Parallelizing the C++ Standard Template Library

Grant Mercer (gmercer015@gmail.com)
Daniel Bourgeois (dcbourg@gmail.com)

CppCon2015
About Grant Mercer

- Third year student at UNLV, computer science major
- Recent work with the STE||AR research group
- Primarily worked on C++ Standards Proposal N4505 inside of HPX
- N4505 is a technical specification for extensions for parallelism
About Daniel Bourgeois

- Fourth year student at LSU, mathematics major
- Currently works with the STE||AR Research group
- Primarily worked on C++ Standards proposals N4505 and N4406 inside of HPX
Background Information

- STE||AR is about shaping a scalable future with a new approach to parallel computation

- Most notable ongoing project by STE||AR is HPX: A general purpose C++ runtime system for parallel and distributed applications of any scale
HPX

- HPX enables programmers to write fully asynchronous code using hundreds of millions of threads

- First open source implementation of the ParallelX execution model
  - Starvation
  - Latencies
  - Overhead
  - Waiting
Focus Points

● Reasons we should parallelize the STL
● Features these algorithms should offer
● Our experience at HPX
● Benchmarking
● Future work
So Why Parallelize the STL?

- Multiple cores are here to stay, parallel programming is becoming more and more important.
  - Amping up processor speed only gives so much. Memory lag, RC delay and Power are all reasons why increasing the processor speed is not the answer

- Scalable performance gains, user flexibility

- Build widespread existing practice for parallelism in the C++ standard algorithms library
Moores law will eventually slow down
Parallelism is growing!
Standards Proposal N4505

- A technical specification for C++ extensions for parallelism, or implementation details for a parallel STL
- Not all algorithms can be parallelized (e.g. std::accumulate), so N4505 defines a list of algorithms to be reimplemented
# Proposed Algorithms

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjacent_difference</td>
<td>adjacent_find</td>
<td>all_of</td>
<td>any_of</td>
</tr>
<tr>
<td>copy</td>
<td>copy_if</td>
<td>copy_n</td>
<td>count_of</td>
</tr>
<tr>
<td>count_if</td>
<td>equal</td>
<td>exclusive_scan</td>
<td>count_fill</td>
</tr>
<tr>
<td>fill_n</td>
<td>find</td>
<td>find_end</td>
<td>find_first_of</td>
</tr>
<tr>
<td>find_if</td>
<td>find_if_not</td>
<td>for_each</td>
<td>for_each_n</td>
</tr>
<tr>
<td>generate</td>
<td>generate_n</td>
<td>includes</td>
<td>inclusive_scan</td>
</tr>
<tr>
<td>inner_product</td>
<td>inplace_merge</td>
<td>is_heap</td>
<td>is_heap_until</td>
</tr>
<tr>
<td>is_partitioned</td>
<td>is_sorted</td>
<td>is_sorted_until</td>
<td>lexicographical_compare</td>
</tr>
<tr>
<td>max_element</td>
<td>merge</td>
<td>min_element</td>
<td>minmax_element</td>
</tr>
<tr>
<td>mismatch</td>
<td>move</td>
<td>none_of</td>
<td>nth_element</td>
</tr>
<tr>
<td>partial_sort</td>
<td>partial_sort_copy</td>
<td>partition</td>
<td>partition_copy</td>
</tr>
<tr>
<td>reduce</td>
<td>remove</td>
<td>remove_copy</td>
<td>remove_copy</td>
</tr>
<tr>
<td>remove_if</td>
<td>replace</td>
<td>replace_copy</td>
<td>replace_copy</td>
</tr>
<tr>
<td>replace_if</td>
<td>reverse</td>
<td>reverse_copy</td>
<td>replace_copy_if</td>
</tr>
<tr>
<td>rotate_copy</td>
<td>search</td>
<td>search_n</td>
<td>rotate</td>
</tr>
<tr>
<td>set_intersection</td>
<td>set_symmetric_difference</td>
<td>set_union</td>
<td>set_difference</td>
</tr>
<tr>
<td>stable_partition</td>
<td>stable_sort</td>
<td>swap_ranges</td>
<td>sort</td>
</tr>
<tr>
<td>transform_exclusive_scan</td>
<td>transform_inclusive_scan</td>
<td>transform_reduce</td>
<td>transform</td>
</tr>
<tr>
<td>uninitialized_copy_n</td>
<td>uninitialized_fill</td>
<td>uninitialized_fill_n</td>
<td>unique</td>
</tr>
<tr>
<td>unique_copy</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Aimed for acceptance into C++17

