Tasks & Concurrency (1)

Lecture 14

Hartmut Kaiser

https://teaching.hkaiser.org/spring2025/csc4400/

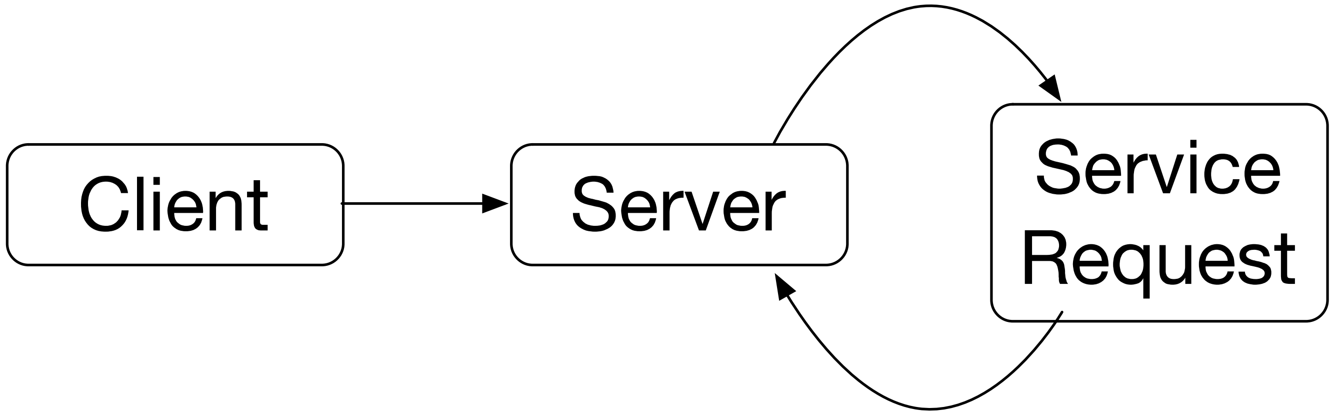
Task-Parallel Model

- Focuses on distributing tasks concurrently performed by processes or threads across different processors (and nodes)
- Task parallelism is distinguished by running many different -- possibly unrelated -- tasks at the same time
  - On the same data
  - On different, even unrelated data
- A common type of task parallelism is pipelining (e.g. chaining tasks)
  - Consists of moving a single set of data through a series of separate tasks
  - Where each task can execute independently of the others
- Explicitly relies on dependencies between tasks
  - Represented by intermediate results computed

Task-Parallel Model

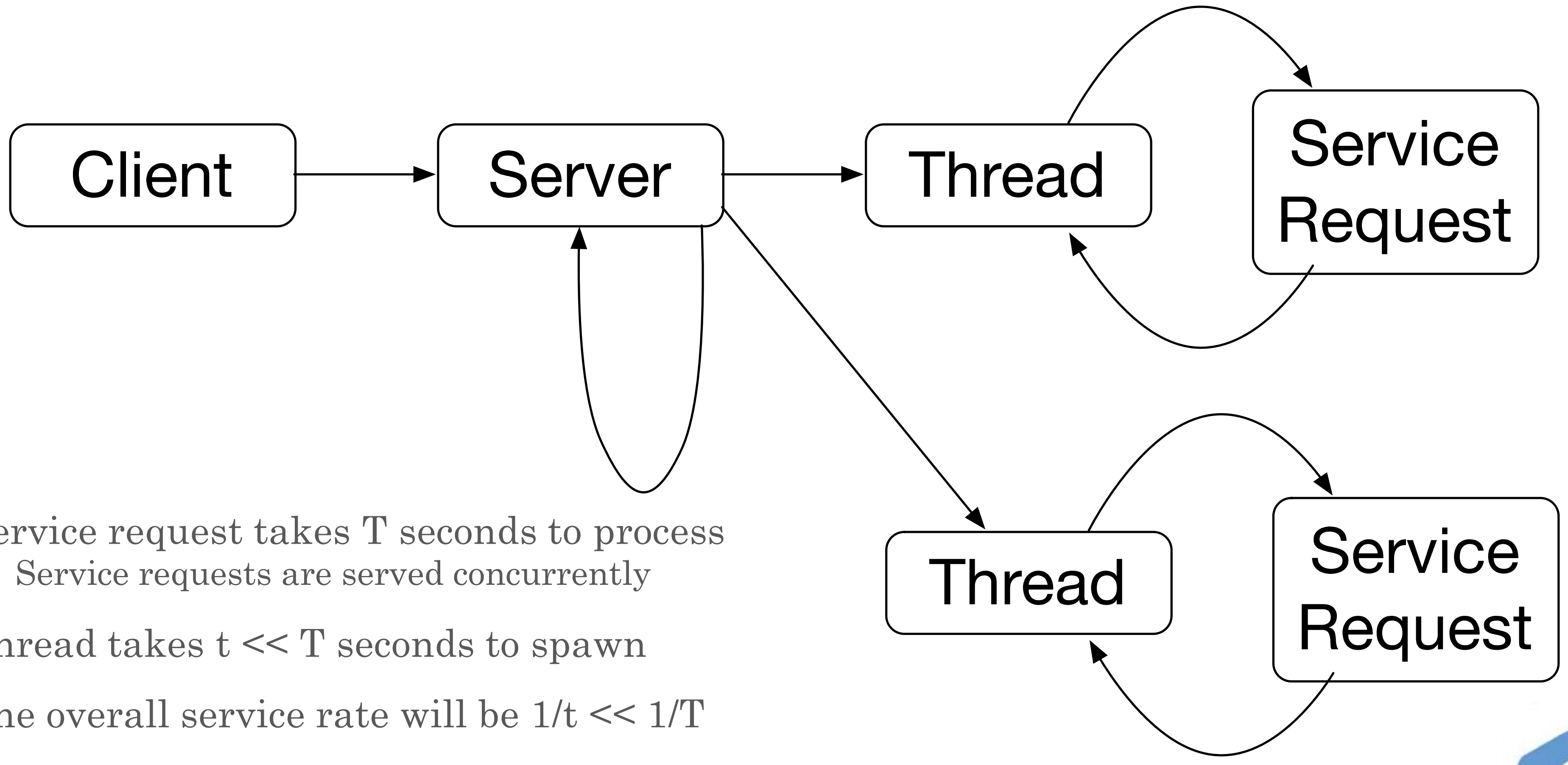
Data parallelism	Task parallelism
Same operations are performed on different subsets of same data	Different operations are performed on the same or different data
Synchronous computation	Asynchronous computation
Speedup is more as there is only one type of execution thread operating on all sets of data	Speedup is less as each processor will execute a different thread or process on the same or different set of data
Amount of parallelization is proportional to the input data size	Amount of parallelization is proportional to the number of independent tasks to be performed
Designed for optimum load balance on multi processor system	Load balancing depends on the availability of the hardware and scheduling algorithms like static and dynamic scheduling

