

# Profiling General Purpose Systems for Parallelism

06.11.2009

Clemens Hammacher, Kevin Streit, Sebastian Hack, Andreas Zeller

Saarland University

Department of Computer Science

Saarbrücken, Germany

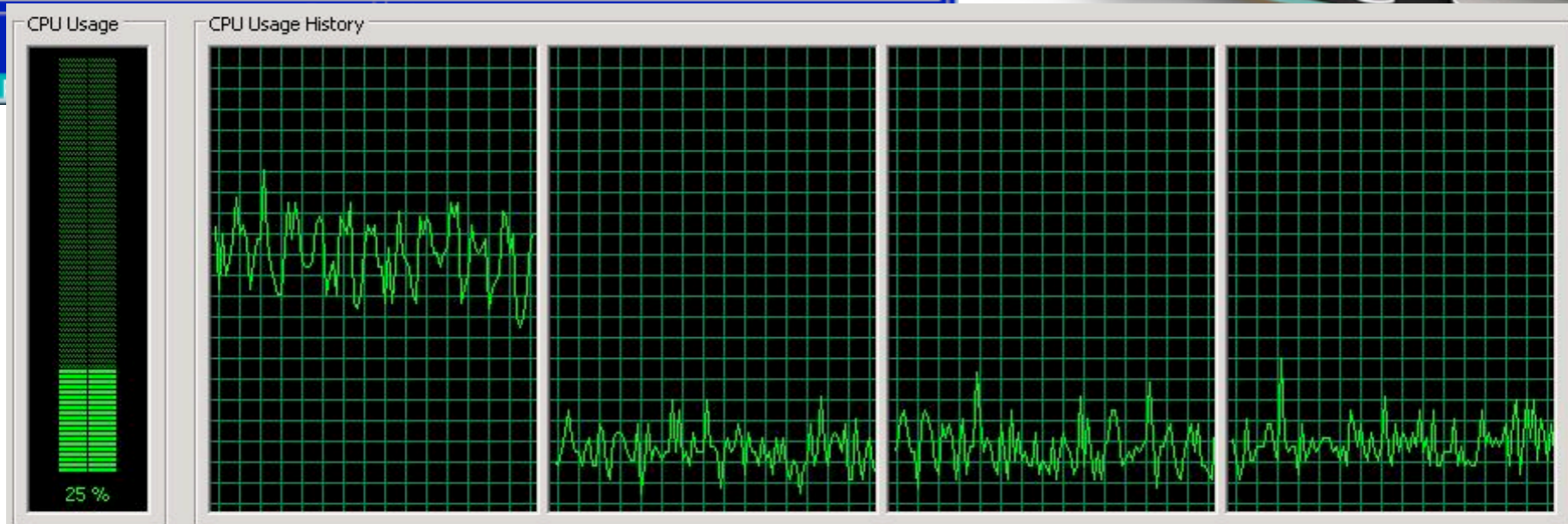
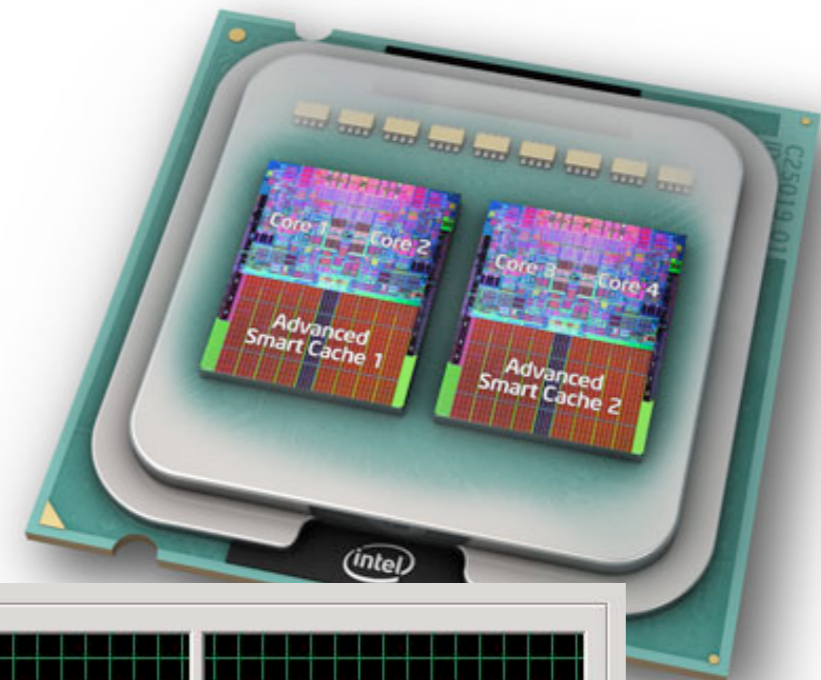
Error Log Progress Console Loop Analysis Result Slicing Criteria

The following are the loops with the most gain.  
If a project is associated with the trace you can double click the line to open the corresponding code.

| class                         | method  | line | gain     | instances |
|-------------------------------|---|------|----------|-----------|
| de.unisb.test.SumUp           | main([Ljava/lang/String;)V                        | 10   | 99.8209% | 1         |
| de.unisb.test.SumUp           | sumTo(I)I   | 21   | 49.9503% | 2500      |
| de.unisb.test.SumUp           | main([Ljava/lang/String;)V                        | 16   | 0.0559%  | 1         |
| java.lang.Integer             | parseInt(Ljava/lang/String;I)I                    | 452  | 0.0006%  | 1         |
| java.util.Hashtable           | get(Ljava/lang/Object;)Ljava/lang/Object;         | 333  | 0.0004%  | 14        |
| java.lang.CharacterDataLatin1 | digit(I)I   | 174  | 0.0003%  | 4         |
| java.lang.Integer             | parseInt(Ljava/lang/String;I)I                    | 414  | 0.0001%  | 1         |
| java.lang.Character           | digit(I)I   | 4532 | 0.0001%  | 4         |
| java.util.Properties          | getProperty(Ljava/lang/String;)Ljava/lang/String; | 932  | 0.0001%  | 13        |

# Motivation

```
Microsoft QuickBASIC
Auto
File Edit View Search Run Debug Options Help
POLSCALO.BAS
REM POLSCALO.BAS creates polar dial hour lines.
REM Mac Oglesby 990709
filename$ = "C:\POLSCALO.txt"
OPEN filename$ FOR OUTPUT AS #1
D2R = 3.141593 / 180
FOR J = 0 TO 67.5 STEP 1.25
H = 76.2
X = H * TAN(J * D2R)
L = 4: M = J * 12
IF M / 45 = INT(M / 45) THEN L = 8
IF M / 90 = INT(M / 90) THEN L = 12
IF M / 180 = INT(M / 180) THEN L = 20
WRITE #1, 1, X, 0
WRITE #1, 2, X, L
NEXT J
WRITE #1, 1, 0, 0
WRITE #1, 2, X, 0
X1 = X + 10 + 76.2
<Shift+F1=Help
```

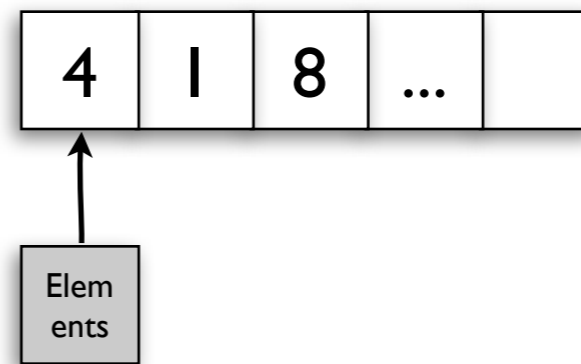
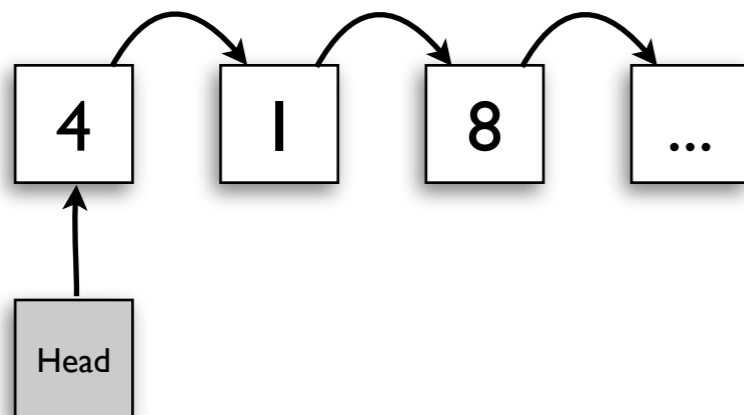
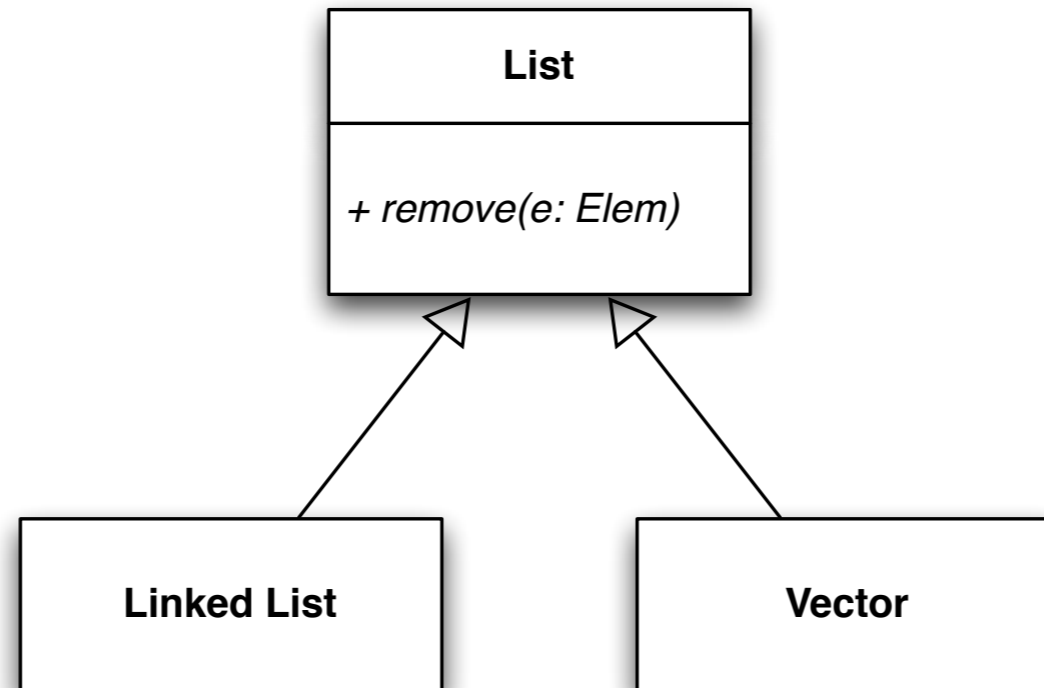


