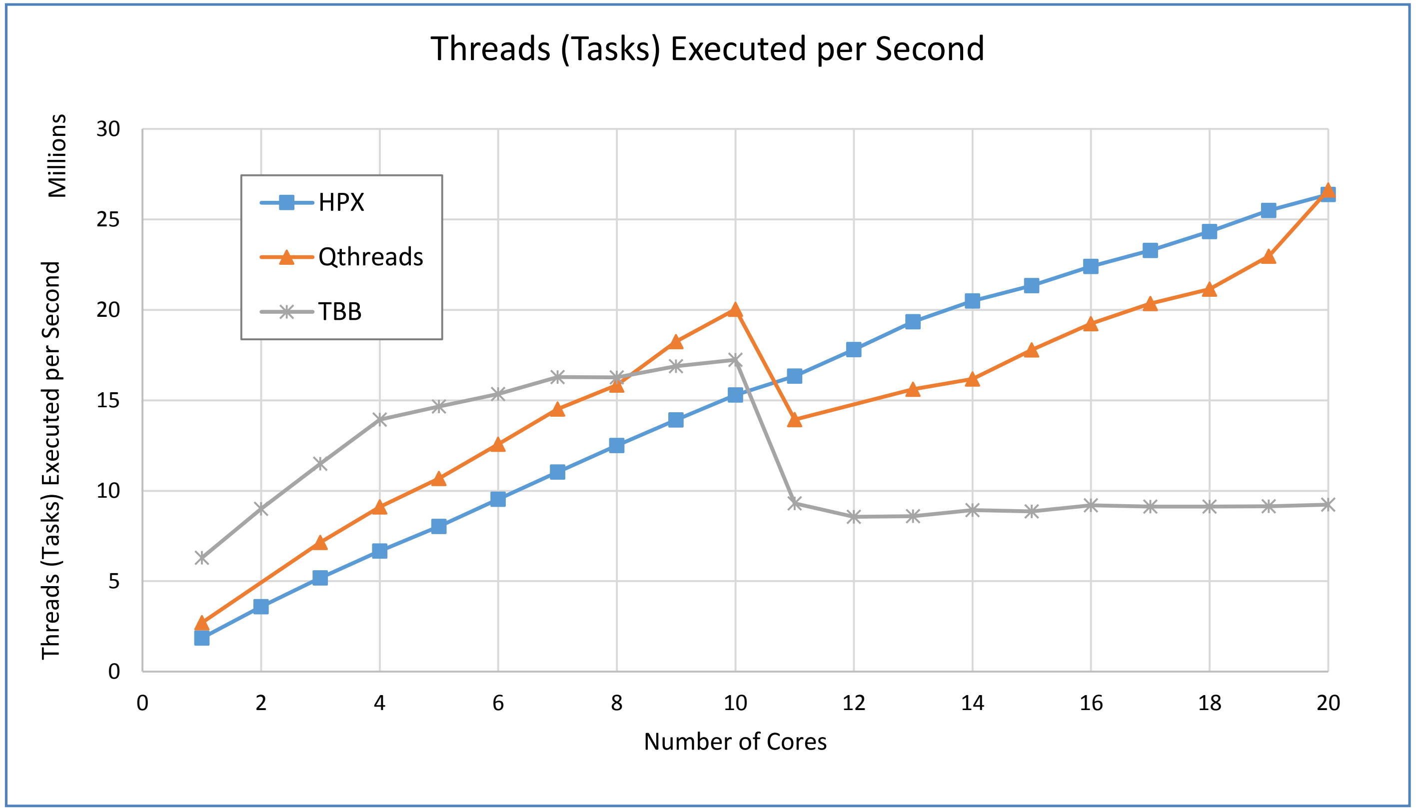
STE||AR GROUP

HPX – The API

- As close as possible to C++11/14 standard library, where appropriate, for instance

• std::thread	hpx::thread
• std::mutex	hpx::mutex
• std::future	hpx::future (including N4107, '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 (N4105, 'Parallelism TS')
• std::parallel::task_region	hpx::parallel::task_region (N4088)

