Collaboration on Legacy Support

HPX - Current State

XPRESS Workshop at LSU, July 10th, 2013
hkaiser@cct.lsu.edu
The 4 Horsemen of the Apocalypse: **SLOW**

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

- **Latencies**
  - Time-distance delay of remote access and services

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

- **Waiting**
  - Delays due to lack of availability of oversubscribed shared resource

**Impose upper bound on both weak and strong scaling**
HPX – A General Purpose Runtime System

• All examples in this talk are based on HPX
• A general purpose runtime system for applications of any scale
  • http://stellar.cct.lsu.edu/
  • http://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.
• 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)
• Current version of HPX provides the following infrastructure on conventional systems as defined by the ParalleX execution model:
  • Active Global Address Space (AGAS)
  • HPX Threads and Thread Management
  • Parcel Transport and Parcel Management
  • Local Control Objects (LCOs)
  • HPX Processes (distributed objects)
  • Namespace and policies management, locality control
  • Monitoring subsystem
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`
  - `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 Boost.Move)
  • 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)

• 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 (N3428)
    • hpx::when_all, hpx::when_any, hpx::when_n
    • hpx::future::then(f)
  • Can be used ‘synchronously’, but do not block
    • Thread is suspended while waiting for result
# 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></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Functions (direct)</strong></td>
<td>f(p...)</td>
<td>async(f, p...)</td>
<td>apply(f, p...)</td>
</tr>
<tr>
<td><strong>Functions (lazy)</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>

C++

C++ Library
What is a (the) future

- A (std) 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
}
```
Example of using Futures

• Calculate Fibonacci numbers in parallel

```c
uint64_t fibonacci(uint64_t n)
{
    // if we know the answer, we return the value
    if (n < 2) return n;

    // asynchronously delay-calculate one of the sub-terms
    future<uint64_t> f = async(&fibonacci, n-1);

    // synchronously calculate the other sub-term
    uint64_t r = fibonacci(n-2);

    // wait for the future and calculate the result
    return f.get() + r;
}
```
HPX & OpenMP

• HPX makes a perfect backbone for an OpenMP implementation
  • Most of OpenMP functionality maps directly onto HPX concepts (threads, tasks, etc.)
HPX & MPI

• MPI is two things
  • Highly efficient and widely supported communication platform (send, receive, isend, ireceive)
  • Programming model (broadcast, all_reduce, data types, etc.)
• We should reuse communication layer
  • New parcelport for HPX
• We should re-implement higher level MPI functions on top of HPX