Parallelism in C++

Lecture 1

Hartmut Kaiser (hkaiser@cct.lsu.edu)

Parallelism

- Preconditions for parallelization
  - Availability of independent work (tasks)
  - Availability of more than one computing elements (cores)
- Parallel computing means
  - Executing more than one thing (thread) concurrently
  - Maintain correct order of execution
  - Protect data that is accessed by more than one thread
  - Synchronize execution in between threads

Amdahl's Law (Strong Scaling)

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

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

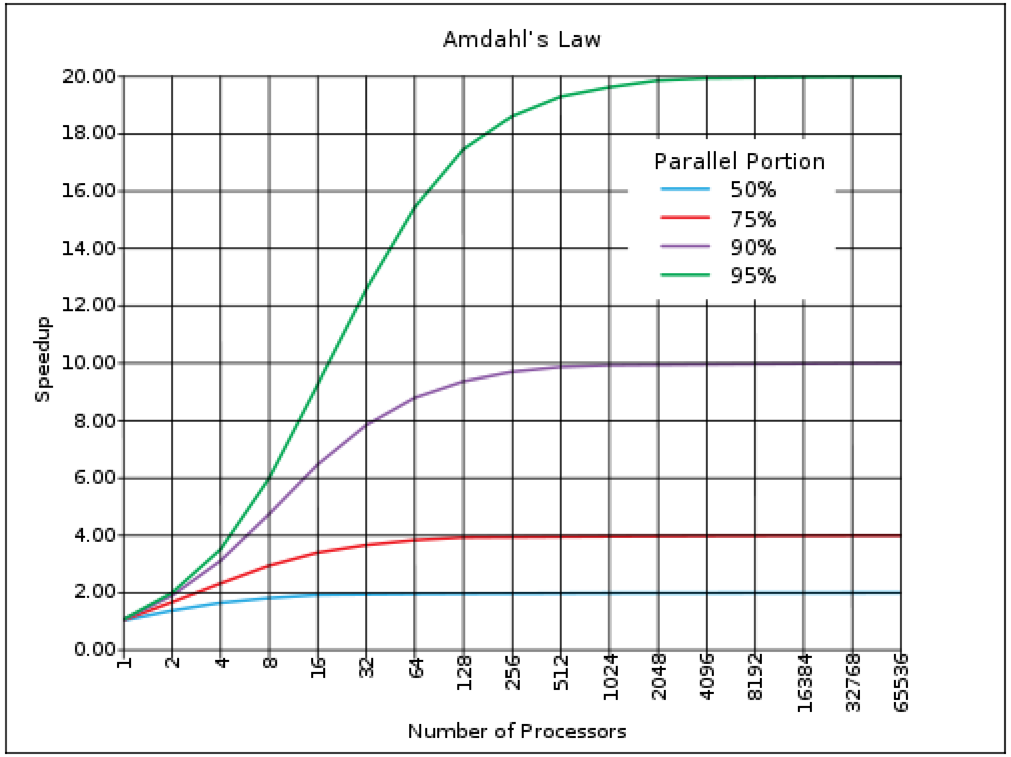


Figure courtesy of Wikipedia (http://en.wikipedia.org/wiki/Amdahl's_law)

