COMP 322: Fundamentals of Parallel Programming
Module 1: Parallelism

February 4, 2016

DRAFT VERSION

Contents

0 Course Organization

The desired learning outcomes from the course fall into three major areas, that we refer to as *modules*:

- *Module 1: Parallelism* — creation and coordination of parallelism (async, finish), abstract performance metrics (work, critical paths), Amdahl's Law, weak vs. strong scaling, data races and determinism, data race avoidance (immutability, futures, accumulators, dataflow), deadlock avoidance, abstract vs. real performance (granularity, scalability), collective and point-to-point synchronization (phasers, barriers), parallel algorithms, systolic algorithms.
- *Module 2: Concurrency* — critical sections, atomicity, isolation, high level data races, nondeterminism, linearizability, liveness/progress guarantees, actors, request-response parallelism, Java Concurrency, locks, condition variables, semaphores, memory consistency models.
- *Module 3: Locality and Distribution* — memory hierarchies, locality, cache affinity, data movement, message-passing (MPI), communication overheads (bandwidth, latency), MapReduce, accelerators, GPGPUs, CUDA, OpenCL.

Each module is further divided into *units*, and each unit consists of a set of *topics*. This document consists of lecture notes for Module 1. The section numbering in the document follows the *unit.topic* format. Thus, Section 1.2 in the document covers topic 2 in unit 1. The same numbering convention is used for the videos hosted on edX.

1 Task-level Parallelism

1.1 Task Creation and Termination (Async, Finish)

To introduce you to a concrete example of parallel programming, let us first consider the following sequential algorithm for computing the sum of the elements of an array of numbers, X:

Algorithm 1: Sequential ArraySum

Input: Array of numbers, X.
Output: sum = sum of elements in array X.
$sum \leftarrow 0$;
for $i \leftarrow 0$ **to** $X.length - 1$ **do**
  $sum \leftarrow sum + X[i]$;
return sum;

This algorithm is simple to understand since it sums the elements of X sequentially from left to right. However, we could have obtained the same algebraic result by summing the elements from right to left instead. This over-specification of the ordering of operations in sequential programs has been classically referred to as the *Von Neumann bottleneck* [2][1]. The left-to-right evaluation order in Algorithm 1 can be seen in the *computation graph* shown in Figure 1. We will study computation graphs formally later in the course. For now, think of each node or vertex (denoted by a circle) as an operation in the program and each edge (denoted by an arrow) as an ordering constraint between the operations that it connects, due to the flow of the output from the first operation to the input of the second operation. It is easy to see that the computation graph in Figure 1 is sequential because the edges enforce a linear order among all nodes in the graph.

How can we go about converting Algorithm 1 to a parallel program? The answer depends on the parallel programming constructs that are available for our use. We will start by learning *task-parallel* constructs. To

[1] These lecture notes include citation such as [2] as references for **optional** further reading.

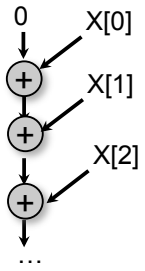

Figure 1: Computation graph for Algorithm 1 (Sequential ArraySum)

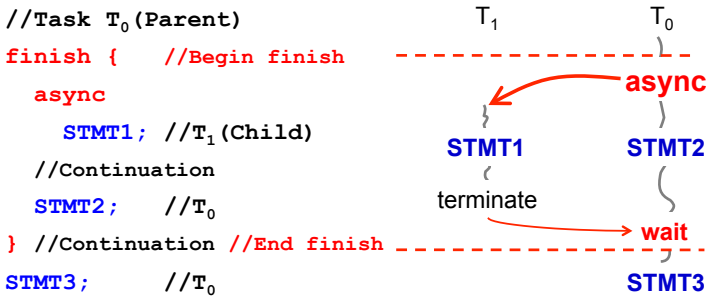

Figure 2: A example code schema with async and finish constructs

understand the concept of tasks informally, let's use the word, *task*, to denote a sequential subcomputation of a parallel program. A task can be made as small or as large as needed e.g., it can be a single statement or can span multiple procedure calls. Program execution is assumed to start as a single "main program" task, but tasks can create new tasks leading to a tree of tasks defined by parent-child relations arising from task creation, in which the main program task is the root. In addition to *task creation*, we will also need a construct for *task termination*, i.e., a construct that can enable certain computations to wait until certain other tasks have terminated. With these goals in mind, we introduce two fundamental constructs for task parallelism, *async* and *finish*, in the following sections[2].

1.1.1 Async notation for Task Creation

The first parallel programming construct that we will learn is called *async*. In pseudocode notation, "**async** ⟨*stmt1*⟩", causes the parent task (i.e., the task executing the async statement to create a new child task to execute the body of the *async*, ⟨*stmt1*⟩, *asynchronously* (*i.e.,* before, after, or in parallel) with the remainder of the parent task. The notation, ⟨*stmt*⟩, refers to any legal program statement *e.g.,* if-then-else, for-loop, method call, or a block enclosed in { } braces. (The use of angle brackets in "⟨*stmt*⟩" follows a standard notational convention to denote units of a program. They are unrelated to the < and > comparison operators used in many programming languages.) Figure 2 illustrates this concept by showing a code schema in which the parent task, T_0, uses an async construct to create a child task T_1. Thus, `STMT1` in task T_1 can potentially execute in parallel with `STMT2` in task T_0.

`async` is a powerful primitive because it can be used to enable any statement to execute as a parallel task, including for-loop iterations and method calls. Listing 1 shows some example usages of `async`. These examples are illustrative of logical parallelism, since it may not be efficient to create separate tasks for all the parallelism created in these examples. Later in the course, you will learn the impact of overheads in determining what subset of logical parallelism can be useful for a given platform.

[2]These constructs have some similarities to the "fork" and "join" constructs available in many languages, including Java's ForkJoin framework (which we will learn later in the course), but there are notable differences.

```
1  // Example 1: execute iterations of a counted for loop in parallel
2  // (we will later see forall loops as a shorthand for this common case)
3  for (int ii = 0; i < A.length; ii++) {
4    final int i = ii; // i is a final variable
5    async { A[i] = B[i] + C[i]; } // value of i is copied on entry to
6  }
7
8  // Example 2: execute iterations of a while loop in parallel
9  pp = first;
10 while ( pp != null ) {
11   T p = pp; // p is an effectively final variable
12   async { p.x = p.y + p.z; } // value of p is copied on entry to async
13   pp = pp.next;
14 }
15
16 // Example 3: Example 2 rewritten as a recursive method
17 static void process(T p) { // parameter p is an effectively final variable
18   if ( p != null ) {
19     async { p.x = p.y + p.z; } // value of p is copied on entry to async
20     process(p.next);
21   }
22 }
23
24 // Example 4: execute method calls in parallel
25 async { left_s = quickSort(left); }
26 async { right_s = quickSort(right); }
```

Listing 1: Example usages of async

All algorithm and programming examples in the module handouts should be treated as "pseudocode", since they are written for human readability with notations that are more abstract than the actual APIs that you will use for programming projects in COMP 322.

In Example 1 in Listing 1, the `for` loop sequentially increments index variable `i`, but all instances of the loop body can logically execute in parallel because of the `async` statement. The pattern of parallelizing counted for-loops in Example 1 occurs so commonly in practice that many parallel languages include a specialized construct for this case, that may be given a name such as `foreach`, `forall` or `forasync`.

In Example 2 in Listing 1, the `async` is used to parallelize computations in the body of a pointer-chasing `while` loop. Though the sequence of `p = p.next` statements is executed sequentially in the parent task, all dynamic instances of the remainder of the loop body can logically execute in parallel with each other.

Example 3 in Listing 1 shows the computation in Example 2 rewritten as a static void recursive method. You should first convince yourself that the computations in Examples 2 and 3 perform the same operations by omitting the `async` keyword in each case, and comparing the resulting sequential versions.

Example 4 shows the use of `async` to execute two method calls as parallel tasks (as was done in the two-way parallel sum algorithm).

As these examples show, a parallel program can create an unbounded number of tasks at runtime. The *parallel runtime system* is responsible for scheduling these tasks on a fixed number of processors. It does so by creating a fixed number of *worker threads* as shown in Figure 3, typically one worker per processor core. Worker threads are allocated by the Operating System (OS). By creating one thread per core, we limit the role of the OS in task scheduling to that of binding threads to cores at the start of program execution, and let the parallel runtime system take over from that point onwards. These workers repeatedly pull work

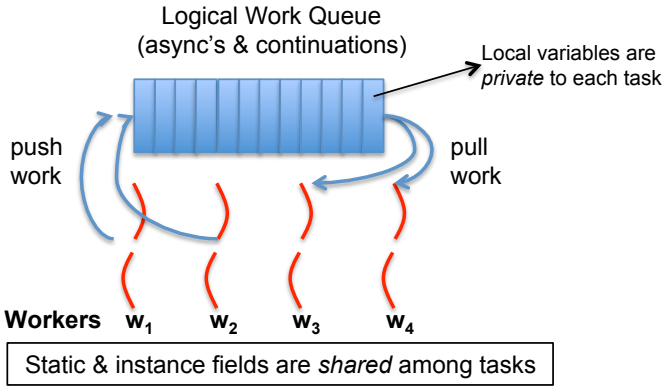

Figure 3: Scheduling an parallel program on a fixed number of workers. (Figure adapted from [6].)

```
1  // Rule 1: a child async may read the value of any outer final local var
2  final int i1 = 1;
3  async { ... = i1; /* i1=1 */ }
4
5  // Rule 2: a child async may also read any "effectively final" outer local var
6  int i2 = 2; // i2=2 is  copied on entry into the async like a method param
7  async { ... = i2; /* i2=2*/}
8  // i2 cannot be modified again, if it is "effectively final"
9
10 // Rule 3: a child async is not permitted to modify an outer local var
11 int i3;
12 async { i3 = ...; /* ERROR */}
```

Listing 2: Rules for accessing local variables across async's

from a shared work queue when they are idle, and push work on the queue when they generate more work. The work queue entries can include *async's* and *continuations*. An async is the creation of a new task, such as T_1 in Figure 2. A continuation[3] represents a potential suspension point for a task, which (as shown in in Figure 2) can include the point after an async creation as well as the point following the end of a finish scope. Continuations are also referred to as *task-switching* points, because they are program points at which a worker may switch execution between different tasks. A key motivation for this separation between tasks and threads is that it would be prohibitively expensive to create a new OS-level worker thread for each `async` task that is created in the program.

An important point to note in Figure 3 is that local variables are *private* to each task, whereas static and instance fields are *shared* among tasks. This is similar to the rule for accessing local variables and static/instance fields within and across methods or lambda expressions in Java. Listing 2 summarizes the rules for accessing local variables across async boundaries. For convenience, as shown in Rules 1 and 2, a child async is allowed to access a local variable declared in an outer async or method by simply capturing the value of the local variable when the async task is created (analogous to capturing the values of local variables in parameters at the start of a method call or in the body of a lambda expression). Note that a child async is not permitted to modify a local variable declared in an outer scope (Rule 3). If needed, you can work around the Rule 3 constraint by replacing the local variable by a static or instance field, since fields can be shared among tasks.

[3]This use of "continuation" is related to, but different from, continuations in functional programming languages.

1.1.2 Finish notation for Task Termination

The next parallel programming construct that we will learn as a complement to *async* is called *finish*. In pseudocode notation, "**finish** ⟨*stmt*⟩" causes the parent task to execute ⟨*stmt*⟩, which includes the possible creation of async tasks, and then wait until all *async* tasks created within ⟨*stmt*⟩ have completed before the parent task can proceed to the statement following the *finish*. Async and finish statements may also be arbitrarily nested.

Thus, the finish statement in Figure 2 is used by task T_0 to ensure that child task T_1 has completed executing `STMT1` before T_0 executes `STMT3`. This may be necessary if `STMT3` in Figure 2 used a value computed by `STMT1`. If T_1 created a child async task, T_2 (a "grandchild" of T_0), T_0 will wait for both T_1 and T_2 to complete in the finish scope before executing `STMT3`.

The waiting at the end of a finish statement is also referred to as a *synchronization*. The nested structure of `finish` ensures that *no deadlock cycle* can be created between two tasks such that each is waiting on the other due to end-finish operations. (A deadlock cycle refers to a situation where two tasks can be blocked indefinitely because each is waiting for the other to complete some operation.) We also observe that each dynamic instance T_A of an `async` task has a unique dynamic Immediately Enclosing Finish (IEF) instance F of a `finish` statement during program execution, where F is the innermost finish containing T_A. Like `async`, `finish` is a powerful primitive because it can be wrapped around any statement thereby supporting modularity in parallel programming.