- Implementation at HPX takes advantage of C++11
- Components of TS will lie in `std::parallel::experimental::v1`. Once standardized, they are expected to be placed in `std`
- HPX implementation lies in `hpx::parallel`
- All algorithms will conform to their predecessors, no new requirements will be placed on the functions

```
template< class ForwardIt1, class ForwardIt2 >
ForwardIt1 search( ForwardIt1 first, ForwardIt1 last, ForwardIt2 s_first, ForwardIt2 s_last );

template< class ForwardIt1, class ForwardIt2, class BinaryPredicate >
ForwardIt1 search( ForwardIt1 first, ForwardIt1 last, ForwardIt2 s_first, ForwardIt2 s_last, 
    BinaryPredicate p );
```
Inside N4505: Execution Policies

- An object of an execution policy type indicates the kinds of parallelism allowed in the execution of the algorithm and express the consequent requirements on the element access functions
- Officially supports `seq, par, par_vec`
std::vector<int> v = ...

// standard sequential sort
std::sort(v.begin(), v.end());

using namespace hpx::parallel;

// explicitly sequential sort
sort(seq, v.begin(), v.end());

// permitting parallel execution
sort(par, v.begin(), v.end());

// permitting vectorization as well
sort(par_vec, v.begin(), v.end());

// sort with dynamically-selected execution
size_t threshold = ...
execution_policy exec = seq;
if (v.size() > threshold)
{
    exec = par;
}

sort(exec, v.begin(), v.end());
• Par: It is the caller’s responsibility to ensure correctness
• Data races and deadlocks are the caller’s job to prevent, the algorithm will not do this for you
• Example of what not to do (data race)

```cpp
using namespace hpx::parallel;

int a[] = {0,1};
std::vector<int> v;

for_each(par, std::begin(a), std::end(a), [&](int i) {
    v.push_back(i*2+1);
});
```
More about parallel execution policies

- Just because you type par, doesn’t mean you’re guaranteed parallel execution due to iterator requirements
- You are permitting the algorithm to execute in parallel, not forcing it
- For example, calling copy with input iterators and a par tag will execute sequentially. Input iterators cannot be parallelized!
Exception reporting behavior

- If temporary resources are required and none are available, throws `std::bad_alloc`
- If the invocation of the element access function terminates with an uncaught exception for `par, seq`: all uncaught exceptions will be contained in an `exception_list`
Task execution policy for HPX

- The task policy was added by us at HPX to give users a choice of when to join threads back into the main program. Returns and `hpx::future` of the result.

```cpp
// permitting parallel execution
auto f =
    sort(par(task), v.begin(), v.end());
...
f.wait();
```
User Interaction with the Algorithms

- Restrictions of execution
- Runtime decision making
- Where work is executed
- Size of work to be executed
- Abstractions usable for the parallel algorithms and elsewhere

```cpp
// sort with dynamically-selected execution
size_t threshold = ...
execution_policy exec = seq;
if (v.size() > threshold)
{
    exec = par;
}
for_each(exec, v.begin(), v.end());
```
Inside N4406: Parallel Algorithms Need Executors

- Let the programmer specify where work is executed
- Attach to parallel algorithms
Extending On Execution Policies

- The .on syntax to attach to parallel algorithms
- Not all combinations of policies and executors should be allowed

```c
// should compile, done in parallel
for_each(par.on(parallel_executor()), f, l, &F)

// should compile, but not done in parallel
for_each(par.on(sequential_executor()), f, l, &F)

// This does not make sense thus should not compile!
for_each(seq.on(parallel_executor()), f, l, &F)
```
But how, N4406? The requirements to be met...

