Shared Memory

SMP Architectures and Programming

Numbers game…

- It is important to understand the basic constants we are working with in high performance computing
- Amdahls law
  - improvements obtained by increasing speed of a component are limited by the fraction of time spent on that component

How fast is a CPU?

- Intel 2.8-3.8 GHz
  - AMD a little lower
- Intel Itanium 2.0-2.4GHz
- IBM 2.2-3.2 GHz
- IBM CELL – 3.2
- SUN 1.0-2.0 GHz

What is a typical CPI?

- Intel 0.3
  - AMD a little lower
- Intel Itanium 0.18
- IBM 0.25 GHz
- IBM CELL 0.07
- SUN 0.25-0.07

Cluster Computing

How fast is memory

- Main Memory
  - 30 ns
- L1 cache
  - 1 ns
- L2 cache
  - 3-6 ns

How fast is the memory bus

- 400-800 MHz
- 1.2-1.6 GHz in new technology

Cluster Computing

How much time to read from memory?

- 3-6 ns to establish L2 miss
- 1.25 ns to get bus slot
- 30 ns to lookup in main memory
- 1.25 ns to get bus slot

- Total: 35.5-38.5ns

Cluster Computing

How long are pipelines?

- Intel 31
- AMD 17
- Intel Itanium 10
- IBM Power 5 16
- IBM CELL 16 and 7
- SUN Ultrasparc 9

How much time to do an interrupt?

- 5-100 cycles at the CPU
- Easily microseconds on the chipset

How long to do a system call?

- 5 cycles to almost a microsecond

Numbers summary

- An instruction is in the order of 0.1 ns
- L1 access is as much as 10 instructions
- L2 access is as much as 60 instructions
- Memory access is as much as 385 instructions
- Interrupts are easily 10000 instructions

Why work with shared memory parallel programming?

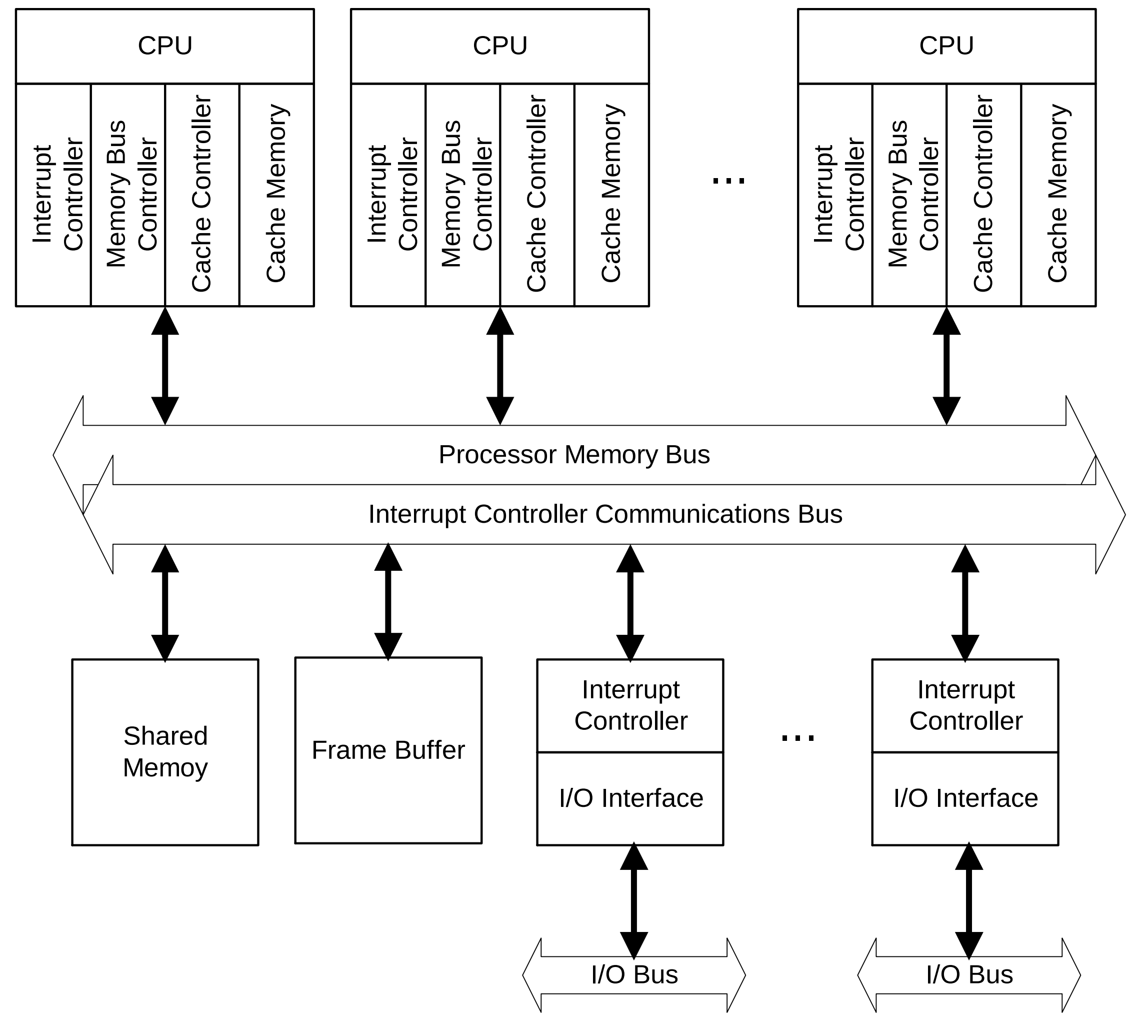
- Speed
- Ease of use
- CLUMPS
- Good starting point

SHMP – a quick refresh

- Shared bus
  - Rather simple
  - Very cheap
  - Only scales to a few processors
  - Maintains the standard memory view

Cluster Computing

Shared Bus

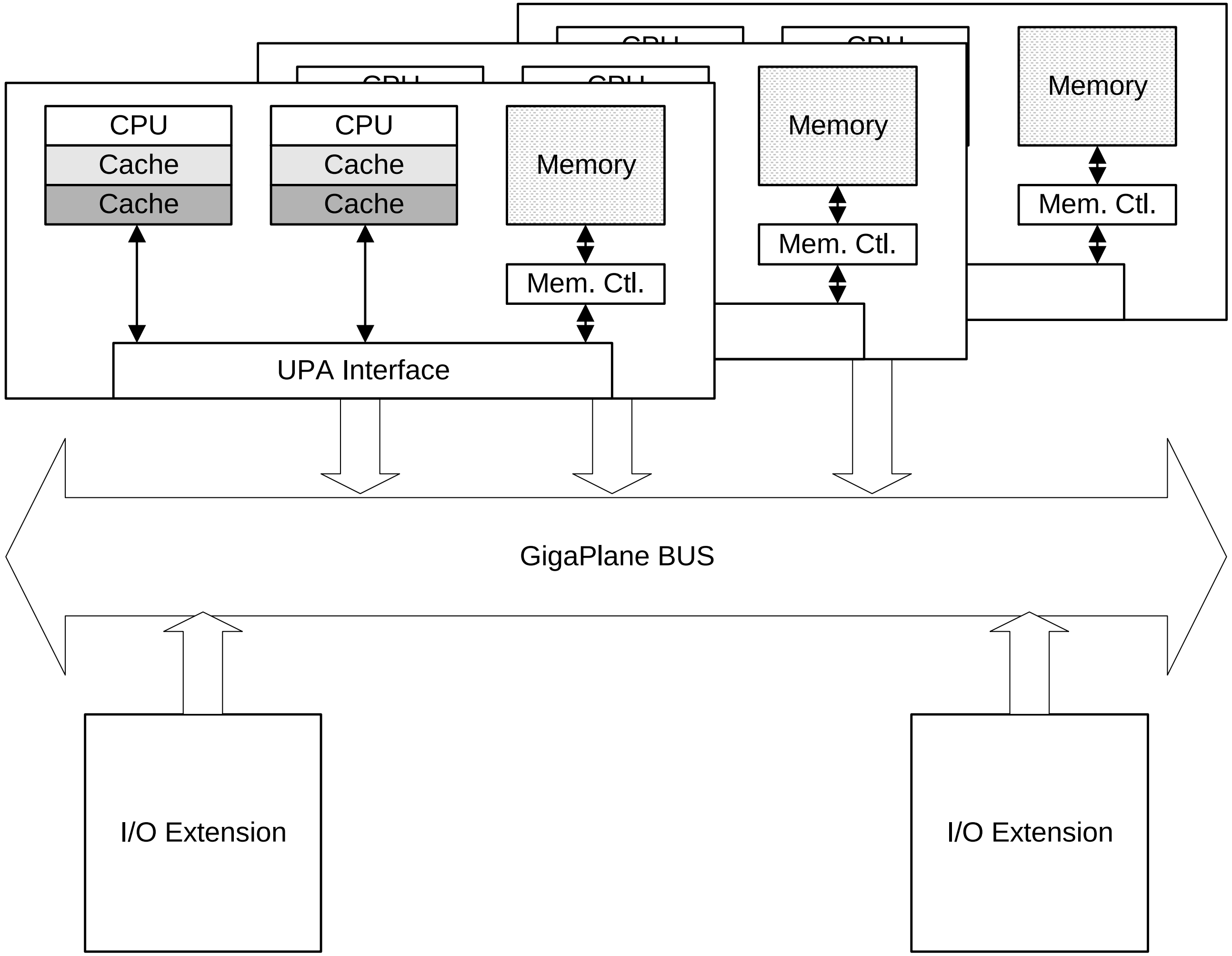


SHMP – a quick refresh

- Crossbar switched
  - Rather complex
  - Quite expensive
  - Can scale to tens of processors
  - Needs a relaxed memory consistency protocol

