HPX
The C++ Standards Library for Concurrency and Parallelism

Hartmut Kaiser (hkaiser@cct.lsu.edu)
The Application Problems
HPX – A General Purpose Runtime System

- The C++ Standards Library for Concurrency and Parallelism
- Exposes a coherent and uniform, C++ standards-conforming API for ease of programming parallel, distributed, and heterogeneous applications.
  - Enables to write fully asynchronous code using hundreds of millions of threads.
  - Provides unified syntax and semantics for local and remote operations.
  - Enables seamless data parallelism orthogonally to task-based parallelism
- HPX represents an innovative mixture of
  - A global system-wide address space (AGAS - Active Global Address Space)
  - Fine grain parallelism and lightweight synchronization
  - Combined with implicit, work queue based, message driven computation
  - Support for hardware accelerators
HPX – A C++ Standard Library

- Widely portable
  - Platforms: x86/64, Xeon/Phi, ARM 32/64, Power, BlueGene/Q
  - Operating systems: Linux, Windows, Android, OS/X

- Well integrated with compiler’s C++ Standard libraries

- Enables writing applications which out-perform and out-scale existing applications based on OpenMP/MPI
  - [http://stellar-group.org/libraries/hpx](http://stellar-group.org/libraries/hpx)
  - [https://github.com/STEllAR-GROUP/hpx/](https://github.com/STEllAR-GROUP/hpx/)

- Is published under Boost license and has an open, active, and thriving developer community.

- Can be used as a platform for research and experimentation
HPX – A C++ Standard Library

C++1y Concurrency/Parallelism APIs

Threading Subsystem

Active Global Address Space (AGAS)

Performance Counter Framework

Local Control Objects (LCOs)

Parcel Transport Layer

Policy Engine/Policies
Programming Model

- Focus on the logical composition of data processing, rather than the physical orchestration of parallel computation
- Provide useful abstractions that shield programmer from low-level details of parallel and distributed processing
- Centered around data dependencies not communication patterns
- Make data dependencies explicit to system thus allows for auto-magic parallelization
- Basis for various types of higher level parallelism, such as iterative, fork-join, continuation-style, asynchronous, data-parallelism
- Enable runtime-based adaptivity while applying application-defined policies
Programming Model

- The consequent application of the Concept of Futures
  - Make data dependencies explicit and visible to the runtime

- Implicit and explicit asynchrony
  - Transparently hide communication and other latencies
  - Makes over-subscription manageable
  - Uniform API for local and remote operation
    - Local operation: create new thread
    - Remote operation: send parcel (active message), create thread on behalf of sender

- Work-stealing scheduler
  - Inherently multi-threaded environment
  - Supports millions of concurrently active threads, minimal thread overhead
  - Enables transparent load balancing of work across all execution resources inside a locality

- API is fully conforming with C++11/C++17 and ongoing standardization efforts
HPX – The API

- As close as possible to C++11/14/17 standard library, where appropriate, for instance
  - std::thread
  - std::mutex
  - std::future
  - std::async
  - std::bind
  - std::function
  - std::tuple
  - std::any
  - std::cout
  - std::for_each(par, ...), etc.
  - std::experimental::task_block
  - hpx::thread
  - hpx::mutex
  - hpx::future (including N4538, ‘Concurrency TS’)
  - hpx::async (including N3632)
  - hpx::bind
  - hpx::function
  - hpx::tuple
  - hpx::any (N3508)
  - hpx::cout
  - hpx::parallel::for_each (C++17)
  - hpx::parallel::task_block (Parallelism TS 2)
Control Model: How is parallelism achieved?

- Explicit parallelism:
  - Low-level: thread
  - Middle-level: async(), dataflow(), future::then()

- Higher-level constructs
  - Parallel algorithms (parallel::for_each and friends, fork-join parallelism for homogeneous tasks)
    - Asynchronous algorithms (alleviates bad effect of fork/join)
  - Task-block (fork-join parallelism of heterogeneous tasks)
    - Asynchronous task-blocks
  - Continuation-style parallelism based on composing futures (task-based parallelism)
  - Data-parallelism on accelerator architectures (vector-ops, GPUs)
    - Same code used for CPU and accelerators
### Parallel Algorithms (C++17)

<table>
<thead>
<tr>
<th>adjacent_difference</th>
<th>adjacent_find</th>
<th>all_of</th>
<th>any_of</th>
</tr>
</thead>
<tbody>
<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>uninitialized_copy</td>
<td>uninitialized_copy_n</td>
<td>uninitialized_fill</td>
<td>uninitialized_fill_n</td>
</tr>
<tr>
<td>unique</td>
<td>unique_copy</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

 STELLAR GROUP
STREAM Benchmark

```cpp
std::vector<double> a, b, c;    // data

// ... init data

auto a_begin = a.begin(), a_end = a.end(), b_begin = b.begin() ...;

// STREAM benchmark
parallel::copy(par, a_begin, a_end, c_begin);   // copy step: c = a
parallel::transform(par, c_begin, c_end, b_begin, 
    [](double val) { return 3.0 * val; });   // scale step: b = k * c
parallel::transform(par, a_begin, a_end, b_begin, b_end, c_begin, 
    [](double val1, double val2) { return val1 + val2; });   // add two arrays: c = a + b
parallel::transform(par, b_begin, b_end, c_begin, c_end, a_begin, 
    [](double val1, double val2) { return val1 + 3.0 * val2; });   // triad step: a = b + k * c
```
Dot-product: Vectorization

```cpp
std::vector<float> data1 = {...};
std::vector<float> data2 = {...};

double p = parallel::inner_product(
    datapar,
    std::begin(data1), std::end(data1),
    std::begin(data2),
    0.0f,
    [](auto t1, auto t2) { return t1 + t2; }, // std::plus<>()
    [](auto t1, auto t2) { return t1 * t2; }  // std::multiplies<>()
);
```
Control Model: How is synchronization expressed?

- Low-level (thread-level) synchronization: mutex, condition_variable, etc.

- Replace (global) barriers with finer-grain synchronization (synchronize of a ‘as-need-basis’)
  - Wait only for immediately necessary dependencies, forward progress as much as possible

- Many APIs hand out a future representing the result
  - Parallel and sequential composition of futures (future::then(), when_all(), etc.)
  - Orchestration of parallelism through launching and synchronizing with asynchronous tasks

- Synchronization primitives: barrier, latch, semaphore, channel,
  - Synchronize using futures
Synchronization with Futures

- 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)
What is a (the) Future?

- Many ways to get hold of a future, simplest way is to use (std) async:

```cpp
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
}
```
Data Model

• AGAS essential underpinning for all data management
  • Foundation for syntactic semantic equivalence of local and remote operations

• Full spectrum of C++ data structures are available
  • Either as distributed data structures or for SPMD style computation

• Explicit data partitioning, manually orchestrated boundary exchange
  • Using existing synchronization primitives (for instance channels)

• Use of distributed data structures, like partitioned_vector
  • Use of parallel algorithms
  • Use of co-array like layer (FORTRAN users like that)

• Load balancing: migration
  • Move objects around in between nodes without stopping the application
Small Example
Extending Parallel Algorithms

Sean Parent: C++ Seasoning, Going Native 2013
Extending Parallel Algorithms

- New algorithm: gather

```cpp
template <typename BiIter, typename Pred>
pair<BiIter, BiIter> gather(BiIter f, BiIter l, BiIter p, Pred pred)
{
    BiIter it1 = stable_partition(f, p, not1(pred));
    BiIter it2 = stable_partition(p, l, pred);
    return make_pair(it1, it2);
}
```
Extending Parallel Algorithms

- New algorithm: gather_async

```cpp
template <typename BiIter, typename Pred>
future<pair<BiIter, BiIter>> gather_async(BiIter f, BiIter l, BiIter p, Pred pred)
{
    future<BiIter> f1 = parallel::stable_partition(par(task), f, p, not1(pred));
    future<BiIter> f2 = parallel::stable_partition(par(task), p, l, pred);
    return dataflow(
        unwrapping([](BiIter r1, BiIter r2) { return make_pair(r1, r2); }),
        f1, f2);
}
```
Extending Parallel Algorithms (await)

- New algorithm: gather_async

```cpp
template <typename BiIter, typename Pred>
future<pair<BiIter, BiIter>> gather_async(BiIter f, BiIter l, BiIter p, Pred pred)
{
    future<BiIter> f1 = parallel::stable_partition(par(task), f, p, not1(pred));
    future<BiIter> f2 = parallel::stable_partition(par(task), p, l, pred);
    co_return make_pair(co_await f1, co_await f2);
}
```
Recent Results
Merging White Dwarfs
Adaptive Mesh Refinement

Speedup and Parallel Efficiency of OctoTiger
(single node, 7 Levels of Refinement, 1641 sub-grids)

Cori II (NERSC)
Adaptive Mesh Refinement

![Number of Sub-grids Processed per Second](image)

