STE||AR GROUP

http://stellar-goup.org

ASYNCHRONOUS COMPUTING IN C++

Hartmut Kaiser (Hartmut.Kaiser@gmail.com)

CppCon 2014

WHAT IS ASYNCHRONOUS COMPUTING?

- Spawning off some work without immediately waiting for the work to finish
  - Asynchronous work
  - May produce result (some value) or not (just trigger)
- Either: wait for the asynchronous work at some later point
- Or: attach a continuation which is automatically run once the work is done
- While this sounds like parallelism, it is not directly related, however
  - May be used to (auto-) parallelize code (this talk will show an example)
  - Runs just as well in single threaded environments
  - Runs just as well in environments with an arbitrary number (millions) of threads

WHAT IS ASYNCHRONOUS COMPUTING?

- Also called ‘reactive computing’, ‘actor computing’, or ‘observer pattern’
  - Propagation of change using the concepts of (static and dynamic) dataflow
- There are many existing asynchronous environments
  - JavaScript, C#, widely adopted in functional languages
  - In this talk, the term ‘asynchronous computing’ is used
    - Presented concepts are not ‘strictly’ reactive
    - Attempt to integrate dataflow with ‘normal’ imperative C++
- All content of this talk is based on using such an environment: HPX
  - HPX is a general purpose parallel runtime system for applications of any scale

STE||AR GROUP

WHY ASYNCHRONOUS COMPUTING?

Tianhe-2’s projected theoretical peak performance: 54.9 PetaFLOPs

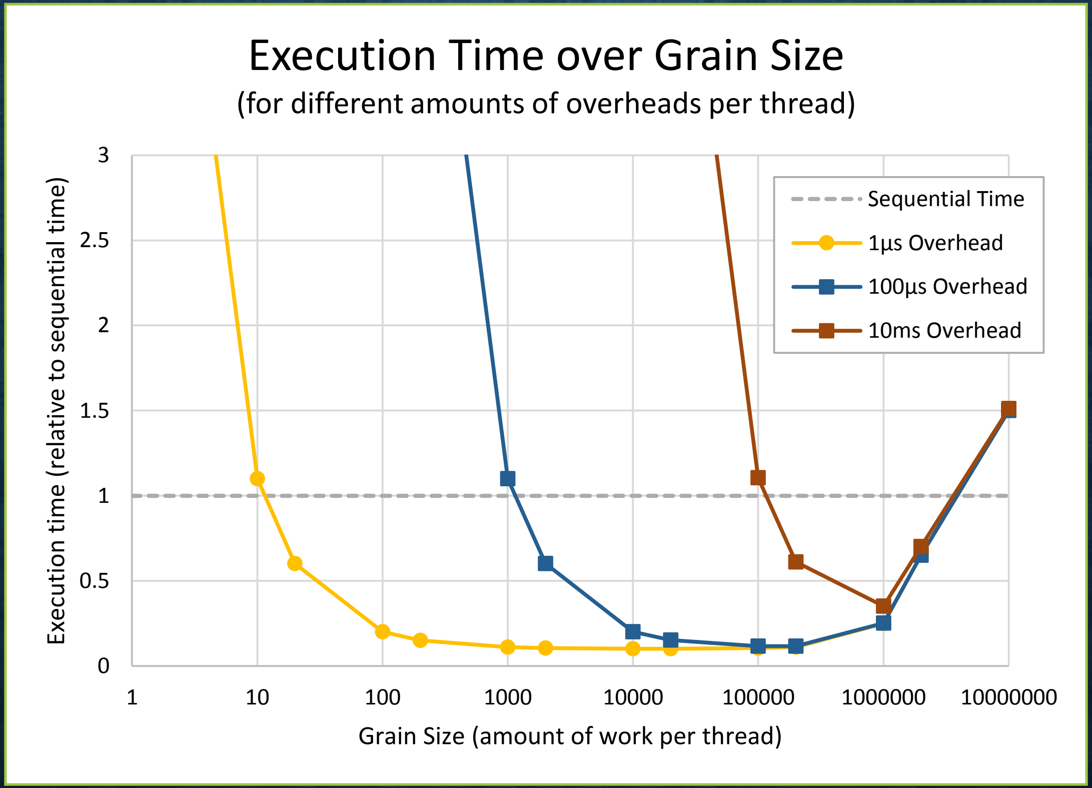
16,000 nodes, ~3,200,000 computing cores (32,000 Intel Ivy Bridge Xeons, 48,000 Xeon Phi Accelerators)

