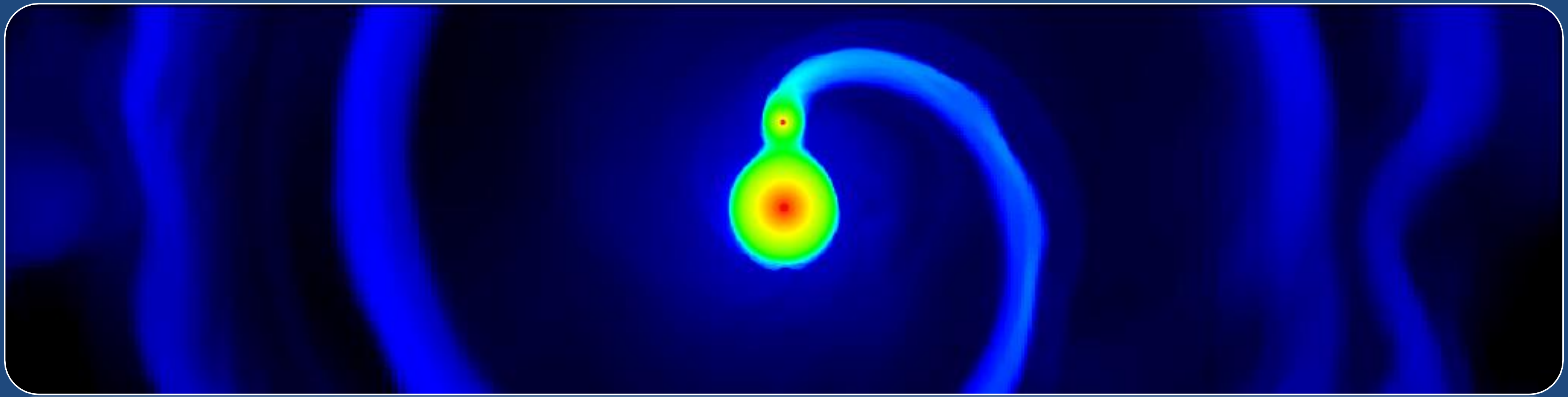
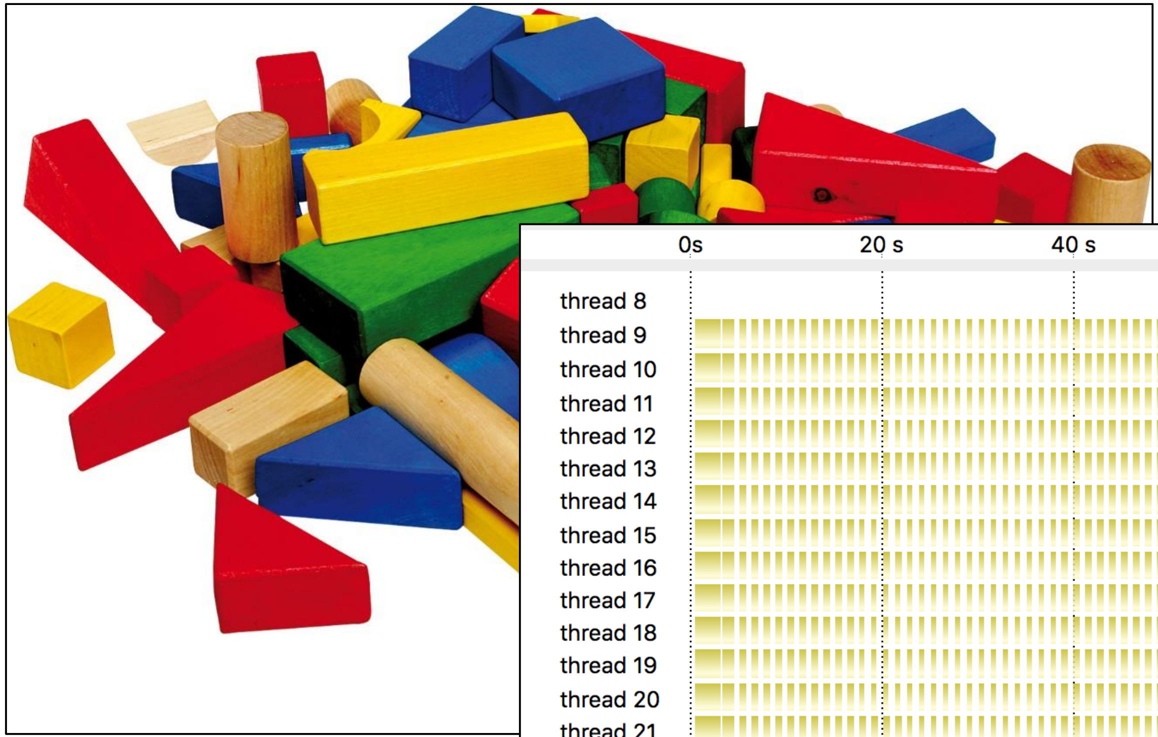
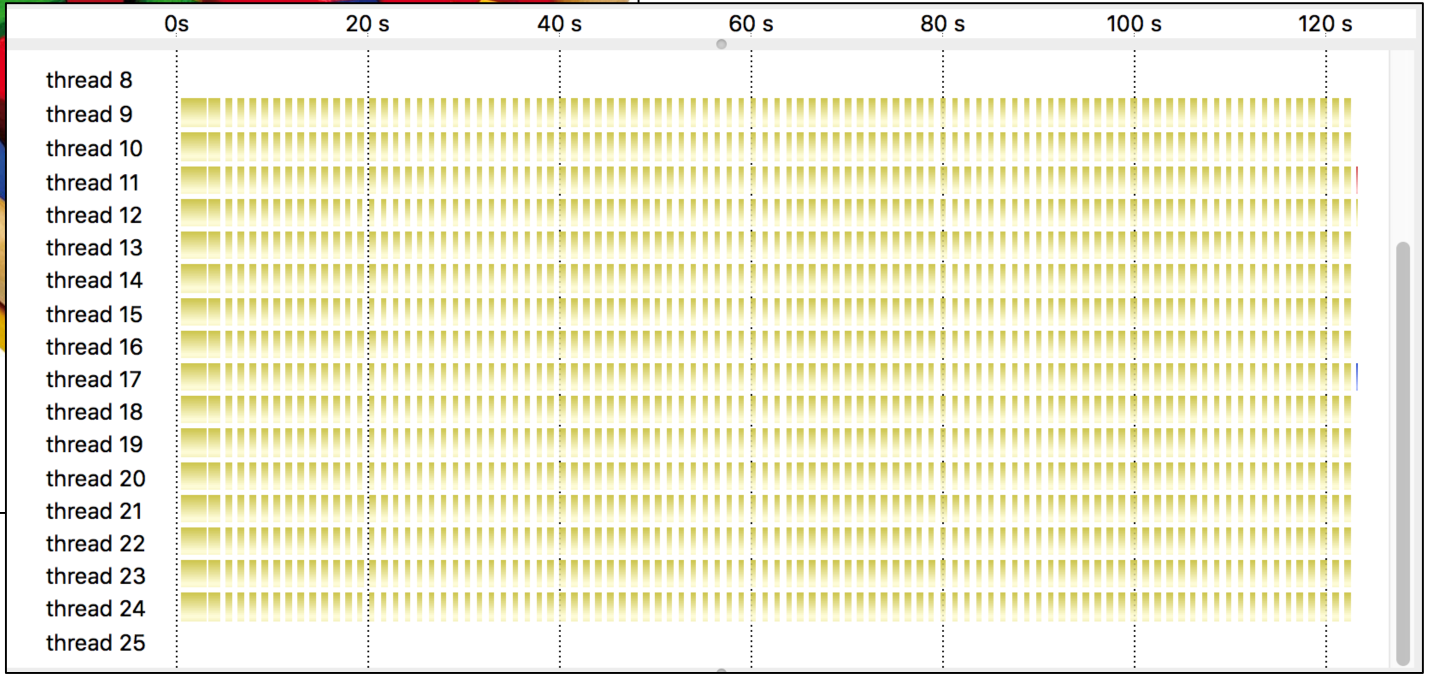


