HCLib Tutorial

Zoran Budimlić
Rice University
zoran@rice.edu

Acknowledgments

- Vincent Cavé
- Sanjay Chatterjee
- Max Grossman
- Vivek Kumar
- Deepak Majeti
- Vivek Sarkar
- Alina Sbîrlea
- Jonathan Sharman
- Sağnak Taşırlar
- Nick Vrvilo
- Yonghong Yan

What is HCLib?

- Task-parallel programming library
  - A C/C++ library implementation of Habanero programming constructs
  - Shares ideas with HJlib (a Java library implementation of Habanero constructs)
  - Supports both C and C++ programming interfaces
  - C++ APIs are based on C++11 lambda functions
  - Supports distributed computing
  - Supports heterogeneous (GPU) computing

Parallel Programming Supported by HCLib

- Dynamic task creation & termination
  - async, forasync, finish
- Data flow programming using futures and promises
  - promise_create, put, get_future
  - async_await, forasync_await
  - async_future, forasync_future
- Support for affinity control
  - Abstract Platfom model (a graph of the execution platform)
  - “Pop” and “steal” paths through the graph
- Integration with distributed programming
  - OpenSHMEM, MPI, UPC++
- Integration with heterogeneous (GPU) programming
  - CUDA
- Easily extensible using the Resource Workers

HCLib Documentation

- http://habanero-rice.github.io/hclib/
  - currently: https://budimlic.github.io/hclib/

Downloading HCLib

- `git clone https://github.com/habanero-rice/hclib.git`
  - currently: `https://github.com/budimlic/hclib.git`
- Dependencies
  - automake
  - gcc >= 4.8.4, or clang >= 3.5 (must support -std=c11 and -std=c++11)
  - libxml2 (with development headers)
  - jsmn JSON parser
  - Modules
    - OpenSHMEM, MPI, UPC++, CUDA

Installing jsmn

- Get jsmn: `git clone https://github.com/zserge/jsmn.git`
- cd to the jsmn directory
- `CFLAGS=-fPIC make`
- set the `JSMN_HOME` variable to the directory where jsmn is installed

Installing OpenSHMEM module

- Dependencies
  - autoconf, automake, libtool, UCX, OpenMPI
- Make sure $(CC) environment variable is pointing to GCC 4.8.4+ or Clang 3.5+
- There's a handy script in:
  - `hclib/scripts/oshmem.ompi.install.sh`
  - Installs all dependencies and OpenSHMEM in /home/user/OpenSHMEM-UCX/usr
- Add /home/user/OpenSHMEM-UCX/usr/bin to your $PATH
- Set $(OPENSHMEM_INSTALL) to /home/user/OpenSHMEM-UCX/usr
- `cd hclib/modules/openshmem; make`
- `cd hclib/modules/system; make`
- Add modules/system/lib and modules/openshmem/lib to $(LD_LIBRARY_PATH)

Installing HCLib

- Make sure your dependencies are downloaded and built
- Make sure $(CC) and $(CXX) environment variables are pointing to GCC 4.8.4+ or Clang 3.5+
- `cd hclib/`
- `./install.sh`
- This will install HCLib in hclib_install in current directory, you can pick another location:
  - `INSTALL_PREFIX=/path/to/hclib/install ./install.sh`
- Set `HCLIB_ROOT` environment variable to the directory where HCLib is installed

Launching HCLib

```
void hclib_launch(async_fct_t fct_ptr, void * arg, const char
**deps, int ndeps);
```

```
int main(int argc, char ** argv) {
    char const *deps[] = { "system" };
    hclib_launch(entrypoint, NULL, deps, 1); //HCLib program running
    . . . // HCLib program is finished, back to sequential code
```