Threads and Asynchrony



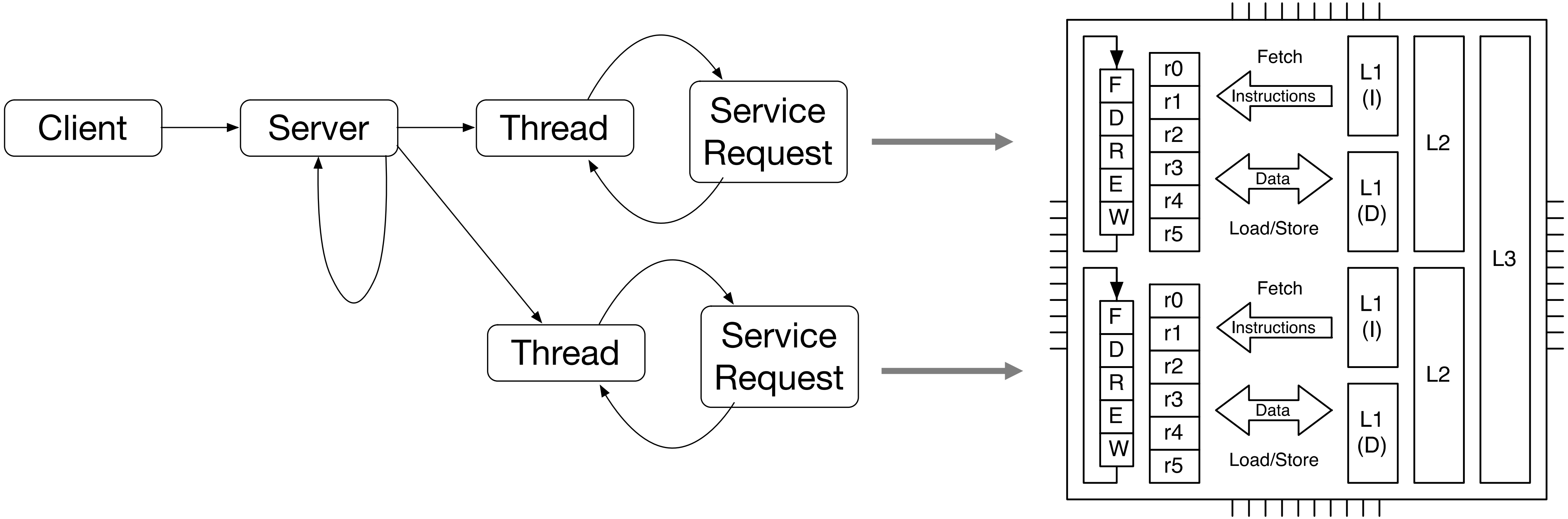
- Service Request takes T seconds to process
- Thus the Service rate will be 1/T