ASYNCHRONOUS ENVIRONMENTS

- Asynchronous computing requires an appropriate runtime system which supports scheduling of work
  - All existing asynchronous environments have such a runtime system
- C++ has the standard library
  - Surprisingly the existing concepts are suitable for this (with some extensions)
  - Main facility is the type 'future<T>'
- Default implementations of 'future<T>' are based on kernel threads
  - Too coarse grain, too much overhead

STE||AR GROUP

WHY IS STD::THREAD TOO SLOW?



STE||AR GROUP

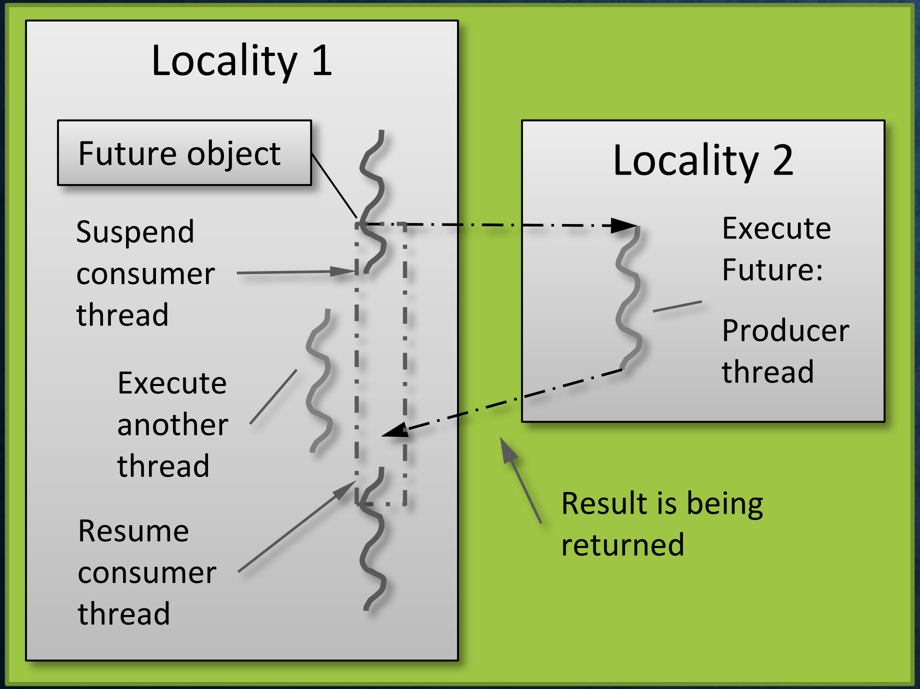
ASYNCHRONOUS ENVIRONMENTS

- Even relatively small amounts of work can benefit from being split into smaller tasks
  - Possibly huge amount of 'threads'
    - In the previous gedankenexperiment we ended up considering up to 10 million threads
    - Best possible scaling is predicted to be reached when using 10000 threads (for 10s 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

CURRENT STD::FUTURE

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

STE||AR GROUP

WHAT IS A (THE) FUTURE?

- Many ways to get hold of a 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, eventually
}
```


STE||AR GROUP

WAYS TO CREATE A FUTURE

- Standard defines 3 possible ways to create a future,
  - 3 different '*asynchronous providers*'
    - std::async
      - See previous example, std::async has caveats
    - std::packaged_task
    - std::promise

PACKAGING A FUTURE

- 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 std::thread
- Converting a callback into a future
  - Observer pattern, allows to wait for a callback to happen

PACKAGING A FUTURE

```
template <typename F, typename ...Arg>
std::future<typename std::result_of<F(Arg...)>::type>
simple_async(F func, Arg&& arg...)
{
    std::packaged_task<F> pt(func);
    auto f = pt.get_future();

    std::thread t(std::move(pt), std::forward<Arg>(arg)...);
    t.detach();

    return std::move(f);
}
```


STE||AR GROUP

PROMISING A FUTURE

- std::promise is also an *asynchronous provider* ("an object that provides a result to a shared state")
  - The promise is the thing that you *set* a result on, so that you can *get* it from the associated future.
  - The promise initially creates the shared state
  - The future created by the promise shares the state with it
  - The shared state stores the value

PROMISING A FUTURE

