18-600 Foundations of Computer Systems

Lecture 26: "Parallel Programming"

John P. Shen & Zhiyi Yu
December 5, 2016

- Required Reading Assignment:
  - **Chapter 12 of CS:APP (3rd edition) by Randy Bryant & Dave O'Hallaron.**
- Recommended Reference:
  "Parallel Computer Organization and Design," by Michel Dubois, Murali Annavaram, Per Stenstrom, Chapters 5 and 7, 2012.

Electrical & Computer ENGINEERING

Carnegie Mellon University

18-600 Foundations of Computer Systems

Lecture 26:
"Parallel Programming"

A. Parallel Programs for Parallel Architectures
B. Parallel Programming Models
C. Shared Memory Model
D. Message Passing Model
E. Thread Level Parallelism Examples

Electrical & Computer ENGINEERING

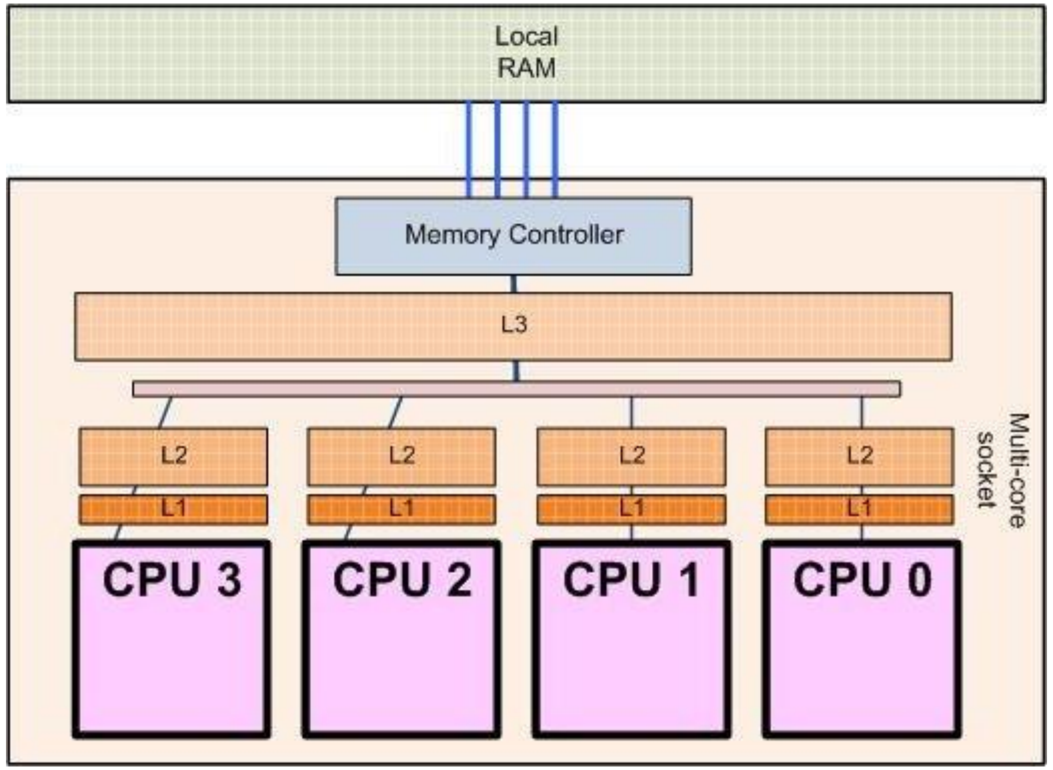
Parallel Architectures: MCP & MCC

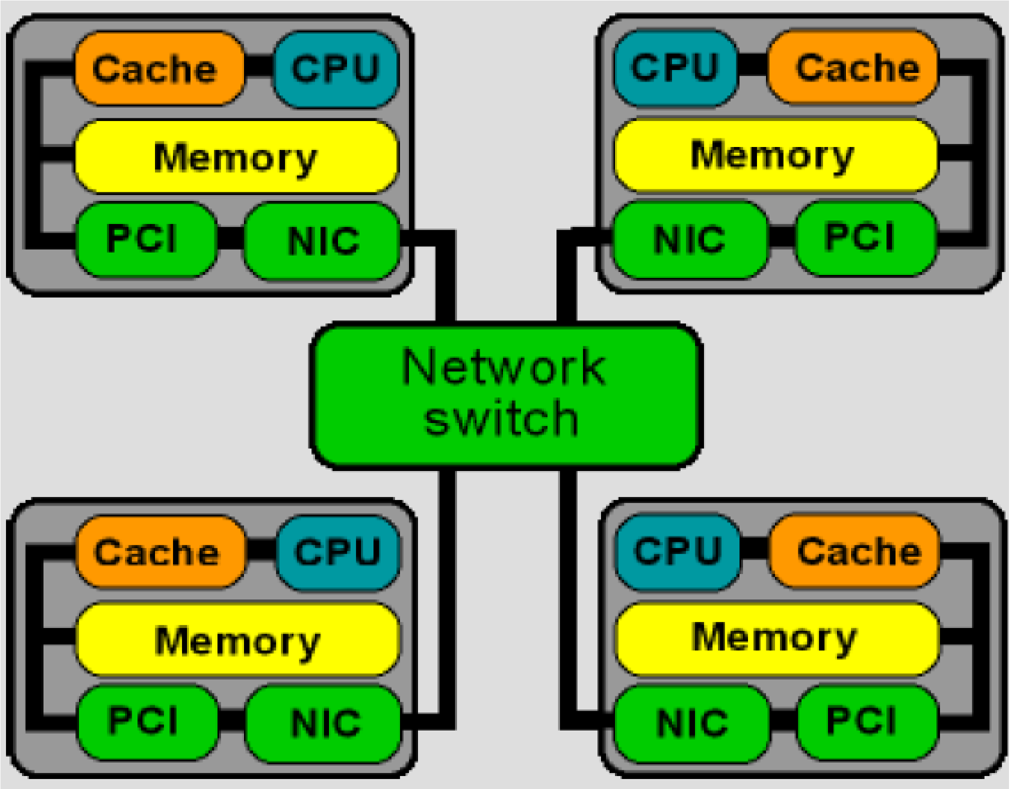
MULTIPROCESSING

Shared Memory Multicore Processors (MCP) or Chip Multiprocessors (CMP)

CLUSTER COMPUTING

Shared File System & LAN Connected Multi-Computer Clusters (MCC)





A. Parallel Programs for Parallel Architectures

- Why is Parallel Programming so hard?
  - Conscious mind is inherently sequential
  - (sub-conscious mind is extremely parallel)
- Identifying parallelism in the problem
- Expressing parallelism to the parallel hardware
- Effectively utilizing parallel hardware (MCP or MCC)
  - MCP: **OpenMP** (Shared Memory)
  - MCC: **Open MPI** (Message Passing)
- Debugging parallel algorithms

Finding Parallelism

1. Functional parallelism
   - Car: {engine, brakes, entertain, nav, …}
   - Game: {physics, logic, UI, render, …}
   - Signal processing: {transform, filter, scaling, …}
2. Request parallelism
   - Web service, shared database, ATM, …
3. Data parallelism
   - Vector, matrix, DB table, pixels, …
4. Multi-threaded Parallelism
   - Decompose/parallelize sequential programs

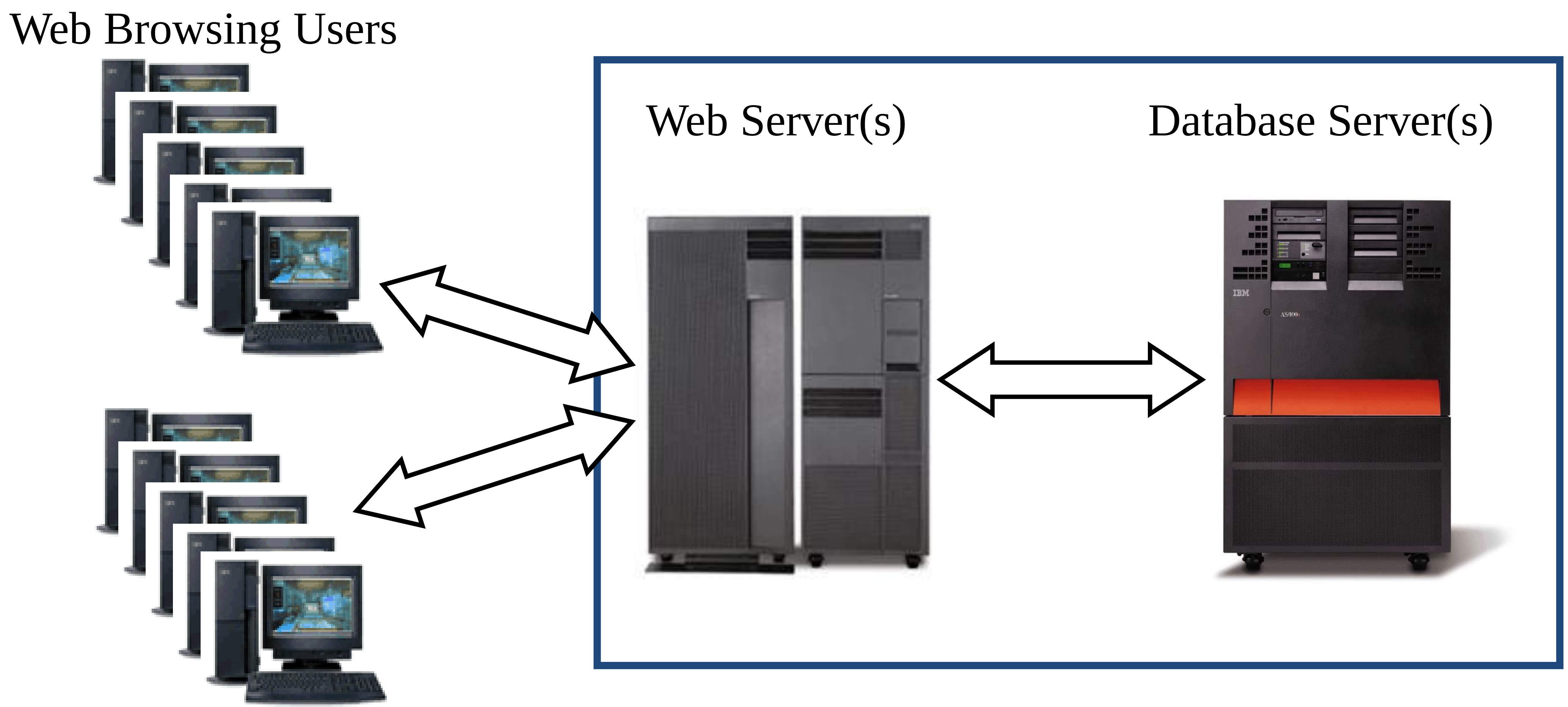
1. Functional Parallelism

Functional parallelism

- Car: {engine, brakes, entertain, nav, …}
- Game: {physics, logic, UI, render, …}
- Signal processing: {transform, filter, scaling, …}

- Relatively easy to identify and utilize
- Provides small-scale parallelism
  - 3x-10x
- Balancing stages/functions is difficult

2. Request Parallelism



- Multiple users => significant parallelism
- Challenges
  - Synchronization, communication, balancing work

3. Data Parallelism

Data parallelism

- Vector, matrix, DB table, pixels, …

- Large data => significant parallelism
- Many ways to express parallelism
  - Vector/SIMD ISA extensions
  - Threads, processes, shared memory
  - Message-passing
- Challenges:
  - Balancing & coordinating work
  - Communication vs. computation at scale

4. Multi-threaded Parallelism

Automatic extraction of parallel threads

- Decompose/Parallelize sequential programs

- Works well for certain application types
  - Regular control flow and memory accesses
- Difficult to guarantee correctness in all cases
  - Ambiguous memory dependences
  - Requires speculation, support for recovery
- Degree of parallelism
  - Large (1000x) for *easy* cases
  - Small (3x-10x) for *difficult* cases

Expressing Parallelism

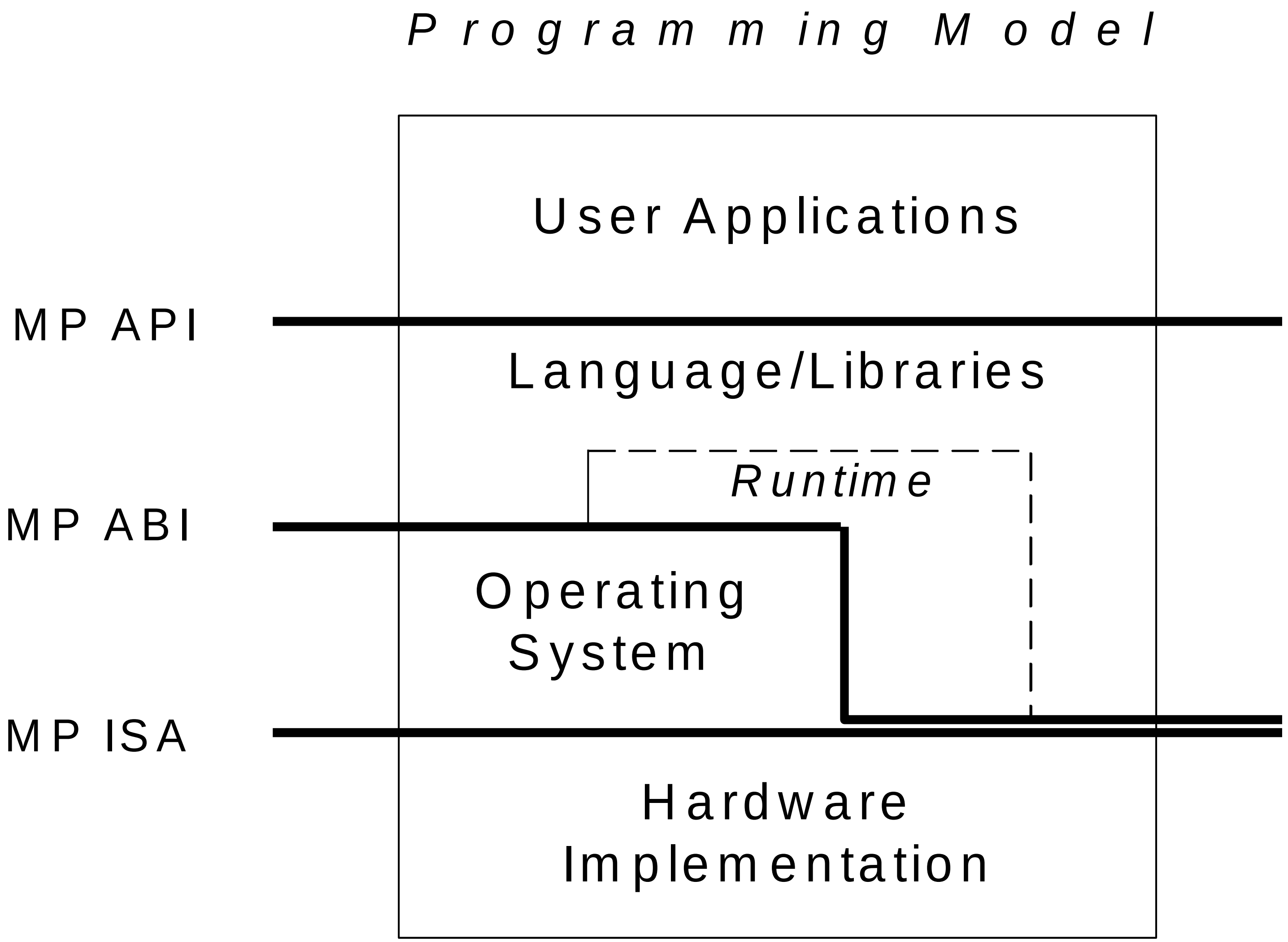
- SIMD – Cray-1 case study
  - MMX, SSE/SSE2/SSE3/SSE4, AVX at small scale
- SPMD – GPGPU model
  - All processors execute same program on disjoint data
  - Loose synchronization vs. rigid lockstep of SIMD
- MIMD – most general (this lecture)
  - Each processor executes its own program/thread
- Expressed through standard interfaces
  - API, ABI, ISA

