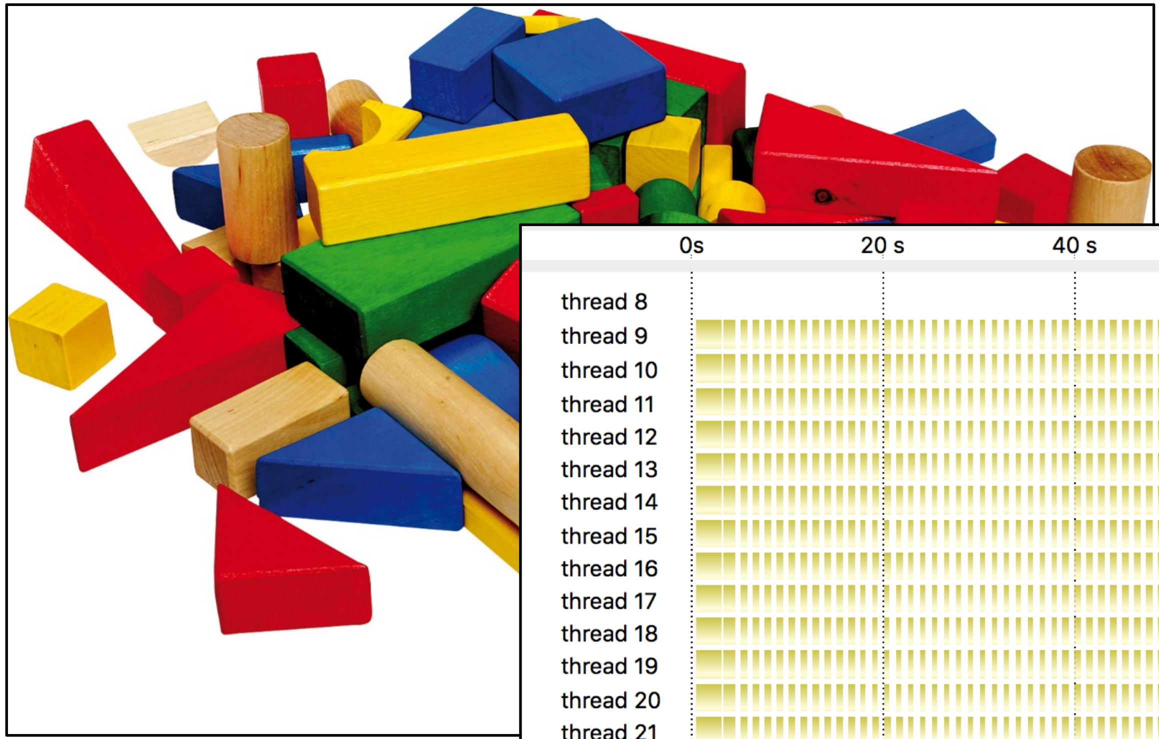
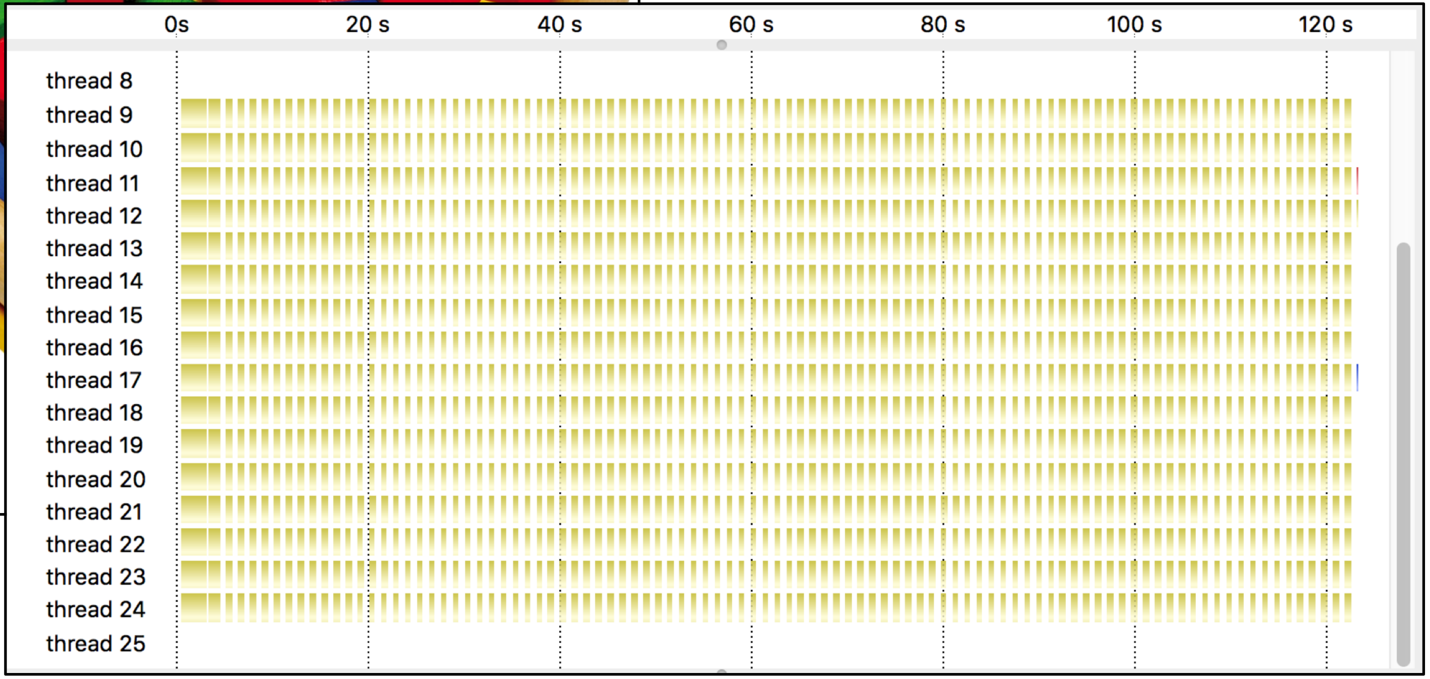
Asynchronous Programming in Modern C++