Rule 1

Parallelize Applications as Much as Humanly Possible

STE||AR GROUP

The 4 Horsemen of the Apocalypse

courtesy of www.albrecht-durer.org

The 4 Horsemen of the Apocalypse

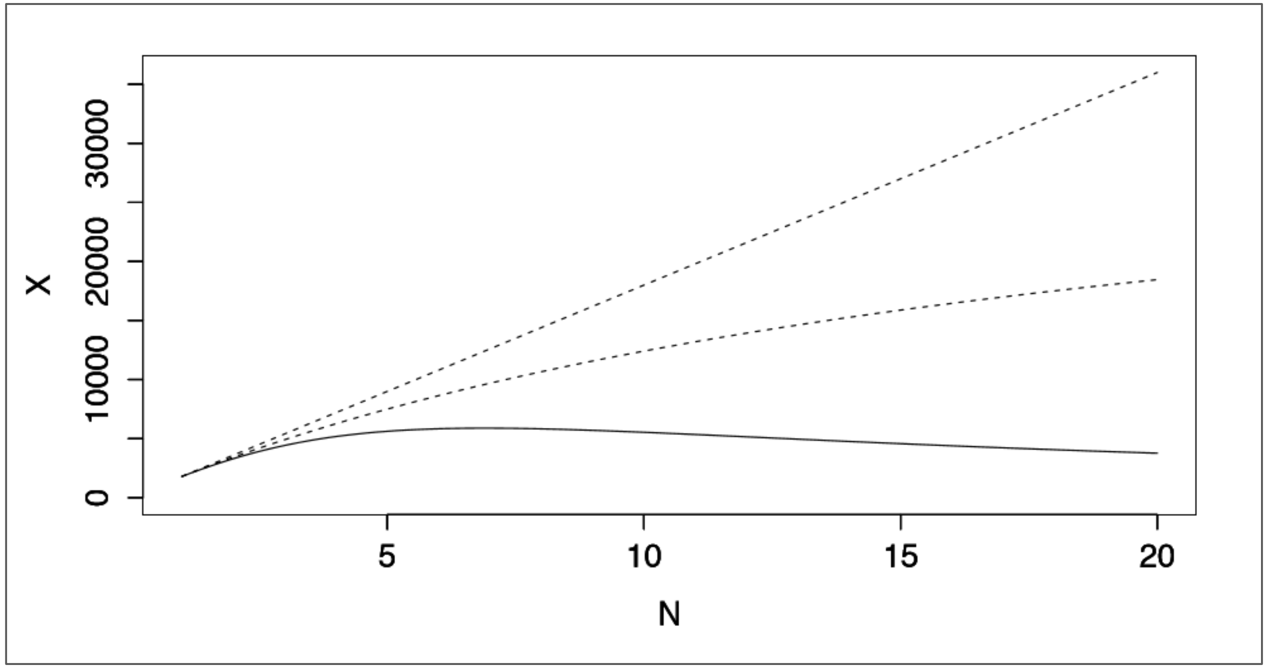
- Starvation
  - Insufficient concurrent work to maintain high utilization of resources
- Latencies
  - Time-distance delay of re
- Overh
  - ns and resources on
    ssary in sequential variant
- Wa for Contention resolution
  - Delays due to lack of availability of oversubscribed shared resources

Impose upper bound on both, weak and strong scaling

courtesy of www.albrecht-durer.org

Universal Scalability Law

$$X(N) = \frac{\lambda N}{1 + \sigma(N-1) + \kappa N(N-1)}$$



- λ: Scaling efficiency
- δ: Contention
- κ: Latencies (‘Crosstalk’)
- N: Number of processors