Thread Overheads



Credit: Bryce Adelstein-Lelbach

Explicit Parallelism

Without Explicit Threads

Explicit Parallelism

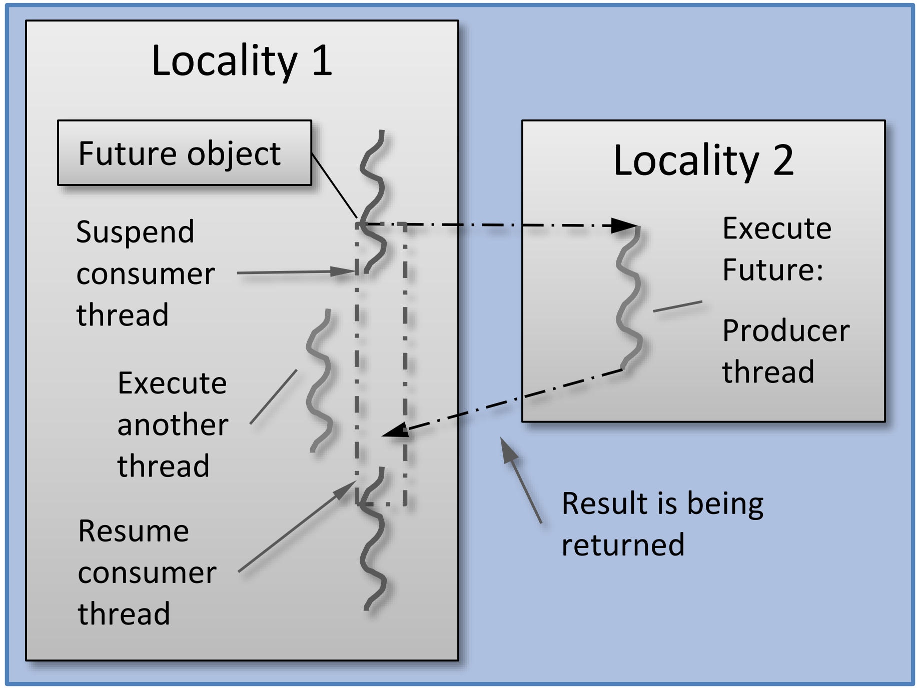
- C++ needs stronger support for higher level parallelism
  - C++11 has some basic facilities: `future`, `promise`, `packaged_task`, `async`
  - More is needed

- Several proposals to the Standardization Committee are under consideration
  - Technical Specification: Transactional Memory
  - Technical Specification: Concurrency (note: misnomer)
  - Technical Specification: Parallelism
  - Other smaller proposals: resumable functions, task regions, executors

STE||AR GROUP

What is a (the) future

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



- Enables transparent synchronization with producer
- Hides notion of dealing with threads
- Makes asynchrony manageable
- Allows for composition of several asynchronous operations
- (Turns concurrency into parallelism)

What is a (the) Future?

- 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, eventually
}
```

STE||AR GROUP

Concurrency TS (N4107)

- Misnomer: should be called 'Asynchrony TS'
- Extensions for `std::future`
- Means of sequential composition (add continuation to a future)
  - `std::experimental::future<>::then()`
- Means of parallel composition
  - `std::experimental::when_all()`, `std::experimental::when_any()`
- Helper facilities
  - `std::experimental::make_ready_future()`
  - `std::experimental::make_exceptional_future()`

Compositional facilities

- Sequential composition of futures

```
future<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
        });

    return f2;
}
```

Compositional facilities

- Parallel composition of futures

```
future<int> 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, etc.