# Unexploited parallelization potential

- Modern OO-Systems difficult to analyze
  - *modularity*: loosely coupled modules
  - runtime reconfiguration
  - *abstraction*: dynamic binding
  - *encapsulation*: data and computation locality

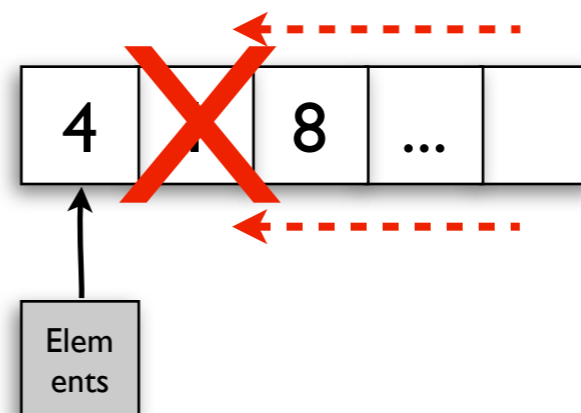
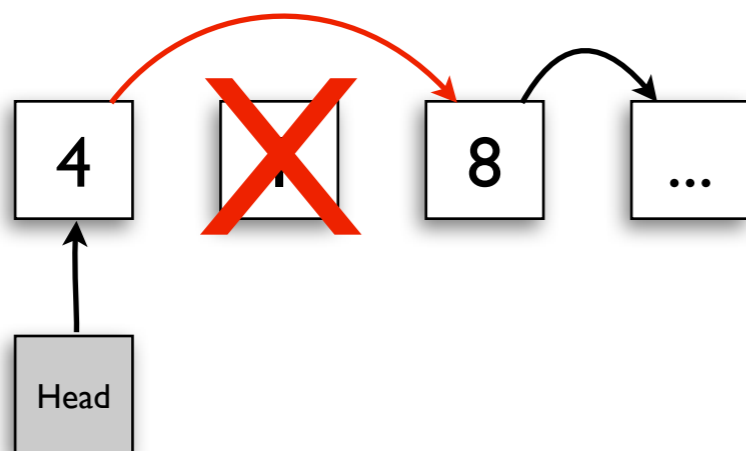
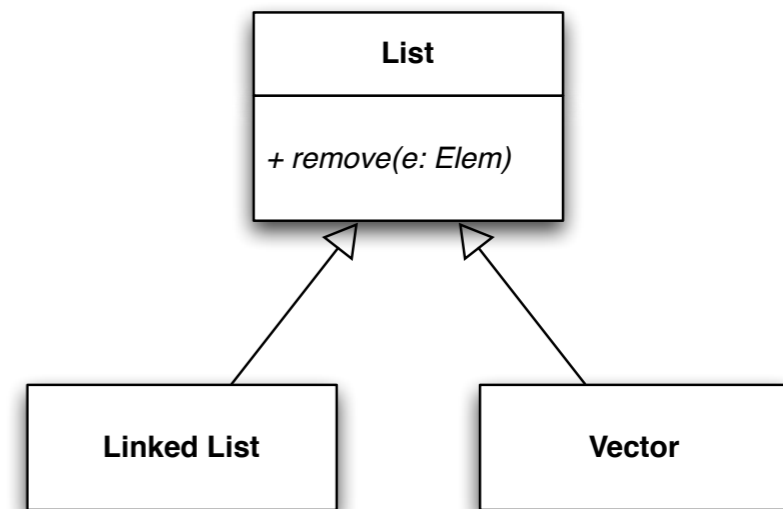
**Can we dynamically  
extend static analyses?**

# Dynamic Binding



# Dynamic Binding

```
public void removeAll(  
    List A, List B)  
{  
    for (Object elem: B) {  
        A.remove(elem);  
    }  
}
```



# First Step: Profiling

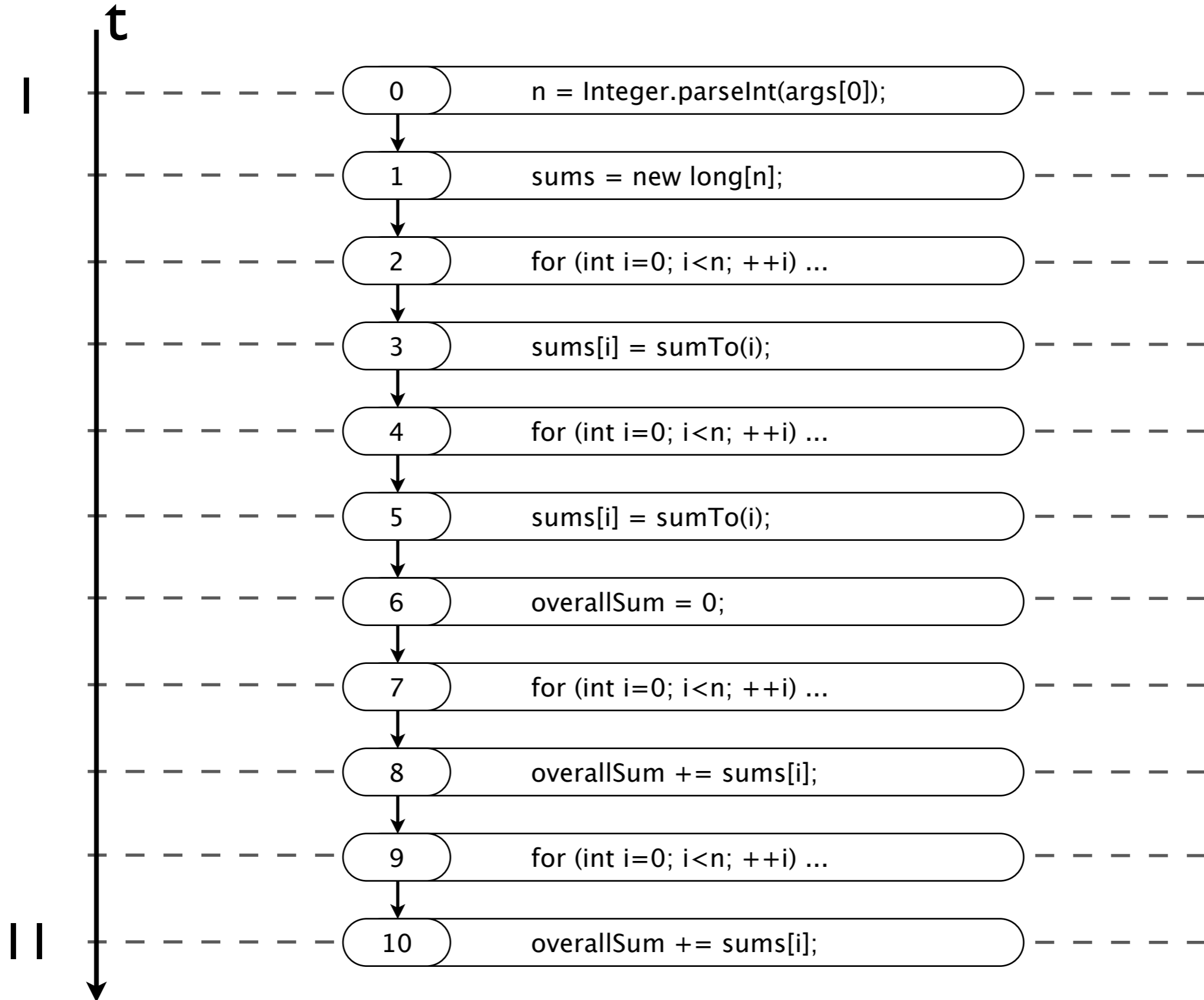
? How much parallelism exists in legacy systems

! Analyse several program runs

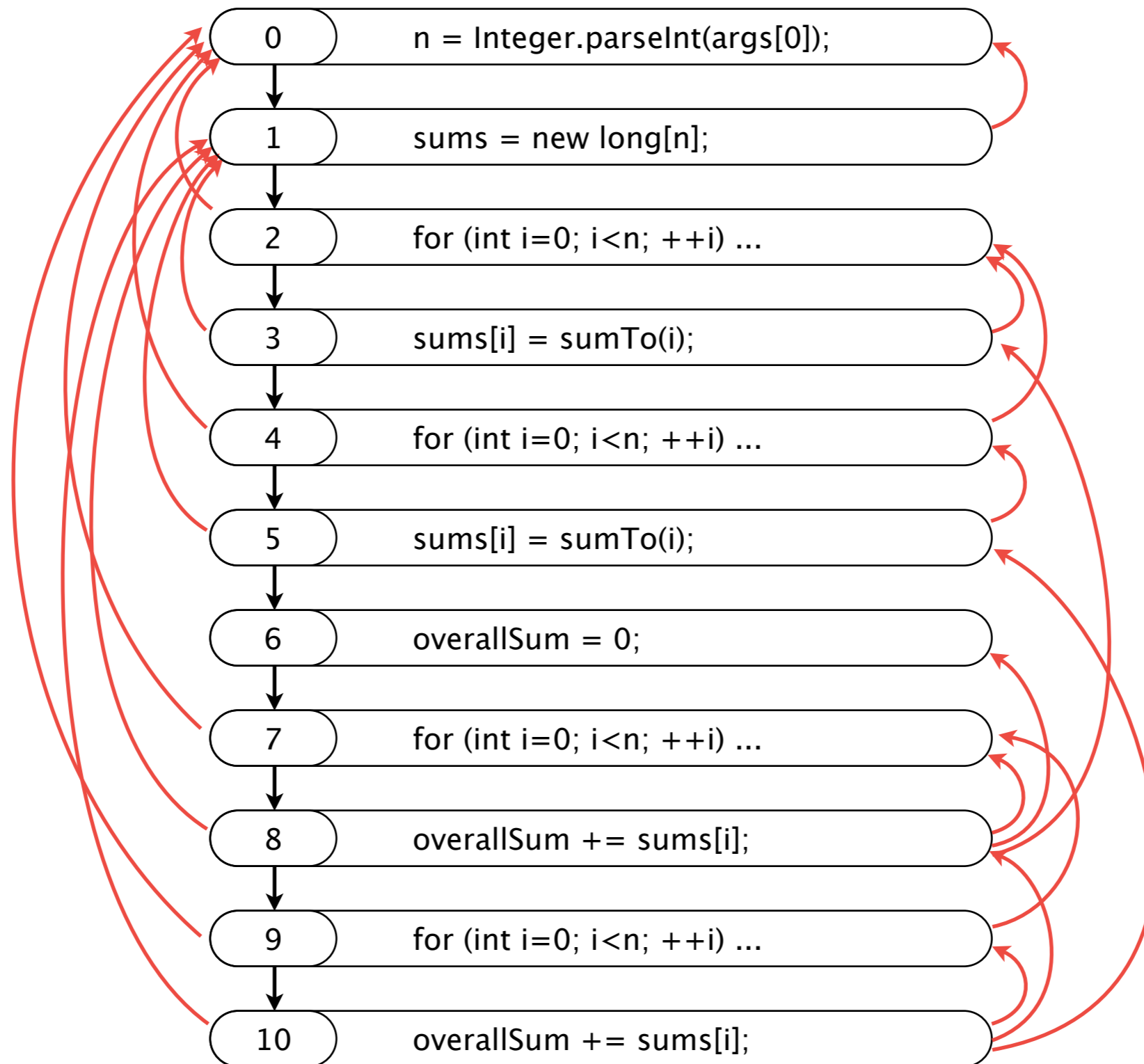
# Example

```
public class SumUp {  
  
    public static void main(String[] args) {  
        int n = Integer.parseInt(args[0]);  
        long[] sums = new long[n];  
  
        for (int i=0; i<n; ++i) {  
            sums[i] = sumTo(i);  
        }  
  
        long overallSum = 0;  
  
        for (int i=0; i<n; ++i) {  
            overallSum += sums[i];  
        }  
    }  
  
    private static long sumTo(int n) {  
        return n == 0 ? 0 : n + sumTo(n-1);  
    }  
}
```