```
template <typename F> class simple_packaged_task;

template <typename R, typename ...Args>
class simple_packaged_task<R(Args...)>    // must be move-only
{
    std::function<R(Args...)> fn;
    std::promise<R> p;                    // the promise for the result
    // ...
public:
    template <typename F>  explicit simple_packaged_task(F && f) : fn(std::forward<F>(f)) {}

    template <typename ...T>
    void operator()(T &&... t) { p.set_value(fn(std::forward<T>(t)...)); }

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

EXTENDING STD::FUTURE

EXTENDING STD::FUTURE

- Several proposals (draft technical specifications) for next C++ Standard
  - Extension for future<>
    - Compositional facilities
      - Parallel composition
      - Sequential composition
  - Parallel Algorithms
  - Parallel Task Regions
- Extended async semantics: dataflow

STE||AR GROUP

MAKE A READY FUTURE

- Create a future which is ready at construction (N3857)

```
future<int> compute(int x)
{
    if (x < 0) return make_ready_future<int>(-1);
    if (x == 0) return make_ready_future<int>(0);

    return async([](int par) { return do_work(par); }, x);
}
```

COMPOSITIONAL FACILITIES

- Sequential composition of futures (see N3857)

```
string make_string()
{
    future<int> f1 = async([]() -> int { return 123; });
    future<string> f2 = f1.then(
        [](future<int> f) -> string {
            return to_string(f.get());    // here .get() won't block
        });
}
```


STE||AR GROUP

COMPOSITIONAL FACILITIES

- Parallel composition of futures (see N3857)

```
void test_when_all() {
    shared_future<int> shared_future1 = async([]() -> int { return 125; });
    future<string> future2 = async([]() -> string { return string("hi"); });

    future<tuple<shared_future<int>, future<string>>> all_f =
        when_all(shared_future1, future2);                    // also: when_any, when_some, etc.

    future<int> result = all_f.then(
        [](future<tuple<shared_future<int>, future<string>>> f) -> int {
            return do_work(f.get());
        });
}
```


