NOVA SCHOOL OF SCIENCE & TECHNOLOGY
COMPUTER SCIENCE DEPARTMENT

Parallel Programming Models and Dependences

lecture 05 (2021-03-29)

Master in Computer Science and Engineering

— Concurrency and Parallelism / 2020-21 —

João Lourenço <joao.lourenco@fct.unl.pt>

NOVA SCHOOL OF SCIENCE & TECHNOLOGY
COMPUTER SCIENCE DEPARTMENT

Parallel Programming Models and Dependences

lecture 05 (2021-03-29)

Master in Computer Science and Engineering

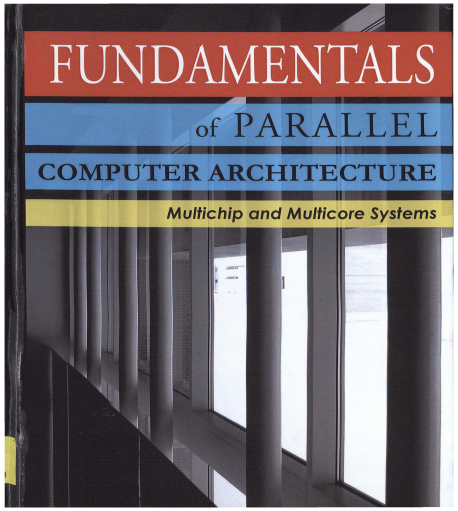
— Concurrency and Parallelism / 2020-21 —

João Lourenço <joao.lourenco@fct.unl.pt>

Outline

- Parallel programming models
- Statement dependences
- Loop dependences

– Bibliography:
  - **(Part of) Chapter 4** of book
    Yan Solihin;
    Fundamentals of Parallel Computer Architecture;
    Solihin Books (2009);
    ISBN: 978-0-98-416300-7



Parallelism, Correctness, and Dependences

- Parallel execution shall always be constrained by the sequence of operations needed to be performed for a correct result
- Parallel execution must address control, data, and system dependences
- A *dependence* arises when one operation (A) depends on an earlier operation (B) to complete and produce a result before (A) can be performed
  - We extend this notion of dependence to resources since some operations may depend on certain resources (e.g., due to where data is located)

Executing Two Statements in Parallel

- Want to execute two statements in parallel
- On a single processor:

```
Processor 1:
   Statement 1;
   Statement 2;
```

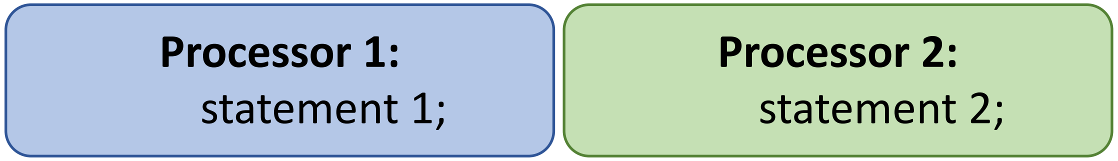
- On two processors:

```
Processor 1:                  Processor 2:
   Statement 1;                  Statement 2;
```

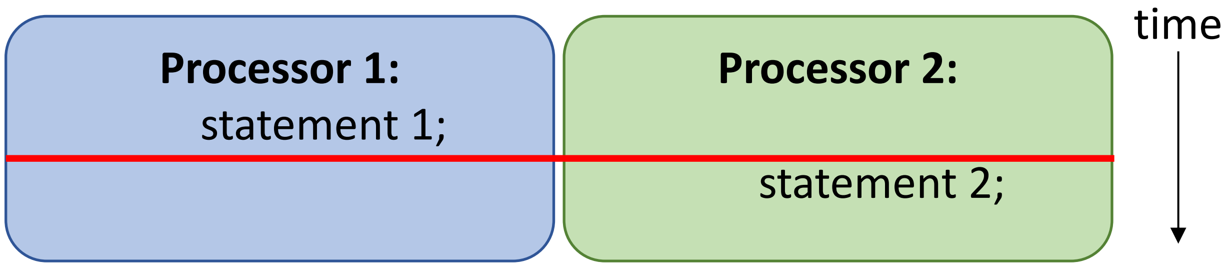
- Fundamental (*concurrent*) execution assumption
  - Processors execute independent of each other
  - No assumptions made about speed of processor execution

