Topic	Supplementary Notes	Book Chapters
Introduction	lec1.pdf	Pacheco, GGKK, Quinn Chapter 1
[Performance] The only reason for parallelism	Quanitifying performance improvements analytically lec2_1.pdf, lec2_2.pdf, lec3.pdf	Pacheco Chapter 2.6, GGKK Chapter 5, Quinn Chapter 7
[Opportunities for parallelism/concurrency] Architecture, compiler and OS support	i. **Data level parallelism** with SIMD, Vector processors lec5.pdf, ii. **Instruction Level Parallelism** with compiler support lec6.pdf, iii. **Thread and process level parallelism** with shared and distributed memory address space multi-processors lec7.pdf, iv. Interconnection networks lec11.pdf	Pacheco Chapter 2.2-2.3, GGKK Chapter 2, 4, Quinn Chapter 2
[Writing parallel programs] C/C++ programming with compiler and library support	i. **OpenMP** Shared memory parallel programming lec4.pdf, ii. **MPI** Message Passing parallel programing lec8.pdf, lec12.pdf	Pacheco Chapter 5, GGKK Chapter 7.10, Quinn Chapter 17, Pacheco Chapter 3, GGKK Chapter 6, Quinn Chapter 4
[Probelms faced in parallelism/concurrency] Ensuring program correctness, correctness-vs-performance trade-offs	i. **Coherence:** Cache coherence and false sharing lec9_lec10.pdf, ii. **Synchronization:** Locks, barriers, transactional memory lec13.pdf, lec14.pdf, lec15.pdf, iii. **Consistency:** Memory consistency models iv. **Fault Tolerance** Additional issue of distributed systems	Hennessy 3.1-3.2, 4.1-4.3, 5.1-5.4
[Parallel program design] Algorithms and data structures		
Case studies		

What extra things are happening in parallel programs compared to sequential programs?

- Two or more threads change the same variable. We need to enforce mutual exclusion.
- Read-compare-store or other code listings, expected to be atomic, might not be so. We need to enforce atomicity.
- What about order of operations? Can there be non-intuitive instruction interleaving?

We need to understand these possible issues to reduce programming bugs.
Bugs might be hard to reproduce and therefore harder to debug.

Sequential code: possibility of compiler reordering

```
X = flag 1

flag1 = 1

if(flag2 == 0)

    printf(“flag2 is 0”)
```

Parallel code: what outputs can a programmer expect?

```
P1:

flag1 = 1

if(flag2 == 0)

    printf(“P1 wins”)
```

```
P2:

flag2 = 1

if(flag1 == 0)

    printf(“P2 wins”)
```

Definition: Shared Memory Consistency Model

The memory consistency model of a shared memory multiprocessor provides a formal specification of how the memory system will appear to the programmer, eliminating the gap between the behavior expected by the programmer and the actual behavior supported by the system.

Definition: Shared Memory Consistency Model

The memory consistency model of a shared memory multiprocessor provides a formal specification of how the memory system will appear to the programmer, eliminating the gap between the behavior expected by the programmer and the actual behavior supported by the system.

- A set of rules governing how the memory system will process memory operations from multiple processors
- Contract between the programmers and the system
- Determines what optimizations can be performed for correct programs

Sequential consistency model

A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operation of each individual processor appear in this sequence in the order specified by its program.

```
P1:

flag1 = 1

if(flag2 == 0)

   printf(“P1 wins”)
```

```
P2:

flag2 = 1

if(flag1 == 0)

   printf(“P2 wins”)
```

Sequential consistency model [too restrictive]

A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operation of each individual processor appear in this sequence in the order specified by its program.

