HPX

A GENERAL PURPOSE C++ RUNTIME SYSTEM FOR PARALLEL AND DISTRIBUTED APPLICATIONS OF ANY SCALE
The Venture Point

TECHNOLOGY DEMANDS NEW RESPONSE
Technology Demands new Response
Technology Demands new Response

Tianhe-2’s projected theoretical peak performance: 54.9 PetaFLOPs
16,000 nodes, ~3,200,000 computing cores (32,000 Intel Ivy Bridge Xeons, 48,000 Xeon Phi Accelerators)
Amdahl’s Law (Strong Scaling)

\[ S = \frac{1}{(1 - P) + \frac{P}{N}} \]

- **S**: Speedup
- **P**: Proportion of parallel code
- **N**: Number of processors

Figure courtesy of Wikipedia (http://en.wikipedia.org/wiki/Amdahl's_law)
The 4 Horsemen of the Apocalypse: **SLOW**

**Starvation**
- Insufficient concurrent work to maintain high utilization of resources

**Latencies**
- Time-distance delay of remote resource services

**Overheads**
- Work for management of parallel actions and resources on critical path which are not necessary

**Waiting**
- Contention resolution
- Delays due to lack of availability of oversubscribed shared resources

**Impose upper bound on both weak and strong scaling**
The Challenges

We need to find a usable way to fully parallelize the applications

Goals are

- Defeat The Four Horsemen
- Provide manageable paradigms for handling parallelism
- Expose asynchrony to the programmer without exposing concurrency
- Make data dependencies explicit, hide notion of ‘thread’, ‘communication’, and ‘data distribution’
Runtime Systems

THE NEW DIMENSION
HPX – A General Purpose Runtime System

Solidly based on a theoretical foundation - ParalleX

- A general purpose runtime system for applications of any scale
  - [http://stellar.cct.lsu.edu/](http://stellar.cct.lsu.edu/)
  - [https://github.com/STEllAR-GROUP/hpx/](https://github.com/STEllAR-GROUP/hpx/)

Exposes an uniform, standards-oriented API for ease of programming parallel and distributed applications.

- Enables to write fully asynchronous code using hundreds of millions of threads.
- Provides unified syntax and semantics for local and remote operations.

Enables writing applications which outperform and out-scale existing ones

Is published under Boost license and has an open, active, and thriving developer community.
HPX – A General Purpose Runtime System

Governing principles

- Active global address space (AGAS) instead of PGAS
- Message driven instead of message passing
- Lightweight control objects instead of global barriers
- Latency hiding instead of latency avoidance
- Adaptive locality control instead of static data distribution
- Moving work to data instead of moving data to work
- Fine grained parallelism of lightweight threads instead of Communicating Sequential Processes (CSP/MPI)
HPX – The API

Fully asynchronous

- All possibly remote operations are asynchronous by default
  - ‘Fire & forget’ semantics (result is not available)
  - ‘Pure’ asynchronous semantics (result is available via hpx::future)
- Composition of asynchronous operations (N3634)
  - hpx::when_all, hpx::when_any, hpx::when_n
  - hpx::future::then(f)
- Can be used ‘synchronously’, but does not block
  - Thread is suspended while waiting for result
  - Other useful work is performed transparently
HPX – The API

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

- `std::thread` → `hpx::thread`
- `std::mutex` → `hpx::mutex`
- `std::future` → `hpx::future` (including N3634)
- `std::async` → `hpx::async`
- `std::bind` → `hpx::bind`
- `std::function` → `hpx::function`
- `std::tuple` → `hpx::tuple`
- `std::any` → `hpx::any` (N3508)
- `std::cout` → `hpx::cout`
- etc.
HPX – The API

Fully move enabled (using C++11 move semantics)
- `hpx::bind, hpx::function, hpx::tuple, hpx::any`

Fully type safe remote operation
- Extends the notion of a ‘callable’ to remote case (actions)
- Everything you can do with functions is possible with actions as well

Data types are usable in remote contexts
- Can be sent over the wire (`hpx::bind, hpx::function, hpx::any`)
- Can be used with actions (`hpx::async, hpx::bind, hpx::function`)

Unifies local and remote operation for the application programmer
- Object migration to other localities
The Future