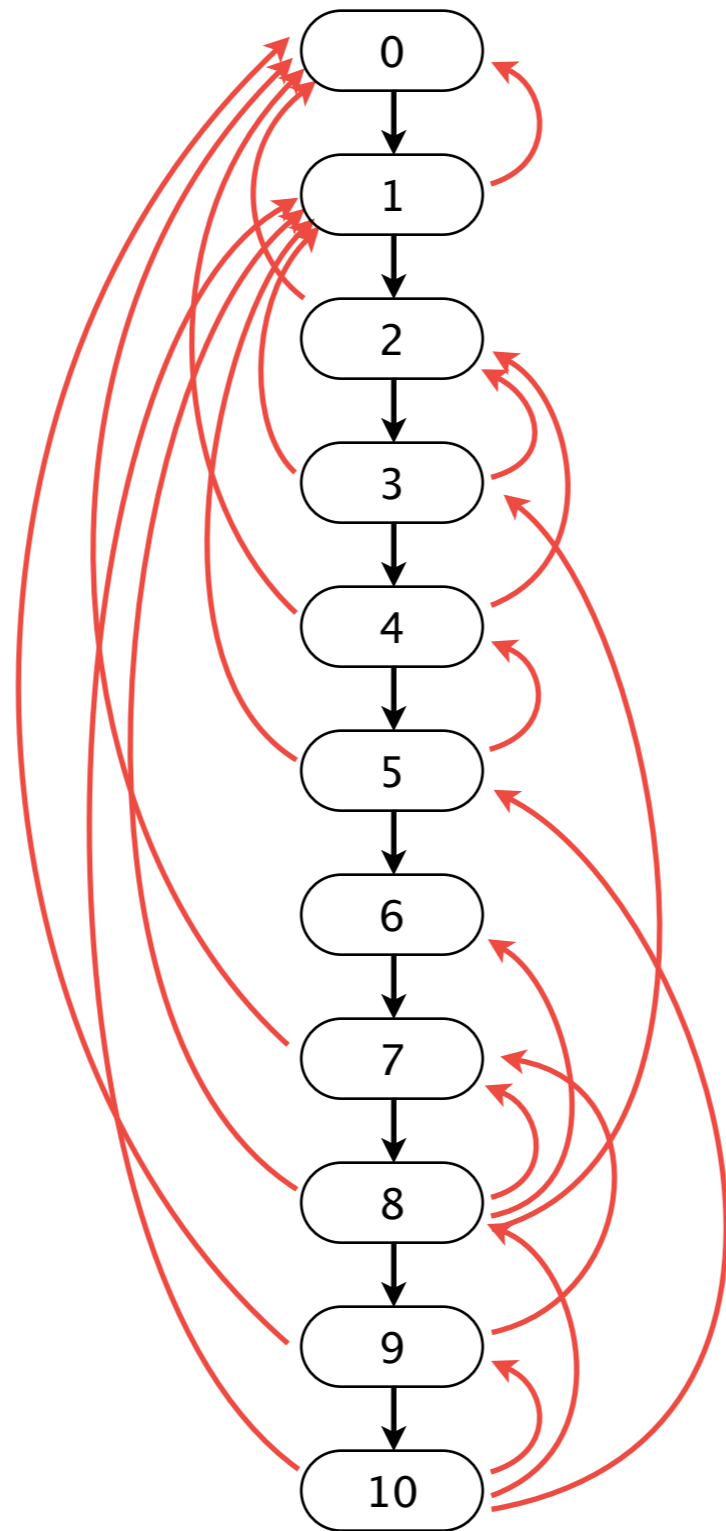
# Execution Trace



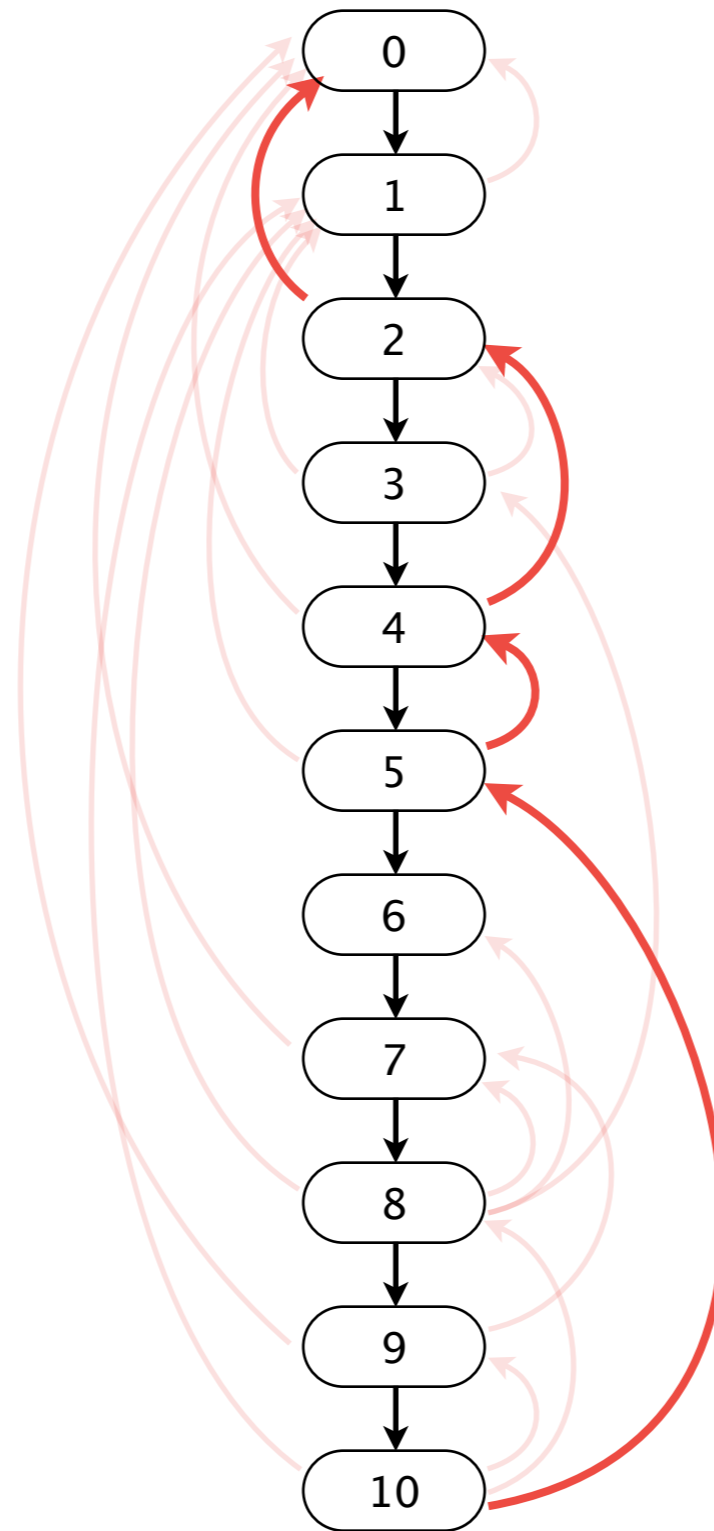
# Data Dependencies



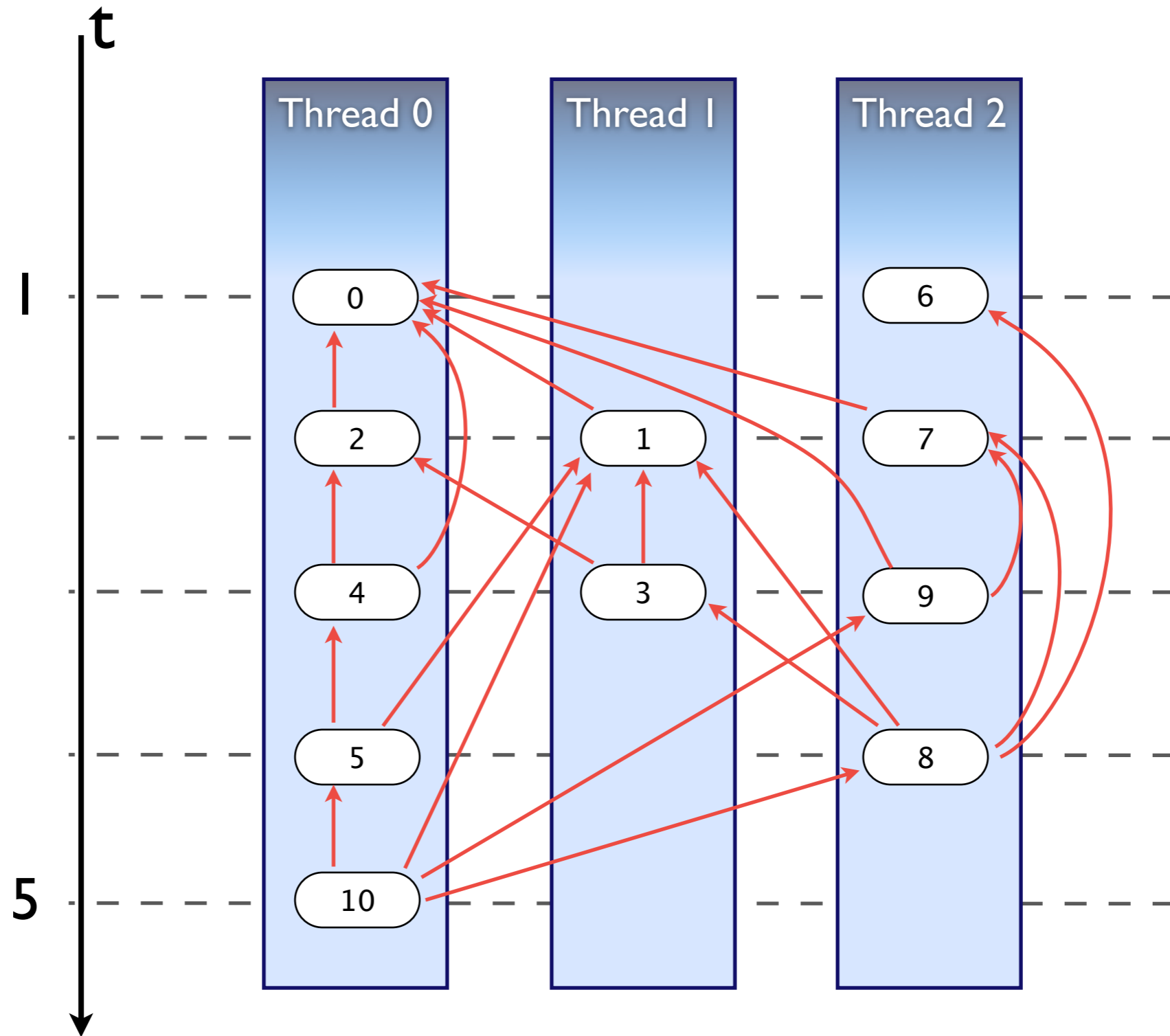
# Data Dependencies



# Critical Path



# Parallel Execution



# Parallelization Potential

$$\text{potential} = \frac{\text{\#nodes}}{\text{\#nodes on CP}}$$

Example:  $\frac{\text{\#nodes}}{\text{\#nodes on CP}} = \frac{11}{5} = 2.2$

# Basics

Dynamic Trace



Data Dependencies



Critical Path



Parallelization Potential

# Applications

- Overall program analysis
- Suggesting parallelization candidates
- Automatic parallelization
  - adaptive
  - “just in time”
- Simulating parallelization strategies on a trace