Cluster Computing

Crossbar switch

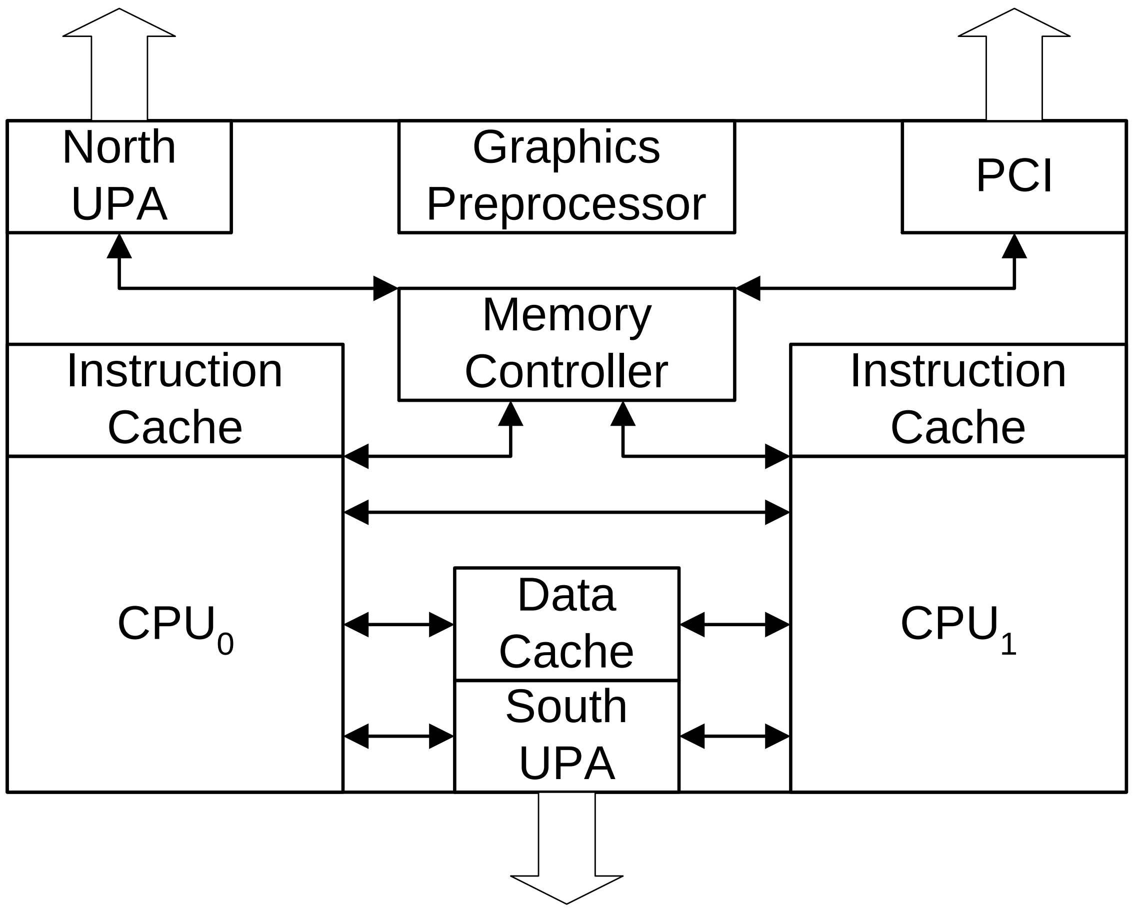


SHMP – a quick refresh

- MP on a chip
  - Extremely simple
  - Extremely cheap
  - Only very few processors per chip (read two)
  - Allows the CPUs to work together more closely