Towards a Standard C++ Asynchronous Programming Model

Hartmut Kaiser (hkaiser@cct.lsu.edu)

Today's Parallel Applications





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.

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
- (CppCon 2017: Asynchronous C++ Programming Model)

STE||AR GROUP

HPX

The C++ Standards Library for Concurrency and Parallelism

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

STE||AR GROUP

HPX – 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.
- Enables using the Asynchronous C++ Standard Programming Model
  - Emergent auto-parallelization, intrinsic hiding of latencies,

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 parallel algorithms
    - One of the first full openly available implementations
    - Extensions:
      - asynchronous execution
      - parallel range based algorithms
      - vectorizing execution policies `simd`/`par_simd`
  - Distributed operation
    - Extending the standard interfaces
    - Global address space, load balancing

STE||AR GROUP

HPX – The API

- As close as possible to C++11/14/17/20 standard library, where appropriate, e.g.

std::thread, std::jthread	hpx::thread (C++11), hpx::jthread (C++20)
std::mutex	hpx::mutex
std::future	hpx::future (including N4538, ‘Concurrency TS’)
std::async	hpx::async (including N3632)
std::for_each(par, …), etc.	hpx::for_each (N4507, C++17)
std::experimental::task_block	hpx::parallel::task_block (N4411)
std::latch, std::barrier, std::for_loop	hpx::latch, hpx::barrier, hpx::for_loop (TS V2)
std::bind	hpx::bind
std::function	hpx::function
std::any	hpx::any (N3508)
std::cout	hpx::cout

STE||AR GROUP

Parallel Algorithms (C++17)

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 (C++17)

- Add Execution Policy as first argument
- Execution policies have associated default executor and default executor parameters
  - `execution::parallel_policy`, generated with `par`
    - parallel executor, static chunk size
  - `execution::sequenced_policy`, generated with `seq`
    - sequential executor, no chunking

```
// add execution policy
std::fill(
    std::execution::par,
    begin(d), end(d), 0.0);
```

STE||AR GROUP

Parallel Algorithms (Extensions)

```
// uses default executor: par
std::vector<double> d = { ... };
fill(execution::par, begin(d), end(d), 0.0);

// rebind par to user-defined executor (where and how to execute)
my_executor my_exec = ...;
fill(execution::par.on(my_exec), begin(d), end(d), 0.0);

// rebind par to user-defined executor and user defined executor
// parameters (affinities, chunking, scheduling, etc.)
my_params my_par = ...
fill(execution::par.on(my_exec).with(my_par), begin(d), end(d), 0.0);
```

Execution Policies (Extensions)

- Extensions: asynchronous execution policies

  - `parallel_task_execution_policy` (asynchronous version of `parallel_execution_policy`), generated with `par(task)`
  - `sequenced_task_execution_policy` (asynchronous version of `sequenced_execution_policy`), generated with `seq(task)`

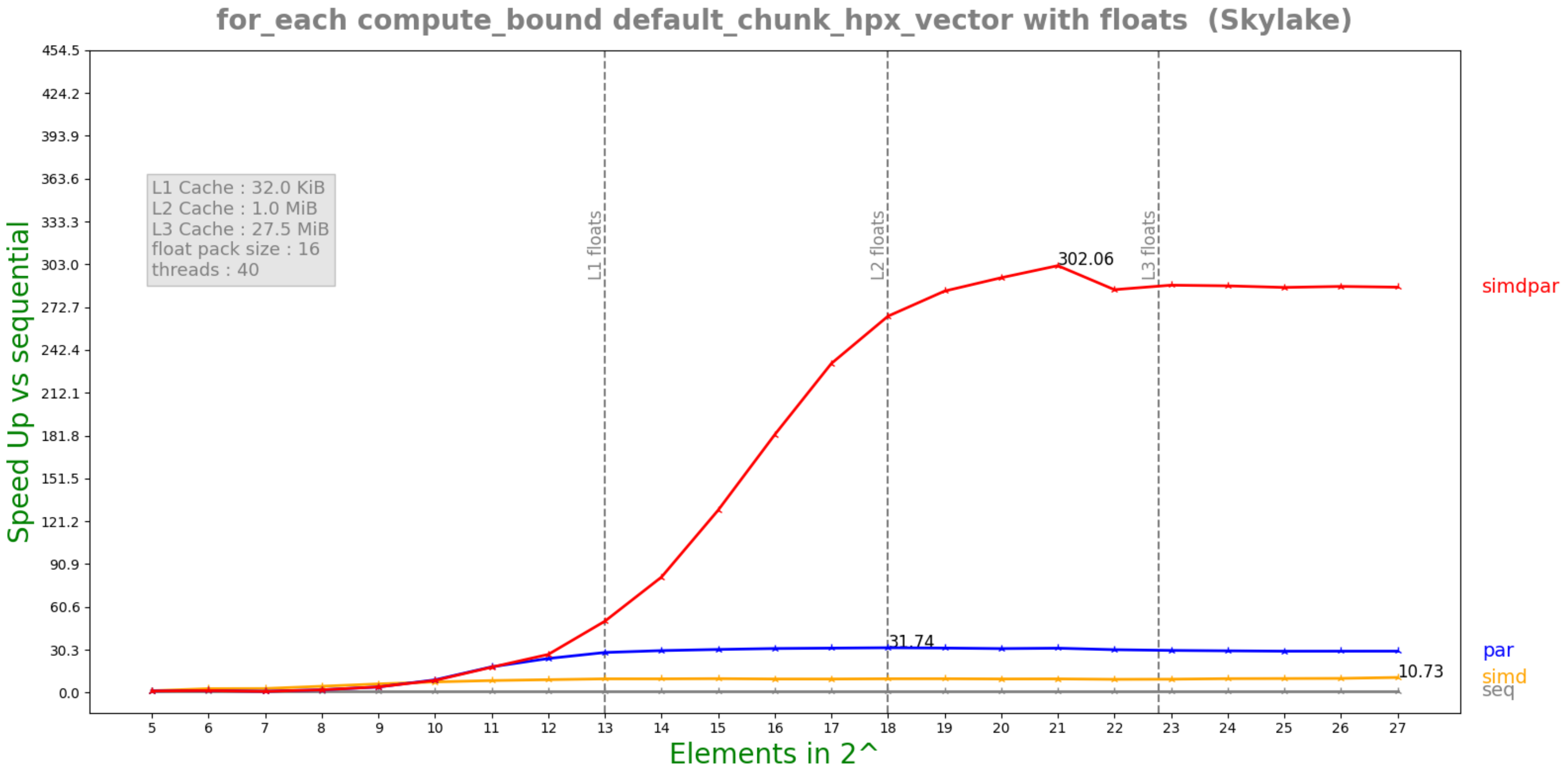
  - In all cases the formerly synchronous functions return a future<>
  - Instruct the parallel construct to be executed asynchronously
  - Allows integration with asynchronous control flow

