COMP 322: Fundamentals of Parallel Programming
Module 1: Parallelism

January 13, 2016

DRAFT VERSION

Contents

0 Course Organization

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

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

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

1 Task-level Parallelism

1.1 Task Creation and Termination (Async, Finish)

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

Algorithm 1: Sequential ArraySum

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

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

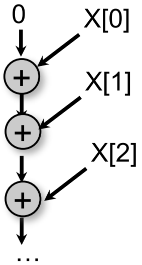
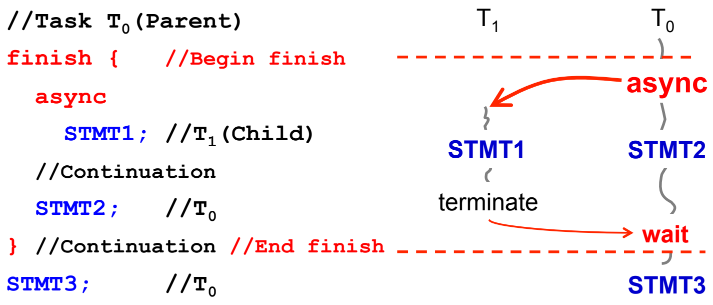


Figure 1: Computation graph for Algorithm 1 (Sequential ArraySum)

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

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

[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.



```
//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

1.1.1 Async notation for Task Creation

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

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

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

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

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

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

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

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

```java
// Example 1: execute iterations of a counted for loop in parallel
// (we will later see forall loops as a shorthand for this common case)
for (int ii = 0; i < A.length; ii++) {
  final int i = ii; // i is a final variable
  async { A[i] = B[i] + C[i]; } // value of i is copied on entry to
}

// Example 2: execute iterations of a while loop in parallel
pp = first;
while ( pp != null ) {
  T p = pp; // p is an effectively final variable
  async { p.x = p.y + p.z; } // value of p is copied on entry to async
  pp = pp.next;
}

// Example 3: Example 2 rewritten as a recursive method
static void process(T p) { // parameter p is an effectively final variable
  if ( p != null ) {
    async { p.x = p.y + p.z; } // value of p is copied on entry to async
    process(p.next);
  }
}

// Example 4: execute method calls in parallel
async { left_s = quickSort(left); }
async { right_s = quickSort(right); }
```

Listing 1: Example usages of async

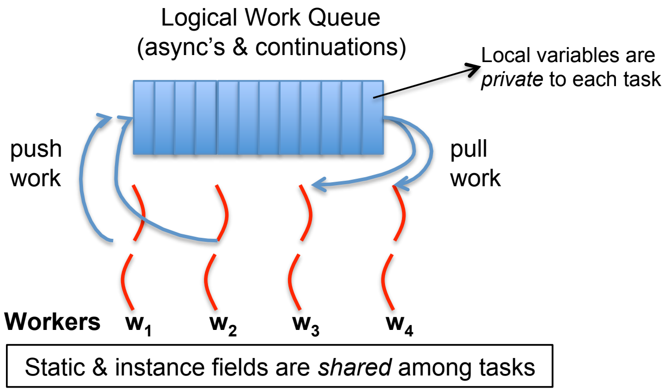

Logical Work Queue
(async's & continuations)
Local variables are *private* to each task
push work
pull work
Workers
w_1
w_2
w_3
w_4
Static & instance fields are *shared* among tasks

Figure 3: Scheduling an parallel program on a fixed number of workers. (Figure adapted from [11].)

```
// Rule 1: a child async may read the value of any outer final local var
final int i1 = 1;
async { ... = i1; /* i1=1 */ }

// Rule 2: a child async may also read any "effectively final" outer local var
int i2 = 2; // i2=2 is  copied on entry into the async like a method param
async { ... = i2; /* i2=2*/}
// i2 cannot be modified again, if it is "effectively final"

// Rule 3: a child async is not permitted to modify an outer local var
int i3;
async { i3 = ...; /* ERROR */}
```

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

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

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

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

1.1.2 Finish notation for Task Termination

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

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

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

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

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

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

1.1.3 Array Sum with two-way parallelism

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

1.2 Computation Graphs

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

```java
// Example 1: Sequential version
for (int i = 0; i < A.length; i++) A[i] = B[i] + C[i];
System.out.println(A[0]);

// Example 1: Incorrect parallel version
for (int ii = 0; i < A.length; ii++) {
  final int i = ii; // i is a final variable
  async { A[i] = B[i] + C[i]; } // value of i is copied on entry to
}
System.out.println(A[0]);

// Example 1: Correct parallel version
finish {
  for (int ii = 0; i < A.length; ii++) {
    final int i = ii; // i is a final variable
    async { A[i] = B[i] + C[i]; } // value of i is copied on entry to
  }
}
System.out.println(A[0]);

// Example 2: Sequential version
p = first;
while ( p != null ) {
  p.x = p.y + p.z; p = p.next;
}
System.out.println(first.x);

// Example 2: Incorrect parallel version
pp = first;
while ( pp != null ) {
  T p = pp; // p is an effectively final variable
  async { p.x = p.y + p.z; } // value of p is copied on entry to async
  pp = pp.next;
}
System.out.println(first.x);

