COMP 322: Fundamentals of Parallel Programming
Module 1: Deterministic Shared-Memory Parallelism

January 21, 2014

DRAFT VERSION – PLEASE DO NOT DISTRIBUTE

Contents

0 Introduction **2**

0.1 What is Parallel Programming? . . . 2

0.2 Why is Parallel Programming important? . . . 3

1 Task-level Parallelism **6**

1.1 Task Creation and Termination (Async, Finish) . . . 6

1.2 Computation Graphs . . . 9

1.3 Ideal Parallelism . . . 14

1.4 Multiprocessor Scheduling . . . 14

1.5 Parallel Speedup and Amdahl's Law . . . 15

1.6 Parallel Quicksort . . . 16

2 Functional Parallelism and Determinism **28**

2.1 Future Tasks and Functional Parallelism . . . 28

2.2 Memoization . . . 30

2.3 Finish Accumulators . . . 30

2.4 Map Reduce . . . 33

2.5 Data Races . . . 36

2.6 Functional and Structural Determinism . . . 38

0 Introduction

0.1 What is Parallel Programming?

Since the dawn of early digital computers and the Von Neumann computing model [19][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 since it's 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 a parallel program, let us first consider the following sequential program for computing the sum of the elements of an array of integers, X:

This program 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 for the program in Listing 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 the program in Listing 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], which we will study in more detail later:

- The statement "**async** ⟨*stmt*⟩" causes the parent task to create a new child task to execute ⟨*stmt*⟩ *asynchronously* (*i.e.,* before, after, or in parallel) with the remainder of the parent task.
- The statement "**finish** ⟨*stmt*⟩" causes the parent task to execute ⟨*stmt*⟩ and then wait until all async tasks created within ⟨*stmt*⟩ have completed.

The notation, ⟨*stmt*⟩, 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 ⟨*stmt*⟩ too. The use of angle brackets in "⟨*stmt*⟩" follows a standard notational convention to denote units of a program. They are unrelated to the < and > comparison operators used in many programming languages.

[1] These lecture notes include citation such as [19] as references for optional further reading.

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

```
1 int sum = 0;
2 for (int i=0 ; i < X.length ; i++ ) sum += X[i];
```

Listing 1: Sequential ArraySum

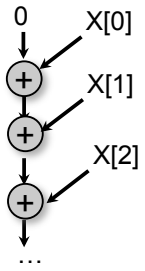

Figure 1: Computation graph for code example in Listing 1 (Sequential ArraySum)

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

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 computers[3] 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 computer 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 becoming important for all computers in the world — even laptops and smart-phones — 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 [18] 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 $\sqrt{2}$ 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

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

```
1 // Start of Task T1 (main program)
2 sum1 = 0; sum2 = 0; // Assume that sum1 & sum2 are fields (not local vars)
3 finish {
4   // Compute sum1 (lower half) and sum2 (upper half) in parallel
5   async for(int i=0 ; i < X.length/2 ; i++) sum1 += X[i]; // Task T2
6   async for(int i=X.length/2 ; i < X.length ; i++) sum2 += X[i]; // Task T3
7 }
8 //Task T1 waits for Tasks T2 and T3 to complete
9 int sum = sum1 + sum2; // Continuation of Task T1
```

Listing 2: Two-way Parallel ArraySum

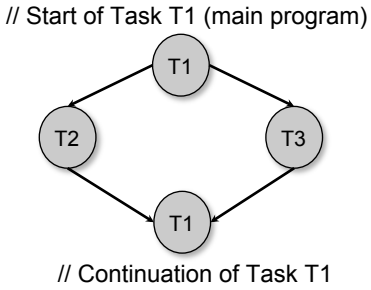

Figure 2: Computation graph for code example in Listing 2 (Two-way Parallel ArraySum)

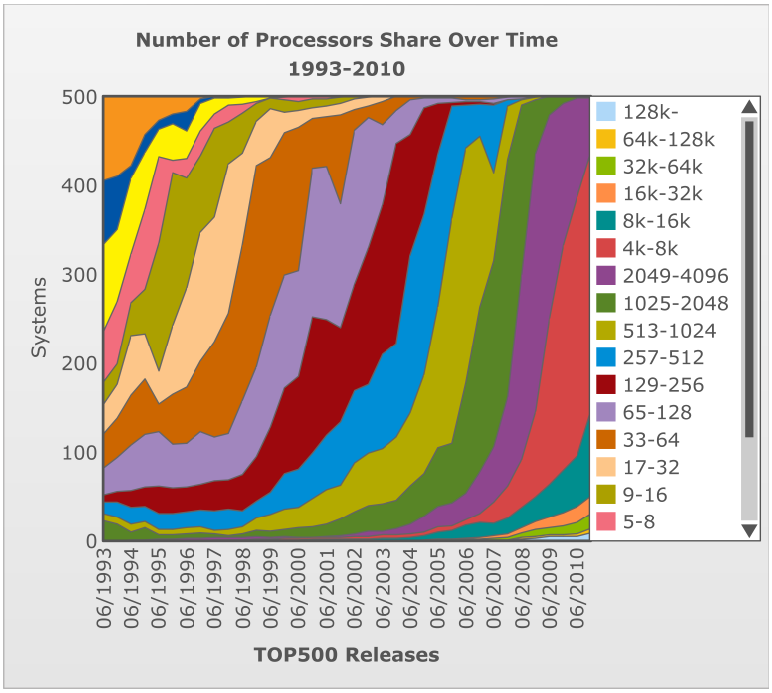

Figure 3: Historical chart showing the distribution of the number of processors used in the Top 500 computers from 1993 to 2010 (source: http://www.top500.org)