Hartmut Kaiser (hkaiser@cct.lsu.edu)

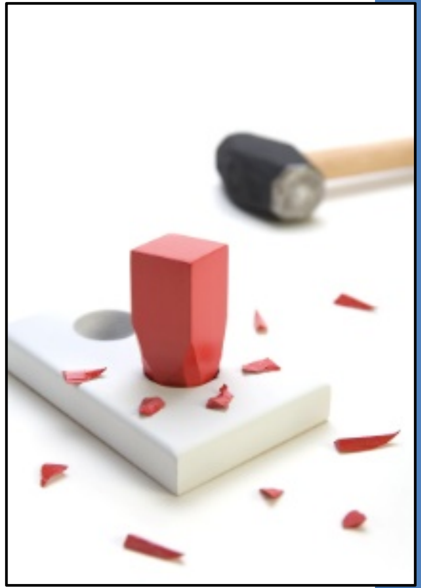
Today’s Parallel Applications





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.

The Challenges

- We need to find a usable way to <u>fully</u> 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 APIs and paradigms for uniformly handling parallelism

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,

HPX – The API

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

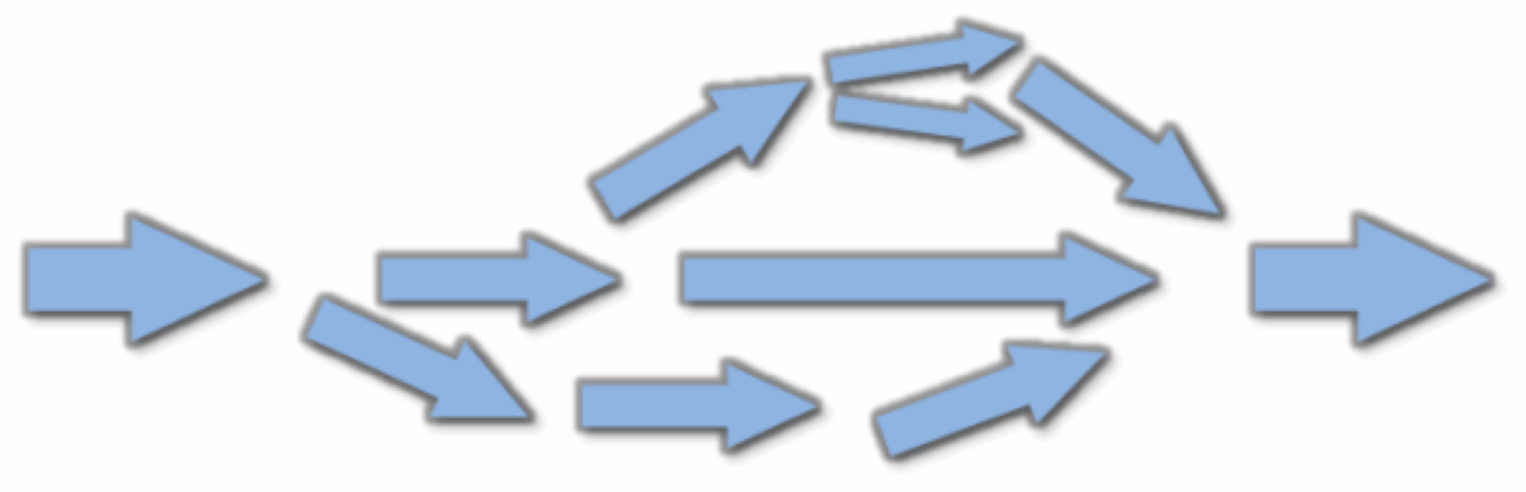
• std::thread, std::jthread	hpx::thread (C++11), hpx::jthread (C++20)
• std::mutex	hpx::mutex
• std::future	hpx::future (including N4538, ‘Concurrency TS’)
• std::async	hpx::async (including N3632)
• std::for_each(par, …), etc.	hpx::for_each (C++17)
• std::latch, std::barrier	hpx::latch, hpx::barrier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

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

The Future of Computation



STE||AR GROUP

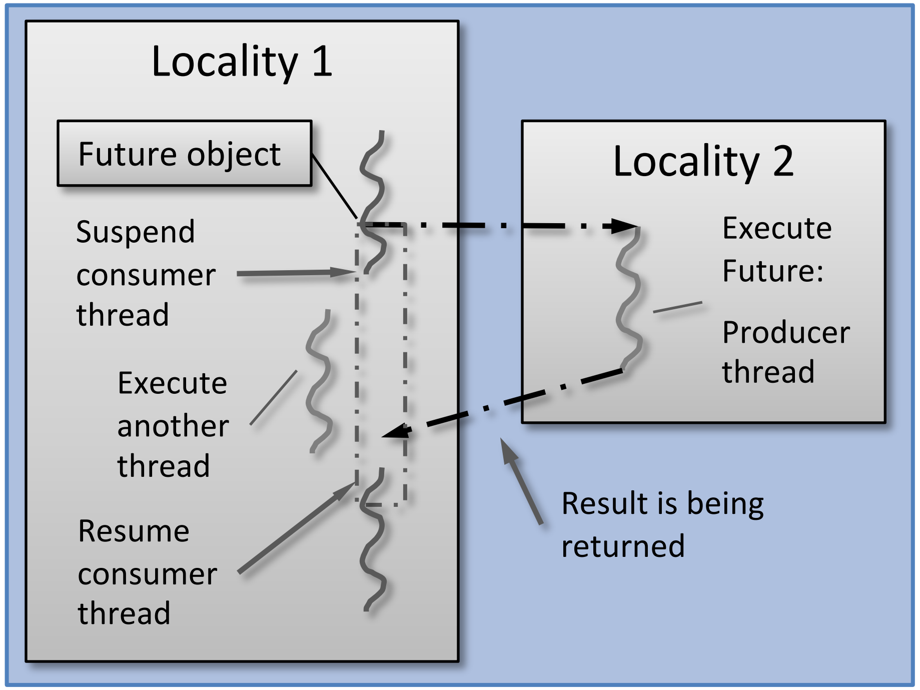
What is a (the) Future?

