Concurrent Systems

3D4 ⟷ CS2016

Operating Systems

Andrew Butterfield
ORI.G39, Andrew.Butterfield@scss.tcd.ie

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

with thanks to Mike Brady

‘Shells’

- A ‘shell’ is a program that allows you to give commands to a computer and get responses.
  - Common GUI-based shells include ‘Finder’ for Mac OS X and ‘Explorer’ for Windows. These are easy to learn but hard to automate.
  - Common UNIX shells are BASH, CSH, TCSH, ASH. These are command-line text-based programs that are a bit more difficult to learn but very powerful indeed. We’ll use BASH, a very-widely used shell in Linux, the BSDs and Mac OS X.
  - The Shell Scripting Primer — search “shell scripting primer pdf”

Important shell principles

- Shell commands generally invoke programs to so the work:
  - e.g. when you write `ls` on the command line, the shell looks for a program called `ls` in a few designated places. Once it finds the program, it executes it.
- Most programs work with standard character-based input sources and output sinks — 'standard input', 'standard output' and 'standard error'. These are 'pipes' and can be connected to files, the terminal window, other programs, network feeds, etc.
- Most of these programs do something simple but very well — to do complex things, you string programs together.

Combining commands

- Multiple commands:
  - Issue a sequence of commands on one line by separating them with a semicolon (“ **;** ”).
- Piping:
  - Using the bar symbol (“ | ”), you can direct (“pipe”) the standard output of one command into the standard input of the next one.
- Automation:
  - You can write a text file containing sequence of commands and shell commands and constructs. This can be executed as a *shell script*.

Communication

- A parallel program consists of two or more separate threads of execution, that run independently except when they have to interact
- To interact, they must *communicate*
- Communication is typically implemented by
  - sharing memory
    - One thread writes data to memory; the other reads it
  - passing messages
    - One thread sends a message; the other gets it

Dependency

- Independent sections of code can run independently.
- We can analyse code for dependencies.
  - To preserve the meaning of the program;
  - To transform the program to reduce dependencies and improve parallelism.

1 – Flow Dependence

S1: write to location L

S2: read from location L

- Flow Dependence: S2 is flow dependent on S1 because S2 reads a location S1 writes to.
  - It must be written to (S1) before it's read from (S2)
  - Also known as True Dependence, and/or Read-After-Write (RAW)

2 – Antidependence

S1: read from location L

↓

S2: write to location L

- Antidependence: S2 is antidependent on S1 because S2 writes to a location S1 reads from.
  - It must be read from (S1) before it can be written to (S2)
  - Also known as Write-After-Read (WAR)