STE||AR GROUP

Execution Policies (Extensions)

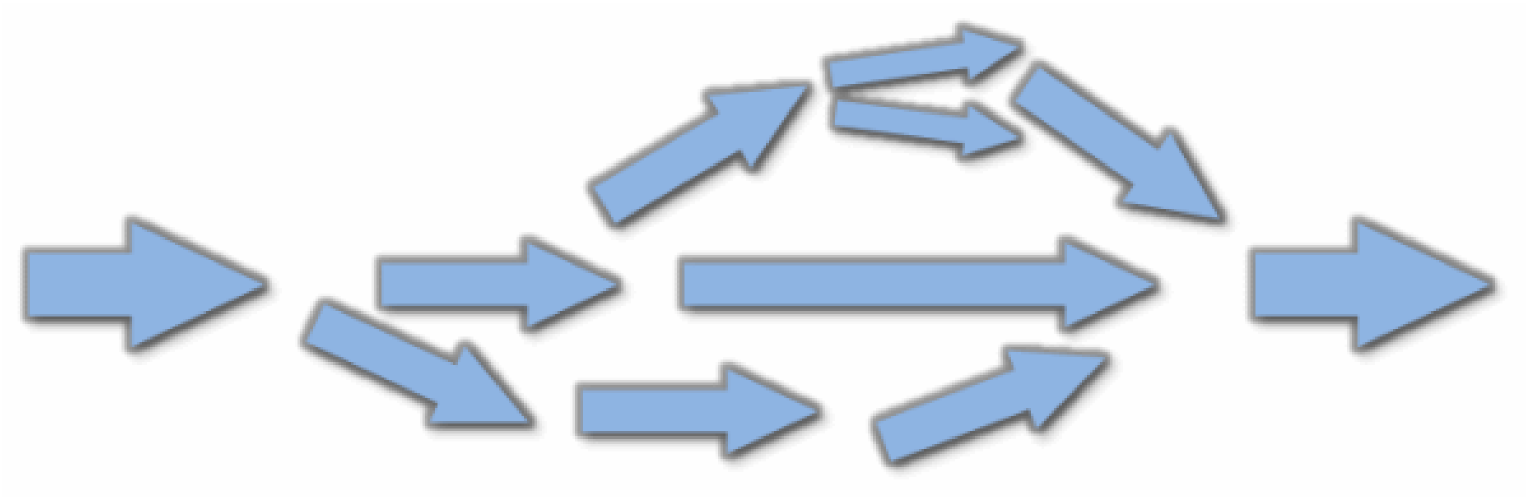
- Extensions: vectorizing execution policies (for Parallelism TS V2)
  - `simd_execution_policy`, `simd_task_execution_policy`, generated with `simd`, `simd(task)`
  - `par_simd_execution_policy`, `par_simd_task_execution_policy`, generated with `par_simd`, `par_simd(task)`
  - Calls iteration function with C++ types that represent vector registers (see: `std::experimental::simd`, gcc v11/clang v12, N4755 – Parallelism TS V2)

Execution Policies (Extensions)



STE||AR GROUP

The Future of Computation



STE||AR GROUP

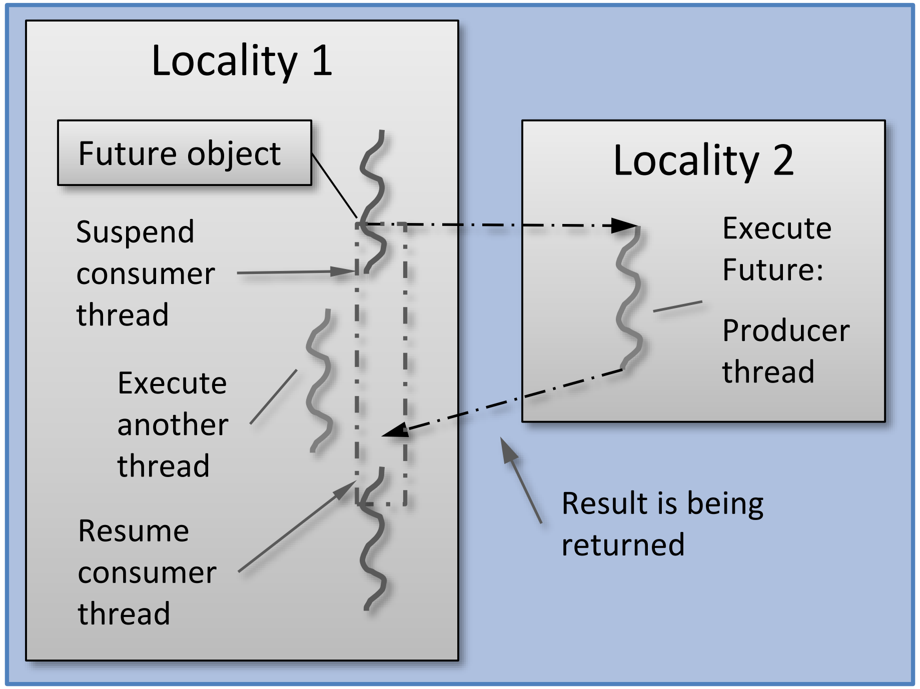
What is a (the) Future?

- 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
- Represents a data-dependency
- Makes asynchrony manageable
- Allows for composition of several asynchronous operations
- (Turns concurrency into parallelism)

STE||AR GROUP

Recursive Parallelism

STE||AR GROUP

Parallel Quicksort

```
template <typename RandomIter>
void quick_sort(RandomIter first, RandomIter last)
{
    ptrdiff_t size = last - first;
    if (size > 1) {
        RandomIter pivot = partition(first, last,
            [p = get_pivot_value(first, size)](auto v) { return v < p; });

        quick_sort(first, pivot);
        quick_sort(pivot, last);
    }
}
```

Parallel Quicksort: Parallel

```
template <typename RandomIter>
void quick_sort(RandomIter first, RandomIter last)
{
    ptrdiff_t size = last - first;
    if (size > threshold) {
        RandomIter pivot = partition(par, first, last,
            [p = get_pivot_value(first, size)](auto v) { return v < p; });

        quick_sort(first, pivot);
        quick_sort(pivot, last);
    }
    else if (size > 1) {
        sort(seq, first, last);
    }
}
```