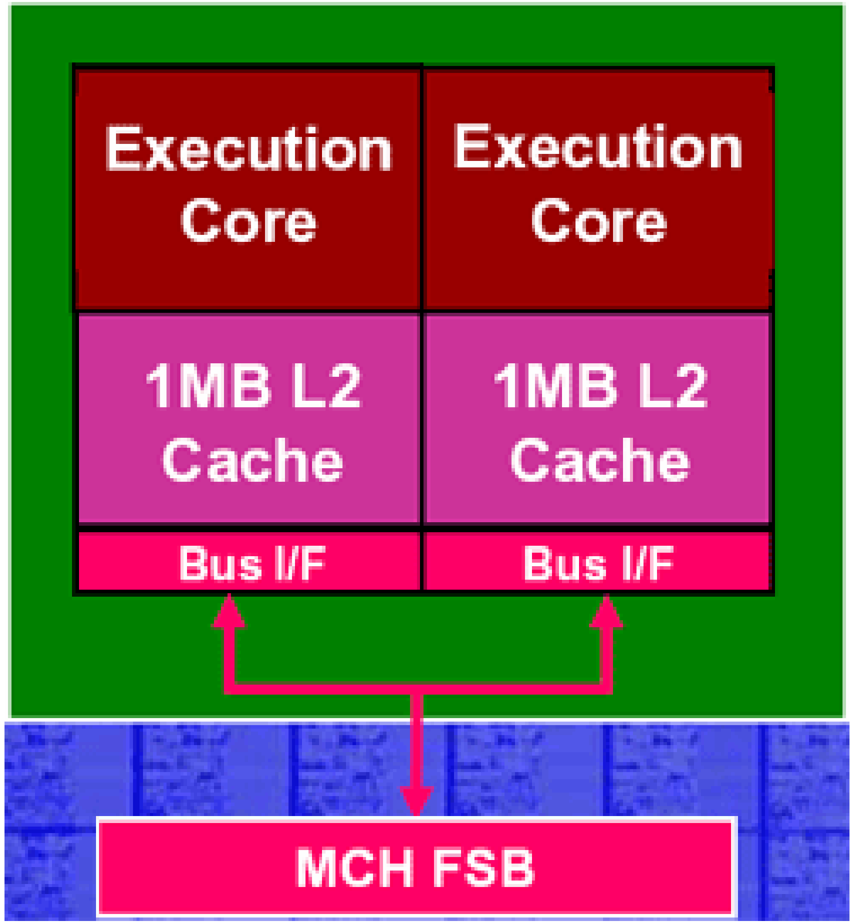
Cluster Computing

MP on a chip

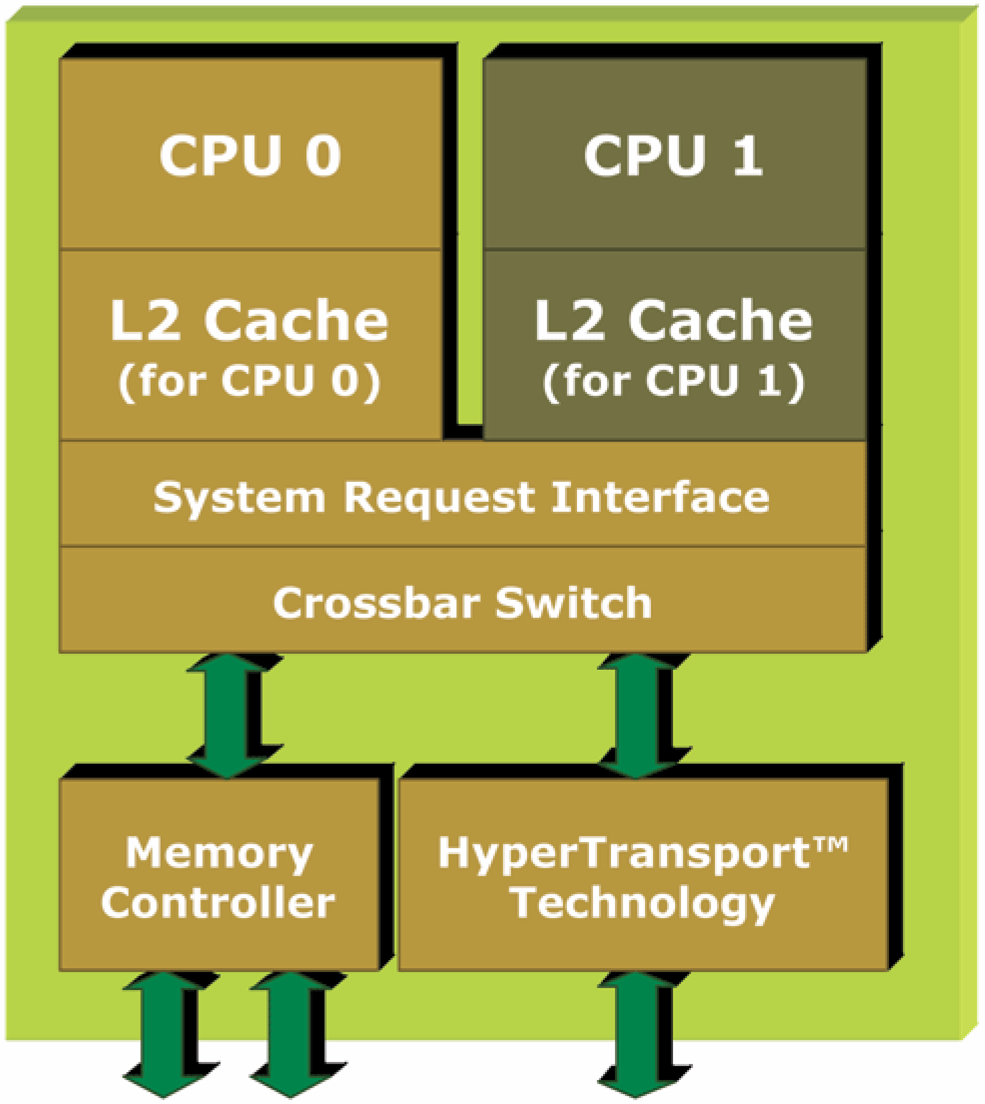


Cluster Computing

Intel dual Core

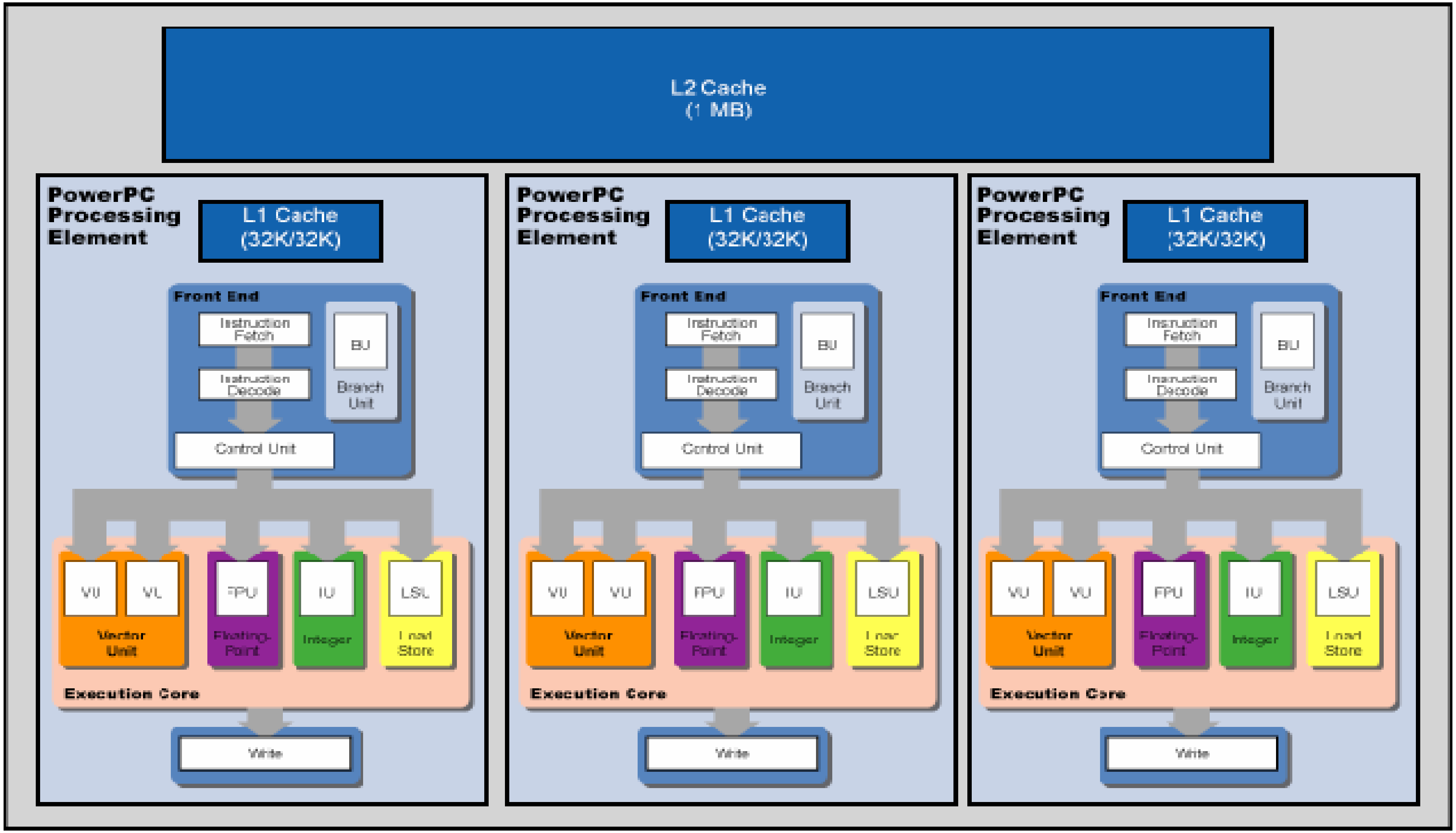


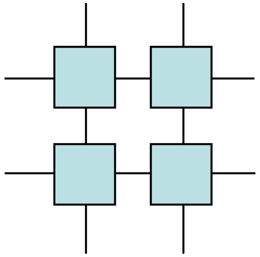
AMD dual core



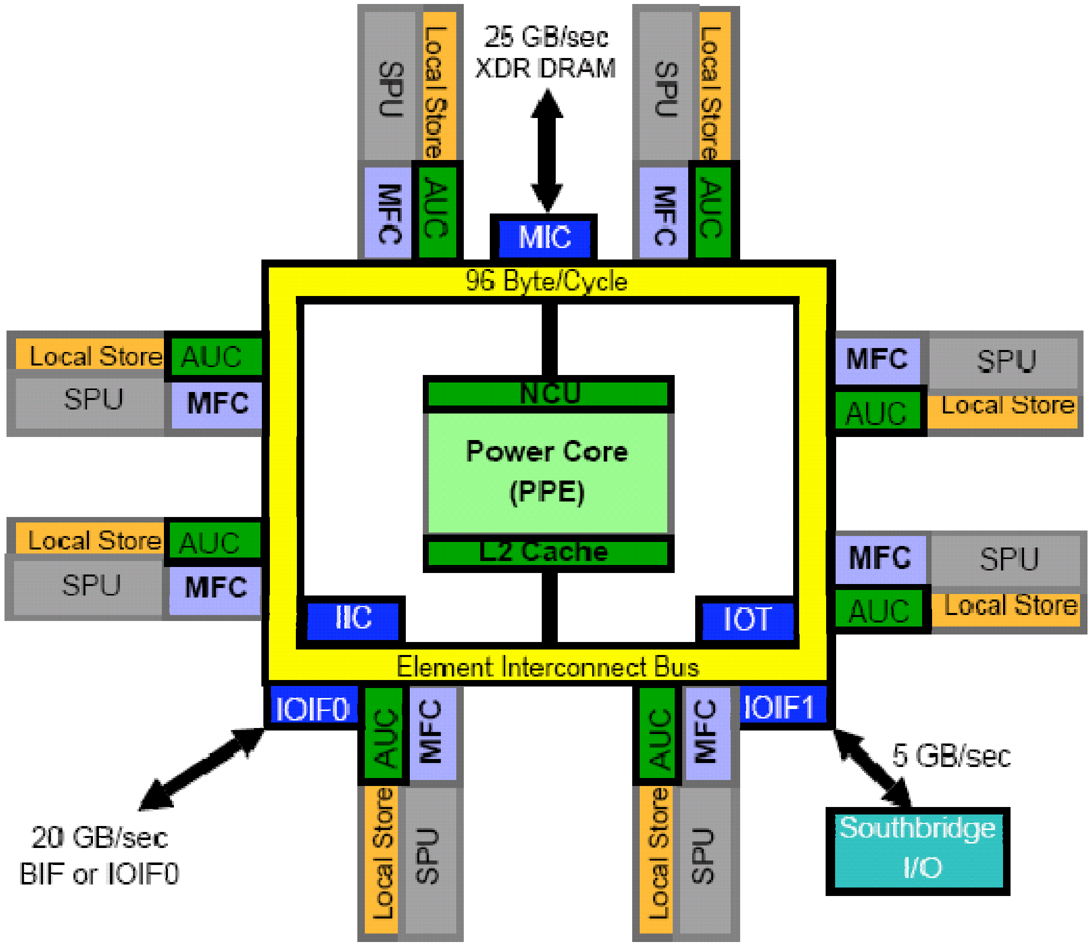
Cluster Computing

Xenon



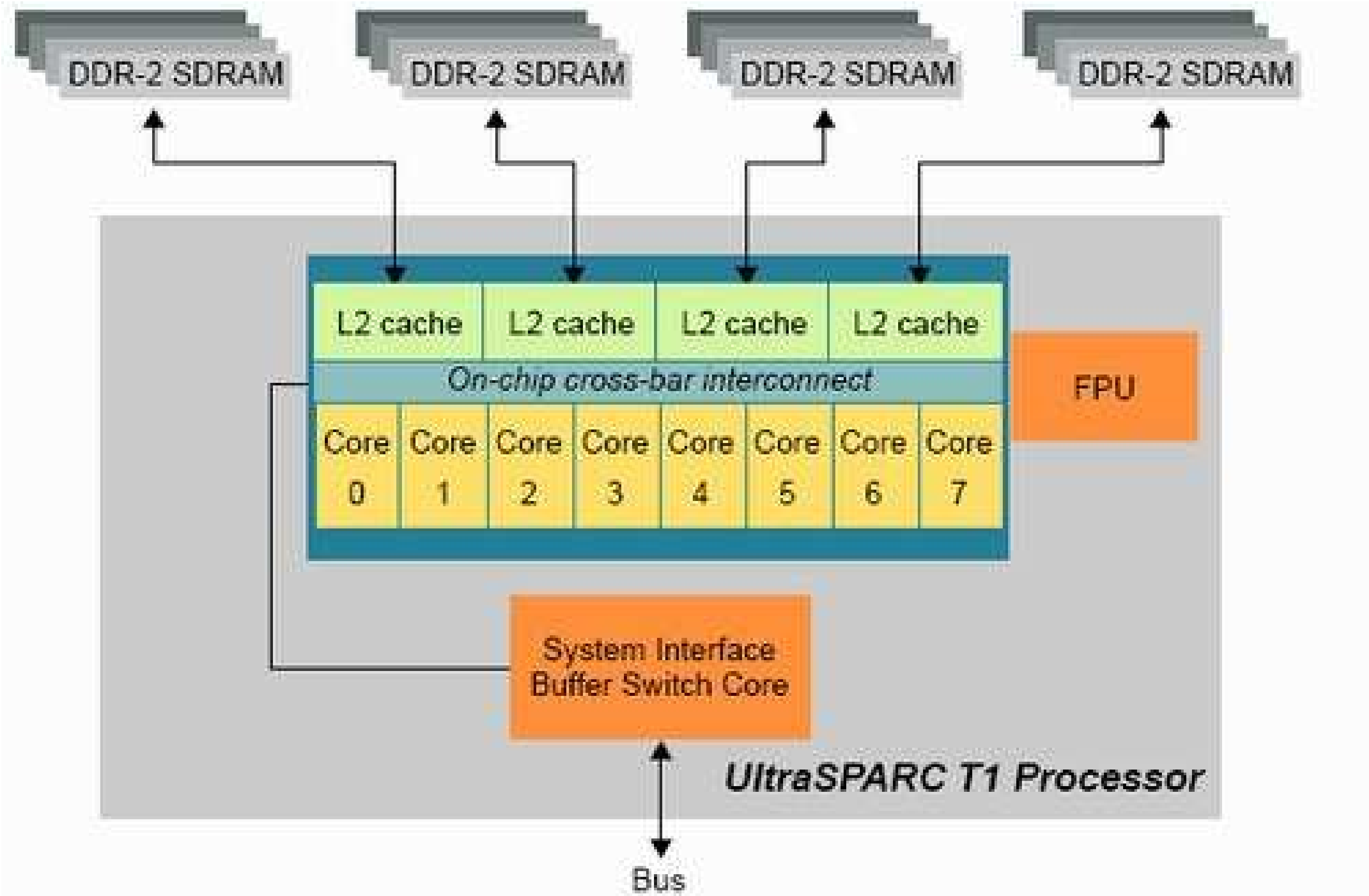

Cluster Computing

CELL


25 GB/sec
XDR DRAM
MIC
96 Byte/Cycle
NCU
Power Core
(PPE)
L2 Cache
IIC
IOT
Element Interconnect Bus
IOIF0
IOIF1
20 GB/sec
BIF or IOIF0
5 GB/sec
Southbridge
I/O
SPU
Local Store
MFC
AUC