B. Parallel Programming Models

- High level paradigms for expressing an algorithm
  - Examples:
    - Functional programs
    - Sequential, procedural programs
    - Shared-Memory parallel programs
    - Message-Passing parallel programs
- Embodied in high level languages that support concurrent execution
  - Incorporated into HLL constructs
  - Incorporated as libraries added to existing sequential language
- Top level features:
  - For conventional models – **shared memory**, **message passing**
  - Multiple threads are conceptually visible to programmer
  - **Communication**/**synchronization** are visible to programmer

MP (Multiprocessing or MIMD) Interfaces

- *Levels of abstraction* enable complex system designs (such as MP computers)
- Fairly natural extensions of uniprocessor model
  - Historical evolution

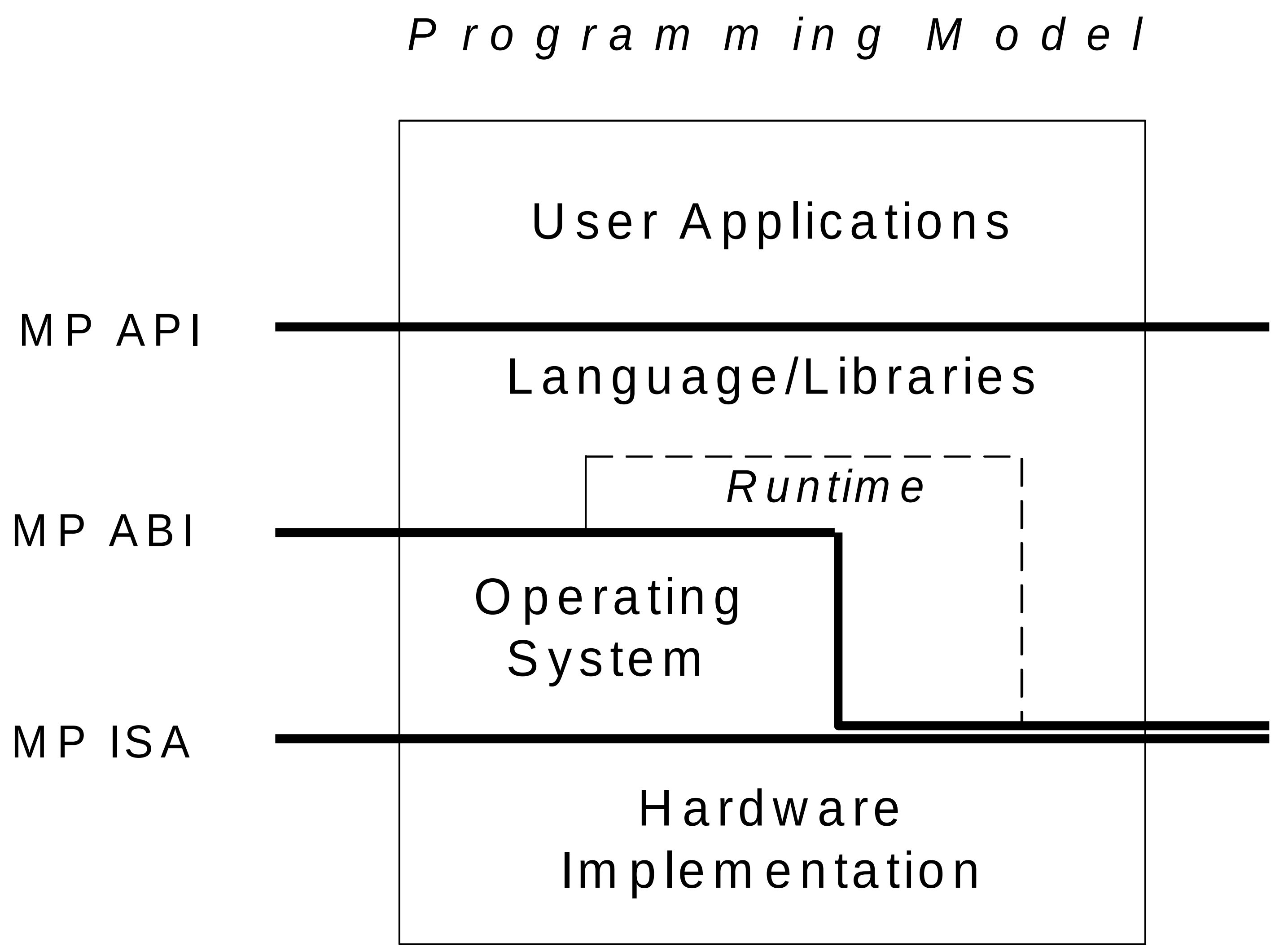


Application Programming Interface (API)

- Interface where HLL programmer works
- High level language plus libraries
  - Individual libraries are sometimes referred to as an "API"
- User level runtime software is often part of API implementation
  - Executes procedures
  - Manages user-level state
- Examples:
  - C and pthreads
  - FORTRAN and MPI

Application Binary Interface (ABI)

- Program in API is compiled to ABI
- Consists of:
  - OS call interface
  - User level instructions (part of ISA)



Instruction Set Architecture (ISA)

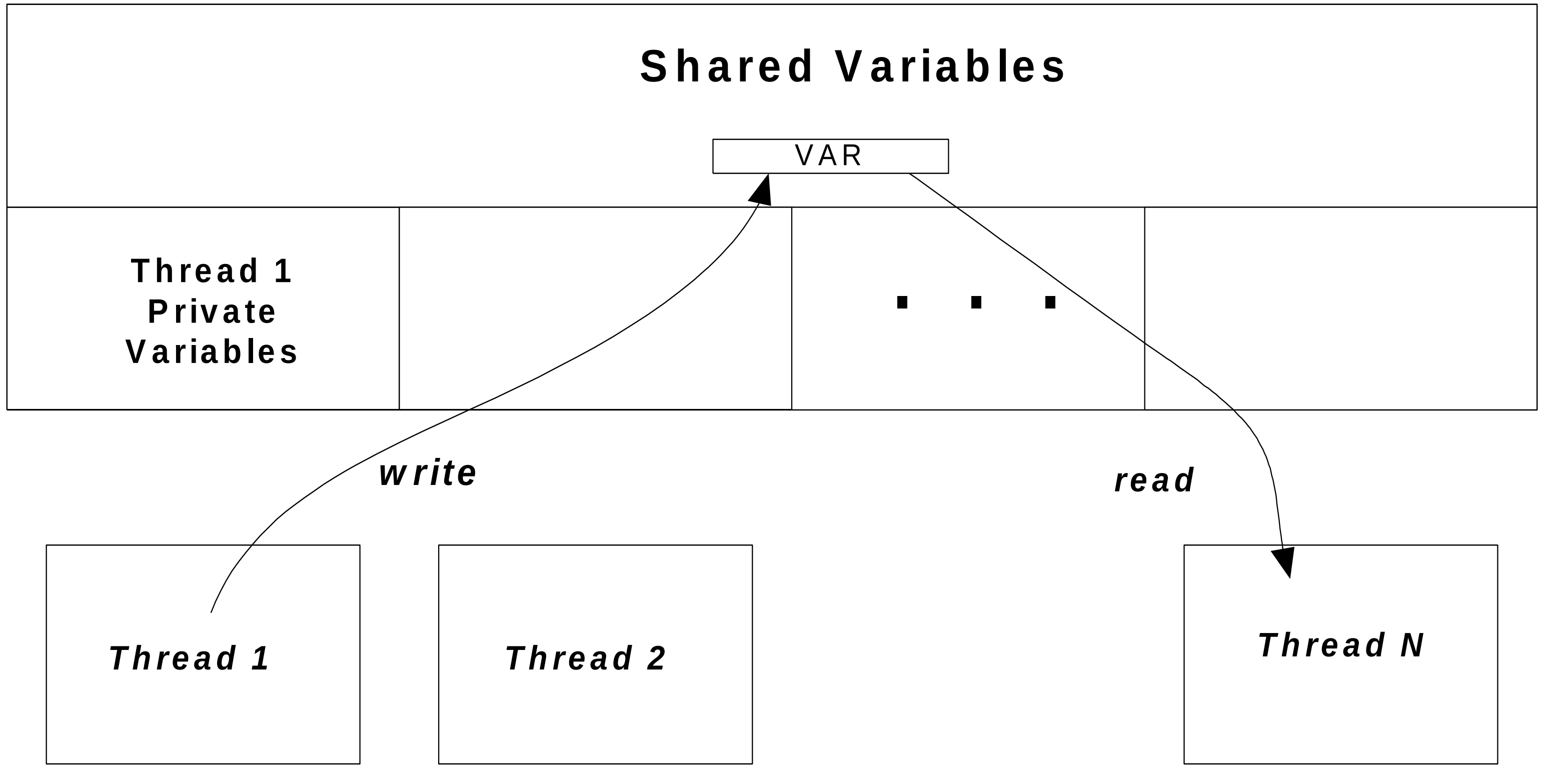
- Interface between hardware and software
  - What the hardware implements
- Architected state
  - Registers
  - Memory architecture
- All instructions
  - May include parallel (SIMD) operations
  - Both non-privileged and privileged
- Exceptions (traps, interrupts)

Major (MP or MIMD) Abstractions

- For both Shared Memory & Message Passing (programming models)
- Processes and Threads (parallelism expressed)
  - ***Process:*** A shared address space and one or more threads of control flows
  - ***Thread:*** A program sequencer and private address space (private stack)
  - ***Task***: Less formal term – part of an overall job
  - Created, terminated, scheduled, etc.
- Communication
  - Passing of data
- Synchronization
  - Communicating control information
  - To ensure reliable, deterministic communication

C. Shared Memory Model

- Flat shared memory or object heap
  - Synchronization via memory variables enables reliable sharing
- Single process
- Multiple threads per process
  - Private memory per thread
- Typically built on shared memory hardware system

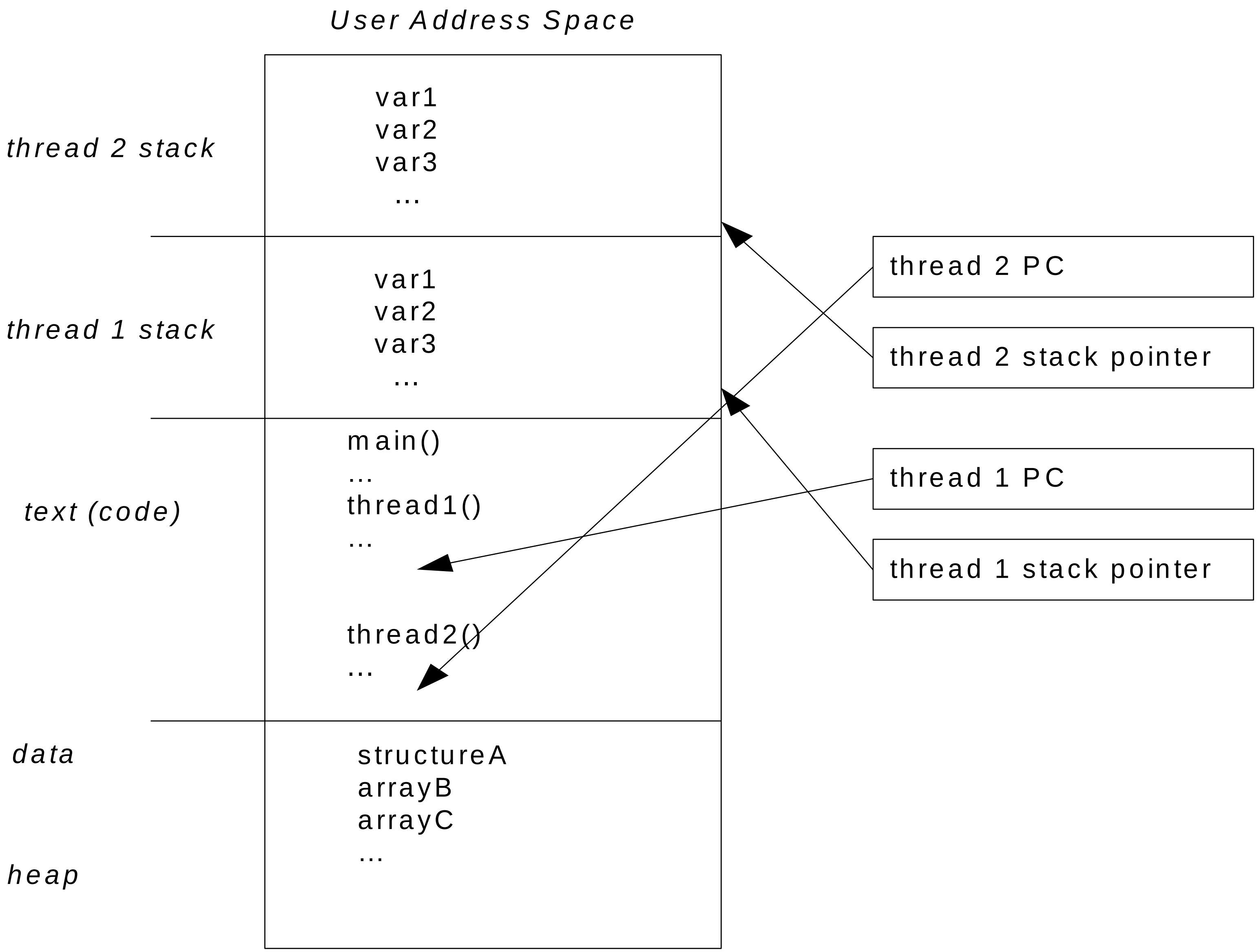


Threads and Processes

- Creation
  - generic -- Fork
    - (Unix forks a process, not a thread)
  - pthread_create(….*thread_function….)
    - creates new thread in current address space
- Termination
  - pthread_exit
    - or terminates when thread_function terminates
  - pthread_kill
    - one thread can kill another

Example

- Unix process with two threads
  - (PC and stack pointer actually part of ABI/ISA implementation)



Shared Memory Communication

- Reads and writes to shared variables via normal language (assignment) statements (e.g. assembly load/store)

Thread 0	Thread 1
	load r1, A
	addi r1, r1, 3
load r1, A	
addi r1, r1, 1	
store r1, A	
	store r1, A

(a)

Thread 0	Thread 1
load r1, A	
addi r1, r1, 1	
store r1, A	
	load r1, A
	addi r1, rl, 3
	store r1, A

(b)

Thread 0	Thread 1
	load r1, A
	addi r1, r1, 3
	store r1, A
load r1, A	
addi r1, r1, 1	
store r1, A	

(c)

Thread 0	Thread 1
load r1, A	
addi r1, r1, 1	
	load r1, A
	addi r1, rl, 3
	store r1, A
store r1, A	

(d)

