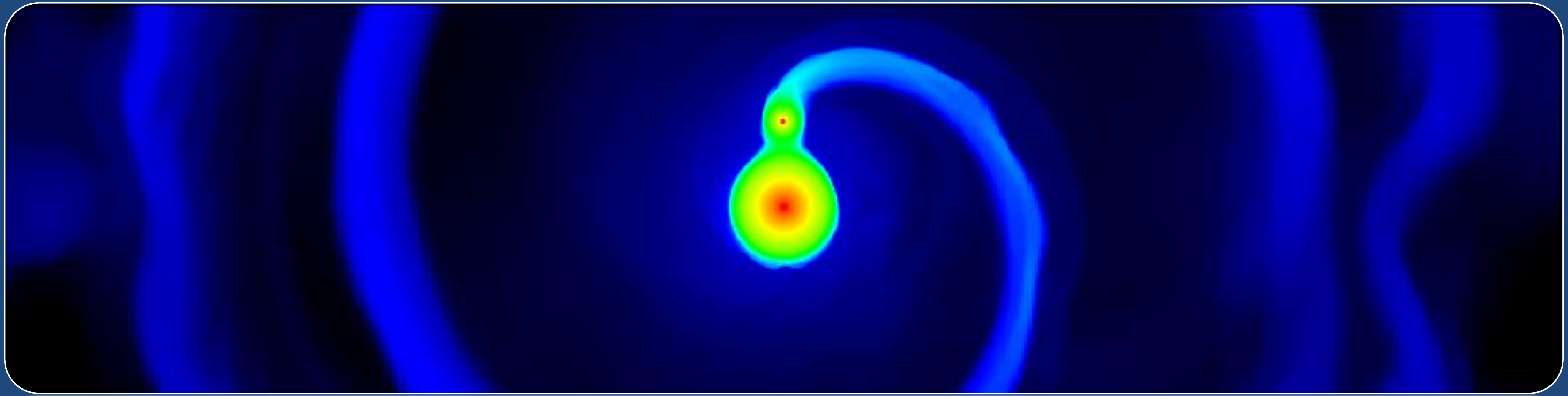
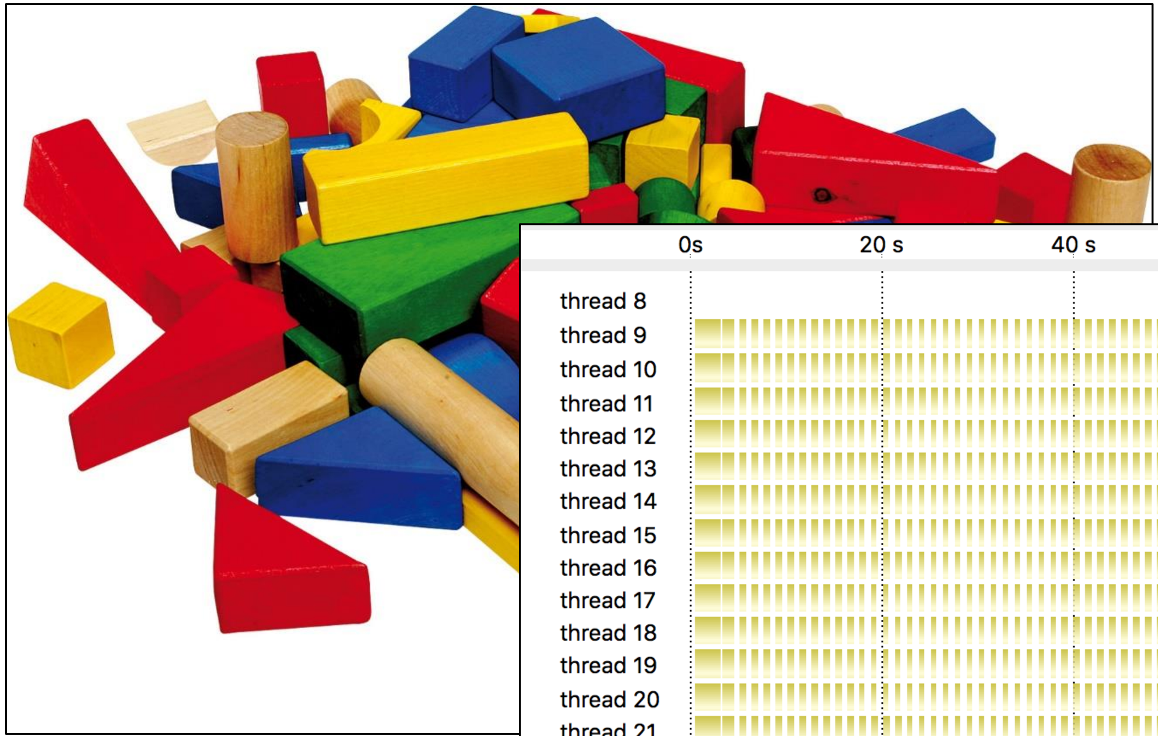


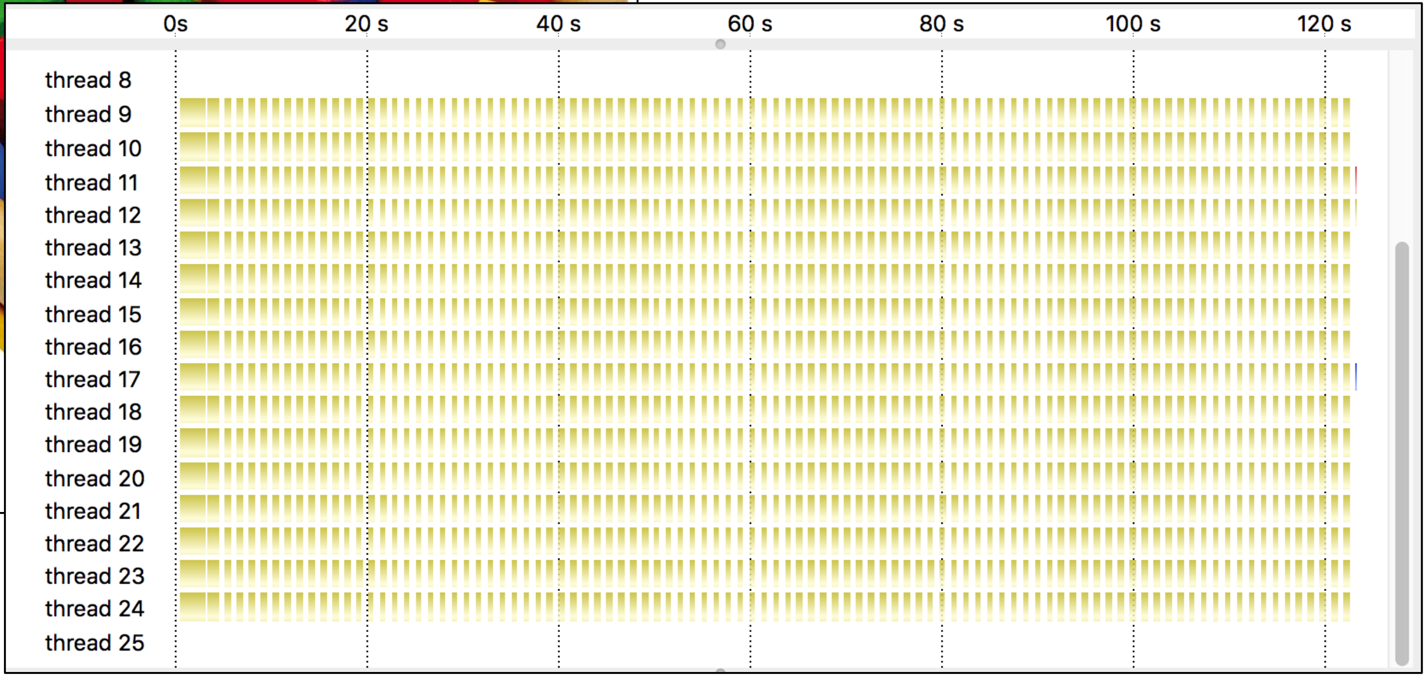
Asynchronous Programming in Modern C++

Futurize All The Things!

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

- Design a programming model and programming environment that:
  - Exposes an API that intrinsically
    - Enables overlap of computation and communication
    - Enables fine-grained parallelism
    - Requires minimal synchronization
    - Makes data dependencies explicit
    - Provides manageable paradigms for handling parallelism
  - Integrates well with existing C++ Standard