using current device technologies varies as the cube of the frequency, $Power \alpha\ (Frequency)^3$. 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. If your laptop has a processor chip named "Intel Core 2 Duo", it means that you have your very own parallel computer at your fingertips. It contains two processor cores that can execute independent instruction streams such as the two async tasks shown in Listing 2. This is a sea change in the history of 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 $Power \alpha\ (Frequency)^3$). 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 A[4]. 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.

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 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. That is why we have created an undergraduate-level course on "Fundamentals of Parallel Programming" that can be taken at the sophomore, or even freshman, level.

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

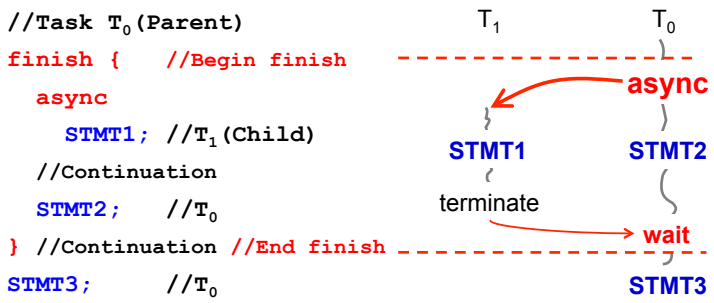

Figure 4: A example code schema with async and finish constructs

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** ⟨*stmt*⟩" causes the parent task to create a new child task to execute ⟨*stmt*⟩ *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 3 shows some example usages of `async`. These examples are illustrative of logical parallelism. Later in the course, you will learn the impact of overheads in determining what subset of ideal parallelism can be useful for a given platform.

Note that all code examples in the module handouts should be treated as "pseudo-code". They were derived from the Habanero-Java research language [5, 12]. However, all programming projects in COMP 322 will be done in standard Java using the newly created Habanero-Java library (HJlib).

In Example 1 in Listing 3, 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. 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). There are no data races in the parallel programs shown in Examples 1–4.

The pattern of parallelizing counted for-loops in Example 1 occurs so commonly in practice that many parallel languages include a separate construct for this case. In HJ, the parallel loop constructs are called `forasync` and `forall`, which you will study later in the course.

In Example 2 in Listing 3, 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 instances of the remainder of the loop body can logically execute in parallel.

Example 3 in Listing 3 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 may not have completed. It's only the enclosing `finish` that awaits the completion of 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.

```
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 3: Example usages of async

As these examples show, an HJ program can create an unbounded number of tasks at runtime. The HJ 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 the role of the OS in task scheduling to that of binding threads to cores at the start of program execution, and let the HJ runtime system take over from that point onwards. 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*. 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 4 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. However, be warned that errors in reading and writing shared

[5]This use of "continuation" is different from continuations in functional programming languages.

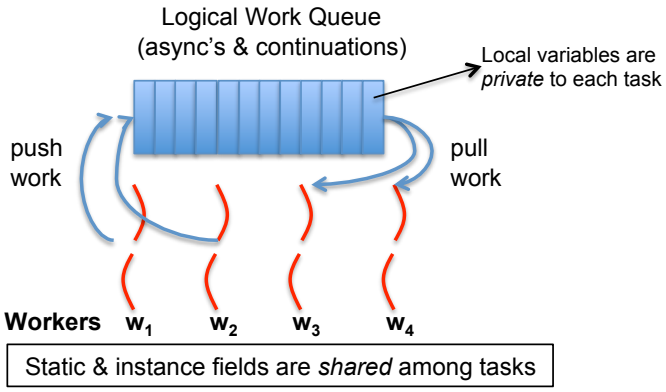

Figure 5: Scheduling an HJ program on a fixed number of workers. (Figure adapted from [10].)

```
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 4: Rules for accessing local variables across async's

fields and arrays among tasks can lead to *data races*, which are notoriously hard to debug.

1.1.2 Finish notation for Task Termination

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** ⟨*stmt*⟩" causes the parent task to execute ⟨*stmt*⟩ and then wait until all async tasks created within ⟨*stmt*⟩ 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 `main()` 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.) Also, 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 5 extends the first two code examples from Listing 3 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.

`async` and `finish` statements also jointly define what statements can potentially be executed in parallel with each other. Consider the HJ code fragment and finish-async nesting structure shown in Figure 6:

Figure 6 reveals which pairs of statements can potentially execute in parallel with each other. For example, task $A2$ can potentially execute in parallel with tasks $A3$ and $A4$ since `async` $A2$ was launched before entering the finish $F2$, which is the Immediately Enclosing Finish for $A3$ and $A4$. However, Part 3 of Task $A0$ cannot execute in parallel with tasks $A3$ and $A4$ since it is performed after finish $F2$ is completed.

