

COMP 322: Fundamentals of Parallel Programming

Module 1: Parallelism

©2015 by Vivek Sarkar

February 24, 2015

DRAFT VERSION – PLEASE DO NOT DISTRIBUTE

Contents

0	Introduction	3
0.1	What is Parallel Programming?	3
0.2	Why is Parallel Programming important?	5
0.3	Course Organization	7
1	Task-level Parallelism	8
1.1	Task Creation and Termination (Async, Finish)	8
1.2	Computation Graphs	14
1.3	Ideal Parallelism	16
1.4	Multiprocessor Scheduling	17
1.5	Parallel Speedup and Amdahl’s Law	17
1.6	Parallel Quicksort	19
2	Functional Parallelism and Determinism	30
2.1	Future Tasks and Functional Parallelism	30
2.2	Memoization	32
2.3	Finish Accumulators	34
2.4	Map Reduce	35
2.5	Data Races	38
2.6	Functional and Structural Determinism	40
3	Loop-level Parallelism	43
3.1	Parallel Loops	43
3.2	Parallel Matrix Multiplication	43
3.3	Iteration Grouping: Chunking of Parallel Loops	45
3.4	Barriers in Parallel Loops	46
3.5	One-Dimensional Iterative Averaging	52
4	Dataflow Synchronization and Pipelining	54

4.1	Fuzzy Barriers	54
4.2	Point-to-point Synchronization with Phasers	55
4.3	One-Dimensional Iterative Averaging Example with Point-to-Point Synchronization	63
4.4	Pipeline Parallelism	65
4.5	Data-Driven Futures	68
A	Optional topic: Abstract vs. Real Performance	70
A.1	Work-sharing vs. Work-stealing schedulers	71
A.2	Modeling and Measurement of Overhead	72
A.3	The “seq” clause for Async tasks	75

0 Introduction

0.1 What is Parallel Programming?

Since the dawn of early digital computers and the Von Neumann computing model [24]¹, programming has been viewed as a *sequential* abstraction of computation. Sequential programming is a step-by-step specification of each operation in a computation as a sequence — a program is a sequence of statements, a loop is a sequence of iterations, an expression is a sequence of operations, and so on. The sequential programming model has served the computing industry well for over six decades as the default model for the vast majority of programming languages. Sequential programming has also simplified reasoning about program execution because a sequential program always performs its operations in a predefined order. However, in many respects, sequential programming can be considered “unnatural” because many application domains modeled by software (*e.g.*, physical systems, social networks) are inherently parallel rather than sequential.

The concept of *parallelism* is often used to denote multiple events occurring side-by-side in space and time. In Computer Science, we use it to denote simultaneous computation of operations on multiple processing units. Thus, *parallel programming* is a specification of operations in a computation that can be executed in parallel on different processing units. This course will focus on the *fundamental concepts* that underlie parallel programming so as to provide you with the foundations needed to understand any parallel programming model that you encounter in the future.

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]. 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. 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. We can think of the start of program execution as a single root task. We now informally introduce two constructs, *async* and *finish*²:

- The statement “**async** ($stmt1$)” causes the parent task to create a new child task to execute the body of

¹These lecture notes include citation such as [24] as references for optional further reading.

²These constructs have some similarities to the “fork” and “join” constructs available in many languages, but there are notable differences as well, as you will learn later in the course.

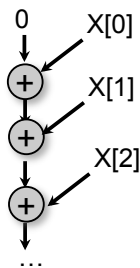


Figure 1: Computation graph for Algorithm 1 (Sequential ArraySum)

the *async*, $\langle stmt1 \rangle$, *asynchronously* (i.e., before, after, or in parallel) with the remainder of the parent task.

- The statement “**finish** $\langle stmt2 \rangle$ ” causes the parent task to execute the body of the *finish*, $\langle stmt2 \rangle$, and then wait until all *async* tasks created within $\langle stmt2 \rangle$ have completed.

The notation, $\langle stmt \rangle$, refers to any legal program statement *e.g.*, if-then-else, for-loop, method call, or a block enclosed in { } braces. *Async* and *finish* statements may be arbitrarily nested, so they can be contained in $\langle stmt \rangle$ too. (The use of angle brackets in “ $\langle stmt \rangle$ ” follows a standard notational convention to denote units of a program. They are unrelated to the < and > comparison operators used in many programming languages.)

We can use *async* and *finish* to obtain a simple parallel program for computing an array sum as shown in Algorithm 2. The graph structure for Algorithm 2 is shown in Figure 2. 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.

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*;

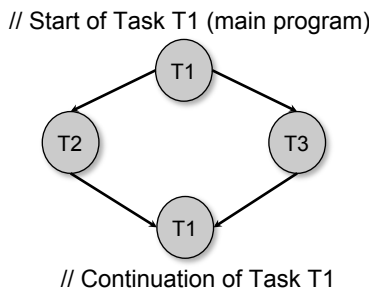


Figure 2: Computation graph for code example in Algorithm 2 (Two-way Parallel ArraySum)

0.2 Why is Parallel Programming important?

Historically, parallel programming was motivated by the need for speed and memory. When a problem is too large to fit or solve quickly on a single computer, we use multiple computers to solve it instead. Figure 3 shows the historical trend of the number of processors used in the Top 500 computers³ during 1993–2010. In 1993, the maximum number of processors used was in the 1025–2048 range, whereas that number increased to over 128,000 in 2010 and is projected to reach one billion in the next decade!

Until recently, all computer users could reasonably expect the speed of a single processor to increase rapidly with new generations of hardware technology, and the motivation for parallel computing was low among mainstream computer users. Now, parallel programming has become important for all computers in the world — even laptops and mobile devices — because clock frequencies for individual processors are no longer increasing. For example, in 1980, a major milestone for researchers was to build a microprocessor that could execute one million instructions per second (1 MIPS) with clock speeds in the range of 1 Megahertz (10^6 cycles per second). Now, all commodity personal computers and laptops have processors with clock speeds that exceed 1 Gigahertz (10^9 cycles per second). This increase in speed was predicted by two observations from the 1960’s and 1970’s that are now referred to as Moore’s Law [20] and Dennard Scaling [7]. Moore’s Law predicted that the transistor density of semiconductor chips would double roughly every 18 months *i.e.*, the transistor feature size would decrease by a factor of $\sqrt{2}$ roughly every 18 months. Dennard Scaling predicted that as transistors get smaller, they will switch faster and use less power. Even though this ability to increase the number of transistors in a chip continues unabated, a major disruption for the computer industry is that clock speeds are not expected to increase beyond the Gigahertz level in the foreseeable future.

There are multiple complex factors that have contributed to clock speeds plateauing at the Gigahertz level. They involve a detailed understanding of semiconductor physics and silicon technologies, but here’s a simple summary. In a nutshell, for clock speeds in the Gigahertz or higher range, the power consumed by a processor using current device technologies varies as the cube of the frequency, $Power \propto (Frequency)^3$. Processor chips in laptop and personal computers are typically limited to consume at most 75 Watts because a larger power consumption would lead to chips overheating and malfunctioning, given the limited air-cooling capacity of these computers. This power limitation has been referred to as the “Power Wall”. The Power Wall is even more critical for smart-phones and handheld devices than for server systems because larger power consumption leads to a shorter battery life. Thus, the dilemma facing the computer industry is: what to do when the number of transistors continues to double every 18 months, but the power budget and clock frequency can no longer be increased?

This dilemma is being solved by the introduction of *multicore* processor chips, each of which contains multiple processor cores. The earliest multicore processors contained two processor cores that can execute independent instruction streams, such as the two async tasks shown in Algorithm 2. This is a sea change in the history of

³This list is compiled by measuring the rate at which each computer can perform a pre-assigned dense matrix computation called High Performance LINPACK (HPL).

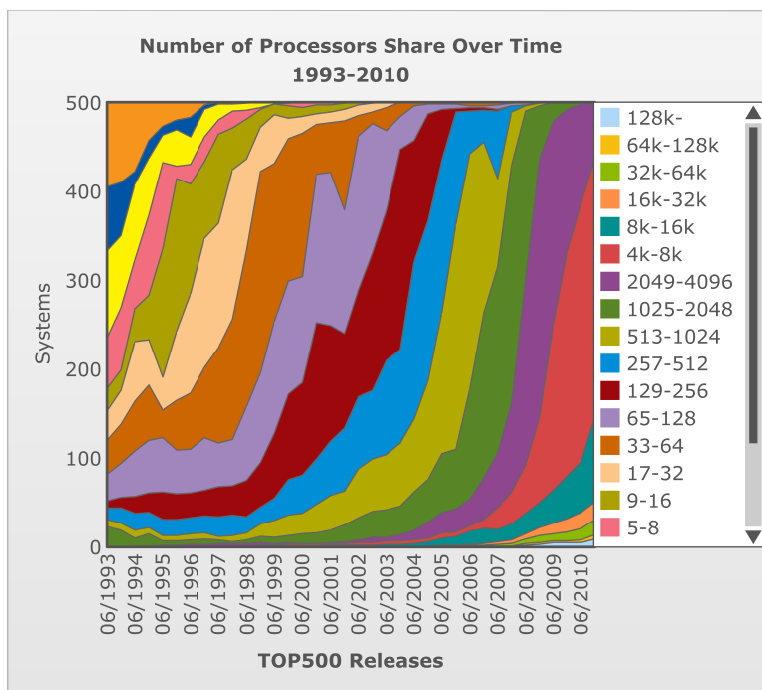


Figure 3: Historical chart showing the distribution of the number of processors used in the Top 500 computers from 1993 to 2010 (source: <http://www.top500.org>)

personal computers, since the only parallel computers available until 2005 were expensive servers with prices that started at around 100,000 US dollars. How did the introduction of multicore processors help with overcoming the Power Wall? Consider a 1 GHz processor chip that consumes power P , and two options for how to use the extra transistors when the number of transistors is doubled. With Option A, the clock frequency is doubled to 2 GHz thereby leading to a power consumption of $8P$ (recall that $Power \propto (Frequency)^3$). With Option B, two cores are created, each with clock speed still at 1 GHz, thereby leading to a power consumption of $2P$ (with power P consumed by each core). In theory, both options deliver the same computation capacity, but Option B does so with one-fourth the power of Option A⁴. Extrapolating to the future, the number of cores per chip will have to continue to increase because of the Power Wall. Some projections are that the number of cores in general-purpose chips will exceed one thousand by 2020. Some specialized processors such as NVIDIA’s Kepler GPGPU chip already contain more than 1500 cores today.

In addition to future hardware roadmaps, there is another important trend in 21st computing that is fueling an increased demand for parallel computing — this is the trend popularly referred to as *big data*. Analyzing data at volumes and velocities that are growing at an exponential rate is now emerging as the “fourth paradigm” for scientific discovery, and has also become a critical source of innovation and disruption across a wide spectrum of industries. Analysis of this large volume of complex data to derive knowledge, therefore, requires parallel computing by necessity since it is impractical to process terabytes or petabytes of data on a single processor.

The final observation is that the challenge of exploiting parallelism across these large numbers of cores falls squarely on the shoulders of software. When hardware makes multiple cores available to the user, it expects that software will somehow find a way of decomposing its workload to execute in parallel across the cores. Of course, the majority of computer users will not need to worry about parallel programming, just like the majority of computer users today don’t worry about sequential programming. However, given that the entire

⁴Caveat: a number of simplifying assumptions were made in this analysis.

computing industry has bet on parallelism, there is a desperate need for all Computer Science majors to be aware of the fundamentals of parallel software and parallel programming. Further, it is widely acknowledged that many commonly available parallel programming models provide low-level primitives that can be very difficult to compose, debug or reason about. As an example, Prof. John Hennessy, a prominent computer scientist (and also president of Stanford University) has observed that the “golden opportunity” for advances in Computer Engineering is “a programming and architectural model that makes parallel programming easy (in the sense that achieving both correctness and performance is not very difficult) and the result remains reasonably portable. If a group solves this problem, they will be heroes.” The importance of parallel computing, and the challenges of developing parallel software using many current approaches provided the motivation for our creating a sophomore-level course on “Fundamentals of Parallel Programming” at Rice that provides a pedagogic approach to structured parallel programming to prepare you for our brave new world of ubiquitous parallel computing.

0.3 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.

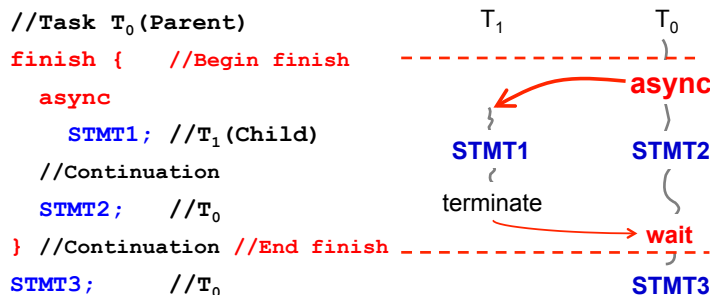


Figure 4: A example code schema with async and finish constructs

1 Task-level Parallelism

1.1 Task Creation and Termination (Async, Finish)

1.1.1 Async notation for Task Creation

As mentioned earlier in Section 0.1, the statement “**async** $\langle stmt \rangle$ ” causes the parent task to create a new child task to execute $\langle stmt \rangle$ *asynchronously* (i.e., before, after, or in parallel) with the remainder of the parent task. Figure 4 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.

All algorithm and code examples in the module handouts should be treated as “pseudo-code”, which is different from the actual notation 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 are equivalent by omitting the **async** keyword in each case. An interesting point to note in Example 3 is that a method can return even though its child **async** tasks might not have completed. It’s only the enclosing **finish** that awaits the completion of all the **async**’s. If you need to ensure that all **async**’s are completed before a method returns, then the method body should be enclosed in a **finish**.

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 5, typically one worker per processor core. Worker threads are allocated by the Operating System (OS). By creating one thread per core, we limit


```

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 i = 0; i < A.length; i++)
4     async { A[i] = B[i] + C[i]; } // value of i is copied on entry to async
5
6 // Example 2: execute iterations of a while loop in parallel
7 p = first;
8 while ( p != null ) {
9     async { p.x = p.y + p.z; } // value of p is copied on entry to async
10    p = p.next;
11 }
12
13 // Example 3: Example 2 rewritten as a recursive method
14 static void process(T p) {
15     if ( p != null ) {
16         async { p.x = p.y + p.z; } // value of p is copied on entry to async
17         process(p.next);
18     }
19 }
20
21 // Example 4: execute method calls in parallel
22 async left_s = quickSort(left);
23 async right_s = quickSort(right);

```

Listing 1: Example usages of async

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 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 4. A *continuation*⁵ represents a potential suspension point for a task, which (as shown in in Figure 4) 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 5 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 Java methods. Listing 2 summarizes the rules for accessing local variables across *async* boundaries. For convenience, as shown in Rules 1 and 2, an inner *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). Note that an inner *async* is not permitted to modify a local variable declared in an outer scope (Rule 3) because doing so will violate the assumption that local variables are private to each task. If needed, you can work around this constraint by replacing the local variable by a static or instance field, since fields can be shared among tasks.

1.1.2 Finish notation for Task Termination

Let us look back at the code schema in Figure 4. It included a **finish** statement, which was defined as follows in Section 0.1: the statement “**finish** $\langle stmt \rangle$ ” causes the parent task to execute $\langle stmt \rangle$ and then wait

⁵This use of “continuation” is related to, but different from continuations in functional programming languages.

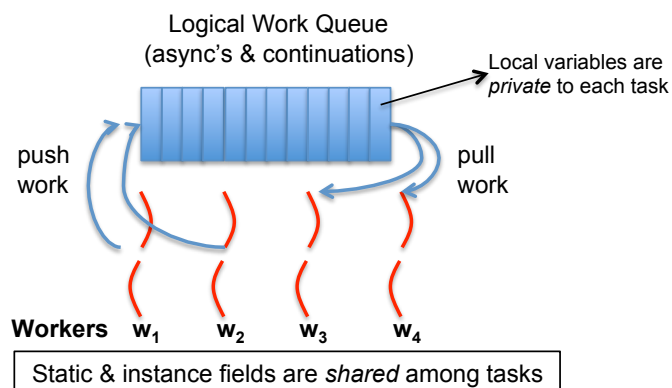


Figure 5: Scheduling an parallel program on a fixed number of workers. (Figure adapted from [11].)

```
1 // Rule 1: an inner 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: an inner async may read the value of any 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 = 3; // This assignment is not seen by the above async
9
10 // Rule 3: an inner 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

until all `async` tasks created within $\langle stmt \rangle$ have completed.

Thus, the `finish` statement in Figure 4 is used by task T_0 to ensure that child task T_1 has completed executing `STMT1` before T_0 executes `STMT3`. 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`. This waiting is also referred to as a *synchronization*. There is an implicit `finish` scope surrounding the body of the main program, so program execution will only end after all `async` tasks have completed. 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 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. Why do you need `finish` constructs beyond the implicit `finish` in `main()`? Because a parent (ancestor) task needs to be sure that its child (descendant) tasks have completed before it can access the results that they computed. This could occur if `STMT3` in Figure 4 used a value computed by `STMT1`. You saw a concrete example of this case in the Two-way Parallel ArraySum program discussed in Section 0.1 where the `finish` was used to ensure that `sum1` and `sum2` were fully computed before their values were used to compute `sum`.

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 6. It reveals which pairs of statements can potentially execute in parallel with each other. For example, task A_2 can potentially execute in parallel with tasks A_3 and A_4 since `async` A_2 was launched before entering the `finish` F_2 , which is the Immediately Enclosing Finish for A_3 and A_4 . However, Part 3 of Task A_0 cannot execute in parallel with tasks A_3 and A_4 since it is performed after `finish` F_2 is completed.

1.1.3 Exception Semantics

Besides termination detection, the `finish` statement plays an important role with regard to exception semantics. If any `async` throws an exception, then its IEF statement throws a *MultiException* [6] formed from the collection of all exceptions thrown by all `async`’s in the IEF. Consider Listing 5, which is Listing 4 extended with two try-catch blocks. If `async` task A_3 throws an exception, the exception will not be caught as `e1` (line 8) but instead as `e2` (line 11), since the catch block for `e2` encloses the `finish` statement containing A_2 . The catch block for `e1` only catches any errors that may occur in the creation of an `async` (*e.g.*, an `OutOfMemoryError`), not the execution of the `async`.

If the program has no try-catch blocks, then an error message is displayed on the console upon program exit.