- Execution policies should accept an executor
- An executor should advertise restrictions
- uniform API for parallel algorithms
Executor Traits for N4406

- Can be called with objects that meet the requirements of an executor
- Executor_traits provides four main function calls
  - async_execute - asynchronously calls a function once
  - async_execute - asynchronously calls a function more than once
  - execute - calls a function once
  - execute - calls a function more than once
Executor Traits for N4406: Example

// Some Definitions

some_executor_type exec;
some_shape_type inputs;

auto f1 = []((){ /*..compute..*/ return t_1; });
auto f2 = [](T t_a){ /*..compute..*/ return t_2; };

typedef executor_traits<some_executor_type> traits;
Executor Traits for N4406: Example

// Calls f1, returns a future containing the result of f1
future<T> myfut1 = traits::async_execute(exec, f1);

// Calls f2 for each of the inputs,
// returns a future indicating the completion of all of the calls
future<void> myfut2 = traits::async_execute(exec, f2, inputs);

// Calls f1, returns the result
T myval1 = traits::execute(exec, f1);

// Calls f2 and returns once all calls are completed
traits::execute(exec, f2, inputs);
HPX and N4406: Yes and Not Quite

Yes
- algorithms can be extended with the .on syntax
- executor_traits provides a convenient, uniform launch mechanism
- easy to define an object meeting executor requirements
- work can be executed in bulk quantities
HPX and N4406: Yes and Not Quite

Not Quite

- Want to minimize waiting

```cpp
future<void> myfut = N4406_traits::async_execute(exec, f2, inputs);
// Has to wait for all functions to finish before my_next_function gets called
myfut2.then(my_next_function);
```

- The HPX solution

```cpp
std::vector<future<T>> myfuts = HPX_traits::async_execute(exec, f2, inputs);
// my_other_next_funcion can be called once each element in myfuts is ready
when_each(my_other_next_function, myfuts);
```
Executor Traits for HPX

template <typename Executor> // requires is_executor<Executor>
struct executor_traits
{
    using Executor = executor_type;

    using execution_category = /* category of Executor */;

    template <typename T>
    using future = /* future type of Executor or hpx::future<T> */;

    // ... apply_execute, async_execute and execute implementation
};
Additional Traits

- **executor_information_traits**
  - retrieve number of processing units
  - test if pending closures exist

- **timed_executor_traits**
  - inherits from executor_traits
  - at and after functions
Parallel executor

```c++
struct parallel_executor : executor_tag {
  explicit parallel_executor(BOOST_SCOPED_ENUM(launch) l = launch::async) : l_(l) {}

  template <typename F>
  hpx::future<typename hpx::util::result_of<typename hpx::util::decay<F>::type()>>
  async_execute(F && f) {
    return hpx::async(l_, std::forward<F>(f));
  }

private:
  /* . . . */
};
```
Sequence of Execution

- Primer on work stealing, N3872

```c
e();
spawn f();
g();
sync;
h();

for(int i=0; i<n; ++i)
    spawn f(i);
sync;
```
Types of Executors in HPX

- **standard executors**
  - parallel, sequential
- **this thread executors**
  - static queue, static priority queue
- **thread pool executors, and thread pool os executors**
  - local queue, local priority queue
  - static queue, static priority queue
- **service executors**
  - io pool, parcel pool, timer pool, main pool
- **distribution policy executor**
Taking a Step Back

- Executors provide a mechanism for launching work
- Flexible decision making
- Need a general mechanism for grain size control
Executor Parameters

- grain size control
- passing information to the partitioner
- Similar to OpenMP Dynamic, Static, Guided
Extending with Execution Policies