If you want to convert a sequential program into a parallel program, one approach is to insert `async` statements at points where the parallelism is desired, and then insert `finish` statements to ensure that the parallel version produces the same result as the sequential version. Listing 3 extends the first two code examples from Listing 1 to show the sequential version, an incorrect parallel version with only `async`'s inserted, and a correct parallel version with both `async`'s and `finish`'s inserted.

The source of errors in the incorrect parallel versions are *data races*, which are notoriously hard to debug. As you will learn later in the course, a *data race* occurs if two parallel computations access the same shared location in an "interfering" manner *i.e.,* such that at least one of the accesses is a write (so called because the effect of the accesses depends on the outcome of the "race" between them to determine which one completes first). Data races form a class of bugs that are specific to parallel programming.

`async` and `finish` statements also jointly define what statements can potentially be executed in parallel with each other. Consider the finish-async nesting structure shown in Figure 4. It reveals which pairs of statements can potentially execute in parallel with each other. For example, task $A2$ can potentially execute in parallel with tasks $A3$ and $A4$ since `async` $A2$ was launched before entering the finish $F2$, which is the Immediately Enclosing Finish for $A3$ and $A4$. However, Part 3 of Task $A0$ cannot execute in parallel with tasks $A3$ and $A4$ since it is performed after finish $F2$ is completed.

1.1.3 Array Sum with two-way parallelism

We can use *async* and *finish* to obtain a simple parallel program for computing an array sum as shown in Algorithm 2. The computation graph structure for Algorithm 2 is shown in Figure 5. Note that it differs from Figure 1 since there is no edge or sequence of edges connecting Tasks $T2$ and $T3$. This indicates that tasks $T2$ and $T3$ can execute in parallel with each other; for example, if your computer has two processor cores, $T2$ and $T3$ can be executed on two different processors at the same time. We will see much richer examples of parallel programs using *async*, *finish* and other constructs during the course.

1.2 Computation Graphs

A *Computation Graph* (CG) is a formal structure that captures the meaning of a parallel program's execution. When you learned sequential programming, you were taught that a program's execution could be understood as a *sequence* of operations that occur in a well-defined *total order*, such as the left-to-right evaluation order for expressions. Since operations in a parallel program do not occur in a fixed order, some other abstraction is needed to understand the execution of parallel programs. Computation Graphs address this need by focusing on the extensions required to model parallelism as a *partial order*. Specifically, a Computation

```
1  // Example 1: Sequential version
2  for (int i = 0; i < A.length; i++) A[i] = B[i] + C[i];
3  System.out.println(A[0]);
4
5  // Example 1: Incorrect parallel version
6  for (int ii = 0; i < A.length; ii++) {
7    final int i = ii; // i is a final variable
8    async { A[i] = B[i] + C[i]; } // value of i is copied on entry to
9  }
10 System.out.println(A[0]);
11
12 // Example 1: Correct parallel version
13 finish {
14   for (int ii = 0; i < A.length; ii++) {
15     final int i = ii; // i is a final variable
16     async { A[i] = B[i] + C[i]; } // value of i is copied on entry to
17   }
18 }
19 System.out.println(A[0]);
20
21 // Example 2: Sequential version
22 p = first;
23 while ( p != null ) {
24   p.x = p.y + p.z; p = p.next;
25 }
26 System.out.println(first.x);
27
28 // Example 2: Incorrect parallel version
29 pp = first;
30 while ( pp != null ) {
31   T p = pp; // p is an effectively final variable
32   async { p.x = p.y + p.z; } // value of p is copied on entry to async
33   pp = pp.next;
34 }
35 System.out.println(first.x);
36
37 // Example 2: Correct parallel version
38 pp = first;
39 finish while ( pp != null ) {
40   T p = pp; // p is an effectively final variable
41   async { p.x = p.y + p.z; } // value of p is copied on entry to async
42   pp = pp.next;
43 }
44 System.out.println(first.x);
```

Listing 3: Incorrect and correct parallelization with async and finish

```
1  finish { // F1
2    // Part 1 of Task A0
3    async {A1; async A2;}
4    finish { // F2
5      // Part 2 of Task A0
6      async A3;
7      async A4;
8    }
9    // Part 3 of Task A0
10 }
```

Listing 4: Example usage of async and finish

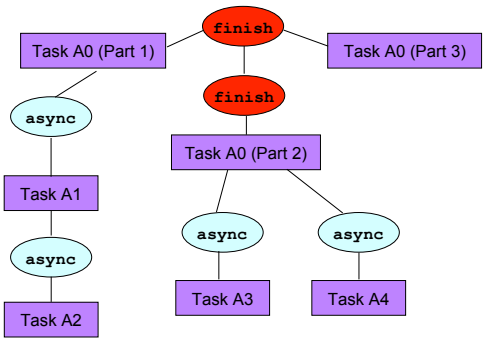

Figure 4: Finish-async nesting structure for code fragment in Listing 4

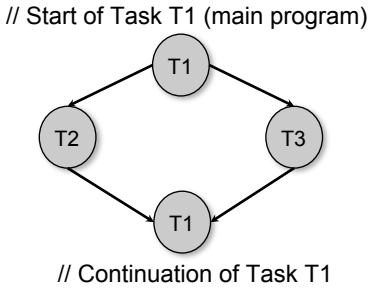

Figure 5: Computation graph for code example in Algorithm 5 (Two-way Parallel ArraySum)

Algorithm 2: Two-way Parallel ArraySum

Input: Array of numbers, X.
Output: sum = sum of elements in array X.

```
// Start of Task T1 (main program)
sum1 ← 0; sum2 ← 0;
// Compute sum1 (lower half) and sum2 (upper half) in parallel.
finish{
    async{
        // Task T2
        for i ← 0 to X.length/2 − 1 do
            sum1 ← sum1 + X[i];
    };
    async{
        // Task T3
        for i ← X.length/2 to X.length − 1 do
            sum2 ← sum2 + X[i];
    };
};
// Task T1 waits for Tasks T2 and T3 to complete
// Continuation of Task T1
sum ← sum1 + sum2;
return sum;
```

Graph consists of:

- A set of *nodes*, where each node represents a *step* consisting of an arbitrary sequential computation. For programs with `async` and `finish` constructs, a task's execution can be divided into steps by using *continuations* to define the boundary points. Recall from Section 1.1.1 that a continuation point in a task is the point after an async creation or a point following the end of a finish scope. It is acceptable to introduce finer-grained steps in the CG if so desired *i.e.,* to split a step into smaller steps. The key constraint is that a step should not contain any parallelism or synchronization *i.e.,* a continuation point should not be internal to a step.
- A set of *directed edges* that represent ordering constraints among steps. For `async`–`finish` programs, it is useful to partition the edges into three cases [6]:
    1. *Continue* edges that capture sequencing of steps within a task — all steps within the same task are connected by a chain of *continue* edges.
    2. *Spawn* edges that connect parent tasks to child `async` tasks. When an `async` is created, a *spawn* edge is inserted between the step that ends with the `async` in the parent task and the step that starts the `async` body in the new child task.
    3. *Join* edges that connect descendant tasks to their Immediately Enclosing Finish (IEF) operations. When an `async` terminates, a *join* edge is inserted from the last step in the `async` to the step in the ancestor task that follows the IEF operation.

Consider the example program shown in Listing 5 and its Computation Graph shown in Figure 6. There are 6 tasks in the CG, T_1 to T_6. This example uses finer-grained steps than needed, since some steps (*e.g.,* $v1$ and $v2$) could have have been combined into a single step. In general, the CG grows as a program executes and a complete CG is only available when the entire program has terminated. The three classes of edges (continue, spawn, join) are shown in Figure 6. Even though they arise from different constructs, they all have the same effect *viz.,* to enforce an ordering among the steps as dictated by the program.

In any execution of the CG on a parallel machine, a basic rule that must be obeyed is that a successor node B of an edge (A, B) can only execute after its predecessor node A has completed. This relationship between nodes A and B is referred to as a *dependence* because the execution of node B depends on the execution of node A having completed. In general, node Y depends on node X if there is a path of directed edges from X to Y in the CG. Therefore, dependence is a *transitive* relation: if B depends on A and C depends on B, then C must depend on A. The CG can be used to determine if two nodes may execute in parallel with each other. For example, an examination of Figure 6 shows that all of nodes $v3 \ldots v15$ can potentially execute in parallel with node $v16$ because there is no directed path in the CG from $v16$ to any node in $v3 \ldots v15$ or vice versa.

It is also important to observe that the CG in Figure 6 is *acyclic i.e.,* it is not possible to start at a node and trace a cycle by following directed edges that leads back to the same node. An important property of CGs is that all CGs are *directed acyclic graphs*, also referred to as *dags*. As a result, the terms "computation graph" and "computation dag" are often used interchangeably.

1.3 Ideal Parallelism

In addition to providing the dependence structure of a parallel program, Computation Graphs can also be used to reason about the *ideal parallelism* of a parallel program as follows:

- Assume that the execution time, *time(N)*, is known for each node N in the CG. Since N represents an uninterrupted sequential computation, it is assumed that *time(N)* does not depend on how the CG is scheduled on a parallel machine. (This is an idealized assumption because the execution time of many operations, especially memory accesses, can depend on when and where the operations are performed in a real computer system.)
- Define *WORK(G)* to be the sum of the execution times of the nodes in CG G,

$$WORK(G) = \sum_{\text{node } N \text{ in } G} time(N)$$

  Thus, *WORK(G)* represents the total amount of work to be done in CG G.
- Define *CPL(G)* to be the length of the longest path in G, when adding up the execution times of all nodes in the path. There may be more than one path with this same length. All such paths are referred to as *critical paths*, so *CPL* stands for *critical path length*.

Consider again the CG, G, in Figure 6. For simplicity, we assume that all nodes have the same execution time, *time(N)* = 1. It has a total of 23 nodes, so *WORK(G)* = 23. In addition the longest path consists of 17 nodes as follows, so *CPL(G)* = 17:

$v1 \rightarrow v2 \rightarrow v3 \rightarrow v6 \rightarrow v7 \rightarrow v8 \rightarrow v10 \rightarrow v11 \rightarrow v12 \rightarrow v13 \rightarrow v14 \rightarrow v18 \rightarrow v19 \rightarrow v20 \rightarrow v21 \rightarrow v22 \rightarrow v23$

Given the above definitions of *WORK* and *CPL*, we can define the *ideal parallelism* of Computation Graph G as the ratio, *WORK(G)/CPL(G)*. The ideal parallelism can be viewed as the maximum performance improvement factor due to parallelism that can be obtained for computation graph G, even if we ideally had an unbounded number of processors. It is important to note that ideal parallelism is independent of the number of processors that the program executes on, and only depends on the computation graph

1.3.1 Abstract Performance Metrics

While Computation Graphs provide a useful abstraction for reasoning about performance, it is not practical to build Computation Graphs by hand for large programs. The Habanero-Java (HJ) library used in the course includes the following utilities to help programmers reason about the CGs for their programs:

- *Insertion of calls to* `doWork()`. The programmer can insert a call of the form `perf.doWork(N)` anywhere in a step to indicate execution of N application-specific abstract operations *e.g.,* floating-point

```
1  // Task T1
2  v1; v2;
3  finish {
4    async {
5      // Task T2
6      v3;
7      finish {
8        async { v4; v5; } // Task T3
9        v6;
10       async { v7; v8; } // Task T4
11       v9;
12     } // finish
13     v10; v11;
14     async { v12; v13; v14; } // Task T5
15     v15;
16   }
17   v16; v17;
18 } // finish
19 v18; v19;
20 finish {
21   async { v20; v21; v22; }
22 }
23 v23;
```

Listing 5: Example program

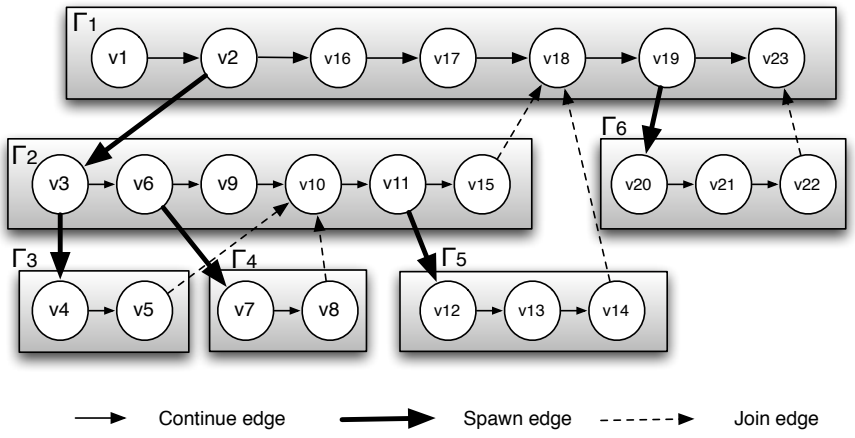

Figure 6: Computation Graph G for example program in Listing 5

operations, comparison operations, stencil operations, or any other data structure operations. Multiple calls to `perf.doWork()` are permitted within the same step. They have the effect of adding to the abstract execution time of that step. The main advantage of using abstract execution times is that the performance metrics will be the same regardless of which physical machine the HJ program is executed on. The main disadvantage is that the abstraction may not be representative of actual performance on a given machine.