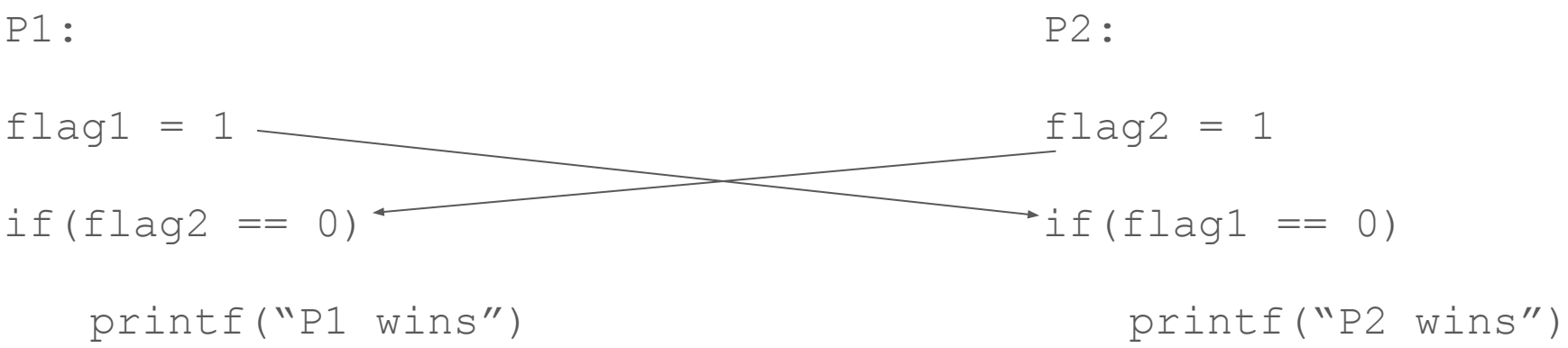
```
P1:

flag1 = 1

if(flag2 == 0)

   printf("P1 wins")
```

```
P2:

flag2 = 1

if(flag1 == 0)

   printf("P2 wins")
```

Both cannot win in the sequential consistency model. Compiler reordering will be prohibited.

Sequential consistency model [too restrictive]

A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operation of each individual processor appear in this sequence in the order specified by its program.



Effect of each instruction immediately visible in all processors. Use of write buffers will be prohibited.

Sequential consistency model

A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operation of each individual processor appear in this sequence in the order specified by its program.

```
P1:

for (i = 0; i < N; i++)

    arr[i] = 0
```

```
P2:

printf("%d",arr[1])

printf("%d", i)
```

Sequential consistency model [too restrictive]

A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operation of each individual processor appear in this sequence in the order specified by its program.

```
P1:

for (i = 0; i < N; i++)

    arr[i] = 0
```

```
P2:

printf("%d",arr[1])

printf("%d", i)
```

Compiler will be restricted from using register variables.

Relaxed consistency model - weak ordering

In this model, the memory operations are divided into two categories,

data operations and **synchronization operations**.

Relaxed consistency model - weak ordering

In this model, the memory operations are divided into two categories,

data operations and **synchronization operations**.

- Intuition: Reordering memory operations in data regions between synchronization operations does not typically affect program correctness.
- Synchronization operations enforce program order by disallowing reordering of code around them
- Temporary view is maintained between synchronization operations

OpenMP synchronization operation

- flush() is the key synchronization operation
  - `#pragma omp flush(list)`
- Prevents reordering of memory accesses across flush

Decker’s algorithm for critical section

```
P1:

flag1 = 1

if(flag2 == 0)

    Critical Section
```

```
P2:

flag2 = 1

if(flag1 == 0)

    Critical Section
```

Decker's algorithm for critical section: Incorrect flushes

```
P1:

flag1 = 1

flush(flag1)

flush(flag2)

if(flag2 == 0)

    Critical Section
```

```
P2:

flag2 = 1

flush(flag2)

flush(flag1)

if(flag1 == 0)

    Critical Section
```

Decker's algorithm for critical section: Correct flushes

```
P1:

flag1 = 1

flush(flag1, flag2)

if(flag2 == 0)

    Critical Section
```

```
P2:

flag2 = 1

flush(flag1, flag2)

if(flag1 == 0)

    Critical Section
```

OpenMP synchronization operation

- flush() is the key synchronization operation
  - `#pragma omp flush(list)`
- Prevents reordering of memory accesses across flush
- Implicit flushes
  - Barriers
  - Entry/Exit from parallel, parallel for, critical
  - Lock functions
  - Entry and exit from atomic (only variables which are updated)

OpenMP consistency model: Release consistency

- Further relaxation of weak consistency
- Synchronization operations are further divided
  - Acquire: operations like lock
  - Release: operations like unlock
- Acquire:
  - must complete before all following memory accesses
- Release:
  - All memory access operations before release must complete
  - Accesses after release in program order need not wait for release

Types of concurrency bugs in presence of compiler reordering, OS scheduling, coherence protocols

- Order violation
- Atomicity violation
- Sequential consistency violation
- Deadlock
- Starvation
- Livelock