COMP 322: Fundamentals of Parallel Programming

Lecture 17: Midterm Review (Part 1)

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

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

Announcements

- No lecture on Friday, Feb 22nd.
- No labs or lab quizzes this week
- No new lecture quiz this week. The lecture quiz for Weeks 5 & 6 is due by Tuesday night.
- Homework 3 is due by by 11:55pm on Friday, February 22, 2013
- Take-home midterm exam (Exam 1) will be given after lecture on Wednesday, February 20, 2013
  - will need to be returned to Sherry Nassar (Duncan Hall 3137) by 4pm on Friday, February 22, 2013
  - Closed-book, closed computer written exam that can be taken in any 2-hour duration during that period

Scope of Midterm Exam

- Midterm exam will cover material from Lectures 1 - 15
  - Lecture 16 on Phaser Accumulators and Bounded Phasers is excluded
- Excerpts from midterm exam instructions
  - "Since this is a written exam and not a programming assignment, syntactic errors in program text will not be penalized (e.g., missing semicolons, incorrect spelling of keywords, etc) so long as the meaning of your solution is unambiguous."
  - "If you believe there is any ambiguity or inconsistency in a question, you should state the ambiguity or inconsistency that you see, as well as any assumptions that you make to resolve it."

Async and Finish Statements for Task Creation and Termination (Lecture 1)

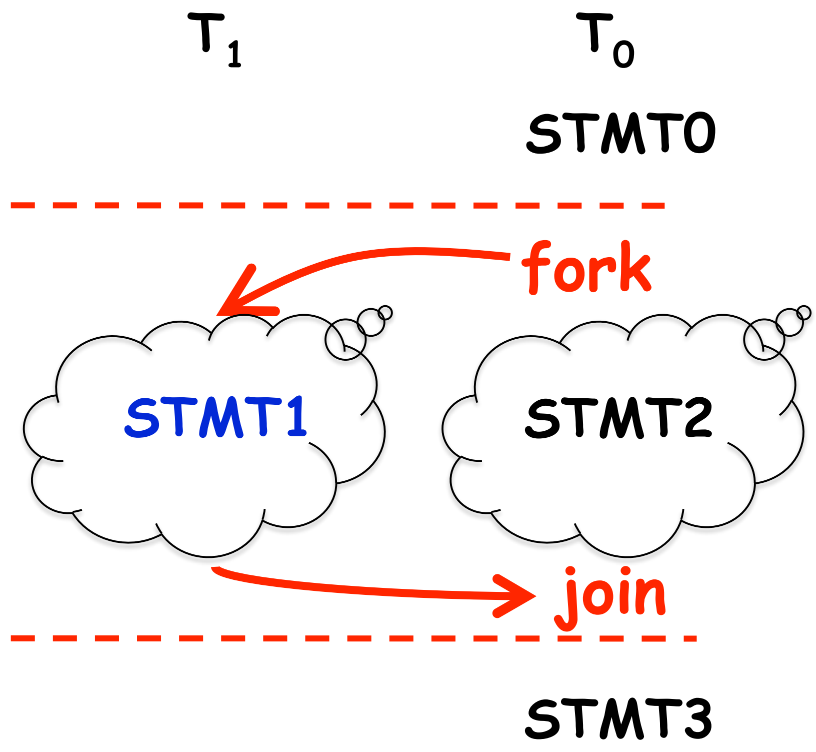
async S

- Creates a new child task that executes statement S

finish S

- Execute S, but wait until *all* asyncs in S's scope have terminated.

```
// 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
```



Worksheet #1 solution: Insert finish to get correct Two-way Parallel Array Sum program

```
// Start of Task T0 (main program)
sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
finish {
  async { // Task T1 computes sum of upper half of array
    for(int i=X.length/2; i < X.length; i++)
      sum2 += X[i];
  }
  // T0 computes sum of lower half of array
  for(int i=0; i < X.length/2; i++) sum1 += X[i];
}
// Task T0 waits for Task T1 (join)
return sum1 + sum2;
```

Solution to Homework 1, Question 1.1

- ... insert finish statements in the incorrect parallel version so as to make it correct i.e., to ensure that the parallel version computes the same result as the sequential version, while maximizing the potential parallelism.

- One possible solution:

```
finish
   for (int I = 0 ; I < N ; I++)
       for (int J = 0 ; J < N ; J++)
           async C[I][J] = 0;

finish
   for (int I = 0 ; I < N ; I++)
       for (int J = 0 ; J < N ; J++)
           async
               for (int K = 0 ; K < N ; K++)
                    C[I][J] += A[I][K] * B[K][J];

System.out.println(C[0][0]);
```

Computation Graphs for HJ Programs (Lecture 2)

- A Computation Graph (CG) captures the dynamic execution of an HJ program, for a specific input
- CG nodes are "steps" in the program's execution
  - A step is a sequential subcomputation without any async, begin-finish and end-finish operations
- CG edges represent ordering constraints
  - "Continue" edges define sequencing of steps within a task
  - "Spawn" edges connect parent tasks to child async tasks
  - "Join" edges connect the end of each async task to its IEF's end-finish operations
- All computation graphs must be acyclic
  - It is not possible for a node to depend on itself
- Computation graphs are examples of "directed acyclic graphs" (dags)

Complexity Measures for Computation Graphs

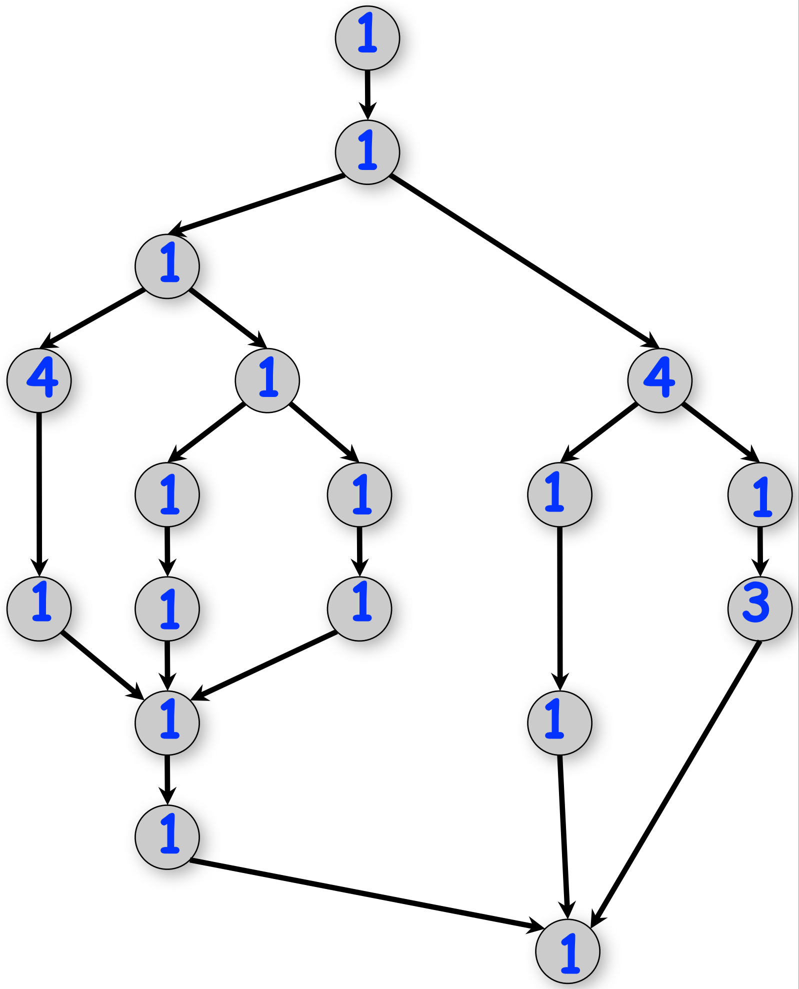
Define

- TIME(N) = execution time of node N
- WORK(G) = sum of TIME(N), for all nodes N in CG G
  - WORK(G) is the total work to be performed in G
- CPL(G) = length of a longest path in CG G, when adding up execution times of all nodes in the path
  - Such paths are called *critical paths*
  - CPL(G) is the length of these paths (critical path length)
  - CPL(G) is also the smallest possible execution time for the computation graph

