COMP 322: Fundamentals of Parallel Programming
Module 1: Parallelism

©2016 by Vivek Sarkar

January 22, 2016

DRAFT VERSION

Contents

0 Course Organization 2

1 Task-level Parallelism 2
  1.1 Task Creation and Termination (Async, Finish) 2
  1.2 Computation Graphs 6
  1.3 Ideal Parallelism 10
  1.4 Multiprocessor Scheduling 12
  1.5 Parallel Speedup and Amdahl’s Law 13

2 Functional Parallelism and Determinism 15
  2.1 Future Tasks and Functional Parallelism 15
  2.2 Memoization 17
  2.3 Finish Accumulators 19
  2.4 Map Reduce 20
  2.5 Data Races 23
  2.6 Functional and Structural Determinism 25
0 Course Organization

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

- **Module 1: Parallelism** — creation and coordination of parallelism (async, finish), abstract performance metrics (work, critical paths), Amdahl’s Law, weak vs. strong scaling, data races and determinism, data race avoidance (immutability, futures, accumulators, dataflow), deadlock avoidance, abstract vs. real performance (granularity, scalability), collective and point-to-point synchronization (phasers, barriers), parallel algorithms, systolic algorithms.

- **Module 2: Concurrency** — critical sections, atomicity, isolation, high level data races, nondeterminism, linearizability, liveness/progress guarantees, actors, request-response parallelism, Java Concurrency, locks, condition variables, semaphores, memory consistency models.

- **Module 3: Locality and Distribution** — memory hierarchies, locality, cache affinity, data movement, message-passing (MPI), communication overheads (bandwidth, latency), MapReduce, accelerators, GPGPUs, CUDA, OpenCL.

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

1 Task-level Parallelism

1.1 Task Creation and Termination (Async, Finish)

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

```
Algorithm 1: Sequential ArraySum

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

sum ← 0;
for i ← 0 to X.length − 1 do
    sum ← sum + X[i];
return sum;
```

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

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

---

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

1.1.1 Async notation for Task Creation

The first parallel programming construct that we will learn is called async. In pseudocode notation, “async ⟨stmt1⟩”, causes the parent task (i.e., the task executing the async statement to create a new child task to execute the body of the async, ⟨stmt1⟩, asynchronously (i.e., before, after, or in parallel) with the remainder of the parent task. The notation, ⟨stmt⟩, refers to any legal program statement e.g., if-then-else, for-loop, method call, or a block enclosed in { } braces. (The use of angle brackets in “⟨stmt⟩” follows a standard notational convention to denote units of a program. They are unrelated to the < and > comparison operators used in many programming languages.) Figure 2 illustrates this concept by showing a code schema in which the parent task, T0, uses an async construct to create a child task T1. Thus, STMT1 in task T1 can potentially execute in parallel with STMT2 in task T0.

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

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

```
//Task T0 (Parent)
finish { //Begin finish
  async
    STMT1; //T1 (Child)
    //Continuation
    STMT2; //T0
} //Continuation //End finish
STMT3; //T0
```

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

---

2 These constructs have some similarities to the “fork” and “join” constructs available in many languages, including Java’s ForkJoin framework (which we will learn later in the course), but there are notable differences.
The key idea is to use the `async` keyword to create a new task for each loop iteration. The body of each task is then executed in parallel. This approach is particularly useful for loops where the loop body is computationally intensive but the loop index is not expensive to compute. By using the `async` keyword, we can ensure that each iteration is executed concurrently, allowing us to make better use of the available CPU resources.

In Example 2, the `async` keyword is used to parallelize computations in the body of a pointer-chasing while loop. Though the sequence of `p = p.next` statements is executed sequentially in the parent task, all dynamic instances of the remainder of the loop body can logically execute in parallel with each other. Example 3 shows the computation in Example 2 rewritten as a static void recursive method. You should first convince yourself that the computations in Examples 2 and 3 perform the same operations by omitting the `async` keyword in each case, and comparing the resulting sequential versions.