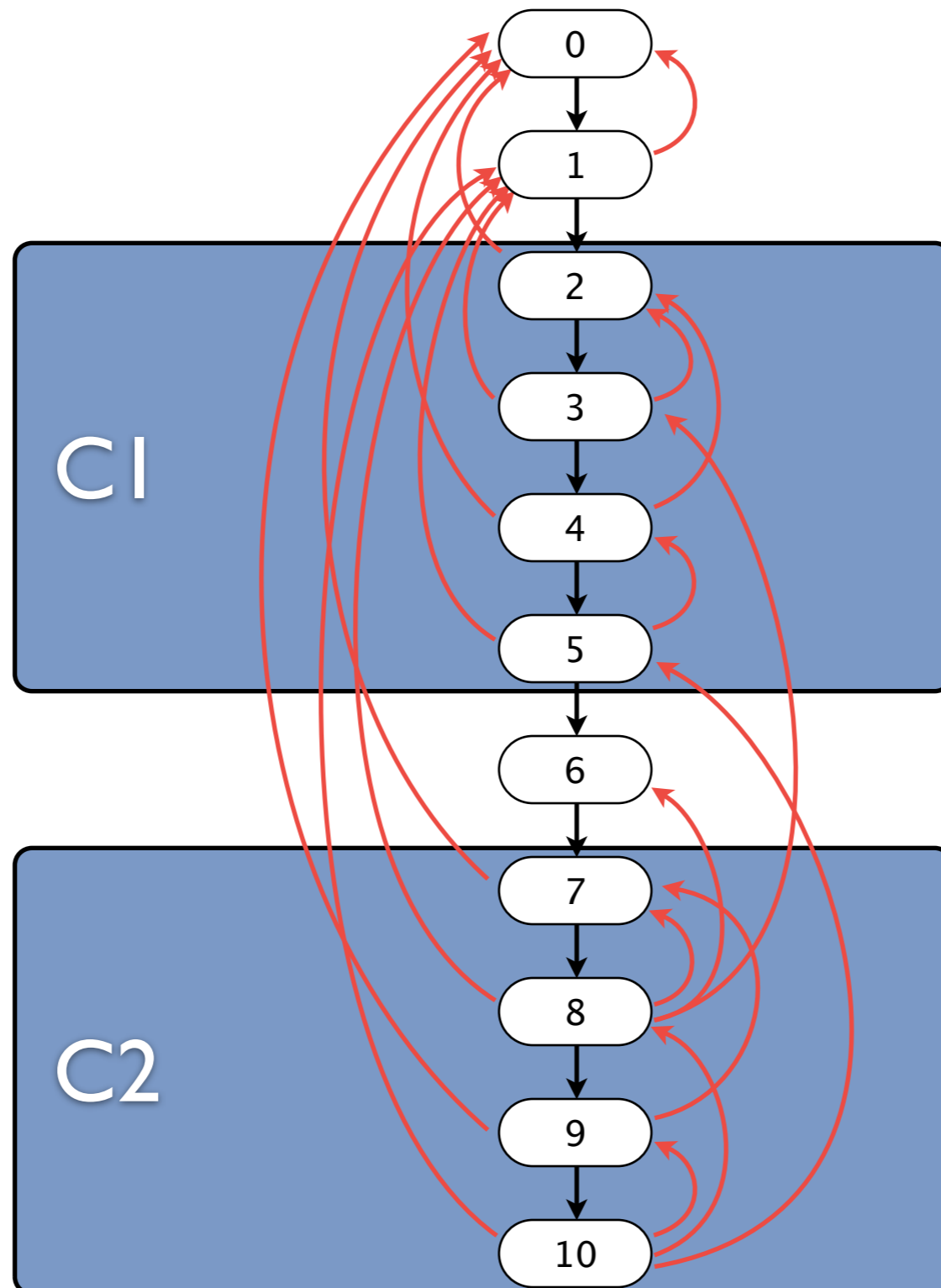
# Overall program analysis

| <i>program</i> | <i>description</i>        | <i>trace length</i><br><i>[<math>\times 10^6</math> insns]</i> | <i>potential</i> |
|----------------|---------------------------|--|------------------|
| antlr          | parser generator          | 212  | 384.28           |
| jython         | python interpreter        | 2,812  | 185.13           |
| pmd            | Java source code analyzer | 19   | 84.09            |

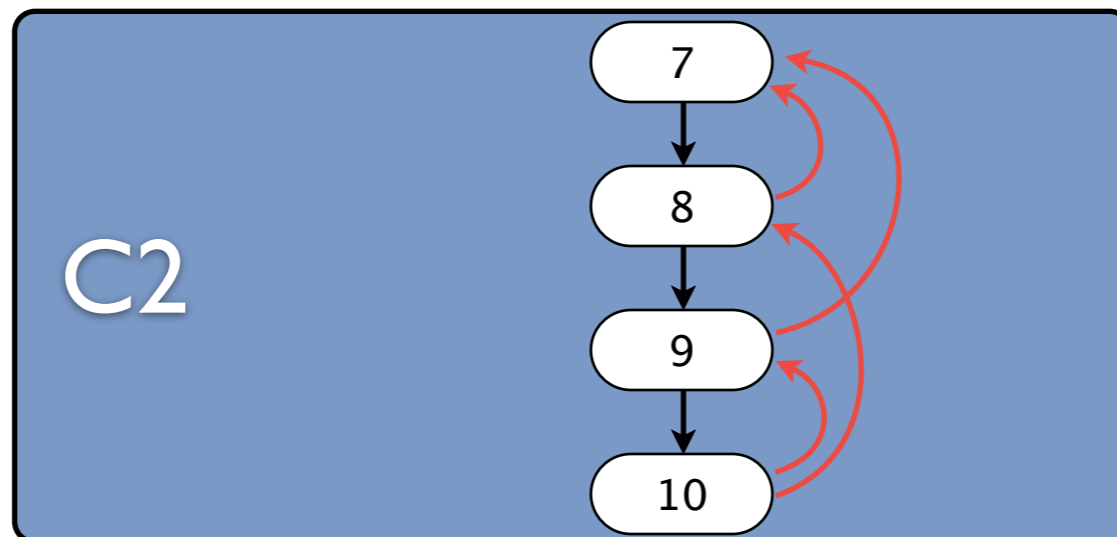
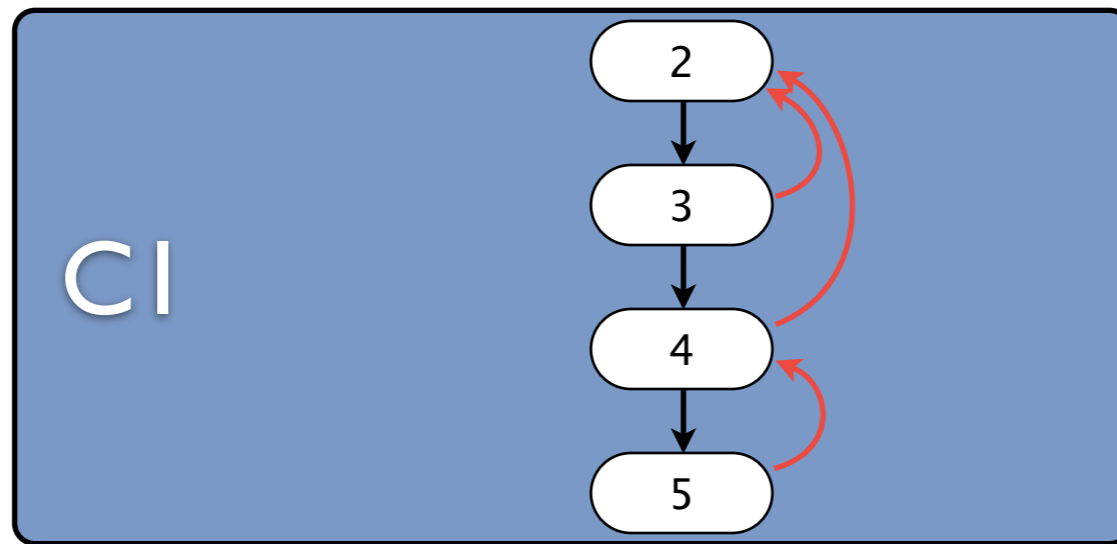
# Parallelization Candidates

```
public class SumUp {  
  
    public static void main(String[] args) {  
        int n = Integer.parseInt(args[0]);  
        long[] sums = new long[n];  
  
        C1  
        for (int i=0; i<n; ++i) {  
            sums[i] = sumTo(i);  
        }  
  
        long overallSum = 0;  
  
        C2  
        for (int i=0; i<n; ++i) {  
            overallSum += sums[i];  
        }  
  
    }  
  
    private static long sumTo(int n) {  
        return n == 0 ? 0 : n + sumTo(n-1);  
    }  
}
```