Ideal Parallelism

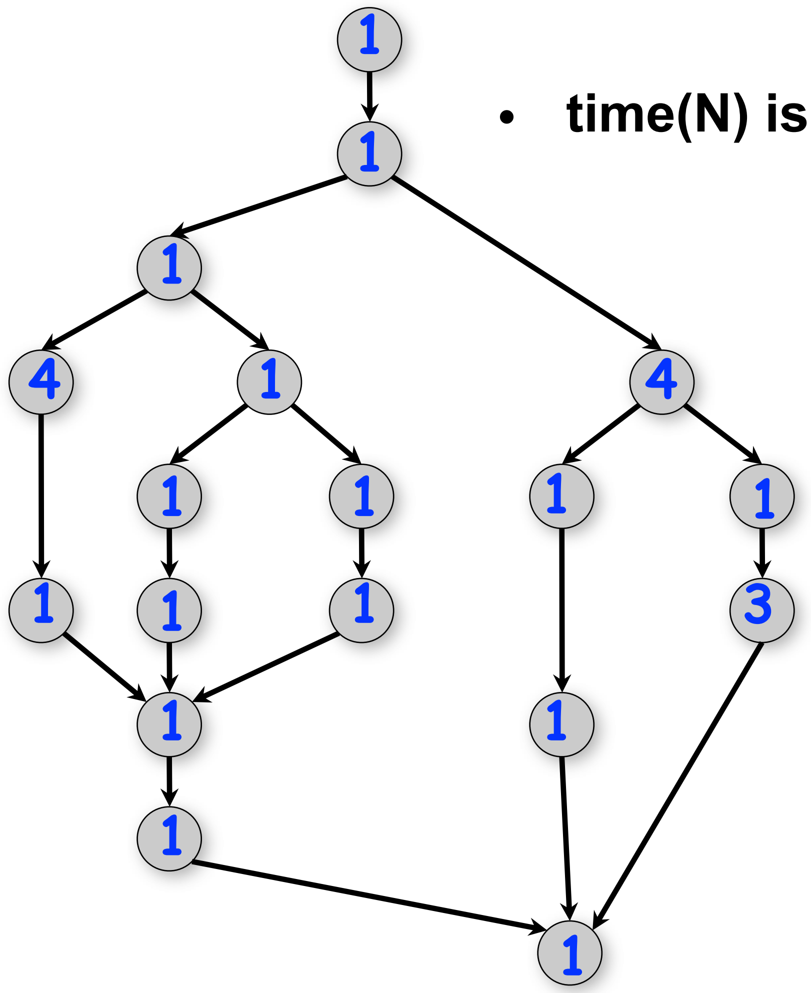
Define ideal parallelism of Computation G Graph as the ratio, WORK(G)/CPL(G)

Ideal Parallelism is independent of the number of processors that the program executes on, and only depends on the computation graph



Solution to Worksheet #2: what is the critical path length and ideal parallelism of this graph?

CPL(G) = length of a longest path in computation graph G



- time(N) is labeled for all nodes N in the graph

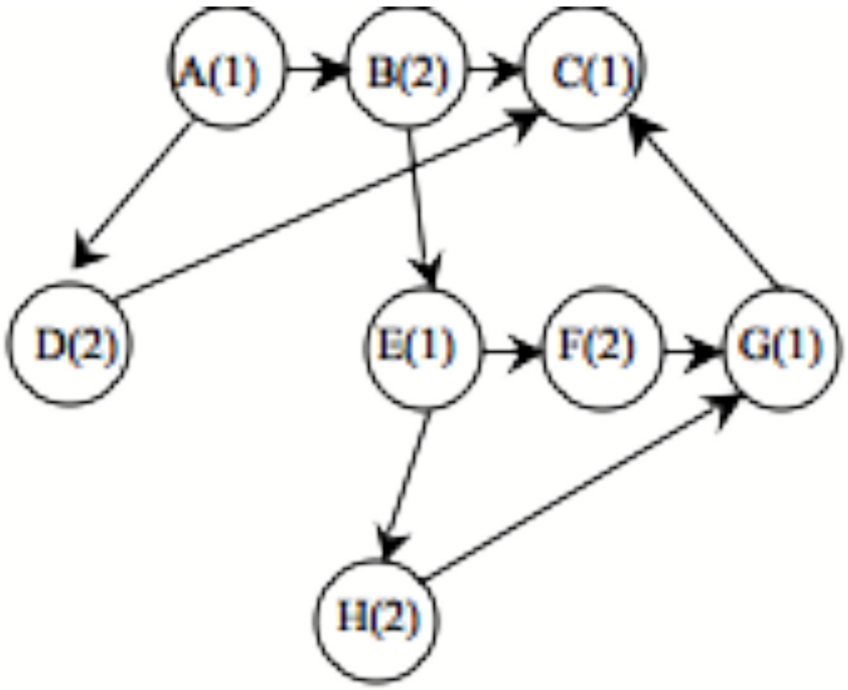
WORK(G) = 26

CPL(G) = 11

Ideal Parallelism
= WORK(G)/CPL(G)
= 26 / 11 ~ 2.36

Solution to Homework 1, Question 1.2.1

- Calculate the total WORK and CPL (critical path length) for this task graph. Each node is labeled with the steps name and execution time e.g., B(2) refers to step B with an execution time of 2 units.

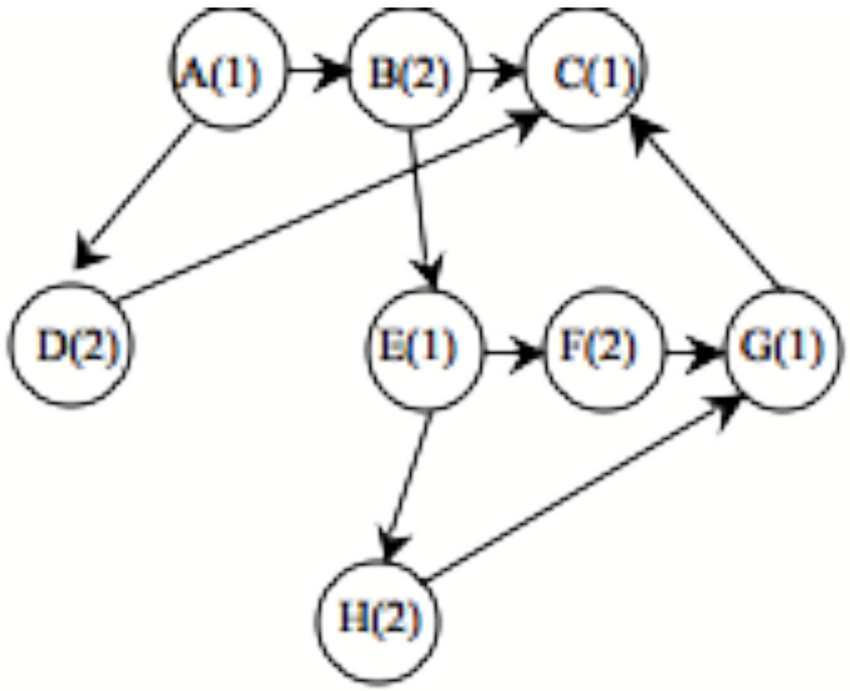


- WORK = 12
- CPL = 8

Solution to Homework 1, Question 1.2.2

- **Write a Habanero-Java program with basic finish and async constructs (no futures) that can generate this computation graph.**
- **One possible solution:**

```
finish {
    A;
    async D;
    B;
    async {
        E;
        finish { async H; F; }
        G;
    }
}
C;
```



Bounding the performance of Greedy Schedulers (Lecture 3)

Combine lower and upper bounds to get

$\max(\text{WORK}(G)/P, \text{CPL}(G)) \leq T_P \leq \text{WORK}(G)/P + \text{CPL}(G)$

Corollary 1: Any greedy scheduler achieves execution time T_P that is within a factor of 2 of the optimal time (since max(a,b) and (a+b) are within a factor of 2 of each other, for any $a \geq 0, b \geq 0$).