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

As these examples show, a parallel program can create an unbounded number of tasks at runtime. The parallel runtime system is responsible for scheduling these tasks on a fixed number of processors. It does so by creating a fixed number of worker threads as shown in Figure 3, typically one worker per processor core. Worker threads are allocated by the Operating System (OS). By creating one thread per core, we limit the role of the OS in task scheduling to that of binding threads to cores at the start of program execution, and let the parallel runtime system take over from that point onwards. These workers repeatedly pull work from a work queue and execute the assigned tasks.
from a shared work queue when they are idle, and push work on the queue when they generate more work. The work queue entries can include async’s and continuations. An async is the creation of a new task, such as $T_1$ in Figure 2. A continuation represents a potential suspension point for a task, which (as shown in Figure 2) can include the point after an async creation as well as the point following the end of a finish scope. Continuations are also referred to as task-switching points, because they are program points at which a worker may switch execution between different tasks. A key motivation for this separation between tasks and threads is that it would be prohibitively expensive to create a new OS-level worker thread for each async task that is created in the program.

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

3 This use of “continuation” is related to, but different from, continuations in functional programming languages.
1.1.2 Finish notation for Task Termination

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

Thus, the finish statement in Figure 2 is used by task T₀ to ensure that child task T₁ has completed executing STMT1 before T₀ executes STMT3. This may be necessary if STMT3 in Figure 2 used a value computed by STMT1. If T₁ created a child async task, T₂ (a “grandchild” of T₀), T₀ will wait for both T₁ and T₂ to complete in the finish scope before executing STMT3.

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

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

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

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

1.1.3 Array Sum with two-way parallelism

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

1.2 Computation Graphs

A Computation Graph (CG) is a formal structure that captures the meaning of a parallel program’s execution. When you learned sequential programming, you were taught that a program’s execution could be understood as a sequence of operations that occur in a well-defined total order, such as the left-to-right evaluation order for expressions. Since operations in a parallel program do not occur in a fixed order, some other abstraction is needed to understand the execution of parallel programs. Computation Graphs address this need by focusing on the extensions required to model parallelism as a partial order. Specifically, a Computation
Listing 3: Incorrect and correct parallelization with async and finish
```javascript
// F1
// Part 1 of Task A0
async { A1; async A2; }

// F2
// Part 2 of Task A0
async A3;
async A4;

// Part 3 of Task A0
}
```

Listing 4: Example usage of async and finish

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

// Start of Task T1 (main program)

Figure 5: Computation graph for code example in Algorithm 5 (Two-way Parallel ArraySum)
Algorithm 2: Two-way Parallel ArraySum

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

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

Graph consists of:

- A set of nodes, where each node represents a step consisting of an arbitrary sequential computation. For programs with async and finish constructs, a task’s execution can be divided into steps by using continuations to define the boundary points. Recall from Section 1.1.1 that a continuation point in a task is the point after an async creation or a point following the end of a finish scope. It is acceptable to introduce finer-grained steps in the CG if so desired i.e., to split a step into smaller steps. The key constraint is that a step should not contain any parallelism or synchronization i.e., a continuation point should not be internal to a step.

- A set of directed edges that represent ordering constraints among steps. For async-finish programs, it is useful to partition the edges into three cases:

  1. Continue edges that capture sequencing of steps within a task — all steps within the same task are connected by a chain of continue edges.
  2. Spawn edges that connect parent tasks to child async tasks. When an async is created, a spawn edge is inserted between the step that ends with the async in the parent task and the step that starts the async body in the new child task.
  3. Join edges that connect descendant tasks to their Immediately Enclosing Finish (IEF) operations. When an async terminates, a join edge is inserted from the last step in the async to the step in the ancestor task that follows the IEF operation.

Consider the example program shown in Listing 5 and its Computation Graph shown in Figure 6. There are 6 tasks in the CG, T1 to T6. This example uses finer-grained steps than needed, since some steps (e.g., v1 and v2) could have have been combined into a single step. In general, the CG grows as a program executes and a complete CG is only available when the entire program has terminated. The three classes of edges (continue, spawn, join) are shown in Figure 6. Even though they arise from different constructs, they all have the same effect viz., to enforce an ordering among the steps as dictated by the program.
In any execution of the CG on a parallel machine, a basic rule that must be obeyed is that a successor node \( B \) of an edge \((A, B)\) can only execute after its predecessor node \( A \) has completed. This relationship between nodes \( A \) and \( B \) is referred to as a *dependence* because the execution of node \( B \) depends on the execution of node \( A \) having completed. In general, node \( Y \) depends on node \( X \) if there is a path of directed edges from \( X \) to \( Y \) in the CG. Therefore, dependence is a *transitive* relation: if \( B \) depends on \( A \) and \( C \) depends on \( B \), then \( C \) must depend on \( A \). The CG can be used to determine if two nodes may execute in parallel with each other. For example, an examination of Figure 6 shows that all of nodes \( v_{3} \ldots v_{15} \) can potentially execute in parallel with node \( v_{16} \) because there is no directed path in the CG from \( v_{16} \) to any node in \( v_{3} \ldots v_{15} \) or vice versa.

