COMP 322: Fundamentals of Parallel Programming

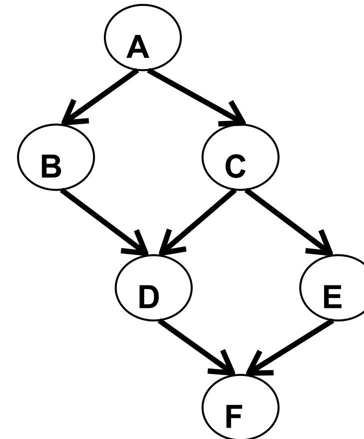
Lecture 7: Futures (contd), Parallel Design Patterns, Finish Accumulators

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

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Worksheet #6 solution: Computation Graphs for Async-Finish and Future Constructs

1) Can you write an HJ program with async-finish constructs that generates a Computation Graph with the same ordering constraints as the graph on the right?
No



2) Can you write an HJ program with future async-get constructs that generates a Computation Graph with the same ordering constraints as the graph on the right? If so, provide a sketch of the program.
Yes, see program sketch with void futures. A dummy return value can also be used.

```
1. // Return statement is optional for void futures
2. final future<void> a = async<void> { A;};
3. final future<void> b = async<void> { a.get(); B;};
4. final future<void> c = async<void> { a.get(); C;};
5. final future<void> d = async<void> { b.get();
6.                                      c.get(); D;};
7. final future<void> e = async<void> { c.get(); E;};
8. final future<void> f = async<void> { d.get();
9.                                      e.get(); F;};
10. f.get(); // Or wrap lines 1-9 in finish
```

Outline of Today's Lecture

- **Futures --- Tasks with Return Values (contd)**
- **Design Patterns for Parallel Programming**
- **Finish Accumulators**

Acknowledgments

- COMP 322 Module 1 handout, Chapter 5, Chapter 6
- "A Pattern language for Parallel Programming"
  - Presentation by Beverly Sanders, U. Florida
  - http://www.cise.ufl.edu/research/ParallelPatterns/sasplas.ppt
- "Parallel Programming with Microsoft .NET"
  - Book published by Microsoft; free download available at http://parallelpatterns.codeplex.com
- "Introduction to Concurrency in Programming Languages", Matthew J. Sottile, Timothy G. Mattson, and Craig E Rasmussen
  - Book published by CRC press; slides available from book web site, http://www.parlang.com/
- "Parallel Programming Patterns", ME964 lecture, Oct 2008, U. Wisconsin
  - http://sbel.wisc.edu/Courses/ME964/2008/.../me964Oct16.ppt

Extending Async Tasks with Return Values (Recap)

- **Example Scenario in PseudoCode**

```
1. // Parent task creates child async task
2. final future<int> container =
3.                  async<int>  { return computeSum(X, low, mid); };
4. . . .
5. // Later, parent examines the return value
6. int sum = container.get();
```

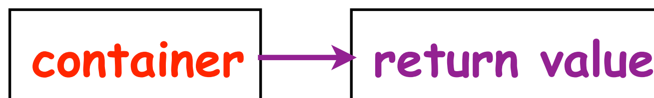
- **Two issues to be addressed:**
  1) Distinction between container and value in container (box)
  2) Synchronization to avoid race condition in container accesses

Parent Task

```
container = async {...}
. . .
container.get()
```

Child Task

```
computeSum(...)
return ...
```



Reduction Tree Schema in ArraySum1 (Recap)

X[0] X[1] X[2] X[3] X[4] X[5] X[6] X[7]

stride = 1, size = 4

X[0] X[2] X[4] X[6]

stride = 2, size = 2

X[0] X[4]

stride = 4, size = 1

X[0]

Questions:

