HPX

A C++ Library for Parallelism and Concurrency

HARTMUT KAISER

Cppcon
The C++ Conference

2022
September 12th-16th

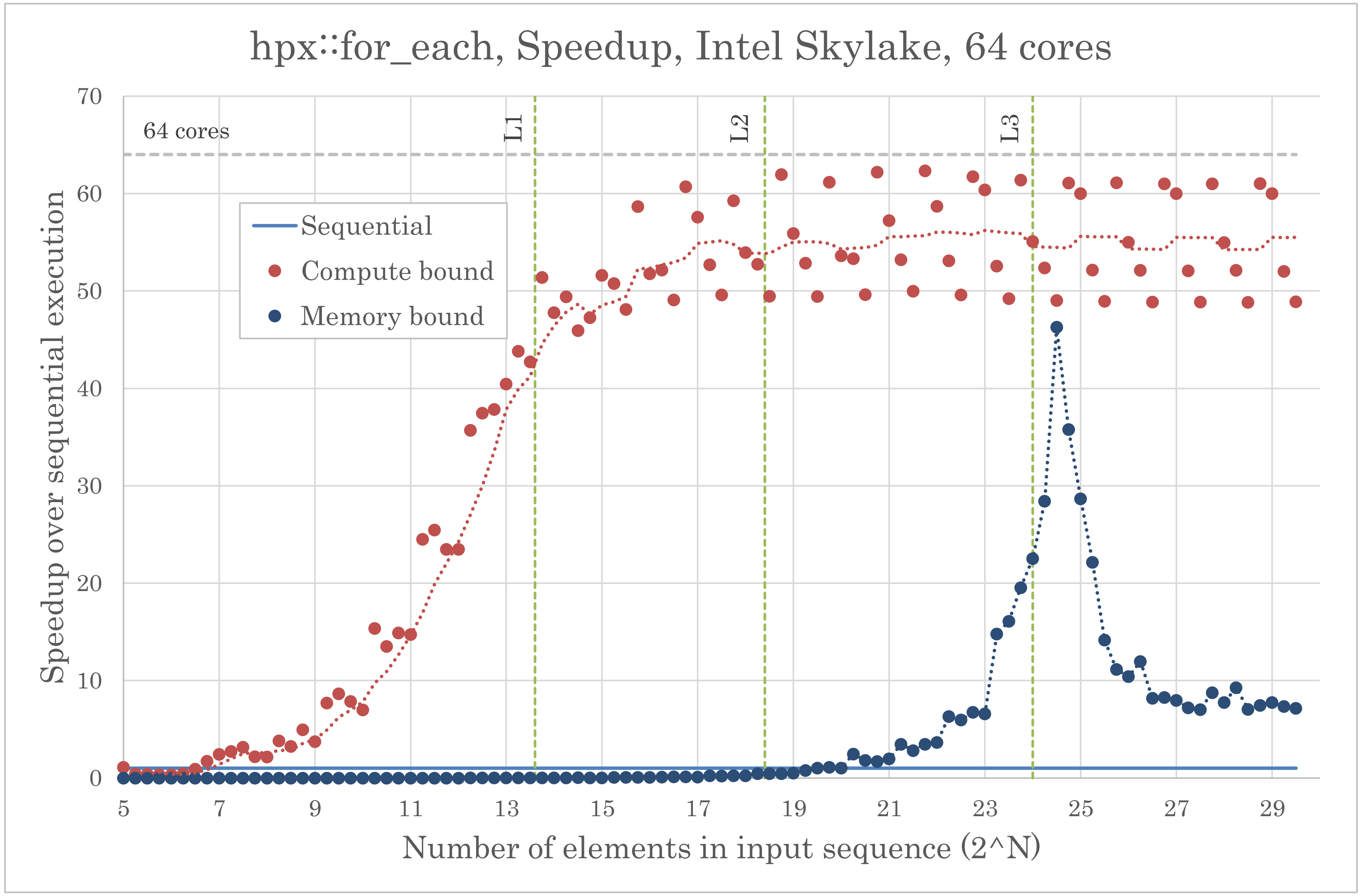
HPX
A C++ Library for Parallelism and Concurrency

Hartmut Kaiser (hkaiser@cct.lsu.edu)

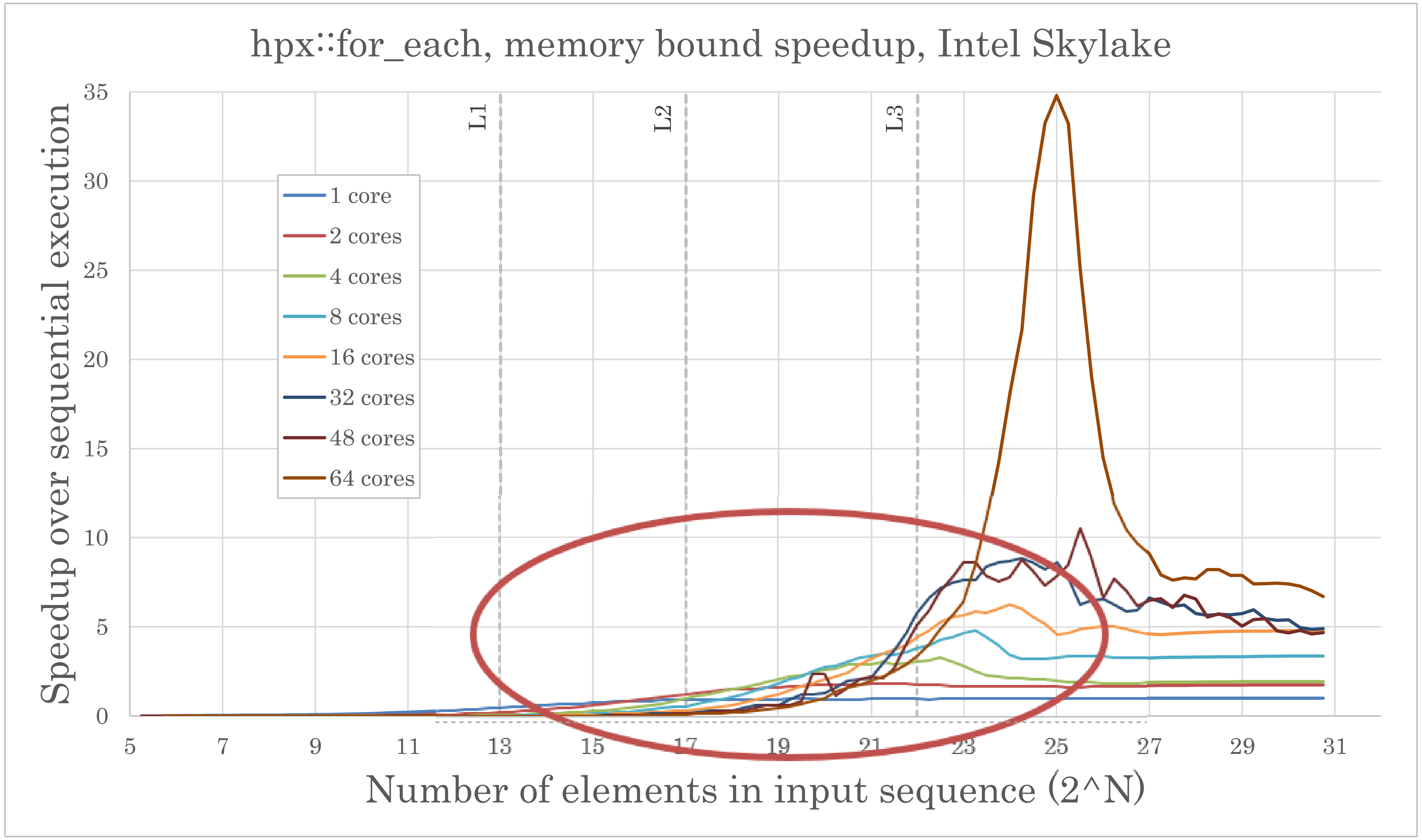
CppCon, September 12, 2022

A Real World Story

STE||AR GROUP


hpx::for_each, Speedup, Intel Skylake, 64 cores
64 cores
L1
L2
L3
Sequential
Compute bound
Memory bound
Speedup over sequential execution
0
10
20
30
40
50
60
70
5
7
9
11
13
15
17
19
21
23
25
27
29
Number of elements in input sequence (2^N)