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

### 1.3 Ideal Parallelism

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

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

- Define \( WORK(G) \) to be the sum of the execution times of the nodes in CG \( G \),

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

Thus, \( WORK(G) \) represents the total amount of work to be done in CG \( G \).

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

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

\[
v_{1} \rightarrow v_{2} \rightarrow v_{3} \rightarrow v_{6} \rightarrow v_{7} \rightarrow v_{8} \rightarrow v_{10} \rightarrow v_{11} \rightarrow v_{12} \rightarrow v_{13} \rightarrow v_{14} \rightarrow v_{18} \rightarrow v_{19} \rightarrow v_{20} \rightarrow v_{21} \rightarrow v_{22} \rightarrow v_{23}
\]

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

### 1.3.1 Abstract Performance Metrics

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

- *Insertion of calls to doWork().* The programmer can insert a call of the form `perf.doWork(N)` anywhere in a step to indicate execution of \( N \) application-specific abstract operations e.g., floating-point...
// Task T1
v1; v2;
finish {
  async {
    // Task T2
    v3;
    finish {
      async { v4; v5; } // Task T3
      v6;
      async { v7; v8; } // Task T4
      v9;
    } // finish
    v10; v11;
    async { v12; v13; v14; } // Task T5
    v15;
  }
  v16; v17;
} // finish
v18; v19;
finish {
  async { v20; v21; v22; }
}
operations, comparison operations, stencil operations, or any other data structure operations. Multiple calls to `perf.doWork()` are permitted within the same step. They have the effect of adding to the abstract execution time of that step. The main advantage of using abstract execution times is that the performance metrics will be the same regardless of which physical machine the HJ program is executed on. The main disadvantage is that the abstraction may not be representative of actual performance on a given machine.

- **Printout of abstract metrics.** If an HJlib program is executed with a specified option, abstract metrics are printed at the end of program execution that capture the total number of operations executed (`WORK`) and the critical path length (`CPL`) of the CG generated by the program execution. The ratio, `WORK/CPL` is also printed as a measure of ideal parallelism.

- **Visualization of computation graph.** A tool called HJ-viz is also provided that enables you to see an image of the computation graph of a program executed with abstract performance metrics.

### 1.4 Multiprocessor Scheduling

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

1. **Capacity bound:** `t_P \geq WORK(G)/P`. It is not possible for a schedule to complete in time less than `WORK(G)/P` because that’s how long it would take if all the work was perfectly divided among `P` processors.

2. **Critical path bound:** `t_P \geq CPL(G)`. It is not possible for a schedule to complete in time less than `CPL(G)` because any legal schedule must obey the chain of dependences that form a critical path.

Note that the critical path bound does not depend on `P`.

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

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

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

- **`t_1 = WORK(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 [10]. An informal sketch of the proof is as follows. At any given time in the schedule, we can declare the time slot to be complete if all `P` processors are busy at that time and incomplete otherwise. The number of complete time slots must add up to at most `t_1/P` since each such time slot performs `P` units of work.
In addition, the number of incomplete time slots must add up to at most \( t_\infty \) 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:

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

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

### 1.5 Parallel Speedup and Amdahl’s Law

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

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

This observation follows directly from a lower bound on parallel execution time that you are familiar with, namely \( t_P \geq \text{CPL}(G) \), where \( t_P \) is the execution time of computation graph \( G \) on \( P \) processors and \( \text{CPL} \) is the critical path length of graph \( G \). If fraction \( q \) of \( \text{WORK}(G) \) is sequential, it must be the case that \( \text{CPL}(G) \geq q \times \text{WORK}(G) \). Therefore, \( \text{Speedup}(P) = t_1/t_P \) must be \( \leq \text{WORK}(G)/(q \times \text{WORK}(G)) = 1/q \) since \( t_1 = \text{WORK}(G) \) for greedy schedulers.
The consequence of Amdahl’s Law is illustrated in Figure 7. The x-axis shows the number of processors increasing in powers of 2 on a log scale, and the y-axis represents speedup obtained for different values of \( q \). Specifically, each curve represents a different value for the parallel portion, \((1 - q)\), assuming that all the non-sequential work can be perfectly parallelized. Even when the parallel portion is as high as 95\%, the maximum speedup we can obtain is 20\( \times \) since the sequential portion is 5\%. The ideal case of \( q = 0 \) and a parallel portion of 100\% is not shown in the figure, but would correspond to the \( y = x \) line which would appear to be an exponential curve since the x-axis is plotted on a log scale.

