COMP 322: Fundamentals of Parallel Programming

Lecture 4: Futures -- Tasks with Return Values

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu

Acknowledgments for Today's Lecture

- COMP 322 Lecture 4 handout

HJ Abstract Performance Metrics (Recap)

- Serial code sequence
  - Dynamic sequence of instructions with no parallel operations
- Calls to perf.addLocalOps()
  - *Programmer* inserts calls of the form, perf.addLocalOps(N), inside a step to indicate execution of N application-specific abstract operations e.g., floating-point ops, stencil ops, data structure ops, etc.
  - Multiple calls add to the execution time of the step
- -perf=true runtime option
  - If an HJ program is executed with this option, abstract metrics are printed at end of program execution with WORK(G), CPL(G), *Ideal Speedup = WORK(G)/ CPL(G)*

Question: What should be included in perf.addLocalOps()?

- Answer: It depends. We will tell you what to count in HW3, but here's the general idea …
- We'll say that a cost function Cost(n) is "order f(n)", or simply "$O(f$(n))" (read "Big-O of f(n))") if
  - —Cost-X(n) < factor * f(n), for sufficiently large n, for some constant factor
- Examples:
  - —Cost-A(n) = $2*n^3 + n^2 + 1$ Cost-A is $O(n^3)$
  - —Cost-B(n) = $3*n^2 + 10$ Cost-B is $O(n^2)$
  - —Cost-C(n) = 2^n Cost-C is $O(2^n)$

Famous "Complexity Classes"

- $O(1)$ constant-time (head, tail)
- $O(\log n)$ logarithmic (binary search)
- $O(n)$ linear (vector multiplication)
- $O(n * \log n)$ "n logn" (sorting)
- $O(n^2)$ quadratic (matrix addition)
- $O(n^3)$ cubic (matrix multiplication)
- $n^{O(1)}$ polynomial (...many! ...)
- $2^{O(n)}$ exponential (guess password)

Question: What should be included in perf.addLocalOps()?

- Focus on key metric of interest in your algorithm
- Don't count operations that may be incidental properties of your implementation
  - —e.g., don't count operations that may not be needed in a better engineered implementation
- Since big-O analysis does not care about differences within a constant factor, you can just a unit 1 as a stand-in for a constant number of operations

HJ Futures: Tasks with Return Values

async<T> { <Stmt-Block> }

- Creates a new child task that executes Stmt-Block, which must terminate with a return statement returning a value of type T
- Async expression returns a reference to a *container* of type future<T>, and parent task immediately to operation following the async
- Values of type future<T> can only be assigned to *final variables*

Expr.get()

- Evaluates Expr, and blocks if Expr's value is unavailable
- Expr must be of type future<T>
- Return value from Expr.get() will then be T
- Unlike finish which waits for all tasks in the finish scope, a get operation only waits for the specified async expression

Example: Two-way Parallel Array Sum using Future Tasks

```
// Parent Task T1 (main program)
// Compute sum1 (lower half) and sum2 (upper half) in parallel
final future<int> sum1 = async { // Future Task T2
  int sum = 0;
  for(int i=0 ; i < X.length/2 ; i++) sum += X[i];
  return sum;
}; //NOTE: semicolon needed to terminate assignment to sum1
final future<int> sum2 = async { // Future Task T3
  int sum = 0;
  for(int i=X.length/2 ; i < X.length ; i++) sum += X[i];
  return sum;
}; //NOTE: semicolon needed to terminate assignment to sum2
//Task T1 waits for Tasks T2 and T3 to complete
int sum = sum1.get() + sum2.get();
```

Listing 1: Two-way Parallel ArraySum using Future Tasks

Why are these semicolons needed?