STE||AR GROUP

PARALLEL ALGORITHMS

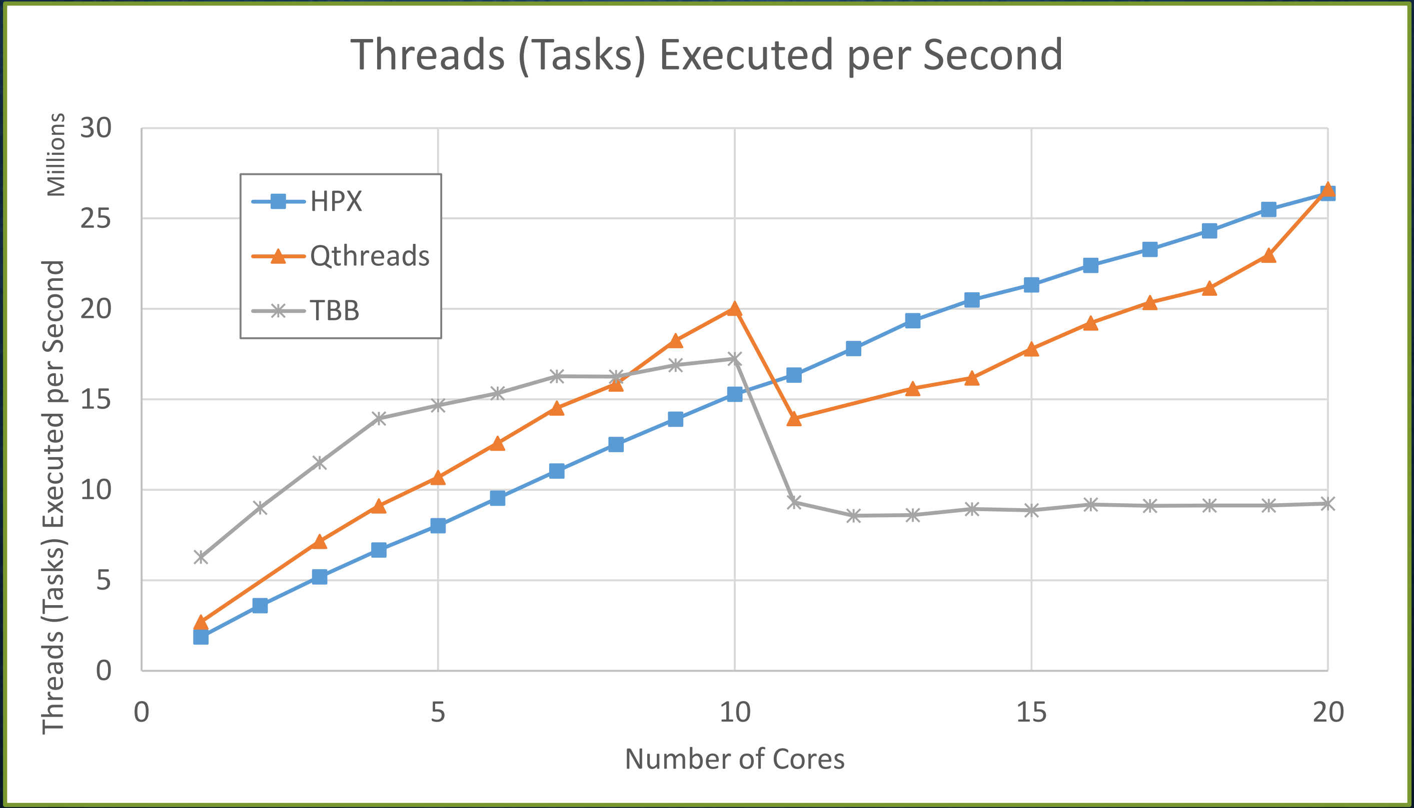
- Parallel algorithms (N4071)
  - Mostly, same semantics as sequential algorithms
  - Additional, first argument: execution_policy (seq, par, etc.)
- Extension
  - task_execution_policy
  - Algorithm returns future<>

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		

HPX – A GENERAL PURPOSE RUNTIME SYSTEM

- Solidly based on a theoretical foundation - ParalleX
  - A general purpose parallel runtime system for applications of any scale
    - http://stellar-group.org/libraries/hpx
    - https://github.com/STEllAR-GROUP/hpx/
- Exposes an uniform, standards-oriented API for ease of programming parallel and distributed applications.
  - Enables to write fully asynchronous code using hundreds of millions of threads.
  - Provides unified syntax and semantics for local and remote operations.
- Enables writing applications which out-perform and out-scale existing ones
- Is published under Boost license and has an open, active, and thriving developer community.
- Can be used as a platform for research and experimentation

THREAD OVERHEADS



HPX – THE API

- As close as possible to C++11/14 standard library, where appropriate, for instance

• std::thread	hpx::thread
• std::mutex	hpx::mutex
• std::future	hpx::future (including N3857)
• std::async	hpx::async (including N3632)
• std::bind	hpx::bind
• std::function	hpx::function
• std::tuple	hpx::tuple
• std::any	hpx::any (N3508)
• std::cout	hpx::cout
• std::parallel::for_each, etc.	hpx::parallel::for_each (N4071)
• std::parallel::task_region	hpx::parallel::task_region (N4088)

EXTENDING ASYNC: DATAFLOW

- What if one or more arguments to ‘async’ are futures themselves?
- Normal behavior: pass futures through to function
- Extended behavior: wait for futures to become ready before invoking the function:

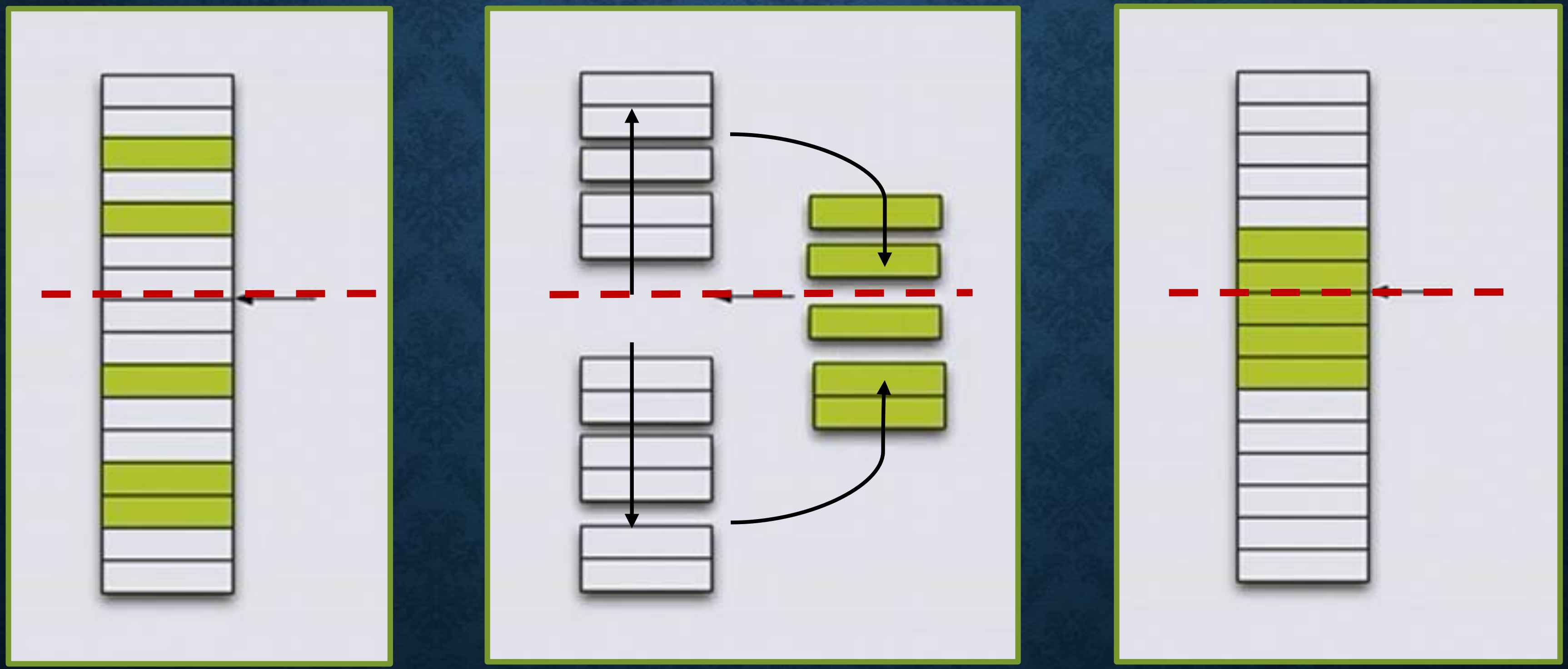
```
template <typename F, typename… Arg>
future<typename result_of<F(Args…)>::type> dataflow(F&& f, Arg&&… arg);
```

- If ArgN is a future, then the invocation of F will be delayed
- Non-future arguments are passed through