- Cori II (NERSC)
The Solution to the Application Problem

<table>
<thead>
<tr>
<th>Thread</th>
<th>Time</th>
<th>Time</th>
<th>Time</th>
<th>Time</th>
<th>Time</th>
<th>Time</th>
<th>Time</th>
<th>Time</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread 8</td>
<td>0s</td>
<td>20s</td>
<td>40s</td>
<td>60s</td>
<td>80s</td>
<td>100s</td>
<td>120s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram showing thread activity over time.
The Solution to the Application Problem
HPX + Cling + Jupyter

This tutorial works in a special Jupyter notebook that can be used in one of two ways:

- From this website: https://hpx-jupyter.cct.lsu.edu
- From the docker image: stevenrbrandt/fedora-hpx-cling

Normally, each cell should contain declarations, e.g. definitions of functions, variables, or `#include` statements.

```
#include using namespace std;
```

If you wish to process an expression, e.g. `cout << "hello world\n"` you can put `.expr` at the front of the cell.

```
..expr cout << "hello, world\n";
```

Sometimes you will want to test a cell because you are uncertain whether it might cause a segfault or some other error that will kill your kernel. Other times, you might want to test a definition without permanently adding it to the current namespace. You can do this by prefixing your cell with `.test`. Whatever is calculated in a test cell will be thrown away after evaluation and will not kill your kernel.

```
..test.expr int foo[5]; foo[10] = 1;
```

## Docker Instructions

1. First, install Docker on your local resource
2. Second, start Docker, e.g. `sudo service docker start`
3. Third, run the fedora-hpx-cling container, e.g.

```
docker run -it -p 8000:8000 stevenrbrandt/fedora-hpx-cling
```

After you do this, docker will respond with something like

```
http://0.0.0.0:8000/?token=5d1eb8a4797851910de481985a54c2f3c3be802802350bac5`
```

Paste that URL into your browser, and you will be able to interact with the notebook.

4. Fourth, play with the existing ipynb files or create new ones.
5. Fifth, save your work! This is an important step. If you simply quit the container, everything you did will be lost. To save your work, first find your docker image using `docker ps`.
```
$ docker ps CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES 4f806b5f4fb3 stevenrbrandt/fedora-hpx-cling /bin/sh -c 'jupyter ' 11 minutes ago Up 11 minutes 0.0.0.0:8000- >8000/tcp dreamy_turing```

Once you have it (in this case, it's 4f806b5f4fb3), you can use `docker cp` to transfer files to or from your image.

```
docker cp 4f806b5f4fb3:/home/jup/HPX_by_example.ipynb. docker cp HPX_by_example.ipynb 4f806b5f4fb3:/home/jup```

In [1]: `#include <hpx/hpx.hpp>`

Out[1]:
What is a (the) Future?

Many ways to get hold of a future, simplest way is to use (std) async:

```cpp
using namespace std;
using namespace hpx;

In [2]:
```

```cpp
In [3]:
```}

```cpp
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
}
```

```cpp
Out[3]:
```

If we want to do something other than a declaration, use the ".expr" prefix.

```cpp
In [4]:
```}

```cpp
.compr deep_thought()
```

```cpp
Out[4]:
```

Compositional Facilities

```cpp
In [5]:
```}

```cpp
future<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
        });
    return f2;
}
```

```cpp
Out[5]:
```

```cpp
In [6]:
```}

```cpp
.compr cout << make_string().get() << endl;
```

```cpp
Out[6]:
```

```cpp
Out[7]:
```
In [7]:
```
int do_work(hpx::lcos::future<hpx::util::tuple<hpx::lcos::future<int>,
   hpx::lcos::future<std::basic_string<char> > > >& w) {
    // extract the value of the first argument.
    return hpx::util::get<0>(w.get()).get();
}
```

future<int> test_when_all()
{
    future<int> future1 = async([]()->int {
        return 125;
    });
    future<string> future2 = async([]()->string {
        return string("hi");
    });

    auto all_f = when_all(future1,future2);

    future<int> result = all_f.then(
        [](auto f)->int {
            return do_work(f);
        });
    return result;
}
```

Out[7]:
0

In [8]:
```
.date.expr cout << test_when_all().get() << endl;
```

125
```
Test
```

Out[8]:

### Parallel Algorithms

HPX allows you to write loop parallel algorithms in a generic fashion, applying to specify the way in which parallelism is achieved (i.e. threads, distributed, cuda, etc.) through policies.

In [9]:
```
#include <hpx/include/parallel_for_each.hpp>
#include <hpx/parallel/algorithms/transform.hpp>
#include <boost/iterator/counting_iterator.hpp>
```

Out[9]:

In [10]:
```
vector<int> v = { 1, 2, 3, 4, 5, 6 };
```

Out[10]:

### Transform

Here we demonstrate the transformation of a vector, and the various mechanisms by which it can performed in parallel.
In [11]:
// This parallel tranformation of vector v
// is done using thread parallelism. An
// implicit barrier is present at the end.
 parallel::transform (  
    parallel::execution::par,  
    begin(v), end(v), begin(v),  
    [](int i) -> int  
    {  
        return i+1;  
    });  
for(int i : v) cout << i << ",";  

2,3,4,5,6,7,

Out[11]:

In [12]:
// This parallel transformation of vector v
// is done using thread parallelism. There
// is no implicit barrier. Instead, the
// transform returns a future.
 auto f = parallel::transform (  
    parallel::par (parallel::execution::task),  
    begin(v), end(v), begin(v),  
    [](int i) -> int  
    {  
        return i+1;  
    });  
// work here...  
// wait for the future to be ready.
 f.wait();  
for(int i : v) cout << i << ",";  

3,4,5,6,7,8,

Out[12]:

In [13]:
#include <hpx/include/parallel_fill.hpp>
#include <hpx/include/compute.hpp>
#include <hpx/include/parallel_executors.hpp>

Out[13]:

In [14]:
auto host_targets = hpx::compute::host::get_local_targets();
typedef hpx::compute::host::block_executor<> executor_type;
executor_type exec(host_targets);

Out[14]:
In [15]:

// Print out a list of the localities, i.e. hosts
// that can potentially be involved in this calculation.
// This notebook will probably show 1, alas.
for(auto host : host_targets)
    cout << host.get_locality() << endl;

{0000000010000000, 0000000000000000}

Out[15]:

Other Algorithms

There are a great many algorithms. Here we demonstrate a handful of them.

In [16]:

std::vector<float> vd;
for(int i=0;i<10;i++) vd.push_back(1.f);
parallel::fill(parallel::execution::par.on(exec),vd.begin(),vd.end(),0.0f);

Out[16]:

In [17]:

#include <hpx/parallel/algorithms/reverse.hpp>

Out[17]:

In [18]:

std::vector<float> vd;
for(int i=0;i<10;i++) vd.push_back(1.f*i);
parallel::reverse(parallel::par,vd.begin(),vd.end());
for(int val : vd) cout << val << " ";

9 8 7 6 5 4 3 2 1 0

Out[18]:

In [19]:

#include <hpx/include/parallel_minmax.hpp>

Out[19]:

In [20]:

std::vector<float> vd;
for(int i=0;i<10;i++) vd.push_back(1.f*rand());
auto ptr = parallel::max_element(parallel::par,vd,std::less<float_>());
for(float val : vd) cout << val << " ";
cout << endl << *ptr << endl;

8.02369e+08 1.63599e+09 1.60543e+09 3.22506e+08 4.34983e+08 1.87237e+09 2.04466e+09 1.82667e+09 1.27975e+09 1.95976e+09
9 2.04466e+09 1.82667e+09 1.27975e+09 1.95976e+09

Out[20]:
In [21]:
```cpp
#include "hpx/traits/is_executor.hpp"
#include "hpx/include/parallel_executors.hpp"
```

Out[21]:

In [22]:
```cpp
int count_async = 0;
struct test_async_executor
{
    typedef hpx::parallel::parallel_execution_tag execution_category;

    template <typename F, typename ... Ts>
    static hpx::future<typename hpx::util::result_of<F&&(Ts&&...)>::type>
        async_execute(F && f, Ts &&... ts)
    {
        ++count_async;
        return hpx::async(hpx::launch::async, std::forward<F>(f),
                          std::forward<Ts>(ts)...);
    }
};
```

Out[22]:

In [23]:
```cpp
// Note that the exact way to specify this trait for an executor is in
// flux
// and the code here is tied to the specific version of HPX on the tes
// t machine.
namespace hpx { namespace traits {

    template<>
    struct is_two_way_executor<test_async_executor>
    : std::true_type
    {
        ...
    }
}
```

Out[23]:

In [24]:
```cpp
// This parallel tranformation of vector v
// is done using using distributed parallelism.
    test_async_executor e;
    parallel::transform (    parallel::execution::par.on(e),
        begin(v), end(v), begin(v),
        [](int i) -> int
        {
            return i+1;
        });
    cout << "count=" << count_async << endl;

    count=3
```

Out[24]:

Test
Let’s Parallelize It – Adding Real Asynchrony

Here we take a step back. Instead of using a pre-designed parallel operation on a vector, we instead introduce task-level parallelism to an existing program.

Calculate Fibonacci numbers in parallel (1st attempt)

In [25]:
   uint64_t fibonacci(uint64_t n)
   {
       // if we know the answer, we return the value
       if (n < 2) return n;
       // asynchronously calculate one of the sub-terms
       future<uint64_t> f = async(launch::async, &fibonacci, n-2);
       // synchronously calculate the other sub-term
       uint64_t r = fibonacci(n-1);
       // wait for the future and calculate the result
       return f.get() + r;
   }

    Out[25]:

In [26]:
   .expr cout << fibonacci(10) << endl;

      55

    Out[26]:

Let’s Parallelize It – Introducing Control of Grain Size

Parallel calculation, switching to serial execution below given threshold
In [27]:

```cpp
const int threshold = 20;

uint64_t fibonacci_serial(uint64_t n)
{
    if (n < 2) return n;
    uint64_t f1 = fibonacci_serial(n-2);
    uint64_t f2 = fibonacci_serial(n-1);
    return f1 + f2;
}

uint64_t fibonacci2(uint64_t n)
{
    if (n < 2) return n;
    if (n < threshold) return fibonacci_serial(n);
    // asynchronously calculate one of the sub-terms
    future<uint64_t> f = async(launch::async, &fibonacci2, n-2);
    // synchronously calculate the other sub-term
    uint64_t r = fibonacci2(n-1);
    // wait for the future and calculate the result
    return f.get() + r;
}
```

Out[27]:

In [28]:

```cpp
.expr cout << fibonacci2(22) << endl;
```

17711

Out[28]:

Let's Parallelize It – Apply Futurization

Parallel way, futurize algorithm to remove suspension points

In [29]:

```cpp
future<uint64_t> fibonacci3(uint64_t n)
{
    if(n < 2) return make_ready_future(n);
    if(n < threshold) return make_ready_future(fibonacci_serial(n));

    future<uint64_t> f = async(launch::async, &fibonacci3, n-2);
    future<uint64_t> r = fibonacci3(n-1);

    return dataflow(
        [](future<uint64_t> f1, future<uint64_t> f2) {
            return f1.get() + f2.get();
        },
        f, r);
}
```

Out[29]:

```cpp
```
Let’s Parallelize It – Unwrap Argument Futures

```cpp
#include <hpx/util/unwrapped.hpp>

using hpx::util::unwrapping;

future<

future<uint64_t> fibonacci4(uint64_t n)
{
    if(n < 2) return make_ready_future(n);
    if(n < threshold) return make_ready_future(fibonacci_serial(n));

    future<

    return dataflow(  
        unwrapping([[](uint64_t f1, uint64_t f2) {  
            return f1+f2;
        }],  
        f, r);
}
```

Excercise: Parallelize a sort

Test what you’ve learned. See if you can speed up the quicksort program below by find a place to:

1. parallelize the code with async
2. use parallel transforms

```cpp
#include <unistd.h>
#include <stdlib.h>
#include <iostream>
#include <vector>
#include <functional>

using namespace std;

function<void(vector<int>&)> myqsort = [] (vector<int>& v) -> void {};
```

Out[30]: 17711
Out[31]: 17711
Out[32]: 17711
Out[33]:
Discussion
We want to define the myqsort function repeatedly, and call it recursively. This is hard to do in C++. So we define
it as a std::function<>. There is a slight awkwardness to this. If you want to call myqsort() with an async function,
you have to do it like this:

```cpp
auto f = hpx::async([&arg](){ myqsort(arg); });
```

Not like this

```cpp
auto f = hpx::async(myqsort,arg);
```

```cpp
def myqsort(vector<int>& v): return;
    vector<int> pre, eq, post;
    int pivot = v[rand() % v.size()];
    for(int val : v) {
        if(val < pivot) pre.push_back(val);
        else if(pivot < val) post.push_back(val);
        else eq.push_back(val);
    }
    myqsort(pre);
    myqsort(post);
    for(int i=0;i<eq.size();i++) v[i+pre.size()] = eq[i];
    parallel::transform(parallel::par,
        eq.begin(), eq.end(), v.begin()+pre.size(),[] (int i) { return i; });
    for(int i=0;i<post.size();i++) v[i+pre.size()+eq.size()] = post[i];
}
```

```cpp
vector<int> vv{20};
for(int i=0;i<20;i++) vv.push_back(rand() % 100);
for(int val : vv) cout << val << " ";
cout << endl;
myqsort(vv);
for(int val : vv) cout << val << " ";
cout << endl;
```

Test

```
20 26 32 84 5 66 50 81 7 5 53 69 45 84 94 59 21 80 96 17 6
5 5 6 7 17 20 21 26 32 45 50 53 59 66 69 80 81 84 84 94 96
```
The Wave Equation

This problem file sets up a very simple physical system, a wave propagating in one dimension. It is a nice example because it requires several loops in sequence and presents an opportunity to practice creating and using HPX parallel algorithms.

In [1]:
#include <hpx/hpx.hpp>
#include <vector>
#include <hpx/include/parallel_for_each.hpp>
#include <hpx/parallel/algorithms/transform.hpp>
#include <boost/iterator/counting_iterator.hpp>

Out[1]:

Basic Variables

We are going to evolve two variables, phi and psi on a grid described by $N$, $dx$, and $x_0$. The system of equations we will solve is

$$\partial_t \phi = c \partial_x \psi$$

$$\partial_t \psi = c \partial_x \phi$$

In [2]:
const int N = 300;
std::vector<double> phi(N), psi(N);
const double dx = 0.01, x0 = -1.5;

Out[2]:

https://hpx-jupyter.cct.lsu.edu/user/sbrandt/nbconvert/html/share/WaveToy.ipynb?download=false
```cpp
#include <fstream>
#include <sstream>
#include <iostream>
#include <vector>
#include <content.hpp>

/**
 * The following function plots an array of vectors
 * of doubles in a single overlapping plot.
 */
void plot_vector(std::vector<std::vector<double>> v, std::string iname) {
    // Store the data in a text file
    const char *fname = "data.txt";
    std::ofstream o(fname);
    for(int n=0; n<v.size(); n++) {
        const std::vector<double>& vv = v[n];
        for(int i=0; i<vv.size(); i++) {
            if(i > 0) o << ' ';
            o << i;
        }
        o << std::endl;
        for(int i=0; i<vv.size(); i++) {
            if(i > 0) o << ' ';
            o << vv[i];
        }
        o << std::endl;
    }
    o.close();

    // Create a python script to run matplotlib
    std::ostringstream cmd;
    cmd << "import matplotlib\n";
    cmd << "matplotlib.use('Agg')\n";
    cmd << "import numpy as np\n";
    cmd << "import matplotlib.pyplot as plt\n";
    cmd << "f = np.genfromtxt('" << fname << ")\n";
    cmd << "plt.figure()\n";
    cmd << "for n in range(0, f.shape[0], 2):
";
    cmd << "  plt.plot(f[n, :], f[n+1, :])\n";
    cmd << "plt.savefig('" << iname << ".png')\n";
    cmd << "exit(0)\n";
    std::ofstream o2("p.py");
    o2 << cmd.str();
    o2.close();
    system("python3 p.py");