STE||AR GROUP


hpx::for_each, memory bound speedup, Intel Skylake
Speedup over sequential execution
Number of elements in input sequence (2^N)
35
30
25
20
15
10
5
0
5
7
9
11
13
15
17
19
21
23
25
27
29
31
L1
L2
L3
1 core
2 cores
4 cores
8 cores
16 cores
32 cores
48 cores
64 cores

STE||AR GROUP

Conventions

- Namespaces

`std::`	namespace std
`stdex::`	namespace std::execution
`stdexp::`	namespace std::experimental
`hpx::`	namespace hpx
`hpxex::`	namespace hpx::execution
`hpxexp::`	namespace hpx::execution::experimental
`hpxtt::`	namespace hpx::this_thread::experimental

STE||AR GROUP

HPX

The C++ Standards Library for Concurrency and Parallelism

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

STE||AR GROUP

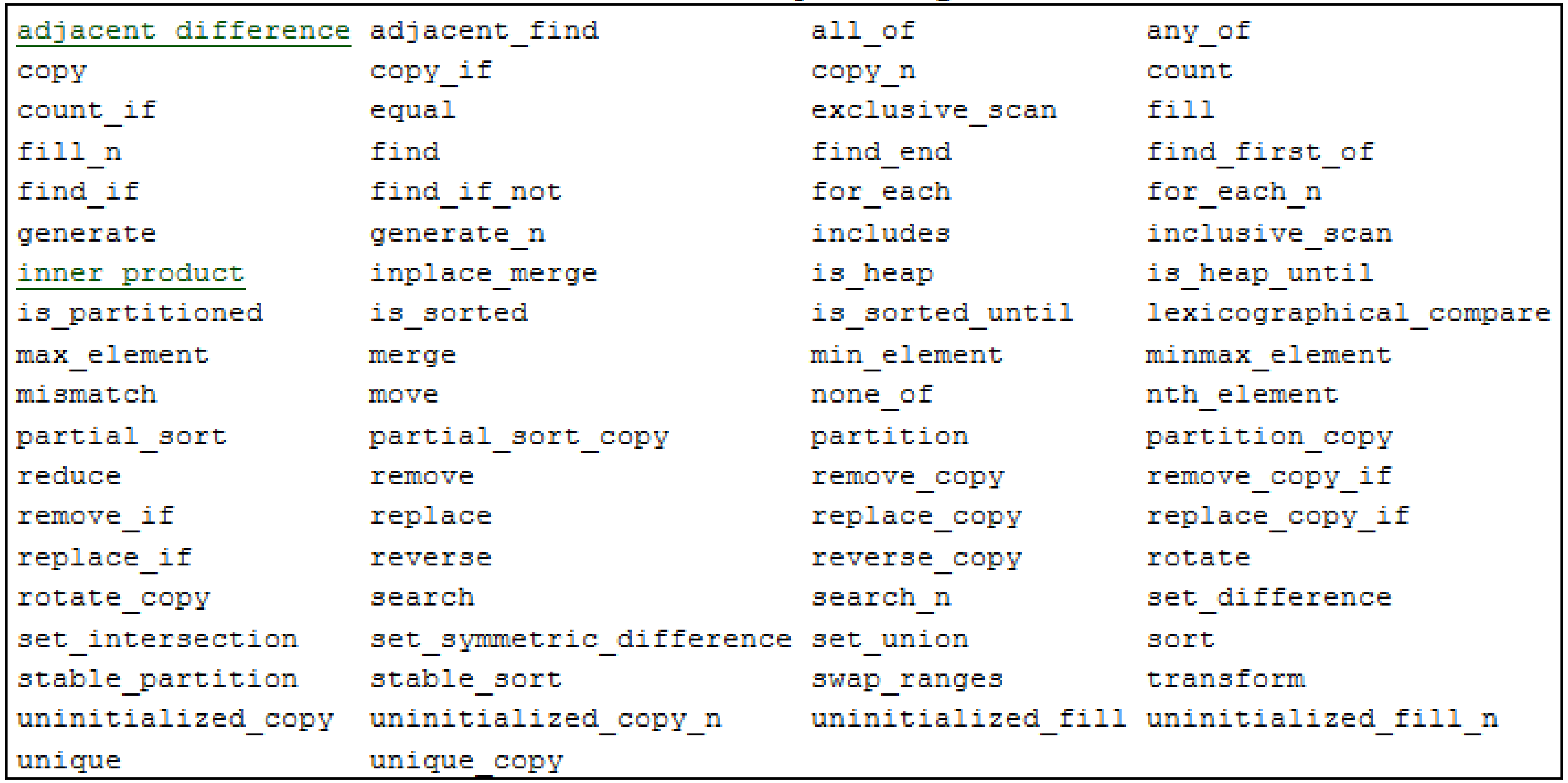
HPX – An Asynchronous Many-task Runtime System

- At it's heart, HPX is a very efficient threading implementation
- Several functional layers are implemented on top:
  - C++ standards-conforming API exposing everything related to parallelism and concurrency
  - Full set of C++17/C++20/C++23 (parallel) algorithms
    - One of the first full openly available implementations
    - Extensions:
      - asynchronous execution
      - parallel range based algorithms
      - vectorizing execution policies `simd`/`par_simd`
  - Full set of senders/receivers (currently being discussed for standardization)
    - Implemented using C++17
  - Distributed operation
    - Extending the standard interfaces for use on tightly coupled clusters (super-computers)
    - Global address space, load balancing, uniform API for local and remote operations

STE||AR GROUP

HPX – An Asynchronous Many-task Runtime System

- Full set of C++17/C++20/C++23 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		

Parallel Algorithms

- Simple iterative algorithms
  - One pass over the input sequence
  - `for_each`, `copy`, `fill`, `generate`, `reverse`, etc.
- Iterative algorithms ‘with a twist’
  - One pass over the input sequence
  - Parallel execution requires additional operation after first pass, most of the time this is a reduction step
  - `min_element`, `all_of`, `find`, `count`, `equal`, etc.
- Scan based algorithms
  - At least three algorithmic steps
  - `inclusive_scan`, `exclusive_scan`, etc.
- Auxilliary algorithms
  - Sorting, heap operations, set operations, `rotate`

Parallel Algorithms

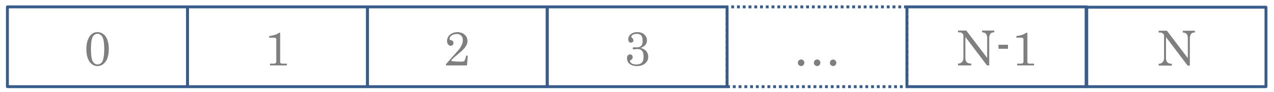
- How does parallelization work?
- On CPUs
  - Split input sequence into pieces (chunks) of theoretically arbitrary size
  - Run algorithm on more than one core, each core on it's own chunk
  - Perform necessary synchronization and reduction
- On GPUs
  - Split input sequence into pieces (chunks) that are sized to fit into a warp
  - Run algorithm on more than one warp, each warp on it's own chunk, each core on its own element
  - Perform necessary synchronization and reduction

Parallelize Loops

STE||AR GROUP

Parallelize Loops

Sequence of elements:



```
std::vector<int> d = {...};
hpx::for_each(d.begin(), d.end(), [](int val) {...});
```

```
template <typename Iterator, typename F>
void for_each(Iterator b, Iterator e, F f)
{
    while (b != e)
        f(*b++);
}
```