```
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 i = 0; i < A.length; i++) async A[i] = B[i] + C[i];
7 System.out.println(A[0]);
8
9 // Example 1: Correct parallel version
10 finish for (int i = 0; i < A.length; i++) async A[i] = B[i] + C[i];
11 System.out.println(A[0]);
12
13 // Example 2: Sequential version
14 p = first;
15 while ( p != null ) {
16     p.x = p.y + p.z; p = p.next;
17 }
18 System.out.println( first.x );
19
20 // Example 2: Incorrect parallel version
21 p = first;
22 while ( p != null ) {
23     async { p.x = p.y + p.z; }
24     p = p.next;
25 }
26 System.out.println( first.x );
27
28 // Example 2: Correct parallel version
29 p = first;
30 finish while ( p != null ) {
31     async { p.x = p.y + p.z; }
32     p = p.next;
33 }
34 System.out.println( first.x );
```

Listing 3: 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

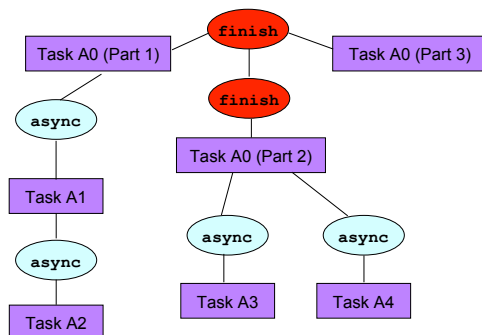


Figure 6: Finish-async nesting structure for code fragment in Listing 4

```
1 finish { // F1
2   // Part 1 of Task A0
3   async {A1; async A2;}
4   try {
5     finish { // F2
6       // Part 2 of Task A0
7       try { async A3; }
8       catch (Throwable e1) { }; // will not catch exception in A3
9       async A4;
10    }
11  } catch (Throwable e2) { }; // will catch exception in A3
12  // Part 3 of Task A0
13 }
```

Listing 5: Extension of Listing 4 with try-catch blocks

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 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 [11]:
 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 6 and its Computation Graph shown in Figure 7. 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 7. 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 7 shows that all of nodes $v3 \dots v15$ can potentially execute in parallel with node $v16$ because there is no directed path in the CG from $v16$ to any node in $v3 \dots v15$ or vice versa.

It is also important to observe that the CG in Figure 7 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 // 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 6: Example program

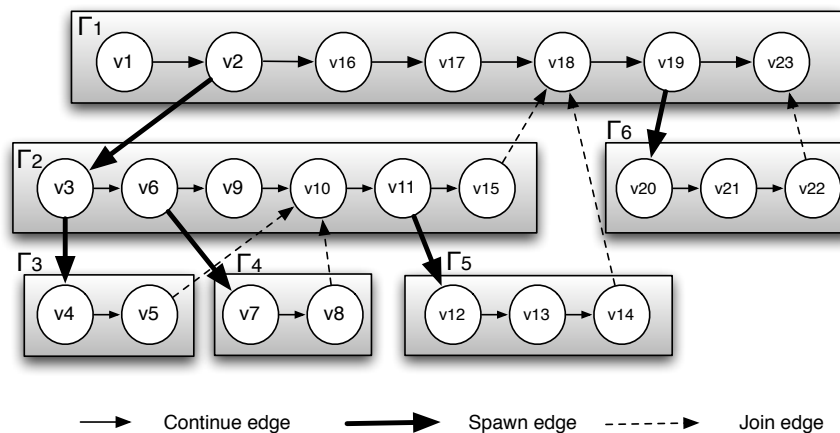


Figure 7: Computation Graph G for example program in Listing 6

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 7. 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 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*: $t_p \geq \text{WORK}(G)/P$. It is not possible for a schedule to complete in time less than $\text{WORK}(G)/P$ because that's how long it would take if all the work was perfectly divided among P processors.
2. *Critical path bound*: $t_p \geq \text{CPL}(G)$. It is not possible for a schedule to complete in time less than $\text{CPL}(G)$ 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 $t_p \geq \max(\text{WORK}(G)/P, \text{CPL}(G))$. 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:

- $t_1 = \text{WORK}(G)$. 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.
- $t_\infty = \text{CPL}(G)$. Any greedy scheduler executing with an unbounded (infinite) number of processors must complete its execution with time = $\text{CPL}(G)$, since all nodes can be scheduled as early as possible.
- $t_P \leq t_1/P + t_\infty = \text{WORK}(G)/P + \text{CPL}(G)$. This is a classic result due to Graham [10]. 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. 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:

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

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, $\text{WORK}(G)/P$ and $\text{CPL}(G)$. 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 $\text{Speedup}(P) = t_1/t_P$. $\text{Speedup}(P)$ is the factor by which the use of P processors speeds up

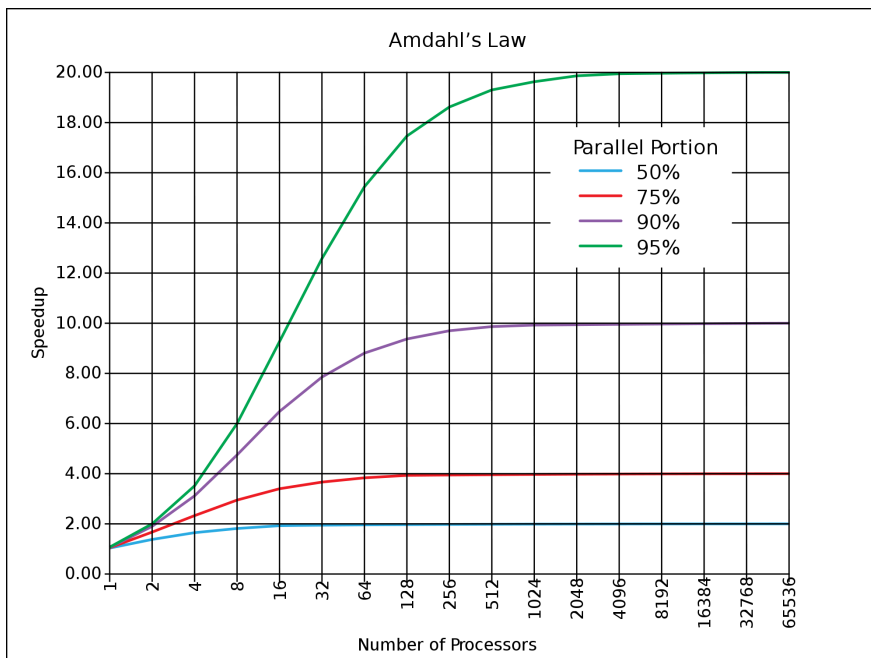


Figure 8: Illustration of Amdahl's Law (source: http://en.wikipedia.org/wiki/Amdahl's_law)

execution time relative to 1 processor, for a fixed input size. For ideal executions without overhead, $1 \leq \text{Speedup}(P) \leq P$. The term *linear speedup* is used for a program when $\text{Speedup}(P) = k \times P$ 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 $\text{Speedup}(P) \leq 1/q$. 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 $t_P \geq \text{CPL}(G)$, where t_P is the execution time of computation graph G on P processors and $\text{CPL}(G)$ is the *critical path length* of graph G . If fraction q of $\text{WORK}(G)$ is sequential, it must be the case that $\text{CPL}(G) \geq q \times \text{WORK}(G)$. Therefore, $\text{Speedup}(P) = t_1/t_P$ must be $\leq \text{WORK}(G)/(q \times \text{WORK}(G)) = 1/q$ since $t_1 = \text{WORK}(G)$ for greedy schedulers.

The consequence of Amdahl's Law is illustrated in Figure 8. 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 $20\times$ 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 [13] 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*.

1.6 Parallel Quicksort

In this section, we introduce the well-known sequential Quicksort algorithm, and discuss three approaches to parallelizing this algorithm.

1.6.1 Parallelizing Quicksort: Approach 1

Quicksort is a classical sequential sorting algorithm introduced by C.A.R. Hoare in 1961 [16], and is still very much in use today. There are a number of factors that have contributed to its longevity. First, it is simple to implement. Second, even though it has a worst case $O(n^2)$ execution time complexity, it executes sequentially in $O(n \log n)$ time in practice. Third, it is an “in place” sorting algorithm *i.e.*, it does not need the allocation of a second copy of the array. For many computers today, the “in place” characteristic may not be as important now as it was 50 years ago. Finally, Quicksort epitomizes the *divide-and-conquer* paradigm of computer algorithms, which naturally lend themselves to parallel implementations.

As a historical note, Figure 9 contains a facsimile of the exact description of the original Quicksort algorithm from 1961 [16]. (Note that the comment section is longer than the code for the algorithm!) A call to *quicksort*(*A*,*M*,*N*) has the effect of sorting array elements $A[M \dots N]$ in place. The **value** keyword indicates that integer parameters *M* and *N* will not be changed (as in Java’s **final** declaration). The body of the procedure is enclosed in the keywords, **begin** and **end**. The divide and conquer stages in this algorithm are captured as follows:

Divide: The call to *partition*(*A*,*M*,*N*,*I*,*J*) partitions the subarray, $A[M \dots N]$ into disjoint sub-arrays, $A[M \dots J]$ and $A[I \dots N]$ (where $M \leq J < I \leq N$) such that all elements in $A[M \dots J]$ are guaranteed to be \leq all elements in $A[J + 1 \dots I - 1]$, which in turn are guaranteed to be \leq all elements in $A[I \dots N]$. A facsimile of the original specification of the *partition* procedure⁶ is included in Figure 10. It uses a randomly selected element in $A[M \dots N]$ to serve as the *pivot* for partitioning the array.

Conquer: The two recursive calls to *quicksort* operate on sub-arrays $A[M \dots J]$ and $A[I \dots N]$. The recursion terminates when $M < N$ is false *i.e.*, when $M \geq N$. There is no combining to be done after the two sub-arrays have been sorted, since they can just be left in place to obtain a sorted version of $A[M \dots N]$.

The divide-and-conquer structure of the *quicksort* procedure naturally leads to parallel execution of the recursive calls, akin to the recursive parallel algorithm for the Array Sum problem that you studied earlier. In particular, the two recursive calls to *quicksort* can be executed in parallel as shown below in Algorithm 3. We refer to this parallel algorithm as Approach 1.

Algorithm 3: ParallelQuickSort (Approach 1)

Input: Array of comparable objects, *A*; start of range to be sorted, *M*; end of range to be sorted, *N*.

Output: Updated version of *A*, in which elements $M \dots N$ are sorted.

// Main program calls **finish** { **ParallelQuickSort**(*A*, *M*, *N*)}

if $M < N$ **then**

 // Call sequential partition() algorithm in Figure 10.

I, *J* \leftarrow *partition*(*A*, *M*, *N*);

async{

 | **ParallelQuickSort**(*A*, *M*, *J*)

 };

async{

 | **ParallelQuickSort**(*A*, *I*, *N*)

 };

⁶Modern implementations of this algorithm usually replace the **go to** statements by **while**, **continue**, and **break** statements.

```

procedure quicksort (A,M,N); value M,N;
    array A; integer M,N;
comment Quicksort is a very fast and convenient method of
sorting an array in the random-access store of a computer. The
entire contents of the store may be sorted, since no extra space is
required. The average number of comparisons made is  $2(M-N) \ln(N-M)$ ,
and the average number of exchanges is one sixth this
amount. Suitable refinements of this method will be desirable for
its implementation on any actual computer;
begin    integer I,J;
        if M < N then begin partition (A,M,N,I,J);
            quicksort (A,M,J);
            quicksort (A, I, N)
        end
end    quicksort

```

Figure 9: Algorithm 64: Quicksort (source: [16])

What is the parallel execution time complexity of this approach to parallelization? For simplicity, let us perform a big-O analysis assuming *perfect pivoting*, i.e., that each call to *partition()* is able to divide the input array into two sub-arrays of equal size. In that case, the first call to *partition()* takes $O(n)$ time, the next two calls execute in parallel and take $O(n/2)$ time each, the next four calls execute in parallel and take $O(n/4)$ time each, and so on. Thus, the *critical path length* for this algorithm is $CPL(n) = O(n) + O(n/2) + O(n/4) + \dots = O(n)$, and the total computation performed by the algorithm is $WORK(n) = O(n \log n)$. The computation graph structure in Figure 11 illustrates the derivation of $WORK(n)$ and $CPL(n)$ for Approach 1. With $O(n)$ processors, the best parallel speedup that we can expect from this algorithm, even with an unbounded number of processors, is $WORK(n)/CPL(n) = O(\log n)$, which is highly limiting.

1.6.2 Parallelizing Quicksort: Approach 2

We studied Approach 1 to parallelizing Quicksort in Section 1.6, by using `async` statements to parallelize recursive calls. An alternate approach is to leave the calls to *quicksort()* unchanged and instead focus on *parallelizing* the computation within each call to *partition()*. This can be very challenging if we use the code in Figure 10 as a starting point, since that algorithm performs sequential incremental updates to the array using calls to `exchange()`. Instead, we can exploit parallelism by making a copy of the sub-array in a new location, identifying elements that are $<$, $=$, or $>$ the pivot value, and then packing elements from the copy back into the original sub-array as shown in Listing 7.

The program in Listing 7 has more detail than a high-level algorithmic description, but it is informative to see what computations are necessary to perform this kind of parallelization. The `forall` statement in lines 8–13 computes in parallel the values of `lt[k]`, `eq[k]`, and `gt[k]`, each of which is set to 0 or 1 depending on whether $A[M+k] < A[M+\text{pivot}]$, $A[M+k] == A[M+\text{pivot}]$ or $A[M+k] > A[M+\text{pivot}]$ respectively. The `forall` also copies sub-array $A[M \dots N]$ into `buffer[0 .. N - M + 1]`. We can use the parallel prefix sum algorithm described in Section 1.6.5 to obtain a parallel implementation of the `computePrefixSums()` procedure called in lines 14, 15, and 16. We assume that the `computePrefixSums()` procedure replaces the input `int[]` array by the prefix sums, and also returns an `int` value containing the total sum of the input array (same as the last element in the prefix sum). The three prefix sums can also be combined into a single scan operation when using *segmented scans* [3].

The `forall` loop in lines 17–21 permutes the elements in sub-array $A[M \dots N]$ so as to pack the sub-arrays containing $<$, $=$, or $>$ values relative to the pivot value. For the $<$ case, line 18 copies `buffer[k]` into $A[M + \text{lt}[k] - 1]$. For example, if only one element, $A[M + k]$ satisfies this condition, then its prefix sum is `lt[k]=1`, and line 18 assigns `buffer[k]` (which contains a copy of the original $A[M + k]$) to $A[M]$. Similarly,

Description of procedure partition(A,M,N,I,J) from [15]:

```

comment  I and J are output variables, and A is the array (with
subscript bounds M:N) which is operated upon by this procedure.
Partition takes the value X of a random element of the array A,
and rearranges the values of the elements of the array in such a
way that there exist integers I and J with the following properties:
      M ≤ J < I ≤ N provided M < N
      A[R] ≤ X for M ≤ R ≤ J
      A[R] = X for J < R < I
      A[R] ≥ X for I ≤ R ≤ N
The procedure uses an integer procedure random (M,N) which
chooses equiprobably a random integer F between M and N, and
also a procedure exchange, which exchanges the values of its two
parameters;
begin   real X; integer F;
        F := random (M,N); X := A[F];
        I := M; J := N;
up:     for I := I step 1 until N do
        if X < A [I] then go to down;
        I := N;
down:   for J := J step -1 until M do
        if A[J]<X then go to change;
        J := M;
change: if I < J then begin exchange (A[I], A[J]);
        I := I + 1; J := J - 1;
        go to up
        end
else   if I < F then begin exchange (A[I], A[F]);
        I := I + 1
        end
else   if F < J then begin exchange (A[F], A[J]);
        J := J - 1
        end;
end   partition

```

Figure 10: Algorithm 63: Partition (source: [15])

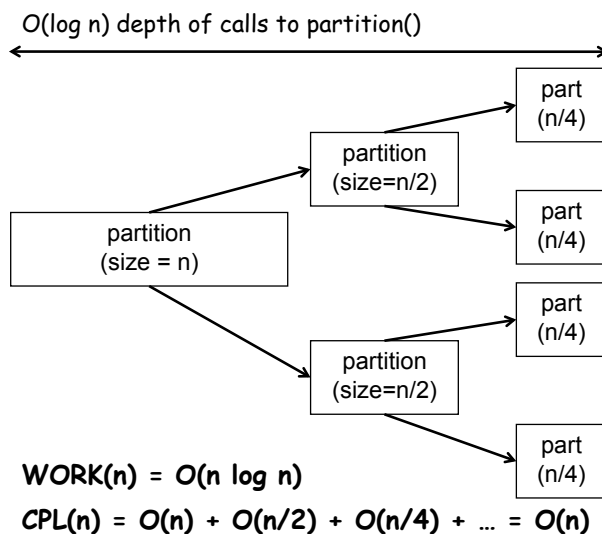


Figure 11: Computation Graph for Approach 1

for the = case, line 19 copies `buffer[k]` into `A[M + ltCount + eq[k] - 1]`, and for the > case, line 20 copies `buffer[k]` into `A[M + ltCount + eqCount + gt[k] - 1]`. Finally, the procedure returns `[I, J]` based on the three cases in lines 22, 23, and 24.

What is the parallel execution time complexity of this implementation of `partition()`? The total computation performed is $WORK_{\text{partition}}(n) = O(n)$, and the critical path length is $CPL_{\text{partition}}(n) = O(\log n)$ due to the calls to `computePrefixSums()`. When combined with sequential calls to `quicksort()` as in [15], the total work will be $O(n)$ for the first call to `partition()`, $2 \times O(n/2) = O(n)$ for the 2 calls to `partition()` with sub-arrays of size $n/2$, $4 \times O(n/4) = O(n)$ for the 4 calls to `partition()` with sub-arrays of size $n/4$, and so on. Thus, the total work is still $WORK(n) = O(n \log n)$ as in Approach 1. The critical path length can be defined by the recurrence, $CPL(n) = CPL_{\text{partition}}(n) + 2CPL(n/2) = \log n + 2CPL(n/2)$, which can be solved as $CPL(n) = O(n)$. The computation graph structure in Figure 12 illustrates the derivation of $WORK(n)$ and $CPL(n)$ for Approach 2. Thus, Approach 2 has the same low level of ideal parallelism as Approach 1.

1.6.3 Parallelizing Quicksort: Approach 3

Approaches 1 and 2 considered two complementary strategies for exploiting parallelism in `quicksort()` — execution of the recursive calls as parallel tasks, and exploitation of parallelism with a single call of `partition()` respectively. Can we get the best of both worlds? What if we combine the parallel version of `quicksort()` in Algorithm 3 with the parallel version of `partition` in Listing 7? The total computation remains the same, $WORK(n) = O(n \log n)$. The computation graph structure is shown in Figure 13. It shows that the critical path length is of the form $\log(n) + \log(n/2) + \log(n/4) + \dots = O(\log^2 n)$. The computation graph structure in Figure 13 illustrates the derivation of $WORK(n)$ and $CPL(n)$ for Approach 3. Thus, the ideal parallelism for this approach is $WORK(n)/CPL(n) = (n \log n)/\log^2 n = O(n/(\log n))$, which is much better than that of Approach 1 and Approach 2.

1.6.4 Parallel Array Sum Reduction using Async and Finish

Let's start with the sequential Array Sum algorithm that you saw in Algorithm 1 (Section 0.1). You already know how to decompose this computation into two tasks for two-way parallelism. However, a more general parallelization structure can be obtained by using the *reduction tree* schema shown in Figure 14. This schema is designed to allow elements of the original array to be overwritten so that the final sum is written in `X[0]`. If this poses a problem, a copy of the original array should be saved before executing this algorithm.

