Habanero Extreme Scale Software Research Project
Comp215: Functional Programming and Parallelism

Zoran Budimlić (Rice University)
All computers are parallel computers!
"The way the processor industry is going, is to add more and more cores, but nobody knows how to program those things. I mean, two, yeah; four, not really; eight, forget it."
What is parallel programming?

Specification of operations that can be executed in parallel

A parallel program is decomposed into sequential subcomputations called tasks

Parallel programming constructs define task creation, termination, and interaction
Sequential program: array sum

```java
int sum = 0;
for (int i=0 ; i < X.length ; i++)
    sum += X[i];
```

Observations:
The decision to sum up the elements from left to right was arbitrary
The computation graph shows that all operations must be executed sequentially
Parallel sum strategy

Basic idea:
Decompose problem into two tasks for partial sums
Combine results to obtain final answer
Parallel divide-and-conquer pattern

**Task 0:** Compute sum of lower half of array

**Task 1:** Compute sum of upper half of array

Compute total sum
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
    //Wait for T₁
} //End finish
STMT3; //Continue in T₀

finish S
- Execute S, but wait until all asyncs in S’s scope have terminated.

Acknowledgment: X10 and Habanero projects
Habanero-Java library

- Pure Java library without any other dependencies

- Uses Java 8 Lambda Expressions
  - Required for terse syntax
Parallel sum

// Start of Task T0 (main program)
sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
finish(() -> {
    async(() -> {
        // Child task computes sum of lower half of array
        for(int i=0; i < X.length/2; i++) sum1 += X[i];
    });
    // Parent task computes sum of upper half of array
    for(int i=X.length/2; i < X.length; i++) sum2 += X[i];
});
// Parent task waits for child task to complete (join)
return sum1 + sum2;
async seq(cond) <stmt> ≡ if (cond) <stmt> else async <stmt>

- “seq” clause specifies condition under which async should be executed sequentially
  - False \( \Rightarrow \) an async is created
  - True \( \Rightarrow \) the parent executes async body sequentially
- Avoids the need to duplicate code for both cases

asyncSeq(size < thresholdSize, () -> computeSum(X, lo, mid));
protected static void quicksort(
    final Comparable[] A, final int M, final int N) {
    if (M < N) {
        // A point in HJ is an integer tuple
        HjPoint p = partition(A, M, N);
        int I = p.get(0);
        int J = p.get(1);
        asyncSeq(I - M <= 100, () -> quicksort(A, M, I));
        asyncSeq(N - J <= 100, () -> quicksort(A, J, N));
    }
}
HjFuture f = future (()->{ ... future task ...});

Create a child task to compute the value of the future f

Integer i = f.get();

Get the value of the future, wait if it’s not available yet
Unlike finish, it only waits for the task that computes the future
Parallel sum using futures

// Parent Task T1 (main program)
// Compute sum1 (lower half) and sum2 (upper half) in parallel
final HjFuture sum1 = future (() -> {
    int sum = 0;
    for(int i=0 ; i < X.length/2 ; i++) sum += X[i];
    return sum;
});
final HjFuture sum2 = future (() ->{
    int sum = 0;
    for(int i=X.length/2 ; i < X.length ; i++) sum += X[i];
    return sum;
});

// Task T1 waits for Tasks T2 and T3 to complete
int total = sum1.get() + sum2.get();
Beyond two-way parallelism
static int computeSum(int[] X, int lo, int hi) {
    if (lo > hi) return 0;
    else if (lo == hi) return X[lo];
    else {
        int mid = (lo+hi)/2;
        final HjFuture sum1 = future(() -> { return computeSum(X, lo, mid); });
        final HjFuture sum2 = future(() -> { return computeSum(X, mid+1, hi); });
        // Parent now waits for the container values
        return sum1.get() + sum2.get();
    }
}

// computeSum

. . .

int sum = computeSum(X, 0, X.length-1); // main program

These two function calls can execute in parallel
Dataflow programming

“Macro-dataflow” = extension of dataflow model from instruction-level to task-level operations

General idea: build an arbitrary task graph, but restrict all inter-task communications to single-assignment variables

Static dataflow ==> graph fixed when program execution starts
Dynamic dataflow ==> graph can grow dynamically
Semantic guarantees: race-freedom, determinism
Deadlocks are possible due to unavailable inputs (but they are deterministic)
Data-driven futures and tasks

HjDataDrivenFuture<T1> ddfA = newDataDrivenFuture();
Allocate an instance of a data-driven-future object (container)
Object in container must be of type T1
asyncAwait(ddfA, ddfB, ..., () -> Stmt);
Create a new data-driven-task to start executing Stmt after all of ddfA, ddfB, ... become available (i.e., after task becomes “enabled”)

ddfA.put(V);
Store object V (of type T1) in ddfA, thereby making ddfA available
Single-assignment rule: at most one put is permitted on a given DDF

ddfA.get();
Return value (of type T1) stored in ddfA
Can only be performed by async’s that await on ddfA (hence no blocking is necessary for DDF gets)
Futures vs. Data-driven futures

