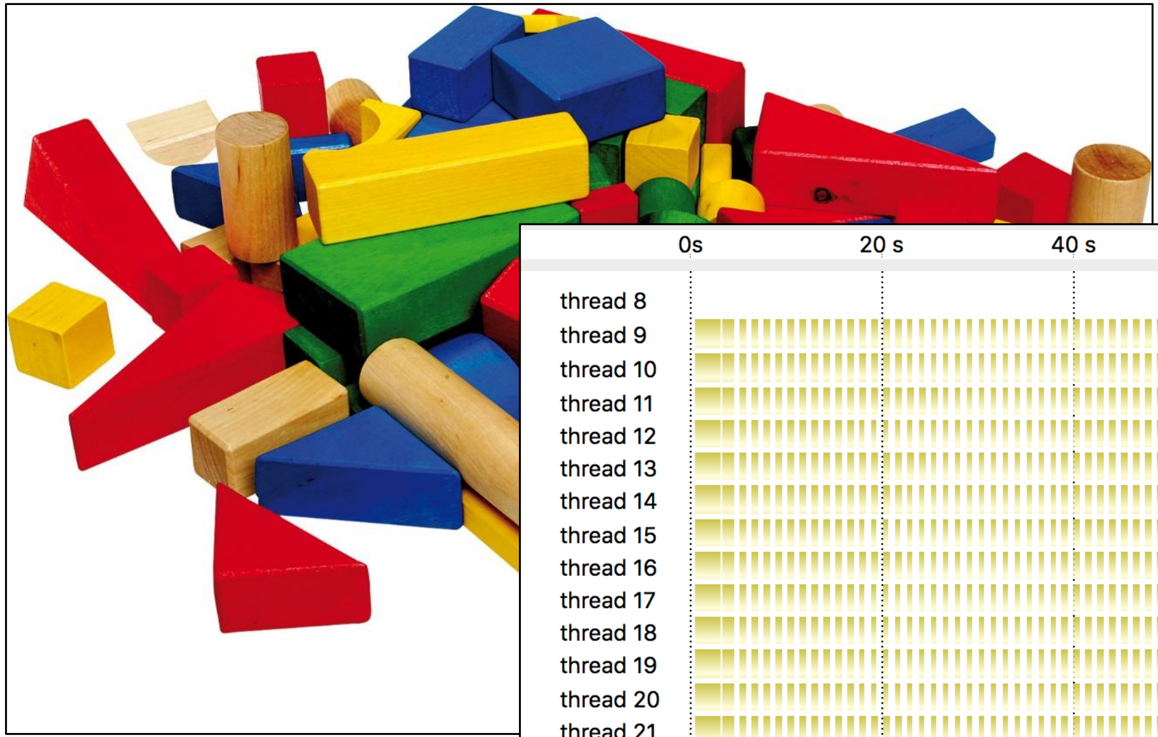
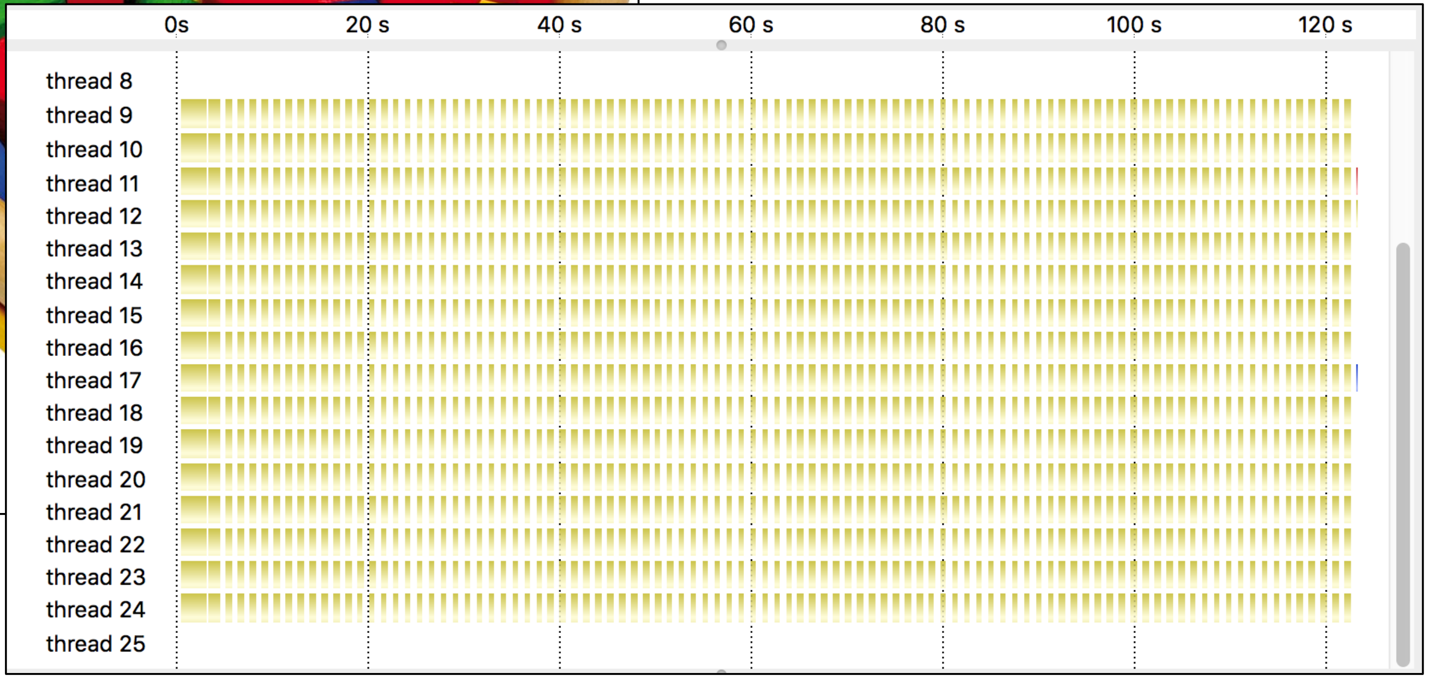
The Asynchronous C++ Parallel Programming Model

Hartmut Kaiser (Hartmut.Kaiser@gmail.com)

The Application Problems





Amdahl’s Law (Strong Scaling)

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

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

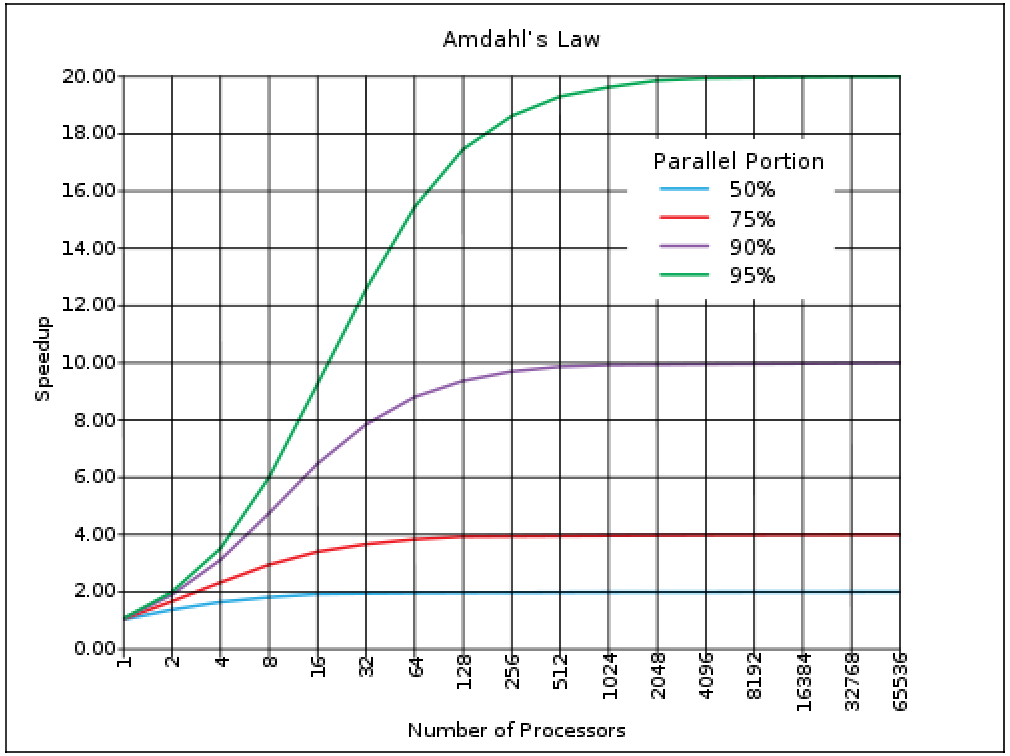


Figure courtesy of Wikipedia (http://en.wikipedia.org/wiki/Amdahl's_law)