HPX

The C++ Standards Library for Concurrency and Parallelism

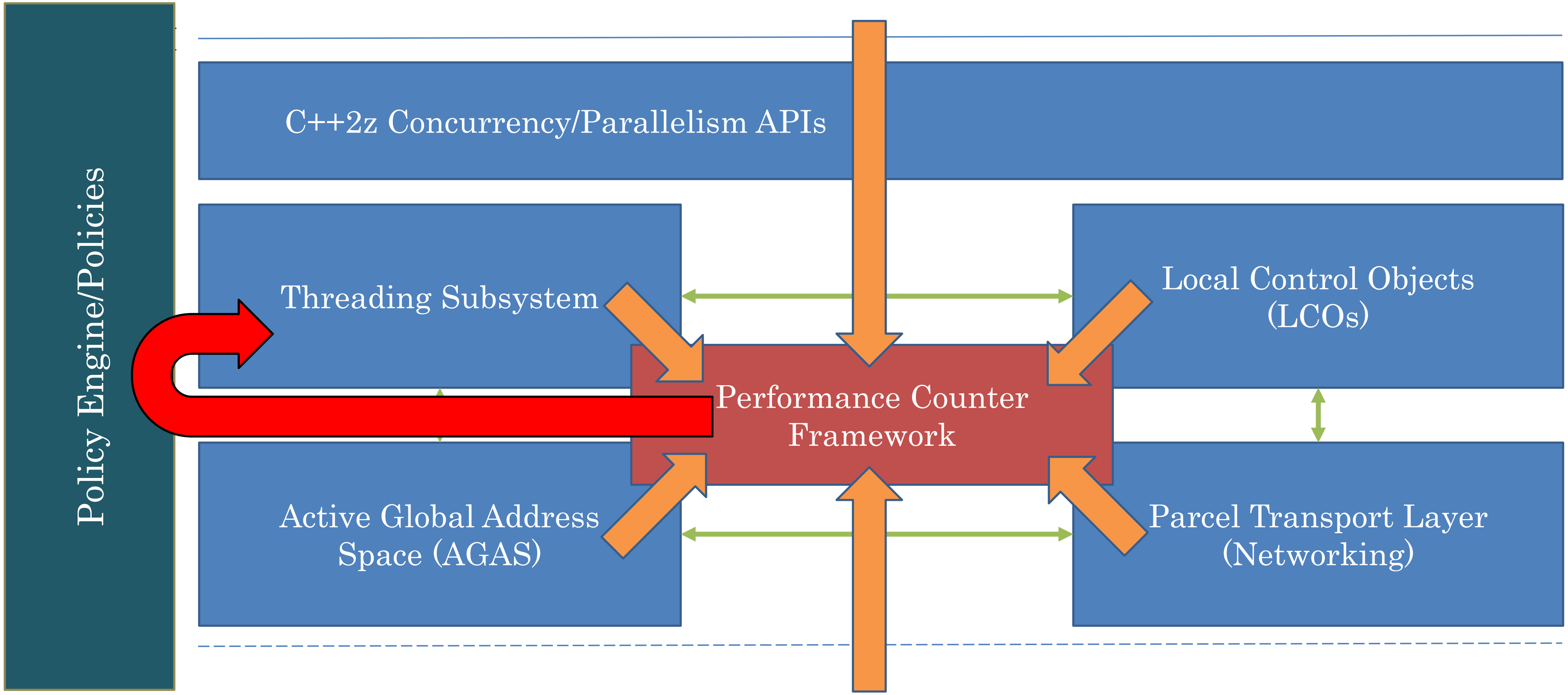
https://github.com/STEllAR-GROUP/hpx

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 – A C++ Standard Library



STE||AR GROUP

HPX – The API

- As close as possible to C++11/14/17/20 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::parallel::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::parallel: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);
```

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);
```

STE||AR GROUP

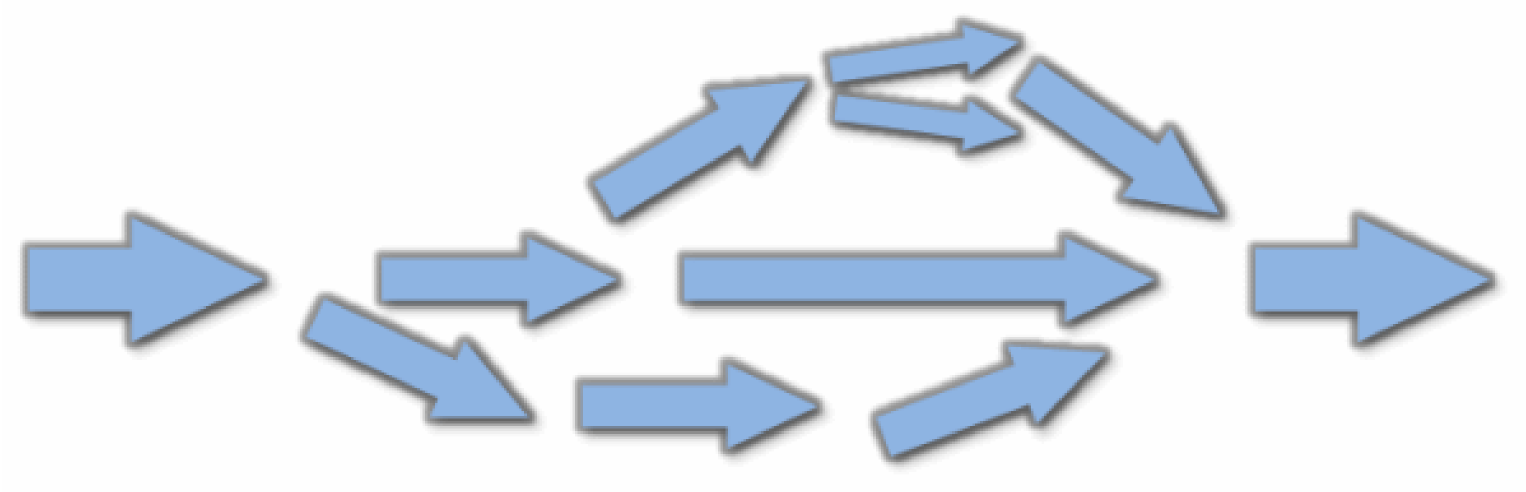
Execution Policies (Extensions)

- 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

The Future of Computation



STE||AR GROUP

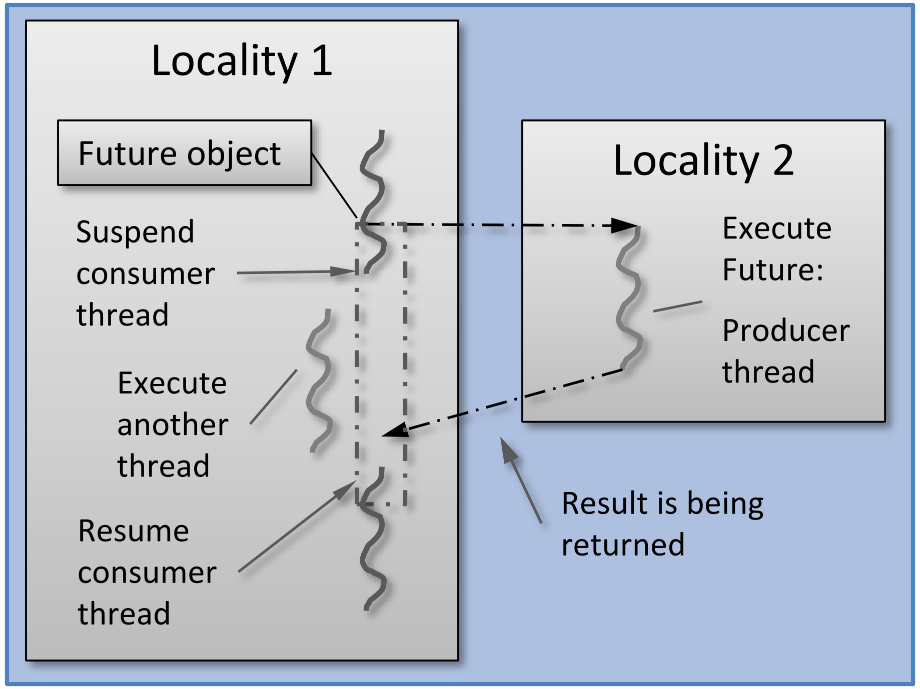
What is a (the) Future?

- 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

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

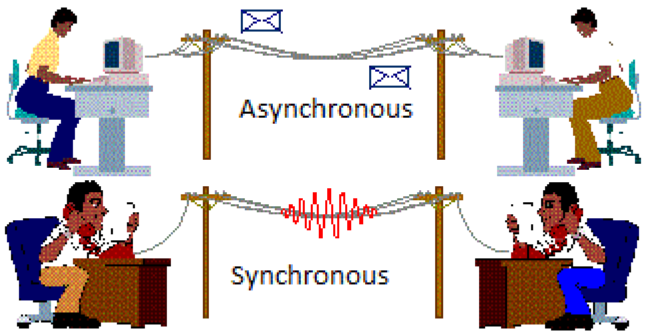
```
template <typename RandomIter>
future<void> quick_sort(RandomIter first, RandomIter last)
{
    ptrdiff_t size = last - first;
    if (size > threshold) {
        future<RandomIter> pivot = partition(par(task), first, last,
            [p = first[size / 2]](auto v) { return v < p; });

        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

```
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

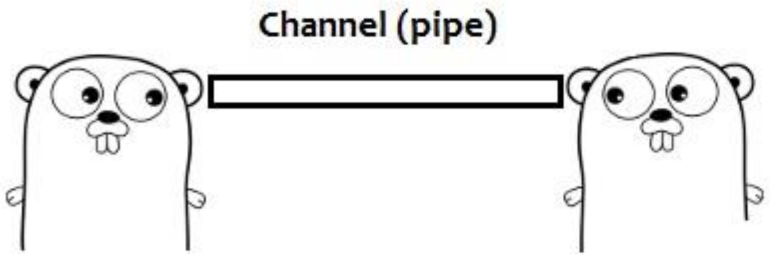


Asynchronous Communication

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

Phylanx

An Asynchronous Distributed Array Processing Toolkit

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

STE||AR GROUP

Phylanx: An Asynchronous Distributed Array Processing Toolkit

- High Performance Computing Challenges
  - Algorithms: need to be made work in distributed, requires data tiling
  - Programming Languages and Models: don't directly support distributed execution
  - Heterogeneous hardware: difficult to deal with various programming models
- Domain experts, specially in the field of machine learning, have traditionally shied away from utilizing HPC resources due to such challenges
- HPC resources are (becoming) the only viable solution with the ever increasing size of datasets.
- Goal: Abstract away complexities of programming on High Performance Computing resources from domain experts.

