Chapter 19 – Parallel Processing Part II

CPSC 240-09
John Overton

Review

Parallel Processing

- Moore's Law:

Gordon Moore, then co-founder of Fairchild Semiconductor and future CEO of Intel, said in a 1965 paper that the number of components on an integrated circuit would double every year. In 1975, he revised his prediction to doubling every two years. This prediction has proved very accurate for about 37 years until it slowed in 2012.

- Effectively, with a slowing of Moore's law, the increase in the capability of new processors (in the form of integrated circuits) has also slowed.
- So, we have reached close to the maximum clock rate on a single processor, therefore we must use multiple processors to distribute the work

Parallel Processing

- Parallel processing is also referred to as concurrency
- Concurrency can refer to either two (or more) programs (or sets of code) executing on separate processors or two (or more) programs executing interleaved, on one processor as if to appear as if two programs were executing on two different processors
- Parallel processing has a connotation that implies processes executing simultaneously, working together to solve a single complex problem

Parallel Processing

- The book talks about two approaches to parallel processing:
  - Multiprocessing is executing tasks on multiple processors, usually in the same computer, to solve a single problem
  - Distributed computing is executing tasks on different computers to solve a single problem (Book example: Folding@home)

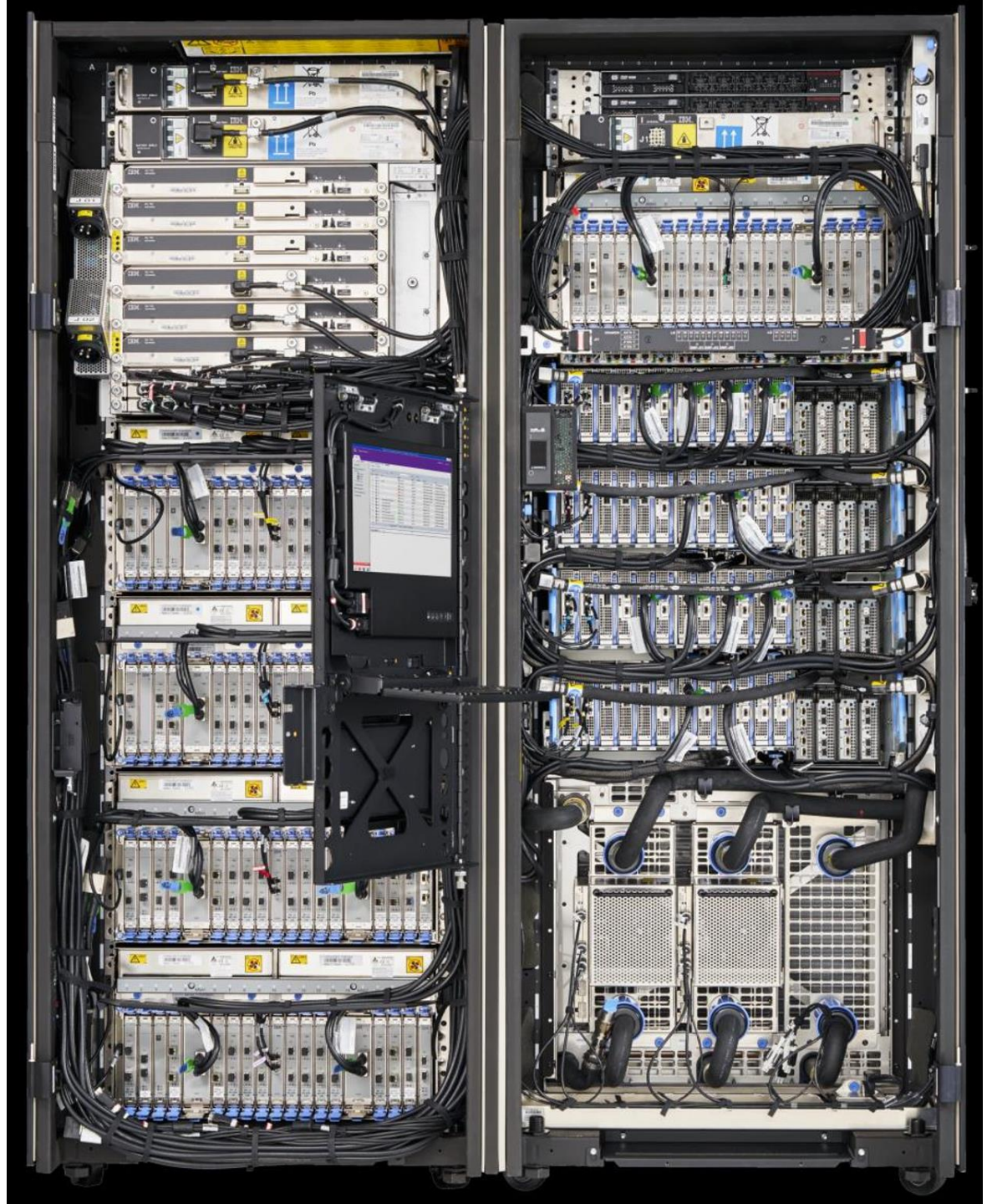
Parallel Processing

- Dividing a programming problem into multiple tasks is quite difficult
- However, if we can break a problem down into sub-problems and if these sub-problems could be executed at the same time, then we would have a total solution that could execute significantly faster using multiple processors than if a single “monolithic” serial-executing solution was running on a single processor
- Not all problems can be divided up and run in parallel

Parallel Processing

- In a multithreaded assembly program, each thread would have their own registers
- Each thread would have its own stack – that is, each thread would have stack space allocated for them from the same shared address space
- All threads share the same .data, .text and .bss section (the share the same address space). This can be a problem if multiple threads were updating a variable in memory at the same time
- However, we can use the stack to allocate “local” variables used by a single thread

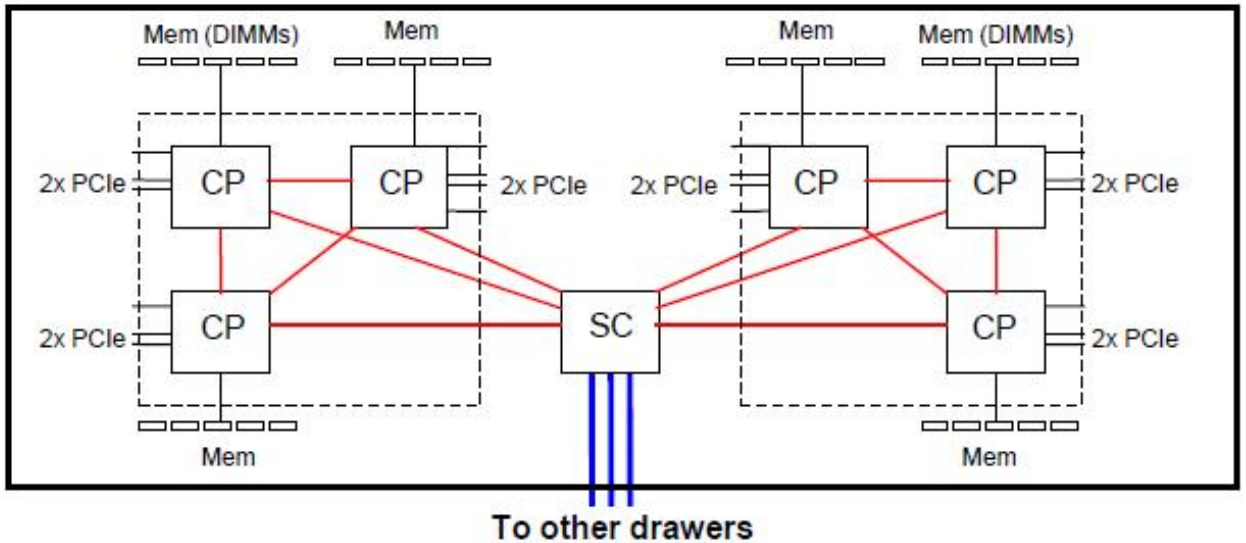
Parallel Processing Case Study
IBM z14 – The modern mainframe

