Parallelizing the C++ Standard Template Library

Grant Mercer (Gmercer015@gmail.com)
C++Now 2015

N4352

About me

- Second year student at UNLV, computer science major
- Recent work with STE||AR research group
- Primarily worked on C++ Standards proposal N4352 inside of HPX
- N4352 is a technical specification for extensions for parallelism

Background Information

- HPX : A general purpose C++ runtime system for parallel and distributed applications of any scale

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

Additional Info HPX

- First open source implementation of the ParallelX execution model
  - Starvation
  - Latencies
  - Overhead
  - Waiting
- Enables programmers to write fully asynchronous code using hundreds of millions of threads

Focus Points

- Reasons for a parallel STL
- Our experience at HPX
- Benchmarking
- Pros vs Cons

Why Parallelize the STL?

- Multiple cores are here to stay, parallel programming is becoming more and more important.

- Scalable performance gains, user flexibility

- Build widespread existing practice for parallelism in the C++ standard algorithms library

Standards Proposal N4352

- 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 N4352 defines a list of algorithms to be reimplemented

Proposed algorithms

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
transform_exclusive_scan	transform_inclusive_scan	transform_reduce	uninitialized_copy
uninitialized_copy_n	uninitialized_fill	uninitialized_fill_n	unique
unique_copy			

N4352

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*

N4352

- 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 N4352: 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());
```

N4352

- 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)

```
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);
});
```

N4352

- 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 an hpx::future of the result

```
// permitting parallel execution
auto f =
    sort(par(task), v.begin(), v.end());

...

f.wait()
```

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 used 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} | {5, 6, 7, 8} | {9, 10, 11, 12}

f([1,2,3,4]) | f([5,6,7,8]) | 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));
}
```

N4352

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} | {5, 6, 7, 8} | {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);
        }));
}
```

N4352

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

```
Point result =
    std::experimental::parallel::reduce(
        std::experimental::parallel::par,
        std::begin (values),
        std::end(values),
        Point{0.0, 0.0},
        [](Point res, Point curr)
        {
            return Point{
                res.x * res.y + curr.x * curr.y, 1.0};
        }
    );
```

N4352

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);
        });
```

- N4352 is the newest revision to include *transform_reduce* , as proposed by N4167
- Without *transform_reduce* , the solution was 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)
{

    // check if first == last , return initial value if true

    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_n(++part_begin, --part_size, 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);
        }));
}
```

N4352

simplified dot product

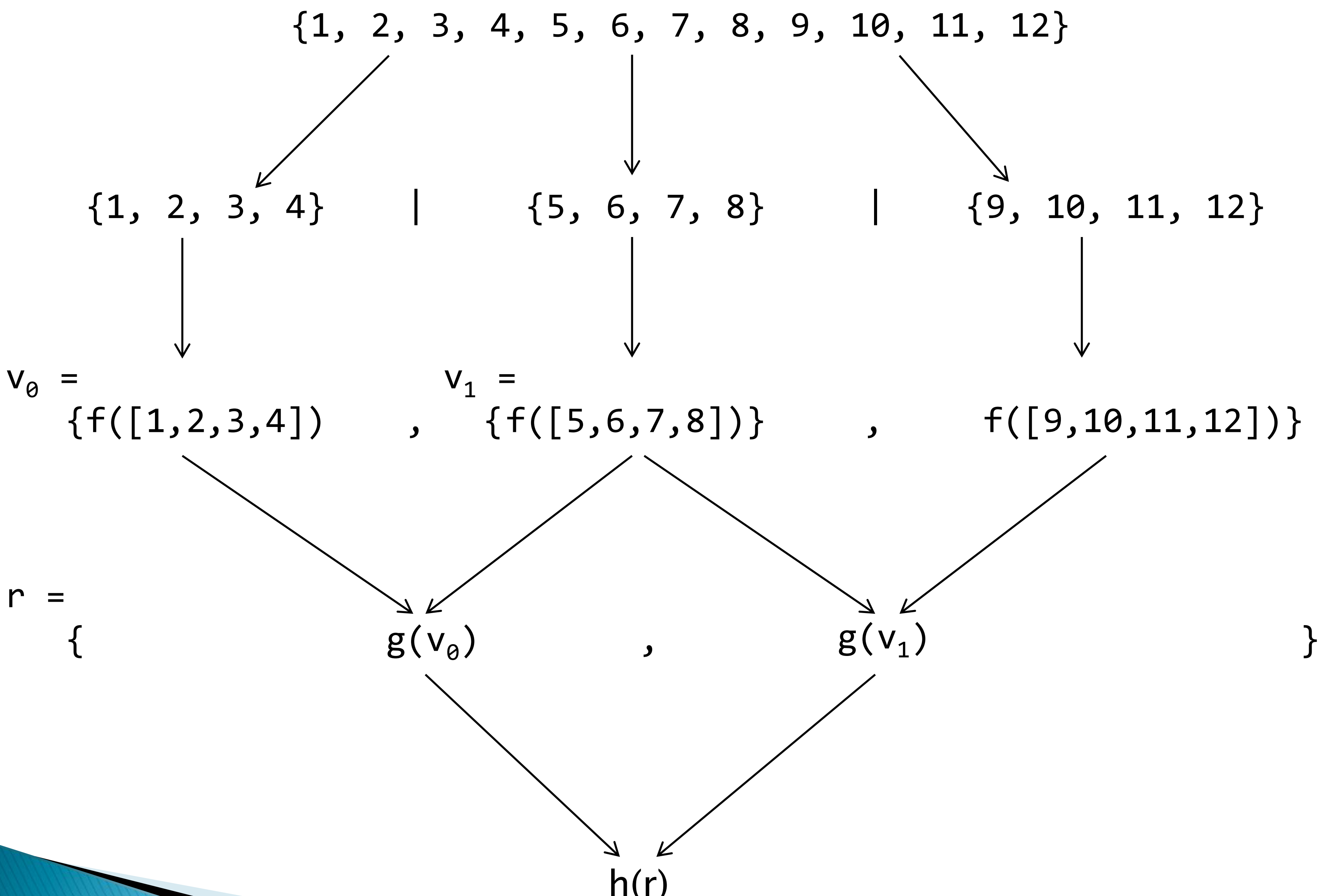
```
int hpx_main()
{
    std::vector<double> xvalues(10007);
    std::vector<double> yvalues(10007);
    std::fill(boost::begin(xvalues), boost::end(xvalues), 1.0);
    std::fill(boost::begin(yvalues), boost::end(yvalues), 1.0);