Phylanx: An Asynchronous Distributed Array Processing Toolkit

- Uses a decorator, @Phylanx, to access the Python AST
  - Reinterpret the AST as C++ data structures
- Integrated job submission, performance measurement and visualization
- Consists of many parts
  - HPX
  - Blaze
  - APEX
  - Traveler
  - Agave/Tapis
  - Jupyter

STE||AR GROUP

Phylanx: An Asynchronous Distributed Array Processing Toolkit

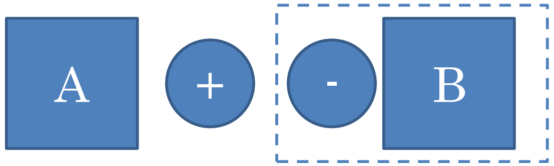
- Combine performance of HPC systems with the ease of programming in a high level language
- Python frontend to abstract away complexities of lower level implementations
  - Integration with Jupyter notebooks
- Run NumPy code directly in Phylanx
- Distributed task graphs are generated from Python
- HPX acts as the execution engine to execute the task graphs
- Promising initial results with execution time comparable to NumPy on shared memory systems.

Phylanx Structure

Frontend

Expression: A + (-B)

Matrices A and B

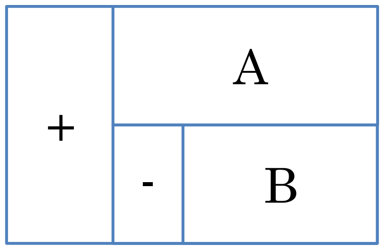


Python:

```
@Phylanx
def work(A, B):
    return A + (-B)
```

Middleware

Internal representation
(Abstract Syntax Tree)