    // Create the html. The pid is added to prevent caching.
    // Note that the pid changes with every cell with the
    // current implementation of HPX/cling.
    std::ostringstream html;
    html << "<img src='" << iname << ".png?pid=" << getpid() << "-" << rand() << ">";
    std::string htmls = html.str();
    create_content(htmls.c_str());
}
```
In [4]:
// What follows is a random test of the plot function.
std::vector<double> v1 = {1,2,3,4,3.5,2,9,9};
std::vector<double> v2 = {5,6,7,8,8.5,7.4,2,4};
std::vector<std::vector<double>> plots;
plots.push_back(v1);
plots.push_back(v2);
plot_vector(plots,"test");

Out[3]:

Out[4]:

In [8]:
// This version of plot_vector plots a single vector only.
void plot_vector(const std::vector<double>& v) {
    std::vector<std::vector<double>> vv;
    vv.push_back(v);
    plot_vector(vv,"vec");
}

Out[8]:

https://hpx-jupyter.cct.lsu.edu/user/sbrandt/nbconvert/html/share/WaveToy.ipynb?download=false
// Apply boundary conditions. In this case, we are using periodic boundary
// conditions, i.e. if we move N-2 points to the right we come back to
where we were.
void boundary(std::vector<double>& vv) {
    const int n = vv.size();
    vv[0] = vv[n-2];
    vv[n-1] = vv[1];
}

// The following are auxiliary variables which are required by our Runge-Kutta
// time integration scheme.
std::vector<double> phi2(N), psi2(N), phi3(N), psi3(N);
std::vector<double> k1_phi(N), k1_psi(N), k2_phi(N), k2_psi(N), k3_phi(N), k3_psi(N);

The Wave Equation Evolution Code
The sequence of loops below will perform an evolution of the wave equation, essentially, a sine wave that
propagates unchanged to the right. The important thing is not understanding the physics, but in parallelizing the
loops and recognizing the dependencies between them.

As an example, the first loop is already parallelized with for_each. Note: You should be able to find a place to use
parallel::execution::task.
In [15]:
"expr"
std::vector<std::vector<double>> plots;
double t = 0.0, t_end = 1.0;
double t_every = t_end/10;
double t_plot = 0;
double dt = dx/2.0;

const double w = 1;
const double pi = 4.0*atan2(1.0,1.0);
const double k = 2.0*pi/(dx*(N-2));

const double c = w/k;
const double A = 1.0;
const double B = c*k*A/w; // == 1.0

// Initialize the physical variables
/*
for(int i=0;i<N;i++) {
    // psi = A*cos(k*x + w*t)  // phi = B*cos(k*x + w*t)
    // d(phi)/dt = c*d(psi)/dx  // w*B*sin(k*x + w*t) = c*k*A*sin(k*x + w*t)
    // B = -c*k*A/w  // d(psi)/dt = c*d(phi)/dx  // w*A*sin(k*x + w*t) = c*k*B*sin(k*x + w*t)
    double x = x0 + i*dx;
    phi[i] = B*sin(k*x);
    psi[i] = A*sin(k*x);
} */

hpx::parallel::for_each(
    hpx::parallel::par,
    boost::counting_iterator<int>(0),
    boost::counting_iterator<int>(N),
    [&]( int i ) {
        double x = x0 + i*dx;
        phi[i] = B*sin(k*x);
        psi[i] = A*sin(k*x);
    });

while(t < t_end) {
    for(int i=1;i<N-1;i++) {
        k1_phi[i] = c*(psi[i+1]-psi[i-1])/(2*dx);
        k1_psi[i] = c*(phi[i+1]-phi[i-1])/(2*dx);
    }
    for(int i=1;i<N-1;i++) {
        phi2[i] = phi[i] + (1./3.)*dt*k1_phi[i];
        psi2[i] = psi[i] + (1./3.)*dt*k1_psi[i];
    }

    // the boundary routines can be parallelized also
    boundary(phi2);
    boundary(psi2);
    for(int i=1;i<N-1;i++) {
        k2_phi[i] = c*(psi2[i+1]-psi2[i-1])/(2*dx);
        k2_psi[i] = c*(phi2[i+1]-phi2[i-1])/(2*dx);
    }
    for(int i=1;i<N-1;i++) {
        phi3[i] = phi[i] + (2./3.)*dt*k1_phi[i];
        psi3[i] = psi[i] + (2./3.)*dt*k1_psi[i];
    }
    boundary(phi3);
    boundary(psi3);
    for(int i=1;i<N-1;i++) {
        k3_phi[i] = c*(psi3[i+1]-psi3[i-1])/(2*dx);
        k3_psi[i] = c*(phi3[i+1]-phi3[i-1])/(2*dx);
    }
    for(int i=1;i<N-1;i++) {
        phi[i] = phi[i] + 0.5*dt*(k2_phi[i]+k3_phi[i]);
        psi[i] = psi[i] + 0.5*dt*(k2_psi[i]+k3_psi[i]);
    }
    boundary(phi);
    boundary(psi);
    t += dt;
    t_plot += dt;
    if(t_plot >= t_every) {
        // do something
    }
}
Install Docker image from USB

- Install Docker (if necessary)
- Insert USB key, open a terminal, navigate to key directory and:
  - (sudo) docker load -i fedora-hpx-cling
  - (sudo) docker pull stevenrbrandt/fedora-hpx-cling
  - (sudo) docker run -it -p 8000:8000 stevenrbrandt/fedora-hpx-cling
- “sudo” may be necessary on some machines
Outline

• Introduction to APEX – *Autonomic Performance Environment for eXascale*
• Motivation, overview, API
• Integration with HPX
• Building HPX with APEX
• APEX event listeners
• Postmortem analysis of HPX applications
  – Gnuplot/Python visualizations of APEX data
  – OTF2 output to Vampir
  – Profile output to TAU
APEX Measurement: Motivation

- Originally designed as a performance measurement library for distributed, asynchronous tasking models/runtimes
  - i.e. HPX, but there are others
- Why another measurement library?
  - “not invented here” mentality? Reinventing the wheel? No.
- New challenges:
  - Lightweight measurement (tasks <1ms)
  - High concurrency (both OS threads and tasks in flight)
  - Distinction between OS and runtime (HPX) thread context
  - Lack of a traditional call stack
    - Task dependency chain instead
  - Runtime controlled task switching
    - *Dynamic runtime control*
APEX Runtime Adaptation: Motivation

- Controlling concurrency
  - Energy efficiency
  - Performance

- Parametric variability
  - Granularity for this machine / dataset?

- Load Balancing
  - When to perform AGAS migration?

- Parallel Algorithms (for_each...)
  - Separate what from how

- Address the “SLOW(ER)” performance model
Introduction to APEX

• **Performance awareness and performance adaptation**
• **Top down and bottom up** performance mapping / feedback
  – Make node-wide resource utilization data and analysis, energy consumption, and health information available in real time
  – Associate performance state with policy for feedback control
• **APEX introspection**
  – OS: track system resources, utilization, job contention, overhead
  – Runtime (HPX): track threads, queues, concurrency, remote operations, parcels, memory management
  – Application timer / counter observation
• **Post-mortem performance analysis**
  – “secondary” goal, but has been useful
• **Integrated with HPX performance counters**
APEX architecture

- APEX Introspection
  - Synchronous
  - Asynchronous

- System Info
  (/proc, getusage, LM Sensors, etc.)

- APEX State
  - Triggered
  - Periodic

- APEX Policy Engine

- Application
- HPX

meta-events

events

actuators
APEX Introspection

• **APEX collects data through “inspectors”**
  
  – *Synchronous* uses an event API and event “listeners”
    
    • Initialize, terminate, new thread – added to HPX runtime
    • Timer start, stop, yield*, resume* - added to HPX task scheduler
    • Sampled value (counters from HPX)
    • Custom events (meta-events)
  
  – *Asynchronous* do not rely on events, but occur periodically

• **APEX exploits access to performance data from lower stack components**
  
  – “Health” data through other interfaces (/proc/stat, cpuinfo, meminfo, net/dev, self/status, lm_sensors, power*, etc.)
APEX Measurement API (subset)

**Starting Timers:**

```cpp
apex::profiler* apex::start(const string &name, void** data_ptr = 0LL);
apex::profiler* apex::start(const uint64_t address, void** data_ptr = 0LL);
apex::profiler* apex::resume(const string name, void** context = 0LL);
apex::profiler* apex::resume(const uint64_t address, void** context = 0LL);
```

**Stopping Timers:**

```cpp
void apex::stop(apex::profiler* p);
void apex::yield(apex::profiler* p);
```

**Sampling a counter:**

```cpp
void apex::sample_value(const string name, double value);
```
• Note about yield, resume – exist for accurate counting of “number of calls” in the face of pre-emption (usually handled by HPX scheduler):

<table>
<thead>
<tr>
<th>Timer Start Command</th>
<th>Timer Stop Command</th>
<th>Call count is increased by</th>
</tr>
</thead>
<tbody>
<tr>
<td>apex::start()</td>
<td>apex::stop()</td>
<td>1</td>
</tr>
<tr>
<td>apex::start()</td>
<td>apex::yield()</td>
<td>0</td>
</tr>
<tr>
<td>apex::resume()</td>
<td>apex::yield()</td>
<td>0</td>
</tr>
<tr>
<td>apex::resume()</td>
<td>apex::stop()</td>
<td>0</td>
</tr>
</tbody>
</table>
void foo(int x) {
    // sample the argument, for example
    apex::sample_counter("foo(x)", x);
    // start a timer
    apex::profiler* p = apex::start(&foo);
    /* do some work in function foo */
    ...
    // stop the current timer to wait on some asynchronous subtask
    apex::yield(p);
    /* wait on result from "subtask" */
    result = some_future.get();
    // "resume" the APEX timer
    p = apex::start(&foo);
    /* do some more work in function foo */
    ...
    // stop the timer
    apex::stop(p);
}
HPX and APEX - Integration

- In HPX, all tasks scheduled by the thread scheduler are “automatically” timed – with some caveats
- HPX registered actions are automatically timed
- All threads/tasks are timed, attribution is the required user intervention
  - Asynchronous functions, direct actions are correctly attributed if wrapped with an `hpx::util::annotated_function` object.
  - See notebook examples for details
Annotative examples:

```cpp
// Forward declaration of the action
uint64_t fibonacci_a(uint64_t n);
// This is to generate the required boilerplate we need for the remote
// invocation to work.
HPX_PLAIN_ACTION(fibonacci_a, fibonacci_action);
/* ... */
future<uint64_t> n1 = async(fib, locality_id, n - 1);
```