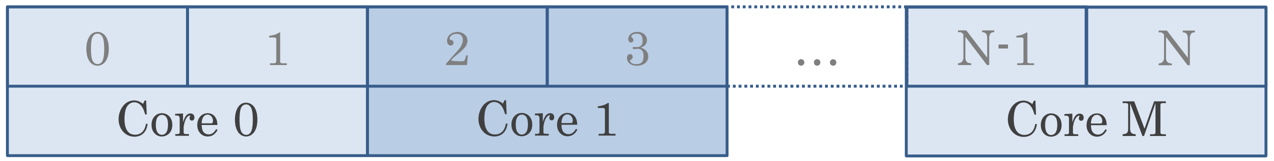
Execution Policies

- Standard introduces: `std::seq`, `std::par`, `std::unseq` (C++20), `std::par_unseq`
  - Passed as additional first argument to algorithm
- Convey guarantees/requirements imposed by loop body
  - `seq`: execute in-order (sequenced) on current thread
  - `unseq`: allow out-of-order execution (unsequenced) on current thread - vectorization
  - `par`: allow parallel execution on different threads
  - `par_unseq`: allow parallel out-of-order (vectorized) execution on different threads
- Proposed for standardization (P0350: Integrating simd with parallel algorithms): `stdex::simd`
  - Enable *explicit* vectorization that relies on special C++ types representing vector registers (`stdexp::simd`, see: Parallelism TS V2, latest draft: N4808)
- HPX introduces:
  - Asynchronous policies, e.g. `par(task)`: allow asynchronous operation
  - Explicit parallelized vectorization: `par_simd`
  - Executors: attached to execution policies using `.on()`

See: wg21.link/p0350, wg21.link/n4808

Parallelize Loops

Sequence of elements:



```
std::vector<int> d = {...};
hpx::for_each(par, d.begin(), d.end(), [](int val) {...});
```

```
template <typename Iterator, typename F>
void for_each(parallel_policy, Iterator b, Iterator e, F f)
{
    auto size = std::distance(b, e);                                  // Iterator should be random access
    std::vector<hpx::future<void>> v;
    for (size_t chunk = 0; chunk != NUM_CHUNKS; ++chunk) {            // assume: cleanly divisible
        v.push_back(hpx::async([&]() {                                // async() launches new thread, returns future
            auto begin = std::next(b, (chunk * size) / NUM_CHUNKS);
            hpx::for_each(begin, std::next(begin, size / NUM_CHUNKS), f);   // sequential for_each()
        }));
    }
    hpx::wait_all(v);
}
```

STE||AR GROUP

Parallelize Loops: Observations

- Parallelization concurrently runs sequential operations on parts of the input
  - At least for CPU based implementations
  - GPU based algorithms are usually different
- Iterators should be random access
  - Otherwise performance might be bad
- NUM_CHUNKS is a magic number!
  - How should we select it?
  - What are the criteria for best performance?
- NUM_CORES is another magic number
- AFFINITIES are important too (NUMA awareness!), control task placement

A Bit of Background

Why is it so difficult to efficiently parallelize execution?

STE||AR GROUP

Amdahl's Law (Strong Scaling)

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

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

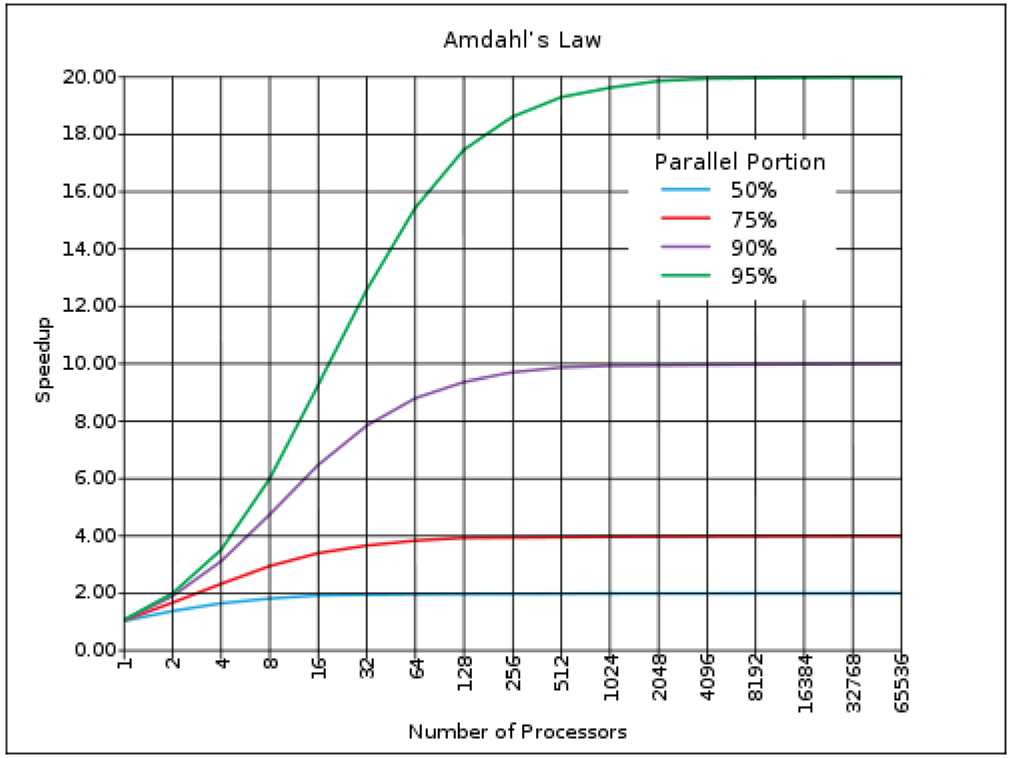


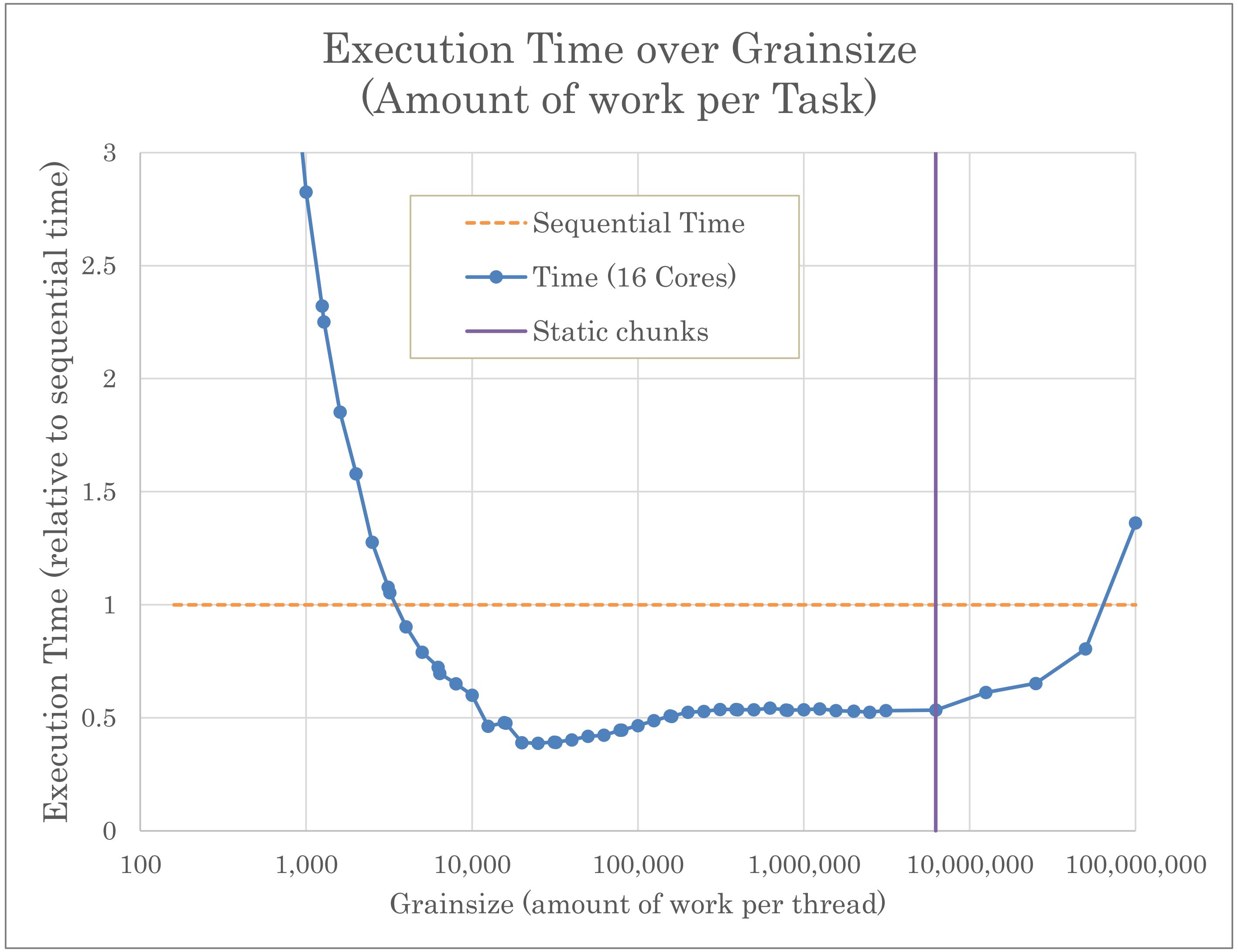
Figure courtesy of Wikipedia (http://en.wikipedia.org/wiki/Amdahl's_law)