Cluster Computing

Ultrasparc T1



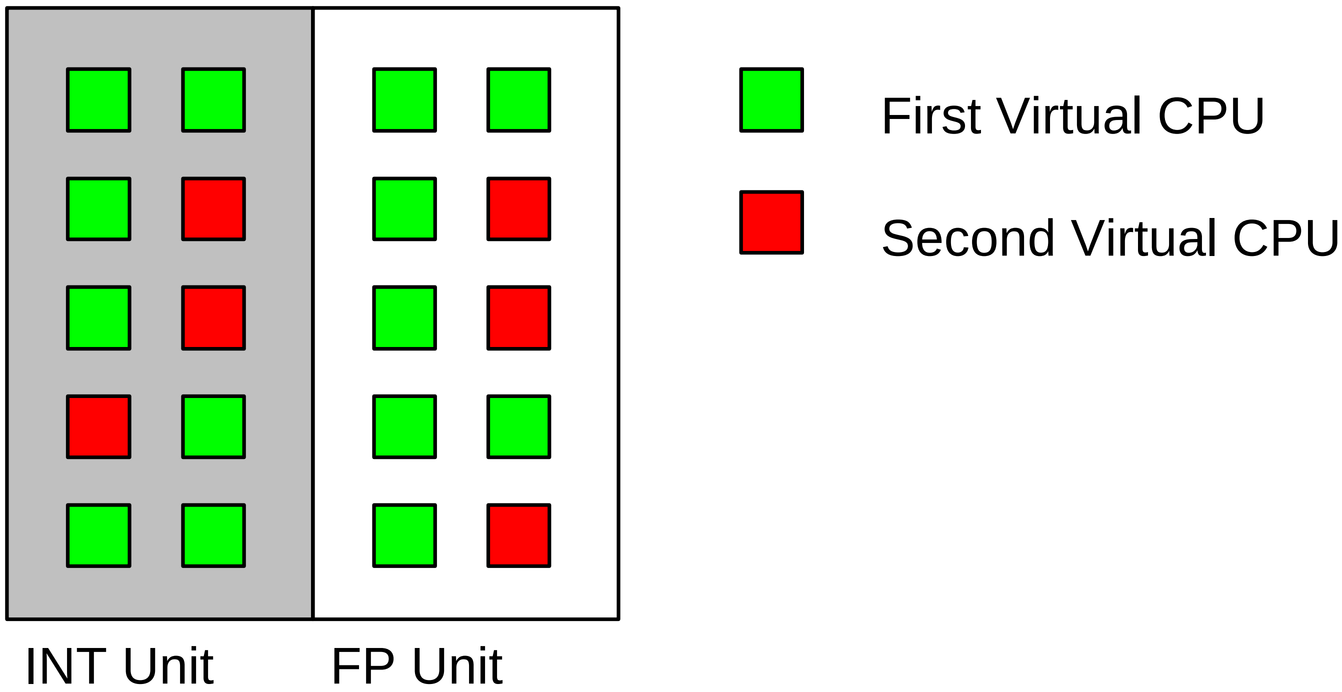
SHMP – a quick refresh

- Virtual MP on a chip
  - Named Hyper-threading
  - Extremely cheap – only an extra register-file per VP and some control logic
  - Virtual depth can be quite large but few applications may take advantage of it
  - Allows us much better utilization of the CPU area

Hyperthreading

- Hardware threads shifts are activated either on cache miss or every cycle
- Cache-miss activated yielding addresses the idea behind HT directly
- The every-cycle approach is simple and requires less overhead

MP on a chip

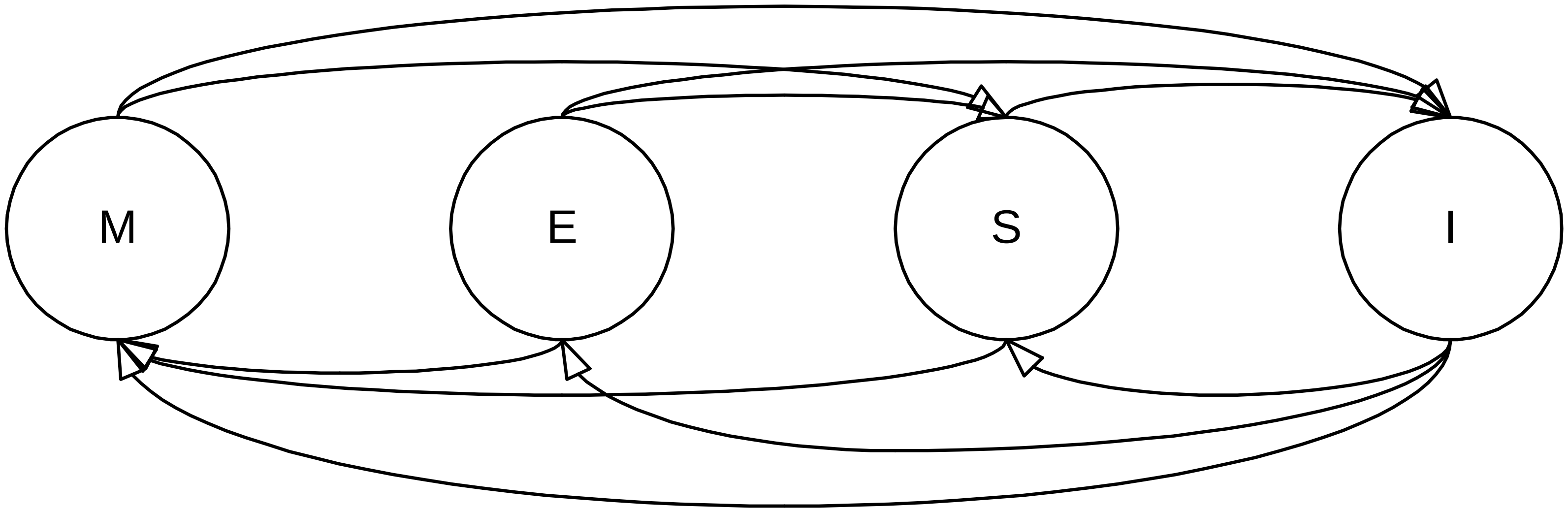


The MESI Protocol

- Common protocol for ensuring sequential consistency
- States are
  - Modified
  - Exclusive
  - Shared
  - Invalid

Cluster Computing

MESI Protocol

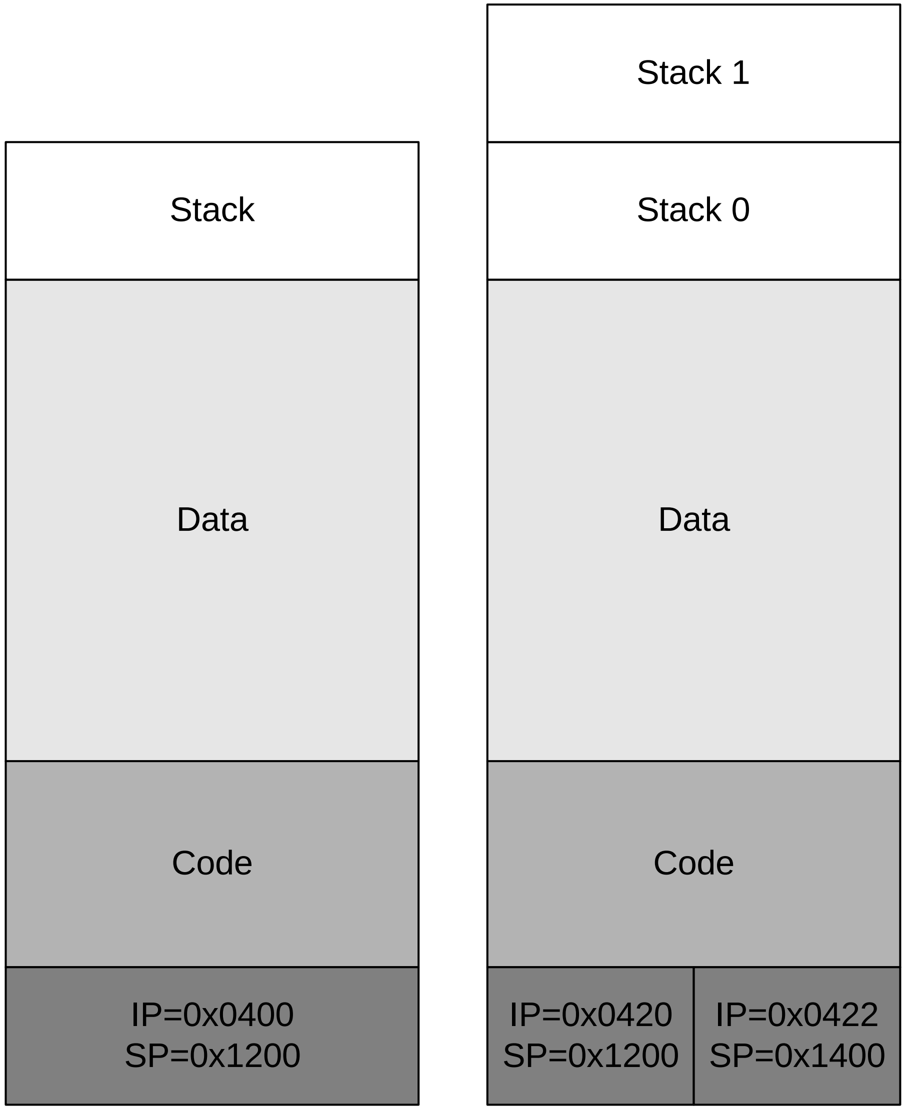


Processes and Threads

- Threads are often referred to as lightweight processes
- A thread is simply a process which shares the address space of the process it resides in with the other threads in that process

Cluster Computing

Processes and threads



Cluster Computing

Thread types

- User level
- Kernel level
- Mixes

Thread Packages

- POSIX Threads
- Solaris Threads
- Java Threads
- + 10^6 custom packages

Types of Threads

- Non-preemptive
- Preemptive
- User level
- Kernel level
- Mixed

User Level Threads

- Non-preemptive switching is fast, Preemptive is slow
- Creating a new thread is fast as is destroying a thread
- Unable to utilize more than one processor

Kernel Level threads

- Preemptive switching is (relatively) fast, Non-preemptive is (relatively) slow
- Creating and destroying threads is slow
- Can utilize more than one processor

Mixed (or both)