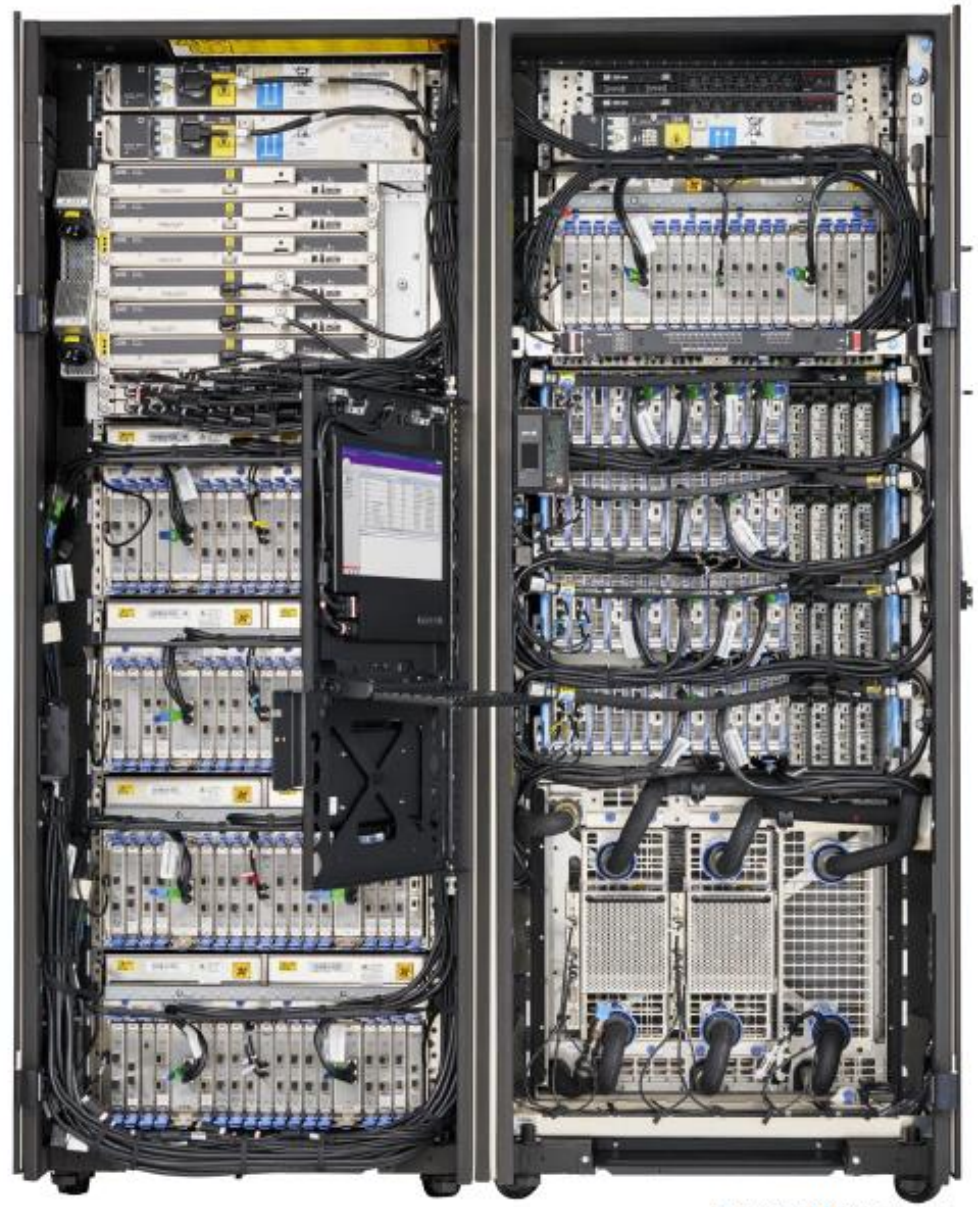


Parallel Processing - Case: IBM z14 – The modern mainframe

- A two rack system
- Contains 4 drawers for processors –
  - Each drawer has up to 6 CP chips and one SC chip
  - Each CP chip contains up to 10 processors (240 total processors)
  - Out of the 240, 170 are used as main processors running at 5.2gHz (with spares)
  - Each CP chip has 6.1B transistors, 14 miles of wire internally
  - An SC chip is responsible for communicating between drawers and I/O channels and also has a L4 (level 4) cache
  - Each SC chip has 9.7B transistors, 13.5 miles of wire internally
- Up to 32 TB of DRAM main memory
- Up to 85 LPARs (virtual machines)