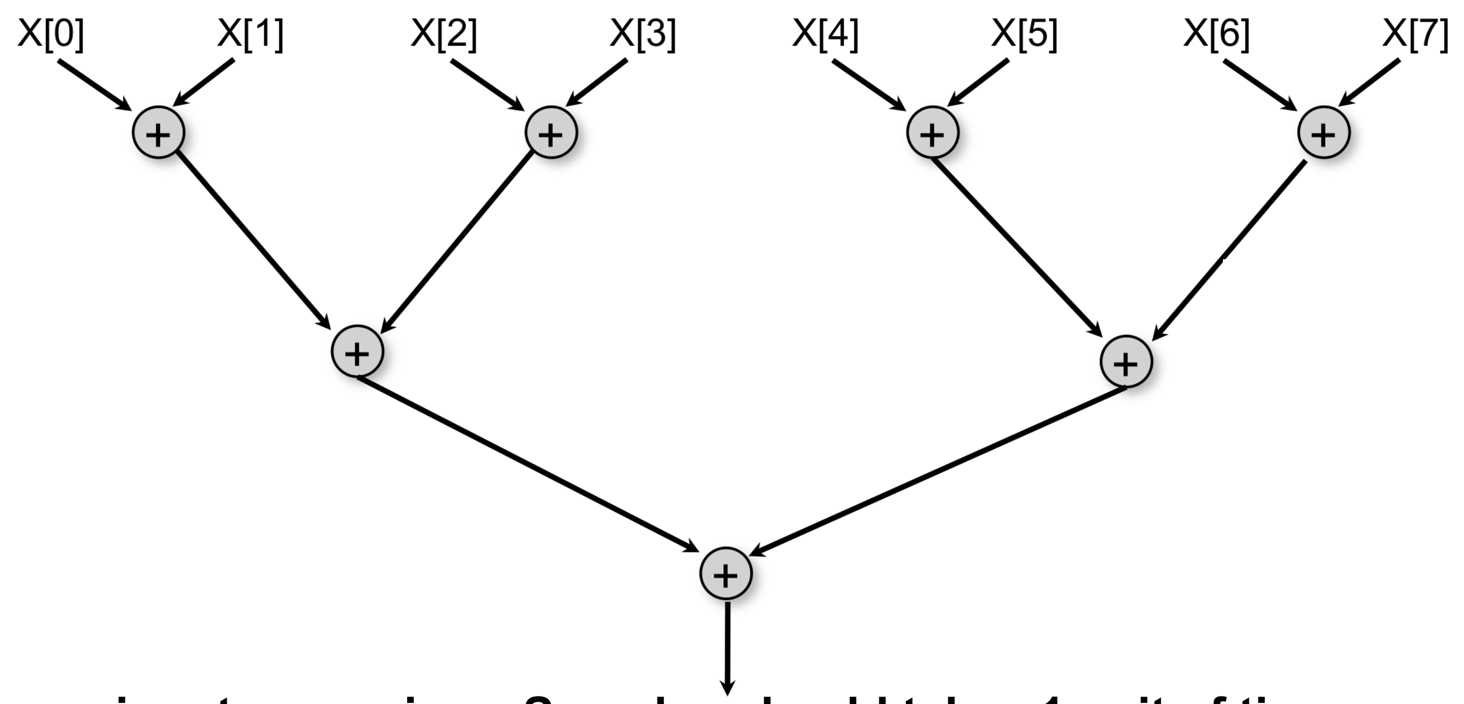
Corollary 2: Lower and upper bounds approach the same value whenever

- **There's lots of parallelism, WORK(G)/CPL(G) >> P**
- **Or there's little parallelism, WORK(G)/CPL(G) << P**

Strong Scaling and Speedup

- Define Speedup(P) = T_1 / T_P
  - Factor by which the use of P processors speeds up execution time relative to 1 processor, for a fixed input size
  - For ideal executions without overhead, $1 <= \text{Speedup}(P) <= P$
  - Linear speedup
    - When Speedup(P) = k*P, for some constant k, $0 < k < 1$
- Referred to as “strong scaling” because input size is fixed

Reduction Tree Schema for computing Array Sum in parallel



Assume input array size = S, and each add takes 1 unit of time:

- **WORK(G) = S-1**
- **CPL(G) = log2(S)**
- **Assume T_P = WORK(G)/P + CPL(G) = (S-1)/P + log2(S)**
  - **Within a factor of 2 of any schedule's execution time**

Implementation of Reduction Tree Schema in HJ (ArraySum1)

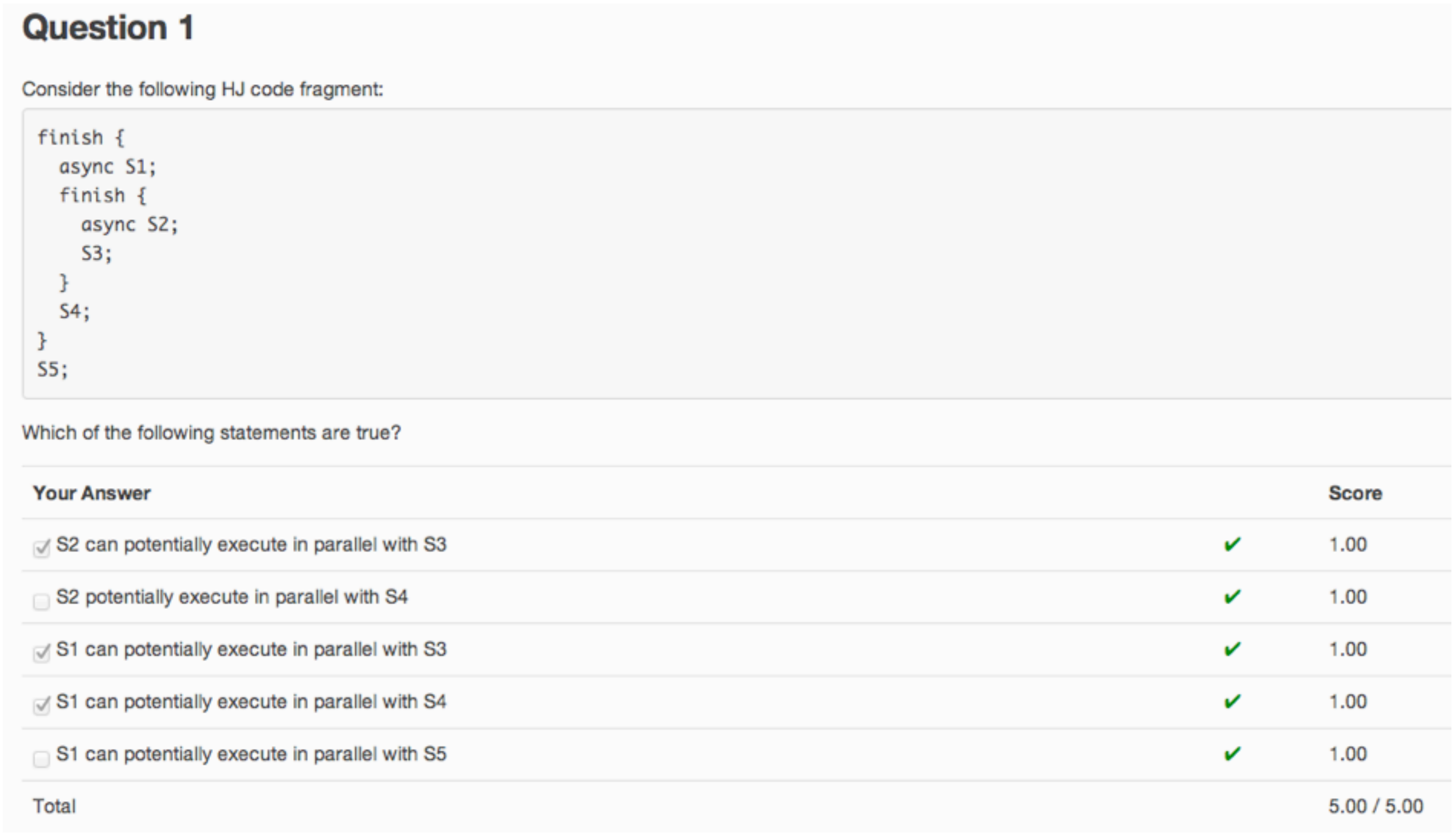
```
for ( int stride = 1; stride < X.length ; stride *= 2 ) {
  // Compute size = number of adds to be performed in stride
  int size=ceilDiv(X.length,2*stride);
  finish for(int i = 0; i < size; i++)
    async {
      if ( (2*i+1)*stride < X.length )
        X[2*i*stride] += X[(2*i+1)*stride];
    } // finish-for-async
} // for

// Divide x by y, and round up to next largest int
static int ceilDiv(int x, int y) { return (x+y-1) / y; }
```

Solution to Worksheet #3: Strong Scaling for Array Sum

- **Assume T(S,P) ~ WORK(G,S)/P + CPL(G,S) = (S-1)/P + log2(S) for a parallel array sum computation with input size S on P processors**
- **Strong scaling**
  - **Assume S = 1024 ==> log2(S) = 10**
  - **Compute Speedup(P) for S=1024 on 10, 100, 1000 processors**
    - **T(P) = 1023/P + 10**
    - **Speedup(10) = T(1)/T(10) ~ 9.2**
    - **Speedup(100) = T(1)/T(100) ~ 51.1**
    - **Speedup(1000) = T(1)/T(1000) ~ 102.3**
    - **Ideal parallelism = T(1)/T($_\infty$) = 1033/10 = 103.3**
  - **Why is it worse than linear?**
    - **The critical path limits speedup as P increases (speedup is limited by ideal parallelism)**

Week 1 Lecture Quiz Solution



Question 1

Consider the following HJ code fragment:

```
finish {
  async S1;
  finish {
    async S2;
    S3;
  }
  S4;
}
S5;
```

Which of the following statements are true?