    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)),
            [](tuple<double, double> r)
            {
                return get<0>(r) * get<1>(r);
            },
            0.0,
            std::plus<double>()
        );

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

N4352

scan_partitioner

- The scan partitioner has 3 steps, the first step partitions the data and invokes the first function. Step two invokes a second function as soon as the current and left partition are ready and lastly a third function is invoked in the resultant vector of step 2.

- Specific cases such as *copy_if, inclusive/exclusive_scan*


{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
{1, 2, 3, 4} | {5, 6, 7, 8} | {9, 10, 11, 12}
v_0 =
v_1 =
{f([1,2,3,4]) , {f([5,6,7,8])} , f([9,10,11,12])}
r =
{ $g(v_0)$, $g(v_1)$ }
h(r)

N4352

copy_if

```
int lst[] = {1, 1, 1, 1, 2, 2, 1, 1, 2, 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 1 2 2 1

1 1 1 1     1 1   1     1
```

- Not just as simple as copying what returns true, the result array is not '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::util::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 the elements into dest in parallel
    }
);
```

Designing Parallel Algorithms

- Some algorithms are easy to implement, others …. 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

- parallel fill_n is now implemented in just two lines total taking advantage of for_each_n

```
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

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~~
transform_exclusive_scan	transform_inclusive_scan	~~transform_reduce~~	~~uninitialized_copy~~
~~uninitialized_copy_n~~	~~uninitialized_fill~~	~~uninitialized_fill_n~~	unique
unique_copy			