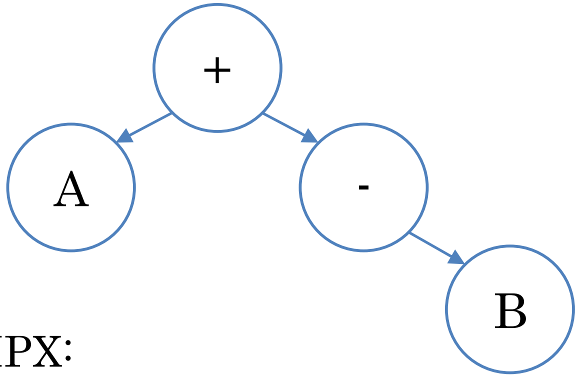


PhySL:

```
define(work, A, B, A + (-B))
```

Backend

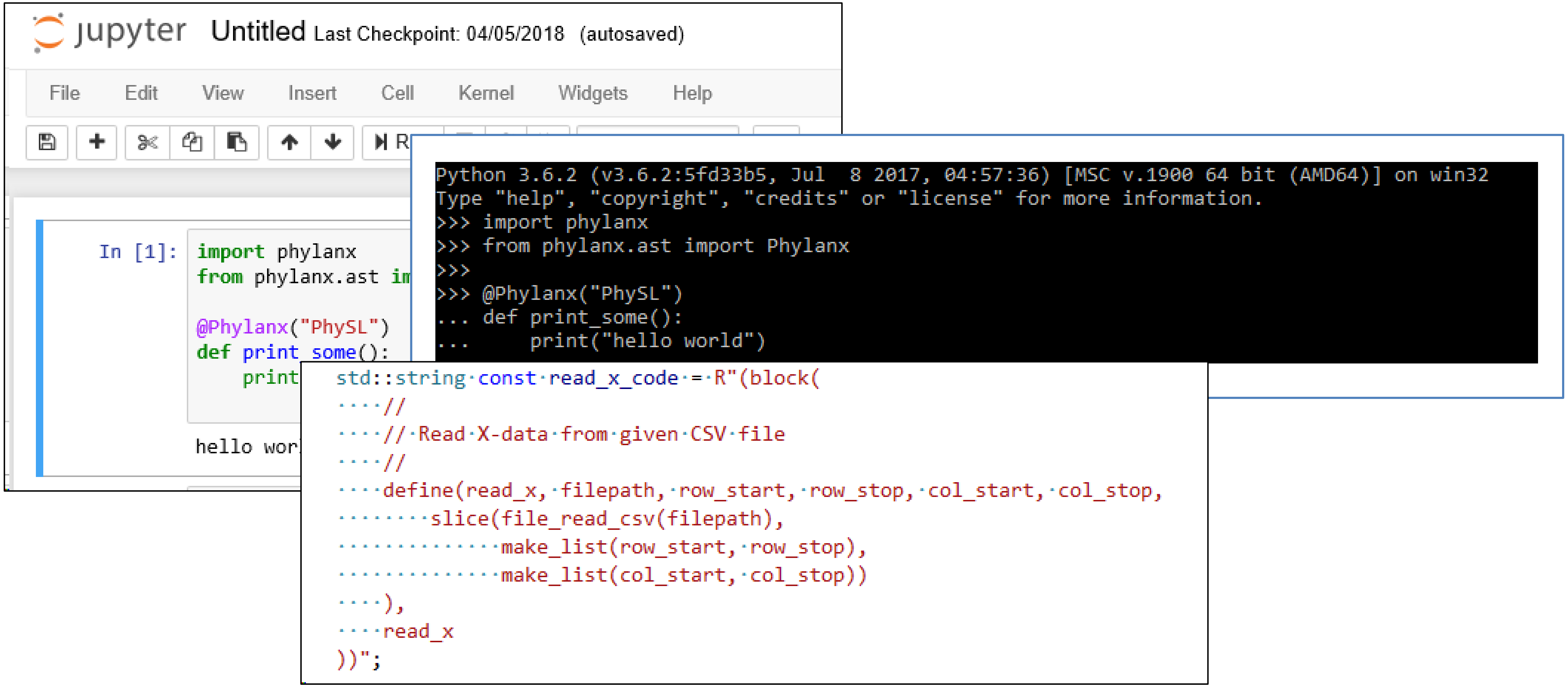
(Distributed) Execution Tree



HPX:

```
hpx::dataflow(...)
```

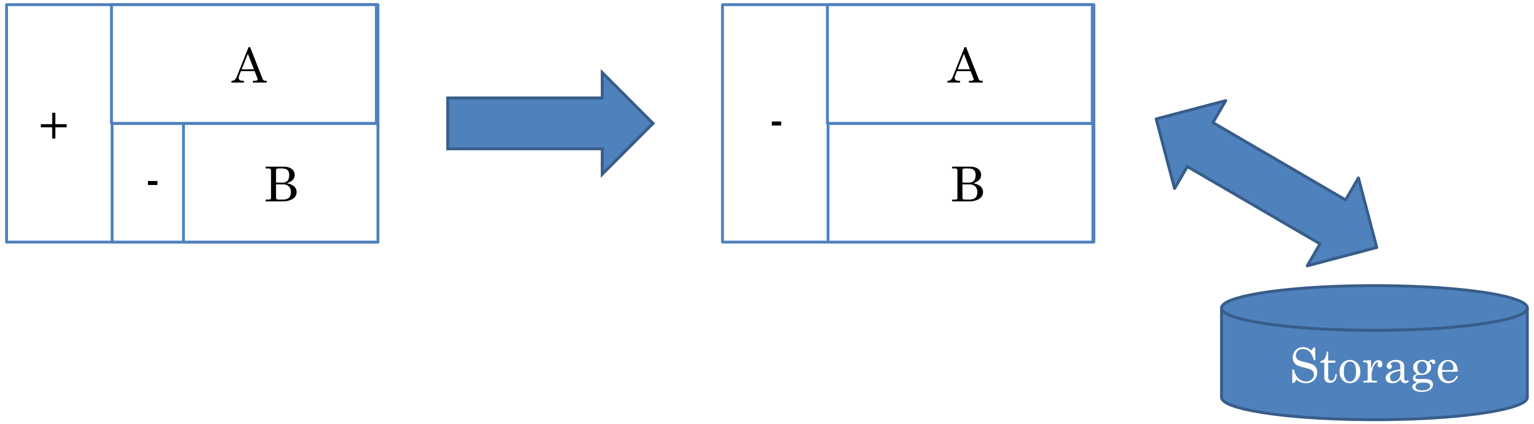
Phylanx: Frontend



STE||AR GROUP

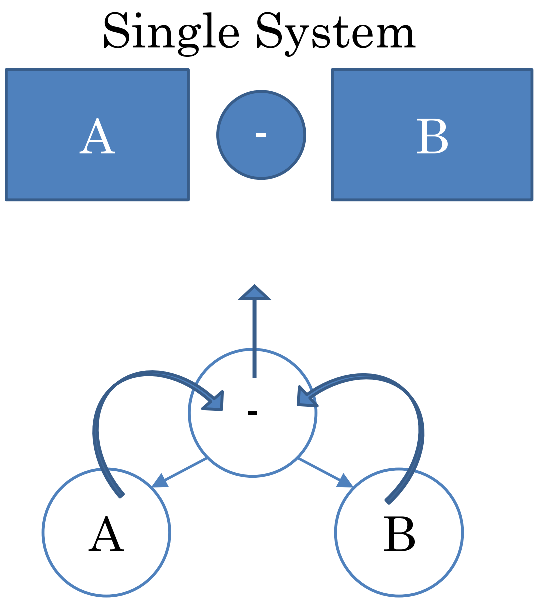
Phylanx: Middleware

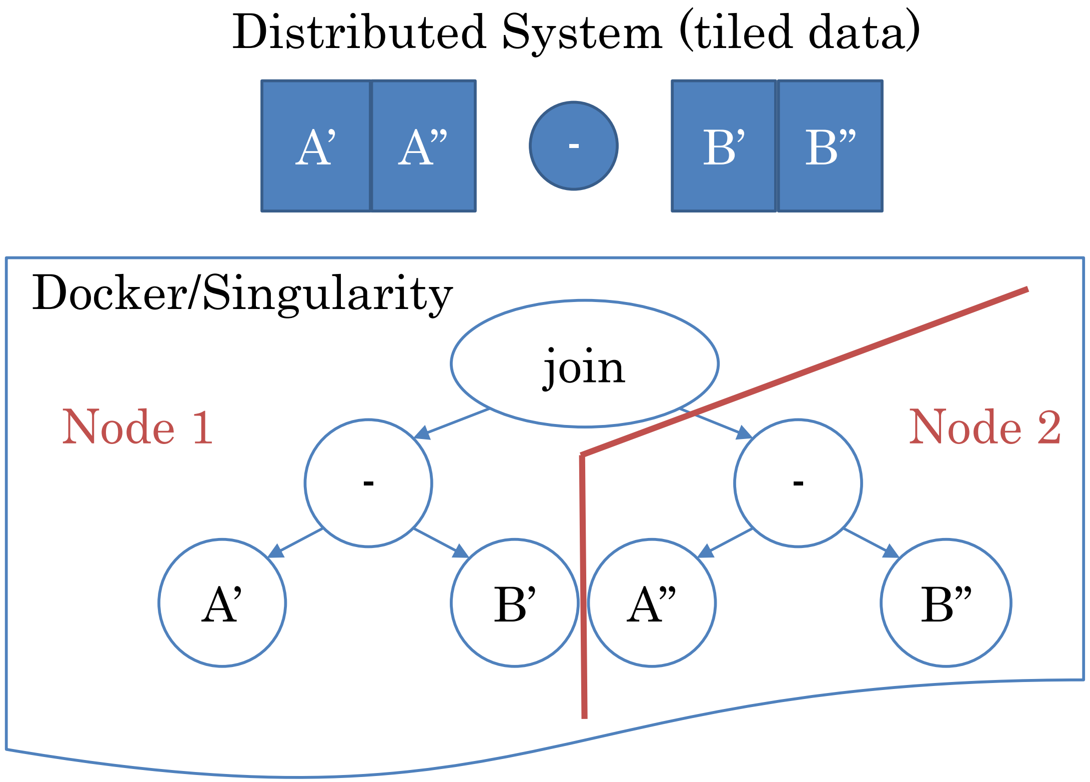
- Various transformations
  - Optimizations
  - Data Tiling and distribution
- Goal: Minimize computation and communication
- Specific for expression to be evaluated



Phylanx: Backend

- Adaptive, asynchronous execution using HPX
  - Maximum speed, pure C++





Run Code Remotely (Jupyter/Agave)

```
In [4]: def fib(n):
            if n < 2:
                return n
            else:
                return fib(n-1)+fib(n-2)

        fibno = randint(10,15)
        print('fib(',fibno,')=...',sep='',flush=True)

        job = remote_run(uv, fib, (fibno,), queue='fork', nodes=1, ppn=1)
        job.wait()
        print("result:",job.get_result())
        viz(job)
```

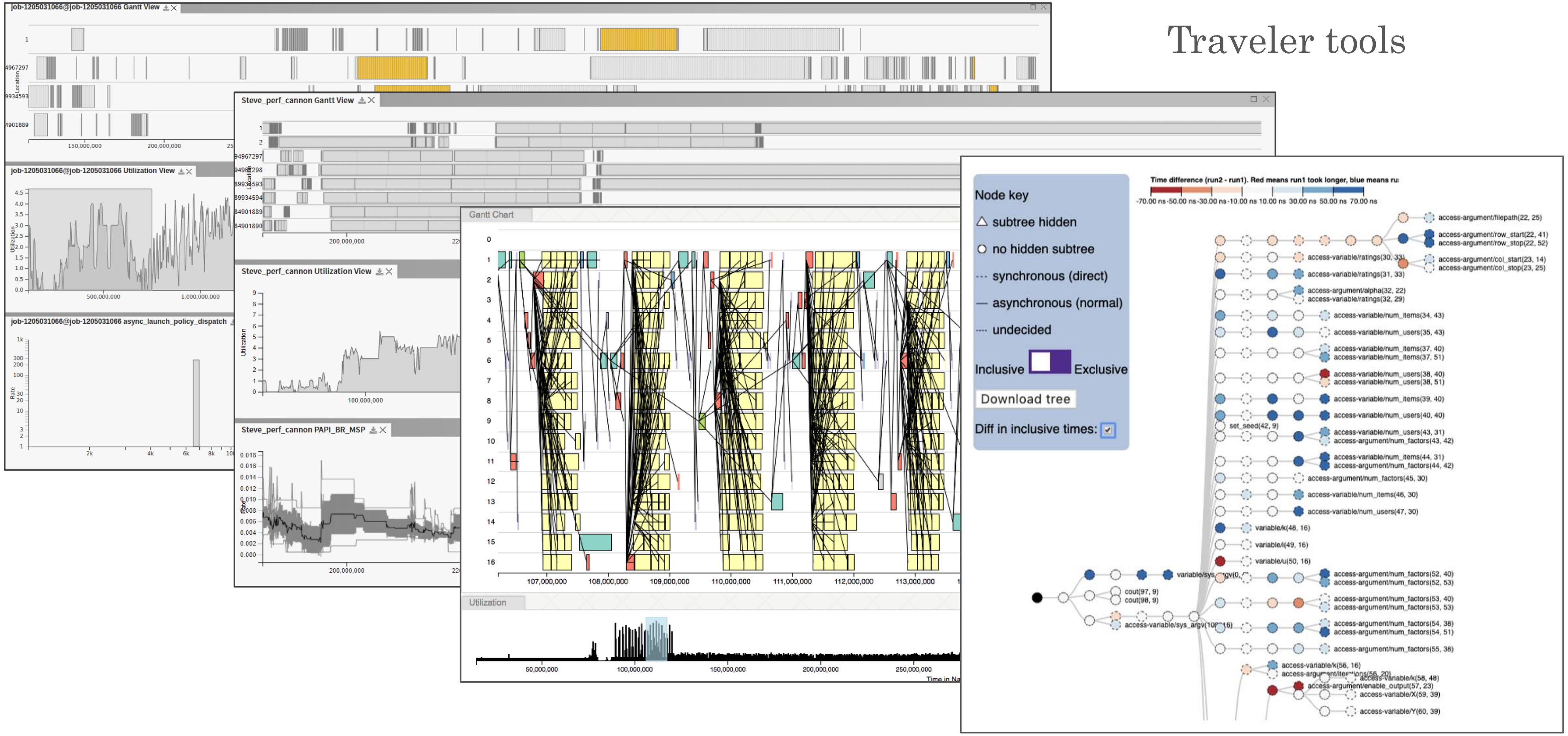
```
fib(11)=...
Job ID: ef66cf7a-c1bd-4044-8d15-749d32e85d49-007
STAGING_INPUTS
SUBMITTING
RUNNING
Cleanup: rostam-sbrandt-storage-tg457049/tjob/tg457049/py-fun-5602761392...done
FINISHED
result: 89
```

fib(11)@ef66cf7a-c1bd-4044-8d15-749d32e85d49-007

- Start with Python
- Call remote_run()
  - Send to remote resource
  - Convert to PhySL
  - Run through queue
  - Collect Performance Data
  - Collect Results
- Click link to visualize performance data

Phylanx: Visualizing Performance

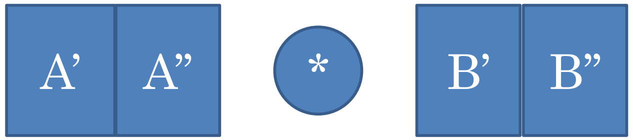
Traveler tools

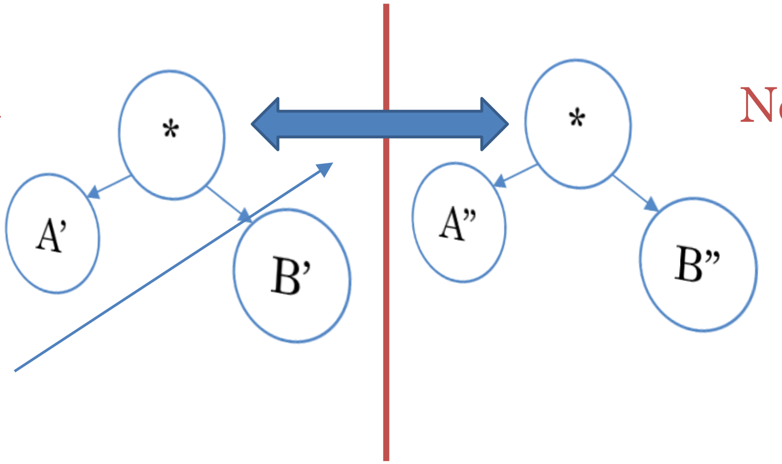


Phylanx: Backend

- Distributed execution model, mostly SPMD
  - Duplicate execution trees
  - Nodes communicate as needed

Dot product





Asynchronous communication

STE||AR GROUP

Phylanx: Futurized Execution

```
// uses hpx::component for distributed operation
struct add : hpx::component<Node>
{
    // futurized implementation
    future<Data> eval(std::vector<Data> params) const override
    {
        // concurrently evaluate child nodes
        future<Data> lhs = children[0].eval(params);
        future<Data> rhs = children[1].eval(params);

        // simplify code with C++20
        co_return co_await lhs + co_await rhs;    // co_await for results
    }

    std::vector<Node> children;
};
```

STE||AR GROUP

Phylanx: CUDA Graph Execution

```
// uses hpx::component for distributed operation
struct cuda_graph : hpx::component<Node>
{
    // futurized implementation
    future<Data> eval(std::vector<Data> params) const override
    {
        // evaluate children, execute CUDA graph when done
        auto args = co_await map(eval, children, params);
        co_return execute_cuda_graph(graph, args);
    }

    cudaGraph_t graph;
    std::vector<Node> children;
};
```