- *Printout of abstract metrics.* If an HJlib program is executed with a specified option, abstract metrics are printed at the end of program execution that capture the total number of operations executed ($WORK$) and the critical path length (CPL) of the CG generated by the program execution. The ratio, $WORK/CPL$ is also printed as a measure of *ideal parallelism*.
- *Visualization of computation graph.* A tool called HJ-viz is also provided that enables you to see an image of the computation graph of a program executed with abstract performance metrics.

1.4 Multiprocessor Scheduling

Now, let us discuss the execution of CG G on an idealized parallel machine with P processors. It is idealized because all processors are assumed to be identical, and the execution time of a node is assumed to be independent of which processor it executes on. Consider all legal schedules of G on P processors. A *legal schedule* is one that obeys the dependence constraints in the CG, such that for every edge (A, B) the scheduled guarantees that B is only scheduled after A completes. Let t_P denote the execution time of a legal schedule. While different schedules may have different execution times, they must all satisfy the following two *lower bounds*:

1. *Capacity bound:* $t_p \geq WORK(G)/P$. It is not possible for a schedule to complete in time less than $WORK(G)/P$ because that's how long it would take if all the work was perfectly divided among P processors.
2. *Critical path bound:* $t_p \geq CPL(G)$. It is not possible for a schedule to complete in time less than $CPL(G)$ because any legal schedule must obey the chain of dependences that form a critical path. Note that the critical path bound does not depend on P.

Putting these two *lower bounds* together, we can conclude that $t_p \geq \max(WORK(G)/P, CPL(G))$. Thus, if the observed parallel execution time t_P is larger than expected, you can investigate the problem by determining if the capacity bound or the critical path bound is limiting its performance.

It is also useful to reason about the *upper bounds* for t_P. To do so, we have to make some assumption about the "reasonableness" of the scheduler. For example, an unreasonable scheduler may choose to keep processors idle for an unbounded number of time slots (perhaps motivated by locality considerations), thereby making t_P arbitrarily large. The assumption made in the following analysis is that all schedulers under consideration are "greedy" i.e., they will never keep a processor idle when there's a node that is available for execution.

We can now state the following properties for t_P, when obtained by greedy schedulers:

- $t_1 = WORK(\mathrm{G})$. Any greedy scheduler executing on 1 processor will simply execute all nodes in the CG in some order, thereby ensuring that the 1-processor execution time equals the total work in the CG.
- $t_\infty = CPL(\mathrm{G})$. Any greedy scheduler executing with an unbounded (infinite) number of processors must complete its execution with time = $CPL(\mathrm{G})$, since all nodes can be scheduled as early as possible.
- $t_P \leq t_1/P + t_\infty = WORK(G)/P + CPL(G)$. This is a classic result due to Graham [5]. An informal sketch of the proof is as follows. At any given time in the schedule, we can declare the time slot to be *complete* if all P processors are busy at that time and *incomplete* otherwise. The number of complete time slots must add up to at most t_1/P since each such time slot performs P units of work.

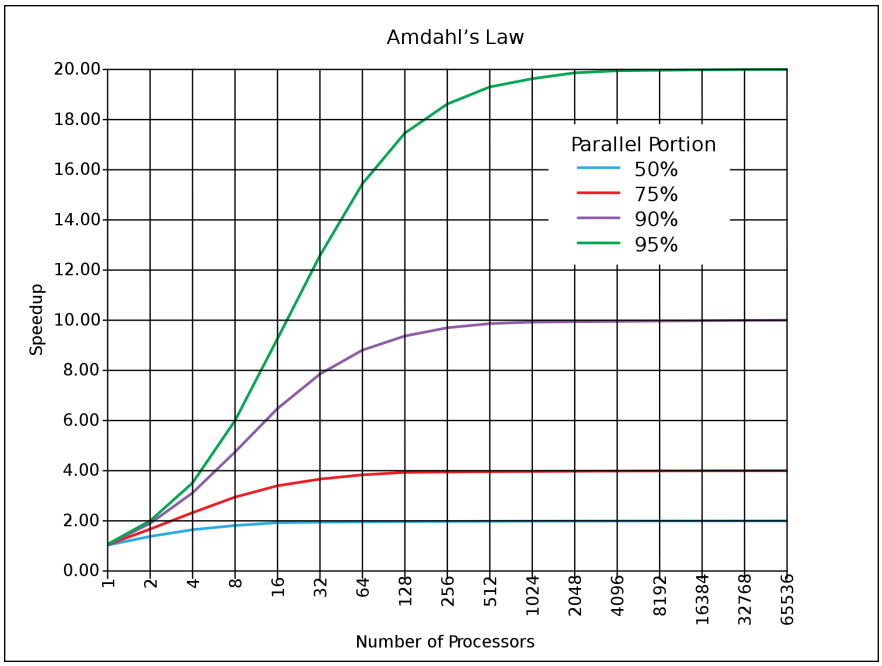

Figure 7: Illustration of Amdahl's Law (source: http://en.wikipedia.org/wiki/Amdahl's_law)

In addition, the number of incomplete time slots must add up to at most t_∞ since each such time slot must advance 1 time unit on a critical path. Putting them together results in the *upper bound* shown above. Combining it with the lower bound, you can see that:

$$\max(WORK(G)/P, CPL(G)) \leq t_P \leq WORK(G)/P + CPL(G)$$

It is interesting to compare the lower and upper bounds above. You can observe that they contain the *max* and *sum* of the same two terms, $WORK(G)/P$ and $CPL(G)$. Since $x + y \leq 2\max(x, y)$, the lower and upper bounds vary by at most a factor of $2\times$. Further, if one term dominates the other *e.g.,* $x \gg y$, then the two bounds will be very close to each other.

1.5 Parallel Speedup and Amdahl's Law

Given definitions for t_1 and t_P, the speedup for a given schedule of a computation graph on P processors is defined as $Speedup(P) = t_1/t_P$. *Speedup(P)* is the 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, $1 \leq Speedup(P) \leq P$. The term *linear speedup* is used for a program when $Speedup(P) = k \times P$ as P varies, for some constant k, $0 < k < 1$.

We can now summarize a simple observation made by Gene Amdahl in 1967 [1]: 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 $Speedup(P) \leq 1/q$. As in the Computation Graph model studied earlier, this observation assumes that all processors are *uniform i.e.,* they all execute at the same speed.

This observation follows directly from a lower bound on parallel execution time that you are familiar with, namely $t_P \geq CPL(G)$, where t_P is the execution time of computation graph G on P processors and *CPL* is the *critical path length* of graph G. If fraction q of $WORK(G)$ is sequential, it must be the case that $CPL(G) \geq q \times WORK(G)$. Therefore, $Speedup(P) = t_1/t_P$ must be $\leq WORK(G)/(q \times WORK(G)) = 1/q$ since $t_1 = WORK(G)$ for greedy schedulers.

The consequence of Amdahl's Law is illustrated in Figure 7. The x-axis shows the number of processors increasing in powers of 2 on a log scale, and the y-axis represents speedup obtained for different values of q. Specifically, each curve represents a different value for the *parallel portion*, $(1 - q)$, assuming that all the non-sequential work can be perfectly parallelized. Even when the parallel portion is as high as 95%, the maximum speedup we can obtain is 20× since the sequential portion is 5%. The ideal case of $q = 0$ and a parallel portion of 100% is not shown in the figure, but would correspond to the $y = x$ line which would appear to be an exponential curve since the x-axis is plotted on a log scale.

Amdahl's Law reminds us to watch out for sequential bottlenecks both when designing parallel algorithms and when implementing programs on real machines. While it may paint a bleak picture of the utility of adding more processors to a parallel computing, it has also been observed that increasing the data size for a parallel program can reduce the sequential portion [7] thereby making it profitable to utilize more processors. The ability to increase speedup by increasing the number of processors for a fixed input size (fixed $WORK$) is referred to as *strong scaling*, and the ability to increase speedup by increasing the input size (increasing $WORK$) is referred to as *weak scaling*.

2 Functional Parallelism and Determinism

2.1 Future Tasks and Functional Parallelism

2.1.1 Tasks with Return Values

The `async` construct introduced in previous sections provided the ability to execute any statement as a parallel task, and the `finish` construct provided a mechanism to await termination of all tasks created within its scope. `async` task creation leads to a natural *parent-child* relation among tasks, *e.g.,* if task T_A creates async task T_B, then T_A is the parent of T_B and T_B is the child of T_A. Thus, an *ancestor* task, T_A, can use a `finish` statement to ensure that it is safe to read values computed by *all descendant tasks*, T_D enclosed in the scope of the `finish`. These values are communicated from T_D to T_A via shared variables, which (in the case of Java tasks) must be an instance field, static field, or array element.

However, there are many cases where it is desirable for a task to explicitly wait for the return value from a specific single task, rather than all descendant tasks in a `finish` scope. To do so, it is necessary to extend the regular `async` construct with return values, and to create a container (proxy) for the return value which is done using *future objects* as follows:

- A variable of type `future<T>`[4] is a reference to a *future* object *i.e.,* a container for a return value of type `T` from an `async` task.
- There are exactly two operations that can be performed on a variable, `V1`, of type `future<T1>`, assuming that type `T2` is a subtype of, or the same as, type `T1`:
  1. *Assignment* — variable `V1` can be assigned a reference to an `async` with return value type `T2` as described below, or `V1` can be assigned the value of a variable `V2` with type `future<T2>`.
  2. *Blocking read* — the operation, `V1.get()`, waits until the `async` referred to by `V1` has completed, and then propagates the return value of the `async` to the caller as a value of type `T1`. This semantics also avoids the possibility of a race condition on the return value.
- An `async` with a return value is called a *future task*, and can be defined by introducing two extensions to regular `async`'s as follows:
  1. The body of the async must start with a type declaration, `async<T1>`, in which the type of the async's return value, `T1`, immediately follows the `async` keyword.
  2. The body itself must consist of a compound statement enclosed in { } braces, dynamically terminating with a `return` statement. It is important to note that the purpose of this `return` statement is to communicate the return value of the enclosing `async` and not the enclosing method.

Listing 6 revisits the two-way parallel array sum example discussed earlier, but using *future tasks* instead of regular `async`'s. There are two variables of type `future<int>` in this example, `sum1` and `sum2` Each future task can potentially execute in parallel with it parent, just like regular `async`'s. However, unlike regular `async`'s, there is no `finish` construct needed for this example since the parent task $T1$, performs `sum1.get()` to wait for future task $T2$ and `sum2.get()` to wait for future task $T3$.

In addition to waiting for completion, the `get()` operations are also used to access the return values of the future tasks. This is an elegant capability because it obviates the need for shared fields or shared arrays, and avoids the possibility of race conditions on those shared variables. Notice the three declarations for variables `sum` in lines 4, 9, and 14. Each occurrence of `sum` is local to a task, and there's no possibility of race conditions on these local variables or the return values from the future tasks. These properties have historically made future tasks well suited to express parallelism in functional languages [8].

[4] "`future`" is a pseudocode keyword, and will need be replaced by the appropriate data type in real code.

```
1  // Parent Task T1 (main program)
2  // Compute sum1 (lower half) and sum2 (upper half) in parallel
3  future<int> sum1 = async<int> { // Future Task T2
4    int sum = 0;
5    for(int i=0 ; i < X.length/2 ; i++) sum += X[i];
6    return sum;
7  }; //NOTE: semicolon needed to terminate assignment to sum1
8  future<int> sum2 = async<int> { // Future Task T3
9    int sum = 0;
10   for(int i=X.length/2 ; i < X.length ; i++) sum += X[i];
11   return sum;
12 }; //NOTE: semicolon needed to terminate assignment to sum2
13 //Task T1 waits for Tasks T2 and T3 to complete
14 int sum = sum1.get() + sum2.get();
```

Listing 6: Two-way Parallel ArraySum using Future Tasks

2.1.2 Computation Graph Extensions for Future Tasks

Future tasks can be accommodated very naturally in the Computation Graph (CG) abstraction introduced in Section 1.2. The main CG extensions required to accommodate the `get()` operations are as follows:

- A `get()` operation is a new kind of *continuation* operation, since it can involve a blocking operation. Thus, `get()` operations can only occur on the boundaries of steps. To fully realize this constraint, it may be necessary to split a statement containing one or more `get()` operations into multiple sub-statements such that a `get()` occurs in a sub-statement by itself.
- A *spawn* edge connects the parent task to a child future task, just as with regular `async`'s.
- When a future task, T_F, terminates, a *join* edge is inserted from the last step in T_F to the step in the ancestor task that follows its Immediately Enclosing Finish (IEF) operation, as with regular `async`'s. In addition, a *join edge* is also inserted from T_F's last step to every step that follows a `get()` operation on the future task. Note that new `get()` operations may be encountered even after T_F has terminated.