**Future version:**

```java
final HjFuture<T> f = future(() -> { return g(); });
S1
async(() -> {
    ... = f.get();
    S2;
    S3;
});
```

**Data-driven future version:**

```java
HjDataDrivenFuture<T> f = newDataDrivenFuture();
async(() -> { f.put(g()); });
S1
asyncAwait(f, () -> {
    ... = f.get();
    S2;
    S3;
});
```
Futures vs. Data-driven futures

Consumer task blocks on \texttt{get()} for each future that it reads, whereas \texttt{async-await} does not start execution till all DDFs are available.

Future tasks cannot deadlock, but it is possible for a DDT to block indefinitely ("deadlock") if one of its input DDFs never becomes available.

DDTs and DDFs are more general than futures:
- Producer task can only write to a single future object, whereas a DDT can write to multiple DDF objects.
- The choice of which future object to write to is tied to a future task at creation time, whereas the choice of output DDF can be deferred to any point with a DDT.

DDTs and DDFs can be implemented more efficiently than futures:
- An "asyncAwait" statement does not block the worker, unlike a future.get()
Habanero-Scala

Scala - OO and functional programming language
Compiles to JVM
Can be mixed freely with Java

async { <stmt> }
  creates a new child task that executes <stmt>
  parent task proceeds to operation following the async

asyncSeq(<cond>) { S } ≡ if (<cond>) S else async { S }

finish { <stmt> }
  execute <stmt>, but wait until all (transitively) spawned asyncs in <stmt>’s scope have terminated

Futures, data-driven futures, data-driven tasks
Fibonacci in Scala using DDFs

```scala
finish {
  val res = ddf[Int]()
  async {
    fib(N, res)
  }
}
println("fib(" + N + ") = " + res.get())

def fib(n: Int, v: DataDrivenFuture[Int]): Unit = {
  if (n < 2) {
    v.put(n)
  } else {
    val (res1, res2) = (ddf[Int](), ddf[Int]())
    async {
      fib(n - 1, res1)
    }
    async {
      fib(n - 2, res2)
    }
    asyncAwait(res1, res2) {
      v.put(res1.get() + res2.get())
    }
  }
}
```
Phasers

Used for long-running tasks that need to synchronize with each other from time to time

Supports Collective and Point-to-Point synchronization

Tasks can register in
  - signal-only/wait-only mode for producer/consumer synchronization
  - signal-wait mode for barrier synchronization

next operation is guaranteed to be deadlock-free

Habanero programs with phasers, finish, async, async-await are guaranteed to be deterministic if they are data-race-free
Iterative averaging using phasers

```scala
val myPhasers = Array.tabulate[n + 2](i => phaser())
for (index <- 1 to n) {
  val (me, left, right) = (index, index - 1, index + 1)
  val leftPhaser = myPhasers(left).inMode(PhaserMode.WAIT)
  val selfPhaser = myPhasers(me).inMode(PhaserMode.SIG)
  val rightPhaser = myPhasers(right).inMode(PhaserMode.WAIT)

  asyncPhased(leftPhaser, selfPhaser, rightPhaser) {
    for (iter <- 0 until N) {
      val loopVal = 0.5 * (dataArray(left) + dataArray(right))
      // Allow others to proceed and modify dataArray
      next
      // update the 'owning' element
      dataArray(me) = loopVal
      // notify others that value has been updated
      next
    }
  }
}
```
Habanero-C++ (HC++)

Dynamic task creation & termination
  async, finish
  forasync1D, forasync2D, forasync3D

Data-Driven Tasks and Data-Driven Futures
  asyncAwait, ddf_put(), ddf_get()

Support for affinity control
  hierarchical places trees (HPT)

Collective and point-to-point synchronization
  asyncPhased

Extensions for PGAS programming
  UPC and MPI

Uses C++ 11 lambda functions
C++11 lambdas and Habanero

// create a function
auto lambda = [ capture_list ] (argument_list) -> return_type {
    Statements;
};

// apply function
lambda(argument_list)

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

finish ( [capture_list] ( ) {
    <Stmt>
});
Fibonacci in HC++

```cpp
int fib(int n) {
    if(n <= THRESHOLD) return fib_serial(n);
    else {
        int x, y;
        finish ( [n, &x, &y] ( ) {
            async ( [n, &x] ( ) {
                x = fib(n-1);
            });
            y= fib(n-2);
        });
        return x+y;
    }
}
```
Parallel computers are everywhere

Functional programming paradigms extremely well-fit for parallel programming
  - Used as a programming language
  - Used for implementing parallel constructs

Much more on this in COMP322

The concepts you learned in COMP215 will be very useful later