Habanero-Java

The Habanero Java (HJ) language under development at Rice University builds on past work on X10 v1.5. HJ is intended for use in teaching Computer Science at the undergraduate level (COMP 322), as well as to serve as a research testbed for new language, compiler, and runtime software technologies for extreme scale systems. HJ proposes an execution model for multicore processors based on four orthogonal dimensions for portable parallelism:

1. Lightweight dynamic task creation and termination using async, finish, future, forall, forasync, ateach constructs.
2. Collective and point-to-point synchronization using phasers
3. Mutual exclusion and isolation using isolated
4. Locality control using hierarchical place trees

Since HJ is based on Java, the use of certain non-blocking primitives from the Java Concurrency Utilities is also permitted in HJ programs, most notably operations on Java Concurrent Collections such as java.util.concurrent.ConcurrentHashMap and on Java Atomic Variables.

A short summary of the HJ language is included below, and the following paper provides an overview of the language:

Habanero-Java: the New Adventures of Old X10. 9th International Conference on the Principles and Practice of Programming in Java (PPPJ), August, 2011.

We also have a new library implementation of HJ called HJ-lib that can be used with any standard Java 8 implementation. HJ-lib puts a particular emphasis on the usability and safety of parallel constructs. HJ-lib is built using Java 8 closures and can run on any Java 8 JVM. Older JVMs can be targeted by relying on external bytecode transformations tools for compatibility.

More details can be found in the papers in the Habanero publications web page.

A download of the HJ language implementation can be found here. The instructions for the download and installation of the HJ-lib jar file is available here.

HJ Language Summary

(Following standard conventions for syntax specification, the [...] square brackets below refer to optional clauses.)

async [at (place)]

[phased [(ph1<mode1>, ...)]]

[seq (condition)]

[await (ddf1, ...)] Stmt

- async — Asynchronously start a new child task to execute Stmt
- at –- A destination place may optionally be specified for where the task should execute
- phased — Task may optionally be phased on a specified subset of its parent’s phasers with specified modes (e.g., phaser ph1 with mode1), or on the entire set of the parent's phasers and modes (by default, if no subset is specified)
- seq — A boolean condition may optionally be specified as a tuning parameter to determine if the async should just be executed sequentially in the parent task. The seq clause can not be combined with phased or await clauses
- await — Task may optionally be delayed to only start after all specified events (data-driven futures) become available

finish [(accum1, ...)] Stmt

- Execute Stmt, but wait until all (transitively) spawned asyncs and futures in Stmt’s scope have terminated
- Propagate a multiset of all exceptions thrown by asyncs spawned within Stmt’s scope
- Optionally, a set of accumulators (e.g., accum1) can be specified as being registered with this finish scope

final future<T> f = async<T> [at (place)]

[phased [(ph1<mode1>, ...)]] Stmt-Block-with-Return

- Asynchronously start a new child task to evaluate Stmt-Block-with-Return with optional at and phased clauses as in async
- f is a reference to object of type future<T>, which is a container for the value to be computed by the future task; T may be a primitive type (including void) or an object type (class)
- Stmt-Block-with-Return is a statement block that dynamically terminates with a return statement as in a method body; a return statement is not needed if the return type is void

f.get()

- Wait until future f has completed execution, and propagate its return value; if T = void, then f.get() is evaluated as a statement (like a method call with a void return value)
- get() also propagates any exception thrown by Stmt-Block-with-Return

point

- A point is an n-dimensional tuple of int's
- A point variable can hold values of different ranks e.g., point p; p = [1]; … p = [2,3]; …

for (point [i1, …] : [lo1:hi1, …]) Stmt

- Execute multiple instances of Stmt sequentially in lexicographic order, one per iteration in rectangular region [lo1:hi1, …]

forall (point [i1, …] : [lo1:hi1, …]) Stmt

- Create multiple parallel instances of Stmt as child tasks, one per forall iteration in the rectangular region [lo1:hi1, …]
- An implicit finish is included for all iterations of the forall
- Each forall instance has an anonymous pre-allocated phaser shared by all its iterations; no explicit phased clause is permitted for a forall

forasync (point [i1, …] : [lo1:hi1, …]) [phased [(ph1<mode1>, ...)]] Stmt

- Like the forall, create multiple instances of Stmt as child tasks, one per forasync iteration in the rectangular region [lo1:hi1, …]
    - There is no implicit finish in forasync
    - As with async, a forasync iteration may optionally be phased on a specified subset, (ph1<mode1>, ...), of its parent's phasers or on the entire set

new phaser(mode1)

- Allocate a phaser with the specified mode, which can be one of SIG, WAIT, SIG_WAIT, SINGLE
- Scope of phaser is limited to immediately enclosing finish

next ;

- Advance each phaser that this task is registered on to its next phase, in accordance with this task's registration mode
- Wait on each phaser that task is registered on with a wait capability (WAIT, SIG_WAIT, SINGLE)

next single Stmt

- Execute a single instance of Stmt during the phase transition performed by next
- All tasks executing the next single statement must be registered with all its phasers in SINGLE mode

signal ;

- signal each phaser that task is registered on with a signal capability (SIG, SIG_WAIT, SIGNAL)
- signal is a non-blocking operation --- computation between signal and next serves as a "split phase barrier"

isolated Stmt

- Execute Stmt in isolation (mutual exclusion) relative to all other instances of isolated statements
- Stmt must not contain any parallel constructs
- Weak atomicity: no guarantee of isolation with respect tonon-isolated statements

isolated [(obj1, ...)] Stmt

- Object-based isolation — mutual exclusion is only guaranteed for a pair of isolated statements with a non-empty intersection of their object sets
- If no object set is specified, then the default set is the universe of all objects
- A null value for an object is treated like an empty contribution to the set
- Weak atomicity: no guarantee of isolation with respect to non-isolated statements

complex32, complex64

- HJ includes complex as a primitive type e.g.,

complex32 cf = (1.0f, 2.0f); complex64 cd = (1.0, 2.0);

- The following operations are supported on complex:

+,-,*,/, ==, !=, toString(), exp(), sin(), cos(), sqrt(), pow()

array views

T[.] declares a view on a 1-D Java array e.g.,

double[.] view = new arrayView(baseArray, offset, [lo1:hi1, …])

where

baseArray = base 1-D Java array

offset = starting offset in baseArray for view

[lo1:hi1, …] = rectangular region for view

abstract performance metrics

- Programmer inserts calls of the form, perf.addLocalOps(N), in sequential code
- HJ implementation computes total work and critical path length in units of programmer's local ops