To compute the computation graph for the example in Listing 6, we will need to split the statement in line 14 into the following sub-statements:

```
14a    int temp1 = sum1.get();
14b    int temp2 = sum2.get();
14c    int sum = temp1 + temp2;
```

The resulting CG is shown in Figure 8. Note that the end of each step in a future task has two outgoing *join edges* in this example, one to the `get()` operation and one to the implicit *end-finish* operation in the main program.

2.1.3 Why should future references be effectively final?

In this section, we elaborate on an important programming principle for futures, viz., all variables containing references to future objects should be *effectively final* (either declared final or participating in a single assignment), which means that the variable cannot be modified after initialization. To motivate this rule, consider the buggy program example in Listing 7. *WARNING: this is an example of bad parallel programming practice that you should not attempt!*

This program declares two static non-final future reference fields, `f1` and `f2`, in lines 1 and 2 and initializes them to `null`. The `main()` programs then creates two future tasks, $T1$ and $T2$, in lines 5 and 6 and assigns

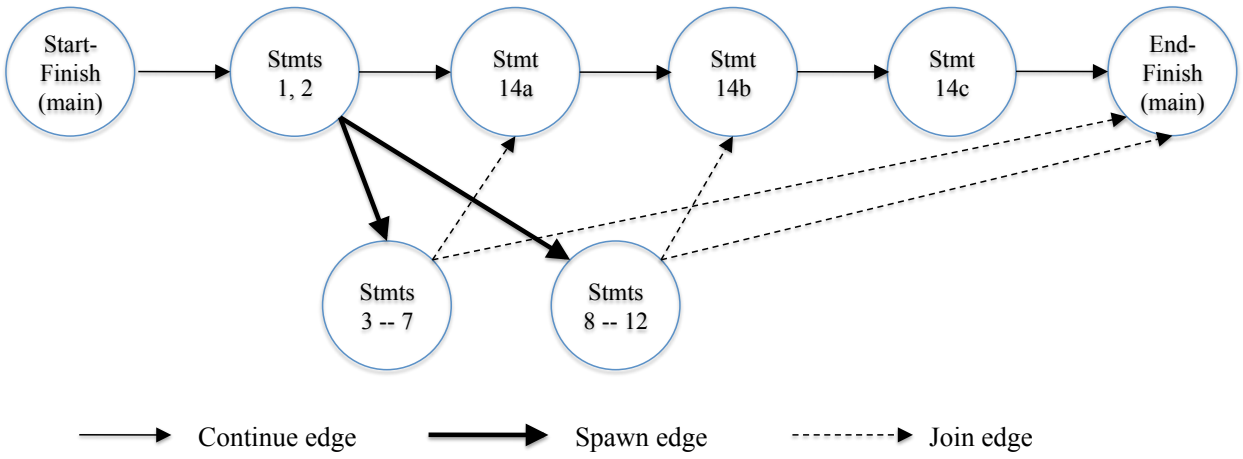

Figure 8: Computation Graph G for example parallel program in code:TwoParArraySumFuture, with statement 14 split into statements 14a, 14b, 14c

them to `f1` and `f2` respectively. Task $T1$ uses a "spin loop" in line 10 to wait for `f2` to become non-null, and task $T2$ does the same in line 15 to wait for `f1` to become non-null. After exiting the spin loop, each task performs a `get()` operation on the other thereby attempting to create a *deadlock cycle* in the computation graph. Fortunately, the rule that all variables containing references to future objects should be effectively final can avoid this situation.

2.1.4 Future Tasks with a void return type

A key distinction made thus far between future tasks and regular `async`'s is that future tasks have return values but regular `async`'s do not. However, there is a construct that represents a hybrid of these two task variants, namely a future task, T_V, with a `void` return type. This is analogous to Java methods with `void` return types. In this case, a `get()` operation performed on T_V has the effect of waiting for T_V to complete, but no return value is communicated from T_V.

Figure 9 shows Computation Graph $G3$ that cannot be generated using only `async` and `finish` constructs, and Listing 8 shows the code that can be used to generate $G3$ using future tasks. This code uses futures with a `void` return type, and provides a systematic way of converting any CG into a task-parallel program using futures.

2.2 Memoization

The basic idea of memoization is to remember results of function calls $f(x)$ as follows:

1. Create a data structure that stores the set $\{(x_1, y_1 = f(x_1)), (x_2, y_2 = f(x_2)), \ldots\}$ for each call $f(x_i)$ that returns y_i.
2. Look up data structure when processing calls of the form $f(x')$ when x' equals one of the x_i inputs for which $f(x_i)$ has already been computed.

The memoization pattern lends itself easily to parallelization using futures by modifying the memoized data structure to store $\{(x_1, y_1 = \mathit{future}(f(x_1))), (x_2, y_2 = \mathit{future}(f(x_2))), \ldots\}$. The lookup operation can then be extended with a *get()* operation on the future value if a future has already been created for the result of a given input.

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

Listing 7: Buggy Use of Future Tasks due to missing final declarations

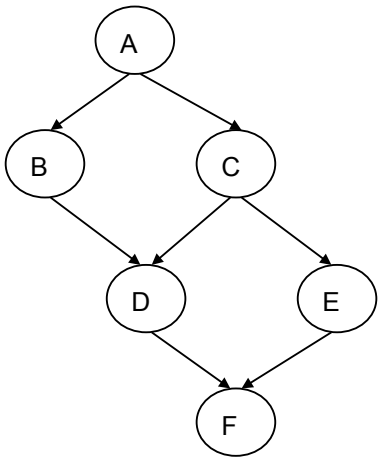

Figure 9: Computation Graph $G3$

```
1  // NOTE: return statement is optional when return type is void
2  future<void> A = async<void> { . . . ; return;}
3  future<void> B = async<void> { A.get(); . . . ; return;}
4  future<void> C = async<void> { A.get(); . . . ; return;}
5  future<void> D = async<void> { B.get(); C.get(); . . . ; return;}
6  future<void> E = async<void> { C.get(); . . . ; return;}
7  future<void> F = async<void> { D.get(); E.get(); . . . ; return;}
```

Listing 8: Task-parallel code with futures to generate Computation Graph $G3$ from Figure 9

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

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

Figure 10: Example of accumulator API

2.3 Finish Accumulators

In this section, we introduce the programming interface and semantics of *finish accumulators*. Finish accumulators support parallel reductions, which represent a common pattern for computing the aggregation of an associative and commutative operation, such as summation, across multiple pieces of data supplied by parallel tasks. There are two logical operations, *put*, to remit a datum and *get*, to retrieve the result from a well-defined synchronization (`end-finish`) point. Section 2.3.1 describes the details of these operations, and Section 2.3.2 describes how user-defined reductions are supported in finish accumulators.

2.3.1 Accumulator Constructs

Figure 10 shows an example of a finish-accumulator programming interface. The operations that a task, T_i, can perform on accumulator, *ac*, are defined as follows.

- **new:** When task T_i performs a "`ac = new accumulator(op, dataType);`" statement, it creates a new accumulator, *ac*, on which T_i is registered as the *owner task*. Here, *op* is the reduction operator that the accumulator will perform, and *dataType* is the type of the data upon which the accumulator operates. Currently supported predefined reduction operators include `SUM`, `PROD`, `MIN`, and `MAX`; `CUSTOM` is used to specify user-defined reductions.
- **put:** When task T_i performs an "*ac*`.put(`*datum*`);`" operation on accumulator *ac*, it sends *datum* to *ac* for the accumulation, and the accumulated value becomes available at a later end-finish point. The runtime system throws an exception if a `put()` operation is attempted by a task that is not the owner and does not belong to a `finish` scope that is associated with the accumulator. When a task performs multiple `put()` operations on the same accumulator, they are treated as separate contributions to the reduction.
- **get:** When task T_i performs an "*ac*`.get()`" operation on accumulator *ac* with predefined reduction operators, it obtains a `Number` object containing the accumulated result. Likewise "*ac*`.customGet()`" on *ac* with a `CUSTOM` operator returns a user-defined `T` object with the accumulated result. When no `put` is performed on the accumulator, `get` returns the identity element for the operator, *e.g.,* `0` for `SUM`, `1` for `PROD`, `MAX_VALUE`/`MIN_VALUE` for `MIN`/`MAX`, and user-defined identity for `CUSTOM`.
- **Summary of access rules:** The owner task of accumulator *ac* is allowed to perform `put`/`get` operations on *ac* and associate *ac* with any `finish` scope in the task. Non-owner tasks are allowed to

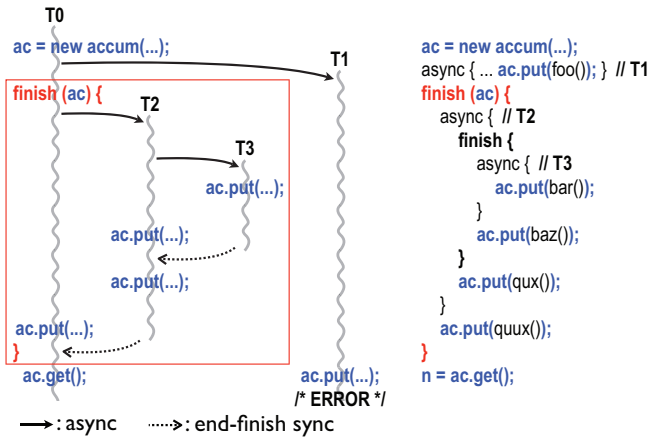

Figure 11: Finish accumulator example with three tasks that perform a correct reduction and one that throws an exception

> access *ac* only within `finish` scopes with which *ac* is associated. To ensure determinism, the accumulated result only becomes visible at the `end-finish` synchronization point of an associated `finish`; `get` operations within a `finish` scope return the same value as the result at the beginning of the `finish` scope. Note that `put` operations performed by the owner outside associated `finish` scopes are immediately reflected in any subsequent `get` operations since those results are deterministic.

In contrast to traditional reduction implementations, the `put()` and `get()` operations are separate, and reduction results are not visible until the end-finish point.

To associate a `finish` statement with multiple accumulators, T_{owner} can perform a special `finish` statement of the form, "`finish` $(ac_1, ac_2, \cdots, ac_n)\langle stmt \rangle$". Note that `finish` (ac) becomes a no-op if *ac* is already associated with an outer `finish` scope.

Figure 11 shows an example where four tasks T_0, T_1, T_2, and T_3 access a finish accumulator *ac*. As described earlier, the `put` operation by T_1 throws an exception due to nondeterminism since it is not the owner and was created outside the `finish` scope associated with accumulator *ac*. Note that the inner `finish` scope has no impact on the reduction of *ac* since *ac* is associated only with the outer `finish`. All `put` operations by T_0, T_2, and T_3 are reflected in *ac* at the `end-finish` synchronization of the outer `finish`, and the result is obtained by T_0's `get` operation.

2.3.2 User-defined Reductions

User-defined reductions are also supported in finish accumulators, and its usage consists of these three steps:
1) specify `CUSTOM` and `reducible.class` as the accumulator's operator and type,
2) define a class that implements the `reducible` interface,
3) pass the implementing class to the accumulator as a type parameter.

Figure 12 shows an example of a user-defined reduction. Class `Coord` contains two double fields, `x` and `y`, and the goal of the reduction is to find the furthest point from the origin among all the points submitted to the accumulator. The `reduce` method computes the distance of a given point from the origin, and updates `x` and `y` if `arg` has a further distance than the current point in the accumulator.

2.4 Map Reduce

Data structures based on key-value pairs are used by a wide range of data analysis algorithms, including web search and statistical analyses. In Java, these data structures are often implemented as instances of the Map interface. An important constraint imposed on sets of key-value pairs is that no key occurs more than once, thereby ensuring that each key can map to at most one value. Thus, a mathematical abstraction of a

```
 1: void foo() {
 2:   accum<Coord> ac = new accum<Coord>(Operation.CUSTOM,
 3:                                      reducible.class);
 4:   finish(ac) {
 5:     forasync (point [j] : [1:n]) {
 6:       while(check(j)) {
 7:         ac.put(getCoordinate(j));
 8:   } } }
 9:   Coord c = ac.customGet();
10:   System.out.println("Furthest: " + c.x + ", " + c.y);
11: }
12:
13: class Coord implements reducible<Coord> {
14:   public double x, y;
15:   public Coord(double x0, double y0) {
16:     x = x0;  y = y0;
17:   }
18:   public Coord identity(); {
19:     return new Coord(0.0, 0.0);
20:   }
21:   public void reduce(Coord arg) {
22:     if (sq(x) + sq(y) < sq(arg.x) + sq(arg.y)) {
23:       x = arg.x;  y = arg.y;
24:   } }
25:   private double sq(double v) { return v * v; }
26: }
```

Figure 12: User-defined reduction example

Map data structure is as a set of pairs, $S = \{(k_1, v_1), \ldots, (k_n, v_n)\}$, such that each (k_i, v_i) pair consists of a key, k_i, and a value, v_i and $k_i \neq k_j$ for any $i \neq j$.