Comparison of Future Task and Regular Async Versions

- Future task version initializes two references to future objects, sum1 and sum2, and both are declared as final
- No finish construct needed in this example
  - Instead parent task waits for child tasks by performing sum1.get() and sum2.get()
- Guaranteed absence of race conditions in Future Task example
  - No race on sum because it is a local variable in tasks T1, T2, T3
  - No race on sum1 and sum2 because of blocking-read semantics

Future Task Declarations and Uses

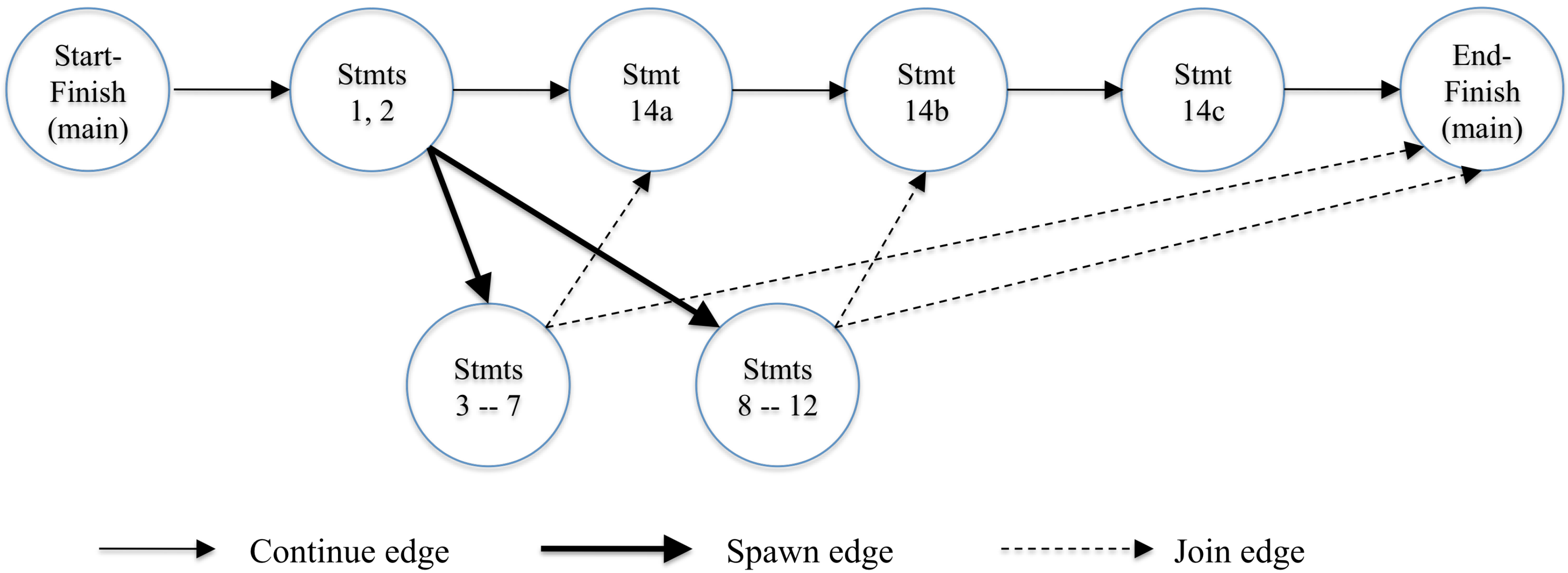
- Variable of type future<T> is a reference to a *future object*
  - Container for return value of T from future task
  - The reference to the container is also known as a handle
- Two operations that can be performed on variable V1 of type future<T1> (assume that type T2 is a subtype of type T1):
  - Assignment: V1 can be assigned value of type future<T2>
  - Blocking read: V1.get() waits until the future task referred to by V1 has completed, and then propagates the return value
- Future task body must start with a type declaration, async<T1>, where T1 is the type of the task's return value
- Future task body must consist of a statement block enclosed in { } braces, terminating with a return statement