z14 On-Drawer and System Topology





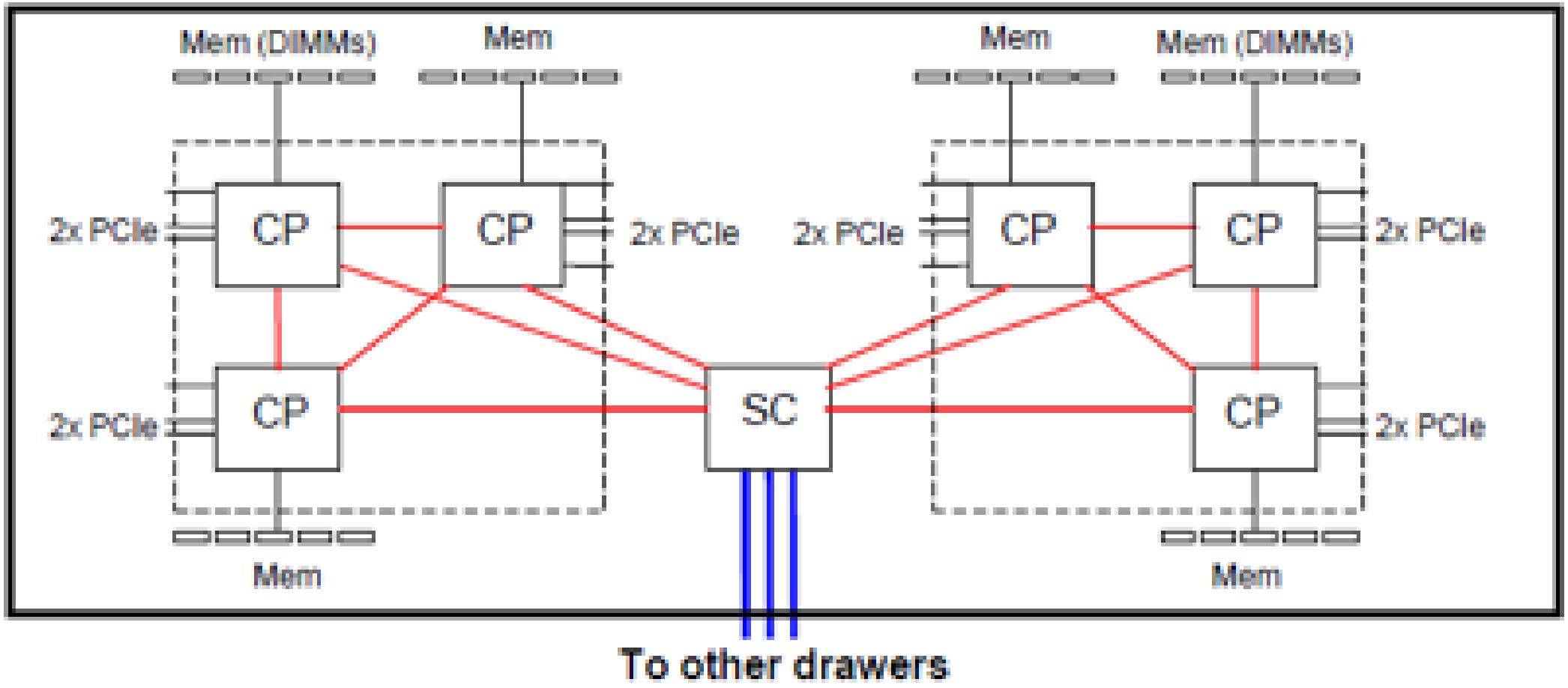
CP chip, 696 sqmm, 14nm, 17 layers of metal
- 10 cores, each 2+4MB I+D L2 cache
- Shared 128MB L3 cache

SC chip, 696 sqmm, 14nm, 17 layers of metal
- System interconnect & coherency logic
- Shared 672MB L4 cache

Max System:
- 24 CP sockets in SMP interconnect
- 32TB RAIM-protected memory
- 40 PCI gen3x16 fanouts to IO-drawers
- 320 IO cards