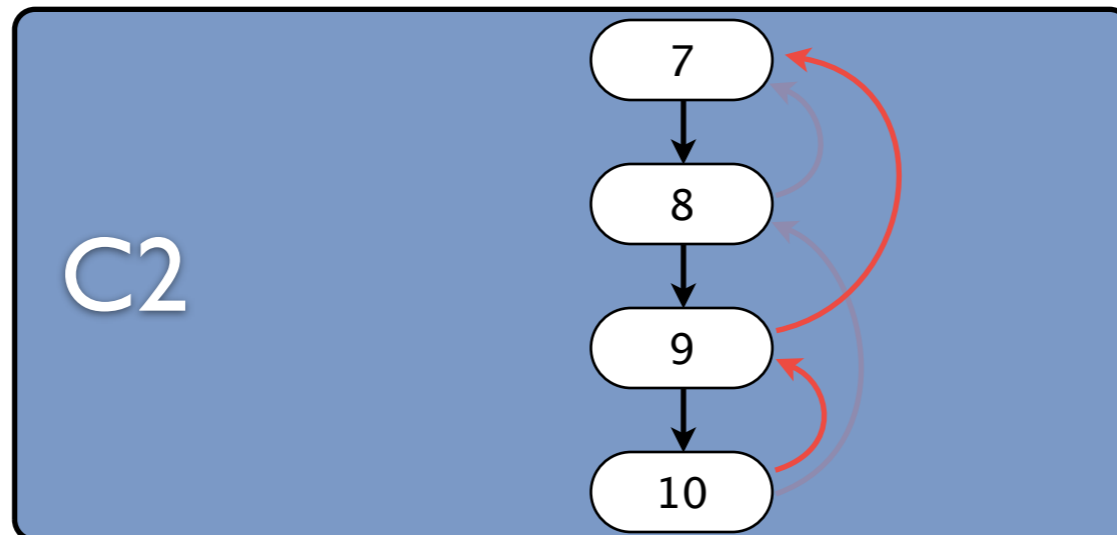
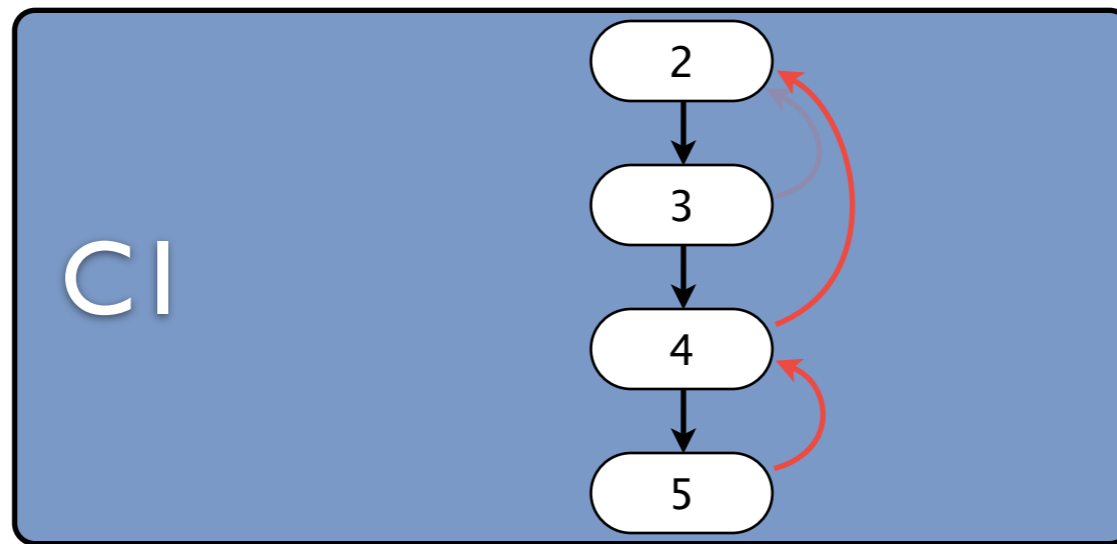
# Execution Trace



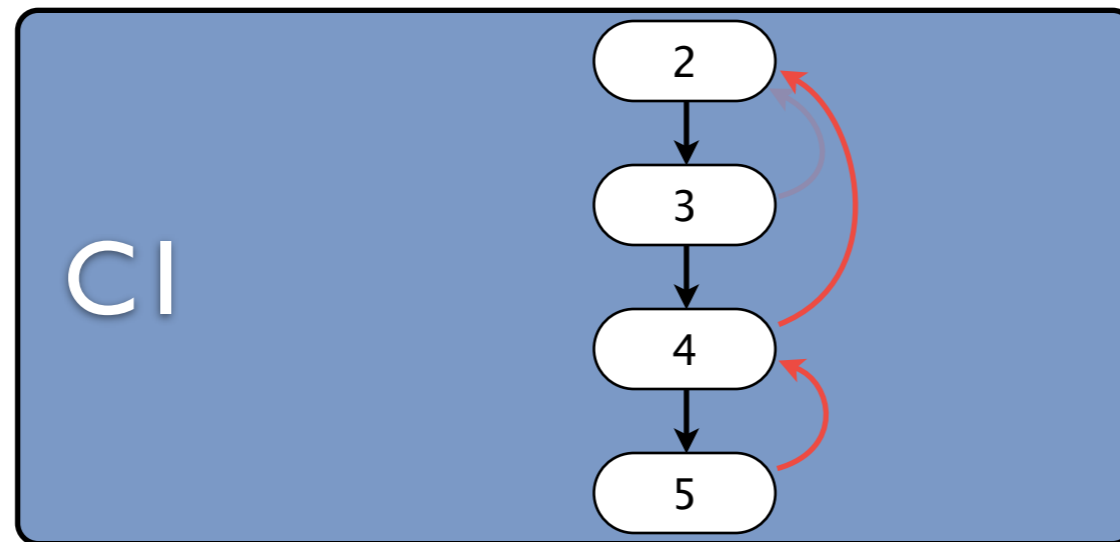
# Execution Trace



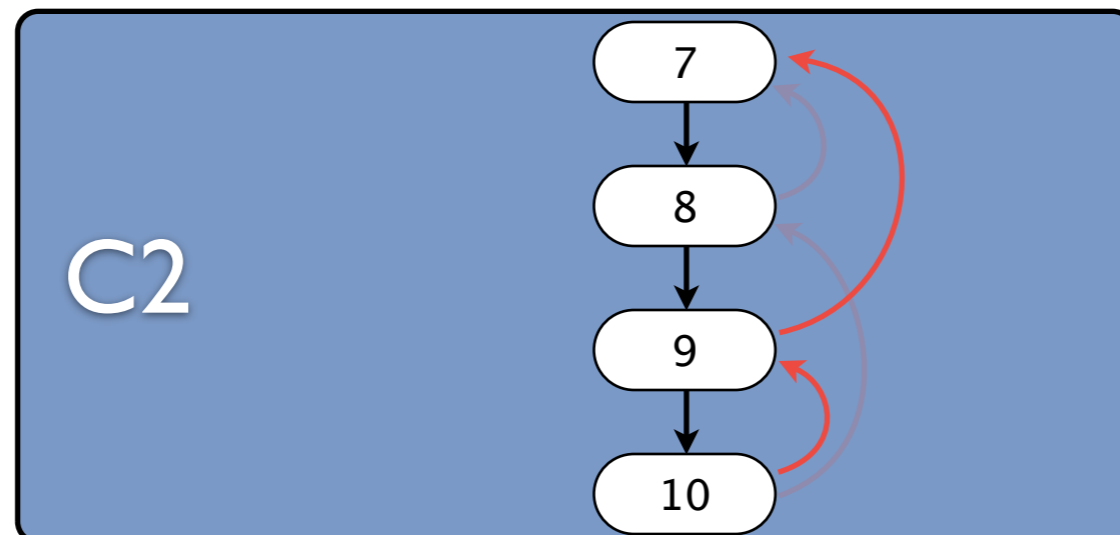
# Critical Path



# Critical Path



Potential:  $\frac{4}{3} = 1.33$



Potential:  $\frac{4}{3} = 1.33$

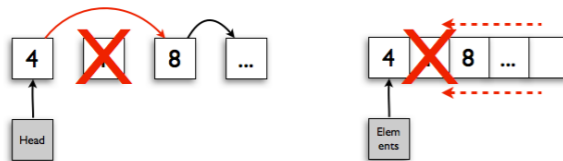
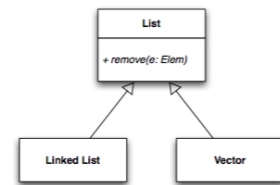
# Work in Progress

- Automatically parallelizing just-in-time compiler
- Continuously enhancing parallelized regions
  - choose reasonable initial configuration
  - keep tracked information across program runs
- Fallback to serial execution on collisions

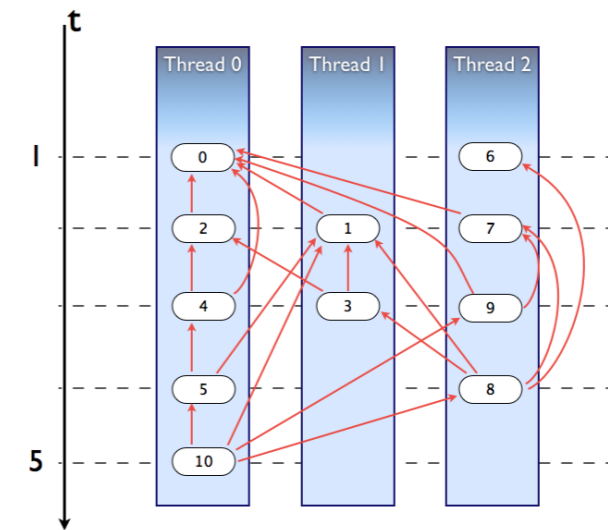
# Summary

## Dynamic Binding

```
public void removeAll(  
    List A, List B)  
{  
    for (Object elem: B) {  
        A.remove(elem);  
    }  
}
```



## Parallel Execution



## Parallelization Candidates

```
public class SumUp {  
    public static void main(String[] args) {  
        int n = Integer.parseInt(args[0]);  
        long[] sums = new long[n];  
  
        C1 for (int i=0; i<n; ++i) {  
            sums[i] = sumTo(i);  
        }  
  
        long overallSum = 0;  
  
        C2 for (int i=0; i<n; ++i) {  
            overallSum += sums[i];  
        }  
  
        private static long sumTo(int n) {  
            return n == 0 ? 0 : n + sumTo(n-1);  
        }  
    }  
}
```

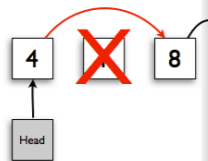
## Work in Progress

- Automatically parallelizing just-in-time compiler
- Continuously enhancing parallelized regions
  - choose reasonable initial configuration
  - keep tracked information across program runs
- Fallback to serial execution on collisions

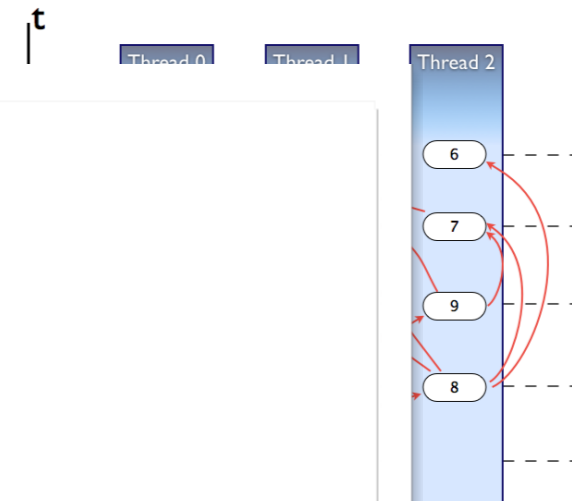
# Summary

## Dynamic Binding

```
public void removeAll(  
    List A, List B  
)  
{  
    for (Object elem:  
        A.remove(elem))  
    }  
}
```



## Parallel Execution



Can we dynamically  
extend static analyses?

