NC STATE UNIVERSITY

Course Overview

Lecture 1
(Chapter 1)
http://go.ncsu.edu/ece506

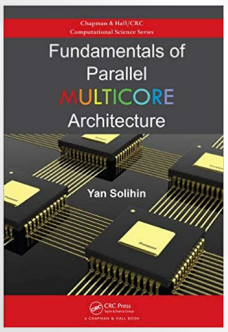
1

Learning Objectives

1. Understand the problem of race conditions in concurrent systems,
2. Learn how to decompose a program for parallel execution,
3. Be able to write simple parallel programs in the important programming models,
4. Understand the operation of common cache-coherence protocols, both bus-based and network-based, and
5. Learn about common memory-consistency models, and appreciate the advantages and disadvantages of each.

2

Textbook



3

TAs

Abhishek Bajaj

Tripti Samal

4

Instructor

5

Where do you find hope, truth, and life?

We encourage you to apply yourself and engage your mind fully in the pursuit of knowledge and academic training in your field of study. College can be one of the best experiences of your life; lasting friendships are developed and your future path is chosen. While here you will no doubt be exposed to various philosophies. Jesus said, *"I am the Way, the Truth, and the Life. No one comes to the Father except through me."* Jesus never forced or coerced anyone to follow him or believe in him. Neither do we. We simply invite you to explore his life and perhaps find new life. We are a group of faculty and staff who are united in our discovery and experience that Jesus Christ provides intellectually and spiritually satisfying answers to life's most important questions. Interested? Have questions? Talk with us or go to EveryStudent.com or MeetTheProf.com.

Contact us at cfsn-ncsu.org or info@cfsn-ncsu.org – Sponsored and paid for by the Christian Faculty Staff Network at NCSU

Everette Gray Allen – IST IT Specialist
Dr. Chris Austin – CSAPC
Dr. Steve H. Barr – Management, Innovation & Entrepreneurship
Valerie Basham – NC State Veterinary Hospital
Carrie Baum-Lane – Applied Ecology
Dr. Mark Beasley – Department of Accounting
Denise Benton – Communications
Dr. Emily Zechman Berglund – Civil, Const. & Environ. Engineering
Dr. Roy Borden – Prof. Emeritus Civil Engineering
Dr. Michael Boyette – Biological & Agricultural Engineering
Dr. Marianne Bradford – Poole College of Management
Kirsten Branch – Department of Computer Science
Dr. Rick L. Brandenburg – Entomology & Plant Pathology
Dr. Joseph Brazel – Department of Accounting
Dr. Steve Broome – Crop & Soil Science
Dr. Andy Hale – Biological & Agricultural Engineering
Dr. Steven Hall – Biological & Agricultural Engineering
Alicia Harris – College of Humanities & Social Science
Felicia Harris – Office of Global Engagement (IEP)
Dr. Gabriel Keith Harris – Food, Bioprocessing, & Nutrition Sciences
Nikki Harris – College of Natural Resources
Dr. Robert B. Hayes – Nuclear Engineering
Dr. Gary Hodge – Forestry & Environmental Resources
Dr. Dennis Hazel – Forestry & Environmental Resources
Gail Hill – GTI
Dr. Daniel Israel – Crop & Soil Sciences
Lee Ivy – Horticultural Science
Dr. Jeffrey Johnson– Biological & Agricultural Engineering
Dr. Chad Jordan – Plant & Microbial Biology
Dr. David L. Jordan – Crop Science
Dr. Jason Painter – The Science House
Dr. Michael Parker – Horticultural Science
Lacy Parrish – Biological & Agricultural Engineering
Dr. Bob Patterson – Crop & Soil Sciences
Jill Phipps – CFI Business Services
Dr. Carrie Pickworth – Animal Sciences
Wayne Pollard – Health & Exercise Studies
Dr. Samuel B. Pond III – Industrial Organizational Psychology
Dr. Daniel H. Poole – Animal Sciences
Terry K. Price – Poole College of Management
Cynthia P. Pullen – Friday Institute
Dr. Bobby Puryear – Department of Economics
Dr. Scott Rugen – The Science House
Dr. Gary Roberson – Biological & Agricultural Engineering
James E. Robinson III – Friday Institute

6

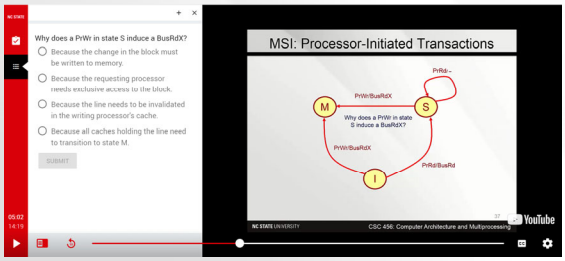
“Attendance” requirement

- You are required to “attend” 20 of the 26 classes.
  - **16** of these must be in the classroom.
  - “Attend” → Respond intelligently to ≥ ½ of Google forms
  - Each one not passed ⇒ –0.5% on semester average.
- You are required to pass 24 of 25 daily quizzes, plus the Syllabus Quiz. First one due Wednesday!
  - “Passed” ⇒ score of ≥ 80%
  - Each one not passed ⇒ –0.5% on semester average.
- You are required to team with 3 students on homework.
  - Each teammate you are lacking ⇒ –0.5% on semester average

7

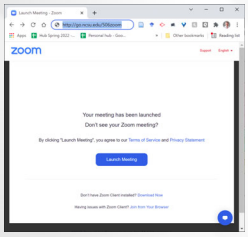
Playposit quizzes

- Three lectures (9, 15, 23) will be videos to watch.
- They have embedded quizzes.
  - Do the quizzes to get attendance credit.



8

Zoom session



http://go.ncsu.edu/506zoom

If you join the Zoom session from the classroom, be sure to let me know.

9

Grading

Homework 50%	4 programs: 24%
	3 problem sets: 18%
	1 peer-reviewed exercise: 8%
Tests 50%	Test 1: 10%
	Test 2: 15%
	Final exam: 25%

10

Homework

- 4 programs
- 3 problem sets*
- 1 peer-reviewed madeup problem

*One of the problem sets will be *dual submission*.

11

Tests

- Two 120-minute midterm tests (10%, 15% of grade)
- 150-minute final (24% of grade)
- Open notes
- No books, computers or communication devices

12

Extra Credit

- All activities for which extra credit is given must *help other students* to learn the course material.
- Examples
  - Making outstanding contributions to answering other students' questions on **Piazza**
  - Contributing useful practice problems via **Peerwise**
  - Doing extra peer reviews of madeup problems submitted to Expertiza
  - Suggesting Web or print resources that will help othe write useful madeup problems

Outline for Lecture 1

- Architectural trends
- Types of parallelism
- Flynn taxonomy
- Scope of CSC/ECE 506

Key Points

- More and more components can be integrated on a single chip
- Speed of integration tracks Moore's law, doubling every 18–24 months.
- *Exercise:* Look up how the number of transistors per chip has changed, esp. since 2006. Submit here.
- Until recently, performance tracked speed of integration
- At the architectural level, two techniques facilitated this:
  - Cache memory
  - Instruction-level parallelism
- Performance gain from uniprocessor system was high enough that multiprocessor systems were not viable for most uses.

Illustration

- 100-processor system with perfect speedup
- Compared to a single processor system
  - Year 1: 100x faster
  - Year 2: 62.5x faster
  - Year 3: 39x faster
  - …
  - Year 10: 0.9x faster
- Single-processor performance catches up in just a few years!
- Even worse
  - It takes longer to develop a multiprocessor system
  - Low volume means prices must be very high
  - High prices delay adoption
  - Perfect speedup is unattainable

How did uniprocessor performance grow so fast?

- ≈ half from circuit improvement (smaller transistors, faster clock, etc.)
- ≈ half from architecture/organization:
- Instruction-level parallelism (ILP)
  - Pipelining: RISC, CISC with RISC back-end
  - Superscalar
  - Out-of-order execution
- Memory hierarchy (caches)
  - Exploit spatial and temporal locality
  - Multiple cache levels

But uniprocessor perf. growth has stalled

- Source of performance growth had been ILP
  - Parallel execution of independent instructions from a single thread
- But ILP improvement has slowed abruptly
  - Memory wall: Processor speed grows at 55%/year, memory speed grows at 7% per year
  - ILP wall: achieving higher ILP requires quadratically increasing complexity (and power)
- Power efficiency
- Thermal packaging limit vs. cost

Types of parallelism

- Instruction level (cf. ECE 563)
  - Pipelining

A (a load)	IF	ID	EX	MEM	WB			
B		IF	ID	EX	MEM	WB		
C			IF	ID	EX	MEM	WB	

19

Types of parallelism, cont.

- Superscalar/VLIW
- Original:

```
LD    F0, 34(R2)
ADDD  F4, F0, F2
LD    F7, 45(R3)
ADDD  F8, F7, F6
```

- Schedule as:

```
LD    F0, 34(R2)   | LD    F7, 45(R3)
ADDD  F4, F0, F2   | ADDD  F8, F0, F6
```

+ Moderate degree of parallelism
– Requires fast communication (register level)

20

Why ILP is slowing

- Number of pipeline stages is already deep (≈ 20–30 stages)
  - But critical dependence loops do not change
  - Memory latency requires more clock cycles to satisfy
- Branch-prediction accuracy is already > 90%
  - Hard to improve it even more
- Cache size
  - Effective, but also shows diminishing returns
  - In general, size must be doubled to reduce miss rate by half.

21

Current trends: multicore and manycore

Aspect	Intel Clovertown	AMD Barcelona	IBM Cell
# cores	4	4	8+1
Clock frequency	2.66 GHz	2.3 GHz	3.2 GHz
Core type	OOO Superscalar	OOO Superscalar	2-issue SIMD
Caches	2x4MB L2	512KB L2 (private), 2MB L3 (sh'd)	256KB local store
Chip power	120 watts	95 watts	100 watts

Exercise: Browse the Web (or the textbook ☺) for information on more recent processors, and for each processor, fill out this form. (You can view the submissions.)

22

Scope of CSC/ECE 506

- ***Parallelism***
  - ***Loop-level and task-level parallelism***
- Flynn taxonomy
  - SIMD (vector architecture)
  - MIMD
    - Shared memory machines (SMP and DSM)
    - Clusters
- Programming Model
  - Shared memory
  - Message-passing
  - Hybrid
  - Data parallel

23

Loop-level parallelism

- Sometimes each iteration can be performed independently.

```
for (i=0; i<8; i++)
  a[i] = b[i] + c[i];
```

- Sometimes iterations cannot be performed independently ⇒ no loop-level parallelism.

```
for (i=0; i<8; i++)
  a[i] = b[i] + a[i-1];
```

+ Very high parallelism > 1K
+ Often easy to achieve load balance
– Some loops are not parallel
– Some apps do not have many loops

24

Task-level parallelism

- Arbitrary code segments in a single program
- Across loops:

```
…
for (i=0; i<n; i++)
  sum = sum + a[i];
for (i=0; i<n; i++)
  prod = prod * a[i];
…
```

- Subroutines:

```
Cost = getCost();
A = computeSum();
B = A + Cost;
```

- Threads: e.g., editor: GUI, printing, parsing

+ Larger granularity ⇒ low overheads, communication
– Low degree of parallelism
– Hard to balance

Program-level parallelism

- Various independent programs execute together
- gmake:
  - `gcc -c code1.c` // assign to proc1
  - `gcc -c code2.c` // assign to proc2
  - `gcc -c main.c` // assign to proc3
  - `gcc main.o code1.o code2.o`

+ No communication
– Hard to balance
– Few opportunities

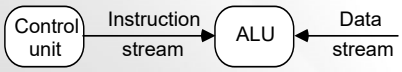
Scope of CSC/ECE 506

- Parallelism
  - Loop-level and task-level parallelism
- *Flynn taxonomy*
  - *SIMD (vector architecture)*
  - *MIMD*
    - *Shared-memory machines (SMP and DSM)*
    - *Clusters*
- Programming Model
  - Shared memory
  - Message-passing
  - Hybrid
  - Data parallel

Taxonomy of parallel computers

The Flynn taxonomy

- *Single* or *multiple instruction* streams.
- *Single* or *multiple data* streams.
- **1. SISD machine**
  - Only one instruction fetch stream
  - Some not-too-ancient laptops or desktops

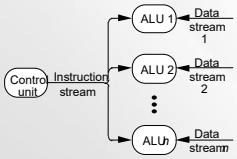


SIMD

- Examples: Vector processors, SIMD extensions (MMX), GPUs
- A single instruction operates on multiple data items.

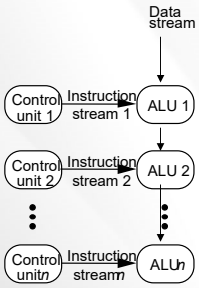
```
SISD:
for (i=0; i<8; i++)
  a[i] = b[i] + c[i];
```

```
SIMD:
a = b + c;   // vector addition
```



MISD

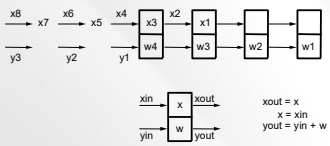
- Example: CMU Warp
- Systolic arrays



Systolic arrays (contd.)

