Lecture 1: Task Level Parallelism

07 August 2020 12:59

Task Creation & Termination

Sequential Algorithm as a sequence of steps: S1, S2, S3 … Sn

The whole idea behind parallel programming with multiple processors is to determine which of these steps can run in parallel with each other and how there parallism is coordinated.

Example: An array of integers having two halves. Calculate sum of all numbers in this array.

Sum (total) = **Async** [Sum (left sub array) + Sum (right sub array)]

Async means that the computation can happen before what follows, after what follow or more interestingly in parallel to what follows.

To compute Sum(Total): we need to result of sum(left) and result of sum(right). We can ensure that by wrapping these inside a notation **Finish**

Finish: we can have multiple tasks running with this block and at the end of the finish scope we're guaranteed that all specified tasks would have completed before we can proceed.

Run 1 component on core 1, other on core 2.

S1,
S2, Async
S3, Async
S4, Async
finish
S5

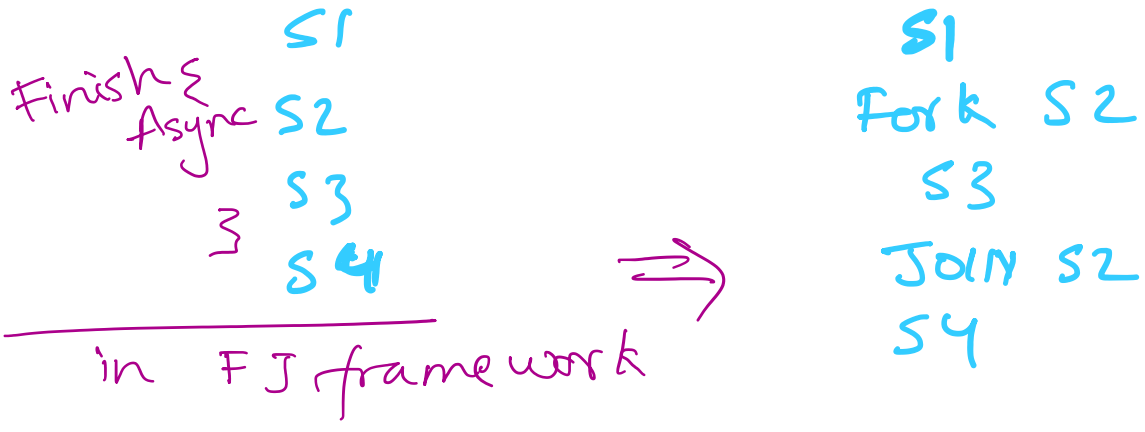
Tasks in Java's Fork / Join Framework

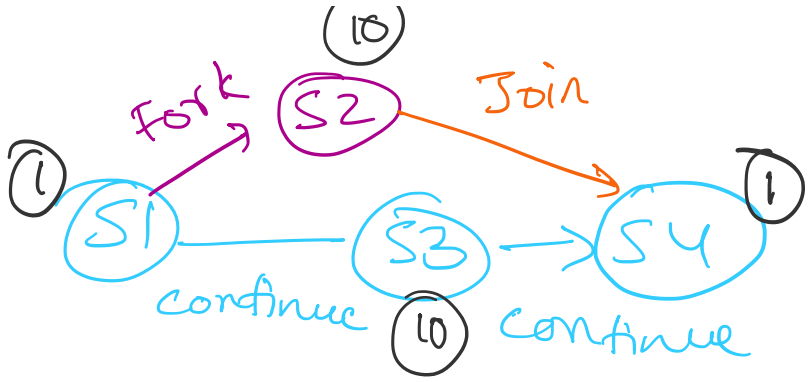
most popular ways of exploiting multi-core parallelism in Java today.