STE||AR GROUP

Parallel Quicksort: Futurized

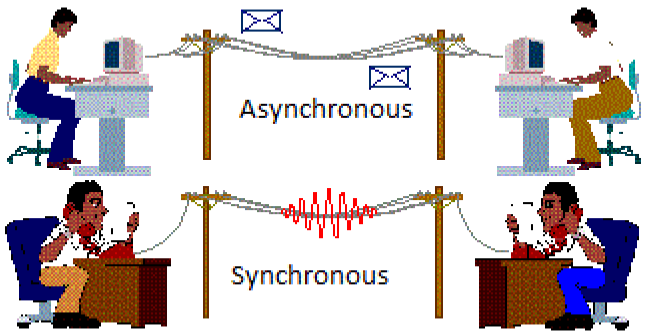
```
template <typename RandomIter>
future<void> quick_sort(RandomIter first, RandomIter last)
{
    ptrdiff_t size = last - first;
    if (size > threshold) {
        future<RandomIter> pivot = partition(par(task), first, last,
            [p = get_pivot_value(first, size)](auto v) { return v < p; });

        return pivot.then([=](auto pf) {
            auto pivot = pf.get();
            return when_all(quick_sort(first, pivot), quick_sort(pivot, last));
        });
    }
    else if (size > 1) {
        sort(seq, first, last);
    }
    return make_ready_future();
}
```

STE||AR GROUP

Parallel Quicksort: co_await

```
template <typename RandomIter>
future<void> quick_sort(RandomIter first, RandomIter last)
{
    ptrdiff_t size = last - first;
    if (size > threshold) {
        RandomIter pivot = co_await partition(par(task), first, last,
            [p = get_pivot_value(first, size)](auto v) { return v < p; });

        co_await when_all(
            quick_sort(first, pivot), quick_sort(pivot, last));
    }
    else if (size > 1) {
        sort(seq, first, last);
    }
}
```

