
Java™
ORACLE

Towards a better Parallelism

Vaibhav Choudhary (@vaibhav_c)
Java Platforms Team
Principal Member of Technical Staff
Bangalore JUG, Leader
https://blogs.oracle.com/vaibhav

Java Your Next (Cloud)

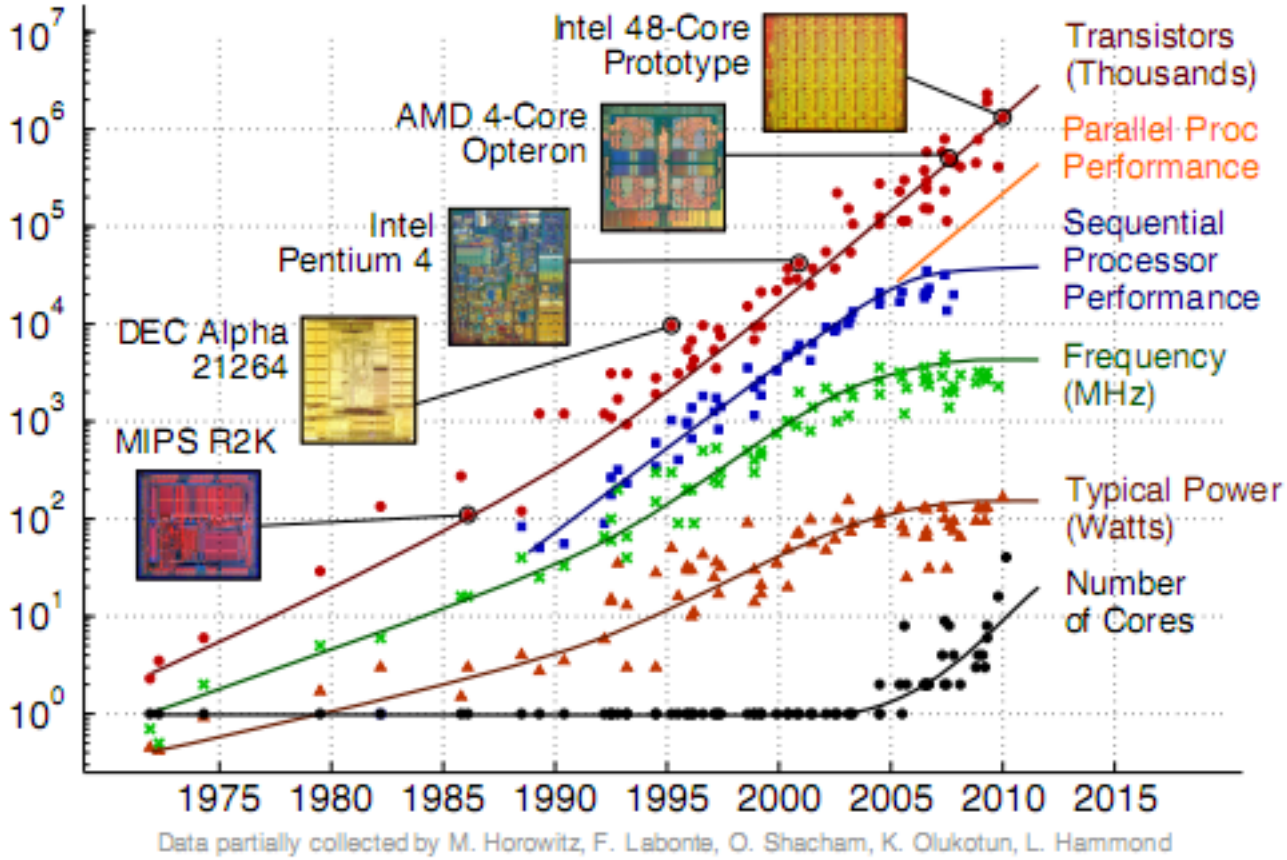
Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Agenda of the day …