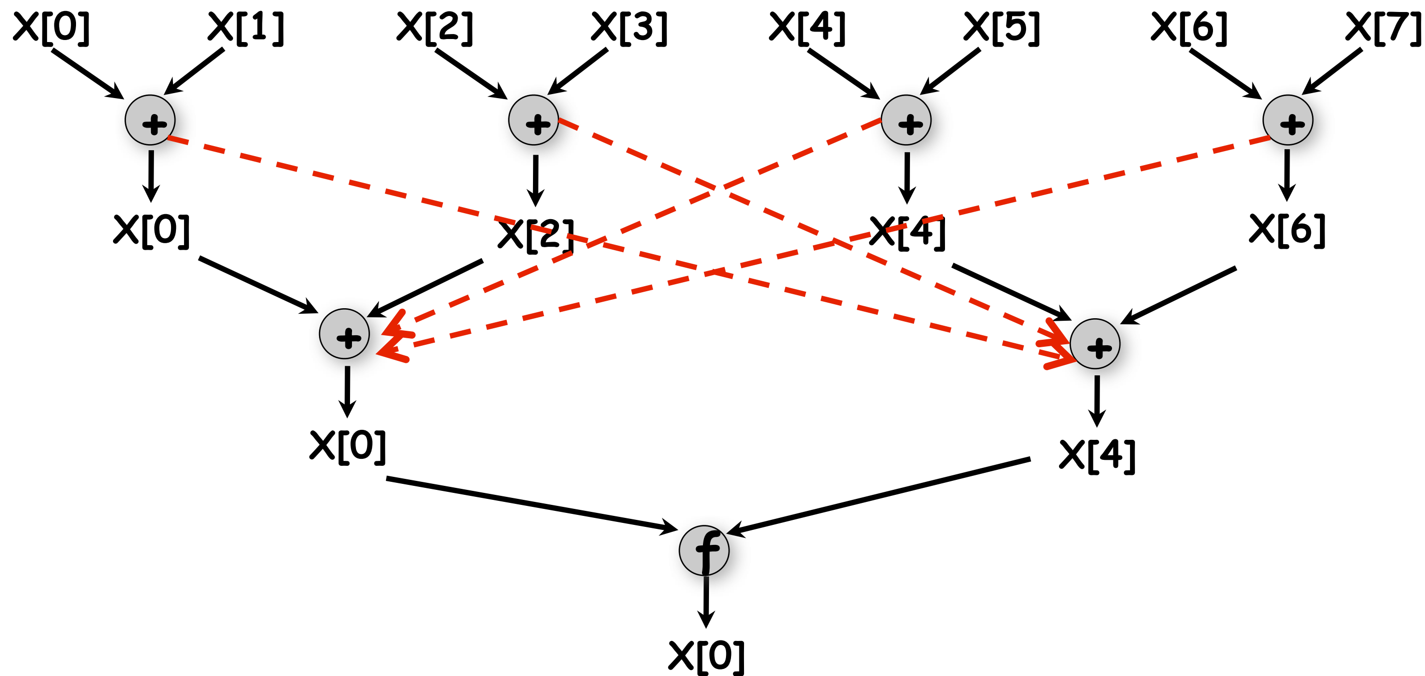
- **How can we implement this schema using future tasks instead?**
- **Can we avoid overwriting elements of array X?**

Array Sum using Future Tasks (Recap)

Recursive divide-and-conquer pattern

```
1.  static int computeSum(int[] X, int lo, int hi) {
2.    if ( lo > hi ) return 0;
3.    else if ( lo == hi ) return X[lo];
4.    else {
5.      int mid = (lo+hi)/2;
6.      final future<int> sum1 =
7.            async<int> { return computeSum(X, lo, mid); };
8.      final future<int> sum2 =
9.           async<int> { return computeSum(X, mid+1, hi); };
10.     // Parent now waits for the container values
11.     return sum1.get() + sum2.get();
12.   }
13. } // computeSum
14. int sum = computeSum(X, 0, X.length-1); // main program
```

Extra dependences in ArraySum1 program (for-finish-for-async)



Extra dependence edges due to finish-async stages
(not present in ArraySum2 divide-and-conquer version)

- **Which of ArraySum1 or ArraySum2 will perform better if the time taken by the reduction operator depends on its inputs e.g., as in Lab 2?**

Divide-and-conquer Array Sum using async-finish instead of futures (ArraySum4)

```
1. // Use single-element arrays to represent boxed mutable ints
2. static void computeSum(int[] X, int lo, int hi, int[] retVal) {
3.    if ( lo > hi ) retVal[0] = 0;
4.    else if ( lo == hi ) retVal[0] = X[lo];
5.    else {
6.      int mid = (lo+hi)/2;
7.      int[] sum1 = new int[1]; int[] sum2 = new int[1];
8.      finish {
9.        async { computeSum(X, lo, mid, sum1); };
10.       async { computeSum(X, mid+1, hi, sum2); };
11.     }
12.     retVal[0] = sum1[0] + sum2[0];
13.    }
14.  } // computeSum
15. . . .
16. // main program
17. int[] sum = new int[1];
18. computeSum(X, 0, X.length-1, sum);
```