Threads and Asynchrony

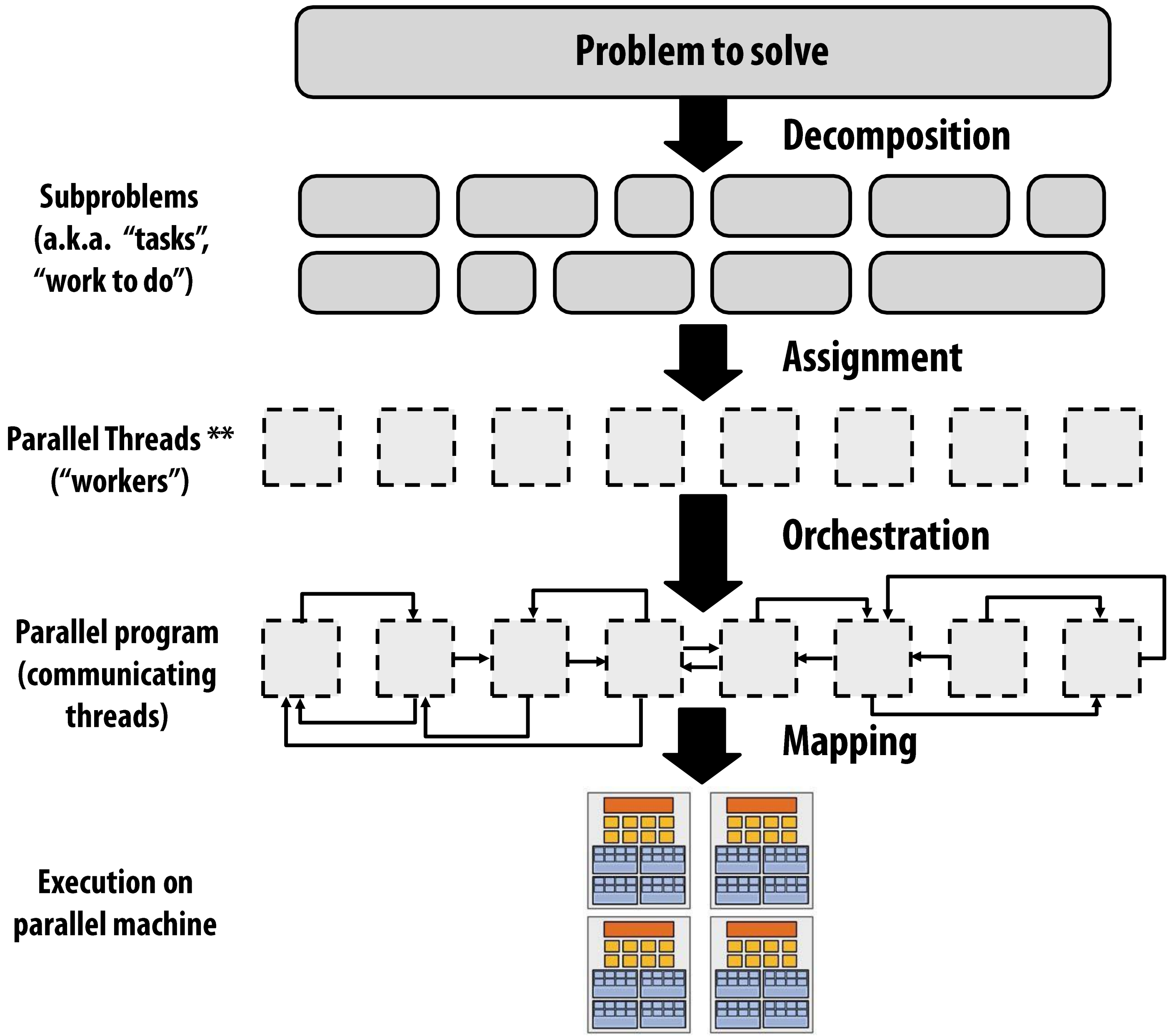


- Service request takes T seconds to process
  - Service requests are served concurrently
- Thread takes t << T seconds to spawn
- The overall service rate will be 1/t << 1/T

Multitasking on Multicore

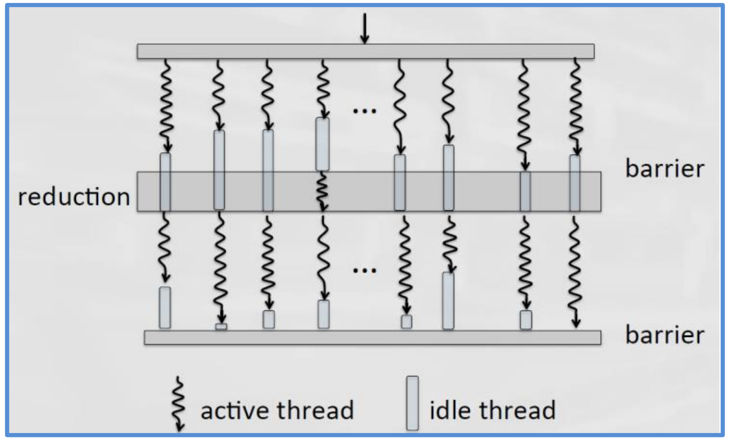


- On multiple cores, concurrent tasks can run in parallel


Problem to solve
Decomposition
Subproblems (a.k.a. “tasks”, “work to do”)
Assignment
Parallel Threads ** (“workers”)
Orchestration
Parallel program (communicating threads)
Mapping
Execution on parallel machine

The Challenges

- We need to find a usable way to fully parallelize our applications
  - Remember Amdahl's Law
  - Avoid 'sequential' pieces of execution by all means possible
  - Fork/Join parallelism has hidden sequential pieces



- Goals are:
  - Expose asynchrony to the programmer
  - 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 (HPX' AGAS: active global address space)
  - Tasks operate on C++ objects possibly distributed across the system