```cpp
using namespace hpx::util;
future<uint64_t> f = async(launch::async, annotated_function(unwrapping(&fibonacci3), "fibonacci3"), n-2);
```
APEX Event Listeners

• Profiling listener
  – Start event: input name/address, get timestamp, return profiler handle
  – Stop event: get timestamp, put profiler object in a queue for back-end processing, return
  – Sample event: put the name & value in the queue
  – Asynchronous consumer thread: process profiler objects and samples to build statistical profile (in HPX, processed/scheduled as a thread/task)
  – Optional: screen/CSV output, build task scatterplot, build taskgraph, etc.

• TAU Listener (postmortem analysis)
  – Synchronously passes all measurement events to TAU to build an offline profile

• OTF2 Listener (postmortem analysis)
  – Synchronously passes all measurement events to libotf2 for trace analysis

• Concurrency listener (postmortem analysis)
  – Start event: push timer ID on stack
  – Stop event: pop timer ID off stack
  – Asynchronous consumer thread: periodically log current timer for each thread, output report at termination
**Policies** are rules that decide on outcomes based on observed state

- *Triggered* policies are invoked by introspection API events
- *Periodic* policies are run periodically on asynchronous thread

- Polices are registered with the Policy Engine
  - Applications, runtimes, and/or OS register callback functions

- Callback functions define the policy rules
  - “If \( x < y \) then…”

- Enables runtime adaptation using introspection data
  - Feedback and control mechanism
  - Engages actuators across stack layers
  - Could also be used to involve online auto-tuning support*
    - Active Harmony [http://www.dyninst.org/harmony](http://www.dyninst.org/harmony)
APEX Policy API (subset)

```cpp
apex_event_type
apex::register_custom_event(const string &name);
apex::custom_event(apex_event_type, void* event_data);
apex_tuning_session_handle
apex::setup_custom_tuning(std::function<double(void)> metric,
apex_event_type event_type, int num_inputs, long** inputs, long* mins,
long* maxs, long* steps);
```
APEX Policy API example

See last example in notebook:

\texttt{apex::register_custom_event()} in \texttt{setup_counters()}

\texttt{apex::get_profile()} (as potential exercise) in \texttt{get_counter_value()}

\texttt{apex::setup_custom_tuning()} in \texttt{do_1d_solve_repart()}
Building HPX with APEX

$ cmake <usual HPX settings>...

-DHPX_WITH_APEX=TRUE \ 
-DAPEX_WITH_ACTIVEHARMONY=TRUE/FALSE \ 
-DAPEX_WITH_PAPI=TRUE/FALSE \ 
-DAPEX_WITH_MSR=TRUE/FALSE \ 
-DAPEX_WITH_OTF2=TRUE/FALSE \ 
...

$ make (as usual) i.e. “make –j 8 core examples tests”
### APEX environment variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>APEX_DISABLE</td>
<td>0</td>
</tr>
<tr>
<td>APEX_SUSPEND</td>
<td>0</td>
</tr>
<tr>
<td>APEX_PAPI_SUSPEND</td>
<td>0</td>
</tr>
<tr>
<td>APEX_PROCESS_ASYNC_STATE</td>
<td>1</td>
</tr>
<tr>
<td>APEX_TAU</td>
<td>0</td>
</tr>
<tr>
<td>APEX_POLICY</td>
<td>1</td>
</tr>
<tr>
<td>APEX_MEASURE_CONCURRENCY</td>
<td>0</td>
</tr>
<tr>
<td>APEX_MEASURE_CONCURRENCY_PERIOD</td>
<td>1000000</td>
</tr>
<tr>
<td>APEX_SCREEN_OUTPUT</td>
<td>1</td>
</tr>
<tr>
<td>APEXPROFILE_OUTPUT</td>
<td>0</td>
</tr>
<tr>
<td>APEX_CSV_OUTPUT</td>
<td>0</td>
</tr>
<tr>
<td>APEX_TASKGRAPH_OUTPUT</td>
<td>0</td>
</tr>
<tr>
<td>APEX_PROC_CPUINFO</td>
<td>0</td>
</tr>
<tr>
<td>APEX_PROC_MEMINFO</td>
<td>0</td>
</tr>
<tr>
<td>APEX_PROC_NET_DEV</td>
<td>0</td>
</tr>
<tr>
<td>APEX_PROC_SELF_STATUS</td>
<td>0</td>
</tr>
<tr>
<td>APEX_PROC_SELF_IO</td>
<td>0</td>
</tr>
<tr>
<td>APEX_PROC_STAT</td>
<td>1</td>
</tr>
<tr>
<td>APEX_PROC_PERIOD</td>
<td>1000000</td>
</tr>
<tr>
<td>APEX_THROTTLE_CONCURRENCY</td>
<td>0</td>
</tr>
<tr>
<td>APEX_THROTTLING_MAX_THREADS</td>
<td>4</td>
</tr>
<tr>
<td>APEX_THROTTLING_MIN_THREADS</td>
<td>1</td>
</tr>
<tr>
<td>APEX_THROTTLE_ENERGY</td>
<td>0</td>
</tr>
<tr>
<td>APEX_THROTTLE_ENERGY_PERIOD</td>
<td>1000000</td>
</tr>
<tr>
<td>APEX_THROTTLING_MAX_WATTS</td>
<td>300</td>
</tr>
<tr>
<td>APEX_THROTTLING_MIN_WATTS</td>
<td>150</td>
</tr>
<tr>
<td>APEX_PTHREAD.WRAPPER_STACK_SIZE</td>
<td>0</td>
</tr>
<tr>
<td>APEX_OMPT.REQUIRED_EVENTS_ONLY</td>
<td>0</td>
</tr>
<tr>
<td>APEX_OMPT_HIGH_OVERHEAD_EVENTS</td>
<td>0</td>
</tr>
<tr>
<td>APEX_PIN_APEX_THREADS</td>
<td>1</td>
</tr>
<tr>
<td>APEX_TASK_SCATTERPLOT</td>
<td>1</td>
</tr>
<tr>
<td>APEX_POLICY_DRAIN_TIMEOUT</td>
<td>1000</td>
</tr>
<tr>
<td>APEX_PAPI_METRICS</td>
<td></td>
</tr>
<tr>
<td>APEX_PLUGIN</td>
<td></td>
</tr>
<tr>
<td>APEX_PLUGIN_PATH</td>
<td>./</td>
</tr>
<tr>
<td>APEX_OTF2</td>
<td>0</td>
</tr>
<tr>
<td>APEX_OTF2_ARCHIVE_PATH</td>
<td>OTF2_archive</td>
</tr>
<tr>
<td>APEX_OTF2_ARCHIVE_NAME</td>
<td>APEX</td>
</tr>
</tbody>
</table>
Note about environment variables...

- Some batch submission systems do not pass environment variables to the application without considerable effort.
- APEX will read environment variables from $PWD/apex.conf file.
- All variables also have API calls for getting/setting.
  - Note: some are only effective at startup because they change/set the APEX configuration.
    - i.e. APEX_DISABLE, APEX_TAU, APEX_PAPI_METRICS,...
Post-mortem output examples
**Example Screen Output**

Elapsed time: 2.40514 seconds  
Cores detected: 4  
Worker Threads observed: 5  
Available CPU time: 9.62057 seconds

<table>
<thead>
<tr>
<th>Timer</th>
<th>#calls</th>
<th>mean</th>
<th>total</th>
<th>% total</th>
</tr>
</thead>
<tbody>
<tr>
<td>someThread(void*)</td>
<td>2</td>
<td>1.39e+00</td>
<td>2.79e+00</td>
<td>28.965</td>
</tr>
<tr>
<td>foo(int)</td>
<td>131072</td>
<td>2.00e-05</td>
<td>2.63e+00</td>
<td>27.306</td>
</tr>
<tr>
<td>bar(int, apex::profiler**, void**)</td>
<td>131072</td>
<td>9.69e-06</td>
<td>1.27e+00</td>
<td>13.198</td>
</tr>
<tr>
<td>someUntimedThread(void*)</td>
<td>2</td>
<td>1.12e+00</td>
<td>2.24e+00</td>
<td>23.271</td>
</tr>
<tr>
<td>main</td>
<td>1</td>
<td>1.40e+00</td>
<td>1.40e+00</td>
<td>14.548</td>
</tr>
<tr>
<td>APEX MAIN</td>
<td>1</td>
<td>2.41e+00</td>
<td>2.41e+00</td>
<td>100.000</td>
</tr>
</tbody>
</table>

Total timers: 262,150  

To enable:  
```cpp  
export APEX_SCREEN_OUTPUT=1  
or, call apex:::dump(bool reset);  
```
Concurrency View (before fix)

Interactive HPC: Using C++ and HPX inside Jupyterhub to Write Performant Portable Parallel Code (APEX)
Concurrency View (after fix)

To enable:

```bash
export APEX_MEASURE_CONCURRENCY=1
export APEX_MEASURE_CONCURRENCY_PERIOD=100000
```
OTF2 View in Vampir

To enable:

```
export APEX_OTF2=1
```
Interactive HPC: Using C++ and HPX inside Jupyterhub to Write Performant Portable Parallel Code (APEX)

Profile View in TAU ParaProf

<table>
<thead>
<tr>
<th>Metric: TIME</th>
<th>Value: Exclusive</th>
</tr>
</thead>
</table>

To enable:

```bash
export APEX_PROFILE_OUTPUT=1
```

Or:

```bash
export APEX_TAU=1
```

and run with “tau_exec”
Task Scatterplot Analysis (prototype)

To enable:
```sh
export APEX_TASK_SCATTERPLOT=1
```
Taskgraph View (prototype)

To enable:
```
export APEX_TASKGRAPH_OUTPUT=1
```
HPX + APEX Policy Examples
Example: HPX+APEX

- Heat diffusion
- 1D stencil code
- Data array partitioned into chunks
- 1 node with no hyperthreading
- Performance increases to a point with increasing worker threads, then decreases
Concurrency & Performance

• Region of maximum performance correlates with thread queue length runtime performance counter
  – Represents # tasks currently waiting to execute
• Could do introspection on this to control concurrency throttling policy (see next example)
100,000,000 elements, 1000 partitions

- 48 worker threads (with hyperthreading, on Edison)
- Actual concurrency much lower
  - Implementation is memory bound
- Large variation in concurrency over time
  - Tasks waiting on prior tasks to complete

Where calculation takes place

Event-generated metrics

138 secs
1d_stencil w/optimal # of Threads

- 12 worker threads on Edison
- Greater proportion of threads kept busy
  - Less interference between active threads and threads waiting for memory
- Much faster
  - 61 sec. vs 138 sec.
1d_stencil Adaptation with APEX

- Initially 32 worker threads
- ActiveHarmony searches for minimal thread queue length
- Quickly converges on 12
Adapting Block Size

- Is 1000 partitions of 100000 cells the best partitioning?
- Parametric studies say “no”.
- Can we modify the example to repartition as necessary to find better performance?
1d_stencil: adapting block size

Interactive HPC: Using C++ and HPX inside Jupyterhub to Write Performant Portable Parallel Code (APEX)
Support for this work was provided through Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (and Basic Energy Sciences/Biological and Environmental Research/High Energy Physics/Fusion Energy Sciences/Nuclear Physics) under award numbers DE-SC0008638, DE-SC0008704, DE-FG02-11ER26050 and DE-SC0006925.

This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

This material is (will be?) based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1737803.

The authors acknowledge that the Talapas HPC cluster and the High Performance Computing Research Core Facility at The University of Oregon have contributed to the research results reported within this paper.
HPX + Cling + Jupyter

This tutorial works in a special Jupyter notebook that can be used in one of two ways:

- From this website: https://hpx-jupyter.cct.lsu.edu
- From the docker image: stevenrbrandt/fedora-hpx-cling
- Normally, each cell should contain declarations, e.g. definitions of functions, variables, or #include statements.

```
#include using namespace std;
```
- If you wish to process an expression, e.g. cout << "hello world\n" you can put .expr at the front of the cell.

```
..expr cout << "hello, world\n";
```
- Sometimes you will want to test a cell because you are uncertain whether it might cause a segfault or some other error that will kill your kernel. Other times, you might want to test a definition without permanently adding it to the current namespace. You can do this by prefixing your cell with .test. Whatever is calculated in a test cell will be thrown away after evaluation and will not kill your kernel.

```
..test.expr int foo[5]; foo[10] = 1;
```

## Docker Instructions

First, install Docker on your local resource
- Second, start Docker, e.g.
  ```
sudo service docker start
  ```
- Third, run the fedora-hpx-cling container, e.g.
  ```
docker run -it -p 8000:8000
stevenrbrandt/fedora-hpx-cling
```

After you do this, docker will respond with something like

`http://0.0.0.0:8000/?token=5d1eb8a4797851910de481985a54c2fcd3be80280023bac5`

Paste that URL into your browser, and you will be able to interact with the notebook.

- Fourth, play with the existing ipynb files or create new ones.
- Fifth, save your work! This is an important step. If you simply quit the container, everything you did will be lost. To save your work, first find your docker image using docker ps.

```
$ docker ps CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
4f806b5f4fb3 stevenrbrandt/fedora-hpx-cling "/bin/sh -c "jupyter ". 11 minutes ago Up 11 minutes 0.0.0.0:8000->8000/tcp dreamy_turing```

Once you have it (in this case, it's 4f806b5f4fb3), you can use docker cp to transfer files to or from your image.

```
docker cp 4f806b5f4fb3:/home/jupyter/HPX_by_example.ipynb docker cp
HPX_by_example.ipynb 4f806b5f4fb3:/home/jupyter```

Measuring HPX performance

HPX is integrated with APEX, "An Autonomic Performance Environment for eXascale". APEX serves two primary roles in HPX - to measure the HPX runtime and application tasks, and to use introspection of those measurements to control behavior.

Measurement

APEX provides an API for measuring actions within a runtime. The API includes methods for timer start/stop, as well as sampled counter values. APEX is designed to be integrated into a runtime, library and/or application and provide performance introspection for the purpose of runtime adaptation. While APEX can provide rudimentary post-mortem performance analysis measurement, there are many other performance measurement tools that perform that task much better (such as TAU [http://tau.uoregon.edu](http://tau.uoregon.edu)). That said, APEX includes an event listener that integrates with the TAU measurement system, so APEX events can be forwarded to TAU and collected in a TAU profile and/or trace to be used for post-mortem performance analysis. That process is not covered in this tutorial, but for more information, see [http://github.com/khuck/xpress-apex](http://github.com/khuck/xpress-apex).

Runtime Adaptation

APEX provides a mechanism for dynamic runtime feedback and control, either for autotuning or adaptation to changing environment. The infrastructure that provides the adaptation in APEX is the Policy Engine, which executes policies either periodically or triggered by events. The policies have access to the performance state as observed by the APEX introspection API. APEX is integrated with Active Harmony ([http://www.dyninst.org/harmony](http://www.dyninst.org/harmony)) to provide dynamic search for autotuning.

Fibonacci example - what's the performance?

Using the first fibonacci implementation from the HPX introduction, let's examine the performance. To get a simple text report of performance from HPX in a regular program, you would set the `APEX_SCREEN_OUTPUT` environment variable to a positive number (i.e. "1"). In the Jupyter notebook, we will use the `apex::dump(bool reset);` method instead. Because the HPX environment is continuously running in the Jupyter kernel, we also need to reset the timers before executing our test.

First, include the HPX header for our example, and declare some useful namespaces.

```cpp
#include <hpx/hpx.hpp>
using namespace std;
using namespace hpx;
```

Next, we will define the first implementation of the fibonacci algorithm from the previous presentation.
In [ ]: uint64_t fibonacci(uint64_t n)
{
    // if we know the answer, we return the value
    if (n < 2) return n;
    // asynchronously calculate one of the sub-terms
    future<uint64_t> f = async(launch::async, &fibonacci, n-2);
    // synchronously calculate the other sub-term
    uint64_t r = fibonacci(n-1);
    // wait for the future and calculate the result
    return f.get() + r;
}

And we will execute that definition:

In [ ]: .expr
    apex::reset(0L);
    cout << fibonacci(22) << endl;
    apex::dump(true);

We get some useful information, but what is task_object::apply? That is the HPX runtime executing asynchronous tasks. To get a useful label for that function (and distinguish the different task types), we will use hpx::util::annotated_function. Note carefully that we will add the hpx::util namespace, and also that we will change the name to fibonacci2, to distinguish from the previous definition (that still exists in this compilation unit). We will use that renaming pattern throughout this tutorial.

(markdown padding for proper formatting)

In [ ]: using namespace hpx::util;

In [ ]: uint64_t fibonacci2(uint64_t n)
{
    // if we know the answer, we return the value
    if (n < 2) return n;
    // asynchronously calculate one of the sub-terms
    future<uint64_t> f = async(launch::async, annotated_function(unwrapping(&fibonacci2), "fibonacci2 asynchronous"), n-2);
    // synchronously calculate the other sub-term
    apex::profiler *p = apex::start("fibonacci2 synchronous");
    uint64_t r = fibonacci2(n-1);
    apex::stop(p);
    // wait for the future and calculate the result
    return f.get() + r;
}

Next, we will execute the fibonacci2 method with the annotated function:
That's better, but not quite right. Note carefully that the fibonacci2 synchronous events are double-counting -- the synchronous timer for \( n \) includes the time spent computing both \( n-1 \) and \( n-2 \). Let's see what happens with the example that uses a serial cutoff:

```cpp
const int threshold = 10;

uint64_t fibonacci_serial(uint64_t n) {
    if (n < 2) return n;
    uint64_t f1 = fibonacci_serial(n-2);
    uint64_t f2 = fibonacci_serial(n-1);
    return f1 + f2;
}
```