Sequential Consistency in Parallel Execution

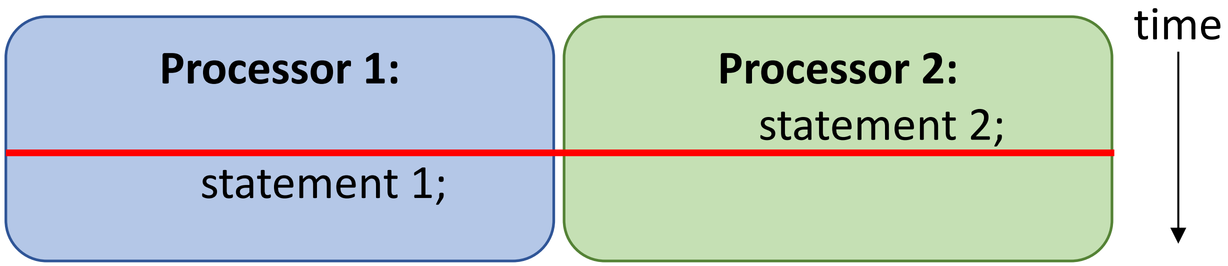
- Parallel execution of



- Case 1:

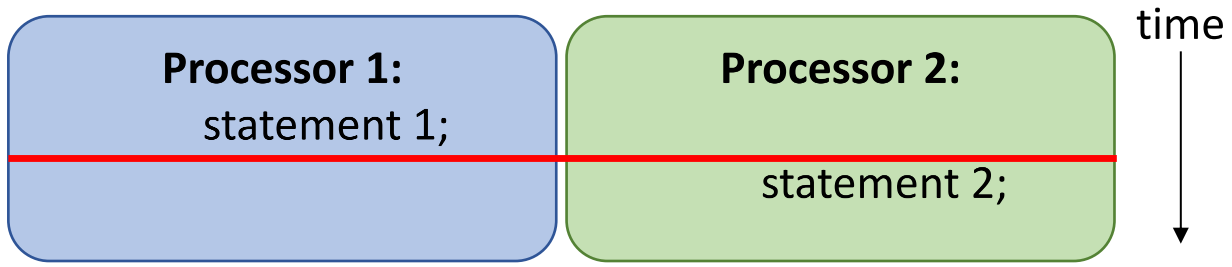


- Case 2:

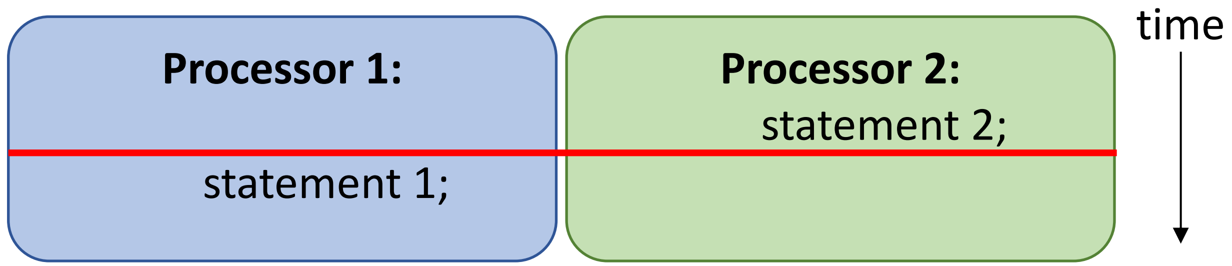


Sequential Consistency in Parallel Execution

- Sequential consistency
  - Statement execution does not interfere with each other
  - Computation result equal to either "Case 1" or "Case 2"
- Case 1:



- Case 2:



Independent versus Dependent Statements

- When the execution of

```
statement1;
statement2;
```

  is equivalent to

```
statement2;
statement1;
```

- Their order of execution must not matter!
- That means the statements are *independent* of each other
- Two statements are *dependent* when the order of their execution affects the computation outcome

True Dependence and Anti-Dependence

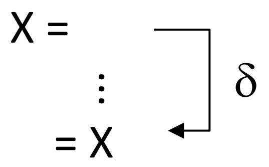
- Given statements S1 and S2 written as,

  S1;

  S2;

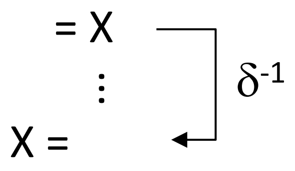
- S2 has a *true (flow) dependence* on S1

  if and only if **S2 reads a value written by S1**
  (RAW – Read After Write)



- S2 has a *anti-dependence* on S1

  if and only if **S2 writes a value read by S1**
  (WAR – Write After Read)



Output Dependence

- Given statements S1 and S2 written as,

  S1;

  S2;

- S2 has an *output dependence* on S1

  if and only if **S2 writes a variable written by S1** (WAW – Write After Write)

$$\begin{tikzcd} X = \arrow[dd, bend left, "\delta^o"] \\ \vdots \\ X = \end{tikzcd}$$

- Anti- and output dependences are "name" dependences
  - How can we get rid of anti- and output dependences?