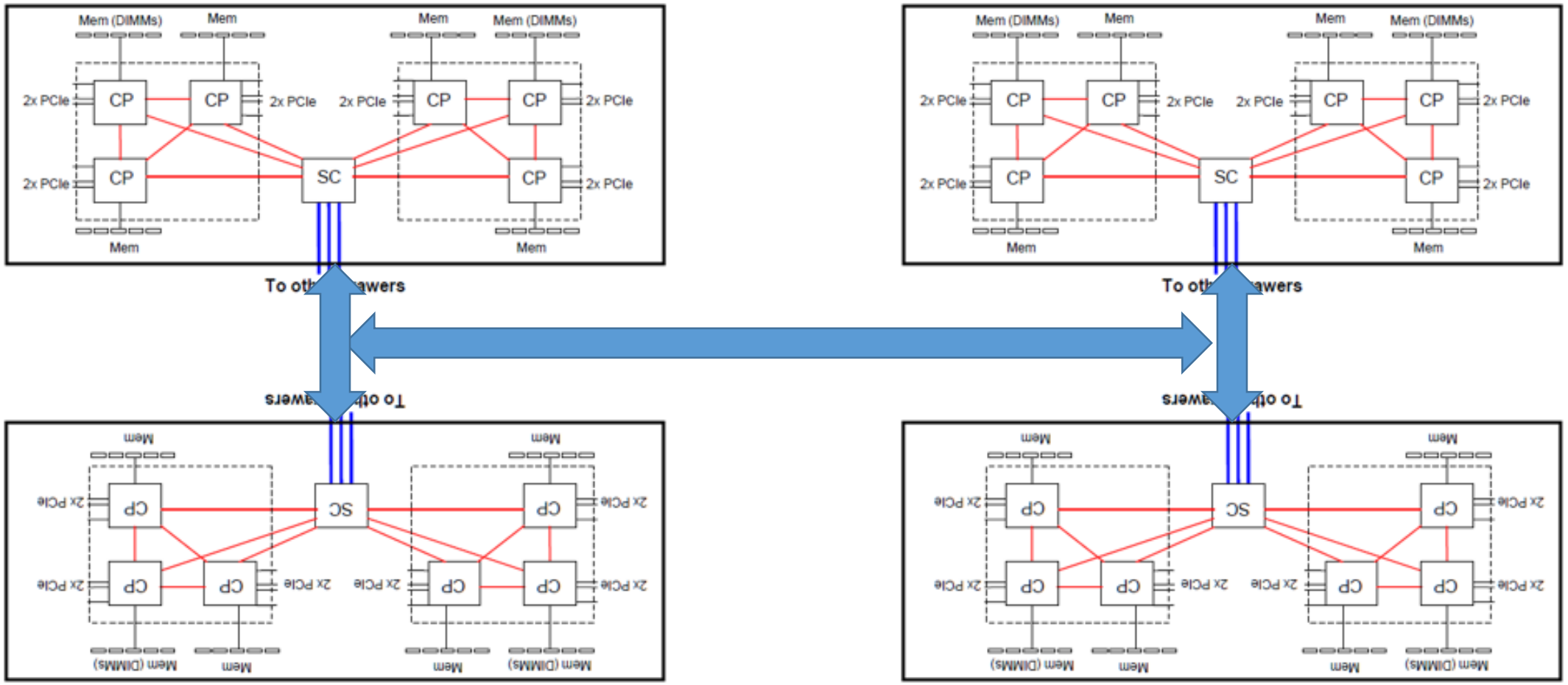
Parallel Processing - Case: IBM z14 – The modern mainframe

NUMA – Non-Uniform Memory Access



Parallel Processing - Case: IBM z14 – The modern mainframe

NUMA – Non-Uniform Memory Access



Parallel Processing – Processes and Threads

- How operating systems keep track of logical units of work: Processes and Threads
  - A process is a program executing in its own “address space” (early definition)
  - A thread is program executing in an existing “address space”
  - In some Oses, a process has a single thread until more threads are created
  - A thread is often referred to as a light-weight process since it will use the initial process’ allocations and address space, thus making it quicker to start (create)
- A threaded approach will not scale to the extent of the distributed approach because a single computer has a finite number of processors and memory, which place an upper bound on the number of simultaneous computations
- However, the distributed solution is bound by network communication (network speed/bandwidth)

Parallel Processing - pthreads

- pThreads is short for POSIX Threads
- pThreads is a thread library that works with the operating system to create and manage threads
- We will use pthread_create(), pthread_join() in our project from the C library
- pthread_create() is a function that will create a new thread and pass a parameter to the new thread. It will return the thread id of the newly created thread
- pthread_join() is a function to wait until a specific thread has ended. Can be used by the main thread to wait on a created thread

Race conditions

- When multiple threads simultaneously write to the same location at the same time
- NOT GOOD!
- Example from book:

$$\sum_{i=0}^{MAX-1} myValue = \left(\frac{myValue}{X}\right) + Y$$

```
for (int i=0; i < MAX; i++)
    myValue = (myValue / X) + Y;
```