STE||AR GROUP

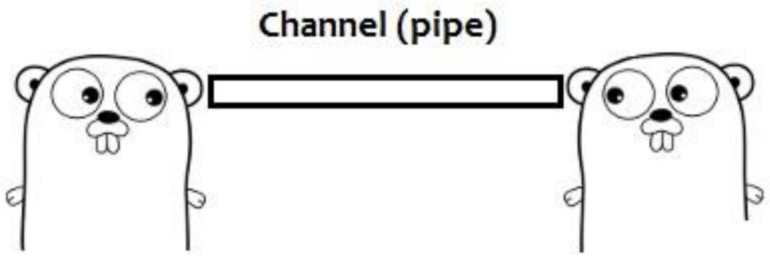


Asynchronous Communication

STE||AR GROUP

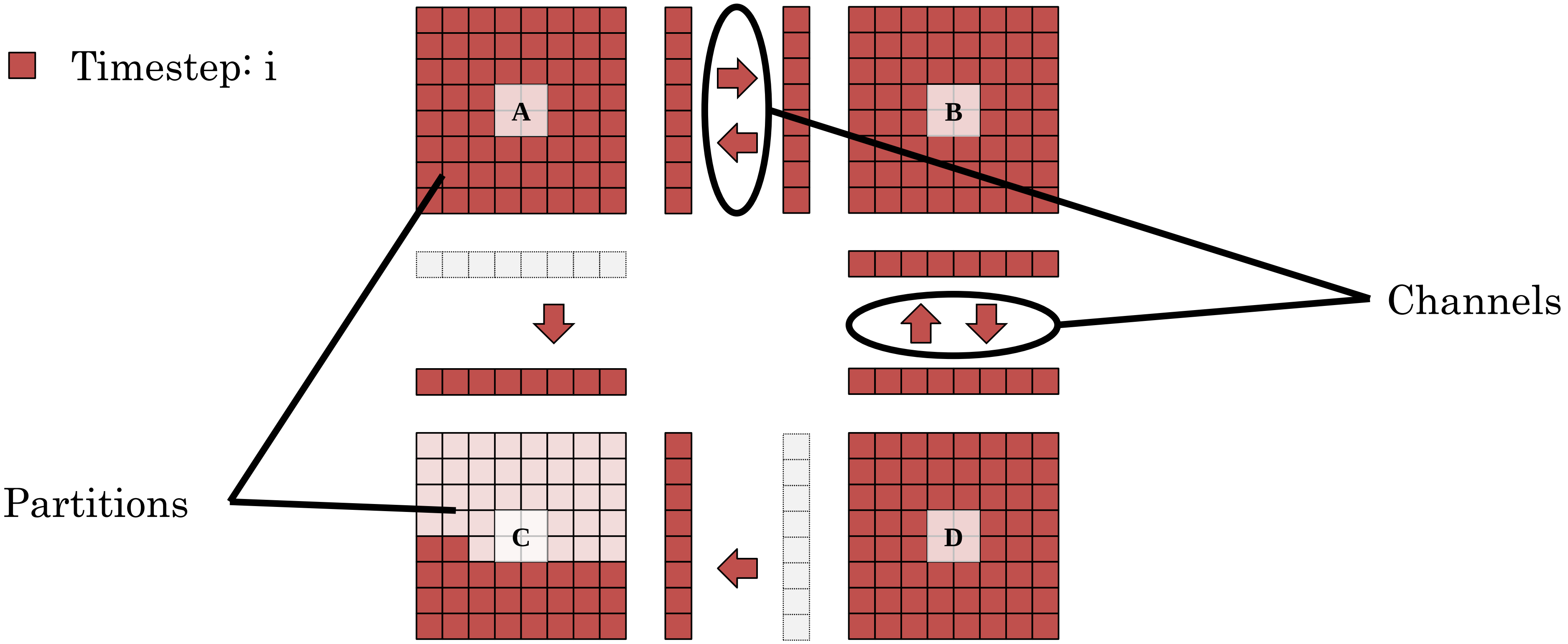
Asynchronous Channels

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



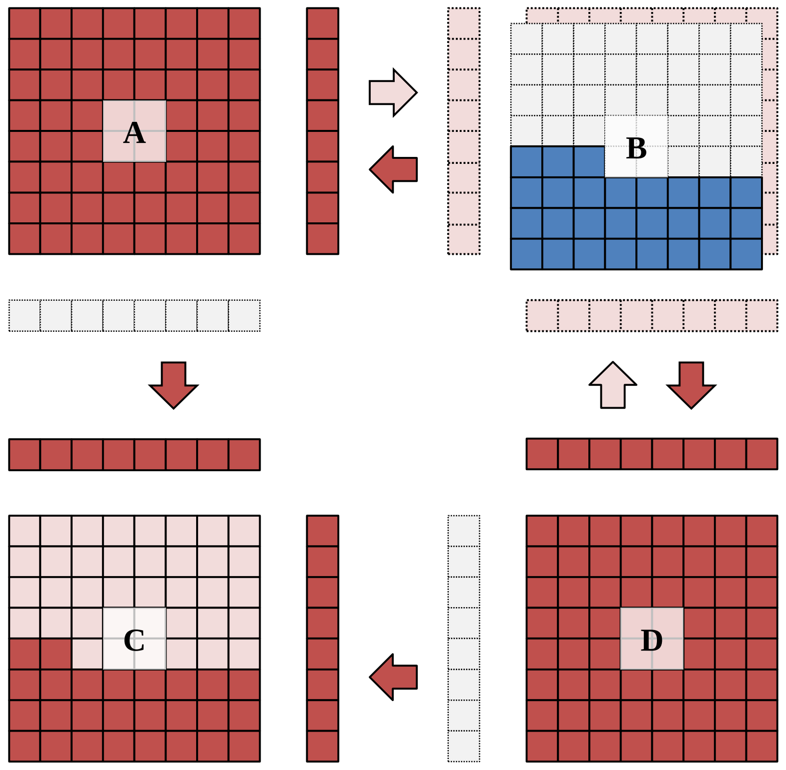
STE||AR GROUP

Futurized 2D Stencil: Timestep i

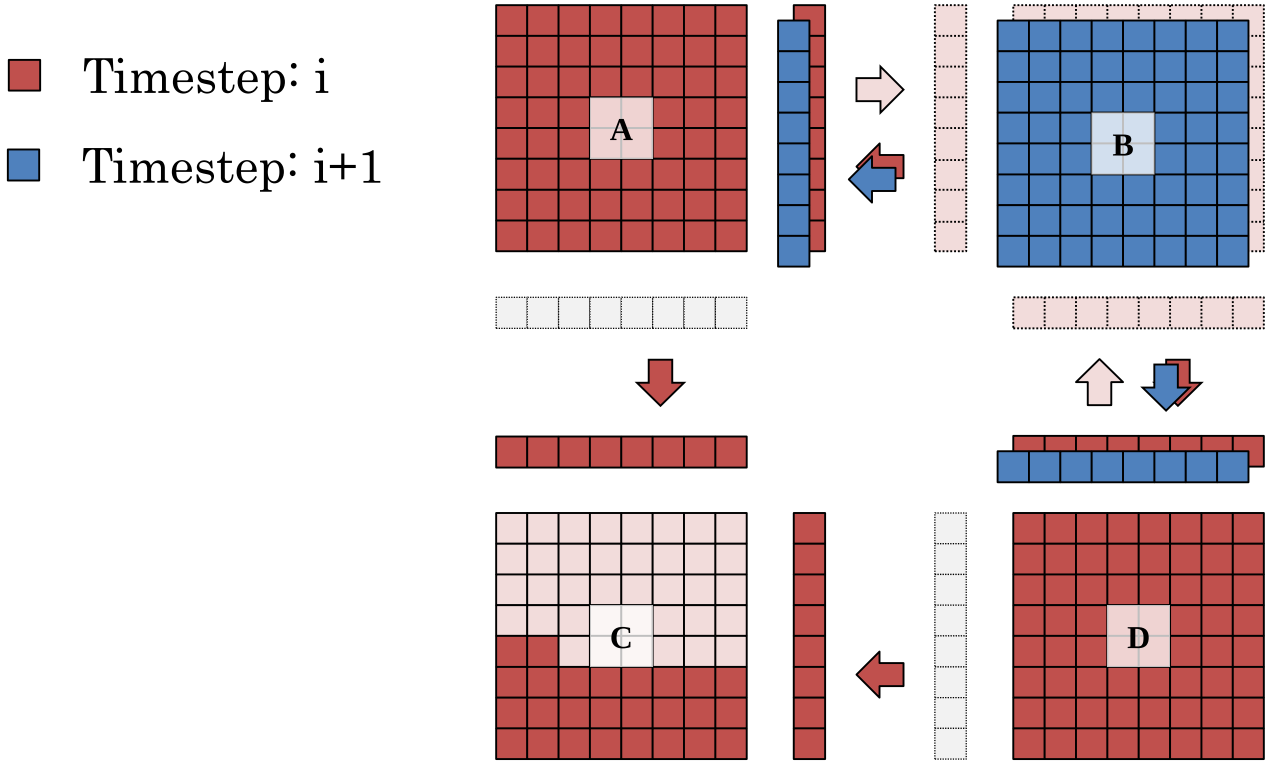


Futurized 2D Stencil: Timestep i+1

- Timestep: i
- Timestep: i+1



Futurized 2D Stencil



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)

STE||AR GROUP

Futurized 2D Stencil: Main Loop

```
// execute this for each partition concurrently
hpx::future<void> simulate(std::size_t steps)
{
    for (size_t t = 0; t != steps; ++t)
    {
        co_await perform_one_time_step(t);
    }
}
```

STE||AR GROUP

One Timestep: Update Boundaries

```
future<void> upper_boundary(int t);  // same for other boundaries

future<void> perform_one_time_step(int t)
{
    // Update our boundaries from neighbors
    co_await when_all(upper_boundary(t), right_boundary(t),
        lower_boundary(t), left_boundary(t));

    // Apply stencil to partition
    co_await for_loop(par(task), min + 1, max - 1,
        [&](size_t idx) { /* apply stencil to each inner point */ });
}
```

STE||AR GROUP

One Timestep: Interior

```
future<void> upper_boundary(int t)
{
    // Update upper boundary from upper neighbor
    vector<double> data = co_await channel_up_from.get(t);

    // process upper ghost-zone data using received data
    for_loop(seq, 1, size(data) - 1,
        [&](size_t idx) { /* apply stencil to each point in data */ });

    // send new ghost zone data to upper neighbor
    co_await channel_up_to.set(std::move(data), t + 1);
}
```