STE||AR GROUP

Rule 1

Parallelize Applications as Much as Humanly Possible

STE||AR GROUP

The 4 Horsemen of the Apocalypse

courtesy of www.albrecht-durer.org

STE||AR GROUP

The 4 Horsemen of the Apocalypse

- **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

- **S**tarvation
  - Insufficient concurrent work to maintain high utilization of resources
- **L**atencies
  - Time-distance delay of re
- **O**verh
  - ns and resources on
    ssary in sequential variant
- **W**a g for Contention resolution
  - Delays due to lack of availability of oversubscribed shared resources

Impose upper bound on both, weak and strong scaling


courtesy of www.albrecht-durer.org

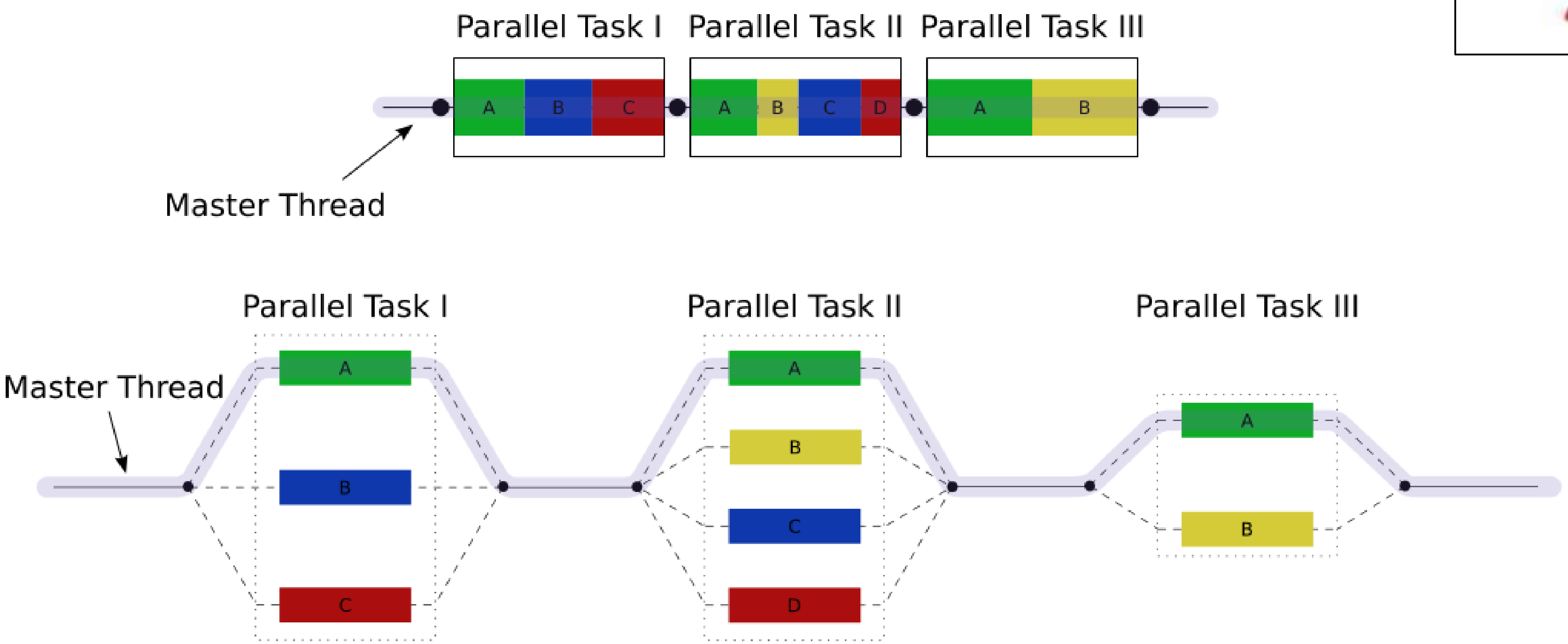
STE||AR GROUP

Real-world Problems

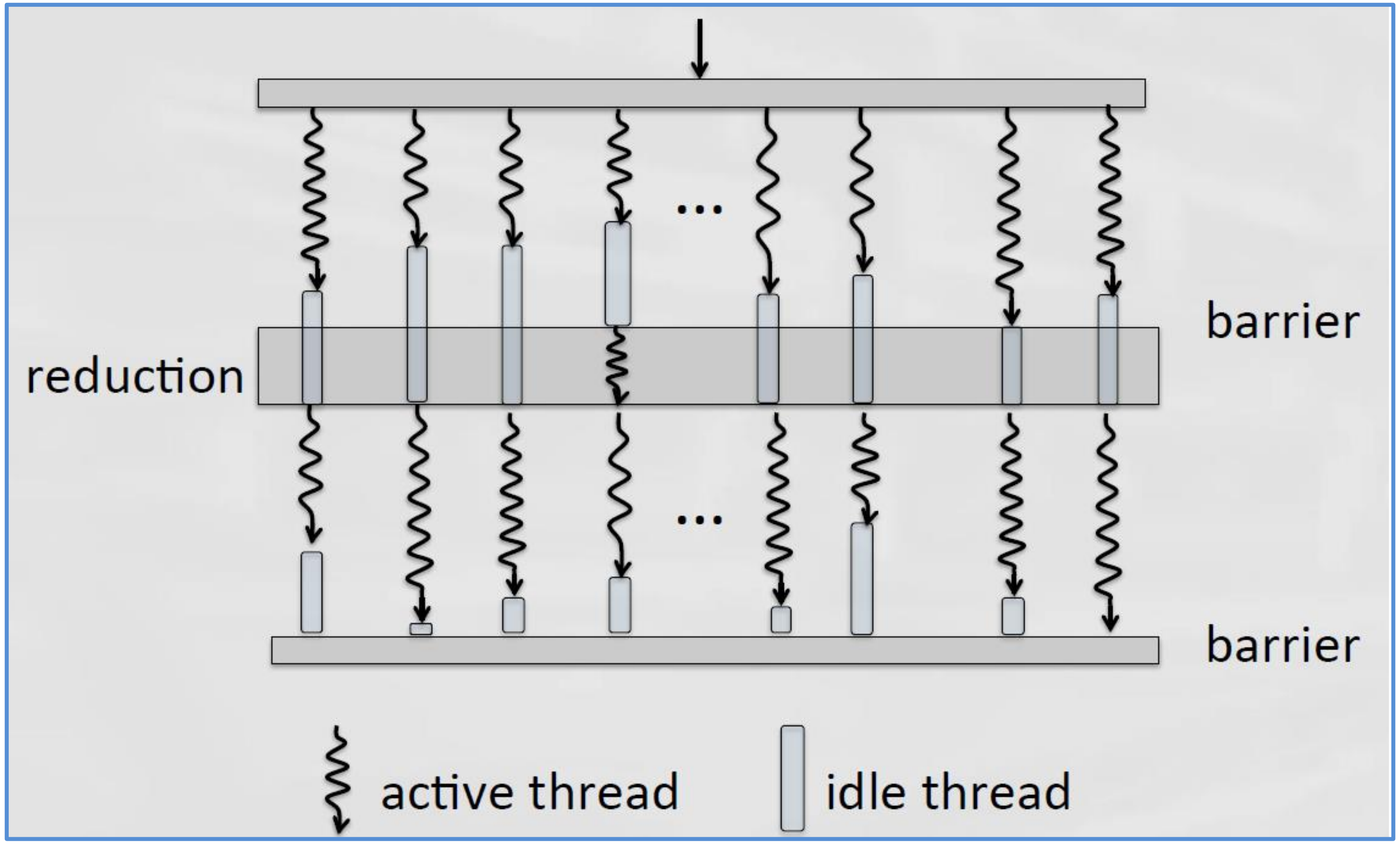
- Insufficient parallelism imposed by the programming model
  - OpenMP: enforced barrier at end of parallel loop
  - MPI: global (communication) barrier after each time step
- Over-synchronization of more things than required by algorithm
  - MPI: Lock-step between nodes (ranks)
- Insufficient coordination between on-node and off-node parallelism
  - MPI+X: insufficient co-design of tools for off-node, on-node, and accelerators
- Distinct programming models for different types of parallelism
  - Off-node: MPI, On-node: OpenMP, Accelerators: CUDA, etc.