```

1  static point partition(int [] A, int M, int N) {
2  int I, J;
3  final int pivot = M + new java.util.Random().nextInt(N-M+1);
4  final int [] buffer = new int [N-M+1];
5  final int [] ltOrig = new int [N-M+1]; final int [] lt = new int [N-M+1];
6  final int [] gtOrig = new int [N-M+1]; final int [] gt = new int [N-M+1];
7  final int [] eqOrig = new int [N-M+1]; final int [] eq = new int [N-M+1];
8  forall(point [k] : [0:N-M]) {
9  ltOrig[k] = (A[M+k] < A[pivot] ? 1 : 0); lt[k] = ltOrig[k];
10 eqOrig[k] = (A[M+k] == A[pivot] ? 1 : 0); eq[k] = eqOrig[k];
11 gtOrig[k] = (A[M+k] > A[pivot] ? 1 : 0); gt[k] = gtOrig[k];
12 buffer[k] = A[M+k];
13 }
14 final int ltCount = computePrefixSums(lt); //update lt with prefix sums
15 final int eqCount = computePrefixSums(eq); //update eq with prefix sums
16 final int gtCount = computePrefixSums(gt); //update gt with prefix sums
17 forall(point [k] : [0:N-M]) {
18 if(ltOrig[k]==1) A[M+lt[k]-1] = buffer[k];
19 else if(eqOrig[k]==1) A[M+ltCount+eq[k]-1] = buffer[k];
20 else A[M+ltCount+eqCount+gt[k]-1] = buffer[k];
21 }
22 if(M+ltCount == M) return [M+ltCount+eqCount, M+ltCount];
23 else if(M+ltCount == N) return [M+ltCount, M+ltCount - 1];
24 else return [M+ltCount+eqCount, M+ltCount - 1];
25 } // partition

```

Listing 7: Parallel algorithm for procedure partition()

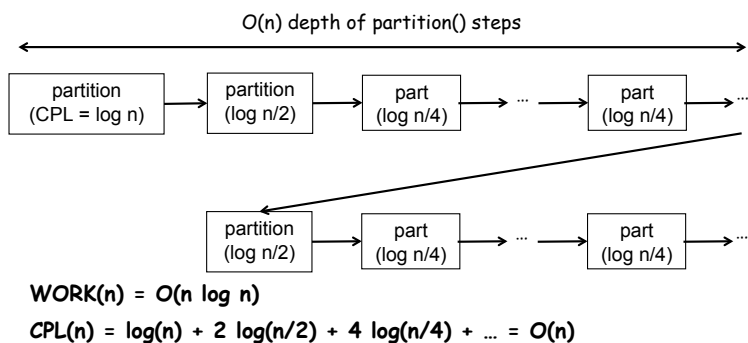


Figure 12: Computation Graph for Approach 2

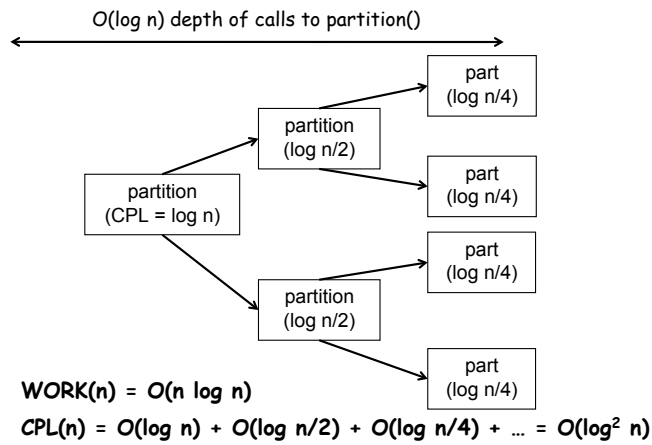


Figure 13: Computation Graph for Approach 3

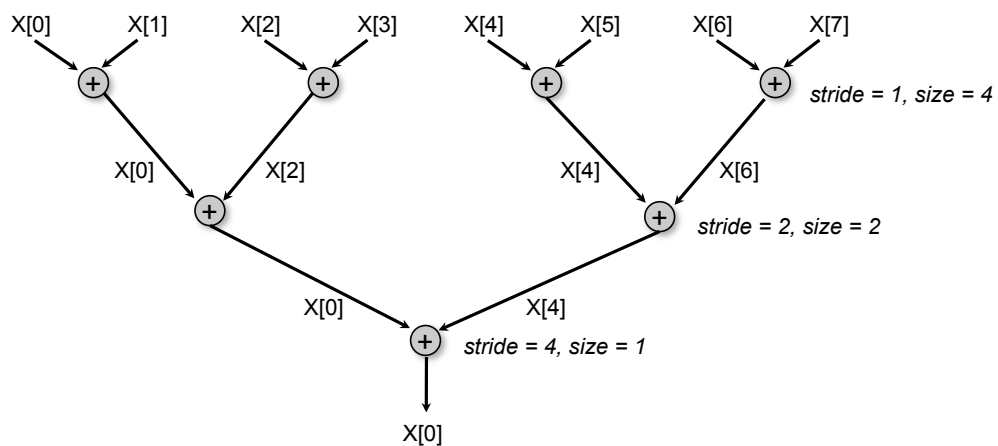


Figure 14: Reduction tree schema for computing ArraySum in parallel, assuming X.length=8. Note that elements of the input array are over-written with partial sums, and the final sum is written in X[0].


```

1  finish { // stride = 1, size = 4 parallel additions
2      async X[0]+=X[1]; async X[2]+=X[3]; async X[4]+=X[5]; async X[6]+=X[7]; }
3  finish { // stride = 2, size = 2 parallel additions
4      async X[0]+=X[2]; async X[4]+=X[6]; }
5  finish { // stride = 4, size = 1 parallel additions
6      async X[0]+=X[4]; }
    
```

Listing 8: Async-Finish code to implement reduction tree schema in Figure 14

How can the schema shown in Figure 14 be converted to executable code? By designing a parallel program that generates a computation graph that satisfies the dependences in Figure 14. One such program for the $X.length = 8$ case is shown in Listing 8:

Looking at the dependence structure in Figure 14, we see that the computation can be decomposed into three stages with 4 parallel additions in the first stage, 2 parallel additions in the second stage, and 1 addition in the third stage. The code in Listing 8 follows this structure by using three `finish` scopes for the three stages. The *stride* in each stage is the distance between the subscripts of the two array element operands for each addition in the stage. Thus, the stride values for the three stages are 1, 2 and 4 respectively. The *size* of each stage is the number of parallel additions that can be performed in the stage. The sizes of the three stages are 4, 2, and 1 respectively. Note that $stride * size = X.length / 2 = 4$, in all stages. Though the last stage just contains a single addition, $X[0] += X[4]$, we treat it as a degenerate case of parallel additions for convenience.

After verifying that Listing 8 correctly implements the schema in Figure 14 for the $X.length = 8$ case, the next question is: how do we implement a program that works in the general case for an array of arbitrary size? The answer can be found in Listing 9. The stages are defined by sequential iterations of the outer `for` loop starting at line 1. This loop initializes $stride = 1$ and doubles $stride$ for each stage, terminating when $stride$ becomes $\geq X.length$. The *size* value for each stage can be computed by using the observation that $stride * size = X.length / 2$. Thus, we set $size = X.length / (2 * stride)$, using the `ceilDiv` function to round up. (`ceilDiv(x, y)` returns $\lceil x/y \rceil$, where $\lceil \cdot \rceil$ represents the *ceiling function*, which rounds up.) The parallel additions in each stage can then be implemented using the `finish-for-async` pattern in line 4. (This pattern occurs commonly in practice, and can be replaced by a `forall` loop which you will learn later in the course.) Each iteration performs the statement, $X[2^i * stride] += X[(2^{i+1}) * stride]$; in line 8, where i iterates from 0 to $size - 1$. The `if` condition in line 5 ensures that the addition is only performed when both operands are available. It is needed to handle the general case when $X.length$ may not be a power of 2. (An alternate approach is to use a recursive method to implement a parallel divide-and-conquer algorithm to compute the sum of the elements of an array.)

As discussed in class, the *overhead* of creating an `async` on a real machine will be much larger than the time taken to execute the instructions in its body (lines 6 and 8). You will learn about these overhead considerations later in the course. For now, let us analyze the performance of this code using abstract performance metrics. For simplicity let's assume that each execution of line 8 takes 1 unit of time (since line 8 contains the actual addition operation), and assume that everything else has zero cost. If $n = X.length$ is the size of the array, then the total *WORK* (number of array element additions) performed by the program in Listing 9 is $n - 1$ or $O(n)$. Also, the critical path length (*CPL*) for this program is $\lceil \log_2(n) \rceil$ or $O(\log(n))$. Thus, the ideal parallelism is $O(n / \log(n))$, which shows that the available parallelism increases as n , the size of the input array, increases.

As a final note, let us determine how many processors could best be used to execute this program (ignoring overhead issues). If we have p processors, we could have each processor compute partial sums for n/p array elements in parallel, and then use the program in Listing 9 to sum the resulting p values. In this case, the total work will be $O(n)$, the critical path length will be $O(n/p + \log p)$, and the resulting parallelism will be $O(n) / O(n/p + \log p)$. If we set $p = n / (\log n)$, then the resulting parallelism will be $O(n) / O(\log n + \log n -$

```

1  for ( int stride = 1; stride < X.length ; stride *= 2 ) {
2      // Compute size = number of additions to be performed in stride
3      int size=ceilDiv(X.length,2*stride); //Divide X.length/2*stride & round up
4      finish for(int i = 0; i < size; i++)
5      async {
6          if ( (2*i+1)*stride < X.length ) //Check if both operands are available
7              // This condition will always be true when X.length is a power of 2
8              X[2*i*stride]+=X[(2*i+1)*stride]; //Array subscripts differ by stride
9      } // finish-for-async
10 } // for
11
12 // Divide x by y, round up to next largest int, and return result
13 static int ceilDiv(int x, int y) { return (x+y-1) / y; }

```

Listing 9: Async-Finish code to implement schema in Figure 9 for arbitrarily sized arrays

```

1  // Copy input array A into output array X
2  X = new int[A.length]; System.arraycopy(A,0,X,0,A.length);
3  // Update array X with prefix sums
4  for (int i=1 ; i < X.length ; i++ ) X[i] += X[i-1];

```

Listing 10: Sequential Prefix Sum

$\log \log n = O(n/(\log n)) = O(p)$, which matches the number of processors. A larger value for p (e.g., $p = n$) will lead to available parallelism that is less than p (e.g., parallelism = $n/(\log n)$). In this sense, we can say that the algorithm in Listing 9 is *optimal* for $p = n/(\log n)$ or fewer processors.

1.6.5 Parallel Prefix Sum

Let us now consider an extension of the Array Sum computation known as *Prefix Sum*. Prefix Sum computations are useful building blocks in a wide range of algorithms such as team standings, index of last occurrence, lexical string comparison, and multi-precision arithmetic [3, 18].

The Prefix Sum problem can be stated as follows:

Given an input array A , compute an output array X as follows (where $n = A.length$):

$$X[i] = \sum_{0 \leq j \leq i} A[j] \quad (1)$$

Thus, each output element, $X[i]$, is the sum of the *prefix* of array A consisting of elements $A[0 \dots i]$. This computation is also referred to as a *scan* operation [3]. In general, the sum operation in Equation 1 can be replaced by any binary associative operation.

What is the most efficient sequential algorithm for this problem? The problem statement in 1 may suggest that $O(n^2)$ additions are required because $X[0]$ requires 0 additions, $X[1]$ requires 1 addition, $X[2]$ requires 2 additions, and so on. However, it is easy to see that the sequential Prefix Sum program in Listing 10 computes the desired prefix sums in $O(n)$ time.

The program in Listing 10 appears to be inherently sequential since output $X[i]$ depends on output $X[i-1]$. How can we execute it in parallel? One approach could be to invoke the parallel Array Sum reduction algorithm from Section 1.6.4 independently for each element in output array X as follows:

Recall that the parallel execution time for a single sum reduction operation was $O(\log n)$. Since all $X[i]$ elements in Listing 11 are computed in parallel, the parallel execution time for computing all output elements