The 4 Horsemen of the Apocalypse: SLOW

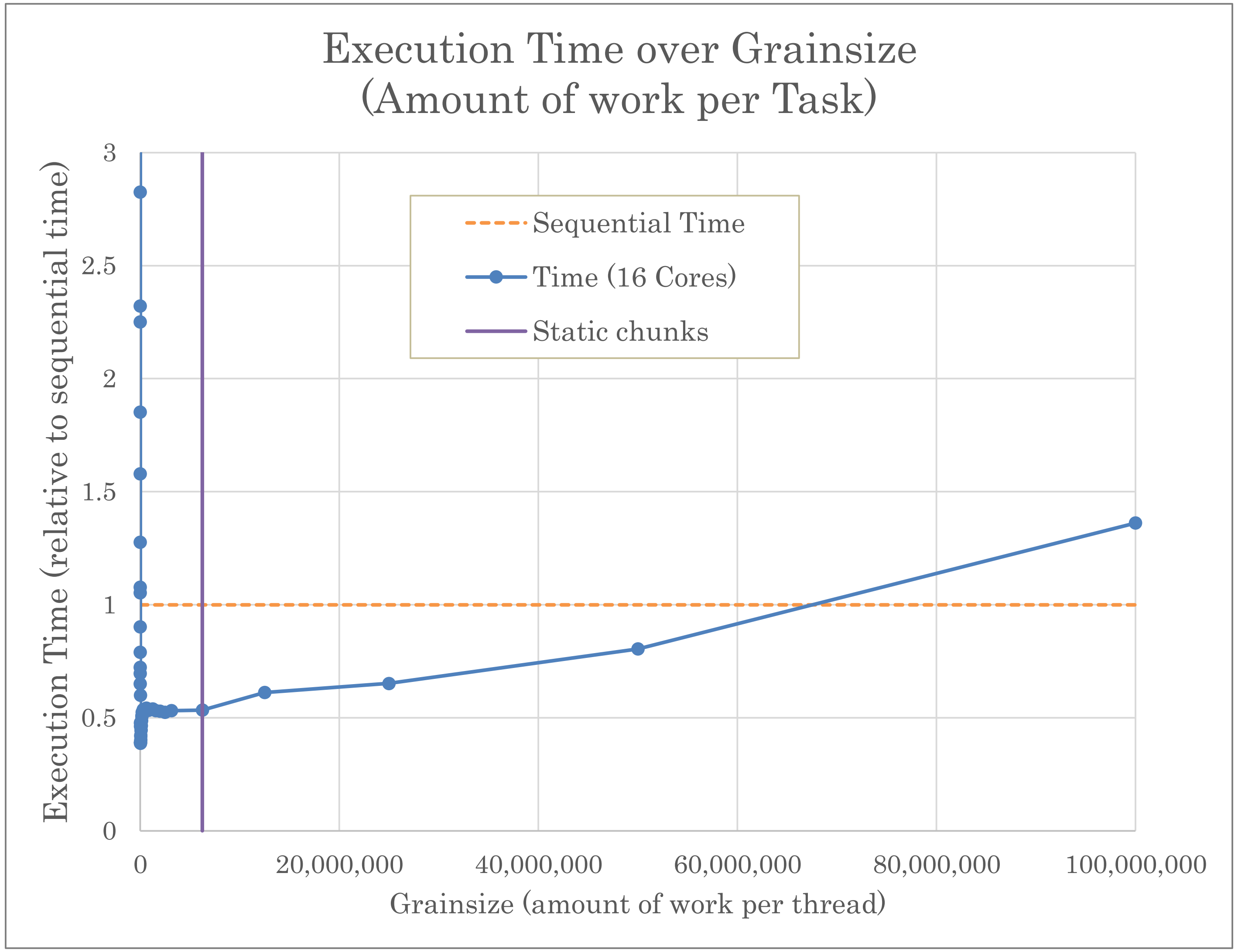
- Starvation
  - Insufficient concurrent work to maintain high utilization of resources
- Latencies
  - Time-distance delay of remote resource access and services
- Overheads
  - Work for management of parallel actions and resources on critical path which are not necessary in sequential variant
- Waiting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

courtesy of www.albrecht-durer.org

STE||AR GROUP


Execution Time over Grainsize
(Amount of work per Task)
Sequential Time
Time (16 Cores)
Static chunks
Execution Time (relative to sequential time)
3
2.5
2
1.5
1
0.5
0
100
1,000
10,000
100,000
1,000,000
10,000,000
100,000,000
Grainsize (amount of work per thread)

