departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Parallel Programming Models and Dependencies

Concurrency and Parallelism — 2016-17

Master in Computer Science

(Mestrado Integrado em Eng. Informática)

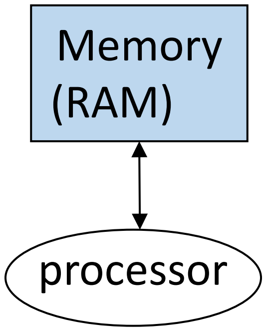
Joao Lourenço <joao.lourenco@fct.unl.pt>

Source: Parallel Computing, CIS 410/510, Department of Computer and Information Science

Outline

- Parallel programming models
- Dependencies

Parallel Models

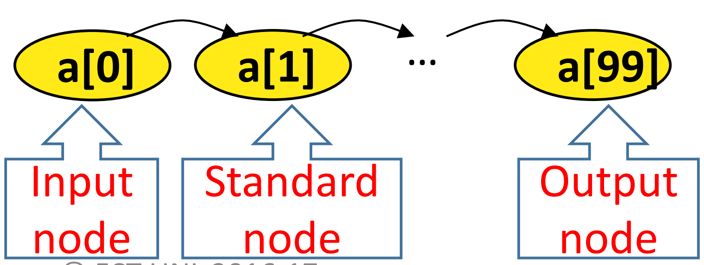


- Sequential models
  - von Neumann (RAM) model
- Parallel model
  - A parallel computer is simple a collection of *processors interconnected* in some manner to *coordinate* activities and *exchange data*
  - Models that can be used as general frameworks for describing and analyzing parallel algorithms
    - *Simplicity*: description, analysis, architecture independency
    - *Implementability*: able to be realized, reflect performance
- Three common parallel models
  - Directed acyclic graphs, shared-memory, network

Directed Acyclic Graphs (DAG)

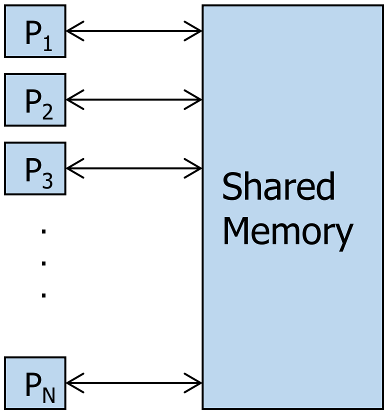
- Captures data flow parallelism
- Nodes represent operations to be performed
  - Inputs are nodes with no incoming arcs
  - Output are nodes with no outgoing arcs
  - Think of nodes as tasks
- Arcs are paths for flow of data results
- DAG represents the operations of the algorithm and implies precedent constraints on their order

```
for (i=1; i<100; i++)
     a[i] = a[i-1] + 100;
```

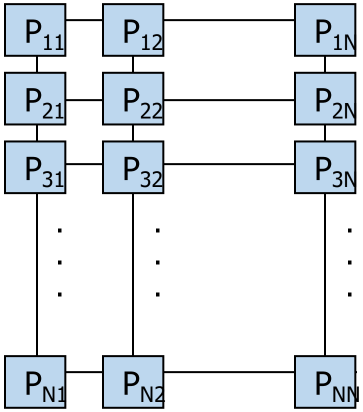


Shared Memory Model

- Parallel extension of RAM model (PRAM)
  - Memory size is infinite
  - Number of processors in unbounded
  - Processors communicate via the memory
  - Every processor accesses any memory location in the same number of cycles
  - Synchronous
    - All processors execute same algorithm synchronously
      - READ phase
      - COMPUTE phase
      - WRITE phase
    - Some subset of the processors can stay idle
  - Asynchronous



Network Model



- G = (N,E)
  - N are processing nodes
  - E are bidirectional communication links
- Each processor has its own memory
- No shared memory is available
- Network operation may be synchronous or asynchronous
- Requires communication primitives
  - Send (X, i)
  - Receive (Y, j)
- Captures message passing model for algorithm design

Parallelism

- Ability to execute different parts of a computation concurrently on different computing elements
- Why do you want parallelism?
  - Shorter running time or handling more work
- What is being parallelized?
  - *Task*: instruction, statement, procedure, …
  - *Data*: data flow, size, replication
  - Parallelism granularity
    - Coarse-grain versus fine-grained
- Evaluation
  - Was the parallelization successful?