Examples

- Example 1

```
S1: a=1;
S2: b=1;
```

- ❐ Statements are independent

- Example 2

```
S1: a=1;
S2: b=a;
```

- ❐ Dependent (*true (flow) dependence*)
  - ❍ Second is dependent on first
  - ❍ Can you remove dependence?

- Example 3

```
S1: a=f(x);
S2: a=b;
```

- ❐ Dependent (*output dependence*)
  - ❍ Second is dependent on first
  - ❍ Can you remove dependence? How?

- Example 4

```
S1: a=b;
S2: b=1;
```

- ❐ Dependent (*anti-dependence*)
  - ❍ First is dependent on second
  - ❍ Can you remove dependence? How?

Statement Dependence Graphs

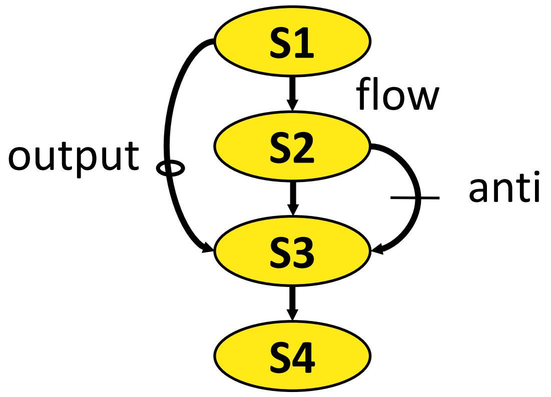
- Can use graphs to show dependence relationships
- Example

  S1: a=1;

  S2: b=a;

  S3: a=b+1;

  S4: c=a;



- $S_1 \ \delta \ S_2$: S_2 is flow-dependent on S_1
- $S_1 \ \delta^0 \ S_3$: S_3 is output-dependent on S_1
- $S_2 \ \delta^{-1} \ S_3$: S_3 is anti-dependent on S_2

When can two statements execute in parallel?

- Statements S1 and S2 can execute in parallel if and only if there are *no dependences* between them, i.e., no
  - *True dependences*; nor
  - *Anti-dependences*; nor
  - *Output dependences*.
- Some dependences can be removed by modifying the program
  - Rearranging statements
  - Eliminating statements

How do you compute dependences?

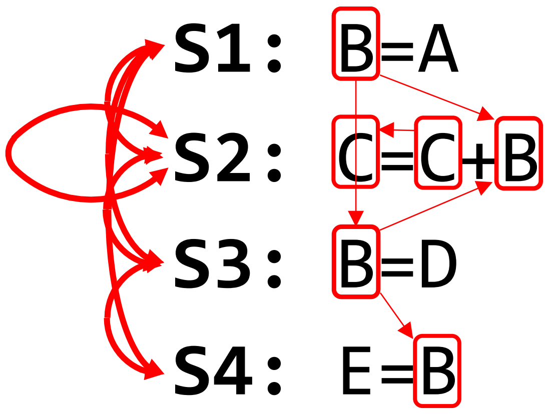
- Data dependence relations can be found by comparing the IN and OUT sets of each node
- The IN and OUT sets of a statement S are defined as:
  - IN(S) : set of memory locations (variables) that may be used (read) in S
  - OUT(S) : set of memory locations (variables) that may be modified (written) by S
- Note that these sets include all memory locations that may be fetched or modified
  - As such, the sets can be conservatively large

IN / OUT Sets and Computing Dependences

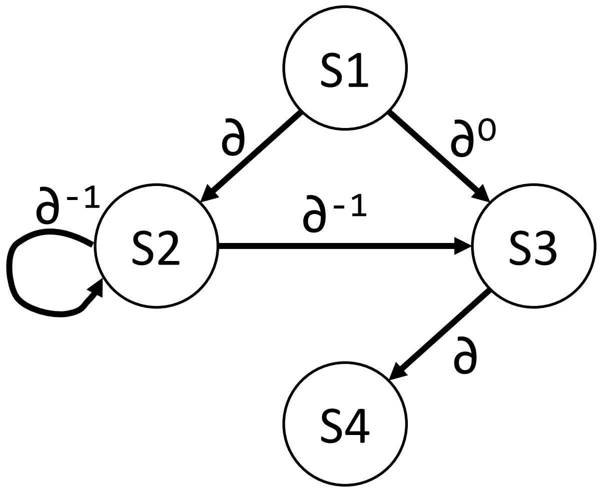
- Assuming that there is an execution path from S1 to S2 , the following shows how to intersect their IN and OUT sets to test for data dependence