- Best of both worlds (BOB)
  - All the advantages of user-level threads combined with MP support
- May introduce a new level of threading

Thread Packages

- Java Threads
- POSIX threads
- Solaris threads
- WIN32 threads

Java Threads

- Integrated into the language

```
class dummyThread extends Thread {
  int id;
  public dummyThread(int id){this.id=id;}
  public void run(){
    System.out.println(“Hello World from thread “+id);
  }
}

dummyThread dt = new dummyThread(42);
dt.start();
dt.join();
```

POSIX Threads

- Language independent library

```
pthread_create(&thread, NULL, worker,(void *)job);
pthread_join(thread);
```

Solaris Threads

- Similar to POSIX however a thread is called a Lightweight process (LWP)
- Introduces a new level of threading on top of LWPs called threads
- LWP are kernel level
- Threads are user level

Cluster Computing

WIN32 Threads

- API is designed to match the rest of the WIN32 API
- Introduces a second level of threading called fibers
- Threads are kernel level
- Fibers are user-level – and non-preemptive

Programming with threads

- Divide your application onto different tasks
  - One task per functionality
  - One task per data block
- Create the threads
- Perform the necessary control over the threads

Thread Control

- Critical regions
- Signal/Wait
- Barriers
- Monitors

Critical regions

- Critical regions are code portions that access data which may be accessed concurrently by another thread
- Unfortunate notation
  - The critical region is really in data
  - But the guards are in code

Critical Region

do {

entry region

critical region

leave region

remainder

} **while (1)**;

Mutex mechanism

- The mechanism that performs this check is called a mutex.
- A mutex has two states, that are usually referred to as **unlocked** and **locked**:
  - **unlocked** mutex indicates that the critical region is empty
  - **locked** mutex indicates that there is some thread inside the critical region.

Mutex – how it works

A thread that wishes to access a resource checks the mutex associated with that resource:

- If the mutex is unlocked, it means there is no thread in the critical section:
  - The thread locks the mutex and enters the critical section.
  - When the thread leaves the critical section it should unlock the mutex.
- If the mutex is locked, it means that there is another thread in the critical section:
  - the thread (that is trying to lock the mutex and enter) waits until the mutex becomes unlocked.

Mutex states

- There are two operations defined on a mutex (beside initializing and destroying):
  - **Lock**: checks the state of the mutex
    - locks the mutex if it is unlocked
    - waits until it becomes unlocked.
  - **Unlock**: unlocks the mutex
    - allows any one waiting thread to lock the mutex

Defining and initializing a mutex

A mutex is defined with the type p*thread_mutex_t*, and it needs to be assigned the initial value: *PTHREAD_MUTEX_INITIALIZER*

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

Lock

- A mutex is locked with the function: **pthread_mutex_lock(pthread_mutex_t *mutex)**
- This function gets a pointer to the mutex it is trying to lock.
- The function returns when the mutex is locked, or if an error occurred
  - a locked mutex is not an error, if a mutex is locked the function waits until it is unlocked.

Trylock

- A mutex lock attempt can be made with the function: **pthread_mutex_trylock(pthread_mutex_t *mutex)**
- The function returns true if the mutex is locked
  - false otherwise

Unlocking a mutex

- A mutex is unlocked with the function:

pthread_mutex_unlock(pthread_mutex_t *mutex)

Signal wait

- Used to coordinate progress between threads
- When a thread need another thread to progress before it can continue it will wait
- When the other thread have progressed it will signal the other thread
- Schoolbook example is the producer consumer model

Condition variables

- Address communications between threads that share a mutex
- They are based upon programmer specified conditions

Notifying threads of events

- Problem:
  - Notify another thread that an event has occurred right now (synch) !
  - Thread start waiting until event happens (regardless the past)

Cluster Computing

Notifying threads - operations

- **wait** - wait until an event occurs.
- **signal** - notify one waiting thread that an even has occurred.
- **broadcast** - notify all waiting threads that an even has occurred.

condition-variable

The pthread library supply a tool for this kind of synchronization.

- The three operations defined on condition-variables are:
  - **wait** - blocks the thread.
  - **signal** - wakes one thread that is waiting on the condition-variable
  - **broadcast** - wakes all threads that are waiting on the condition-variable

How threads wait for a signal

- Just like mutexes every condition variable has a list of threads that are waiting to be signaled
- When a thread calls wait(& c) it adds itself to the waiting list and removes itself from the ready queue
- When signal(& c) is called one thread is extracted from the waiting list and is returned to the ready queue

Note

- The basic operations on conditions are: **signal** and **wait** for the condition
- A condition variable must always be associated with a **mutex**
- **WHY?????**

Note

- What happens when a thread signals on a conditional variable and there is no thread currently waiting?
- A signal is not preserved
  - If one thread signals on a condition variable and no thread is waiting at that moment, the signal "goes away"
  - When a thread waits on the same condition variable it does not catch the previous signal, and has to wait for a new signal

Wait syntax

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

- Atomically unlocks the mutex and waits for the condition variable to be signaled.
- The thread execution is suspended and does not consume any CPU time until the condition variable is signaled.
- The mutex must be locked by the calling thread on entrance to pthread_cond_wait

Signal syntax

int pthread_cond_signal(pthread_cond_t *cond);

- Restarts one of the threads that are waiting on the condition variable
- If no threads are waiting nothing happens.
- If several threads are waiting on exactly one is restarted, but it is not specified which.

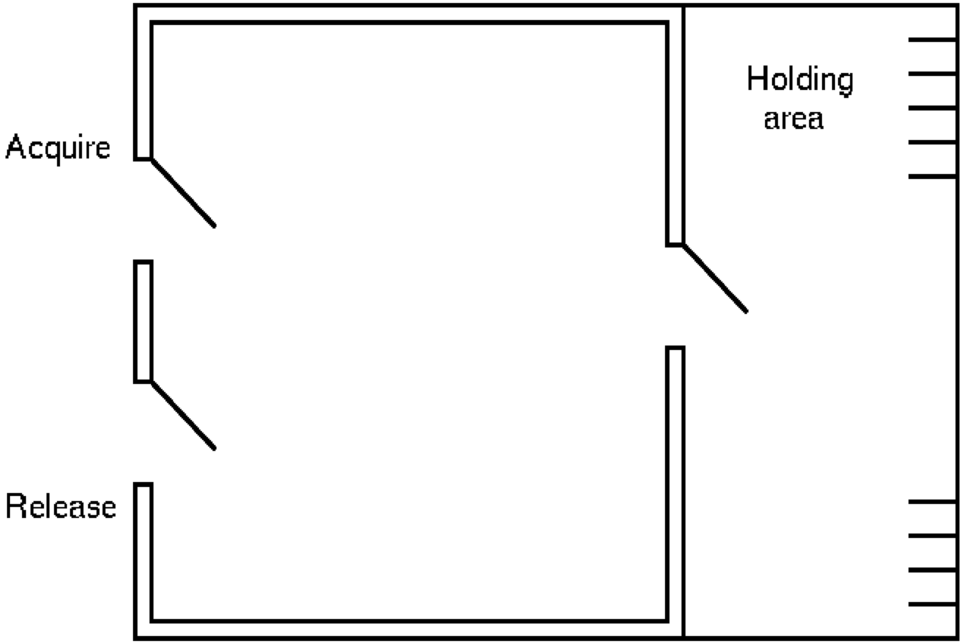
Barriers

- Barriers are used to allow a set of threads to ’meet up’
- Only after all threads have called the barrier are they allowed to continue
- Pthreads no longer has a barrier call ☹

Monitors (C.A.R. Hoare)

- higher level construct than semaphores
- a package of grouped procedures, variables and data
- processes call procedures within a monitor but cannot access internal data
- can be built into programming languages
- synchronization enforced by the compiler
- only one process allowed within a monitor at one time
- wait and signal operations on condition variables

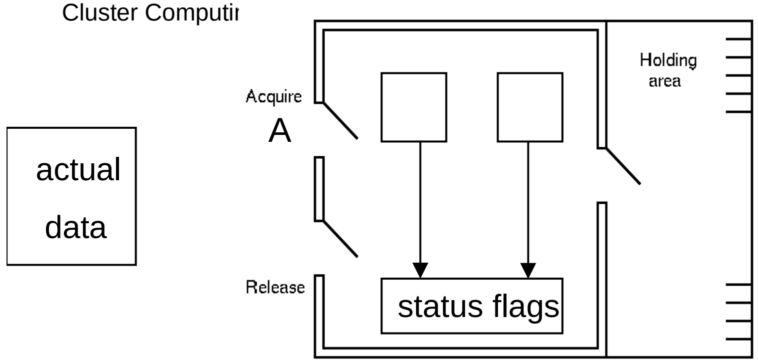
Blocked processes go into a *Holding Area*



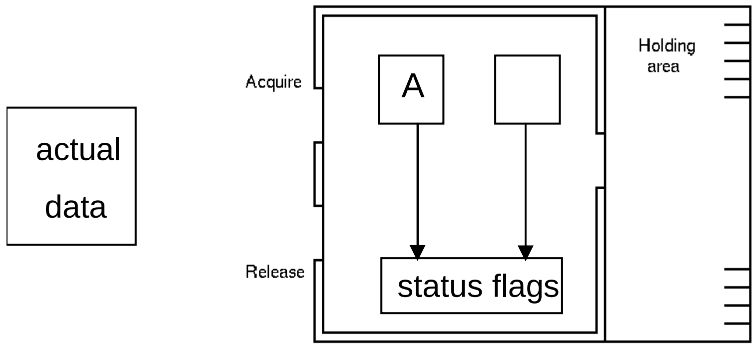
- Possibilities for running signaled and signaling processes
  - let newly signaled process run immediately, and make signaling process wait in holding area
  - let signaling process continue in monitor, and run signaled process when it leaves