1.1.3 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 7, which is Listing 6 extended with two try-catch blocks. If async task $A3$ 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 $A2$. 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.

1.2 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

```
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 5: 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 6: Example usage of async and finish

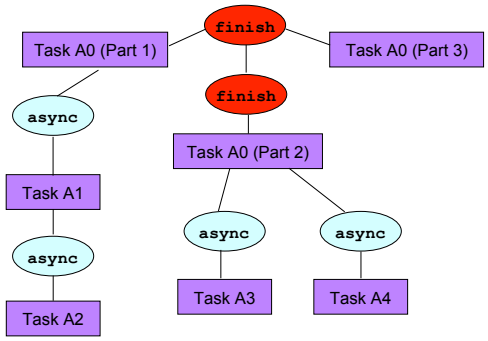

Figure 6: Finish-async nesting structure for code fragment in Listing 6

```
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 7: Extension of Listing 6 with try-catch blocks

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 HJ 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 an HJ 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 [10]:
  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 HJ program shown in Listing 8 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 HJ 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 $v3 \ldots v15$ can potentially execute in parallel with node $v16$ because there is no directed path in the CG from $v16$ to any node in $v3 \ldots v15$ or vice versa.

It is also important to observe that the CG in Figure 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.

In addition to providing the dependence structure of a parallel program, Computation Graphs can also be used to reason about the *execution time complexity* 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.)

```
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 8: Example HJ program

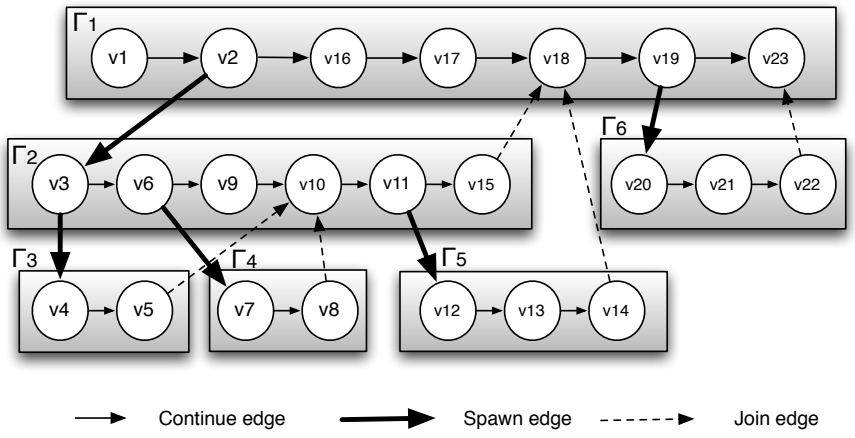

Figure 7: Computation Graph G for example HJ program in Listing 8

- 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:

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

1.3 Ideal Parallelism

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

1.4 Multiprocessor Scheduling

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

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

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

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

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

- $t_1 = WORK(\text{G})$. Any greedy scheduler executing on 1 processor will simply execute all nodes in the CG in some order, thereby ensuring that the 1-processor execution time equals the total work in the CG.

- $t_\infty = CPL(G)$. Any greedy scheduler executing with an unbounded (infinite) number of processors must complete its execution with time = $CPL(G)$, since all nodes can be scheduled as early as possible.
- $t_P \leq t_1/P + t_\infty = WORK(G)/P + CPL(G)$. This is a classic result due to Graham [9]. 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:

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

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

1.4.1 HJ 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 HJ programs. HJ includes the following utilities to help programmers reason about the CGs for their programs:

- *Insertion of calls to* `perf.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.
- *Execution with the* `-perf=true` *option.* If an HJ program is executed with this option, abstract metrics are printed at 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 speedup* for the abstract metrics.

1.5 Parallel Speedup and Amdahl's Law

Given definitions for t_1 and t_P, the speedup for a given schedule of a computation graph on P processors is defined as $Speedup(P) = t_1/t_P$. *Speedup(P)* is the factor by which the use of P processors speeds up execution time relative to 1 processor, for a fixed input size For ideal executions without overhead, $1 \leq Speedup(P) \leq P$. The term *linear speedup* is used for a program when $Speedup(P) = k \times P$ as P varies, for some constant k, $0 < k < 1$.

We conclude this unit with a simple observation made by Gene Amdahl in 1967 [1] that can be summarized as follows: if $q \leq 1$ is the fraction of $WORK$ in a parallel program that must be executed *sequentially*, then the best speedup that can be obtained for that program, even with an unbounded number of processors, is $Speedup \leq 1/q$. As in the Computation Graph model studied earlier, this observation assumes that all processors are *uniform i.e.,* they all execute at the same speed.

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

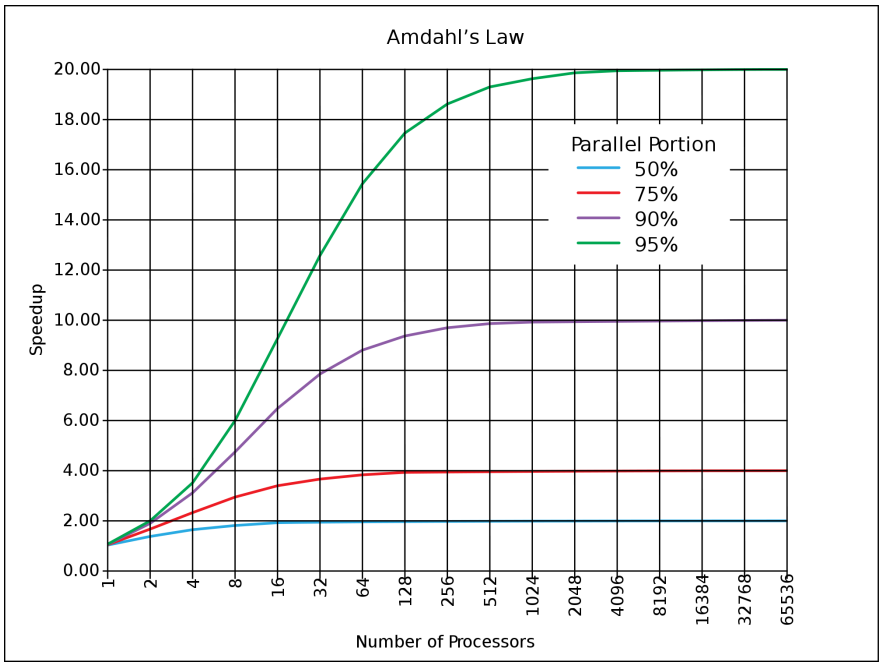

Figure 8: Illustration of Amdahl's Law (source: http://en.wikipedia.org/wiki/Amdahl's_law)

$CPL(G) \geq qWORK$. Therefore, $Speedup = t_1/t_P$ must be $\leq WORK/(qWORK) = 1/q$ since $t_1 = WORK$ 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 [11] 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

1.6.1 Parallelizing Quicksort: Approach 1

Quicksort is a classical sequential sorting algorithm introduced by C.A.R. Hoare in 1961 [15], 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.