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

    return result;
}
```

STE||AR GROUP

Dataflow – The New 'async' (HPX)

- 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

STE||AR GROUP

Parallelism TS (N4105)

- Fork-Join parallelism
  - Used for years: OpenMP, CILK, Java concurrency Framework, Task Parallel Library for .NET

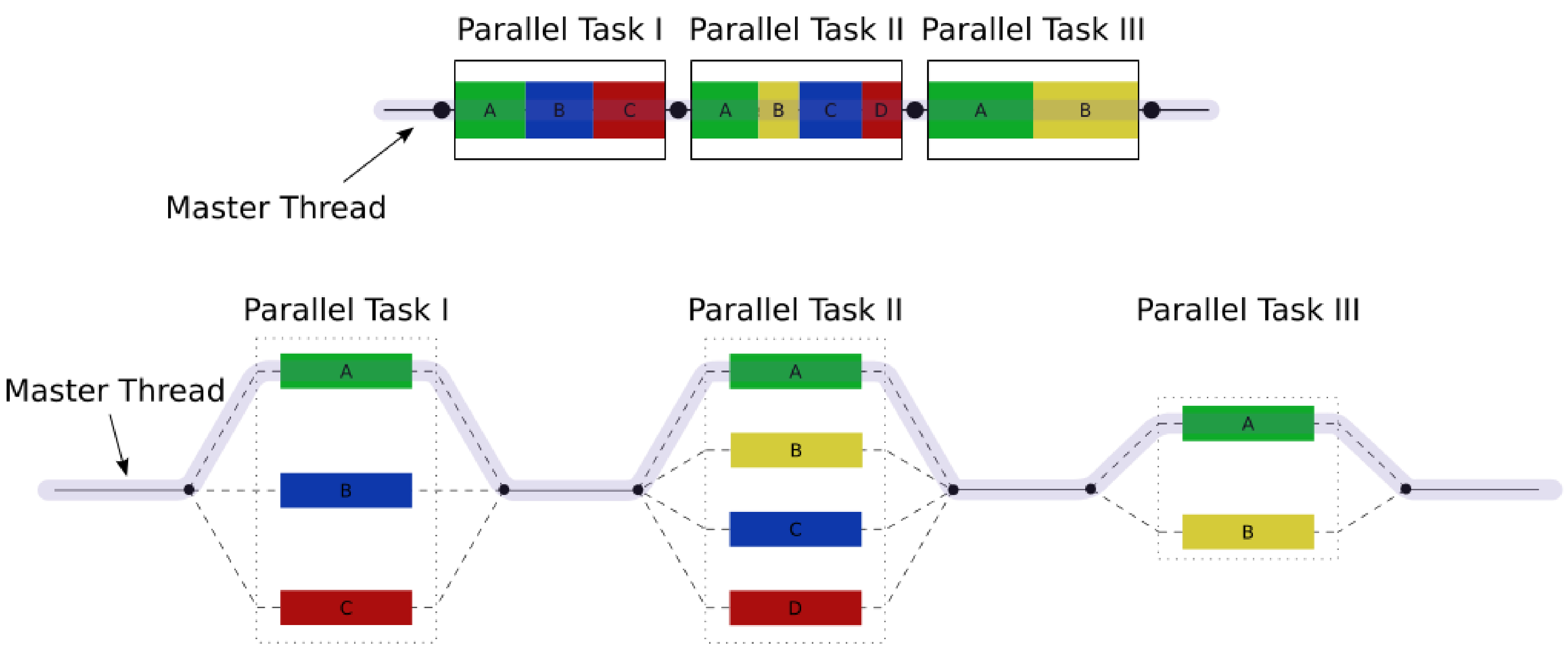
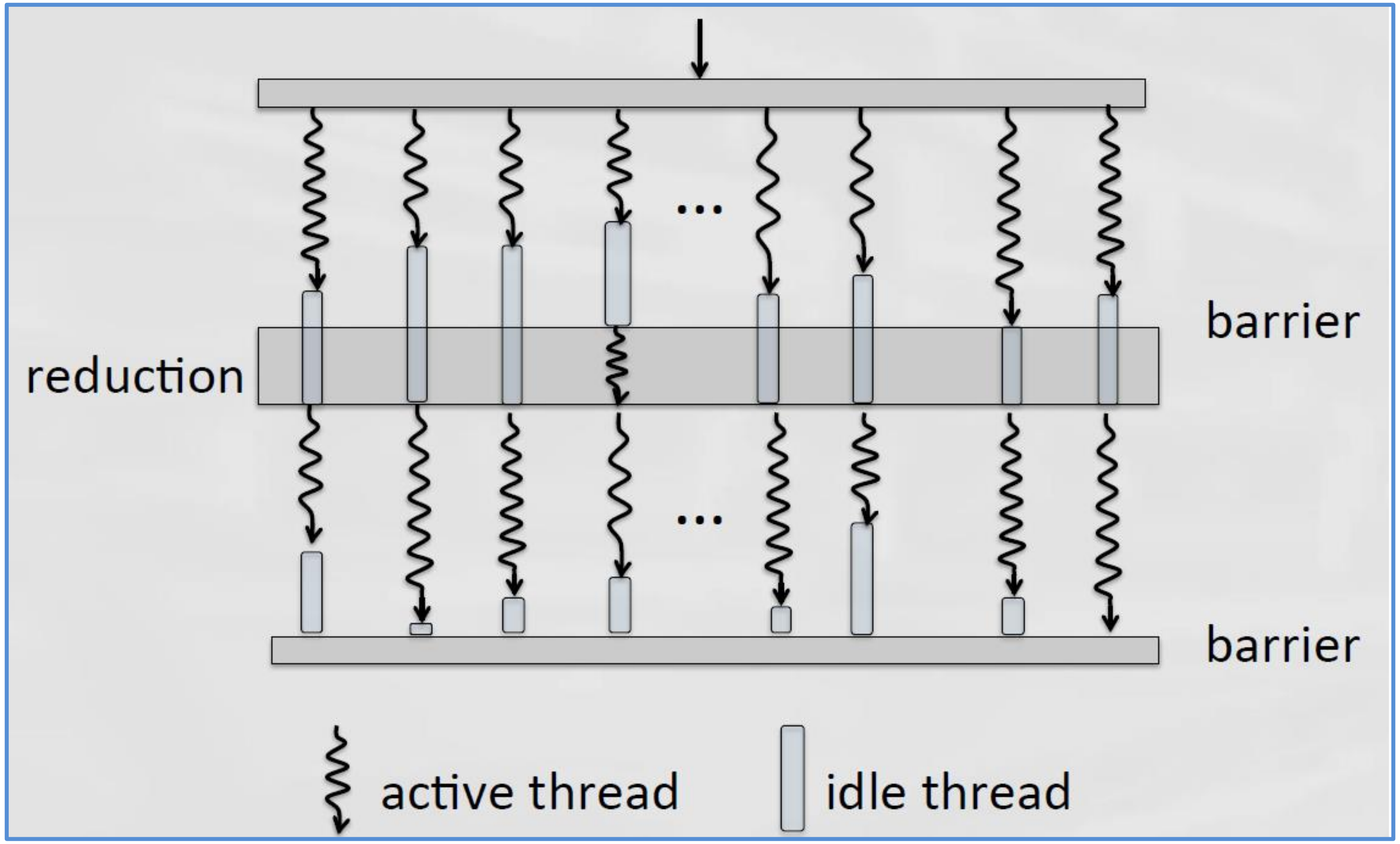


Image courtesy of: Wikipedia: http://en.wikipedia.org/wiki/Fork%E2%80%93join_model

Fork/Join Parallelism



Parallel Algorithms

- Mostly, same semantics as sequential algorithms
  - Additional, first argument: `execution_policy`
    - `sequential_execution_policy`: `seq`
    - `parallel_execution_policy`: `par`
    - `parallel_vector_execution_policy`: `par_vec`
  - Special rules related to exception handling
- Entirely fork/join as algorithms return only after all work has been done
  - Performance of those algorithms depends on high quality schedulers

STE||AR GROUP

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
uninitialized_copy	uninitialized_copy_n	uninitialized_fill	uninitialized_fill_n
unique	unique_copy		

STE||AR GROUP

Parallel Algorithms