Example

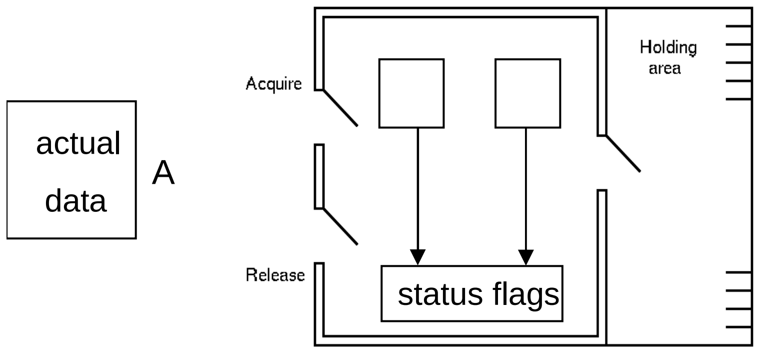
Cluster Computi



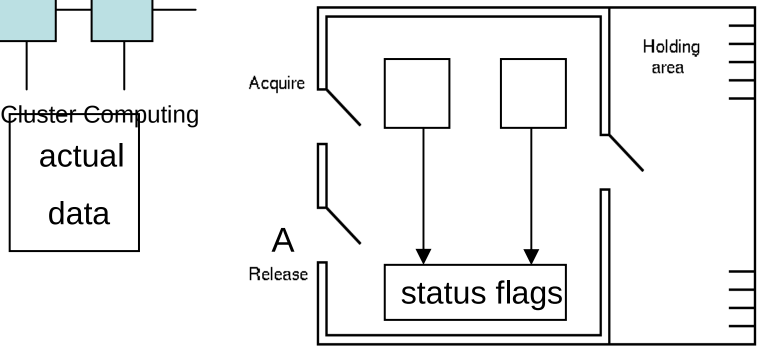
– process A entering monitor to request permission to access data



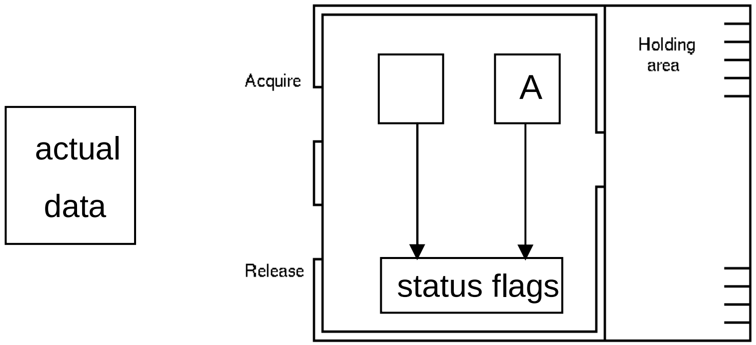
– receiving permission to access data



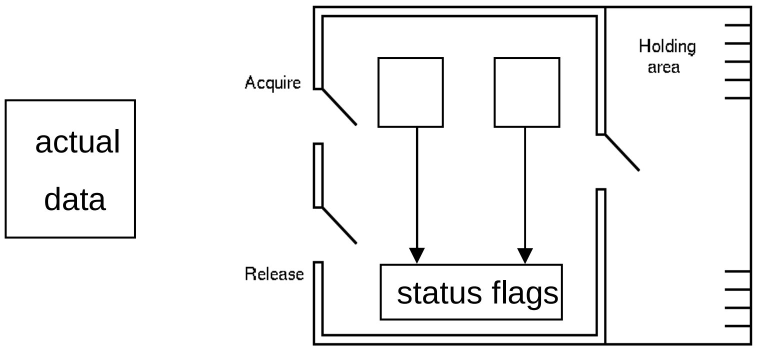
– leaving monitor to access data



– process A entering monitor to release access permission to data



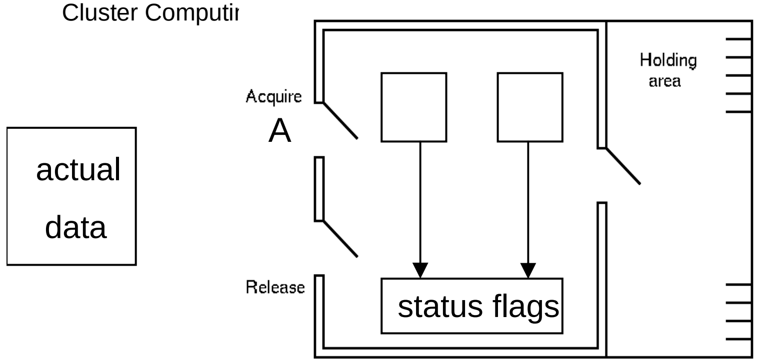
– releasing access permission to data



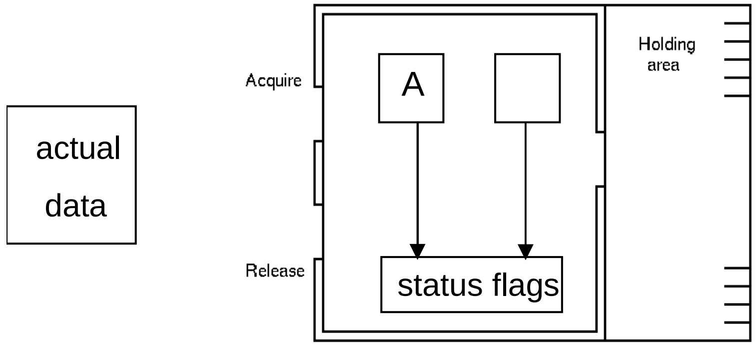
– leaving monitor

Cluster Computi

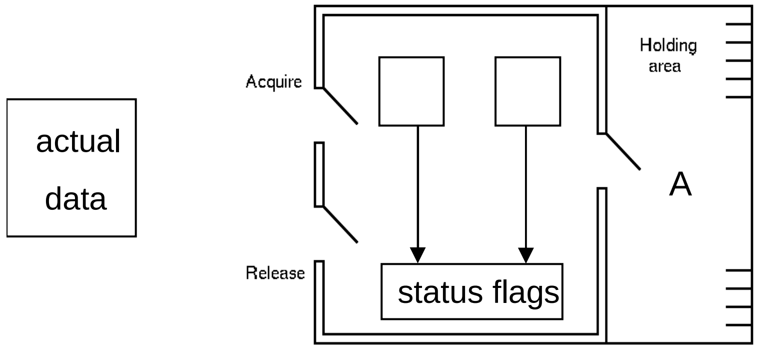
Example



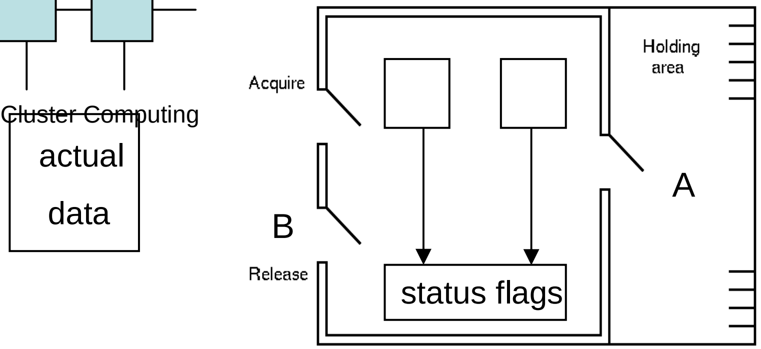
– process A entering monitor to get permission to access to data



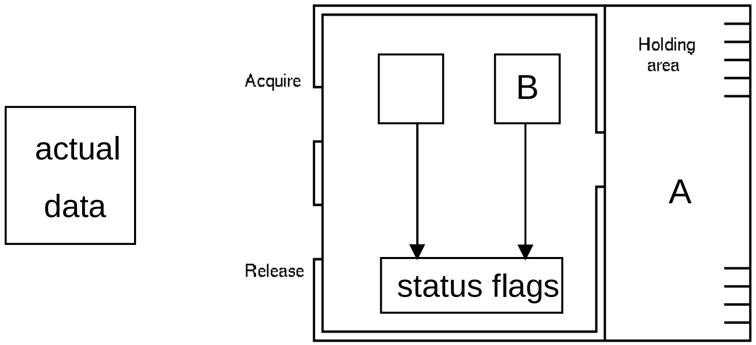
– entering monitor and *not* receiving permission to access data



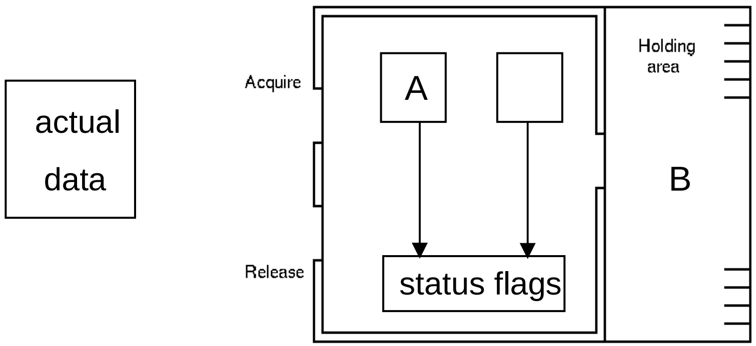
– having to wait in holding area



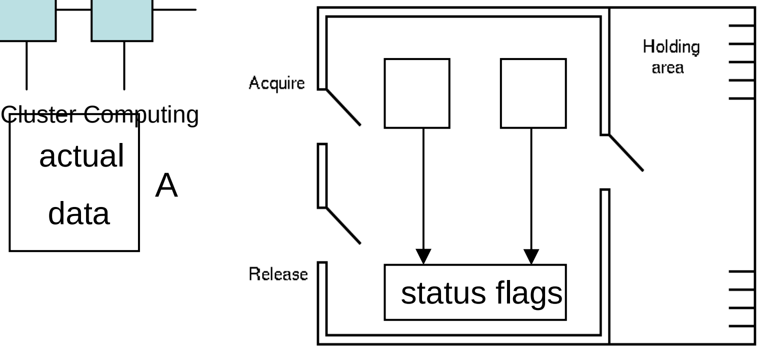
– process B entering monitor to release access to data



– process B releasing access to data

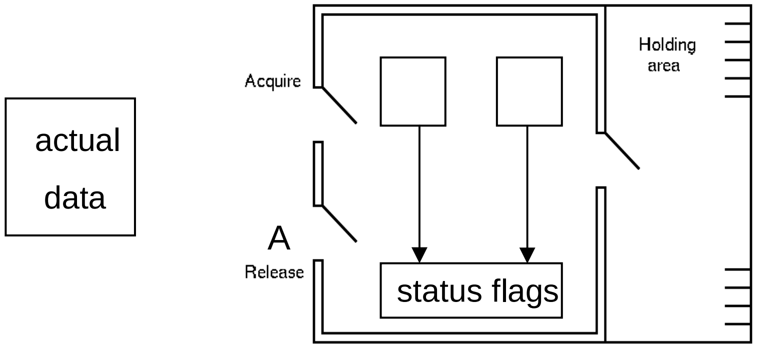


– process B entering holding area whilst process A re-enters monitor to get access permission to data