STE||AR GROUP

Asynchrony Everywhere

STE||AR GROUP

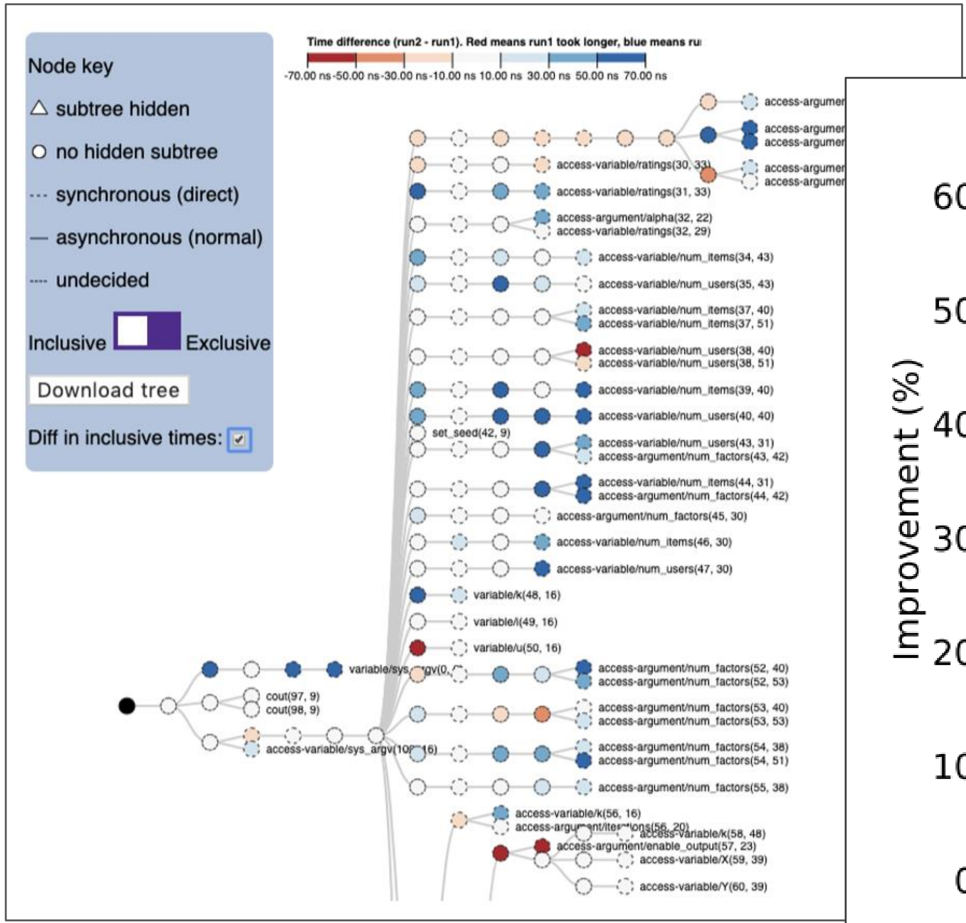
Futurization

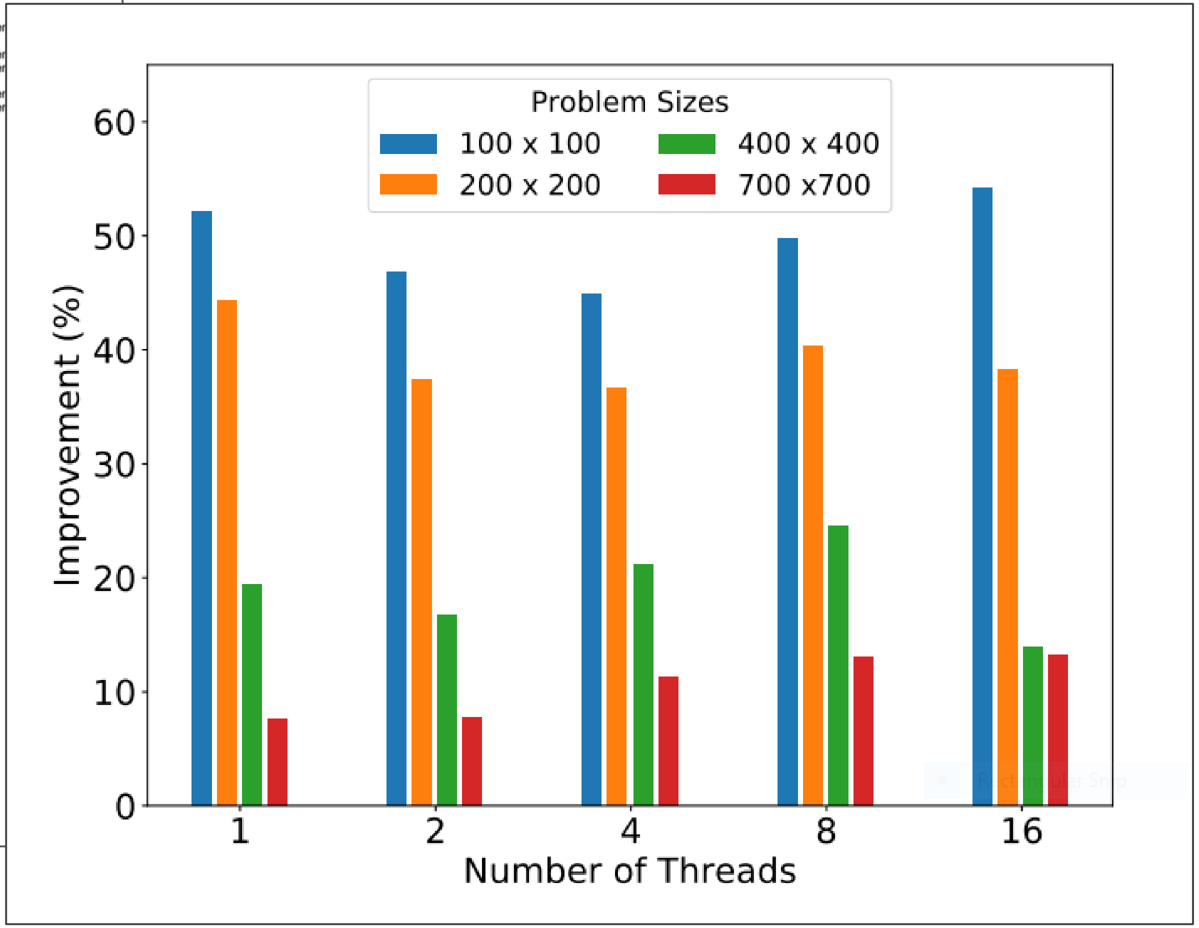
- Technique allowing to automatically transform code
  - Delay direct execution in order to avoid synchronization
  - Turns 'straight' code into 'futurized' code
  - Code no longer calculates results, but generates an execution tree representing the original algorithm
  - If the tree is executed it produces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