## Paralleliz

```
public cl  
publ
```

C1

C2

```
    overallSum += sumTo(i);  
}  
  
private static long sumTo(int n) {  
    return n == 0 ? 0 : n + sumTo(n-1);  
}  
}
```

## gress

ist-in-time

allelized regions

configuration

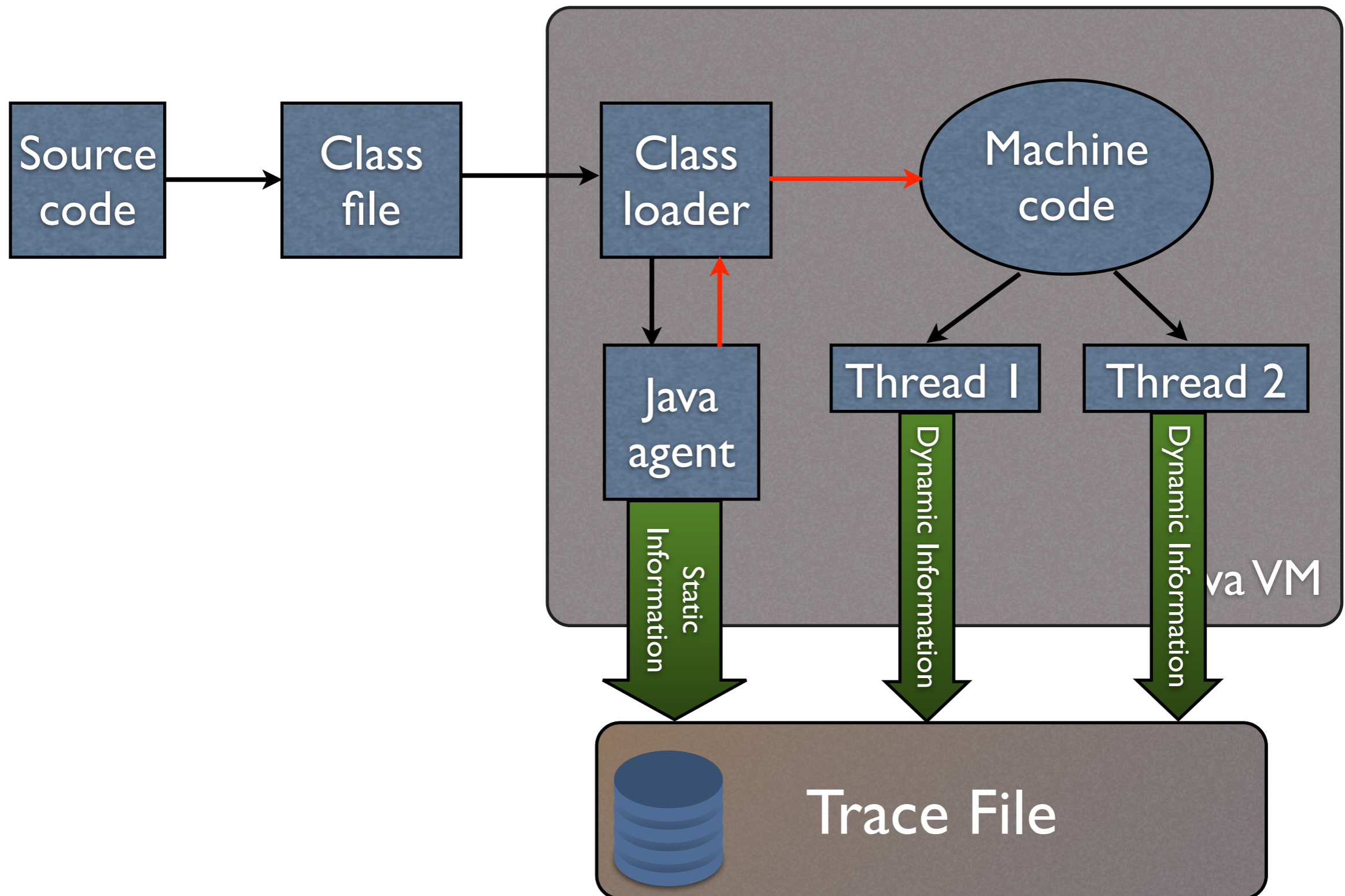
- keep tracked information across program runs
- Fallback to serial execution on collisions

# Appendix

# References

- Christoph von Praun, Rajesh Bordawekar, and Calin Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In PPOPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pages 185–196, New York, NY, USA, 2008. ACM.
- Clemens Hammacher. Design and Implementation of an Efficient Dynamic Slicer for Java. Bachelor's Thesis, November 2008. <http://www.st.cs.uni-saarland.de/javaslicer/>
- Clemens Hammacher, Kevin Streit, Sebastian Hack, and Andreas Zeller. Profiling Java Programs for Parallelism. In Proc. 2nd International Workshop on Multi-Core Software Engineering (IWMSE), May 2009.

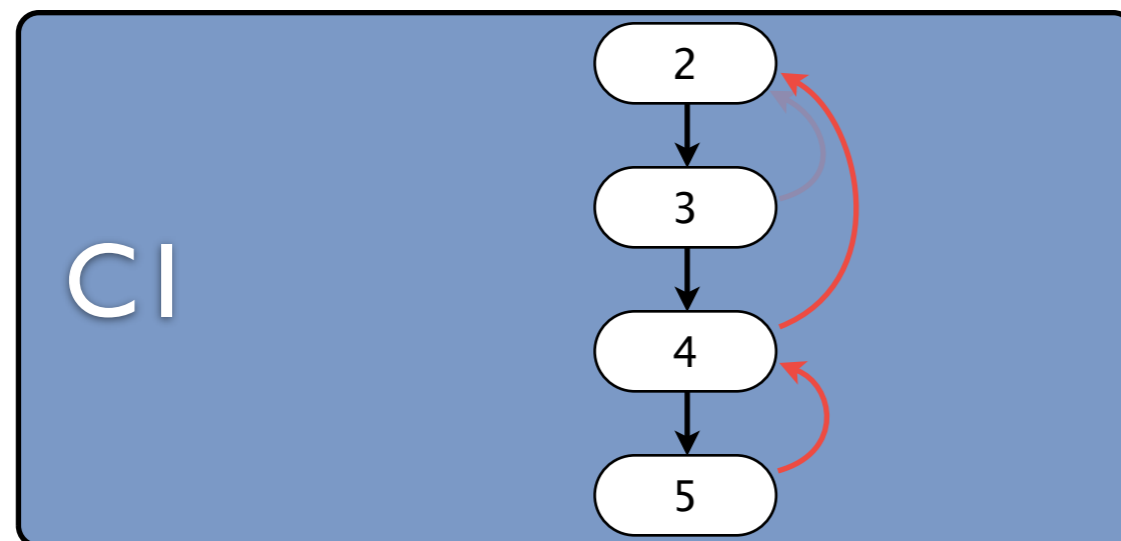
# Hooking into the VM



# Gain Measurement

$$\begin{aligned} \textit{gain} &= \left( 1 - \frac{1}{\textit{potential}} \right) \cdot \textit{influence} \\ &= \textit{influence} - \frac{\textit{influence}}{\textit{potential}} \end{aligned}$$