- 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)

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 = first[size / 2]](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 = first[size / 2]](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

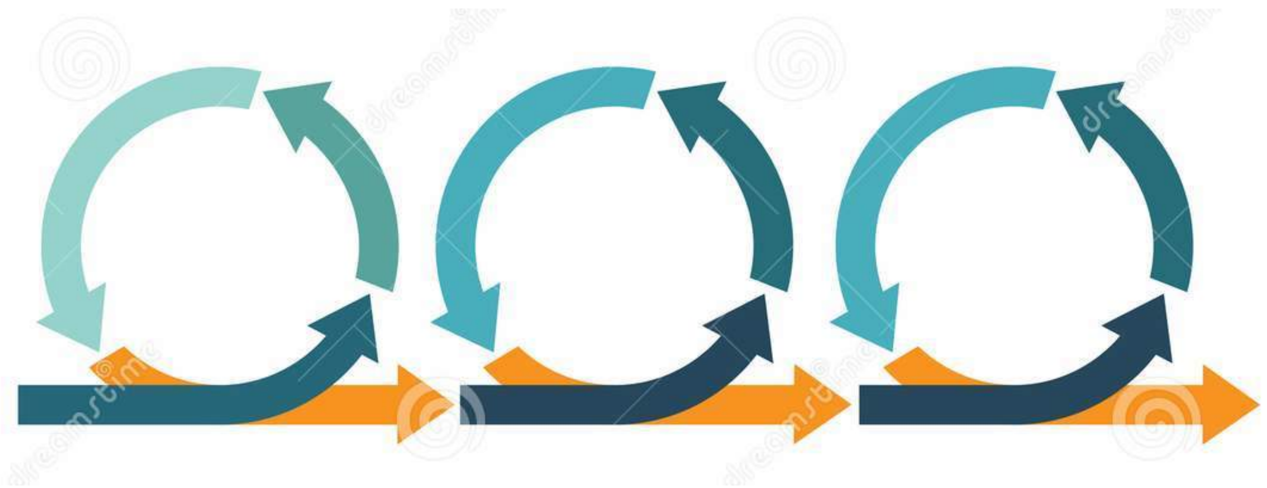
```
template <typename RandomIter>
future<void> quick_sort(RandomIter first, RandomIter last)
{
    ptrdiff_t size = last - first;
    if (size > threshold) {
        future<RandomIter> pivot = partition(par(task), first, last,
            [p = first[size / 2]](auto v) { return v < p; });

        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 (C++20)

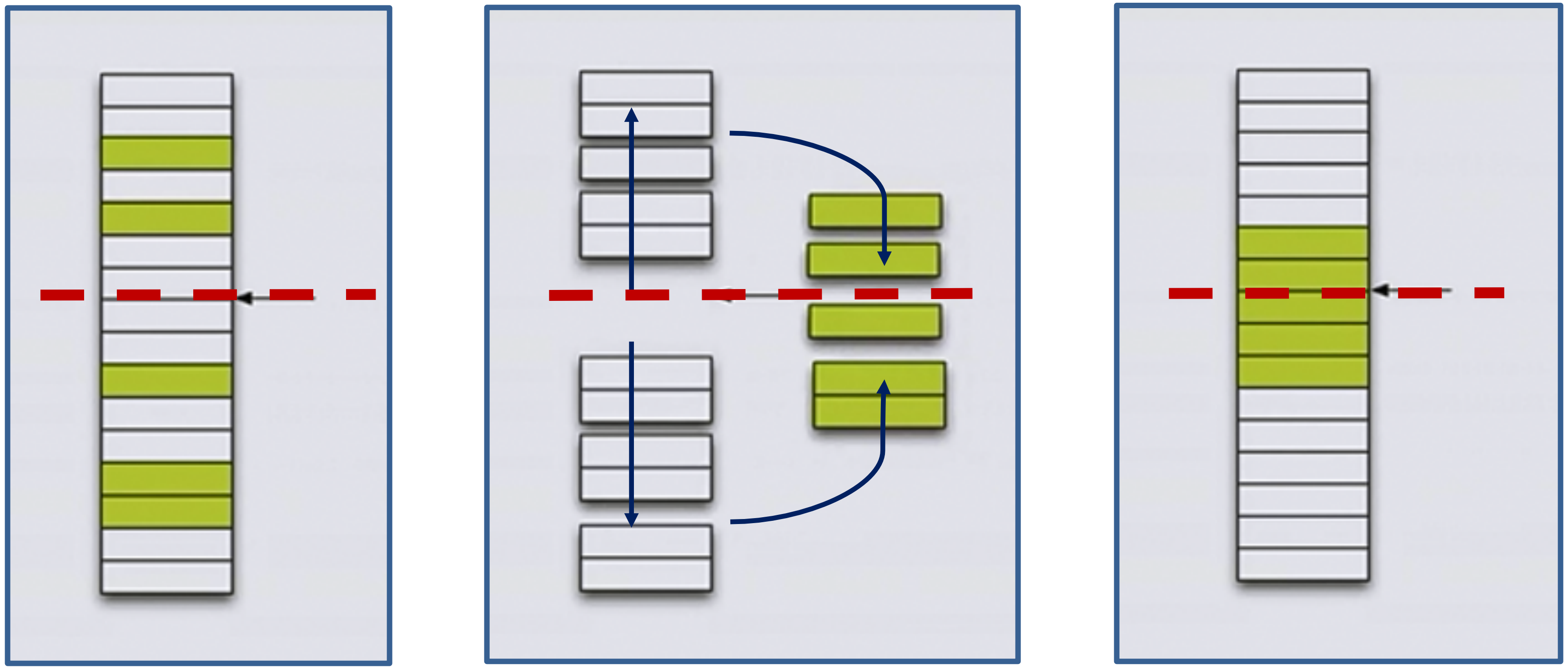
```
template <typename RandomIter>
future<void> quick_sort(RandomIter first, RandomIter last)
{
    ptrdiff_t size = last - first;
    if (size > threshold) {
        RandomIter pivot = co_await partition(par(task), first, last,
            [p = first[size / 2]](auto v) { return v < p; });