Example: Systolic array for 1-D convolution

y(i) = w1 ´ x(i) + w2 ´ x(i + 1) + w3 ´ x(i + 2) + w4 ´ x(i + 3)

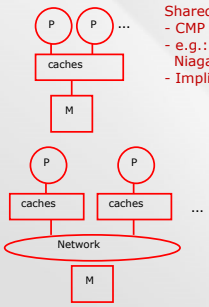


- Practical realizations (e.g. iWARP) use quite general processors
  - Enable variety of algorithms on same hardware
- But dedicated interconnect channels
  - Data transfer directly from register to register across channel
- Specialized, and same problems as SIMD
  - General-purpose systems work well for same algorithms (locality etc.)

MIMD

- Independent processors connected together to form a *multiprocessor* system.
- Physical organization
  - Determines which memory hierarchy level is shared
- Programming abstraction
  - Shared Memory:
    - on a chip: Chip Multiprocessor (CMP)
    - Interconnected by a bus: Symmetric multiprocessors (SMP)
    - Point-to-point interconnection: Distributed Shared Memory (DSM)
  - Distributed Memory:
    - Clusters, Grid

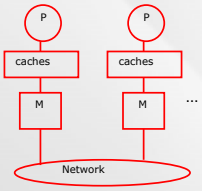
MIMD Physical Organization



Shared-cache architecture:
- CMP (or Simultaneous Multi-Threading)
- e.g.: Pentium 4 chip, IBM Power4 chip, Sun Niagara, Pentium D, etc.
- Implies shared-memory hardware

UMA (Uniform Memory Access)
Shared Memory :
- Pentium Pro Quad, Sun Enterprise, etc.
- What interconnection network?
  - Bus
  - Multistage
  - Crossbar
  - etc.
- Implies shared-memory hardware

MIMD Physical Organization (2)



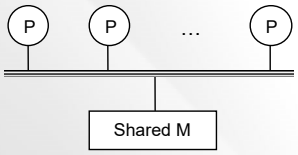
NUMA (Non-Uniform Memory Access)
Shared Memory :
- SGI Origin, Altix, IBM p690, AMD Hammer-based system
- What interconnection network?
  - Crossbar
  - Mesh
  - Hypercube
  - etc.

Scope of CSC/ECE 506

- Parallelism
  - Loop-level and task-level parallelism
- Flynn taxonomy
  - MIMD
    - Shared memory machines (SMP and DSM)
- ***Programming Model***
  - ***Shared memory***
  - ***Message-passing***
  - ***Hybrid***
  - ***Data parallel***

Programming models: shared memory

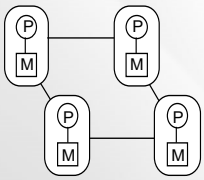
- Shared Memory / Shared Address Space:
  - Each processor can see the entire memory



  - Programming model = thread programming in uniprocessor systems

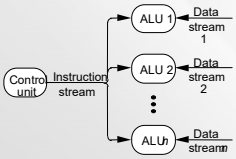
Programming models: message-passing

- Distributed Memory / Message Passing / Multiple Address Space:
  - A processor can directly access only its local memory.
  - All communication happens by explicit messages.



Programming models: data parallel

- Programming model
  - Operations performed in parallel on each element of data structure
  - Logically single thread of control, performs sequential or parallel steps
  - Conceptually, a processor associated with each data element



Data parallel (cont.)

- Architectural model
  - Array of many simple, cheap *processing elements* (PEs) each with little memory
    - Processing elements don't sequence through instructions
  - PEs are attached to a control processor that issues instructions
  - Specialized and general communication, cheap global synchronization
- Original motivation
  - Matches simple differential equation solvers
  - Centralize high cost of instruction fetch/sequencing

Top 500 supercomputers

- http://www.top500.org
- Let's look at the Earth Simulator, #1 in 2004
- Hardware:
  - 5,120 (640 8-way nodes) 500 MHz NEC CPUs
  - 8 GFLOPS per CPU (41 TFLOPS total)
    - 30s TFLOPS sustained performance!
  - 10 TB total memory
- Now (Nov. 2023)
  - Frontier, at Oak Ridge National Laboratory, is #1
  - 8.7 million cores
  - 1194 PFLOP/s max performance (Rmax)
  - 1680 PFLOP/s peak performance (Rpeak)

Exploring the Top 500 list …

- Lists > Top500 > November 2023 > The list
  - See a list of the top systems
- Statistics > List Statistics > Vendors
  - Lenovo is top vendor, more than double HPE
- Statistics > List Statistics > Architecture
  - Clusters are overwhelmingly dominant
- Statistics > Developm't over Time > Countries
  - China comes from nowhere to lead in # of systems
  - But US still leads in performance share

Exercise

- Go to http://www.top500.org and look at the Lists and Statistics menus in the top menu bar.
- From the Statistics dropdown,
  - choose either List Statistics or Development over time,
  - then select one of the statistics, e.g., Vendors, Processor Architecture, and
  - examine what kind of systems are prevalent. Then do the same for earlier lists, and report on the trend.
- You can go all the way back to the first list from 1993.
- Submit your results here.

Three parallel-programming models

- *Shared-memory* programming is like using a “bulletin board” where you can communicate with colleagues.
- *Message-passing* is like communicating via e-mail or telephone calls. There is a well defined event when a message is sent or received.
- *Data-parallel* programming is a “regimented” form of cooperation. Many processors perform an action separately on different sets of data, then exchange information globally before continuing en masse.

User-level communication primitives are provided to realize the programming model

- There is a mapping between language primitives of the programming model and these primitives

These primitives are supported directly by hardware, or via OS, or via user software.

In the early days, the kind of programming model that could be used was closely tied to the architecture.

Today—

- Compilers and software play important roles as bridges
- Technology trends exert a strong influence

The result is convergence in organizational structure, and relatively simple, general-purpose communication primitives.

A shared address space

In the shared-memory model, processes can access the same memory locations.

Communication occurs implicitly as result of loads and stores

This is convenient.

- Wide range of granularities supported.
- Similar programming model to time-sharing on uniprocessors, except that processes run on different processors
- Wide range of scale: few to hundreds of processors

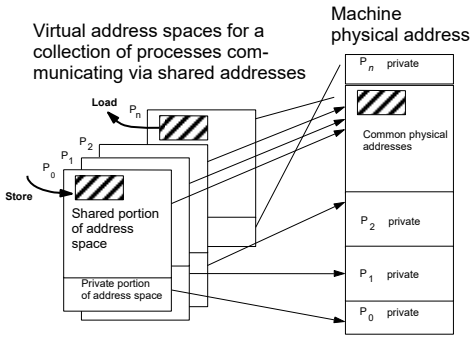
Good throughput on multiprogrammed workloads.

This is popularly known as the *shared memory* model, even though memory may be physically distributed among processors.

The shared-memory model

A process is a virtual address space plus one or more threads of control.

Portions of the address spaces of tasks are shared.



What does the private region of the virtual address space usually contain? Stack and any private data.

Conventional memory operations can be used for communication.

Special atomic operations are used for synchronization.

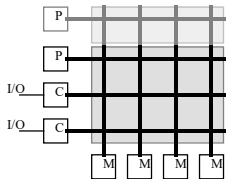
The interconnection structure

The interconnect in a shared-memory multiprocessor can take several forms.

It may be a *crossbar switch*.

Each processor has a direct connection to each memory and I/O controller.

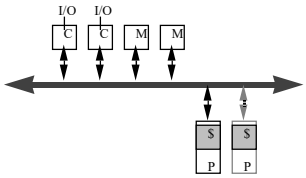
Bandwidth scales with the number of processors.



Unfortunately, cost scales with the square of the # of processors.

This is sometimes called the “mainframe approach.”

At the other end of the spectrum is a *shared-bus* architecture.



All processors, memories, and I/O controllers are connected to the bus. Cost scales linearly with the number of processors.

Such a multiprocessor is called a symmetric multiprocessor (SMP).

What are some advantages and disadvantages of organizing a multiprocessor this way? List them here.

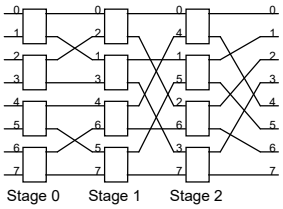
-
-
-

A compromise between these two organizations is a *multistage interconnection network.*

The processors are on one side, and the memories and controllers are on the other.

Each memory reference has to traverse the stages of the network.

Why is this called a compromise between the other two strategies?



Because it allows more parallel transactions than a shared bus, but there’s still a chance of two transactions conflicting.

For small configurations, however, a shared bus is quite viable.

Message passing

In a message-passing architecture, a complete computer, including the I/O, is used as a building block.

Communication is via explicit I/O operations, instead of loads and stores.

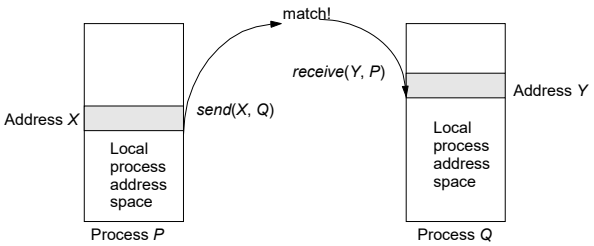
- A program can directly access only its private address space (in local memory).
- It communicates via explicit messages (*send* and *receive*).

It is like a network of workstations (clusters), but more tightly integrated.

Easier to build than a scalable shared-memory machine.

Send-receive primitives

The programming model is further removed from basic hardware operations.



Library or OS intervention is required to do communication.

- *send* specifies a buffer to be transmitted, and the receiving process.
- *receive* specifies sending process, and a storage area to receive into.
- A memory-to-memory copy is performed, from the address space of one process to the address space of the other.
- There are several possible variants, including whether *send* completes—

  when the *receive* has been executed,

  when the send buffer is available for reuse, or

  when the message has been sent.
- Similarly, a *receive* can wait for a matching *send* to execute, or simply fail if one has not occurred.

There are many overheads: copying, buffer management, protection. Let's describe each of these. Submit your descriptions here.

- Why is there an overhead to copying, compared to a share-memory machine?

- Describe the overhead of buffer management.

- What is the overhead for protection?

Here's an example from the textbook of the difference between shared address-space and message-passing programming.

A shared-memory system uses the thread model:

```
int a, b, signal;
…
void dosum(<args>) {
  while (signal == 0) {}; // wait until instructed to work
  printf("child thread> sum is %d", a + b);
  signal = 0;  // my work is done
}

void main() {
  signal = 0;
  thread_create(&dosum);  // spawn child thread
  a = 5, b = 3;
  signal = 1;             // tell child to work
  while (signal == 1) {}  // wait until child done
  printf("all done, exiting\n");
}
```

Message-passing uses the process model:

```
int a, b;
…
void dosum() {
  recvMsg(mainID, &a, &b);
  printf("child process> sum is %d", a + b);
}

void main() {
  if (fork() == 0)  // I am the child process
    dosum();
  else {            // I am the parent process
    a = 5, b = 3;
    sendMsg(childID, a, b);
```



```
    wait(childID);
    printf("all done, exiting\n");
  }
}
```

Here's the relevant section of documentation on the **`fork()`** function: "Upon successful completion, **`fork()`** and **`fork1()`** return 0 to the child process and return the process ID of the child process to the parent process."

Interconnection topologies

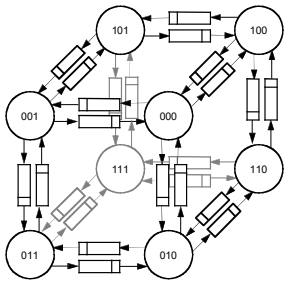
Early message-passing designs provided hardware primitives that were very close to the message-passing model.

Each node was connected to a fixed set of neighbors in a regular pattern by point-to-point links that behaved as FIFOs.

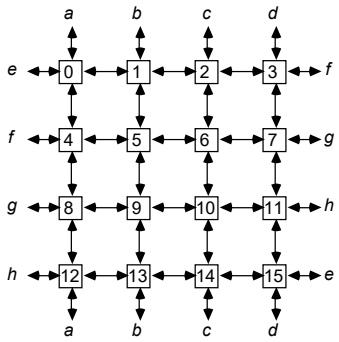
A common design was a *hypercube*, which had $2 \times n$ links per node, where n was the number of dimensions.

The diagram shows a 3D cube.

One problem with hypercubes was that they were difficult to lay out on silicon.



Because of that, 2D meshes eventually supplanted hypercubes.



Here is an example of a 16-node mesh. Note that the last element in one row is connected to the first element in the next.

If the last element in each row were connected to the first element in the same row, we would have a *torus* instead.

Early message-passing machines used a FIFO on each link.

- Thus, software sends were implemented as synchronous hardware operations at each node.

  What was the problem with this, for multi-hop messages? You needed interrupts at all the intermediate processors.

- Synchronous ops were replaced by DMA, enabling non-blocking operations
  - A DMA device is a special-purpose controller that transfers data between memory and an I/O device without processor intervention.
  - Messages were buffered by the message layer of the system at the destination until a *receive* took place.
  - When a *receive* took place, the data was copied to the destination process's address space.

The diminishing role of topology.

- With store-and-forward routing, topology was important.

Parallel algorithms were often changed to conform to the topology of the machine on which they would be run.

- Introduction of pipelined ("wormhole") routing made topology less important.

In current machines, it makes less difference how far the data travels.

This simplifies programming; cost of interprocessor communication is essentially independent of which processor is receiving the data.

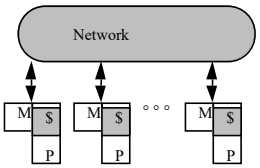
Toward architectural convergence

In 1990, there was a clear distinction between message-passing and shared-memory machines. Today, there isn't a distinct boundary.