WHERE DO WE GO?
What is a (the) future

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

- Enables transparent synchronization with producer
- Hides notion of dealing with threads
- Makes asynchrony manageable
- Allows for composition of several asynchronous operations
- Turns concurrency into parallelism
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
}
```
# HPX – The API

<table>
<thead>
<tr>
<th></th>
<th>Synchronous (return R)</th>
<th>Asynchronous (return future&lt;R&gt;)</th>
<th>Fire &amp; Forget (return void)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R f(p...)</strong></td>
<td>f(p...)</td>
<td>async(f, p...)</td>
<td>apply(f, p...)</td>
</tr>
<tr>
<td><strong>Functions</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(direct)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>f(p...)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Functions</strong></td>
<td>bind(f, p...)(...)</td>
<td>async(bind(f, p...), ...)</td>
<td>apply(bind(f, p...), ...)</td>
</tr>
<tr>
<td>(lazy)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Actions</strong></td>
<td>HPX_ACTION(f, a) a(id, p...)</td>
<td>HPX_ACTION(f, a) a(id, p...)</td>
<td>HPX_ACTION(f, a) a(id, p...)</td>
</tr>
<tr>
<td>(direct)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HPX_ACTION(f, a) a(id, p...)</td>
<td>HPX_ACTION(f, a) a(id, p...)</td>
<td></td>
</tr>
<tr>
<td><strong>Actions</strong></td>
<td>bind(a, id, p...)(...)</td>
<td>HPX_ACTION(f, a) bind(a, id, p...)(...)</td>
<td></td>
</tr>
<tr>
<td>(lazy)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HPX_ACTION(f, a) bind(a, id, p...)(...)</td>
<td>HPX_ACTION(f, a) bind(a, id, p...), ...</td>
<td></td>
</tr>
</tbody>
</table>
HPX – The API

Additional constructs for composing futures

° Sequential composition (attach continuation):
  
  ```cpp
  future<decltype(F(future<T>))>
  future<T>::then(F f);
  ```

° Parallel composition:
  
  ```cpp
  future<tuple<future<T>, ...>>
  when_all(future<T> f, ...);
  ```

  ```cpp
  future<tuple<future<T>, ...>>
  when_any(future<T> f, ...);
  ```

° Dataflow:
  
  ```cpp
  future<decltype(F(future<T> f, ...))>
  dataflow(F f, future<T> f, ...);
  ```
Futurization – An Example
Stupidest Way to Calculate Fibonacci Numbers

Synchronous way:

```c
// watch out: O(2^n)
int fibonacci_serial(int n)
{
    if (n < 2) return n;
    return fibonacci_serial(n - 1) + fibonacci_serial(n - 2);
}

cout << fibonacci_serial(10) << endl;    // will print: 55
```

---

Serial Calculation of Fibonacci Numbers
Stupidest Way to Calculate Fibonacci Numbers

Computational complexity is $O(2^n)$ – alright, however

This algorithm is representative for a whole class of applications
- Tree based recursive data structures
  - Adaptive Mesh Refinement – important method for wide range of physics simulations
  - Game theory
- Graph based algorithms
  - Breadth First Search

Characterized by very tightly coupled data dependencies between calculations
- But fork/join semantics make it simple to reason about parallelization

Let’s spawn a new thread for every other sub tree on each recursion level
Let’s Parallelize It – Adding Real Asynchrony

Calculate Fibonacci numbers in parallel (1st attempt)

```cpp
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;
}
```
Let’s Parallelize It – Adding Real Asynchrony

Fibonacci (1st Parallel Version)

Fibonacci - Scaling (1st Parallel Version)

HPX - A GENERAL PURPOSE RUNTIME SYSTEM FOR PARALLEL APPLICATIONS OF ANY SCALE
(HTTP://STELLAR.CCT.LSU.EDU/)

May 21st, 2014
Let’s Parallelize It – Adding Real Asynchrony

What’s wrong? While it does scale, it is still 100 times slower than the serial execution.

Creates a new future for each invocation of `fibonacci()` (spawns an HPX thread)
- Millions of threads with minimal work each
- Overheads of thread management (creation, scheduling, execution, deletion) are much larger than the amount of useful work
  - Future overheads: \(~1\mu s\) (Thread overheads: \(~400\text{ns}\) )
  - Useful work: \(~50\text{ns}\)

Let’s introduce the notion of granularity of work (grain size of work)
- The amount of work executed in one thread
Let’s Parallelize It –
Introducing Control of Grain Size

Parallel calculation, switching to serial execution below given threshold

