STE||AR
stellar.cct.lsu.edu

Managing Asynchrony in C++

VINAY AMATYA (VAMATYA@CCT.LSU.EDU)

HARTMUT KAISER (HKAISER@BOOST.ORG)

The Venture Point

TECHNOLOGY DEMANDS NEW RESPONSE

Technology Demands new Response

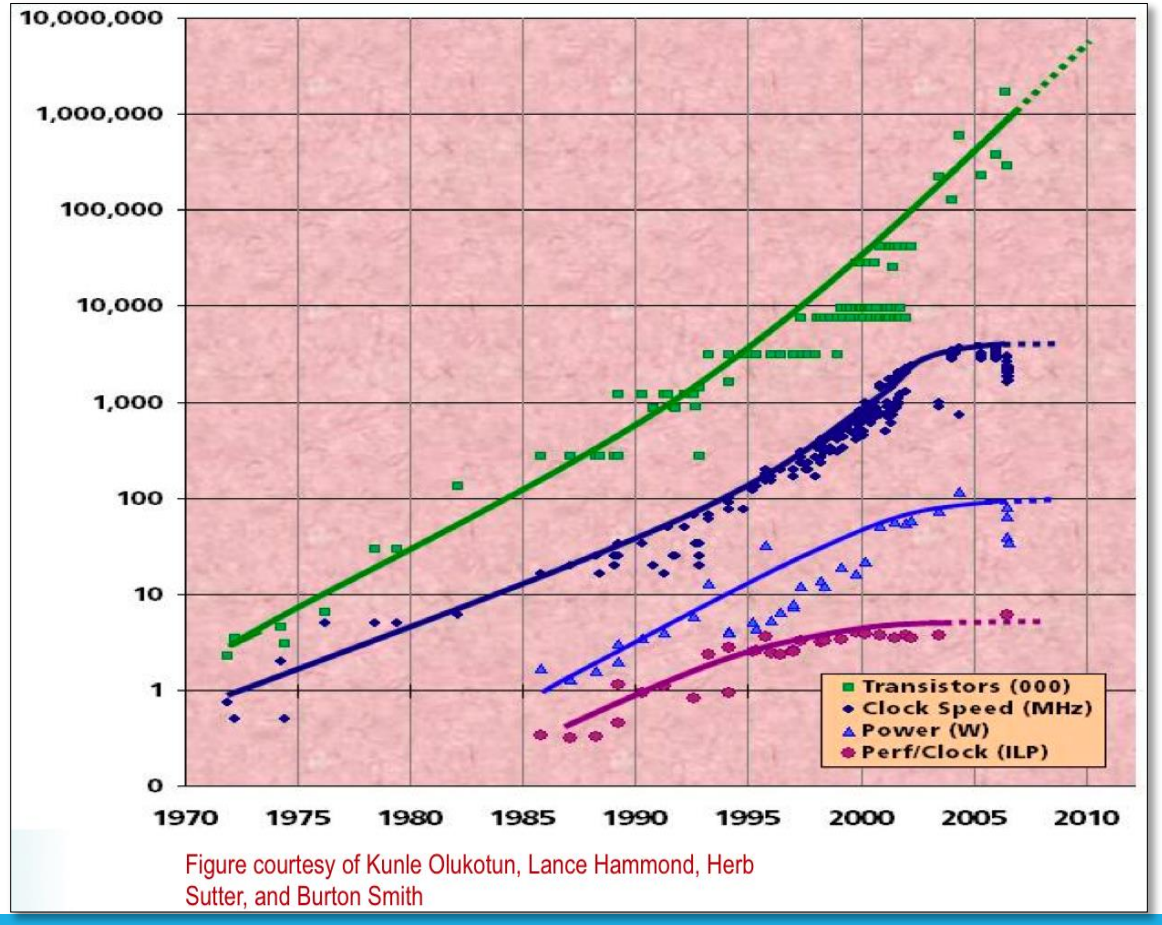


Figure courtesy of Kunle Olukotun, Lance Hammond, Herb Sutter, and Burton Smith

Technology Demands new Response

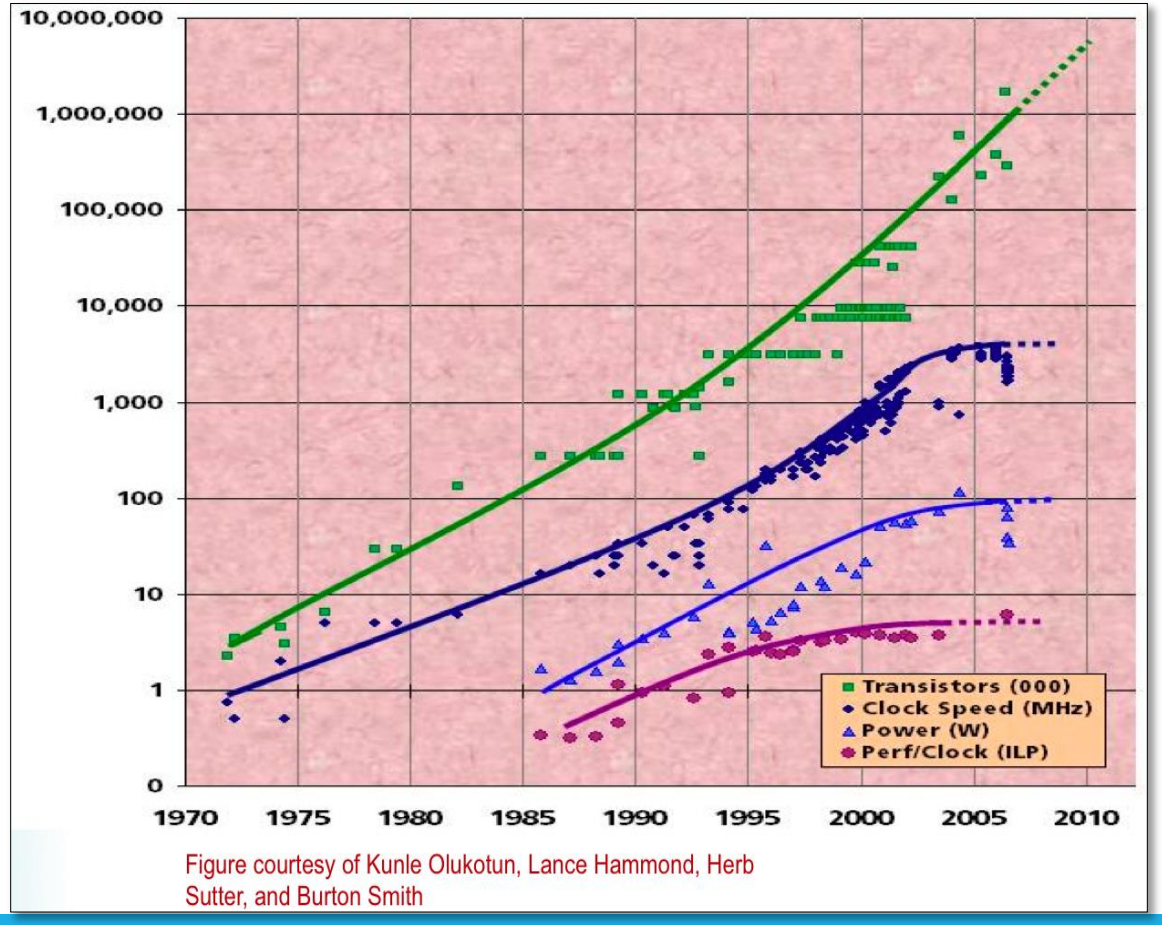
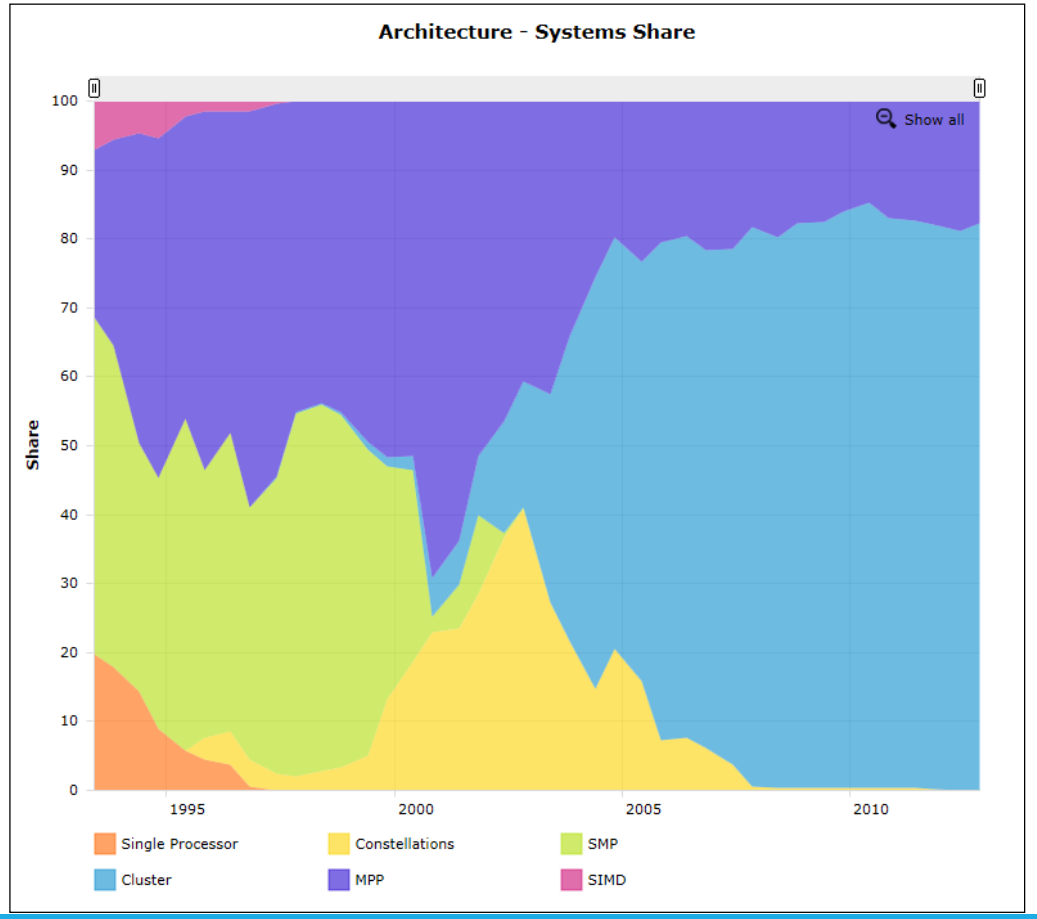


