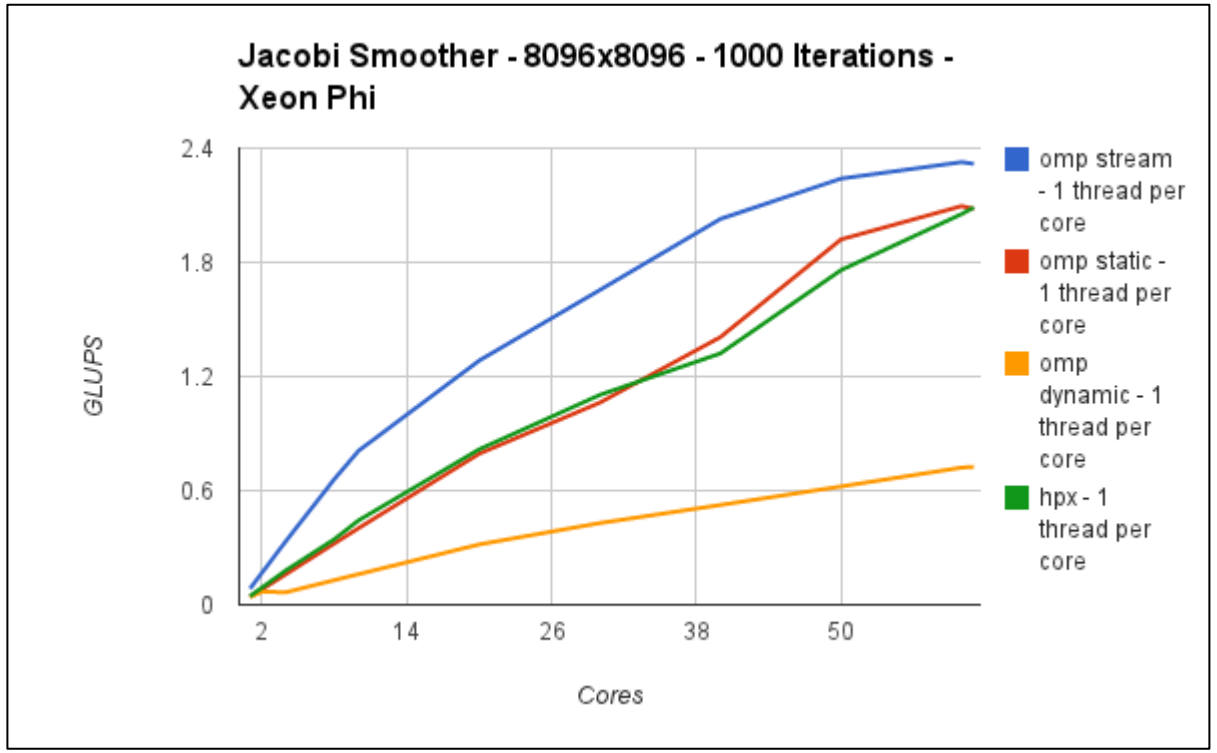
HPX

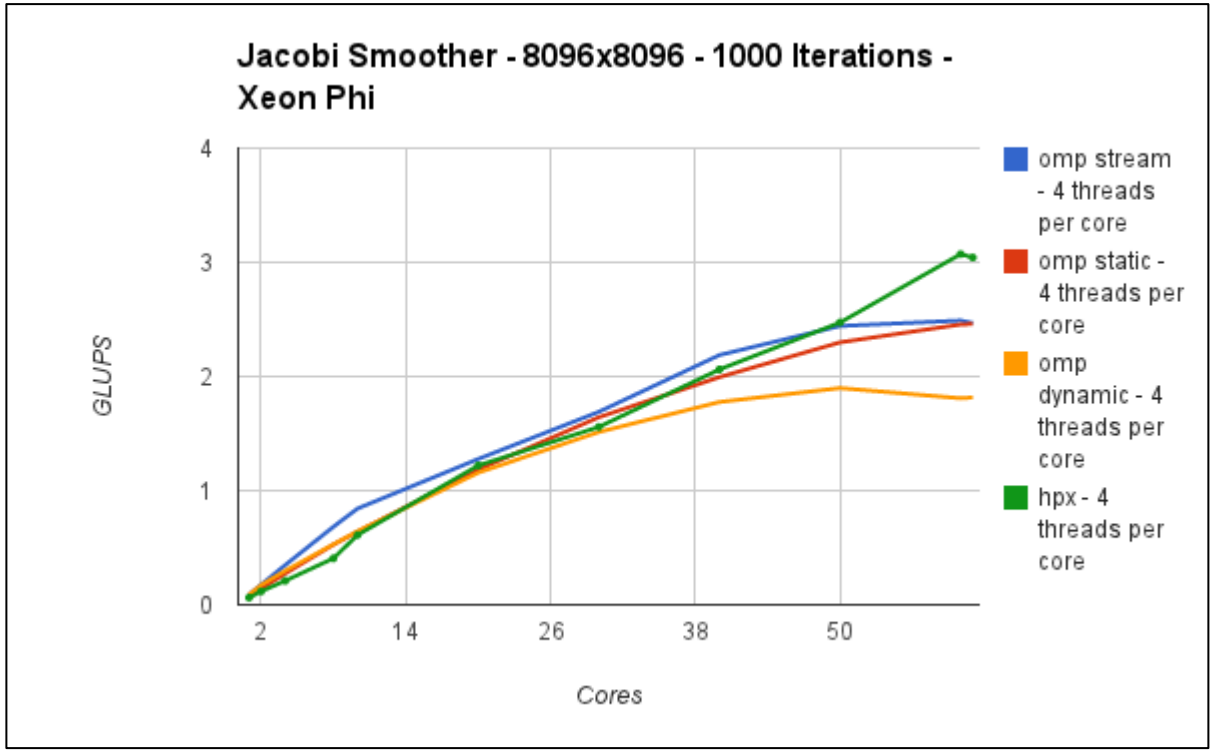
A GENERAL PURPOSE C++ RUNTIME SYSTEM FOR PARALLEL AND DISTRIBUTED APPLICATIONS OF ANY SCALE

HARTMUT KAISER (HKAISER@CCT.LSU.EDU)

HPX on the Xeon Phi



HPX on the Xeon Phi



Some Simple Examples

HELLO WORLD ANYONE?

Hello HPX World

```
#include <hpx/hpx.hpp>
#include <hpx/iostream.hpp>

int hpx_main(int argc, char* argv[])
{
    hpx::cout << "Hello HPX World!\n";
    return hpx::finalize();
}

int main(int argc, char* argv[])
{
    return hpx::init(argc, argv);
}
```

Hello HPX World

```
void say_hello()
{
    hpx::cout << "Hello HPX World from locality: " <<
              << hpx::get_locality_id() << "!\n";
}
HPX_PLAIN_ACTION(say_hello);   // defines say_hello_action

int hpx_main()
{
    say_hello_action sayit;
    for (auto loc: hpx::find_all_localities())
        hpx::apply(sayit, loc);
    return hpx::finalize();
}
```

Hello HPX World

```
int hpx_main()
{
   std::vector<hpx::future<void> > ops;
   say_hello_action sayit;

   for (auto loc: hpx::find_all_localities())
       ops.push_back(hpx::async(sayit, loc));

   hpx::wait_all(ops);
   return hpx::finalize();
}
```