```
std::vector<int> v = { 1, 2, 3, 4, 5, 6 };
std::experimental::transform(
    std::experimental::par, begin(v), end(v),
    [](int i) -> int
    {
        return i + 1;
    });
```

Parallel Algorithms (HPX)

- Extensions: Make all algorithms asynchronous, if needed
  - `parallel_task_execution_policy` (asynchronous version of `parallel_execution_policy`), generated with `par(task)`
  - `sequential_task_execution_policy` (asynchronous version of `sequential_execution_policy`), generated with `seq(task)`
  - In both cases the algorithms return a future<>
- More is needed
  - Make partition results available to programmer
  - Integration with Eric Niebler's Ranges

STE||AR GROUP

Other Proposals: Executors (N4143)

- Executors
  - Originally planned for Concurrency TS, however not included, now N4143
  - Goal: provide alternative for `std::async()`
  - Core API: `void spawn(Func&&)`
    - Any object exposing this function is an executor
  - Defines various executor types
    - `thread_per_task_executor`, `thread_pool_executor`, `system_executor`, etc.
- Hides the notion of threads by enforcing to think about self-contained tasks
  - Almost functional
  - No races if 'tasks' are side effect free and act on value-type arguments only

Executors (HPX)

- Added possibility to pass executors to `async()` and `dataflow()`
- Enables control over where (in what context) the function is spawned
  - Could be UI – thread, or dedicated thread used for IO
- Added executors representing different scheduler types
  - FIFO, LIFO schedulers
  - Non-work-stealing schedulers
  - NUMA aware schedulers.
  - Etc.
- User can easily create new executors for finer control

STE||AR GROUP

Resumable Functions (D4134)

- Highly scalable (to hundreds of millions of concurrent co-routines)
- Highly efficient (resume and suspend operations comparable in cost to a function call overhead)
- Seamless interaction with existing facilities **with no overhead**
- Open ended co-routine machinery allowing library designers to develop co-routine libraries exposing various high-level semantics, such as generators, go-routines, tasks and more.
- Usable in environments where exceptions are forbidden or not available

Resumable Functions (D4134)

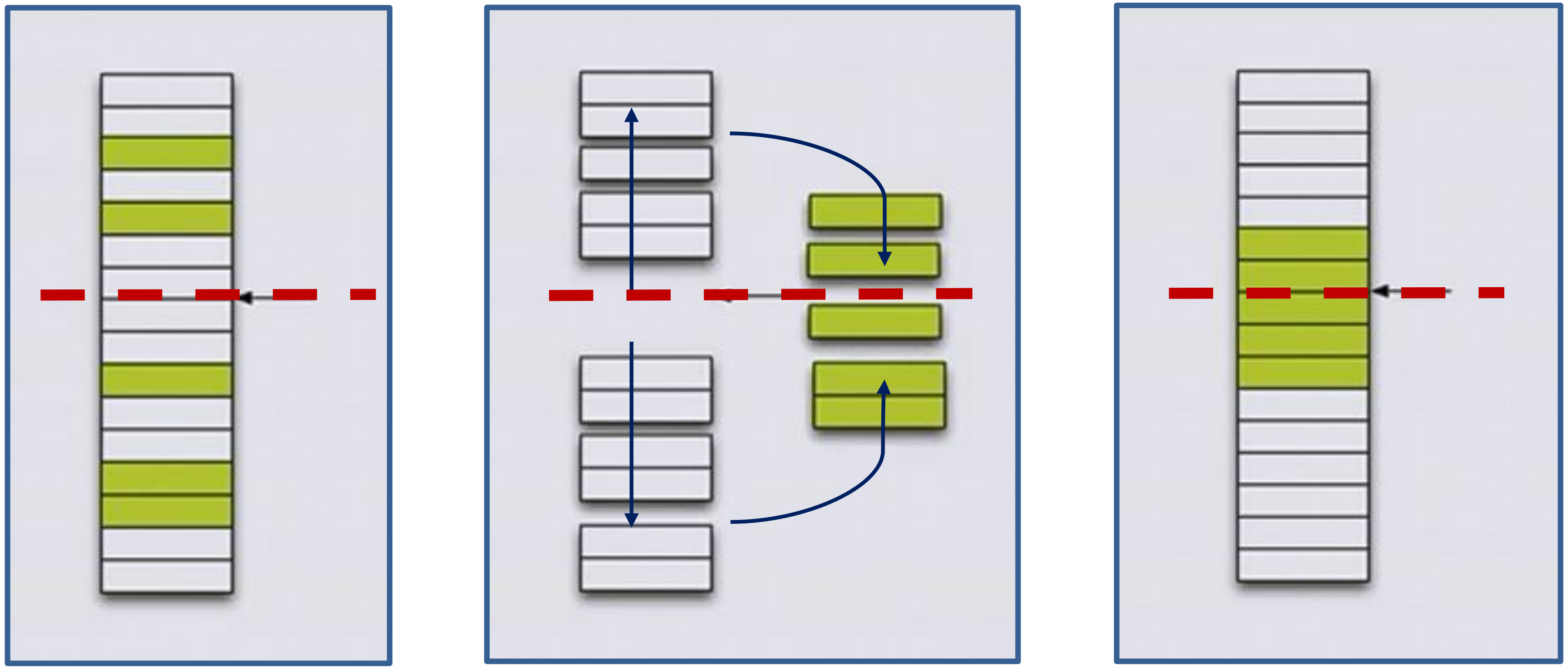
- Canonical example:

```
std::future<std::ptrdiff_t> tcp_reader(int total)
{
    char buf[64 * 1024];
    std::ptrdiff_t result = 0;
    auto conn = await Tcp::Connect("127.0.0.1", 1337);
    do {
        auto bytesRead = await conn.Read(buf, sizeof(buf));
        total -= bytesRead;
        result += std::count(buf, buf + bytesRead, 'c');
    } while (total > 0);
    return result;
}
```

Gor Nishanov: await 2.0, CppCon 2014, https://www.youtube.com/watch?v=KUhSjfSbINE

Two Examples

Extending Parallel Algorithms



Sean Parent: C++ Seasoning, Going Native 2013

STE||AR GROUP

Extending Parallel Algorithms

- New algorithm: gather

```
template <typename BiIter, typename Pred>
pair<BiIter, BiIter> gather(BiIter f, BiIter l, BiIter p, Pred pred)
{
    BiIter it1 = stable_partition(f, p, not1(pred));
    BiIter it2 = stable_partition(p, l, pred);
    return make_pair(it1, it2);
}
```

Sean Parent: C++ Seasoning, Going Native 2013

Extending Parallel Algorithms

- New algorithm: gather_async

```
template <typename BiIter, typename Pred>
future<pair<BiIter, BiIter>> gather_async(BiIter f, BiIter l, BiIter p, Pred pred)
{
    future<BiIter> f1 = parallel::stable_partition(par(task), f, p, not1(pred));
    future<BiIter> f2 = parallel::stable_partition(par(task), p, l, pred);
    return dataflow(
        unwrapped([](BiIter r1, BiIter r2) { return make_pair(r1, r2); }),
        f1, f2);
}
```

STE||AR GROUP

Extending Parallel Algorithms (await)

- New algorithm: gather_async

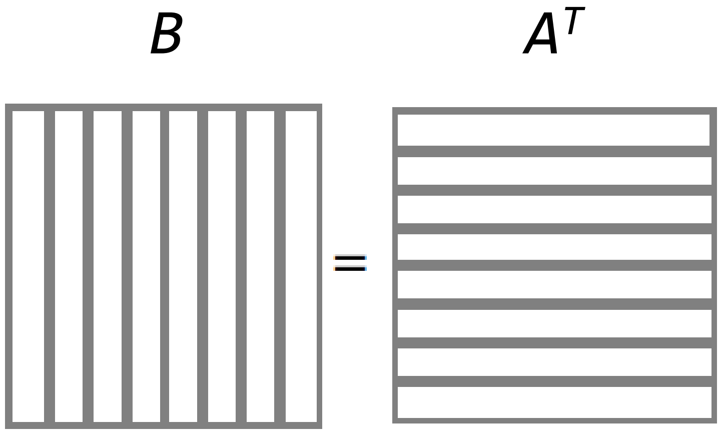
```
template <typename BiIter, typename Pred>
future<pair<BiIter, BiIter>> gather_async(BiIter f, BiIter l, BiIter p, Pred pred)
{
    future<BiIter> f1 = parallel::stable_partition(par(task), f, p, not1(pred));
    future<BiIter> f2 = parallel::stable_partition(par(task), p, l, pred);
    return make_pair(await f1, await f2);
}
```

Matrix Transposition

An extended Example

Matrix Transposition

$B = A^T \Rightarrow$



Matrix Transposition

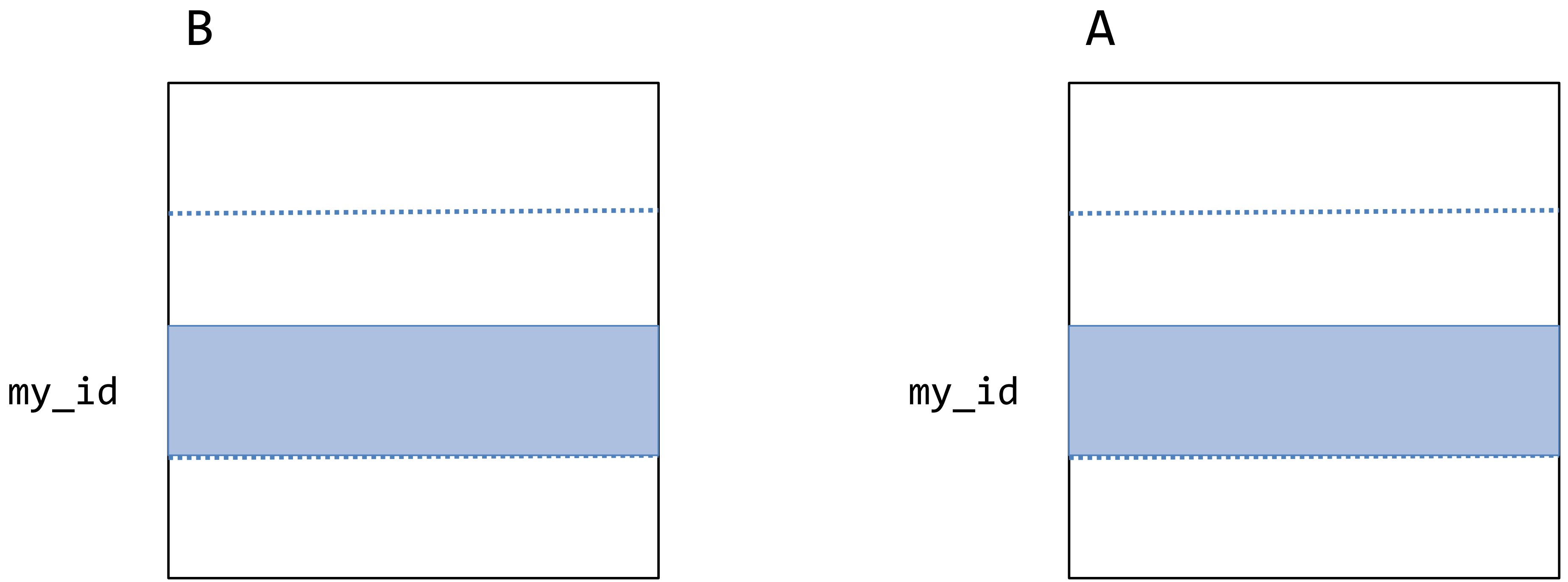
```
void transpose(std::vector<double>& A, std::vector<double>& B)
{
    #pragma omp parallel for
    for (std::size_t i = 0; i != order; ++i)
        for (std::size_t j = 0; j != order; ++j)
            B[i + order * j] = A[j + order * i];
}