Race conditions – Example from Book

```
MAX     equ     1000000000

; Perform MAX / 2 iterations to update myValue
        mov     rcx, MAX
        shr     rcx, 1                  ; divide by 2
        mov     r10, qword [x]
        mov     r11, qword [y]

incLoop0:       ; myValue = (myValue / x) + y
        mov     rax, qword [myValue]
        cqo
        div     r10
        add     rax, r11
        mov     qword [myValue], rax
        loop    incLoop0
        ret

        section .data
myValue dq      0
x       dq      1
y       dq      1
```

Race conditions – Example from Book

Since each thread is independent of other threads, chances are they may be doing different things at different times and could save a value to a variable at an inconvenient time

Step	Code: Core 0, Thread 0	Code: Core 1, Thread 1
1	`mov rax, qword [myValue]`	
2	`cqo`	`mov rax, qword [myValue]`
3	`div qword [x]`	`cqo`
4	`add rax, qword [y]`	`div qword [x]`
5	`mov qword [myValue], rax`	`add rax, qword [y]`
6		`mov qword [myValue], rax`

Ways to safeguard resources : Mutex lock and unlock

- A mutex (mutual exclusion) is a way to synchronize access to a resource (a variable or a data structure)
- A mutex is also called a lock: A thread can lock and unlock – when locked, it means one thread has obtained the right to access the resource
- Lock contention is when one or more threads request (attempt) to acquire a lock but another thread has already acquired the lock
- A deadlock is a case where thread A has acquired lock 1 and wants to acquire lock 2, which is already acquired by thread B, and thread B wants to acquire lock 1, already acquired by thread A. Both tasks are waiting for the other task's owned resource
- pthread_mutex_lock()
- Pthread_mutex_unlock()

Ways to safeguard resources : semaphore

- A semaphore is an integer whose value is never allowed to fall below zero.
- A semaphore is set to some initial number to indicate the number of resources it is to protect
- Then, two operations can be used by threads that need access to one or more of the set of resources
  - sem_wait() is called by a thread to request access to resources. If the semaphore is already zero, the thread waits until it is not zero
  - sem_post() is called by a thread to release access to the resources. It increments the semaphore so that other threads can use the resources. If there was a waiting thread, then it woken up, the semaphore is decremented and the thread has access to the resources
- A semaphore can be initialized to 1 – the semaphore would work like a mutex

Ways to safeguard resources – Underlying architecture

- We need an atomic operation
  - An atomic operation means, do not interrupt me while I do this (sequence of) operation(s)
  - Or, if you need to interrupt me, act as if it never got started
- Two major architectural solutions: compare-and-swap and test-and-set
  - Compare-and-swap compares a value in memory and if it is equal, swap it with another value
  - Test-and-set makes a note of the address of a variable, then does a “test” on the value of the variable and if true, sets the variable to a new value. The note of the address is used to cause any other processor to wait if they were also trying to test-and-set using the address of the variable
- What does the x86 have? xchg and the lock prefix

Using local storage

- To be continued…

Assignment – Phase 2

In phase 1, you wrote a program that creates only one thread and runs a subroutine that prints “Hello World” to the console.

In phase 2, you will create multiple threads to print “Hello World”

Create a constant equate at the top of your program that defines the number of threads to create:

```
MAX_THREADS    equ    2
```

Create an array to store the thread ids for each of the threads created. Pass this index to the thread so that it can put the index number in the message, for example, print (use printf() if you want to print out the message):

Hello World from thread n

Use separate for loops for calling pthread_create() and pthread_join()

For phase 2, you will need to refresh how to write a for loop…

```
For ( initializers; loop-tests; other-expressions ) {
        statement(s)
}
```

```
initializers here (like mov dword [i], 10)
loop:
        Do loop-test here
        …
        statement(s)
        …
        other-expressions
        jmp    loop
```

```
for ( int i = 0; i < MAX_THREADS; i++ ) {
        statement(s)
}
```

```
MAX_THREADS   equ   10

        mov   qword [i], 0
loop:
        cmp    qword [i], MAX_THREADS
        jge    loop_end
        …
        statement(s)
        …
        inc     qword [i]
        jmp     loop
loop_end:
        section .data
i       dq      0
```