- The .with syntax to extend parallel algorithms

```cpp
auto par_auto = par.with(auto_chunk_size()); // equivalent to par
auto par_static = par.with(static_chunk_size());
auto my_policy = par.with(my_exec).on(my_chunk_size);
auto my_task_policy = my_policy(task);`
The Concepts for Execution Policies

<table>
<thead>
<tr>
<th>Property</th>
<th>C++ Concept Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution restrictions</td>
<td>execution_policy</td>
</tr>
<tr>
<td>Sequence of execution</td>
<td>executor</td>
</tr>
<tr>
<td>Where execution happens</td>
<td>executor</td>
</tr>
<tr>
<td>Grain size of work items</td>
<td>executor_parameter</td>
</tr>
</tbody>
</table>
Initial Parallel Design: Partitioning

- All algorithms given by the proposal are passed a range, which must be partitioned and executed in parallel.
- There are a couple different types of partitioners we implemented at HPX.
foreach_partitioner

- The simplest of partitioners, splits a set of data into equal partitions and invokes a passed function on each subset of the data.
- Mainly used in algorithms such as `foreach, fill` where each element is independent and not part of any bigger picture.
\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}

\{1, 2, 3, 4\} \quad | \quad \{5, 6, 7, 8\} \quad | \quad \{9, 10, 11, 12\}

f([1,2,3,4]) \quad | \quad f([5,6,7,8]) \quad | \quad f([9, 10, 11, 12])
for_each_n

template<typename ExPolicy, typename F>
static typename detail::algorithm_result<ExPolicy, Iter>::type
parallel(ExPolicy const& policy, Iter first, std::size_t count, F && f)
{
    if(count != 0)
    {
        return util::foreach_n_partitioner<ExPolicy>::call(policy, first, count,
            [f](Iter part_begin, std::size_t part_size)
            {
                util::loop_n(part_begin, part_size, [&f](Iter const& curr)
                {
                    f(*curr);
                });
            });
    }
    return detail::algorithm_result<ExPolicy, Iter>::get( std::move(first));
}
partitioner

- Similar to foreach, but the result of the invocation of the function on each subset is stored in a vector and an additional function is invoked and passed that vector.
- Useful in a majority of algorithms *copy, find, search, etc*...
\[
\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}
\]

\[
\{1, 2, 3, 4\} \quad | \quad \{5, 6, 7, 8\} \quad | \quad \{9, 10, 11, 12\}
\]

\[
v = \{f([1, 2, 3, 4]) , f([5, 6, 7, 8]) , f([9, 10, 11, 12])\}
\]

\[
g(v)
\]
reduce

template <typename ExPolicy, typename FwdIter, typename T_, typename Reduce>
static typename detail::algorithm_result<ExPolicy, T>::type
parallel(ExPolicy, const& policy, FwdIter first, FwdIter last, T_ && init, Reduce && r)
{
  // check if first == last, return initial value if true

  return util::partitioner<ExPolicy, T>::call( policy,
    first, std::distance(first, last),
    [r](FwdIter part_begin, std::size_t part_size) -> T
    {
      T val = *part_begin;
      return util::accumulate_n(++part_begin, --part_size,
        std::move(val), r);
    },
    hpx::util::unwrapped([init, r](std::vector<T> && results)
    {
      return util::accumulate_n(boost::begin(results),
        boost::size(results), init, r);
    }));
}
parallel vector dot product

- No intermediate function, forces us to use a tuple instead of a simple double
- Reduce requirements can not be worked around, a new function is needed

```cpp
int xvalues[] = //...
int yvalues[] = //...