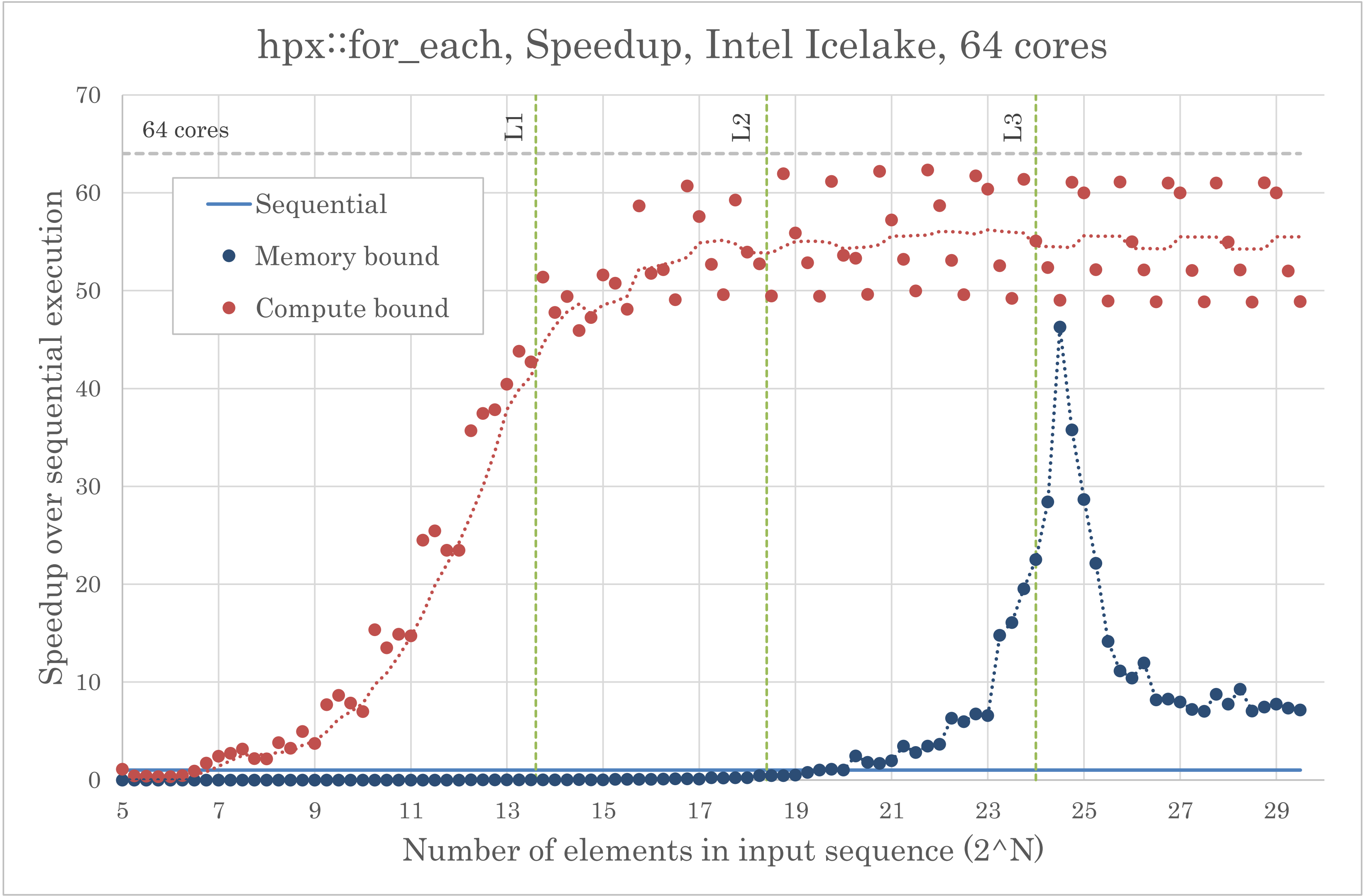
STE||AR GROUP


Execution Time over Grainsize
(Amount of work per Task)
Sequential Time
Time (16 Cores)
Static chunks
Execution Time (relative to sequential time)
Grainsize (amount of work per thread)
0
0.5
1
1.5
2
2.5
3
0
20,000,000
40,000,000
60,000,000
80,000,000
100,000,000

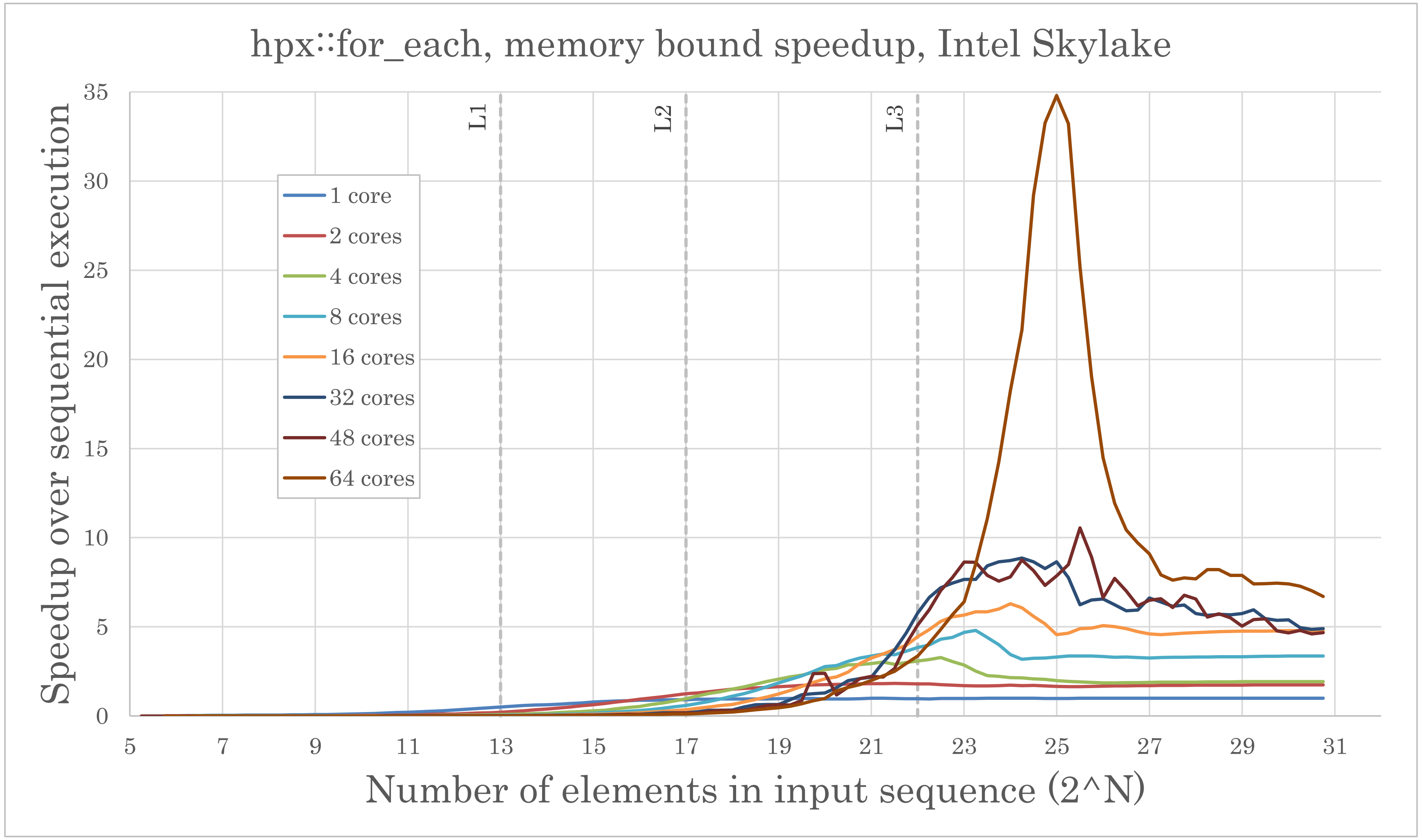
STE||AR GROUP

A Real World Story

STE||AR GROUP


hpx::for_each, Speedup, Intel Icelake, 64 cores
Speedup over sequential execution
Number of elements in input sequence (2^N)
64 cores
L1
L2
L3
Sequential
Memory bound
Compute bound
0
10
20
30
40
50
60
70
5
7
9
11
13
15
17
19
21
23
25
27
29

STE||AR GROUP


hpx::for_each, memory bound speedup, Intel Skylake
Speedup over sequential execution
Number of elements in input sequence (2^N)
0
5
10
15
20
25
30
35
5
7
9
11
13
15
17
19
21
23
25
27
29
31
L1
L2
L3
1 core
2 cores
4 cores
8 cores
16 cores
32 cores
48 cores
64 cores

STE||AR GROUP

Executors

Executors

- Need abstraction of **How**, **When**, and **Where** to execute tasks
  - Need an API abstracting execution:
    - Fire & forget
    - Synchronously/asynchronously
    - Single/multiple tasks
    - Dependency tracking
  - Control parameters of execution
    - Chunk sizes?
    - Number of chunks?
    - What cores to use (where, number)?
    - Thread attributes (annotations, priorities, affinities, etc.)?

Executors

- Executors abstract different task launching infrastructures
  - Synchronization using futures
    - HPX historically uses futures as main means of coordinating
  - Synchronization using sender/receivers (C++26?)
    - C++ standardization focusses on developing an infrastructure for anything related to asynchrony and parallelism
      - P2300: std::execution (senders & receivers)
    - Computational basis for asynchronous programming
    - Current discussions focus on integrating parallel algorithms
- In HPX, all facilities that launch tasks are implemented on top of executors
  - Parallel algorithms (attached to execution policies: `par.on(exec)`)
  - `hpx::async`, `hpx::dataflow`, `hpx::future::then`, etc. (passed directly as additional argument: `hpx::async(exec, f, ...)`)

See: wg21.link/p2300

Executors: Parallel Algorithms

- HPX supports associating an executor with execution policies:

```
// Parallel execution using default executor
std::vector v = {1.0, 2.0, ... };
hpx::for_each(par, v.begin(), v.end(), [](double val) { ... });

// Parallel execution using parallel_executor
hpxex::parallel_executor exec;
hpx::for_each(par.on(exec), v.begin(), v.end(), [](double val) { ... });

// Parallel asynchronous (eager) execution using parallel_executor
future auto f = hpx::for_each(par(task).on(exec), v.begin(), v.end(), [](double val) { ... });
f.get();   // wait for completion

// Parallel execution using sender_executor
hpxexp::sender_executor sr_exec;
hpx::for_each(par.on(sr_exec), v.begin(), v.end(), [](double val) { ... });

// Parallel asynchronous (lazy) execution using sender_executor
sender auto s = hpx::for_each(par(task).on(sr_exec), v.begin(), v.end(), [](double val) { ... });
hpxtt::sync_wait(s);   // start execution and wait for completion
```

STE||AR GROUP

Executors: Parallel Algorithms

- HPX integrates parallel algorithms with senders/receivers

```
auto exec = ex::sender_executor();
auto result =
    hpxexp::just(std::begin(c), std::end(c), [](auto) { ... })
  |  hpx::for_each(par(task).on(exec))
  |  hpxtt::sync_wait();
```

