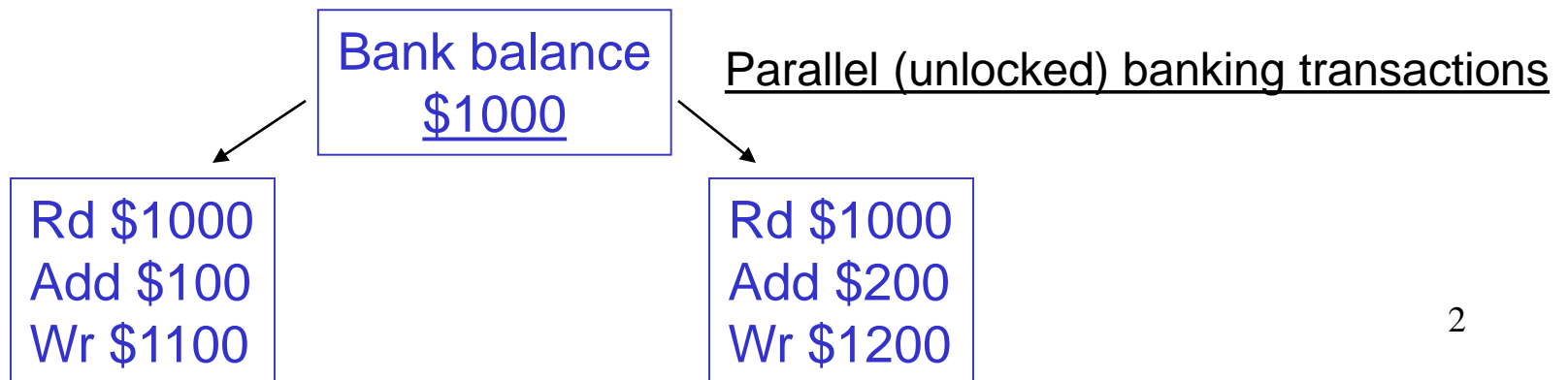# Lecture 25: Multiprocessors

- Today's topics:
    - Synchronization
    - Consistency
    - Shared memory vs message-passing
    - Simultaneous multi-threading (SMT)

Constructing Locks

- Applications have phases (consisting of many instructions) that must be executed atomically, without other parallel processes modifying the data

- A lock surrounding the data/code ensures that only one program can be in a critical section at a time

- The hardware must provide some basic primitives that allow us to construct locks with different properties

Bank balance
$1000

Parallel (unlocked) banking transactions

Rd $1000
Add $100
Wr $1100

Rd $1000
Add $200
Wr $1200

Synchronization

- The simplest hardware primitive that greatly facilitates synchronization implementations (locks, barriers, etc.) is an atomic read-modify-write

- Atomic exchange: swap contents of register and memory

- Special case of atomic exchange: test & set: transfer memory location into register and write 1 into memory (if memory has 0, lock is free)

- lock: t&s   register, location
                bnz   register, lock
                CS
                st   location, #0

> When multiple parallel threads execute this code, only one will be able to enter CS

Coherence Vs. Consistency

- Recall that coherence guarantees (i) write propagation (a write will eventually be seen by other processors), and (ii) write serialization (all processors see writes to the same location in the same order)

- The consistency model defines the ordering of writes and reads to different memory locations – the hardware guarantees a certain consistency model and the programmer attempts to write correct programs with those assumptions

Consistency Example

- Consider a multiprocessor with bus-based snooping cache coherence

Initially A = B = 0

P1	P2
A ← 1	B ← 1
…	…
if (B == 0)	if (A == 0)
Crit.Section	Crit.Section

Consistency Example

- Consider a multiprocessor with bus-based snooping cache coherence

```
Initially A = B = 0
  P1                    P2
A ← 1                 B ← 1
…                     …
if (B == 0)           if (A == 0)
  Crit.Section          Crit.Section
```

The programmer expected the
above code to implement a
lock – because of ooo, both processors
can enter the critical section

The consistency model lets the programmer know what assumptions
they can make about the hardware's reordering capabilities

6

Sequential Consistency

- A multiprocessor is sequentially consistent if the result of the execution is achieveable by maintaining program order within a processor and interleaving accesses by different processors in an arbitrary fashion

- The multiprocessor in the previous example is not sequentially consistent

- Can implement sequential consistency by requiring the following: program order, write serialization, everyone has seen an update before a value is read – very intuitive for the programmer, but extremely slow

