Habanero-C

The Habanero-C (HC) language under development in the Habanero project at Rice University builds on past work on Habanero-Java, which in turn was derived from X10 v1.5

Overview

The Habanero-C (HC) language under development in the Habanero project at Rice University builds on past work on Habanero-Java, which in turn was derived from X10 v1.5. HC serves as a research testbed for new compiler and runtime software technologies for extreme scale systems for homogeneous and heterogeneous processors. Like HJ, HC implements an execution model for multicore processors based on four orthogonal dimensions for portable parallelism:

1. Lightweight dynamic task creation and termination using *async, finish, data-driven futures.*
2. Collective and point-to-point synchronization using *phasers*
3. Mutual exclusion and isolation using *isolated (not yet supported)*
4. Locality control using *hierarchical place trees*

Unlike HJ which needs a JVM to run, Habanero-C is designed to be mapped onto hardware platforms with lightweight system software stacks, such as the Customizable Heterogeneous Platform (CHP) being developed in the NSF Expeditions Center for Domain-Specific Computing (CDSC) which includes CPUs, GPUs, and FPGAs. The C foundation also makes it easier to integrate HC with communication middleware for cluster systems, such as MPI and GASNet.

The Habanero-C compiler is written in C++ and is built on top of the ROSE compiler infrastructure, which was also used in the DARPA-funded PACE project at Rice University. The bulk of the Habanero-C runtime has been written from scratch in portable ANSI C. However, a few library routines for low-level synchronization and atomic operations are written in assembly language for the target platform. To date, the Habanero-C runtime has been ported and tested on Intel X86, Cyclops 64, Power7, Sun Niagara 2 and Intel SCC multicore platforms.

A short summary of the HC language is included below. Details on the underlying implementation technologies can be found in the Habanero publications web page. The HC implementation is still evolving at an early stage. If you would like to try out HC, please contact one of the following people: Zoran Budimli, Vincent Cave, or Vivek Sarkar.

- Habanero-C has two basic primitives for the task parallel programming model borrowed from X10: *async* and *finish*. The *async* state ment, *async* <*stmt*>, causes the parent task to fork a new child task that executes <*stmt*>. Execution of the *async* statement returns immediately, i.e., the parent task can proceed to its following statement without waiting for the child task to complete. The *finish* state ment, *finish* <*stmt*>, performs a join operation that causes the parent task to execute <stmt> and then wait until all the tasks created within <*stmt*> have terminated (including transitively spawned tasks). The Habanero-C runtime uses a work-stealing scheduler that supports work-first and help-first policies along with places for locality
- Habanero-C uses *phasers* for synchronization. Phasers are programming constructs that unify collective and point-to-point synchronization in task parallel programming. Phasers are designed for ease of use and safety, helping programmer productivity in task parallel programming and debugging. The use of phasers guarantees two safety properties: deadlock-freedom and phase-ordering. These properties, along with the generality of its use for dynamic parallelism, distinguish phasers from other synchronization constructs such as barriers, counting semaphores and X10 clocks. In Habanero-C tasks can register on a phaser in on of the 3 modes: SIGNAL_WAIT_MODE, SIGNAL_ONLY_MODE, WAIT_ONLY_MODE.
- For locality, Habanero-C uses *Hierarchical Place Trees(HPTs)*. HPTs abstract the underlying hardware using hierarchical trees, allowing the program to spawn tasks at *places*, which for example could be cores, groups of cores sharing cache, nodes, groups of nodes, or other devices such as GPUs or FPGAs. The work-stealing runtime takes advantage of the hardware hierarchy to preserve locality when executing tasks.

HC Language Summary

Task Creation and Termination

async [(place)] [IN (var1, var2, ...)] [OUT (var1, var2, ...)] [INOUT (var1, var2, ...)]

[AWAIT (ddf1, ddf2, ...)] [phased] Stmt

- Asynchronously start a new task to execute Stmt in parallel with the parent. A destination place can optionally be specified for where the task should execute. The place can be obtained from the runtime using HC runtime functions (see HPT).

- Any local variable declared in an outer scope that is used in the async has to be specified in an IN (for variables read by the async), OUT(for variables written by the async), or INOUT(for variables both read and written by the async) clauses. The IN/OUT/OUT clauses have copy-in/copy-out semantics for local variables; selected variables are copied in from the parent scope at the start of the async, and out into the parent scope at the end of the async task.

- an AWAIT clause can optionally be specified, listing all the data-driven futures (DDF's) that the task should wait on before starting its execution.

- a phased clause can optionally be specified, registering the async on all the phasers specified in the list (ph1, ph2, ...), or on all the phasers of the parent (if the list is not specified).

finish Stmt

- execute Stmt, but wait until all (transitively) spawned asyncs in Stmt's scope have terminated before advancing to the next statement.

Data-Driven Task Synchronization

DDF_CREATE() --- a library function that creates a Data-Driven Future (DDF), and returns a pointer to a DDF_t type. A DDF is a single-assignment container that is initially empty, and becomes full after a DDF_PUT operation is performed on it.

DDF_GET(DDF_t * ddf) --- if ddf is full, return the value stored in ddf's container. If ddf is empty, the runtime will exit with an assertion failure.

DDF_PUT(DDF_t * ddf, void * value) --- if ddf is empty, perform a put of value into ddf. If ddf already has a value, the runtime will exit with an assertion failure.

async AWAIT (ddf1, ddf2, ...) Stmt --- wait until all the DDF's in the list (ddf1, ddf2, ...) have their values filled in before asynchronously starting the execution of Stmt. Stmt can safely perform a GET on the DDF's specified in the list.