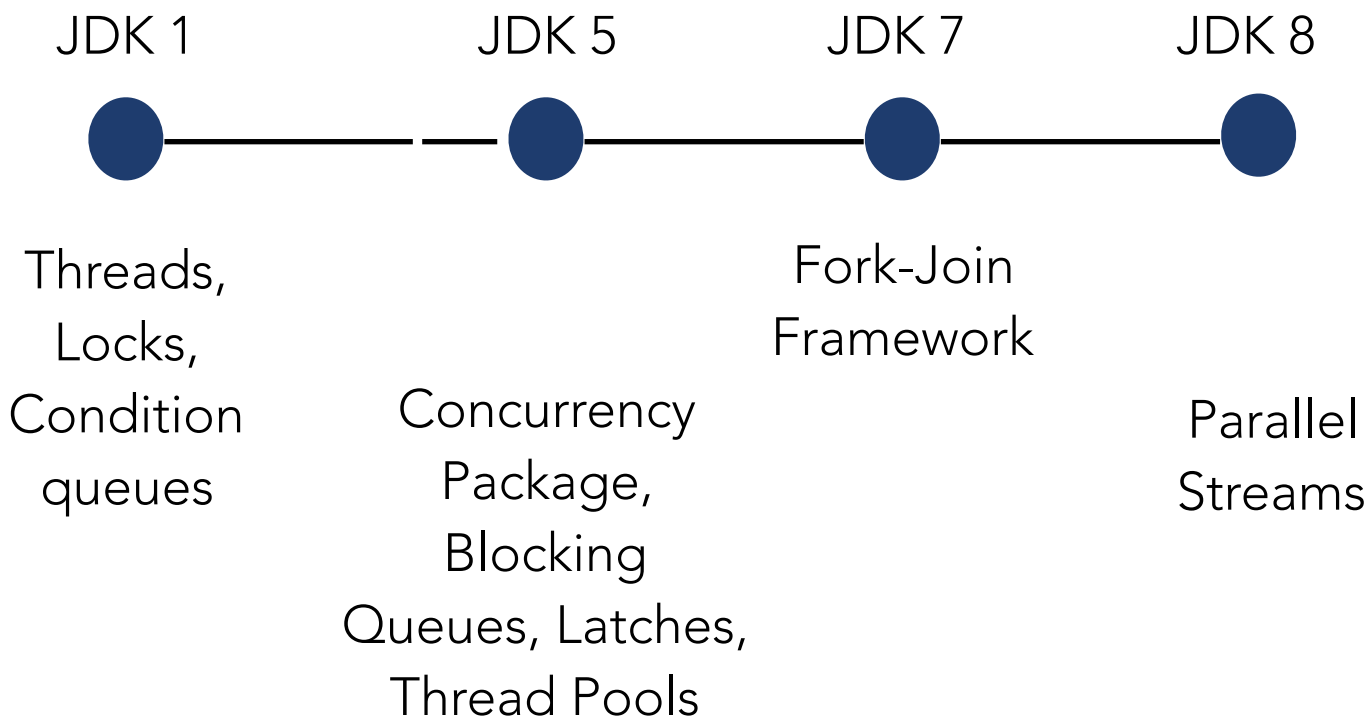
1. Future of Hardware
2. Software followed Hardware
3. Productive Parallelism
4. Performance Consideration
5. A glimpse of Parallel Streams
6. Where we are heading

Where future is trending …



Prepared by C. Batten - School of Electrical and Computer Engineering - Cornell University - 2005 - retrieved Dec 12 2012 - http://www.csl.cornell.edu/courses/ece5950/handouts/ece5950-overview.pdf

Software followed Hardware



What is Parallelism …

- Breaking the tasks into sub-tasks and get results faster.
- Faster and only faster (with more resources)
- Golden rule
  - Analyze -> Implement -> Test -> Repeat
- If results are not better (?), drop the idea.

Reference: Internet Movie Firearms Database (http://www.imfdb.org/wiki/Matrix,_The)

Any overhead ?

- Parallel computation always involves “more” task than its alternative sequential computation.
- Always a slow startup.

DECOMPOSE THE PROBLEM INTO TASK

LAUNCH TASK, MANAGE TASK, WAIT FOR COMPLETION

COMBINE RESULTS

+

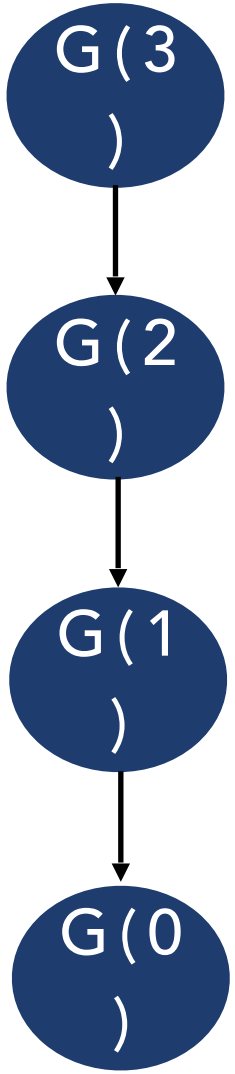
SOLVE THE PROBLEM

Data Flow Dependency

- Consider a problem and look at the data flow dependency.
- If it's a real iterative problem - Bad idea to think of parallelism.