N4352

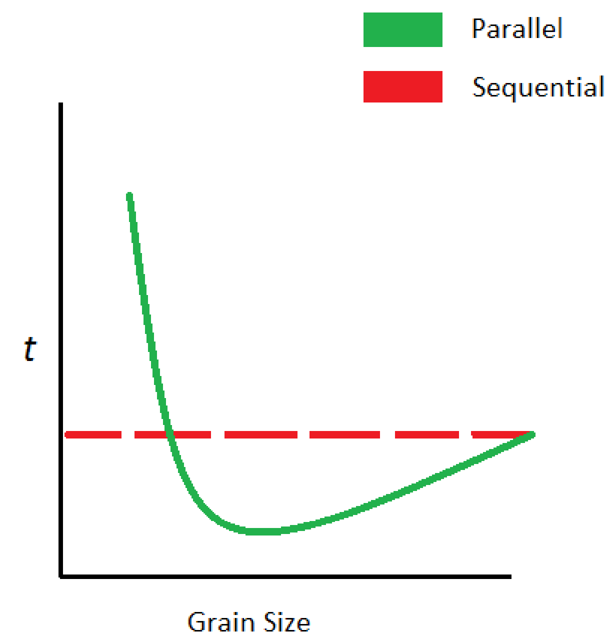
```
void measure_parallel_foreach(std::size_t size)
{
    std::vector<std::size_t> data_representation(size);
    std::iota(boost::begin(data_representation), boost::end(data_representation),
              std::rand());

    // invoke parallel for_each
    hpx::parallel::for_each(hpx::parallel::par(chunk_size),
        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();

    // run test_count times to get an average execution time
    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:

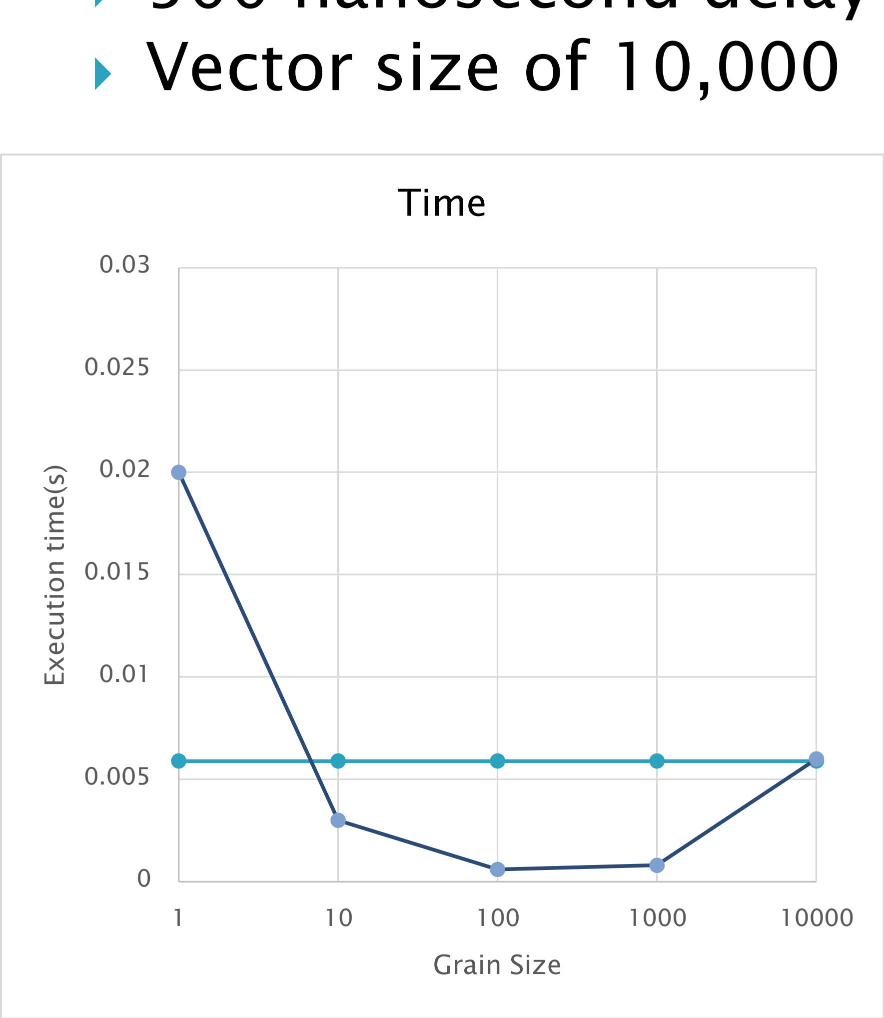