Async and Finish Statements for Task Creation and Termination

async S

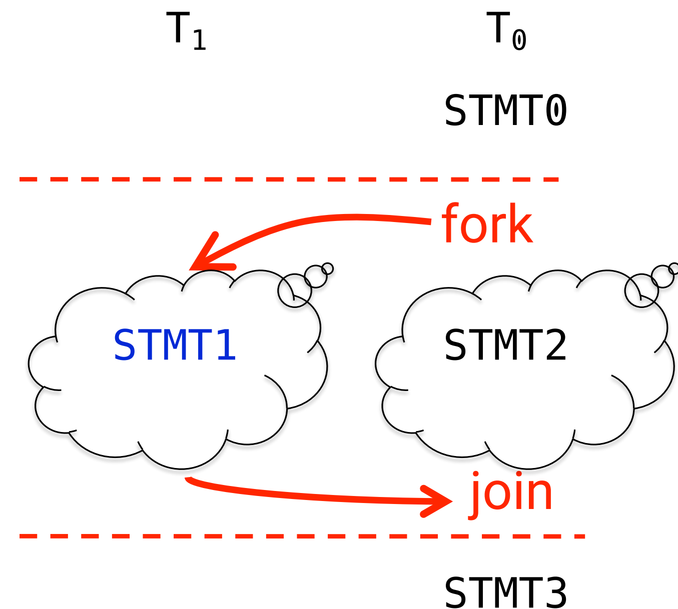
- Creates a new child task that executes statement S

```
// T0(Parent task)
STMT0;
finish {    //Begin finish
  async {
    STMT1; //T1(Child task)
  }
  STMT2;    //Continue in T0

}           //End finish (wait for T1)
STMT3;      //Continue in T0
```

finish S

- Execute S, but wait until *all* asyncs in S's scope have terminated.



Async and Finish using the C interface

```
void async_fct(void * arg) {<Stmt>}



hclib_async(async_fct, NULL, NO_FUTURE, 0, ANY_PLACE);


hclib_start_finish();
hclib_end_finish();
```

Async using the C interface

```
void hclib_async(generic_frame_ptr fp, void *arg,
        hclib_future_t **futures, const int nfutures,
        hclib_locale_t *locale);
```

Async and Finish using the C++ interface

```
hclib::async ( [capture_list] ( ) {
   <Stmt>
});

hclib::finish ([capture_list] ( ) {
   <Stmt>
});
```

Async and Finish using the C++ interface

```
using namespace hclib;

async ( [capture_list] ( ) {
   <Stmt>
});

finish ([capture_list] ( ) {
   <Stmt>
});
```

Fibonacci using async and finish in C++

```
1. #include "hclib.h"
2. using namespace hclib;
3. int fib(int n) {
4.    if(n <= THRESHOLD) {
5.       return fib_serial(n);
6.    }
7.    else {
8.       int x, y;
9.       finish ([n, &x, &y]( ) {
10.          async ([n, &x]( ) {
11.             x = fib(n-1);
12.          });
13.          y = fib(n-2);
14.       });
15.       return x + y;
16.    }
17. }
```

Parallel-for loop in HCLib

```
for ( int i = low; i < high; i+=stride ) { <Stmt> }
```

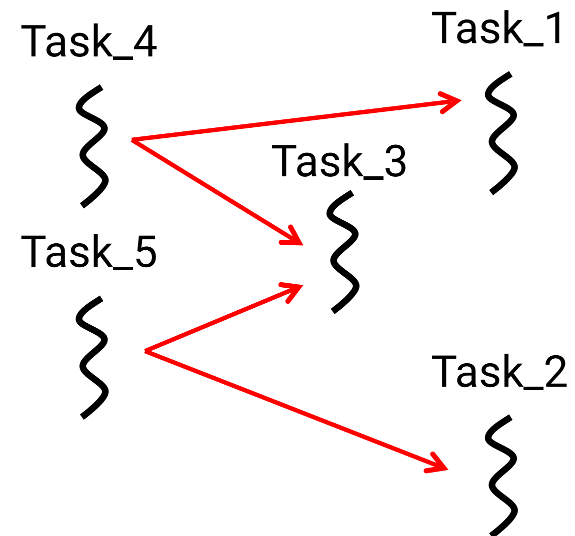
```
hclib::loop_domain_t loop = {low, high, stride, tile_size};
hclib::forasync1D (loop_domain_t* loop, [capture_list] ( int i )
{
    <Stmt>
}, int mode);
```

Internally uses hclib async to launch iterations

See also: forasync2D, forasync3D

Productivity Benefits of Dataflow Programming

- Producer-consumer relationship between tasks ?