- Nicely integrates with existing Standard, does not require learning new APIs

STE||AR GROUP

Executors

- HPX executors are (small) objects that expose an API supporting launching tasks:
  - `post` : fire & forget execution of given function
  - `sync_execute` : synchronously execute given function
  - `async_execute` : asynchronously execute given function, return awaitable
  - `bulk_async_execute` : asynchronously execute given function N times, return awaitable
  - `bulk_sync_execute` : asynchronously execute given function N times
  - `then_execute` : execute given function after given awaitable is ready
  - `bulk_then_execute` : execute given function N times after given awaitable is ready
- Executors need to minimally implement `async_execute` only
  - Missing functions are emulated

STE||AR GROUP

Executors: async_execute

- Example implementation using futures:

```
template <typename Executor, typename F, typename ... Ts>
auto async_execute(Executor&& exec, F&& f, Ts&&... ts)
{
    hpx::promise<std::invoke_result_t<F, Ts...>> p;
    auto f = p.get_future();
    exec.sched.launch([=, p = std::move(p)]() {     // copy arguments for brevity
        p.set_value(std::invoke(f, ts...));         // assume non-void return value
    });
    return f;
}
```

STE||AR GROUP

Executors: async_execute

- Example implementation using senders/receivers

```
template <typename Executor, typename F, typename ... Ts>
auto async_execute(Executor&& exec, F&& f, Ts&&... ts)
{
    return
        hpxexp::on(exec.sched)
      | hpxexp::then([=]() { return std::invoke(f, ts...); }));
}
```

STE||AR GROUP

Executors: bulk_async_execute

- Example implementation agnostic to underlying execution machinery:

```
template <typename Executor, typename Shape, typename F, typename ... Ts>
auto bulk_async_execute(Executor&& exec, Shape const& shape, F&& f, Ts&&... ts)
{
    std::vector<decltype(async_execute(f, 0, ts...))> results;
    results.reserve(shape);
    for (size_t i : range(0, shape))
        results.push_back(async_execute(exec, f, i, ts...));
    return when_all(results);
}
```

STE||AR GROUP

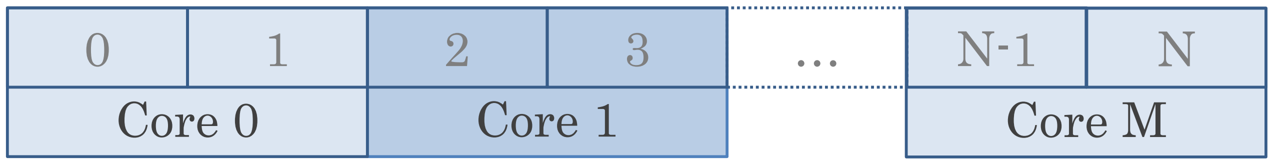
Executors: bulk_async_execute

- Example implementation specific to senders/receivers:

```
template <typename Executor, typename Shape, typename F, typename ... Ts>
auto bulk_async_execute(Executor&& exec, Shape const& shape, F&& f, Ts&&... ts)
{
    return
        hpxexp::on(exec.sched)
      | hpxexp::bulk(shape, [=](auto idx) { std::invoke(f, idx, ts...); });
}
```

Parallelize Loops: Executors

Sequence of elements:



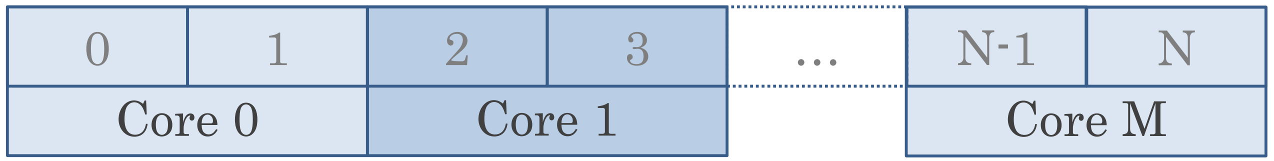
```
std::vector<int> d = {...};
for_each(par, d.begin(), d.end(), [](int val) {...});
```

```
template <typename Iterator, typename F>
auto for_each(parallel_policy policy, Iterator begin, Iterator end, F f)
{
    auto num_chunks = calculate_number_of_chunks(policy, begin, end);
    auto chunk_size = (end - begin) / num_chunks;                              // assume: cleanly divisible
    return wait_all(
        bulk_async_execute(
            policy.executor(), num_chunks,
            [=](size_t idx) {
                auto start_idx = chunk_size * idx;
                std::for_each(begin + start_idx, begin + start_idx + chunk_size, f);  // sequential execution of chunks
            }));
}
```

STE||AR GROUP

Parallelize Loops: NUM_CHUNKS

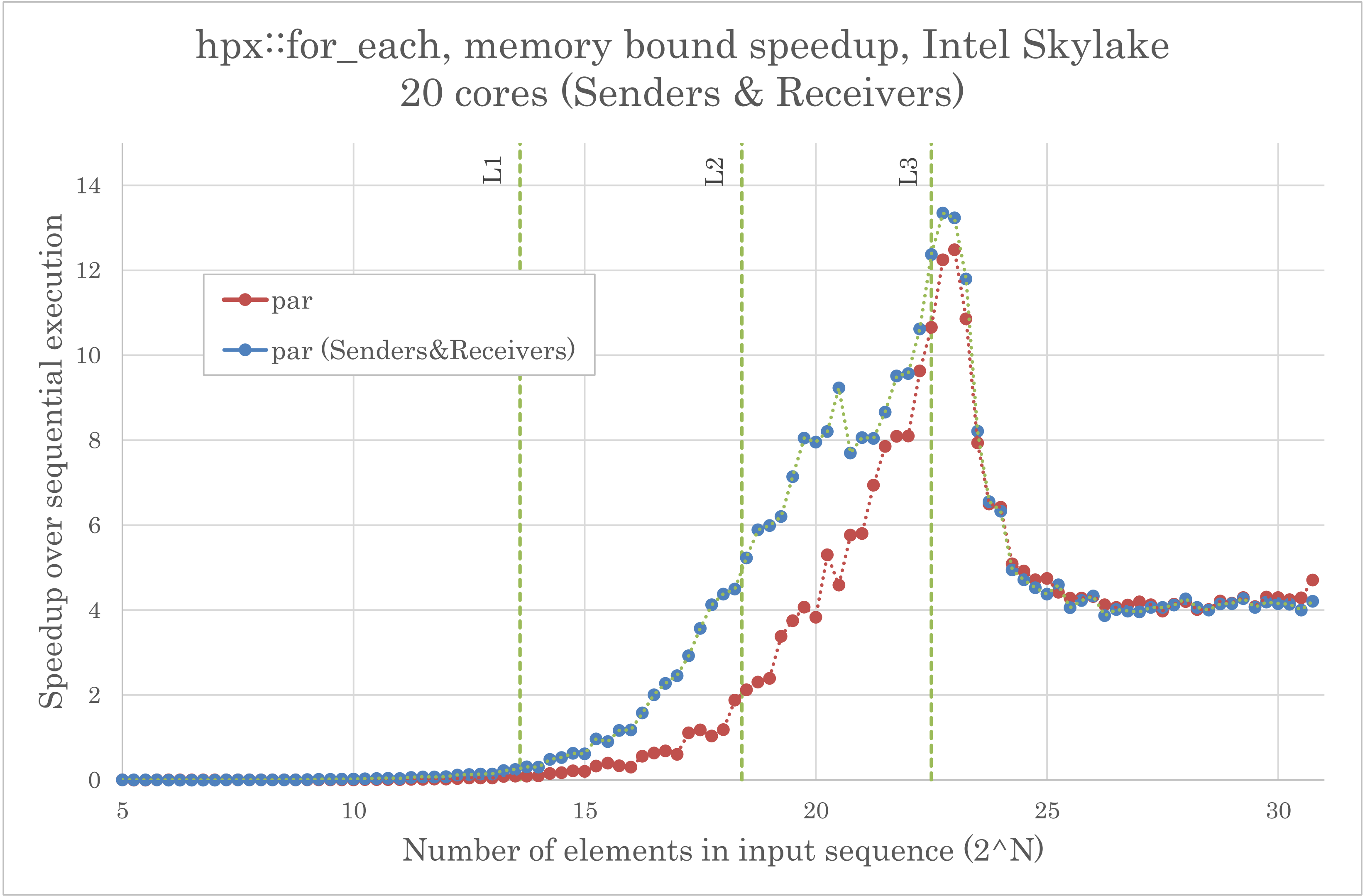
Sequence of elements:



```
std::vector<int> d = {...};
hpx::for_each(with_number_of_chunks(par, NUM_CHUNKS), d.begin(), d.end(), [](int val) {...});
```

```
template <typename ExPolicy, typename Iterator, typename F>
auto for_each(ExPolicy&& policy, Iterator begin, Iterator end, F f)
{
    auto num_chunks = calculate_number_of_chunks(policy, begin, end);             // extract NUM_CHUNKS if given
    auto chunk_size = (end - begin) / num_chunks;                                  // assume: cleanly divisible
    return bulk_async_execute(
        policy.executor(), num_chunks,
        [=](size_t idx) {
            auto start_idx = chunk_size * idx;
            hpx::for_each(begin + start_idx, begin + start_idx + chunk_size, f);  // sequential execution of chunks
        });
}
```