Real-world Problems

- Even standard algorithms added to C++17 enforce fork-join semantics



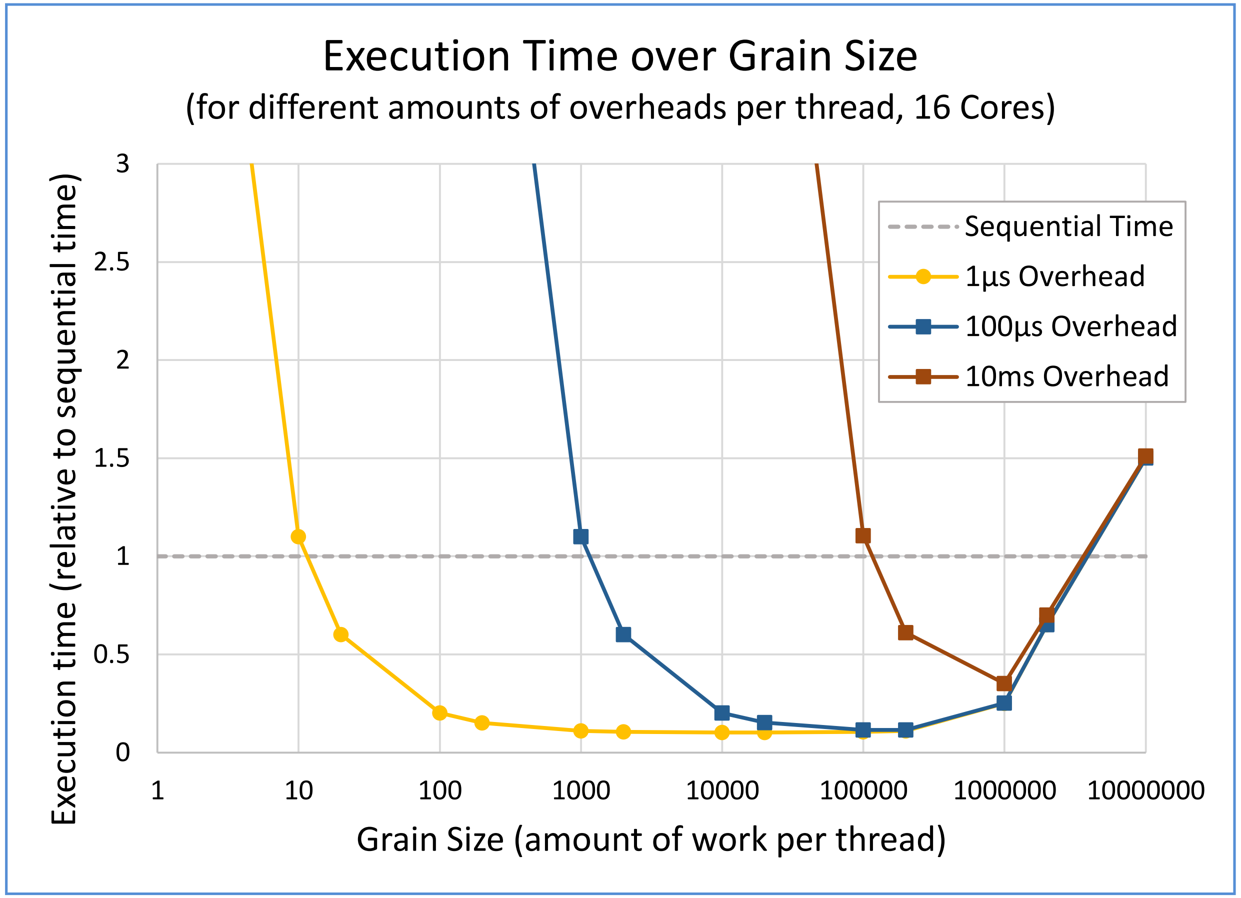
Fork/Join Parallelism



Rule 2

Use a Programming Environment that Embraces SLOW

Overheads: Thought-Experiment



STE||AR GROUP

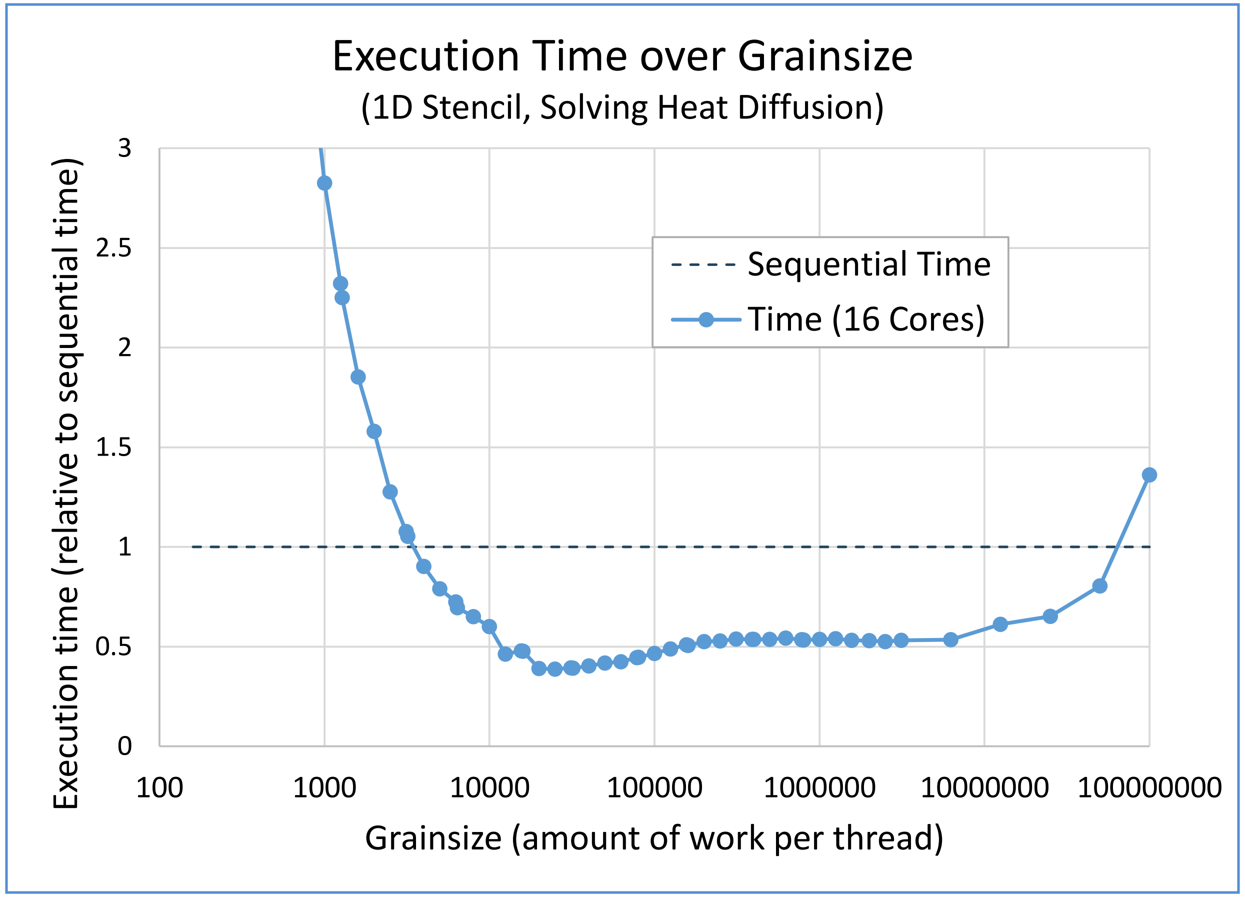
Overheads: The Worst of All?

- Even relatively small amounts of work can benefit from being split into smaller tasks
  - Possibly huge amount of 'threads'
    - In the previous thought-experiment we ended up considering up to 10 million threads
    - Best possible scaling is predicted to be reached when using 10000 threads (for 1 second 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

Rule 3

Allow for your
Grainsize to be Variable

Overheads: The Worst of All?



STE||AR GROUP

Rule 4

Oversubscribe and Balance Adaptively

STE||AR GROUP

The Challenges

- We need to find a usable way to fully parallelize our applications
- Goals are:
  - Expose asynchrony to the programmer without exposing additional concurrency
  - Make data dependencies explicit, hide notion of ‘thread’ and ‘communication’
  - Provide manageable paradigms for handling parallelism