Many data analysis algorithm can be specified as sequences of *map* and *reduce* operations on sets of key-value pairs. For a given key-value pair, (k_i, v_i), a map function f generates a sets of output key-value pairs, $f(k_i, v_i) = \{(k_1, v_1), \ldots, (k_m, v_m)\}$. The k_j keys can be different from the k_i key in the input of the map function. When applied to a set of key-value pairs, the map function results in the union of the output set generated from each input key-value pair as follows:

$$f(S) = \bigcup_{(k_i, v_i) \in S} f(k_i, v_i)$$

$f(S)$ is referred to as a set of *intermediate key-value pairs* because it will serve as an input for a reduce operation, g. Note that it is possible for $f(S)$ to contain multiple key-value pairs with the same key. The reduce operation groups together intermediate key-value pairs, $\{(k, v_j)\}$ with the sam key k, and generates a reduced key-value pair, (k, v), for each such k, using a reduce function g on all v'_j values with the same intermediate key k'. Therefore $g(f(S))$ is guaranteed to satisfy the unique-key property.

Listing 9 shows the pseudocode for one possible implementation of map-reduce operations using finish and async primitives. The basic idea is to complete all operations in the map phase before any operation in the reduce phase starts. Alternate implementations are possible that expose more parallelism.

As an example, Listing 10 shows how the *WordCount* problem can be solved using map and reduce operations on sets of key-value pairs. All map operations in step a) (line 4) can execute in parallel with only local data accesses, making the map step highly amenable to parallelization. Step b) (line 5) can involve a major reshuffle of data as all key-value pairs with the same key are grouped (gathered) together. Finally, step c) (line 6) performs a standard reduction algorithm for all values with the same key.

```
1  finish { // map phase
2    for each (ki,vi) pair in input set S
3      async compute f(ki,vi) and append output to f(S); // map operation
4  }
5  finish { // reduce phase
6    for each key k' in intermediate set f(S)
7      async { // reduce operation
8        temp = identity;
9        for each value v'' such that (k',v'') is in f(S) {
10         temp = g(temp, v'');
11       }
12       append (k',temp) to output set, g(f(S);
13     }
14 }
```

Listing 9: Pseudocode for one possible implementation of map-reduce operations using finish and async primitives

```
1  Input: set of words
2  Output: set of (word,count) pairs
3  Algorithm:
4  a) For each input word W, emit (W, 1) as a key-value pair (map step).
5  b) Group together all key-value pairs with the same key (intermediate
6  key-value pairs).
7  c) Perform a sum reduction on all values with the same key (reduce step).
```

Listing 10: Computing *Wordcount* using map and reduce operations on sets of key-value pairs

```
1 // Sequential version
2 for ( p = first; p != null; p = p.next) p.x = p.y + p.z;
3 for ( p = first; p != null; p = p.next) sum += p.x;
4
5 // Incorrect parallel version
6 for ( p = first; p != null; p = p.next)
7   async p.x = p.y + p.z;
8 for ( p = first; p != null; p = p.next)
9   sum += p.x;
```

Listing 11: Sequential and incorrect parallel versions of example program

2.5 Data Races

2.5.1 What are Data Races?

The fundamental primitives for task creation (async) and termination (finish) that you have learned thus far are very powerful, and can be used to create a wide range of parallel programs. You will now learn about a pernicious source of errors in parallel programs called *data races*, and how to avoid them.

Consider the example program shown in Listing 11. The parallel version contains an error because the writes to instances of `p.x` in line 7 can potentially execute in parallel with the reads of instances of `p.x` in line 9. This can be confirmed by building a computation graph for the program and observing that there is no chain of dependence edges from step instances of line 7 to step instances of line 9. As a result, it is unclear whether a read in line 9 will receive an older value of `p.x` or the value written in line 7. This kind of situation, where the outcome depends on the relative completion times of two events, is called a *race condition*. When the race condition applies to read and write accesses on a shared location, it is called a *data race*. A shared location must be a static field, instance field or array element, since it is not possible for interfering accesses to occur in parallel on a local variable in a method.

Data races are a challenging source of errors in parallel programming, since it is usually impossible to guarantee that all possible orderings of the accesses to a location will be encountered during program testing. Regardless of how many tests you write, so long as there is one ordering that yields the correct answer it is always possible that the correct ordering in encountered when testing your program and an incorrect ordering is encountered when the program executes in a production setting. For example, while testing the program, it is possible that the task scheduler executes all the `async` tasks in line 7 of Listing 11 before executing the continuation starting at line 8. In this case, the program will appear to be correct during test, but will have a latent error that could be manifest at any arbitrary time in the future.

Formally, *a data race occurs on location L in a program execution with computation graph CG* if there exist steps S_1 and S_2 in CG such that:

1. S_1 does not depend on S_2 and S_2 does not depend on S_1 *i.e.,* there is no path of dependence edges from S_1 to S_2 or from S_2 to S_1 in CG, and
2. both S_1 and S_2 read or write L, and at least one of the accesses is a write.

Programs that are guaranteed to never exhibit a data race are said to to be *data-race-free*. It is also common to refer to programs that may exhibit data races as "*racy*".

There are a number of interesting observations that follow from the above definition of a data race:

1. *Immutability property: there cannot be a data race on shared immutable data.* Recall that shared data in a parallel Habanero-Java program consists of static fields, instance fields, and array elements. An immutable location, L_i, is one that is only written during initialization, and can only be read after

```
1 finish {
2   String s1 = "XYZ";
3   async { String s2 = s1.toLowerCase(); ... }
4   System.out.println(s1);
5 }
```

Listing 12: Example of immutable string operations in a parallel program

initialization. In this case, there cannot be a data race on L_i because there will only be one step that writes to L_i in CG, and all steps that read from L must follow the write. This property applies by definition to static and non-static *final* fields. It also applies to instances of any *immutable class e.g.,* `java.lang.String`.

2. *Single-task ownership property: there cannot be a data race on a location that is only read or written by a single task.* Let us say that step S_i in CG *owns* location L if it performs a read or write access on L. If step S_i belongs to Task T_j, we can also say that Task T_j owns L when executing S_i. (Later in the course, it will be useful to distinguish between *read ownership* and *write ownership.*) Consider a location L that is only owned by steps that belong to the same task, T_j. Since all steps in Task T_j must be connected by *continue* edges in CG, all reads and writes to L must be ordered by the dependences in CG. Therefore, no data race is possible on location L.

3. *Ownership-transfer property: there cannot be a data race on a location if all steps that read or write it are totally ordered in CG.* The single-task-ownership property can be generalized to the case when all steps that read or write a location L are totally ordered by dependences in CG, even if the steps belong to different tasks *i.e.,* for any two steps S_i and S_j that read or write L, it must be the case that there is a path of dependence edges from S_i to S_j or from S_j to S_i. In this case, no data race is possible on location L. We can think of the ownership of L being "transferred" from one step to another, even across task boundaries, as execution follows the path of dependence edges.

4. *Local-variable ownership property: there cannot be a data race on a local variable.* If L is a local variable, it can only be written by the task in which it is declared (L's owner). Though it may be read by a descendant task, the "copy-in" semantics for local variables (Rule 2 in Listing 2 of Section 1.1.1) ensures that the value of the local variable is copied on `async` creation thus ensuring that there is no race condition between the read access in the descendant task and the write access in L's owner.

2.5.2 Avoiding Data Races

The four observations in Section 2.5.1 directly lead to the identification of programming tips and best practices to avoid data races. There is considerable effort under way right now in the research community to provide programming language support for these best practices, but until they enter the mainstream it is your responsibility as a programmer to follow these tips on avoiding data races:

1. *Immutability tip:* Use immutable objects and arrays as far as possible. Sometimes this may require making copies of objects and arrays instead of just modifying a single field or array element. Depending on the algorithm used, the overhead of copying could be acceptable or prohibitive. For example, copying has a small constant factor impact in the Parallel Quicksort algorithm.

   Consider the example program in Listing 12. The parent task initializes `s1` to the string, `"XYZ"` in line 2, creates a child task in line 3, and prints out `s1` in line 4. Even though the child task invokes the `toLowerCase()` method on `s1` in line 3, there is no data race between line 3 and the parent task's print statement in line 4 because `toLowerCase()` returns a new copy of the string with the lower-case conversion instead of attempting to update the original version.

```
1  finish { // Task T1
2    int[] A = new int[n]; // A is owned by T1
3    // ... initialize array A ...
4    // create a copy of array A in B
5    int[] B = new int[A.length]; System.arraycopy(A,0,B,0,A.length);
6    async { // Task T2 now owns B
7      int sum = computeSum(B,0,B.length-1); // Modifies B
8      System.out.println("sum = " + sum);
9    }
10   // ... update Array A ...
11   System.out.println(Arrays.toString(A)); //printed by task T1
12 }
```

Listing 13: Example of single-task ownership

2. *Single-task ownership tip:* If an object or array needs to be written multiple times after initialization, then try and restrict its ownership to a single task. This will entail making copies when sharing the object or array with other tasks. As in the Immutability tip, it depends on the algorithm whether the copying overhead can be acceptable or prohibitive.

   In the example in Listing 13, the parent Task $T1$ allocates and initializes array `A` in lines 2 and 3, and creates an `async` child Task $T2$ to compute its sum in line 6. Task $T2$ calls the `computeSum()` method that actually modifies its input array. To avoid a data race, Task $T1$ acts as the owner of array `A` and creates a copy of `A` in array `B` in lines 4 and 5/ Task $T2$ becomes the owner of `B`, while Task $T1$ remains the owner of `A` thereby ensuring that each array is owned by a single task.

3. *Ownership-transfer tip:* If an object or array needs to be written multiple times after initialization and also accessed by multiple tasks, then try and ensure that all the steps that read or write a location L in the object/array are totally ordered by dependences in CG. Ownership transfer is even necessary to support single-task ownership. In Listing 13, since Task $T1$ initializes array `B` as a copy of array `A`, $T1$ is the original owner of A. The ownership of `B` is then transferred from $T1$ to $T2$ when Task $T2$ is created with the `async` statement.

4. *Local-variable tip:* You do not need to worry about data races on local variables, since they are not possible. However, local variables in Java are restricted to contain primitive data types (such as `int`) and references to objects and arrays. In the case of object/array references, be aware that there may be a data race on the underlying object even if there is no data race on the local variable that refers to (points to) the object.

You will learn additional mechanisms for avoiding data races later in the course, when you study the *future*, *phaser*, *accumulator*, *isolated* and *actor* constructs.

2.6 Functional and Structural Determinism

A computation is said to be *functionally deterministic* if it always computes the same answer, when given the same inputs. By default, any sequential computation is expected to be deterministic with respect to its inputs; if the computation interacts with the environment (*e.g.,* a GUI event such as a mouse click, or a system call like System.nanoTime()) then the values returned by the environment are also considered to be inputs to the computation. Further, a computation is said to be *structurally deterministic* if it always computes the same computation graph, when given the same inputs.

The presence of data races often leads to functional and/or structural nondeterminism because a parallel program with data races may exhibit different behaviors for the same input, depending on the relative scheduling and timing of memory accesses involved in a data race. In general, the absence of data races

```
1    p.x = 0; q = p;
2    async p.x = 1; // Task T1
3    async p.x = 2; // Task T2
4    async { // Task T3
5      System.out.println("First read = " + p.x);
6      System.out.println("Second read = " + q.x);
7      System.out.println("Third read = " + p.x);
8    }
9     async { // Task T4
10     System.out.println("First read = " + p.x);
11     System.out.println("Second read = " + p.x);
12     System.out.println("Third read = " + p.x);
13   }
```

Listing 14: Example of a parallel program with data races

is not sufficient to guarantee determinism. However, the parallel constructs introduced in this module ("Module 1: Determinism") were carefully selected to ensure the following *Determinism Property*:

> If a parallel program is written using the constructs introduced in Module 1 *and is guaranteed to never exhibit a data race*, then it must be both functionally and structurally deterministic.

Note that the determinism property states that all data-race-free programs written using the constructs introduced in Module 1 are guaranteed to be deterministic, but it does not imply that all racy programs are non-deterministic.

The determinism property is a powerful semantic guarantee since the constructs introduced in Module 1 span a wide range of parallel programming primitives that 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 ("Concurrency").