Figure courtesy of Kunle Olukotun, Lance Hammond, Herb Sutter, and Burton Smith



Technology Demands new Response

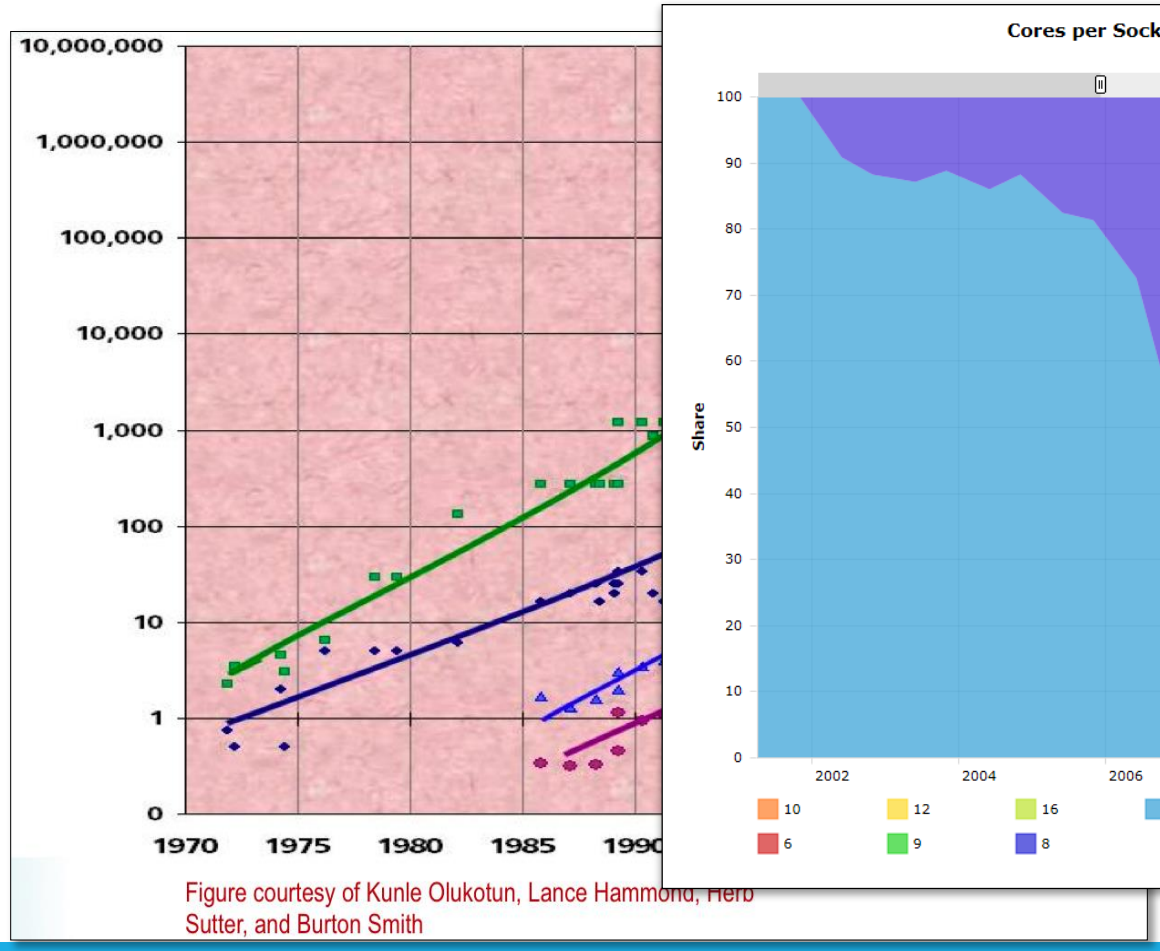
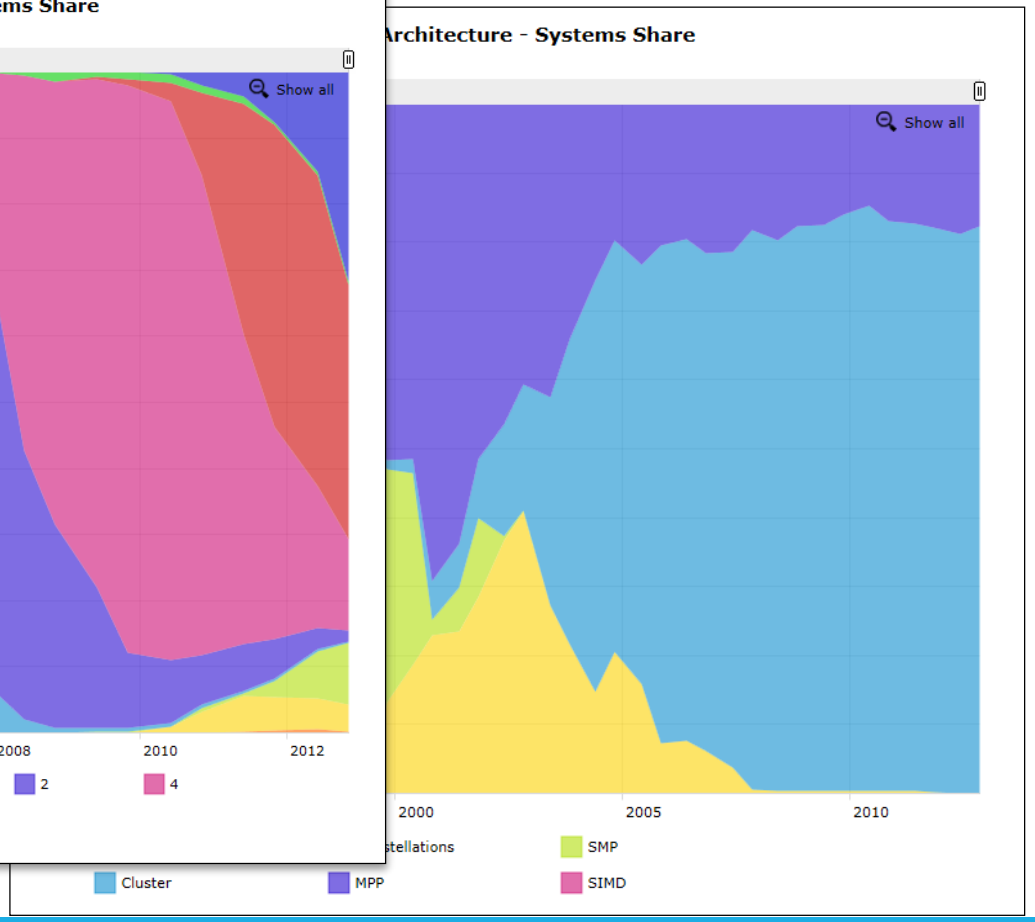


Figure courtesy of Kunle Olukotun, Lance Hammond, Herb Sutter, and Burton Smith



Technology Demands new Response

Peak performance: 1.17 PetaFLOPs

112,896 computing cores (18,816 2.6 GHz six-core AMD Opteron processors)

Amdahl's Law (Strong Scaling)

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

S: Speedup

P: Proportion of parallel code

N: Number of processors

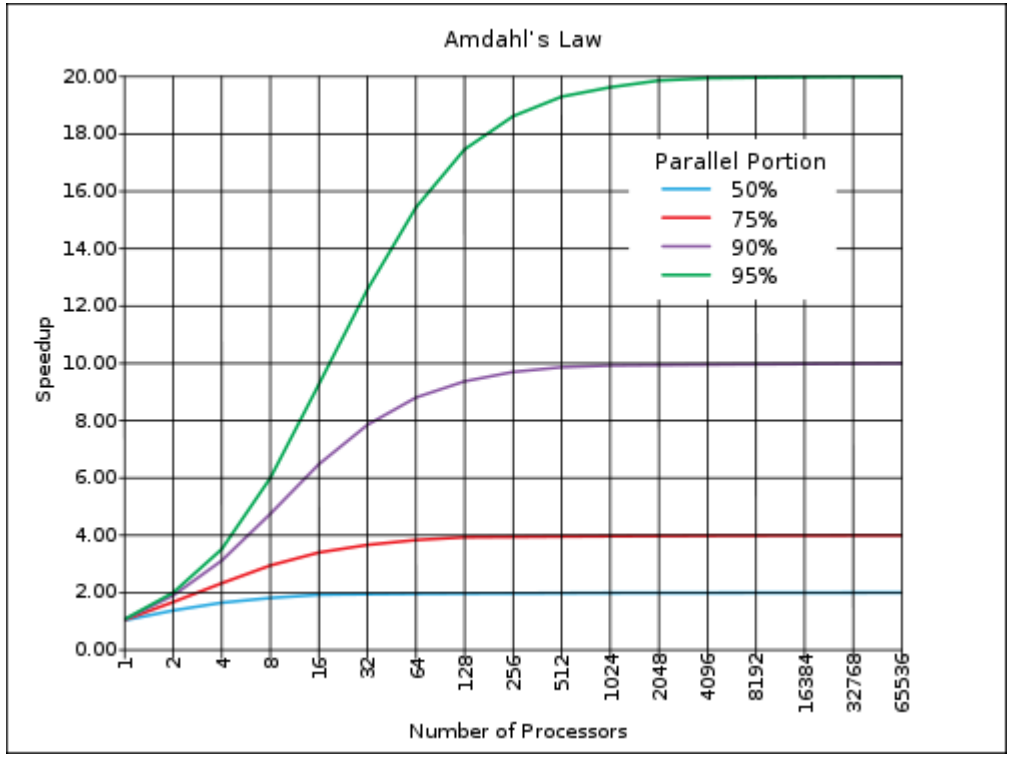


Figure courtesy of Wikipedia (http://en.wikipedia.org/wiki/Amdahl's_law)

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

The 4 Horsemen of the Apocalypse: SLOW

Starvation

- Insufficient concurrent work to maintain high utilization of resources

Latencies

- Time-distance delay of remote re… services

Overhea…

- …
- …ot necessary

Wa… Contention resolution

- De…ys due to lack of availability of oversubscribed shared resources

Impose upper bound on both weak and strong scaling


courtesy of www.albrecht-durer.org

The Challenges

We need to find a usable way to fully parallelize the applications

Goals are

- Defeat The Four Horsemen
- Provide manageable paradigms for handling parallelism
- Expose asynchrony to the programmer without exposing concurrency
- Make data dependencies explicit, hide notion of ‘thread’, ‘communication’, and ‘data distribution’

Runtime Systems

THE NEW DIMENSION

HPX – A General Purpose Runtime System

All examples in this talk are based on HPX

A general purpose runtime system for applications of any scale

- http://stellar.cct.lsu.edu/
- https://github.com/STEllAR-GROUP/hpx/

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.

Is published under Boost license and has an open, active, and thriving developer community.

HPX – A General Purpose Runtime System

Governing principles

- Active global address space (AGAS) instead of PGAS
- Message driven instead of message passing
- Lightweight control objects instead of global barriers
- Latency hiding instead of latency avoidance
- Adaptive locality control instead of static data distribution
- Moving work to data instead of moving data to work
- Fine grained parallelism of lightweight threads instead of Communicating Sequential Processes (CSP/MPI)

HPX – The API

Fully asynchronous

- All possibly remote operations are asynchronous by default
  - ‘Fire & forget’ semantics (result is not available)
  - ‘Pure’ asynchronous semantics (result is available via hpx::future)
- Composition of asynchronous operations (N3634)
  - hpx::when_all, hpx::when_any, hpx::when_n
  - hpx::future::then(f)
- Can be used ‘synchronously’, but does not block
  - Thread is suspended while waiting for result
  - Other useful work is performed transparently

HPX – The API

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

- std::thread → hpx::thread
- std::mutex → hpx::mutex
- std::future → hpx::future
- std::async → hpx::async
- std::bind → hpx::bind
- std::function → hpx::function
- std::tuple → hpx::tuple
- std::any → hpx::any (N3508)
- std::cout → hpx::cout
- etc.

HPX – The API

Fully move enabled (using Boost.Move)

- hpx::bind, hpx::function, hpx::tuple, hpx::any

Fully type safe remote operation

- Extends the notion of a ‘callable’ to remote case (actions)
- Everything you can do with functions is possible with actions as well

Data types are usable in remote contexts

- Can be sent over the wire (hpx::bind, hpx::function, hpx::any)
- Can be used with actions (hpx::async, hpx::bind, hpx::function)

HPX – The API

R f(p...)	Synchronous (return R)	Asynchronous (return future<R>)	Fire & Forget (return void)
Functions (direct)	`f(p…)`	`async(f, p…)`	`apply(f, p…)`
Functions (lazy)	`bind(f, p…)(…)`	`async(bind(f, p…), …)`	`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(f, 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…), …)`

C++

C++ Library

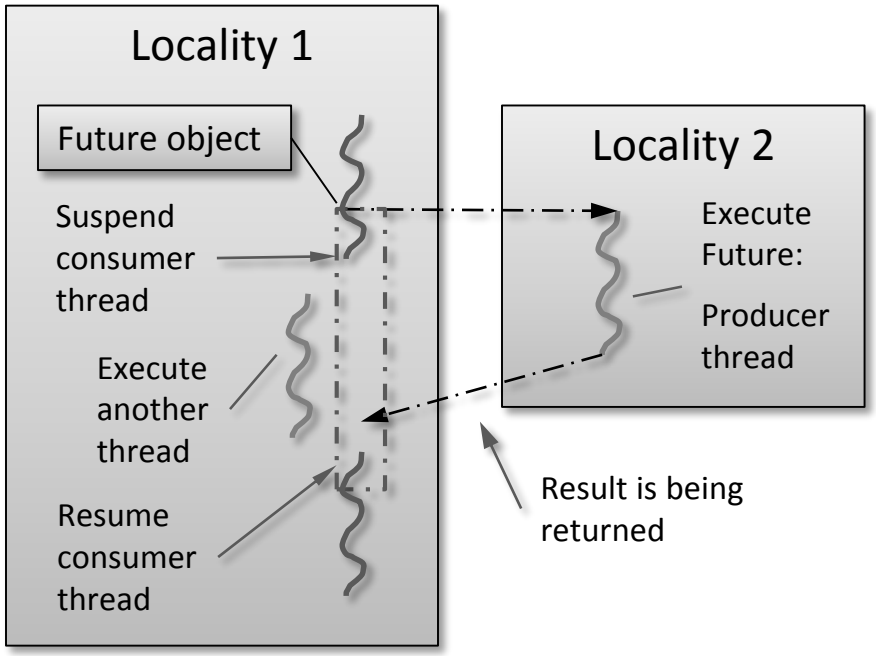
HPX

The Future

A CLOSER LOOK

What is a (the) future

A (std) 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
}
```

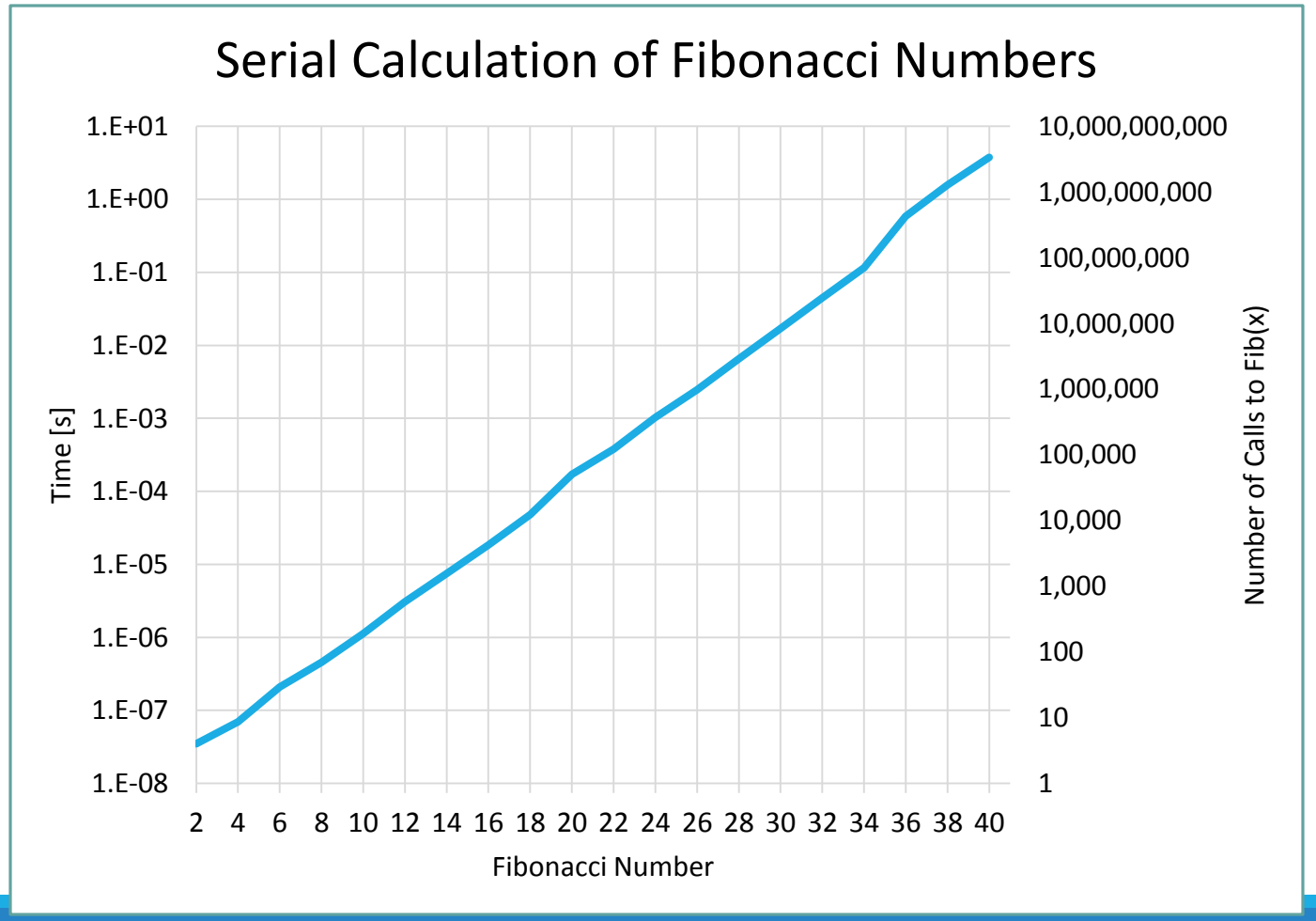
Stupidest Way to Calculate Fibonacci Numbers

Synchronous way:

```
// watch out: O(2^n)
int fibonacci_serial(int n)
{
    if (n < 2) return n;
    return fibonacci_serial(n-1) + fibonacci_serial(n-2);
}

cout << fibonacci_serial(10) << endl;    // will print: 55
```

Stupidest Way to Calculate Fibonacci Numbers

Complexity: $O(2^n)$



Stupidest Way to Calculate Fibonacci Numbers

Computational complexity is $O(2^n)$ – alright, however

This algorithm is representative for a whole class of applications

- Tree based recursive data structures
  - Adaptive Mesh Refinement – important method for wide range of physics simulations
  - Game theory
- Graph based algorithms
  - Breadth First Search

Characterized by very tightly coupled data dependencies between calculations

- But fork/join semantics make it simple to reason about parallelization

Let's spawn a new thread for every other sub tree on each recursion level

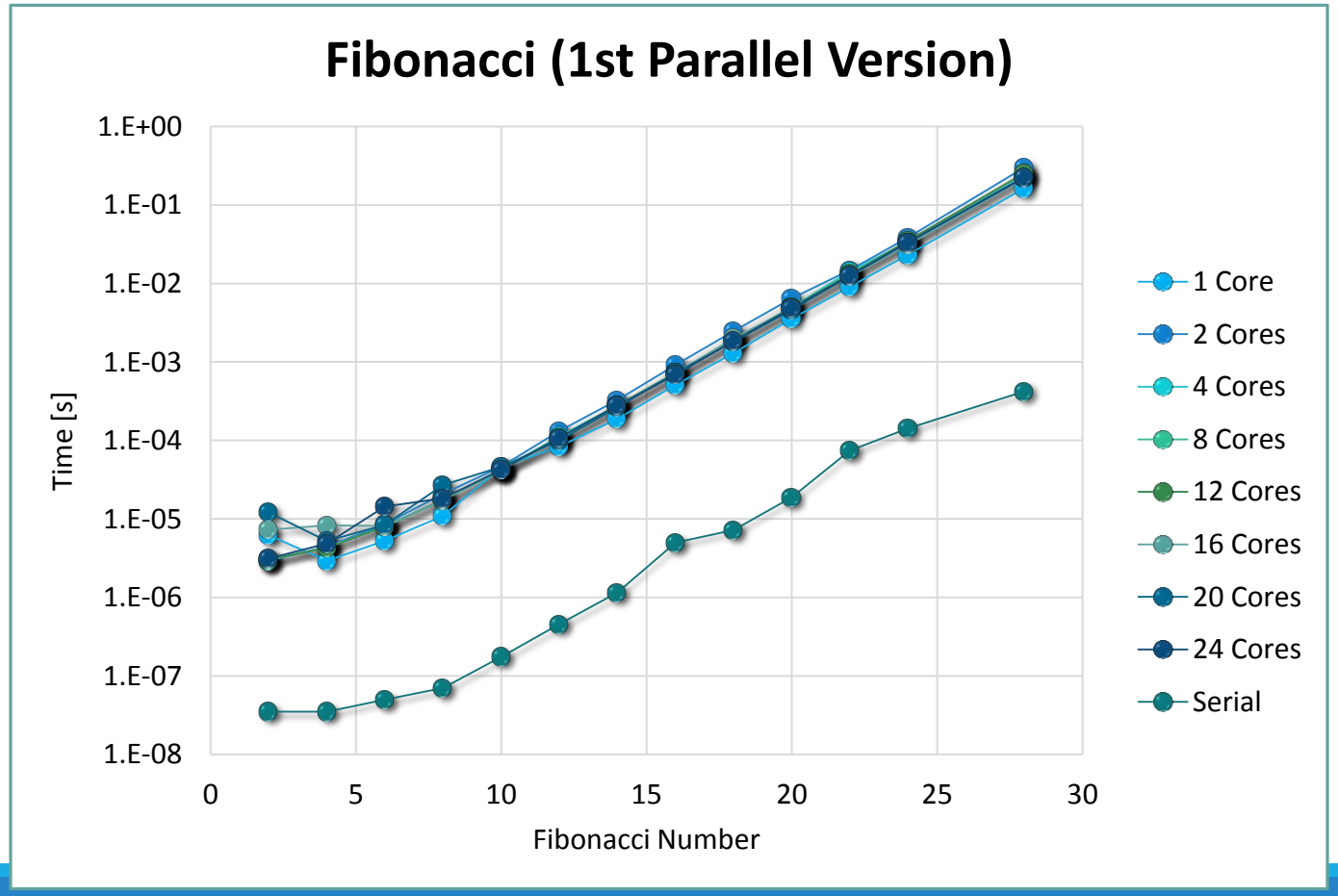
Explicit Asynchrony

Let's Parallelize It! What could be easier?

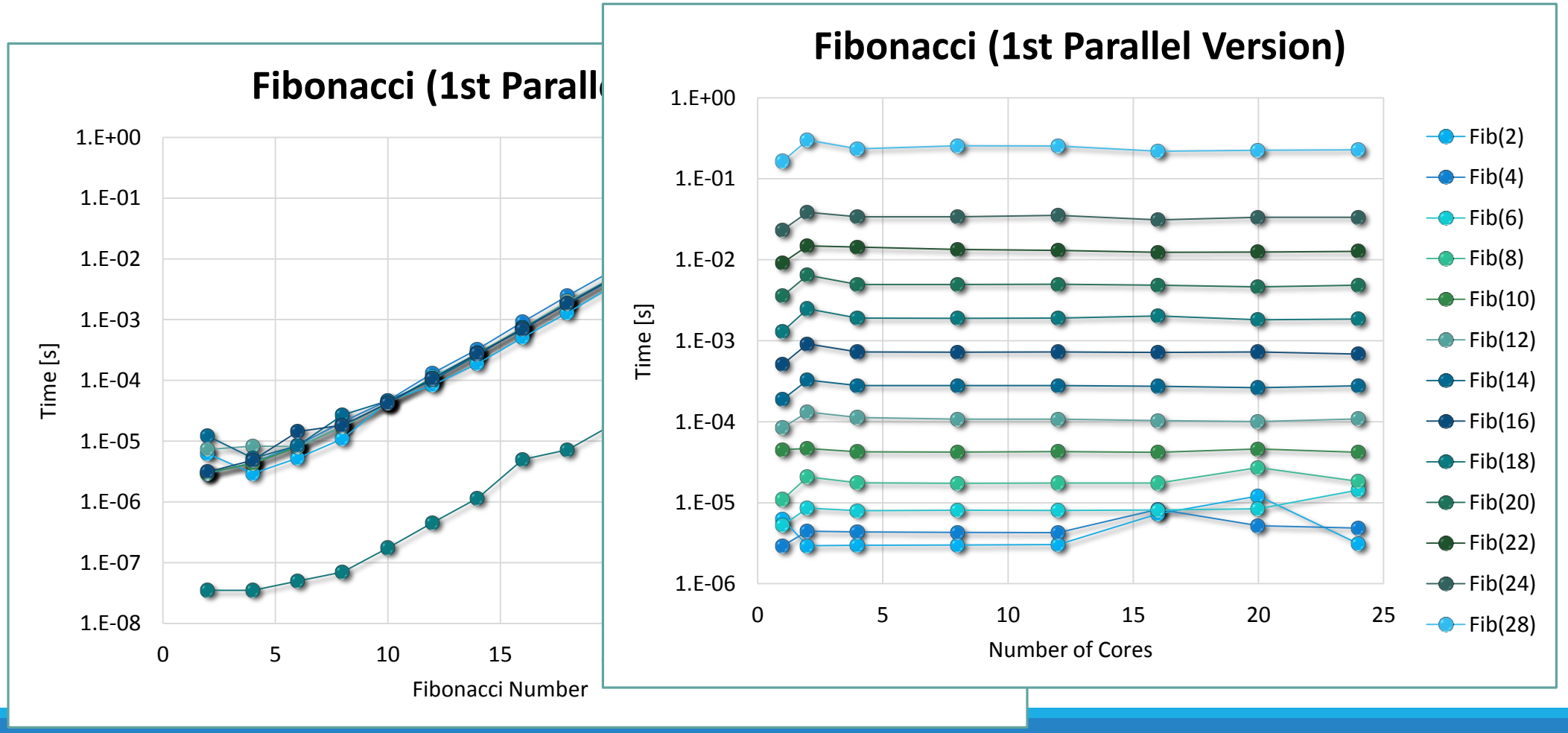
Calculate Fibonacci numbers in parallel (1st attempt)

```
uint64_t fibonacci(uint64_t n)
{
    // if we know the answer, we return the value
    if (n < 2) return n;

    // asynchronously delay-calculate one of the sub-terms
    future<uint64_t> f = async(launch::deferred, &fibonacci, n-1);

    // synchronously calculate the other sub-term
    uint64_t r = fibonacci(n-2);

    // wait for the future and calculate the result
    return f.get() + r;
}
```

Let's Parallelize It! What could be easier?



Let's Parallelize It! What could be easier?



Let's Parallelize It! Could it be easier?

Parallel calculation (1st attempt), why is it slow? Why doesn't it scale?

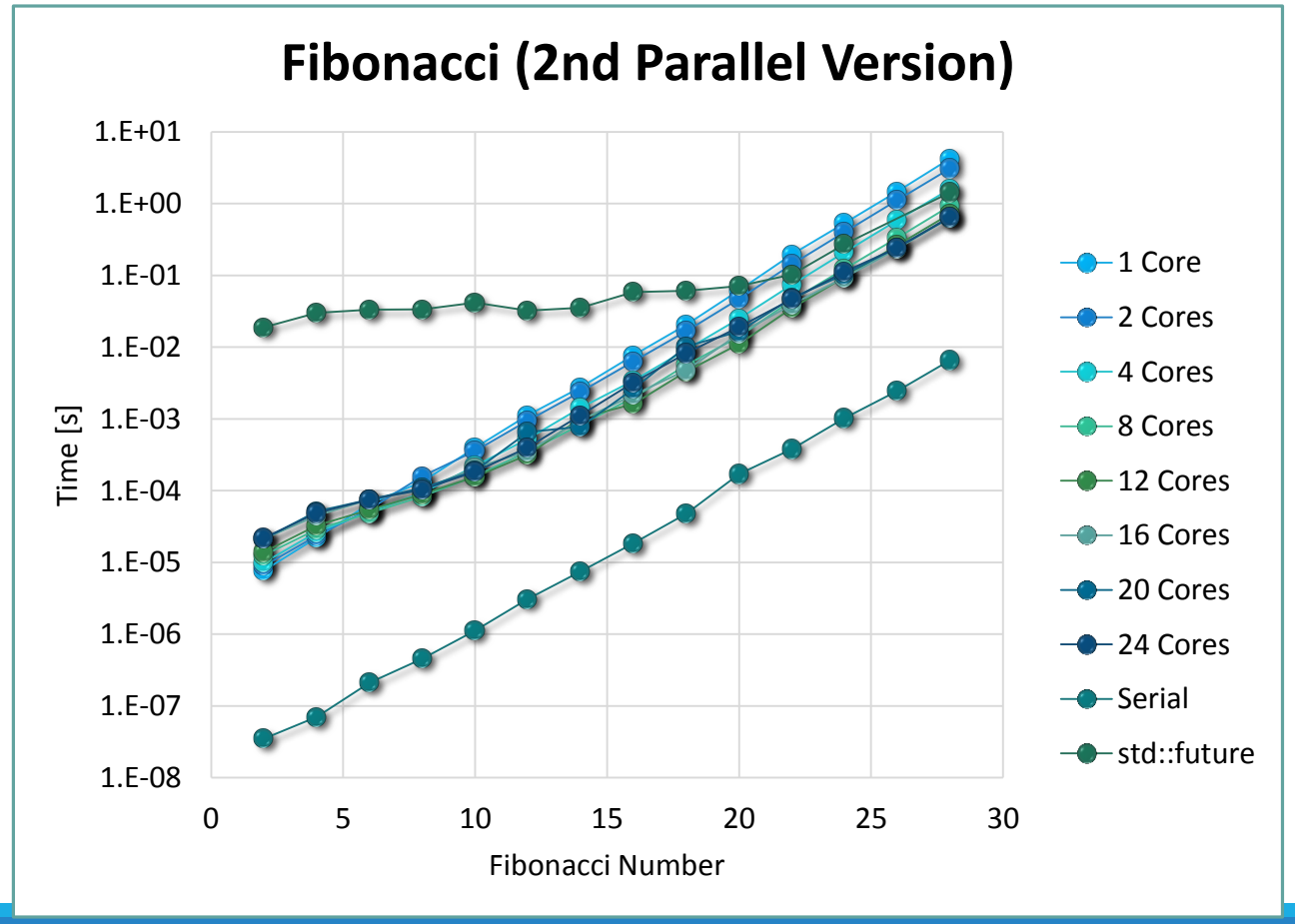
```
uint64_t fibonacci(uint64_t n)
{
    // if we know the answer, we return the value
    if (n < 2) return n;