```
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: [15])

As a historical note, Figure 9 contains a facsimile of the exact description of the original Quicksort algorithm from 1961 [15]. (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 $A[M \ldots N]$ 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, $A[M \ldots N]$ into disjoint sub-arrays, $A[M \ldots J]$ and $A[I \ldots N]$ (where $M \leq J < I \leq N$) such that all elements in $A[M \ldots N]$ are guaranteed to be $\leq$ all elements in $A[J+1 \ldots I-1]$, which in turn are guaranteed to be $\leq$ all elements in $A[I \ldots N]$. A facsimile of the original specification of the *partition* procedure is included in Figure 10. It uses a randomly selected element in $A[M \ldots N]$ to serve as the *pivot* for partitioning the array.

Conquer: The two recursive calls to *quicksort* operate on sub-arrays $A[M \ldots J]$ and $A[I \ldots N]$. 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 $A[M \ldots N]$.

Listing 9 shows a sequential implementation of the *quicksort()* and *partition()* procedures from Figures 9 and 10. The *partition()* procedures uses the *point* data type to return the `I` and `J` values in a single object, and also replaces the `go to` statements by `while`, `continue`, and `break` statements. This code initializes $I = M$ and $J = N$. Line 16 repeatedly increments I while scanning elements from left-to-right (moving "up") that are $\leq$ the pivot value. Similarly, line 17 repeatedly decrements J while scanning elements from right-to-left (moving "down") that are $\geq$ the pivot value. After this, an *exchange* is performed, and the `while` loop in line 15 is repeated as needed. If one of the conditions in line 20 or line 21 is satisfied, control exits from the `while` loop to the `return` statement in line 23.

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 with each other as shown in Listing 10.

What is the parallel execution time complexity of the program in Listing 10? For simplicity, let us assume perfect pivoting in this analysis 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

Description of procedure partition(A,M,N,I,J) from [14]:

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:

$M \leqq J < I \leqq N$ provided $M < N$
$A[R] \leqq X$ for $M \leqq R \leqq J$
$A[R] = X$ for $J < R < I$
$A[R] \geqq X$ for $I \leqq R \leqq N$

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: [14])

```
1  static void quicksort(int[] A, int M, int N) {
2    if (M < N) {
3      point p = partition(A, M, N); int I=p.get(0); int J=p.get(1);
4      quicksort(A, M, J);
5      quicksort(A, I, N);
6    }
7  } //quicksort
8
9  static point partition(int[] A, int M, int N) {
10   int I, J;
11   Random rand = new Random();
12   // Assign pivot to random integer in M..N
13   int pivot = M + rand.nextInt(N-M+1);
14   I = M; J = N;
15   while (true) {
16     /* up */ while (I<= N && A[I] <= A[pivot]) I++; if (I > N) I = N;
17     /* down */ while (J>= M && A[J] >= A[pivot]) J--; if (J < M) J = M;
18     /* change */
19     if (I < J) { exchange(A, I, J); I++; J--; continue; }
20     else if (I < pivot) { exchange(A, I, pivot); I++; break; }
21     else if (pivot < J) { exchange(A, pivot, J); J--; break; }
22     else break;
23   } // while
24   return [I, J];
25 } // partition
26
27 static void exchange(int[] A, int x, int y) {
28   int temp = A[x]; A[x] = A[y]; A[y] = temp;
29 }
```