Your Answer		Score
[✓] S2 can potentially execute in parallel with S3	✓	1.00
[ ] S2 potentially execute in parallel with S4	✓	1.00
[✓] S1 can potentially execute in parallel with S3	✓	1.00
[✓] S1 can potentially execute in parallel with S4	✓	1.00
[ ] S1 can potentially execute in parallel with S5	✓	1.00
Total		5.00 / 5.00

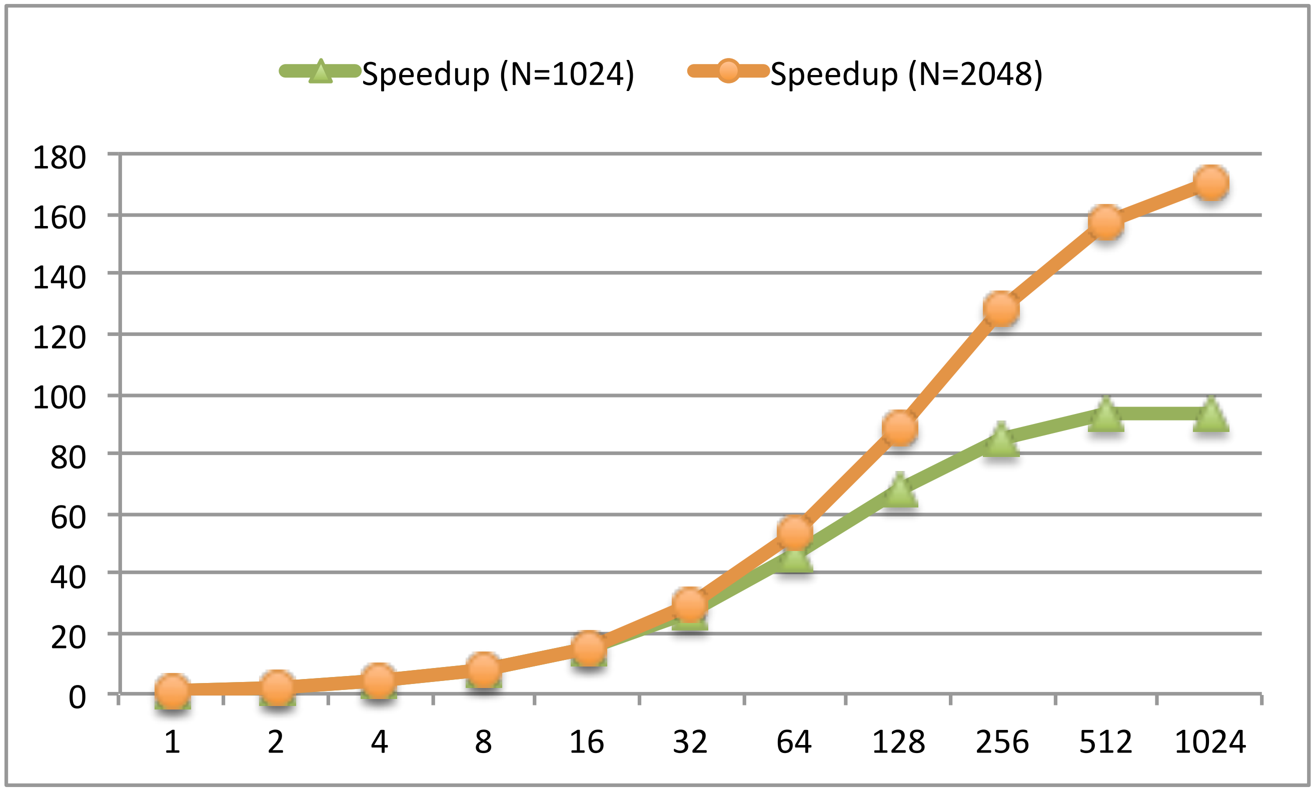
How many processors should we use? (Lecture 4)

- Efficiency(P) = Speedup(P)/ P = $T_1/(P * T_P)$
  - —Processor efficiency --- figure of merit that indicates how well a parallel program uses available processors
  - —For ideal executions without overhead, 1/P <= Efficiency(P) <= 1
- Half-performance metric
  - —$S_{1/2}$ = input size that achieves Efficiency(P) = 0.5 for a given P
  - —Figure of merit that indicates how large an input size is needed to obtain efficient parallelism
  - —A larger value of $S_{1/2}$ indicates that the problem is harder to parallelize efficiently
- How many processors to use?
  - —Common goal: choose number of processors, P for a given input size, S, so that efficiency is at least 0.5

ArraySum: Speedup as function of array size, S, and number of processors, P

- Speedup(S,P) = T(S,1)/T(S,P) = S/(S/P + $\log_2$(S))
- Asymptotically, Speedup(S,P) --> S/$\log_2$S, as P --> infinity

Speedup(S,P)



P

Solution to Worksheet #4: how many processors should we use for ArraySum?

For ArraySum on P processors and input array size, S,

$\text{Speedup}(S,P) = T(S,1)/T(S,P) = S/(S/P + \log_2(S))$

- Question: For a given N, what value of P should we choose to obtain Efficiency(P) = 0.5? Recall that Efficiency(P) = 0.5 $\Rightarrow$ Speedup(N,P)/P = 0.5.

- Answer (derive value of P as a symbolic function of N):

  $. . . \Rightarrow P = N/\log_2(N)$

- Check answer by observing that $N/P = \log_2(N) \Rightarrow$

  $\text{Speedup}(N,P) = N/(\log_2(N) + \log_2(N)) = N/(2*\log_2(N)) = P/2$

Amdahl's Law [1967]

- If $q \leq 1$ is the fraction of WORK in a parallel program that <u>must be executed sequentially</u> for a given input size S, then the best speedup that can be obtained for that program is Speedup(S,P) $\leq 1/q$.
- Observation follows directly from critical path length lower bound on parallel execution time
  - CPL >= q * T(S,1)
  - T(S,P) >= q * T(S,1)
  - Speedup(S,P) = T(S,1)/T(S,P) <= 1/q
- This upper bound on speedup simplistically assumes that work in program can be divided into sequential and parallel portions
  - Sequential portion of WORK = q
    - also denoted as f_S (fraction of sequential work)
  - Parallel portion of WORK = 1-q
    - also denoted as f_p (fraction of parallel work)
- Computation graph is more general and takes dependences into account

Homework 2, Question 1.1

- In Lecture 4, you learned the following statement of Amdahl's Law:
  - —If $q \leq 1$ is the fraction of WORK in a parallel program that must be executed sequentially, then the best speedup that can be obtained for that program, even with an unbounded number of processors, is Speedup $\leq 1/q$.
- Now, consider the following generalization of Amdahl's Law. Let q1 be the fraction of WORK in a parallel program that must be executed sequentially, and q2 be the fraction of WORK that can use at most 2 processors. Assume that the fractions of WORK represented by q1 and q2 are disjoint. Your assignment is to provide an upper bound on the Speedup as a function of q1 and q2, and justify why it is a correct upper bound. (Hint: to check your answer, consider the cases when q1=0 or q2=0.)

Solution to Homework 2, Question 1.1

- Total WORK can be divided into three categories
  - q1*WORK = work that must be executed sequentially
  - q2*WORK = work can can use at most 2 processors
  - (1-q1-q2)*WORK = work that can use an unbounded number of processors

- Lower bound on execution time on P processors
  - T(p) >= q1*WORK + q2*WORK/2 + (1-q1-q2)*WORK/P

- Upper bound on Speedup
  - Speedup(P) = T(1) / T(P)
  - <= WORK / (q1*WORK + q2*WORK/2 + (1-q1-q2)*WORK/P)

⇒ SpeedUp(P) <= 1/ (q1 + q2/2 + (1-q1 - q2) / P) <= 1/(q1 + q2/2)