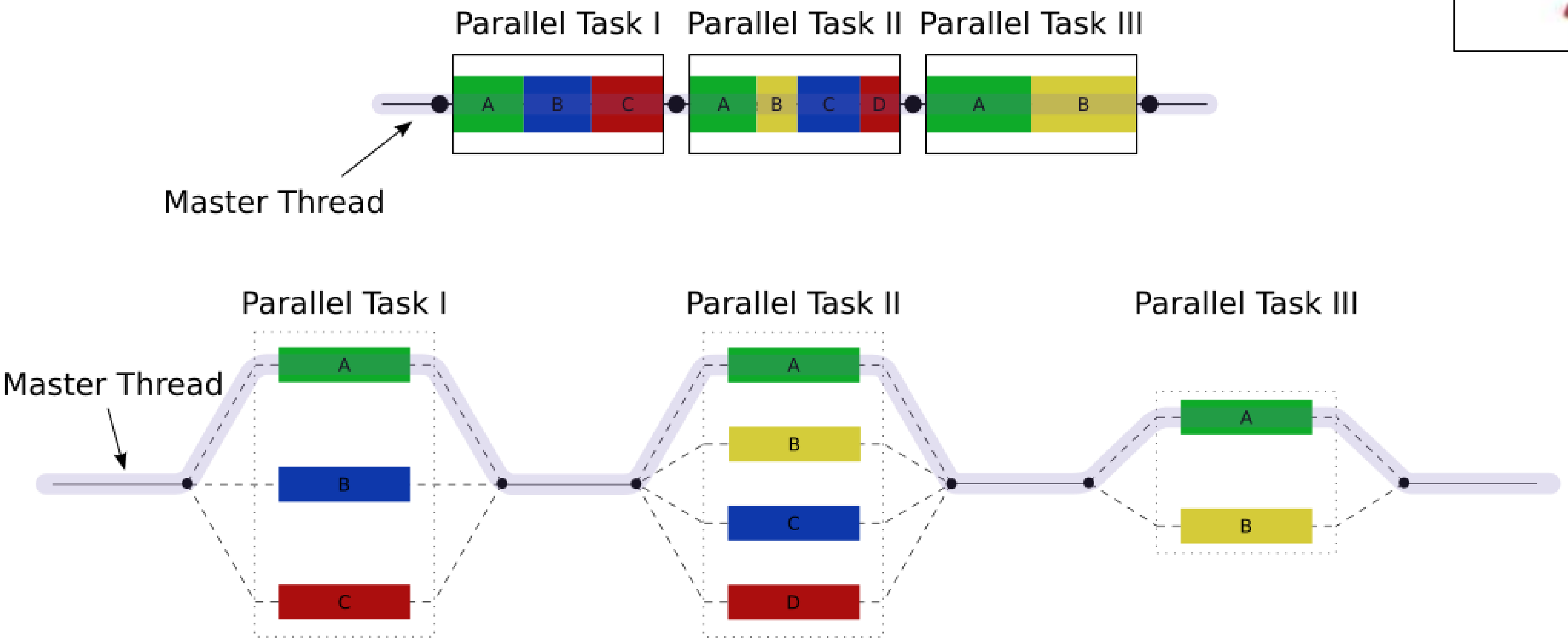
STE||AR GROUP

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.