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

Asynchronous Tasks

Think in Terms of Tasks, not Threads

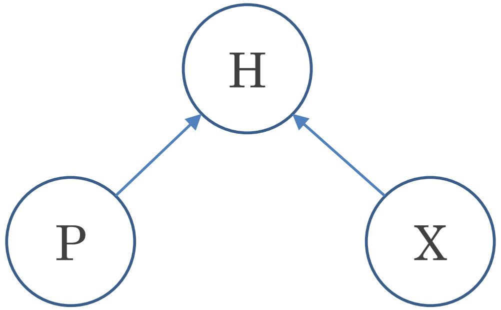
- A thread is an implementation concept, a way of thinking about the machine
- A task is an application notion, something you'd like to do
  - Preferably concurrently with other tasks
  - Sequencing happens based on data dependencies, not function order
  - Application concepts are easier to reason about
- Try to reason about “What to do?”, “What needs to be done before this?” (tasks)
- Rather to think about “How to do things?” (threads)

Synchronous Programming

- Simple code:

```
auto P = compute_p();
auto X = compute_x();
auto H = compute_h(P, X);
```

- Dependency graph (implicit):

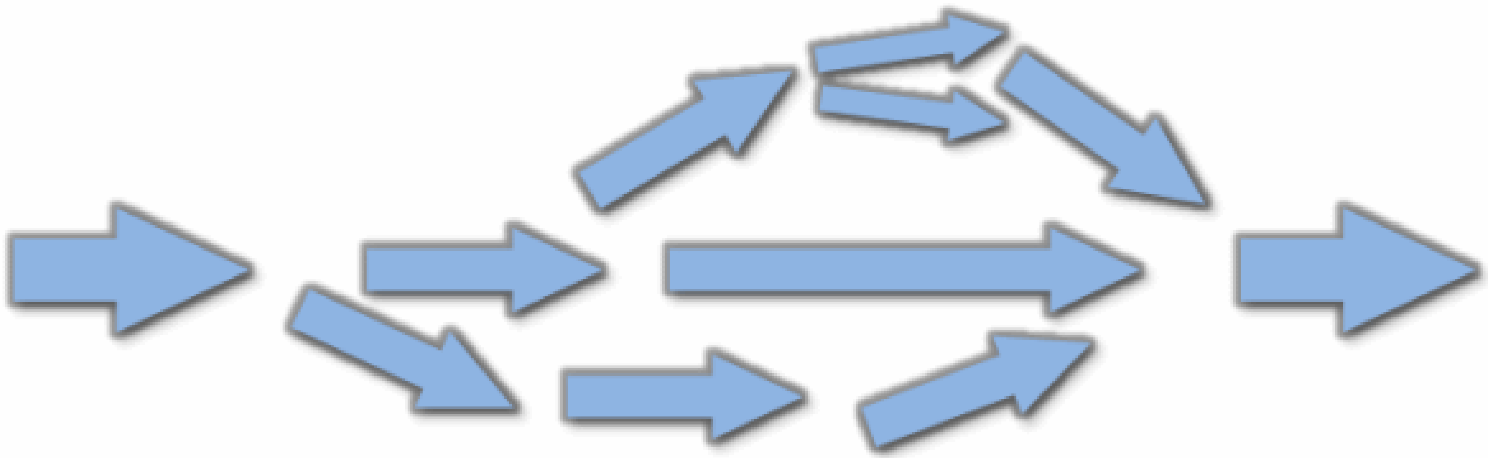


- The program is executed line by line
- Each time a function is called the calling code waits until the functions finishes
- We could compute `P` and `X` at the same time, since the data is independent

Asynchronous Tasks

- As we have seen before, using plain threads
  - Make it complicated to 'return' results from separate computations
  - Often requires additional thread-safety measures to aggregate over many results
- Today, we will see what facilities can be applied to overcome these limitations
- The idea is to create mechanisms that allow to directly access the return value of a function spawned asynchronously
  - That also allows for keeping the asynchronous functions side-effect free
- Additionally we will develop an asynchronous programming model that minimizes thread suspension and synchronization
  - We will introduce C++ language features that directly support this model

Aside: 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:

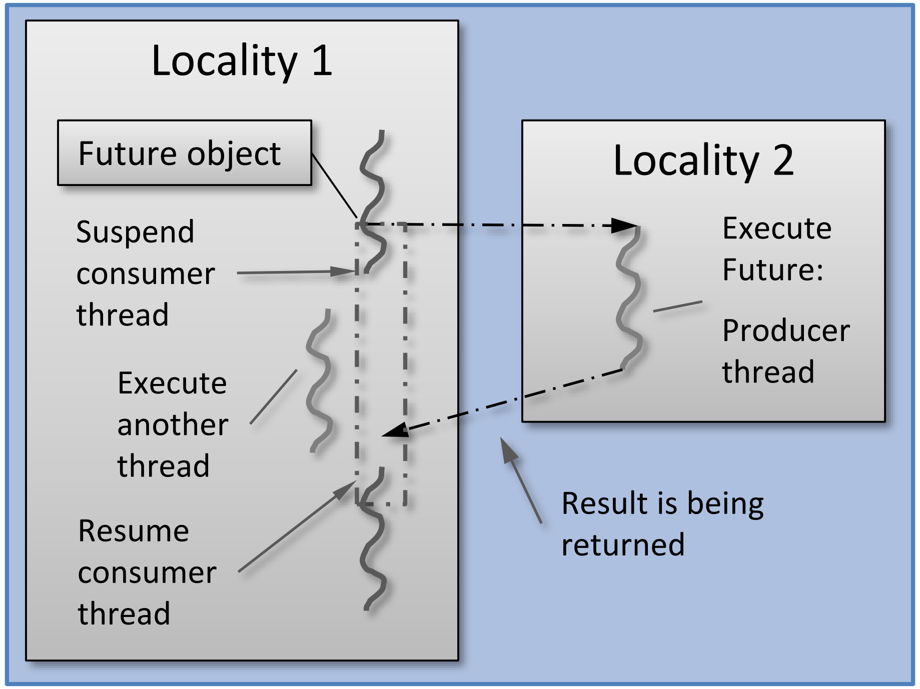
```
int universal_answer() { return 42; }

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 `std::future` provides a mechanism to access the result of an asynchronous operations
  - Like one created by `std::async` and
  - It provides methods for synchronization with the result
- Synchronization:
  - `.get()` suspends until the computation has finished and returns the result of the function
  - `.wait()` waits until the computation has finished
  - `.wait_for(std::chrono::seconds(1))` returns if the result is not available after the specified timeout duration
  - `.wait_until(std::chrono::now()+std::chrono::seconds(1))` waits for a result to become available until given point in time
  - Both block until specified timeout time has been reached or the result becomes available, whichever comes first.

What is a (the) future

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



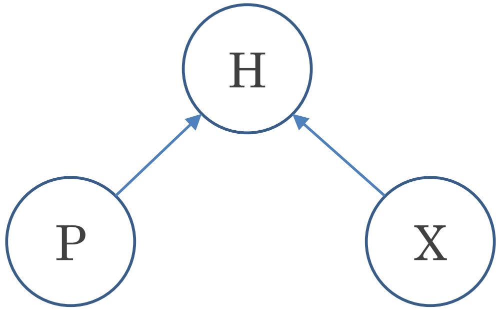
- Enables transparent synchronization with producer
- Hides notion of dealing with threads
- Represents a data-dependency
- Makes asynchrony manageable
- Allows for composition of several asynchronous operations
- (Turns concurrency into parallelism)

Synchronous Programming

- Simple code:

```cpp
auto P = compute_p();
auto X = compute_x();
auto H = compute_h(P, X);
```

- Dependency graph (implicit):



- The program is executed line by line
- Each time a function is called the calling code waits until the functions finishes
- We could compute `P` and `X` at the same time, since the data is independent

Asynchronous Programming

- Computing independent things concurrently:

```
std::future<int> f = std::async(compute_p);
auto X = compute_x();
auto H = compute_h(f.get(), X);
```

- The program is still executed line by line
- However, `std::async` returns right away, even if the `compute_p` has not finished yet
  - Returned future `f`, which represents the result that will be computed
  - It can be used to synchronize with the execution of `compute_p`
- In the code above `compute_p` and `compute_x` are being run concurrently (if compute resources are available)

Asynchronous Execution of Functions

- Testing whether a given number is a prime:

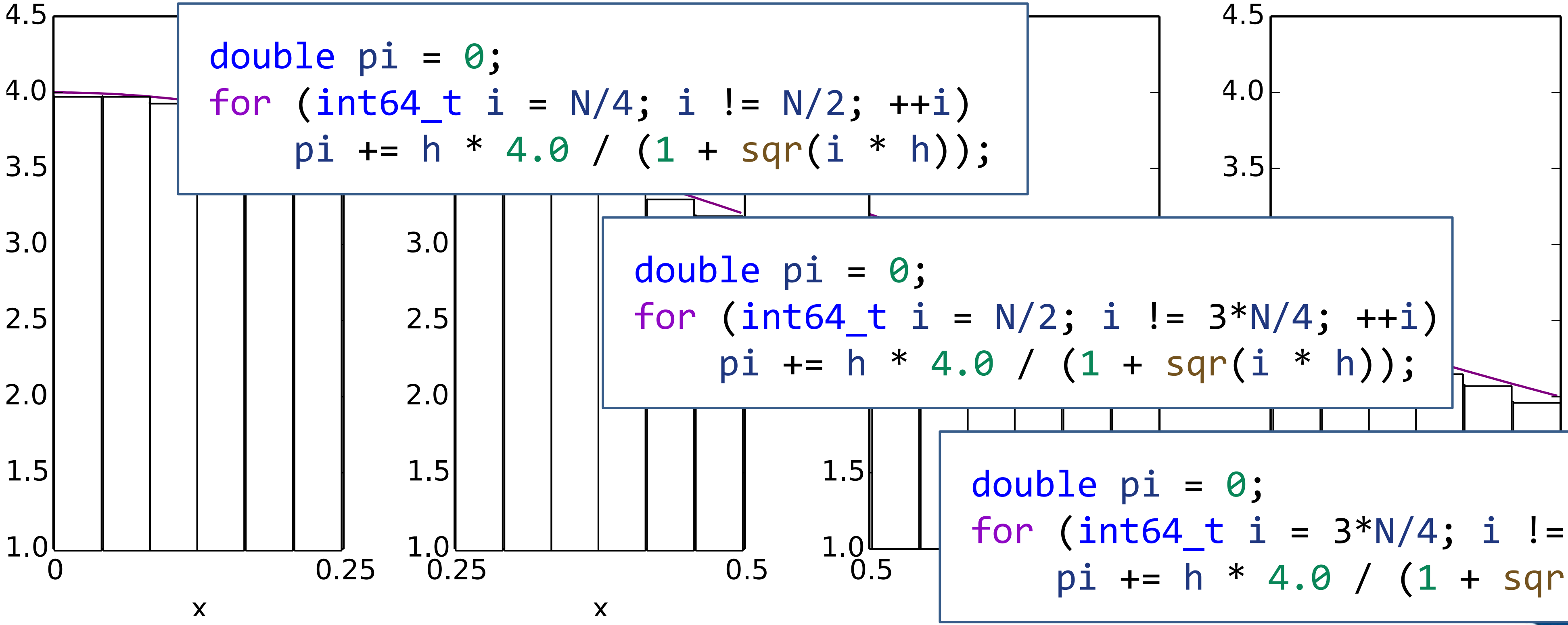
```
bool is_prime (long x) {
    std::println("Calculating. Please , wait ...");
    long limit = std::sqrt(x);
    for (long i = 2; i < limit; ++i)
        if (x % i == 0) return false;
    return true;
}