Recent Results

STE||AR GROUP

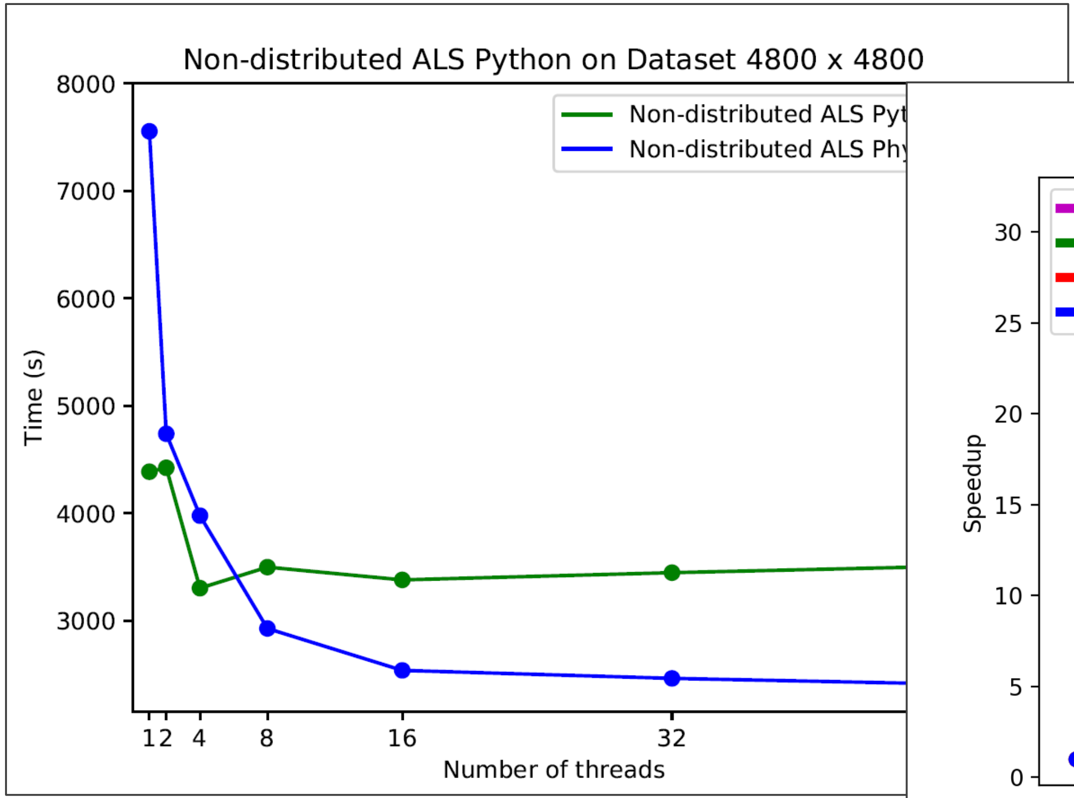
Phylanx: Adaptive Inlining

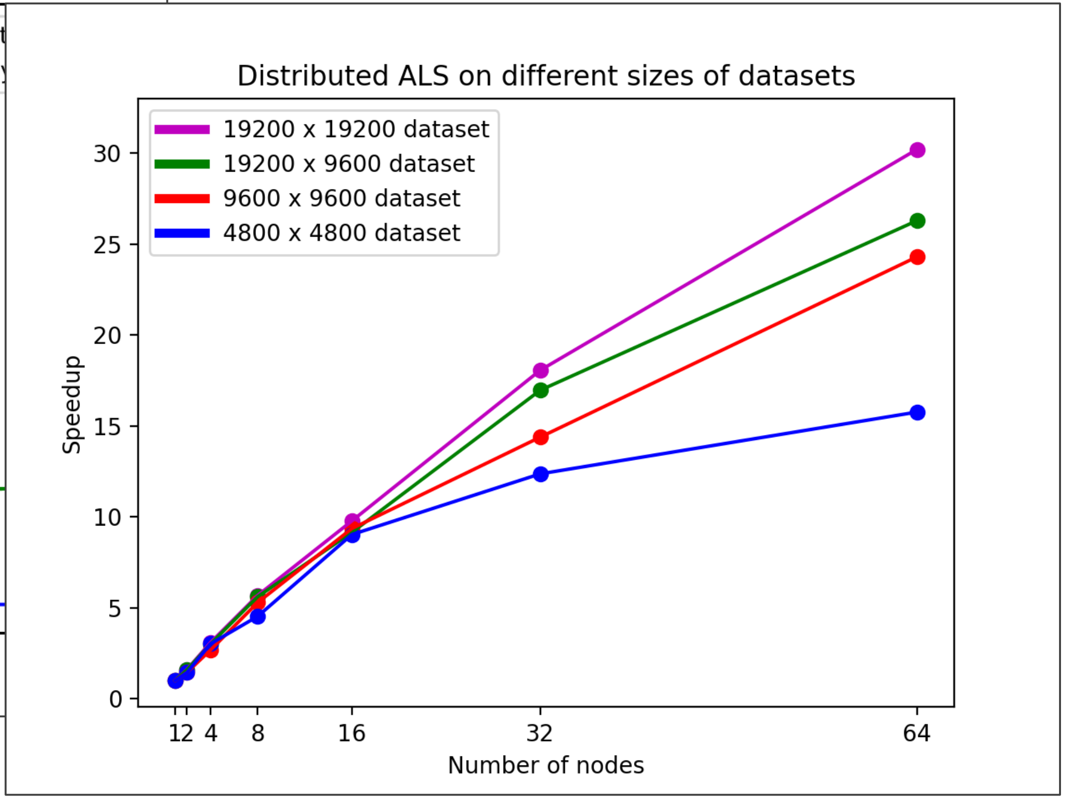




STE||AR GROUP

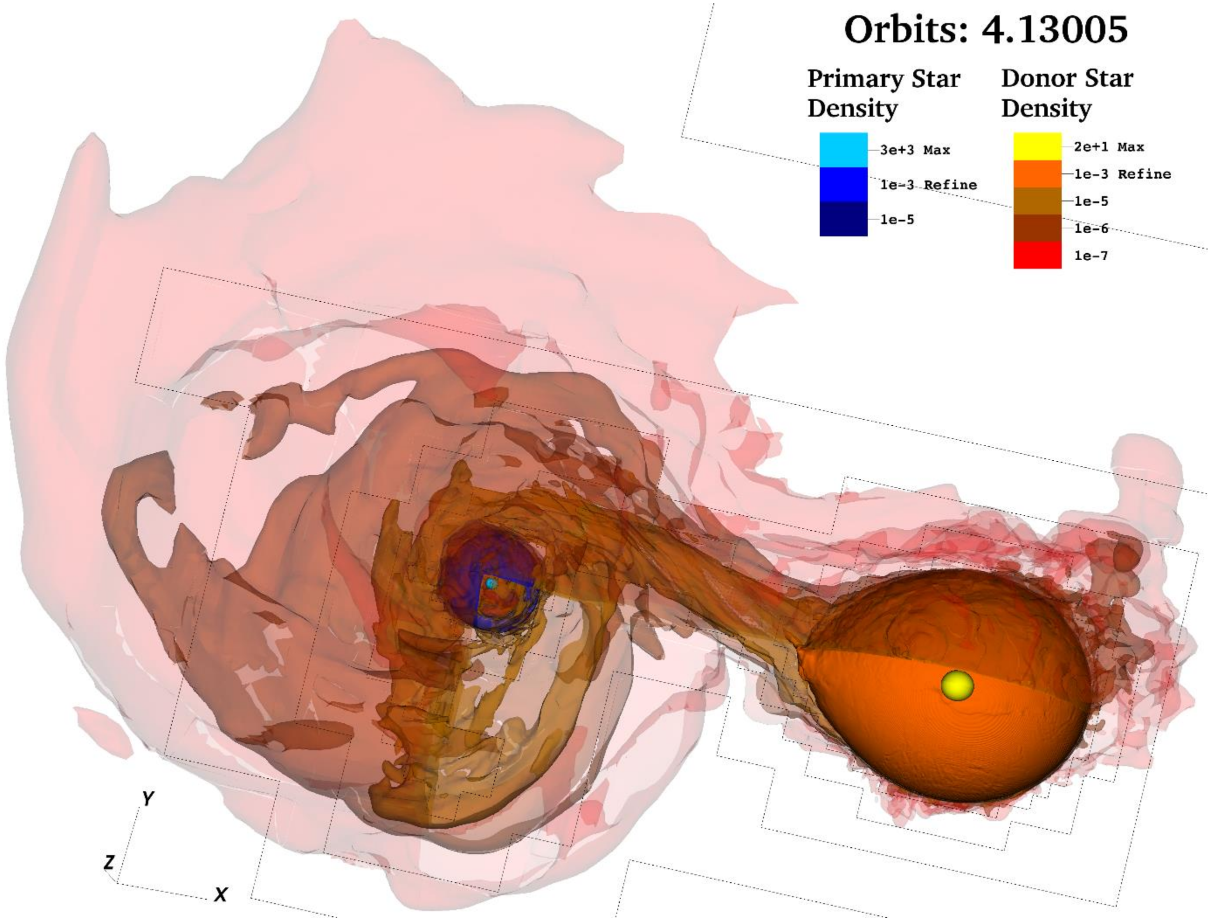
Phylanx: Scaling Results





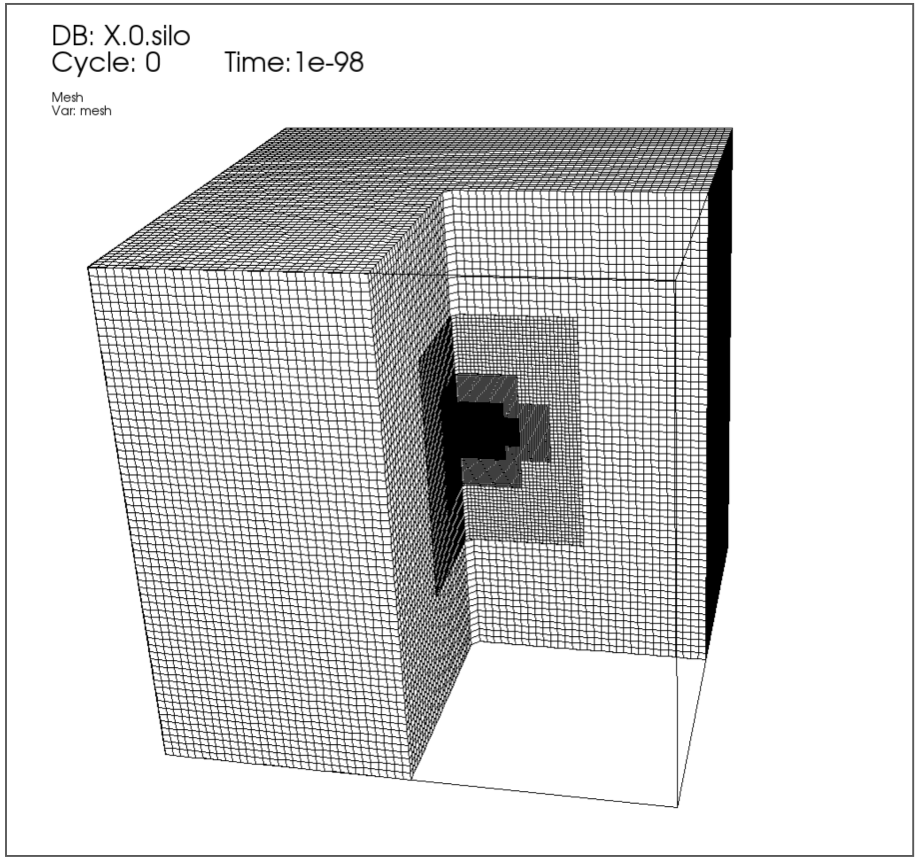
STE||AR GROUP