Why is parallel programming important?

- Parallel programming has matured
  - Standard programming models
  - Common machine architectures
  - Programmer can focus on computation and use suitable programming model for implementation
- Increase portability between models and architectures
- Reasonable hope of portability across platforms
- Problem
  - Performance optimization is still platform-dependent
  - Performance portability is a problem
  - Parallel programming methods are still evolving

Parallel Algorithm

- Recipe to solve a problem "in parallel" on multiple processing elements
- Standard steps for constructing a parallel algorithm
  - Identify work that can be performed concurrently
  - Partition the concurrent work on separate processors
  - Properly manage input, output, and intermediate data
  - Coordinate data accesses and work to satisfy dependencies
- Which steps are hard to do?

Parallelism Views

- Where can we find parallelism?
- Program (task) view
  - Statement level
    - Between program statements
    - Which statements can be executed at the same time?
  - Block level / Loop level / Routine level / Process level
    - Larger-grained program statements
- Data view
  - How is data operated on?
  - Where does data reside?
- Resource view
  - When to access and use a shared resource?

Parallelism, Correctness, and Dependencies

- 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 dependencies
- A *dependency* arises when one operation depends on an earlier operation to complete and produce a result before this later operation can be performed
- We extend this notion of dependency to resources since some operations may depend on certain resources
  - For example, due to where data is located

Executing Two Statements in Parallel

- Want to execute two statements in parallel
- On one 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

- Case 1:

Processor 1:	Processor 2:	time
statement 1;		↓
	statement 2;	

- Case 2:

Processor 1:	Processor 2:	time
	statement 2;	↓
statement 1;		

- Sequential consistency
  - Statements execution does not interfere with each other
  - Computation results are the same (independent of order)

Independent versus Dependent

- In other words the execution of

```
statement1;
statement2;
```

  must be equivalent to

```
statement2;
statement1;
```

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

Examples

- Example 1

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

- Statements are independent

- Example 2

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

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

- Example 3

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

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

- Example 4

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

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

True Dependency and Anti-Dependency

- Given statements S1 and S2,

  S1;

  S2;

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

  if and only if **S2 reads a value written by S1**

  (RAW – Read After Write)

  X =
  ⋮
  = X
  δ

- S2 has a *anti-dependency* on S1

  if and only if **S2 writes a value read by S1**

  (WAR – Write After Read)

  = X
  ⋮
  X =
  δ^{-1}

Output Dependency

- Given statements S1 and S2,

  S1;

  S2;

- S2 has an *output dependency* on S1

  if and only if **S2 writes a variable written by S1**

  (WAW – Write After Write)

X =

⋮

X =

δ^0

- Anti- and output dependencies are "name" dependencies
  - Are they "true" dependencies?
- How can you get rid of output dependencies?

Statement Dependency Graphs

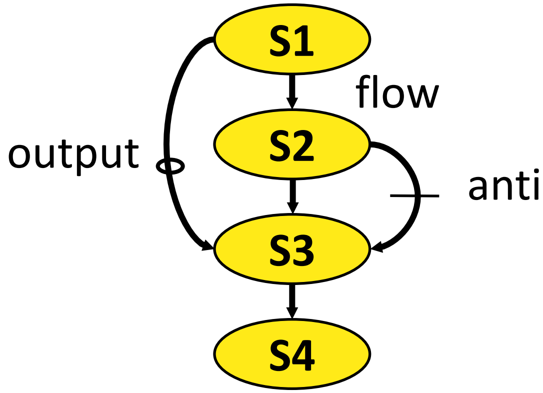
- Can use graphs to show dependency relationships
- Example

  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 dependencies* between them, i.e., no
  - *True dependencies*; nor
  - *Anti-dependencies*; nor
  - *Output dependencies*.
- Some dependencies can be removed by modifying the program
  - Rearranging statements
  - Eliminating statements

How do you compute dependencies?