STE||AR GROUP

Asynchrony Everywhere

Futurization

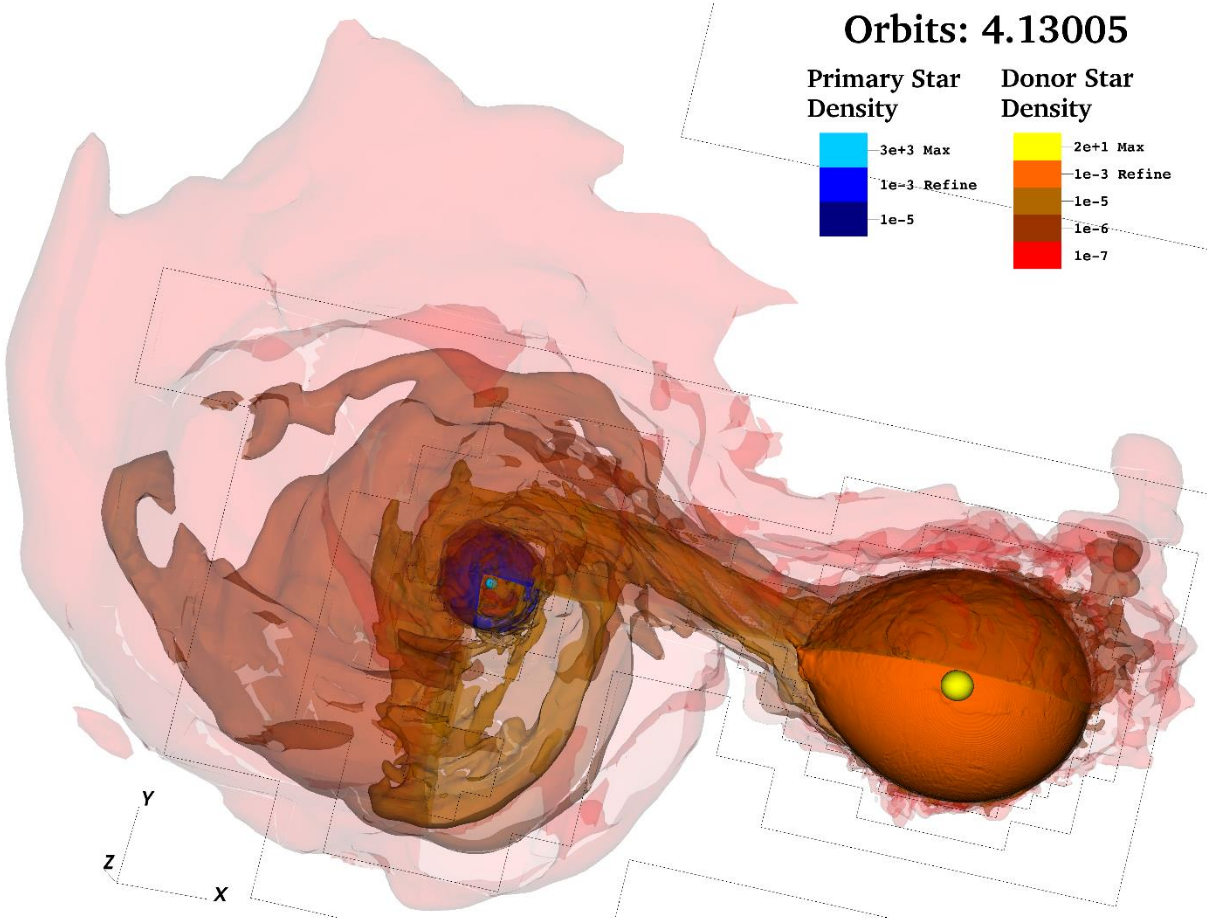
- Technique allowing to automatically transform code
  - Delay direct execution in order to avoid synchronization
  - Turns 'straight' code into 'futurized' code
  - Code no longer calculates results, but generates an execution tree representing the original algorithm
  - If the tree is executed it produces 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
- Execution exposes the emergent property of being auto-parallelized