$$G(N) = F(G(N-1)), \text{ IF } N > 0$$

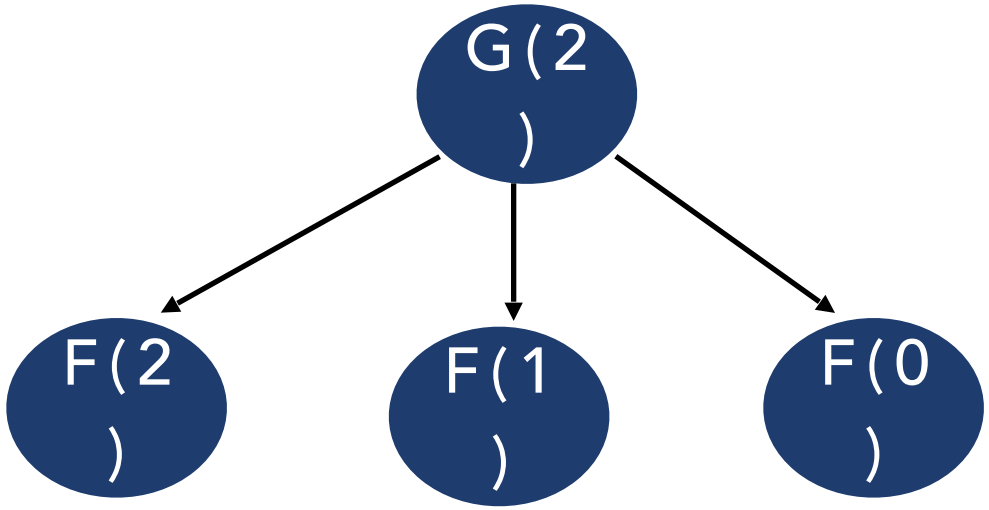
$$G(0) = F(0)$$

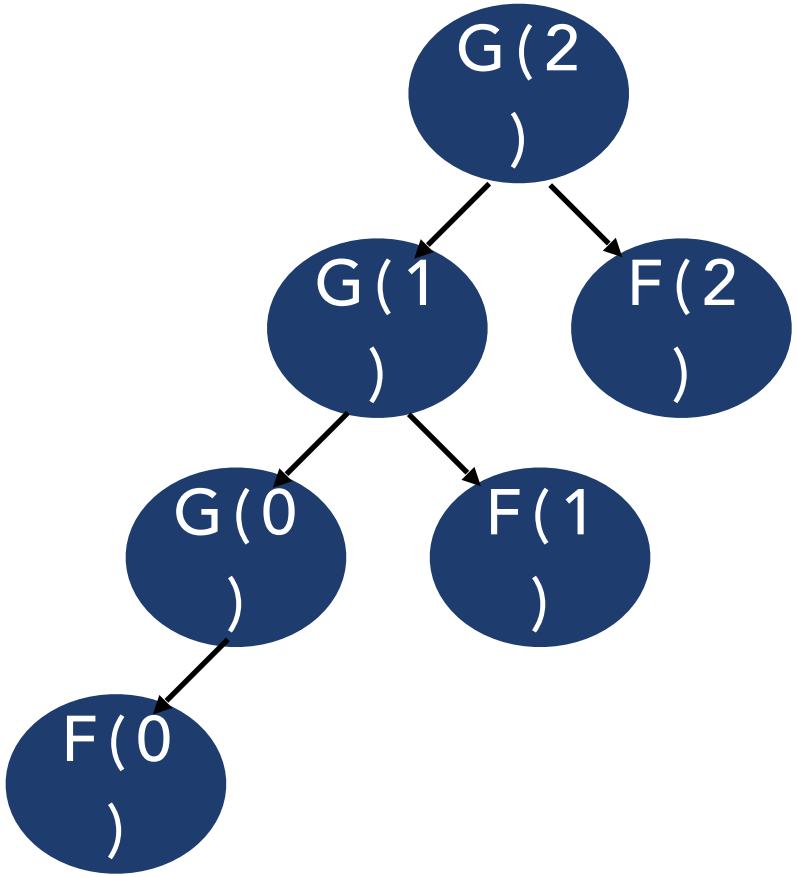


Parallelism can be exploited

$$G(N) = F(N) + G(N-1), \text{ IF } N>0$$

$$G(0) = F(0)$$





Let’s reach out to the parallel computation

```
int sum (int[] arr) {
    int sum = 0;
    for (int i : arr)
        sum = sum + i;
    return sum;
}
```