STE||AR GROUP

TWO EXAMPLES

EXTENDING PARALLEL ALGORITHMS



EXTENDING PARALLEL ALGORITHMS

- New algorithm: gather

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


STE||AR GROUP

EXTENDING PARALLEL ALGORITHMS

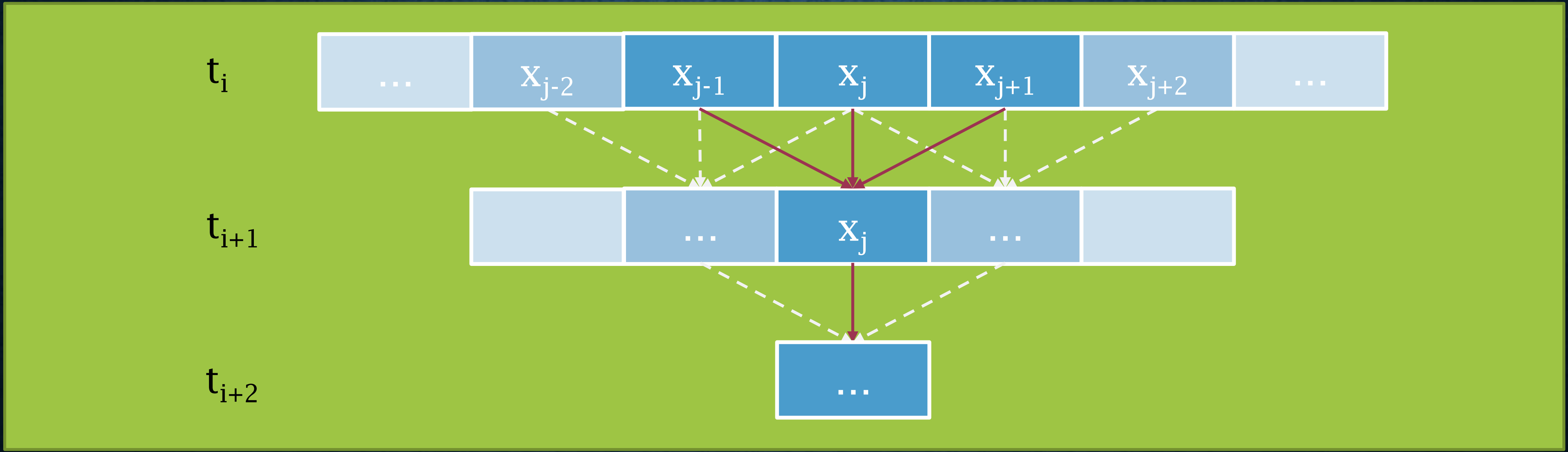
- New algorithm: gather_async

```
template <typename BiIter, typename Pred>
future<pair<BiIter, BiIter>> gather_async(BiIter f, BiIter l, BiIter p, Pred pred)
{
    return dataflow(
        unwrapped([](BiIter r1, BiIter r2) { return make_pair(r1, r2); }),
        parallel::stable_partition(task, f, p, not1(pred)),
        parallel::stable_partition(task, p, l, pred));
}
```


STE||AR GROUP

1D HEAT EQUATION

- Iteratively simulating 1D heat diffusion



STE||AR GROUP

1D HEAT EQUATION

- Kernel: simple iterative heat diffusion solver, 3 point stencil

```
double heat(double left, double middle, double right)
{
    return middle + (k*dt/dx*dx) * (left - 2*middle + right);
}
```

1D HEAT EQUATION

- One time step, periodic boundary conditions:

```
void heat_timestep(std::vector<double>& next, std::vector<double> const& curr)
{
    #pragma omp parallel for
    for (std::size_t i = 0; i != nx; ++i)
        next[i] = heat(current[idx(i-1, nx)], current[i], current[idx(i+1, nx)]);
}
```


STE||AR GROUP

1D HEAT EQUATION

- Time step iteration:

```
std::array<std::vector<double>, 2> U = { std::vector<double>(nx), std::vector<double>(nx) };
for (std::size_t t = 0; t != nt; ++t)
{
    std::vector<double> const& current = U[t % 2];
    std::vector<double>& next = U[(t + 1) % 2];

    heat_timestep(next, curr);
}
```