Shared Memory Synchronization

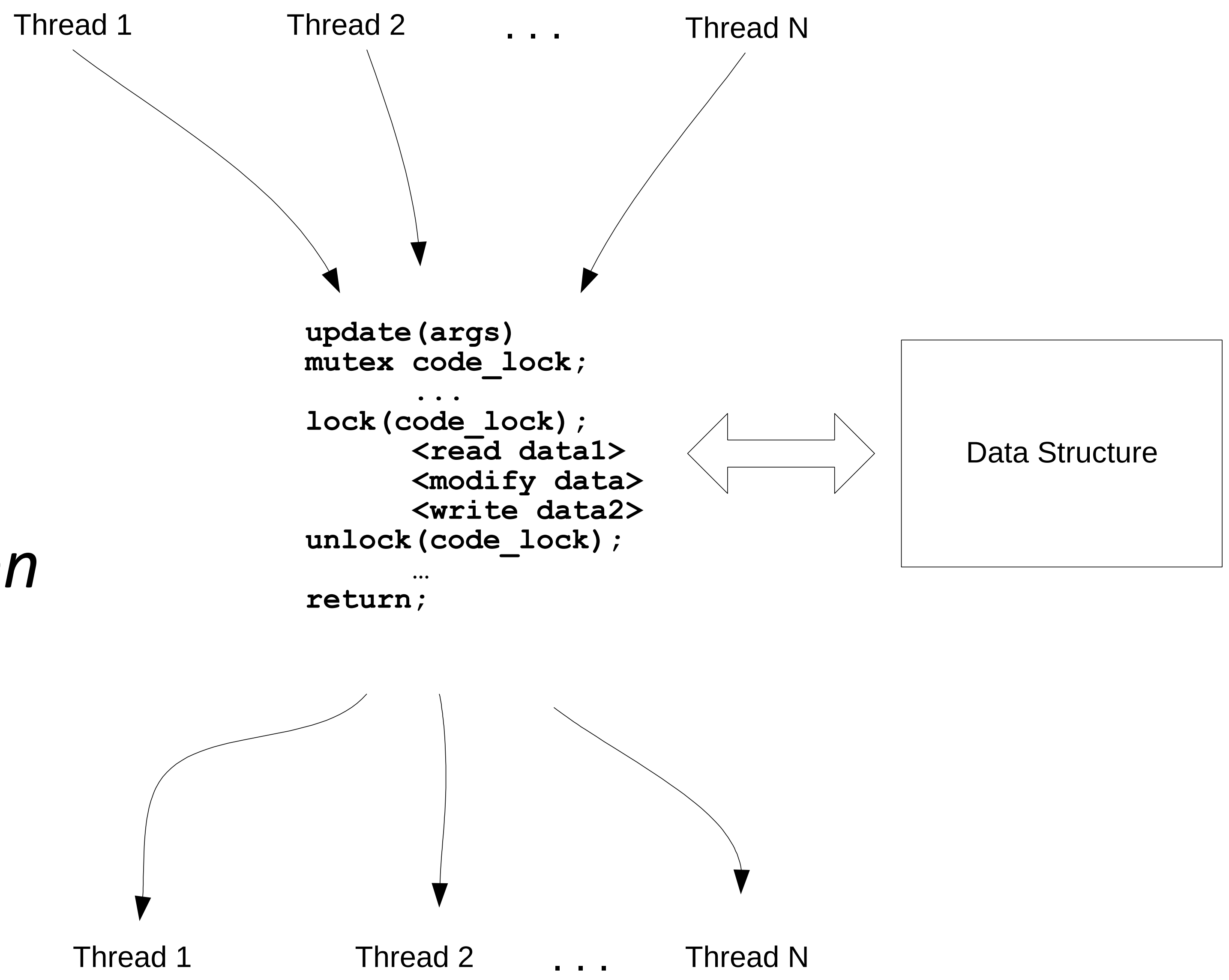
- What really gives shared memory programming its structure
- Usually explicit in shared memory model
  - Through language constructs or API
- Three major classes of synchronization
  - Mutual exclusion (mutex)
  - Point-to-point synchronization
  - Rendezvous
- Employed by *application design patterns*
  - *A general description or template for the solution to a commonly recurring software design problem.*

Mutual Exclusion (mutex)

- Assures that only one thread at a time can access a code or data region
- Usually done via *locks*
  - One thread acquires the lock
  - All other threads excluded until lock is released
- Examples
  - pthread_mutex_lock
  - pthread_mutex_unlock
- Two main application programming patterns
  - Code locking
  - Data locking

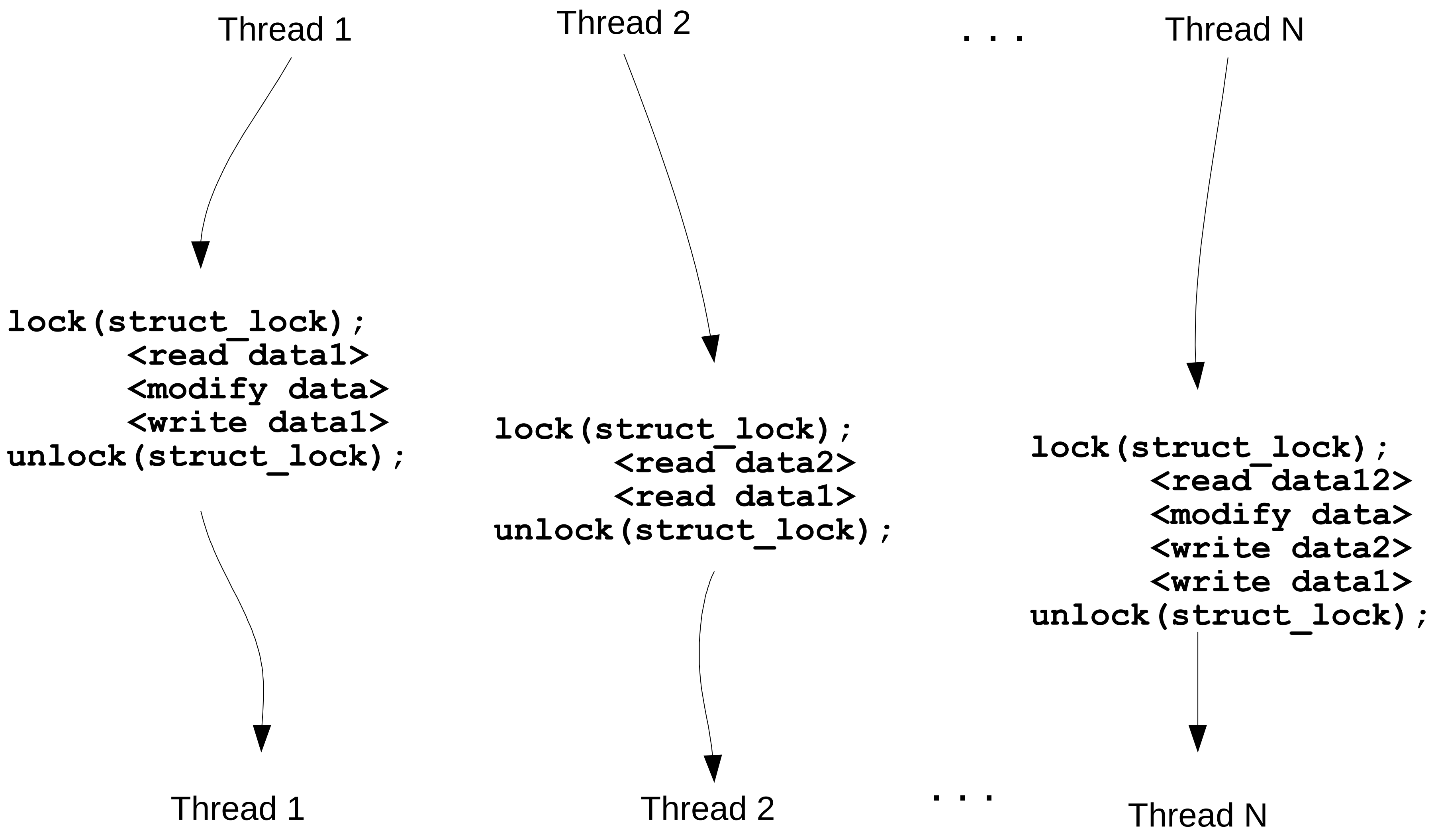
Code Locking

- Protect shared data by locking the code that accesses it
- Also called a *monitor* pattern
- Example of a *critical section*



Data Locking

- Protect shared data by locking data structure



Data Locking

- Preferred when data structures are read/written in combinations
- Example:

```
<thread 0>
Lock(mutex_struct1)
Lock(mutex_struct2)
   <access struct1>
   <access struct2>
Unlock(mutex_data1)
Unlock(mutex_data2)
```

```
<thread 1>
Lock(mutex_struct1)
Lock(mutex_struct3)
   <access struct1>
   <access struct3>
Unlock(mutex_data1)
Unlock(mutex_data3)
```

```
<thread 2>
Lock(mutex_struct2)
Lock(mutex_struct3)
   <access struct2>
   <access struct3>
Unlock(mutex_data2)
Unlock(mutex_data3)
```

Deadlock

- Data locking is prone to deadlock
  - If locks are acquired in an unsafe order
- Example:

<thread 0>

```
Lock(mutex_data1)
Lock(mutex_data2)
   <access data1>
   <access data2>
Unlock(mutex_data1)
Unlock(mutex_data2)
```

<thread 1>

```
Lock(mutex_data2)
Lock(mutex_data1)
   <access data1>
   <access data2>
Unlock(mutex_data1)
Unlock(mutex_data2)
```

- Complexity
  - Disciplined locking order must be maintained, else deadlock
  - Also, composability problems
    - Locking structures in a nest of called procedures

Efficiency

- Lock Contention
  - Causes threads to wait
- Function of lock *granularity*
  - Size of data structure or code that is being locked
- Extreme Case:
  - "One big lock" model for multithreaded OSes
  - Easy to implement, but very inefficient
- Finer granularity
  - \+ Less contention
  - \- More locks, more locking code
  - \- Perhaps more deadlock opportunities
- Coarser granularity
  - Opposite +/- of above

Point-to-Point Synchronization

- One thread signals another that a condition holds
  - Can be done via API routines
  - Can be done via normal load/stores
- Examples
  - pthread_cond_signal
  - pthread_cond_wait
    - suspends thread if condition not true
- Application program pattern
  - Producer/Consumer

<Producer>

```
while (full == 1){}; wait
buffer = value;
full = 1;
```

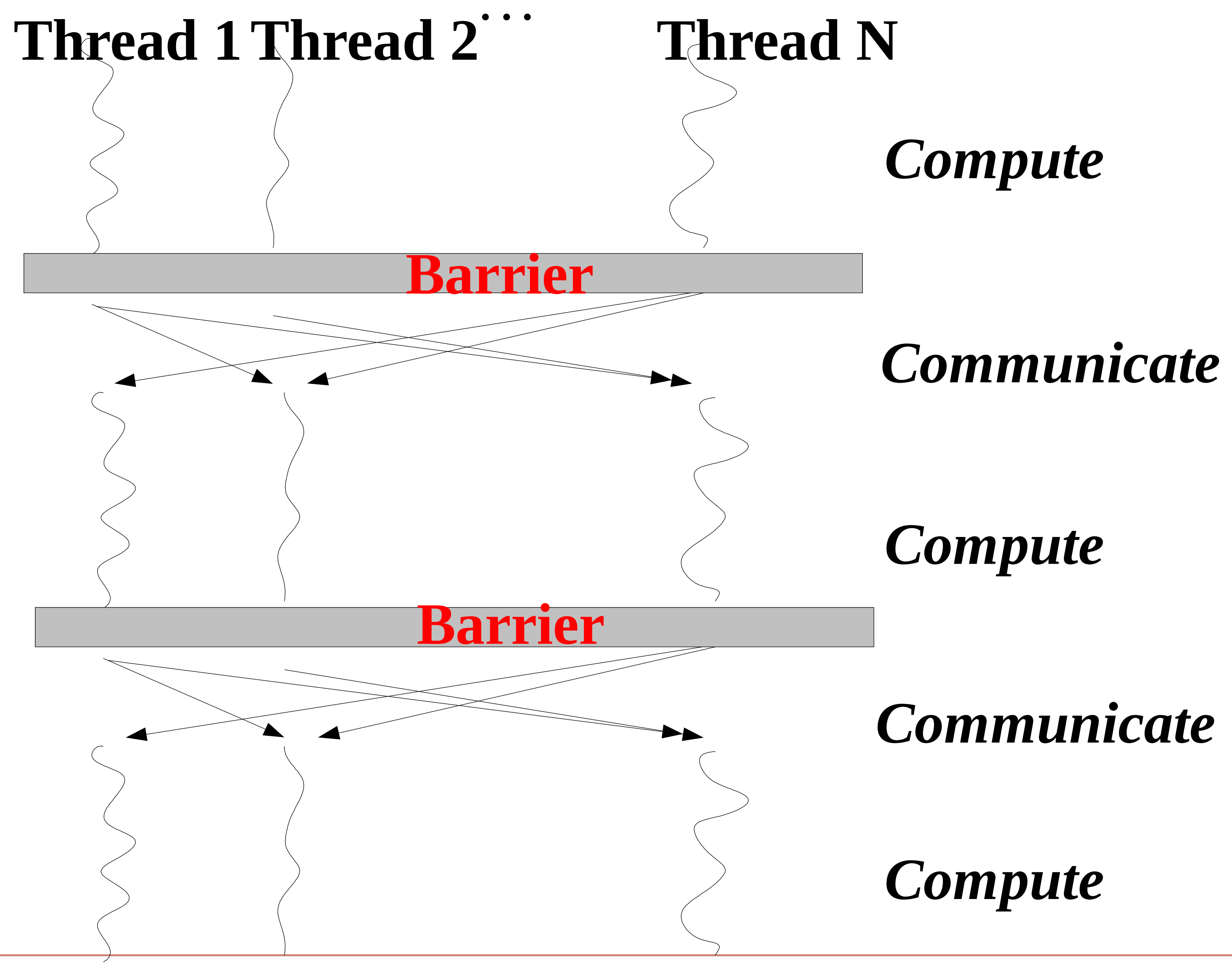
<Consumer>

```
while (full == 0){}; wait
b = buffer;
full = 0;
```