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

STE||AR GROUP



Iterative Parallelism

Extending Parallel Algorithms



Sean Parent: C++ Seasoning, Going Native 2013

Extending Parallel Algorithms

- New algorithm: gather

```
template <typename BiIter, typename Pred>
pair<BiIter, BiIter> gather(BiIter f, BiIter l, BiIter p, Pred pred)
{
    BiIter it1 = stable_partition(f, p, not1(pred));
    BiIter it2 = stable_partition(p, l, pred);
    return make_pair(it1, it2);
}
```

Sean Parent: C++ Seasoning, Going Native 2013

Extending Parallel Algorithms

- New algorithm: gather_async

```
template <typename BiIter, typename Pred>
future<pair<BiIter, BiIter>> gather_async(BiIter f, BiIter l, BiIter p, Pred pred)
{
    future<BiIter> f1 = stable_partition(par(task), f, p, not1(pred));
    future<BiIter> f2 = stable_partition(par(task), p, l, pred);
    co_return make_pair(co_await f1, co_await f2);
}
```

STE||AR GROUP

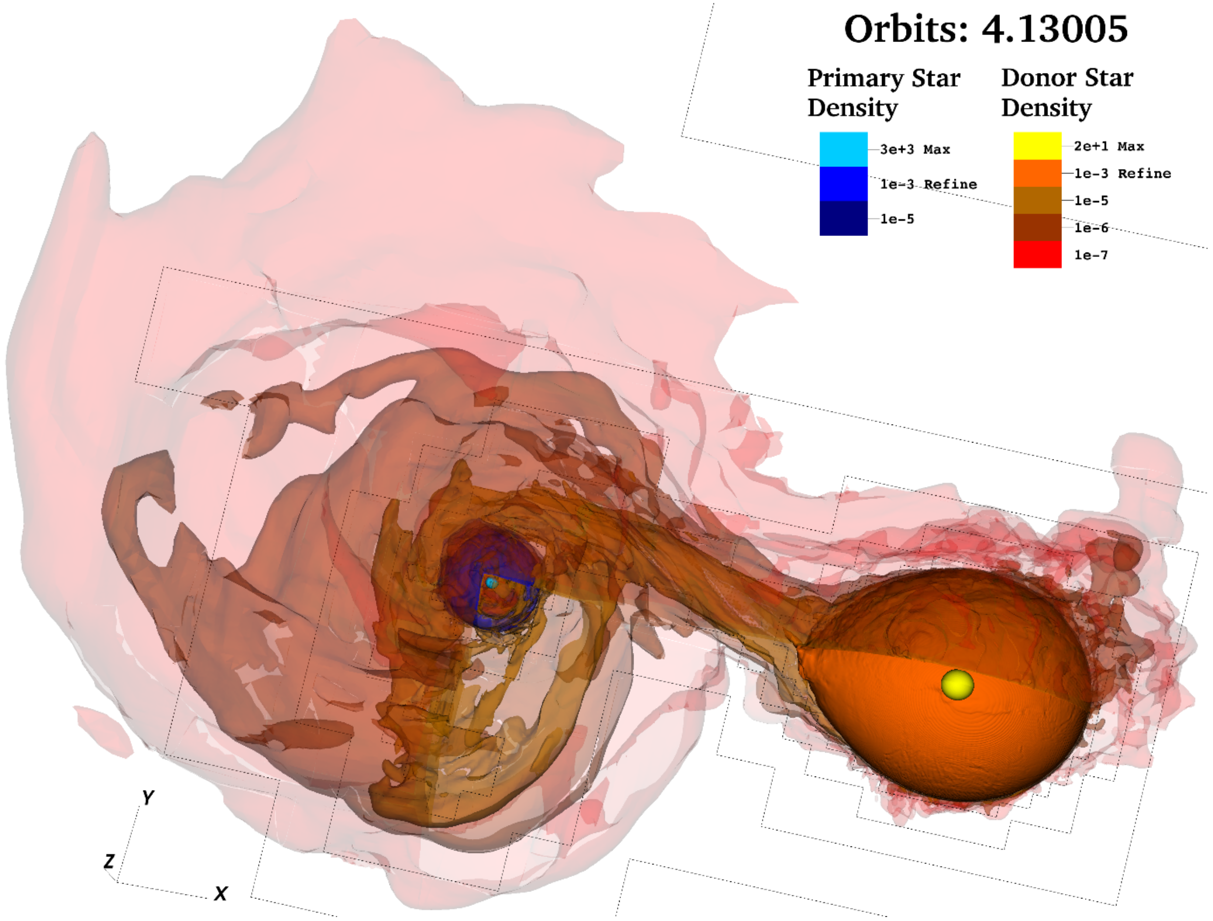
Futurization

- Technique allowing to automatically transform code
  - Delay direct execution in order to reduce 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