STE||AR GROUP

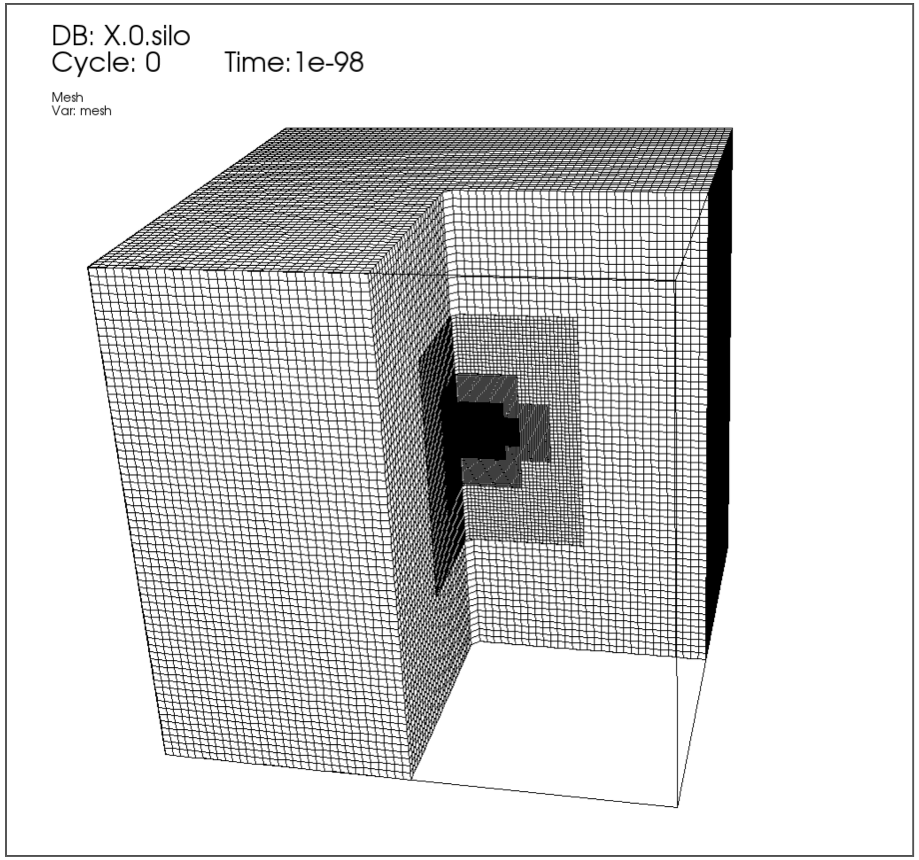
Recent Results

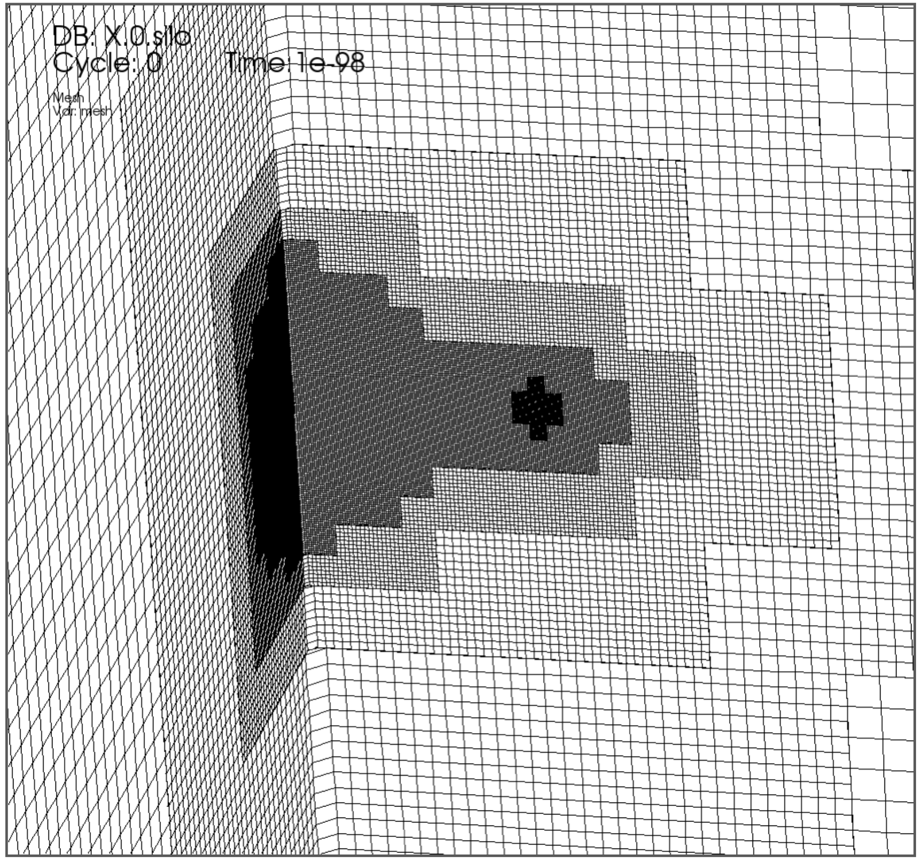
STE||AR GROUP

Merging White Dwarfs

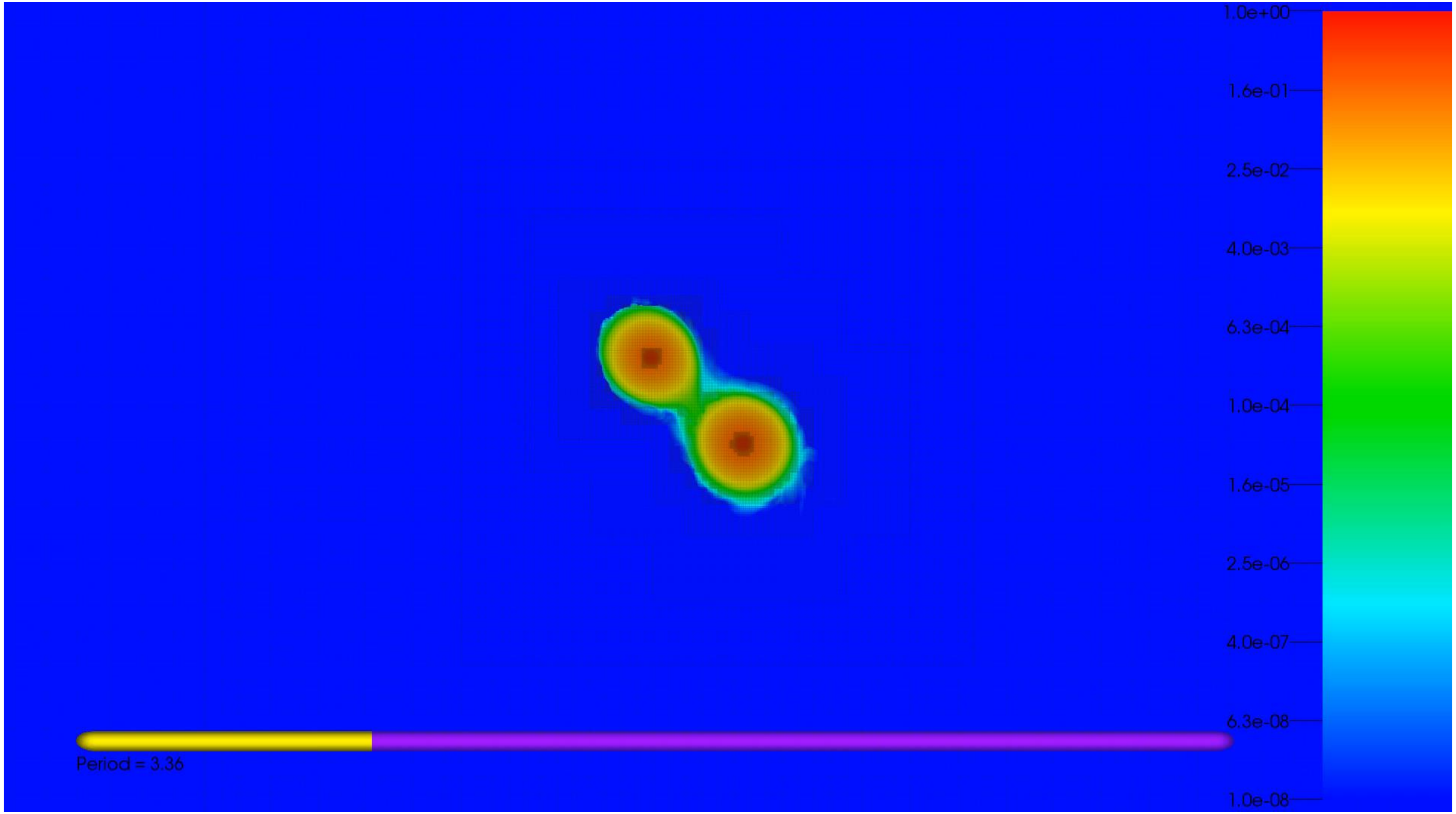


Adaptive Mesh Refinement

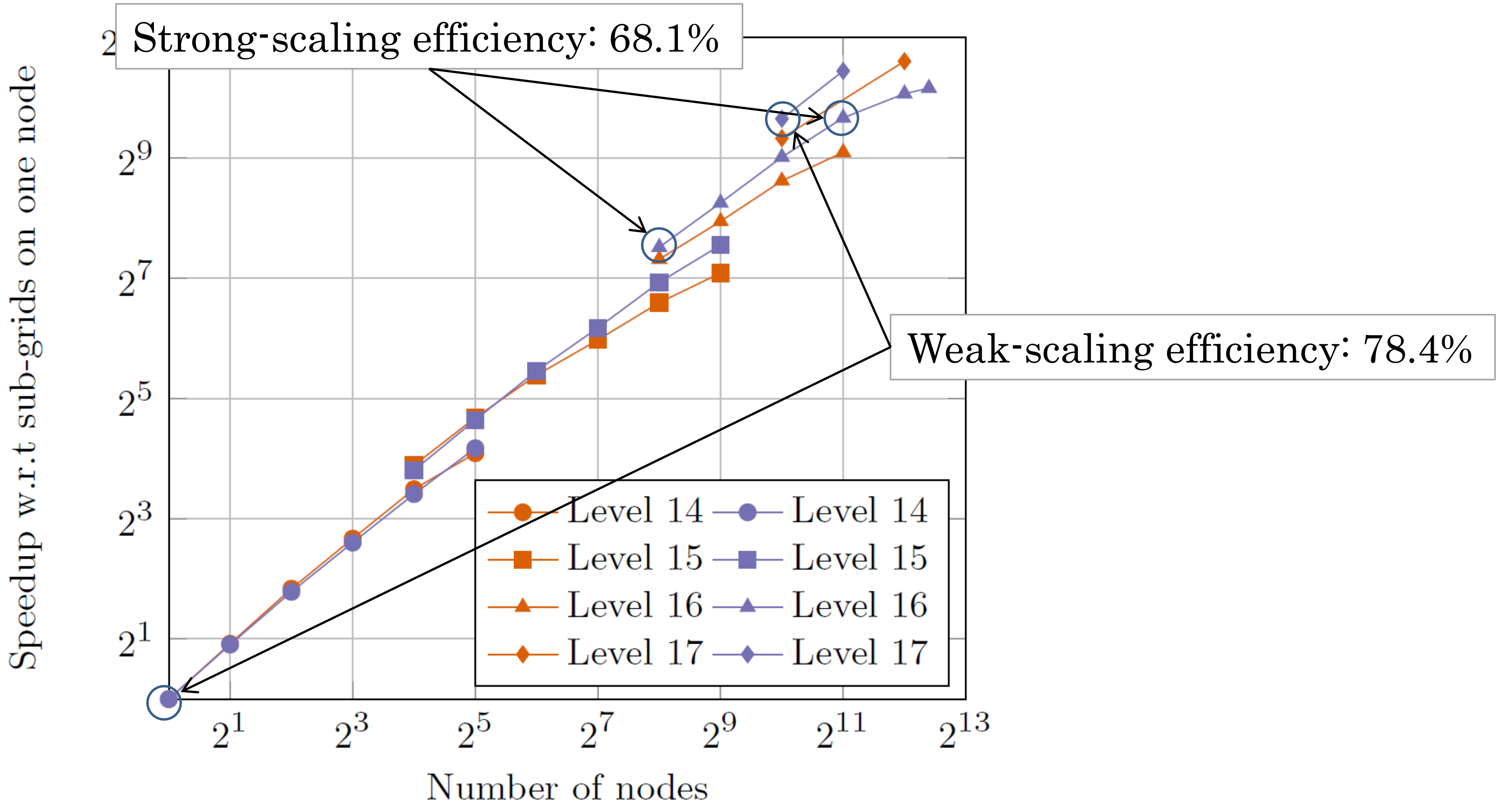




STE||AR GROUP

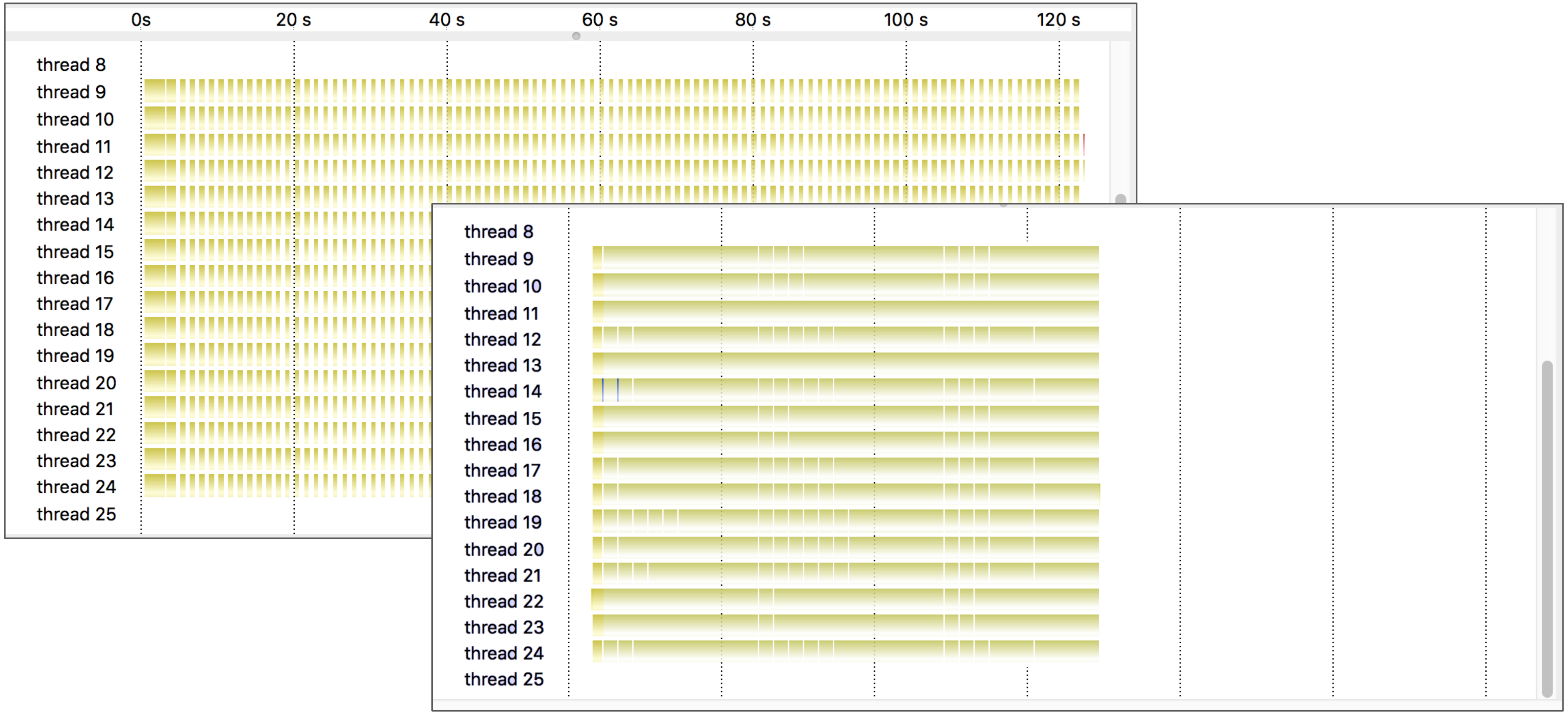

1.0e+00
1.6e-01
2.5e-02
4.0e-03
6.3e-04