double result =
    std::accumulate(
        make_zip_iterator(std::begin(xvalues), std::being(yvalues)),
        make_zip_iterator(std::end(xvalues), std::end(yvalues)),
        0.0,
        [](double result, reference it) {
            return result + get<0>(it) + get<1>(it)
        });
```
parallel vector dot product

tuple<double, double> result =
  hpx::parallel::reduce(hpx::parallel::par,
    make_zip_iterator(boost::begin(xvalues), boost::begin(yvalues)),
    make_zip_iterator(boost::end(xvalues), boost::end(yvalues)),
    hpx::util::make_tuple(0.0, 0.0),
    [](tuple<double, double> res, reference it) {
      return hpx::util::make_tuple(
        get<0>(res) + get<0>(it) * get<1>(it),
        1.0);
    });

- N4505 is the newest revision to include \textit{transform\_reduce}, as proposed by N4167
- Without \textit{transform\_reduce} the solution is horribly hacky
transform_reduce

template <typename ExPolicy, typename FwdIter, typename T_, typename Reduce, //...
static typename detail::algorithm_result<ExPolicy, T>::type
parallel(ExPolicy const& policy, FwdIter first, FwdIter last, T_ && init, Reduce && r, Convert && conv)
{
    typedef typename std::iterator_traits<FwdIter>::reference reference;
    return util::partitioner<ExPolicy, T>::call(policy, first,
        std::distance(first, last),
        [r, conv](FwdIter part_begin, std::size_t part_size) -> T
        {
            T val = conv(*part_begin);
            return util::accumulate(++part_begin, --partsize, std::move(val),
                [r, &conv](T const& res, reference next)
                {
                    return r(res, conv(next));
                });
        },
        hpx::util::unwrapped([&init, r](std::vector<T> && results)
        {
            return util::accumulate_n(boost::begin(results),
                boost::size(results) init, r);
        }));
}
simplified dot product

int hpx_main()
{
    std::vector<double> xvalues(10007);
    std::vector<double> yvalues(10007);

    using ...;

    double result =
        hpx::parallel::transform_reduce(hpx::parallel::par,
            make_zip_iterator(boost::begin(xvalues), boost::begin(yvalues)),
            make_zip_iterator(boost::end(xvalues), boost::end(yvalues)),
            0.0,
            std::plus<double>(),
            [](tuple<double, double> r)
            {
                return get<0>(r) * get<1>(r);
            }
        );

    hpx::cout << result << hpx::endl;
    return hpx::finalize();
}
scan_partitioner

- The scan partitioner has 3 steps
  - Partition the data and invoke the first function
  - Invoke a second function as soon as the current and left partition are ready
  - Invoke a third function on the resultant vector of step 2

- Specific cases such as `copy_if`, `inclusive/exclusive_scan`
\[
\begin{align*}
\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\} \\
\{1, 2, 3, 4\} & \quad \{5, 6, 7, 8\} & \{9, 10, 11, 12\} \\
\{1, 2, 3, 4\} \\
v_0 = \{\{f[1,2,3,4]\} \quad \{f[5,6,7,8]\} \quad f[9,10,11,12]\} \\
r = \{g(v_0) \quad g(v_1)\} \\
h(r)
\end{align*}
\]
copy_if

int lst[] = {1, 1, 1, 1, 2, 2, 1, 1, 2, 2, 1};

int res[8];

hpx::parallel::copy_if(par, boost::begin(lst), boost::end(lst), boost::begin(res),
    [](int i){ return i == 1; });

1 1 1 1 2 2 1 1 2 2 1
1 1 1 1 1 1 1 1

- Not just as simple as copying what returns true, the resultant arrays need's to be squashed
copy_if

typedef util::scan_partitioner<ExPolicy, Iter, std::size_t> scan_partitioner_type;
return scan_partitioner_type::call(
    policy, hpx::util::make_zip_iterator(first, flags.get()),
    count, init,
    [f](zip_iterator part_begin, std::size_t part_size) -> std::size_t
    {
        // flag any elements to be copied
    },
    hpx::until::unwrapped( [](std::size_t const& prev, std::size_t const& curr)
    {
        // determine distance to advance dest iter for each partition
        return prev + curr;
    } ),
    [=](std::vector<hpx::shared_future<std::size_t> > && r,
        std::vector<std::size_t> const& chunk_sizes) mutable -> result_type
    {
        // copy element to dest in parallel;
    } );
Designing Parallel Algorithms

- Some algorithms are easy to implement, other … not so much
- Start simple, work up the grape vine towards more difficult algorithms
- Concepts from simple algorithms can be brought into more difficult and complex solutions
fill_n

- fill_n can be implemented in two lines using for_each_n