Calculating Fibonacci

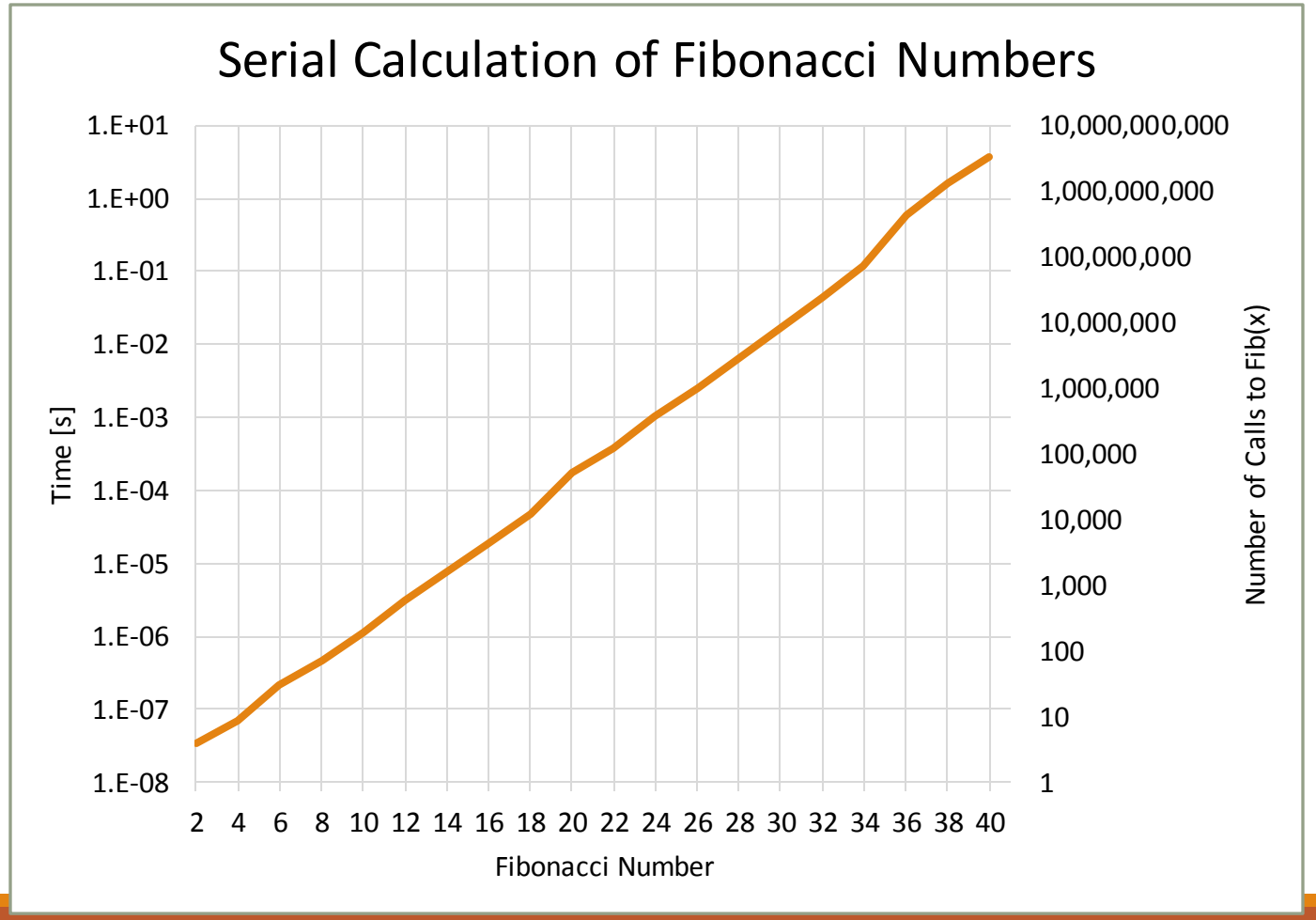
FUTURIZATION OF RECURSIVE ALGORITHMS

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