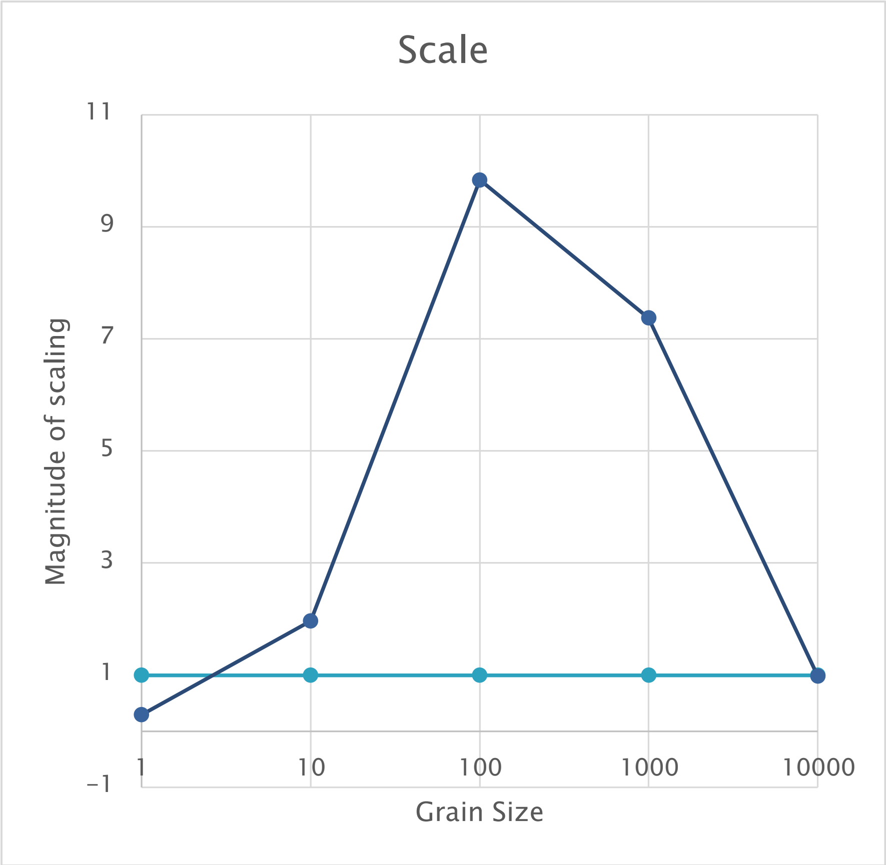
Hardware Used

Classification	Name	Wedge	Deneb	Tycho	Trillian	Lyra	Sheliak	Ariel	Marvin	Beowulf
	Role	Head + I/O	Development	GPGPU development	Fat compute	GPGPU/Fat compute	Fat compute	Fast compute	Thin compute	Thin compute
# of Nodes		1	1	1	2	2	2	2	16	16
System	OEM	Dell	HP	Supermicro	Dell	HP	Sun	Dell	Dell	HP
	Model	PowerEdge R720xd 12G	z800	X8DTG-D	PowerEdge R815 11G	ProLiant DL785 G6	Sun Fire X4600 M2	PowerEdge R620 12G	PowerEdge M520 12G	ProLiant DL120 G6
CPU(s)	IDM	Intel	Intel	Intel	AMD	AMD	AMD	Intel	Intel	Intel