// Example 2: Correct parallel version
pp = first;
finish while ( pp != null ) {
  T p = pp; // p is an effectively final variable
  async { p.x = p.y + p.z; } // value of p is copied on entry to async
  pp = pp.next;
}
System.out.println(first.x);
```

Listing 3: Incorrect and correct parallelization with async and finish

```
finish { // F1
  // Part 1 of Task A0
  async {A1; async A2;}
  finish { // F2
    // Part 2 of Task A0
    async A3;
    async A4;
  }
  // Part 3 of Task A0
}
```

Listing 4: Example usage of async and finish

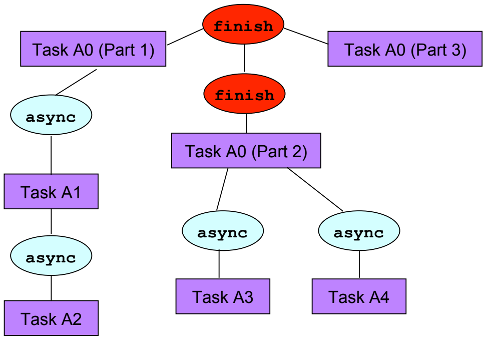


Figure 4: Finish-async nesting structure for code fragment in Listing 4

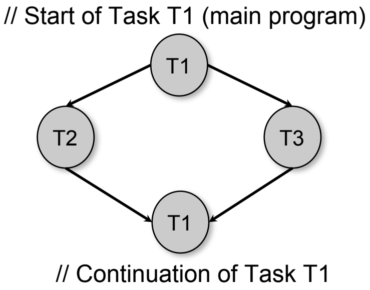

// Start of Task T1 (main program)
T1
T2
T3
T1
// Continuation of Task T1

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)
$sum1 \leftarrow 0$; $sum2 \leftarrow 0$;
// Compute sum1 (lower half) and sum2 (upper half) in parallel.
finish{
    async{
        // Task T2
        for $i \leftarrow 0$ to $X.length/2 - 1$ do
            $sum1 \leftarrow sum1 + X[i]$;
    };
    async{
        // Task T3
        for $i \leftarrow X.length/2$ to $X.length - 1$ do
            $sum2 \leftarrow sum2 + X[i]$;
    };
};
// Task T1 waits for Tasks T2 and T3 to complete
// Continuation of Task T1
$sum \leftarrow sum1 + sum2$;
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 [11]:
    1. *Continue* edges that capture sequencing of steps within a task — all steps within the same task are connected by a chain of *continue* edges.
    2. *Spawn* edges that connect parent tasks to child `async` tasks. When an `async` is created, a *spawn* edge is inserted between the step that ends with the `async` in the parent task and the step that starts the `async` body in the new child task.
    3. *Join* edges that connect descendant tasks to their Immediately Enclosing Finish (IEF) operations. When an `async` terminates, a *join* edge is inserted from the last step in the `async` to the step in the ancestor task that follows the IEF operation.

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

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

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

1.3 Ideal Parallelism

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

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

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

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

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

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

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

1.3.1 Abstract Performance Metrics

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

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