Rendezvous

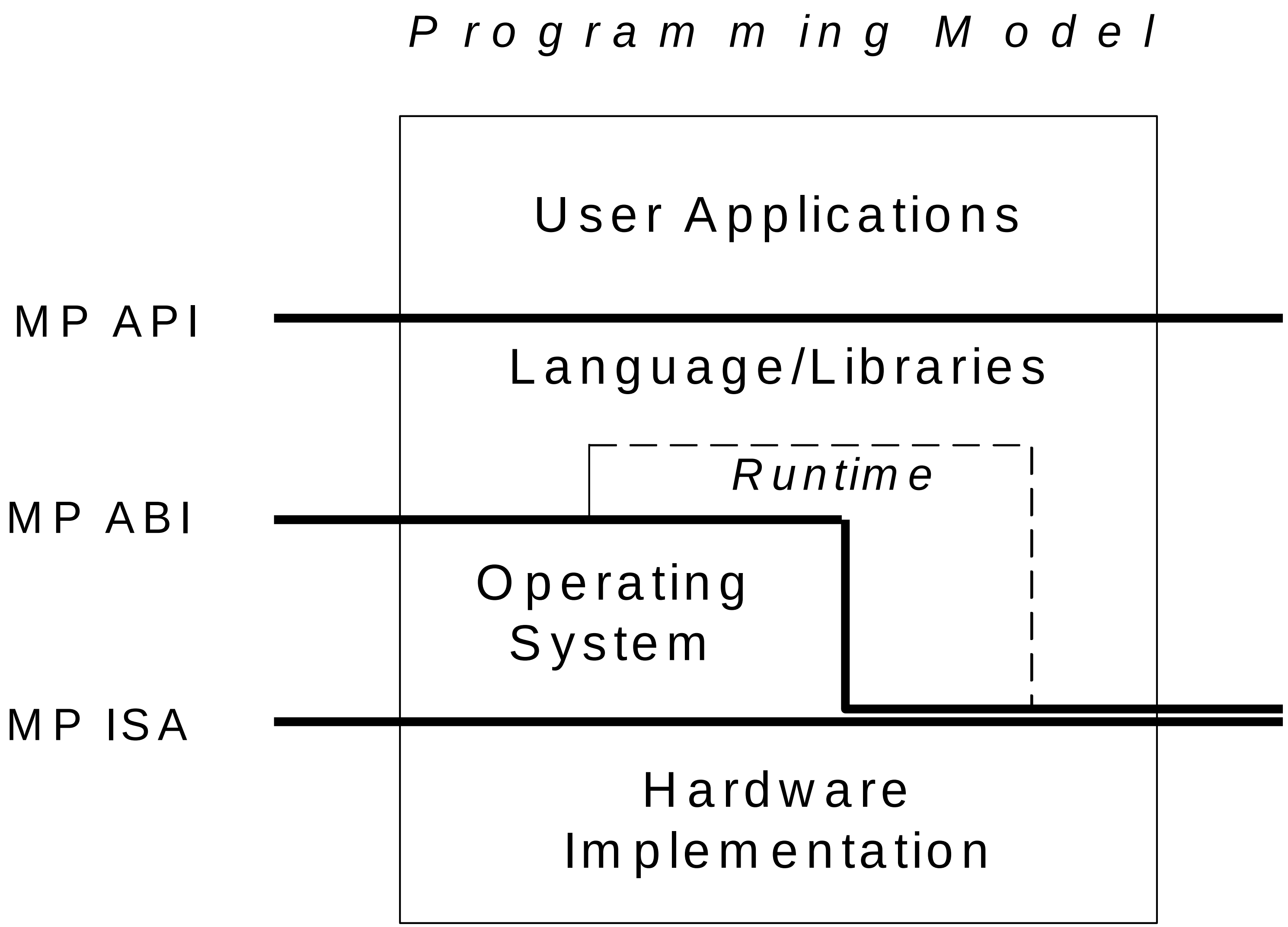
- Two or more cooperating threads must reach a program point before proceeding
- Examples
  - Wait for another thread at a join point before proceeding
    - example: pthread_join
  - Barrier synchronization
    - many (or all) threads wait at a given point
- Application program pattern
  - Bulk synchronous programming pattern

Bulk Synchronous Program Pattern



API Implementation

- Implemented at ABI and ISA level
  - OS calls
  - Runtime software
  - Special instructions
- Processes and Threads
  - OS processes
  - OS threads
  - User threads



OS Processes

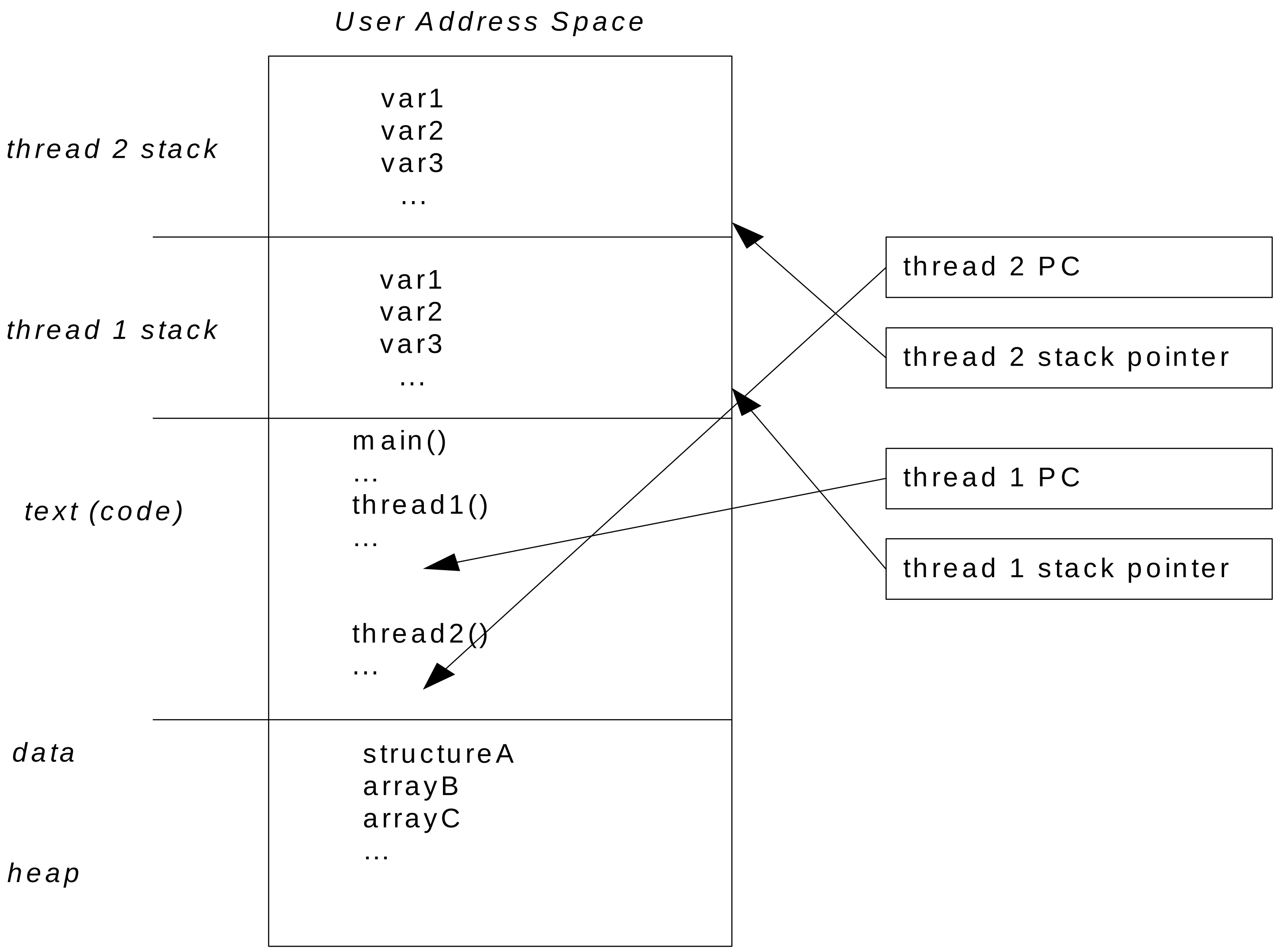
- Processes
- Use OS fork to create processes
- Use OS calls to set up shared address space
- OS manages processes (and threads) via run queue
- Heavyweight thread switches
  - OS call followed by:
  - Switch address mappings
  - Switch process-related tables
  - Full register switch
- Advantage
  - Processes have protected private memory

OS (Kernel) Threads

- API pthread_create() maps to Linux clone()
  - Allows multiple threads sharing same memory address space
- OS manages threads via run queue
- Lighter weight thread switch
  - Still requires OS call
  - OS switches architected register state and stack pointer

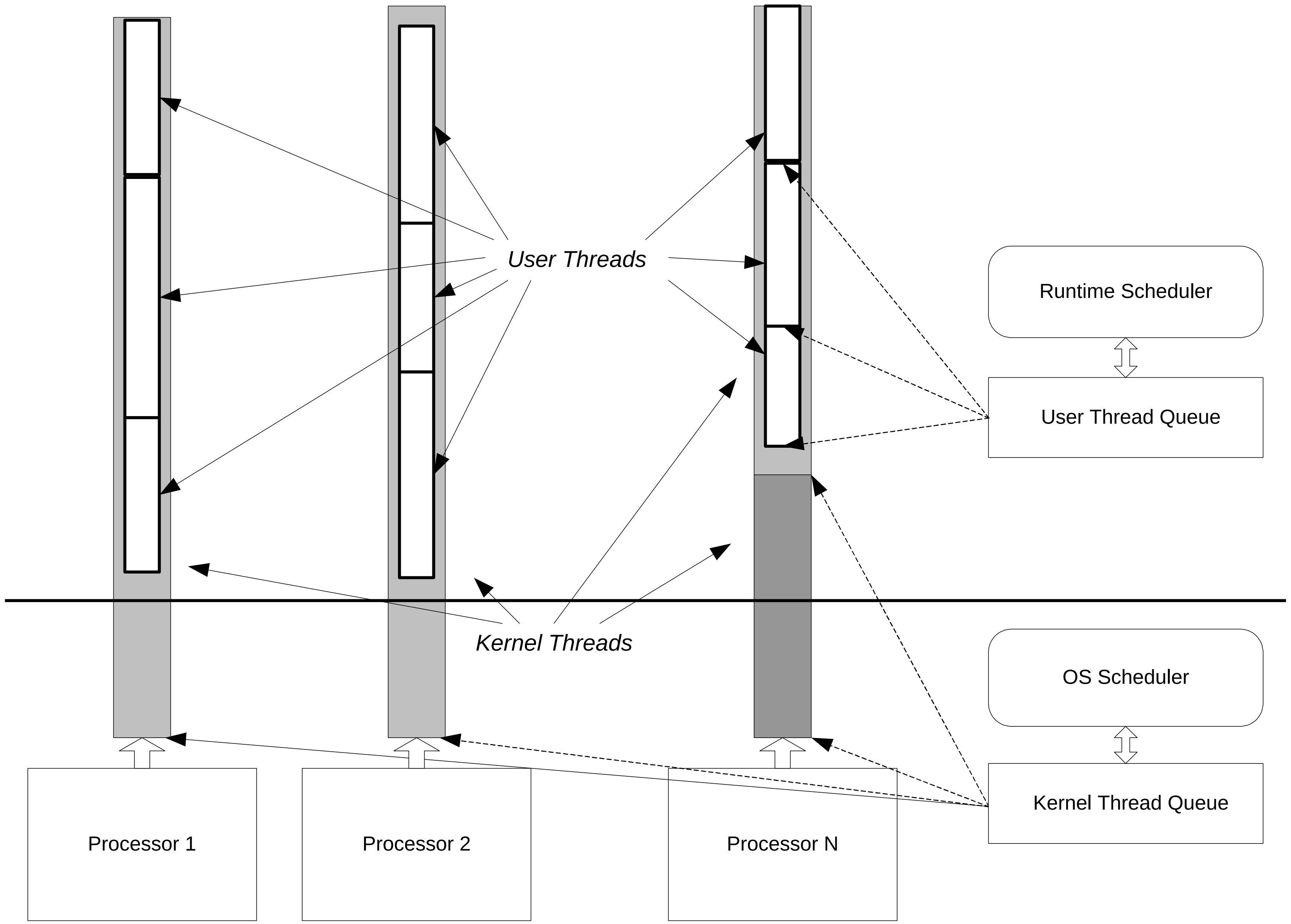
User Threads

- If memory mapping doesn't change, why involve OS at all?
- Runtime creates threads simply by allocating stack space
- Runtime switches threads via user level instructions
  - thread switch via jumps



Implementing User Threads

- Multiple kernel threads needed to get control of multiple hardware processors
- Create kernel threads (OS schedules)
- Create user threads that runtime schedules onto kernel threads



Lock Implementation

- Reliable locking can be done with *atomic* read-modify-write instruction
- Example: test&set
  - read lock and write a one
  - some ISAs also set CCs (test)

<thread 1>

```
        .
LAB1:   Test&Set R1, Lock
        Branch LAB1 if R1==1
        .
        <critical section>
        .
        Reset Lock
```

<thread 2>

```
        .
LAB2:   Test&Set R1, Lock
        Branch LAB2 if R1==1
        .
        <critical section>
        .
        Reset Lock
```

Atomic Read-Modify-Write

- Many such instructions have been used in ISAs

```
Test&Set(reg,lock)
reg ← mem(lock);
mem(lock) ← 1;
```

```
Fetch&Add(reg,value,sum)
reg ← mem(sum);
mem(sum)← mem(sum)+value;
```

```
Swap(reg,opnd)
temp ← mem(opnd);
mem(opnd)← reg;
reg ← temp
```

- More-or-less equivalent
  - One can be used to implement the others
  - Implement Fetch&Add with Test&Set:

```
try:  Test&Set(lock);
      if lock == 1 go to try;
      reg ← mem(sum);
      mem(sum) ← reg+value;
      reset (lock);
```

Lock Efficiency

- Spin Locks
  - tight loop until lock is acquired

```
LAB1:  Test&Set R1, Lock
       Branch LAB1 if R1==1
```

- Inefficiencies:
  - Memory/Interconnect resources, spinning on read/writes
  - With a cache-based systems,
    writes $\Rightarrow$ lots of coherence traffic
  - Processor resource
    - not executing useful instructions

Efficient Lock Implementations

- Test&Test&Set
  - spin on check for unlock only, then try to lock
  - with cache systems, all reads can be local
    - no bus or external memory resources used

```
test_it: load       reg, mem(lock)
         branch     test_it if reg==1
lock_it: test&set   reg, mem(lock)
         branch     test_it if reg==1
```

- Test&Set with Backoff
  - Insert delay between test&set operations (not too long)
  - Each failed attempt $\Rightarrow$ longer delay
    (Like Ethernet collision avoidance)

Efficient Lock Implementations

- Solutions just given save memory/interconnect resource
  - Still waste processor resource
- Use runtime to suspend waiting process
  - Detect lock
  - Place on wait queue
  - Schedule another thread from run queue
  - When lock is released move from wait queue to run queue

Point-to-Point Synchronization

- *Can* use normal variables as flags

```
while (full ==1){}   ;spin        while (full == 0){} ;spin
a = value;                        b = value;
full = 1;                         full = 0;
```

- Assumes sequential consistency
  - Using normal variables may cause problems with relaxed consistency models
- May be better to use special opcodes for flag set/clear

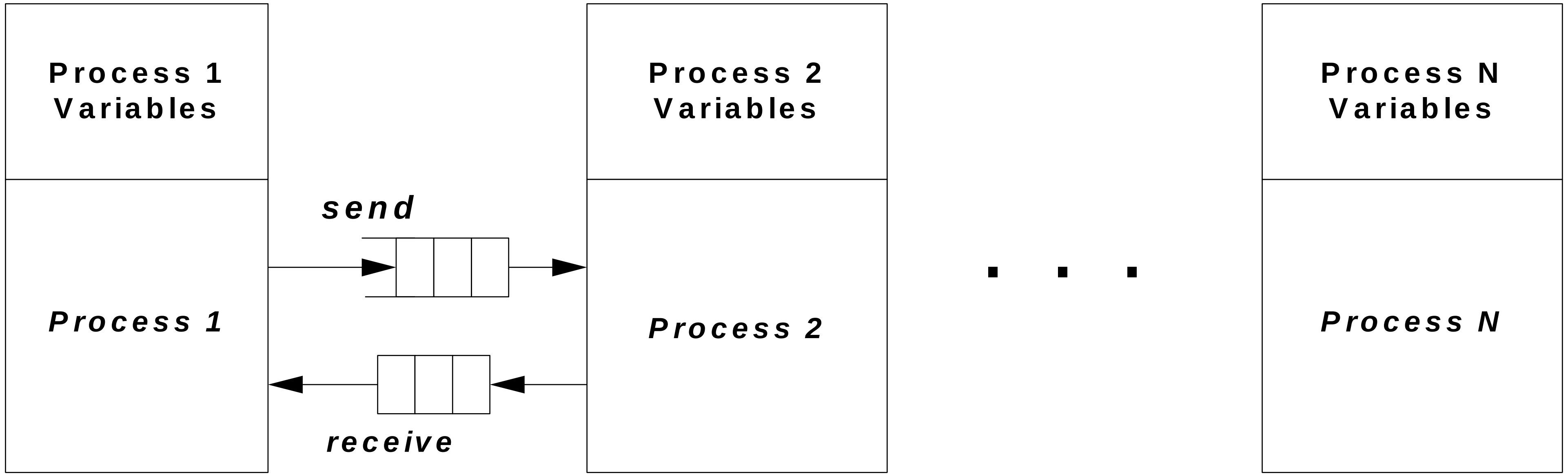
Barrier Synchronization

- Uses a lock, a counter, and a flag
  - lock for updating counter
  - flag indicates all threads have incremented counter

```
Barrier (bar_name, n) {
   Lock (bar_name.lock);
   if (bar_name.counter = 0)  bar_name.flag = 0;
   mycount = bar_name.counter++;
   Unlock (bar_name.lock);
   if (mycount == n) {
         bar_name.counter = 0;
         bar_name.flag = 1;
   }
   else  while(bar_name.flag = 0) {}; /* busy wait */
}
```