Amdahl’s Law reminds us to watch out for sequential bottlenecks both when designing parallel algorithms and when implementing programs on real machines. While it may paint a bleak picture of the utility of adding more processors to a parallel computing, it has also been observed that increasing the data size for a parallel program can reduce the sequential portion \([13]\) thereby making it profitable to utilize more processors. The ability to increase speedup by increasing the number of processors for a fixed input size (fixed WORK) is referred to as strong scaling, and the ability to increase speedup by increasing the input size (increasing WORK) is referred to as weak scaling.
2 Functional Parallelism and Determinism

2.1 Future Tasks and Functional Parallelism

2.1.1 Tasks with Return Values

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

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

- A variable of type $\text{future<T>}$ is a reference to a future object i.e., a container for a return value of type $T$ from an async task.
- There are exactly two operations that can be performed on a variable, $V_1$, of type $\text{future<T>}$, assuming that type $T_2$ is a subtype of, or the same as, type $T_1$:
  1. Assignment — variable $V_1$ can be assigned a reference to an async with return value type $T_2$ as described below, or $V_1$ can be assigned the value of a variable $V_2$ with type $\text{future<T_2>}$.  
  2. Blocking read — the operation, $V_1.get()$, waits until the async referred to by $V_1$ has completed, and then propagates the return value of the async to the caller as a value of type $T_1$. 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, $\text{async<T_1>}$, in which the type of the async’s return value, $T_1$, immediately follows the async keyword.
  2. The body itself must consist of a compound statement enclosed in { } braces, dynamically terminating with a return statement. It is important to note that the purpose of this return statement is to communicate the return value of the enclosing async and not the enclosing method.

Listing 6 revisits the two-way parallel array sum example discussed earlier, but using future tasks instead of regular async’s. There are two variables of type $\text{future<int>}$ in this example, $\text{sum1}$ and $\text{sum2}$. Each future task can potentially execute in parallel with it parent, just like regular async’s. However, unlike regular async’s, there is no finish construct needed for this example since the parent task $T_1$, performs $\text{sum1.get()}$ to wait for future task $T_2$ and $\text{sum2.get()}$ to wait for future task $T_3$.

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

---

4 "future" is a pseudocode keyword, and will need be replaced by the appropriate data type in real code.
2.1.2 Computation Graph Extensions for Future Tasks

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

- A `get()` operation is a new kind of *continuation* operation, since it can involve a blocking operation. Thus, `get()` operations can only occur on the boundaries of steps. To fully realize this constraint, it may be necessary to split a statement containing one or more `get()` operations into multiple sub-statements such that a `get()` occurs in a sub-statement by itself.

- A `spawn` edge connects the parent task to a child future task, just as with regular `async`’s.

- When a future task, $T_F$, terminates, a `join` edge is inserted from the last step in $T_F$ to the step in the ancestor task that follows its Immediately Enclosing Finish (IEF) operation, as with regular `async`’s. In addition, a `join` edge is also inserted from $T_F$’s last step to every step that follows a `get()` operation on the future task. Note that new `get()` operations may be encountered even after $T_F$ has terminated.

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

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

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

2.1.3 Why should future references be effectively final?

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

```java
Listing 7: Example of buggy program using future tasks
```

This program declares two static non-final future reference fields, $f1$ and $f2$, in lines 1 and 2 and initializes them to `null`. The `main()` programs then creates two future tasks, $T1$ and $T2$, in lines 5 and 6 and assigns
them to \( f_1 \) and \( f_2 \) respectively. Task \( T_1 \) uses a “spin loop” in line 10 to wait for \( f_2 \) to become non-null, and task \( T_2 \) does the same in line 15 to wait for \( f_1 \) to become non-null. After exiting the spin loop, each task performs a \texttt{get()} operation on the other thereby attempting to create a \textit{deadlock cycle} in the computation graph. Fortunately, the rule that all variables containing references to future objects should be effectively final can avoid this situation.