	Model	Xeon E5-2670	Xeon E5649	Xeon E5620	Opteron 6272	Opteron 8431	Opteron 8384	Xeon E5-2690	Xeon E5-2450	Xeon X3430
	Frequency [GHz]	2.6	2.5	2.4	2.1	2.4	2.7	2.9	2.1	2.4
	# of CPUs	2	2	2	4	8	8	2	2	1
	# of Cores	16	12	8	64	48	32	16	16	4
Main Memory	Type	Registered	Unregistered	???	Registered	Registered	???	Unregistered	Registered	Registered
	Form Factor	DDR3	DDR3	DDR3	DDR3	DDR2	DDR2	DDR3	DDR3	DDR3
	Speed [MT/s]	1600	1333	1333	1333	533	333	1333	1333	1333
	# DIMMs	16	8	6	32	48		8	6	4
	RAM [GB]	128	32	24	128	96	64	32	48	12
Storage	Controller	Dell PERC H710	LSI SAS1068E	Intel 82801JI ICH	Dell PERC H200	HP Smart Array P400i	???	Dell PERC H310	Dell PERC S110	Intel BD3400 PCH
	Bus	???	???	???	???	SAS1/SATA1	???	???	???	SATA-2
	Frequency [RPM]	10000	7200	7200	7200	10000	???	7200	7200	7200
	# of Disk Drives	5	1	1	1	1	???	1	1	1
	Storage [TB]	3	1.5	0.32	1	0.3	???	1	1	0.25
Network	# of GigE ports	4	2	2	4	2	4	4	2	2
	# of QDR IB ports	1	0	0	1	1	0	1	1	0
Max Load [W]		750	???	???	1100	???	???	750	N/A	???
Out-of-band management		iDRAC7 Express	N/A	???	iDRAC6 Express	???	???	iDRAC7 Express	iDRAC7 Express	N/A

Sequential vs. Parallel

- 500 nanosecond delay per iteration
- Vector size of 10,000

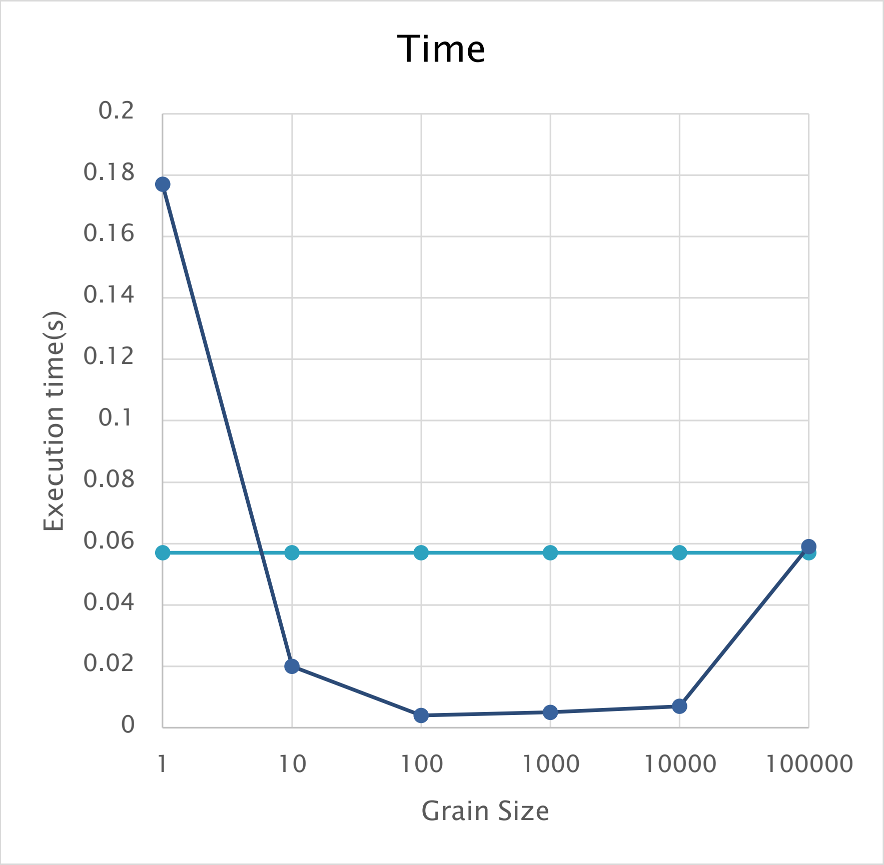