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

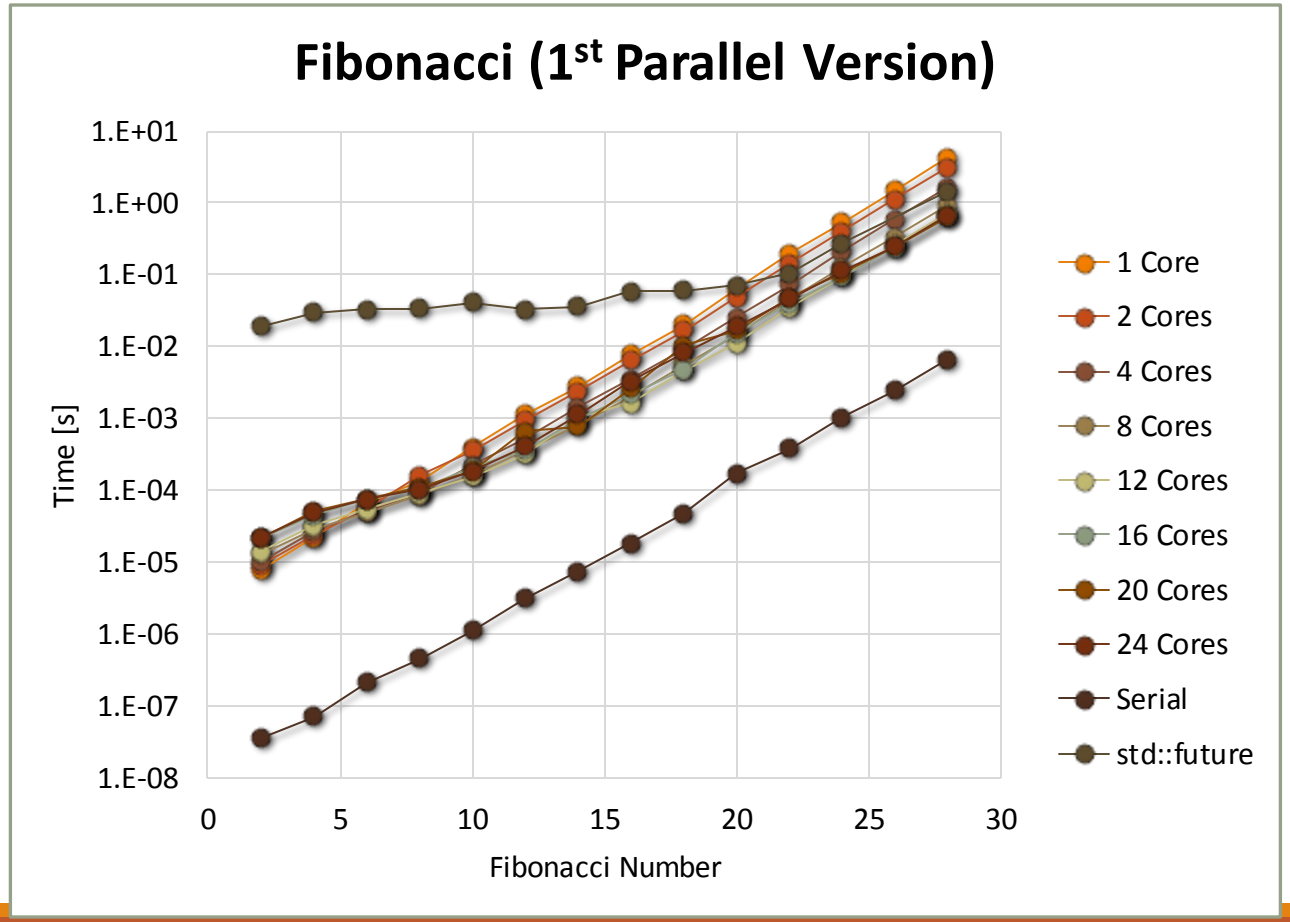
Let's spawn a new thread for every other sub tree on each recursion level