Recent Results

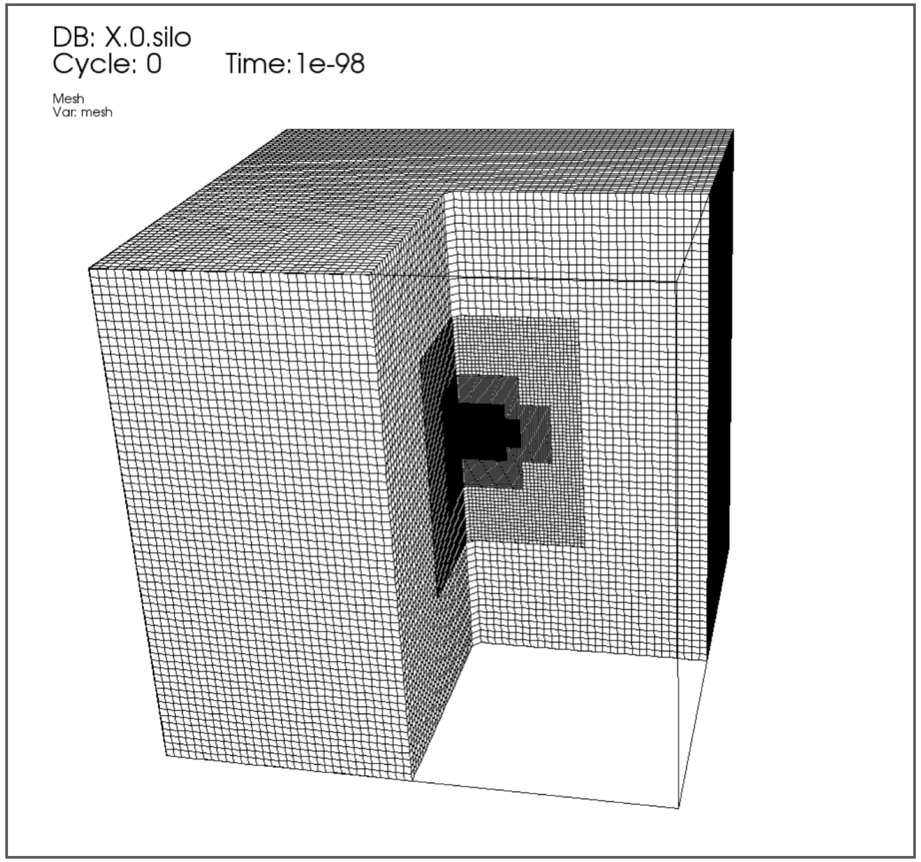
STE||AR GROUP

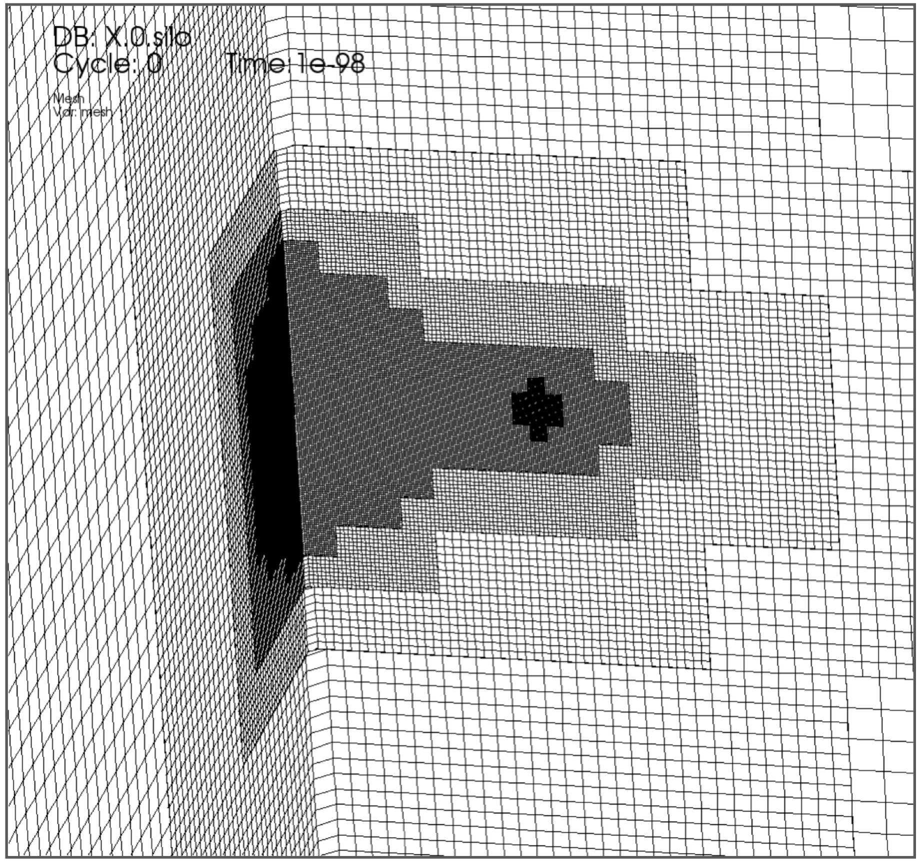
Merging White Dwarfs