```

1  finish {
2      for (int i=0 ; i < X.length ; i++ )
3          async X[i] = computeSum(A, 0, i);
4  }
```

Listing 11: Inefficient Parallel Prefix Sum program

is also $O(\log n)$. However, if you count each addition as taking 1 unit of time, then the total *WORK* performed by the program in Listing 11 is $\sum_{0 \leq i < n} i = O(n^2)$. This means that you will need at least $O(n^2/(\log n))$ processors to perform the computation in $O(\log n)$ time, which is highly inefficient considering that the sequential algorithm takes $O(n)$ time. For optimality, you need a parallel algorithm that ideally takes $O(\log n)$ time on $O(n/(\log n))$ processors, like the parallel algorithm for Sum Reduction.

To develop such a parallel algorithm, the key observation is that each prefix sum consists of some number of partial sums that can be obtained by grouping together the terms in the prefix into powers of 2. For example, the expression for $X[6]$ can be rewritten as

$$\begin{aligned} X[6] &= A[0] + A[1] + A[2] + A[3] + A[4] + A[5] + A[6] \\ &= (A[0] + A[1] + A[2] + A[3]) + (A[4] + A[5]) + A[6] \end{aligned}$$

We then observe that there will be at most $O(\log n)$ terms in () parentheses for each such $X[i]$ and that these terms can be reused across different $X[i]$ values. Further, these terms are naturally computed as partial sums in the parallel Sum Reduction algorithm that you learned earlier.

Figure 15 shows the “upward sweep” for the Parallel Prefix algorithm, following the reduction tree schema studied earlier. The rules to be followed by each non-leaf node in this reduction tree are:

1. Receive values from left child and right child.
2. Store the left child’s value in the node’s “box”. The left child’s value is the sum of all array elements that are descendants of the left child, and can be reused as a common term in the partial sums for elements that are descendants of the right child.
3. Compute the sum of the left child’s and right child’s values, and send that value up to the parent.

For the example in Figure 15, the input array is [3, 1, 2, 0, 4, 1, 1, 3]. If you look at the shaded box in the figure, it receives the values 3 and 1 from its children, and propagates $3 + 1 = 4$ to its parent. In addition, it stores 3 in the box. All other boxes are filled in the same way. The critical path length for this sweep is the height of the reduction tree, which is $O(\log n)$. This sweep can be performed in $O(\log n)$ time using $(n/(\log n))$ processors, as discussed earlier.

Figure 16 shows the “downward sweep” necessary to complete the prefix sum calculation. The rules to be followed for each box are:

1. Receive value from its parent. This value contains the sum of the array elements located to the left of the subtree rooted at the current box. Thus, the root box receives a zero (identity) value to start with.
2. Send value received from its parent to its left child, since its left child’s subtree has the same prefix as it does.
3. Compute the sum of the value receive from its parent and the value stored in its box from the upward sweep, and send that value to its right child. Recall that the value stored in its box is the sum of the elements belong to its left child’s subtree. By adding these two values, we obtain the sum of the array elements located to the left of the subtree rooted at the right child.

Upward sweep

1. Receive values from children
2. Store left value in box
3. Send left+right value to parent

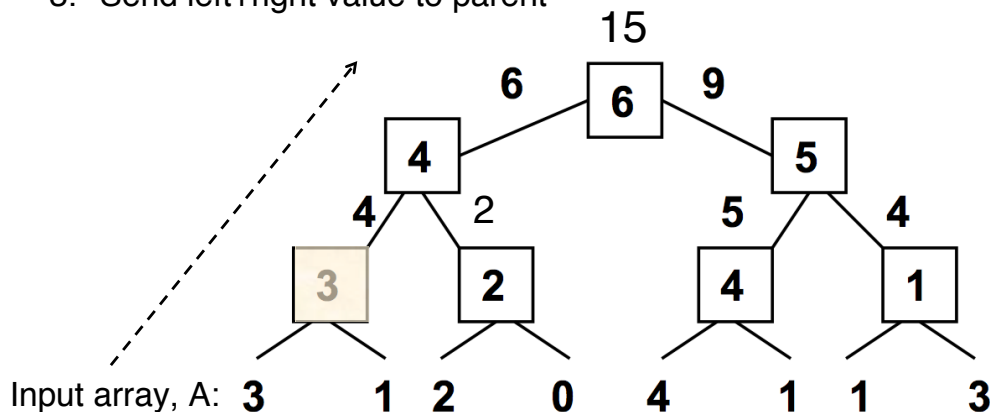


Figure 15: Upward sweep for Parallel Prefix algorithm

After the downward sweep has completed, each leaf will contain the sum of the array element to its left (also known as the *pre-scan*). To compute the prefix sum, all that remains to be done is to add the pre-scan value to the array element value at that position, which can be performed in $O(1)$ time in parallel on $O(n)$ processors or $O(\log n)$ time in parallel on $O(n/(\log n))$ processors.

If you look at the shaded box in Figure 16, it receives the value 6 from its parent which is the sum of all the elements to the left of the subtree rooted at the shaded box (see input array in Figure 15). This value of 6 is propagated to the left child, whereas the value of $6 + 5 = 11$ is propagated to the right child. As with the upward sweep, the downward sweep can be performed in $O(\log n)$ time using $(n/(\log n))$ processors.

Downward sweep

1. Receive value from parent (root receives 0)
 2. Send parent's value to left child
 3. Send parent+box value to right child
- Add $X[i]$ to get prefix sum

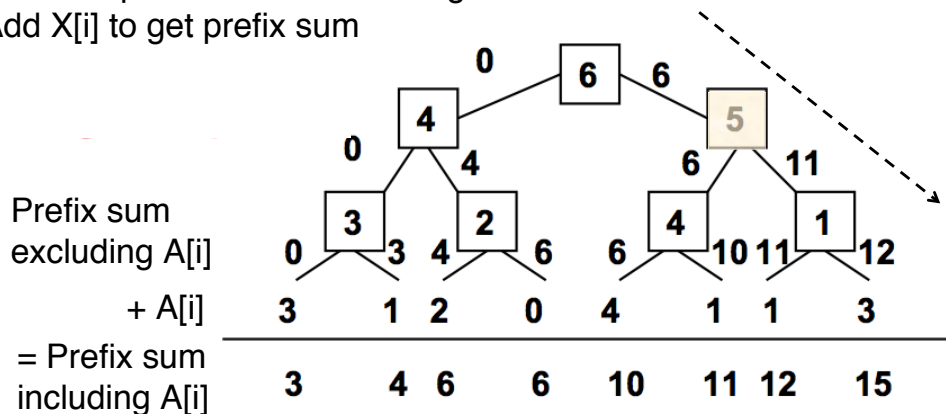


Figure 16: Downward sweep for Parallel Prefix algorithm

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>`⁷ is a reference to a *future* object *i.e.*, a container for a return value of type `T` from an `async` task. An important parallel programming rule is that *all variables of type `future<T>` should be declared with a `final` modifier*, thereby ensuring that the value of the reference cannot change after initialization. This rule ensures that the synchronization associated with reading future values cannot lead to a deadlock cycle.
- 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 12 revisits the two-way parallel array sum example discussed in Section 0.1, but using *future tasks* instead of regular `async`'s. There are two variables of type `future<int>` in this example, `sum1` and `sum2`, and both are declared as `final`. Each future task can potentially execute in parallel with its parent, just like regular `async`'s. However, unlike regular `async`'s, there is no `finish` construct needed for this example since the parent task T_1 , performs `sum1.get()` to wait for future task T_2 and `sum2.get()` to wait for future task T_3 .

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

⁷“`future`” is a pseudocode keyword, and will need to 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 final 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 final 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 12: Two-way Parallel ArraySum using Future Tasks

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 [14].

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 12, 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 17. 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 declared as final?

In the previous section, we stated the rule that all variables containing references to future objects should be declared as final. This means that the variable cannot be modified after initialization. To motivate this rule, consider the buggy program example in Listing 13. *WARNING: this is an example of bad parallel programming practice that you should not attempt!*

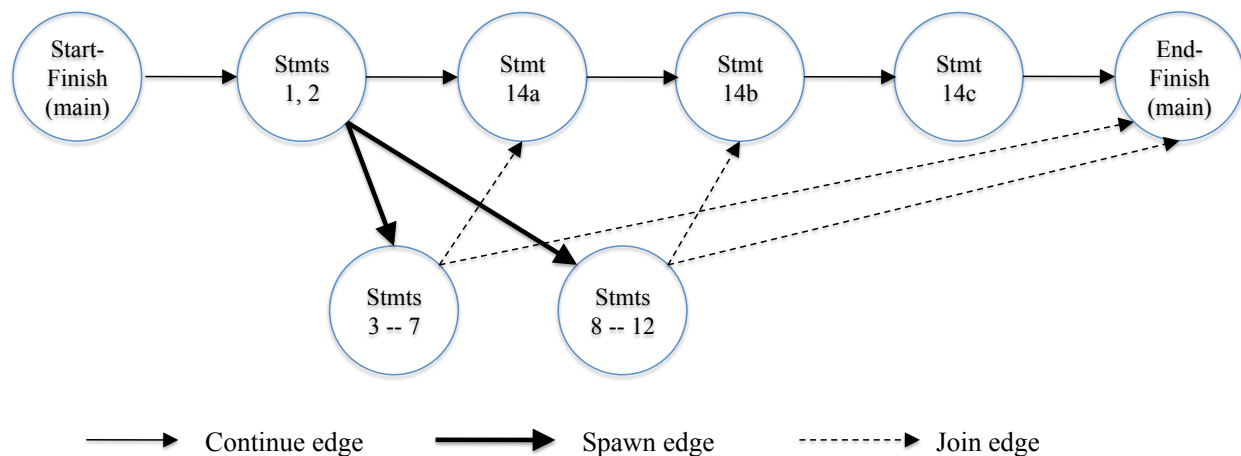


Figure 17: Computation Graph G for example parallel program in code:TwoParArraySumFuture, with statement 14 split into statements 14a, 14b, 14c

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 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 declared to be 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 18 shows Computation Graph $G3$ that cannot be generated using only `async` and `finish` constructs, and Listing 14 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 $\{(x_1, y_1 = f(x_1)), (x_2, y_2 = f(x_2)), \dots\}$ 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 $\{(x_1, y_1 = \text{future}(f(x_1))), (x_2, y_2 = \text{future}(f(x_2))), \dots\}$. 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 13: Buggy Use of Future Tasks due to missing final declarations

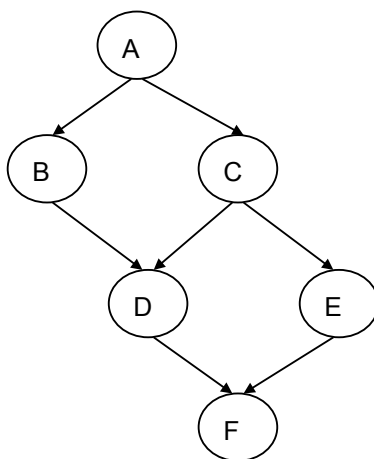


Figure 18: Computation Graph G_3

```

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

```

Listing 14: Task-parallel code with futures to generate Computation Graph G_3 from Figure 18

```
// 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 19: 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 19 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

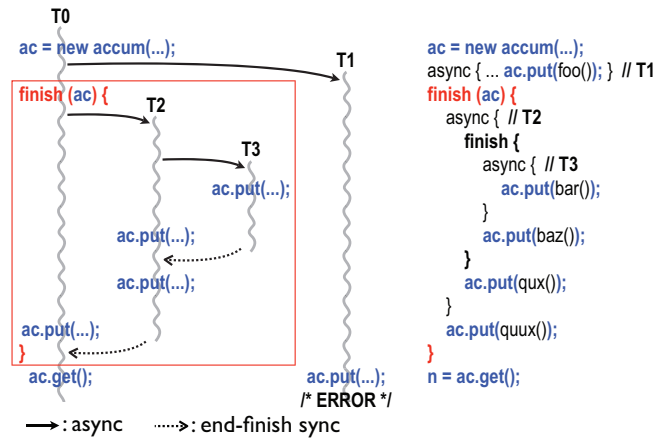


Figure 20: Finish accumulator example with three tasks that perform a correct reduction and one that throws an exception

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 (ac1, ac2, ..., acn) { stmt }`”. Note that `finish (ac)` becomes a no-op if *ac* is already associated with an outer `finish` scope.

Figure 20 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 21 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 21: User-defined reduction example

Map data structure is as a set of pairs, $S = \{(k_1, v_1), \dots, (k_n, v_n)\}$, 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, $f(k_i, v_i) = \{(k_1, v_1), \dots, (k_m, v_m)\}$. 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 15 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 16 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 15: 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 16: 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 17: 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 17. 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 is 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 17 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 18: 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 18. 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 19: 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 19, 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 19, 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 20: 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 20. 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 [17] 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 21: Rewrite of Task T3 from Listing 20

mainstream application programmers. For example, suppose an application programmer decided to rewrite the body of Task T3 as shown in Listing 21. 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 [9] 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 20, 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 [5] 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 is to provide a sequential method that implements the body of Task *T3* in Listing 20, 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 21 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.

```

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 22: 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 23: Examples of three parallel loops using forall (pseudocode)

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 22. In this pseudocode, we assume that the `for` construct can be used to express sequential multidimensional (nested) loops. For example, 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 22. Following past parallel languages, we use the `forall` notation to identify loops with single or multi-dimensional parallelism and an implicit finish. Listing 23, shows how the loops in Listing 22 can be rewritten using the `forall` notation.

3.2 Parallel Matrix Multiplication

Consider the pseudocode fragment for a parallel matrix multiplication example in Listing 24.

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

```

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 24: 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 25: Matrix multiplication program using forall

sequential and parallel loops in Listing 24 is not immediately easy to discern. Instead, the same program can be rewritten more compactly and clearly using `forall` loops as shown in Listing 25.

There are a number of features worth noting in Listing 25:

- 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. 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 25:

- `p1.rank` — returns rank of point `p1`
- `p1.get(i)` — returns element in dimension `i` of point `p1`, or element in dimension $(i \bmod p1.rank)$ 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 = p2.rank`.

```

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 26: Unchunked and chunked versions of a forall loop

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 $[1], \dots, [10]$, and $[1 : 10, -5 : 5]$ is a 2-dimensional region consisting of 110 points since the first dimension has 10 values ($1 \dots 10$) and the second dimension has 11 values ($-5 \dots 5$). 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 25. 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 `forsync` statement that is like `forall` but *does not* include an implicit `finish` statement. The statement `forsync (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 `forsync`.

There are other differences between `forsync` and `forall` which you will study later, most notably the fact that `forall` includes an implicit barrier, whereas `forsync` can be used for explicit phaser declarations.

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.

Listing 26 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()`,

```

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

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 integers⁸, 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 27. 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 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).

⁸`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 28: Hello-Goodbye forall loop

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 28. 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 tasks⁹. 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 28 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 29.

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 29, `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 22 illustrates how the barrier synchronization (`next` statement) works for the program example in Listing 29 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 22, 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) 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

⁹The 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 29: Hello-Goodbye forall loop with barrier (next) statement

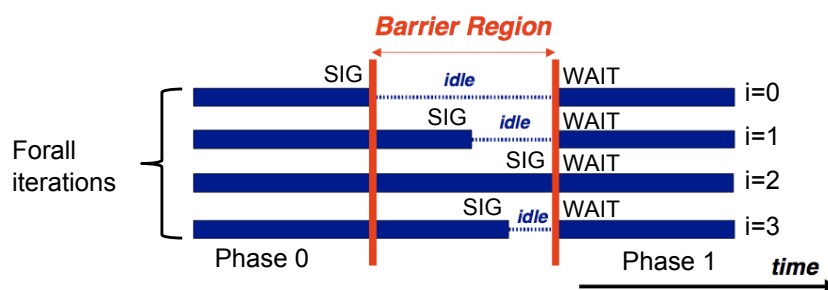


Figure 22: Illustration of barrier synchronization (next statement) for program example in Listing 29

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` statement¹⁰. 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 30 illustrates this point.

The iteration numbered i in the `forall- i` loop in line 1 of Listing 30 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 23 which illustrates how the set of `forall` iterations synchronizing on the barrier decreases after each phase in this example.

Figure 24 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

¹⁰Later, 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 30: Example of forall loop with varying numbers of next operations across different iterations

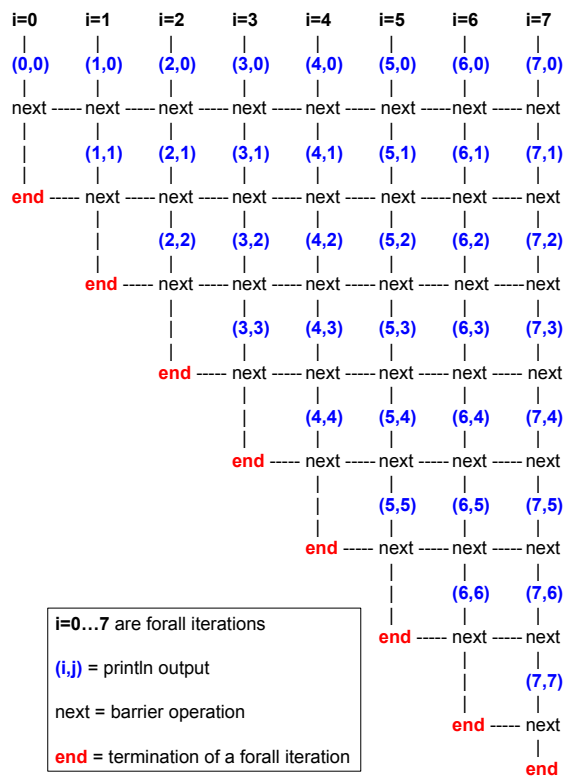


Figure 23: Illustration of the execution of forall example in Listing 30

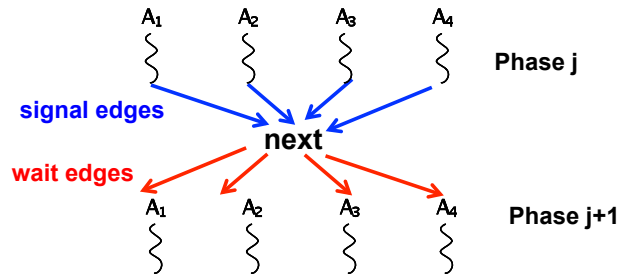


Figure 24: Modeling a next operation in the Computation Graph with Signal and Wait edges and a Next node

```

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 31: Hello-Goodbye program in Listing 29 extended with a second barrier to print a log message between the Hello and Goodbye phases

`next` statement causes the insertion of a single `next` node in the CG as shown in Figure 24. 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. Though 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 31. 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 25.

Listing 32 shows how a `next-with-single` statement can be used to perform the same computation as in Listing 31 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 iteration of the `forall` loop.

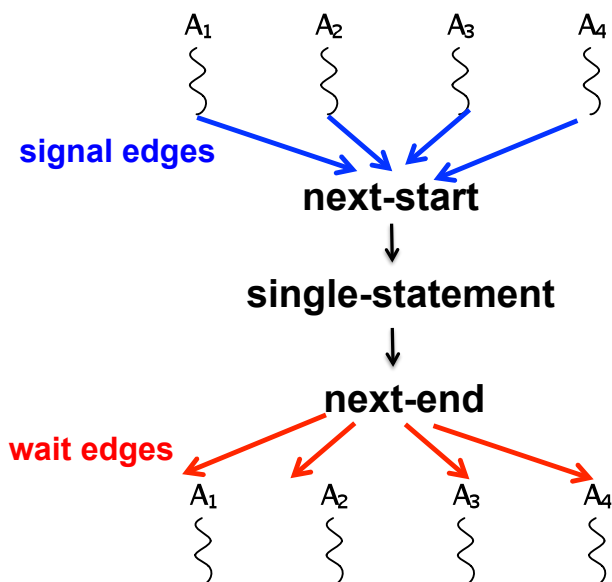


Figure 25: Modeling a next-with-single statement in the Computation Graph

```

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 32: Listing 31 extended with a next-with-single statement in lines 15-17

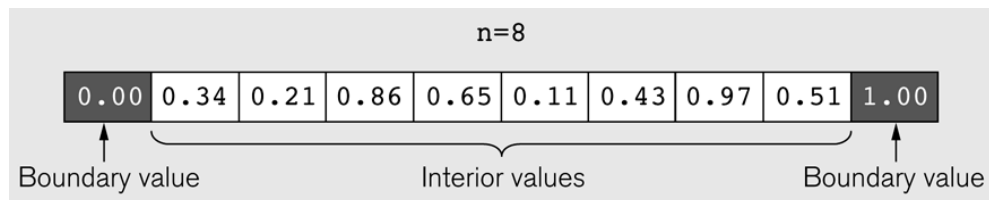


Figure 26: Illustration of the One-Dimensional Iterative Averaging Example for $n = 8$ (source: Figure 6.19 in [18])

```

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

3.5 One-Dimensional Iterative Averaging

To further motivate the need for barriers, consider the one-dimensional iterative averaging algorithm illustrated in Figure 26. 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 $1 \dots n$) 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 $1 \dots n$.

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 33 uses a `for-forall` structure, whereas the second version in Listing 34 uses a `forall-for-next` structure.

The first version in Listing 33 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 33, 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 34 shows an alternate implementation of the iterative averaging example using the `next` (barrier) statement. Now, the `forall` loop has moved to the outer level