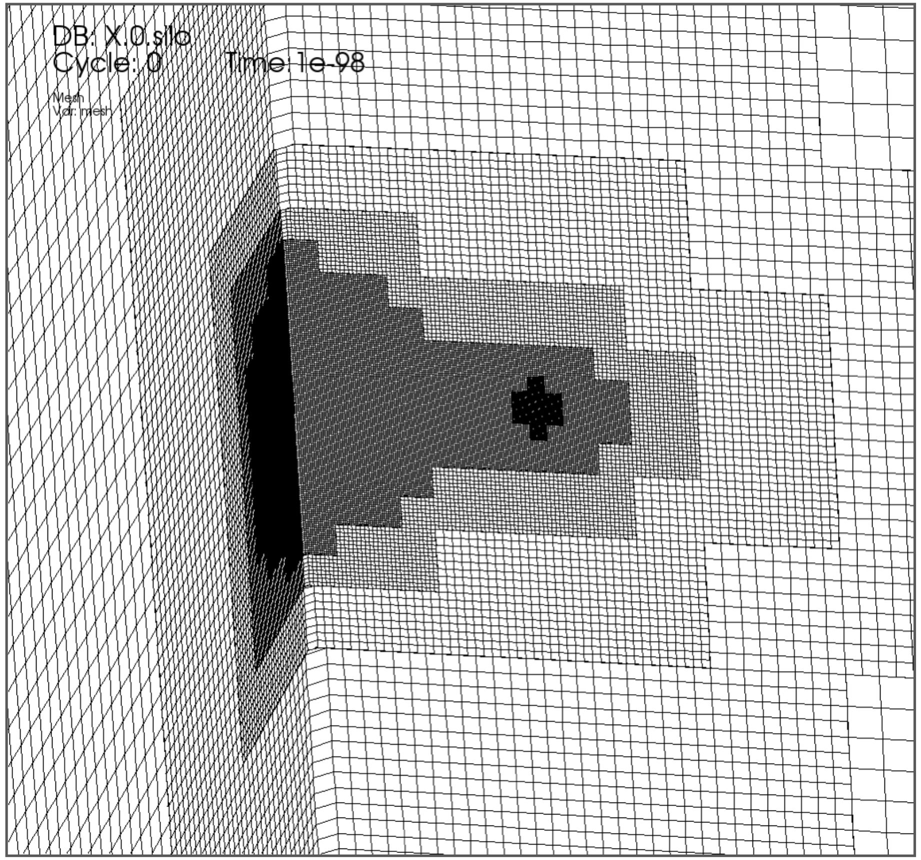
Astrophysics: Merging White Dwarfs



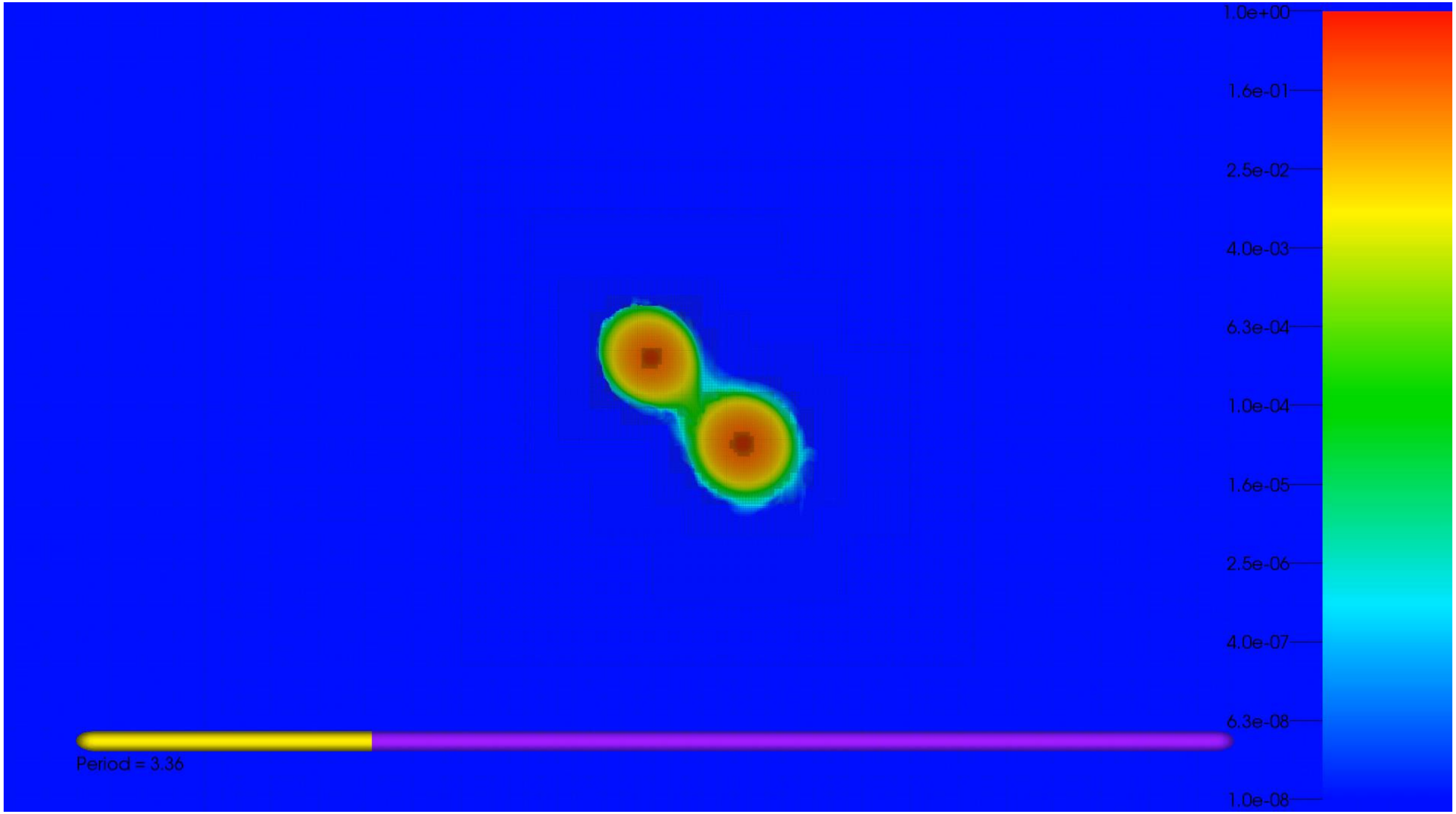
STE||AR GROUP

Adaptive Mesh Refinement

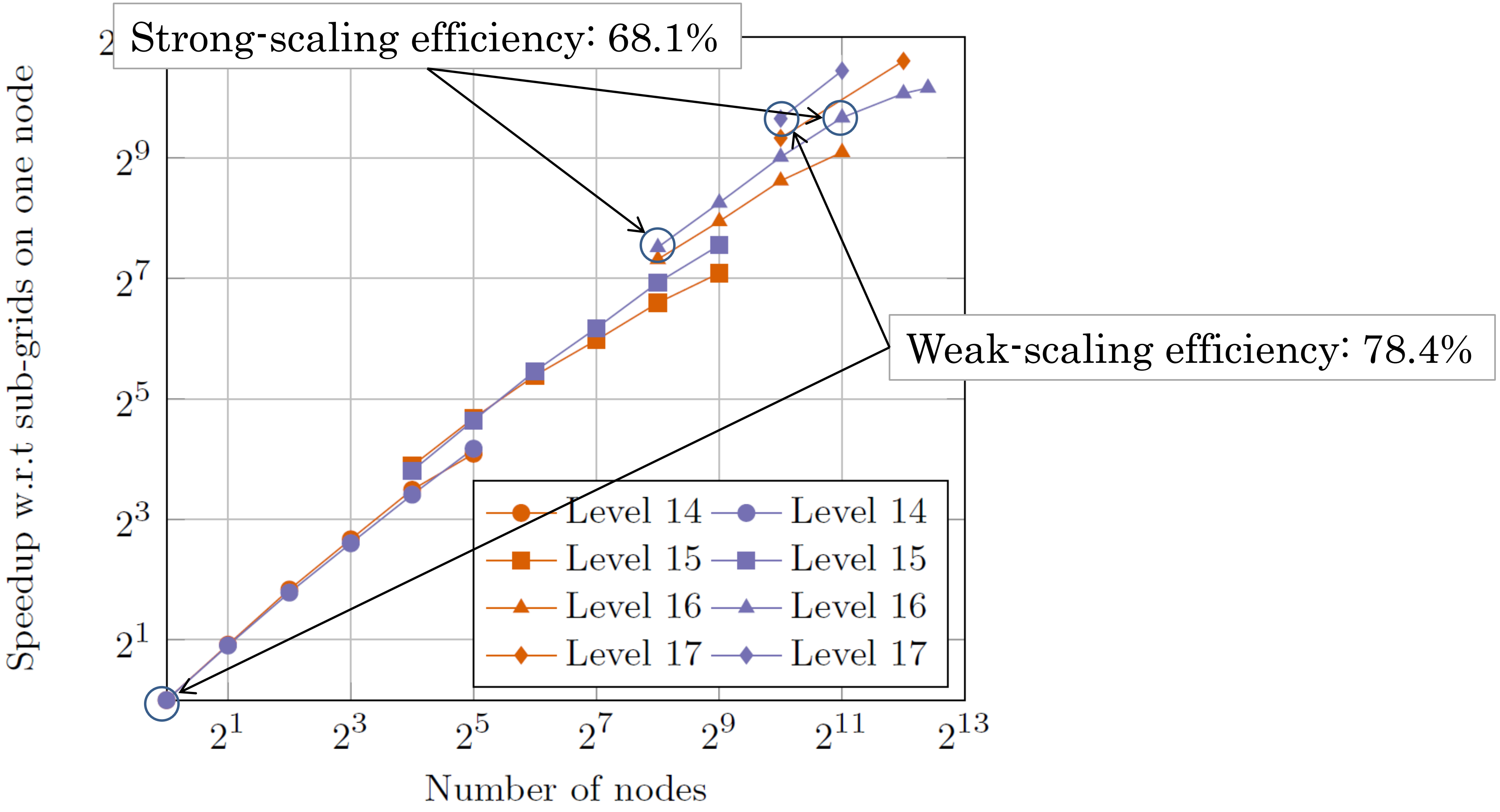




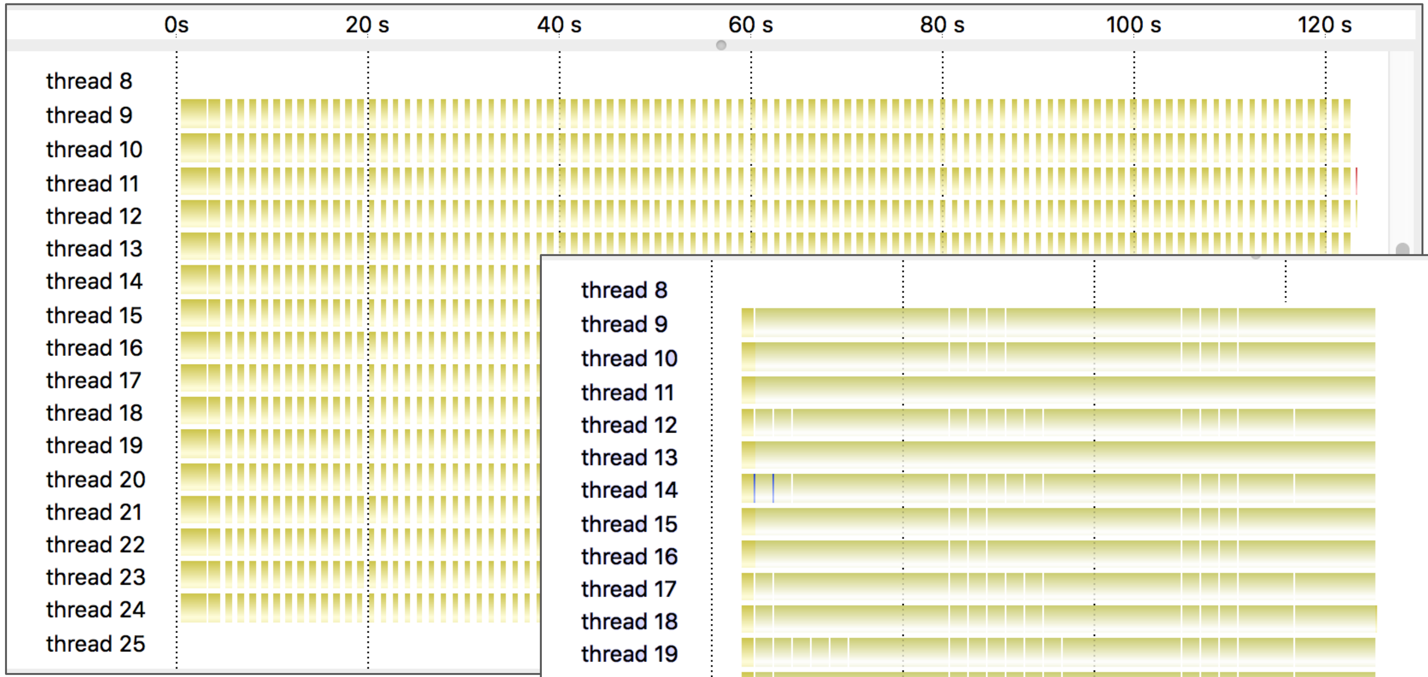
STE||AR GROUP


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