Listing 9: procedure partition() from [14]

```
1 static void quicksort(int[] A, int M, int N) {
2   if (M < N) {
3     point p = partition(A, M, N); int I=p.get(0); int J=p.get(1);
4     async quicksort(A, M, J);
5     async quicksort(A, I, N);
6   }
7 } //quicksort
```

Listing 10: Parallelization of recursive calls in procedure quicksort() from [15]

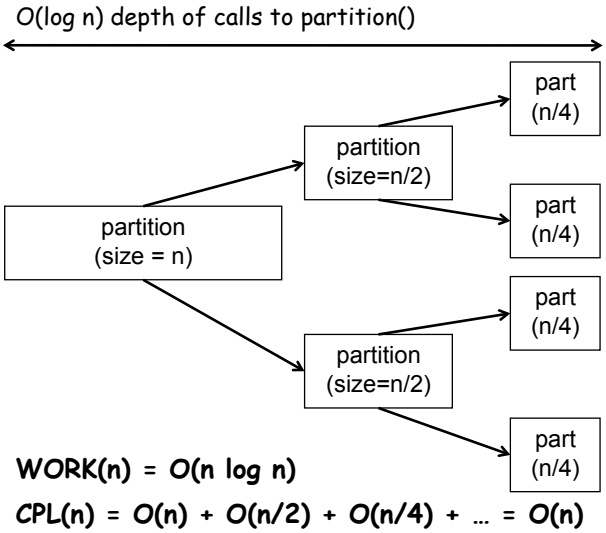

Figure 11: Computation Graph for Approach 1

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 $CPL(n) = O(n) + O(n/2) + O(n/4) + \ldots = O(n)$, 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 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 algorithm in Listing 9 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 11.

The program in Listing 11 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+pivot]`, `A[M+k] == A[M+pivot]` or `A[M+k] > A[M+pivot]` respectively. The `forall` also copies sub-array $A[M \ldots N]$ into $buffer[0 \ldots N - M + 1]$. We can use the parallel prefix sum algorithm described below in Section 1.6.5 to obtain a parallel implementation of the `computePrefixSums()`

```
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 11: Parallel algorithm for procedure partition()

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 $A[M \ldots N]$ so as to pack the sub-arrays containing $<$, $=$, or $>$ values relative to the pivot value. For the $<$ case, line 18 copies *buffer*$[k]$ into $A[M + lt[k] - 1]$. 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, 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 [14], 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 is as inefficient as Approach 1

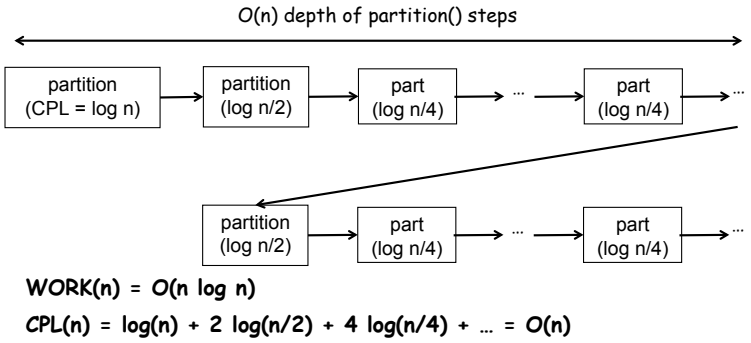

Figure 12: Computation Graph for Approach 2

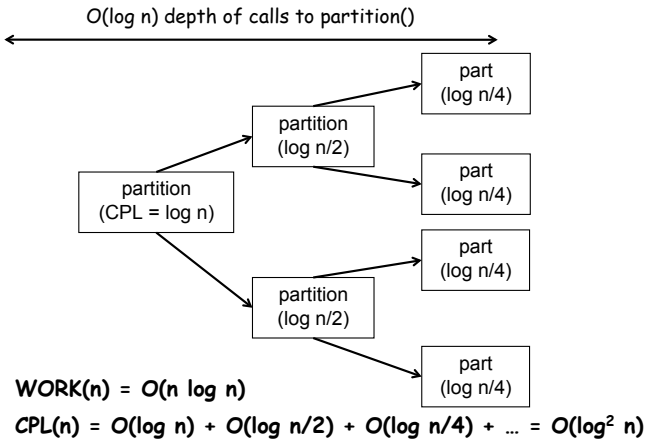

Figure 13: Computation Graph for Approach 3

1.6.3 Parallelizing Quicksort: Approach 3

Approaches 1 and 2 to 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 Listing 10 with the parallel version of *partition* in Listing 11? 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 $\log(n) + \log(n/2) + \log(n/4) + \ldots = O(\log^2 n)$. 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 program that you saw in Section 0.1, Listing 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.

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 12:

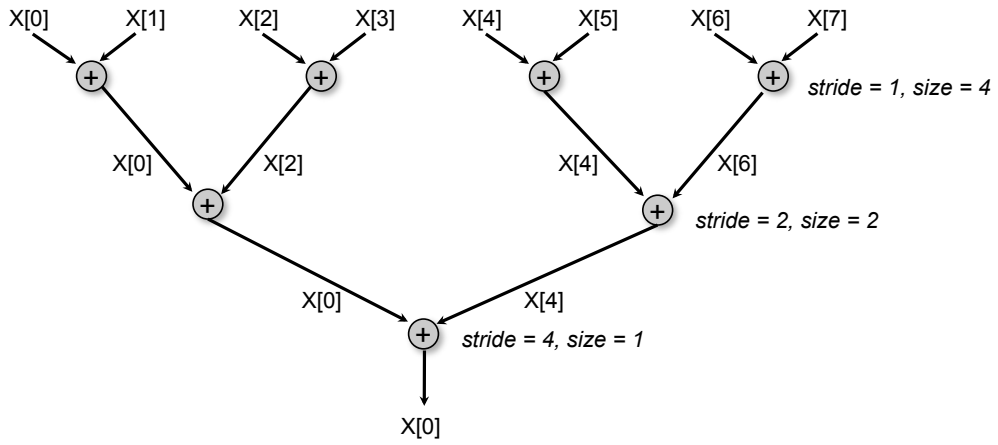

Figure 14: Reduction tree schema for computing ArraySum in parallel, assuming X.length=8. Note that elements of the input array are over-written with partial sums, and the final sum is written in X[0].