```cpp
future<uint64_t> fibonacci3(uint64_t n) {
    if(n < 2) return make_ready_future(n);
    if(n < threshold) return make_ready_future(fibonacci_serial(n));

    future<uint64_t> f = async(launch::async, annotated_function(unwrapping(&fibonacci3), "fibonacci3"), n-2);
    future<uint64_t> r = fibonacci3(n-1);

    return dataflow(
        [](future<uint64_t> f1, future<uint64_t> f2) {
            return f1.get() + f2.get();
        },
        f, r);
}
```

Note that the timers are only around the calls where \( 10 > n \geq 22 \). We are now under-counting, because the synchronous executions aren't measured. If we were interested in a careful evaluation of the serial execution, we would include a third function with a timer that is called once. That function would then recursively call fibonacci_serial. Feel free to implement that version as an exercise.

What if we want to compare the different methods? We can put them all into one compilation unit, and execute them in the same cell for a clearer comparison:
In [ ]: 
```cpp
apex::reset(0L);
cout << fibonacci(22) << endl;
cout << fibonacci2(22) << endl;
cout << fibonacci3(22).get() << endl;
apex::dump(true);
```

Well, there is some useful data in there - but we need to do some aggregation. We could add them up, or we could wrap the top level calls with a timer and get the wall clock time for each approach:

In [ ]: 
```cpp
apex::reset(0L);
apex::profiler *p1 = apex::start("Version 1");
cout << fibonacci(22) << endl;
apex::stop(p1);
apex::profiler *p2 = apex::start("Version 2");
cout << fibonacci2(22) << endl;
apex::stop(p2);
apex::profiler *p3 = apex::start("Version 3");
cout << fibonacci3(22).get() << endl;
apex::stop(p3);
apex::dump(true);
```

**Heat Equation Example**
In [ ]:
```cpp
#include <hpx/include/parallel_algorithm.hpp>
#include <boost/range/irange.hpp>
#include <boost/format.hpp>

#include <cstddef>
#include <cstdint>
#include <iostream>
#include <memory>
#include <utility>
#include <vector>
#include <stdexcept>
#include <string>

void print_time_results(
    std::uint32_t num_localities,
    std::uint64_t num_os_threads,
    std::uint64_t elapsed,
    std::uint64_t nx,
    std::uint64_t np,
    std::uint64_t nt,
    bool header
)
{
    if (header)
```
std::cout << "Localities,OS_Threads,Execution_Time_sec,"
   "Points_per_Partition,Partitions,Time_Steps\n"
   << std::flush;

std::string const locs_str = boost::str(boost::format("%u," ) % num_localities);
std::string const threads_str = boost::str(boost::format("%lu," ) % num_os_threads);
std::string const nx_str = boost::str(boost::format("%lu," ) % nx);
std::string const np_str = boost::str(boost::format("%lu," ) % np);
std::string const nt_str = boost::str(boost::format("%lu " ) % nt);

std::cout << ( boost::format("%-6s %-6s %.14g, %-21s %-21s %-21s
"
   % locs_str % threads_str % (elapsed / 1e9) %nx_str % np_str
   % nt_str) << std::flush;
}

void print_time_results(
    std::uint64_t num_os_threads,
    std::uint64_t elapsed,
    std::uint64_t nx,
    std::uint64_t np,
    std::uint64_t nt,
    bool header)
{
    if (header)
    {
        std::cout << "OS_Threads,Execution_Time_sec,"
           "Points_per_Partition,Partitions,Time_Steps\n"
           << std::flush;

        std::string const threads_str = boost::str(boost::format("%lu," ) % num_os_threads);
        std::string const nx_str = boost::str(boost::format("%lu," ) % nx);
        std::string const np_str = boost::str(boost::format("%lu," ) % np);
        std::string const nt_str = boost::str(boost::format("%lu " ) % nt);

        std::cout << ( boost::format("%-21s %.14g, %-21s %-21s %-21s
"
           % threads_str % (elapsed / 1e9) %nx_str % np_str
           % nt_str) << std::flush;
    }

void print_time_results(
    std::uint64_t num_os_threads,
    std::uint64_t elapsed,
    std::uint64_t nx,
    std::uint64_t nt,
    bool header)
{
    if (header)
    {
        std::cout << "OS_Threads,Execution_Time_sec,"
           "Grid_Points,Time_Steps\n"
           << std::flush;

        std::string const threads_str = boost::str(boost::format("%lu," ) % num_os_threads);
        std::string const nx_str = boost::str(boost::format("%lu," ) % nx);
        std::string const np_str = boost::str(boost::format("%lu," ) % np);
        std::string const nt_str = boost::str(boost::format("%lu " ) % nt);

        std::cout << ( boost::format("%-21s %.14g, %-21s %-21s %-21s
"
           % threads_str % (elapsed / 1e9) %nx_str % np_str
           % nt_str) << std::flush;
    }
<< std::flush;

std::string const threads_str = boost::str(boost::format("%lu," ) % num_os_threads);
std::string const nx_str = boost::str(boost::format("%lu," ) % nx);
std::string const nt_str = boost::str(boost::format("%lu ") % nt);

std::cout << ( boost::format("%-21s %10.12s, %-21s %-21s
") % threads_str % (elapsed / 1e9) %nx_str % nt_str) << std::flush;
}

///////////////////////////////////////////////////////////////////////
// Command-line variables
bool header = true; // print csv heading
double k = 0.5; // heat transfer coefficient
double dt = 1.; // time step
double dx = 1.; // grid spacing

inline std::size_t idx(std::size_t i, int dir, std::size_t size)
{
    if(i == 0 && dir == -1)
        return size-1;
    if(i == size-1 && dir == +1)
        return 0;
    HPX_ASSERT((i + dir) < size);

    return i + dir;
}

///////////////////////////////////////////////////////////////////////
// Our partition data type
struct partition_data
{
    public:
    partition_data(std::size_t size)
        : data_(new double[size]), size_(size)
    {}

    partition_data(std::size_t size, double initial_value)
        : data_(new double[size]),
          size_(size)
    {
        double base_value = double(initial_value * size);
        for (std::size_t i = 0; i != size; ++i)
            data_[i] = base_value + double(i);
    }

    partition_data(partition_data && other)
        : data_(std::move(other.data_)), size_(other.size_)
    {}

    double& operator[](std::size_t idx) { return data_[idx]; }
double operator[](std::size_t idx) const { return data_[idx]; }

std::size_t size() const { return size_; }

private:
    std::unique_ptr<double[]> data_
    std::size_t size_
};

std::ostream& operator<<(std::ostream& os, partition_data const& c)
{
    os << "{";
    for (std::size_t i = 0; i != c.size(); ++i)
    {
        if (i != 0)
            os << ", ";
        os << c[i];
    }
    os << "}";
    return os;
}

struct stepper
{
    // Our data for one time step
    typedef hpx::shared_future<partition_data> partition;
    typedef std::vector<partition> space;

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

    // The partitioned operator, it invokes the heat operator above on all
    // elements of a partition.
    static partition_data heat_part(partition_data const& left,
        partition_data const& middle, partition_data const& right)
    {
        apex_wrapper profiler("partition_data::heat_part", 0L);
        std::size_t size = middle.size();
        partition_data next(size);
        next[0] = heat(left[size-1], middle[0], middle[1]);
        for(std::size_t i = 1; i != size-1; ++i)
        {
            next[i] = heat(middle[i-1], middle[i], middle[i+1]);
        }
        next[size-1] = heat(middle[size-2], middle[size-1], right[0]);
        return next;
    }
}
// do all the work on 'np' partitions, 'nx' data points each, for 'nt'
// time steps, limit depth of dependency tree to 'nd'
hpx::future<space> do_work(std::size_t np, std::size_t nx, std::size_t nt,
                          std::uint64_t nd)
{
    using hpx::dataflow;
    using hpx::util::unwrapping;

    // U[t][i] is the state of position i at time t.
    std::vector<space> U(2);
    for (space& s: U)
        s.resize(np);

    // Initial conditions: f(0, i) = i
    std::size_t b = 0;
    auto range = boost::irange(b, np);
    using hpx::parallel::execution::par;
    hpx::parallel::for_each(par, std::begin(range), std::end(range),
                [&U, nx](std::size_t i)
                {
                    U[0][i] = hpx::make_ready_future(partition_data(nx, double(i)));
                });

    // limit depth of dependency tree
    hpx::lcos::local::sliding_semaphore sem(nd);

    auto Op = hpx::util::annotated_function(unwrapping(&stepper::heat_part),
                          "stepper::heat_part");

    // Actual time step loop
    for (std::size_t t = 0; t != nt; ++t)
    {
        space const& current = U[t % 2];
        space& next = U[(t + 1) % 2];

        for (std::size_t i = 0; i != np; ++i)
        {
            next[i] = dataflow(
                Op, current[idx(i, -1, np)], current[i], current[i + dx(i, +1, np)]);
        }

        // every nd time steps, attach additional continuation which will
        // trigger the semaphore once computation has reached this point
        if ((t % nd) == 0)
        {
            next[0].then([&sem, t](partition &&)
void do_1d_solve(std::uint64_t np, std::uint64_t nx, std::uint64_t nt, std::uint64_t nd, bool results)
{
    header = false;

    // Create the stepper object
    stepper step;

    // Measure execution time.
    std::uint64_t t = hpx::util::high_resolution_clock::now();

    // Execute nt time steps on nx grid points and print the final solution.
    hpx::future<stepper::space> result = step.do_work(np, nx, nt, nd);
    stepper::space solution = result.get();
    hpx::wait_all(solution);

    std::uint64_t elapsed = hpx::util::high_resolution_clock::now() - t;

    // Print the final solution
    if (results)
    {
        for (std::size_t i = 0; i != np; ++i)
        {
            std::cout << "U[" << i << "] = " << solution[i].get() << std::endl;
        }
    }

    std::uint64_t const os_thread_count = hpx::get_os_thread_count();
}
Heat Equation with an APEX policy
In [ ]:
```
#include <hpx/hpx_init.hpp>
#include <hpx/hpx.hpp>