Let's Parallelize It – Adding Real Asynchrony

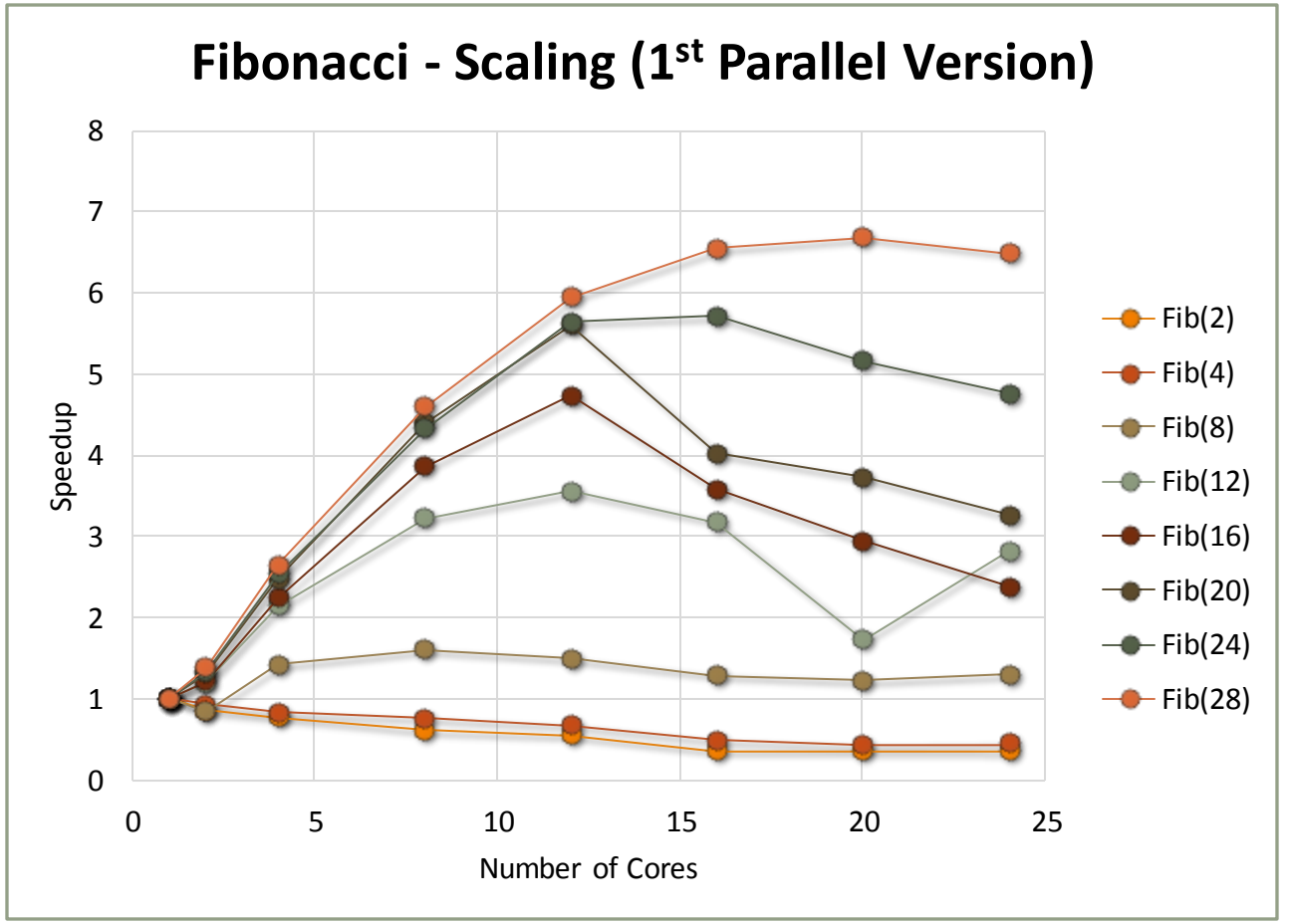
Calculate Fibonacci numbers in parallel

```
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-2);

    // synchronously calculate the other sub-term
    uint64_t r = fibonacci(n-1);

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

Let's Parallelize It – Adding Real Asynchrony



Let's Parallelize It – Adding Real Asynchrony



Let's Parallelize It –Adding Real Asynchrony

What's wrong? While it does scale, 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)

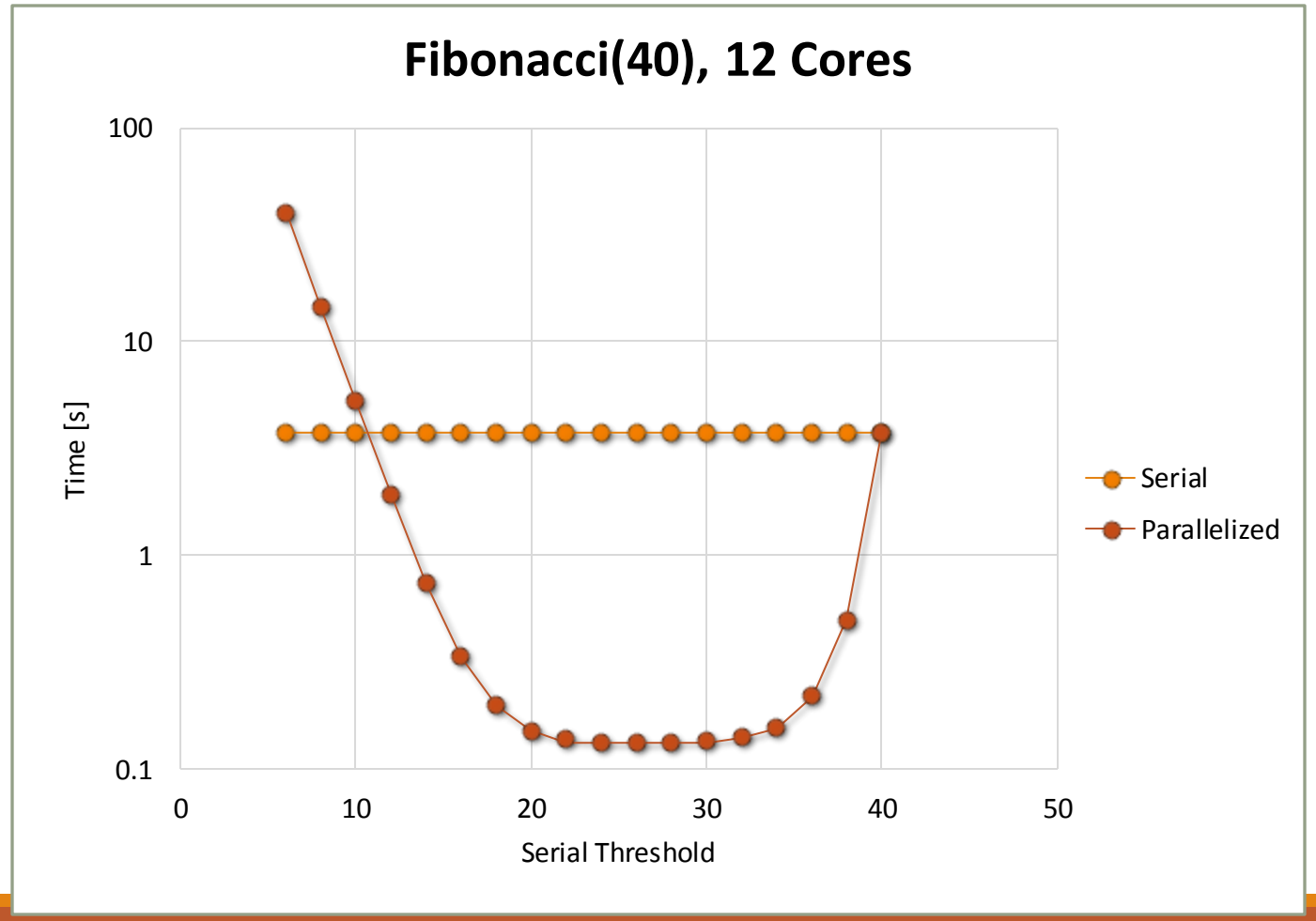
- The amount of work executed in one thread

Let's Parallelize It – Introducing Control of Grain Size

Parallel calculation, 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-2);

    // synchronously calculate the other sub-term
    uint64_t r = fibonacci(n-1);

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

Let's Parallelize It – Introducing Control of Grain Size



Grain Size 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

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 = func();`
`future<T> n = async(&func, …);`	`future<future<T> > n = async(&func, …);`

Let's Parallelize It – Apply Futurization

Parallel way, 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-2);
    future<uint64_t> r = fibonacci(n-1);

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

Let's Parallelize It – Unwrap Inner Futures

```
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-2).unwrap();
    future<uint64_t> r = fibonacci(n-1);

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

Let's Parallelize It – Unwrap Argument Futures

```
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-2);
    future<uint64_t> r = fibonacci(n-1);

    return dataflow(
        unwrapped([](uint64_t r1, uint64_t r2) {
            return r1 + r2;
        }),
        f, r);
}
```

Guess what? – This is 10% faster than straight version!

So What's the Deal?

Too much parallelism is as bad as is too little

- Sweet spot 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