2.6.1 Optional topic: Memory Models and assumptions that can be made in the presence of Data Races

Since the current state-of-the-art lacks a fool-proof approach for avoiding data races, this section briefly summarizes what assumptions can be made for parallel programs that may contain data races.

A *memory consistency model*, or *memory model*, is the part of a programming language specification that defines what write values a read may see in the presence of data races. Consider the example program in Listing 14. It exhibits multiple data races since location `p.x` can potentially be written in parallel by Tasks $T1$ and $T2$ and read in parallel by Tasks $T3$ and $T4$. $T3$ and $T4$ each read and print the value of `p.x` three times. (Note that `q.x` and `p.x` both refer to the same location.) It is the job of the memory model to specify what outputs are legally permitted by the programming language.

There is a wide spectrum of memory models that have been proposed in the literature. We briefly summarize three models for now, and defer discussion of a fourth model, the Java Memory Model, to later in the course:

1. *Sequential Consistency:* The Sequential Consistency (SC) memory model was introduced by Leslie Lamport in 1979 [9] and builds on a simple but strong rule *viz.,* all steps should observe writes to all locations in the same order. Thus, the SC memory model will not permit Task $T3$ to print "0, 1, 2" and Task $T4$ to print "0, 2, 1".

   While the SC model may be intuitive for expert system programmers who write operating systems and multithreaded libraries such as `java.util.concurrent`, it can lead to non-obvious consequences for

```
1   async { // Task T3
2     int p_x = p.x;
3     System.out.println("First read = " + p_x);
4     System.out.println("Second read = " + q.x);
5     System.out.println("Third read = " + p_x);
6   }
```

Listing 15: Rewrite of Task T3 from Listing 14

mainstream application programmers. For example, suppose an application programmer decided to rewrite the body of Task T3 as shown in Listing 15. The main change is to introduce a local variable `p_x` that captures the value of `p.x` in line 2, and replaces `p.x` by `p_x` in lines 3 and 5. This rewrite is perfectly legal for a sequential program, and should be legal for computations performed within a sequential step. However, a consequence of this rewrite is that Task $T3$ may print "0, 1, 0" as output, which would not be permitted by the SC model. Thus, an apparently legal code transformation within a sequential step has changed the semantics of the parallel program under the SC model.

2. *Location Consistency:* The Location Consistency (LC) memory model [4] was introduced to provide an alternate semantics to address the code transformation anomalies that follow from the SC model. The LC rule states that a read of location L in step S_i may receive the value from any *Most Recent Write* (MRW) of L relative to S_i in the CG. A MRW is a write operation that can potentially execute in parallel with S_i, or one that precedes S_i by a chain of dependence edges such that there is no other write of L on that chain. LC is a *weaker* model than SC because it permits all the outputs that SC does, as well as additional outputs that are not permitted by SC. For the program in Listing 14, the LC model permits Task $T3$ to print "0, 1, 2" and Task $T4$ to print "0, 2, 1" in the same execution, and also permits Task $T3$ to print "0, 1, 0" in a different execution.

3. *C++ Memory Model:* The proposed memory model for the new C++0x standard [3] makes the following assumption about data races:

   > "We give no semantics to programs with data races. There are no benign C++ data races."

   A data race that cannot change a program's output with respect to its inputs is said to be *benign*. A special case of benign races is when all write accesses to a location L (including the initializing write) write the same value to L. It is benign, because it does not matter how many writes have been performed on L before a read occurs, since all writes update L with the same value.

   Thus, the behavior of a program with data races is completely undefined in the C++ memory model. While this approach may be acceptable for systems programming languages like C/C++, it is unacceptable for type-safe languages like Java that rely on basic safety guarantees for pointers and memory accesses.

Why should you care about these memory models if you write bug-free code without data races? Because the code that you write may be used in conjunction with other code that causes your code to participate in a data race. For example, if your job to provide a sequential method that implements the body of Task $T3$ in Listing 14, the program that uses your code may exhibit data races even though your code may be free of bugs. In that case, you should be aware what the impact of data races may be on the code that you have written, and whether or not a transformation such as the one in Listing 15 is legal. The type of the shared location also impacts the assumptions that you make. On some systems, the guarantees for 64-bit data types such as `long` and `double` are weaker than those for smaller data types such as `int` and Java object references.

3 Loop-level Parallelism

3.1 Parallel Loops

As mentioned earlier, the `finish` and `async` constructs can be used to create parallel loops using the `finish`–`for`–`async` pattern shown in Listing 16. In this pseudocode, we assume that the `for` construct can be used to express sequential multidimensional (nested) loops. Unlike Java lambdas, we assume that the non-final values of i and j in the pseudocode are copied automatically when the `async` is created, thereby avoiding the possibility of data races on i and j. (There are other programming languages that support this convention, most notably `C++11` lambdas with the `=` capture clause.)

The `for` loop in Case 1 expresses a two-dimensional loop with $m \times n$ iterations. Since the body of this loop is an `async` statement, both loops i and j can run in parallel in Case 1. However, only loop i can run in parallel in Case 2, and only loop j can run in parallel in Case 3.

Most parallel programming languages include special constructs to embody the commonly used `finish`–`for`–`async` parallel loop pattern shown above in Listing 16. Following the notation used in other parallel languages and the $\forall$ mathematical symbol, we use the `forall` keyword to identify loops with single or multi-dimensional parallelism and an implicit finish. Listing 17, shows how the loops in Listing 16 can be rewritten using the `forall` notation.

```
1  // Case 1: loops i,j can run in parallel
2  finish for (point[i,j] : [0:m-1,0:n-1]) async A[i][j] = F(A[i][j]) ;
3
4  // Case 2: only loop i can run in parallel
5  finish for (point[i] : [1:m-1]) async
6    for (point[j] : [1:n-1]) // Equivalent to   for (j=1;j<n;j++)
7      A[i][j] = F(A[i][j-1]) ;
8
9  // Case 3: only loop j can run in parallel
10 for (point[i] : [1:m-1]) // Equivalent to   for (i=1;i<m;j++)
11   finish for (point[j] : [1:n-1]) async
12     A[i][j] = F(A[i-1][j]) ;
```

Listing 16: Examples of three parallel loops using finish-for-async (pseudocode)

```
1  // Case 1: loops i,j can run in parallel
2  forall (point[i,j] : [0:m-1,0:n-1]) A[i][j] = F(A[i][j]) ;
3
4  // Case 2: only loop i can run in parallel
5  forall (point[i] : [1:m-1])
6    for (point[j] : [1:n-1]) // Equivalent to   for (j=1;j<n;j++)
7      A[i][j] = F(A[i][j-1]) ;
8
9  // Case 3: only loop j can run in parallel
10 for (point[i] : [1:m-1]) // Equivalent to   for (i=1;i<m;j++)
11   forall (point[j] : [1:n-1])
12     A[i][j] = F(A[i-1][j]) ;
```

Listing 17: Examples of three parallel loops using forall (pseudocode)

```
1  finish {
2    for (int i = 0 ; i < n ; i++)
3      for (int j = 0 ; j < n ; j++)
4        async C[i][j] = 0;
5  }
6  finish {
7    for (int i = 0 ; i < n ; i++)
8      for (int j = 0 ; j < n ; j++)
9        async
10         for (int k = 0 ; k < n ; k++)
11           C[i][j] += A[i][k] * B[k][j];
12 }
13 System.out.println(C[0][0]);
```

Listing 18: Matrix multiplication program using finish-async

```
1  forall (point [i,j] : [0:n-1,0:n-1]) C[i][j] = 0;
2  forall (point [i,j] : [0:n-1,0:n-1])
3    for (point[k] : [0:K-1])
4      C[i][j] += A[i][k] * B[k][j];
5  System.out.println(C[0][0]);
```

Listing 19: Matrix multiplication program using forall

3.2 Parallel Matrix Multiplication

Consider the pseudocode fragment for a parallel matrix multiplication example in Listing 18.

This program executes all (i, j) iterations for line 4 in parallel to initialize array C, waits for all the iterations to complete at line 5, and then executes all (i, j) iterations for lines 10–11 in parallel (each of which executes the k loop sequentially). Since `async` and `finish` are powerful and general constructs, the structure of sequential and parallel loops in Listing 18 is not immediately easy to discern. Instead, the same program can be rewritten more compactly and clearly using `forall` loops as shown in Listing 19.

There are a number of features worth noting in Listing 19:

- The combination of `for`–`for`–`async` is replaced by a single keyword, `forall`. Multiple loops can be collapsed into a single `forall` with a multi-dimensional iteration space. (In Listing refcode:finish-async, both loop nests are two-dimensional.)
- The iteration variable for a `forall` is a *point* (integer tuple) such as $[i, j]$.
- The loop bounds can be specified as a rectangular *region* (dimension ranges) such as $[0 : n-1, 0 : n-1]$.
- We also extend the sequential `for` statement so as to iterate sequentially over a rectangular region, as in line 5.

We now briefly discuss the point and region constructs used in our pseudocode. A *point* is an element of an k-dimensional Cartesian space ($k \geq 1$) with integer-valued coordinates, where k is the rank of the point. A point's dimensions are numbered from 0 to $k-1$. Points can be used outside of `forall` loops, if so desired. For completeness, we summarize the following operations that are defined on a point-valued expression `p1`, even though it is unlikely that you will need to go beyond the use of points shown in Listing 19:

- `p1.rank` — returns rank of point `p1`

```
1  // Unchunked version
2  forall (point [i] : [0:n-1]) X[i] = Y[i] + Z[i];
3  . . .
4  // Chunked version
5  int nc = numWorkerThreads() ; // Set number of chunks to number of worker threads
6  int size = (n+nc-1)/nc; // chunk size = ceiling(n/nc) for integers n>=0, nc>0
7  forall (point [ii] : [0:nc-1]) {
8    int myLo = ii*size;
9    int myHi = Math.min(n-1, (ii+1)*size-1);
10   for(int i = myLo; i <= myHi; i++)
11     X[i] = Y[i] + Z[i];
12   }
13 }
```

Listing 20: Unchunked and chunked versions of a forall loop

- `p1.get(i)` — returns element in dimension `i` of point `p1`, or element in dimension ($i \mod p1.rank$) if $i < 0$ or $i \geq p1.rank$.
- `p1.lt(p2)`, `p1.le(p2)`, `p1.gt(p2)`, or `p1.ge(p2)` returns true if and only if `p1` is lexicographically $<$, $\leq$, $>$, or $\geq$ `p2`. These operations are only defined when `p1.rank` = `p1.rank`.

A k-dimensional *region* is a set of k-dimensional points, defined as a Cartesian product of *low:high* contiguous subranges in each dimension. Thus, $[1:10]$ is a 1-dimensional region consisting of the 10 points $[1], \ldots, [10]$, and $[1:10, -5:5]$ is a 2-dimensional region consisting of 110 points since the first dimension has 10 values $(1 \ldots 10)$ and the second dimension has 11 values $(-5 \ldots 5)$. Likewise, the region `[0:200,1:100]` specifies a collection of two-dimensional points `(i,j)` with `i` ranging from `0` to `200` and `j` ranging from `1` to `100`. Regions are used to define the range for sequential point-wise `for` and parallel `forall` loops.

A task executes a point-wise `for` statement by sequentially enumerating the points in its region in canonical lexicographic order, and binding the components of the points to the index variables defined in the `for` statement e.g., variable `k` in line 3 of Listing 19. A convenience relative to the standard Java idiom, "`for (int i = low; i <= high; i++)`", is that the upper bound, `high`, is re-evaluated in each iteration of a Java loop, but it is only evaluated once in a `[low:high]` region expression. Another convenience is that loops can be easily converted from sequential to parallel (or vice versa) by replacing `for` by `forall`.

Finally, we include a `forasync` statement that is like `forall` but *does not* include an implicit `finish` statement. The statement `forasync (point p : R) S` supports parallel iteration over all the points in region `R` by launching each iteration `S` as a separate `async`. Just as with standard `async` statements, a separate `finish` construct is needed to await termination of all iterations in a `forasync`.