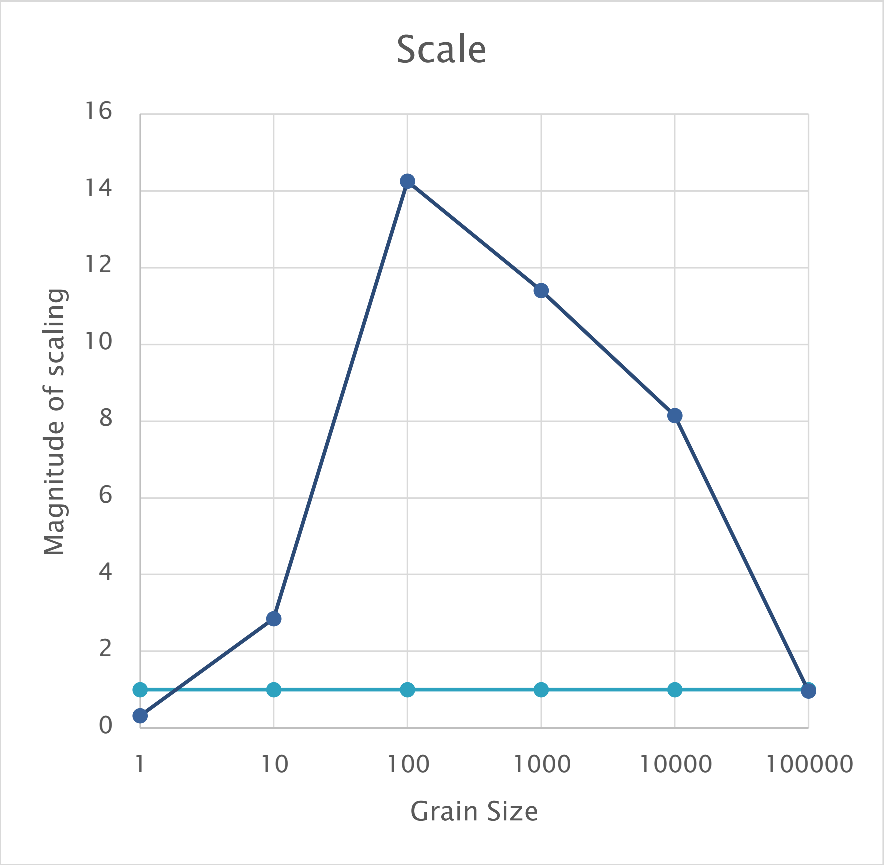


Sequential vs. Parallel

- 1000 nanosecond delay per iteration
- Vector size of 100,000

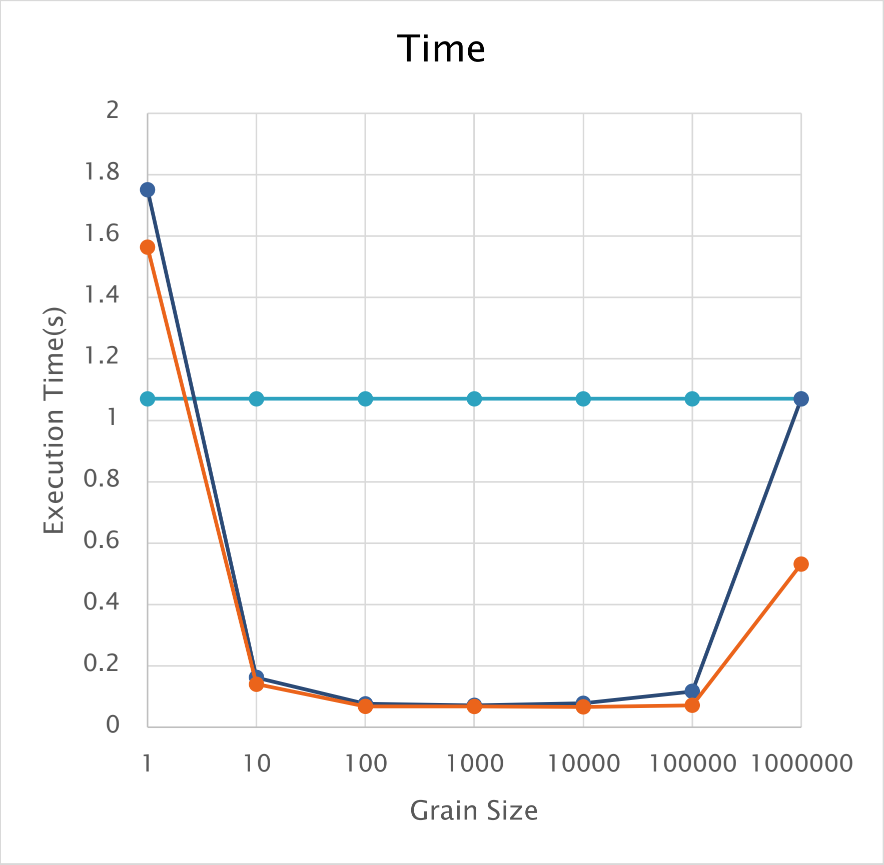
Sequenti al
Parallel

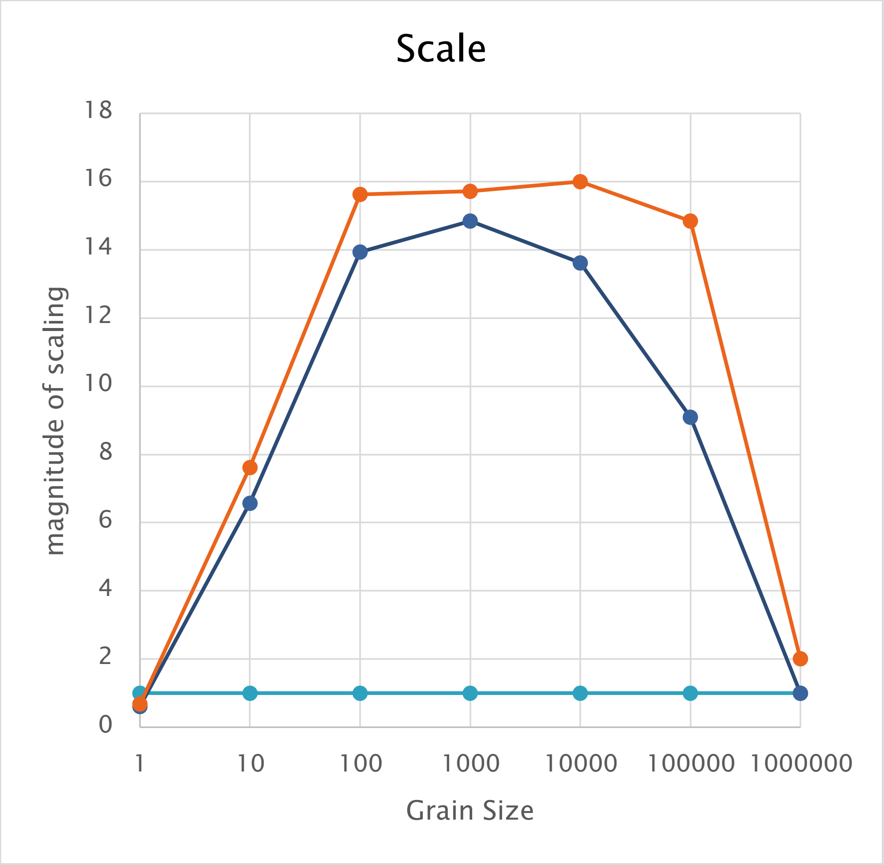




Parallel vs. Task

- 1000 nanosecond delay per iteration
- Vector size of 1,000,000





N4352

HPXCL: OpenCL backend

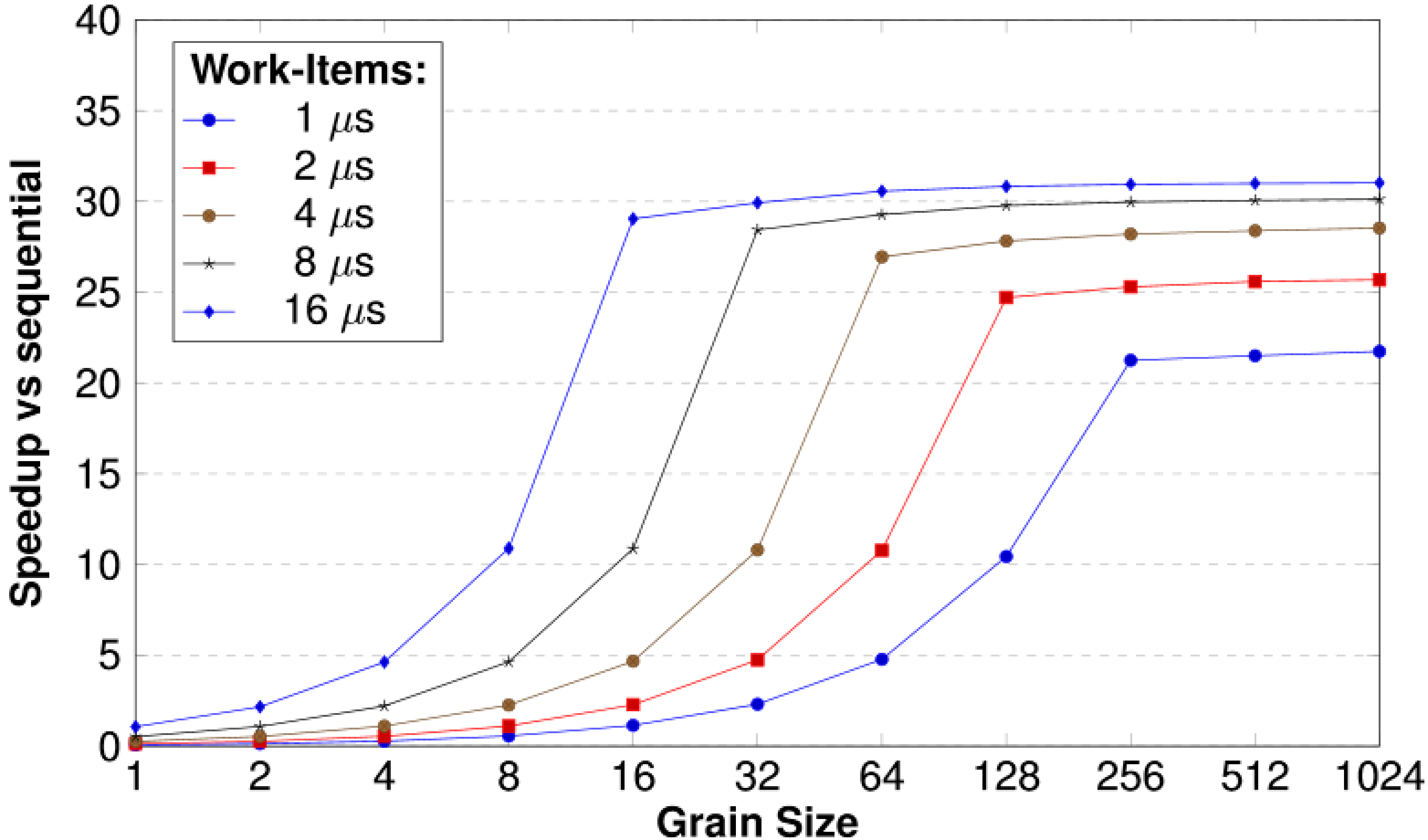
- OpenCL is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs GPUs DSPs FPGAs
- Highly portable

HPXCL : OpenCL backend

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