### 2.1.4 Future Tasks with a \texttt{void} return type

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

Figure 9 shows Computation Graph \( G_3 \) that cannot be generated using only \texttt{async} and \texttt{finish} constructs, and Listing 8 shows the code that can be used to generate \( G_3 \) using future tasks. This code uses futures with a \texttt{void} return type, and provides a systematic way of converting any CG into a task-parallel program using futures.

### 2.2 Memoization

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

1. Create a data structure that stores the set \{ \((x_1, y_1 = f(x_1)), (x_2, y_2 = f(x_2)), \ldots\) \} for each call \( f(x_i) \) that returns \( y_i \).

2. Look up data structure when processing calls of the form \( f(x') \) when \( x' \) equals one of the \( x_i \) inputs for which \( f(x_i) \) has already been computed.

The memoization pattern lends itself easily to parallelization using futures by modifying the memoized data structure to store \{ \((x_1, y_1 = \text{future}(f(x_1))), (x_2, y_2 = \text{future}(f(x_2))), \ldots\) \}. The lookup operation can then be extended with a \texttt{get()} operation on the future value if a future has already been created for the result of a given input.
```java
static future<int> f1=null;
static future<int> f2=null;

static void main(String [] args) {
    f1 = async<int> { return a1(); }; // Task T1
    f2 = async<int> { return a2(); }; // Task T2
}

int a1() {
    while (f2 == null); // spin loop
    return f2.get(); // T1 waits for T2
}

int a2() {
    while (f1 == null); // spin loop
    return f1.get(); // T2 waits for T1
}
```

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

![Computation Graph G3](image)

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

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

2.3.1 Accumulator Constructs

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

- **new**: When task Ti performs a “ac = new accumulator(op, dataType);” statement, it creates a new accumulator, ac, on which Ti 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 Ti 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 Ti 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 `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, \ldots, ac_n)⟨stmt⟩`”. Note that `finish (ac)` becomes a no-op if `ac` is already associated with an outer `finish` scope.

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

2.3.2 User-defined Reductions

User-defined reductions are also supported in finish accumulators, and its usage consists of these three steps:

1) specify `CUSTOM` and `reducible.class` as the accumulator’s operator and type,
2) define a class that implements the `reducible` interface,
3) pass the implementing class to the accumulator as a type parameter.

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

2.4 Map Reduce

Data structures based on key-value pairs are used by a wide range of data analysis algorithms, including web search and statistical analyses. In Java, these data structures are often implemented as instances of the `Map` interface. An important constraint imposed on sets of key-value pairs is that no key occurs more than once, thereby ensuring that each key can map to at most one value. Thus, a mathematical abstraction of a
void foo() {
  accum<Coord> ac = new accum<Coord>(Operation.CUSTOM,
  reducible.class);
  finish(ac) {
    forasync (point [j] : [1:n]) {
      while(check(j)) {
        ac.put(getCoordinate(j));
      } }
  }
  Coord c = ac.customGet();
  System.out.println("Furthest: " + c.x + ", " + c.y);
}

class Coord implements reducible<Coord> {
  public double x, y;
  public Coord(double x0, double y0) {
    x = x0; y = y0;
  }
  public Coord identity(); {
    return new Coord(0.0, 0.0);
  }
  public void reduce(Coord arg) {
    if (sq(x) + sq(y) < sq(arg.x) + sq(arg.y)) {
      x = arg.x; y = arg.y;
    }
  }
  private double sq(double v) { return v * v; }
}

Figure 12: User-defined reduction example

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

Many data analysis algorithm can be specified as sequences of map and reduce operations on sets of key-value pairs. For a given key-value pair, $(k_i, v_i)$, a map function $f$ generates a sets of output key-value pairs, $f(k_i, v_i) = \{(k_1, v_1), \ldots, (k_m, v_m)\}$. The $k_i$ 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 same 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 10 shows the pseudocode for one possible implementation of map-reduce operations using finish and async primitives. The basic idea is to complete all operations in the map phase before any operation in the reduce phase starts. Alternate implementations are possible that expose more parallelism.