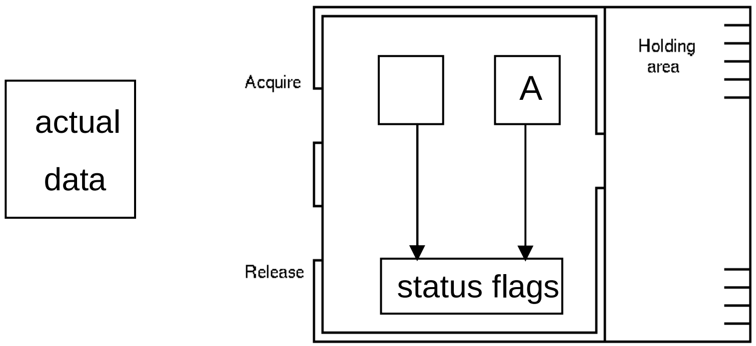


– process A is accessing data

– process B has left holding area and left the monitor



– process A entering monitor to release access to data



– process A releasing access to data

– finally process A leaves monitor

"Monitors" in Java

- Every object of a class that has a *synchronized* method has a "monitor" associated with it
- Any such method is guaranteed by the Java Virtual Machine execution model to execute mutually exclusively from any other synchronized methods for that object
- Access to individual objects such as arrays can also be synchronized
  - also complete class definitions
- Based around use of *threads*
- *One* condition variable per monitor
  - *wait()* releases a lock I.e.enters holding area
  - *notify()* signals a process to be allowed to continue
  - *notifyAll()* allows all waiting processes to continue

Example: producer/consumer

```
class ProCon {
    private int contents;
    private boolean available = false;

    public synchronized int get() {
        while (available==false) {
            try { wait(); }
            catch (InterruptedException e) { }
        }
        available = false;
        notify();
    return contents;
    }

    public synchronized int put(int value) {
        while (available==true) {
            try { wait(); }
            catch (InterruptedException e) { }
        contents = value;
        available = true;
        notify();
    }
}
```

Java monitor implementation of User-level semaphores

```
class Semaphore {
        private int value;

        Semaphore (int initial) { value = initial; }          // constructor

        synchronized public void P() {
                while (value==0) {
                        try { wait(); }
                        catch (InterruptedException e) { }
                }
                value = value-1;
        }

        synchronized public void V() {
                value = value+1;
                notify();
        }
}
```

- since the thread calling *notify()* may continue, or another thread execute, and invalidate the condition, it is safer to retest the condition in a *while* loop

```
class BoundedSemaphore {
        private int value, bound;

        Semaphore (int initial, int bound) {                    // constructor
                value = initial;
                this.bound = bound;
        }

        synchronized public void P() {
                while (value==0) {
                        try { wait(); }
                        catch (InterruptedException e) { }
                }
                value = value-1;
                notifyAll();
        }

        synchronized public void V() {
                 while (value==bound) {
                        try { wait(); }
                        catch (InterruptedException e) { }
                }
                value = value+1;
                notifyAll();
        }
}
```

Java Monitors - CONCERNS

- Threads yield non-determinacy (and, therefore, scheduling sensitivity) straight away ...
- No help provided to guard against race hazards ...
- Overheads too high (> 30 times **???**)
- Learning curve is long …
- Scalability (both in logic and performance) **???**
- Theoretical foundations **???**
  - (deadlock / livelock / starvation analysis **???**)
  - (rules / tools **???**)

“Wot, No Chickens!”

- Peter Welch, University of Kent
- Five Philosophers (consumers)
  - Think
  - Go to Canteen to get Chicken for dinner
  - Repeat
- Chef (producer)
  - produces four chickens at a time and delivers to canteen

“Wot, No Chickens!”

- Philosopher 0 is greedy -- never thinks
- Other philosophers think 3 time units before going to eat
- Chef takes 2 time units to cook four chickens
- Chef takes 3 time units to deliver chickens
  - occupies canteen while delivering

- Simplified code follows -- leaves out exception handling try-catch

```
class Canteen {

  private int n_chickens = 0;

  public synchronized int get(int id) {
    while (n_chickens == 0) {
      wait(); // Wot, No Chickens!
    }
    n_chickens--;// Those look good...one please
    return 1;
  }

  public synchronized void put(int value) {
    Thread.sleep(3000); // delivering chickens..
    n_chickens += value;
    notifyAll (); // Chickens ready!
} }
```

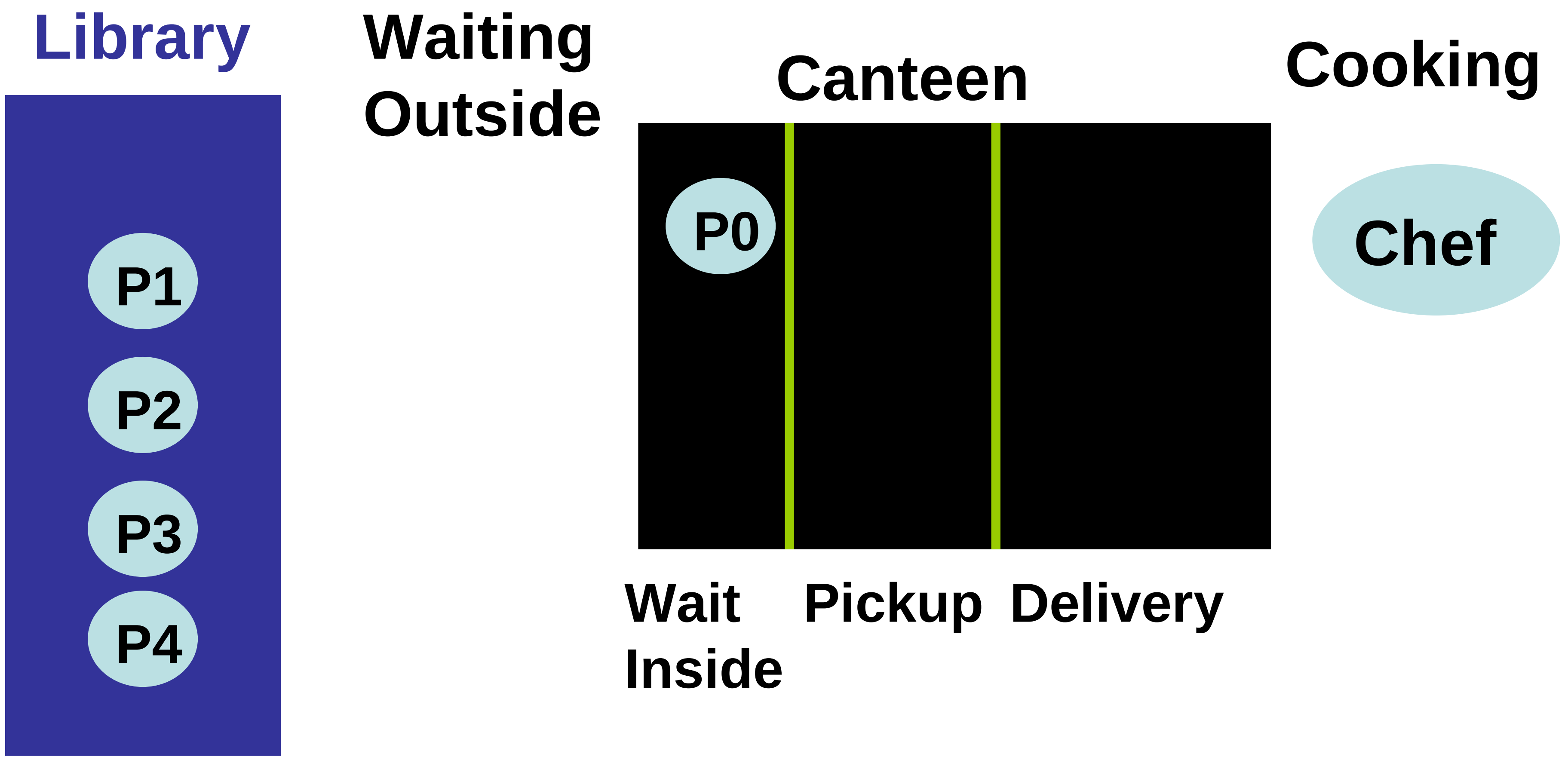
```
class Chef extends Thread {
  private Canteen canteen;

  public Chef (Canteen canteen) {
    this.canteen = canteen;
    start ();
  }

  public void run () {
    int n_chickens;
    while (true) {
      sleep (2000);//  Cooking...
      n_chickens = 4;
      canteen.put (n_chickens);
    }
  }
}
```

```
class Phil extends Thread {
  private int id;
  private Canteen canteen;

  public Phil(int id, Canteen canteen) {
    this.id = id;
    this.canteen = canteen;
    start ();
  }
  public void run() {
    int chicken;
    while (true) {
      if (id > 0) {
        sleep(3000);            // Thinking...
      }
      chicken = canteen.get(id);// Gotta eat...
    } // mmm...That's good
} }
```