int main()
{
    std::vector<double> A(order * order);
    std::vector<double> B(order * order);

    transpose(A, B);
}
```

Matrix Transposition

```
// parallel for
std::vector<double> A(order * order);
std::vector<double> B(order * order);
auto range = irange(0, order);
for_each(par, begin(range), end(range),
    [&](std::size_t i)
    {
        for (std::size_t j = 0; j != order; ++j)
        {
            B[i + order * j] = A[j + order * i];
        }
    });
```

STE||AR GROUP

Matrix Transposition (distributed)



Matrix Transposition (distributed)

```
std::size_t my_id = hpx::get_locality_id();
std::size_t num_blocks = hpx::get_num_localities();
std::size_t block_order = order / num_blocks;

std::vector<block> A(num_blocks);
std::vector<block> B(num_blocks);
```

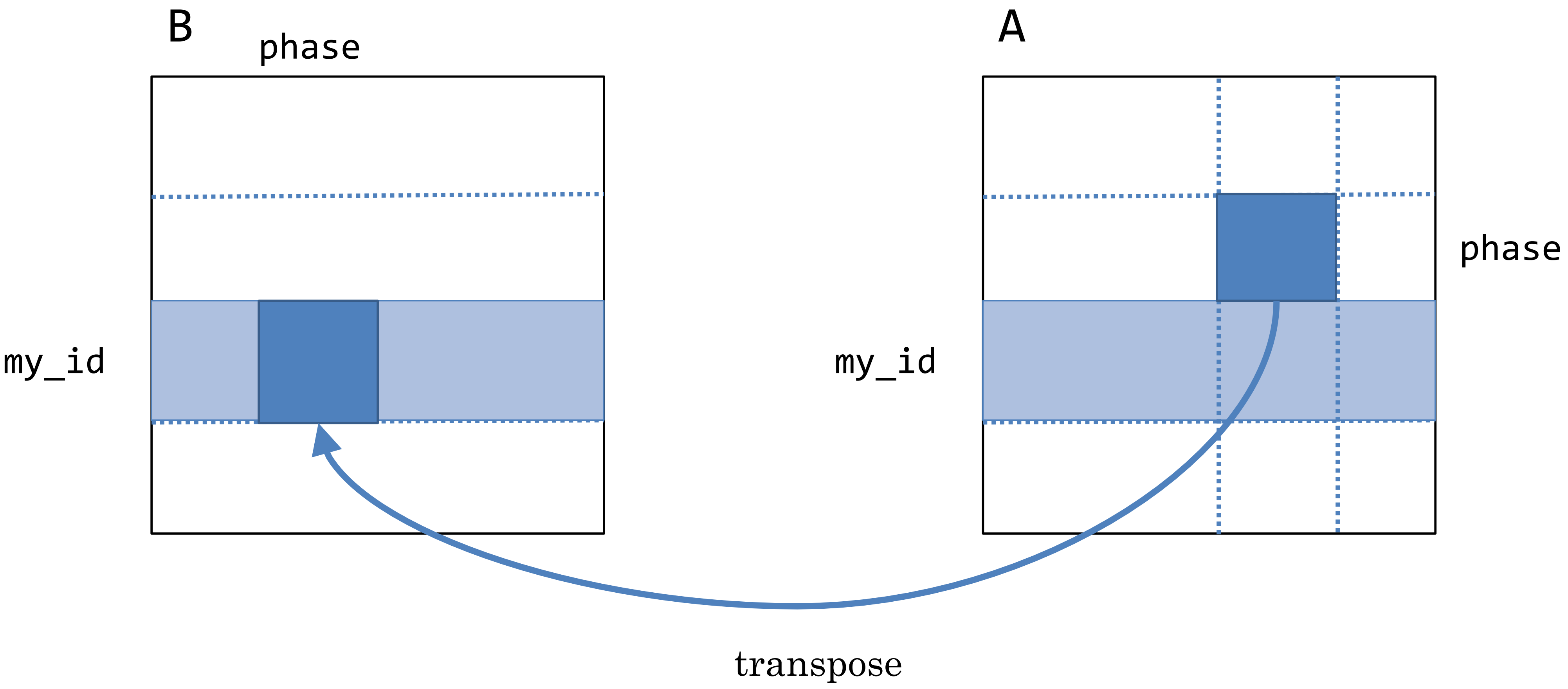
STE||AR GROUP

Matrix Transposition (distributed)