D. Message Passing Model

- Multiple processes (or threads)
- Logical data partitioning
  - No shared variables
- Message Passing
  - Threads of control communicate by sending and receiving messages
  - May be implicit in language constructs
  - More commonly explicit via API



MPI – Message Passing Interface API (Open MPI)

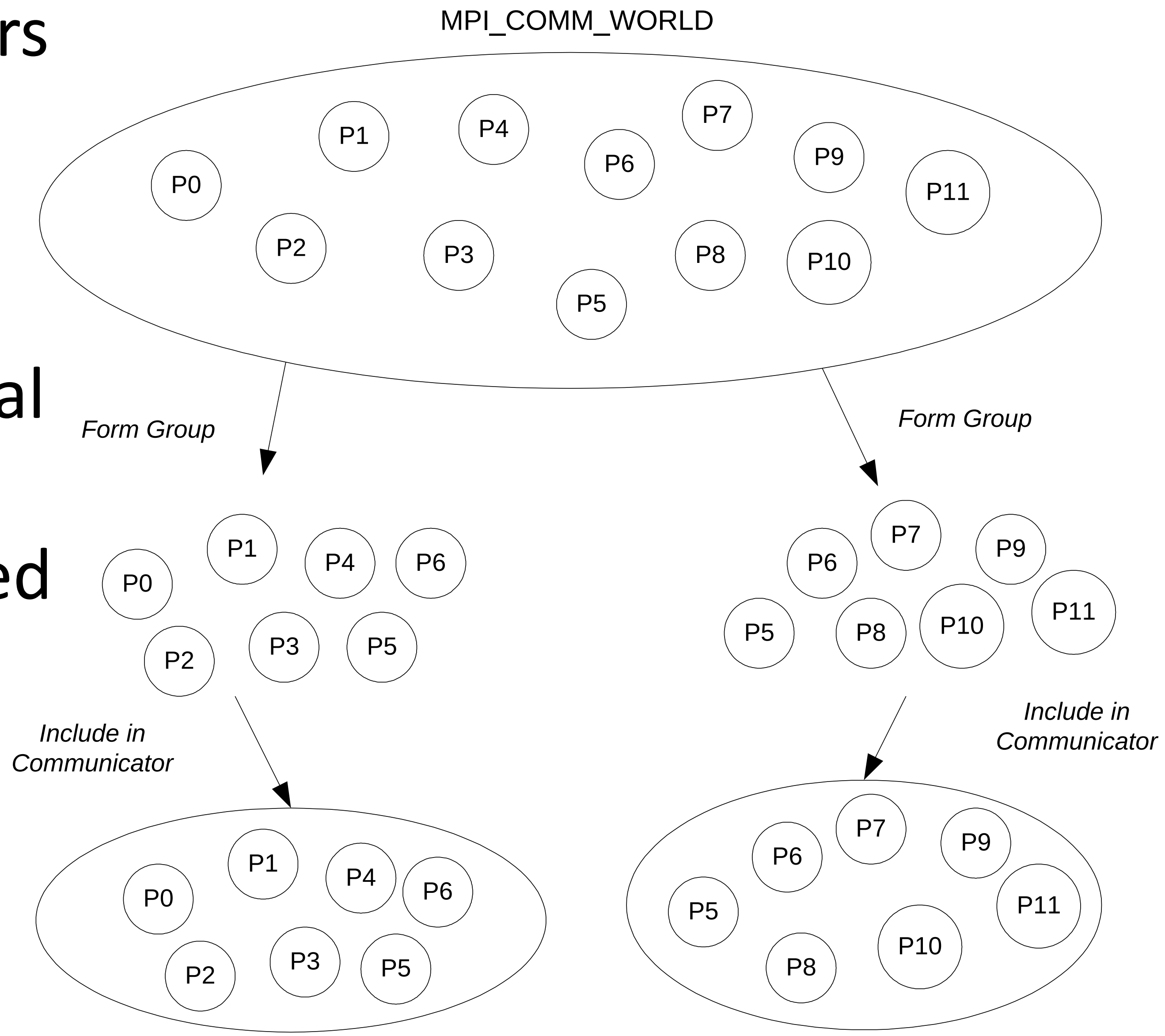
- A widely used standard
  - For a variety of distributed memory systems
    - SMP Clusters, workstation clusters, MPPs, heterogeneous systems
- Also works on Shared Memory MPs (OpenMP)
  - Easy to emulate distributed memory on shared memory HW
- Can be used with a number of high level languages

Processes and Threads

- Lots of flexibility (advantage of message passing)
  1) Multiple threads sharing an address space
  2) Multiple processes sharing an address space
  3) Multiple processes with different address spaces and different OSes
- 1) and 2) are easily implemented on shared memory hardware (with single OS)
  - Process and thread creation/management similar to shared memory
- 3) probably more common in practice
  - Process creation often external to execution environment; e.g. shell script
  - Hard for user process on one system to create process on another OS

Process Management

- Processes are given identifiers (PIDs)
  - "rank" in MPI
- Process can acquire own PID
- Operations can be conditional on PID
- Message can be sent/received via PIDs
- Organize into groups
  - For collective management and communication



Communication and Synchronization

- Combined in the message passing paradigm
  - Synchronization of messages part of communication semantics
- Point-to-point communication
  - From one process to another
- Collective communication
  - Involves groups of processes
  - e.g., broadcast

Point to Point Communication

- Use sends/receives primitives
- Send(RecProc, SendBuf,…)
  - RecProc is destination (wildcards may be used)
  - SendBuf names buffer holding message to be sent
- Receive(SendProc, RecBuf,…)
  - SendProc names sending process (wildcards may be used)
  - RecBuf names buffer where message should be placed

MPI Examples

- MPI_Send(buffer,count,type,dest,tag,comm)
  - buffer – address of data to be sent
  - count – number of data items
  - type – type of data items
  - dest – rank of the receiving process
  - tag – arbitrary programmer-defined identifier
    - tag of send and receive must match
  - comm – communicator number
- MPI_Recv(buffer,count,type,source,tag,comm,status)
  - buffer – address of data to be sent
  - count – number of data items
  - type – type of data items
  - source – rank of the sending process; may be a wildcard
  - tag – arbitrary programmer-defined identifier; may be a wildcard
    - tag of send and receive must match
  - comm – communicator number
  - status – indicates source, tag, and number of bytes transferred

Message Synchronization

- After a send or receive is executed…
  - *Has message actually been sent? or received?*
- Asynchronous vs. Synchronous
  - Higher level concept
- Blocking vs. non-Blocking
  - Lower level – depends on buffer implementation
    - *but is reflected up into the API*

Synchronous vs. Asynchronous

- Synchronous Send
  - Stall until message has actually been received
  - Implies a message acknowledgement from receiver to sender
- Synchronous Receive
  - Stall until message has actually been received
- Asynchronous Send and Receive
  - Sender and receiver can proceed regardless
  - Returns *request handle* that can be tested for message receipt
  - Request handle can be tested to see if message has been sent/received

Blocking vs. Non-Blocking

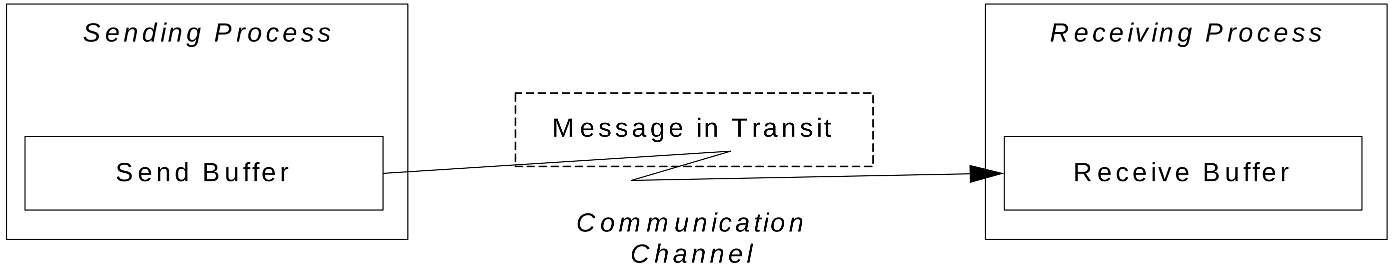
- *Blocking send* blocks if send buffer is not available for new message
- *Blocking receive* blocks if no message in its receive buffer
- Non-blocking versions don't block…
- Operation depends on buffering in implementation

Blocking vs. Non-Blocking

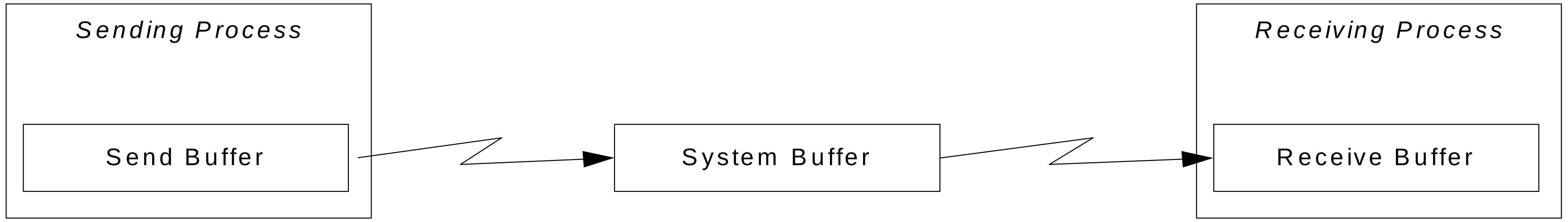
- Buffer implementations

  a) Message goes directly from sender to receiver reduces copying time

  b) Message is buffered by system in between may free up send buffer sooner (less blocking)



(a)



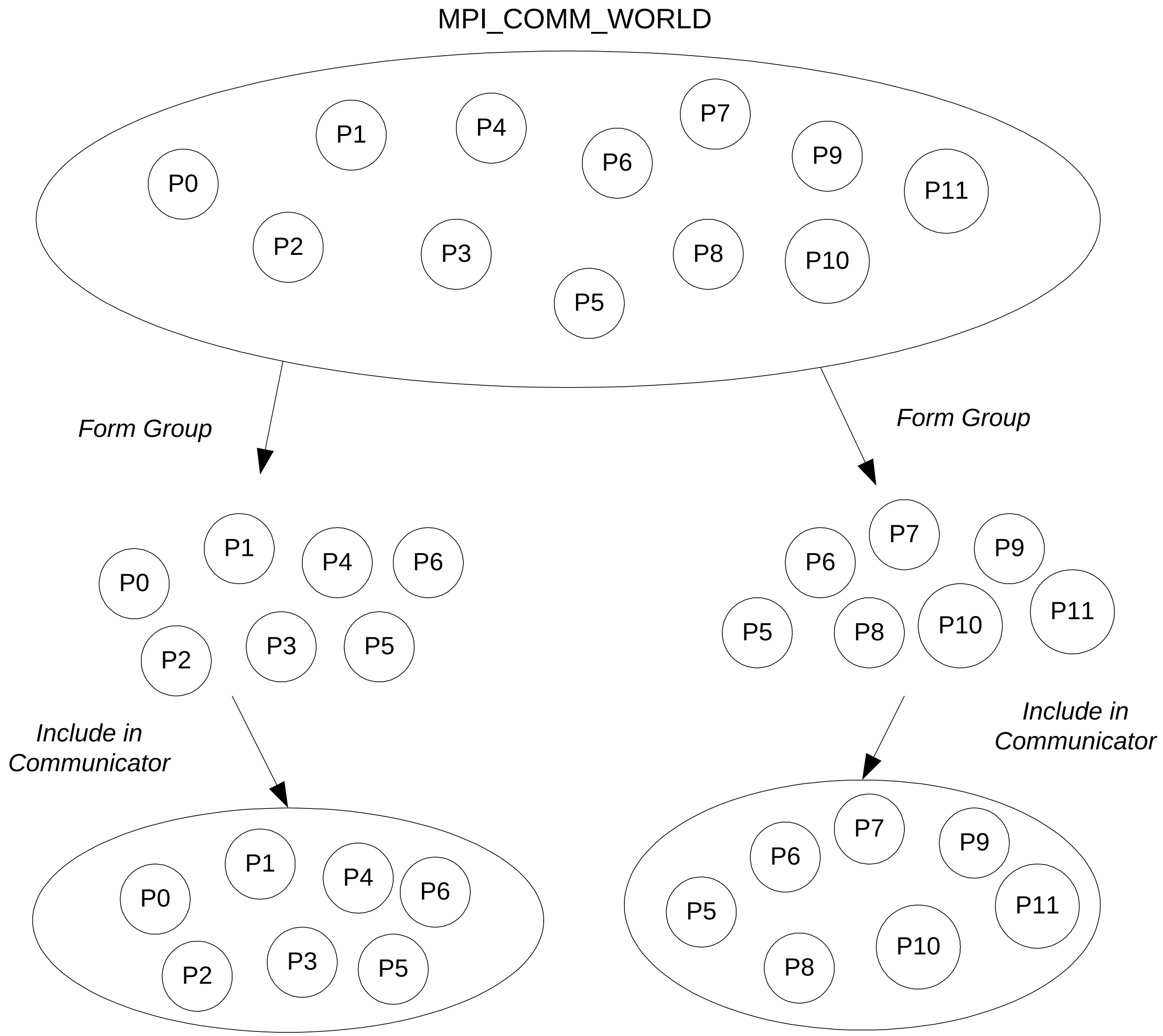
(b)

