# Summary: Task Level Parallelism

07 August 2020 13:27

## Task Creation & Termination

the ***async*** notation for task creation: "*async ⟨stmt1⟩*", causes the parent task (*i.e.*, the task execu
statement) to create a new child task to execute the body of the *async, ⟨stmt1⟩, asynchronously*
parallel) with the remainder of the parent task.
the ***finish*** notation for task termination: "*finish ⟨stmt2⟩*" causes the parent task to execute *⟨stmt*
*⟨stmt2⟩* and all *async* tasks created within *⟨stmt2⟩* have completed. Async and finish constructs r
nested.

```
finish {
  async Sum1; // asynchronously compute sum of the left half of the array
  async  Sum2;      // compute sum of the right half of the array in parallel with S1
}
Sum3 = Sum1 + Sum2; // combine the two partial sums after both Sum1 and Sum2 have finished
```

## Tasks in Java's Fork / Join Framework

A task can be specified in the **compute()** method of a user-defined class that extends the standa
class in the FJ framework. In our Array Sum example, we created class **SumNumber** with fields **A**
and **HI** for the subrange for which the sum is to be computed, and **SUM** for the result for that su
of this user-defined class (e.g., **L** in the lecture), we learned that the method call, **L.fork()**, create
executes L's compute() method. This implements the functionality of the async construct that w
call to **L.join()** then waits until the computation created by **L.fork()** has completed. Note that **join**
primitive than finish because join() waits for a specific task, whereas finish implicitly waits for all
scope. To implement the finish construct using join() operations, you have to be sure to call join
in the finish scope.

A sketch of the Java code for the **SumNumber** class is as follows:

```
class SumNumber extends RecursiveAction {
  int[] A; // input array
```

uting the async
(*i.e.*, before, after, or in

*2)*, and then wait until
may be arbitrarily

d

rd **RecursiveAction**
**A** for the input array, **LO**
ubrange. For an instance
s a new task that
e learned earlier. The
**n()** is a lower-level
tasks created in its
() on every task created

```
  int LO, HI; // subrange
  int SUM; // return value
  @Override
  protected void compute() {
    SUM = 0;
    for (int i = LO; i <= HI; i++)
         SUM += A[i];
  } // compute()
}
```

FJ tasks are executed in a ForkJoinPool, which is a pool of Java threads. This pool supports the in combines both the fork and join operations by executing a set of tasks in parallel, and waiting fo example, ForkJoinTask.invokeAll(left,right) implicitly performs fork() operations on left and right operations on both objects.

# Computation Graphs, Work, Span

Computation Graphs (CGs) models the execution of a parallel program as a partially ordered set consists of:

- A set of vertices or nodes, in which each node represents a step consisting of an arbitrary computation.
- A set of directed edges that represent ordering constraints among steps.

For fork–join programs, it is useful to partition the edges into three cases:

1. Continue edges that capture sequencing of steps within a task.
2. Fork edges that connect a fork operation to the first step of child tasks.
3. Join edges that connect the last step of a task to all join operations on that task.

CGs can be used to define data races, an important class of bugs in parallel programs. We say th location L in a computation graph, G, if there exist steps S1 and S2 in G such that there is no path S1 to S2 or from S2 to S1 in G, and both S1 and S2 read or write L (with at least one of the access two parallel reads do not pose a problem).

CGs can also be used to reason about the ideal parallelism of a parallel program as follows:

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

Define **SPAN(G)** to be the length of a longest path in G, when adding up the execution times of a The longest paths are known as critical paths, so SPAN also represents the critical path length (C

Given the above definitions of WORK and SPAN, we define the ideal parallelism of Computation

vokeAll() method that
r their completion. For
, followed by join()

. Specifically, a CG

sequential

at a data race occurs on
n of directed edges from
ses being a write, since

ll nodes in the path.
PL) of G.

Graph G as the ratio,

WORK(G)/SPAN(G). The ideal parallelism is an upper limit on the speedup factor that can be obt
execution of nodes in computation graph G. Note that ideal parallelism is only a function of the
does not depend on the actual parallelism available in a physical computer.

## Multiprocessor Scheduling, Parallel Speedup

we will study the possible executions of a Computation Graph (CG) on an idealized parallel mach
It is idealized because all processors are assumed to be identical, and the execution time of a no
fixed, regardless of which processor it executes on. A *legal schedule* is one that obeys the depen
CG, such that for every directed edge (*A, B*), the schedule guarantees that step *B* is only schedul
step *A* completes. Unless otherwise specified, we will restrict our attention in this course to sche
no *unforced idleness,* i.e., schedules in which a processor is not permitted to be idle if a CG node
scheduled on it. Such schedules are also referred to as "greedy" schedules.

We defined *T(*P) as the execution time of a CG on *P* processors, and observed that

$T(\infty) \leq T(P) \leq T(1)$

We also saw examples for which there could be different values of $T\_PP$ for different schedules
on *P* processors.

We then defined the parallel speedup for a given schedule of a CG on *P* processors as *Speedup(P*
observed that *Speedup(P)* must be ≤ the number of processors *P* , and also ≤ the ideal parallelism

## Amdahl's Law

If $q \leq 1$ is the fraction of *WORK* in a parallel program that must be executed *sequentially*, then th
can be obtained for that program for any number of processors, *P* , is $Speedup(P) \leq 1/q$.

This observation follows directly from a lower bound on parallel execution time that you are fam
namely $T\_PP \geq SPAN(G)$. If fraction *q* of *WORK(G)* is sequential, it must be the case that $SPAN(G)$
Therefore, $Speedup(P) = T(1)/T(P)$ must be $\leq WORK(G)/(q \times WORK(G)) = 1/q$ since $T(1) = WORK(G$

Amdahl's Law reminds us to watch out for sequential bottlenecks both when designing parallel a
implementing programs on real machines. As an example, if $q = 10\%$, then Amdahl's Law remind
possible speedup must be ≤ 10 (which equals $1/q$ ), regardless of the number of processors avail

ained from parallel
parallel program, and

hine with *P* processors.
de is assumed to be
dence constraints in the
ed after
edules that have
e is available to be

of the same CG

*P) = T*(1)/*T*(*P)*, and
m, WORK/SPAN.

he best speedup that

miliar with,
*≥ q × WORK(G)*.
*G)* for greedy schedulers.

algorithms and when
ls us that the best
lable.