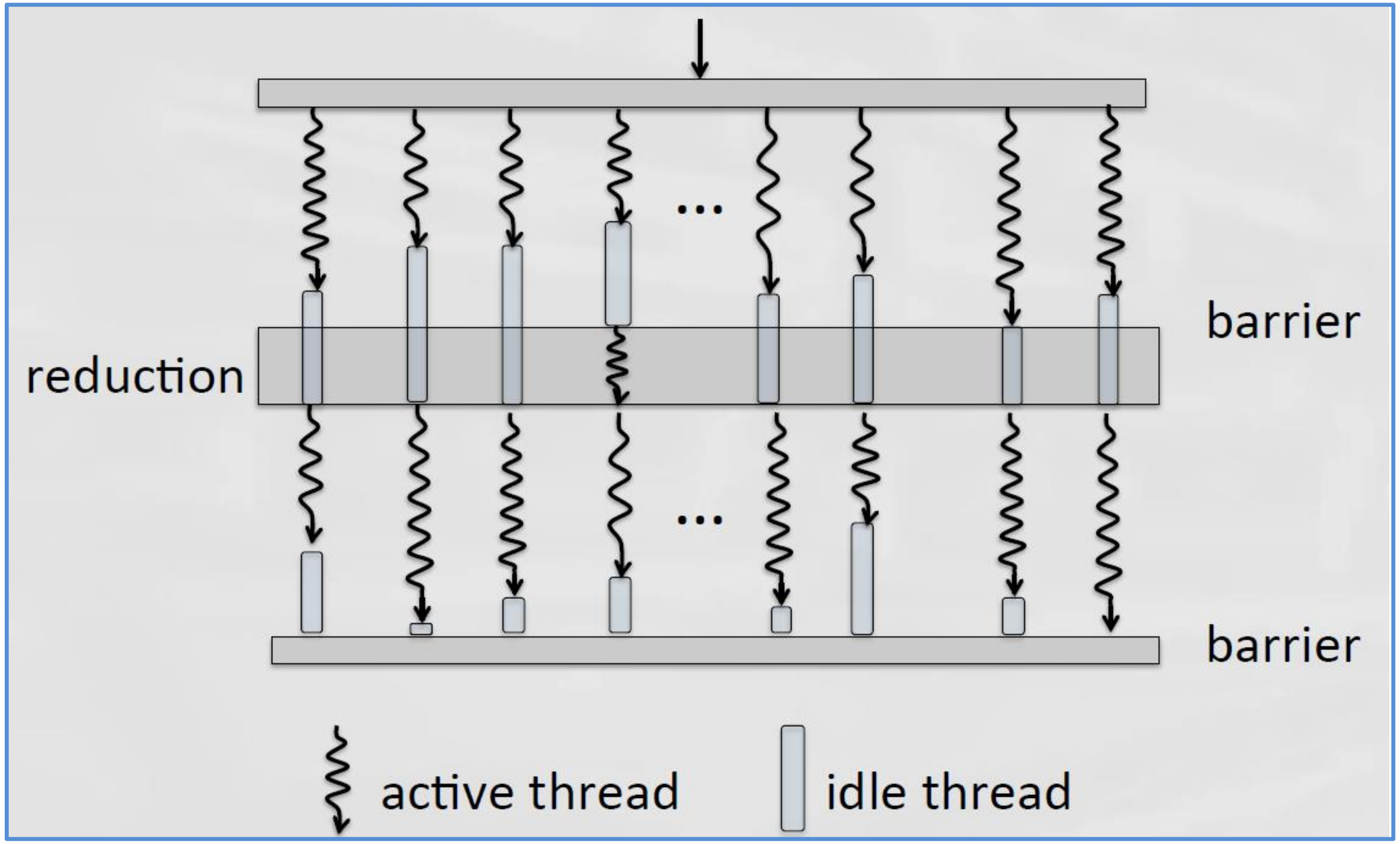
Real-world Problems

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