- Message-passing operations are supported on most shared-memory machines.
- A shared virtual address space can be constructed on a message-passing machine, by sharing *pages* between processors.
  - When a missing page is accessed, a page fault occurs.
  - The OS fetches the page from the remote node via message-passing.

At the machine-organization level, the designs have converged too.

The block diagrams for shared-memory and message-passing machines look essentially like this:



In shared memory, the network interface is integrated with the memory controller.

It initiates a transaction to access memory at a remote node.

In message-passing, the network interface is essentially an I/O device.

What does Solihin say about the ease of writing shared-memory and message-passing programs on these architectures?

- Which model is easier to program for initially?
- Why doesn't it make much difference in the long run?

The limits of parallelism: Amdahl's law

Speedup is defined as

$$\frac{\text{time for serial execution}}{\text{time for parallel execution}}$$

or, more precisely, as

$$\frac{\text{time for serial execution of best serial algorithm}}{\text{time for parallel execution of our algorithm}}$$

Give two reasons why it is better to define it the second way than the first.

[§4.3.1] If some portions of the problem don't have much concurrency, the speedup on those portions will be low, lowering the average speedup of the whole program.

Exercise: Submit your answers to the questions below.

Suppose that a program is composed of a serial phase and a parallel phase.

- The whole program runs for 1 time unit.
- The serial phase runs for time s, and the parallel phase for time $1-s$.

Then regardless of how many processors N are used, the execution time of the program will be at least ___

and the speedup will be no more than ___. This is known as *Amdahl's law.*

For example, if 25% of the program's execution time is serial, then regardless of how many processors are used, we can achieve a speedup of no more than __.

Efficiency is defined as

$$\frac{\text{speedup}}{\text{number of processors}}$$

Let us normalize computation time so that

- the serial phase takes time 1, and
- the parallel phase takes time p if run on a single processor.

Then if run on a machine with N processors, the parallel phase takes p/N.

Let α be the ratio of serial time to total execution time. Thus

$$\alpha = \frac{1}{1+p/N} = \frac{N}{N+p}.$$

For large N, α approaches __, so efficiency approaches ___.

Does it help to add processors?

Gustafson's law: But this is a pessimistic way of looking at the situation.

In 1988, Gustafson et al. noted that as computers become more powerful, people run larger and larger programs.

Therefore, as N increases, p tends to increase too. Thus, as N increases, α does not get very close to 1, and efficiency remains reasonable.

There may be a maximum to the amount of speedup for a given problem size, but since the problem is "scaled" to match the processing power of the computer, there is no clear maximum to "scaled speedup."

Gustafson's law states that any sufficiently large problem can be efficiently parallelized.

The limits of parallelism: Amdahl's law

Speedup is defined as

$$\frac{\text{time for serial execution}}{\text{time for parallel execution}}$$

or, more precisely, as

$$\frac{\text{time for serial execution of best serial algorithm}}{\text{time for parallel execution of our algorithm}}$$

Give two reasons why it is better to define it the second way than the first.

[§4.3.1] If some portions of the problem don't have much concurrency, the speedup on those portions will be low, lowering the average speedup of the whole program.

Exercise: Submit your answers to the questions below.

Suppose that a program is composed of a serial phase and a parallel phase.

- The whole program runs for 1 time unit.
- The serial phase runs for time s, and the parallel phase for time $1-s$.

Then regardless of how many processors N are used, the execution time of the program will be at least s

and the speedup will be no more than $1/s$. This is known as *Amdahl's law.*

For example, if 25% of the program's execution time is serial, then regardless of how many processors are used, we can achieve a speedup of no more than 4.

Efficiency is defined as

$$\frac{\text{speedup}}{\text{number of processors}}$$

Let us normalize computation time so that

- the serial phase takes time 1, and
- the parallel phase takes time p if run on a single processor.

Then if run on a machine with N processors, the parallel phase takes p/N.

Let α be the ratio of serial time to total execution time. Thus

$$\alpha = \frac{1}{1+p/N} = \frac{N}{N+p}.$$

For large N, α approaches 1, so efficiency approaches 0.

Does it help to add processors? Not much.

Gustafson's law: But this is a pessimistic way of looking at the situation.

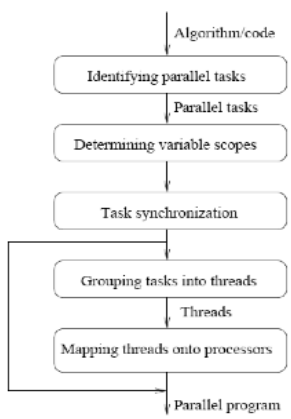
In 1988, Gustafson et al. noted that as computers become more powerful, people run larger and larger programs.

Therefore, as N increases, p tends to increase too. Thus, as N increases, α does not get very close to 1, and efficiency remains reasonable.

There may be a maximum to the amount of speedup for a given problem size, but since the problem is "scaled" to match the processing power of the computer, there is no clear maximum to "scaled speedup."

Gustafson's law states that any sufficiently large problem can be efficiently parallelized.

Shared-Memory Parallel Programming



[§3.1] Solihin identifies several steps in parallel programming.

The first step is identifying parallel tasks. Can you give an example?

The next step is identifying variable scopes. What does this mean?

The next step is grouping tasks into threads. What factors need to be taken into account to do this? Tasks should not have a lot of data dependencies, because that requires a lot of synchronization. They should not be too fine grained, There should be enough to make use of the available processors, and the load should be balanced among the processors.

Then threads must be synchronized. How did we see this done in the three parallel-programming models?

What considerations are important in mapping threads to processors?

Solihin says that there are three levels of parallelism:

- program level
- algorithm level
- code level

Identifying loop-level parallelism

[§3.2] *Goal:* given a code, without knowledge of the algorithm, find parallel tasks.

Focus on loop-dependence analysis.

Notations:

- S is a statement in the source code
- $S[i, j, \ldots]$ denotes a statement in the loop iteration $[i, j, \ldots]$
- "$S1$ then $S2$" means that $S1$ *happens before* $S2$
- If $S1$ then $S2$:

  $S1 \rightarrow_T S2$ denotes true dependence, i.e., $S1$ writes to a location that is read by $S2$

  $S1 \rightarrow_A S2$ denotes anti-dependence, i.e., $S1$ reads a location written by $S2$

  $S1 \rightarrow_O S2$ denotes output dependence, i.e., $S1$ writes to the same location written by $S2$

Example:

S1: $x = 2$;
S2: $y = x$;
S3: $y = x + 4$;
S4: $x = y$;

Exercise: Identify the dependences in the above code.

Loop-independent vs. loop-carried dependences

[§3.2] Loop-carried dependence: dependence exists across iterations; i.e., if the loop is removed, the dependence *no longer exists.*

Loop-independent dependence: dependence exists within an iteration; i.e., if the loop is removed, the dependence still exists.

Example:

```
for (i=1; i<n; i++) {
  S1: a[i] = a[i-1] + 1;
  S2: b[i] = a[i];
}

for (i=1; i<n; i++)
  for (j=1; j< n; j++)
    S3: a[i][j] = a[i][j-1] + 1;

for (i=1; i<n; i++)
  for (j=1; j< n; j++)
    S4: a[i][j] = a[i-1][j] + 1;
```

`S1[i]` →T **`S1[i+1]`**: loop-carried

`S1[i]` →T **`S2[i]`**: loop-independent

`S3[i,j]` →T **`S3[i,j+1]`**:

- loop-carried on **`for`** `j` loop
- no loop-carried dependence in **`for`** `i` loop

`S4[i,j]` →T **`S4[i+1,j]`**:

- no loop-carried dependence in **`for`** `j` loop
- loop-carried on **`for`** `i` loop

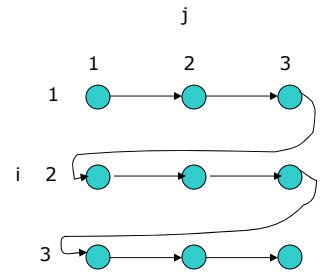
Iteration-space Traversal Graph (ITG)

[§3.2.1] The ITG shows graphically the order of traversal in the iteration space. This is sometimes called the *happens-before relationship*. In an ITG,

- A *node* represents a point in the iteration space
- A *directed edge* indicates the next point that will be encountered after the current point is traversed

Example:

```
for (i=1; i<4; i++)
  for (j=1; j<4; j++)
    S3: a[i][j] = a[i][j-1] + 1;
```

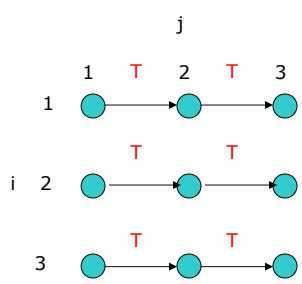



Loop-carried Dependence Graph (LDG)

- LDG shows the true/anti/output dependence relationship graphically.
- A *node* is a point in the iteration space.
- A *directed edge* represents the dependence.

Example:

```
for (i=1; i<4; i++)
  for (j=1; j<4; j++)
    S3: a[i][j] = a[i][j-1] + 1;
```

Another example:

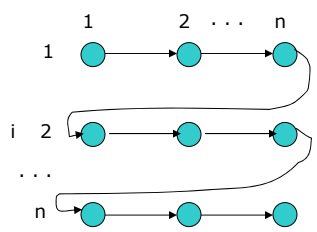
```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    S1: a[i][j] = a[i][j-1] + a[i][j+1] + a[i-1][j] + a[i+1][j];

for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S2: a[i][j] = b[i][j] + c[i][j];
    S3: b[i][j] = a[i][j-1] * d[i][j];
  }
```

- Draw the ITG
- List all the dependence relationships

Note that there are two "loop nests" in the code.

- The first involves S1.
- The other involves S2 and S3.

What do we know about the ITG for these nested loops?



Dependence relationships for Loop Nest 1

- True dependences: Current iteration needs to write before next iteration reads
  - **`S1[i,j]`** →T **`S1[i,j+1]`**
  - **`S1[i,j]`** →T **`S1[i+1,j]`**
- Output dependences:
  - None
- Anti-dependences: Current iteration needs to read before other code overwrites.
  - **`S1[i,j]`** →A **`S1[i+1,j]`**
  - **`S1[i,j]`** →A **`S1[i,j+1]`**

Exercise: Suppose we dropped off the first half of S1, so we had

```
S1: a[i][j] = a[i-1][j] + a[i+1][j];
```

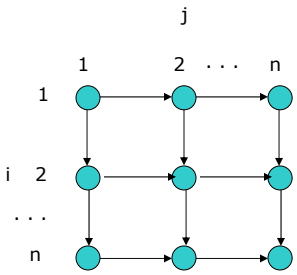
or the last half, so we had

```
S1: a[i][j] = a[i][j-1] + a[i][j+1];
```

Which of the dependences would still exist?

1st half	2nd half
`S1[i,j]` →T `S1[i+1,j]`, `S1[i,j]` →A `S1[i+1,j]`	`S1[i,j]` →T `S1[i,j+1]`, `S1[i,j]` →A `S1[i,j+1]`
These are the dependences on the same row, as you would expect, because iteration is only being done using points in the same row.	These are the dependences on the same column, as you would expect, b/c iteration is only being done using points in the same column.

Draw the LDG for Loop Nest 1.

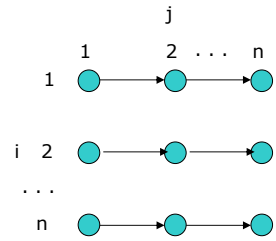


Note: each edge represents both true and anti-dependences

Dependence relationships for Loop Nest 2

- True dependences:
  - `S2[i,j]` →T `S3[i,j+1]`
- Output dependences:
  - None
- Anti-dependences:
  - `S2[i,j]` →A `S3[i,j]` (loop-independent dependence)

Draw the LDG for Loop Nest 2.



Note: each edge represents only true dependences

Why are there no vertical edges in this graph? Answer here.

Why is the anti-dependence not shown on the graph?

Exercise: Consider this code sequence.

```
for (i = 3; i < n; i++) {
     for (j = 0; j < n - 3; j++) {
          S1: A[i][j] = A[i - 3][j] + A[i][j + 3];
          S2: B[i][j] = A[i][j] / 2;
     }
}
```

List the dependences, and say whether they are loop independent or loop carried. Then draw the ITG and LDG (you don't need to submit these).

Finding parallel tasks across iterations

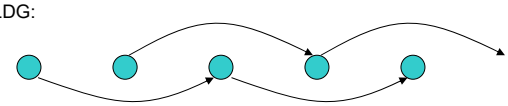
[§3.3.1] Analyze loop-carried dependences:

- Dependences must be enforced (especially true dependences; other dependences can be removed by privatization)
- There are opportunities for parallelism when some dependences are not present.