# Shared-Memory Vs. Message-Passing

Shared-memory:
- Well-understood programming model
- Communication is implicit and hardware handles protection
- Hardware-controlled caching

Message-passing:
- No cache coherence → simpler hardware
- Explicit communication → easier for the programmer to restructure code
- Software-controlled caching
- Sender can initiate data transfer

Ocean Kernel

```
Procedure Solve(A)
begin
  diff = done = 0;
  while (!done) do
    diff = 0;
    for i ← 1 to n do
      for j ← 1 to n do
        temp = A[i,j];
        A[i,j] ← 0.2 * (A[i,j] + neighbors);
        diff += abs(A[i,j] – temp);
      end for
    end for
    if (diff < TOL) then done = 1;
  end while
end procedure
```

Row 1

Row k

Row 2k

Row 3k

…
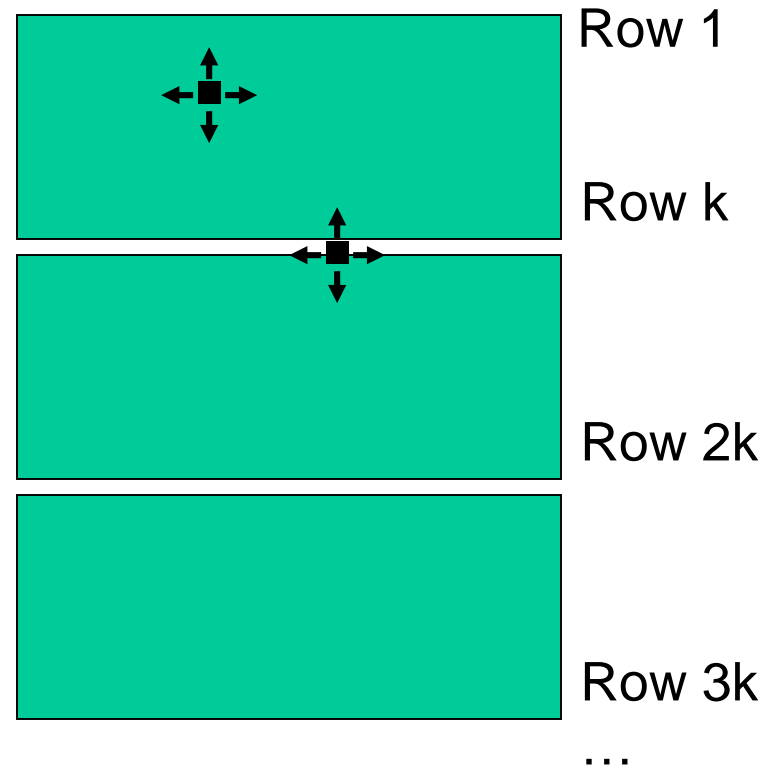
Shared Address Space Model

```
int  n, nprocs;
float  **A, diff;
LOCKDEC(diff_lock);
BARDEC(bar1);


main()
begin
  read(n); read(nprocs);
  A ← G_MALLOC();
  initialize (A);
  CREATE (nprocs,Solve,A);
  WAIT_FOR_END (nprocs);
end main
```

```
procedure Solve(A)
    int i, j, pid, done=0;
    float temp, mydiff=0;
    int mymin = 1 + (pid * n/procs);
    int mymax = mymin + n/nprocs -1;
    while (!done) do
      mydiff = diff = 0;
      BARRIER(bar1,nprocs);
      for i ← mymin to mymax
        for j ← 1 to n do
            …
          endfor
        endfor
      LOCK(diff_lock);
      diff += mydiff;
      UNLOCK(diff_lock);
      BARRIER (bar1, nprocs);
      if (diff < TOL) then done = 1;
      BARRIER (bar1, nprocs);
    endwhile
```