std::future<bool> f = std::async(is_prime, 313222313);
// ... Do other things
std::println("313222313 is {}a prime", f.get() ? "" : "not ");
```

Asynchronous Execution of Functions

- `std::async`
  - The first argument is a function (pointer) to execute, any ‘invocable’ works
  - The second argument is the first argument of the function, and so on
  - The return value is a `std::future<T>` where `T` is the return type of the function
- For each call, `std::async` launches a new thread to execute the function
- The returned future can be used to ‘wait’ for the result to be available
  - The value eventually provided by the future is whatever the function returned that was passed to `std::async`

Numeric Integration with Tasks

```
double pi = 0;
for (int64_t i = 0; i != N/4; ++i)
    pi += h * 4.0 / (1 + sqr(i * h));
```




```
double pi = 0;
for (int64_t i = N/4; i != N/2; ++i)
    pi += h * 4.0 / (1 + sqr(i * h));
```

```
double pi = 0;
for (int64_t i = N/2; i != 3*N/4; ++i)
    pi += h * 4.0 / (1 + sqr(i * h));
```

```
double pi = 0;
for (int64_t i = 3*N/4; i != N; ++i)
    pi += h * 4.0 / (1 + sqr(i * h));
```

Numeric Integration with Tasks

```
double calculate_pi()
{
    int64_t N = 1'000'000'000;    // really large number
    double h = 1.0 / (double) num_intervals; // Decomposition
    double pi = 0.0;
    int num_blocks = 4;
    int64_t block_size = N / num_blocks;

    // For each set of discretized points
    for (int k = 0; k < num_blocks; ++k) {
        pi += pi_helper(n * block_size, (n + 1) * block_size, h);
    }
    return pi;
}
```

Numeric Integration with Tasks

- Helper function that computes the sub-result for one of the blocks

```
double pi_helper(int64_t begin, int64_t end, double h)
{
    double local_pi = 0.0;
    for (int64_t i = begin; i != end; ++i)
        local_pi += h * 4.0 / (1 + sqr(i * h));
    return local_pi;
}
```

Numeric Integration with Tasks

```cpp
int main(int argc, char* argv[])
{
    int64_t N = 1'000'000'000;    // really large number
    double h = 1.0 / N;
    int num_blocks = 4;
    double pi =  0.0;
    int64_t block_size = N / num_blocks;

    std::vector<std::future<double>> part_pi;
    for (int64_t n = 0; n != num_blocks; ++n)
        part_pi.push_back(
            std::async(pi_helper, n * block_size, (n + 1) * block_size, h));

    for (int n = 0; n != num_blocks; ++n)
        pi += part_pi[n].get();

    std::println("pi: {} ", pi);
}
```

Numeric Integration with Tasks

- All tasks run in parallel (as long as compute resources are available
  - The helper function `pi_helper` has no 'awareness' that it is run on a separate thread
  - It is still a 'normal' function
  - Simply returns calculated result
- No synchronization is needed (at all)
- Little change compared to sequential version

Results

```
pi_async 1000000000 1
pi: 3.1415926545900716, time: 1068 [ms]

pi_async 1000000000 2
pi: 3.1415926545897657, time: 575 [ms]

pi_async 1000000000 4
pi: 3.141592654589842, time: 386 [ms]

pi_async 1000000000 6
pi: 3.141592646589882, time: 313 [ms]

pi_async 1000000000 8
pi: 3.141592654589781, time: 298 [ms]
```

- Sequential
- 2 threads, speedup of ~2
- 4 threads, speedup of ~3
- 6 threads, speedup of ~4
- 8 threads, speedup of ~4

Launch Policies for `std::async`

- `std::async` takes an optional first argument:

```
for (int n = 0; n != num_blocks; ++n)
    part_pi.push_back(std::async(std::launch::async, pi_helper,
        n * block_size, (n + 1) * block_size, h));
```

- Launch policy, could be:

`std::launch::async`	could the task is executed on a different thread, potentially by creating and launching it first
`std::launch::deferred`	the task is executed on the calling thread the first time its result is requested (lazy evaluation)

- The default is 'whatever'
  - Always make sure your specify the launch policy

Results when using deferred