    // asynchronously delay-calculate one of the sub-terms
    future<uint64_t> f = async(launch::deferred, &fibonacci, n-1);

    // synchronously calculate the other sub-term
    uint64_t r = fibonacci(n-2);

    // wait for the future and calculate the result
    return f.get() + r;
}
```

Let's Parallelize It - 2nd Attempt What could be easier?

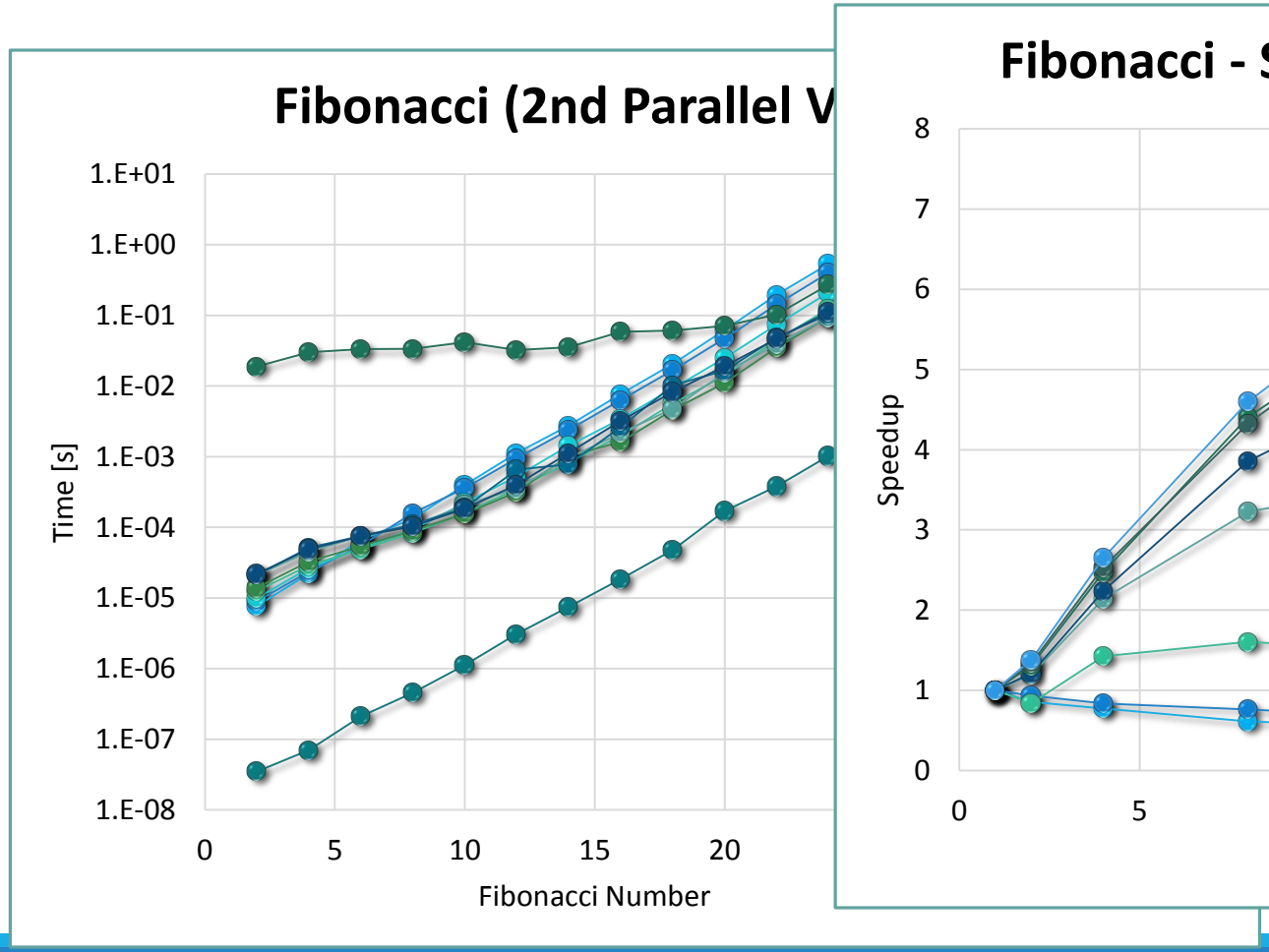
Calculate Fibonacci numbers in parallel (2nd attempt)

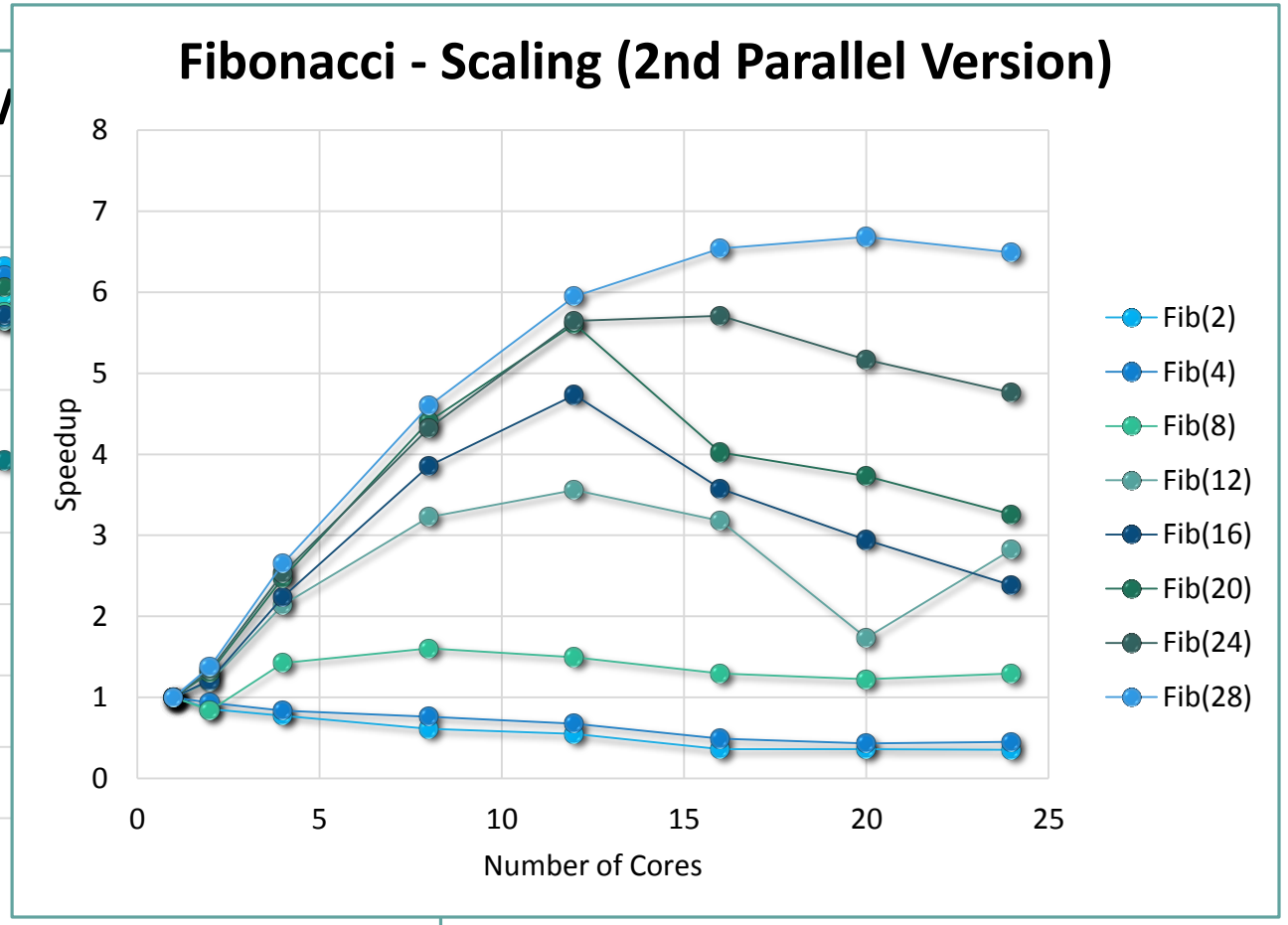
```
uint64_t fibonacci(uint64_t n)
{
    // if we know the answer, we return the value
    if (n < 2) return n;

    // asynchronously calculate one of the sub-terms
    future<uint64_t> f = async(launch::async, &fibonacci, n-1);

    // synchronously calculate the other sub-term
    uint64_t r = fibonacci(n-2);

    // wait for the future and calculate the result
    return f.get() + r;
}
```

Let's Parallelize It - 2nd Attempt
What could be easier?



Let's Parallelize It - 2nd Attempt
What could be easier?





Let's Parallelize It - 2nd Attempt What could be easier?

What's wrong? While it scales it is still 100 times slower than the serial execution

Creates a new future for each invocation of fibonacci() (spawns an HPX thread)

- Millions of threads with minimal work each
- Overheads of thread management (creation, scheduling, execution, deletion) are much larger than the amount of useful work
  - Future overheads: ~1µs (Thread overheads: ~400ns)
  - Useful work: ~50ns

Let's introduce the notion of granularity of work (grain size of work)

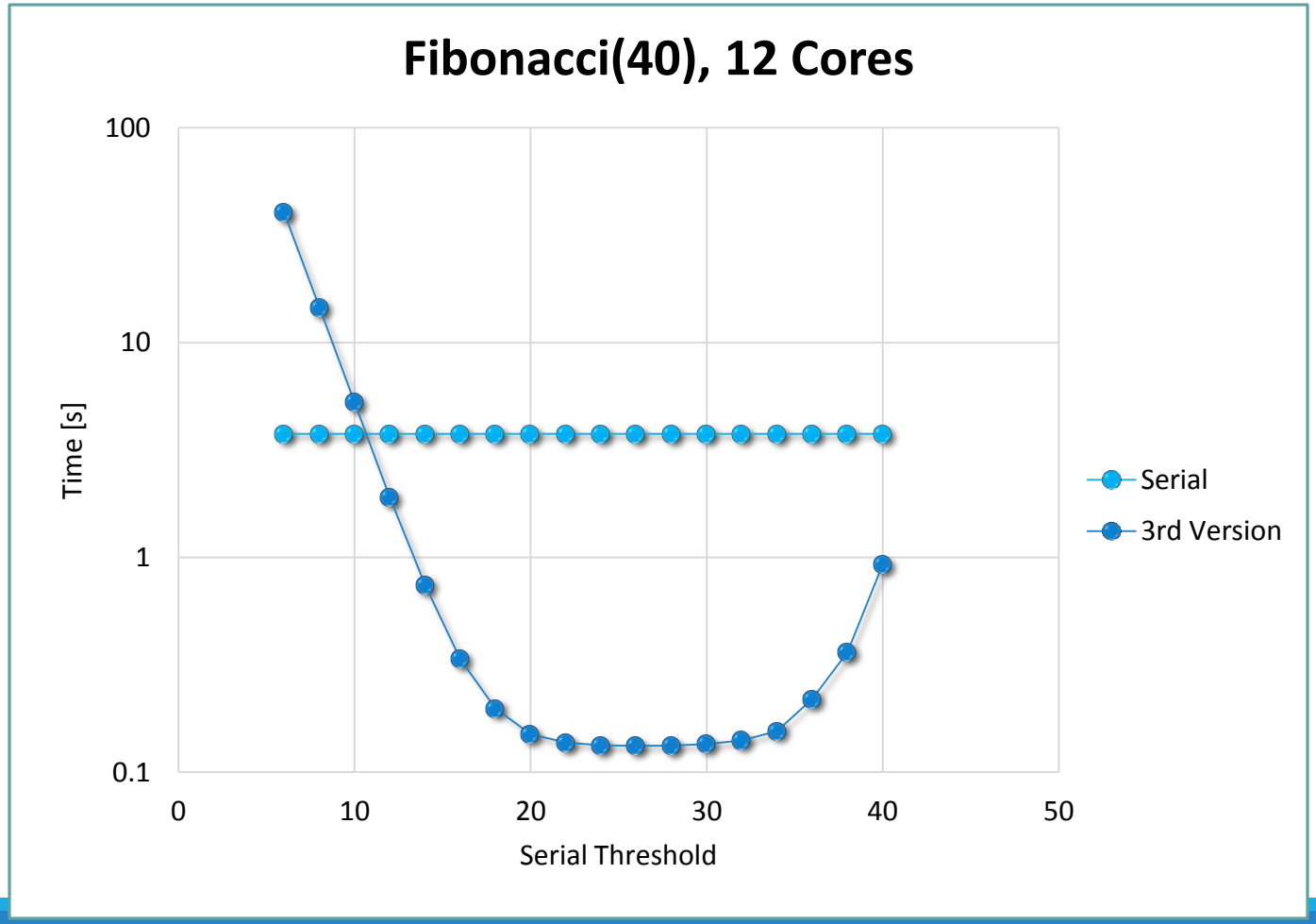
- The amount of work executed in one thread

Let's Parallelize It – 3rd Attempt What could be easier?

Parallel calculation (3rd attempt), switching to serial execution below given threshold