Example 1

```
for (i=2; i<=n; i++)
  S: a[i] = a[i-2];
```

LDG:



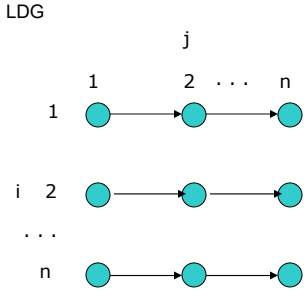
We can divide the loop into two parallel tasks (one with odd iterations and another with even iterations):

```
for (i=2; i<=n; i+=2)
  S: a[i] = a[i-2];
for (i=3; i<=n; i+=2)
  S: a[i] = a[i-2];
```


Example 2

```
for (i=0; i<n; i++)
  for (j=0; j< n; j++)
    S3: a[i][j] = a[i][j-1] + 1;
```

LDG

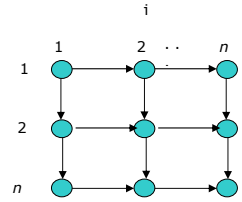


How many parallel tasks are there here? *n*

Example 3

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

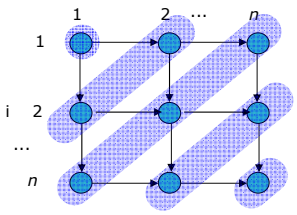
LDG



Note: each edge represents both true, and anti-dependences

Identify which nodes are not dependent on each other

In each anti-diagonal, the nodes are independent of each other



Note: each edge represents both true, and anti-dependences

We need to rewrite the code to iterate over anti-diagonals:

Calculate number of anti-diagonals
for each anti-diagonal do
Calculate the number of points in the current anti-diagonal
for_all points in the current anti-diagonal do
Compute the value of the current point in the matrix

Parallelize the loops highlighted above.

```
for (i=1; i <= 2*n-1; i++) {// 2n-1 anti-diagonals
  if (i <= n) {
    points = i;           // number of points in anti-diag
    row = i;              // first pt (row,col) in anti-diag
    col = 1;              // note that row+col = i+1 always
  }
  else {
    points = 2*n - i;
    row = n;
    col = i-n+1;          // note that row+col = i+1 always
  }
  for_all (k=1; k <= points; k++) {
    a[row][col] = …       // update a[row][col]
    row--; col++;
  }
}
```


DOACROSS Parallelism

[§3.3.2] Suppose we have this code:

Can we execute anything in parallel?

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

Well, we can't run the iterations of the **for** loop in parallel, because …

S[i] →T **S[i+1]** (There is a loop-carried dependence.)

But, notice that the **b[i]*c[i]** part has no loop-carried dependence.

This suggests breaking up the loop into two:

```
for (i=1; i<=N; i++) {
  S1: temp[i] = b[i] * c[i];
}
for (i=1; i<=N; i++) {
  S2: a[i] = a[i-1] + temp[i];
}
```

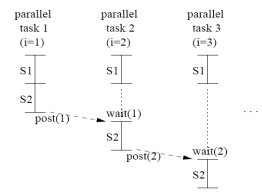
The first loop is ||izable. The second is not.

Execution time: $N\times(T_{S1}+T_{S2})$

What is a disadvantage of this approach? It uses more memory, for the temp array. Execution time = $NT_{S1} + NT_{S2}$

Here's how to solve this problem:

```
post(0);
for_all (i=1; i<=N; i++) {
  S1: temp = b[i] * c[i];
  wait(i-1);
  S2: a[i] = a[i-1] + temp;
  post(i);
}
```



What is the execution time now? $T_{S1} + N\times T_{S2}$

Function parallelism

- [§3.3.3] Identify dependences in a loop body.
- If there are independent statements, can split/distribute the loops.

Example:

```
for (i=0; i<n; i++) {
  S1: a[i] = b[i+1] * a[i-1];
  S2: b[i] = b[i] * coef;
  S3: c[i] = 0.5 * (c[i] + a[i]);
  S4: d[i] = d[i-1] * d[i];
}
```

Loop-carried dependences:

S1[i] →T **S1[i+1]**
S1[i] →A **S2[i+1]**
S4[i] →T **S4[i+1]**

Loop-indep. dependences:
S1[i] →T **S3[i]**

Note that S4 has no dependences with other statements

After loop distribution:

```
for (i=0; i<n; i++) {
  S1: a[i] = b[i+1] * a[i-1];
  S2: b[i] = b[i] * coef;
  S3: c[i] = 0.5 * (c[i] + a[i]);
}

for (i=0; i<n; i++) {
  S4: d[i] = d[i-1] * d[i];
}
```

Each loop is a parallel task.

This is called *function parallelism.*

It can be distinguished from *data parallelism*, which we saw in DOALL and DOACROSS.

Further transformations can be performed (see p. 64 of text).

"**S1[i]** →A **S2[i+1]**" implies that S2 at iteration *i*+1 must be executed after S1 at iteration *i*. Hence, the dependence is not violated if all S2s execute after all S1s.

Characteristics of function parallelism:

- Parallelism is of modest size, does not grow with input.
- Little sync, only at beginning and end.

- Difficult to balance load.

Can use function parallelism along with data parallelism when data parallelism is limited.

DOPIPE Parallelism

[§3.3.4] Another strategy for loop-carried dependences is pipelining the statements in the loop.

Consider this situation:

```
for (i=2; i<=N; i++) {
  S1: a[i] = a[i-1] + b[i];
  S2: c[i] = c[i] + a[i];
}
```

Loop-carried dependences:

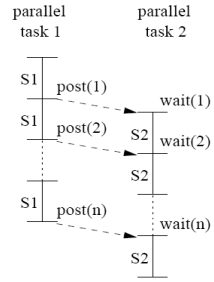
S1[i] →T **S1[i+1]**

Loop-indep. dependences:

S1[i] →T **S2[i]**

To parallelize, we just need to make sure the two statements are executed in sync:

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

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



Question: What's the difference between DOACROSS and DOPIPE?

Determining variable scope

[§3.6] This step is specific to the shared-memory programming model. For each variable, we need to decide how it is used. There are three possibilities:

- Read-only: *variable is only read by multiple tasks*
- R/W non-conflicting: *variable is read, written, or both by only one task*
- R/W conflicting: *variable is written by one task and may be read by another*

Intuitively, why are these cases different? RO … no updates ever occur, so you can copy without having to keep copies up to date. With RWn, you don't have to worry about other tasks. With RWc, you need to synchronize access.

Example 1

Let's assume each iteration of the **for** *i* loop is a parallel task.

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S2: a[i][j] = b[i][j] + c[i][j];
    S3: b[i][j] = a[i][j-1] * d[i][j];
  }
```

Fill in the tableaus here.

Read-only	R/W non-conflicting	R/W conflicting
n, *c*, *d*	*a*, *b*	*i*, *j*

Now, let's assume that each **for** *j* iteration is a separate task.

Read-only	R/W non-conflicting	R/W conflicting
n, *i*, *c*, *d*	*b*	*a*, *j*

Do these two decompositions create the same number of tasks?
No, for I creates *n* tasks; for *j* creates n^2

Example 2

Let's assume that each **for** *j* iteration is a separate task.

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S1: a[i][j] = b[i][j] + c[i][j];
    S2: b[i][j] = a[i-1][j] * d[i][j];
    S3: e[i][j] = a[i][j];
  }
```

Read-only	R/W non-conflicting	R/W conflicting
n, *i*, *c*, *d*	*a*, *b*, *e*	*j*

Exercise: Suppose each **for** *i* iteration were a separate task …

Read-only	R/W non-conflicting	R/W conflicting
n, *c*, *d*	*b*, *e*	*a*, *i*, *j*

To test your knowledge of this approach, try the recent homework problem on the following page:

Problem *k*. *(15 points)* The following code is a commonly used algorithm in image processing applications.

Consider an image *f* with width=ImageWidth and height=ImageHeight. *f* is a 2D grid of pixels. *k* is a kernel of width=2*w*+1 and height=2*h*+1 where (2*w*+1) < ImageWidth and (2*h*+1) < ImageHeight. The image *f* is processed using the kernel *k* to produce a new image *g* as shown:

```
for y = 0 to ImageHeight do
    for x = 0 to ImageWidth do
        sum = 0
        for i= -h to h do
            for j = –w to w do
                sum = sum + k[j,i] * f[x – j, y – i]
            end for
        end for
        g[x y] = sum
    end for
end for
```

(a). Identify the read-only, R/W non-conflicting and R/W conflicting variables, if the **for** *y* loop is parallelized.

Read only	R/W non-conflicting	R/W conflicting

(b). Identify the read-only, R/W non-conflicting and R/W conflicting variables, if (only) the **for** *i* loop is parallelized. Assume that the **for** *i* tasks for the previous value of *x* must complete before the **for** *i* tasks of the current value of *x* are started.

Read only	R/W non-conflicting	R/W conflicting

(c). Identify the read-only, R/W non-conflicting and R/W conflicting variables, if the **for** *i* loop is parallelized. Assume that the **for** *i* tasks for the previous value of *x* do not have to complete before the **for** *i* tasks of the current value of *x* are started.

Read only	R/W non-conflicting	R/W conflicting

Privatization

Privatization means making private copies of a shared variable.

What is the advantage of privatization?
Tasks can run in parallel without paying attention to what other task is accessing the variable.

Of the three kinds of variables in the table above, which kind(s) does it make sense to privatize? R/W conflicting; other variables can simply be accessed where they reside in memor.

Under what conditions is a variable privatizable?

- If it is always defined (=written) in program order by a task before use (=read) by the same task (Case 1).
- If its values in different parallel tasks are known ahead of time, allowing private copies to be initialized to the known values (Case 2).

When a variable is privatized, one private copy is made for each thread (not each task).

Result of privatization: R/W conflicting → R/W non-conflicting

Let's revisit the examples.

Example 1

With each **for** *i* iteration a separate task, which of the R/W conflicting variables are privatizable? *i*, *j*

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S2: a[i][j] = b[i][j] + c[i][j];
    S3: b[i][j] = a[i][j-1] * d[i][j];
  }
```

Which case does each such variable fall into?
i falls into Case 2 (value known ahead of time)

j is Case 1 (always written by a task before being read by the task)

We can think of privatized variables as arrays, indexed by process ID:

Example 2

Parallel tasks: each **for** *j* loop iteration.

Is the R/W conflicting variable *j* privatizable? If so, which case does it represent? Yes, Case 2.

Reduction

Reduction is another way to remove conflicts. It is based on partial sums.

Suppose we have a large matrix, and need to perform some operation on all of the elements—let's say, a sum of products—to produce a single result.

We could have a single processor undertake this, but this is slow and does not make good use of the parallel machine.

So, we divide the matrix into portions, and have one processor work on each portion.

Then after the partial sums are complete, they are combined into a global sum. Thus, the array has been "reduced" to a single element.

Examples:

- addition (+), multiplication (*)
- Logical (**and**, **or**, …)

The *reduction variable* is the scalar variable that is the result of a reduction operation.

Criteria for reducibility:

- Reduction variable is updated by each task, and the order of update is not important.

- Hence, the reduction operation must be associative and commutative.

Goal: Compute

$y = y\_init$ **op** $x1$ **op** $x2$ **op** $x3$ … **op** x_n

op is a reduction operator if it is *commutative*

u **op** $v = v$ **op** u

and *associative*

(u **op** v) **op** $w = u$ **op** (v **op** w)

Summary of scope criteria

Should be declared private	Should be declared shared	Should be declared reduction	Non-privatizable R/W conflicting
privatizable variables	read-only vars. R/W non-conflicting	reduction variables	declare as shared, protect by synch.

Example 1

with **for** *i* parallel tasks

Fill in the answers here.

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S2: a[i][j] = b[i][j] + c[i][j];
    S3: b[i][j] = a[i][j-1] * d[i][j];
  }
```

Read-only	R/W non-conflicting	R/W conflicting
n, c, d	*a, b*	*i, j*

Declare as shared	Declare as private
n, c, d a, b	*i, j*

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S1: a[i][j] = b[i][j] + c[i][j];
    S2: b[i][j] = a[i][j] * d[i][j];
    S3: e[i][j] = a[i][j];
  }
```


Example 2

with **for** *j* parallel tasks

Fill in the answers here.

Read-only	R/W non-conflicting	R/W conflicting
n, i, c, d	*a, b, e*	*j*

Declare as shared	Declare as private
n, i, c, d a, b, e	*j*

Example 3

Consider matrix multiplication.

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++) {
    C[i][j] = 0.0;
    for (k=0; k<n; k++) {
      C[i][j] = C[i][j] + A[i][k]*B[k][j];
    }
  }
}
```

Exercise: Suppose the parallel tasks are **for** *k* iterations. Determine which variables are conflicting, which should be declared as private, and which need to be protected against concurrent access by using a critical section.

Read-only	R/W non-conflicting	R/W conflicting
A, B, i, j, n		*C, k*

Declare as shared	Declare as private
A, B, i, j, n, C	*k*

Which variables, if any, need to be protected by a critical section?

C

Now, suppose the parallel tasks are **for** *i* iterations. Determine which variables are conflicting, which should be declared as private, and

which need to be protected against concurrent access by using a critical section.

Read-only	R/W non-conflicting	R/W conflicting
A, B, n	*C*	*i, j, k*

Declare as shared	Declare as private
A, B, n, C	*i, j, k*

Which variables, if any, need to be protected by a critical section?
None.

Synchronization

Synchronization is how programmers control the sequence of operations that are performed by parallel threads.

Three types of synchronization are in widespread use.

- *Point-to-point:*
  - a pair of *post*() (or *signal*())and *wait*()
  - a pair of *send*() and *recv*() in message passing
- *Lock*
  - a pair of *lock*() and *unlock*()
  - only one thread is allowed to be in a locked region at a given time
  - ensures mutual exclusion
  - used, for example, to serialize accesses to R/W concurrent variables.
- *Barrier*
  - a point past which a thread is allowed to proceed only when all threads have reached that point.

Lock

What problem may arise here?

```
// inside a parallel region
for (i=start_iter; i<end_iter; i++)
  sum = sum + a[i];
```

Two threads may read sum and increment it by a[i] before the other has finished. Then one of the increments will be lost.