```

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

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 34 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 $t \ll n$ 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 35 shows an alternate implementation of the iterative averaging example. The `forall` loop is again at the outermost level (line 4), as in Listing 34. 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.

```

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 35: 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)

4 Dataflow Synchronization and Pipelining

4.1 Fuzzy Barriers

Thus far, you have studied the use of `next` as shorthand for `signal` and `wait` operations performed by a task. In fact, the *deadlock freedom* property for phasers depends on the fact that the programmer does not have the ability to perform a `wait` operation separately from `signal` operations. However, there are some cases when it can be useful to perform a `signal` operation for a given phase ahead of a `next` operation. This idiom is referred to as *split-phase barriers* or *fuzzy barriers* [12].

A typical use case of this occurs when a task has a combination of *shared work* and *local work* to perform in a given phase. Listing 36 contains a simple example to illustrate this point.

Each of tasks $T1$ and $T2$ in Listing 36 contain shared work and local work in phase 0. Task $T1$ computes `a` in phase 0 (line 4) whose value is to be shared with task $T2$ in phase 1 (line 15). Likewise, task $T2$ computes `x` in phase 0 (line 11) whose value is to be shared with task $T1$ in phase 1 (line 8). The use of `next` in lines 7 and 14 provides the necessary synchronization to ensure that this sharing can occur without creating a data race.

However, each of tasks $T1$ and $T2$ also have some local work to perform between phases 0 and 1. In the case of $T1$, it is the computation of `b` in line 6, and in the case of $T2$, it is the computation of `y` in line 13. These local computations can logically be performed before or after the `next` operation, but it would be ideal if they could be performed in parallel with the phase transition. The use of `signal` statements in lines 5 and 12 achieves this desired effect.

When a task T performs a `signal` operation, it notifies all the phasers it is registered on that it has completed all the work expected by other tasks in the current phase (the “shared” work). Since `signal` is a non-blocking operation, an early execution of `signal` cannot create a deadlock. Later on, when T performs a `next` operation, the `next` degenerates to a `wait` since a `signal` has already been performed in the current phase. The execution of “local work” between `signal` and `next` is performed during phase transition; hence the use of names such as “split-phase barrier” and “fuzzy barrier” to describe this concept.

Figure 27 shows the Computation Graph (CG) corresponding to Listing 36. Note that there is a path of dependence edges from statement 4 in task $T1$ to statement 15 in task $T2$, but no dependence path from statement 6 in task $T1$ to statement 15 in task $T2$. Likewise, there is a path of dependence edges from statement 11 in task $T2$ to statement 8 in task $T1$, but no dependence path from statement 13 in task $T2$ to statement 8 in task $T1$. The ability to overlap statements 6 and 13 with the phase transition reduces the

```

1  finish {
2    phaser ph = new phaser(phaserMode.SIG_WAIT);
3    async phased { // Task T1
4      a = ... ;    // Shared work in phase 0
5      signal;     // Signal completion of a's computation
6      b = ... ;    // Local work in phase 0
7      next;       // Barrier — wait for T2 to compute x
8      b = f(b,x); // Use x computed by T2 in phase 0
9    }
10   async phased { // Task T2
11     x = ... ;    // Shared work in phase 0
12     signal;     // Signal completion of x's computation
13     y = ... ;    // Local work in phase 0
14     next;       // Barrier — wait for T1 to compute a
15     y = f(y,a); // Use a computed by T1 in phase 0
16   }
17 } // finish

```

Listing 36: Example of split-phase barrier

overall critical path length of the CG, compared to the case if the `signal` statements in lines 5 and 12 were removed.

The HJ language provides two utility functions `ph.getSigPhase()` and `ph.getWaitPhase()` to indicate which phase a task is in with respect to phaser `ph`. At the start of phase i , both `ph.getSigPhase()` and `ph.getWaitPhase()` will return i . However, if a `signal` operation is performed before a `next`, then `ph.getSigPhase()` returns $i + 1$ while `ph.getWaitPhase()` still returns i at any program point between `signal` and `next`. After the `next`, both `ph.getSigPhase()` and `ph.getWaitPhase()` will return $i + 1$. These signal and wait phase numbers can be used to establish the following *phase-ordering property*, which states how the phase numbers can be used to establish ordering among steps in a Computation Graph.

Phase-ordering property: Given two CG steps S_1 and S_2 in the CG, if there exists a phaser ph such that $ph.getSigPhase(S_1) < ph.getWaitPhase(S_2)$ then there must be a path of dependence edges from S_1 to S_2 in the CG.

As an example, statement 4 in Figure 27 would return `ph.getSigPhase() = 0` but statement 6 in Figure 27 would return `ph.getSigPhase() = 1`. Since statement 15 would return `ph.getWaitPhase() = 1`, the phase-ordering property guarantees that statement 4 (but not statement 6) must precede statement 15.

The idea of split-phase barriers can be used to further increase the parallelism in the optimized one-dimensional iterative averaging example using point-to-point synchronization in Listing 45. The key observation is that only the boundary elements need to be communicated between tasks in each iteration. Listing 37 shows how the split-phase concept can be combined with point-to-point synchronization. The computations of `myNew[start]` and `myNew[end]` have now been “peeled” out in lines 12 and 13, after which the `signal` operation in line 14 indicates that all shared work has been completed. The computation of interior elements `start+1:end-1` is local work and can be overlapped with phase transition by placing the `for` loop in lines 15 and 16 between the `signal` (line 14) and `next` (line 17) operations.

4.2 Point-to-point Synchronization with Phasers

Your repertoire of fundamental parallel programming constructs now includes task creation (`async`, `async await`), task termination (`finish`, `future.get()`), and barrier synchronization (`next` in `forall` loops). These constructs can all be modeled by a unified Computation Graph in which each node represents a sequential indivisible *step*, and each (directed) edge represents one of the following cases:

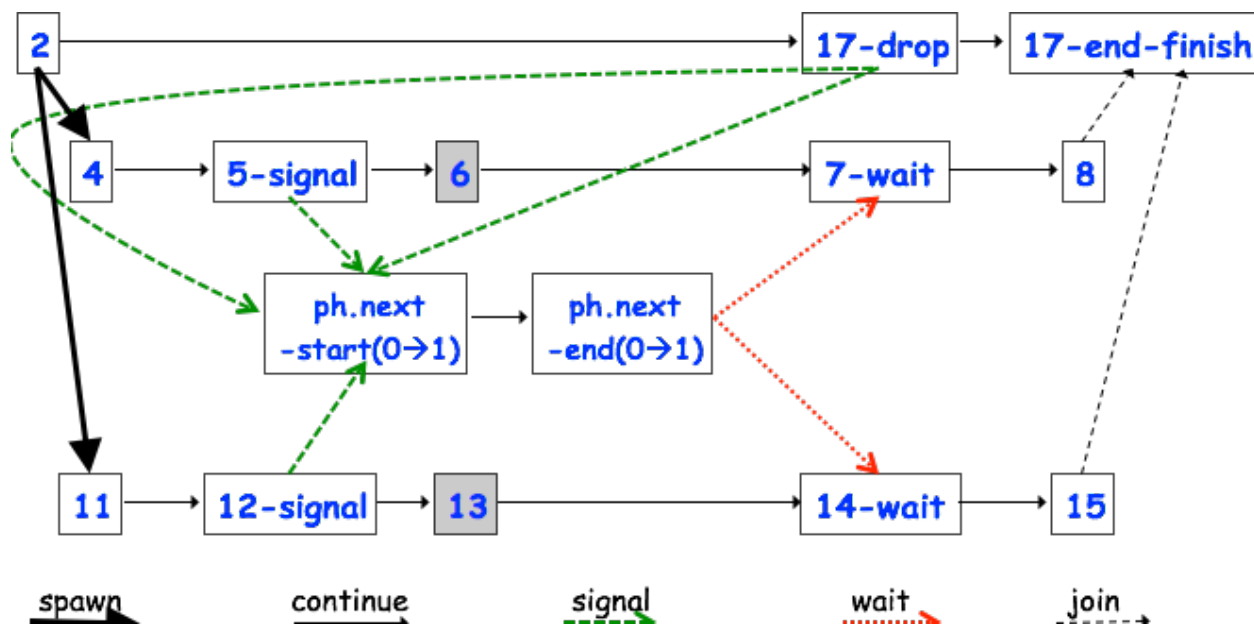


Figure 27: Computation Graph for HJ program in Listing 36. Shaded steps 6 and 13 represent local work executed in parallel with phase transition.

```

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  finish {
5    phaser ph = new phaser[t+2];
6    forall(point [i]:[0:t+1]) ph[i]=new phaser(phaserMode.SIG_WAIT);
7    for (point [i] : [1:t])
8      async phased(ph[i]<SIG>, ph[i-1]<WAIT>, ph[i+1]<WAIT>) {
9        double [] myVal = val1; double myNew = val2; double [] temp = null;
10       int start=(i-1)*batchSize+1; int end=Math.min(start+batchSize-1,n);
11       for (point [iter] : [0:iterations-1]) {
12         myNew[start] = (myVal[start-1] + myVal[start+1])/2.0;
13         myNew[end] = (myVal[end-1] + myVal[end+1])/2.0;
14         signal; // signal ph[i]
15         for (point [j] : [start+1:end-1])
16           myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
17         next; // wait on ph[i-1] and ph[i+1]
18         temp = myNew; myNew = myVal; myVal = temp; // swap(myNew, myVal)
19       } // for
20     } // for-async
21 } // finish

```

Listing 37: Optimized One-Dimensional Iterative Averaging Example using signal statements for split-phase point-to-point synchronization


```

1  finish {
2    for (point [i] : [1:m])
3      async {
4        doPhase1(i);
5        doPhase2(i);
6      }
7  }

```

Listing 38: A simple parallel loop

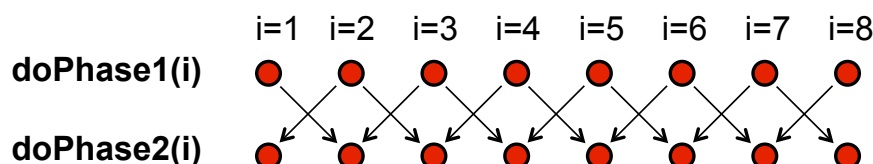


Figure 28: Illustration of left-neighbor right-neighbor point-to-point synchronization in a loop like Listing 38, assuming $m = 8$

- *Continue edges* capture sequencing of steps within a task.
- *Spawn edges* capture creation of child tasks, as in `async` and `forall`.
- *Join edges* capture task termination as in `finish` and `future.get()`.
- *Signal* and *wait edges* capture directed synchronization as in the `next` statement.

While barrier synchronization is an important special case of directed synchronization, there are many other synchronization patterns that can be expressed using *signal* and *wait* edges in a Computation Graph including *nearest-neighbor synchronization* and *pipeline parallelism*. These patterns build on the idea of *point-to-point synchronization* between two tasks that date back to the concept of *binary semaphores* [8]¹¹.

As one example of point-to-point synchronization, consider the following parallel loop expressed using the expanded `finish-for-async` notation rather than a `forall`. (The reason for the expansion will become clear later.) Each iteration i of the parallel loop in Listing 38 performs two method calls in sequence, `doPhase1(i)` and `doPhase2(i)`. Earlier, you learned how to insert a `next` statement in a `forall` loop to implement a barrier. Now, consider a less constrained case when iteration i only needs to wait for iterations $i - 1$ and $i + 1$ to complete their work in `doPhase1()` before it starts `doPhase2(i)` *i.e.*, iteration i only needs to synchronize on its *left neighbor* and *right neighbor*.

Figure 28 illustrates this synchronization pattern. Since it is different from a barrier, you will learn how the necessary point-to-point synchronization can be implemented using the more general phaser construct. Note the boundary conditions in Figure 28 where iteration $i = 1$ only needs to wait for iteration $i + 1 = 2$ to complete `doPhase1()`, and iteration $i = m - 1 = 8$ only needs to wait for iteration $i = m - 1 = 7$ to complete `doPhase1()`. Continuing the dining analogy discussed earlier for barriers, we can think of this synchronization as enforcing a rule that you cannot advance to your next course (Phase 2) until the neighbors on your left and right have completed their previous course (Phase 1), which is a less stringent requirement than a barrier between the two phases.

The previous examples motivated the need for point-to-point synchronization in multiple scenarios. We now introduce *phasers* [22]. Phasers were added to the HJ language as an extension to the *clock* con-

¹¹When you learn about semaphores in an Operating Systems class, you can relate them to this class by thinking of semaphores as lower-level OS mechanisms that can be used to implement programming constructs such as `finish`, `future.get()`, `isolated`, and `next`.

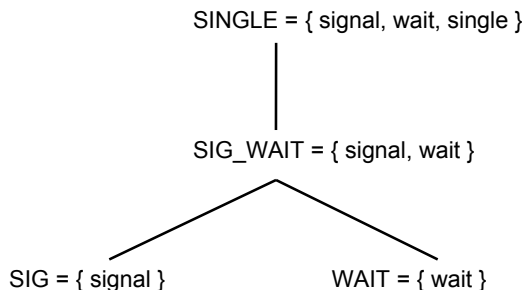


Figure 29: Capability hierarchy for phaser registration modes

struct developed in the X10 language [6]. A limited version of phasers was also added to the Java 7 `java.util.concurrent.Phaser` library [19]. A phaser is a special kind of *synchronization object* that unifies point-to-point and barrier synchronization in a single construct, while supporting *dynamic parallelism i.e.*, the ability for tasks to drop phaser registrations and for new tasks to add new phaser registrations.

Each `async` task has the option of registering with a phaser in *signal-only* (`signal`) or *wait-only* mode for point-to-point synchronization or *signal-wait* mode for barrier synchronization. As with barriers, a `next` statement is used to advance each phaser that a task is registered on (according to the registration modes), and the `next` statement can optionally include a `single` statement which is guaranteed to be executed exactly once during a phase transition. The `forall` barrier synchronization that you learned earlier is actually implemented by allocating an implicit phaser for each `forall` instance, and registering all `forall` iterations on that phaser in *signal-wait* mode.

At any point in time, a task can be registered in one of four modes with respect to a phaser: `SINGLE`, *signal-wait*, *signal-only*, or *wait-only*. The mode defines the set of capabilities — `signal`, `wait`, `single` — that the task has with respect to the phaser. The subset relationship defines a natural hierarchy of the registration modes, as shown in Figure 29. Thus, *signal-wait* mode is above *signal-only* and *wait-only* modes in the hierarchy because the capability set for registration mode *signal-wait* is `{signal, wait}` and the capability sets for *signal-only* and *wait-only* are `{signal}` and `{wait}` respectively.

The four primary phaser operations that can be performed by a task, A_i , are defined as follows. You will learn some additional (advanced) phaser operations later in the course:

1. **new:** When A_i performs a `new phaser(MODE)` operation, it results in the creation of a new phaser, ph , such that A_i is registered with ph according to `MODE`. If `MODE` is omitted, the default mode assumed is *signal-wait*. Phaser creation also initializes the phaser to its first phase (phase 0). At this point, A_i is the only task registered on ph . Additional descendant tasks of A_i can add registrations on ph , using the `async phased` construct described in item 2. The scope of the phaser is limited to the Immediately Enclosing Finish (IEF) for the `new` statement.
2. **phased async:** `async phased (ph1(mode1), ...) Aj`
When task A_i creates an `async` child task A_j , it has the option of registering A_j with any *subset* of phaser capabilities possessed by A_i . This subset is specified by the registration mode (`modek` for phaser ph_k) contained in the `phased` clause. We also permit the “`async phased Aj`” syntax as shorthand to indicate that A_i is transmitting all its capabilities on all phasers that it is registered with to A_j .

The subset rule is also referred to as the *Capability Rule* for phasers *i.e.*, a child task can only inherit a subset of its parent task’s phaser capabilities. However, it can add new capabilities by creating new phasers in its scope.

3. **drop:** A_i implicitly drops its registration on all phasers when it terminates. In addition, when A_i executes an end-finish instruction for finish statement F , it implicitly de-registers from each phaser

```

1  finish {
2    ph = new phaser (); // Default mode is SIG_WAIT
3    async phased(ph<SIG>){doA1Phase1 (); next; doA1Phase2 ();} //A1 (SIG mode)
4    async phased{doA2Phase1 (); next; doA2Phase2 ();} //A2 (SIG_WAIT mode)
5    async phased{doA3Phase1 (); next; doA3Phase2 ();} //A3 (SIG_WAIT mode)
6    async phased(ph<WAIT>){doA4Phase1 (); next; doA4Phase2 ();} //A4 (WAIT mode)
7  }

```

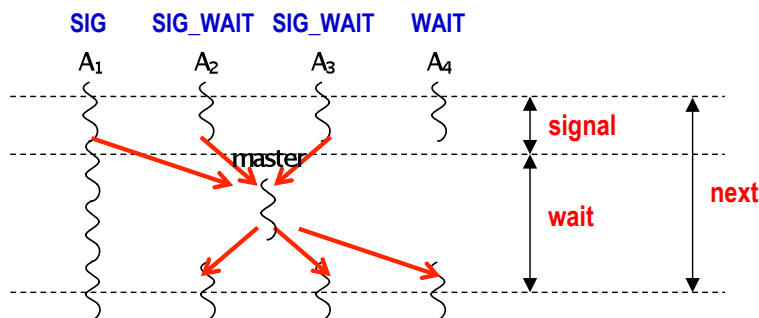
Listing 39: Simple example with four `async` tasks and one phaser

Semantics of `next` depends on registration mode

SIG_WAIT: `next = signal + wait`

SIG: `next = signal`

WAIT: `next = wait`



A master thread (worker) gathers all signals and broadcasts a barrier completion

Figure 30: “`next`” actions performed by tasks with different registration modes on the same phaser

ph for which F is the Immediately Enclosing Finish (IEF) for ph ’s creation. Finally, any task A_i can explicitly de-register from phaser ph by executing $ph.drop()$.

- next:** The `next` operation has the effect of advancing each phaser on which A_i is registered to its next phase, thereby synchronizing all tasks registered on the same phaser. As described in [22], the semantics of `next` depends on the registration mode that A_i has with a specific phaser, ph .

Listing 39 contains a simple example to illustrate these concepts. A single phaser, `ph` is allocated in line 2, and its IEF is the `finish` statement that spans lines 1–7. Four `async` tasks are created in lines 3–6 with different registration modes defined by their `phased` clauses. Figure 30 illustrates these concepts by showing the actions performed by the `next` statements in tasks $A_1 \dots A_4$ as Computation Graph edges. The unification of point-to-point synchronization can be seen in the fact that tasks A_2 and A_3 perform barrier (*signal-wait*) operations at each `next` statement, whereas tasks A_1 and A_4 perform point-to-point synchronizations.

Let us now revisit the example in Listing 38, and first consider the simple case when $m = 3$. Listing 40 shows the use of phasers to ensure proper synchronization for the $m = 3$ case. The synchronizations enforced by the phasers `ph1`, `ph2`, and `ph3` match the pattern in Figure 28, including the boundary conditions for $i = 1$ and $i = 3$, which need to wait on one phaser each. This code could also have been written by creating separate `future async` tasks for method calls `doPhase1()` and `doPhase2()`, and by using `future.get()` operations instead of `next`. However, doing so would incur the same overheads and inconveniences discussed earlier for replacing barriers by `finish-async` constructs.