$out(S_1) \cap in(S_2) \neq \emptyset$	$S_1 \, \delta \, S_2$	flow dependence
$in(S_1) \cap out(S_2) \neq \emptyset$	$S_1 \, \delta^{-1} \, S_2$	anti - dependence
$out(S_1) \cap out(S_2) \neq \emptyset$	$S_1 \, \delta^{0} S_2$	output dependence

Example



```
S1:  B=A
S2:  C=C+B
S3:  B=D
S4:  E=B
```



Loop-Level Parallelism

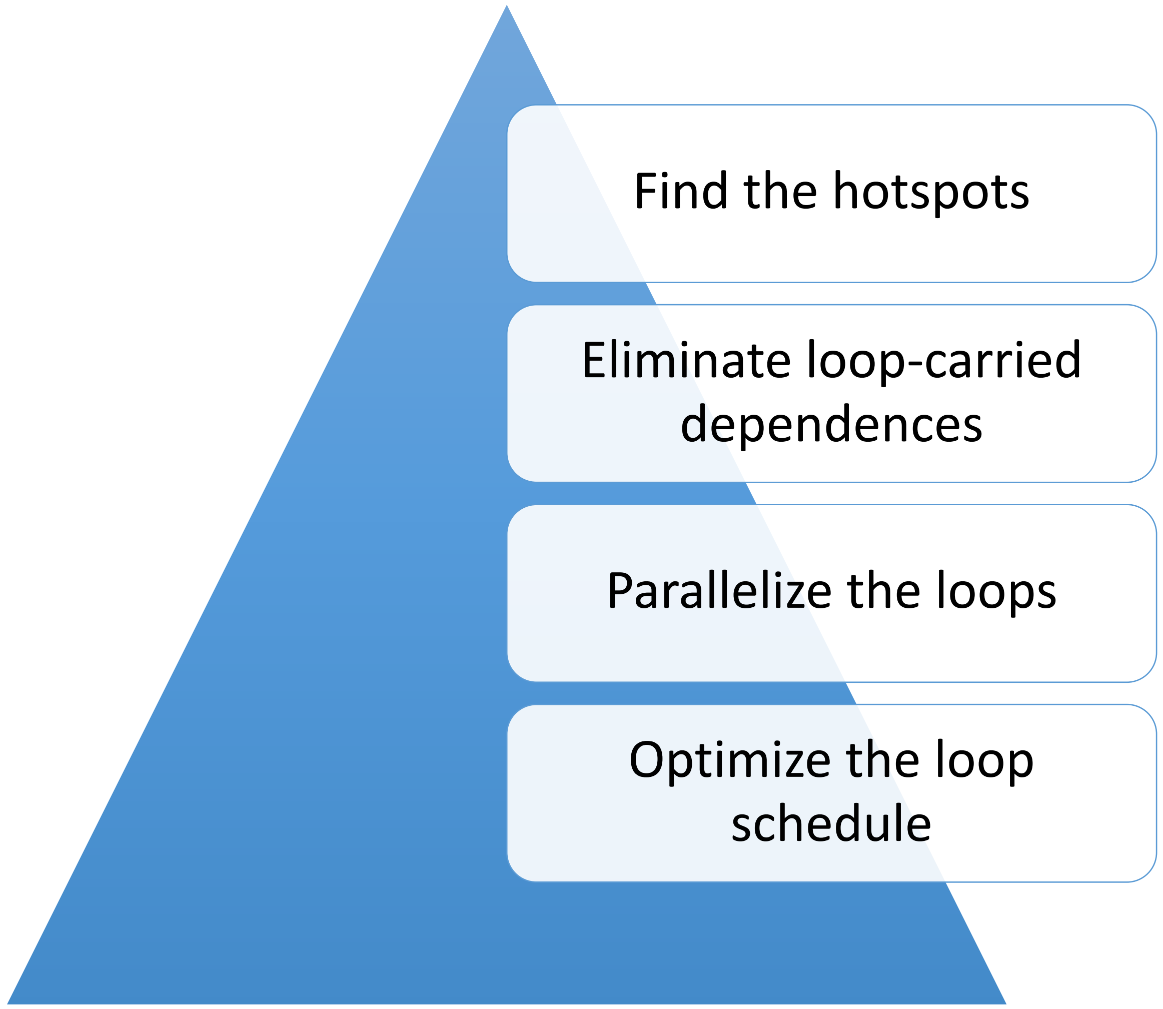
- Significant parallelism can be identified **within loops**

```
for (i=0; i<100; i++)
    S1: a[i] = i;
```

```
parallel_for (i=0; i<100; i++)
{
    S1: a[i] = i;
    S2: b[i] = 2*a[i];        ?
}
```

- Dependences? What about *i*, the loop index?
- *DOALL* loop (a.k.a. *foreach* loop)
  - All iterations are independent of each other
  - All statements will be executed in parallel at the same time
    - Is this really true?