As an example, Listing 10 shows how the WordCount problem can be solved using map and reduce operations on sets of key-value pairs. All map operations in step a) (line 4) can execute in parallel with only local data accesses, making the map step highly amenable to parallelization. Step b) (line 5) can involve a major reshuffle of data as all key-value pairs with the same key are grouped (gathered) together. Finally, step c) (line 6) performs a standard reduction algorithm for all values with the same key.
### Listing 9: Pseudocode for one possible implementation of map-reduce operations using finish and async primitives

```plaintext
finish { // map phase
    for each (ki, vi) pair in input set S
        async compute f(ki, vi) and append output to f(S); // map operation
}
finish { // reduce phase
    for each key k' in intermediate set f(S)
        async { // reduce operation
            temp = identity;
            for each value v'' such that (k', v'') is in f(S) {
                temp = g(temp, v'');
            }
            append (k', temp) to output set, g(f(S));
        }
}
```

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

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

2.5.1 What are Data Races?

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

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

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

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

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

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

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

1. Immutability property: there cannot be a data race on shared immutable data. Recall that shared data in a parallel Habanero-Java program consists of static fields, instance fields, and array elements. An immutable location, \( L_i \), is one that is only written during initialization, and can only be read after
initialization. In this case, there cannot be a data race on $L_i$ because there will only be one step that writes to $L_i$ in CG, and all steps that read from $L$ must follow the write. This property applies by definition to static and non-static final fields. It also applies to instances of any immutable class e.g., java.lang.String.

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

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

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

2.5.2 Avoiding Data Races

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

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

Consider the example program in Listing 12. The parent task initializes s1 to the string, "XYZ" in line 2, creates a child task in line 3, and prints out s1 in line 4. Even though the child task invokes the toLowerCase() method on s1 in line 3, there is no data race between line 3 and the parent task’s print statement in line 4 because toLowerCase() returns a new copy of the string with the lower-case conversion instead of attempting to update the original version.
2. **Single-task ownership tip:** If an object or array needs to be written multiple times after initialization, then try and restrict its ownership to a single task. This will entail making copies when sharing the object or array with other tasks. As in the Immutability tip, it depends on the algorithm whether the copying overhead can be acceptable or prohibitive.

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

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

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

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

### 2.6 Functional and Structural Determinism

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

The presence of data races often leads to functional and/or structural nondeterminism because a parallel program with data races may exhibit different behaviors for the same input, depending on the relative scheduling and timing of memory accesses involved in a data race. In general, the absence of data races...
is not sufficient to guarantee determinism. However, the parallel constructs introduced in this module ("Module 1: Determinism") were carefully selected to ensure the following Determinism Property:

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

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

The determinism property is a powerful semantic guarantee since the constructs introduced in Module 1 span a wide range of parallel programming primitives that include async, finish accumulators, futures, data-driven tasks (async await), forall, barriers, phasers, and phaser accumulators. The notable exceptions are critical sections, isolated statements, and actors, all of which will be covered in Module 2 ("Concurrency").

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

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

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

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

1. Sequential Consistency: The Sequential Consistency (SC) memory model was introduced by Leslie Lamport in 1979 [17] and builds on a simple but strong rule viz., all steps should observe writes to all locations in the same order. Thus, the SC memory model will not permit Task T3 to print “0, 1, 2” and Task T4 to print “0, 2, 1”.

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

Listing 14: Example of a parallel program with data races

```java
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 }
```
async {  // Task T3
    int p.x = p.x;
    System.out.println("First read=", + p.x);
    System.out.println("Second read=", + q.x);
    System.out.println("Third read=", + p.x);
}

Listing 15: Rewrite of Task T3 from Listing[14]

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

2. Location Consistency: The Location Consistency (LC) memory model [9] was introduced to provide an alternate semantics to address the code transformation anomalies that follow from the SC model. The LC rule states that a read of location $L$ in step $S_i$ may receive the value from any Most Recent Write (MRW) of $L$ relative to $S_i$ in the CG. A MRW is a write operation that can potentially execute in parallel with $S_i$, or one that precedes $S_i$ by a chain of dependence edges such that there is no other write of $L$ on that chain. LC is a weaker model than SC because it permits all the outputs that SC does, as well as additional outputs that are not permitted by SC. For the program in Listing[14] the LC model permits Task T3 to print “0, 1, 2” and Task T4 to print “0, 2, 1” in the same execution, and also permits Task T3 to print “0, 1, 0” in a different execution.

3. C++ Memory Model: The proposed memory model for the new C++0x standard [5] makes the following assumption about data races:

“We give no semantics to programs with data races. There are no benign C++ data races.”

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

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

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