Promises, Futures and Future-driven Tasks

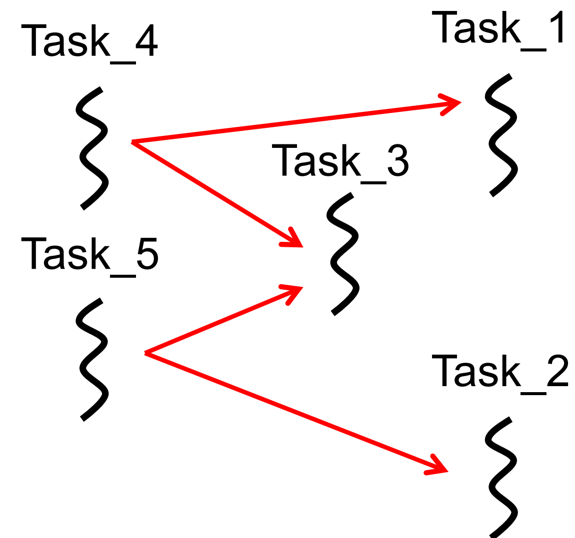
- **Promise**
  - An object that serves as a data container
  - Promises are write-only
- **Future**
  - A read-only handle to a promise's data
  - Can be used to block on the satisfaction of that handle
  - Can express a task dependency on that promise
- **Future-driven task**
  - A task whose execution is predicated on the satisfaction of some promise
  - This dependency is expressed using the future associated with that promise

Promises, Futures and Future-driven Tasks

- Creation
  - Create an empty promise object container for an integer
    - `promise_t<int> *promise = new promise_t<int>();`
- Resolution (put)
  - Write to the future object associated with a promise
    - `promise->put(V); //Single Assignment`
- Get the value stored in the future object of a promise
  - `promise->get_future()->get()`
- Future-driven Tasks
  -
```
hclib::async_await([capture_list] ( ) {
            <Stmt>
      }, future1, …, future4);
```

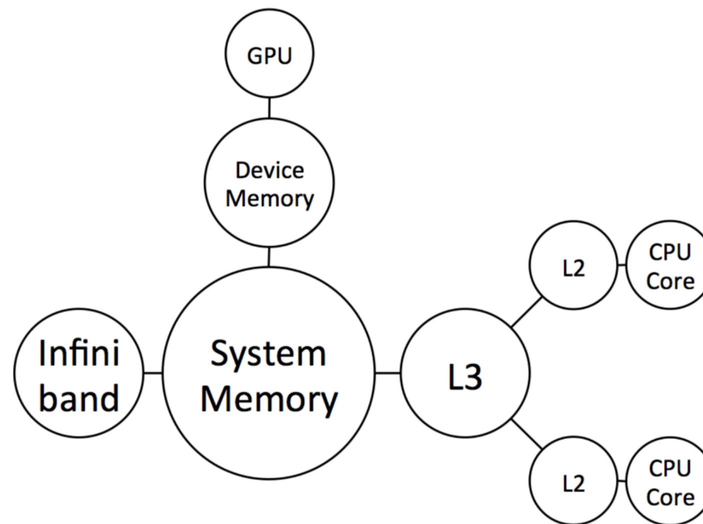
Promises, Futures and Future-driven Task

```
1.  promise_t<int> *left = new promise_t<int>();
2.  promise_t<int> *right = new promise_t<int>();
3.  finish([=]() {
4.      async_await([=]() {
5.          task_1(left->get_future()->get());
6.      }, left->get_future());
7.      async_await([=]() {
8.          task_2(right->get_future()->get());
9.      },right->get_future());
10.     async_await([=]() {
11.         task_3(left->get_future()->get(),
12.                 right->get_future()->get());
13.     }, left->get_future(), right->get_future());
14.     async([=]() {
15.         task_4();
16.         left->put(value);
17.     });
18.     async([=]() {
19.         task_5();
20.         right->put(value);
21.     });
22. });
```