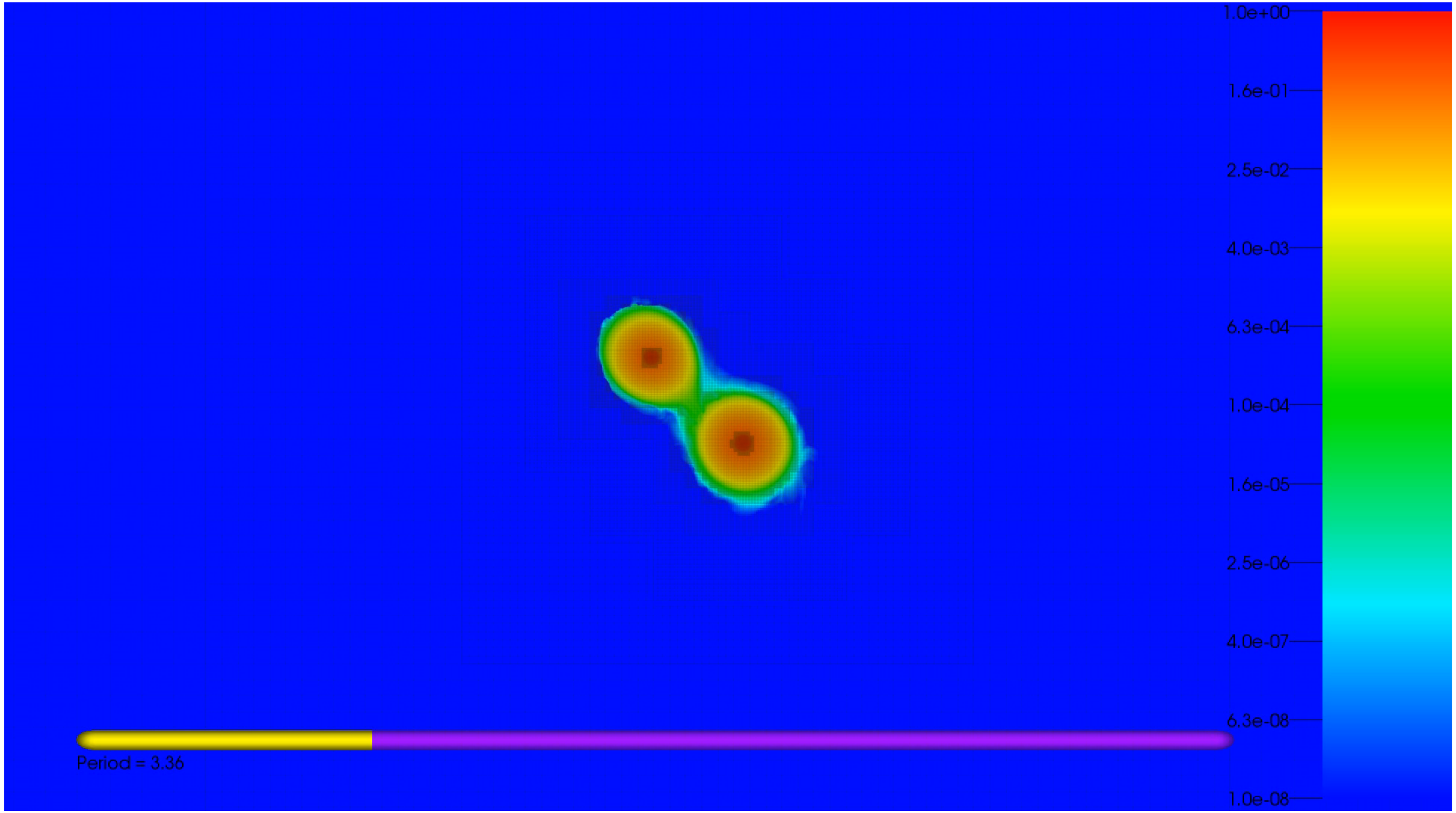


STE||AR GROUP

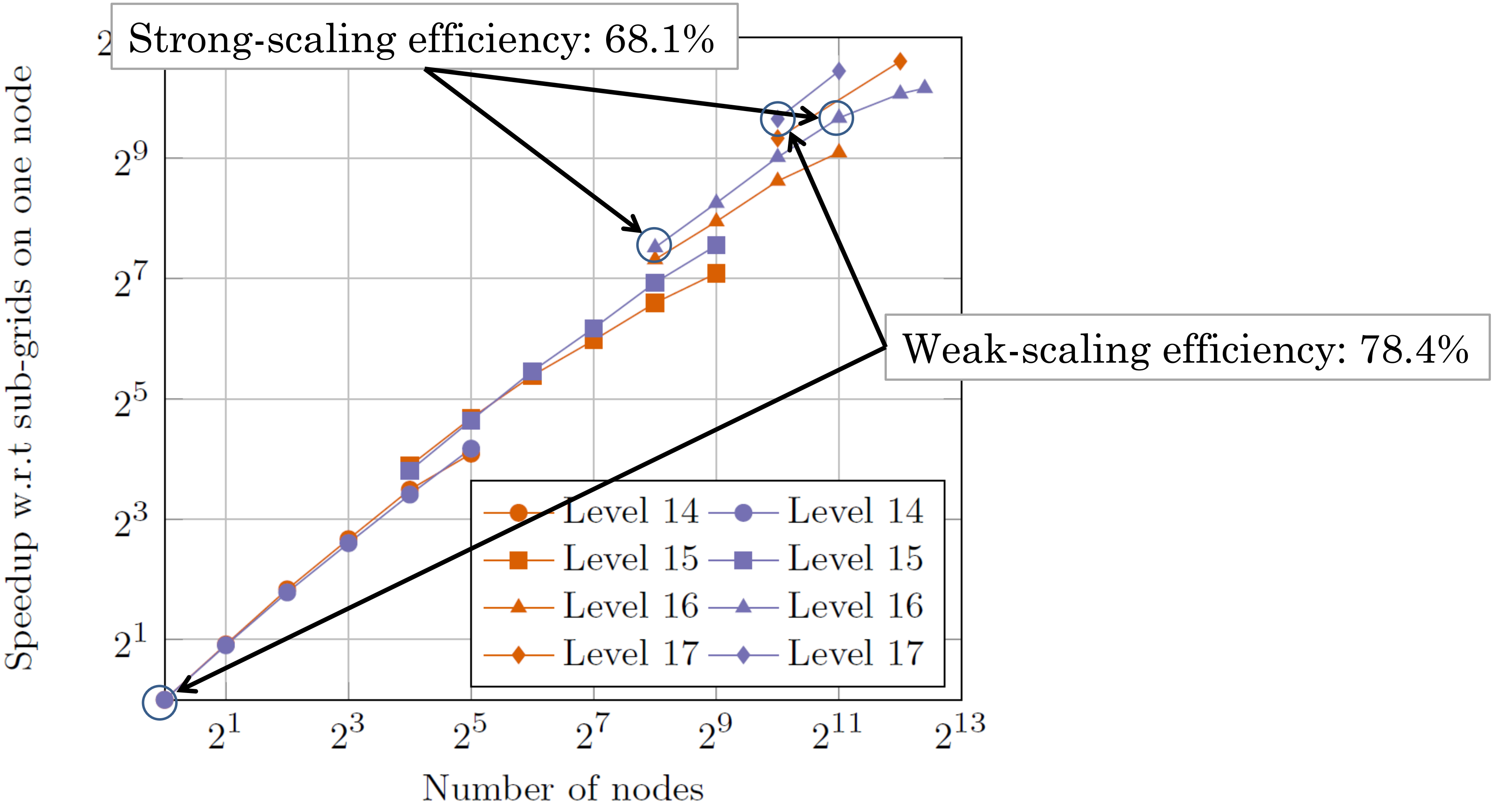
Adaptive Mesh Refinement