```
// 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; }
}
v23;
```

Listing 5: Example program

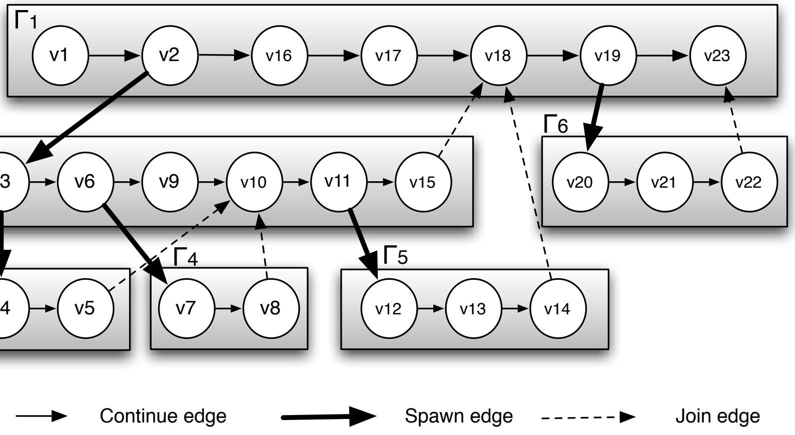


Figure 6: Computation Graph G for example program in Listing 5

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

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

1.4 Multiprocessor Scheduling

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

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

- $t_1 = \mathit{WORK}(\mathrm{G})$. Any greedy scheduler executing on 1 processor will simply execute all nodes in the CG in some order, thereby ensuring that the 1-processor execution time equals the total work in the CG.
- $t_\infty = \mathit{CPL}(\mathrm{G})$. Any greedy scheduler executing with an unbounded (infinite) number of processors must complete its execution with time $= \mathit{CPL}(\mathrm{G})$, since all nodes can be scheduled as early as possible.
- $t_P \leq t_1/P + t_\infty = \mathit{WORK}(G)/P + \mathit{CPL}(G)$. 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.

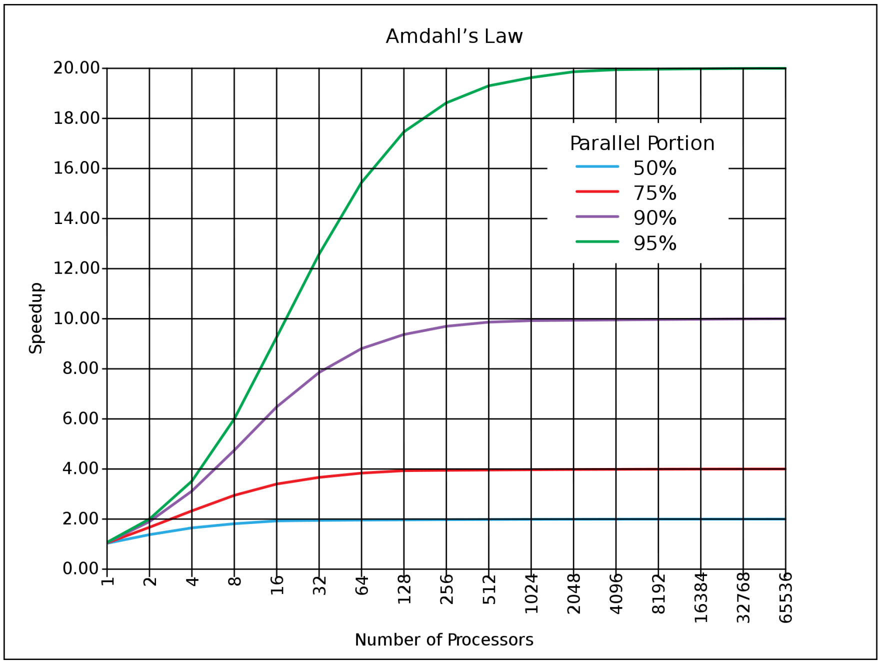


Figure 7: Illustration of Amdahl's Law (source: http://en.wikipedia.org/wiki/Amdahl's_law)

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

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

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

1.5 Parallel Speedup and Amdahl's Law

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

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

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

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

Amdahl's Law reminds us to watch out for sequential bottlenecks both when designing parallel algorithms and when implementing programs on real machines. While it may paint a bleak picture of the utility of adding more processors to a parallel computing, it has also been observed that increasing the data size for a parallel program can reduce the sequential portion [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*.

References

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

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

[3] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. Comput.*, 38:1526–1538, November 1989. ISSN 0018-9340. URL `http://dx.doi.org/10.1109/12.42122`.

[4] R. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46 (5):720–748, 1999. ISSN 0004-5411. URL `http://doi.acm.org/10.1145/324133.324234`.

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

[6] Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Christoph von Praun, Vijay Saraswat, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pages 519–538, NY, USA, 2005. ISBN 1-59593-031-0. URL `http://doi.acm.org/10.1145/1094811.1094852`.

[7] ROBERT H. DENNARD, FRITZ H. GAENSSLEN, HWA-NIEN YU, V. LEO 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] Edsger W. Dijkstra. The structure of the "the"-multiprogramming system. In *Proceedings of the first ACM symposium on Operating System Principles*, SOSP '67, pages 10.1–10.6, New York, NY, USA, 1967. ACM. URL `http://doi.acm.org/10.1145/800001.811672`.

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

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

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

[12] Rajiv Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. In *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, ASPLOS-III, pages 54–63, New York, NY, USA, 1989. ACM. ISBN 0-89791-300-0. URL `http://doi.acm.org/10.1145/70082.68187`.

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

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

[15] C. A. R. Hoare. Algorithm 63: partition. *Commun. ACM*, 4:321–, July 1961. ISSN 0001-0782. URL `http://doi.acm.org/10.1145/366622.366642`.

[16] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4:321–, July 1961. ISSN 0001-0782. URL `http://doi.acm.org/10.1145/366622.366644`.

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

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

[19] Alex Miller. Set your java 7 phasers to stun, 2008. URL `http://tech.puredanger.com/2008/07/08/java7-phasers/`.

[20] Gordon E. Moore. Readings in computer architecture. chapter Cramming more components onto integrated circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000. ISBN 1-55860-539-8. URL `http://portal.acm.org/citation.cfm?id=333067.333074`.

[21] Cherri M. Pancake. Knowing When to Parallelize: Rules-of-Thumb based on User Experiences. URL `http://web.engr.oregonstate.edu/~pancake/presentations/sdsc.pdf`.

[22] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 277–288, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-158-3. doi: http://doi.acm.org/10.1145/1375527.1375568.

[23] Sagnak Tasirlar. Scheduling macro-dataflow programs on task-parallel runtime systems. M.S. Thesis, Department of Computer Science, Rice University, May 2011 (expected).

[24] John von Neumann. First draft of a report on the edvac. *IEEE Ann. Hist. Comput.*, 15:27–75, October 1993. ISSN 1058-6180. URL `http://dx.doi.org/10.1109/85.238389`.

[25] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004. ISBN 0131405632.