Weak Scaling

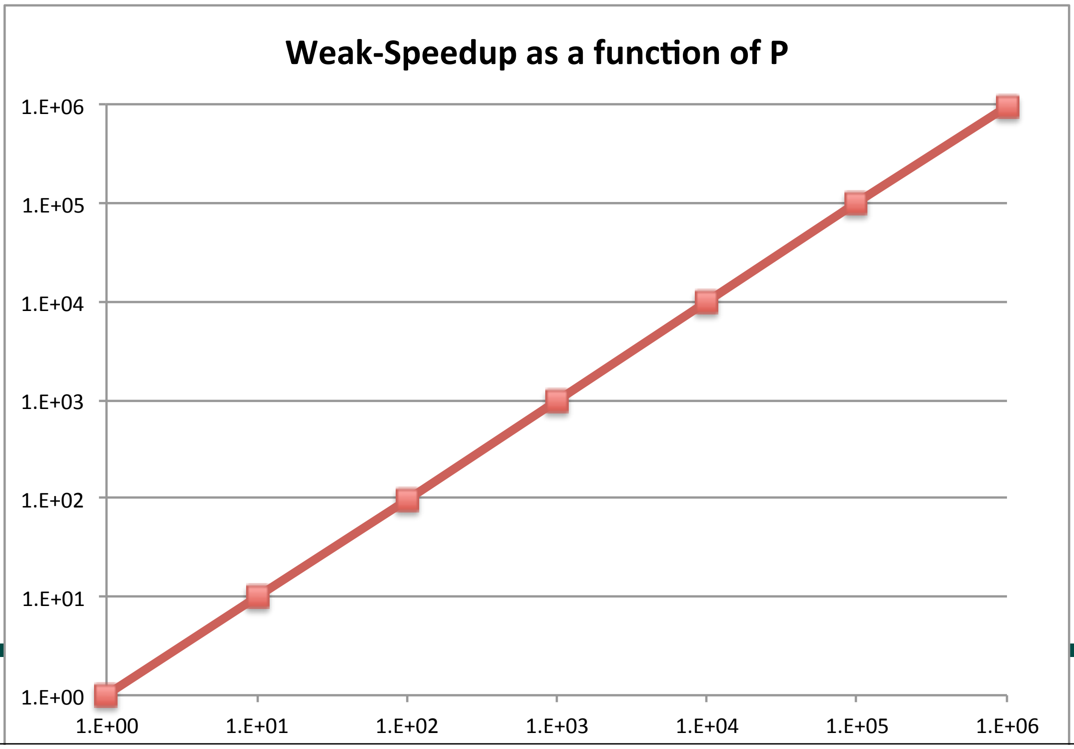
- Consider a computation graph, CG, in which all node execution times are parameterized by input size S
  - —TIME(N,S) = time to execute node N with input size S
  - —WORK(G,S) = sum of TIME(N,S) for all nodes N
  - —CPL(G,S) = critical path length for G, assuming node N takes TIME(N,S)
- Let T(S,P) = time to execute CG with input size S on P processors
- Weak scaling
  - —Allow input size S to increase with number of processors i.e., make S a function of P
  - —Define Weak-Speedup(S(P),P) = T(S(P),1)/T(S(P),P), where input size S(P) increases with P
    - – Note that T(S(P),1) is a hypothetical projection of running a larger problem size, S(P), on 1 processor

Weak Scaling for Array Sum

- Recall that T(S,P) = (S-1)/P + log2(S) for a parallel array sum computation
- For weak scaling, assume S(P) = 1024*P

==> Weak-Speedup(S(P),P) = T(S(P),1)/T(S(P),P)

= ((1024*P-1)+log2(1024*P)) / ((1024*P-1)/P+log2(1024*P)) ~ P



Formal Definition of Data Races (Lecture 5)

Formally, a data race occurs on location L in a program execution with computation graph CG if there exist steps (nodes) S1 and S2 in CG such that:

1. **S1 does not depend on S2 and S2 does not depend on S1 i.e., there is no path of dependence edges from S1 to S2 or from S2 to S1 in CG, and**
2. **Both S1 and S2 read or write L, and at least one of the accesses is a write. (L must be a shared location i.e., a static field, instance field, or array element.)**

Data races are challenging because of

- **Nondeterminism: different executions of the parallel program with the same input may result in different outputs.**
- **Debugging and Testing: it is usually impossible to guarantee that all possible orderings of the accesses to a location will be encountered during program debugging and testing.**

Relating Data Races and Determinism

- A parallel program is said to be deterministic with respect to its inputs if it always computes the same answer when given the same inputs.
- Structural Determinism Property
  - If a parallel program is written using the constructs in Module 1 and is guaranteed to be race-free, then it must be deterministic with respect to its inputs. The final computation graph is also guaranteed to be the same for all executions of the program with the same inputs.
- Constructs introduced in Module 1 ("Deterministic Shared-Memory Parallelism") include async, finish, finish accumulators, futures, data-driven tasks (async await), forall, barriers, phasers, and phaser accumulators.
  - The notable exceptions are critical sections, isolated statements, and actors, all of which will be covered in Module 2 ("Nondeterministic Shared-Memory Parallelism")

Solution to Worksheet #5: Data Races and Determinism

Consider a modified String Search program that returns true if any occurrence is found, rather than the count of all occurrences:

```
static boolean found = false; // static field
. . .
finish for (int i = 0; i <= N - M; i++)
  async {
    int j;
    for (j = 0; j < M; j++)
      if (text[i+j] != pattern[j]) break;
    if (j == M) found = true; // found at offset i
  } // finish-for-async
```

Questions:

1. **Does this program have a data race?**

   Yes. Multiple async tasks can write to the same static field, found.

2. **Is it deterministic?**

   Yes. The answer will be the same regardless of the order of the writes.

3. **Is it structurally deterministic?**

   Yes. The computation graph will always be the same for the same input.

HJ Futures: Tasks with Return Values (Lecture 6)

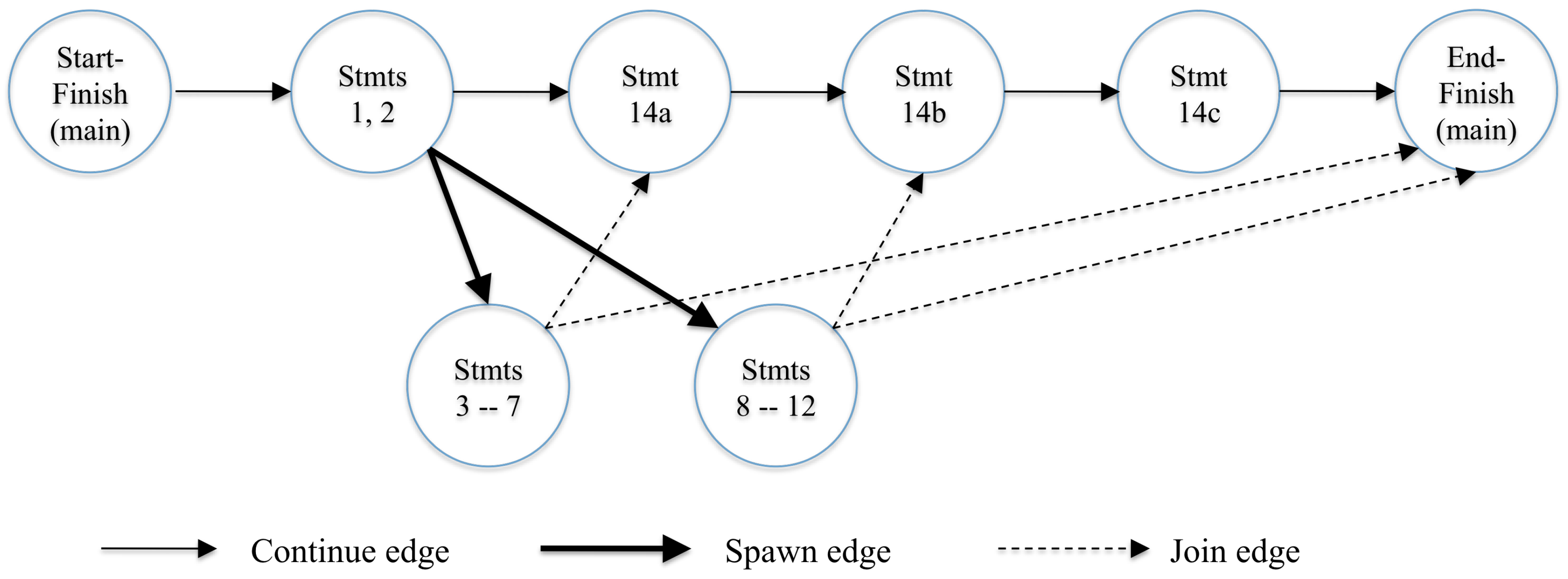
`async<T> { Stmt-Block }`

- 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>
- 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

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



NOTE: DrHJ's data race detection tool does not support futures as yet (it only supports finish, async, and isolated constructs)