Getty

Proposed Solution

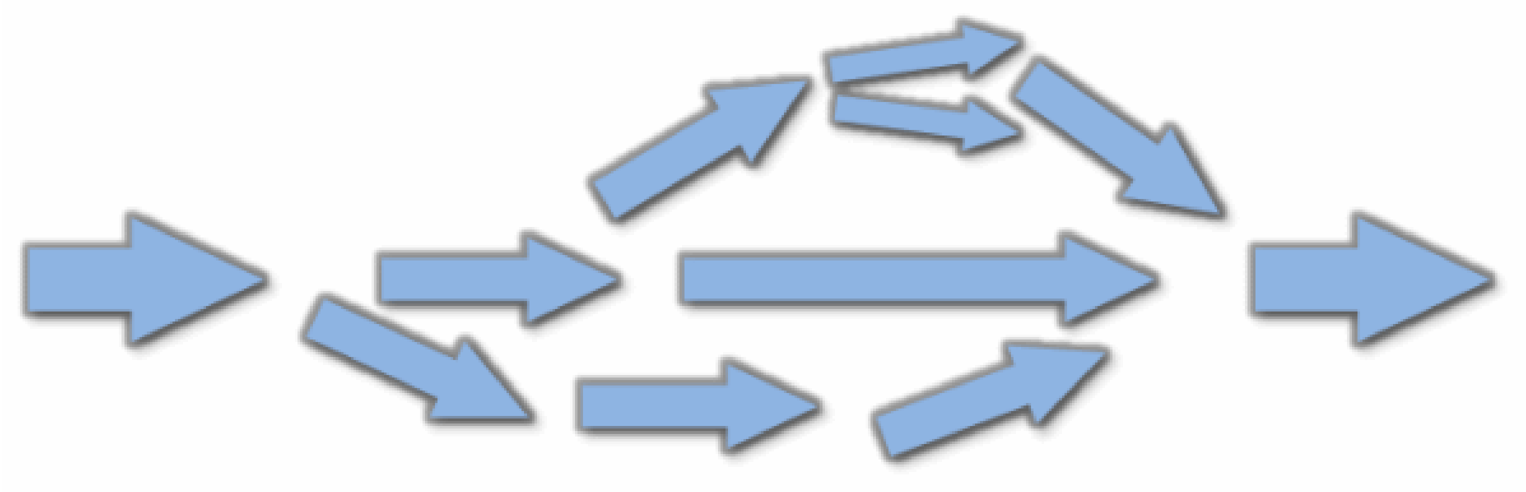
- Asynchronous programming model
  - Objects interact using asynchronous functions calls
    - Remote calls are sent as active messages
  - Futures are used to represent data dependencies in asynchronous execution and dataflow
  - View the entire (super-)computer as a single C++ abstract machine (AGAS: active global address space)
  - Tasks operate on C++ objects possibly distributed across the system

STE||AR GROUP

Proposed Solution

- Semantic and syntactic equivalence of local and remote operation
  - Enables performance portability
  - Unified approach to vector-, core-, and node- level parallelism
- Futurization technique
  - Formal way of transforming sequential code into auto-parallelized, asynchronous code
- Fully conforming to API as prescribed by C++ Standard

The Future of Computation



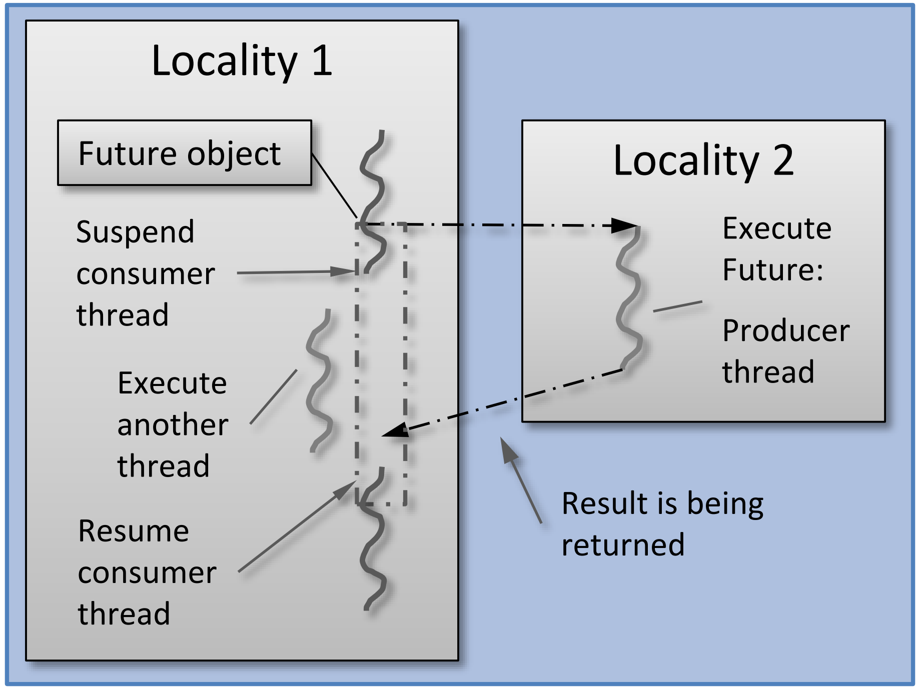
What is a (the) Future?

- Many ways to get hold of a (the) 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
}
```

What is a (the) future

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



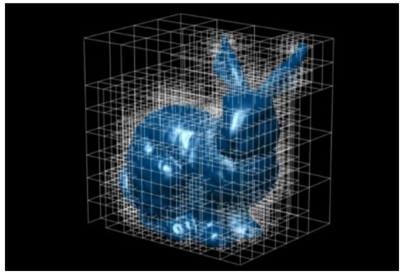
- Enables transparent synchronization with producer
- Hides notion of dealing with threads
- Makes asynchrony manageable
- Allows for composition of several asynchronous operations
- (Turns concurrency into parallelism)

STE||AR GROUP

Recursive Parallelism

STE||AR GROUP

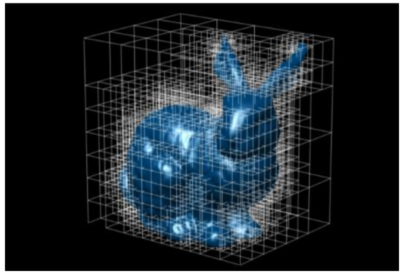
Traversing a Recursive Tree



```
T tree_node::traverse()
{
    if (has_children()) {
        std::array<T, 8> results; // 8 for children.

        for (int i = 0; i != 8; ++i)
            results[i] = children[i].traverse();

        return combine_results(results, compute_result());
    }
    return compute_result();     // perform calculations for leaf
}
```

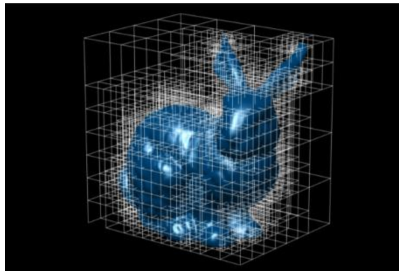
STE||AR GROUP

Traversing a Recursive Tree



```
T traverse(tree_node const &t)
{
    if (t.has_children()) {
        std::array<future<T>, 8> results;      // 8 for children

        for (int i = 0; i != 8; ++i)
            results[i] = async(traverse, t.children[i]);
        T r = t.compute_result();

        wait_all(results);

        return t.combine_results(results, r);
    }
    return t.compute_result();
}
```

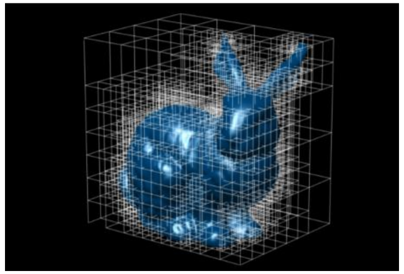
Traversing a Recursive Tree



```
future<T> traverse(tree_node const &t)
{
    if (t.has_children()) {
        std::array<future<T>, 8> results;      // 8 for children

        for (int i = 0; i != 8; ++i)
            results[i] = async(traverse, t.children[i]);

        return when_all(results, t.compute_result()).then(
             [](auto f, auto r) { return combine_results(f, r); }
        );
    }
    return t.compute_result();
}
```

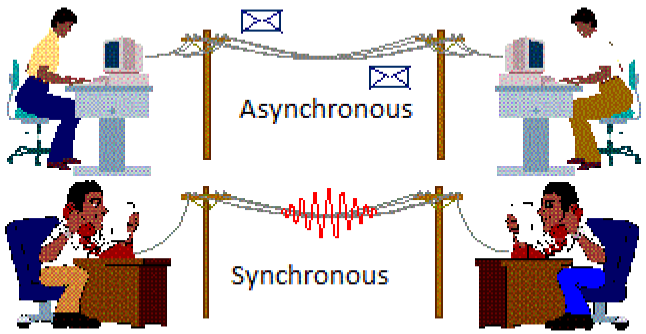
STE||AR GROUP

Traversing a Recursive Tree: co_await



```
future<T> traverse(tree_node const &t)
{
    if (t.has_children()) {
        std::array<future<T>, 8> results;    // 8 for children.