- Data dependency 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 in S
  - OUT(S) : set of memory locations (variables) that may be modified 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 Dependencies

- Assuming that there is a path from S1 to S2 , the following shows how to intersect the IN and OUT sets to test for data dependency

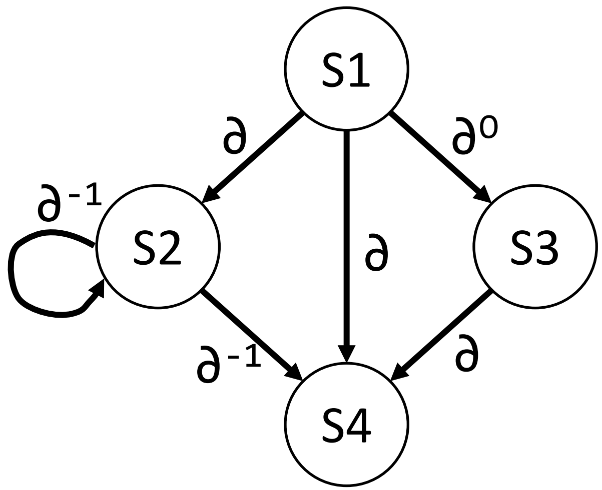
$$out(S_1) \cap in(S_2) \neq \emptyset \qquad S_1\ \delta\ \ S_2 \quad \text{flow dependence}$$

$$in(S_1) \cap out(S_2) \neq \emptyset \qquad S_1\ \delta^{-1}\ S_2 \quad \text{anti - dependence}$$

$$out(S_1) \cap out(S_2) \neq \emptyset \qquad S_1\ \delta^{0} S_2 \quad \text{output dependence}$$