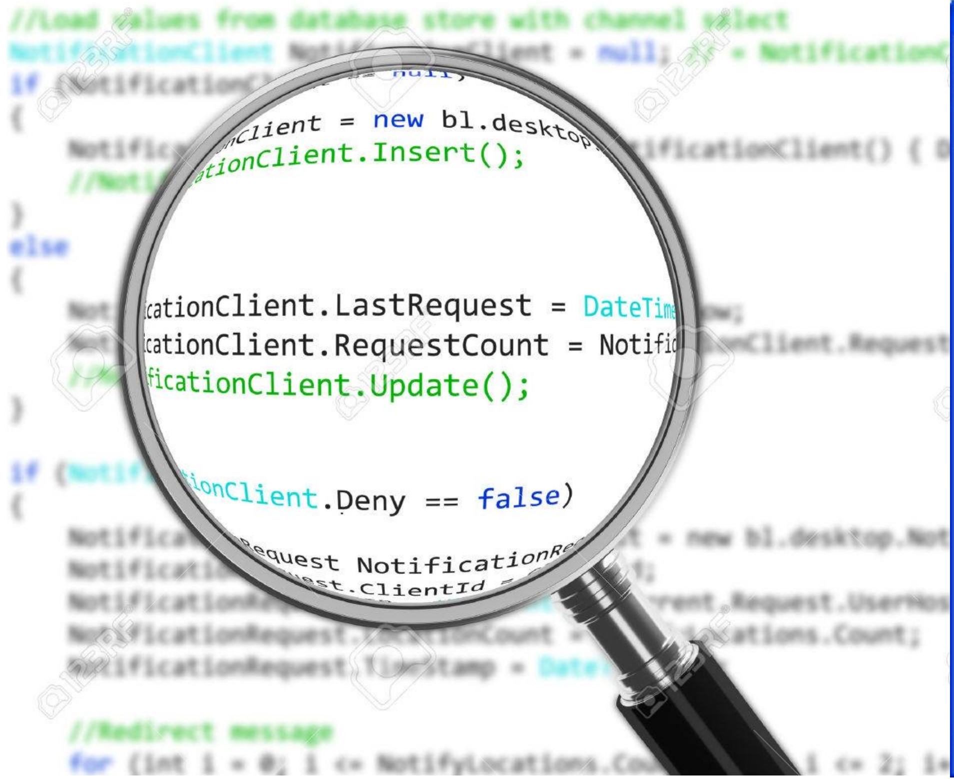
General Approach for Loop Parallelism


Find the hotspots
Eliminate loop-carried dependences
Parallelize the loops
Optimize the loop schedule

Find the hotspots

- By code inspection

- By using performance analysis tools

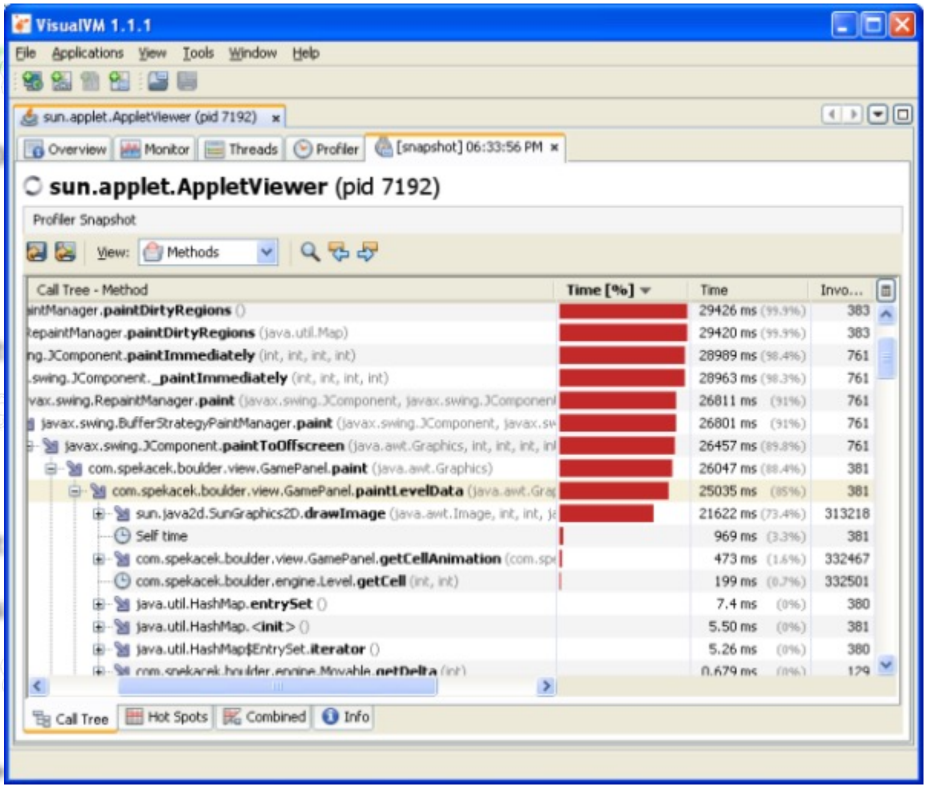


```
nClient = new bl.desktop
ationClient.Insert();

cationClient.LastRequest = DateTim
cationClient.RequestCount = Notific
ficationClient.Update();

ionClient.Deny == false)

Request NotificationRe
```