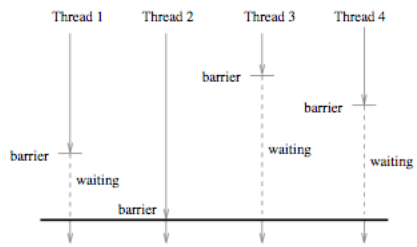
A lock prevents more than one thread from being inside the locked region.

```
// inside a parallel region
for (i=start_iter; i<end_iter; i++) {
    lock(x);
    sum = sum + a[i];
    unlock(x);
}
```

Issues:

- What granularity to lock?
- How to build a lock that is correct and fast.

Barrier: Global event synchronization



A barrier is used when the code that follows requires that all threads have gotten to this point. Example: Simulation that works in terms of timesteps.

Load balance is important.

Execution time is dependent on the slowest thread.

This is one reason for gang scheduling and avoiding time sharing and context switching.

Question 1. (a) *(18 points)* Considert the following algorithm. If we are to parallelize this algorithm for each **for** loop, fill in the table appropriately for each variable used.

```
for (i = 0; i < n; i++){
      for (j = 0; j < n; j++){
            for (k = 0; k < n; k++){
                  if (d[i][j] - d[j][k] < d[i][k])     {
                        b[j][k] = k*j;
                        d[i][j] = d[j][k] + d[i][k];
                  }
            }
      }
}
```

Which loop parallelized? →	**for** *i*	**for** *j*	**for** *k*
Read-only	*n*	*i, n*	
RW non-conflicting		*b*	
RW conflicting	*i, j, k, b, d*	*d, j, k*	
Private	*i, j, k*	*j, k*	
Shared	*b, d, n*	*d, i, n, b*	

(b) *(2 points)* Do any of the shared variables need to be protected by a critical section? Explain.

Simulating ocean currents

We will study a parallel application that simulates ocean currents.

Goal: Simulate the motion of water currents in the ocean. Important to climate modeling.

The overall structure of the program looks like this:

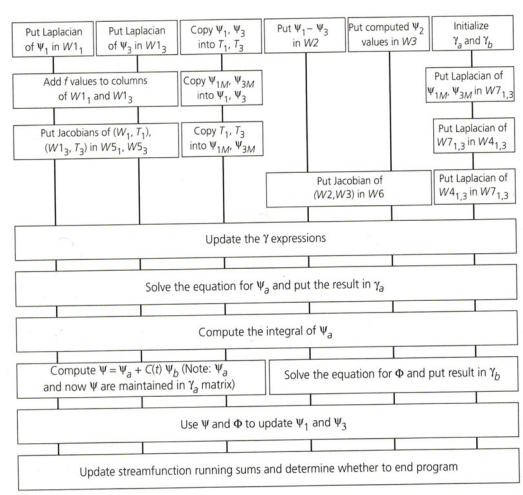
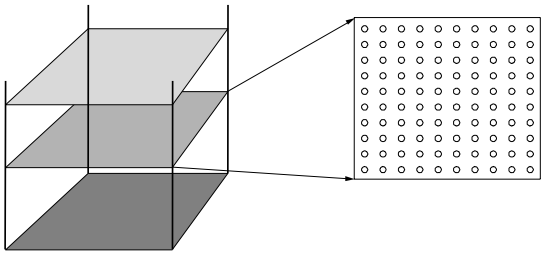


FIGURE 3.14 Ocean: The phases in a time-step and the dependences among grid computations. Each box is a grid computation (or pair of similar computations). Computations connected by vertical lines are dependent while others, such as those in the same row, are independent. The parallel program treats each horizontal row as a phase and synchronizes between phases.

The program offers opportunities for function parallelism (the different blocks in a row) and data parallelism (parallelism within a block).

We will concentrate on solving the equation for ψ_a (data parallelism).

Motion depends on atmospheric forces, friction with ocean floor, and "friction" with ocean walls.



(a) Cross sections

To predict the state of the ocean at any instant, we need to solve complex systems of equations.

The problem is *continuous* in both space and time.
But to solve it, we *discretize* it over both dimensions.

Every important variable, e.g.,

- pressure
- velocity
- currents

has a value at each grid point.

This model uses a set of 2D horizontal cross-sections through the ocean basin.

Equations of motion are solved at all the grid points in one time-step.

- The state of the variables is updated, based on this solution.
- The equations of motion are solved for the next time-step.

Tasks

The first step is to divide the work into *tasks*.

- A task is an arbitrarily defined portion of work.
- It is the smallest unit of concurrency that the program can exploit.

Example: In the ocean simulation, a task can be computations on—

- a single grid point,
- a row of grid points, or
- any arbitrary subset of the grid.

Tasks are chosen to match some natural granularity in the work.

- If the grain is small, the decomposition is called fine grained.
- If it is large, the decomposition is called coarse grained.

Threads

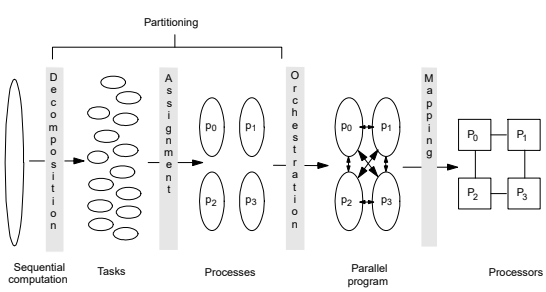
A *thread* is an abstract entity that performs tasks.

- A program is composed of cooperating threads.
- Each thread is assigned to a processor.
- Threads need not correspond 1-to-1 with processors!

Example: In the ocean simulation, an equal number of rows may be assigned to each thread.

Four steps in parallelizing a program:

- *Decomposition* of the computation into tasks.
- *Assignment* of tasks to threads.
- *Orchestration* of the necessary data access, communication, and synchronization among threads.
- *Mapping* of threads to processors.



Together, decomposition and assignment are called *partitioning*.

They break up the computation into tasks to be divided among threads.

The number of tasks available at a time is an upper bound on the achievable parallelism.

Table 2.1 Steps in the Parallelization Process and Their Goals

Step	Architecture-Dependent?	Major Performance Goals
Decomposition	Mostly no	Expose enough concurrency but not too much
Assignment	Mostly no	Balance workload Reduce communication volume
Orchestration	Yes	Reduce noninherent communication via data locality Reduce communication and synchronization cost as seen by the processor Reduce serialization at shared resources Schedule tasks to satisfy dependences early
Mapping	Yes	Put related processes on the same processor if necessary Exploit locality in network topology

Parallelization of an Example Program

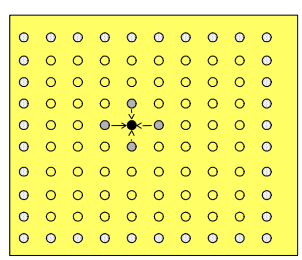
[§2.3] In this lecture, we will consider a parallelization of the kernel of the Ocean application.

The serial program

The equation solver solves a PDE on a grid.

It operates on a regular 2D grid of (n+2) by (n+2) elements.

- The *boundary elements* in the border rows and columns do not change.
- The interior n-by-n points are updated, starting from their initial values.



Expression for updating each interior point:

$$A[i,j] = 0.2 \times (A[i,j] + A[i,j-1] + A[i-1,j] + A[i,j+1] + A[i+1,j])$$

- The old value at each point is replaced by the weighted average of itself and its 4 nearest-neighbor points.
- Updates are done from left to right, top to bottom.
  - The update computation for a point sees the new values of points above and to the left, and
  - the old values of points below and to the right.

  This form of update is called the Gauss-Seidel method.

During each sweep, the solver computes how much each element has changed since the last sweep.

- If the sum of these differences is less than a “tolerance” parameter, the solution has converged.
- If so, we exit solver; if not, we do another sweep.

Here is the code for the solver.

```
1.  int n;                                  /*size of matrix: (n + 2-by-n + 2) elements*/
2.  double **A, diff = 0;

3.  main()
4.  begin
5.     read(n) ;                            /*read input parameter: matrix size*/
6.     A ← malloc (a 2-d array of size n + 2 by n + 2 doubles);
7.     initialize(A);                       /*initialize the matrix A somehow*/
8.     Solve (A);                           /*call the routine to solve equation*/
9.  end main

10. procedure Solve (A)                     /*solve the equation system*/
11.    double **A;                          /*A is an (n + 2)-by-(n + 2) array*/
12. begin
13.    int i, j, done = 0;
14.    float diff = 0, temp;
15.    while (!done) do                     /*outermost loop over sweeps*/
16.       diff = 0;                         /*initialize maximum difference to 0*/
17.       for i ← 1 to n do                 /*sweep over nonborder points of grid*/
18.          for j ← 1 to n do
19.             temp = A[i,j];              /*save old value of element*/
20.             A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                A[i,j+1] + A[i+1,j]); /*compute average*/
22.             diff += abs(A[i,j] - temp);
23.          end for
24.       end for
25.       if (diff/(n*n) < TOL) then done = 1;
26.    end while
27. end procedure
```

Answer these questions about the solver.

Why is the array size (n+2)×(n+2) rather than n×n?

Why is it necessary to use a *temp* variable?

Why is the denominator in Line 25 $n*n$?

Decomposition

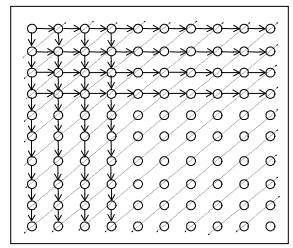
A simple way to identify concurrency is to look at loop iterations.

Is there much concurrency in this example? Does the algorithm let us perform more than one sweep concurrently? No, we can only start the i+1st iteration after we finish the ith.

Note that—

- Computation proceeds from left to right and top to bottom.
- Thus, to compute a point, we use
  - the updated values from the point above and the point to the left, but
  - the “old” values of the point itself and its neighbors below and to the right.

Here is a diagram that illustrates the dependences.



The horizontal and vertical lines with arrows indicate dependences.

The dashed lines along the antidiagonal connect points with no dependences that can be computed in parallel.

Check: If `A[3,4]` is being computed, which updated values are used in the calculation? A[2, 4], A[3, 3]

Which of the following points can be updated in parallel?

Of the $O(n^2)$ work in each sweep, ∃ concurrency proportional to the number of antidiagonals. (Give your answer in terms of n; how many points along an antidiagonal can be computed in parallel?)

How could we exploit this parallelism?

- We can *leave loop structure alone* and let loops run in parallel, inserting *synchronization ops* to make sure a value is computed before it is used.

  Why isn't this a good idea?

- We can *change the loop structure*, making
  - the outer **for** loop (line 17) iterate over anti-diagonals, and
  - the inner **for** loop (line 18) iterate over elements within an antidiagonal.

  Why isn't this a good idea?

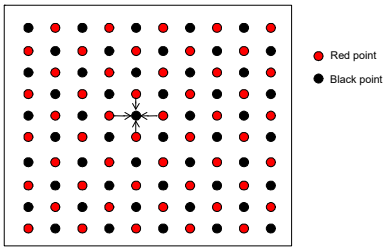
The Gauss-Seidel algorithm doesn't *require* us to update the points from left to right and top to bottom.

It is just a convenient way to program on a uniprocessor.

We can compute the points in another order, as long as we use updated values frequently enough (if we don't, the solution will converge, but more slowly).

Red-black ordering

Let's divide the points into alternating “red” and “black” points:



To compute a red point, we don't need the updated value of any other red point. But we need the updated values of 2 black points.

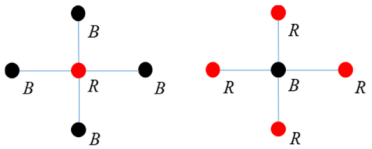
And similarly for computing black points.

Thus, we can divide each sweep into two phases.

- First we compute all red points.
- Then we compute all black points.

True, we don't use any updated black values in computing red points.

But we use *all* updated red values in computing black points.



Whether this converges more slowly or faster than the original ordering depends on the problem.

But it does have important advantages for parallelism.

- Which points can be computed in parallel?
- Altogether, how many red points can be computed in parallel?
- How many black points can be computed in parallel?

Red-black ordering is effective, but it doesn't produce code that can fit on a single display screen.

A simpler decomposition

Another ordering that is simpler but still works reasonably well is just to ignore dependences between grid points within a sweep.

A sweep just updates points based on their nearest neighbors, regardless of whether the neighbors have been updated yet.

Global synchronization is still used between sweeps, however.

Now execution is no longer deterministic. (Does this matter?)

The number of sweeps needed, and the results, may depend on the number of processors used.

But for most reasonable assignments of processors, the number of sweeps will not vary much.

Let's look at the code for this.

```
while (!done) do                    /*a sequential loop*/
  diff = 0;
  for_all i ← 1 to n do             /*a parallel loop nest*/
    for_all j ← 1 to n do
      temp = A[i,j];
      A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
        A[i,j+1] + A[i+1,j]);
      diff += abs(A[i,j] - temp);
    end for_all
  end for_all
  if (diff/(n*n) < TOL) then done = 1;
end while
```

The *only* difference is that **for** has been replaced by **for_all**.

A **for_all** just tells the system that all iterations can be executed in parallel.

With **for_all** in both loops, all n^2 iterations of the nested loop can be executed in parallel.

We could write the program so that the computation of one row of grid points must be assigned to a single processor. How would we do this? Make the outer loop **for_all**, but the inner loop would change back to **for**.