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];
}
```

?

- Dependencies? 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?

Loop-Level Parallelism

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

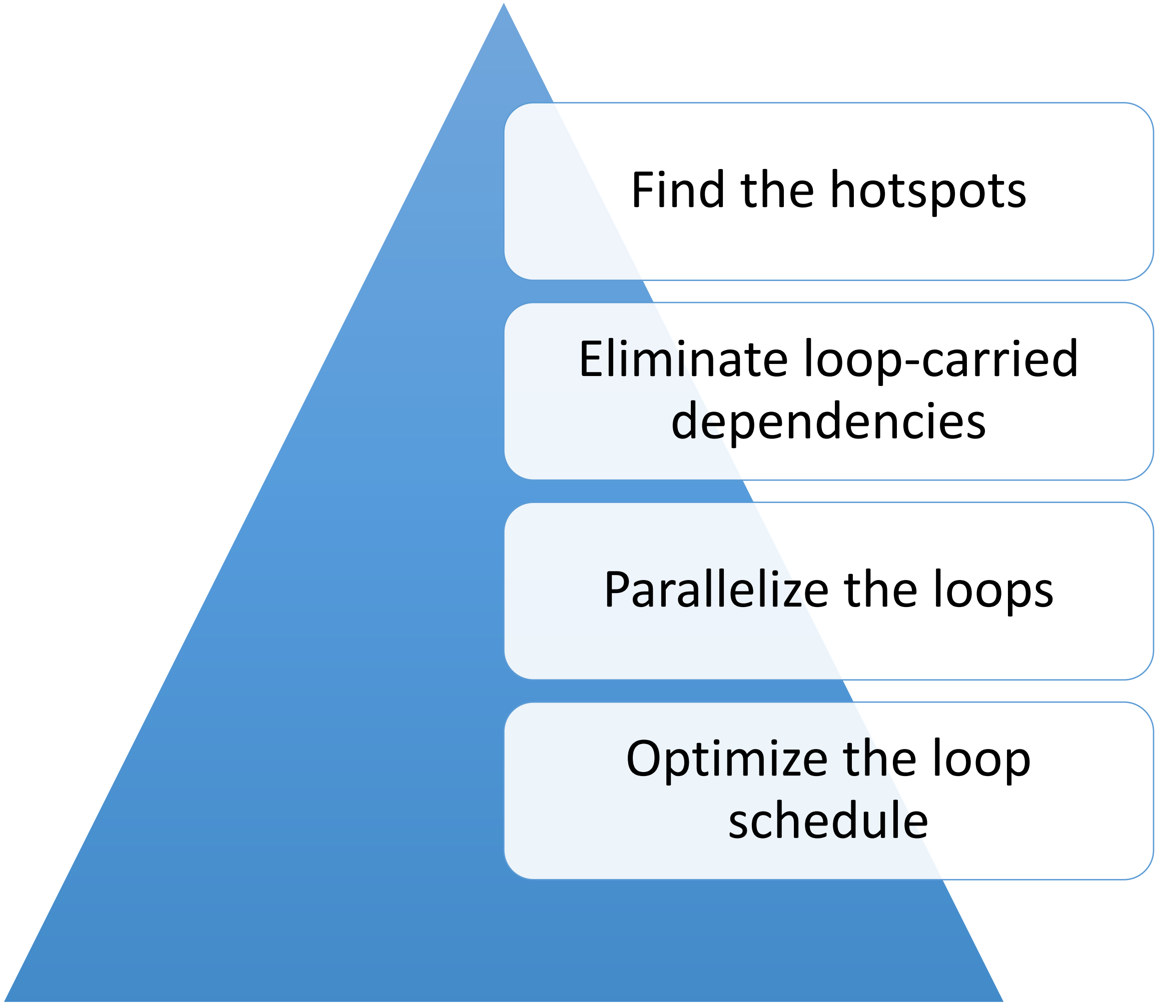
```
for (i=0; i<100; i++)
     S1: a[i] = i;
```

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

?

- Dependencies? What about *i*, the loop index?

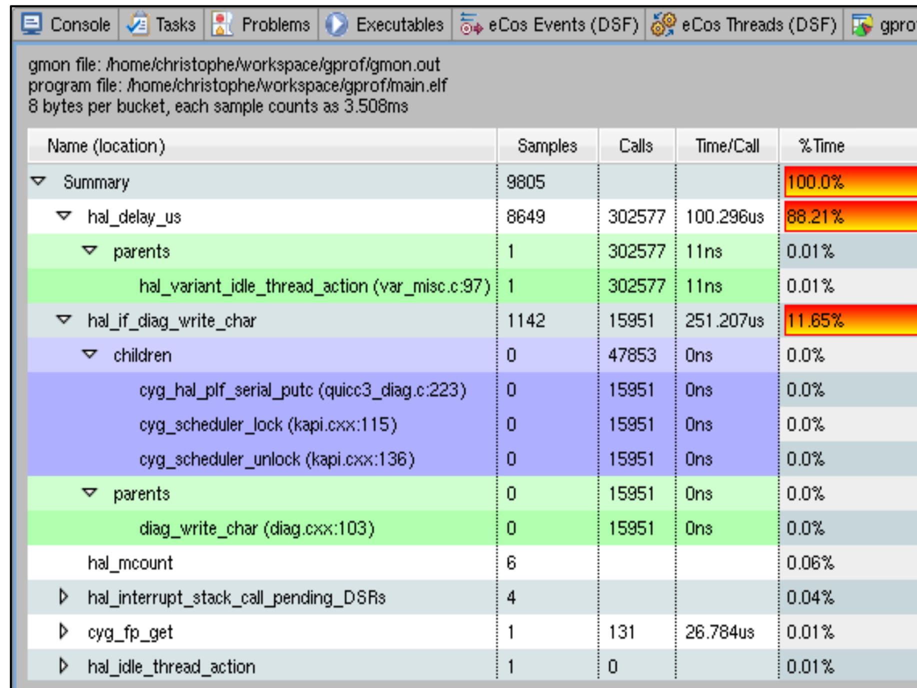
General Approach for Loop Parallelism



Find the hotspots

- By code inspection

- By using performance analysis tools



gmon file: /home/christophe/workspace/gprof/gmon.out
program file: /home/christophe/workspace/gprof/main.elf
8 bytes per bucket, each sample counts as 3.508ms

Name (location)	Samples	Calls	Time/Call	% Time
Summary	9805			100.0%
hal_delay_us	8649	302577	100.296us	88.21%
parents	1	302577	11ns	0.01%
hal_variant_idle_thread_action (var_misc.c:97)	1	302577	11ns	0.01%
hal_if_diag_write_char	1142	15951	251.207us	11.65%
children	0	47853	0ns	0.0%
cyg_hal_plf_serial_putc (quicc3_diag.c:223)	0	15951	0ns	0.0%
cyg_scheduler_lock (kapi.cxx:115)	0	15951	0ns	0.0%
cyg_scheduler_unlock (kapi.cxx:136)	0	15951	0ns	0.0%
parents	0	15951	0ns	0.0%
diag_write_char (diag.cxx:103)	0	15951	0ns	0.0%
hal_mcount	6			0.06%
hal_interrupt_stack_call_pending_DSRs	4			0.04%
cyg_fp_get	1	131	26.784us	0.01%
hal_idle_thread_action	1	0		0.01%

Eliminate loop-carried dependencies

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

Loop Dependencies

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

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

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

S1 ∂ S2

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

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

S1[j] ∂ S1[j+1]

Loop Dependencies

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

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

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

S1 ∂^{-1} S2

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

Anti-dependency — 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 Dependencies

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

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

Output dependency — 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 dependency — 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 Dependencies

- A *loop-carried* dependency is a dependency between two statements instances in two different iterations of a loop
- Otherwise, it is *loop-independent*
- Loop-carried dependencies can prevent loop iteration parallelization
- The dependency is *lexically forward* if the source comes before the target or *lexically backward* otherwise
  - Unroll the loop to see

Loop dependencies: 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 dependencies

- Analyze how each variable is used within a loop iteration:
- Is the variable read and never written?
  => no dependencies!
- For each written variable: can there be any accesses in other iterations than the current?
  => there are dependencies!

Simple rule of thumb

- A loop that matches the following criteria has no dependencies 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];
}
```

```
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 dependencies!

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];
}
```

```
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 dependencies!

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];
}
```