VisualVM 1.1.1
File Applications View Tools Window Help
sun.applet.AppletViewer (pid 7192)
Overview Monitor Threads Profiler [snapshot] 06:33:56 PM
Profiler Snapshot
View: Methods
Call Tree Hot Spots Combined Info

sun.applet.AppletViewer (pid 7192)

Call Tree - Method	Time [%]	Time	Invo...
intManager.**paintDirtyRegions** ()	🟥	29426 ms (99.9%)	383
RepaintManager.**paintDirtyRegions** (java.util.Map)	🟥	29420 ms (99.9%)	383
ng.JComponent.**paintImmediately** (int, int, int, int)	🟥	28989 ms (98.4%)	761
.swing.JComponent.**_paintImmediately** (int, int, int, int)	🟥	28963 ms (98.3%)	761
vax.swing.RepaintManager.**paint** (javax.swing.JComponent, javax.swing.JComponent	🟥	26811 ms (91%)	761
javax.swing.BufferStrategyPaintManager.**paint** (javax.swing.JComponent, javax.sw	🟥	26801 ms (91%)	761
javax.swing.JComponent.**paintToOffscreen** (java.awt.Graphics, int, int, int, in	🟥	26457 ms (89.8%)	761
com.spekacek.boulder.view.GamePanel.**paint** (java.awt.Graphics)	🟥	26047 ms (88.4%)	381
com.spekacek.boulder.view.GamePanel.**paintLevelData** (java.awt.Gra	🟥	25035 ms (85%)	381
sun.java2d.SunGraphics2D.**drawImage** (java.awt.Image, int, int, ja	🟥	21622 ms (73.4%)	313218
Self time	🟥	969 ms (3.3%)	381
com.spekacek.boulder.view.GamePanel.**getCellAnimation** (com.spe	🟥	473 ms (1.6%)	332467
com.spekacek.boulder.engine.Level.**getCell** (int, int)	🟥	199 ms (0.7%)	332501
java.util.HashMap.**entrySet** ()		7.4 ms (0%)	380
java.util.HashMap.**<init>** ()		5.50 ms (0%)	381
java.util.HashMap$EntrySet.**iterator** ()		5.26 ms (0%)	380
com.spekacek.boulder.engine.Movable.**getDelta** (int)		0.679 ms (0%)	129

Eliminate loop-carried dependences

- Statements’ dependences include true dependences, anti-dependences and output dependences.
- Loop dependences also include those, carried from one execution of the loop to another.

Loop Dependences

- A *loop-carried* dependence is a dependence between two statements instances in two different iterations of a loop
- Otherwise, it is *loop-independent*
- Loop-carried dependences can prevent loop iteration parallelization

Loop Dependences

- A *loop-carried* dependence is a dependence between two statements instances in two different iterations of a loop

```
S1: a = 5;
S2: b = a;
```

True dependence — the memory location ‘a’ is written (in S1) before it is read (in S2)

S1 ∂ S2

```
for (i=1; i<n; i++) {
    S1: a[i] = a[i-1];
}
```

True dependence — a memory location ’a[j]’ is written before it is read in the next iteration of the loop

S1[j] ∂ S1[j+1]

Loop Dependences

- A *loop-carried* dependence is a dependence between two statements instances in two different iterations of a loop

```
S1: b = a;
S2: a = 5;
```

Anti-dependence — the memory location ‘a’ is read (in S1) before it is written (in S2)

S1 ∂^{-1} S2

```
for (i=0; i<n-1; i++) {
    S1: a[i] = a[i+1];
}
```

Anti-dependence — a memory location ’a[j]’ is read before it is written in the next iteration of the loop

S1[j] ∂^{-1} S1[j+1]