```
uint64_t fibonacci(uint64_t n)
{
    if (n < 2) return n;
    if (n < threshold) return fibonacci_serial(n);

    // asynchronously calculate one of the sub-terms
    future<uint64_t> f = async(launch::async, &fibonacci, n-1);

    // synchronously calculate the other sub-term
    uint64_t r = fibonacci(n-2);

    // wait for the future and calculate the result
    return f.get() + r;
}
```

Let's Parallelize It – 3rd Attempt What could be easier?



Granularity Control
The New Dimension

Parallelizing code introduces Overheads (SLOW)

Overheads are caused by code which

- Is executed in the parallel version only
- Is on the critical path (we can't 'hide' it behind useful work)
- Is required for managing the parallel execution
  - i.e. task queues, synchronization, data exchange
  - NUMA and core affinities

Controlling not only the amount of resources used but also the granularity of work is an important factor

Controlling the grain size of work allows finding the sweet-spot between too much overheads and too little parallelism

N3634: Improvements to std::future<T> and Related APIs

Combining futures

- when_all(), when_any()
  - Allows waiting for a combination of passed in future instances
  - Return a future representing the entire operation

Defining continuations

- future::then()
  - Attaches a function to be executed once the future gets ready
  - Returns a future representing the result of the continuation function

Unwrapping futures

- Asynchronous operations may return future<future<T> >
  - future::unwrap() returns inner future instance

Create futures which are ‘ready’

- future<decay<T>::type> make_ready_future(T);

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3634.pdf

Futurization

Special technique allowing to automatically transform code

- Delay direct execution in order to avoid synchronization
- Turns ‘straight’ code into ‘futurized’ code
- Code no longer calculates results, but generates an execution tree representing the original algorithm
- If the tree is executed it produces the same result as the original code
- The execution of the tree is performed with maximum speed, depending only on the data dependencies of the original code
- Simple transformation rules:

Straight Code	Futurized Code
`T func() {…}`	`future<T> func() {…}`
`rvalue: n`	`make_ready_future(n)`
`T n = func(…);`	`future<T> n = async(&func, …);`

Let's Parallelize It – 4th Attempt What could be easier?

Parallel way (4th attempt), futurize algorithm to remove suspension points

```
future<uint64_t> fibonacci(uint64_t n)
{
    if (n < 2) return make_ready_future(n);
    if (n < threshold) return make_ready_future(fibonacci_serial(n));

    future<future<uint64_t>>  f = async(launch::async, &fibonacci, n-1);
    future<uint64_t> r = fibonacci(n-2);

    return when_all(f.get() , r).then(
        [](future<std::vector<future<uint64_t>>> fv) -> uint64_t {
            std::vector<future<uint64_t>> v = fv.get();
            return v[0].get() + v[1].get();
        });
}
```

Let's Parallelize It – 4th Attempt What could be easier?

Parallel way (4th attempt), futurize algorithm to remove suspension points

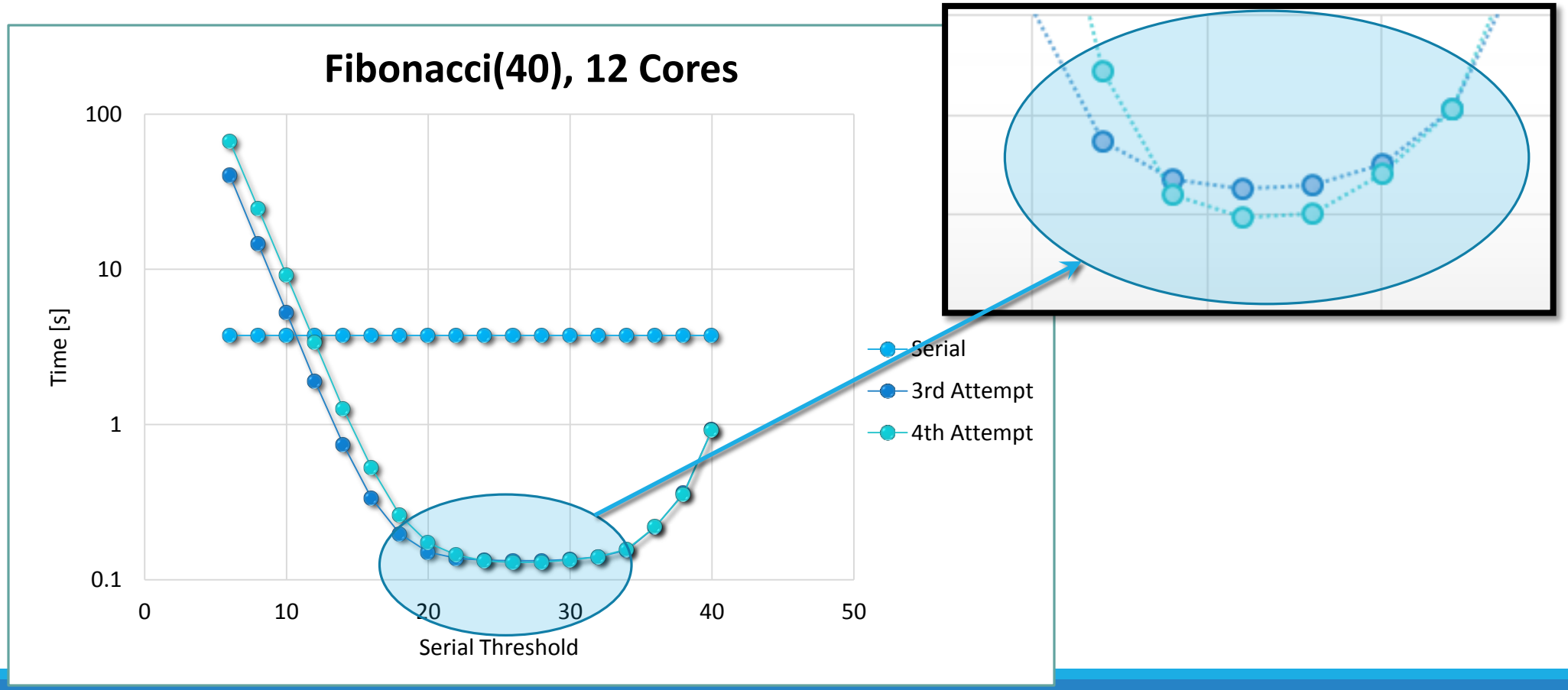
```
future<std::vector<future<uint64_t>>> f = when_all(f.get(), r);
future<uint64_t> result = f.then(
    [](future<std::vector<future<uint64_t>>> fv) -> uint64_t {
        std::vector<future<uint64_t>> v = fv.get();
        return v[0].get() + v[1].get();
    });
return result;
```

Let's Parallelize It – 4th Attempt What could be easier?

Parallel way (4th attempt), futurize algorithm to remove suspension points

```
future<uint64_t> fibonacci(uint64_t n)
{
    if (n < 2) return make_ready_future(n);
    if (n < threshold) return make_ready_future(fibonacci_serial(n));

    future<future<uint64_t>> f = async(launch::async, &fibonacci, n-1);
    future<uint64_t> r = fibonacci(n-2);

    return when_all(f.get(), r).then(
        [](future<std::vector<future<uint64_t>>> f) -> uint64_t {
            std::vector<future<uint64_t>> v = f.get();
            return v[0].get() + v[1].get();
        });
}
```

Let's Parallelize It – 4th Attempt
What could be easier?



Let's Parallelize It – 4th Try
What could be easier?

Parallel way (4th attempt), futurize algorithm to remove suspension points

```
future<uint64_t> fibonacci(uint64_t n)
{
    if (n < 2) return make_ready_future(n);
    if (n < threshold) return make_ready_future(fibonacci_serial(n));

    future<future<uint64_t>> f = async(launch::async, &fibonacci, n-1);
    future<uint64_t> r = fibonacci(n-2);

    return when_all(f.get(), r).then(
        [](future<std::vector<future<uint64_t>>> f) -> uint64_t {
            std::vector<future<uint64_t>> v = f.get();
            return v[0].get() + v[1].get();
        });
}
```

Let's Parallelize It – 5th Attempt What could be easier?

Parallel way (5th attempt), unwrapping inner future

```
future<uint64_t> fibonacci(uint64_t n)
{
    if (n < 2) return make_ready_future(n);
    if (n < threshold) return make_ready_future(fibonacci_serial(n));

    future<uint64_t> f = async(launch::async, &fibonacci, n-1).unwrap();
    future<uint64_t> r = fibonacci(n-2);

    return when_all(f, r).then(
        [](future<std::vector<future<uint64_t> > > f) -> uint64_t {
            std::vector<future<uint64_t> > v = f.get();
            return v[0].get() + v[1].get();
        });
}
```

Guess what? – This is the fastest implementation so far!

N3650: Resumable Functions

Large amount of overheads in HPX are caused by stacks

- We don't know which thread will suspend, thus every thread needs its own stack segment (~8kByte)
  - Virtual memory segmentation, TLB thrashing, physical memory exhaustion
- Not always possible to reuse stack segments as too many threads are being suspended
  - Fibonacci: many threads get created just to be suspended almost immediately
  - Figuring out what threads need to be executed first in order to make progress is a NP complete problem

Using when_all(), then(), etc. is an alternative, however

- Complex constructs
- Integrate badly with straight serial code and control structures

N3650: Resumable Functions

HPX can ‘simulate’ synchrony while performing asynchronous operations:

```
int f(stream str)
{
    std::vector<char> buf;
    future<int> count = str.read(512, buf);
    // …
    return count.get() + 11;    // get() will suspend
}
```

However, this leaves a half-filled stack frame behind, moreover caller can’t proceed until done

N3650: Resumable Functions

Introduces 2 new keywords: `async` and `await`

```
future<int> f(stream str) async
{
    shared_ptr<vector<char>> buf = ...;
    int count = await str.read(512, buf);     // returns from f() if not ready!
    return count + 11;
}
```

From N3650:

- A resumable function is a function that is capable of split-phase execution, meaning that the function may be observed to return from an invocation without producing its final logical result or all of its side-effects.

This allows

- Writing asynchronous code as if it was synchronous
- Avoids creating stack frames as the resumable function always leaves the stack in 'virgin' state

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3650.pdf

N3650: Resumable Functions

Disadvantage: requires compiler support

- Local variables and parameters have to be placed in heap allocated memory
- Certain transformations have to be applied to function bodies to create current continuations and to allow for re-entrance
- Heap based allocations necessary to store local variables and parameters
- Surprising semantics (return type, side effects of surrounding code, etc.)

Advantages

- Only one stack segment is required for each OS-thread
- Simplified code
- More asynchrony as functions can proceed whenever a called function awaits

Let's Parallelize It – 6th Attempt What could be easier?

Parallel way (6th attempt), using resumable functions

```
future<uint64_t> fibonacci(uint64_t n) async
{
    if (n < 2) return make_ready_future(n);
    if (n < threshold) return make_ready_future(fibonacci_serial(n));

    future<uint64_t> f = async(launch::async, &fibonacci, n-1); // .unwrap()
    future<uint64_t> r = fibonacci(n-2);

    return await f + await r;
}
```

Let's Parallelize It – 6th Attempt
What could be easier?

Fastest parallelization results, outperforms all others by a significant amount

- Scales better
- Runs faster
- Uses less memory (less stack segments)
- Creates less threads

Resumable functions are a valuable addition to the language

- Simplify code, makes asynchronous code look synchronous
- Required code transformation are almost trivial and well understood

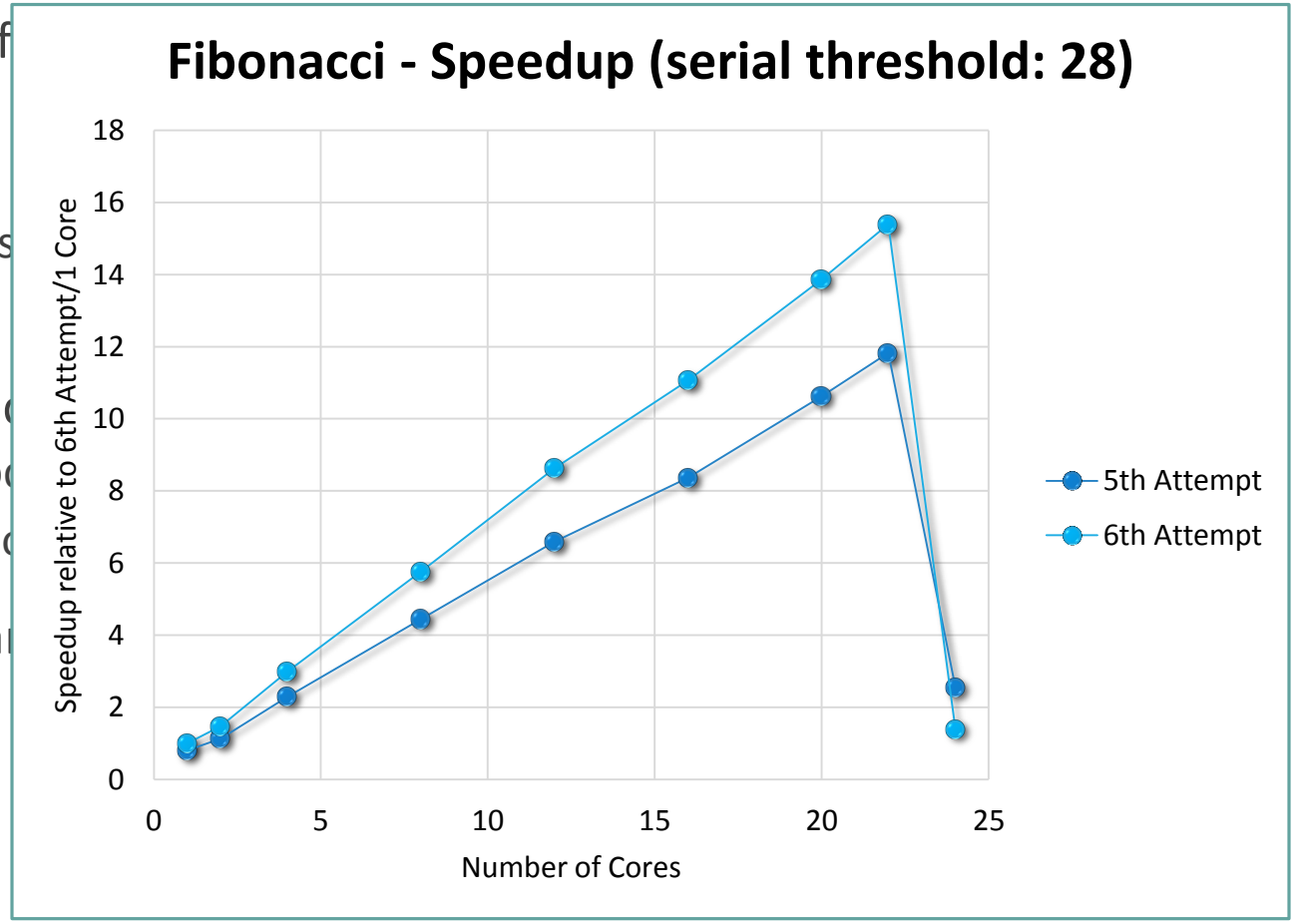
However, resumable function alone are not sufficient, they work best on code which has already been parallelized

Let's Parallelize It – 6th Attempt What could be easier?

Fastest parallelization results, outperf
- Scales better
- Runs faster
- Uses less memory (less stack segments
- Creates less threads

Resumable functions are a valuable a
- Simplify code, makes asynchronous co
- Required code transformation are almo

However, resumable function alone a
been parallelized



Let's Parallelize It – 7th Attempt What could be easier?

Parallel way (7th attempt), using resumable functions