Collective Communications

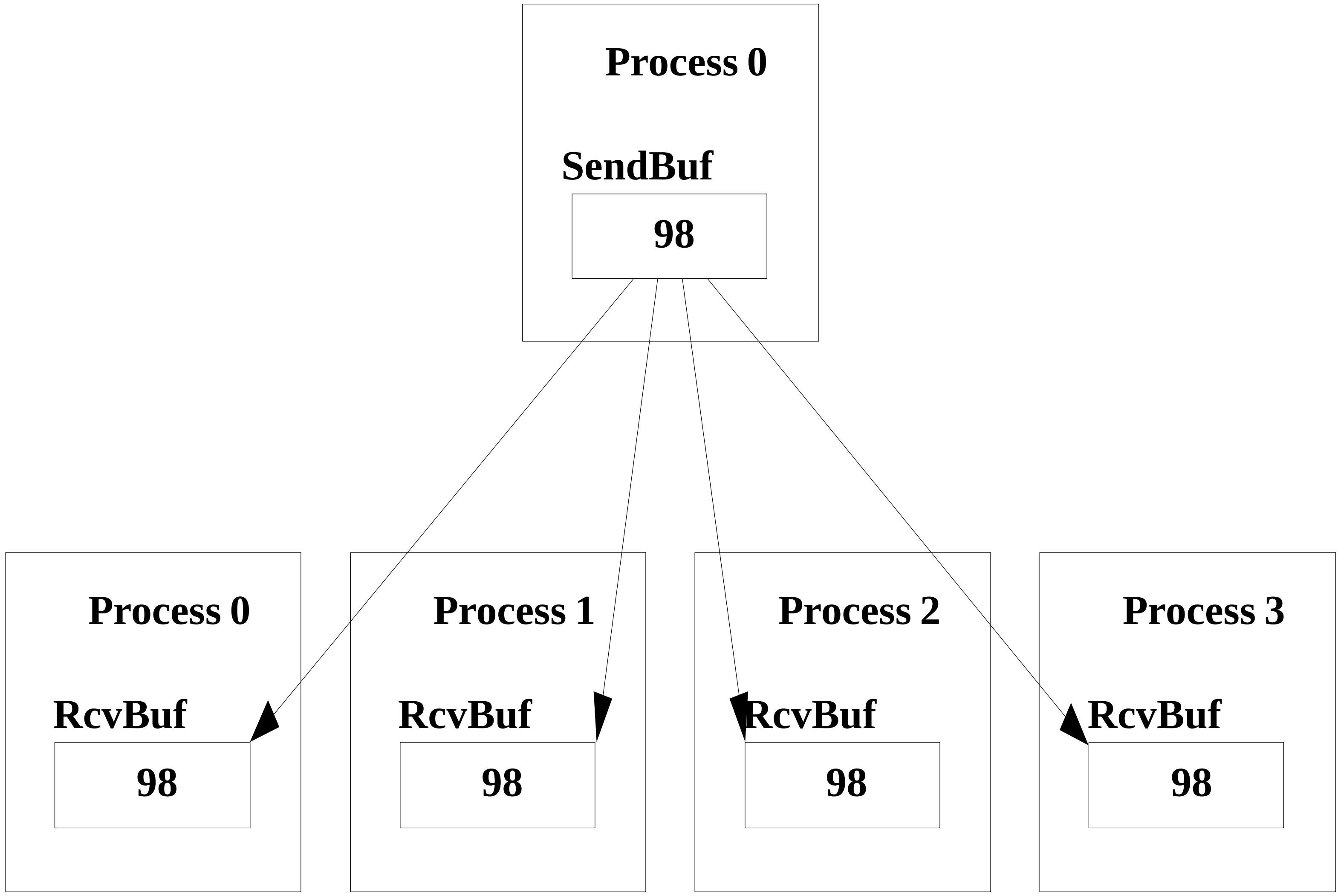
- Involve all processes within a communicator
- Blocking
- MPI_Barrier (comm)
  - Barrier synchronization
- MPI_Bcast (*buffer,count,datatype,root,comm)
  - Broadcasts from process of rank “root” to all other processes
- MPI_Scatter (*sendbuf,sendcnt,sendtype,*recvbuf,
  ...... recvcnt,recvtype,root,comm)
  - Sends different messages to each process in a group
- MPI_Gather (*sendbuf,sendcnt,sendtype,*recvbuf,
  ...... recvcount,recvtype,root,comm)
  - Gathers different messages from each process in a group
- Also reductions

Communicators and Groups

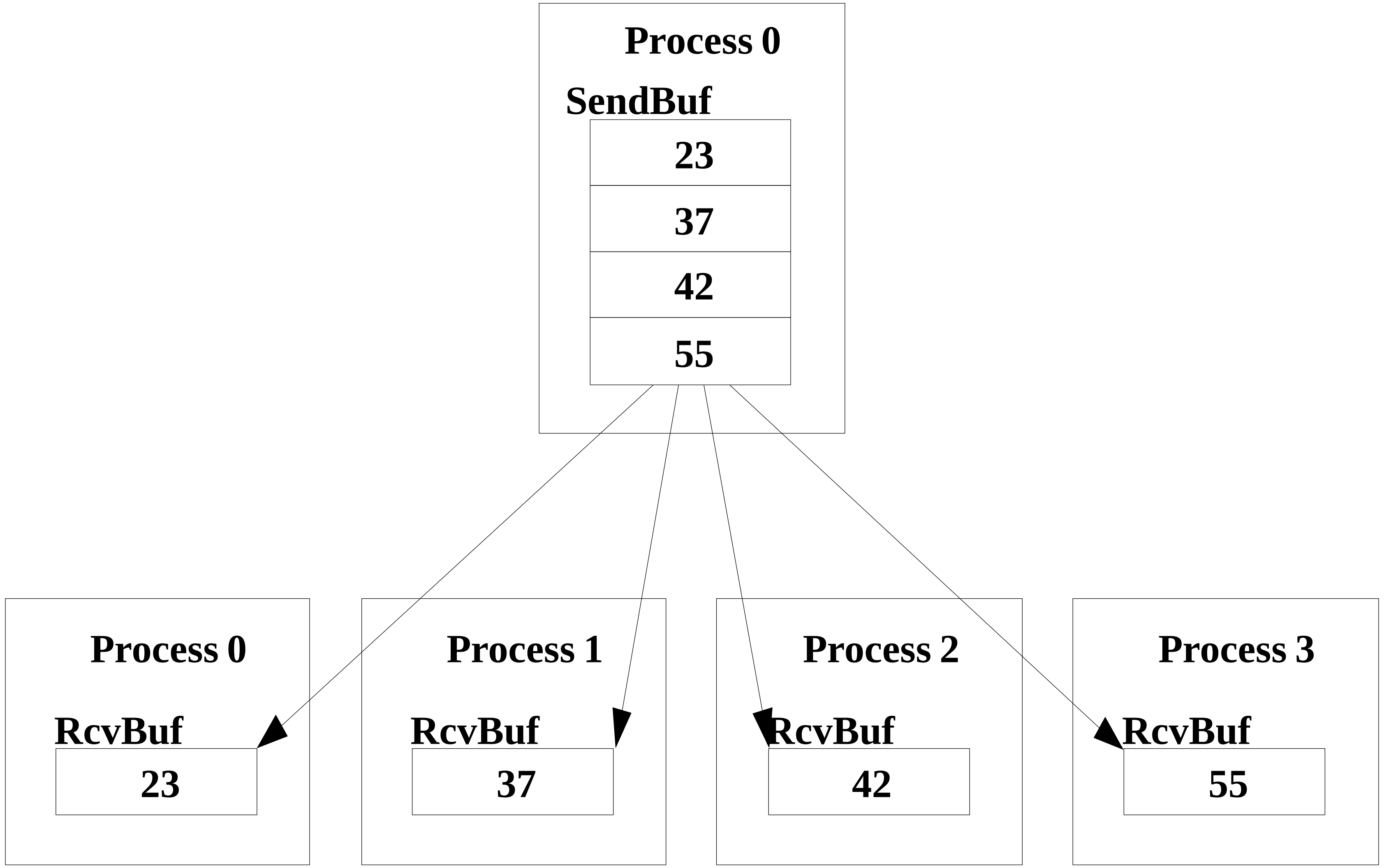
- Define collections of processes that may communicate
  - Often specified in message argument
  - MPI_COMM_WORLD – predefined communicator that contains all processes



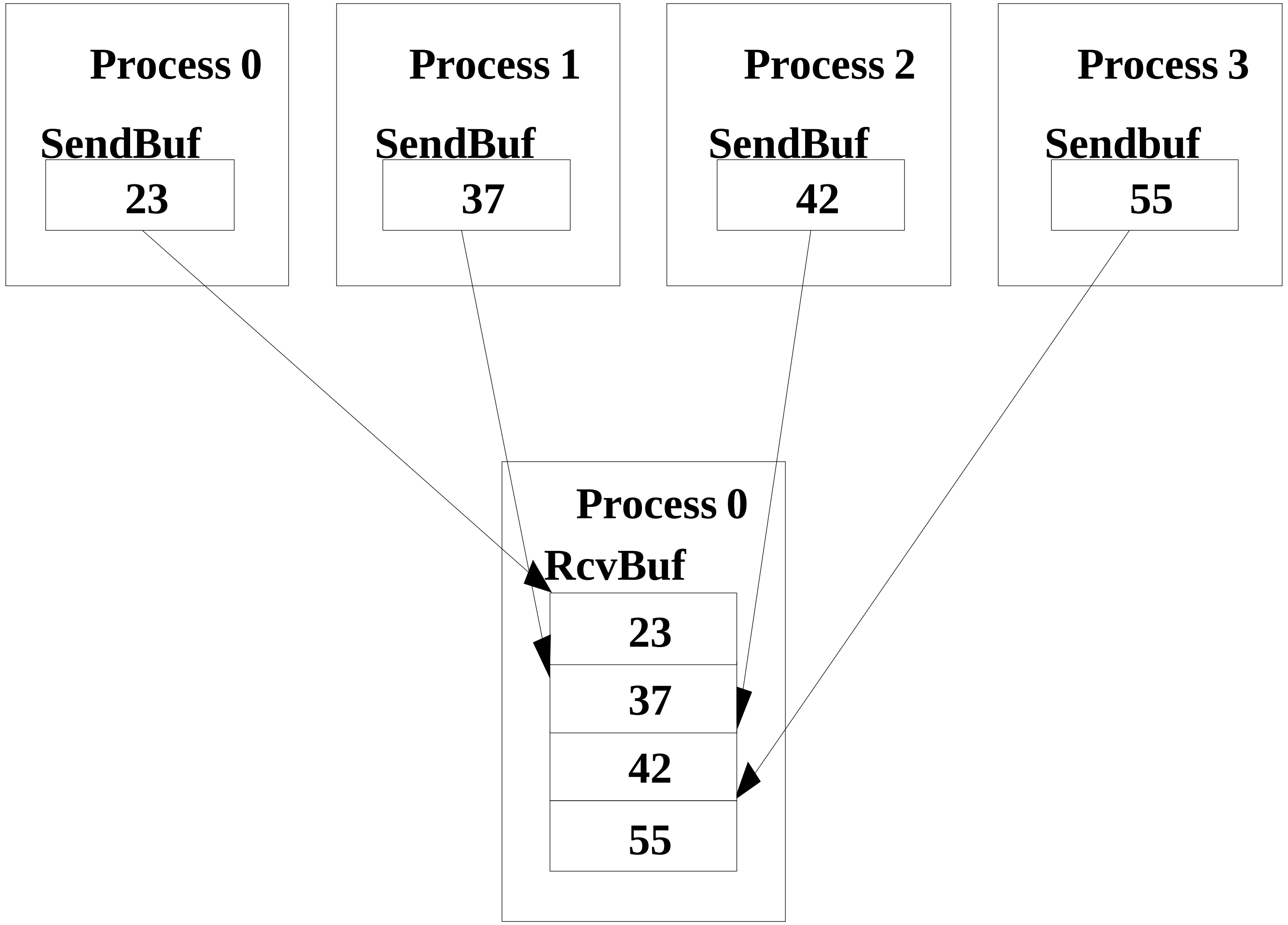
Broadcast Example



Scatter Example



Gather Example



Message Passing Implementation

- At the ABI and ISA level
  - No special support (beyond that needed for shared memory)
  - Most of the implementation is in the runtime
    - user-level libraries
  - Makes message passing relatively portable
- Three implementation models
  1) Multiple threads sharing an address space
  2) Multiple processes sharing an address space
  3) Multiple processes with non-shared address space (and different OSes)

Multiple Threads Sharing Address Space

- Runtime manages buffering and tracks communication
  - Communication via normal loads and stores using shared memory
- Example: Send/Receive
  - Send calls runtime, runtime posts availability of message in runtime-managed table
  - Receive calls runtime, runtime checks table, finds message
  - Runtime copies data from send buffer to store buffer via load/stores
- Fast/Efficient Implementation
  - May even be advantageous over shared memory paradigm
    - considering portability, software engineering aspects
  - Can use runtime thread scheduling
  - Problem with protecting private memories and runtime data area

Multiple Processes Sharing Address Space

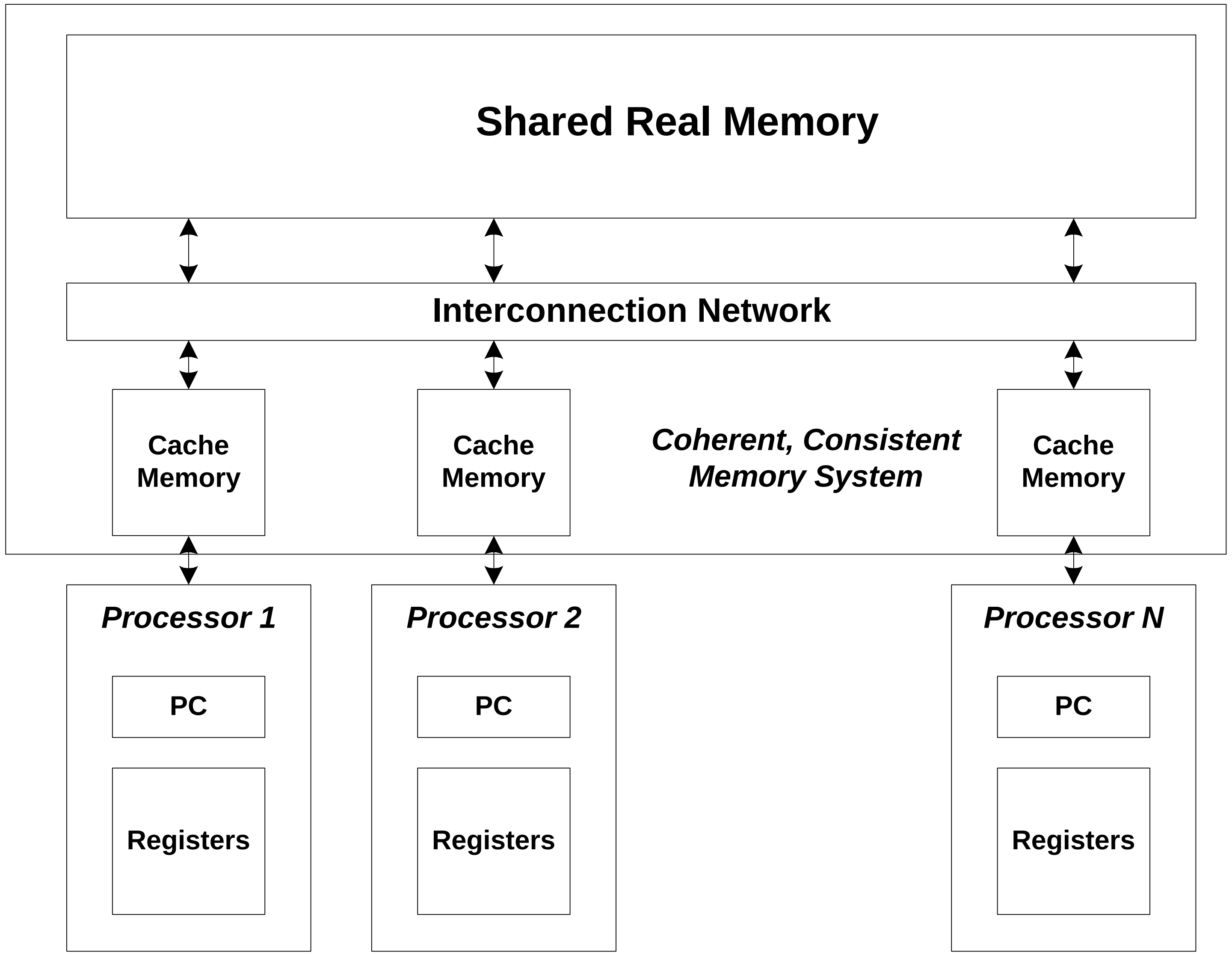
- Similar to multiple threads sharing address space
- Would rely on kernel scheduling
- May offer more memory protection
  - With intermediate runtime buffering
  - User processes can not access others' private memory