```
for (std::size_t b = 0; b != num_blocks; ++b) {
    if (b == my_id) {
        A[b] = block(block_order * order);
        B[b] = block(block_order * order);
        hpx::register_id_with_basename("A", A[b], b);
        hpx::register_id_with_basename("B", B[b], b);
    }
    else {
        A[b] = hpx::find_id_from_basename("A", b);
        B[b] = hpx::find_id_from_basename("B", b);
    }
}
```

STE||AR GROUP

Matrix Transposition (distributed)



Matrix Transposition (distributed)

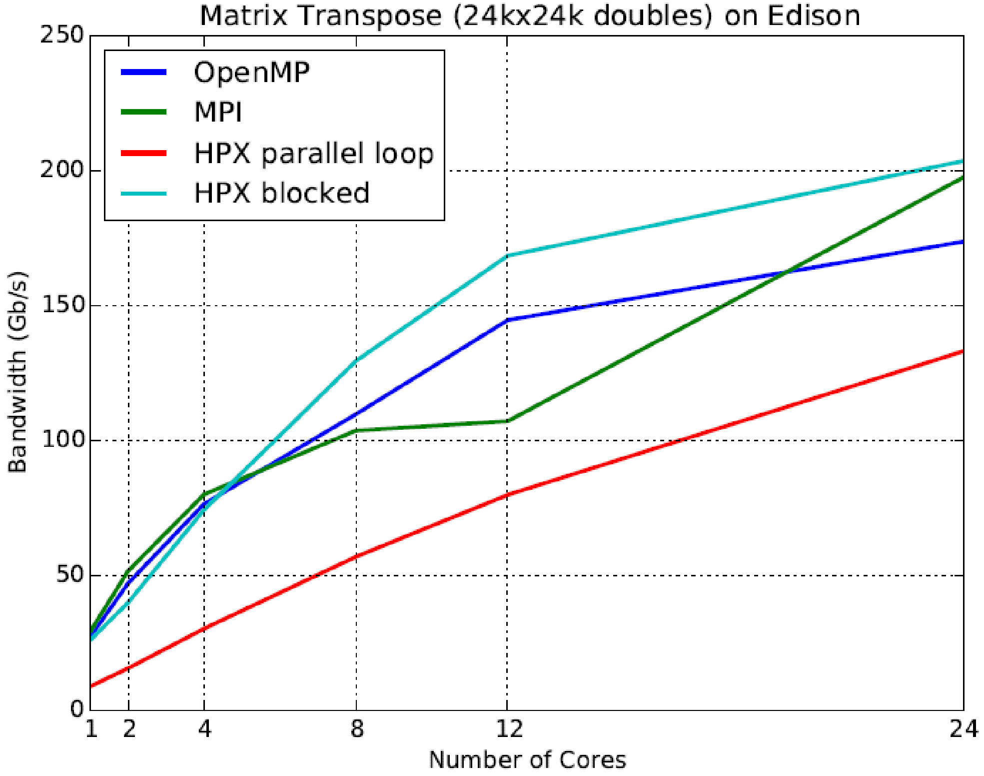
```
std::vector<future<void>> results;
auto range = irange(0, num_blocks);
for_each(seq, begin(range), end(range),
    [&](std::size_t phase)
    {
        future<block_data> f1 = A[phase].get_data(my_id, block_size);
        future<block_data> f2 = B[my_id].get_data(phase, block_size);
        results.push_back(hpx::dataflow(unwrapped(transpose), f1, f2));
    });
wait_all(results);
```

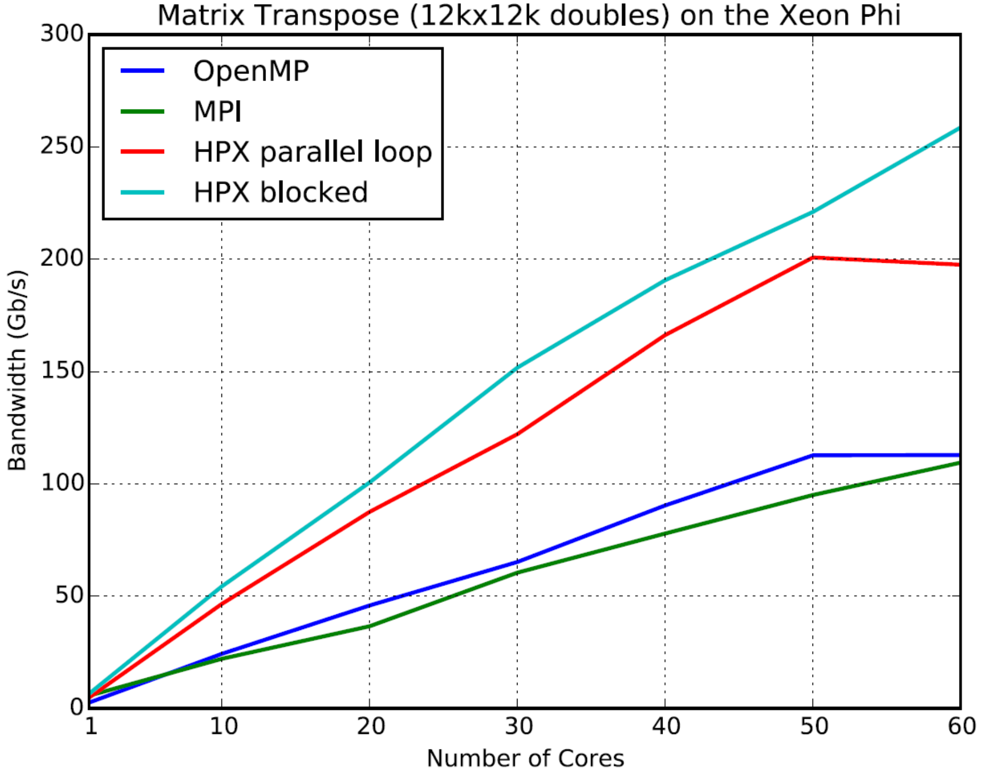
Matrix Transposition (await)