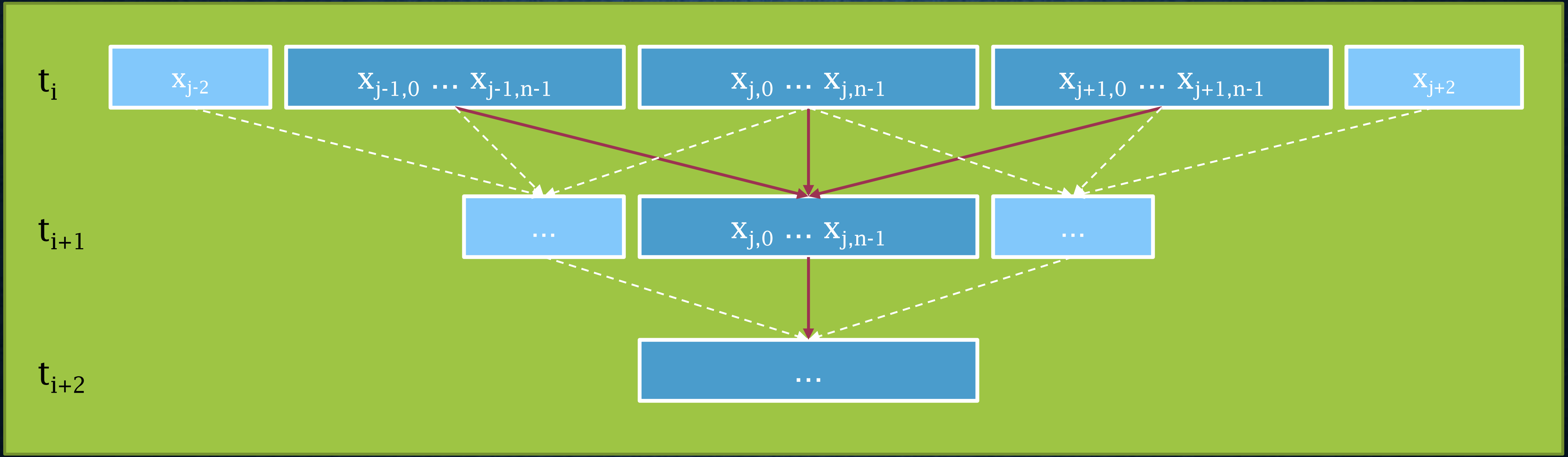
1D HEAT EQUATION, FUTURIZED

- One time step, periodic boundary conditions:

```
void heat_timestep(
    std::vector<shared_future<double>>& next,
    std::vector<shared_future<double>> const& curr)
{
    for (std::size_t i = 1; i != nx-1; ++i) {
        next[i] = dataflow(unwrapped(heat),
            current[idx(i-1, nx)], current[i], current[idx(i+1, nx)]);
    }
}
```

1D HEAT EQUATION, PARTITIONED

- Partitioning data into parts to control grain size of work



1D HEAT EQUATION, FUTURIZED

- Time step iteration:

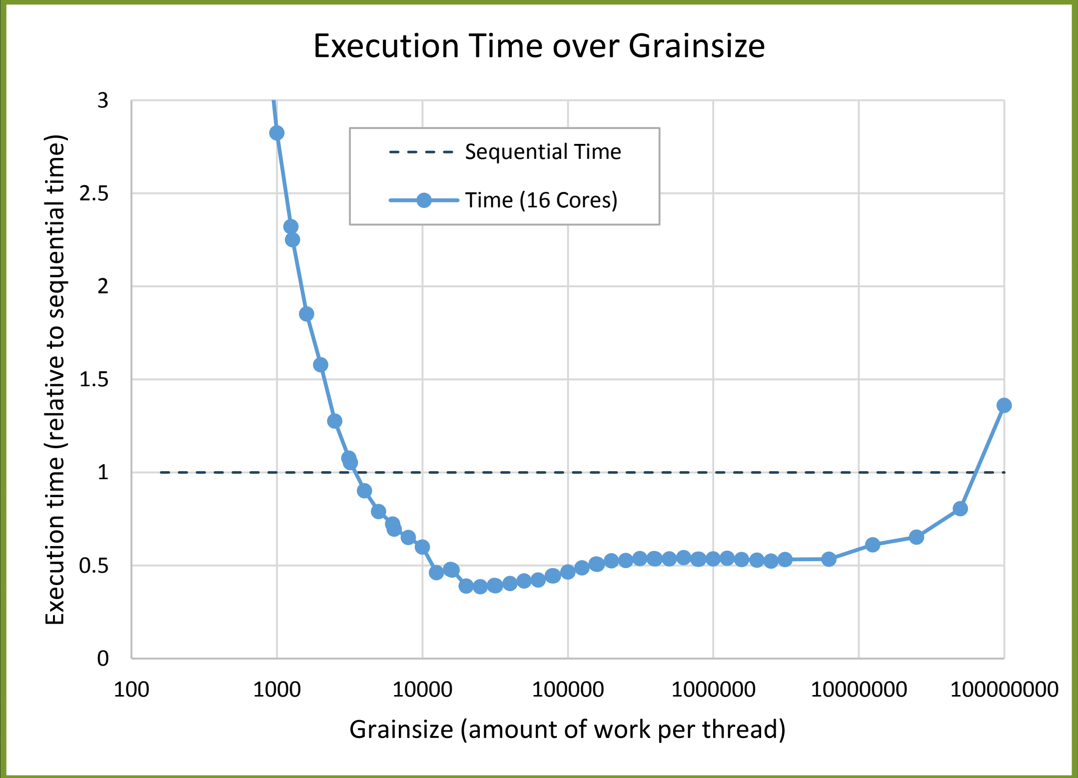
```
std::array<std::vector<shared_future<std::vector<double>>>, 2> U { ... };
for (std::size_t t = 0; t != nt; ++t)
{
    std::vector<shared_future<std::vector<double>>> const& current = U[t % 2];
    std::vector<shared_future<std::vector<double>>>& next = U[(t + 1) % 2];

    heat_timestep(next, curr);
}
```