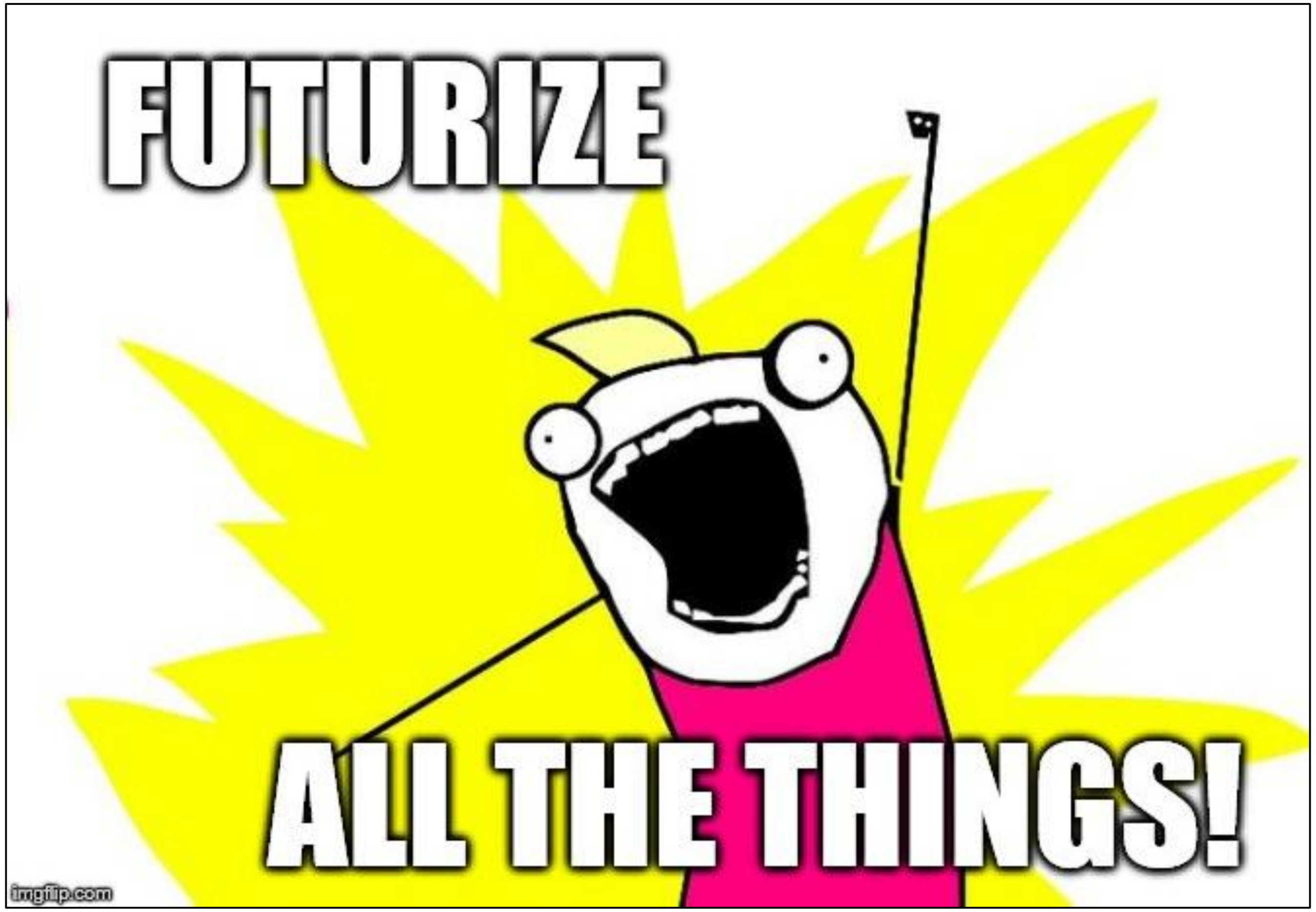
Adaptive Mesh Refinement



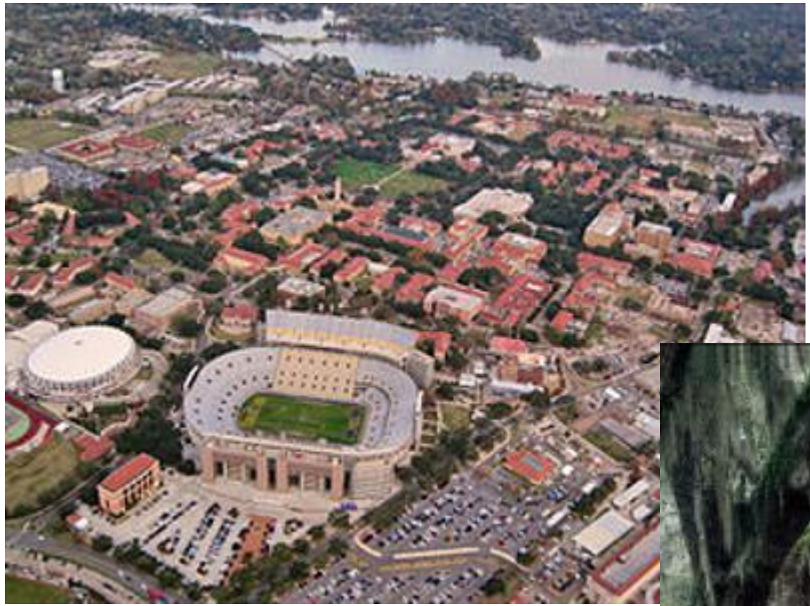
The Solution to the Application Problem



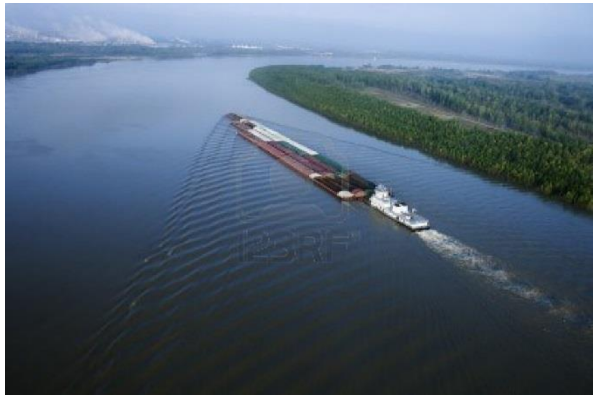
The Solution to the Application Problems



STE||AR GROUP



LSU CENTER FOR COMPUTATION & TECHNOLOGY



STE||AR GROUP