3.3 Iteration Grouping: Chunking of Parallel Loops

Though the `forall` construct is convenient for the programmer, the approach of creating a separate `async` task for each iteration can lead to excessive overheads. For a parallel loop, there are additional choices available. A natural approach to reduce the overhead of parallel loops is that of batching or "chunking" groups of iterations together so that iterations in the same chunk execute sequentially within a single `async` task, but parallelism can be exploited across chunks. The chunks size plays a critical role in determining the effectiveness of chunking. If it is too small, then the overhead can still be an issue due to the small size of `async` tasks. If it is too large, then there is a danger of losing too much parallelism. Fortunately, it is possible to set up chunking of parallel loops such that the number of chunks (or equivalently, the chunk size) can be specified as a runtime parameter that can be "tuned" for a given input size and parallel machine. For loops in which the amount of work per iteration is fixed, a common approach is to set the number of chunks to the number of available processors.

```
1  // Return range for chunk ii if range [rLo:rHi] is divided into nc chunks
2  static region getChunk(int rLo, rHi, int nc, int ii) {
3    if (rLo > rHi) return [0:-1]; // Empty region
4    assert(nc > 0); // number of chunks must be > 0
5    assert(0 <= ii && ii < c); // ii must be in [0:c-1] range
6    int chunkSize = (rHi-rLo+c-1)/c;
7    int myLo = rLo +  ii*chunkSize;
8    int myHi = Math.min(rHi, rLo + (ii+1)*chunkSize - 1);
9    return [myLo:myHi]; // range for chunk ii
10 }
11
12 // Chunked version using getChunk function
13 int nc = numWorkerThreads() ; // Set number of chunks to number of worker threads
14 forall (point [ii] : [0:nc-1]) {
15   region myRange = getChunk(rLo, rHi, nc, ii);
16   int myLo = myRange.rank(0).low();
17   int myHi = myRange.rank(0).high();
18   for(int i = myLo; i <= myHi; i++)
19     X[i] = Y[i] + Z[i];
20   }
21 }
```

Listing 21: Unchunked and chunked versions of a one-dimensional forall loop

Listing 20 includes unchunked and chunked version of an example `forall` loop. The chunking is achieved by creating an outer parallel `forall` loop with number of chunks = `nc` and an inner sequential `for` loop that executes the iterations in a given chunk. We assume the availability of a library call, `numWorkerThreads()`, that returns the number of worker threads with which the parallel program execution was initiated; this is a suitable value for the number of chunks in this example. The `size` variable is then set to the expected chunk size i.e., number of iterations per chunk. If `nc` evenly divides `n`, then we could just set `size` to equal `n/nc`. Otherwise, we'd like to set `size` to be $\lceil$`n/nc`$\rceil$. Since Java does not provide a convenient primitive for performing this operation on integers[5], we use the mathematical property that $\lceil x/y \rceil$ equals $\lfloor (x+y-1)/y \rfloor$ for integers x, y such that $y > 0$. (Recall that standard integer division in languages like Java and C truncates downwards like the floor function.)

After `nc` and `size` have been identified, the outer `forall` loop is set up to execute for `nc` iterations with index variable `ii` ranging from 0 to `nc` -1. Each `forall` iteration then computes the range of iterations for its chunk, `myLo...myHi` as a function of its index variable `ii`. The use of the `Math.min` function ensures that the last chunk stays within the bounds of the original loop (in the case that `nc` does not evenly divide `n`). This division into chunks guarantees that each iteration of the original loop is assigned to exactly one chunk and that all chunks have the same size when `n` is a multiple of `nc`.

The above calculation can get more complicated when the lower bound of the original loop is non-zero, and when the original `forall` has a multidimensional region. For general loop bounds, we can introduce a helper function called `GetChunk()` as shown in Listing 21. For multidimensional regions, the `GetChunk()` function can simply be performed separately in each dimension, provided that the total number of chunks is also given as a multidimensional region that specifies the number of chunks in each dimension.

3.4 Barriers in Parallel Loops

Thus far, you have learned the fundamentals of task creation (`async`, `async await`) and task termination (`finish`, *future*`.get()`). There are many algorithms that also need to implement some form of *directed synchronization* among tasks, with well defined *predecessor* and *successor* steps in the computation graph. While

[5] `Math.ceiling()` only operates on values of type `double`.

```
1 rank.count = 0; // rank object contains an int field, count
2 forall (point[i] : [0:m-1]) {
3   int square = i*i;
4   System.out.println("Hello from task " + i + " with square = " + square);
5   System.out.println("Goodbye from task " + i + " with square = " + square);
6 }
```

Listing 22: Hello-Goodbye forall loop

the `finish` and *future*.`get()` constructs impose directed synchronization, they only apply to cases where the predecessor has to terminate for the synchronization to be enabled (via *join* edges in the computation graph).

To understand the need for other forms of directed synchronization, especially a *barrier* synchronization, consider the simple "Hello-Goodbye" `forall` example program shown in Listing 22. This example contains a single `forall` loop with `m` iterations numbered `0..m-1`. The main program task starts execution at line 1, creates `m` child tasks at line 2, and waits for them to finish after line 7. (Recall that `forall` is shorthand for `finish`–`for`–`async`.) Each `forall` iteration (task) then prints a "Hello" string (line 5) and a "Goodbye" string (line 6). While the Hello and Goodbye strings from the same task must be printed in order, there is no other guarantee on the relative order among print statements from different tasks[6]. For example, the following output is legal for the `m` = 4 case:

```
Hello from task ranked 0 with square = 0
Hello from task ranked 1 with square = 1
Goodbye from task ranked 0 with square = 0
Hello from task ranked 2 with square = 4
Goodbye from task ranked 2 with square = 4
Goodbye from task ranked 1 with square = 1
Hello from task ranked 3 with square = 9
Goodbye from task ranked 3 with square = 9
```

Now, let us consider how to modify the program in Listing 22 so as to ensure that all Hello strings are printed before any Goodbye string. One approach would be to replace the `forall` loop by two `forall` loops, one for the Hellos and one for the Goodbyes. However, a major inconvenience with this approach is that all local variables in the first `forall` loop (such as `square`) need to be copied into objects or arrays so that they can be communicated into the second `forall` loop. The preferred approach instead is to use `next` statements, commonly known as *barriers*, as shown in Listing 23.

The semantics of a `next` statement inside a `forall` is as follows. A `forall` iteration i suspends at `next` until all iterations arrive (*i.e.,* all iterations complete their *previous phase*), after which iteration i can advance to its *next phase*. Thus, in Listing 23, `next` acts as a *barrier* between Phase 0 which corresponds to all the computations executed before `next` and Phase 1 which corresponds to all the computations executed after `next`.

Figure 13 illustrates how the barrier synchronization (`next` statement) works for the program example in Listing 23 for the `m` = 4 case. Each task (iteration) performs a *signal* (SIG) operation when it enters the barrier, and then performs a *wait* (WAIT) operation thereby staying idle until all tasks have entered the barrier. In the scenario shown in Figure 13, iteration $i = 0$ is the first to enter the barrier, and has the longest idle time. Iteration $i = 2$ is the last to enter the barrier, so it has no idle time since its SIGNAL operation releases all iterations waiting at the barrier.

Can you think of real-world situations that can be modeled by barriers? Consider a (somewhat elaborate)

[6]The source of nondeterminism in this example arises from the race conditions among the print statements, which violates the precondition of the Structural Determinism property in Section 2.6.

```
1 rank.count = 0; // rank object contains an int field, count
2 forall (point[i] : [0:m-1]) {
3   // Start of phase 0
4   int square = i*i;
5   System.out.println("Hello from task " + i + " with square = " + square);
6   next; // Acts as barrier between phases 0 and 1
7   // Start of phase 1
8   System.out.println("Goodbye from task " +  i + " with square = " + square);
9 }
```

Listing 23: Hello-Goodbye forall loop with barrier (next) statement

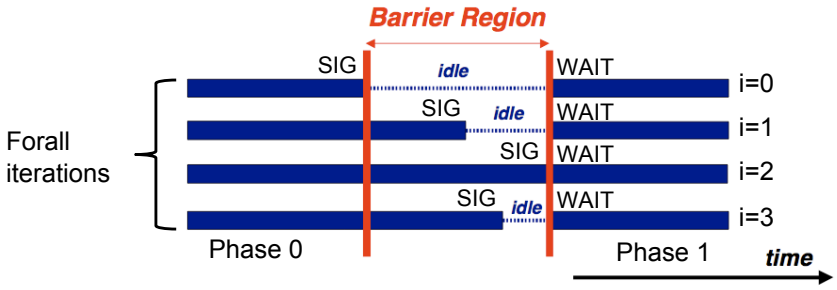

Figure 13: Illustration of barrier synchronization (next statement) for program example in Listing 23

family meal where no one starts eating the main course until everyone has finished their soup, and no one starts eating dessert until everyone has finished their main course. In this case, each family member can be modeled as a `forall` iteration, each course — soup, main dish, and dessert — can be modeled as a *phase*, and each synchronization between phases can be modeled as a barrier.

The `next` statement in a `forall` provides its parallel iterations/tasks with a mechanism to periodically rendezvous with each other. The scope of synchronization for a `next` statement is its closest enclosing `forall` statement[7]. Specifically, when iteration i of the `forall` executes `next`, it is informing the other iterations that it has completed its current phase and is now waiting for all other iterations to complete their current phase by executing `next`. There is no constraint on which statements are executed by a `forall` iteration before or after a `next` statement. There is also no constraint on where a `next` can be performed by a `forall` iteration. For example, a `next` can be performed by a task in the middle of an `if`, `while` or `for` statement, and different `forall` iterations can even perform `next` at different program points in different methods.

When a `forall` iteration i terminates, it also drops its participation in the barrier *i.e.,* other iterations do not wait for iteration i past its termination. This rule avoids the possibility of deadlock with `next` operations. The example in Listing 24 illustrates this point.

The iteration numbered i in the `forall`-i loop in line 1 of Listing 24 performs a sequential `for`-j loop in line 2 with $i+1$ iterations ($0 \leq j \leq i$). Each iteration of the `for`-j loop prints (i, j) before performing a `next` operation. Thus, j captures the phase number for each `forall`-i iteration participating in the barrier. Iteration $i = 0$ of the `forall`-i loop prints $(0, 0)$, performs a `next`, and then terminates. Iteration $i = 1$ of the `forall`-i loop prints $(1, 0)$, performs a `next`, prints $(1, 1)$, performs a `next`, and then terminates. And so on, as shown in Figure 14 which illustrates how the set of `forall` iterations synchronizing on the barrier decreases after each phase in this example.

[7]Later, you will learn how the `next` statement can be used outside `forall`'s as well.

```
1 forall (point[i] : [0:m-1]) {
2   for (point[j] : [0:i]) {
3     // Forall iteration i is executing phase j
4     System.out.println("(" + i + "," + j + ")");
5     next;
6   }
7 }
```

Listing 24: Example of forall loop with varying numbers of `next` operations across different iterations

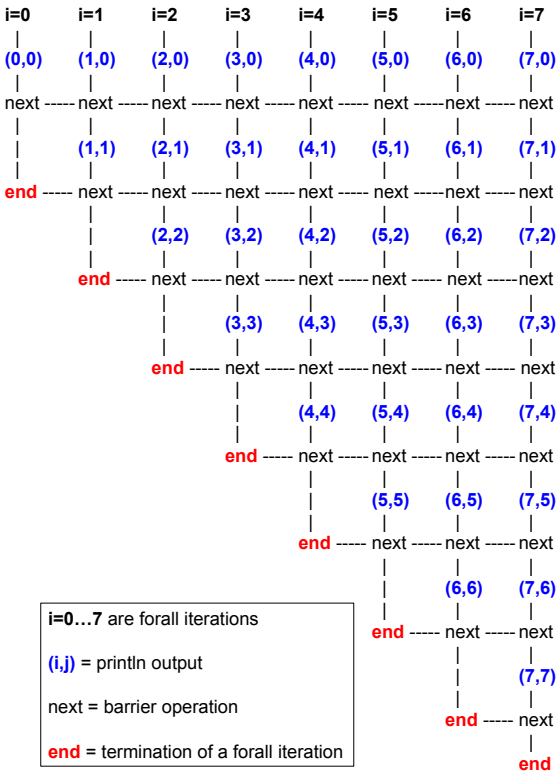

Figure 14: Illustration of the execution of `forall` example in Listing 24

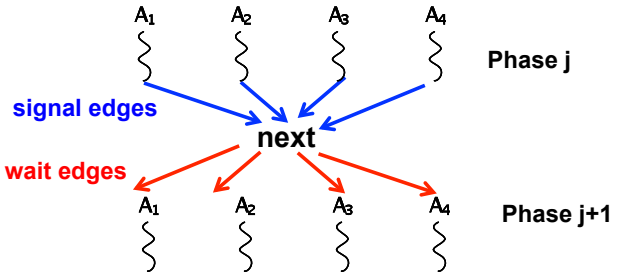

Figure 15: Modeling a next operation in the Computation Graph with Signal and Wait edges and a Next node

```
1  rank.count = 0; // rank object contains an int field, count
2  forall (point[i] : [0:m-1]) {
3    // Start of Hello phase
4    int square = i*i;
5    System.out.println("Hello from task " +  i + " with square = " + square);
6    next; // Barrier
7    if ( i == 0 ) System.out.println("LOG: Between Hello & Goodbye Phases");
8    next; // Barrier
9    // Start of Goodbye phase
10   System.out.println("Goodbye from task " +  i + " with square = " + square);
11 }
```

Listing 25: Hello-Goodbye program in Listing 23 extended with a second barrier to print a log message between the Hello and Goodbye phases

Figure 15 shows how a `next` operation can be modeled in the dynamic Computation Graph by adding SIGNAL and WAIT edges. A_1, A_2, A_3, A_4 represent four iterations in a common `forall` loop. The execution of a `next` statement causes the insertion of a single `next` node in the CG as shown in Figure 15. SIGNAL edges are added from the last step prior to the `next` in each `forall` iteration to the `next` node, and WAIT edges are added from the `next` node to the continuation of each `forall` iteration. Collectively, these edges enforce the barrier property since all tasks must complete their *Phase j* computations before any task can start its *Phase j+1* computation.