1.0e-04
1.6e-05
2.5e-06
4.0e-07
6.3e-08
1.0e-08
Period = 3.36

Adaptive Mesh Refinement



STE||AR GROUP

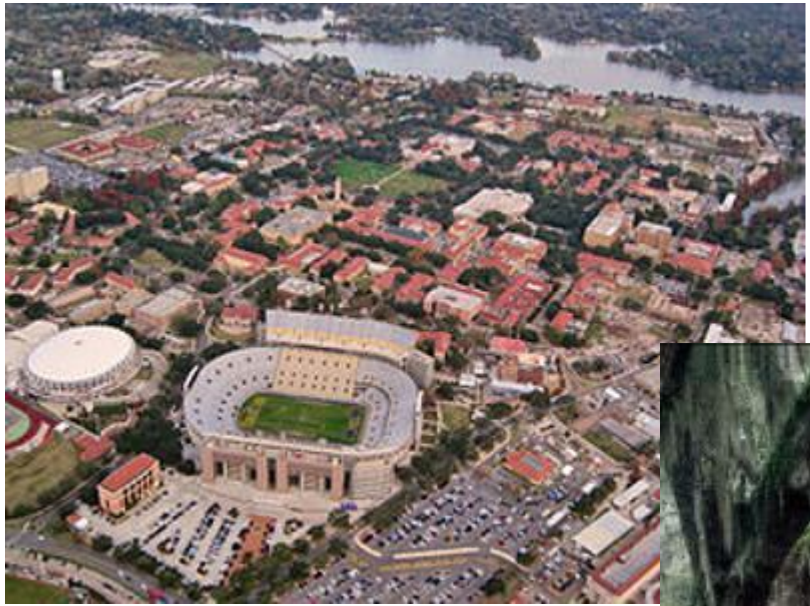
The Solution to the Application Problem



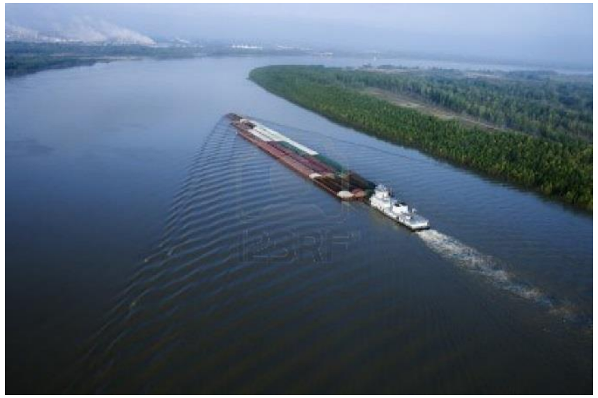
STE||AR GROUP

The Solution to the Application Problems

STE||AR GROUP



LSU CENTER FOR COMPUTATION & TECHNOLOGY



STE||AR GROUP