Message Passing Model

```
main()
  read(n); read(nprocs);
  CREATE (nprocs-1, Solve);
  Solve();
  WAIT_FOR_END (nprocs-1);

procedure Solve()
  int i, j, pid, nn = n/nprocs, done=0;
  float temp, tempdiff, mydiff = 0;
  myA ← malloc(…)
  initialize(myA);
  while (!done) do
    mydiff = 0;
    if (pid != 0)
      SEND(&myA[1,0], n, pid-1, ROW);
    if (pid != nprocs-1)
      SEND(&myA[nn,0], n, pid+1, ROW);
    if (pid != 0)
      RECEIVE(&myA[0,0], n, pid-1, ROW);
    if (pid != nprocs-1)
      RECEIVE(&myA[nn+1,0], n, pid+1, ROW);
```
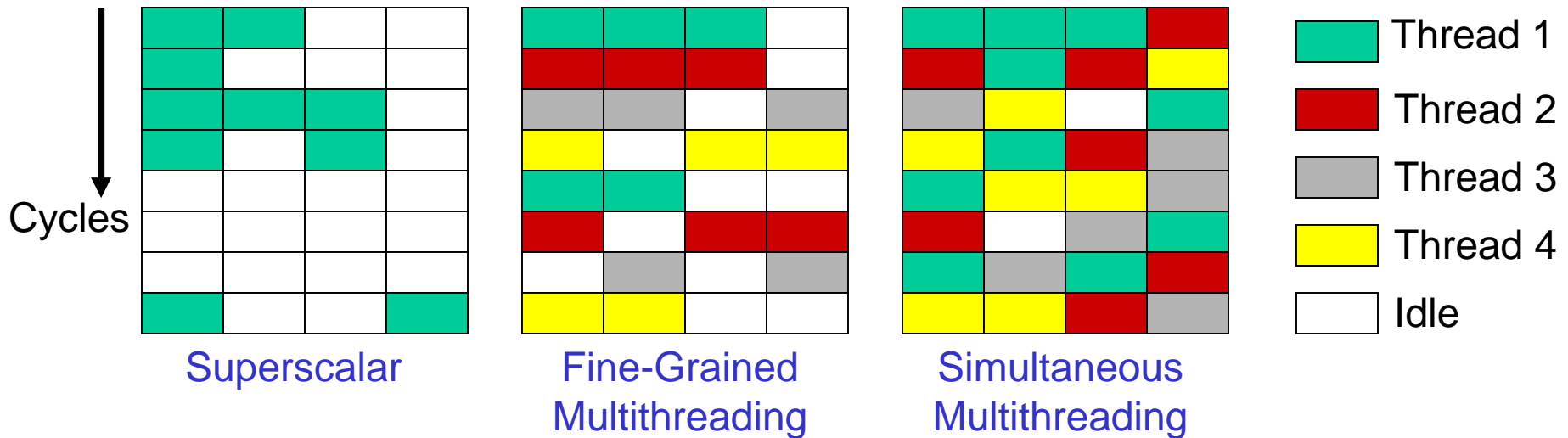
```
    for i ← 1 to nn do
      for j ← 1 to n do
        …
      endfor
    endfor
    if (pid != 0)
      SEND(mydiff, 1, 0, DIFF);
      RECEIVE(done, 1, 0, DONE);
    else
      for i ← 1 to nprocs-1 do
        RECEIVE(tempdiff, 1, *, DIFF);
        mydiff += tempdiff;
      endfor
      if  (mydiff < TOL)  done = 1;
      for i ← 1 to nprocs-1  do
        SEND(done, 1, I, DONE);
      endfor
    endif
  endwhile
```

11

Multithreading Within a Processor

- Until now, we have executed multiple threads of an application on different processors – can multiple threads execute concurrently on the same processor?

- Why is this desireable?
  - ➢ inexpensive – one CPU, no external interconnects
  - ➢ no remote or coherence misses (more capacity misses)

- Why does this make sense?
  - ➢ most processors can't find enough work – peak IPC is 6, average IPC is 1.5!
  - ➢ threads can share resources → we can increase threads without a corresponding linear increase in area

How are Resources Shared?

Each box represents an issue slot for a functional unit. Peak thruput is 4 IPC.

Cycles

Superscalar

Fine-Grained Multithreading

Simultaneous Multithreading

Thread 1
Thread 2
Thread 3
Thread 4
Idle

- Superscalar processor has high under-utilization – not enough work every cycle, especially when there is a cache miss
- Fine-grained multithreading can only issue instructions from a single thread in a cycle – can not find max work every cycle, but cache misses can be tolerated
- Simultaneous multithreading can issue instructions from any thread every cycle – has the highest probability of finding work for every issue slot

13

Performance Implications of SMT

- Single thread performance is likely to go down (caches, branch predictors, registers, etc. are shared) – this effect can be mitigated by trying to prioritize one thread

- With eight threads in a processor with many resources, SMT yields throughput improvements of roughly 2-4

Title

- Bullet