STE||AR GROUP

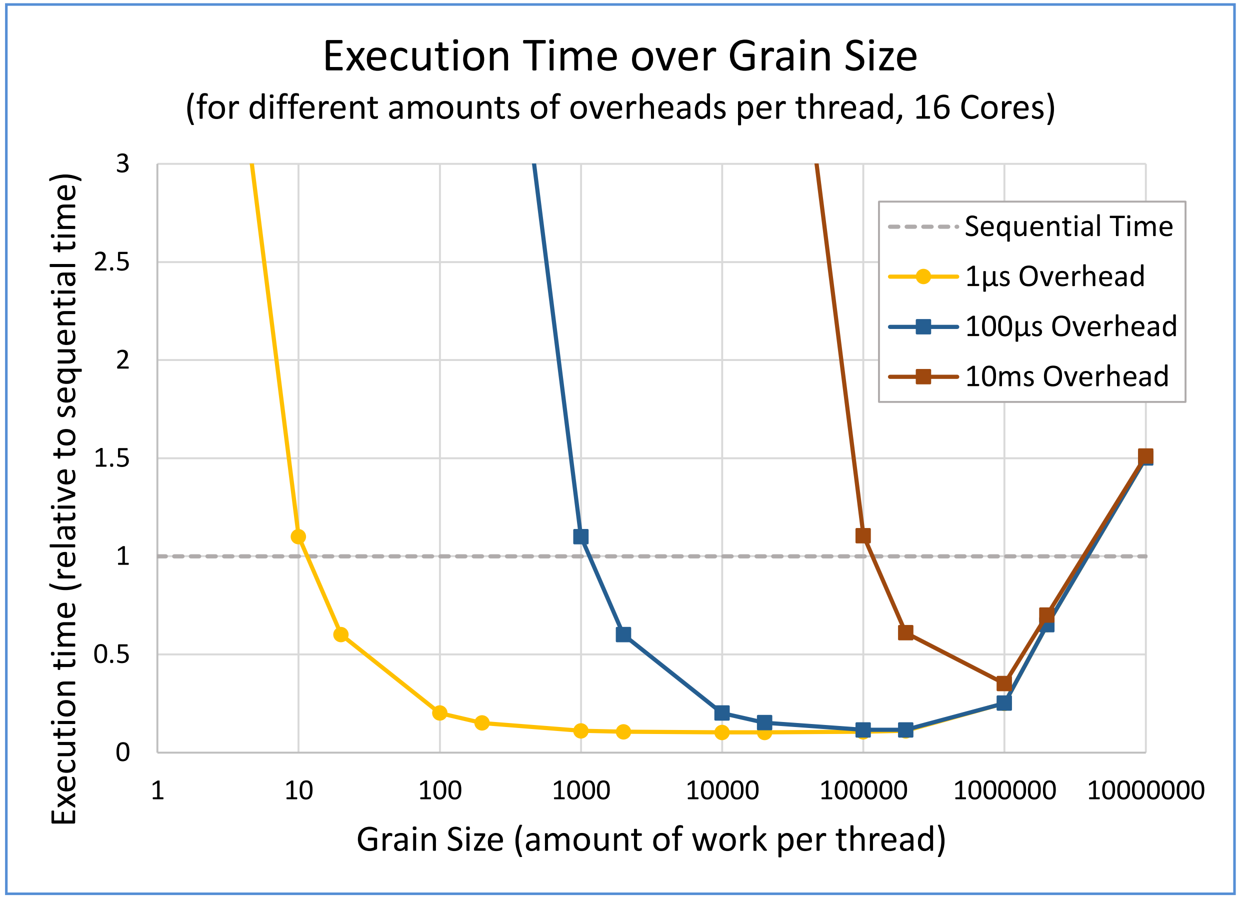
Fork/Join Parallelism



Rule 2

Use a Programming Environment that Embraces SLOW