```
auto range = irange(0, num_blocks);
for_each(par, begin(range), end(range),
    [&](std::size_t phase)
    {
        future<block_data> f1 = A[phase].get_data(my_id, block_order);
        future<block_data> f2 = B[my_id].get_data(phase, block_order);
        transpose(await f1, await f2);
    });
```

STE||AR GROUP

Matrix Transposition, Results





Credit: Thomas Heller, runs performed on Edison and Babbagge (NERSC)

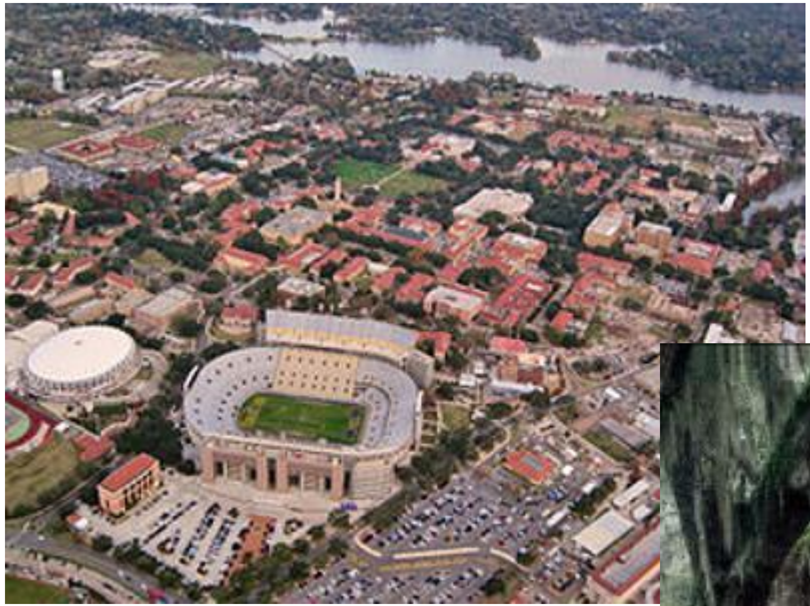
STE||AR GROUP

Conclusions

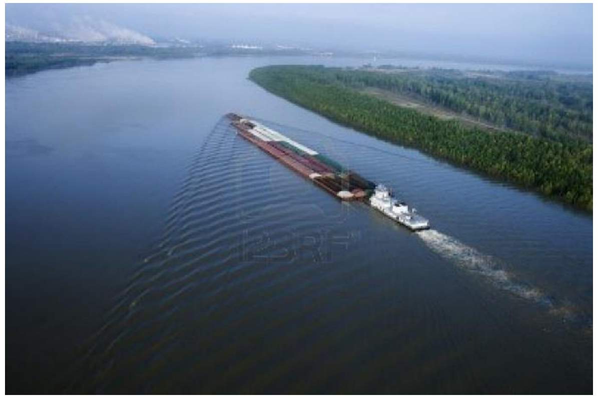
- Multi-core is the new modality and parallelism is here to stay
  - We need higher level abstractions for threading and parallelism
  - Goal should be to make data dependencies explicit
  - Allow reasoning about massively parallel code
- C++11/14 already defines some basic types but more are needed
- Many C++ standardization proposals are being currently discussed
  - Several technical specifications to be released shortly: Parallelism TS, Concurrency TS, and Transactional Memory TS
  - More proposals are in the pipeline

So stop using plain threads after all

STE||AR GROUP



LSU CENTER FOR COMPUTATION & TECHNOLOGY



STE||AR GROUP