```cpp
template <typename ExPolicy, typename T>
static typename detail::algorithm_result<ExPolicy, OutIter>::type
parallel(ExPolicy const& policy, OutIter first, std::size_t count, T const& val)
{
    typedef typename std::iterator_traits<OutIter>::value_type type;

    return for_each_n<OutIter>().call(
        policy, boost::mpl::false_(), first, count,
        [val](type& v) {
            v = val;
        });
}
```
## Completed algorithms as of today

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Algorithm</th>
<th>Algorithm</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjacent_difference</td>
<td>adjacent_find</td>
<td>all_of</td>
<td>any_of</td>
</tr>
<tr>
<td>copy</td>
<td>copy_if</td>
<td>copy_n</td>
<td>count</td>
</tr>
<tr>
<td>count_if</td>
<td>equal</td>
<td>exclusive_scan</td>
<td>fill</td>
</tr>
<tr>
<td>fill_n</td>
<td>find</td>
<td>find_end</td>
<td>find_first_of</td>
</tr>
<tr>
<td>find_if</td>
<td>find_if_not</td>
<td>for_each</td>
<td>for_each_n</td>
</tr>
<tr>
<td>generate</td>
<td>generate_n</td>
<td>includes</td>
<td>inclusive_scan</td>
</tr>
<tr>
<td>inner_product</td>
<td>inplace_merge</td>
<td>is_heap</td>
<td>is_heap_until</td>
</tr>
<tr>
<td>is_partitioned</td>
<td>is_sorted</td>
<td>is_sorted_until</td>
<td>lexicographical_compare</td>
</tr>
<tr>
<td>max_element</td>
<td>merge</td>
<td>min_element</td>
<td>minmax_element</td>
</tr>
<tr>
<td>mismatch</td>
<td>move</td>
<td>none_of</td>
<td>nth_element</td>
</tr>
<tr>
<td>partial_sort</td>
<td>partial_sort_copy</td>
<td>partition</td>
<td>partition_copy</td>
</tr>
<tr>
<td>reduce</td>
<td>remove</td>
<td>remove_copy</td>
<td>remove_copy_if</td>
</tr>
<tr>
<td>remove_if</td>
<td>replace</td>
<td>replace_copy</td>
<td>replace_copy_if</td>
</tr>
<tr>
<td>replace_if</td>
<td>reverse</td>
<td>reverse_copy</td>
<td>rotate</td>
</tr>
<tr>
<td>rotate_copy</td>
<td>search</td>
<td>search_n</td>
<td>set_difference</td>
</tr>
<tr>
<td>set_intersection</td>
<td>set_symmetric_difference</td>
<td>set_union</td>
<td>sort</td>
</tr>
<tr>
<td>stable_partition</td>
<td>stable_sort</td>
<td>swap_ranges</td>
<td>transform</td>
</tr>
<tr>
<td>transform_exclusive_scan</td>
<td>transform_inclusive_scan</td>
<td>transform_reduce</td>
<td>uninitialized_copy</td>
</tr>
<tr>
<td>uninitialized_copy_n</td>
<td>uninitialized_fill</td>
<td>uninitialized_fill_n</td>
<td>unique</td>
</tr>
</tbody>
</table>
void measure_parallel_foreach(std::size_t size)
{
    std::vector<std::size_t> data_representation(size);
    std::iota(boost::begin(data_representation),
              boost::end(data_representation),
              std::rand());

    // create executor parameters object
    hpx::parallel::static_chunk_size cs(chunk_size);

    // invoke parallel for_each
    hpx::parallel::for_each(hpx::parallel::par.with(cs),
                            boost::begin(data_representation),
                            boost::end(data_representation),
                            [](std::size_t) {