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

Loop carried lexically forward dependency (true dependency)

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 dependencies 1

- How to remove this dependency?

```
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 dependencies 2

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

- How to remove this dependency?

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

True dependency inside the loop (x)

Output dependency between iterations (x)

Anti-dependency between iterations (x)

Anti-dependency between iterations (a[i])

- To remove the dependencies on ‘x’ privatize it

Removing dependencies 2

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

- How to remove this dependency?

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

Anti-dependency between iterations (a[i])

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

Removing dependencies 2

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

- How to remove this dependency?

```
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-dependency between iterations (a[i])

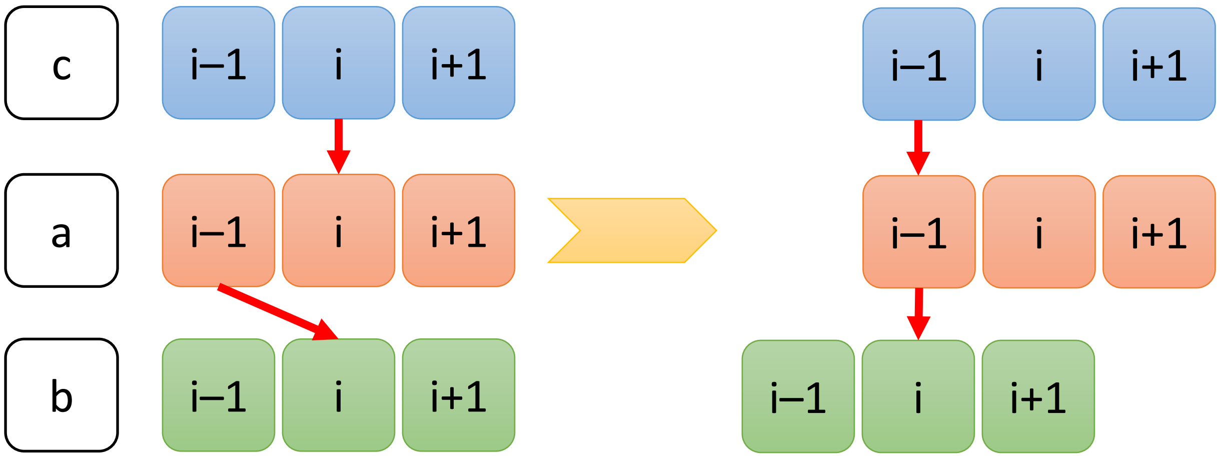
- Both ‘for’ are parallelizable!! *Should we do it?*

Removing dependencies 3

- How to remove this dependency?

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

Use *software pipelining!*



Removing dependencies 3

- How to remove this dependency?

```
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 dependencies 4

The END