```
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);
          });
```

N4352


Work-Items:
1 μs
2 μs
4 μs
8 μs
16 μs
Speedup vs sequential
Grain Size
0
5
10
15
20
25
30
35
40
1
2
4
8
16
32
64
128
256
512
1024

Results

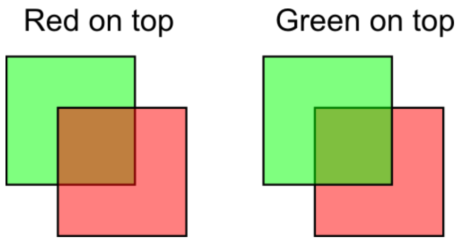
- Parallel executions will give a performance boost proportional to the number of cores available.

- Significant overhead to overcome initially, scales very well however

Example Use Case: Particles

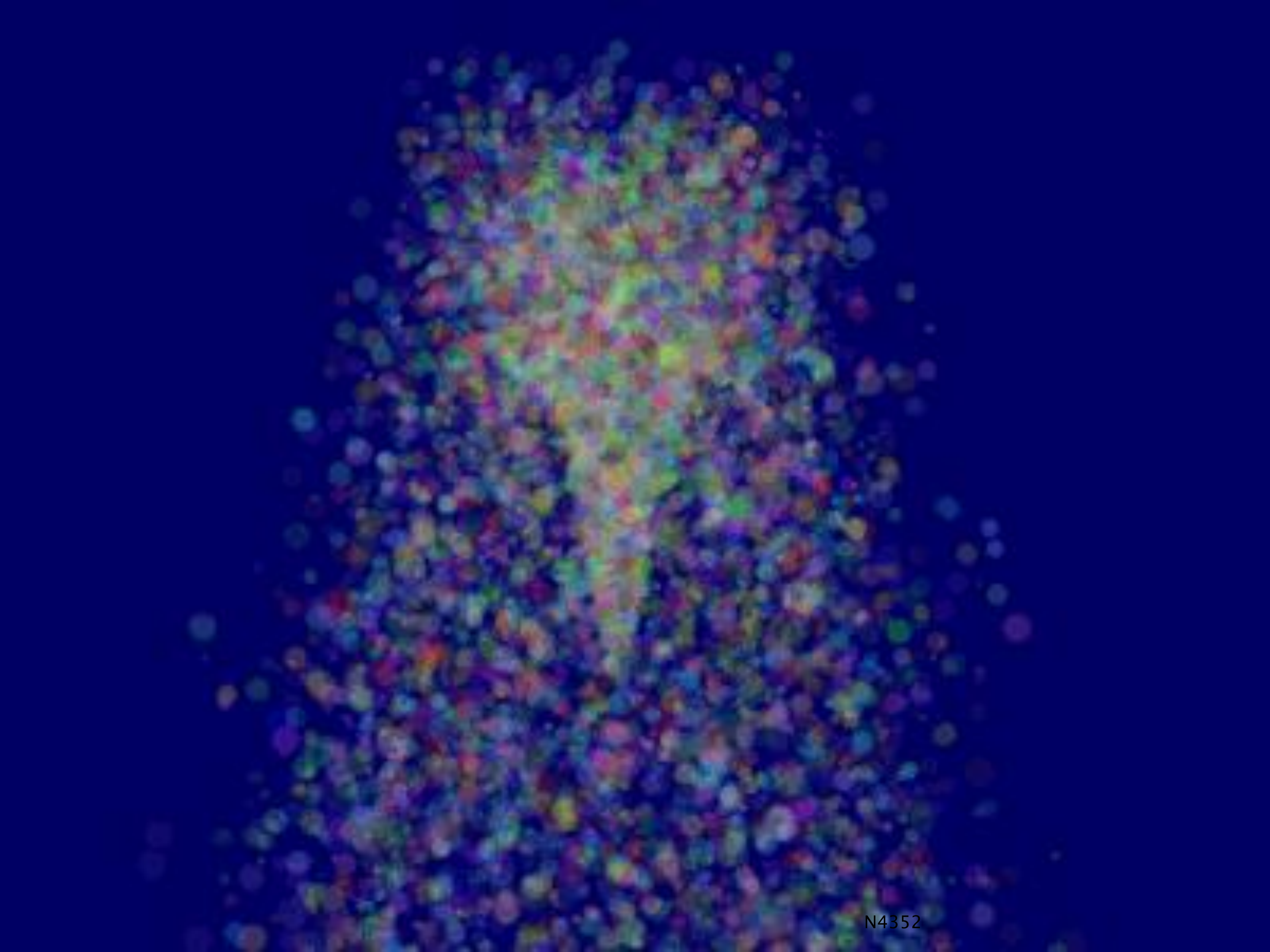
- When displaying particles with varying alpha values, **order matters**

- One easy way correctly draw a stream of particles is to first *sort* them by alpha, and draw in order




```
while( running )
{
    // handle events
    // for ALL particles in container
    // determine next particle location

    // SORT particle container
    // fill OpenGL buffers
    // draw
}
```

N4352


N4352

Drawbacks

- No resource sharing: each call to an algorithm will create, then destroy resources , no chaining possible.
- Getting maximum performance means tweaking your grain size and knowing your execution times. Not always possible

Conclusion

- N4352 has shown very promising results for us at HPX

- Additional user flexibility in C++ would be a huge benefit, especially with highly scalable code

Additional information

- https://github.com/Syntaf
- https://github.com/STEllAR-GROUP/hpx
- http://stellar.cct.lsu.edu/
- https://isocpp.org/blog/2015/01/n4352-53