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 using `future<void>` as shown**
- **Can be useful if a task needs to synchronize on a specific task (instead of finish), but doesn't need a future object to communicate a return value**

```
1. sum1 = 0; sum2 = 0; // Task T1
2. // Assume that sum1 & sum2 are fields
3. final future<void> a1 = async<void> {
4.    for (int i=0; i < X.length/2; i++)
5.       sum1 += X[i]; // Task T2
6. };
7. final future<void> a2 = async<void> {
8.    for (int i=X.length/2; i < X.length; i++)
9.       sum2 += X[i]; // Task T3
10. };
11. // Task T1 waits for Tasks T2 and T3
12. a1.get(); a2.get();
13. // Now fields sum1 and sum2 can be read
```

Why must future variables be declared final?
Time for worksheet #7!

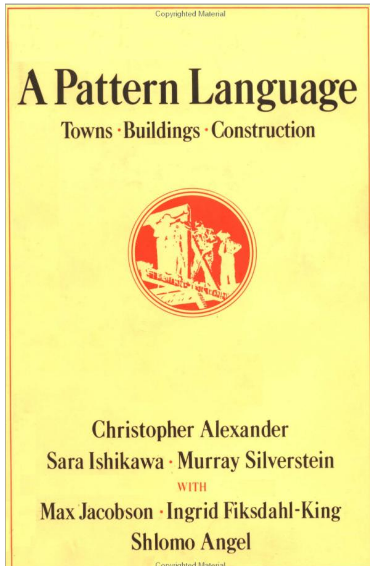
Outline of Today's Lecture

- **Futures --- Tasks with Return Values (contd)**
- **Design Patterns for Parallel Programming**
- **Finish Accumulators**

Acknowledgments

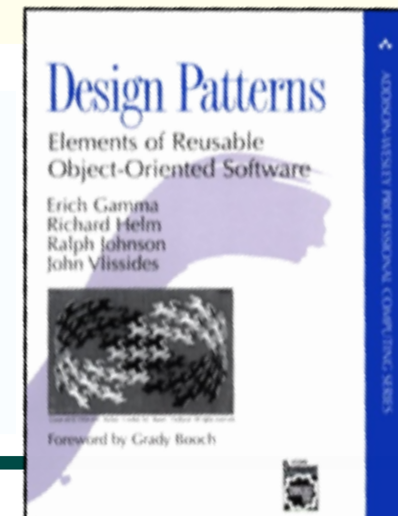
- COMP 322 Module 1 handout, Chapter 5, Chapter 6
- "A Pattern language for Parallel Programming"
  - Presentation by Beverly Sanders, U. Florida
  - http://www.cise.ufl.edu/research/ParallelPatterns/sasplas.ppt
- "Parallel Programming with Microsoft .NET"
  - Book published by Microsoft; free download available at http://parallelpatterns.codeplex.com
- "Introduction to Concurrency in Programming Languages", Matthew J. Sottile, Timothy G. Mattson, and Craig E Rasmussen
  - Book published by CRC press; slides available from book web site, http://www.parlang.com/
- "Parallel Programming Patterns", ME964 lecture, Oct 2008, U. Wisconsin
  - http://sbel.wisc.edu/Courses/ME964/2008/.../me964Oct16.ppt

Design Patterns = formal discipline of design



- **Christopher Alexander's approach to (civil) architecture:**
  - **A design pattern "describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."** *Page x, A Pattern Language,* **Christopher Alexander, 1977**
- **A pattern language is an organized way of tackling an architectural problem using patterns**

- The Design Patterns book turned object oriented design from an "art" to a systematic design discipline.