                                worker_timed(delay);
                            });
}

boost::uint64_t average_out_parallel(std::size_t vector_size)
{
    boost::uint64_t start = hpx::util::high_resolution_clock::now();

    // average out 100 executions to avoid varying results
    for(auto i = 0; i < test_count; i++)
    {measure_parallel_foreach(vector_size);
    }

    return (hpx::util::high_resolution_clock::now() - start) / test_count;
}
Benchmarking

- Comparing seq, par, task execution policies
- Task is special in that executions can be written to overlap
- User can wait to join execution after multiple have been sent off
Getting the most out of performance

- The big question is whether these functions actually offer a gain in performance when used.
- Grain size: amount of work executed per thread.
- In order to test this we look to simulate the typical *strong scaling* graph:
# Hardware Used

<table>
<thead>
<tr>
<th>Classification</th>
<th>Name</th>
<th>Wedge</th>
<th>Deneb</th>
<th>Tycho</th>
<th>Trillian</th>
<th>Lyra</th>
<th>Sheliak</th>
<th>Aiel</th>
<th>Marvin</th>
<th>Beowulf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Role</td>
<td>Head + I/O</td>
<td></td>
<td>Development</td>
<td>GPGPU development</td>
<td>Fat compute</td>
<td>GPGPU/Fat compute</td>
<td>Fat compute</td>
<td>Fast compute</td>
<td>Thin compute</td>
<td>Thin compute</td>
</tr>
<tr>
<td># of Nodes</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>OEM</td>
<td>Dell</td>
<td>HP</td>
<td>Supermicro</td>
<td>Dell</td>
<td>HP</td>
<td>Sun</td>
<td>Dell</td>
<td>HP</td>
<td>Dell</td>
<td>HP</td>
</tr>
<tr>
<td>System</td>
<td>Model</td>
<td>PowerEdge R720xd 12G</td>
<td>z800</td>
<td>PowerEdge R815 11G</td>
<td>2.1</td>
<td>PowerEdge R802 12G</td>
<td>PowerEdge M520 12G</td>
<td>PowerEdge M620 12G</td>
<td>PowerEdge M620 12G</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IDM</td>
<td>Intel</td>
<td>Intel</td>
<td>Intel</td>
<td>AMD</td>
<td>AMD</td>
<td>AMD</td>
<td>Intel</td>
<td>Intel</td>
<td>Intel</td>
</tr>
<tr>
<td></td>
<td>CPU(s)</td>
<td>Xeon E5-2670</td>
<td>Xeon E5649</td>
<td>Xeon E5620</td>
<td>Opteron 6272</td>
<td>Opteron 8431</td>
<td>Opteron 8384</td>
<td>Intel E5-2690</td>
<td>Intel E5-2450</td>
<td>Xeon X3430</td>
</tr>
<tr>
<td></td>
<td>Frequency [GHz]</td>
<td>2.5</td>
<td>2.4</td>
<td>2.4</td>
<td>2.1</td>
<td>2.4</td>
<td>2.7</td>
<td>2.9</td>
<td>2.1</td>
<td>2.4</td>
</tr>
<tr>
<td></td>
<td># of CPUs</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td># of Cores</td>
<td>12</td>
<td>12</td>
<td>8</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Main Memory</td>
<td>Type</td>
<td>Registered</td>
<td>Unregistered</td>
<td>???</td>
<td>Registered</td>
<td>Registered</td>
<td>???</td>
<td>Unregistered</td>
<td>Registered</td>
<td>Registered</td>
</tr>
<tr>
<td></td>
<td>Form Factor</td>
<td>DDR3</td>
<td>DDR3</td>
<td>DDR3</td>
<td>DDR3</td>
<td>DDR2</td>
<td>DDR2</td>
<td>DDR3</td>
<td>DDR3</td>