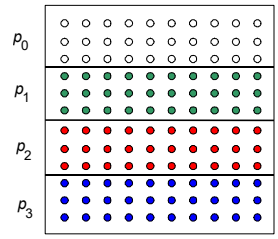
With each row assigned to a different processor, each task has to access about $2n$ grid points that were computed by other processors; meanwhile, it computes n grid points itself.

So the communication-to-computation ratio is $O(1)$.

Assignment

How can we statically assign elements to processes?

- One option is "block assignment"—Row i is assigned to process $\lfloor i/p \rfloor$.



- Another option is "cyclic assignment—Process i is assigned rows i, $i+p$, $i+2p$, etc.
- Another option is 2D contiguous block partitioning.

We could instead use dynamic assignment, where a process gets an index, works on the row, then gets a new index, etc. Is there any advantage to this?

What are advantages and disadvantages of these partitionings?

Static assignment of rows to processes reduces concurrency

But block assignment reduces communication, by assigning adjacent rows to the same processor.

How many rows now need to be accessed from other processors?

So the communication-to-computation ratio is now only O(_____).

Orchestration

Once we move on to the orchestration phase, the computation model constrains our decisions.

Data-parallel model

In the code below, we assume that global declarations are used for shared data, and that any data declared within a procedure is private.

Global data is allocated with *g_malloc*.

Differences from sequential program:

- **for_all** loops
- **decomp** statement
- *mydiff* variable, private to each process
- **reduce** statement

```
1.  int n, nprocs;                  /*grid size (n+2×n+2) and # of processes*/
2.  double **A, diff = 0;

3.  main()
4.  begin
5.    read(n); read(nprocs);        /*read input grid size and # of processes*/
6.    A ← G_MALLOC (a 2-d array of size n+2 by n+2 doubles);
7.    initialize(A);                /*initialize the matrix A somehow*/
8.    Solve (A);                    /*call the routine to solve equation*/
9.  end main

10. procedure Solve(A)              /*solve the equation system*/
11.     double **A;                 /* A is an (n+2×n+2) array*/
12.   begin
13.   int i, j, done = 0;
14.   float mydiff = 0, temp;
14a.    DECOMP A[BLOCK,*, nprocs];
15.   while (!done) do              /*outermost loop over sweeps*/
16.     mydiff = 0;                 /*initialize maximum difference to 0 */
17.     for_all i ← 1 to n do       /*sweep over non-border points of grid*/
18.       for_all j ← 1 to n do
19.         temp = A[i,j];          /*save old value of element*/
20.         A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.           A[i,j+1] + A[i+1,j]);     /* compute average*/
22.         mydiff += abs(A[i,j] - temp);
23.       end for_all
24.     end for_all
24a.      REDUCE (mydiff, diff, ADD);
25.     if (diff/(n*n) < TOL) then done = 1;
26.   end while
```


The **decomp** statement has a twofold purpose.

- It specifies the assignment of iterations to processes.

  The first dimension (rows) is partitioned into *nprocs* contiguous blocks. The second dimension is not partitioned at all.

  Specifying `[CYCLIC, *, nprocs]` would have caused a cyclic partitioning of rows among `nprocs` processes.

  Specifying `[*,CYCLIC, nprocs]` would have caused a cyclic partitioning of columns among `nprocs` processes.

  Specifying `[BLOCK, BLOCK, nprocs]` would have implied a 2D contiguous block partitioning.

  For all of these partitionings, tell which processing element in a 64-PE system would compute A[33, 65]. If the grid is 1024 x 1024?

- It specifies the assignment of grid data to memories on a distributed-memory machine. (Follows the *owner-computes* rule.)

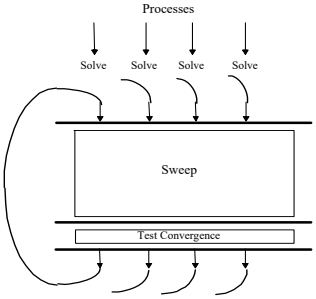
The *mydiff* variable allows local sums to be computed.

The **reduce** statement tells the system to add together all the *mydiff* variables into the shared *diff* variable.

Shared-memory model

In this model, we need mechanisms to create processes and manage them.

After we create the processes, they interact as shown on the right.




```
1.      int n, nprocs;              /*matrix dimension and number of processors to be used*/
2a.     double**A, diff;            /*A is global (shared) array representing the grid*/
                                    /*diff is global (shared) maximum difference in current
                                    sweep*
2b.     LOCKDEC(diff_lock);         /*declaration of lock to enforce mutual exclusion*/
2c.     BARDEC (bar1);              /*barrier declaration for global synchronization between
                                    sweeps*

3.   main()
4.   begin
5.          read(n); read(nprocs);      /*read input matrix size and number of processes */
6.          A ← -         (a two-dimensional array of size n+2 by n+2 doubles);
7.          initialize(A);              /*initialize A in an unspecified way*/
8a.         CREATE (nprocs-1, Solve, A);
8.       Solve(A);                      /*main process becomes a worker
8b.         WAIT_FOR_END (nprocs-1);    /*wait for all child processes created to terminate*/
9.   end main

10.  procedure Solve(A)
11.     double**A;                                  /*A is entire n+2-by-n+2 shared array,
                                                    as in the sequential program*/
12.  begin
13.     int i,j, pid, done = 0;
14.     float temp, mydiff = 0;                     /*private variables*/
14a.    int mymin = 1 + (pid * n/nprocs);           /*assume that n is exactly divisible by*/
14b.    int mymax = mymin + n/nprocs - 1            /*nprocs for simplicity here*/

15.     while (!done) do                        /* outer loop over all diagonal elements*/
16.        mydiff = diff = 0;                   /*set global diff to 0 (okay for all to do it)*/
16a.    BARRIER(bar1, nprocs);                  /*ensure all reach here before anyone modifies diff*/
17.        for i ← mymin to mymax do            /*for each of my rows */
18.                for j ← 1 to n do            /*for all nonborder elements in that row*/
19.                temp = A[i,j];
20.                A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                   A[i,j+1] + A[i+1,j]);
22.                mydiff += abs(A[i,j] - temp);
23.             endfor
24.        endfor                               /*update global diff if necessary*/
25a.       LOCK(diff_lock);
25b.       diff += mydiff;
25c.       UNLOCK(diff_lock);
25d.       BARRIER(bar1, nprocs);               /*ensure all reach here before checking if done*/
25e.       if (diff/(n*n) < TOL) then done = 1;        /*check convergence; all get
                                                       same answer*/
25f.       BARRIER(bar1, nprocs);
26.     endwhile
27.  end procedure
```

What are the main differences between the serial program and this program?

- The first process creates *nprocs*–1 worker processes. All *n* processes execute *Solve*.

  All processes execute the same code.

  But all do *not* execute the same instructions at the same time.

- Private variables like *mymin* and *mymax* are used to control loop bounds.
- All processors need to—

  - complete an iteration before any process tests for convergence. Why?
  - test for convergence before any process starts the next iteration. Why?

  Notice the use of *barrier synchronization* to achieve this.

  What could happen if the barrier at Line 16a was removed?

  What could happen if the barrier at Line 25d was removed?

  What could happen if the barrier at Line 25f was removed?

- Locks must be plsaced around updates to *diff*, so that no two processors update it at once. Otherwise, inconsistent results could ensue.

p_1	p_2	
`r1 ← diff`		{ p_1 gets 0 in its `r1`}
	`r1 ← diff`	{ p_2 also gets 0}
`r1 ← r1+r2`		{ p_1 sets its `r1` to 1}
	`r1 ← r1+r2`	{ p_2 sets its `r1` to 1}
`diff ← r1`		{ p_1 sets *diff* to 1}
	`diff ← r1`	{ p_2 also sets *diff* to 1}

If we allow only one processor at a time to access *diff*, we can avoid this *race condition*.

What is one performance problem with using locks?

Note that at least some processors need to access *diff* as a non-local variable.

What is one technique that our shared-memory program uses to diminish this problem of serialization?

Message-passing model

The program for the message-passing model is also similar, but again there are several differences.

- There's no shared address space, so we can't declare array *A* to be shared.

  Instead, each processor holds the rows of *A* that it is working on.

- The subarrays are of size (*n*/*nprocs* + 2) × (*n* + 2).
  This allows each subarray to have a copy of the boundary rows from neighboring processors. Why is this done?

  These *ghost* rows must be copied explicitly, via **send** and **receive** operations.

  Note that **send** is not synchronous; that is, it doesn't make the process wait until a corresponding **receive** has been executed.

  What problem would occur if it did?

- Since the rows are copied and then not updated by the processors they have been copied from, the boundary values are more out-of-date than they are in the sequential version of the program.

  This may or may not cause more sweeps to be needed for convergence.

- The indexes used to reference variables are *local* indexes, not the "real" indexes that would be used if array *A* were a single shared array.

```
1.  int pid, n, b;                    /*process id, matrix dimension and number of
                                        processors to be used*/
2.  float **myA;
3.  main()
4.  begin
5.      read(n);   read(nprocs);      /*read input matrix size and number of processes*/
8a.     CREATE (nprocs-1, Solve);
8b.     Solve();                      /*main process becomes a worker too*/
8c.     WAIT_FOR_END (nprocs-1);      /*wait for all child processes created to terminate*/
9.  end main

10.  procedure Solve()
11.  begin
13.     int i,j, pid, n' = n/nprocs, done = 0;
14.     float temp, tempdiff, mydiff = 0;    /*private variables*/
6.   myA ← malloc(a 2-d array of size [n/nprocs + 2] by n+2);
                                      /*my assigned rows of A*/
7.  initialize(myA);                  /*initialize my rows of A, in an unspecified way*/

15.  while (!done) do
16.     mydiff = 0;                   /*set local diff to 0*/
16a.    if (pid != 0) then SEND(&myA[1,0],n*sizeof(float),pid-1,ROW);
16b.    if (pid != nprocs-1) then
          SEND(&myA[n',0],n*sizeof(float),pid+1,ROW);
16c.    if (pid != 0) then RECEIVE(&myA[0,0],n*sizeof(float),pid-1,ROW);
16d.    if (pid != nprocs-1) then
          RECEIVE(&myA[n'+1,0],n*sizeof(float), pid+1,ROW);
                                      /*border rows of neighbors have now been copied
                                        into myA[0,*] and myA[n'+1,*]*/
17.     for i ← 1 to n' do            /*for each of my (nonghost) rows*/
18.       for j ← 1 to n do           /*for all nonborder elements in that row*/
19.         temp = myA[i,j];
20.         myA[i,j] = 0.2 * (myA[i,j] + myA[i,j-1] + myA[i-1,j] +
21.           myA[i,j+1] + myA[i+1,j]);
22.         mydiff += abs(myA[i,j] - temp);
23.       endfor
24.     endfor
                                      /*communicate local diff values and determine if
                                        done; can be replaced by reduction and broadcast*/
25a.      if (pid != 0) then                  /*process 0 holds global total diff*/
25b.        SEND(mydiff,sizeof(float),0,DIFF);
25c.        RECEIVE(done,sizeof(int),0,DONE);
25d.      else                                /*pid 0 does this*/
25e.        for i ← 1 to nprocs-1 do          /*for each other process*/
25f.          RECEIVE(tempdiff,sizeof(float),*,DIFF);
25g.        mydiff += tempdiff;               /*accumulate into total*/
25h.      endfor
25i       if (mydiff/(n*n) < TOL) then      done = 1;
25j.        for i ← 1 to nprocs-1 do          /*for each other process*/
25k.          SEND(done,sizeof(int),i,DONE);
25l.        endfor
25m.    endif
26.  endwhile
27.  end procedure
```

There are one or more typos in the **if** statements involving **pids**. Which statement(s)? What are the error(s)?

Data parallel algorithms[1]

(Guy Steele): The data-parallel programming style is an approach to organizing programs suitable for execution on massively parallel computers.

In this lecture, we will—

- characterize the _____ programming style,
- examine the building blocks used to construct data-parallel programs, and
- see how to fit these building blocks together to make useful algorithms.

All programs consist of code and data put together. If you have more than one processor, there are various ways to organize parallelism.

- Control parallelism: Emphasis is on extracting parallelism by orienting the program's organization around the parallelism in the code.
- _____ parallelism: Emphasis is on organizing programs to extract parallelism from the organization of the data.

With data parallelism, typically all the processors are at roughly the same point in the program.

Control and data parallelism vs. SIMD and MIMD.

- You may write a data-parallel program for a MIMD computer, or
- a control-parallel program which is executed on a SIMD computer.

Emphasis in this talk will be on styles of organizing programs. It becomes an engineering issue whether it is appropriate to organize the hardware to match the program.

[1]Video © 1991, Thinking Machines Corporation. This video is available from University Video Communications, http://www.uvc.com.

The sequential programming style, typified by C and Pascal, has building blocks like—

- scalar arithmetic operators,
- control structures like **if** … **then** … **else**, and
- subscripted array references.

The programmer knows essentially how much these operations cost. E.g., addition and subtraction have similar costs; multiplication may be more expensive.

To write data-parallel programs effectively, we need to understand the cost of data-parallel operations.