# Gain Measurement (Example)



Influence:  $\frac{4}{11}$

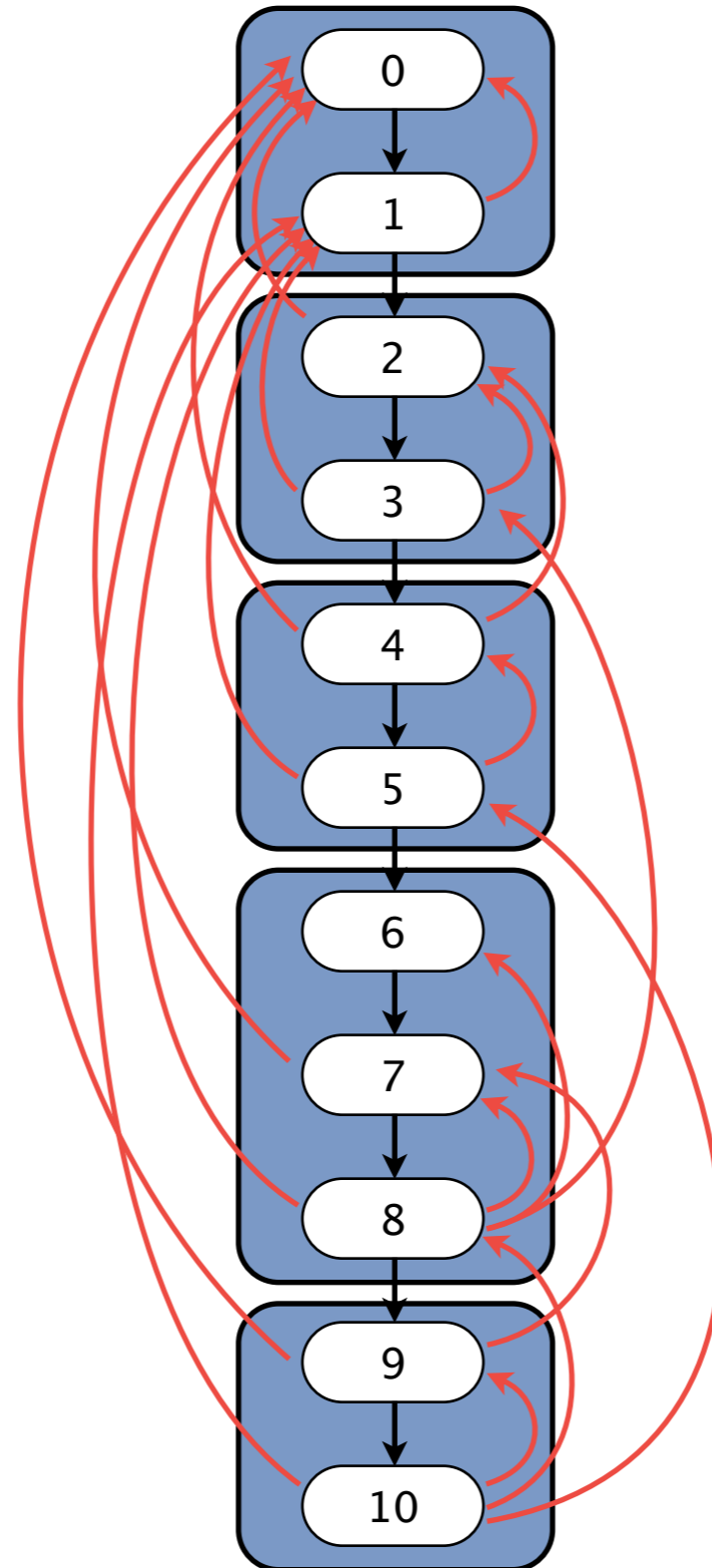
Potential:  $\frac{4}{3} = 1.33$

Gain:  $\frac{1}{4} \cdot \frac{4}{11} = \frac{1}{11}$

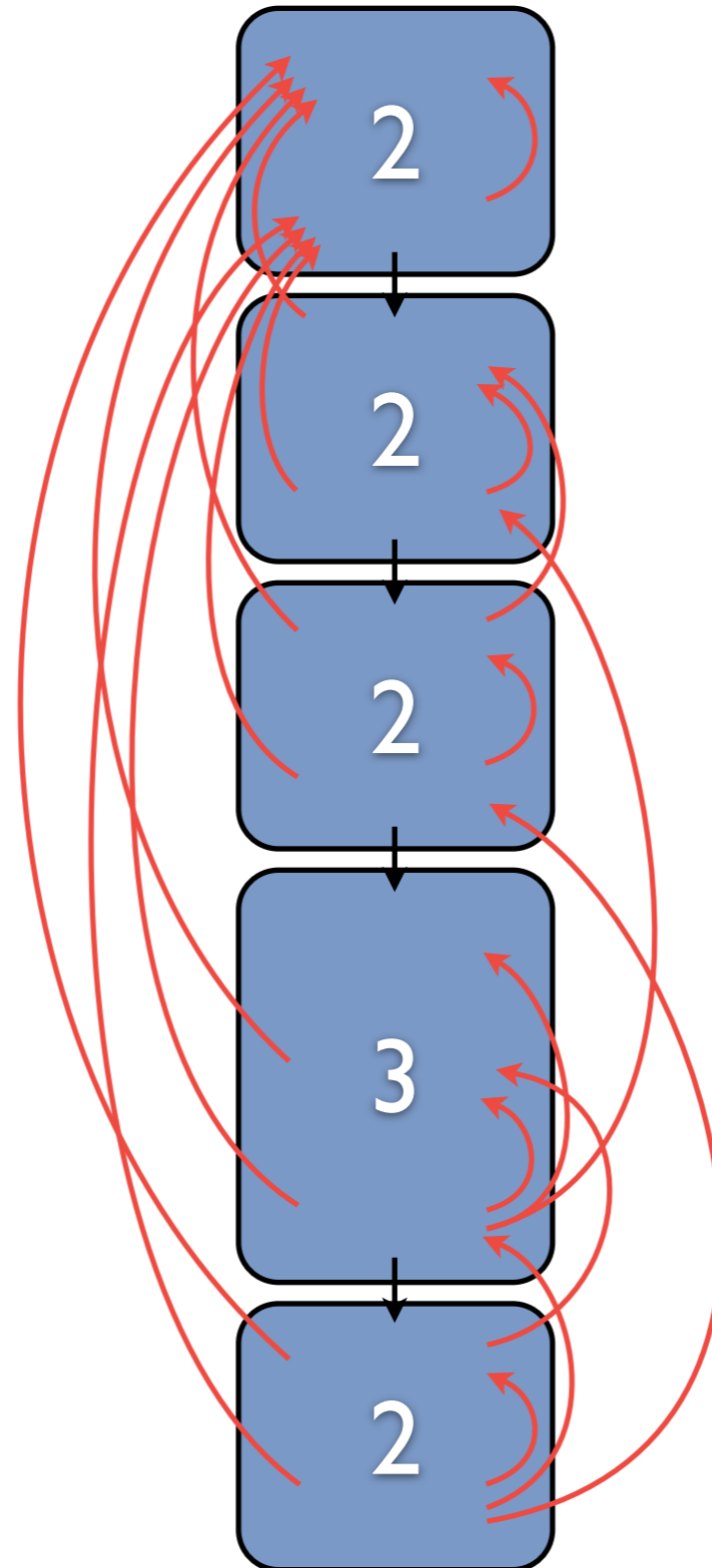
# Granularity

- so far: instruction level
- unrealistic for parallelization
- idea: basic blocks