```
future<uint64_t> fibonacci(uint64_t n)
{
    if (n < 2) return make_ready_future(n);
    if (n < threshold) return make_ready_future(fibonacci_serial(n));

    future<uint64_t> f = async(launch::async, &fibonacci, n-1);
    future<uint64_t> r = fibonacci(n-2);

    return dataflow(
        [](future<uint64_t> f1, future<uint64_t> f2) -> uint64_t {
            return f1.get() + f2.get();
        },
        f, r);
}
```

Thanks to Thomas Heller for coming up with this idea

So What's the Deal?

Too much parallelism is as bad as is too little

- Sweetspot is determined by the Four Horsemen, mainly by contention

Granularity control is crucial

- Optimal grain size depends very little on number of used resources
- Optimal grain size is determined by the Four Horsemen, mainly by overheads, starvation, and latencies

Even problems with (very) strong data dependencies can benefit from parallelization

Doing more is not always bad

- While we added more overheads by futurizing the code, we still gained performance
- This is a result of the complex interplay of starvation, contention and overheads in modern hardware

Avoid explicit suspension as much as possible, prefer continuation style execution flow

- Dataflow style programming is key to managing asynchrony

Predicting the Future

THE UNIVERSAL SCALABILITY LAW

Universal Scalability Law

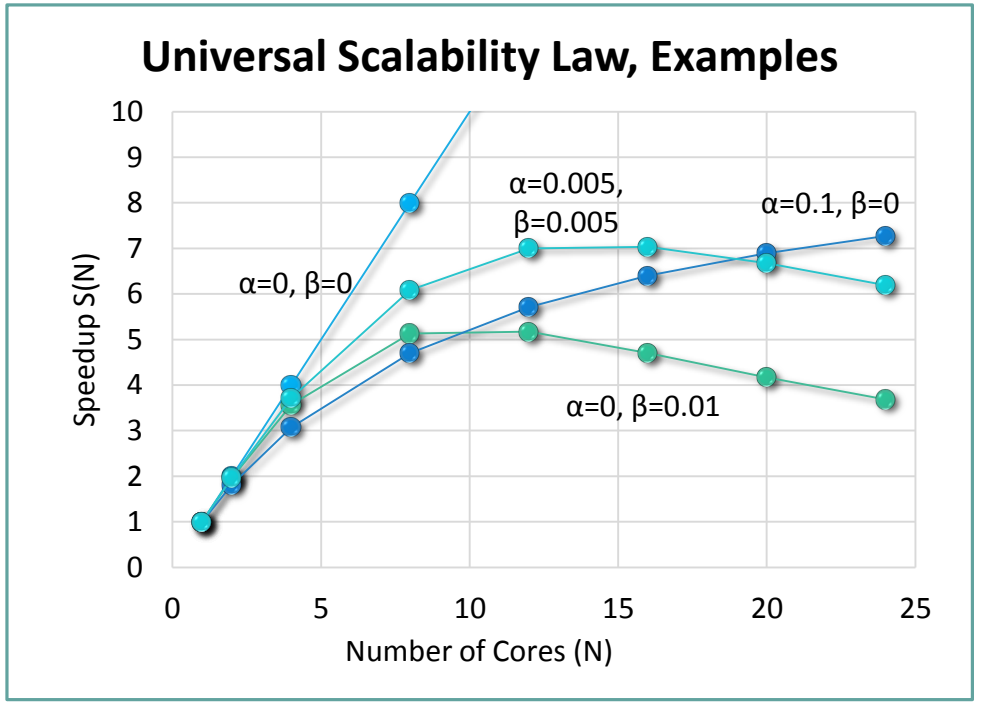
$$S(N) = \frac{N}{1+\alpha\,(N-1)+\beta\,N(N-1)}$$

S(N) – Normalized scaling

1 – Concurrency
- Without interaction the function would scale linearly

$\alpha\,(N-1)$ – Contention, (time spent in scalar mode, i.e. α is serial fraction), starvation
- Represents the degree of serialization on shared writable data and is parameterized by the constant σ.

$\beta\,N(N-1)$ – Latency (Point-to-point), overheads
- Represents the penalty incurred for maintaining consistency of shared writable data



http://www.perfdynamics.com/index.html

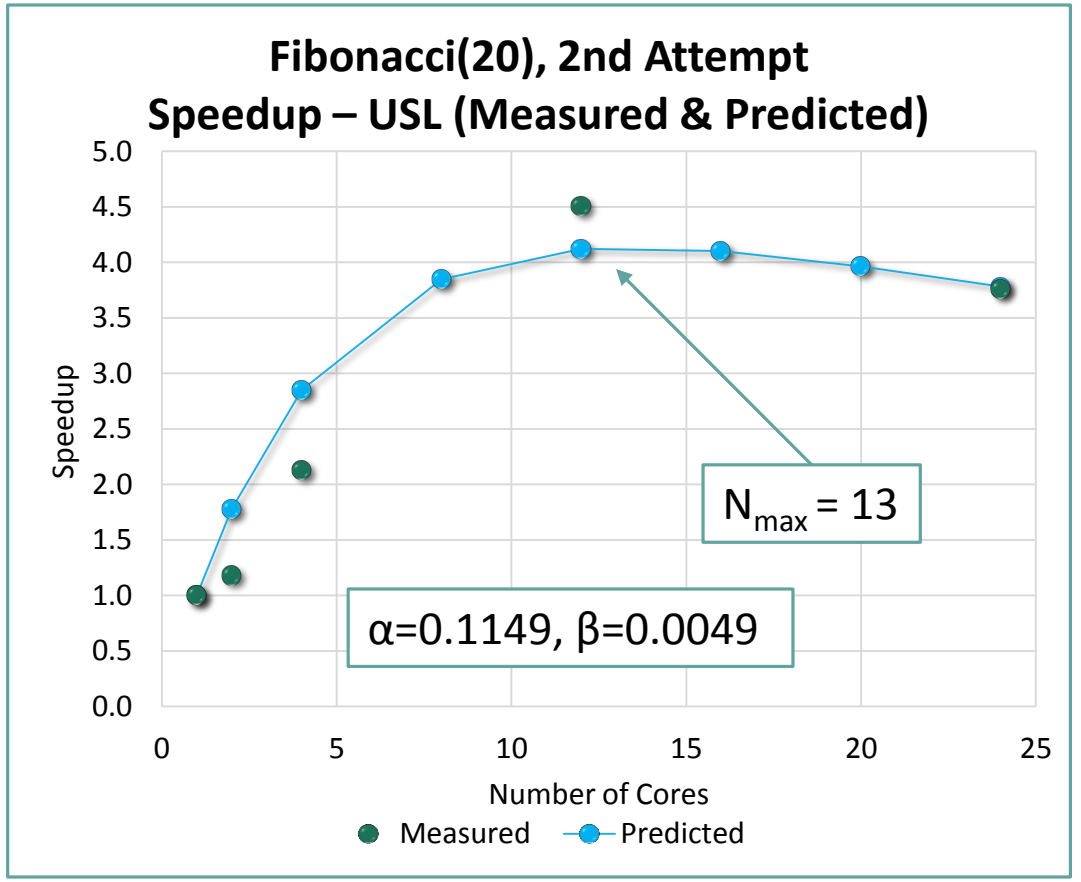
Universal Scalability Law

If we know the coefficients (α, β) we can estimate the N gaining the best speedup:

$$N_{max} = \sqrt{\frac{1-\alpha}{\beta}}$$

This does not tell us what causes the problems, but helps optimizing resource utilization

Can be done at runtime, 4-6 data points are sufficient



Using HPX

EXTENDING THE STANDARD BEYOND ONE NODE

HPX feature(s) for Asynchronous Computing:

Actions

- Actions are essentially functions in conventional sense
- But additionally can be invoked to remote locations

Global Actions

- Functions that could to be invoked remotely.

```
void some_global_function(double d)
{
     cout << d;
}

// This will define the action type 'some_global_action' which represents
// the function 'some_global_function'.
HPX_PLAIN_ACTION(some_global_function, some_global_action);
```

Example

```
// Evaluating Factorial
boost::uint64_t factorial(boost::uint64_t x);
HPX_PLAIN_ACTION(factorial, factorial_action);

boost::uint64_t factorial(boost::uint64_t n)
{
    if (n <= 0) return 1;

    factorial_action fact;
    hpx::future<boost::uint64_t> n1 =
        hpx::async(fact, hpx::find_here(), n - 1);
    return n * n1.get();
}
```

Invoking Actions in HPX

Asynchronous Actions without Synchronization

- Fully asynchronous, controlling thread does not wait for action to start or complete
- Return value ignored if the function had return value

```
some_global_action act;        // define an instance of some_global_action
hpx::apply(act, hpx::find_here(),  data);
```

```
some_component_action act;     // define an instance of some_component_action
hpx::apply(act, id, "42");     // id is component's global id
```

Invoking Actions in HPX

Asynchronous Actions with Synchronization

- Controlling thread does not wait for the function to start or complete
- Waits for the return value of the function

```
some_global_action act;     // define an instance of some_global_action
hpx::future<void> f = hpx::async(act, hpx::find_here(), 2.0);

//
// ... do other stuff here
//

f.get();    // this will possibly wait for the asyncronous operation to 'return'
```

Invoking Actions in HPX

Synchronous Actions

- The invoked function is scheduled immediately
- The calling thread waits for the function to complete

```
some_global_action act;        // define an instance of some_global_action
act(hpx::find_here(), 2.0);
```

```
some_component_action act;     // define an instance of some_component_action
int result = act(id, "42");
```

Invoking Actions in HPX

Asynchronous Actions With Continuation But Synchronization

- Similar to async action with synchronization; but takes additional function argument.
- Similar to future_function().then(….) on the same locality

```
// first action
boost::int32_t action1(boost::int32_t i)
{
    return i+1;
}
HPX_PLAIN_ACTION(action1);    // defines action1_type

// second action
boost::int32_t action2(boost::int32_t i)
{
    return i*2;
}
HPX_PLAIN_ACTION(action2);    // defines action2_type
```

Invoking Actions in HPX

```
action1_type act1;     // define an instance of 'action1_type'
action2_type act2;     // define an instance of 'action2_type‘
// action1_type and action2_type are global function or component function.
hpx::future<int> f =
    hpx::async_continue(act1, hpx::find_here(), 5,
        hpx::make_continuation(act2));
hpx::cout << f.get() << "\n";   // will print: 12 ((5+1) * 2)
```

Continuing to remote locality

The final return of chain of called function goes back to origin

```
hpx::future<int> f =
    hpx::async_continue(act1, hpx::find_here(), 5,
        hpx::make_continuation(act2, hpx::find_here()));
// put component_id in place of find_here() for remote function invocation
hpx::cout << f.get() << "\n";   // will print: 12 ((5 + 1) * 2)
```

Invoking Actions in HPX

Chaining more than 2 operations

```
hpx::future<int> f =
    hpx::async_continue(act1, hpx::find_here(), 5,
        hpx::make_continuation(act2,
            hpx::make_continuation(act1)));

hpx::cout << f.get() << "\n";   // will print: 13 (((5+1) * 2) + 1)
```

Invoking Actions in HPX

Asynchronous Actions With Continuation But Without Synchronization

- Similar to async actions with no synchronization,
- Similar structure as above, but after evaluation of the last function, the return value if any is discarded

```
string greetings(string name) { return "Hello," + name + "!"; }
HPX_PLAIN_ACTION(greetings);

void print(string greetings) { hpx::cout << greetings << '\n'; }
HPX_PLAIN_ACTION(print);

greetings_action_type act1;
print_action_type act2;
hpx::async_continue(act1, hpx::find_here(), "Manuel",
        hpx::make_continuation(act2));