```

1  finish {
2    phaser ph1 = new phaser (); // Default mode is SIG_WAIT
3    phaser ph2 = new phaser (); // Default mode is SIG_WAIT
4    phaser ph3 = new phaser (); // Default mode is SIG_WAIT
5    async phased(ph1<SIG>, ph2<WAIT>) { // i = 1
6      doPhase1(1);
7      next; // Signals ph1, and waits on ph2
8      doPhase2(1);
9    }
10   async phased(ph2<SIG>, ph1<WAIT>, ph3<WAIT>) { // i = 2
11     doPhase1(2);
12     next; // Signals ph2, and waits on ph1 and ph3
13     doPhase2(2);
14   }
15   async phased(ph3<SIG>, ph2<WAIT>) { // i = 3
16     doPhase1(3);
17     next; // Signals ph3, and waits on ph2
18     doPhase2(3);
19   }
20 }

```

Listing 40: Extension of example in Listing 38 with three phasers for $m = 3$

```

1  finish {
2    phaser ph = new phaser [m+2];
3    forall(point [i]:[0:m+1]) ph[i]=new phaser (); //Default mode is SIG_WAIT
4    forasync phased(ph[i]<SIG>, ph[i-1]<WAIT>, ph[i+1]<WAIT>)
5    (point [i] : [1:m]) {
6      doPhase1(i);
7      next; // Signals ph[i], and waits on ph[i-1] and ph[i+1]
8      doPhase2(i);
9    }

```

Listing 41: Extension of example in Listing 38 with array of $m + 2$ phasers for general m

This idea can be extended to general values of m by using the code in Listing 41. Lines 2 and 3 allocate and initialize a *phaser array*. This makes it convenient to specify the phaser registrations as a function of i in the *phased* clause in line 5. No special case *if* conditions need to be specified for the boundary cases of $i = 1$ and $i = m$, since the general formulation works correctly in those cases too. For example, the *async* task for iteration $i = 1$ waits on phaser `ph[0]`. However, no task is registered with a *signal* capability on `ph[0]`, so this wait is essentially a no-op. Listing 41 also illustrates the use of a *phased* clause with a *forasync* construct.

In summary, phasers are general synchronization constructs that can be used to create complex synchronization patterns specified by *phased* clauses in *async* constructs. A task can be registered on multiple phasers in different modes, and a phaser can have multiple tasks registered on it. The use of a *next* statement, instead of individual *signal* and *wait* operations, ensures that each task performs the correct synchronization at the end of each phase. Further, since the expansion of *next* into *signal* and *wait* operations is performed by the compiler and runtime, it is not possible for the programmer to create a deadlock cycle with phasers.

Adding Phaser Operations to the Computation Graph You have already learned how to define Computation Graphs for HJ programs that use `async`, `finish`, and `future.get()` operations. We now add Computation Graph rules for `signal`, `wait`, `next` and `drop` phaser operations.

Recall that a Computation Graph (CG) node represents a *step* consisting of an arbitrary sequential computation, and that a task's execution is divided into steps by using *continuations* to define the boundary points. The set of operations that induce continuation points includes `async` (source of a *spawn* edge), end of a `finish` statement (destination of *join* edges), `future.get()` (destination of a *join* edge), `signal` and `drop` (source of *signal* edges), `wait` (destination of *wait* edges), and `next` (source of *signal* edges and destination of *wait* edges). As discussed earlier, it is acceptable to introduce finer-grained steps in the CG if so desired *i.e.*, to split a step into smaller steps than the boundaries induced by continuation points. The key constraint is that a continuation point should not be internal to a step.

Each `next` operation performed by a task is modeled as two steps, `signal` and `wait`, in the task. Likewise, each end-`finish` operation is modeled as two steps, `drop` (to drop all phaser registrations in the finish scope) and end-`finish`. For convenience, we assume that the CG also includes an unbounded set of pairs of *phase transition* CG nodes, $ph.next-start(i \rightarrow i + 1)$ and $ph.next-end(i \rightarrow i + 1)$, for each phaser ph allocated during program execution that transitions from phase i to phase $i + 1$.

Given a set of CG nodes, the following classes of CG edges enforce ordering constraints among the nodes:

1. *Continue* edges capture sequencing of steps within a task — all steps within the same task are connected by a chain of *continue* edges. In addition, each pair of phase transition CG nodes, $ph.next-start(i \rightarrow i + 1)$ and $ph.next-end(i \rightarrow i + 1)$, is connected by a *continue* edge.
2. *Spawn* edges 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 connect descendant tasks to their Immediately Enclosing Finish (IEF) operations and to `get()` operations for *future* tasks. 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. If the `async` represents a *future* task, then a *join* edge is also inserted from the last step in the `async` to each step that starts with a `get()` operation on that future.
4. *Signal* edges connect each `signal` or `drop` operation performed by a task to the phase transition node, $ph.next-start(i \rightarrow i + 1)$, for each phaser ph that the task is registered on with *signal capability* when that phaser is transitioning from phase i to phase $i + 1$.
5. *Wait* edges connect each phase transition node, $ph.next-end(i \rightarrow i + 1)$ to a `wait` or `next` operation performed by a task, for each phaser ph that the task is registered on with *wait capability* when that phaser is transitioning from phase i to phase $i + 1$.
6. *Single* edges connect each phase transition node, $ph.next-start(i \rightarrow i + 1)$ to the start of a `single` statement instance in a task registered on phaser ph with *single capability* when that phaser is transitioning from phase i to phase $i + 1$, and the end of that single statement to the phase transition node, $ph.next-end(i \rightarrow i + 1)$.

Let us now revisit the code in Listing 42 which shows the use of phasers to ensure proper synchronization for the $m = 3$ case of the left-right neighbor synchronization example studied earlier. The computation graph for this program is shown in Figure 31. The steps in Figure 31 are labeled with statement numbers from Listing 42. The *spawn*, *continue* and *join* edges capture the basic `finish-async` structure of the program as before. The new CG nodes include phaser transition nodes (*e.g.*, $ph1.next-start(0 \rightarrow 1)$ and $ph1.next-end(0 \rightarrow 1)$) and the expansion of `next` statements into `signal` and `wait` nodes (*e.g.*, the `next` operation in line 12 is expanded into two CG nodes, *12-signal* and *12-wait*). The new CG edges are the *signal* (green) and *wait* (red) edges corresponding to phaser operations. For example, a *signal* edge is added from the

```

1  finish {
2  phaser ph1 = new phaser(phaserMode.SIG_WAIT);
3  phaser ph2 = new phaser(phaserMode.SIG_WAIT);
4  phaser ph3 = new phaser(phaserMode.SIG_WAIT);
5  async phased(ph1<phaserMode.SIG>, ph2<phaserMode.WAIT>)
6  { doPhase1(1); // Task T1
7    next; // Signals ph1, and waits on ph2
8    doPhase2(1);
9  }
10 async phased(ph2<phaserMode.SIG>, ph1<phaserMode.WAIT>, ph3<phaserMode.WAIT>)
11 { doPhase1(2); // Task T2
12   next; // Signals ph2, and waits on ph1 and ph3
13   doPhase2(2);
14 }
15 async phased(ph3<phaserMode.SIG>, ph2<phaserMode.WAIT>)
16 { doPhase1(3); // Task T3
17   next; // Signals ph3, and waits on ph2
18   doPhase2(3);
19 }
20 } // finish

```

Listing 42: Example of left-right neighbor synchronization for $m = 3$ case

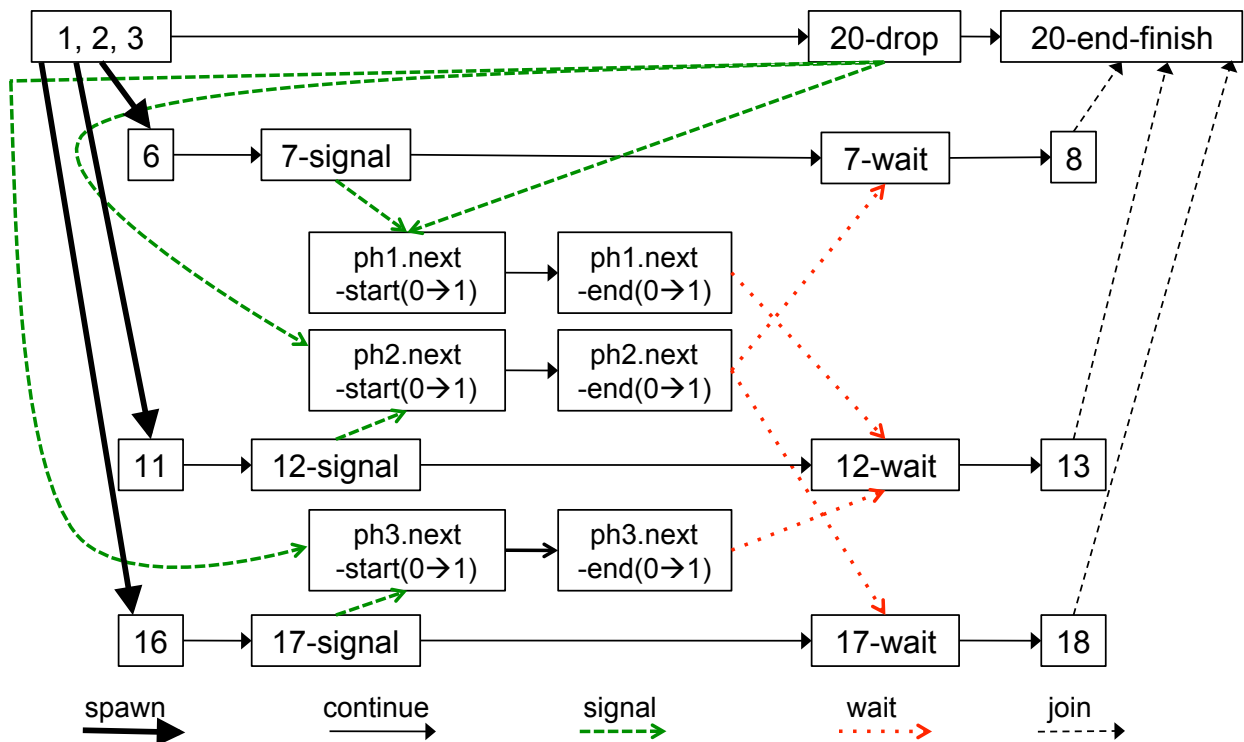


Figure 31: Computation Graph for example HJ program in Listing 42

```

1 rank.count = 0; // rank object contains an int field , count
2 finish {
3     phaser ph = new phaser(PhaserMode.SIG_WAIT);
4     for (point[i] : [0:m-1]) async phased {
5         // Start of phase 0
6         int square = i*i;
7         System.out.println("Hello from task " + i + " with square=" + square);
8         next; // Acts as barrier between phases 0 and 1
9         // Start of phase 1
10        System.out.println("Goodbye from task " + i + " with square=" + square);
11    } // for async phased
12 } // finish

```

Listing 43: Translation of Listing 29 to a finish-for-async-phased code structure (phaser version)

12-signal node to the $ph2.next-start(0 \rightarrow 1)$ node because task $T2$ is registered on phaser $ph2$ in **signal** mode. Likewise, **wait** edges are added from the $ph1.next-end(0 \rightarrow 1)$ and $ph3.next-end(0 \rightarrow 1)$ nodes to the 12-wait node because task $T2$ is registered on phasers $ph1$ and $ph3$ in **wait** mode. Finally, end-finish statement 20 is expanded into two CG nodes, 20-drop and 20-end-finish. The **signal** edges from 20-drop to the three $ph.next-start(0 \rightarrow 1)$ nodes models the synchronization that occurs with the parent task on **next** operations. The 20-end-finish nodes serves as the destination for **join** edges as before. Note that this splitting of an end-finish statement into two CG nodes avoids the creation of a cycle in the CG.

Barriers in forall Statements revisited Earlier, you learned the use of **next** as a barrier statement in **forall** loops with examples such as the one in Listing 29. However, now that you have learned the use of phaser operations in their full generality, the **forall** barrier can be understood as an *implicit* phaser allocated as part of a **forall** statement. Specifically, the HJ programming model implements a **forall** as the finish-for-async-phased code structure shown in Listing 43 with the allocation of a phaser, ph , that is not visible to the programmer. Recall that the default convention for **async phased** is that each child task inherits the same phaser registrations as its parent *i.e.*, **SIG_WAIT** registration on ph in this case. Even though both Listing 29 and Listing 43 are equivalent, it is recommended that you use the **forall** barrier version (Listing 29) whenever possible because of its simplicity. The phaser version (Listing 43) should be used only when more general synchronization patterns are needed.

4.3 One-Dimensional Iterative Averaging Example with Point-to-Point Synchronization

Listing 44 shows the optimized implementation of the One-Dimensional Iterative Averaging Example with a **forall** loop and barrier (**next**) operations. The first optimization implemented in this version is *batching* so that only t tasks (**forall** iterations) are created, and each task sequentially executes iterations $start:end$ in its batch (line 8). The second optimization is the reuse of two arrays instead of allocating a new array in each iteration of the **for-iter** loop in lines 7–12.

Figure 32 shows the synchronization structure for this example, where the red line indicates the barrier operation and the black arrows indicate the actual dependences in the algorithm that need to be satisfied across successive iterations of the **for-iter** loop in lines 7–12. As can be seen in Figure 32, the use of a barrier is overkill since the algorithm only needs left-right neighbor synchronization for correctness. Instead, the only synchronizations needed are the ones shown in blue lines in Figure 33. This can be accomplished using point-to-point synchronization with an array of phasers as shown in Listing 45. Note that the synchronization structure in this example is identical to that of the left-right neighbor synchronization studied earlier.

```

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 44: Optimized 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)

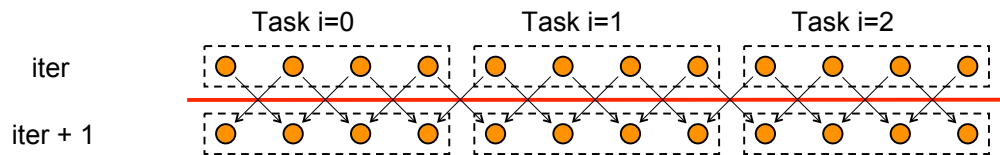


Figure 32: Synchronization structure of HJ program in Listing 44 (red line indicates barrier in line 10, and black arrows indicate actual dependences)

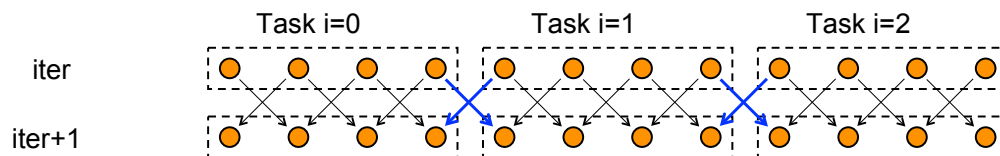


Figure 33: Synchronization structure of HJ program in Listing 45 (blue lines indicate point-to-point phaser synchronization, and black arrows indicate actual dependences)


```

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  finish {
5      phaser ph = new phaser[t+2];
6      forall(point [i]:[0:t+1]) ph[i]=new phaser(phaserMode.SIG_WAIT);
7      for (point [i] : [1:t])
8          async phased(ph[i]<SIG>, ph[i-1]<WAIT>, ph[i+1]<WAIT>) {
9              double[] myVal = val1; double myNew = val2; double[] temp = null;
10             int start = (i-1)*batchSize + 1; int end = Math.min(start+batchSize-1, n);
11             for (point [iter] : [0:iterations-1]) {
12                 for (point [j] : [start:end])
13                     myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
14                 next; // signal ph[i] and wait on ph[i-1] and ph[i+1]
15                 temp = myNew; myNew = myVal; myVal = temp; // swap(myNew, myVal)
16             } // for
17         } // for-async
18     } // finish

```

Listing 45: Optimized One-Dimensional Iterative Averaging Example using point-to-point synchronization, instead of barrier synchronization as in Listing 44

4.4 Pipeline Parallelism

As another example of point-to-point synchronization, consider a one-dimensional *pipeline* of tasks (stages), P_0, P_1, \dots , as shown in Figure 34. Pipelines occur in multiple domains *e.g.*, automobile manufacturing assembly lines and image processing pipelines. Figure 35 shows a pipeline used for seismic images that consists of three stages — simulation, rendering, and formatting. The seismic simulation in the first stage generates a sequence of results, one per *time step*. The rendering application in the second stage takes as input the simulation results for one time step, and generates an image for that time step. Finally, the formatting application in the third stage takes the image as input and outputs it into an animation sequence. Even though the processing is sequential for a single time step, pipeline parallelism can be exploited via point-to-point synchronization between neighboring stages so as to enable multiple stages to process different data items in a coordinated way.