Multiple Processes with Non-Shared Address Space

- Most common implementation
- Communicate via networking hardware
- Send/receive to runtime
  - Runtime converts to OS (network) calls
- Relatively high overhead
  - Most HPC systems use special low-latency, high-bandwidth networks
  - Buffering in receiver's runtime space may save some overhead for receive (doesn't require OS call)

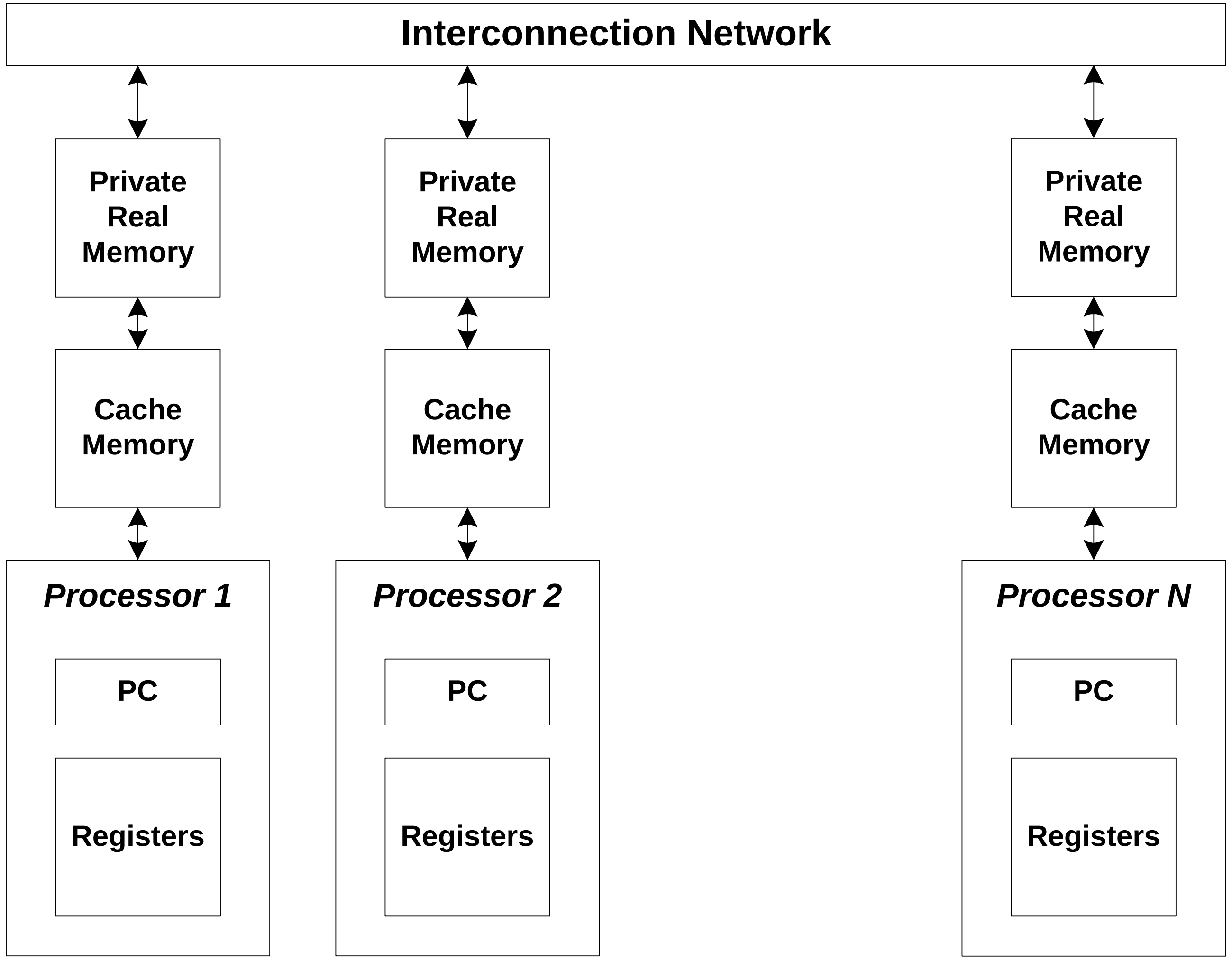
At the ISA Level: Shared Memory Systems

- Multiple processors
- Architected shared virtual memory
- Architected Synchronization instructions
- Architected Cache Coherence
- Architected Memory Consistency



At the ISA Level: Message Passing Systems

- Multiple processors
- Shared or non-shared real memory (multi-computers)
- Limited ISA support (if any)
  - An advantage of distributed memory systems --Just connect a bunch of small computers
  - Some implementations may use shared memory managed by runtime



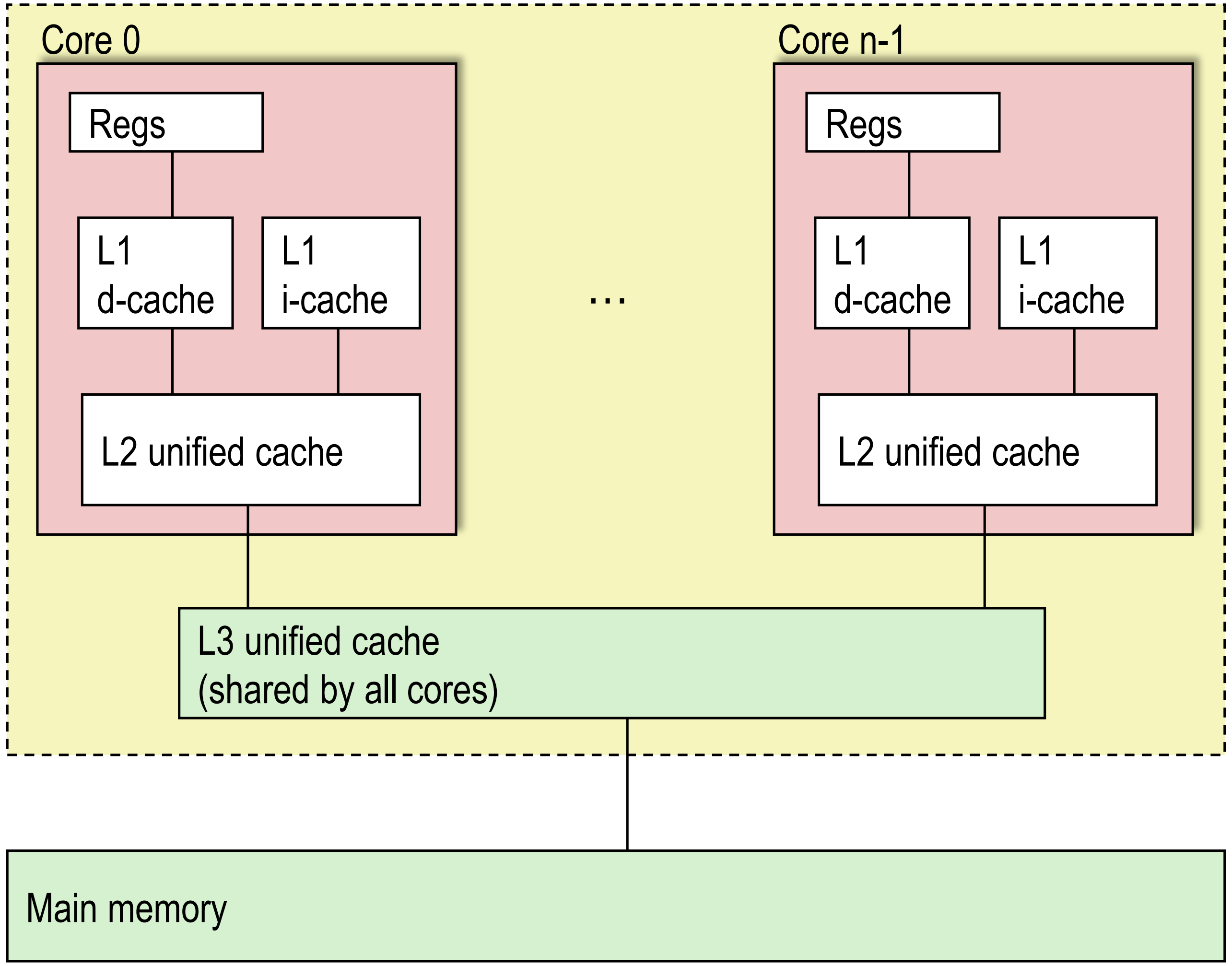
E. Thread Level Parallelism Examples

- **Parallel Computing Hardware**
  - Multicore
    - Multiple separate processors on single chip
  - Hyperthreading
    - Efficient execution of multiple threads on single core
- **Thread-Level Parallelism**
  - Splitting program into independent tasks
    - Example 1: **Parallel summation**
  - Divide-and conquer parallelism
    - Example 2: **Parallel quicksort**
- **Consistency Models**
  - What happens when multiple threads are reading & writing shared state

Exploiting parallel execution

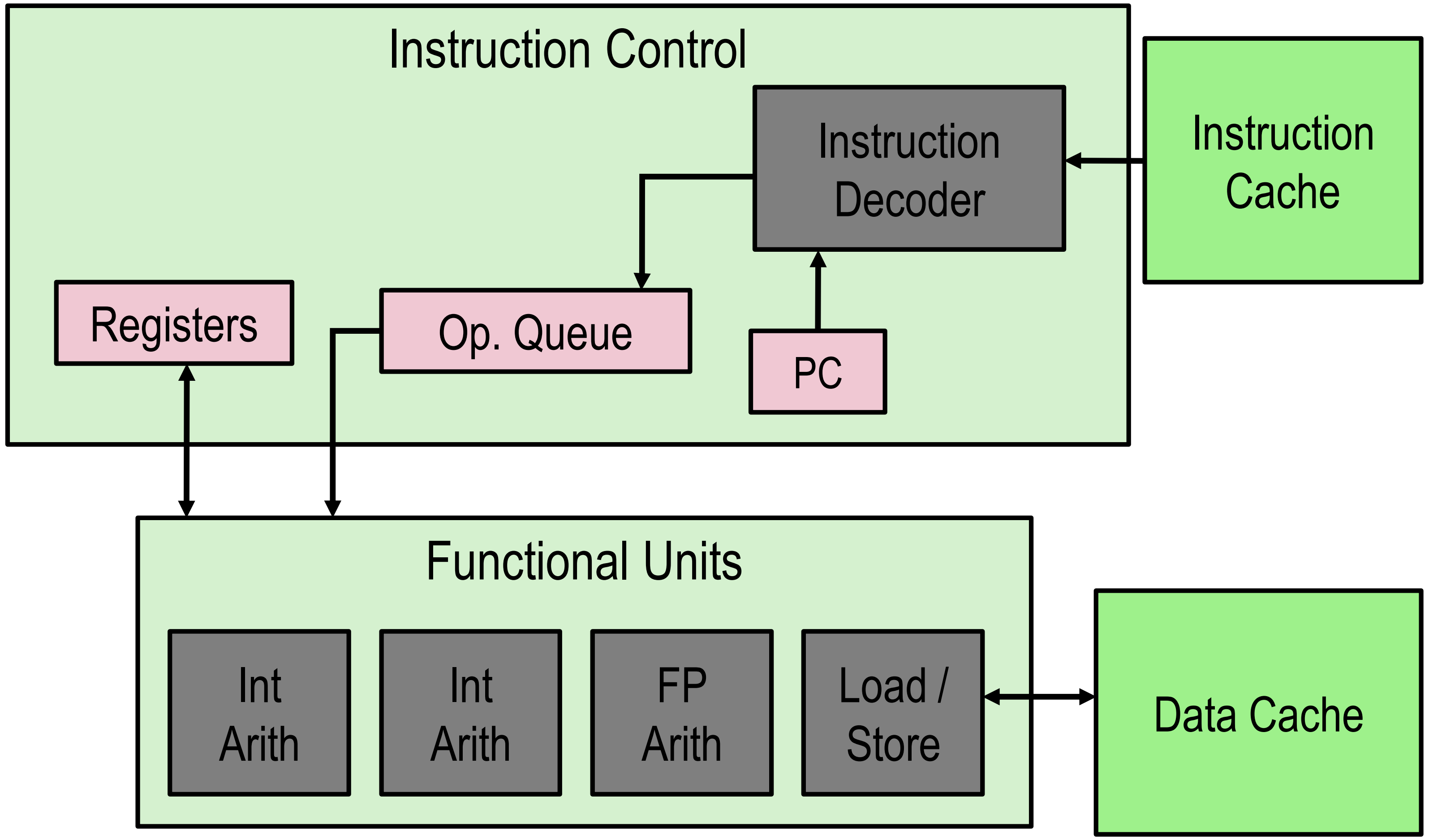
- **So far, we've used threads to deal with I/O delays**
  - e.g., one thread per client to prevent one from delaying another
- **Multi-core/Hyperthreaded CPUs offer another opportunity**
  - Spread work over threads executing in parallel
  - Happens automatically, if many independent tasks
    - e.g., running many applications or serving many clients
  - Can also write code to make one big task go faster
    - by organizing it as multiple parallel sub-tasks