Loop Dependences

- A *loop-carried* dependence is a dependence between two statements instances in two different iterations of a loop

```
S1: c = 8;
S2: c = 15;
```

Output dependence — the same memory location ‘c’ is written (in S1) and then written once again (in S2)

S1 ∂^o S2

```
for (i=0; i<n; i++) {
    S1: c[i] = i;
    S2: c[i+1] = 5;
}
```

Output dependence — the same memory location ’a[j]’ is written (in S2) and then written again in the next iteration of the loop (in S1)

S2[j] ∂^o S1[j+1]

Loop dependences: examples

- The following loop cannot be parallelized (without rewriting)

```
a[0] = 1;
for (i=1; i<N; i++) {
    a[i] = a[i] + a[i-1];
}
```

```
i=1: a[1] = a[1] + a[0];
i=2: a[2] = a[2] + a[1];
i=3: a[3] = a[3] + a[2];
...
```

Each iteration depends on the result of the preceding iteration

Detecting dependences

- Analyze how each variable is used within a loop iteration:
- Is the variable read and never written?

  => no dependences!
- For each written variable: can there be any accesses in other iterations than the current?

  => there are dependences!

Simple rule of thumb

- A loop that matches the following criteria has no dependences and can be parallelized:

1. All assignments to shared data are to arrays:
2. Each element is assigned by at most one iteration; and
3. No iteration reads elements assigned by any other iteration.

Example 1

- Is this loop parallelizable?

```
for (i=1; i<N; i+=2) {
    a[i] = a[i] + a[i-1];
}
```

1. All assignments to shared data are to arrays:
2. Each element is assigned by at most one iteration; and
3. No iteration reads elements assigned by any other iteration.

```
i=1: a[1] = a[1] + a[0];
i=3: a[3] = a[3] + a[2];
i=5: a[5] = a[5] + a[4];
...
```

No dependences!

YES!! It is parallelizable!

Example 2

- Is this loop parallelizable?

```
for (i=0; i<N/2; i++) {
    a[i] = a[i] + a[i+N/2];
}
```

1. All assignments to shared data are to arrays:
2. Each element is assigned by at most one iteration; and
3. No iteration reads elements assigned by any other iteration.

```
i=0: a[0] = a[0] + a[0+N/2];
i=1: a[1] = a[1] + a[1+N/2];
...
i=N/2-1: a[N/2-1] = a[N/2-1] + a[N-1];
```

No dependences!

YES!! It is parallelizable!

Example 3

- Is this loop parallelizable?

```
for (i=0; i<=N/2; i++) {
    a[i] = a[i] + a[i+N/2];
}
```

1. All assignments to shared data are to arrays:
2. Each element is assigned by at most one iteration; and
3. No iteration reads elements assigned by any other iteration.

```
i=0: a[0] = a[0] + a[0+N/2];
i=1: a[1] = a[1] + a[1+N/2];
...
i=N/2: a[N/2] = a[N/2] + a[N];
```

Loop carried true dependence

It is NOT parallelizable!

Example 4

- Is this loop parallelizable?

```
for (i=0; i<N; i++) {
    a[idx[i]] = a[idx[i]] + b[idx[i]];
}
```

```
i=0: a[?_1] = a[?_1] + b[?_1];
i=1: a[?_2] = a[?_2] + b[?_2];
i=3: a[?_3] = a[?_3] + b[?_3];
...
```

Don't know which index is accessed in each iteration of the loop.

It is NOT parallelizable!

Removing dependences 1

- How to remove this dependence?

```
for (i=0; i<=N/2; i++) {
    a[i] = a[i] + a[i+N/2];
}
```

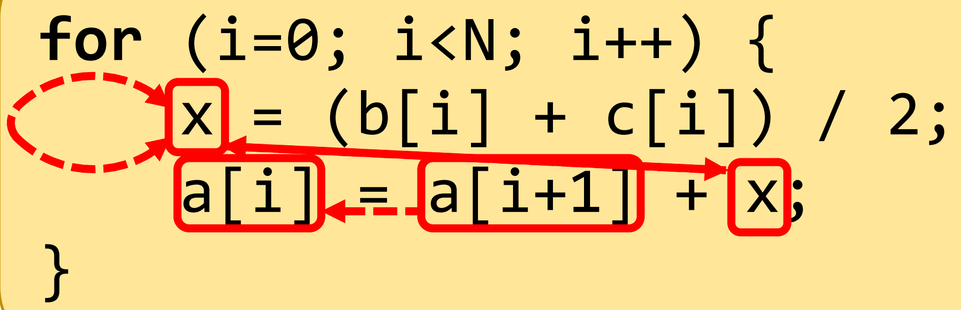
```
for (i=0; i<N/2; i++) {
    a[i] = a[i] + a[i+N/2];
}
a[N/2] = a[N/2] + a[N];
```