```
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 12: Async-Finish code to implement reduction tree schema in Figure 14

```
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 13: Async-Finish code to implement schema in Figure 13 for arbitrarily sized arrays

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 12 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 12 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 13. 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 $\lceil x/y \rceil$.) 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.

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 13 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 13 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

```
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 14: Sequential Prefix Sum

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

Listing 15: Inefficient Parallel Prefix Sum program

will be $O(n)/O(n/p + \log p)$. If we set $p = n/(\log n)$, then the resulting parallelism will be $O(n)/O(\log n + \log n - \log\log n) = O(n/(\log n)) = O(p)$, 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 13 is *optimal* for $p = n/(\log n)$ or fewer processors. We will formalize this notion of optimality later in the course.

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 a useful building block in a wide range of algorithms such as team standings, index of last occurrence, lexical string comparison, and multi-precision arithmetic [3, 17].

The Prefix Sum problem can be stated as follows:

> Given an input array A, compute an output array X as follows (where $n = A.length$):
>
> $$X[i] = \sum_{0 \le j \le i} A[j] \tag{1}$$

Thus, each output element, $X[i]$, is the sum of the *prefix* of array A consisting of elements $A[0 \ldots i]$. 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 14 computes the desired prefix sums in $O(n)$ time.

The program in Listing 14 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 15 are computed in parallel, the parallel execution time for computing all output elements 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 15 is $\sum_{0 \le i < n} i = O(n^2)$. 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.

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.