        for (int i = 0; i != 8; ++i)
            results[i] = async(traverse, t.children[i]);

        co_return t.combine_results(co_await results, co_await t.compute_result());
    }
    co_return t.compute_result();            // perform calculations for leaf
}
```

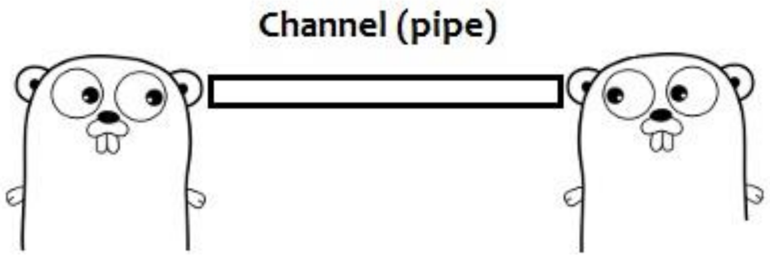
STE||AR GROUP



Asynchronous Communication

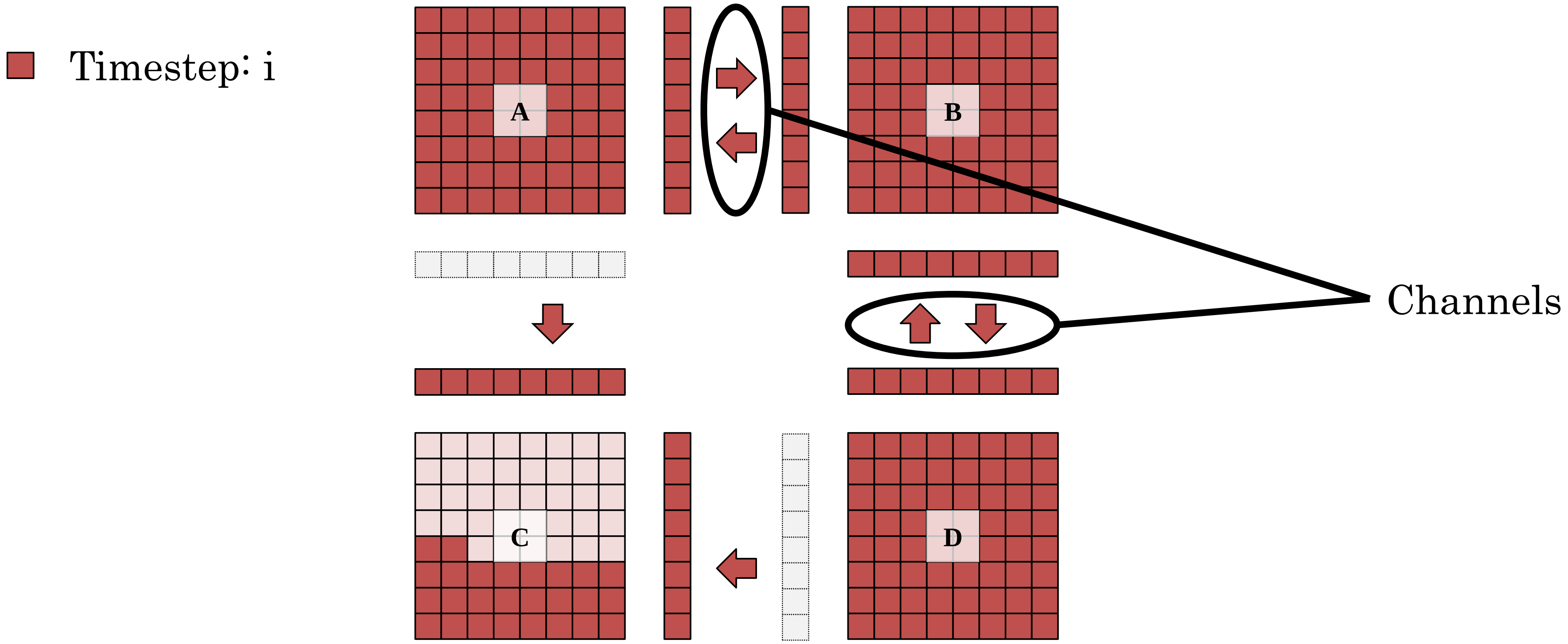
Asynchronous Channels

- High level abstraction of communication operations
  - Perfect for asynchronous boundary exchange
- Modelled after Go-channels
- Create on one locality, refer to it from another locality
  - Conceptually similar to bidirectional P2P (MPI) communicators
- Asynchronous in nature
  - `channel::get()` and `channel::set()` return futures



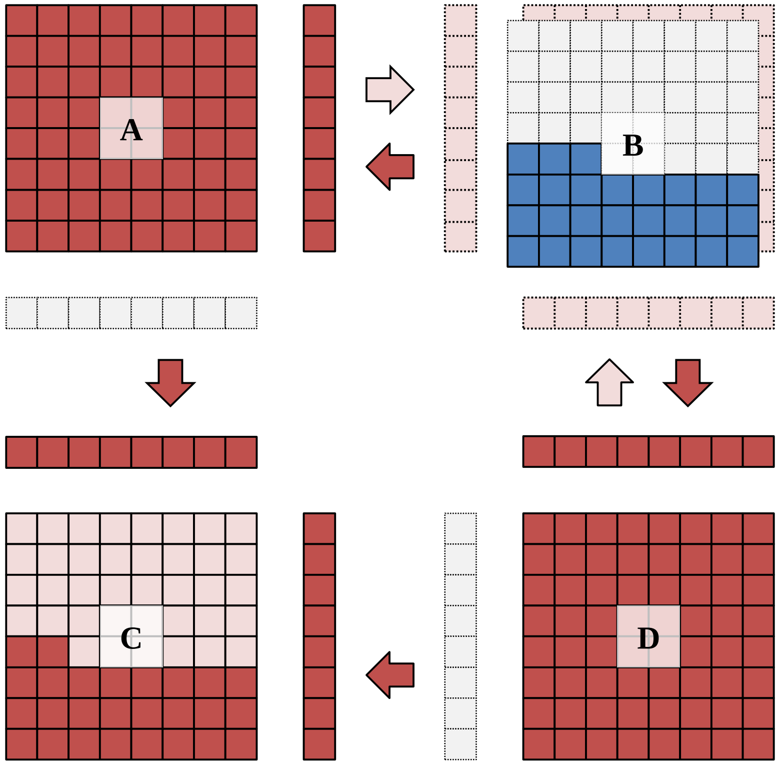
STE||AR GROUP

Futurized 2D Stencil: Timestep i



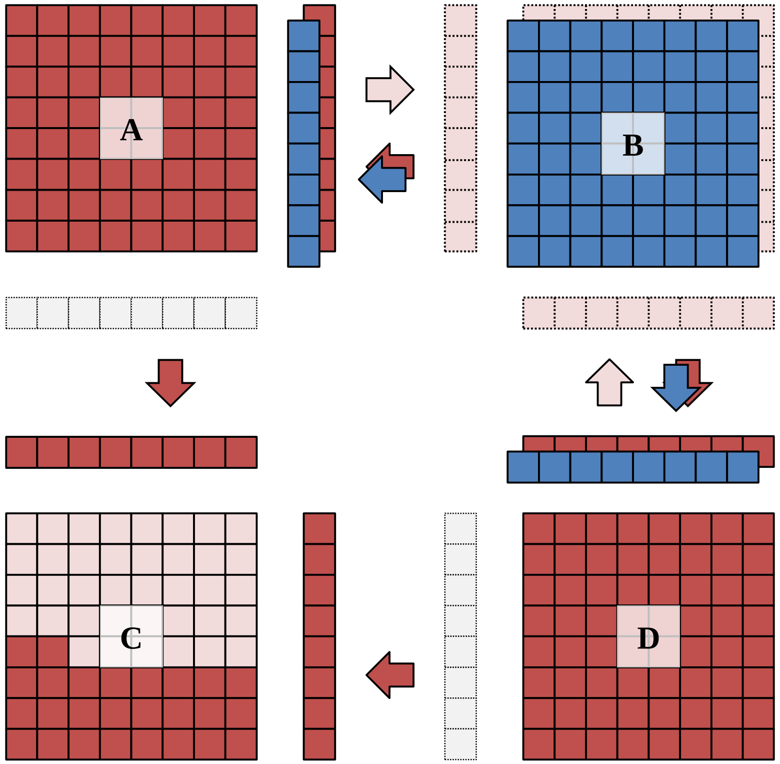
Futurized 2D Stencil: Timestep i+1

- Timestep: i
- Timestep: i+1



Futurized 2D Stencil

- Timestep: i
- Timestep: i+1



STE||AR GROUP

2D Stencil

- Partitions are distributed across machine
- More partitions per node (locality) than cores
  - Oversubscription
- Code equivalent regardless whether neighboring partition is on the same node
- Overlap of communication and computation
  - More parallelism (work) than compute resources (cores)

Asynchrony Everywhere

Futurized 2D Stencil: Main Loop

```
hpx::future<void> step_future = make_ready_future();
for (std::size_t t = 0; t != steps; ++t)
{
    step_future = step_future.then(
        [t](hpx::future<void> &&)
        {
            return perform_one_time_step(t);
        });
}
step_future.get();    // wait for everything to finish
```

STE||AR GROUP

One Timestep: Update Boundaries

```
hpx::future<void> partition::perform_one_time_step(int t)
{
    // Update our boundaries from neighbors
    hpx::future<void> top_boundary_future = channel_up_from.get(t)
        .then([](hpx::future<std::vector<double>> && up_future)
        {
            std::vector<double> data = f.get();    // does not block

            // process ghost-zone data using received data

            // send new ghost zone data to neighbor
            return channel_up_to.set(data);
        });

    // Apply stencil to partition
}
```

STE||AR GROUP

One Timestep: Interior

```
hpx::future<void> partition::perform_one_time_step(int t)
{
    // Update our boundaries from neighbors

    // Apply stencil to partition
    hpx::future<void> interior_future =
        hpx::parallel::for_loop(
            par(task), min+1, max-1,
            [](int idx)
            {
                // apply stencil to each point
            });