Sequential approach. Accumulator Pattern.

```
int sum (int[] arr) {
    int sum = 0;
    INT MID = arr.length/2;
    CONCURRENT {
        { for (int i=0; i< MID; i++)
            sum = sum + arr[i];  }
        { for (int i=MID;i<arr.length; i++)
            sum = sum + arr[i]; }
    }
    return sum;
}
```

//need to atomic

//need to atomic

Cont…

- The last solution is bad. Because we are spending time to safe-guard the data.
- Right approach (in the order) :-
  - Don’t share
  - Don’t mutate
  - Coordinate access

Cont…

```
int sum (int[] arr) {
    int LEFT = 0, RIGHT = 0;
    int MID = arr.length/2;
    CONCURRENT {
        { for (int i=0; i< MID; i++)
            ATOMIC { LEFT = LEFT + arr[i]; } }
        { FOR (int i=MID;i<arr.length; i++)
            ATOMIC { RIGHT = RIGHT + arr[i]; } }
    }
    return LEFT + RIGHT;
}
```

Fairly a good solution

Generic solution is required.

Leverage all the cores, split till sequential gives the better result.

Pseudo-Code for generic solution

```
// PSEUDOCODE
Result solve(Problem problem) {
    if (problem.size < SEQUENTIAL_THRESHOLD)
        return solveSequentially(problem);
    else {
        Result left, right;
        INVOKE-IN-PARALLEL {
            left = solve(extractLeftHalf(problem));
            right = solve(extractRightHalf(problem));
        }
        return combine(left, right);
    }
}
```

Introduction to Fork-Join Framework

- JDK7 introduced Fork And Join Framework.
- Works on Divide and Conquer
- Implements work-stealing algorithm.

```
if (my portion of the work is small enough)
        do the work directly

else
        split my work into two pieces
        invoke the two pieces and wait for the results
```

Introduction of ParallelStreams in Java8

- Build on top of FnJ Framework.
- FnJ was not so easy to use.
- You just have to say “.parallel()”

```
Arrays
    .stream(a)
    .parallel()
    .filter(s -> s > 5000000)
    .sum();
```

Magical, but unfortunately not…

Parallel Stream consideration

Performance Considerations

- Splitting of Source
  - source should be splittable
  - cost of computation for split
  - evenness, predictability of split
- Some source are awesome to split and some are not.
- Arrays (Good), LinkedList (Not good), Tree (so-so)
- Not only splitting, combining result should be considered.

Minimum Data to get benefit (Run Size: 500K)

THREAD/ THRESHOLD	500K	50K	5K	500	50
2 thread	1.0	1.07	1.02	0.82	0.2
4 thread	0.88	3.02	3.2	2.22	0.43
8 thread	1.0	5.29	5.73	4.53	2.03
32 thread	0.98	10.46	17.21	15.34	6.49

The NQ model

- Simple model for parallel performance
  - N: No of elements
  - Q: amount of work per item
  - N X Q > 10,000 -> chance for parallelism
  - Important for say “minimum data”

Locality Issues

- Locality is dangerous.
- CPU should be busy in good work
  - Cache misses are not good.
- Streams.of(int[].sum()) vs Streams.of(Interger[].sum())

Speed Up	N=1K	N=10K	N=1M
int	1x	6.2x	7.9x
Integer	-4.9x	1.5x	3.5x

Order insensitive operations

- Least parallel friendly.
- Always remember the bottle-neck principles.

```
OptionalLong m = Arrays.stream(a).parallel()  // killer
        .filter(s -> s > 5000000)
        .filter(x -> x == 5465759)
        .findFirst();
```

Must visit reference

- Java Concurrency in Practice (Book) - Brian Goetz
- Concurrency to Parallelism
- Introduction to Streams Library
- Java Fork-Join Framework - Doug Lea