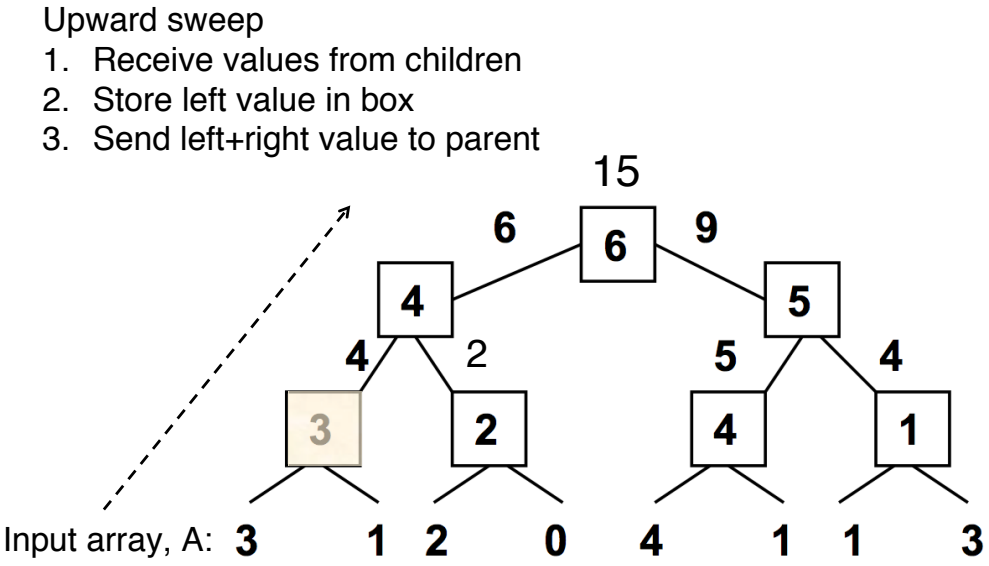

Figure 15: Upward sweep for Parallel Prefix algorithm

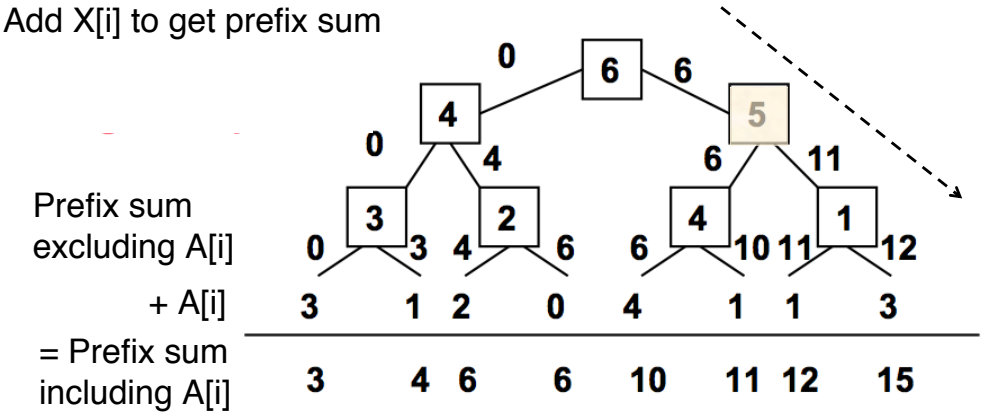

Figure 16: Downward sweep for Parallel Prefix algorithm

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. For example, an *ancestor* task, T_A, uses a `finish` to ensure that it is safe to read values computed by *all descendant tasks*, T_D. These values are communicated from T_D to T_A via shared variables, which in the case of HJ 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>` is a reference to a *future* object *i.e.,* a container for a return value of type `T` from an `async` task. An important constraint in HJ is that *all variables of type* `future<T>` *must be declared with a* `final` *modifier*, thereby ensuring that the value of the reference cannot change after initialization. This constraint 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 16 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 it parent, just like regular `async`'s. However, unlike regular `async`'s, there is no `finish` construct needed for this example since the parent task $T1$, performs `sum1.get()` to wait for future task $T2$ and `sum2.get()` to wait for future task $T3$.

In addition to waiting for completion, the `get()` operations are also used to access the return values of the future tasks. This is an elegant capability because it obviates the need for shared fields as in the example with regular `async`'s, and avoids the possibility of race conditions on `sum1` and `sum2`. Notice the three declarations for variables `sum` in lines 4, 9, and 14. Each occurrence of `sum` is local to a task, and there's no possibility of race conditions on local variables either. These properties have historically made future tasks well suited to express parallelism in functional languages [13].

```
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 16: Two-way Parallel ArraySum using Future Tasks

2.1.2 Computation Graph Extensions for Future Tasks

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

- A `get()` operation is a new kind of *continuation* operation, since it can involve a blocking operation. Thus, `get()` operations can only occur on the boundaries of steps. To fully realize this constraint, it may be necessary to split a statement containing one or more `get()` operations into multiple sub-statements such that a `get()` occurs in a sub-statement by itself. This splitting can always be performed by following HJ's expression evaluation rules, which are the same as those for Java.
- 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 16, 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 must Future References be declared as final?

In the previous section, we stated the rule that all variables containing references to future objects must be declared as final. This means that the variable cannot be modified after initialization. To motivate why this rule was added to HJ, consider the illegal program example in Listing 17. *WARNING: this is an example of bad parallel programming practice that is not permitted in HJ.*

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

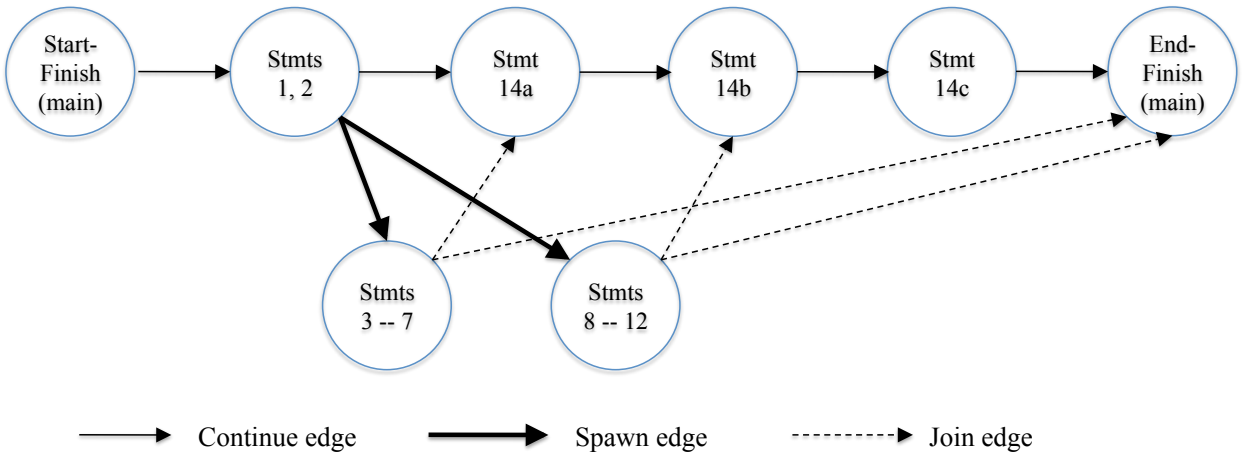

Figure 17: Computation Graph G for example HJ program in code:TwoParArraySumFuture, with statement 14 split into statements 14a, 14b, 14c

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

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, HJ also offers 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 `future` constructs, and Listing 18 shows the HJ 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 an HJ program.

2.2 Memoization

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

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

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

2.3 Finish Accumulators

In this section, we introduce the programming interface and semantics of *finish accumulators* in HJ. Finish accumulators support parallel reductions, which represent a common pattern for computing the aggregation

```
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 17: Illegal Use of Future Tasks in HJ due to missing final declarations