3 – Output Dependence

S1: write to location L

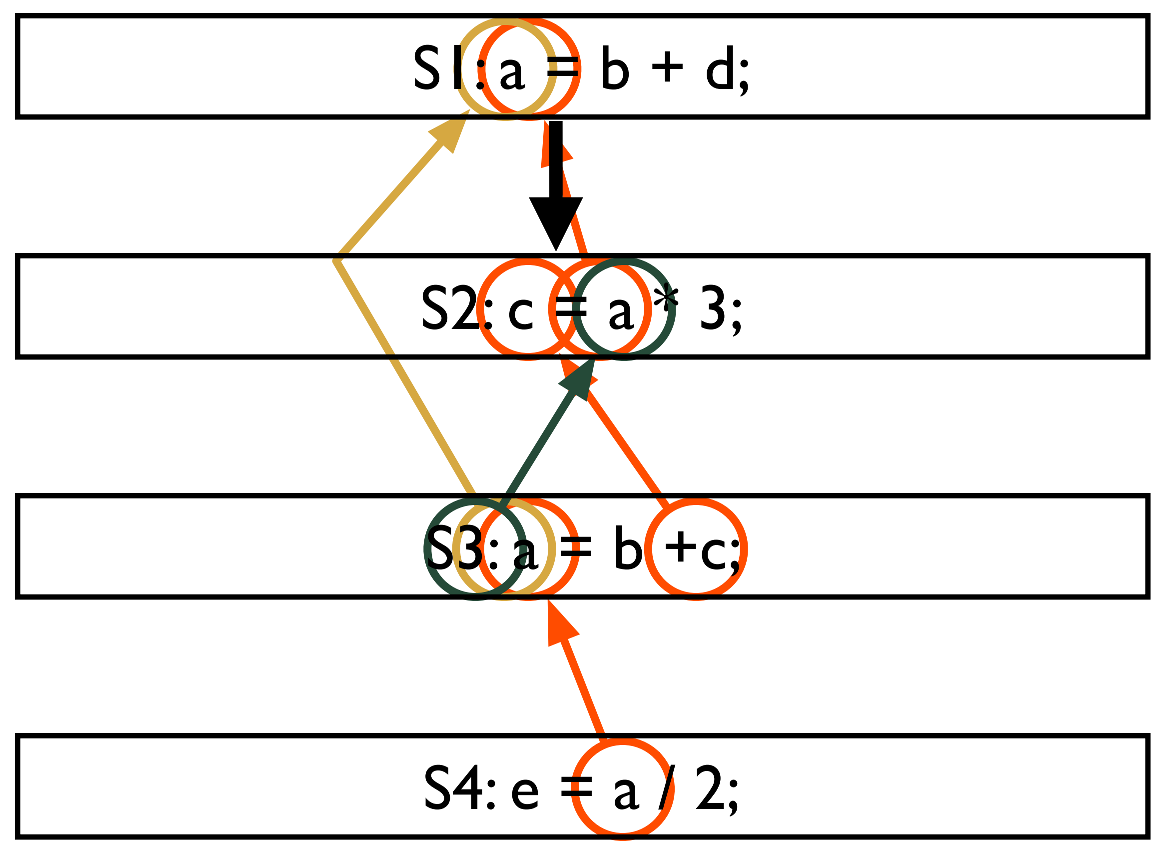
↓

S2: write to location L

- Output dependence: S2 is output dependent on S1 because S2 writes to a location S1 writes to.
  - The value of L is affected by the order of S1 and S2.
  - Also known as Write-After-Write (WAW)

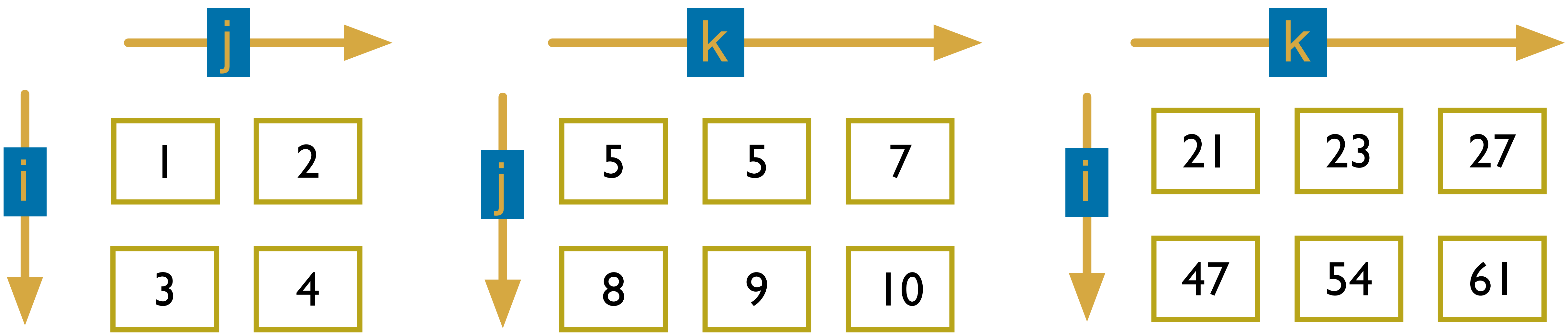
Example

Flow
Anti
Output



Removing Unneeded Dependencies in Loops

- Matrix Multiply. We know, intuitively, we can parallelise this a great deal:
  - each element is c is independent



a b c

Sample Serial Solution

```
for i = 1  to 2 {
  for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[i,j] * b[j,k];
    c[i,k] = sum;
  }
}
```

How can we transform it?

- We can work out how to transform it by locating the dependencies in it.
- Some dependencies are intrinsic to the solution, but
- Some are *artefacts* of the *way* we are *solving* the problem;
    - If we can identify them, perhaps we can modify or remove them.

Try three execution agents:

```
with k = 1;
for i = 1  to 2 {
 for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[1,j] * b[j,k];
    c[i,k] = sum;
 }
}
```

```
with k = 2;
for i = 1  to 2 {
 for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[1,j] * b[j,k];
    c[i,k] = sum;
 }
}
```

```
with k = 3;
for i = 1  to 2 {
 for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[1,j] * b[j,k];
    c[i,k] = sum;
 }
}
```

Issues:

- The variable `sum`, as written, is common to all three programs.
- Solution:
  - Make `sum` private to each program to avoid this dependency.

Try Six Execution Agents

```
with k = 1, i=1;
for i = 1  to 2 {
 for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[1,j] * b[j,k];
    c[i,k] = sum;
 }
}
```

```
with k = 2, i=1;
for i = 1  to 2 {
 for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[1,j] * b[j,k];
    c[i,k] = sum;
 }
}
```

```
with k = 3, i=1;
for i = 1  to 2 {
 for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[1,j] * b[j,k];
    c[i,k] = sum;
 }
}
```

```
with k = 1, i=2;
for i = 1  to 2 {
 for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[1,j] * b[j,k];
    c[i,k] = sum;
 }
}
```

```
with k = 2, i=2;
for i = 1  to 2 {
 for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[1,j] * b[j,k];
    c[i,k] = sum;
 }
}
```

```
with k = 3, i=2;
for i = 1  to 2 {
 for k = 1 to 3 {
    sum = 0.0;
    for j = 1 to 2
      sum = sum + a[1,j] * b[j,k];
    c[i,k] = sum;
 }
}
```

Summarising:

- We could parallelise the original algorithm with some care:
  - Private Variables to avoid unnecessary dependencies
- Actually, we could break this ‘Embarrassingly Parallel’ problem into tiny separate pieces; maybe too small. *(How will we know?)*