Overheads: Thought-Experiment



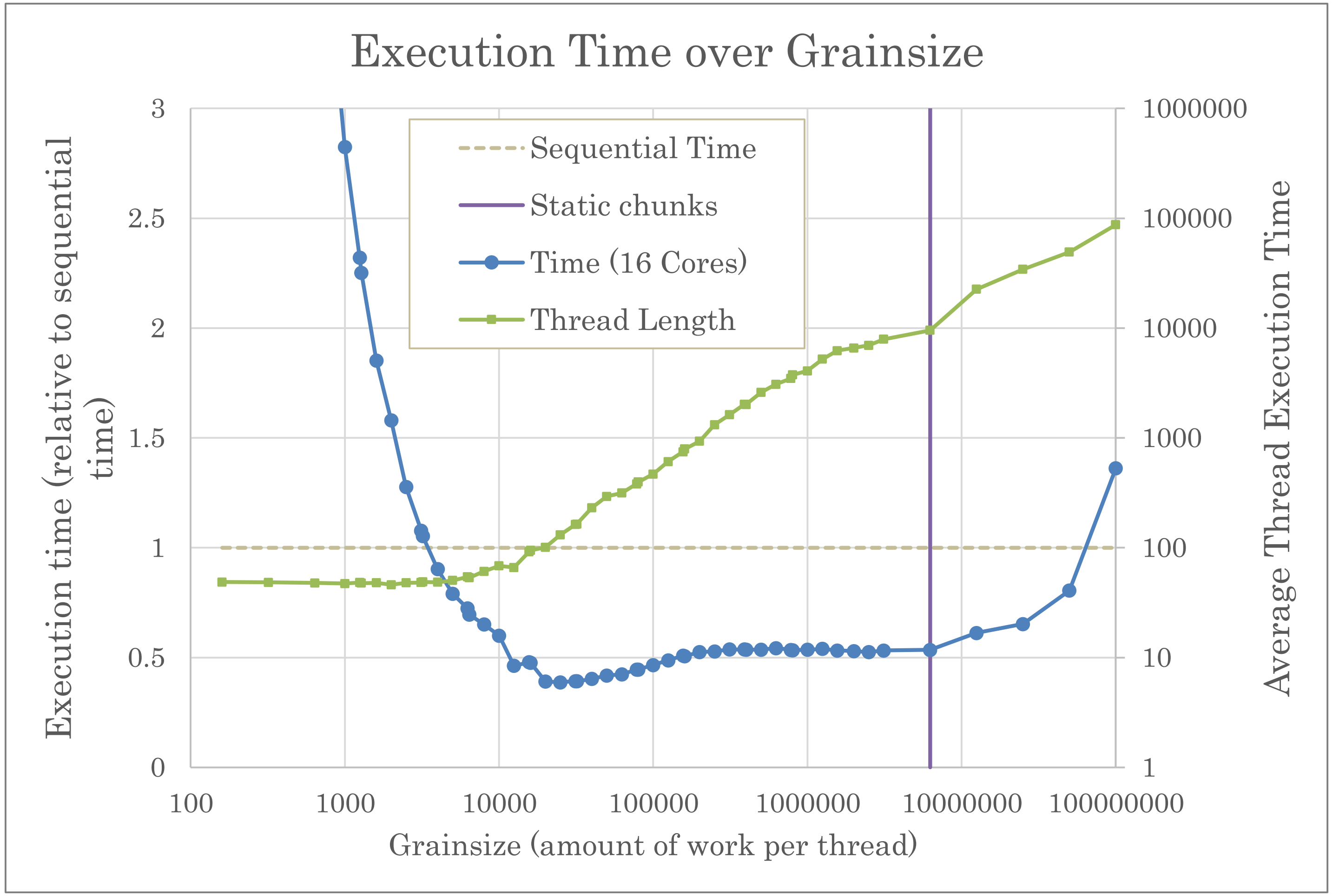
Overheads: The Worst of All?