<td>DDR3</td>
</tr>
<tr>
<td></td>
<td>Speed [MT/s]</td>
<td>1600</td>
<td>1333</td>
<td>1333</td>
<td>1333</td>
<td>533</td>
<td>333</td>
<td>1333</td>
<td>1333</td>
<td>1333</td>
</tr>
<tr>
<td></td>
<td># DIMMs</td>
<td>16</td>
<td>8</td>
<td>6</td>
<td>32</td>
<td>48</td>
<td>32</td>
<td>32</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>RAM [GB]</td>
<td>128</td>
<td>32</td>
<td>24</td>
<td>128</td>
<td>96</td>
<td>64</td>
<td>32</td>
<td>48</td>
<td>12</td>
</tr>
<tr>
<td>Storage</td>
<td>Controller</td>
<td>Dell PERC H710</td>
<td>LSI SAS1068E</td>
<td>Intel 82801JI ICH</td>
<td>Dell PERC H200</td>
<td>HP Smart Array P400i</td>
<td>???</td>
<td>Dell PERC H310</td>
<td>Dell PERC S110</td>
<td>Intel BD3400 PCH</td>
</tr>
<tr>
<td></td>
<td>Bus</td>
<td>???</td>
<td>???</td>
<td>???</td>
<td>???</td>
<td>SAS1/SATA1</td>
<td>???</td>
<td>???</td>
<td>SATA-2</td>
<td>???</td>
</tr>
<tr>
<td></td>
<td>Frequency [RPM]</td>
<td>10000</td>
<td>7200</td>
<td>7200</td>
<td>7200</td>
<td>10000</td>
<td>???</td>
<td>7200</td>
<td>7200</td>
<td>7200</td>
</tr>
<tr>
<td></td>
<td># of Disk Drives</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>???</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Storage [TB]</td>
<td>3</td>
<td>1.5</td>
<td>0.32</td>
<td>1</td>
<td>0.3</td>
<td>???</td>
<td>1</td>
<td>1</td>
<td>0.25</td>
</tr>
<tr>
<td>Network</td>
<td># of GigE ports</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td># of QDR IB ports</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Max Load [W]</td>
<td>750</td>
<td>???</td>
<td>???</td>
<td>1100</td>
<td>???</td>
<td>???</td>
<td>750</td>
<td>N/A</td>
<td>???</td>
</tr>
<tr>
<td>Out-of-band management</td>
<td>iDRAC7 Express</td>
<td>N/A</td>
<td>???</td>
<td>???</td>
<td>iDRAC6 Express</td>
<td>???</td>
<td>???</td>
<td>iDRAC7 Express</td>
<td>iDRAC7 Express</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Sequential vs. Parallel

- 500 nanosecond delay per iteration
- Vector size of 10,000
Sequential vs. Parallel

- 1000 nanosecond delay per iteration
- Vector size of 100,000
Parallel vs. Task

- 1000 nanosecond delay per iteration
- Vector size of 1,000,000
HPXCL: OpenCL backend

- Uses `hpx::parallel::for_each`
  - Grouping work-items into work packets

```cpp
hpx::parallel::for_each(hpx::parallel::par,
    nd_range_iterator::begin(dim_x, dim_y, dim_z),
    nd_range_iterator::end(dim_x, dim_y, dim_z),
    [&ta](nd_pos const& gid)
    {
      workgroup_thread(&ta, gid);
    });
```
Future Work

- Not all of the algorithms are implemented
- Perform more benchmarking on different algorithms
- Grain size control and non-partitioned algorithms
- Experiment with custom policies
  - if_gpu_then.on(numa).with(chunker)
- Introspection tools (using performance counters to make adjustments)
- Minimization executor (power, idle_rate, other performance counter stuff)
Additional Resources

- HPX - https://github.com/STEllAR-GROUP/hpx
- STE||AR - http://stellar.cct.lsu.edu/