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

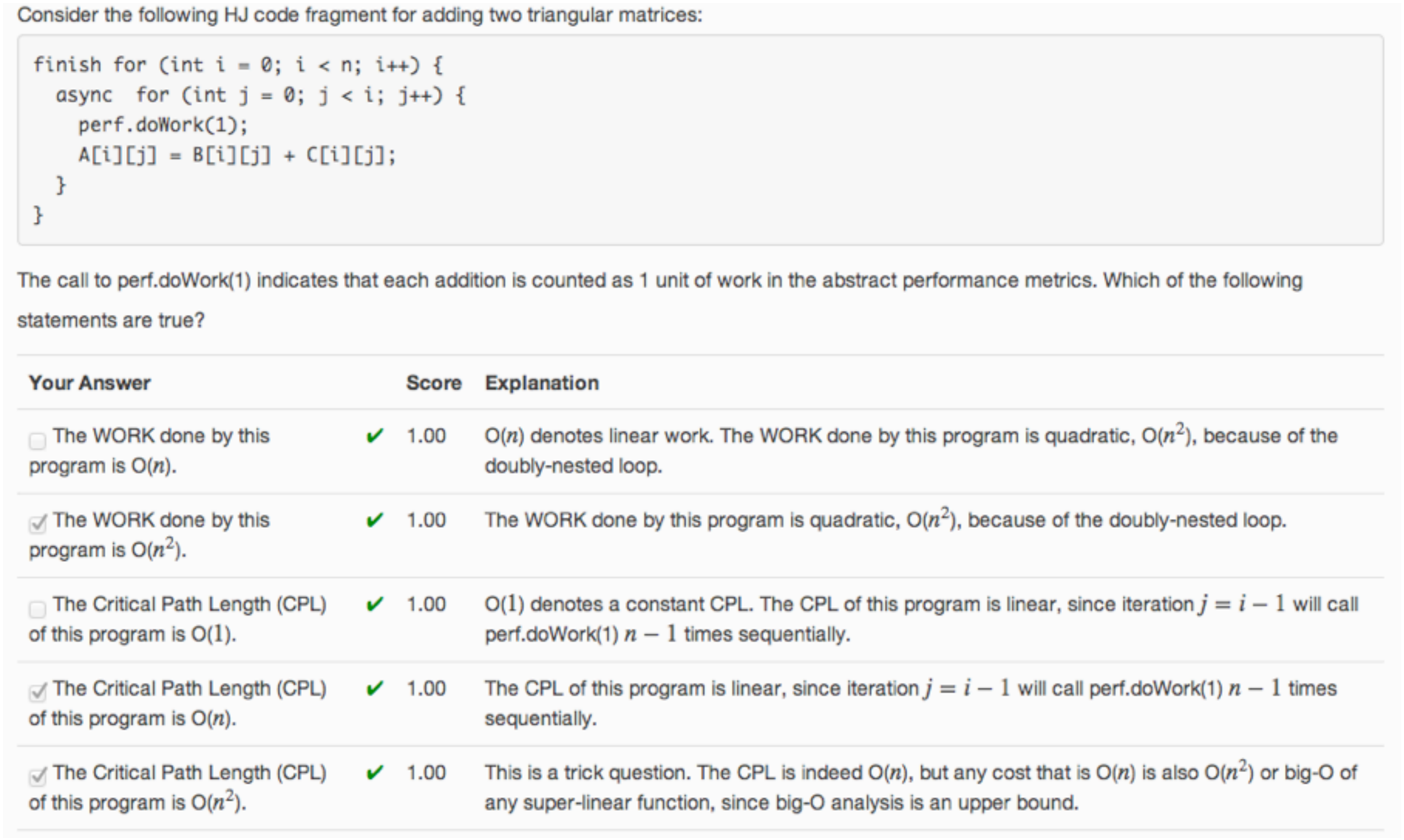
1) Can you write an HJ program with async-finish constructs that generates a Computation Graph with the same ordering constraints as the graph on the right?
No

$$\begin{tikzcd}
& A \arrow[dl] \arrow[dr] & & \\
B \arrow[dr] & & C \arrow[dl] \arrow[dr] & \\
& D \arrow[dr] & & E \arrow[dl] \\
& & F &
\end{tikzcd}$$

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
```

Week 2 Lecture Quiz Solution: Question 1



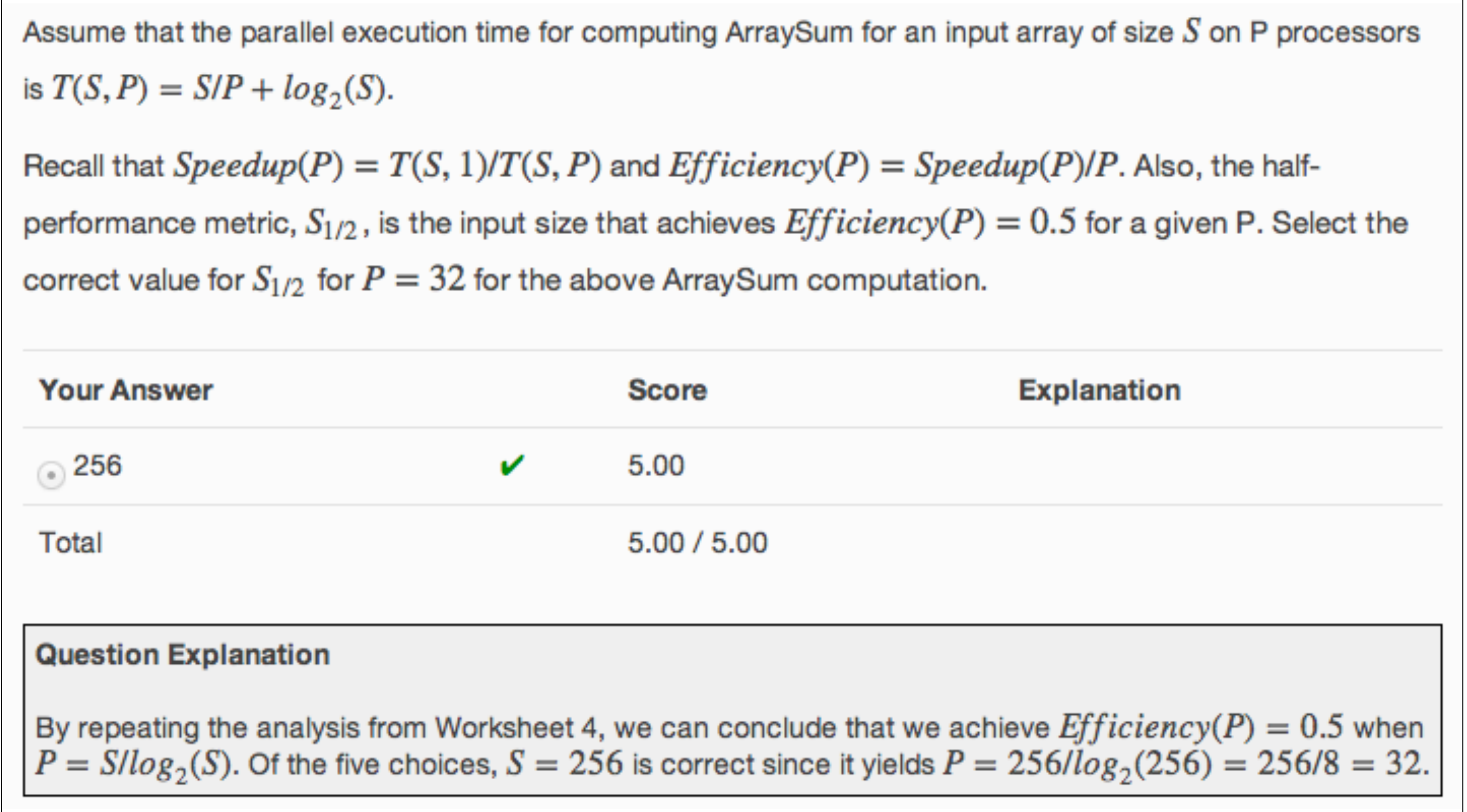
Consider the following HJ code fragment for adding two triangular matrices:

```
finish for (int i = 0; i < n; i++) {
  async  for (int j = 0; j < i; j++) {
    perf.doWork(1);
    A[i][j] = B[i][j] + C[i][j];
  }
}
```

The call to perf.doWork(1) indicates that each addition is counted as 1 unit of work in the abstract performance metrics. Which of the following statements are true?

Your Answer		Score	Explanation
☐ The WORK done by this program is O(n).	✔	1.00	O(n) denotes linear work. The WORK done by this program is quadratic, O(n^2), because of the doubly-nested loop.
☑ The WORK done by this program is O(n^2).	✔	1.00	The WORK done by this program is quadratic, O(n^2), because of the doubly-nested loop.
☐ The Critical Path Length (CPL) of this program is O(1).	✔	1.00	O(1) denotes a constant CPL. The CPL of this program is linear, since iteration $j = i - 1$ will call perf.doWork(1) $n - 1$ times sequentially.
☑ The Critical Path Length (CPL) of this program is O(n).	✔	1.00	The CPL of this program is linear, since iteration $j = i - 1$ will call perf.doWork(1) $n - 1$ times sequentially.
☑ The Critical Path Length (CPL) of this program is O(n^2).	✔	1.00	This is a trick question. The CPL is indeed O(n), but any cost that is O(n) is also O(n^2) or big-O of any super-linear function, since big-O analysis is an upper bound.

Week 2 Lecture Quiz Solution: Question 2



Assume that the parallel execution time for computing ArraySum for an input array of size S on P processors is $T(S,P) = S/P + log_2(S)$.

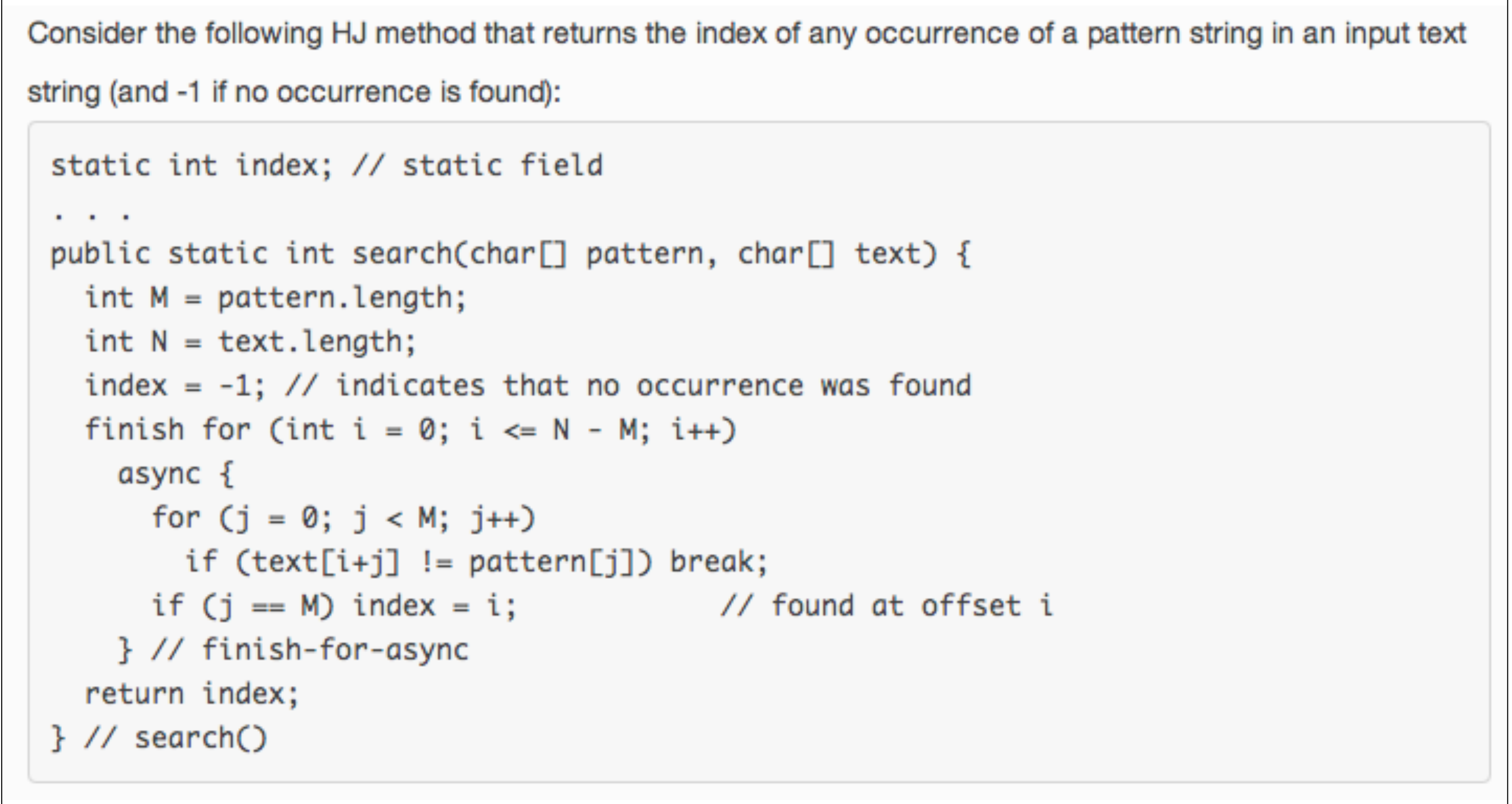
Recall that $Speedup(P) = T(S,1)/T(S,P)$ and $Efficiency(P) = Speedup(P)/P$. Also, the half-performance metric, $S_{1/2}$, is the input size that achieves $Efficiency(P) = 0.5$ for a given P. Select the correct value for $S_{1/2}$ for $P = 32$ for the above ArraySum computation.

Your Answer		Score	Explanation
● 256	✓	5.00	
Total		5.00 / 5.00	

Question Explanation

By repeating the analysis from Worksheet 4, we can conclude that we achieve $Efficiency(P) = 0.5$ when $P = S/log_2(S)$. Of the five choices, $S = 256$ is correct since it yields $P = 256/log_2(256) = 256/8 = 32$.

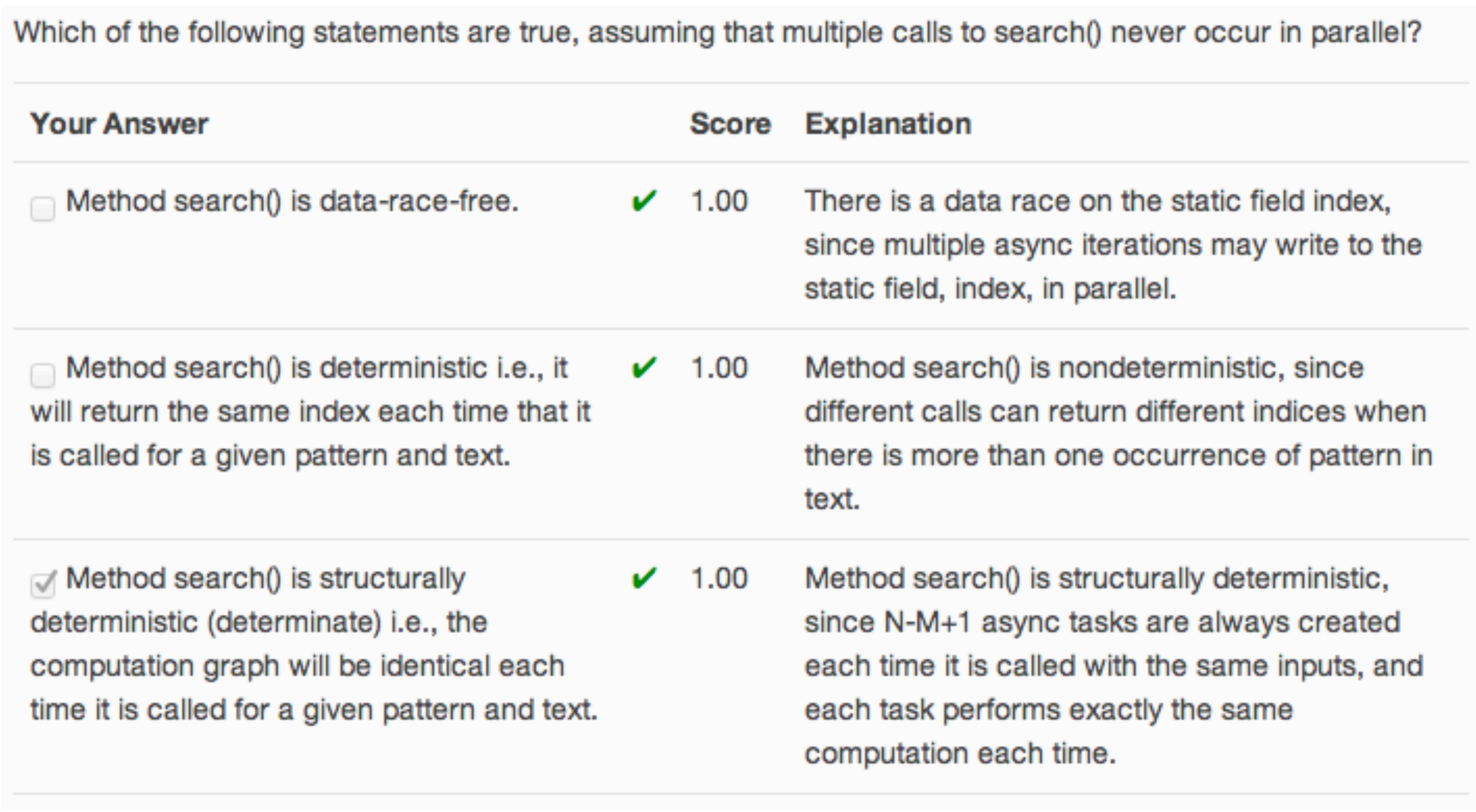
Week 2 Lecture Quiz Solution: Question 3



Consider the following HJ method that returns the index of any occurrence of a pattern string in an input text string (and -1 if no occurrence is found):

