Math 4997-3

Lecture 11: Introduction to HPX

Patrick Diehl

https://www.cct.lsu.edu/~pdiehl/teaching/2021/4997/

Reminder

Lecture 10

What you should know from last lecture

- Conjugate Gradient method
- Solving equation systems using BlazeIterative

What is HPX

Description of HPX[1,2]

HPX (High Performance ParalleX) is a general purpose C++ runtime system for parallel and distributed applications of any scale. It strives to provide a unified programming model which transparently utilizes the available resources to achieve unprecedented levels of scalability. This library strictly adheres to the C++11 Standard and leverages the Boost C++ Libraries which makes HPX easy to use, highly optimized, and very portable.

[1] https://github.com/STEllAR-GROUP/hpx

[2] https://stellar-group.github.io/hpx/docs/sphinx/branches/master/html/index.html

HPX's features

- HPX exposes a uniform, standards-oriented API for ease of programming parallel and distributed applications.
- HPX provides unified syntax and semantics for local and remote operations.
- HPX exposes a uniform, flexible, and extendable performance counter framework which can enable runtime adaptivity
- HPX has been designed and developed for systems of any scale, from hand-held devices to very large scale systems (Raspberry Pi, Android, Server, up to super computers).

Compilation and running

Compilation and running

CMake

```
cmake_minimum_required(VERSION 3.3.2)
project(my_hpx_project CXX)
find_package(HPX REQUIRED)
add_hpx_executable(my_hpx_program
    SOURCES main.cpp
)
```

Running

```
cmake .
make
./my_hpx_program --hpx:threads=4
```

Hello World

A small HPX program

C++

```
int main()
{
    std::cout << "Hello World!\n" << hpx::flush;
    return 0;
}
```

HPX

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

int main()
{
    std::cout << "Hello World!\n" << std::endl;
    return 0;
}
```

Hello world using `hpx::init`

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

int hpx_main(int, char**)
{
    // Say hello to the world!
    std::cout << "Hello World!\n" << std::endl;
    return hpx::finalize();
}

int main(int argc, char* argv[])
{
    return hpx::init(argc, argv);
}
```

Note that here we initialize the HPX runtime explicitly.

Asynchronous programming

Futurization[3]

```
#include <hpx/hpx_init.hpp>
#include <hpx/incldue/lcos.hpp>

int square(int a)
{
    return a*a;
}

int main()
{
    hpx::future<int> f1 = hpx::async(square,10);

    hpx::cout << f1.get() << hpx::flush;

    return EXIT_SUCCESS;
}
```

Note that we just replaced `std` by the namespace `hpx`

[3] Example: hpx::async

Advanced synchronization[4]

```
std::vector<hpx::future<int>> futures;

futures.push_back(hpx::async(square,10);
futures.push_back(hpx::async(square,100);

hpx::when_all(futures).then([](auto&& f){
 auto futures = f.get();
 std::cout << futures[0].get()
         << " and " << futures[1].get();
});
```

[4] Documentation: hpx::when_all

Synchronization[5]

- `when_all`
  It *AND*-composes all the given futures and returns a new future containing all the given futures.
- `when_any`
  It *OR*-composes all the given futures and returns a new future containing all the given futures.
- `when_each`
  It *AND*-composes all the given futures and returns a new future containing all futures being ready.
- `when_some`
  It *AND*-composes all the given futures and returns a new future object representing the same list of futures after n of them finished.

---

[5] Documentation: LCO

Parallel algorithms

Example: Reduce

C++

```
#include<algorithm>
#include<execution>

std::reduce(std::execution::par,
        values.begin(),values.end(),0);
```

HPX

```
#include<hpx/include/parallel_reduce.hpp>
#include<vector>

hpx::ranges::reduce(
        hpx::execution::par,
                values.begin(),values.end(),0);
```

Example: Reduce with future

```
auto f =

hpx::ranges::reduce(
        hpx::execution::par(
                hpx::execution::task),
        values.begin(),
        values.end(),0);

std::cout<< f.get();
```

- `hpx::execution::par` Parallel execution
- `hpx::execution::seq` Sequential execution
- `hpx::execution::task` Task-based execution

Execution parameters

```
#include<hpx/include/parallel_executor_parameters.hpp>

hpx::execution::static_chunk_size scs(10);
hpx::ranges::reduce(
        hpx::execution::par.with(scs),
        values.begin(),
        values.end(),0);
```

- `hpx::execution::static_chunk_size`
  Loop iterations are divided into pieces of a given size and then assigned to threads.
- `hpx::execution::auto_chunk_size`
  Pieces are determined based on the first 1% of the total loop iterations.
- `hpx::execution::dynamic_chunk_size`
  Dynamically scheduled among the cores and if one core finished it gets dynamically assigned a new chunk.

Example: Range-based for loops

```
#include<vector>
#include<iostream>
#include<hpx/include/parallel_for_loop.hpp>

std::vector<double> values = {1,2,3,4,5,6,7,8,9};

hpx::for_loop(
        hpx::execution::par,
        0,
        values.size();
        [](boost::uint64_t i)
                {
                std::cout<< values[i] << std::endl;
                }
        );
```

Summary

Summary

After this lecture, you should know

- What is HPX
- Asynchronous programming using HPX
- Shared memory parallelism using HPX