- Elementwise operations (carried on independently by processors; typically _____ operations and tests).
- Conditional operations (also elementwise, but some processors may not participate, or act in various ways).
- Replication
- _____
- Permutation
- Parallel prefix (scan)

An example of an elementwise operation:

Elementwise addition

$C = A + B$

3	1	4	5	2	1	3	2
6	2	1	3	0	1	1	5
9	3	5	8	2	2	4	7

+

Elementwise test

if $(A > B)$

●	●	○	○	○	●	○	●
3	1	4	5	2	1	3	2
6	2	1	3	0	1	1	5
0	0	0	0	0	0	0	0

>

The results can be used to "conditionalize" future operations:

if $(A > B)$ $C = A + B$

●	●	○	○	○	●	○	●
3	1	4	5	2	1	3	2
6	2	1	3	0	1	1	5
0	0	5	8	2	0	4	0

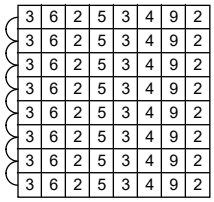
+

The set of bits that is used to conditionalize the operations is frequently called a *condition mask* or a *context*. Each processor can perform different computations based on the data it contains.

Building blocks

Communications operations:

- _____: Get a single value out to all processors. This operation happens so frequently that is worthwhile to support in hardware. It is not unusual to see a hardware bus of some kind.
- Spreading (nearest-neighbor grid). One way is to have each row copied to its nearest neighbor.

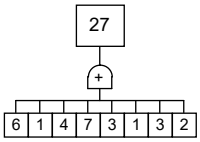


A better way is to use a copy-scan:

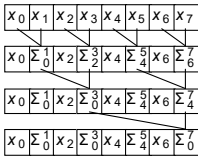
- On the first step, the data is copied to the row that is directly below.
- On the second step, data is copied from each row that has the data to the row that is two rows below.
- On the third step, data is copied from each row to the row that is four rows below.

In this way, the row can be copied in logarithmic time, if we have the necessary interconnections.

- _____—essentially the inverse of broadcasting. Each processor has an element, and you are trying to combine them in some way to produce a single result.



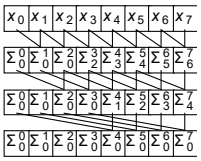
Summing a vector in logarithmic time:



Most of the time during the course of this algorithm, most processors have *not* been busy.

So while it is fast, we haven't made use of all the processors.

Suppose you don't turn off processors; what do you get? Vector sum-prefix (sum-scan).

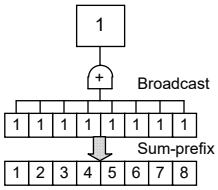


Each processor has received the sum of what it contained, plus all the processors preceding it.

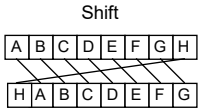
We have computed the sums of all *prefixes*—initial segments—of the array.

This can be called the checkbook operation; if the numbers are a set of credits and debits, then the prefixes are the set of running balances that should appear in your checkbook.

- _____. We wish to assign a different number to each processor.

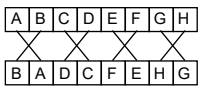


- Regular permutation.

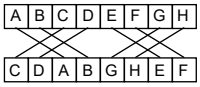


Of course, one can do shifting on two-dimensional arrays too; you might shift it one position to the north.

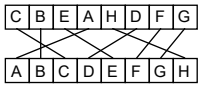
Another kind of permutation is an odd-even swap:



Distance 2^k swap:



Some algorithms call for performing irregular permutations on the data.



The permutation depends on the data. Here we have performed a sort. (Real sorting algorithms have a number of intermediate steps.)

Example: image processing

Suppose we have a rocket ship and need to figure out where it is.

Some of the operations are strictly local. We might focus in on a particular region, and have each processor look at its values and those of its neighbor.

This is a local operation; we shift the data back and forth and have each processor determine whether it is on a boundary.

When we assemble this data and put it into a global object, the communication patterns are dependent on the data; it depends on where the object happened to be in the image.

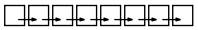
Irregularly organized data

Most of our operations so far were on arrays, regularly organized data.

We may also have operations where the data are connected by pointers.

In this diagram, imagine the processors as being in completely different parts of the machine, known to each other only by an address.

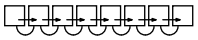
_____ doubling:



I originally thought that nothing could be more essentially sequential than processing a linked list. You just can't find the third one without going through the second one. But I forgot that there is processing power at each node.

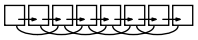
The most important technique is *pointer doubling*. This is the pointer analogue of the spreading operation we looked at earlier to make a copy of a vector into a matrix in a logarithmic number of steps.

In the first step, each processor makes a copy of the pointer it has to its neighbor.



In the rest of the steps, each processor looks at the processor it is pointing to with its extra pointer, and gets a copy of *its* pointer.

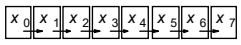
In the first step, each processor has a pointer to the next processor. But in the next step, each processor has a pointer to the processor two steps away in the linked list.



In the next step, each processor has a pointer to the pointer four processors away (except that if you fall off the end of the chain, you don't update the pointer).

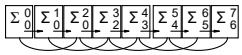
Eventually, in a logarithmic number of steps, each processor has a pointer to the end of the chain.

How can this be used? In partial sums of a linked list.

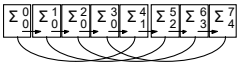


At the first step, each processor takes the pointer to its neighbor.

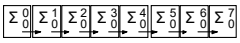
At the next step, each processor takes the value that it holds, and adds it into the value in the place pointed to:



Now we do this again:



And after the third step, you will find that each processor has gotten the sum of its own number plus all the preceding ones in the list.



Speed vs. efficiency: In sequential programming, these terms are considered to be synonymous. But this coincidence of terms comes about only because you have a single processor.

In the parallel case, you may be able to get it to go fast by doing extra work.

Let's take a look at the serial vs. parallel algorithm for summing an array.

_____-Reduction

	Serial	Parallel
Processors	1	N
Time steps	N–1	log N
Additions	N–1	N–1
Cost	N–1	N log N
Efficiency	1	$\frac{1}{\log N}$

Sum – Prefix

	Serial	Parallel
Processors	1	n
Time steps	n–1	log n
Additions	n–1	n (log n–1)

Cost	n–1	n log n
Efficiency	1	$\frac{\log n-1}{\log n}$

The serial version of sum–prefix is similar to the serial version of sum–reduction, but you save the partial sums. You don't need to do any more additions, though.

In the parallel version, the number of additions is much greater. You use n processors, and commit log n time steps, and nearly all of them were busy.

As n gets large, the efficiency is very close to 1. So this is a very efficient algorithm. But in some sense, the efficiency is bogus; we've kept the processors busy doing more work than they had to do. Only n–1 additions are really required to compute sum–prefix. But n(log n–1) additions are required to do it fast.

Thus, the business of measuring the speed and efficiency of a parallel algorithm is tricky. The measures I used are a bit naïve. We need to develop better measures.

Exercise: Submit your answers here.

Calculate the speedup of summing a vector using copy-scan (turning off the processors that are not in use).

- How long does it take to sum the vector serially?
- How long does it take to sum it using copy-scan?
- What is the speedup?

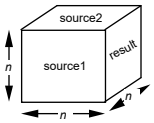
What is the *efficiency* (speedup ÷ # of processors) of summing a vector with copy-scan?

In the parallel version of summing an array via sum-prefix, a "bogus" efficiency is mentioned. What would be the "non-bogus" efficiency of the same algorithm?

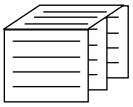
Putting the building blocks together

Let's consider *matrix multiply*.

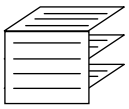
One way of doing this with a brute-force approach is to use n^3 processors.



1. Replicate. The first step is to make copies of the first source array, using a spread operation.

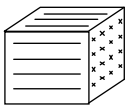


2. Replicate. Then we will do the same thing with the second source, spreading those down the cube.

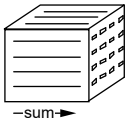


So far, we have used O(log n) time.

3. Elementwise multiply. n^3 operations are performed, one by each processor.



4. Perform a parallel sum operation, using the doubling-reduction method.



We have multiplied two matrices in
$O(\log n)$ time, but at the cost of using n^3 processors.

Brute force: n^3 processors $O(\log n)$ time

Also, if we wanted to add the sum to one of the matrices, it's in the wrong place, and we would incur an additional cost to move it.

Cannon's method

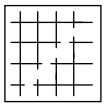
There's another method that only requires n^2 processors. We take the two source arrays and put them in the same n^2 processors. The result will also show up in the same n^2 processors.

We will pre-_____ the two source arrays.

- The first array has its rows skewed by different amounts.

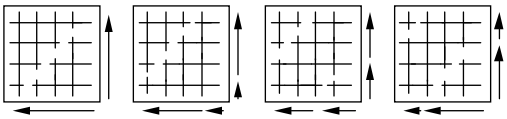
- The columns of the second array are skewed.

The two arrays are overlaid, and they then look like this:



This is a systolic algorithm; it rotates both of the source matrices at the same time.

- The first source matrix is rotated horizontally.
- The second source matrix is rotated vertically.



At the first time step, the 2nd element of the first row and the 2nd element of the first column meet in the upper left corner. They are then multiplied and accumulated.

At the second time step, the 3rd element of the first row and the 3rd element of the first column meet in the upper left corner. They are then multiplied and accumulated.

At the third time step, the 4th element of the first row and the 4th element of the first column meet in the upper left corner. They are then multiplied and accumulated.

At the fourth time step, the 1st element of the first row and the 1st element of the first column meet in the upper left corner. They are then multiplied and accumulated.

The same thing is going on at all the other points of the matrix.

The _____ serves to cause the correct elements of each row and column to meet at the right time.

Cannon's method: n^2 processors $O(n)$ time

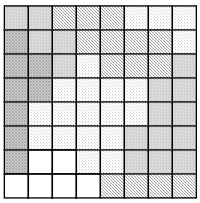
An additional benefit is that the matrix ends up in the right place.

Labeling regions in an image

Let's consider a really big example.

Instead of the rocket ship earlier in the lecture, we'll consider a smaller region. (This is one of the problems in talking about data-parallel algorithms. They're useful for really large amounts of data, but it's difficult to show that on the screen.)

We have a number of regions in this image. There's a large central "green" region, and a "red-orange" region in the upper right-hand corner. Some disjoint regions have the same color.



We would like to compute a result in which each region gets assigned a distinct number.

We don't care which number gets assigned, as long as the numbers are distinct (even for regions of the same color.

0	0	2	2	2	5	5	5
8	0	0	2	2	2	2	5
8	8	0	19	2	2	2	23
8	8	19	19	19	19	23	23
8	19	19	19	19	19	23	23
8	19	19	19	19	23	23	23
8	49	49	19	19	23	23	23
49	49	49	49	60	60	60	60

For example, here the central green region has had all its pixels assigned the value 19.

The squiggly region in the upper left corner has received 0 in all its pixels.

The region in the upper right, even though the same color as the central green region, has received a different value.

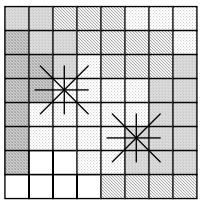
Let's see how all the building blocks we have discussed can fit together to make an interesting algorithm.

First, let's assign each processor a different number.

Here I've assigned the numbers sequentially across the rows, but any distinct numbering would do.

We've seen how the enumeration technique can do this in a logarithmic number of time steps.

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63



Next, we have each of the pixels examine the values of its eight neighbors.

This is easily accomplished using regular _____— namely, shifts of the matrix.

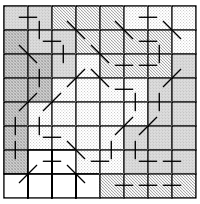
We shift it up, down, left, right, to the northeast, northwest, southeast, and southwest.

This is enough for each processor to do elementwise computation and decide whether it is on the border.

(There are messy details, but we won't discuss them here, since they have little to do with parallelism.)

The next computation will be carried out only by processors that are on the borders (an example of conditional operation).

We have each of the processors again consider the pixel values that came from its neighbors, and



inquire again, using shifting, if each of its neighbors are border elements.

This is enough information to figure out which of your neighbors are border elements in the same region, so you can construct pointers to them.

0	1	2		4	5	6	
8	9	10	11		13	14	15
	17	18	19	20	21	22	23
	25	26		28	29	30	
32	33				37	38	
40	41	42		44	45		
48	49	50	51	52	53	54	55
56			59	60	61	62	63

Now we have stitched together the borders in a linked list.

We now use the pointer-doubling algorithm. Each pixel on the borders considers the number that it was assigned in the enumeration step.

We use the pointer-doubling algorithm to do a reduction step using the **min** operation.

0	0	2		2	5	5	
8	0	0	2	2	2	2	5
	8	0	19	2	2	2	23
	8	19		19	19	23	
8	19				19	23	
8	19	19		19	23		
8	49	49	19	19	23	23	23
49			49	60	60	60	60

Each linked list performs pointer-doubling around that list, and determines which number is the smallest in the list.

Then another pointer-doubling algorithm makes copies of that minimum all around the list.

Finally, we can use _____ operation, not on linked lists, but by operating on the columns (or the rows) to copy the processor labels from the borders to the rows.

Other items, particularly those on the edge, may need the numbers propagated up instead of down. So you do a scan in both directions.

The operation used is a non-commutative operation that copies the old number from the neighbor, unless it comes across a new number.

0	0	2	2	2	5	5	5
8	0	0	2	2	2	2	5
8	8	0	19	2	2	2	23
8	8	19	19	19	19	23	23
8	19	19	19	19	19	23	23
8	19	19	19	19	23	23	23
8	49	49	19	19	23	23	23
49	49	49	49	60	60	60	60

scan

This is known as Lim's algorithm.

Region labeling: $O(n^2)$ processors. $O(\log n)$ time

(Each of the steps was either constant time or $O(\log n)$ time.)