This general idea can be seen in the timing diagram in Figure 36, which illustrates the scheduling of data items on pipeline stages, using the notation from Figure 34. The horizontal axis shows the progress of time from left to right, and the vertical axis shows which data item is being processed by which pipeline stage at a given time. Assuming that the inputs d_0, d_1, \dots arrive sequentially, pipeline parallelism can be exploited by enabling task (stage) P_i to work on item d_{k-i} when task (stage) P_0 is working on item d_k .

We can undertake a simplified analysis of the timing diagram in Figure 36, assuming that each stage takes 1 unit of time to process a single data item. Let n be the number of input items and p the number of stages in the pipeline ($n = p = 10$ in Figure 36.) Then, $WORK = n \times p$ is the total work that must be done for all data items, and $CPL = n + p - 1$ is the *critical path length* of the pipeline schedule shown in Figure 36. Thus, the ideal parallelism is $PAR = WORK/CPL = np/(n + p - 1)$.

This formula can be validated by considering a few boundary cases. When $p = 1$, the ideal parallelism degenerates to $PAR = 1$, which confirms that the computation is sequential when only one stage is available. Likewise, when $n = 1$, the ideal parallelism again degenerates to $PAR = 1$, which confirms that the computation is sequential when only one data item is available. When n is much larger than p ($n \gg p$), then the ideal parallelism approaches $PAR = p$ in the limit, which is the best possible case. Finally, when $n = p$, then the ideal parallelism becomes $p/(2 - 1/p)$, which approaches $PAR = p/2$, when $n = p$ is $\gg 1$.

In summary, assuming that the number of stages, p , is fixed, the ideal parallelism, PAR , increases with

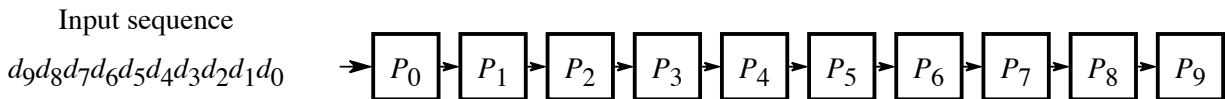


Figure 34: One-dimensional pipeline with tasks, $P_0 \dots P_9$, and sequence of input items, d_0, d_1, \dots (Source: Figure 5.6(a) in [25])

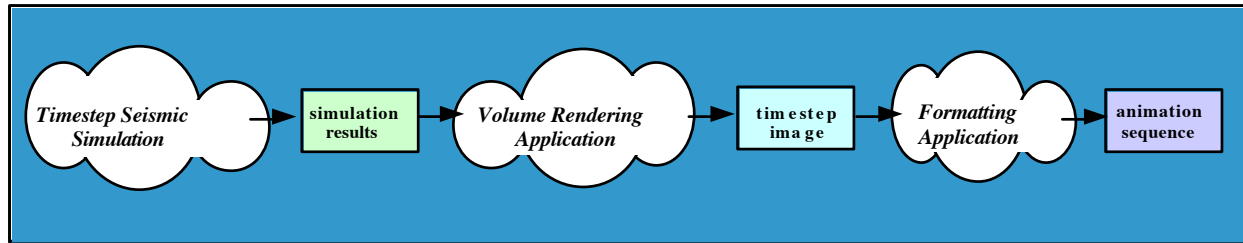


Figure 35: Seismic imaging pipeline (Source: [21])

an increasing number of data items. PAR starts with a degenerate value of 1 when $n = 1$, increases to approximately $p/2$ when $n = p$ and approaches p as n increases further. This analysis will need to be refined for more realistic scenarios (such as the seismic imaging pipeline in Figure 35) when different stages take different amounts of time. Also, pipelines can be multidimensional *e.g.*, in a two-dimensional pipeline, a stage may receive inputs from both the stage to its west and the stage to its north.

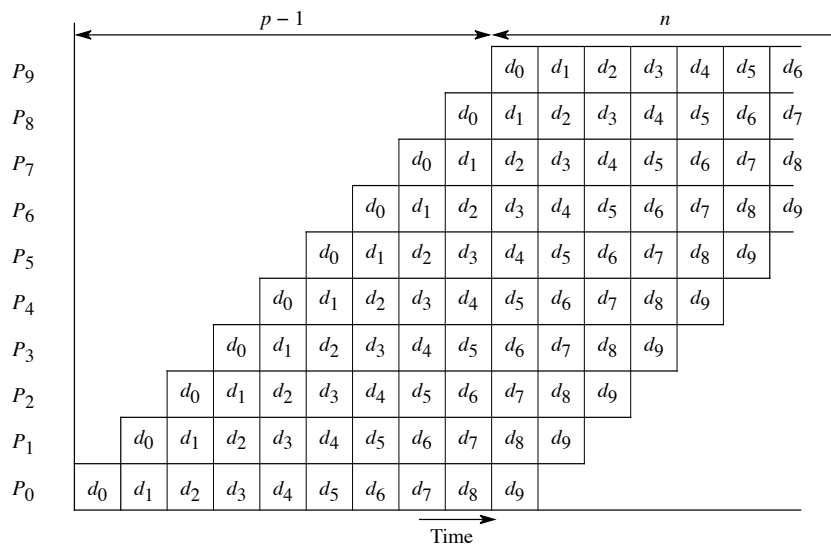


Figure 36: Timing diagram for pipeline shown in Figure 34 assuming $n = p = 10$ (Source: Figure 5.6(b) in [25])

4.5 Data-Driven Futures

Data-Driven Futures [23] are an extension of future tasks in HJ that support the dataflow model. This extension is enabled by adding an `await` clause to the `async` statement as follows:

```
async await (DDFa, DDFb, ...) { statement }
```

Each of DDF_a, DDF_b, \dots is an instance of the standard `DataDrivenFuture` class in HJ. A DDF acts as a container for a single-assignment value, like regular `future` objects. However, unlike `future` objects, DDF's can be used in an `await` clause, and any `async` task can be a potential producer for a DDF (though only one task can be the actual producer at runtime because of the single-assignment property).

Specifically, the following rules apply to `DataDrivenFuture` objects:

- A variable of type `DataDrivenFuture` is a reference to a DDF object. Unlike regular future objects, there is no requirement that variables of type `DataDrivenFuture` be declared with a `final` modifier.
- The following four operations can be performed on variables of type `DataDrivenFuture`:
 1. *Create* — an instance of a DDF container can be created using the standard Java statement, `new DataDrivenFuture()`.
 2. *Produce* — any task can provide a value, V , for a DDF container by performing the operation, `DDF.put(V)`. After the `put` operation, the DDF's value is said to become *available*. If another `put` operation is attempted on the same DDF, the second `put` will throw an exception because of its violation of the single-assignment rule.
 3. *Await* — a new `async` task can be created with an `await` clause, `await (DDFa, DDFb, ...)`. The task will only start execution after all the DDFs in the `await` clause become available.
 4. *Consume* — any task that contains a DDF in its `await` clause is permitted to perform a `get()` operation on the DDF. A *cast* operation will be required to cast the result of `get()` to the expected object type. If a `get()` operation is attempted by a task that has no `await` clause or does not contain the DDF in its `await` clause, then the `get()` will throw an exception. Thus, a `get()` on a DDF will never lead to a blocking operation (unlike a `get()` on a regular `future` object).
- It is possible for an instance of an `async` with an `await` clause to never be enabled, if one of its input DDF never becomes available. This can be viewed as a special case of *deadlock*.

Example The example HJ code fragment in Figure 37 shows five logically parallel tasks and how they are synchronized through DDFs. Initially, two DDFs are created as containers for data items `left` and `right`. Then a `finish` is created with five `async` tasks. The tasks, `leftReader` and `rightReader`, include `left` or `right` in their `await` clauses respectively. The fifth task, `bothReader`, includes both `left` and `right` in its `await` clause. Regardless of the underlying scheduler, the first two `async`s are guaranteed to execute before the fifth `async`.

The computation graph for the five tasks in Figure 37 can be seen in the left side of Figure 39. This exact graph structure can not be created with pure `async` and `finish` constructs without `futures` or DDFs. Instead, an approximate version with extra dependences can be created using `async` and `finish`, as shown in Figure 38 (for which the computation graph can be see in the right side of Figure 39).

Implementing regular Future Tasks using Data-Driven Futures To further understand Data-Driven Futures, we show how regular *future* tasks can be implemented using DDFs. Consider the example Habanero Java `future` construct shown in Figure 40. It can be rewritten to use DDFs, as shown in Figure 41. Once the DDF gets created, it will await the availability of the `f` object, if it has not been produced yet. The execution of that DDF will thus be delayed until `f` becomes available.

```

DataDrivenFuture left = new DataDrivenFuture();
DataDrivenFuture right = new DataDrivenFuture();
finish { // begin parallel region
    async left.put(leftBuilder()); // Task1
    async right.put(rightBuilder()); // Task2
    async await (left) leftReader(left.get()); // Task3
    async await (right) rightReader(right.get()); // Task4
    async await (left, right) bothReader(left.get(), right.get()); //Task5
} // end parallel region

```

Figure 37: Example Habanero Java code fragment with Data-Driven Futures.

```

Object left = new Object();
Object right = new Object();
finish { // begin region
    async left = leftBuilder(); // Task1
    async right = rightBuilder(); // Task2
} // end region
finish { // begin region
    async leftReader(left); // Task3
    async rightReader(right); // Task4
    async bothReader(left, right); //Task5
} // end region

```

Figure 38: A finish-async version of the example in Figure 37

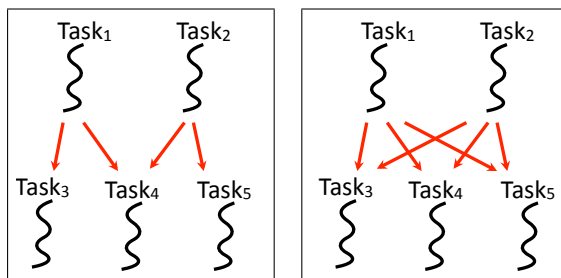


Figure 39: Computation Graphs for Figure 37 (left) and Figure 38 (right)

```

final future<int> f = async<int> { return g(); };
...
int local = f.get();

```

Figure 40: Habanero Java interface for the language construct future

```

DataDrivenFuture f = new DataDrivenFuture();
async { f.put(g()); };
...
async await (f) { local = f.get(); };

```

Figure 41: Data-Driven Future equivalent of Figure 40

```

1  finish {
2    DataDrivenFuture<Void> ddfA = new DataDrivenFuture<Void>();
3    DataDrivenFuture<Void> ddfB = new DataDrivenFuture<Void>();
4    DataDrivenFuture<Void> ddfC = new DataDrivenFuture<Void>();
5    DataDrivenFuture<Void> ddfD = new DataDrivenFuture<Void>();
6    DataDrivenFuture<Void> ddfE = new DataDrivenFuture<Void>();
7    async { . . . ; ddfA.put(); } // Task A
8    async await(ddfA) { . . . ; ddfB.put(); } // Task B
9    async await(ddfA) { . . . ; ddfC.put(); } // Task C
10   async await(ddfB,ddfC) { . . . ; ddfD.put(); } // Task D
11   async await(ddfC) { . . . ; ddfE.put(); } // Task E
12   async await(ddfD,ddfE) { . . . } // Task F
13 } // finish

```

Listing 46: HJ code to generate Computation Graph G_3 using DDFs

Data-Driven Futures with empty objects In Section 2.1.4, you learned how the `future<void>` type can be used to create `future async` tasks with a `void` return type. Likewise, a `DataDrivenFuture<Void>` can be used when a DDF is just intended for synchronization rather than communicating a value. (Actually, any object can be used, since no `get()` operation will be performed on such a DDF.)

Listing 46 shows the HJ code that can be used to generate computation graph G_3 in Figure 18 with DDFs. This code uses DDFs without `get()` operations, and provides a systematic way of converting any Computation Graph into an HJ program. The difference between the regular `future` version studied in Section 2.1.1 and this version is that the `async` tasks in Listing 46 will never be started till all their input DDFs become available.

A Optional topic: Abstract vs. Real Performance

Thus far in the course, we have lived in a “Neverland” where parallel programs never run slower than their sequential counterparts, if their performance is measured using idealized abstract performance metrics. However, in the real world, there are a number of *overheads* that can cause a parallel program to run slower than their sequential counterparts even when they have abundant amounts of parallelism. Here are some of the overheads that are encountered in practice for the constructs that you have studied so far:

Spawn overhead When an `async` construct is executed (the source of a *spawn* edge in the computation graph) there is extra book-keeping work that needs to be performed by the parent task to create necessary data structures for the child task. This extra work is referred to as *spawn overhead*, and it is incurred sequentially for each child task spawned by the same parent task. There is also overhead incurred when the child task begins execution, but that overhead can be incurred in parallel when different child tasks are executed by different workers.

Join overhead When a task T_C terminates (the source of a *join* edge in the computation graph) there is some overhead incurred to check if it is the last task in its Immediately Enclosing Finish (IEF) to terminate. If so, the *worker thread* executing T_C will switch tasks after T_C terminates, and resume execution of the ancestor task containing $IEF(T_C)$ which involves additional *context switching* overhead. If not, nothing further needs to be done and T_C ’s worker can look for other work after T_C terminates. If T_C is a *future task* then additional overhead is incurred to process any pending `get()` operations on the future.

IEF-Join overhead When the end of a `finish` construct F is encountered (the sink of a *join* edge in the computation graph) there is some overhead incurred to check if all tasks with $IEF = F$ have terminated.

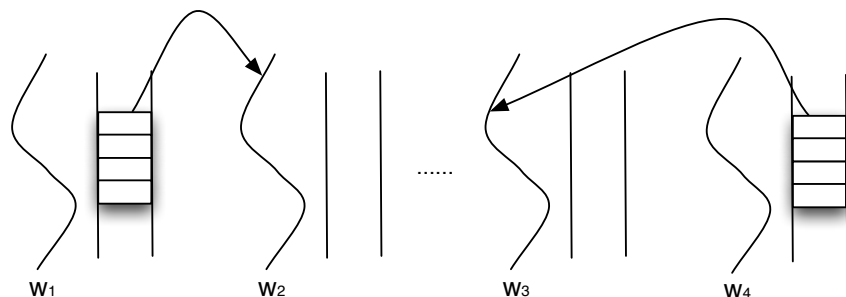


Figure 42: Work-stealing Scheduling Paradigm with distributed task queues (one queue per worker). Arrows represent stealing of tasks by idle workers from busy workers.

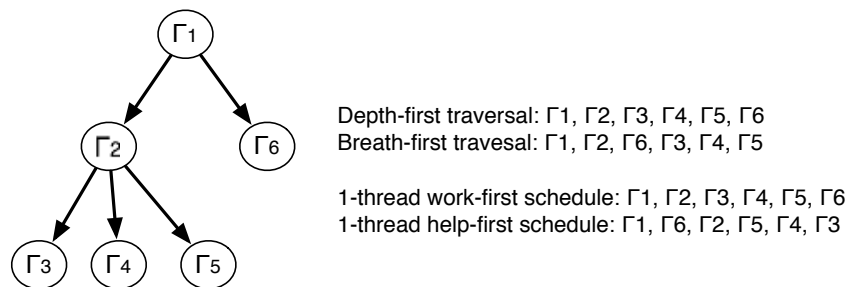


Figure 43: Example spawn tree with work-first and help-first schedules on a single worker. (A spawn tree only contains spawn edges from a computation graph.)

If so, nothing further needs to be done. If not, additional overhead is incurred by the worker executing F to suspend the current task at the end of F , and to look for other work instead (another instance of *context switching* overhead).

A.1 Work-sharing vs. Work-stealing schedulers

Earlier in the course, you learned that the Habanero-Java (HJ) runtime system is responsible for scheduling unbounded numbers of `async` tasks on a fixed number of processors. It does so by creating a fixed number of *worker threads* as shown in Figure 5, typically one worker per processor core. (This approach is used by many other task-parallel programming models.) These workers repeatedly pull work from a logical 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* (Section 1.1.1).

Work-sharing and *work-stealing* are two task scheduling paradigms for task parallelism. In work-sharing, when a new task is created, the creator works eagerly to share the task with other worker. This task redistribution is usually implemented by a centralized task queue as shown in Figure 5. A key limitation with the work-sharing approach is that accesses to the task queue by multiple workers need to be synchronized by the scheduler to avoid race conditions. This *centralized task queue* can become a scalability bottleneck when the number of workers increases, or when many fine-grain tasks are created. (Remember Amdahl's Law.) **Work-sharing is the default scheduling paradigm used in HJ** since it supports all the parallel constructs in the language.

In contrast, the *work-stealing* paradigm is implemented using distributed task queues, as shown in Figure 42. Each worker just adds work (tasks) to its own queue when new work is created. An idle worker then selects a "victim" worker to "steal" work from. If it doesn't obtain any work from its first choice of victim, it then attempts to steal work from another victim worker. Note that the time spent by the idle worker looking for work when no spare work is available will not add to the critical path length (CPL) of the program

```
finish { //startFinish
    for (int i=1; i<k; i++)
        async Ti; // task i
    T0; //task 0
}
```

Figure 44: Iterative Fork-Join Microbenchmark

execution. (Why?) This is one of the reasons why work-stealing is more efficient than work-sharing.

In classical work stealing [4], when a task Γ_a spawns task Γ_b , the worker that spawns Γ_b will start to work on Γ_b eagerly, and the continuation of Γ_a after the spawn might be stolen by other idle workers. This strategy is called the *work-first task scheduling policy* in the work-stealing paradigm. As the alternative, the worker can stay with Γ_a and let another worker help by stealing Γ_b . The worker will work on Γ_b later if Γ_b is not picked up by other workers. This alternative work-stealing strategy is called the *help-first task scheduling policy*.