Example of OO Design Pattern: Visitor

```
1. class Employee {
2.  private int vacationDays; private String SSN;
3.  public void accept(Visitor v) { v.visit(this); }
4.  . . .
5. }
6. abstract class Visitor {
7.  public abstract void visit(Employee emp);
8. }
9. class VacationVisitor extends Visitor {
10.  private int totalDays;
11.  public VacationVisitor() { total_days = 0; }
12.  public void visit(Employee emp) {
13.    totalDays += emp.getVacationDays();
14.  }
15.  public int getTotalDays() { return totalDays; }
16.}
17.. . .
18.VacationVisitor v = new VacationVisitor();
19.emp1.accept(v); emp2.accept(v); ...
20.... v.getTotalDays() ...
21.
```

Patterns in Parallel Programming

- **Can a pattern language/taxonomy providing guidance for the entire development process make parallel programming easier?**
  - Need to identify basic patterns, along with refinements (usually for efficiency)
  - By relating HJ constructs to parallel programming patterns, you can apply HJ concepts to any parallel programming model you encounter in the future
- **Algorithmic Patterns**
  - Selection of task and data decompositions to solve a given problem in parallel
    - Task decomposition = identification of parallel steps
    - Data decomposition = partitioning of data into task-local vs. shared storage classes
  - Examples: Parallel Loops, Parallel Tasks, Reductions, Dataflow, Pipeline
-

Selecting the Right Pattern
(adapted from page 9, Parallel Programming w/ Microsoft .Net)

Application characteristics	Algorithmic pattern	Relevant HJ constructs
Sequential loop with independent iterations	1) Parallel Loop	forall, forasync (to be covered later)
Independent operations with well-defined control flow	2) Parallel Task	async, finish (already covered)
Aggregating data from independent tasks/iterations	3) Parallel Aggregation (reductions)	finish accumulators (to be covered today)
Ordering of steps based on data flow constraints	4) Futures	futures (already covered), data-driven tasks (to be covered later)
Divide-and-conquer algorithms with recursive data structures	5) Dynamic Task Parallelism	async, finish (already covered)
Repetitive operations on data streams	6) Pipelines	streaming phasers (to be covered later)

Outline of Today's Lecture

- **Futures --- Tasks with Return Values (contd)**
- **Design Patterns for Parallel Programming**
- **<u>Finish Accumulators</u>**

Acknowledgments

- COMP 322 Module 1 handout, Chapter 5, Chapter 6
- "A Pattern language for Parallel Programming"
  - Presentation by Beverly Sanders, U. Florida
  - http://www.cise.ufl.edu/research/ParallelPatterns/sasplas.ppt
- "Parallel Programming with Microsoft .NET"
  - Book published by Microsoft; free download available at http://parallelpatterns.codeplex.com
- "Introduction to Concurrency in Programming Languages", Matthew J. Sottile, Timothy G. Mattson, and Craig E Rasmussen
  - Book published by CRC press; slides available from book web site, http://www.parlang.com/
- "Parallel Programming Patterns", ME964 lecture, Oct 2008, U. Wisconsin
  - http://sbel.wisc.edu/Courses/ME964/2008/.../me964Oct16.ppt

Parallel NQueens Example

```
1.  // Challenge: how to count number of solutions found?
2.
3.  finish nqueens_kernel(new int[0], 0);
4.  System.out.println("No. of solutions = " ...);
5.  . . .
6.  void nqueens_kernel(int [] a, int depth) {
7.    if (size == depth) // Solution found: how to count?
8.    else
9.      /* try each possible position for queen at depth */
10.     for (int i =  0; i < size; i++) async {
11.       /* allocate a temporary array and copy array a into it */
12.       int [] b = new int [depth+1];
13.       System.arraycopy(a, 0, b, 0, depth);
14.       b[depth] = i;
15.       if (ok(depth+1,b)) nqueens_kernel(b, depth+1);
16.     } // for-async
17. } // nqueens_kernel()
```

Finish Accumulators in HJ

- Creation

```
accumulator ac = accumulator.factory.accumulator(operator, type);
```

  - *operator can be Operator.SUM, Operator.PROD, Operator.MIN, Operator.MAX or Operator.CUSTOM*
    - ★ *You may need to use the fully qualified name, accumulator.Operator.SUM, in your code if you don't have the appropriate import statement*
  - *type can be* int.class or double.class for standard operators or any object that implements a "reducible" interface for CUSTOM