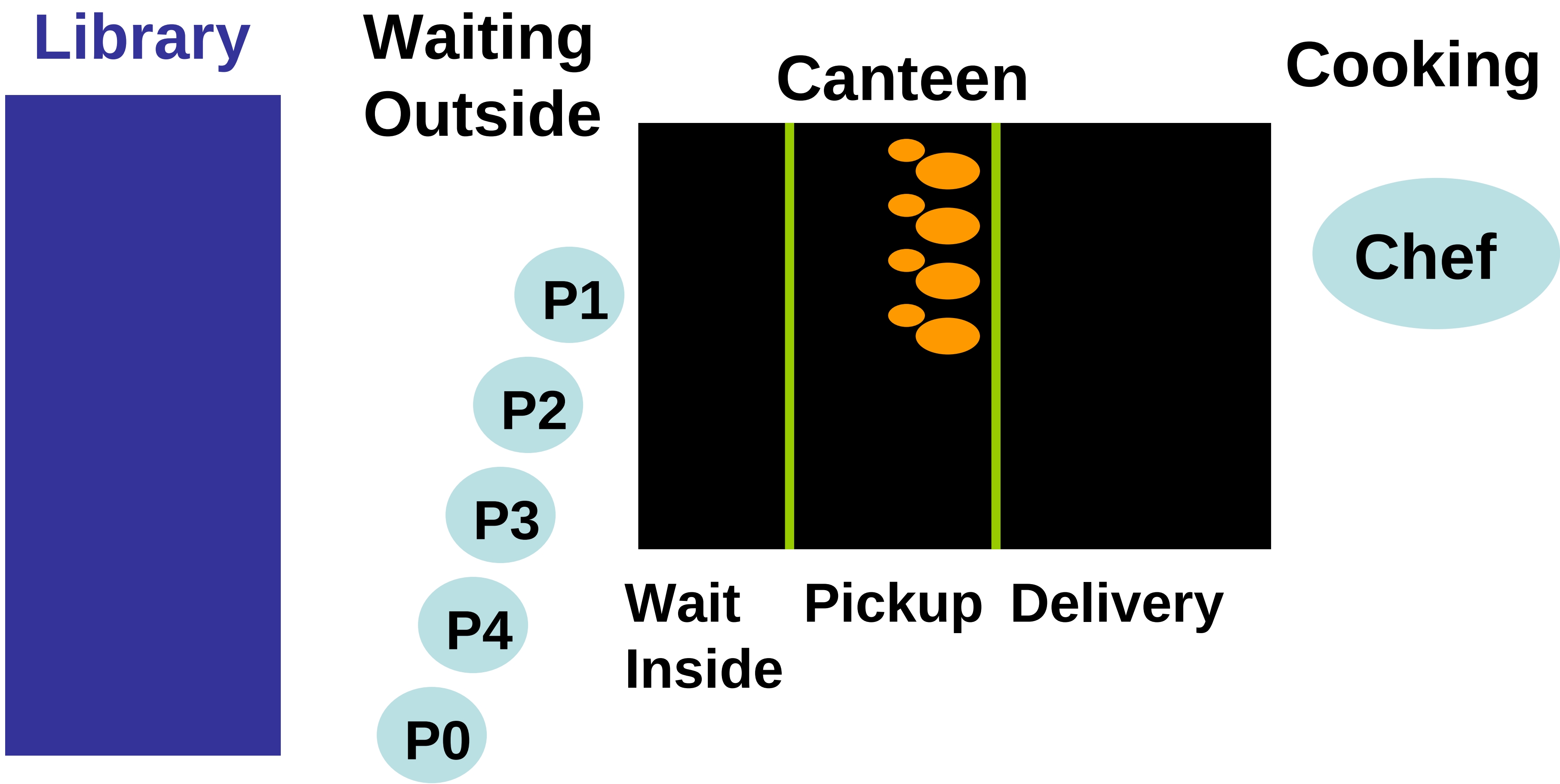
```
class College {

  public static void main (String argv[]) {

    int n_philosophers = 5;
    Canteen canteen = new Canteen ();
    Chef chef = new Chef (canteen);
    Phil[] phil = new Phil[n_philosophers];

    for (int i = 0; i < n_philosophers; i++) {
      phil[i] = new Phil (i, canteen);
    }
  }
}
```

Cluster Computing

"Wot, No Chickens!"

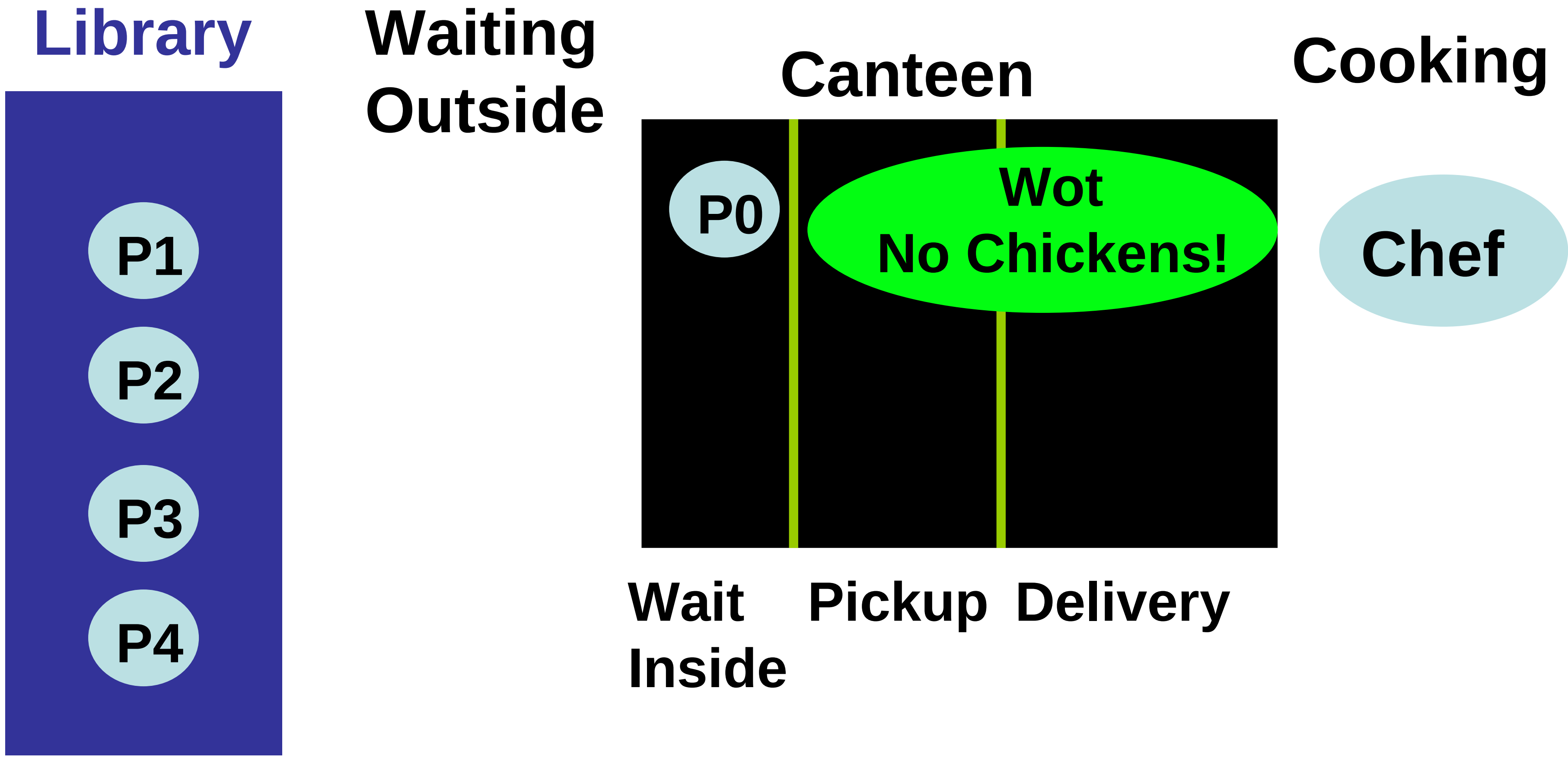


“Wot, No Chickens!”



Cluster Computing

"Wot, No Chickens!"



Compiler generated multithreaded applications

- Programming with threads is not trivial
- Parallel execution opens many new options for bugs
- Debugging is much harder
- Conclusion:
  - Make the compiler take over the job of handling the threading

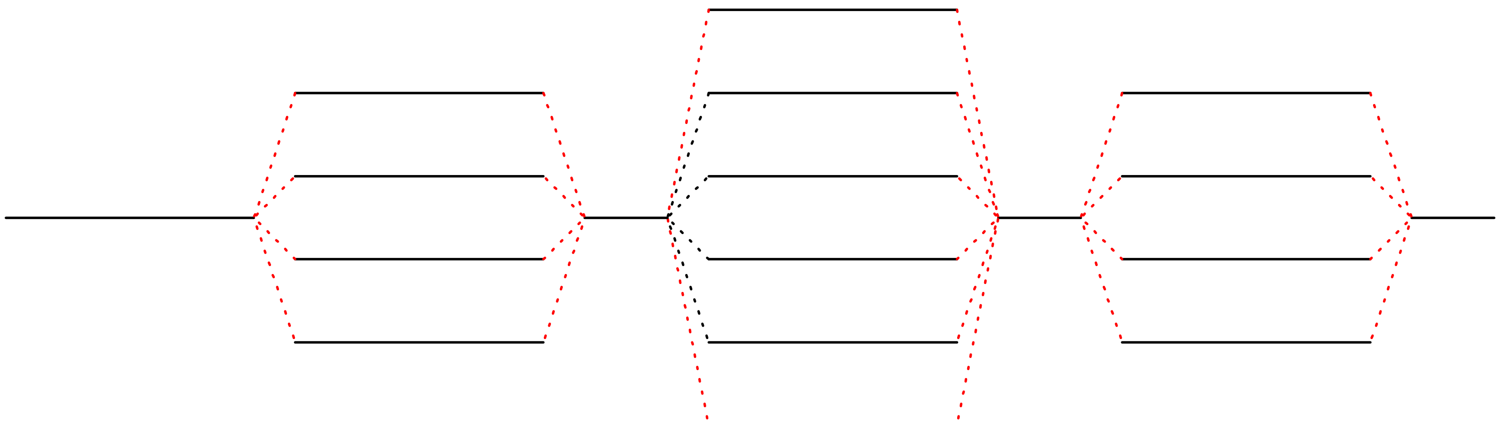
Higher level tools

- High Performance Fortran (Java)
- Open MP

OpenMP

- Industrial standard parallelizing pragmas

```
#pragma omp parallel for
for(i=0; i<length; i++)
  c[i]=a[i]+b[i];
```



High Performance Fortran

- HPF provides parallel pragms similar to those found in OpenMP
- In addition the compiler tries to detect potential parallelism in FORALL loops etc.
- Scalar data-types allow the compiler to use very high performance parallel libraries
  - A=B x C

High Performance Fortran

- HPF also provides the programmer with methods to give hints to the compiler on data layout

```
!HPF$ ALIGN A(*,BLOCK) c Divide A Vertically
!HPF$ ALIGN A(BLOCK,*) c Divide A Horizontally
!HPF$ ALIGN A(BLOCK,BLOCK) c Divide A into tiles
!HPF$ ALIGN A(*,CYCLIC) c Divide A by rows
```

CSP

- Communicating Sequential Processes
- Extreme multitasking
- Each process/thread has a number of ports that are either input or output, when data arrives at an input-port it is processed and sent to an output-port
- Easily programmed using Occam