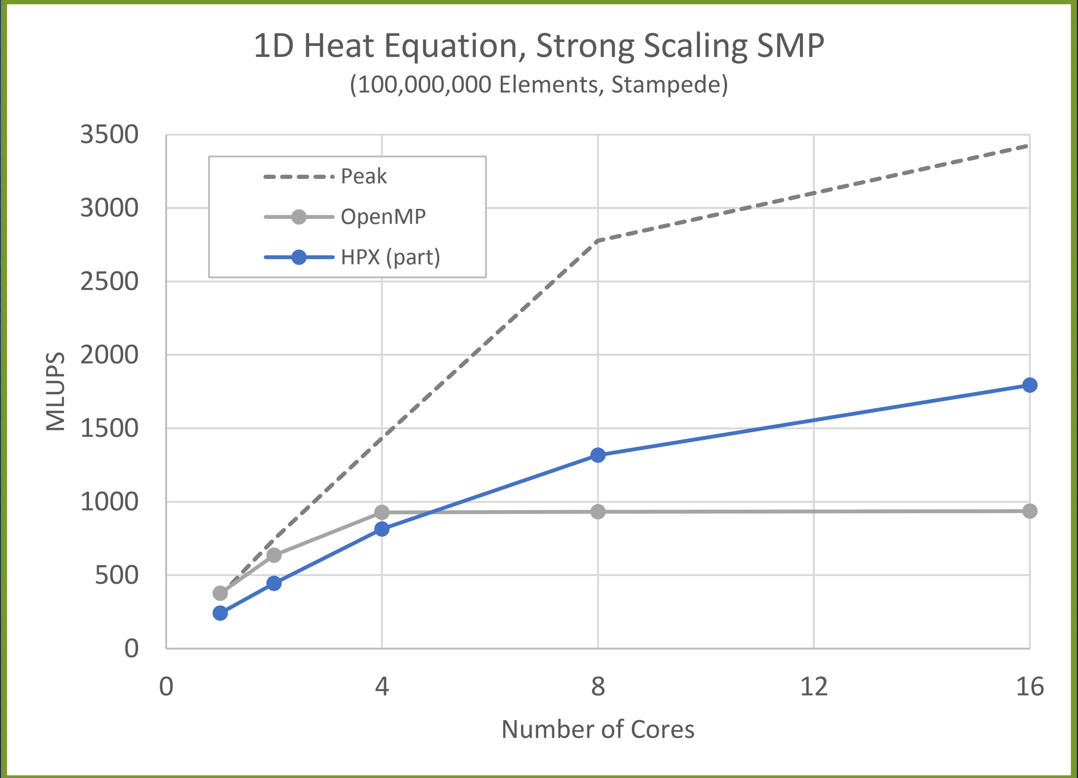
1D HEAT EQUATION, FUTURIZED

- One time step, periodic boundary conditions:

```
void heat_timestep(
    std::vector<shared_future<std::vector<double>>>& next,
    std::vector<shared_future<std::vector<double>>> const& curr)
{
    for (std::size_t i = 0; i != np; ++i) {
        next[i] = dataflow(unwrapped(heat_partition),
            current[idx(i-1, np)], current[i], current[idx(i+1, np)]);
    }
}
```

1D HEAT EQUATION, RESULTS





STE||AR GROUP

CONCLUSIONS

- Asynchronous computing is fun
  - And a possible approach to solve massive parallelization problems
- C++11/14 (and proposals) cover large amount of necessary interfaces
  - However more fine grain parallelism necessary to take full advantage
- One possible option would be to use HPX as a runtime environment
  - HPX also implements a couple of extensions which have proven to be beneficial

LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP

http://stellar-goup.org