```
static int index; // static field
. . .
public static int search(char[] pattern, char[] text) {
  int M = pattern.length;
  int N = text.length;
  index = -1; // indicates that no occurrence was found
  finish for (int i = 0; i <= N - M; i++)
    async {
      for (j = 0; j < M; j++)
        if (text[i+j] != pattern[j]) break;
      if (j == M) index = i;            // found at offset i
    } // finish-for-async
  return index;
} // search()
```

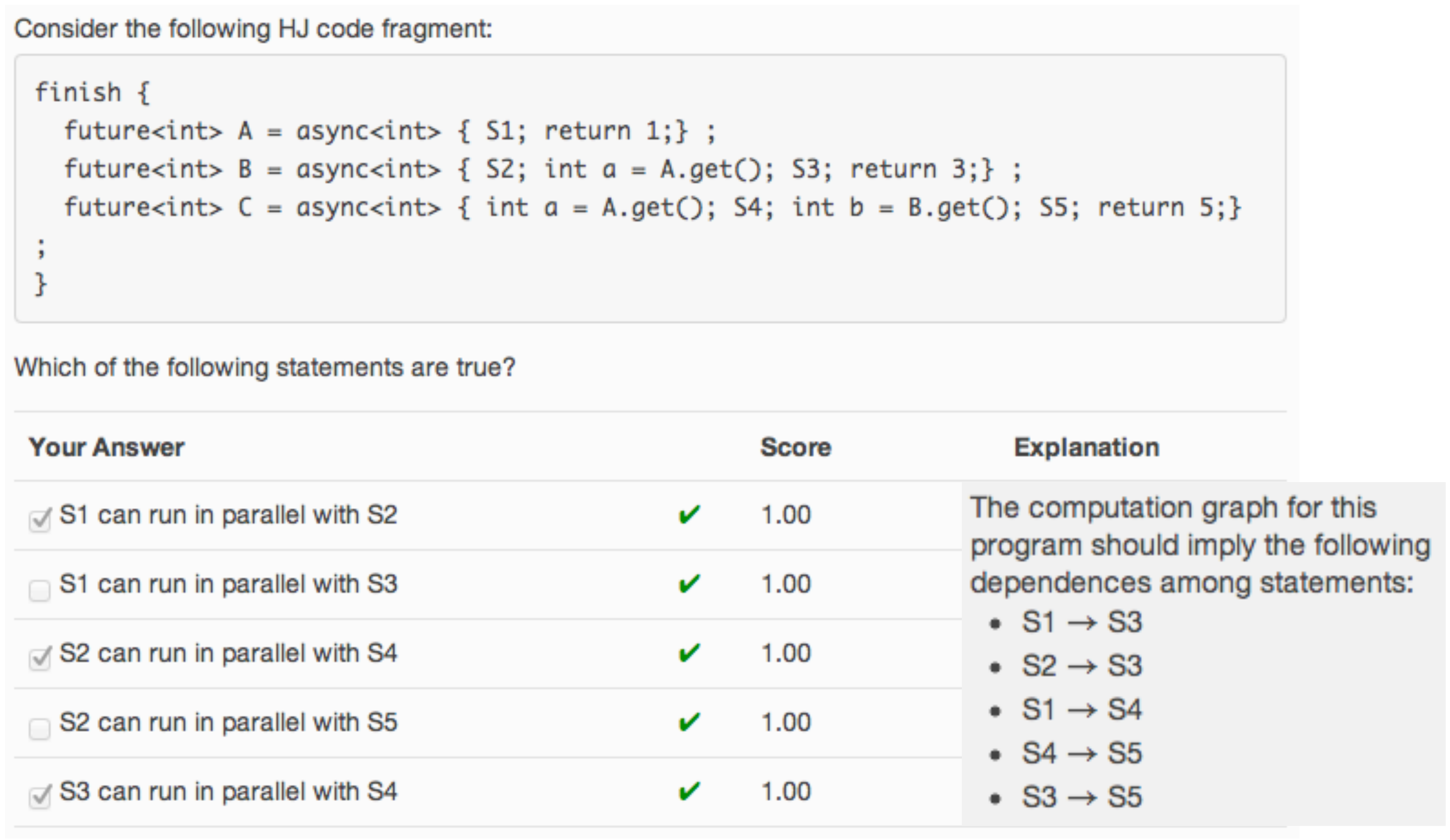
Week 2 Lecture Quiz Solution: Question 3 (contd)



Which of the following statements are true, assuming that multiple calls to search() never occur in parallel?

Your Answer		Score	Explanation
☐ Method search() is data-race-free.	✓	1.00	There is a data race on the static field index, since multiple async iterations may write to the static field, index, in parallel.
☐ Method search() is deterministic i.e., it will return the same index each time that it is called for a given pattern and text.	✓	1.00	Method search() is nondeterministic, since different calls can return different indices when there is more than one occurrence of pattern in text.
✓ Method search() is structurally deterministic (determinate) i.e., the computation graph will be identical each time it is called for a given pattern and text.	✓	1.00	Method search() is structurally deterministic, since N-M+1 async tasks are always created each time it is called with the same inputs, and each task performs exactly the same computation each time.

Week 2 Lecture Quiz Solution: Question 4



Consider the following HJ code fragment:

```
finish {
  future<int> A = async<int> { S1; return 1;} ;
  future<int> B = async<int> { S2; int a = A.get(); S3; return 3;} ;
  future<int> C = async<int> { int a = A.get(); S4; int b = B.get(); S5; return 5;}
;
}
```

Which of the following statements are true?

Your Answer		Score	Explanation
☑ S1 can run in parallel with S2	✔	1.00	The computation graph for this program should imply the following dependences among statements:
☐ S1 can run in parallel with S3	✔	1.00	• S1 → S3 • S2 → S3
☑ S2 can run in parallel with S4	✔	1.00	• S1 → S4
☐ S2 can run in parallel with S5	✔	1.00	• S4 → S5
☑ S3 can run in parallel with S4	✔	1.00	• S3 → S5

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

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.

Yes, a deadlock can occur when future f1 does f2.get() and future f2 does f1.get().

WARNING: such "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) Their semantics depends on the memory model. In HJ's memory model, there's no guarantee that the above spin loops will ever terminate.

```
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
}
```

deadlock

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

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.

No, the final declarations make it impossible for future f1's task (T1) to receive a reference to f2.

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

No.

```
void main(String[] args) {
   final future<int> f1 =
     async<int> {return a1();};
   final future<int> f2 =
     async<int> {return a2(f1);};
}

 int a1() {
 // Task T1 cannot receive a
 // reference to f2

 }

 int a2(future<int> f1) {
 // Task T2 can receive a reference
 // to f1 but that won't cause
 // a deadlock.
 ... f1.get() ...
 }
```

Finish Accumulators (Lecture 7)

- **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*

Sequential solution for NQueens (Lecture 8)

```
static int count;
. . .
count = 0;
nqueens_kernel(new int[0], 0);
System.out.println("No. of solutions = " + count);
. . .
void nqueens_kernel(int [] a, int depth) {
  if (size == depth) count++;
  else
    /* try each possible position for queen at depth */
    for (int i =  0; i < size; i++) {
      /* allocate a temporary array and copy array a into it */
      int [] b = new int [depth+1];
      System.arraycopy(a, 0, b, 0, depth);
      b[depth] = i;
      if (ok(depth+1,b)) nqueens_kernel(b, depth+1);
    } // for-async
} // nqueens_kernel()
```

Parallel Solution to NQueens with Finish Accumulators (counting all solutions)

```
static accumulator count;
. . .
count = accumulator.factory.accumulator(SUM, int.class);
finish(count) nqueens_kernel(new int[0], 0);
System.out.println("No. of solutions = " + count.get().intValue());
. . .
void nqueens_kernel(int [] a, int depth) {
  if (size == depth) count.put(1);
  else
    /* try each possible position for queen at depth */
    for (int i =  0; i < size; i++) async {
      /* allocate a temporary array and copy array a into it */
      int [] b = new int [depth+1];
      System.arraycopy(a, 0, b, 0, depth);
      b[depth] = i;
      if (ok(depth+1,b)) nqueens_kernel(b, depth+1);
    } // for-async
} // nqueens_kernel()
```