// will print: Hello, Manuel!
```

Components and Actions (Distributed Computing – HPX)

Components

- “Remotable” C++ objects
- These are First Class objects in HPX, with a globally unique name, or GID
- Context of the instantiated object is preserved, for the duration of the client side that holds reference to its global name.

Component Actions

- Functions that need to be invoked remotely, but are part of a class object/ component object.

Components in HPX

```
class accumulator
  : public hpx::components::simple_component_base<accumulator>
{
public:
    accumulator() : acculumlate_(0) {}

    void add(double to_add) { accumulate_ += to_add; }
    double query() const { return accumulator_; }

    HPX_DEFINE_COMPONENT_ACTION(accumulator, add, add_action);
    HPX_DEFINE_COMPONENT_CONST_ACTION(accumulator, query, query_action);

private:
    double accumulate_;
};
```

Components in HPX

```
// create new instance of an accumulator
hpx::id_type id = hpx::new_<accumulator>();

accumulator_add_action add_action;

// Explicitly asynchronous invocation
hpx::future<void> f = hpx::async(add_action, id, 42.0);
// ...
f.get();

// Explicitly ‘synchronous’ invocation
accumulator_query_action query_action;
hpx::cout << query_action(id) << "\n";    // prints: 42

// fire and forget version
hpx::apply(add_action, id, 43);
```

The Life of Pi

Embarrassingly Parallel Applications

Easily broken into approximately equal amounts of work per processors

Each Individual Task (per processor) is independent, i.e. minimal communication among processors

We get near-perfect parallel speedup with modest programming efforts

M work load can be divided into N chunk, with each work load equaling M/N

Parallel Overhead due to communication (over work coordination) and/or reduce operation at the end

Examples:

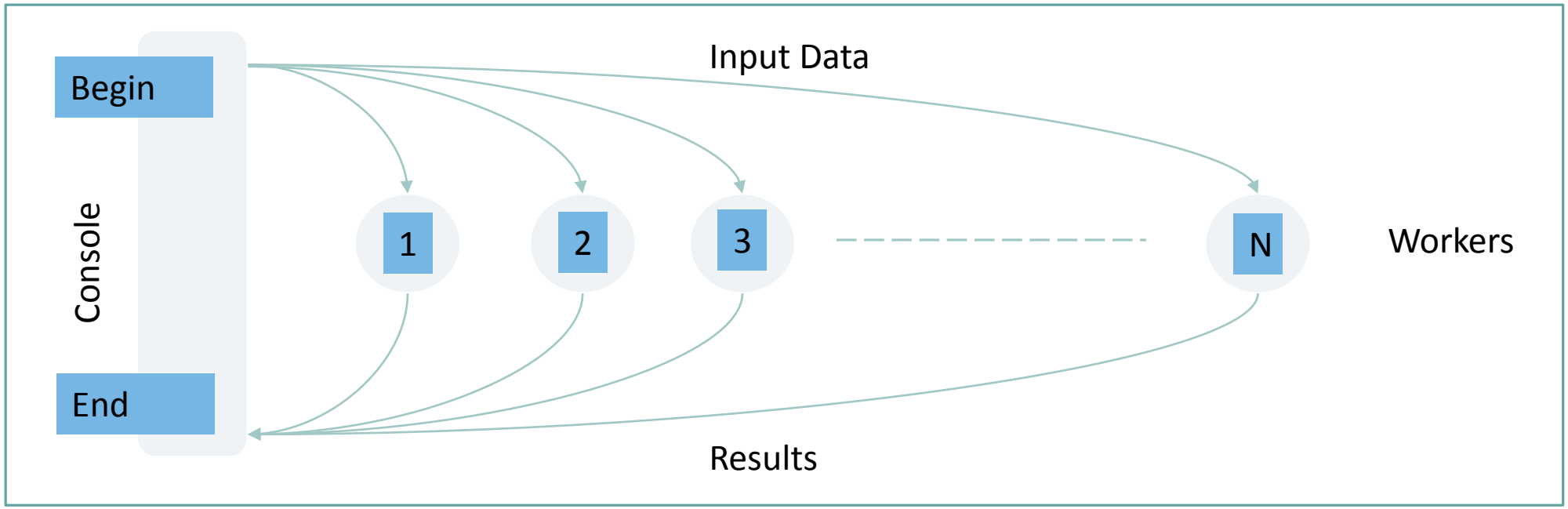
- Image Processing,
- Monte Carlo Simulation
- Random Number Generation
- Encryption, Compression

Embarrassingly Parallel Application

$Y_i = F_i(X_i)$

, where i = 0, 1, 2, … , N.

, x_i -> inputs, y_i -> outputs and, F_i -> pure function



Monte Carlo

Monte Carlo Methods : simulating physical phenomena based on randomness

- Randomly generate a large number of example cases (input) of a phenomenon,
- Perform some computation on these inputs
- Take average of the observed results

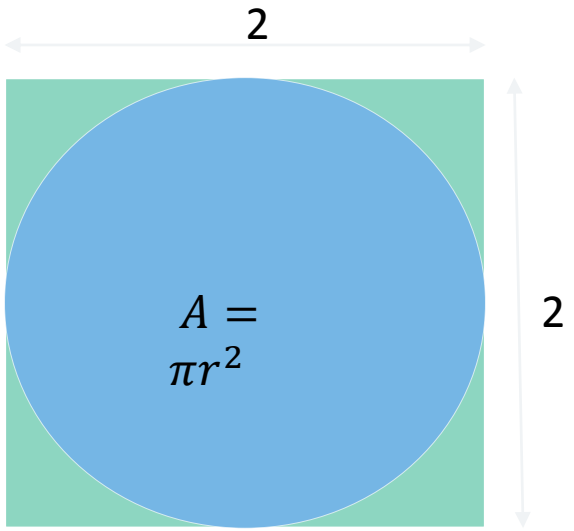
The result is an approximation to some true but unknown quantity

Monte Carlo Simulations are typically embarrassingly parallel

- Each unit simulation is completely independent from all such unit simulations.

Evaluation of Pi (Monte Carlo)

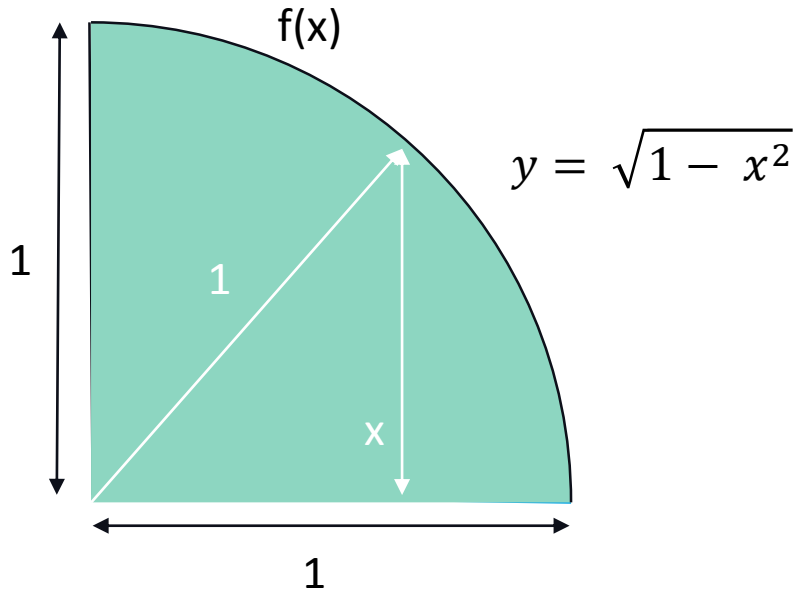
$$\frac{\text{Area of Circle}}{\text{Area of Square}} = \frac{\pi r^2}{2 \times 2} = \frac{\pi}{4}$$



Random Pairs of Numbers, (x_r, y_r)

$$x_r^2 + y_r^2 \leq 1$$

$$\int_0^1 \sqrt{1 - x^2}\, dx = \frac{\pi}{4}$$



$$Area = \int_{x_1}^{x_2} f(x)dx = \lim_{N \to \propto} \frac{1}{N-1} \sum_{i=0}^{N-1} f(x_r)\,(x_2 - x_1)$$

Evaluation of Pi using HPX

```
// define action: can be called remotely
double pi_montecarlo_evaluate(std::size_t num_iteration)
{
    // ...
}
// defines pi_montecarlo_evaluate_action
HPX_PLAIN_ACTION(pi_montecarlo_evaluate);


// retrieve all participating localities (nodes)
std::vector<hpx::id_type> localities =
     hpx::find_all_localities();
```

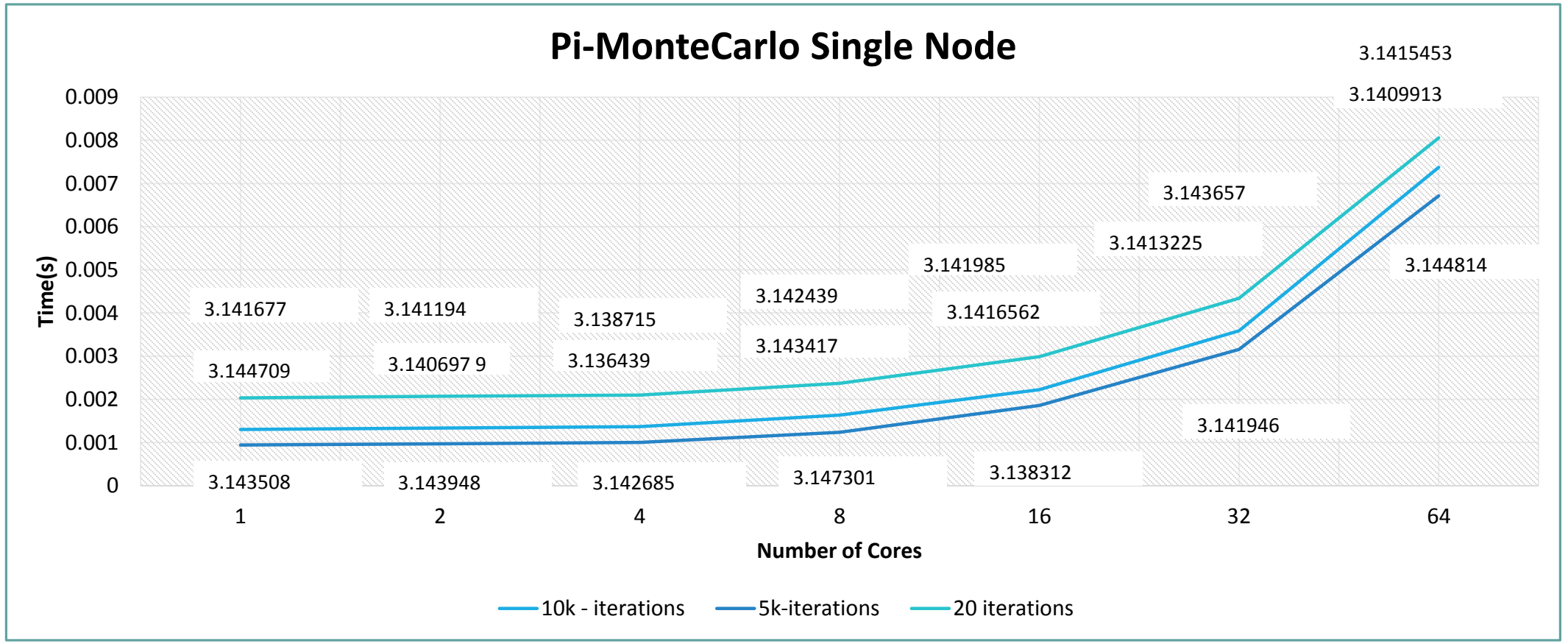
Evaluation of Pi using HPX

```
double pi_montecarlo_evaluate(std::size_t num_iteration)
{
     base_generator_type generator(0.1);
     generator.seed(std::time(0));
     gen_type monte_carlo(generator, distribution_type(0, 1));
     boost::generator_iterator<gen_type> gen_value(&monte_carlo);

     double x, area = 0;

     // dx = 1 / num_iteration;
     for(std::size_t i = 0; i != num_iteration; ++i)
     {
         x = *gen_value++;
         area += std::sqrt(1 - x*x); // f(x) * dx
     }

     // return area of quadrant
     return area / num_iteration;
 }
```

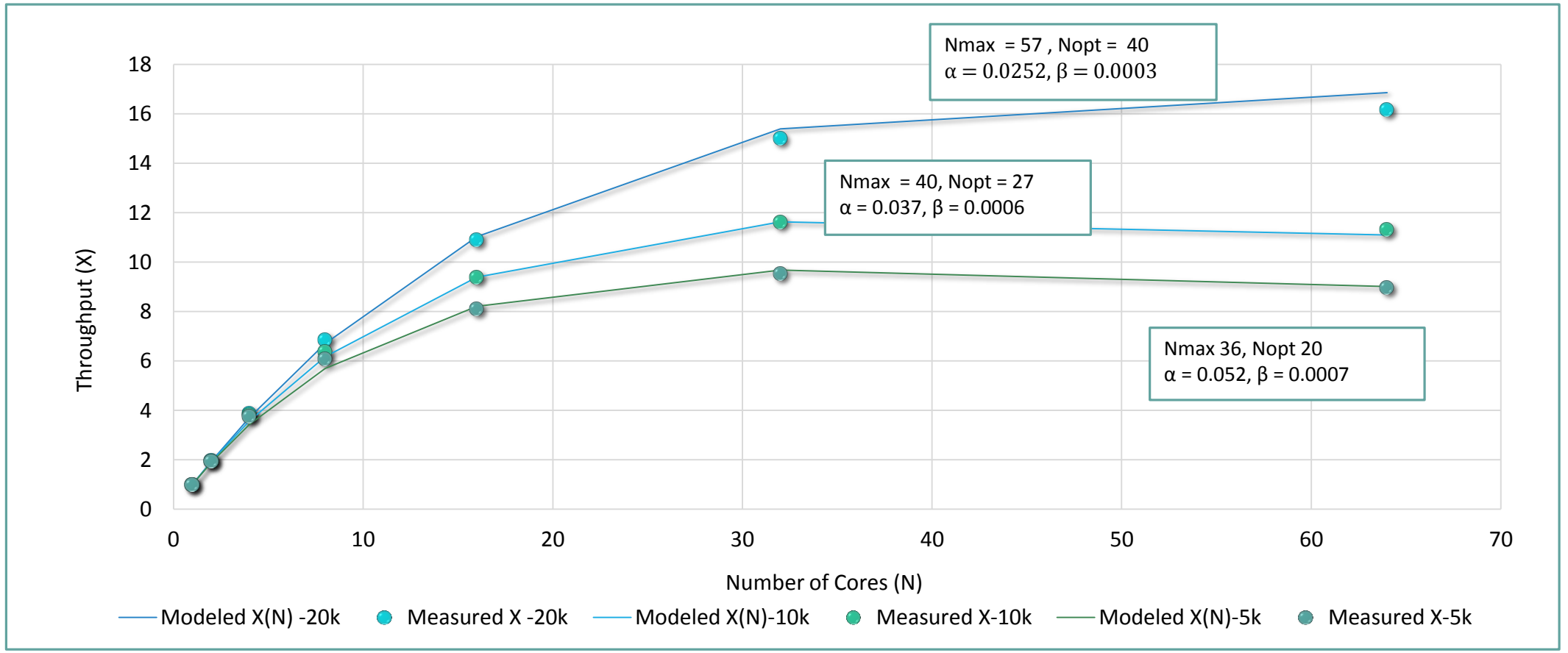
Evaluation of Pi using HPX