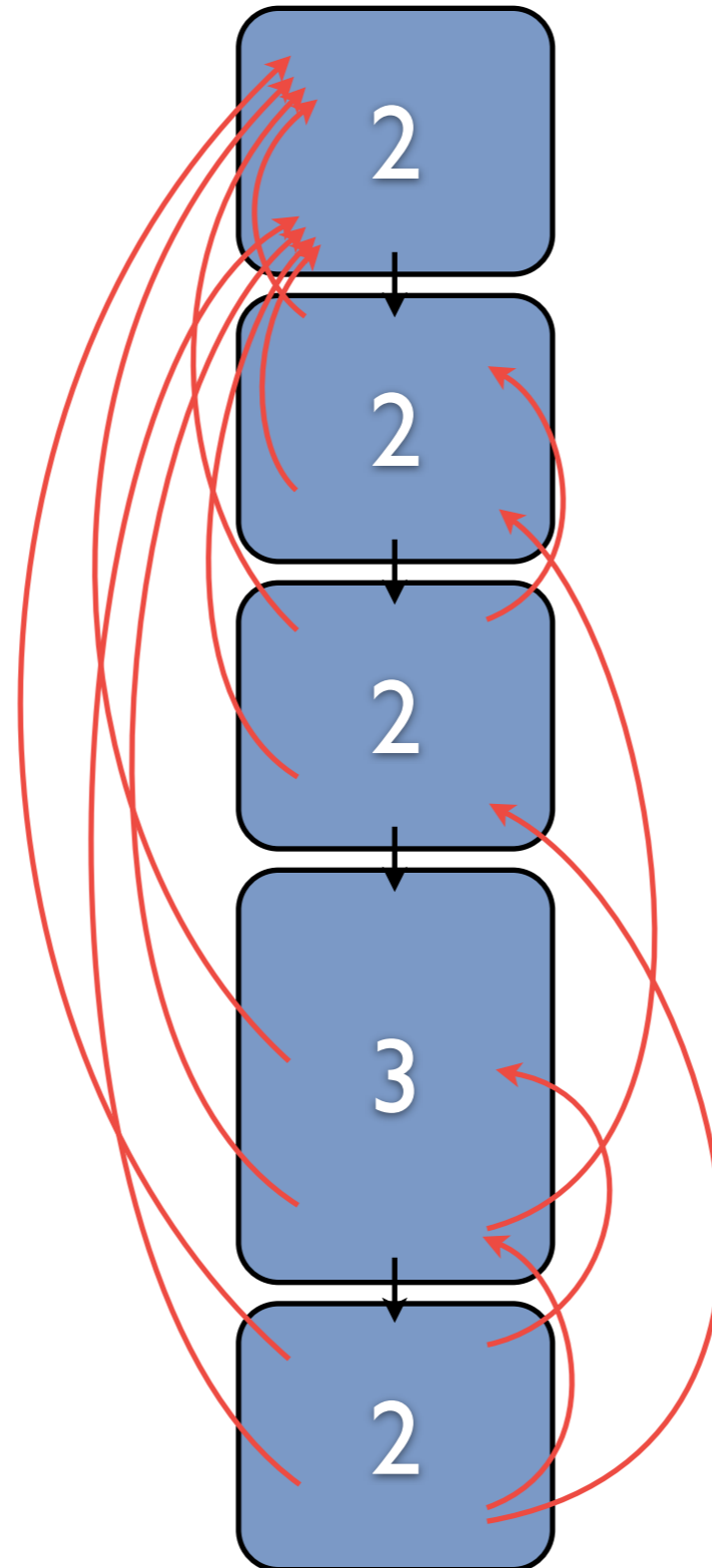
# Basic Blocks



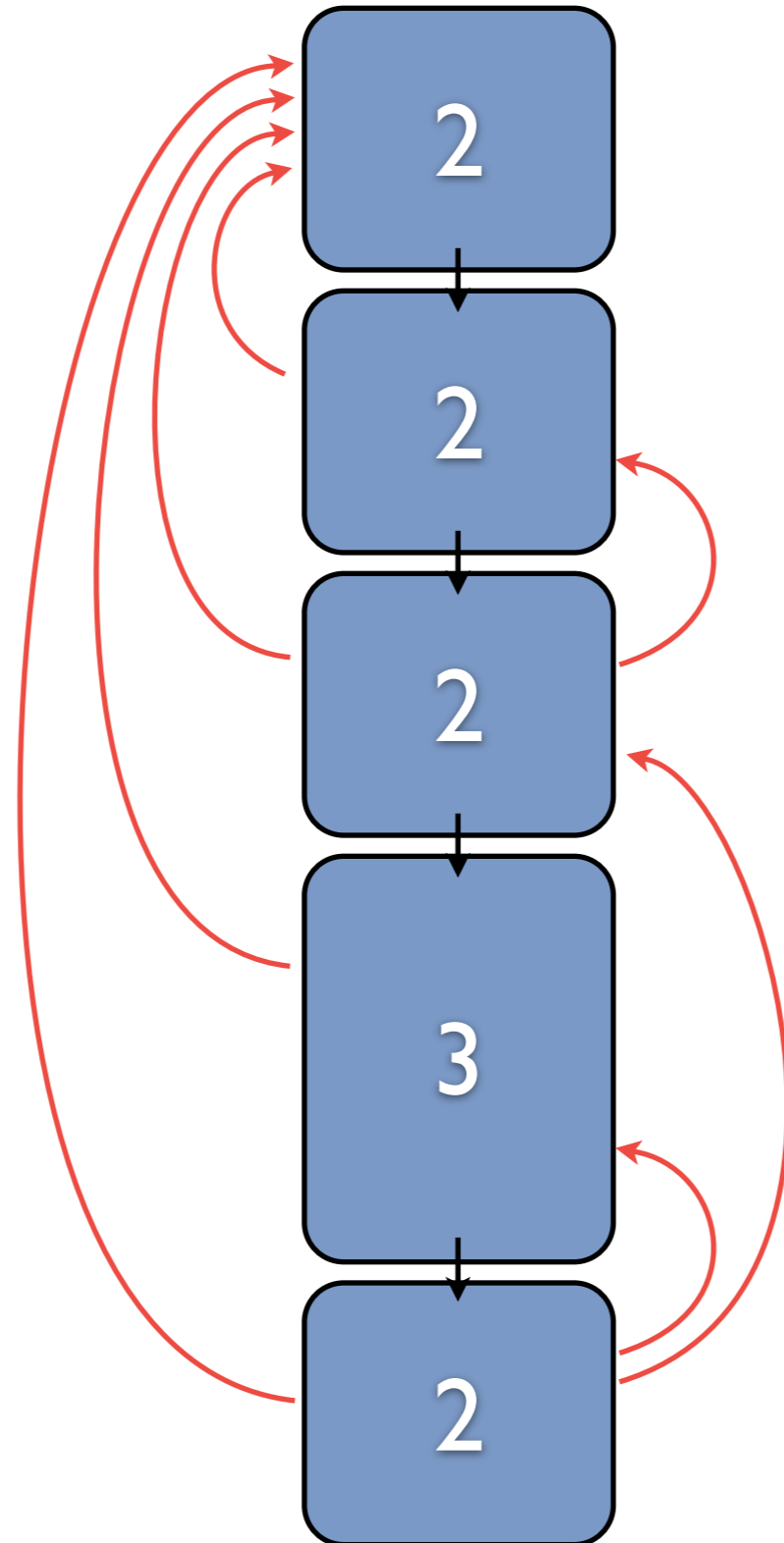
# Basic Blocks



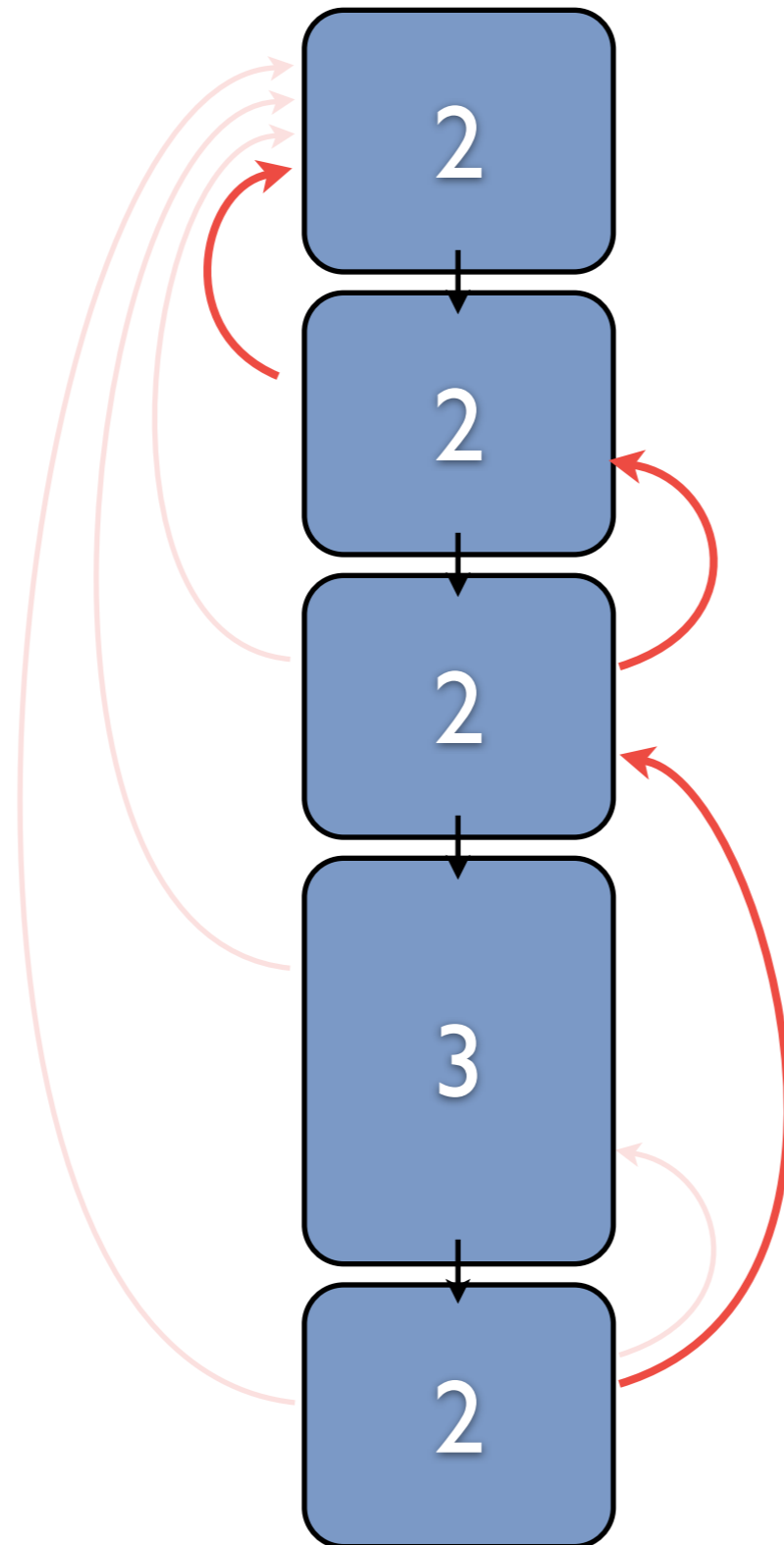
# Basic Blocks



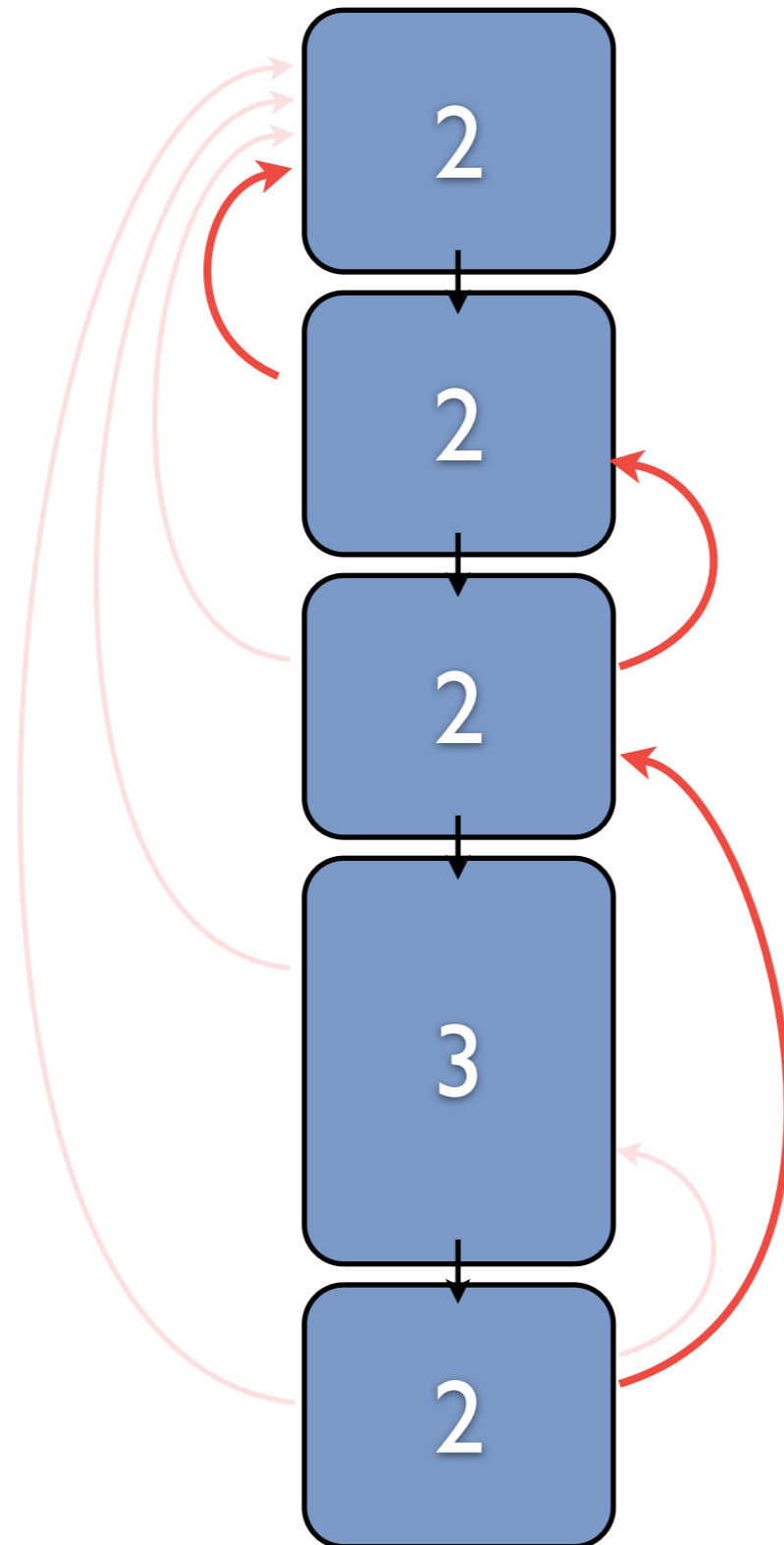
# Basic Blocks



# Basic Blocks

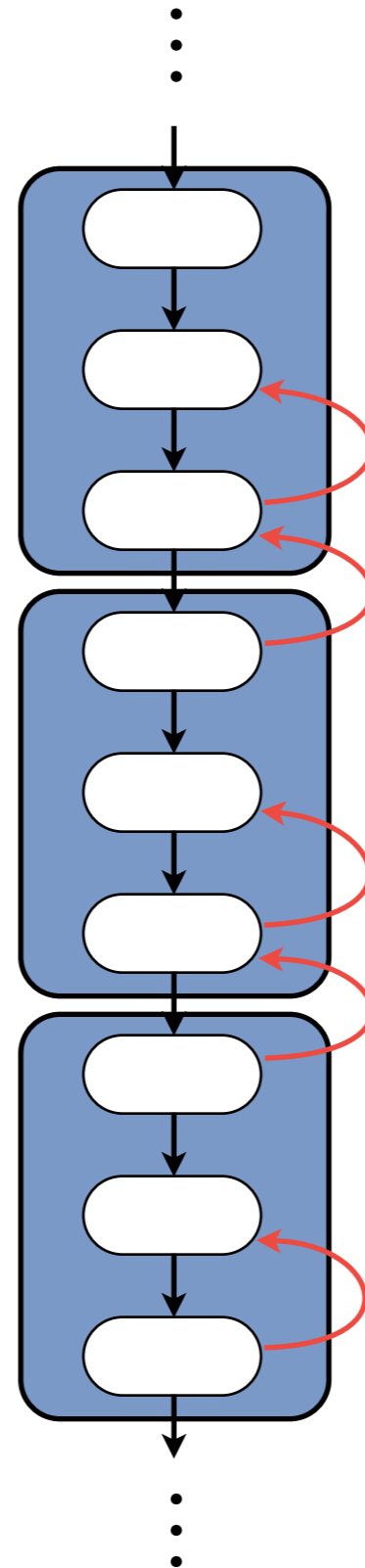


# Basic Blocks



$$\begin{aligned} \text{potential} &= \frac{11}{8} \\ &= 1.375 \end{aligned}$$

# Basic Block Artifacts



# Overall program analysis

| program | description               | trace length | potential (instructions) | potential (blocks) |
|---------|---------------------------|--------------|--------------------------|--------------------|
| antlr   | parser generator          | 212          | 384.28                   | 14.01              |
| jython  | python interpreter        | 2812         | 185.13                   | 16.83              |
| pmd     | Java source code analyzer | 19           | 84.09                    | 8.16               |