#include <hpx/include/parallel_algorithm.hpp>
#include <hpx/include/performance_counters.hpp>

#include <boost/range/irange.hpp>
#include <boost/format.hpp>

#include <algorithm>
#include <cstddef>
#include <cstdint>
#include <iostream>
#include <limits>
#include <memory>
#include <string>
#include <utility>
#include <vector>

#include <apex_api.hpp>

#include <boost/shared_array.hpp>

using hpx::naming::id_type;
using hpx::performance_counters::get_counter;
using hpx::performance_counters::stubs::performance_counter;
using hpx::performance_counters::counter_value;
using hpx::performance_counters::status_is_valid;

static bool counters_initialized = false;
static std::string counter_name = "/threads{locality#0/total}/idle-rate";
static apex_event_type end_iteration_event = APEX_CUSTOM_EVENT_1;
static hpx::naming::id_type counter_id;

void setup_counters() {
    try {
        id_type id = get_counter(counter_name);
        // We need to explicitly start all counters before we can use them. For
// certain counters this could be a no-op, in which case start
// will return
// 'false'.
    performance_counter::start(hpx::launch::sync, id);
    std::cout << "Counter " << counter_name << " initialized " <<
    id << std::endl;
    counter_value value = performance_counter::get_value(hpx::laun
    ch::sync, id);
    std::cout << "Counter value " << value.get_value<std::int64_t>
() << std::endl;
    counter_id = id;
    end_iteration_event = apex::register_custom_event("Repartition
n");
    counters_initialized = true;
}

    catch(hpx::exception const & e) {
        std::cerr << "1d_stencil_4_repart: caught exception: "
        << e.what() << std::endl;
        counter_id = hpx::naming::invalid_id;
        return;
    }
}

double get_counter_value() {
    if (!counters_initialized) {
        std::cerr << "get_counter_value(): ERROR: counter was not init
ialized" << std::endl;
        return false;
    }
    try {
        counter_value value1 =
            performance_counter::get_value(hpx::launch::sync, counter_ id, true);
        std::int64_t counter_value = value1.get_value<std::int64_t>();
        std::cout << "counter_value " << counter_value << std::endl;
        return (double)(counter_value);
    } catch(hpx::exception const & e) {
        std::cout << "get_counter_value(): caught exception: " << e.wh
at() << std::endl;
        return (std::numeric_limits<double>::max());
    }
}

///////////////////////////////////////////////////////////////////////

// Our partition data type
struct partition_data2
{
    public:
        partition_data2(std::size_t size)
            : data_(new double[size]), size_(size)
        {}

    private:
        std::size_t size_;
        double* data_;
```cpp
partition_data2(std::size_t size, double initial_value)
    : data_(new double[size]),
      size_(size)
{
    double base_value = double(initial_value * size);
    for (std::size_t i = 0; i != size; ++i)
        data_[i] = base_value + double(i);
}

partition_data2(std::size_t size, const double * other)
    : data_(new double[size]),
      size_(size)
{
    for(std::size_t i = 0; i != size; ++i) {
        data_[i] = other[i];
    }
}

partition_data2(partition_data2 && other)
    : data_(std::move(other.data_))
      , size_(other.size_)
{}

double& operator[](std::size_t idx) { return data_[idx]; }  // C++11
double operator[](std::size_t idx) const { return data_[idx]; }

void copy_into_array(double * a) const
{
    for(std::size_t i = 0; i != size(); ++i) {
        a[i] = data_[i];
    }
}

std::size_t size() const
{
    return size_;}

private:
    std::unique_ptr<double[]> data_
    , size_t size_
;          

std::ostream& operator<<(std::ostream& os, partition_data2 const& c)
{
    os << "{";
    for (std::size_t i = 0; i != c.size(); ++i)
    {
        if (i != 0)
            os << ", ";
        os << c[i];
    }
    os << "}";
    return os;
}

```

---

struct stepper2
{
    // Our data for one time step
    typedef hpx::shared_future<partition_data2> partition;
    typedef std::vector<partition> space;

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

    // The partitioned operator, it invokes the heat operator above on all
    // elements of a partition.
    static partition_data2 heat_part(partition_data2 const& left,
                                     partition_data2 const& middle, partition_data2 const& right)
    {
        std::size_t size = middle.size();
        partition_data2 next(size);

        if(size == 1) {
            next[0] = heat(left[0], middle[0], right[0]);
            return next;
        }

        next[0] = heat(left[size-1], middle[0], middle[1]);
        for(std::size_t i = 1; i < size-1; ++i)
        {
            next[i] = heat(middle[i-1], middle[i], middle[i+1]);
        }

        next[size-1] = heat(middle[size-2], middle[size-1], right[0]);
        return next;
    }

    // do all the work on 'np' partitions, 'nx' data points each, for 'nt'
    // time steps
    hpx::future<space> do_work(std::size_t np, std::size_t nx, std::size_t nt,
                                boost::shared_array<double> data)
    {
        using hpx::dataflow;
        using hpx::util::unwrapping;

        // U[t][i] is the state of position i at time t.
        std::vector<space> U(2);
        for (space& s: U)
        {
            s.resize(np);
        }

        if (!data) {
            // Initial conditions: f(0, i) = i
            std::size_t b = 0;
        }
    }
auto range = boost::irange(b, np);
using hpx::parallel::execution::par;

hpx::parallel::for_each(par, std::begin(range), std::end(range), [
    &U, nx](std::size_t i)
    {
        U[0][i] = hpx::make_ready_future(
            partition_data2(nx, double(i)));
    
    }
);

else {
    // Initialize from existing data
    std::size_t b = 0;
    auto range = boost::irange(b, np);
    using hpx::parallel::execution::par;
    hpx::parallel::for_each(par, std::begin(range), std::end(range), [
        &U, nx, data](std::size_t i)
        {
            U[0][i] = hpx::make_ready_future(
                partition_data2(nx, data.get() + (i * nx)));
        }
    );
    
    auto Op = hpx::util::annotated_function(unwrapping(&stepper2::heat_part),
        "stepper2::heat_part");
}

    // Actual time step loop
for (std::size_t t = 0; t != nt; ++t)
    {
        space const& current = U[t % 2];
        space& next = U[(t + 1) % 2];

        for (std::size_t i = 0; i != np; ++i)
            {
                next[i] = dataflow(
                    hpx::launch::async, Op,
                    current[idx(i, -1, np)], current[i], current[i]
                    dx(i, +1, np])
                    );
            }

    // Return the solution at time-step 'nt'.
    return hpx::when_all(U[nt % 2]);
}
};

////////////////////////////////////////////////////////////////////////////////

std::uint64_t np{10};  // Number of partitions.
std::uint64_t nx{10};  // Number of grid points. (local x dimensi
on of each partition)
    std::uint64_t nt{45};   // Number of steps.
    std::uint64_t nr{10};   // Number of runs to tune
*/
void do_1d_solve_repart(std::uint64_t nx,
    std::uint64_t nt, std::uint64_t nr, bool results)
{
    setup_counters();

    // Number of partitions dynamically determined    */
    header = false;
    std::uint64_t const
    os_thread_count = hpx::get_os_thread_count();

    // Find divisors of nx
    std::vector<std::uint64_t> divisors;
    // Start with os_thread_count so we have at least as many
    // partitions as we have HPX threads.
    for(std::uint64_t i = os_thread_count; i < std::sqrt(nx); ++i) {
        if(nx % i == 0) {
            divisors.push_back(i);
            divisors.push_back(nx/i);
        }
    } // This is not necessarily correct (sqrt(x) does not always evenly
    // and leads to partition size = 1 which we want to avoid
    //divisors.push_back(static_cast<std::uint64_t>(std::sqrt(nx))); //divisors.push_back(static_cast<std::uint64_t>(std::sqrt(nx)))
    std::sort(divisors.begin(), divisors.end());

    if(divisors.size() == 0) {
        std::cerr << "ERROR: No possible divisors for " << nx
        << " data elements with at least " << os_thread_count
        << " partitions and at least two elements per partition."
        << std::endl;
        return hpx::finalize();
    }

    //std::cerr << "Divisors: ";
    //for(std::uint64_t d : divisors) {
    //    std::cerr << d << " ";
    //}
    //std::cerr << std::endl;

    // Set up APEX tuning
    // The tunable parameter -- how many partitions to divide data int
    long np_index = 1;
    long * tune_params[1] = { 0L };
    long num_params = 1;
    long mins[1] = { 0 };
    long maxs[1] = { (long)divisors.size() };
    long steps[1] = { 1 };
    tune_params[0] = &np_index;
    apex::setup_custom_tuning(get_counter_value, end_iteration_event,
    num_params,
    tune_params, mins, maxs, steps);