STE||AR GROUP

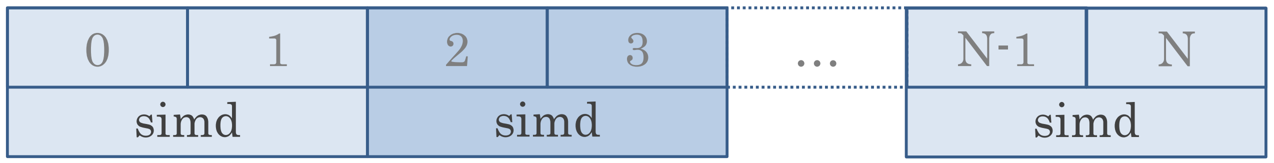

hpx::for_each, memory bound speedup, Intel Skylake
20 cores (Senders & Receivers)
par
par (Senders&Receivers)
L1
L2
L3
Speedup over sequential execution
0
2
4
6
8
10
12
14
5
10
15
20
25
30
Number of elements in input sequence (2^N)

STE||AR GROUP

Explicit Vectorization

Vectorize Loops (explicitly)

Sequence of elements (trivial types):



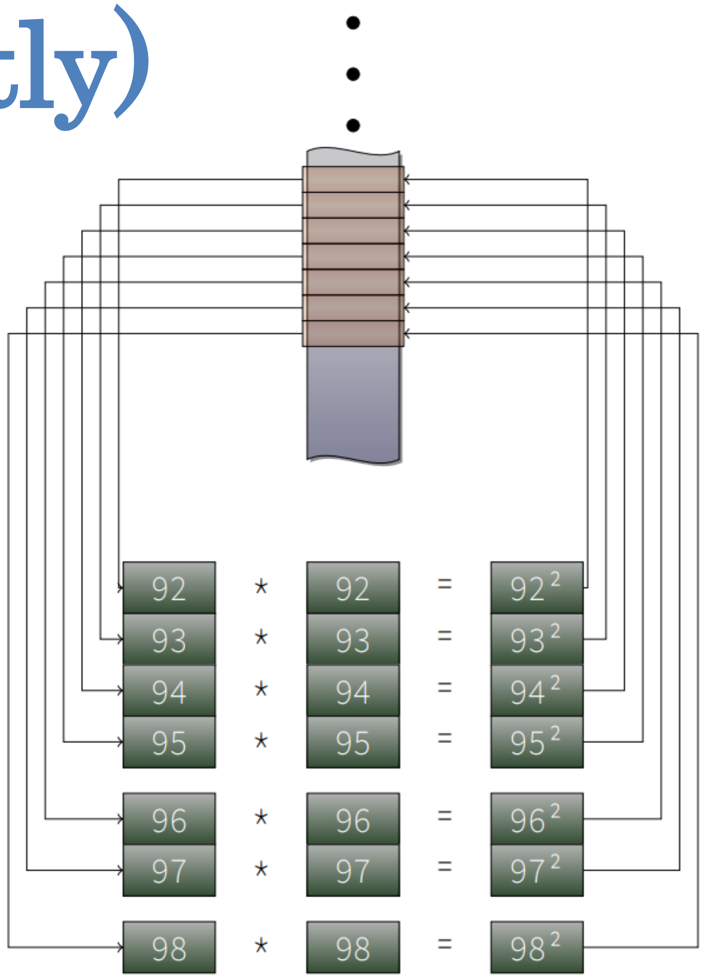
```
std::vector<int> d = {...};
std::for_each(stdexp::simd, d.begin(), d.end(), [](auto val) {...});
```

```
template <typename Iterator, typename F>
void for_each(stdexp::simd_policy, Iterator b, Iterator e, F f)
{
    using V = stdexp::simd<Iterator::value_type>;
    for (/**/; std::distance(b, e) != 0; b += V::size()) {        // Iterator is assumed to be contiguous
        V tmp(std::addressof(*first), aligned);
        f(tmp);
        if constexpr (is_function_argument_mutable_v<F, V>)
            store(tmp, std::addressof(*first), aligned);
    }
}
```

See: wg21.link/p0350

Vectorize Loops (explicitly)

- Provided lambda is called with a `stdexp::simd` type instance instead of a single value (see Parallelism TS V2)
- The `stdexp::simd` type has operators overloaded to make code transition seamless
- Iterator should be contiguous access
  - Otherwise bad things may happen
- Parallel algorithms load underlying sequence into vector register types before invoking loop body
- HPX implements `simd` and `par_simd` policies and their asynchronous variations



Linear Algebra

P1673: A free function linear algebra interface based on the BLAS

STE||AR GROUP

Linear Algebra

- P1673: A free function linear algebra interface based on the BLAS
  - Proposes a C++ Standard Library dense linear algebra interface

```
std::vector x_vec = { 1.0, 2.0, 3.0, ... };   // size: N

std::mdspan x(x_vec.data(), N);               // as of C++23

stdexp::linalg::scale(2.0, x);                // sequential: x = 2.0 * x
stdexp::linalg::scale(stdex::par, 3.0, x);    // parallel: x = 3.0 * x
```

See: wg21.link/p1673

Linear Algebra

- Adding (optional) execution policies to all API functions
  - Allows for customization
  - Reference implementation available: https://github.com/kokkos/stdBLAS
    - CPU based implementation
    - Kokkos based implementation
    - HPX based implementation (under development)

Linear Algebra

- P1673: A free function linear algebra interface based on the BLAS
  - Proposes a C++ Standard Library dense linear algebra interface

```
std::vector x_vec = { 1.0, 2.0, 3.0, ... };   // size: N

std::mdspan x(x_vec.data(), N);               // as of C++23

stdexp::linalg::scale(2.0, x);                // sequential: x = 2.0 * x
stdexp::linalg::scale(stdex::par, 3.0, x);    // parallel: x = 3.0 * x

stdexp::linalg::scale(hpx::par, 3.0, x);      // parallel (HPX): x = 3.0 * x
stdexp::linalg::scale(hpx::par_simd, 3.0, x); // parallel and vectorized: x = 3.0 * x
```

See: wg21.link/p1673

Linear Algebra: linalg::scale (1D)

- Exemplar 1D implementation of policy-based `linalg::scale`

```
std::vector<double> data = { 1.0, 2.0. 3.0, ... };
std::linalg::scale(par, 4.0, std::mdspan(data.data(), data.size()));
```

```
template <typename ExPolicy, typename Scalar, typename MdSpan>
auto scale(ExPolicy&& policy, Scalar alpha, MdSpan x)
{
    if constexpr (!supports_vectorization_v<ExPolicy> ||
                  !allow_vectorization_v<MdSpan>) {            // more conditions may apply
        // fall back to non-vectorized execution
        return hpx::for_each(to_non_simd(policy),
            mditerator_begin(x), mditerator_end(x),
            [&](auto& v) { v*= alpha; });
    } else {
        // possibly explicitly vectorized execution
        return hpx::for_each(policy,
            mditerator_begin(x), mditerator_end(x),
            [&](auto& v) { v *= alpha; });
    }
}
```