Take the dependent iteration out of the loop

Removing dependences 2

```
for (i=0; i<N; i++) {
    x = (b[i] + c[i]) / 2;
    a[i] = a[i+1] + x;
}
```

- How to remove this dependence?



```
for (i=0; i<N; i++) {
    x = (b[i] + c[i]) / 2;
    a[i] = a[i+1] + x;
}
```

True dependence inside the loop (x)

Output dependence between iterations (x)

Anti-dependence between iterations (x)

Anti-dependence between iterations (a[i])

- To remove the dependences on ‘x’ privatize it

Removing dependences 2

```
for (i=0; i<N; i++) {
    x = (b[i] + c[i]) / 2;
    a[i] = a[i+1] + x;
}
```

- How to remove this dependence?

```
for (i=0; i<N; i++) {
    int x = (b[i] + c[i]) / 2;
    a[i] = a[i+1] + x;
}
```

Anti-dependence between iterations (a[i])

- To remove the dependence on ‘a[i]’ make copy of ‘a’

Removing dependences 2

```
for (i=0; i<N; i++) {
    x = (b[i] + c[i]) / 2;
    a[i] = a[i+1] + x;
}
```

- How to remove this dependence?

```
for (i=0; i<N; i++) {
    a2[i] = a[i+1];
}
for (i=0; i<N; i++) {
    int x = (b[i] + c[i]) / 2;
    a[i] = a2[i] + x;
}
```

Anti-dependence between iterations (a[i])

- Both ‘for’ are parallelizable!! *Should we do it?*

Removing dependences 3

- How to remove this dependence?

```
for (i=1; i<N; i++) {
    b[i] += a[i-1];
    a[i] += c[i];
}
```

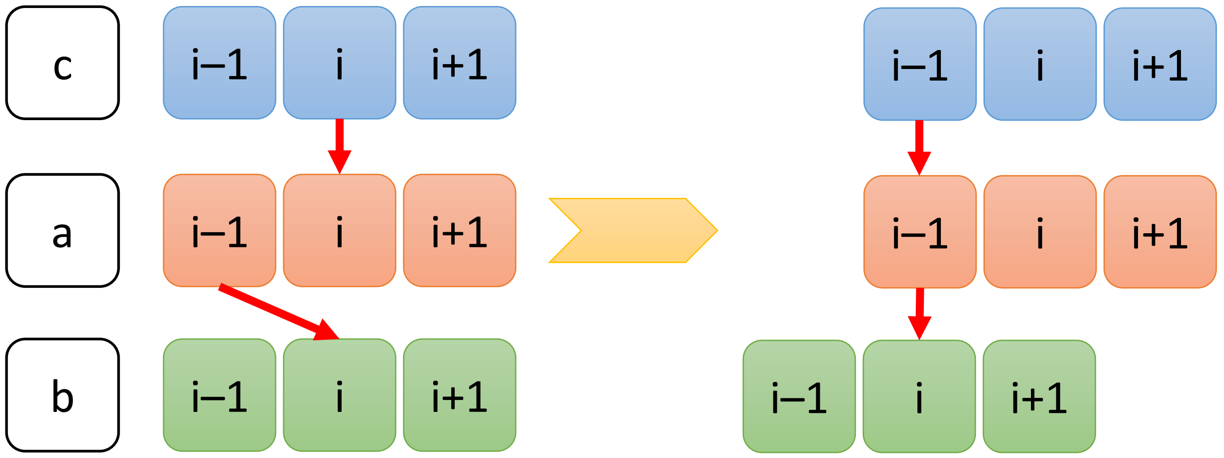
```
i=1: b[1]=b[1]+a[0]; a[1]=a[1]+c[1]
i=2: b[2]=b[2]+a[1]; a[2]=a[2]+c[2]
...
i=N-1 : b[N-1]=b[N-1]+a[N-2]; a[N-1]=a[N-1]+c[N-1]
```

Removing dependences 3

- How to remove this dependence?

```
for (i=1; i<N; i++) {
    b[i] += a[i-1];
    a[i] += c[i];
}
```

Use *software pipelining!*



Removing dependences 3

- How to remove this dependence?

```
for (i=1; i<N; i++) {
    b[i] += a[i-1];
    a[i] += c[i];
}
```

```
b[1] += a[0];
for (i=1; i<N-1; i++) {
     a[i] += c[i];
     b[i+1] += a[i];
}
a[N] += c[N];
```

Removing dependences 4

Not all loops can be made parallel!

The END