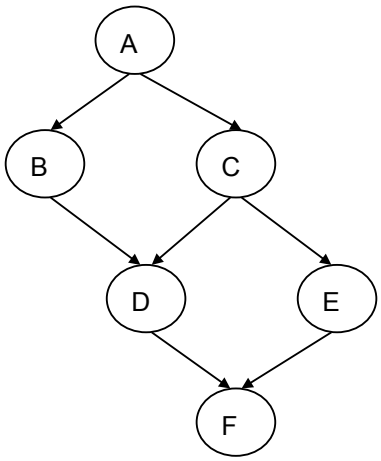

Figure 18: Computation Graph $G3$

```
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 18: HJ code to generate Computation Graph $G3$ 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: accumulator API

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 the 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 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

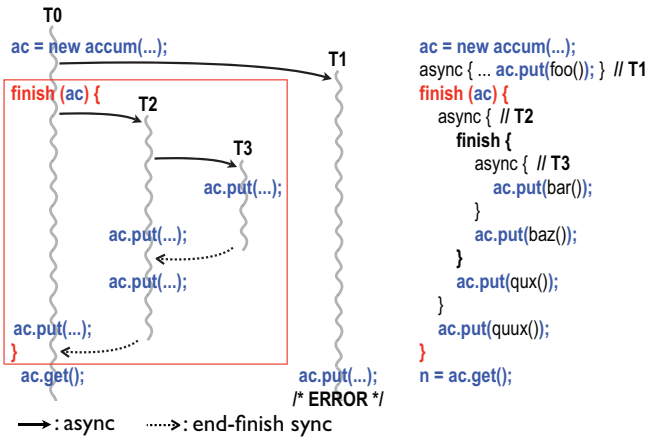

Figure 20: Finish accumulator example with three tasks that perform a correct reduction and one that throws an exception

`finish` scope. Note that `put` operations performed by the owner outside associated `finish` scopes are immediately reflected in any subsequent `get` operations since those results are deterministic.

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

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

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

Many data analysis algorithm can be specified as sequences of *map* and *reduce* operations on sets of key-

```
 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

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

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

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

Listing 19 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 20 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 19: 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 20: 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 21: 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 21. The parallel version contains an error because the writes to instances of `p.x` in line 7 can potentially execute in parallel with the reads of instances of `p.x` in line 9. This can be confirmed by building a computation graph for the program and observing that there is no chain of dependence edges from step instances of line 7 to step instances of line 9. As a result, it is unclear whether a read in line 9 will receive an older value of `p.x` or the value written in line 7. This kind of situation, where the outcome depends on the relative completion times of two events, is called a *race condition*. When the race condition applies to read and write accesses on a shared location, it is called a *data race*. A shared location must be a static field, instance field or array element, since it is not possible for interfering accesses to occur in parallel on a local variable in a method.

Data races are a challenging source of errors in parallel programming, since it is usually impossible to guarantee that all possible orderings of the accesses to a location will be encountered during program testing. Regardless of how many tests you write, so long as there is one ordering that yields the correct answer it is always possible that the correct ordering in encountered when testing your program and an incorrect ordering is encountered when the program executes in a production setting. For example, while testing the program, it is possible that the task scheduler executes all the `async` tasks in line 7 of Listing 21 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 22: 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 4 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 22. 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 23: 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 23, 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 23, 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 in HJ.

2.6 Functional and Structural Determinism

A computation is said to be *functionally deterministic with respect to its inputs* 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. The presence of data races often leads to 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 is not sufficient to guarantee determinism. However, the parallel constructs introduced in this module ("Module 1: Deterministic Shared-Memory Parallelism") were carefully selected to ensure the following *Structural Determinism Property*:

```
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 24: Example of a parallel program with data races

> 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 deterministic with respect to its inputs. Further, the final computation graph is also guaranteed to be the same for all executions of the program with the same input.

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 ("Nondeterministic Shared-Memory Parallelism").

The additional guarantee that the final computation graph will also be the same for all executions of the program with the same input is the reason for referring to this property as "Structural Determinism". This additional guarantee has also been referred to as "determinacy" in past work.

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 24. 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 [16] 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".

```
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 25: Rewrite of Task T3 from Listing 24

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 mainstream application programmers. For example, suppose an application programmer decided to rewrite the body of Task T3 as shown in Listing 25. 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 [8] 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 24, 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 [4] makes the following assumption about data races:

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

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

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

Why should you care about these memory models if you write bug-free code without data races? Because the code that you write may be used in conjunction with other code that causes your code to participate in a data race. For example, if your job to provide a sequential method that implements the body of Task $T3$ in Listing 24, 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 25 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.

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] 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`.

[5] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-Java: the New Adventures of Old X10. In *PPPJ'11: Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java*, 2011.

[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 RIDEOVT, 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] 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`.

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

[10] Yi Guo. *A Scalable Locality-aware Adaptive Work-stealing Scheduler for Multi-core Task Parallelism.* PhD thesis, Department of Computer Science, Rice University, August 2010.

[11] 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`.

[12] Habanero. Habanero Java programming language. http://habanero.rice.edu/hj, Dec 2009.

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

[14] 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`.

[15] 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`.

[16] 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`.

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

[18] 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`.

[19] 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`.