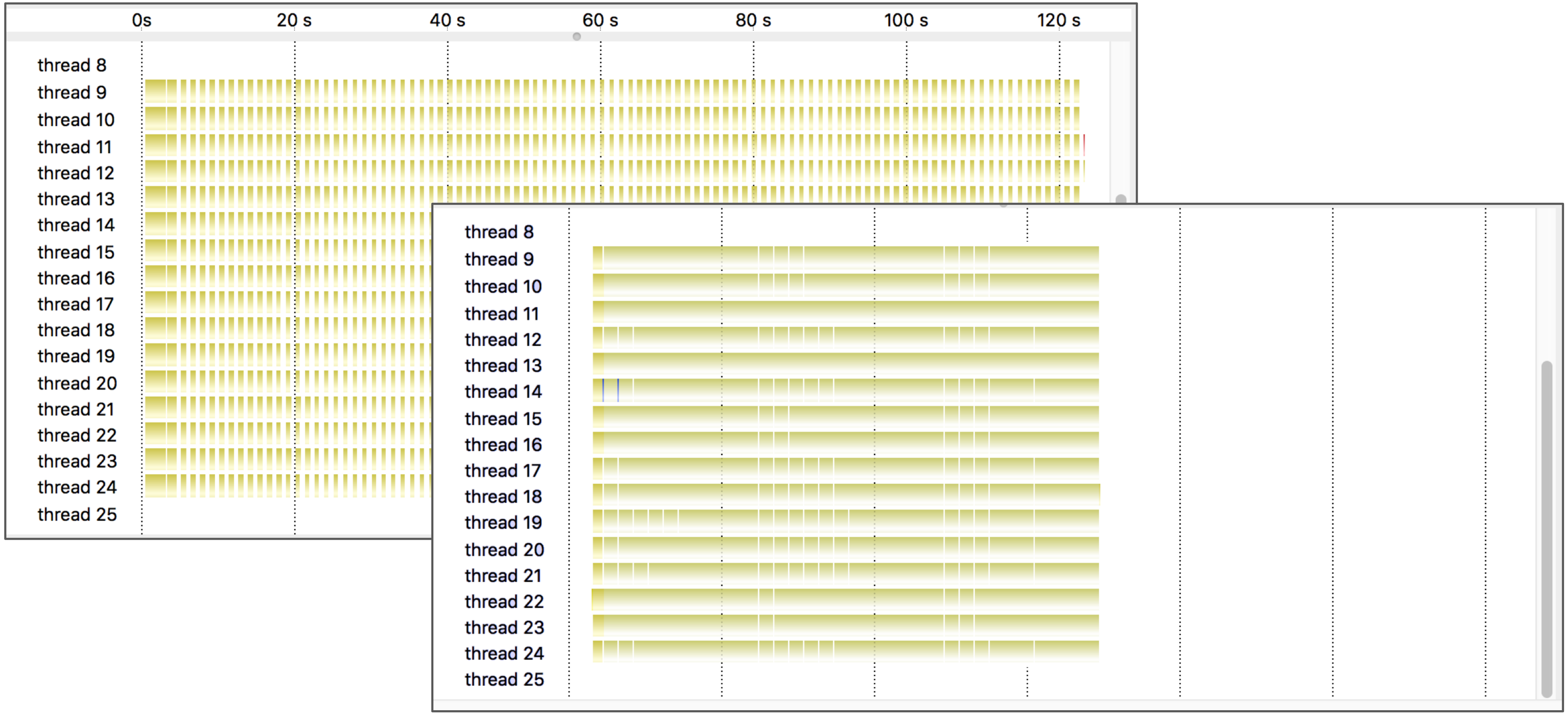



1.0e+00
1.6e-01
2.5e-02
4.0e-03
6.3e-04
1.0e-04
1.6e-05
2.5e-06
4.0e-07
6.3e-08
1.0e-08
Period = 3.36