The naming of the “work-first” and “help-first” policies can be understood from the perspective of the worker of the parent task when executing an `async`. Under the work-first policy, the parent task’s worker will *work* on the child task *first*. Under the help-first policy, the parent task’s worker will ask other workers to *help* execute the child task *first*. The work-first policy is sometimes referred to as a *depth-first* policy because, in a 1-worker execution, all the tasks will be executed in the order of depth-first traversal of the task *spawn tree*. Figure 43 shows a task spawn tree and the 1-thread execution order of the tasks under the work-first policy and the help-first policy.

The work-stealing paradigm with *work-first policy* can be enabled by using the “**hjc -rt w**” option when compiling HJ programs, whereas work-stealing with the *help-first policy* can be enabled by using the “**hjc -rt h**” option. However, the current implementation of the work-stealing paradigm does not support all parallel constructs in the language; notably, it does not support *future tasks*, *data-driven tasks* and *phasers*.

A.2 Modeling and Measurement of Overhead

In this section, we use *microbenchmarks* to model and measure some of the overheads described earlier. A convenient way to measure some of these overheads is to execute a parallel program on a single worker and compare the execution time with that of the sequential counterpart of the parallel program (if one is available).

A.2.1 Single-Worker Execution Times for Iterative Async-Finish example

We break down the single worker execution time (t_1) of a HJ program with a work-stealing scheduler into the following components: serial Java execution (t_s), `async` task spawns, context switches (t_{cs}) before starting a new task, `startFinish` (t_{sf}) and task synchronization at `endFinish`. Depending on the scheduling policy, the overhead of asynchronous task spawns is denoted as t_{aw} for work-first task spawns or t_{ah} for help-first task spawns. The task synchronization at `endFinish` is either trivial or non-trivial. In the trivial case, all tasks created in the finish scope will be completed and the worker will just continue execution. In the non-trivial case, the current serial execution flow will be interrupted, and a context switch is performed before executing new tasks. We use t_{ef} to denote the overhead of non-trivial task synchronization. The overhead of the trivial case is considered to be included in t_{sf} . A similar analysis can be performed for `get()` operations on future tasks, which are omitted for simplicity.

Consider the HJ microbenchmark shown in Figure 44. The microbenchmark performs k tasks; tasks 1 to $k-1$ are performed asynchronously and task 0 is performed serially. Under the work-first policy, the single worker execution incurs no context switch. Thus, the single worker execution time of the whole microbenchmark as a function of k for the work-first policy is

$$t_1^{wf}(k) = t_s(k) + t_{sf} + (k-1)t_{aw} \quad (2)$$

k	$t_s(k)$	$t_1^{wf}(k)$	$t_1^{hf}(k)$	$t_1^{ws}(k)$	Java-thread(k)
1	0.11	0.21	0.22		
2	0.22	0.44	2.80		
4	0.44	0.88	2.95		
8	0.90	1.96	3.92	335	3,600
16	1.80	3.79	6.28		
32	3.60	7.15	10.37		
64	7.17	14.59	19.61		
128	14.47	28.34	36.31	2,600	63,700
256	28.93	56.75	73.16		
512	57.53	114.12	148.61		
1024	114.85	270.42	347.83	22,700	768,000

Table 1: Execution time (in microseconds) of the serial execution time and the single worker HJ execution time of the microbenchmark in Figure 44 with $k = 1, 2, 4, 8, \dots, 1024$ on the Xeon SMP machine

where $t_s(k)$ is the serial execution time of the whole microbenchmark as a function of k . There is no extra synchronization cost (t_{ef}) at the end of the finish scope for single-worker work-first execution.

Under the help-first policy, tasks 1 to $k-1$ will be executed after a context switch and there is one non-trivial task synchronization at `end-finish` for $k > 1$. Thus

$$t_1^{hf}(k) = \begin{cases} t_s(k) + t_{sf}, & k = 1 \\ t_s(k) + t_{sf} + (k-1)(t_{ah} + t_{cs}) + t_{ef}, & k > 1 \end{cases} \quad (3)$$

We assume every task in the microbenchmark shown in Figure 44 contains the same amount of work, t_0 (also called *task granularity*). The task granularity t_0 can be calculated as the slope of the serial execution $t_s(k)$. Based on Equation 2 and 3, we have

$$\begin{aligned} t_{sf} &= t_1^{wf}(1) - t_s = t_1^{hf}(1) - t_s(1) \\ t_{aw} &= SLOPE(k, t_1^{wf}(k)) - t_0 \quad (k \geq 1) \\ t_{ah} + t_{cs} &= SLOPE(k, t_1^{hf}(k)) - t_0 \quad (k > 1) \\ t_{ef} &= t_1^{hf}(2) - t_1^{hf}(1) - t_{aw} + t_{cs} - t_0 \end{aligned}$$

Table 1 shows the execution time of the serial Java execution times and the single worker HJ execution time of the microbenchmark in Figure 44 for $k = 1, 2, 4, 8, \dots, 1024$ on a Xeon SMP machine. A few data points are also provided for work-sharing ($t_1^{ws}(k)$) and Java-thread implementations of `async` tasks.

We calculate t_0 , t_{sf} , t_{aw} , $t_{ah} + t_{cs}$ and t_{ef} from and get the following results from Table 1: $t_0 \approx 0.1\mu s$, or 417 cycles, $t_{sf} \approx 0.1\mu s$ or 417 cycles, $t_{aw} \approx 0.15\mu s$ or 625 cycles, $t_{ah} + t_{cs} \approx 0.22\mu s$ or 917 cycles, $t_{ef} \approx 2.11\mu s$ or 8800 cycles.

A.2.2 Single-Worker Execution Times for Recursive Async example

Consider the HJ microbenchmark shown in Figure 45 and Figure 46. The code shown in Figure 45 uses a global finish to synchronize all task whereas each task in the code shown in Figure 46 waits for all child tasks. For `fib(35)`, both versions spawns 29,860,703 tasks. The code shown in Figure 46 (version 2) will also create 14,930,351 finish instances. Using the earlier definitions and notations, the 1-worker execution times of the version shown in Figure 45 under the work-first and help-first policy are:

$$t_{wf}^1 = t_s + t_{sf} + t_{aw} * 29,860,703$$

```
finish fib(n);

fib (int n) {
    if (n<2) {
        . . .
    } else {
        async fib(n-1);
        async fib(n-2);
    }
}
```

Figure 45: Version 1: Recursive microbenchmark with one global finish scope. Even though the method is called `fin()`, it does not actually compute the Fibonacci sequence.

```
void fib (int n) {
    if (n<2) {
        . . .
    } else {
        finish {
            async fib(n-1);
            async fib(n-2);
        }
    }
}
```

Figure 46: Version 2: Recursive microbenchmark in which every task waits for its child tasks. Even though the method is called `fin()`, it does not actually compute the Fibonacci sequence.

	serial	1-worker work-first	1-worker help-first
Code in Figure 45 (Version 1)	0.103	3.405	6.974
Code in Figure 46 (Version 2)	0.103	5.872	43.18

Table 2: Execution time (in secs) of the serial and 1-worker execution time of the code shown in Figure 45 and 46 under both policies. Both versions have the same sequential counterpart.

```

void fib (int n) {
    if (n<2) {
        . . .
    } else if ( n > THRESHOLD) { // PARALLEL VERSION
        finish {
            async fib(n-1);
            async fib(n-2);
        }
    }
    else { // SEQUENTIAL VERSION
        fib(n-1); fib(n-2);
    }
}

```

Figure 47: Modified version of Figure 46 with threshold test

```

void fib (int n) {
    if (n<2) {
        . . .
    } else {
        finish {
            async seq(n <= THRESHOLD) fib(n-1);
            async seq(n <= THRESHOLD) fib(n-2);
        }
    }
}

```

Figure 48: Figure 47 rewritten with seq clause

$$t_{hf}^1 = t_s + t_{sf} + t_{ef} + (t_{ah} + t_{cs}) * 29,860,703$$

The 1-worker execution time of the code shown in Figure 46 under the work-first and help-first policy are:

$$t_{wf}^2 = t_s + t_{sf} * 14,930,351 + t_{aw} * 29,860,703$$

$$t_{hf}^2 = t_s + (t_{sf} + t_{ef}) * 14,930,351 + (t_{ah} + t_{cs}) * 29,860,703$$

Table 2 shows the serial and 1-worker execution time of the code shown in Figure 45 and 46 under both policies. We compute t_{aw} , $t_{ah} + t_{cs}$, t_{sf} and t_{ef} , and get the following result: $t_{aw} = 0.11\mu s$ (460 cycles), $t_{ah} + t_{cs} = 0.23\mu s$ (960 cycles), $t_{sf} = 0.17\mu s$ (709 cycles), $t_{ef} = 2.26\mu s$ (9400 cycles).

The results from both examples confirm that spawning and executing a task under the help-first policy (about 0.22-0.23 microseconds) is slower than the work-first policy (about 0.11-0.15 microseconds), and the context switch at the `endFinish` instruction is the most expensive operation (about 2.11-2.26 microseconds). The construction and destruction of a finish instance costs about 0.10-0.17 microseconds. We did not include results for work-sharing and Java threads for the recursive microbenchmarks because they would take too long to compute and, in some cases, may even run out of memory.

A.3 The “seq” clause for Async tasks

A very common way to reduce the large overheads seen in Table 2 is for the programmer to add a *threshold test* as shown in Figure 47. A `THRESHOLD` value of T implies that all calls to `Fib()` with parameters $\leq T$ are implemented as sequential calls. However, one of the drawbacks of the approach in Figure 47 is that it violates the *Don't Repeat Yourself (DRY)* principle of software engineering since the same work needs to be specified in the parallel and sequential versions. To avoid this problem, the HJ language supports a `seq`

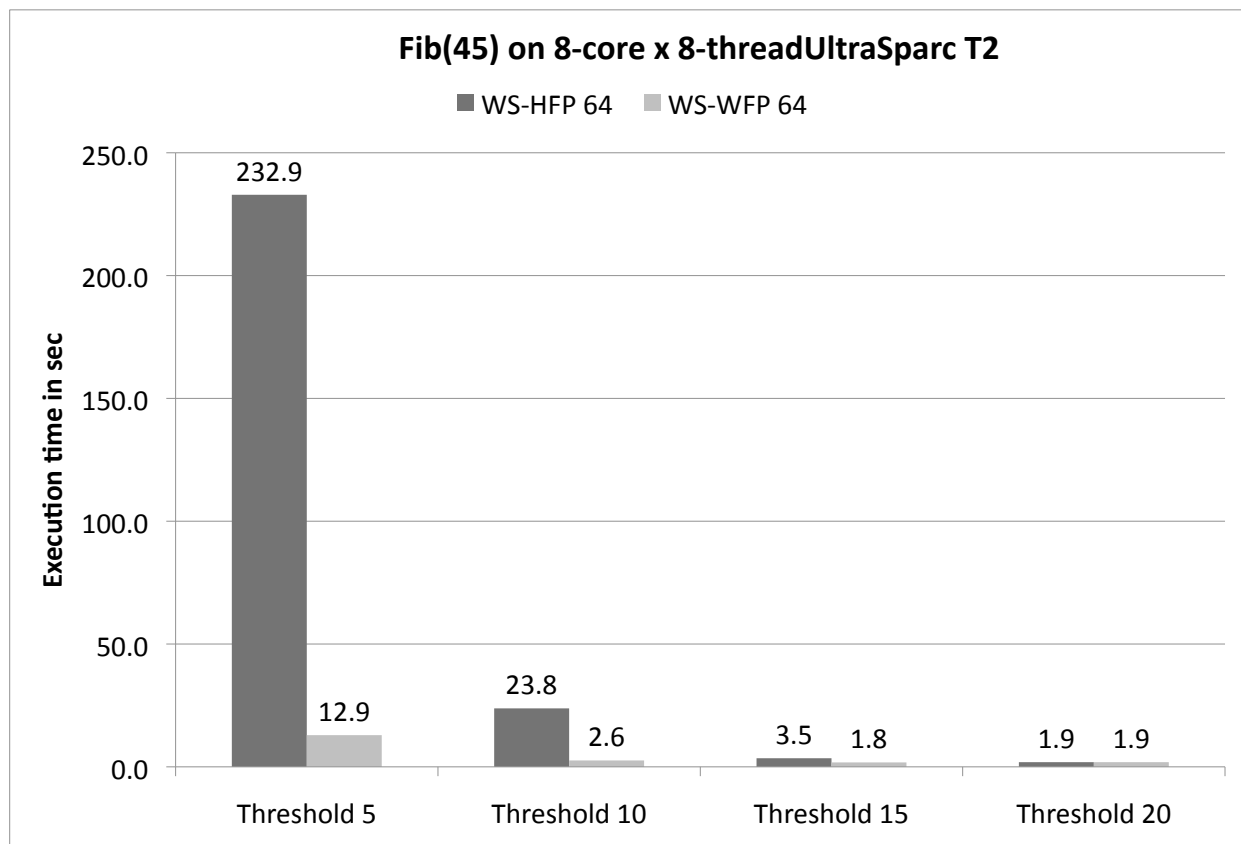


Figure 49: Execution times of `Fib(45)` using 64 workers for HFP (Help-First Policy) and WFP (Work-First Policy) with thresholds 5, 10, 15, and 20 on an UltraSparc T2 system.

clause for an `async` statement with the following syntax and the semantics:

```
async seq(cond) <stmt> ≡ if (cond) <stmt> else async <stmt>
```

The `seq` clause simplifies programmer-controlled serialization of task creation to deal with the overhead issue. However, the `seq` clause is restricted to cases when no blocking operation such as `finish` or `get()` is permitted inside `<stmt>`. The main benefit of the `seq` clause is that it removes the burden on the programmer to specify `<stmt>` twice with the accompanying software engineering hazard of ensuring that the two copies remain consistent. Figure 48 shows how the code in Figure 47 can be rewritten with the `seq` clause. In the future, the HJ system will explore approaches in which the compiler and/or runtime system can select the serialization condition automatically for any `async` statement. The `seq` clause is currently not supported for future `async` tasks.

Figure 49 compares the execution time of the work-first and help-first policies for `Fib(45)` using 64 workers and varying `THRESHOLD` from 5 to 20. As expected, the work-first policy (WS-WFP) significantly outperforms the help-first policy (WS-HFP) for smaller threshold values. This result re-establishes the fact that the work-first policy is well-suited for recursive parallel algorithms with abundant parallelism and small numbers of steals. However, if we increase the threshold to 20 (a value considered to be reasonable for `Fib`), the performance gap between WS-HFP and WS-WFP disappears.

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] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. Comput.*, 38:1526–1538, November 1989. ISSN 0018-9340. URL <http://dx.doi.org/10.1109/12.42122>.
- [4] R. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999. ISSN 0004-5411. URL <http://doi.acm.org/10.1145/324133.324234>.
- [5] 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>.
- [6] Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Christoph von Praun, Vijay Saraswat, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pages 519–538, NY, USA, 2005. ISBN 1-59593-031-0. URL <http://doi.acm.org/10.1145/1094811.1094852>.
- [7] ROBERT H. DENNARD, FRITZ H. GAENSSLEN, HWA-NIEN YU, V. LEO RIDEVOT, ERNEST BASSOUS, and ANDRE R. LEBLANC. Design of ion-implanted mosfet’s with very small physical dimensions. *Solid-State Circuits Newsletter, IEEE*, 12(1):38–50, winter 2007. ISSN 1098-4232. doi: 10.1109/N-SSC.2007.4785543.
- [8] Edsger W. Dijkstra. The structure of the “the”-multiprogramming system. In *Proceedings of the first ACM symposium on Operating System Principles, SOSP '67*, pages 10.1–10.6, New York, NY, USA, 1967. ACM. URL <http://doi.acm.org/10.1145/800001.811672>.
- [9] 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>.
- [10] R. Graham. Bounds for Certain Multiprocessor Anomalies. *Bell System Technical Journal*, (45):1563–1581, 1966.
- [11] Yi Guo. *A Scalable Locality-aware Adaptive Work-stealing Scheduler for Multi-core Task Parallelism*. PhD thesis, Rice University, Aug 2010.
- [12] Rajiv Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. In *Proceedings of the third international conference on Architectural support for programming languages and operating systems, ASPLOS-III*, pages 54–63, New York, NY, USA, 1989. ACM. ISBN 0-89791-300-0. URL <http://doi.acm.org/10.1145/70082.68187>.
- [13] 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>.
- [14] Robert Halstead, JR. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions of Programming Languages and Systems*, 7(4):501–538, October 1985.
- [15] C. A. R. Hoare. Algorithm 63: partition. *Commun. ACM*, 4:321–, July 1961. ISSN 0001-0782. URL <http://doi.acm.org/10.1145/366622.366642>.

- [16] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4:321–, July 1961. ISSN 0001-0782. URL <http://doi.acm.org/10.1145/366622.366644>.
- [17] 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>.
- [18] Calvin Lin and Lawrence Snyder. *Principles of Parallel Programming*. Addison-Wesley, 2009.
- [19] Alex Miller. Set your java 7 phasers to stun, 2008. URL <http://tech.puredanger.com/2008/07/08/java7-phasers/>.
- [20] Gordon E. Moore. Readings in computer architecture. chapter Cramming more components onto integrated circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000. ISBN 1-55860-539-8. URL <http://portal.acm.org/citation.cfm?id=333067.333074>.
- [21] Cherri M. Pancake. Knowing When to Parallelize: Rules-of-Thumb based on User Experiences. URL <http://web.engr.oregonstate.edu/~pancake/presentations/sdsc.pdf>.
- [22] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 277–288, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-158-3. doi: <http://doi.acm.org/10.1145/1375527.1375568>.
- [23] Sagnak Tasirlar. Scheduling macro-dataflow programs on task-parallel runtime systems. M.S. Thesis, Department of Computer Science, Rice University, May 2011 (expected).
- [24] John von Neumann. First draft of a report on the edvac. *IEEE Ann. Hist. Comput.*, 15:27–75, October 1993. ISSN 1058-6180. URL <http://dx.doi.org/10.1109/85.238389>.
- [25] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004. ISBN 0131405632.