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

Figure courtesy of Kunle Olukotun, Lance Hammond, Herb Sutter, and Burton Smith
Technology Demands new Response

Figure courtesy of Kunle Olukotun, Lance Hammond, Herb Sutter, and Burton Smith
Technology Demands new Response

A GENERAL PURPOSE C++ RUNTIME SYSTEM FOR PARALLEL AND DISTRIBUTED APPLICATIONS OF ANY SCALE (HTTP://STELLAR.CCT.LSU.EDU/)
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 access and services

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

**Waiting** for Contention resolution
- Delays due to lack of availability of oversubscribed shared resources
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>Type</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 (direct)</strong></td>
<td>bind(f, p...)(...)</td>
<td>async(bind(f, p...), ...)</td>
<td>apply(bind(f, p...), ...)</td>
</tr>
<tr>
<td><strong>Actions (direct)</strong></td>
<td>HPX_ACTION(f, a) a(id, p...)</td>
<td>HPX_ACTION(f, a) async(a, id, p...)</td>
<td>HPX_ACTION(f, a) apply(f, id, p...)</td>
</tr>
<tr>
<td><strong>Actions (lazy)</strong></td>
<td>HPX_ACTION(f, a) bind(a, id, p...)(...)</td>
<td>HPX_ACTION(f, a) async(bind(a, id, p...), ...)</td>
<td>HPX_ACTION(f, a) apply(bind(a, id, p...), ...)</td>
</tr>
</tbody>
</table>
Some Simple Examples

A CLOSER LOOK
Hello HPX World

#include <hpx/hpx.hpp>
#include <hpx/iostream.hpp>

int hpx_main()
{
    hpx::cout << "Hello HPX World!\n";
    return hpx::finalize();
}

int main(int argc, char* argv[])
{
    return hpx::init(argc, argv);
}
void say_hello()
{
    hpx::cout << "Hello HPX World from locality: "
    << hpx::get_locality_id() << "!\n";
}
HPX_PLAIN_ACTION(say_hello); // defines say_hello_action

int hpx_main()
{
    say_hello_action sayit;
    for (auto loc: hpx::find_all_localities())
        hpx::apply(sayit, loc);
    return hpx::finalize();
}
Hello HPX World

```cpp
int hpx_main()
{
    std::vector<hpx::future<void>> ops;
    say_hello_action sayit;

    for (auto loc: hpx::find_all_localities())
        ops.push_back(hpx::async(sayit, loc));

    hpx::wait_all(ops);
    return hpx::finalize();
}
```
Fibonacci Number Sequence

```cpp
int fibonacci(int n) {
    if (n < 2) return n;
    auto f = hpx::async(fibonacci, n-1);
    int r = fibonacci(n-2);
    return f.get() + r;
}
```
Fibonacci Number Sequence

```cpp
int fibonacci(int n);

HPXPLAIN_ACTION(fibonacci); // defines fibonacci_action
fibonacci_action fib;

int fibonacci(int n)
{
    if (n < 2) return n;
    hpx::id_type loc = hpx::find_here();
    hpx::future<int> f = hpx::async(fib, loc, n - 1);
    int r = f.get(loc, n - 2);
    return f.get() + r;
}
```
Recent Results
N-Body Code based on LibGeoDecomp
N-Body Code based on LibGeoDecomp
N-Body Code based on LibGeoDecomp

Weak Scaling Results for HPX and MPI N-Body Codes
(Host Cores only)

Performance in TFLOPS vs. Number of Nodes, 16 Cores Each
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: #stear on freenode