Lets take the same Sum Array example and see if we can extend it to divide and conquer algorithm. Lets apply the OOPs concepts we learned in earlier classes

```
Class SumNumber extends RecursiveAction{
     Int[] A,
     Int low, high, sum
     Compute() {
          If(low == high)
               Sum = A[low];
          Else if( low > high)
               Sum = 0;
          Else {
               Mid = ( low + high ) / 2;
               L= new SumNumber(A, low, mid)
               R = new SumNumber(A, mid+1, high)
               L.compute(); // async -> L.fork()
               R.compute(); // async
               Sum = L.sum + R.sum; // L.join();
// invokeAll(L, R) : takes care of forking
whats passed inside and then joining them once
the execution is finished.
          }
     }
}
```

Computation Graphs, Work, Span



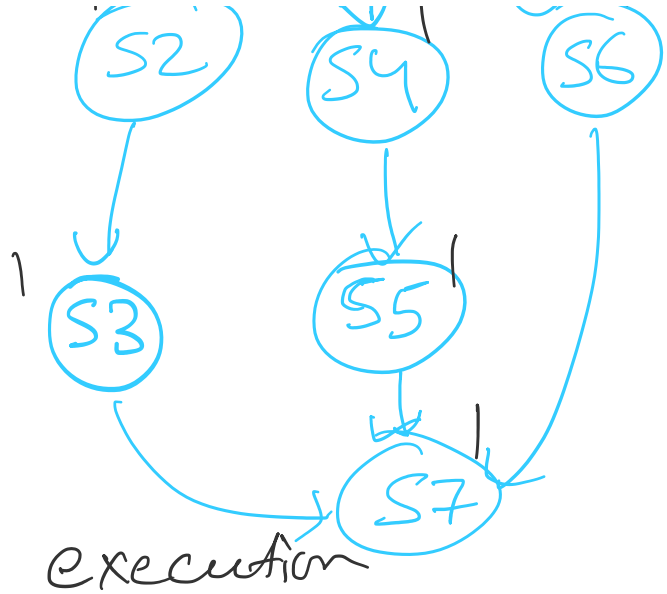


So we have 3 kinds of edge

Each node of this directed graph represents a sequential execution (step). Each edge represents an ordering constraint. If you want to reason about which statements can execute in parallel,
we ask "Is there a path of directed edges from one statement to another?" So for example, there's a path from S2 and S4. So that tells us that S2 and S4 cannot run in parallel with each other. But between S2 and S3, we can see there's a parallel execution that's possible, because there's no path of directed edges between S2 and S3. So that gives the basic property about what can run in parallel. It's a very handy abstraction to think in terms of debugging.
For example, if by mistake in S3, we were starting to try to read some sum computed by S2, and in S2, we were starting to write to that field sum, then we would have an error, because the read and write could go in parallel with S2 and S3 not being connected with each other in the computation graph. There's no path of edges between them. That's a very pernicious kind of bug in parallel programming. That's called a data race.

So let's just say in abstract units, S1 took one unit of time. S3 takes 10, S2 takes 10, and S4 takes one. There are two important metrics that we will work with to reason about the performance. The first is called the **"WORK"**. And that's simply the sum of the execution times of all the nodes. So in this case, it would be 1 plus 10 plus 10 plus 1. That's 22. Now, another metric that's really important is called the "SPAN".
And that's the length of the longest path.

Multiprocessor Scheduling, Parallel Speedup



T_p : execution time on p processes.

P_0	P_1
S1	IDLE
S2	S4
S3	S5
S6	IDLE
S7	IDLE

14

P_0	P_1
S1	IDLE
S6	S2
	S3
	S4
	S5
S7	

12

if $P = 1$ = work = T_1

$= 16$

P_∞ = SPAN = T_∞

$= 12$

$$T_\infty \leq T_p \leq T_1$$

$$\text{speed up} = \frac{T_1}{T_p} \leq \frac{\text{work}}{\text{span}}$$

$\therefore$ ideal parallism

AMDAHL's Law

if we don't have a CG

q = SEQUENTIAL FRACTION

speed up $\leq 1/q$

$q = 0.5 \Rightarrow$ speed up ≤ 2

$= 0.1 \Rightarrow$ speed up ≤ 10

span $\geq$ work $*$ q

speedup $\leq \frac{work}{span}$

$\leq \frac{work}{q * work}$ $\leq$

1

$\frac{1}{9}$