    // Join all asynchronous operations
}
```

STE||AR GROUP

One Timestep: Wrap-up

```
hpx::future<void> partition::perform_one_time_step(int t)
{
    // Update our boundaries from neighbors

    // Apply stencil to partition

    // Join all asynchronous operations
    return when_all(
        top_boundary_future, bottom_boundary_future,
        left_boundary_future, right_boundary_future,
        interior_future);
}
```

STE||AR GROUP

Futurization

- 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

Futurization

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, …);`*

* `future<future<T>>` collapses (unwraps) to `future<T>`

HPX

The C++ Standards Library for Concurrency and Parallelism

STE||AR GROUP

HPX – A General Purpose Runtime System

- The C++ Standards Library for Concurrency and Parallelism
- Exposes a coherent and uniform, standards-oriented API for ease of programming parallel, distributed, and heterogeneous applications.
  - Enables to write fully asynchronous code using hundreds of millions of threads.
  - Provides unified syntax and semantics for local and remote operations.

HPX – A General Purpose Runtime System

- HPX represents an innovative mixture of
  - A global system-wide address space (AGAS - Active Global Address Space)
  - Fine grain parallelism and lightweight synchronization
  - Combined with implicit, work queue based, message driven computation
  - Full semantic equivalence of local and remote execution, and
  - Explicit support for hardware accelerators and vectorization

STE||AR GROUP

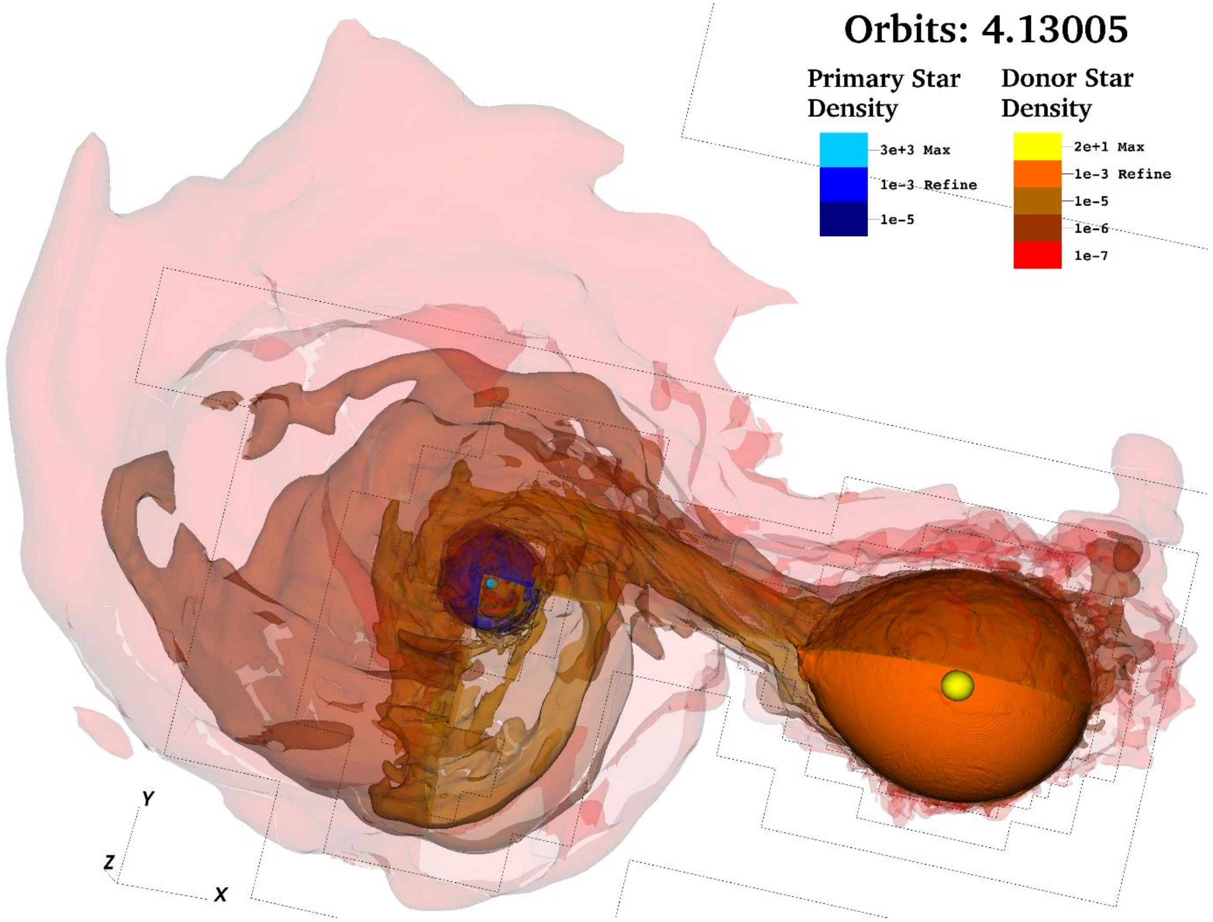
HPX – A General Purpose Runtime System

- Enables writing applications which out-perform and out-scale existing applications based on OpenMP/MPI