```
pi_async 1000000000 1
pi: 3.1415926545900716, time: 1125 [ms]

pi_async 1000000000 2
pi: 3.1415926545897657, time: 1055 [ms]

pi_async 1000000000 4
pi: 3.141592654589842, time: 1062 [ms]

pi_async 1000000000 6
pi: 3.141592646589882, time: 1060 [ms]

pi_async 1000000000 8
pi: 3.141592654589781, time: 1146 [ms]
```

- Sequential
- 2 threads, speedup of ~1
- 4 threads, speedup of ~1
- 6 threads, speedup of ~1
- 8 threads, speedup of ~1

Aside: Name This Famous Couple

Bonnie Parker

Clyde Barrow

Bonnie and Clyde Use ATMs


```
int bank_balance = 300;

void withdraw(std::string const& name, int amount)
{
    int bal = bank_balance;              // Get current balance
    std::println("{} withdraws {}", name, amount);
    bank_balance = bal - amount;         // compute new balance and save it
}

int main() {
    std::println("Starting balance {}", bank_balance);

    std::future<void> f1 =
        std::async(std::launch::async, withdraw, "Bonnie", 100);
    std::future<void> f2 =
        std::async(std::launch::async, withdraw, "Clyde", 100);

    f1.get();
    f2.get();

    std::println("Final balance {}", bank_balance);
    return 0;
}
```

Bonnie and Clyde Use ATMs


```
$ ./a.out
Starting balance is 300
Bonnie withdraws 100
Clyde withdraws 100
```

Why is this not correct?

Prevent Race Condition

```
std::mutex msg_mutex;   // protect printing the message
std::mutex atm_mutex;   // protect the bank balance

void withdraw(std::string const& name, int amount)
{
    std::lock(msg_mutex, atm_mutex);   // Prevent deadlock

    int bal = bank_balance;            // Get current balance

    std::lock_guard msg_lock(msg_mutex, std::adopt_lock);
    std::println("{} withdraws {}", name, amount);

    std::lock_guard atm_lock(atm_mutex, std::adopt_lock);
    bank_balance = bal - amount;       // compute new balance and save it
}
```

Back to the Future

Ways to Create a `future`

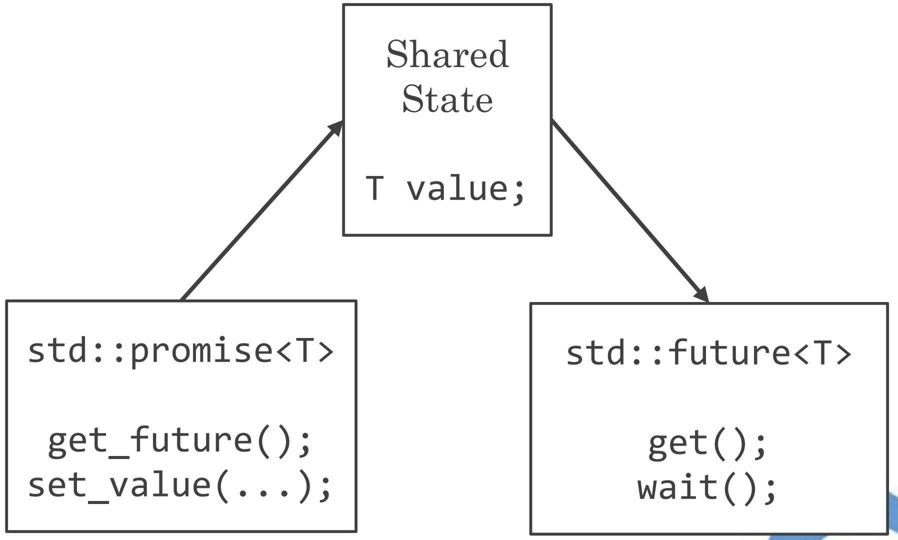
- Standard defines 3 possible ways to create a future
  - 3 different 'asynchronous providers'
    - `std::async`
    - `std::packaged_task`
    - `std::promise`
- As we will see, with HPX there are many more ways to create one

Promising a Future

- `std::promise` is main 'maker' 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'
- Promise and future represent a anonymous connection between a producer and a consumer
  - The producer sets the value in the promise
  - The consumer receives the value from the future
- The `promise` initially creates a shared state
  - The `future` created by the promise shares the state with it
  - The shared state stores the value, etc.

Promising a Future

- Futures are created by promises: `f = promise.get_future()`
- The promise is used to set the result value: `promise.set_value(r)`
- The future is used to access the result value: `r = f.get()`
- The shared state is invisible
  - Stores the value
  - Manages synchronization and ensures thread safety
- Promises and futures are thread-safe



Promising a Future: async