```cpp
uint64_t fibonacci(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, &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;
}
```
Let’s Parallelize It – Introducing Control of Grain Size

Fibonacci(40), 12 Cores

Serial Threshold

Time [s]

Serial
Parallelized
Grain Size Control - The New Dimension

Parallelizing code introduces Overheads (SLOW)

Overheads are caused by code which

- is executed in the parallel version only
- is on the critical path (we can’t ‘hide’ it behind useful work)
- is required for managing the parallel execution
  - i.e. task queues, synchronization, data exchange
  - NUMA and core affinities

Controlling not only the amount of resources used but also the granularity of work is an important factor

Controlling the grain size of work allows finding the sweet-spot between too much overheads and too little parallelism
Futurization

Special technique allowing to automatically transform code

- Delay direct execution in order to avoid synchronization
- Turns ‘straight’ code into ‘futurized’ code
- Code no longer calculates results, but generates an execution tree representing the original algorithm
- If the tree is executed it produces the same result as the original code
- The execution of the tree is performed with maximum speed, depending only on the data dependencies of the original code

<table>
<thead>
<tr>
<th>Straight Code</th>
<th>Futurized Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>T func() {...}</td>
<td>future&lt;T&gt; func() {...}</td>
</tr>
<tr>
<td>rvalue: n</td>
<td>make_ready_future(n)</td>
</tr>
<tr>
<td>T n = func();</td>
<td>future&lt;T&gt; n = func();</td>
</tr>
<tr>
<td>future&lt;T&gt; n = async(&amp;func, ...);</td>
<td>future&lt;future&lt;T&gt; &gt; n = async(&amp;func, ...);</td>
</tr>
</tbody>
</table>
Let’s Parallelize It – Apply Futurization

Parallel way, futurize algorithm to remove suspension points

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

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

    return dataflow(
        [](future<uint64_t> f1, future<uint64_t> f2) {
            return f1.get() + f2.get();
        },
        f.get(), r);
}
```
future<uint64_t> fibonacci(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, &fibonacci, n-2).unwrap();
    future<uint64_t> r = fibonacci(n-1);

    return dataflow(
        [](future<uint64_t> f1, future<uint64_t> f2) {
            return f1.get() + f2.get();
        },
        f, r);
}
Let’s Parallelize It – Unwrap Argument Futures

```cpp
future<uint64_t> fibonacci(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, &fibonacci, n-2);
    future<uint64_t> r = fibonacci(n-1);

    return dataflow(
        unwrapped([](uint64_t r1, uint64_t r2) {
            return r1 + r2;
        }),
        f, r);
}
```

Guess what? – This is 10% faster than straight version!
So What’s the Deal?

Too much parallelism is as bad as is too little
  ◦ Sweet spot is determined by the Four Horsemen, mainly by contention

Granularity control is crucial
  ◦ Optimal grain size depends very little on number of used resources
  ◦ Optimal grain size is determined by the Four Horsemen, mainly by overheads, starvation, and latencies

Even problems with (very) strong data dependencies can benefit from parallelization

Doing more is not always bad
  ◦ While we added more overheads by futurizing the code, we still gained performance
  ◦ This is a result of the complex interplay of starvation, contention and overheads in modern hardware

Avoid explicit suspension as much as possible, prefer continuation style execution flow
  ◦ Dataflow style programming is key to managing asynchrony
Recent Results
N-Body Code based on LibGeoDecomp

![Weak Scaling Results for HPX N-Body Code](image)

*Single Xeon Phi, Futurized*
N-Body Code based on LibGeoDecomp
Mini-Ghost (SMP)
Mini-Ghost (distributed runs)
Conclusions
Conclusions

Be aware of the Four Horsemen

Embrace parallelism, it’s here to stay, avoid concurrency

Asynchrony is your friend if used correctly

Think in terms of data dependencies, make them explicit

Avoid thinking in terms of threads

Continuation style, dataflow based programming is key for successful parallelization

Granularity control allows to find ‘optimal’ mode of operation
Where to get HPX

Main repository: https://github.com/STELLAR-GROUP/hpx/ (Boost licensed)

Main website: http://stellar.cct.lsu.edu/

Mailing lists: hpx-users@stellar.cct.lsu.edu, hpx-devel@stellar.cct.lsu.edu

IRC channel: #stellar on freenode