- Registration

```
finish (ac1, ac2, ...) { ... }
```

  - *Accumulators ac1, ac2, ... are registered with the finish scope*

- Accumulation

```
ac.put(data);
```

  - *can be performed by any statement in finish scope that registers ac*

- Retrieval

```
Number n = ac.get();
```

  - *get() is nonblocking because finish provides the necessary synchronization*
    Either returns initial value before end-finish or final value after end-finish
  - *result from get() will be deterministic if CUSTOM operator is associative and commutative*

Example with Multiple Finish Accumulators

```
1.   // T1 allocates accumulator a and b
2.   accumulator a = accumulator.factory.accumulator(SUM, int.class);
3.   accumulator b = accumulator.factory.accumulator(MIN, double.class);
4.   // T1 can invoke put()/get() on a and b any time
5.   a.put(1); // adds 1 to accumulator a
6.   Number v1 = a.get(); // Returns 1
7.   // T1 creates a finish scope registered on a and b
8.   finish (a, b) {
9.     // Any task can invoke put() within the finish
10.    b.put(2.5); // min operation with accumulator b
11.    finish { // Inner finish inherits registrations for a & b
12.      async a.put(2);
13.      b.put(1.5);
14.    }
15.    // Unlikely case: if a task invokes get() within the finish,
16.    // the value returned value is that on entry to the finish
17.    Number v2 = a.get(); // Returns 1
18.  }
19.  // T1 obtains overall sum and min values after end-finish
20.  Number v3 = a.get(); // Returns 1 + 2 = 3
21.  Number v4 = b.get(); // Returns min(2.5,1.5) = 1.5
```

Summary of accumulator API

```
// Reduction operators
enum Operator {SUM, PROD, MIN, MAX, CUSTOM}

// Predefined reduction
accum(Operator op, Class dataType);    // Constructor
void accum.put(Number datum);          // Remit datum
void accum.put(int datum);
void accum.put(double datum);
Number accum.get();                    // Retrieve result

// User-defined reduction
interface reducible<T> {
  void reduce(T arg);                  // Define reduction
  T identity();                        // Define identity
}
accum<T>(Operator op, Class dataType); // Constructor
void accum.put(T datum);               // Remit datum
T accum.customGet();                   // Retrieve result
```

Error Conditions with Finish Accumulators

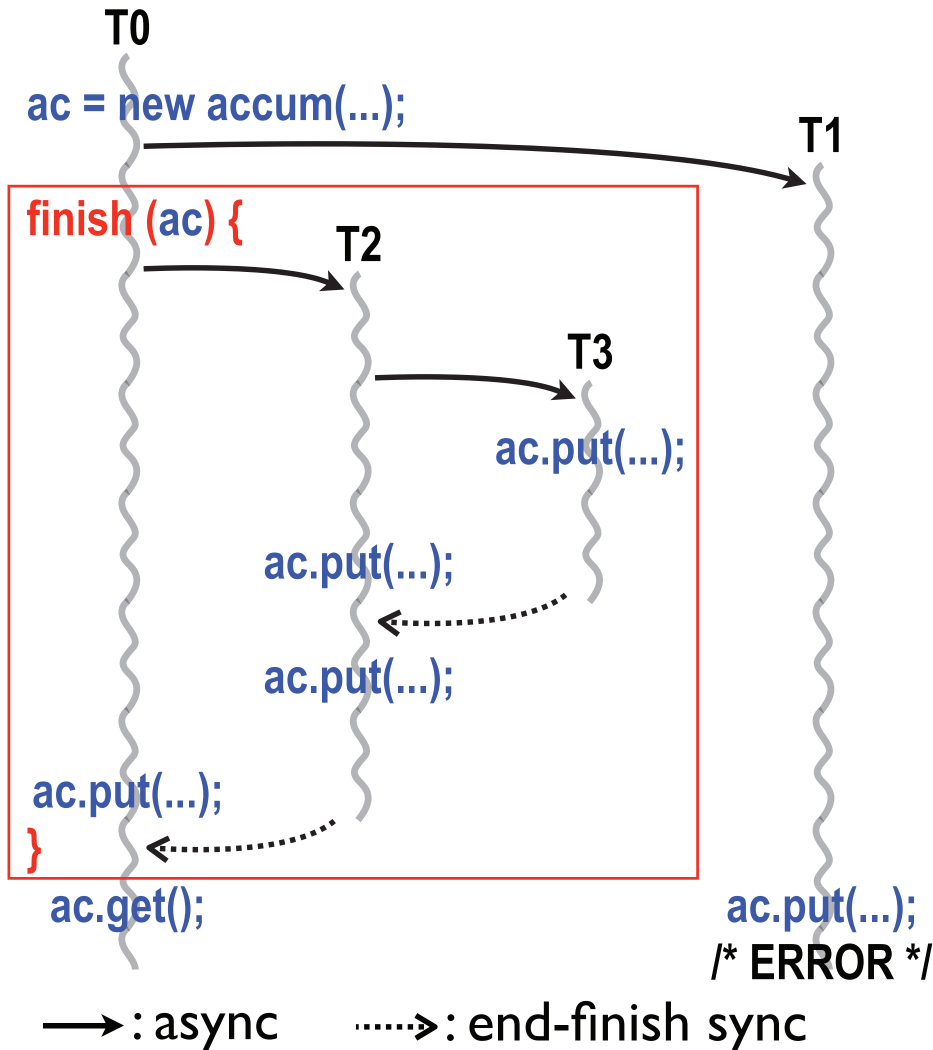
1. Non-owner task cannot access accumulators outside registered finish

```
// T1 allocates accumulator a
accumulator a = accumulator.factory.accumulator(...);
async { // T2 cannot access a
  a.put(1); Number v1 = a.get();
}
```