STE||AR GROUP

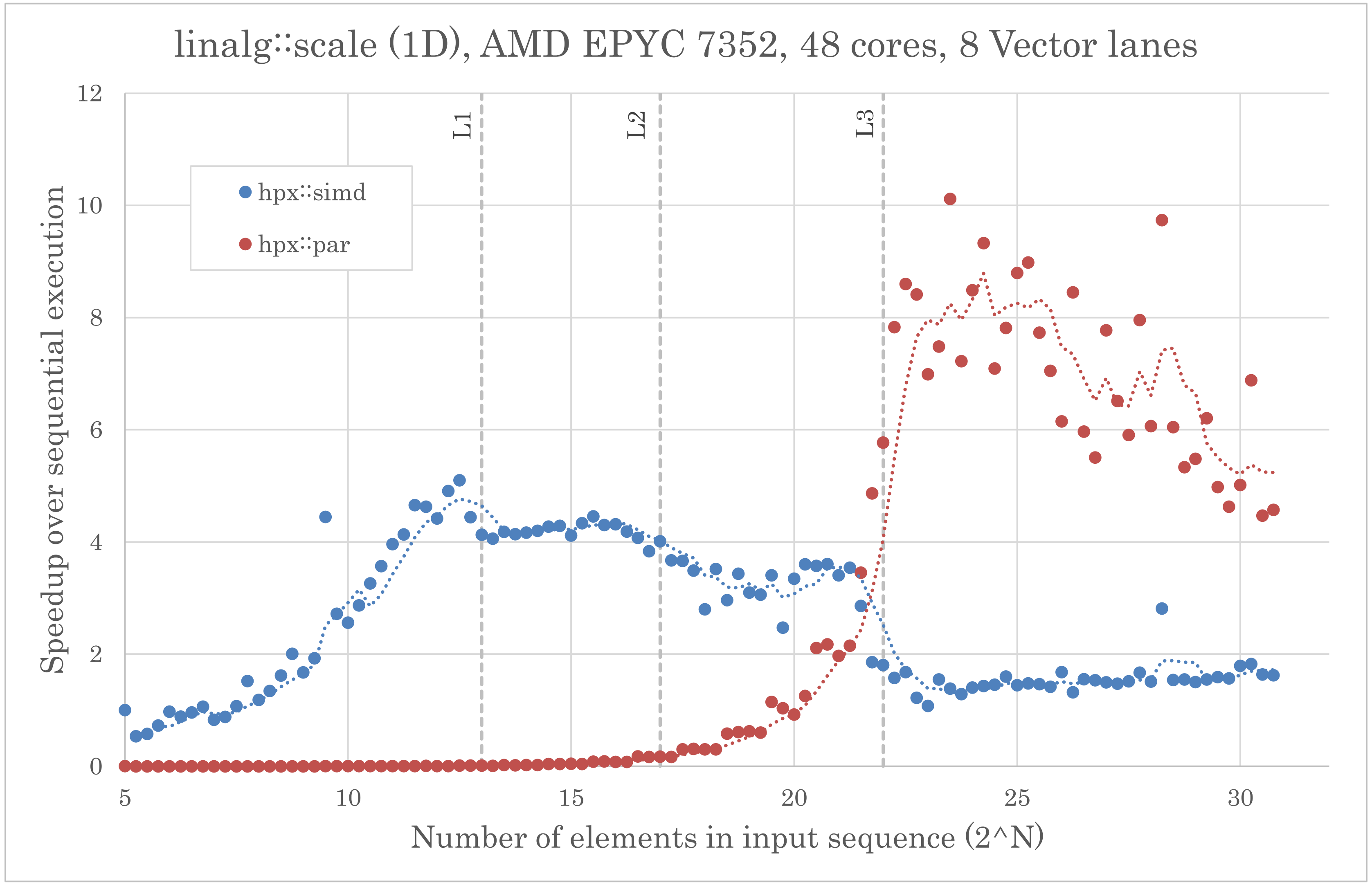
Linear Algebra: linalg::scale (2D)

- Exemplar 2D implementation of policy-based `linalg::scale`

```
std::vector<double> data = { 1.0, 2.0. 3.0, ... };      // size: Nx * Ny
std::linalg::scale(par_simd, 4.0, std::mdspan(data.data(), Nx, Ny));
```

```
template <typename ExPolicy, typename Scalar, typename MdSpan>
auto scale(ExPolicy&& policy, Scalar alpha, MdSpan x)
{
    return hpx::for_each(to_non_simd(policy),       // allow for outer loop to be parallelized
        mditerator_begin(x), mditerator_end(x),
        [&](auto&& sub_x)
        {
            hpx::for_each(to_seq(policy),             // assume inner loop is vectorizable
                mditerator_begin(sub_x), mditerator_end(sub_x),
                [&](auto& v) { v *= alpha; });
        });
}
```

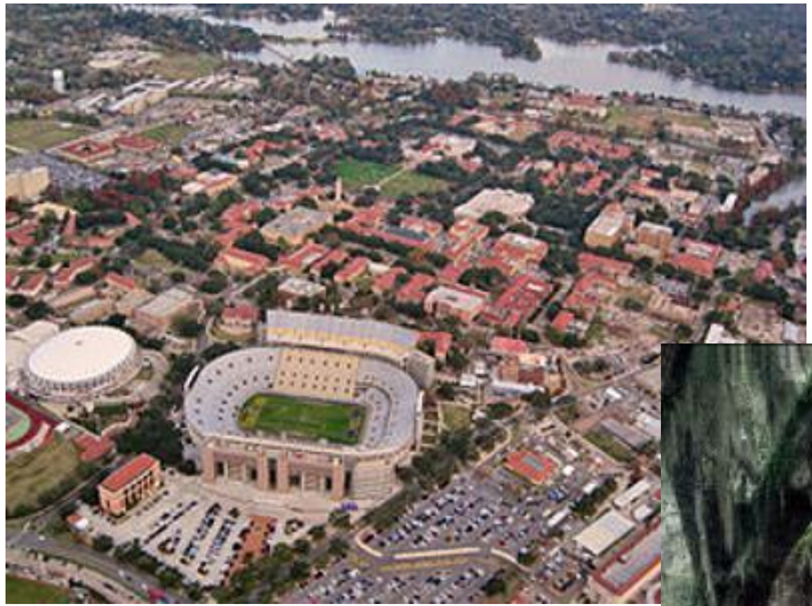
STE||AR GROUP


linalg::scale (1D), AMD EPYC 7352, 48 cores, 8 Vector lanes
Speedup over sequential execution
Number of elements in input sequence (2^N)
● hpx::simd
● hpx::par
L1
L2
L3
0
2
4
6
8
10
12
5
10
15
20
25
30

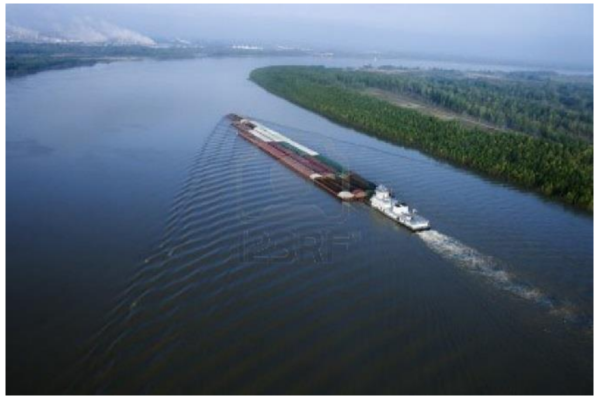
STE||AR GROUP

Conclusions

- Using execution policies for API functions that should allow for customization of execution is a good choice
  - More customization is needed, though
    - Chunking, execution environment, number of cores, etc.
  - Having means of running things asynchronously is important
    - Big hopes for senders/receivers
- Adding higher-level APIs that integrate well with senders/receivers is a must
  - Senders/receivers are fairly low level facilities with a steep learning curve
- Currently new APIs for parallel algorithms in the context of sender/receivers are being discussed
  - We believe that no new APIs are necessary



LSU CENTER FOR COMPUTATION & TECHNOLOGY



STE||AR GROUP