```
template <typename F, typename... Args>
std::future<std::invoke_result_t<F, Args...>> async(F f, Args... args)
{
    using result_type = std::invoke_result_t<F, Args...>;

    std::promise<result_type> p;
    std::future<result_type> f = p.get_future();

    std::thread t([=]() { p.set_value(f(args...)); });  // note: simplified!
    t.detach();    // detach the thread from t

    return f;
}
```

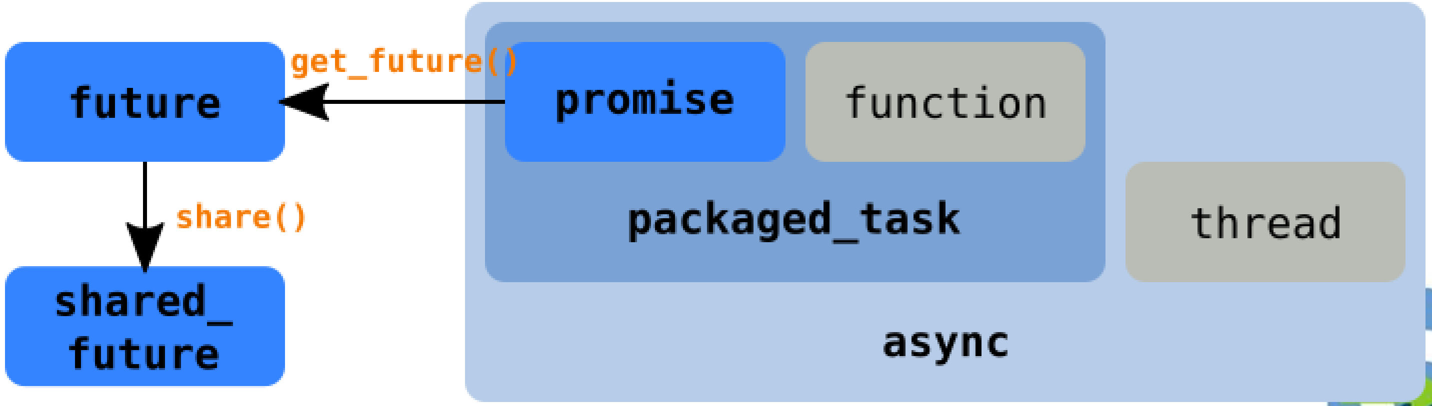
Waiting for the Future

```
int main() {
    std::promise<int> p;
    std::future<int> f = p.get_future();

    std::jthread t([=]() {
        std::this_thread::sleep_for(std::chrono::seconds(5));
        p.set_value(42);
    });

    for (int i = 0; /**/; ++i) {
        std::println("Waiting attempt {} ...", i);
        std::future_status status = f.wait_for(std::chrono::seconds(1));
        if (status != std::future_status::timeout) break;
    }
    std::println("Computed result: {}", f.get());
}
```

```
Waiting attempt 0 ...
Waiting attempt 1 ...
Waiting attempt 2 ...
Waiting attempt 3 ...
Waiting attempt 4 ...
Computed result: 42
```

Packaging a Future: packaged_task

- `std::packaged_task` is a function object
  - It gives away a future representing the result of its invocation
- Can be used as a synchronization primitive
  - Pass to APIs that accepts a callback function
- Converting a callback into a future
  - Observer pattern, allows to wait for a callback to happen



Promising a Future: packaged_task

- Very Simple example:

```
int main()
{
    std::packaged_task<int(int, int)> task(
        [](int a, int b) { return std::pow(a, b); });
    std::future<int> result = task.get_future();

    task(2, 9);

    std::println("task: {}", result.get());    // prints: task: 512 (2^9)

    return 0;
}
```

Promising a Future: packaged_task

```
template <typename F> class packaged_task;

template <typename R, typename... Args>
class packaged_task<R(Args...)>
{
    std::function<R(Args...)> fn;
    std::promise<R> p;                  // the promise for the result

public:
    template <typename F>
    explicit packaged_task(F f) : fn(f) {}

    void operator()(Args... args) { p.set_value(fn(args...)); }

    std::future<R> get_future() { return p.get_future(); }
};
```

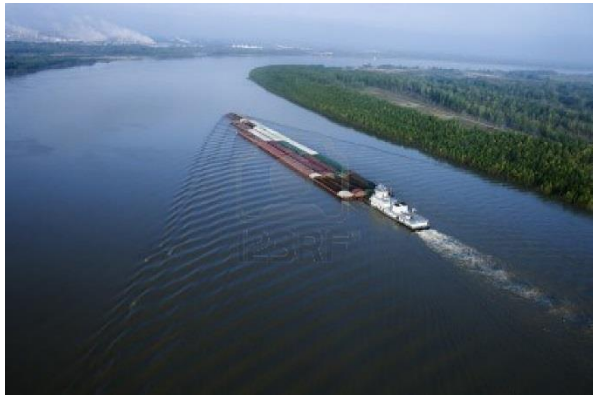
Packaging a Future: async

```cpp
template <typename F, typename... Args>
std::future<std::invoke_result_t<F, Args...>> async(F f, Args... args)
{
    std::packaged_task<F(Args...)> pt(f);
    auto f = pt.get_future();

    std::thread t(pt, args...);     // note: simplified!
    t.detach();    // detach the thread from t

    return f;
}
```

Lessons Learnt so far

- Assume that someone someday will run your code as part of a multi-threaded program
- Avoid data races
- Minimize explicit sharing of writable data
- Think in terms of tasks, rather than threads (`std::async` is your friend!)
- Use RAII, never plain lock()/unlock()
- Use `std::lock()` to acquire multiple mutexes
- Use `std::launch::async` when using `std::async()`

LSU CENTER FOR COMPUTATION & TECHNOLOGY



STE||AR GROUP