3.4.1 Next-with-single Statement

Consider an extension to the Hello-Goodbye program in which we want to print a log message after the Hello phase ends but before the Goodbye phase starts. Thought this may sound like a contrived problem, it is representative of logging functionalities in real servers where status information needs to be printed at every "heartbeat".

A simple solution is to assign this responsibility to iteration $i = 0$ of the `forall` loop as shown in Listing 25. The log message is printed by iteration $i = 0$ on line 8 after a barrier in line 7 and before a second barrier in line 9. Though correct, it is undesirable to use two barriers when trying to log a single phase transition. To mitigate this problem, the `next` statement offers a *next-with-single* option. This option has the form `next single` ⟨*single-statement*⟩, where ⟨*single-statement*⟩ is a statement that is performed exactly once after all tasks have completed the previous phase and before any task begins its next phase. The CG edges for a next-with-single statement are shown in Figure 16.

Listing 26 shows how a next-with-single statement can be used to perform the same computation as in Listing 25 by using one barrier operation (with a single statement) instead of two. Note that no `if` statement is needed in the body of the single statement, since it will be executed exactly once by a randomly selected

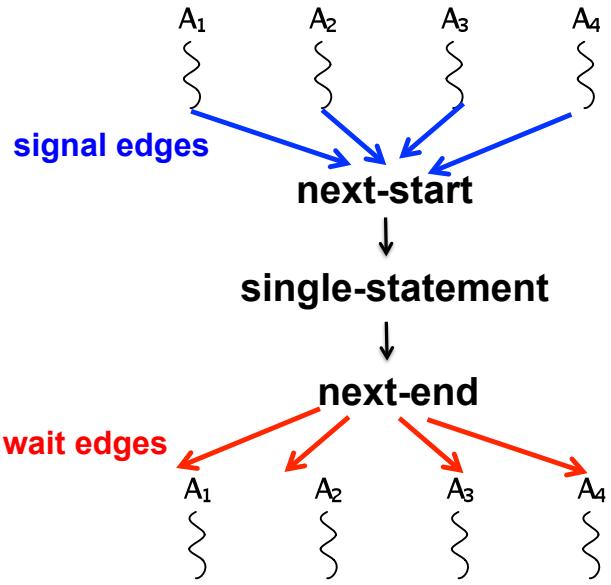

Figure 16: Modeling a next-with-single statement in the Computation Graph

```
1  rank.count = 0; // rank object contains an int field, count
2  forall (point[i] : [0:m-1]) {
3    // Start of Hello phase
4    int square = i*i;
5    System.out.println("Hello from task " +  i + " with square = " + square);
6    next single { // single statement
7      System.out.println("LOG: Between Hello & Goodbye Phases");
8    }
9    // Start of Goodbye phase
10   System.out.println("Goodbye from task " +  i + " with square = " + square);
11 }
```

Listing 26: Listing 25 extended with a next-with-single statement in lines 15–17

iteration of the `forall` loop.

3.5 One-Dimensional Iterative Averaging

To further motivate the need for barriers, consider the one-dimensional iterative averaging algorithm illustrated in Figure 17. The idea is to initialize a one-dimensional array of `double` with `n+2` elements, `myVal`, with boundary conditions, `myVal[0] = 0` and `myVal[n+1] = 1`. Then, in each iteration, each interior element (with index in the range $1 \ldots n$) is replaced by the average of its left and right neighbors. After a sufficient number of iterations, we expect each element of the array to converge to $myVal[i] = i/(n+1)$. For this final quiescent equilibrium state, it is easy to see that $myVal[i] = (myVal[i-1] + myVal[i+1])/2$ must be the average of its left and right neighbors, for all i in the range $1 \ldots n$.

3.5.1 Idealized Implementations of One-Dimensional Iterative Averaging Example

In this section, we discuss two idealized implementations of the one-dimensional iterative averaging example. The first version in Listing 27 uses a `for`–`forall` structure, whereas the second version in Listing 28 uses a `forall`–`for`–`next` structure.

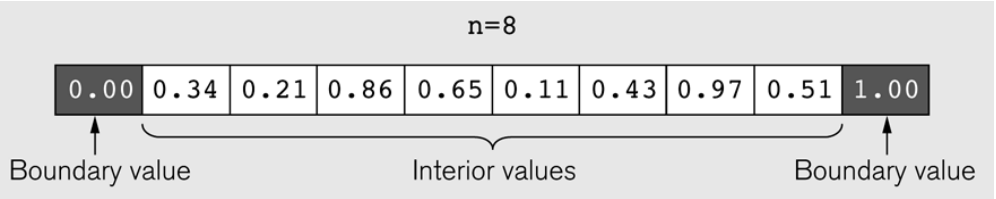

Figure 17: Illustration of the One-Dimensional Iterative Averaging Example for $n = 8$ (source: Figure 6.19 in [10])

```
1  double[] myVal = new double[n]; myVal[0] = 0; myVal[n+1] = 1;
2  for (point [iter] : [0:iterations-1]) {
3    // Output array MyNew is computed as function of
4    // input array MyVal from previous iteration
5    double[] myNew = new double[n]; myNew[0] = 0; myNew[n+1] = 1;
6    forall (point [j] : [1:n]) { // Create n tasks
7      myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
8    } // forall
9    myVal = myNew; // myNew becomes input array for next iteration
10 } // for
```

Listing 27: Idealized One-Dimensional Iterative Averaging program using for-forall computation structure with n parallel tasks working on elements $1 \ldots n$

The first version in Listing 27 contains an outer `for`–*iter* loop in line 2 which is intended to run for a sufficiently large number of `iterations` to guarantee convergence. (Many real-world applications use a `while` loop instead of a counted `for` loop to test for convergence.) Each iteration of the `for`–*iter* loop starts by allocating and initializing a new output array, `myNew`, in line 5. For each instance of the `forall` in lines 6–8, `myVal` is a reference to the array computed in the previous iteration and `myNew` is a reference to the array computed in the current iteration. Line 6 performs the averaging step in parallel due to the `forall`–*j* loop. There are no data races induced by line 6, since the reads and writes are performed on distinct arrays and each write is performed on a distinct location in `myNew`.

You learned earlier that repeated execution of `forall`, as in Listing 27, can incur excessive overhead because each `forall` spawns multiple `async` tasks and then waits for them to complete with an implicit `finish` operation. Keeping this observation in mind, Listing 28 shows an alternate implementation of the iterative averaging example using the `next` (barrier) statement. Now, the `forall` loop has moved to the outer level in line 3 and the `for` loop to the inner level in line 4. Further, the array references `myVal` and `myNew` are stored in fields rather than local variables so that they can be updated inside a `forall` iteration. Finally, a next-with-single statement is used in lines 5–8 to ensure that the "`myVal = myNew;`" copy statement and the allocation of a new array in lines 6 and 7 are executed exactly once during each phase transition.

3.5.2 Optimized Implementation of One-Dimensional Iterative Averaging Example

Though Listing 28 in Section 3.5.1 reduced the number of tasks created by the use of an outer `forall` with a barrier instead of an inner `forall`, two major inefficiencies still remain. First, the allocation of a new array in every iteration of the `for`–*iter* loop is a major source of memory management overhead. Second, the `forall` loop creates one task per array element which is too fine-grained for use in practice.

To address the first problem, we observe that only two arrays are needed for each iteration, an input array and an output array. Therefore, we can get by with two arrays for the entire algorithm by just swapping the roles of input and output arrays in every iteration of the `for`–*iter* loop. To address the second problem, we can use loop chunking as discussed in Section 3.3. Specifically, the `forall` loop can be created for `t` $\ll$ `n`

```
1  // Assume that myVal and myNew are mutable fields of type double[]
2  myNew = new double[n]; myNew[0] = 0; myNew[n+1] = 1;
3  forall (point [j] : [1:n]) { // Create n tasks
4    for (point [iter] : [0:iterations-1]) {
5      next { // single statement
6        myVal = myNew; //  myNew becomes input array for next iteration
7        myNew = new double[n]; myNew[0] = 0; myNew[n+1] = 1;
8      }
9      myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
10   } // for
11 } // forall
```

Listing 28: One-Dimensional Iterative Averaging Example using forall-for-next-single computation structure with n parallel tasks working on elements $1 \ldots n$

```
1  double[] val1 = new double[n]; val[0] = 0; val[n+1] = 1;
2  double[] val2 = new double[n];
3  int batchSize = CeilDiv(n,t); // Number of elements per task
4  forall (point [i] : [0:t-1]) { // Create t tasks
5    double[] myVal = val1; double myNew = val2; double[] temp = null;
6    int start = i*batchSize + 1; int end = Math.min(start+batchSize-1,n);
7    for (point [iter] : [0:iterations-1]) {
8      for (point[j] : [start:end])
9        myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
10     next; // barrier
11     temp = myNew; myNew = myVal; myVal = temp; // swap(myNew, myVal)
12   } // for
13 } // forall
```

Listing 29: One-Dimensional Iterative Averaging Example using forall-for-next computation structure with t parallel tasks working on an array with $n+2$ elements (each task processes a batch of array elements)

iterations, and each iteration of the `forall` loop can be responsible for processing a batch of n/t iterations sequentially.

Keeping these observations in mind, Listing 29 shows an alternate implementation of the iterative averaging example. The `forall` loop is again at the outermost level (line 4), as in Listing 28. However, each iteration of the `forall` now maintains local variables, `myVal` and `myNew`, that point to the two arrays. The *swap(myNew, myVal)* computation in line 11 swaps the two references so that `myNew` becomes `myVal` in the next iteration of the `for` loop. (There are two distinct array objects allocated in lines 1 and 2, whereas `myVal` and `myNew` are pointers to them that are swapped each time line 10 is executed.) Maintaining these pointers in local variables avoids the need for synchronization in the *swap* computation in line 11.

Line 3 computes `batchSize` as $\lceil n/t \rceil$. Line 6 computes the `start` index for batch i, where $0 \leq i \leq t-1$. The `for` loop in line 7 sequentially computes all array elements assigned to batch i. (The `Math.min()` function is used to ensure that the last iteration of the last batch equals n.) This form of batching is very common in real-world parallel programs. In some cases, the compiler can perform the batching (chunking) transformation automatically, but programmers often perform the batching by hand so as to be sure that it is performed as they expect.

The `for`–*iter* loop at line 7 contains a `next` operation in line 10. The barrier semantics of the `next` statement ensures that all elements of `myNew` are computed in line 9 across all tasks, before moving to line 11 and the next iteration of the `iter` loop at line 8. We can see that only t tasks are created in line 4, and the same

tasks repeatedly execute the iterations of the `iter` loop in line 7 with a barrier synchronization in line 10.

References

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. URL `http://doi.acm.org/10.1145/1465482.1465560`.

[2] John Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21:613–641, August 1978. ISSN 0001-0782. URL `http://doi.acm.org/10.1145/359576.359579`.

[3] Hans-J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 68–78, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. URL `http://doi.acm.org/10.1145/1375581.1375591`.

[4] Guang R. Gao and Vivek Sarkar. Location consistency-a new memory model and cache consistency protocol. *IEEE Trans. Comput.*, 49(8):798–813, 2000. ISSN 0018-9340. URL `http://dx.doi.org/10.1109/12.868026`.

[5] R. Graham. Bounds for Certain Multiprocessor Anomalies. *Bell System Technical Journal*, (45):1563–1581, 1966.

[6] Yi Guo. *A Scalable Locality-aware Adaptive Work-stealing Scheduler for Multi-core Task Parallelism*. PhD thesis, Rice University, Aug 2010.

[7] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31:532–533, May 1988. ISSN 0001-0782. URL `http://doi.acm.org/10.1145/42411.42415`.

[8] Robert Halstead, JR. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions of Programming Languages and Systems*, 7(4):501–538, October 1985.

[9] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28:690–691, September 1979. ISSN 0018-9340. URL `http://dx.doi.org/10.1109/TC.1979.1675439`.

[10] Calvin Lin and Lawrence Snyder. *Principles of Parallel Programming*. Addison-Wesley, 2009.