```
std::vector<future<double>> futures;

pi_montecarlo_evaluate_action act;
BOOST_FOREACH(id_type const& node, localities)
{
    futures.push_back(async(act, node, num_iter));
}

wait_all(futures);

double result = 0, pi = 0;
BOOST_FOREACH(future<double>& fut, futures)
{
    result += fut.get();
}

pi = 4 * result;
```

Results: Pi-Monte Carlo



Results: Pi – Monte Carlo USL projected Scalability (SMP)



Fast Fourier Transform (FFT)

Well Known and Understood problem

- Data dependency between elements is well defined

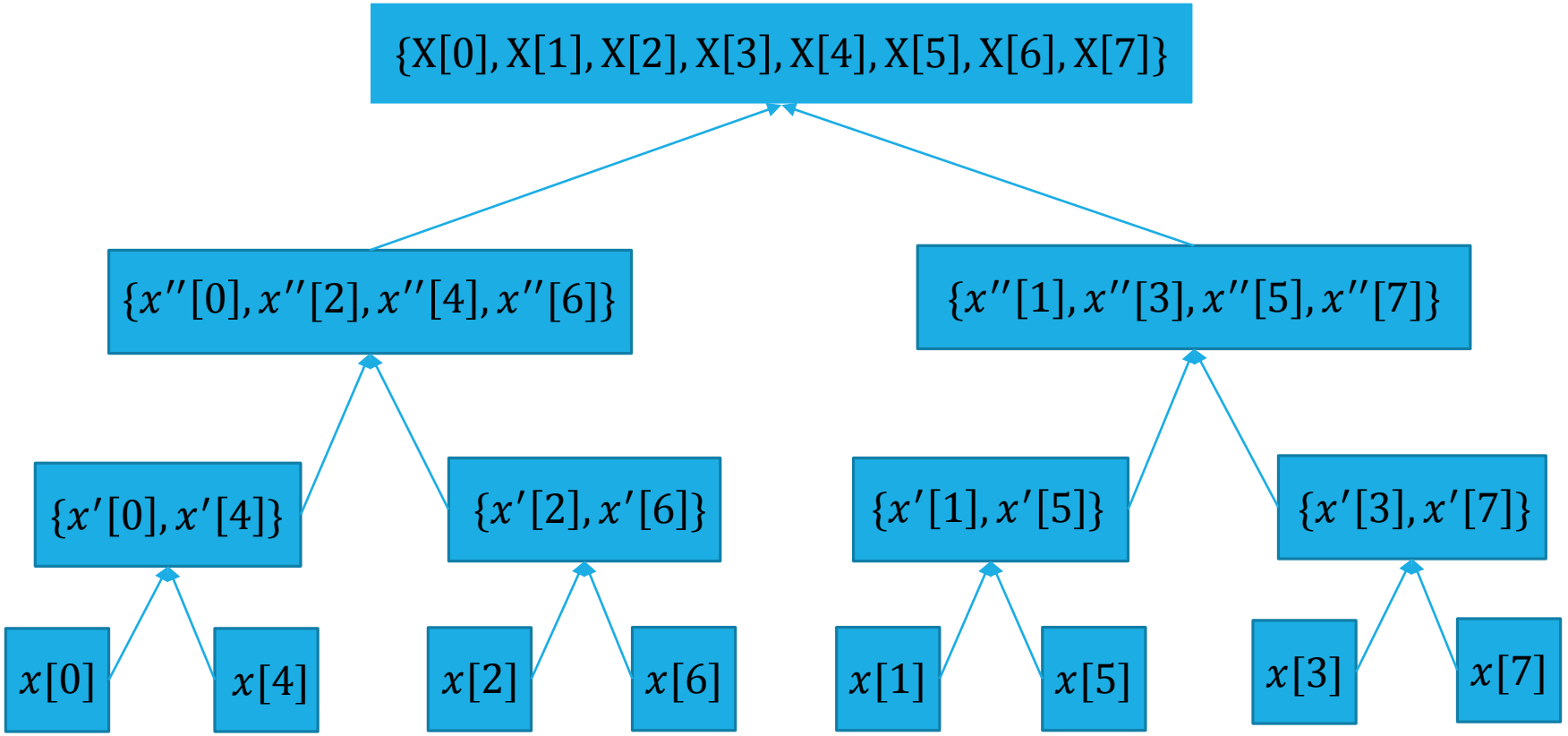
Parallel Implementation of FFT Emerging recently with several parallelization techniques available

- In Distributed FFT, data explicitly need to be moved
  - So a parallel asynchronous computation mechanism would be nice.
  - HPX async

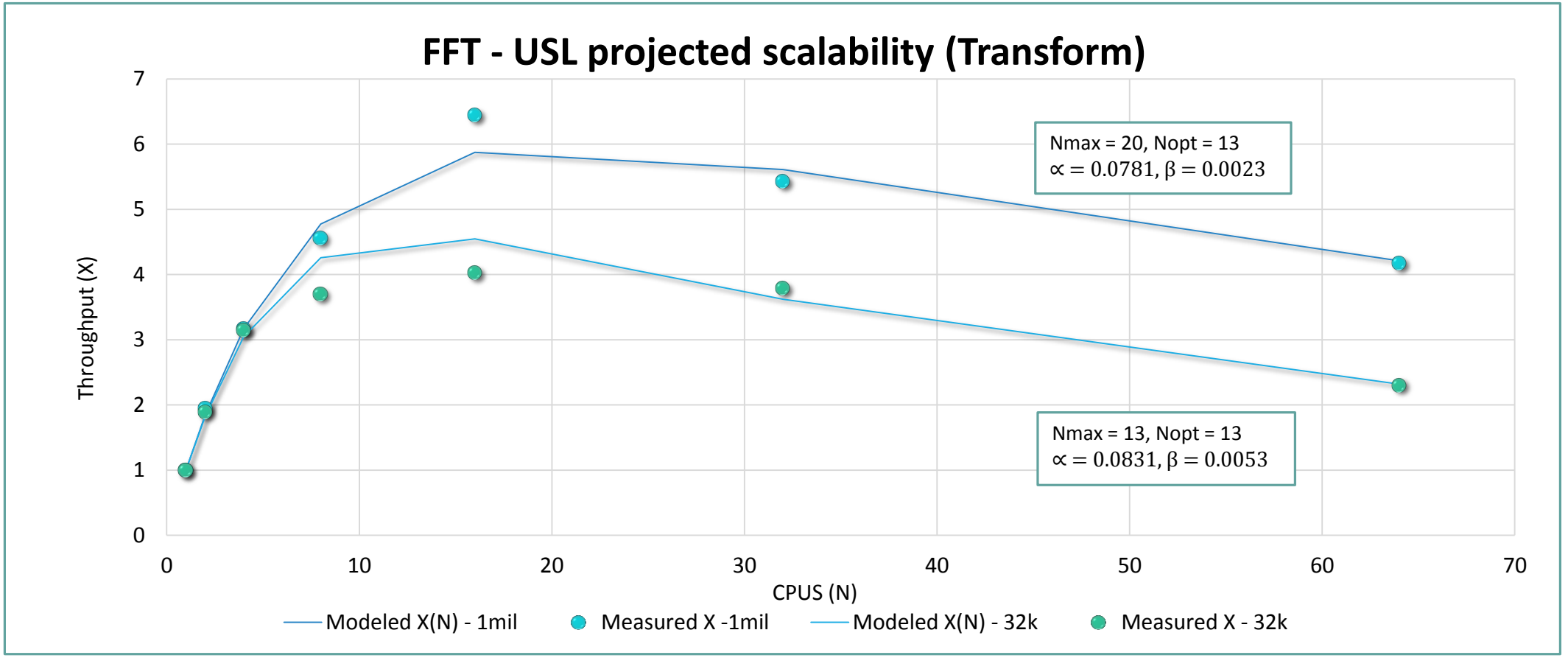
Implementation of Parallel FFT in HPX

- Based on Coley-Tuckey algorithm with Radix – 2DIT

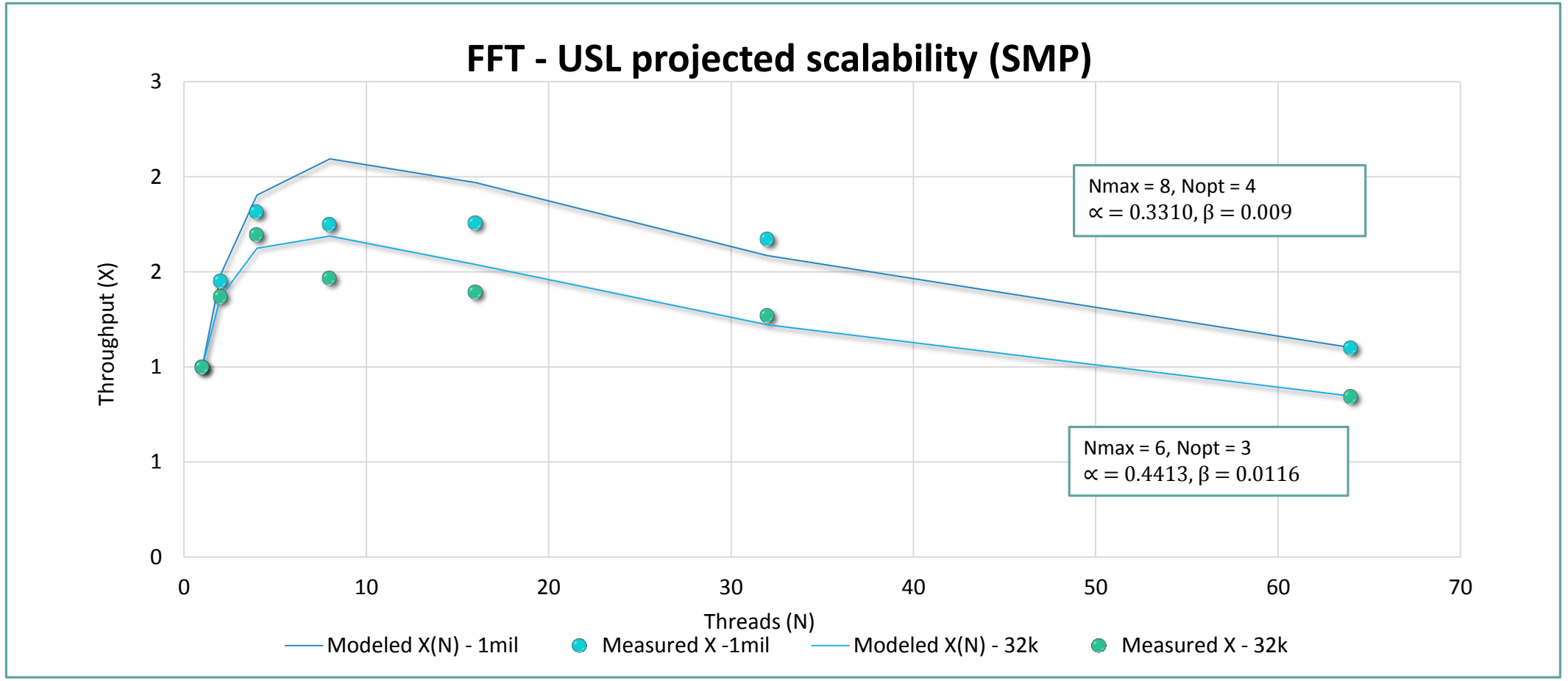
FFT (Decimation in Time)



Results: FFT - USL Projected Scalability (Transform)



Results: FFT - USL Projected Scalability (SMP)



Conclusions

Conclusions

Be aware of the Four Horsemen

Embrace parallelism, it’s here to stay, avoid concurrency

Asynchrony is your friend if used correctly

Think in terms of data dependencies, make them explicit

Avoid thinking in terms of threads

Continuation style, dataflow based programming is key for successful parallelization

Performance modelling can help adjusting parameters

Where to get HPX

Main repository: https://github.com/STEllAR-GROUP/hpx/ (Boost licensed)

Main website: http://stellar.cct.lsu.edu/

Mailing lists: hpx-users@stellar.cct.lsu.edu, hpx-devel@stellar.cct.lsu.edu

IRC channel: #ste||ar on freenode