2. Non-owner task cannot register accumulators with a finish

```
// T1 allocates accumulator a
accumulator a = accumulator.factory.accumulator(...);
async {
  // T2 cannot register a with finish
  finish (a) { async a.put(1);  }
}
```

Example of Finish Accumulator with Three Tasks (Figure 16, Module 1 handout)




```
ac = new accum(...);
async { ... ac.put(foo()); }  // T1
finish (ac) {
  async {  // T2
    finish {
      async {  // T3
        ac.put(bar());
      }
      ac.put(baz());
    }
    ac.put(qux());
  }
  ac.put(quux());
}
n = ac.get();
```

Use of Finish Accumulators to count solutions in Parallel NQueens

```
1.  static accumulator a;
2.  aa = accumulator.factory.accumulator(SUM, int.class);
3.  finish(a) nqueens_kernel(new int[0], 0);
4.  System.out.println("No. of solutions = " + a.get().intValue())
5.  . . .
6.  void nqueens_kernel(int [] a, int depth) {
7.    if (size == depth) aa.put(1);
8.    else
9.      /* try each possible position for queen at depth */
10.     for (int i =  0; i < size; i++) async {
11.       /* allocate a temporary array and copy array a into it */
12.       int [] b = new int [depth+1];
13.       System.arraycopy(a, 0, b, 0, depth);
14.       b[depth] = i;
15.       if (ok(depth+1,b)) nqueens_kernel(b, depth+1);
16.     } // for-async
17. } // nqueens_kernel()
```

Worksheet #7: Why must Future References be declared as final?

Name 1: ______________________

Name 2: ______________________

1) Consider the code on the right with futures declared as non-final static fields (though that's not permitted in HJ). Can a deadlock situation occur between tasks T1 and T2 with this code? Explain why or why not.

```
1. static future<int> f1=null;
2. static future<int> f2=null;
3.
4. void main(String[] args) {
5.   f1 = async<int> {return a1();};
6.   f2 = async<int> {return a2();};
7. }
8.
9. int a1() { // Task T1
10.  while (f2 == null); // spin loop
11.  return f2.get(); //T1 waits for T2
12. }
13.
14. int a2() { // Task T2
15.  while (f1 == null); // spin loop
16.  return f1.get(); //T2 waits for T1
17. }
```

Worksheet #7: Why must Future References be declared as final (contd)?

2) Now consider a modified version of the above code in which futures are declared as final local variables (which is permitted in HJ). Can you add get() operations to methods a1() and a2() to create a deadlock between tasks T1 and T2 with this code? Explain why or why not.

Will your answer be different if f1 and f2 are final fields in objects or final static fields?

```
1. void main(String[] args) {
2.   final future<int> f1 =
3.     async<int> {return a1();};
4.   final future<int> f2 =
5.     async<int> {return a2();};
6. }
7.
8. int a1() { // Task T1
9.
10.
11.
12. }
13.
14. int a2() { // Task T2
15.
16.
17.
18. }
```