- 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

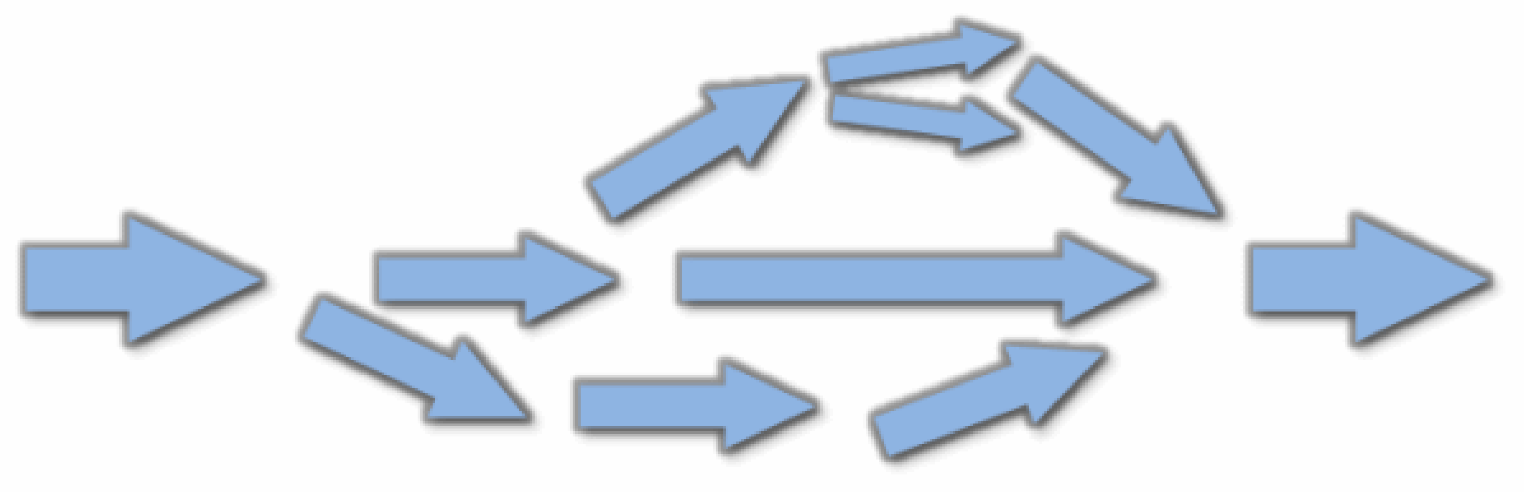
STE||AR GROUP

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

STE||AR GROUP

The Future of Computation



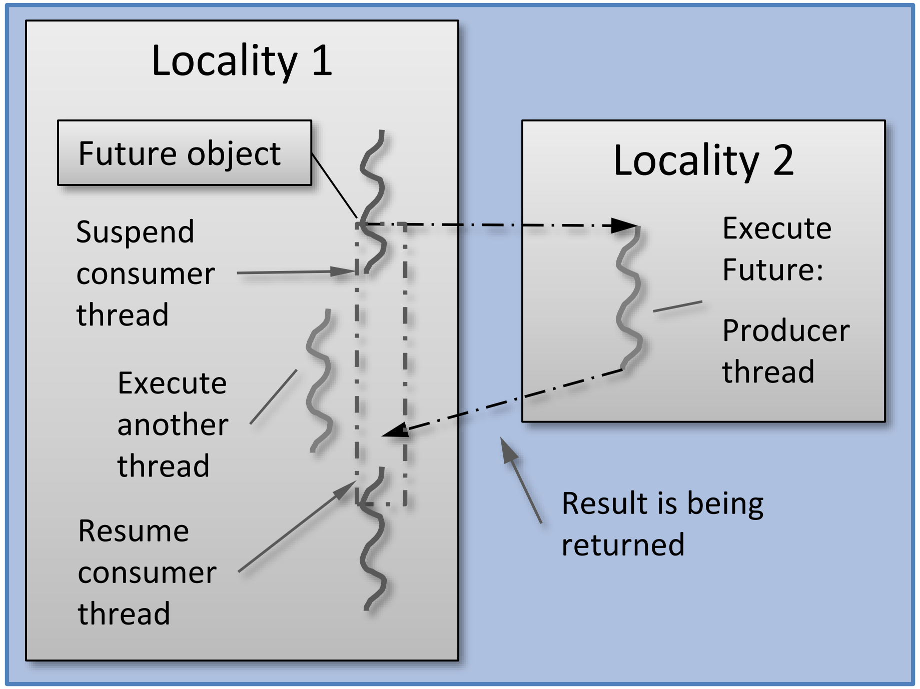
What is a (the) Future?

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

Ways to Create a future

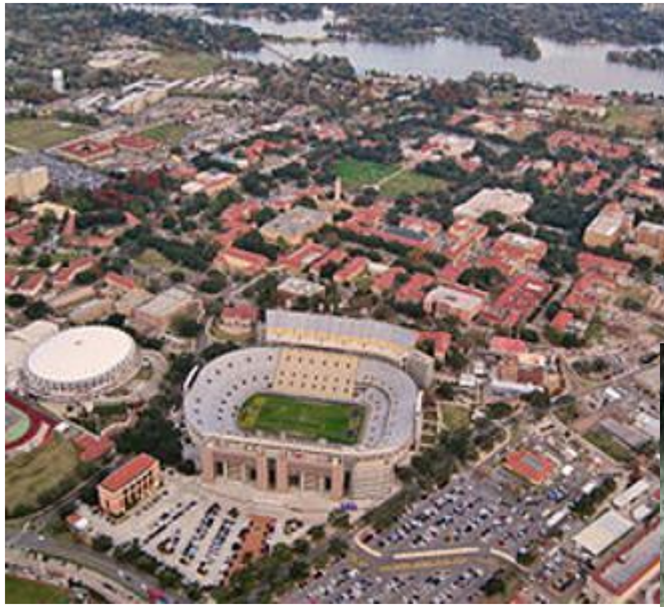
- Standard defines 3 possible ways to create a future,
  - 3 different ‘asynchronous providers’
    - std::async
    - std::packaged_task
    - std::promise

Promising a Future

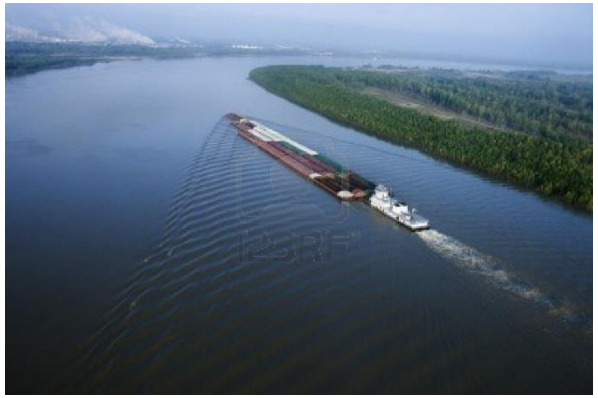
- std::promise is main ‘producer’ of futures
    - It gives away a future representing the value it received
    - Promise/future is a one-shot pipeline where the promise is the ‘sender’ and the future is the ‘receiver’

Demo

STE||AR GROUP



LSU CENTER FOR COMPUTATION & TECHNOLOGY



STE||AR GROUP