Data-parallel programming makes it easy to organize operations on large quantities of data in massively parallel computers.

It differs from sequential programming in that its emphasis is on operations on entire sets of data instead of one element at a time.

You typically find fewer loops, and fewer array subscripts.

On the other hand, data-parallel programs are like sequential programs, in that they have a single thread of control.

In order to write good data-parallel programs, we must become familiar with the necessary building blocks for the construction of data-parallel algorithms.

With one processor per element, there are a lot of interesting operations which can be performed in constant time, and other operations which take logarithmic time, or perhaps a linear amount of time.

This also depends on the connections between the processors. If the hardware doesn't support sufficient connectivity among the processors, a communication operation may take more time than would otherwise be required.

Once you become familiar with the building blocks, writing a data-parallel program is just as easy (and just as hard) as writing a sequential program. And, with suitable hardware, your programs may run much faster.

Exercise: Run through Lim's algorithm on the grid given here.

Questions and answers: [not shown during class] *Question:* (Bert Halstead): Do you ever get into problems when you have highly data-dependent computations, and it's hard to keep more than a small fraction of the processors doing the same operation at the same time?

Answer: Yes. That's one reason for making the distinction between the data-parallel style and _____ hardware. The best way to design a system to give you the most flexibility without making it overly difficult to control is, I think, still an open research question.

Question (Franklin Turback): Your algorithms seem to be based on the assumption that you actually have enough processors to match the size of your problem. If you have more data than processors, it seems that the logarithmic time growth is no longer justified.

Answer: There's no such thing as a free lunch. Making the problem bigger makes it run slower. If you have a much larger problem that won't fit, you're going to have to buy a larger computer.

Question: How about portability of programs to different machines?

Answer: Right now it's very difficult, because so far, we haven't agreed on standards for the right building blocks to support. Some architectures support some building blocks but not others. This is why you end up with non-portabilities of efficiencies of running times.

Question: For dealing with large sparse matrices, there are methods that we use to reduce complexity. If this is true, how do you justify the overhead cost of parallel processing?

Answer: Yes, that is true. It would not be appropriate to use that kind of algorithm on a sparse matrix, just as you don't use the usual sequential triply-nested loop.

_____ processing on a data-parallel computer calls for very different approaches. They typically call for the irregular communication and permutation techniques that I illustrated.

Question: What about non-linear programming and algorithms like branch-and-bound?

Answer: It is sometimes possible to use data-parallel algorithms to do seemingly unstructured searches, as on a game tree, by maintaining a work queue, like you might do in a more control-parallel, and at every step, taking a large number of task items off the queue by using an enumeration step and using the results of that enumeration to assign them to the processors.

This may depend on whether the rest of the work to be done is sufficiently similar. If it's not, then control parallelism may be more appropriate.

Question: With the current software expertise in 4GLs for sequential machines, do you think that developing data-parallel programming languages will end up at least at 4GL level?

Answer: I think we are now at the point where we know how to design data-parallel languages at about the level of expressiveness as C, Fortran, and possibly Lisp. I think it will take awhile before we can raise our level of understanding to the level we need to design 4GLs.

Parallel access to linked data structures

[Solihin Ch. 4] Answer the questions below.

Name some linked data structures. Linked lists, trees, graphs, hash tables.

What operations can be performed on *all* of these structures? Insertion, deletion, search.

Why is it hard to parallelize these operations? Because pointer-chasing involves frequent loop-carried dependences.

Explain how the following code illustrates such a dependence.

```
void addValue(pIntList pList, int key, int x) {
   pIntListNode p = pList->head;
   while (p != NULL) {
     if (p->key == key)
       S1: p->data = p->data + x;
     S2: p = p->next;
 }
}
```

In the notation introduced in Lecture 9, how would the dependence be written?

`S1[i]` →T `S1[i+1]`, `S2[i]` →T `S2[i+1]`, except that there is no `i` in the program.

If we just look at the loops in an "LDS" program, we won't find any parallelism to be exploited.

So, where can we find the opportunity to execute anything in parallel? The "algorithm level"—paralellism between the *operations* that are performed on the LDS.

Conceptually, we can allow several operations to be performed in parallel. What kind of operations? Insertion, deletion, search, etc.

But how do we decide *which* operations can be performed in parallel?

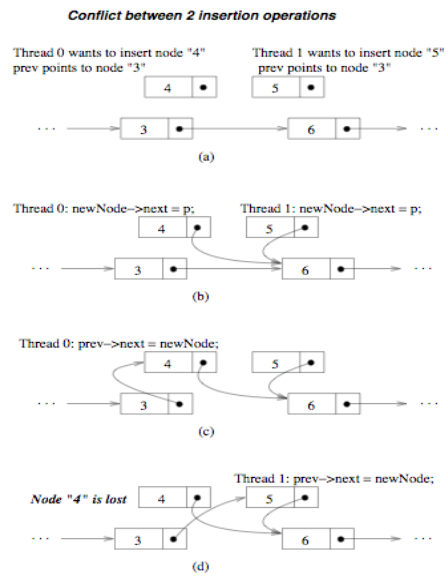
Correctness of parallel LDS operations

Serializability: A parallel execution of a group of operations (or primitives) is said to be *serializable* if there is some sequence of operations (or primitives) that produce an identical result.

Suppose a node insertion i_1 and a node deletion d_1 are performed in parallel. The outcome must be equivalent to either

- i_1 followed by d_1, or
- d_1 followed by i_1.

Conflict between two insertions



Let's look at the simple case of a singly-linked list.

Suppose two items are inserted in parallel: insert both 4 and 5.

Serializable outcomes:

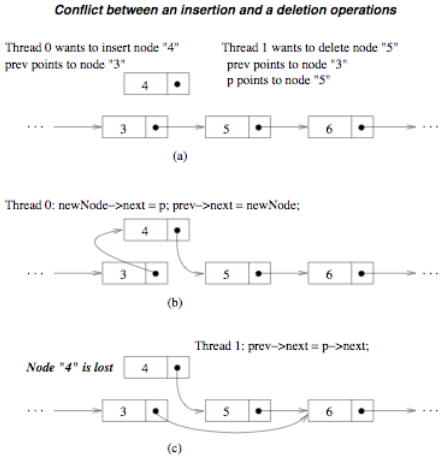
insert 4, then insert 5

or insert 5, then insert 4

In any case, both nodes 4 and 5 must be in the list at the end of execution

What could happen if the operations are not parallelized correctly? Node 4 could be lost, or node 5 could be lost.

Conflict between an insertion and a deletion



Serializable outcome:

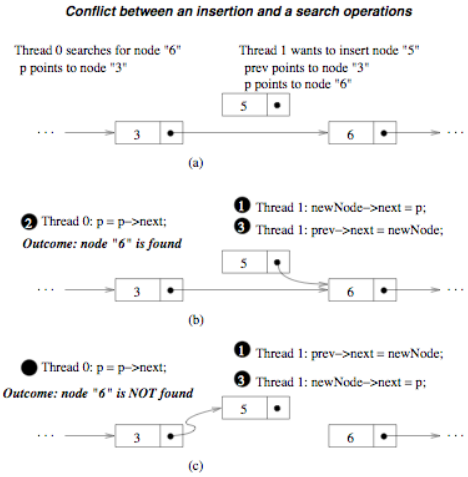
insert 4, then delete 5, or

delete 5, then insert 4

in both cases, at the end of execution, node 4 is in the list, but node 5 is not in the list

In the case shown, node 4 is lost. What would be a sequence that produces *another* incorrect result? What would happen with this sequence? (You may use this worksheet.)

Conflict between an insertion and a search



Suppose we attempt

insert 5, then search 6

or, search 6, then insert 5

in both cases, at the end of execution,

- 5 must be in the list, and
- 6 must be found

Depending on when the insertion code is executed,

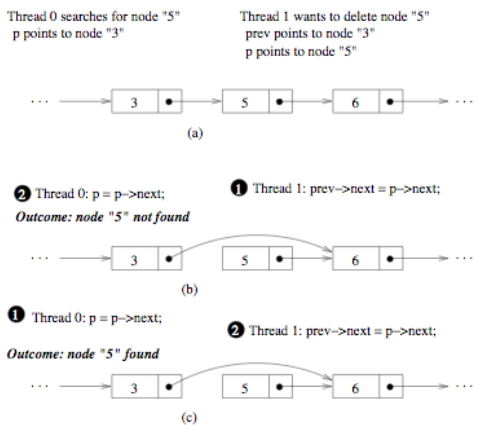
- node 6 will be found, or
- node 6 may not be found, and an uninitialized link may be followed.

Conflict between a deletion and a search

- Deletion and search
  - delete 5, then search for 5
  - search for 5, then delete 5
- Possible outcomes
  - Node 5 may be found or not found
  - Node 5 is deleted from the list

What, if anything, is the problem with these outcomes?
Nothing; the operations are serializable.

Conflict between a deletion and a search operations



Main Observations

- Parallel execution of two operations that affect a *common node*, in which at least one operation *involves writing* to the node, can produce conflicts that lead to *non-serializable outcome*.
- Under some circumstances, a serializable outcome may still be achieved, despite the conflicts mentioned above.
- Conflicts can also occur between LDS operations and memory-management functions such as allocation and deallocation.

Parallelization strategies

- Parallelization among readers
  - Very simple
  - Works well if structure is modified infrequently

- Global lock approach
  - Relatively simple
  - Parallel traversal, followed by sequential list modifications
- Fine-grain lock approach
  - A lock is associated with each node.
  - Each operation locks only nodes that need to be accessed exclusively.
  - Complex: Deadlock can occur; memory allocation and deallocation become more complex

Parallelization among readers

- Basic idea
  - (Read-only) operations that do not modify the list can execute in parallel.
  - (Write) operations that modify the list execute sequentially
- How to enforce
  - A read-only operation acquires a read lock
  - A write operation acquires a write lock
- Construct a lock-compatibility table

Already-granted lock	Read lock requested	Write lock requested
Read lock	Yes	No
Write lock	No	No

Example

```
IntListNode_Search(int x)
{
  acq_read_lock();
  …
  …
  …
  rel_read_lock();
}
```

```
IntListNode_Insert(node *p)
{
  acq_write_lock();
  …
  …
  …
  rel_write_lock();
}
```


Global-lock approach

- Each operation logically has two steps
  - Traversal
    - Node insertion: Find the correct location for the node
    - Node deletion: Find the node to delete
    - Node search: Find the sought-for node
  - List modification
- Basic idea: perform the traversal in parallel, but modify the list in a critical section, i.e., modify the list between the time that a write lock is acquired and when it is released (that's what a c.s. is).
- Pitfall
  - The list may have changed by the time the write-lock is acquired,
  - so the assumptions must be re-validated.

Example

```
IntListNode_Insert(node *p)
{
  …
  /* perform traversal */
  …
  acq_write_lock();
  /* then check validity:
       nodes still there?
       link still valid? */
  /* if not valid, repeat traversal */
  /* if valid, modify list */
  …
  rel_write_lock();
}
```

Fine-grain locking approach

- Associate each node with a lock (read, write).
- Each operation locks only needed nodes.
- (Read and write) operations execute in parallel except when they conflict on some nodes. Fill in the blanks below.
  - Nodes that will be modified are write-locked.

  - Nodes that are read and must remain unchanged are read-locked.
- Pitfall: Deadlock becomes possible.
  - Suppose one operation locks node 1 and then needs to lock node 2, while another operation locks node 2 and then needs to lock node 1.
  - Then neither operation can complete before the other operation frees the lock it is holding.
- Deadlocks can be prevented by imposing a global lock-acquisition order.

Example

```
void insert(pIntList pList, int x){
  int succeed;
  … /* traversal code to find where to insert */

  /* insert the node at head or between prev & p */
  succeed = 0;
  do {
    acq_write_lock(prev);
    acq_read_lock(p);
    if (prev->next != p || prev->deleted || p->deleted)
    {
      rel_write_lock(prev);
      rel_read_lock(p);
      … /* repeat traversal */
    }
    else
      succeed = 1;
  } while (!succeed);

  /* prev and p are now valid, so insert node */
  newNode->next = p;
  if (prev != NULL)
    prev->next = newNode;
  else
    pList->head = newNode;
  rel_write_lock(prev);
  rel_read_lock(p);
```

```
}
```

Questions

What do the tests **prev->deleted** and **p->deleted** mean? They ask whether the node has been deleted (by checking its **deleted** field); nodes are marked deleted rather than deallocated.

Why is garbage collection used, rather than explicit deletion? Because nodes may be deleted only when they are not involved in any operation. This would require keeping reference counts on all the nodes, which is too expensive.

The delete operation is similar; code that is the same is shown in green.

```
void delete(pIntList pList, int x){
  int succeed;
  … /* traversal code to find node to delete */

  /* node has been found; perform the deletion */
  succeed = 0;
  do {
    acq_write_lock(prev);
    acq_write_lock(p);
    if (prev->next != p || prev->deleted || p->deleted)
    {
      rel_write_lock(prev);
      rel_write_lock(p);
      … /* repeat traversal; return if not found */
    }
    else
      succeed = 1;
  } while (!succeed);

  /* prev and p are now valid, so delete node */
  if (prev == NULL) { /* delete head node */
    acq_write_lock(pList);
    pList->head = p->next;
    rel_write_lock(pList);
  }
  else /* delete non-head node */
    prev->next = p->next;
  p->deleted = 1; /*don't deallocate; mark deleted*/
  rel_write_lock(prev);
  rel_write_lock(p);
}
```