Phaser Synchronization

phaser *ph = PHASER_CREATE(mode) --- create a phaser and register the calling task on the phaser with the specified mode.

async phased Stmt --- register the async with all phasers created by the parent in the immediate enclosing finish scope
and asynchronously execute Stmt

async phased SIGNAL_ONLY(ph1, ...) WAIT_ONLY(ph2, ...) SIGNAL_WAIT(ph3, ...) Stmt

-- register an async on specific phasers with specific modes. The parent should be registered on all the phasers in modes that are greater than or equal to the modes of the child as shown below.

SIGNAL_WAIT > SIGNAL_ONLY
SIGNAL_WAIT > WAIT_ONLY
SIGNAL_ONLY = WAIT_ONLY

NEXT--- synchronize on all the phasers that the task is registered on.

Data Parallel Loops

forasync [in (var1, var2, ...)] [point (ind1, ind2, ...)] [size (siz1, siz2, ...)] [seq (seq1, seq2, ...)] Body

-- The semantics of the in clause is the same as in the async case.

-- Loop indices in each dimension are specified by the point clause.

-- The number of iterations in each dimension is specified by the size clause.

-- The tile size is specified by the seq clause.

forasync is lowered and implemented for CPUs in two different ways as follows.

1) Chunked Scheduling: Loop iterations are chunked into blocks of lengths specified by the seq clause.(Default option)

2) Recursive Scheduling: Loop iterations are recursively partitioned until the size of a block size specified by the seq clause is reached. This is similar to the TBB style. (use '-hcc:recursive' option when compiling your program)

forasync targets **heterogeneous platforms** by automatically generating host code and OpenCL device code.

Note: The semantics of forasync does not include a barrier. An explicit finish must enclose the forasync to synchronize all the iterations.

Hierarchical Place Trees (HPT's)

A Hierarchical Place Tree (HPT) is an optional abstraction of the memory hierarchy that a Habanero-C program is executed on, specified as an XML document that conforms with hpt.dtd. If an HPT is not specified, the HC systems assumes a single-level hierarchy consisting of the specified number of workers. For example, one possible .xml file for an 8-core node is as follows:

```
<?xml version="1.0"?>
<!DOCTYPE HPT SYSTEM "hpt.dtd">
<HPT version="0.1" info="an HPT for SUGAR, 2 quad core2 Intel Xeon processors">
   <place num="1" type="mem">
      <place num="2" type="cache"> <!   2 sockets   >
         <place num="2" type="cache"> <!   2 L2 cache per socket   >
            <place num="2" type="cache"> <!   2 L1 cache per L2  >
               <worker num="1"/>
            </place>
         </place>
      </place>
   </place>
</HPT>
```

The HPT is specified using the -hpt option when invoking the HC executable. For example, the following cmmands creates an HC runtime instance with 8 workers organized in accordance with the specified HPT.

./a.out -nproc 8 -hpt sugar.xml

The following API calls enable an HC program to navigate the HPT:

short is_cpu_place(place_t * pl) --- is pl a CPU place?

short is_device_place(place_t * pl) --- is pl a device (GPU or FPGA) place?

short is_nvgpu_place(place_t * pl) --- is pl a NVIDIA GPU place?

place_t* hc_get_current_place() --- get the place where the task is currently executing

int hc_get_num_places(short type) --- get the number of places of the specified type (NVGPU_PLACE, MEM_PLACE or FPGA_PLACE)

void hc_get_places(place_t ** pls, short type) --- get an array of all the places of the specified type

place_t * hc_get_place(short type) --- get any place of the specified type

place_t * get_ancestor_place(hc_workerState * ws)

place_t * hc_get_child_place() --- get the child place on the path from the current place to the leaf place of the current worker

place_t * hc_get_parent_place() --- get the parent place of the current place in HPT

place_t ** hc_get_children_places(int * numChildren) --- get an array of all the child places of the current place

---

Current HC limitations

There are some limitations and pitfalls in the current implementation of the HC programming model. These limitations are not inherent to the programming model, but rather are a result of incompleteness in the current compiler or runtime implementation.

1) *Pointers to stack variables* (including stack-allocated arrays) cannot be reused across "suspendable" points. A suspendable function is a function that can directly or indirectly call a function containing an async statement or a finish statement. A suspendable point is an async statement, the end of a finish statement, or a call to a suspendable function.

Work-around: copy these stack variables to the heap. There is no limitation on the reuse of heap pointers across suspendable points.

2) *Pointers to HC functions* (functions that contain HC constructs or call other HC functions) are not supported in HC.

Work-around: only use pointers to C functions.

3) *const modifiers* are not supported for function parameters or local variables in HC programs.

Work-around: remove these 'const' modifiers. The semantics of a correct program will remain unchanged, since the only purpose of the 'const' modifiers is to enforce additional compiler checking.

4) The *number of tasks registered on a phaser* cannot be larger than the number of worker threads specified with the -nproc option when an HC program is invoked. Otherwise, a deadlock may occur.

5) HC function calls must be in canonical form; either being a statement or being the right hand-side of an assignment.

Canonicalized HC function usage

```
foo();
b = foo();
a = b + c; // 'foo' value must first be assigned to a variable
```

Acknowledgement

Partial support for Habanero-C was provided through the CDSC program of the National Science Foundation with an award in the 2009 Expedition in Computing Program.