Abstract Platform Model

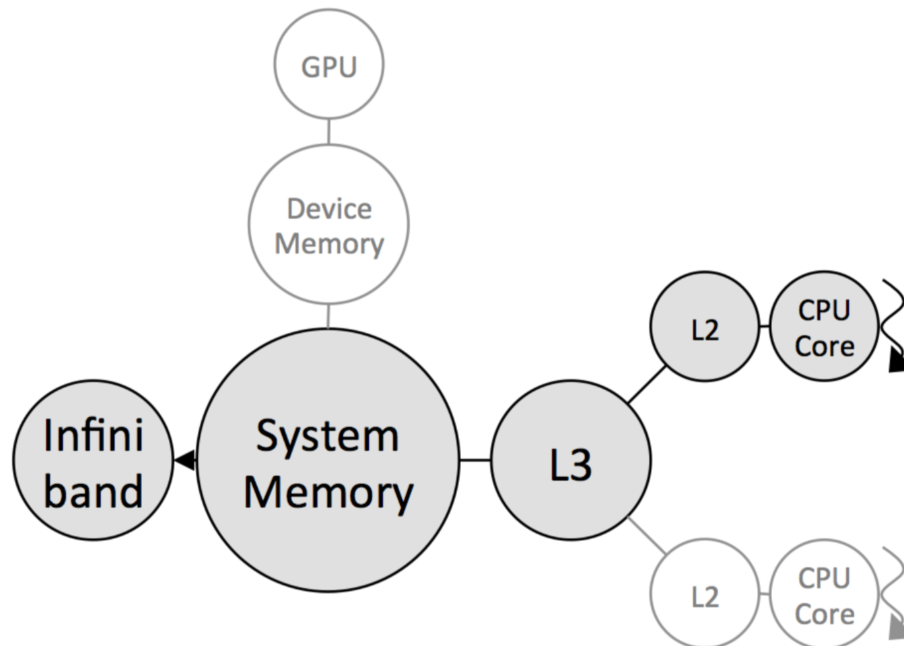
- A generalization of the Habanero Place Trees
- Allows locality control
- Allows different modules to interact with the HCLib tasking runtime



"A Pluggable Framework for Composable HPC Scheduling Libraries". Max Grossman, Vivek Kumar, Nick Vrvilo, Zoran Budimlić, and Vivek Sarkar. The Seventh International Workshop on Accelerators and Hybrid Exascale Systems (AsHES), May 2017.

Abstract Platform Model

You can define a "pop" and "steal" path for every worker thread in the HCLib runtime



An example of the Abstract Platform Model

```
{
    "nworkers": 12,
    "declarations": [
        "sysmem",
        "L2_0_0", "L2_0_1", "L2_0_2", "L2_0_3", "L2_0_4", "L2_0_5",
        "L2_1_0", "L2_1_1", "L2_1_2", "L2_1_3", "L2_1_4", "L2_1_5",
        "Interconnect"
    ],
    "reachability": [
        ["sysmem", "L2_0_0"], ["sysmem", "L2_0_1"], ["sysmem", "L2_0_2"],
            ["sysmem", "L2_0_3"], ["sysmem", "L2_0_4"], ["sysmem", "L2_0_5"],
        ["sysmem", "L2_1_0"], ["sysmem", "L2_1_1"], ["sysmem", "L2_1_2"],
            ["sysmem", "L2_1_3"], ["sysmem", "L2_1_4"], ["sysmem", "L2_1_5"],
        ["sysmem", "Interconnect"]
    ],
    "pop_paths": {
        "default": ["L2_$(id / 6)_$(id % 6)", "sysmem"],
        0: ["L2_0_0", "sysmem", "Interconnect"]
    },
    "steal_paths": {
        "default": ["L2_$(id / 6)_$(id % 6)", "sysmem"],
        0: ["L2_0_0", "sysmem", "Interconnect"]
    }
}
```

Using the Abstract Platform Model for Distributed Execution

```
const char *deps[] = { "system", "openshmem" };
hclib::launch(deps, 2, [argc, argv] {
. . .
// hclib::shmem… code for communication
// HCLib code (finish, async, futures, promises etc.) for intra-node
// parallelism
. . .
});
```

Interaction between communication and tasking

```
void shmem_async_when(mem_addr, wait_for_val, [=] {
    <STMT>
})
```

- Wait for a remote put into `mem_addr` of a value `wait_for_val`, then launch the task

- Currently implemented:

```
void shmem_int_async_when(volatile int *ivar, int cmp, int
cmp_value, lambda)
```

TODOs

- Make the `async` interface when using phasers conform to the current C and C++ APIs.
  - Use Sanjay's standalone phaser library instead?
  - Add the phaser tests to the test suite
- **Allow arbitrary number of futures (or an array of futures) for `async_await`**
- More general implementation of `shmem_async_when`
- Get rid of install.sh
- **Add the regular futures to documentation (get and wait)**
- Testing forasync performance (vs OpenMP & Cilk)
- Testing dataflow performance (vs Legion & OpenMP task dependence)
  - Traversing a graph and creating an async per node
- Focus on C++11 API's in the first release