Computation Graph Extensions for Future Tasks

- Since a get() is a blocking operation, it must also be treated as a *continuation*
  - get()'s must occur on boundaries of CG nodes/steps
  - May require splitting a statement into sub-statements e.g.,
    - 14: int sum = sum1.get() + sum2.get();

    can be split into three sub-statements
    - 14a int temp1 = sum1.get();
    - 14b int temp2 = sum2.get();
    - 14c int sum = temp1 + temp2;
- *Spawn* edge connects parent task to child future task, as before
- *Join edge* connects end of future task to Immediately Enclosing Finish (IEF), as before
- *Additional join edges* are inserted from end of future task to each get() operation on future object

Computation Graph for Two-way Parallel Array Sum using Future Tasks



Why must Future References be declared as final?

```
static future<int> f1=null;

static future<int> f2=null;


void main(String[] args) {

  f1 = async<int> {return a1();};

  f2 = async<int> {return a2();};

}
```

```
int a1() { // Task T1
 while (f2 == null); // spin loop
  return f2.get(); //T1 waits for T2
}


int a2() { // Task T2
  while (f1 == null); // spin loop
  return f1.get(); //T2 waits for T1
}
```

cyclic wait condition

This situation cannot arise in HJ because f1 and f2 must be final

- Final declaration ensures that variable (handle) cannot be modified after initialization

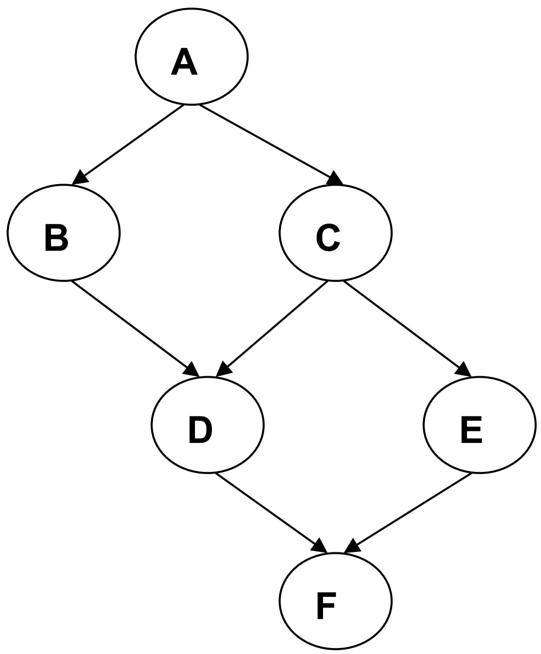
WARNING: spin loops are an example of bad parallel programming practice in application code (they should only be used by expert systems programmers, and even then sparingly)

Future Tasks with void Return Type

- Key difference between regular async's and future tasks is that future tasks have a future<T> return value
- We can get an intermediate capability by setting T=void as shown
- Can be useful if a task needs to synchronize on another task, but doesn't need to use future object for communicating a return value

```
sum1 = 0; sum2 = 0; // Task T1
// Assume that sum1 & sum2 are fields
final future<void> a1 = async<void> {
  for (int i=0; i < X.length/2; i++)
      sum1 += X[i]; // Task T2
};
final future<void> a2 = async<void> {
  for (int i=X.length/2; i < X.length; i++)
      sum2 += X[i]; // Task T3
};
//Task T1 waits for Tasks T2 and T3
a1.get(); a2.get();
int sum = sum1 + sum2;
```

Using Future Tasks to generate Computation Graph CG3 from Homework 2



Computation Graph CG3

NOTE: this is not an acceptable solution for Homework~2 since this code uses future tasks!

```
// NOTE: return statement is optional
   when return type is void

final future<void> A = async<void>
   { . . . ; return;}

final future<void> B = async<void>
   { A.get(); . . . ; return;}

final future<void> C = async<void>
   { A.get(); . . . ; return;}

final future<void> D = async<void>
   { B.get(); C.get(); . . . ; return;}

final future<void> E = async<void>
   { C.get(); . . . ; return;}

final future<void> F = async<void>
   { D.get(); E.get(); . . . ; return;}
```