  - 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

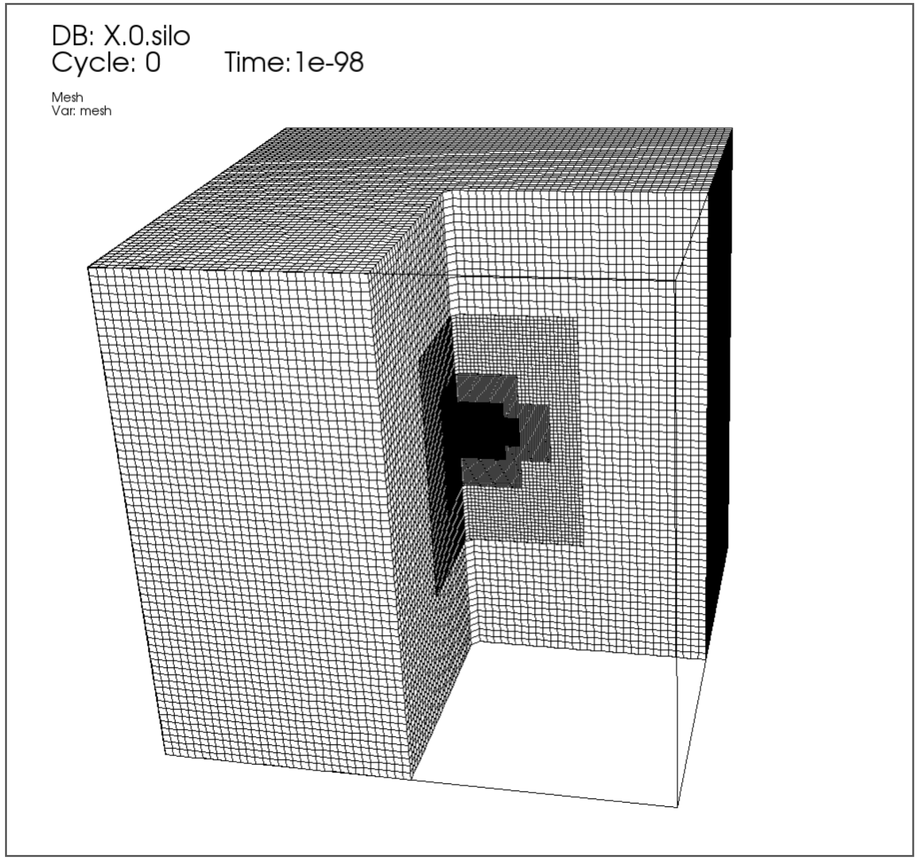
Recent Results

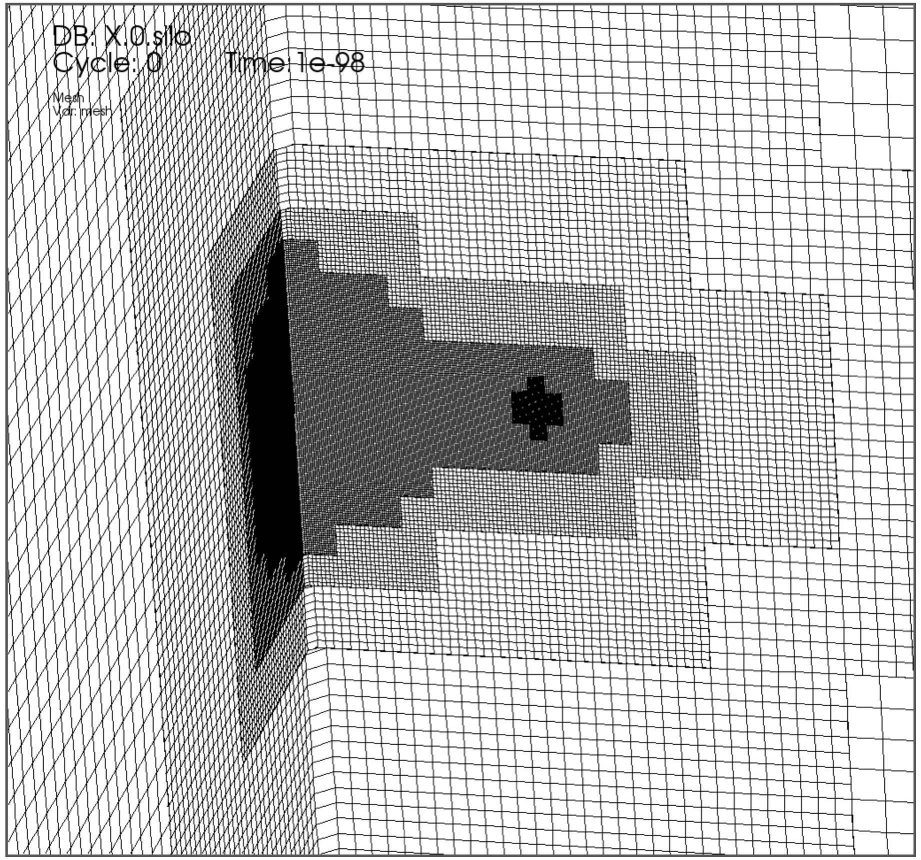
STE||AR GROUP

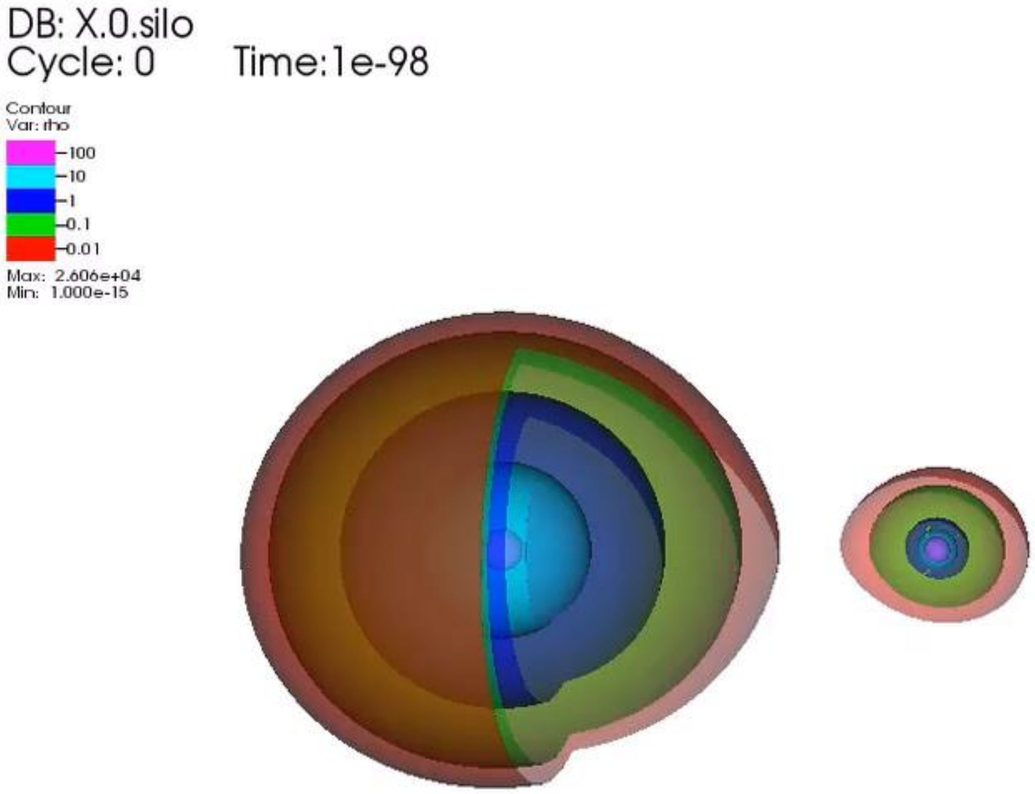
Merging White Dwarfs



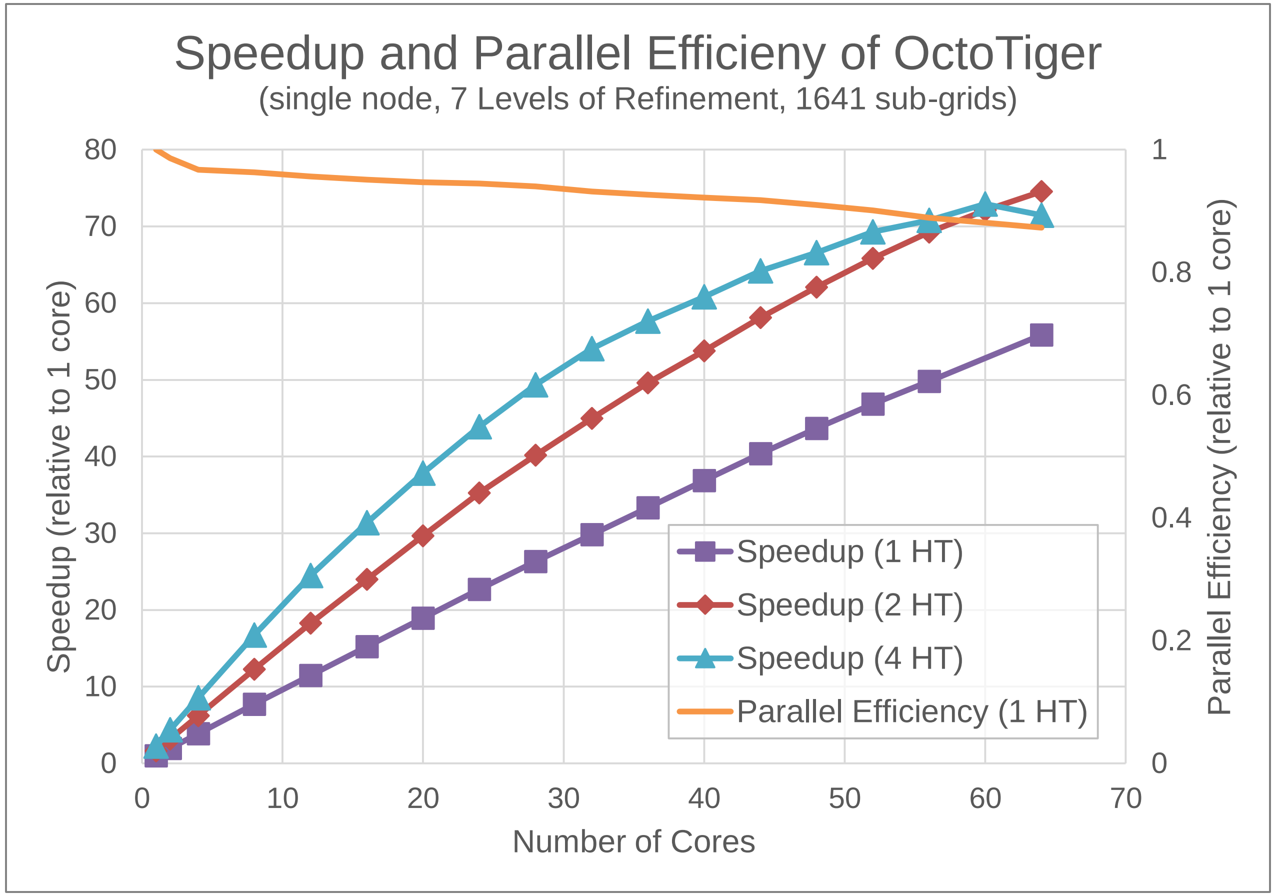
Adaptive Mesh Refinement





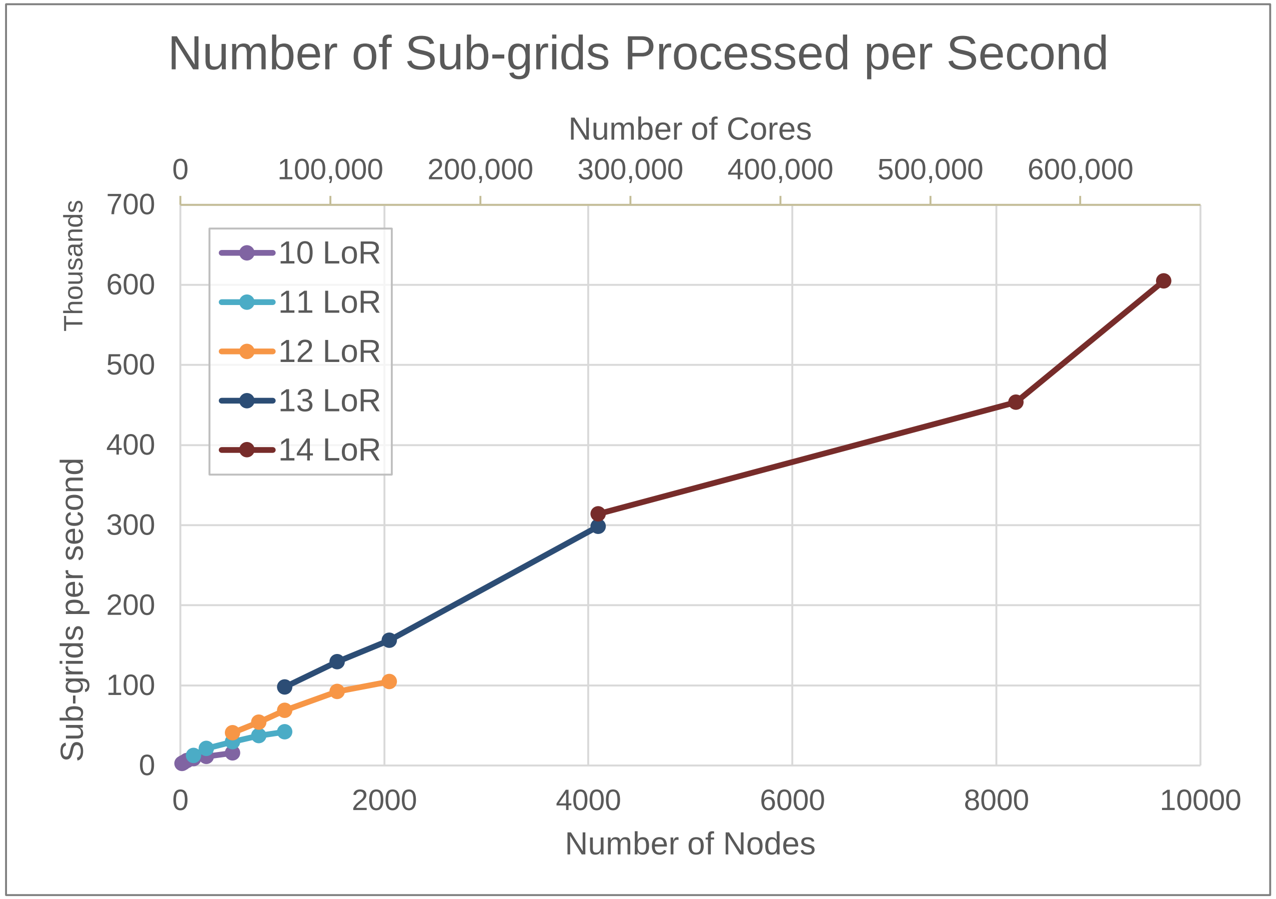

DB: X.0.silo
Cycle: 0
Time:1e-98
Contour
Var: rho
100
10
1
0.1
0.01
Max: 2.606e+04
Min: 1.000e-15

Adaptive Mesh Refinement



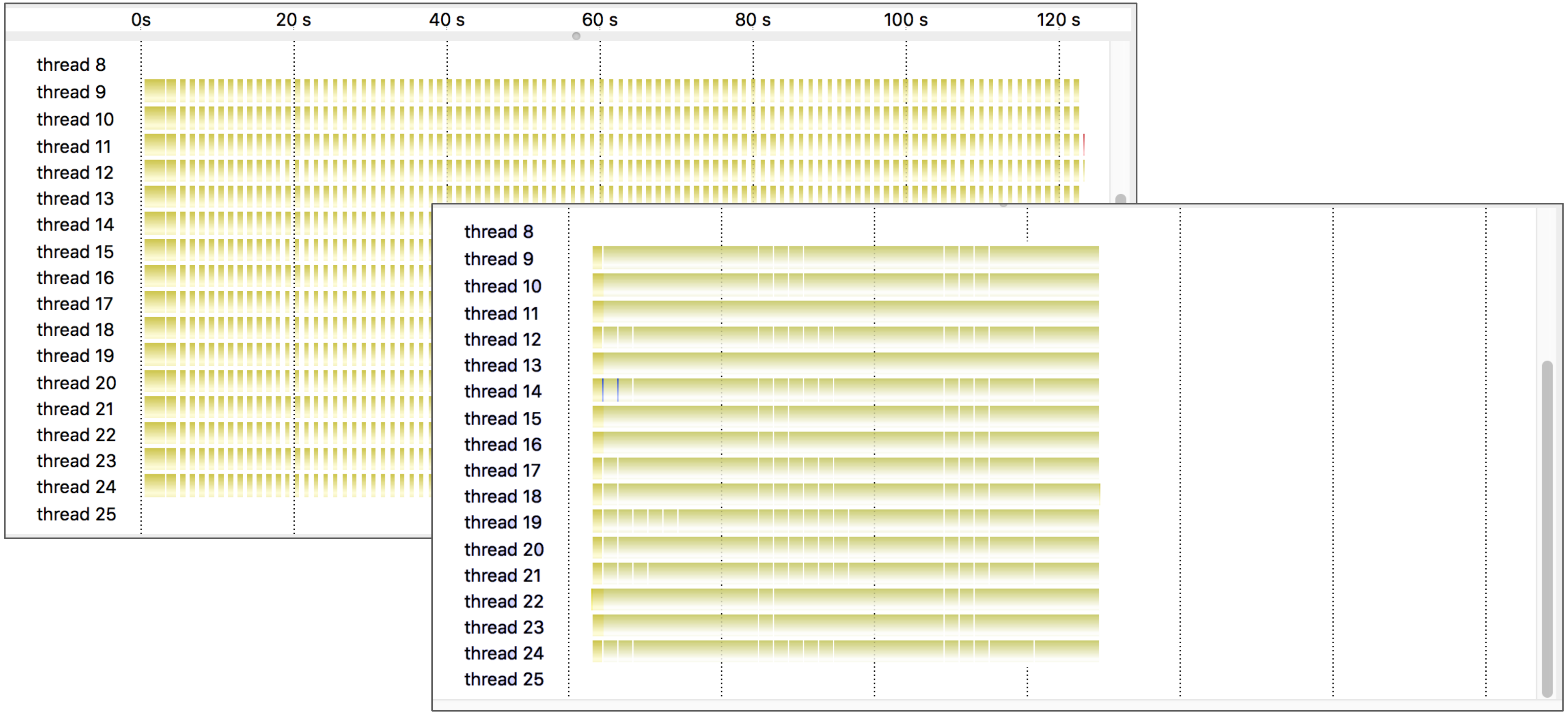
Cori II (NERSC)

Adaptive Mesh Refinement



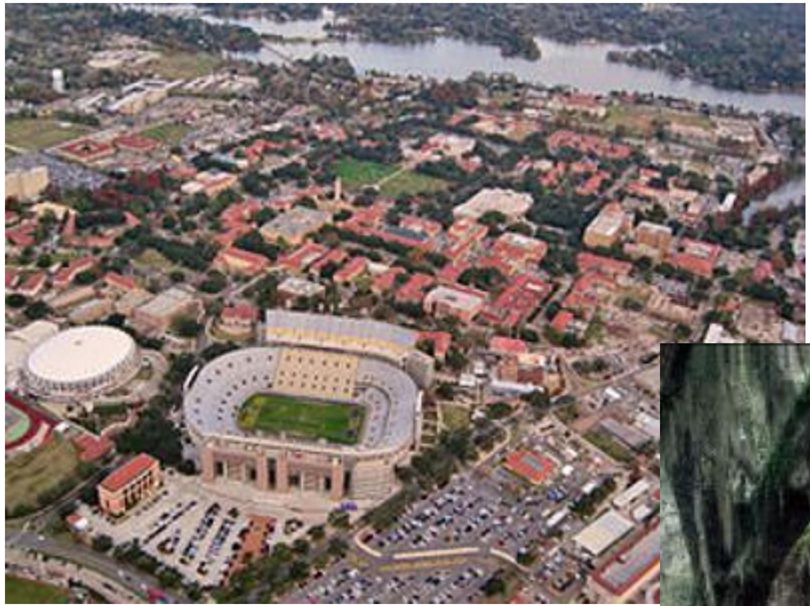
Cori II (NERSC)

The Solution to the Application Problem

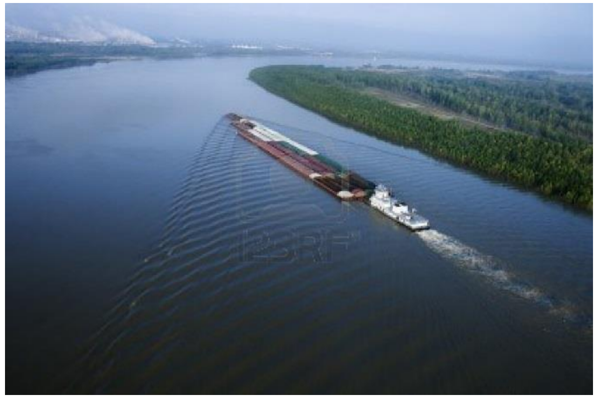


The Solution to the Application Problem

STE||AR GROUP



LSU CENTER FOR COMPUTATION & TECHNOLOGY



STE||AR GROUP