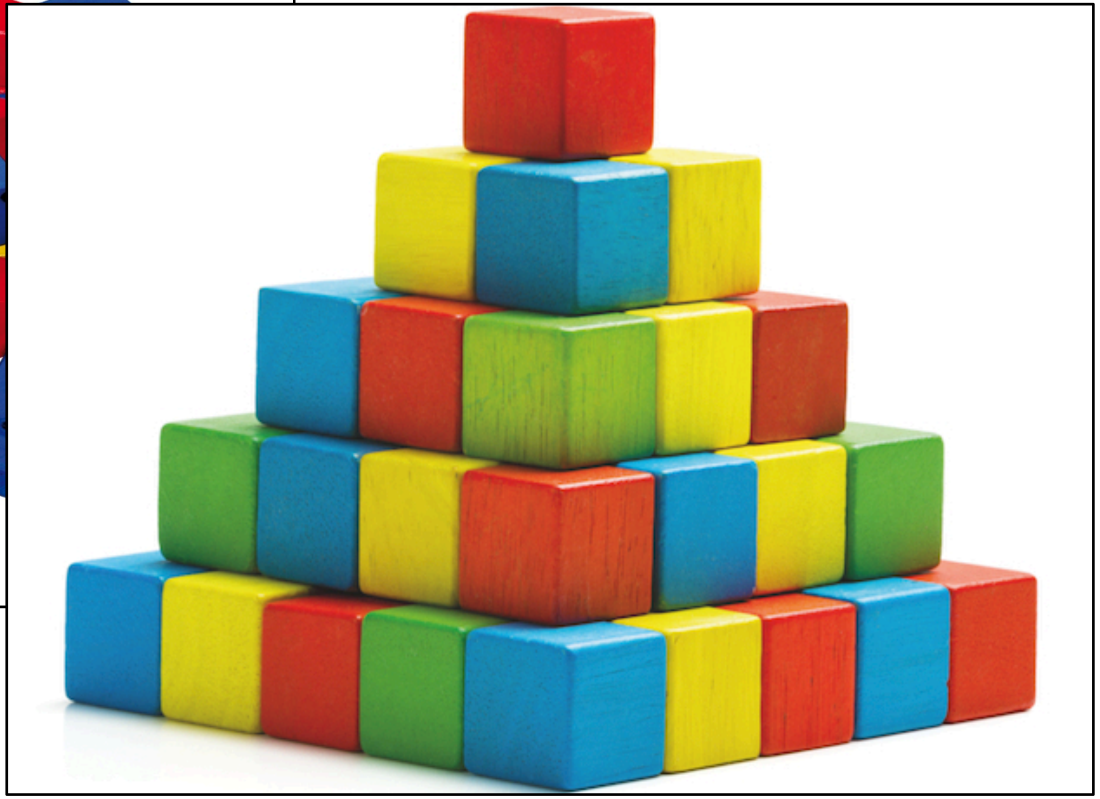
Adaptive Mesh Refinement

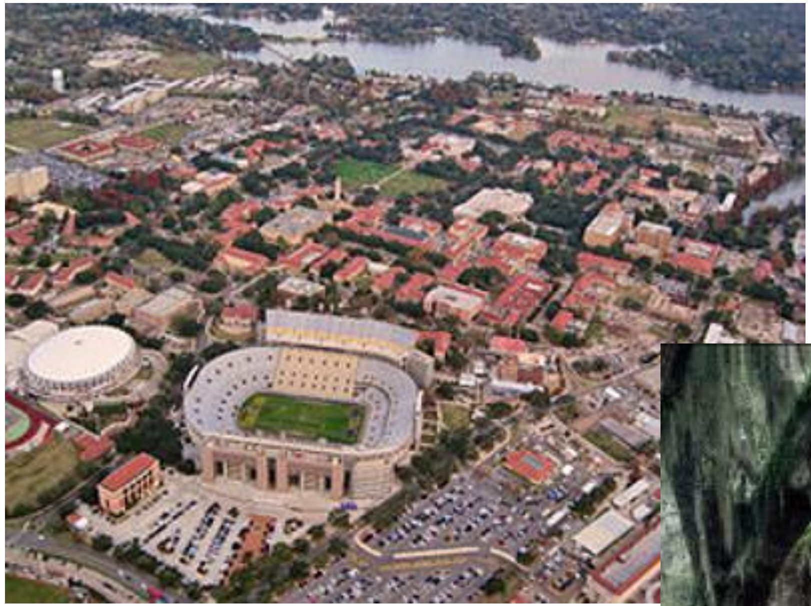


The Solution to the Application Problem

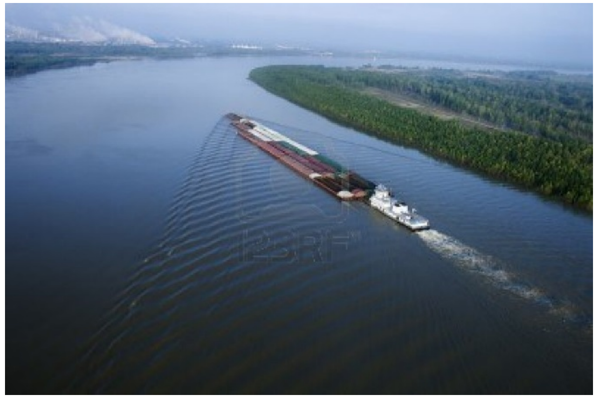


The Solution to the Application Problems





LSU CENTER FOR COMPUTATION & TECHNOLOGY



STE||AR GROUP