Typical Multicore Processor



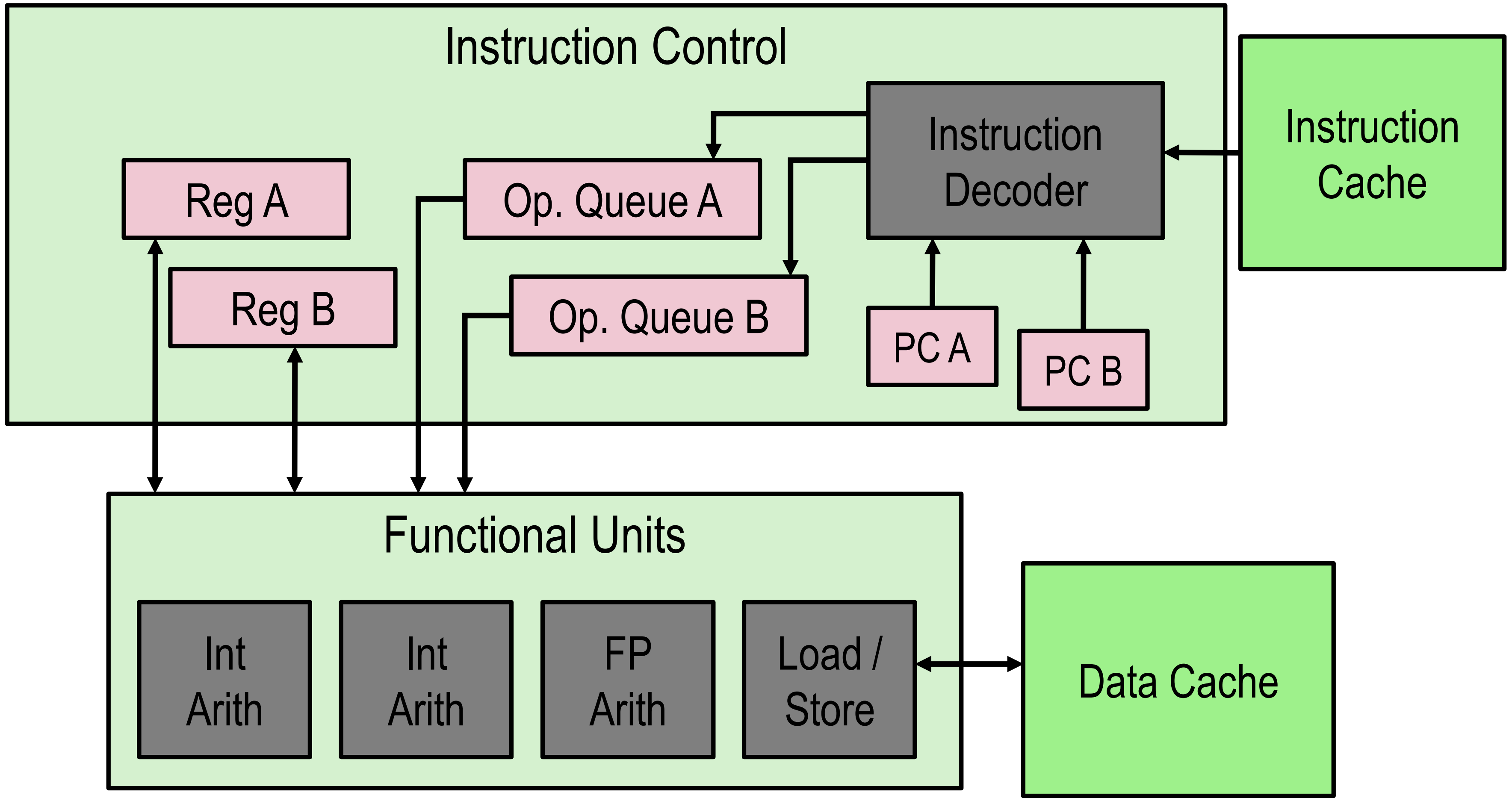
- **Multiple processors operating with coherent view of memory**

Out-of-Order Processor Structure



- **Instruction control dynamically converts program into stream of operations**
- **Operations mapped onto functional units to execute in parallel**

Hyperthreading Implementation



- Replicate enough instruction control to process K instruction streams
- K copies of all registers
- Share functional units

Benchmark Machine

- **Get data about machine from /proc/cpuinfo**
- **Shark Machines**
  - Intel Xeon E5520 @ 2.27 GHz
  - Nehalem, ca. 2010
  - 8 Cores
  - Each can do 2x hyperthreading

Example 1: Parallel Summation

- **Sum numbers *0, …, n-1***
  - Should add up to *((n-1)*n)/2*
- **Partition values *1, …, n-1* into *t* ranges**
  - $\lfloor n/t \rfloor$ values in each range
  - Each of *t* threads processes 1 range
  - For simplicity, assume *n* is a multiple of *t*
- **Let's consider different ways that multiple threads might work on their assigned ranges in parallel**

First attempt: psum-mutex

- **Simplest approach: Threads sum into a global variable protected by a semaphore mutex.**

```
void *sum_mutex(void *vargp); /* Thread routine */

/* Global shared variables */
long gsum = 0;                /* Global sum */
long nelems_per_thread;       /* Number of elements to sum */
sem_t mutex;                  /* Mutex to protect global sum */

int main(int argc, char **argv)
{
    long i, nelems, log_nelems, nthreads, myid[MAXTHREADS];
    pthread_t tid[MAXTHREADS];

     /* Get input arguments */
    nthreads = atoi(argv[1]);
    log_nelems = atoi(argv[2]);
    nelems = (1L << log_nelems);
    nelems_per_thread = nelems / nthreads;
    sem_init(&mutex, 0, 1);
```

psum-mutex.c

psum-mutex (cont)

- **Simplest approach: Threads sum into a global variable protected by a semaphore mutex.**

```
    /* Create peer threads and wait for them to finish */
    for (i = 0; i < nthreads; i++) {
        myid[i] = i;
        Pthread_create(&tid[i], NULL, sum_mutex, &myid[i]);
    }
    for (i = 0; i < nthreads; i++)
        Pthread_join(tid[i], NULL);

    /* Check final answer */
    if (gsum != (nelems * (nelems-1))/2)
        printf("Error: result=%ld\n", gsum);

    exit(0);
}
```

psum-mutex.c

psum-mutex Thread Routine

- **Simplest approach: Threads sum into a global variable protected by a semaphore mutex.**

```
/* Thread routine for psum-mutex.c */
void *sum_mutex(void *vargp)
{
    long myid = *((long *)vargp);          /* Extract thread ID */
    long start = myid * nelems_per_thread; /* Start element index */
    long end = start + nelems_per_thread;  /* End element index */
    long i;

    for (i = start; i < end; i++) {
        P(&mutex);
        gsum += i;
        V(&mutex);
    }
    return NULL;
}
```

psum-mutex.c

`psum-mutex` Performance

- **Shark machine with 8 cores, $n=2^{31}$**

Threads (Cores)	**1 (1)**	**2 (2)**	**4 (4)**	**8 (8)**	**16 (8)**
psum-mutex (secs)	51	456	790	536	681

- **Nasty surprise:**
  - Single thread is very slow
  - Gets slower as we use more cores

Next Attempt: `psum-array`

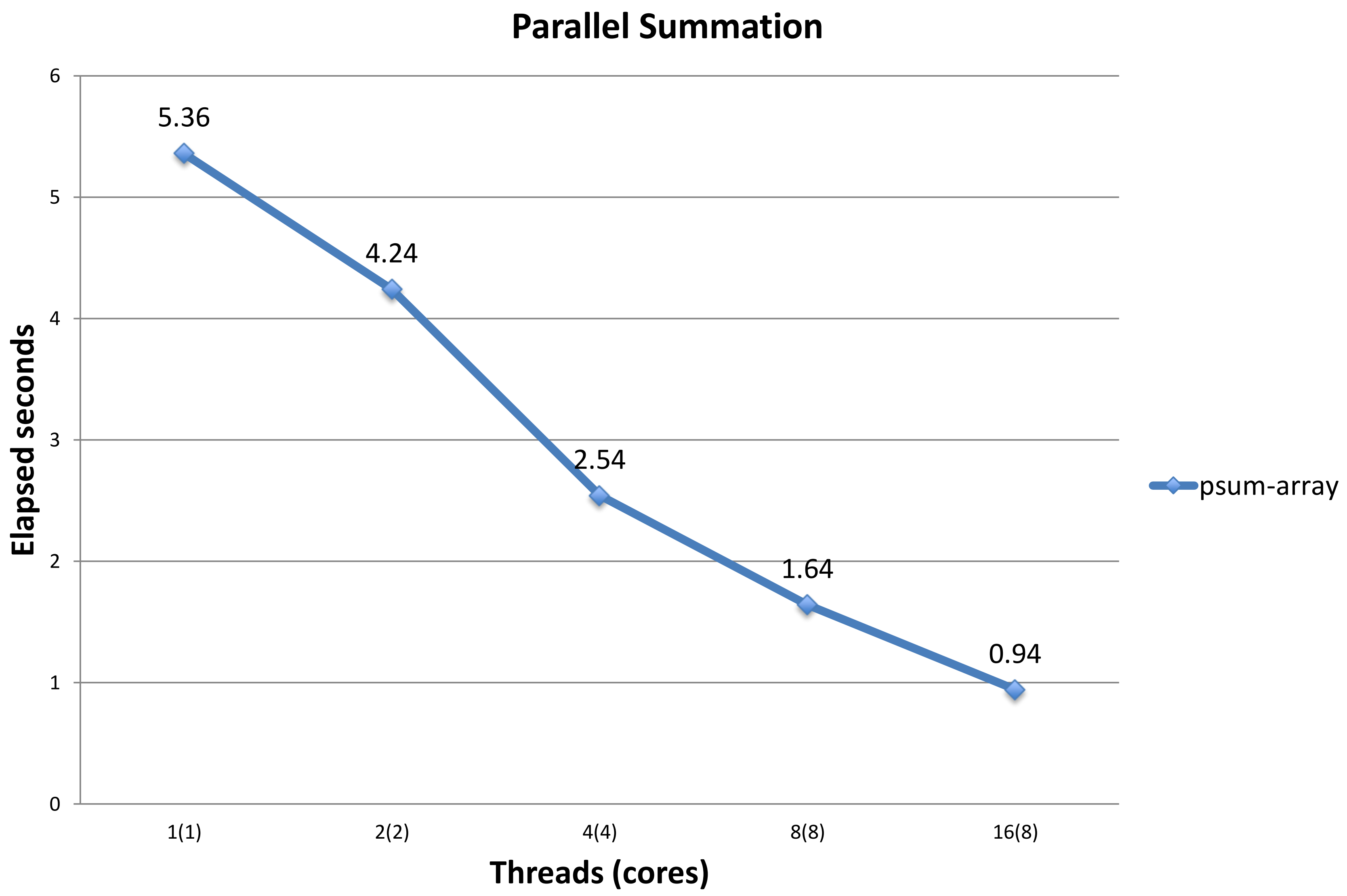
- **Peer thread `i` sums into global array element `psum[i]`**
- **Main waits for theads to finish, then sums elements of `psum`**
- **Eliminates need for mutex synchronization**

```
/* Thread routine for psum-array.c */
void *sum_array(void *vargp)
{
    long myid = *((long *)vargp);          /* Extract thread ID */
    long start = myid * nelems_per_thread; /* Start element index */
    long end = start + nelems_per_thread;  /* End element index */
    long i;

    for (i = start; i < end; i++) {
        psum[myid] += i;
    }
    return NULL;
}
```

psum-array.c

psum-array Performance

- **Orders of magnitude faster than psum-mutex**



Next Attempt: `psum-local`

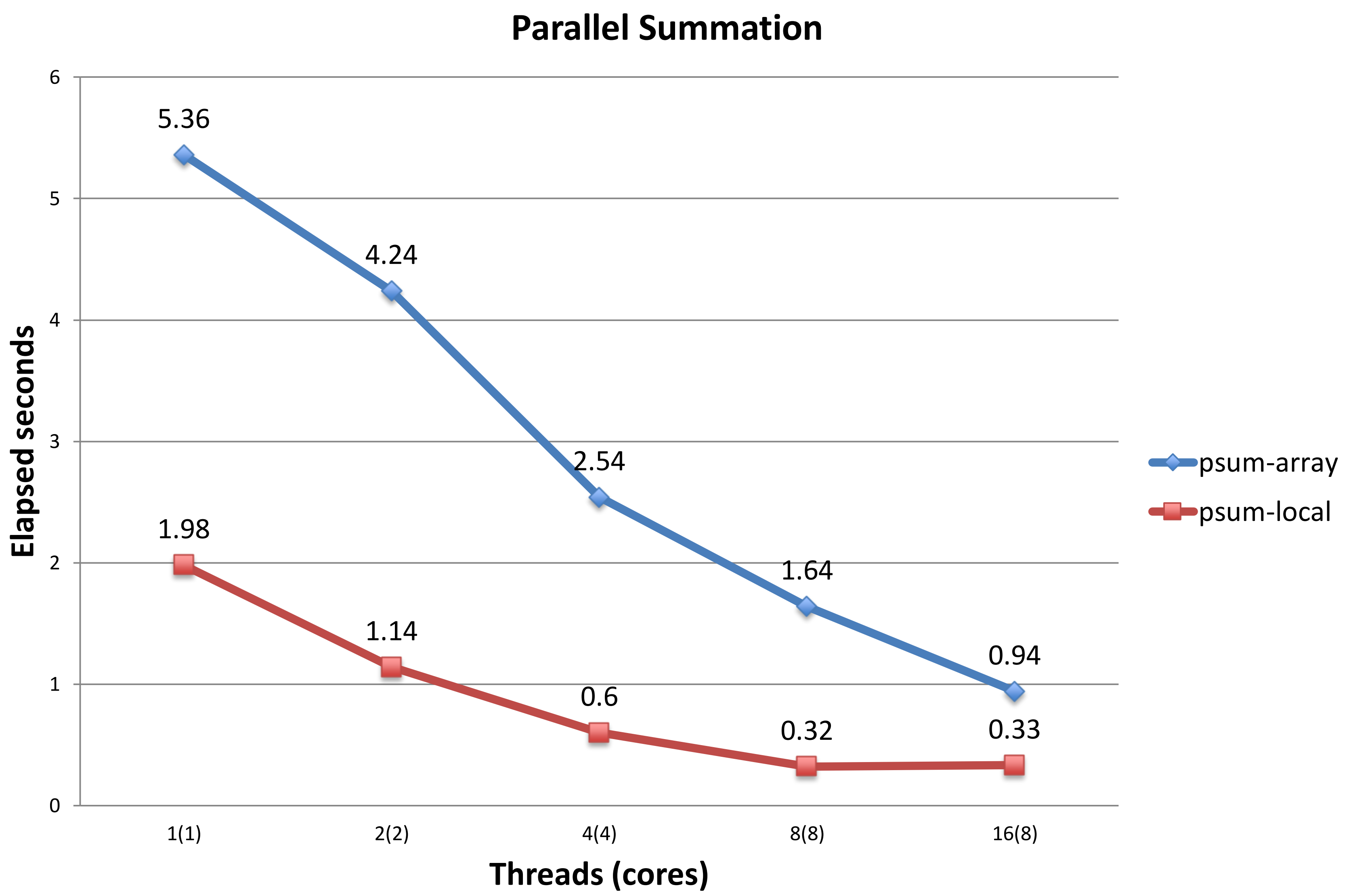
- **Reduce memory references by having peer thread i sum into a local variable (register)**

```
/* Thread routine for psum-local.c */
void *sum_local(void *vargp)
{
    long myid = *((long *)vargp);          /* Extract thread ID */
    long start = myid * nelems_per_thread; /* Start element index */
    long end = start + nelems_per_thread;  /* End element index */
    long i, sum = 0;

    for (i = start; i < end; i++) {
        sum += i;
    }
    psum[myid] = sum;
    return NULL;
}
```

psum-local.c

psum-local Performance

- Significantly faster than `psum-array`



Characterizing Parallel Program Performance

- ***p* processor cores, T_k is the running time using *k* cores**
- ***Def.* Speedup: $S_p = T_1 / T_p$**
  - S_p is *relative speedup* if T_1 is running time of parallel version of the code running on 1 core.
  - S_p is *absolute speedup* if T_1 is running time of sequential version of code running on 1 core.
  - Absolute speedup is a much truer measure of the benefits of parallelism.
- ***Def.* Efficiency: $E_p = S_p /p = T_1/(pT_p)$**
  - Reported as a percentage in the range (0, 100].
  - Measures the overhead due to parallelization

Performance of `psum-local`

Threads (t)	1	2	4	8	16
Cores (p)	1	2	4	8	8
Running time (T_p)	1.98	1.14	0.60	0.32	0.33
Speedup (S_p)	1	1.74	3.30	6.19	6.00
Efficiency (E_p)	100%	87%	82%	77%	75%

- **Efficiencies OK, not great**
- **Our example is easily parallelizable**
- **Real codes are often much harder to parallelize**
  - e.g., parallel quicksort later in this lecture

Amdahl's Law

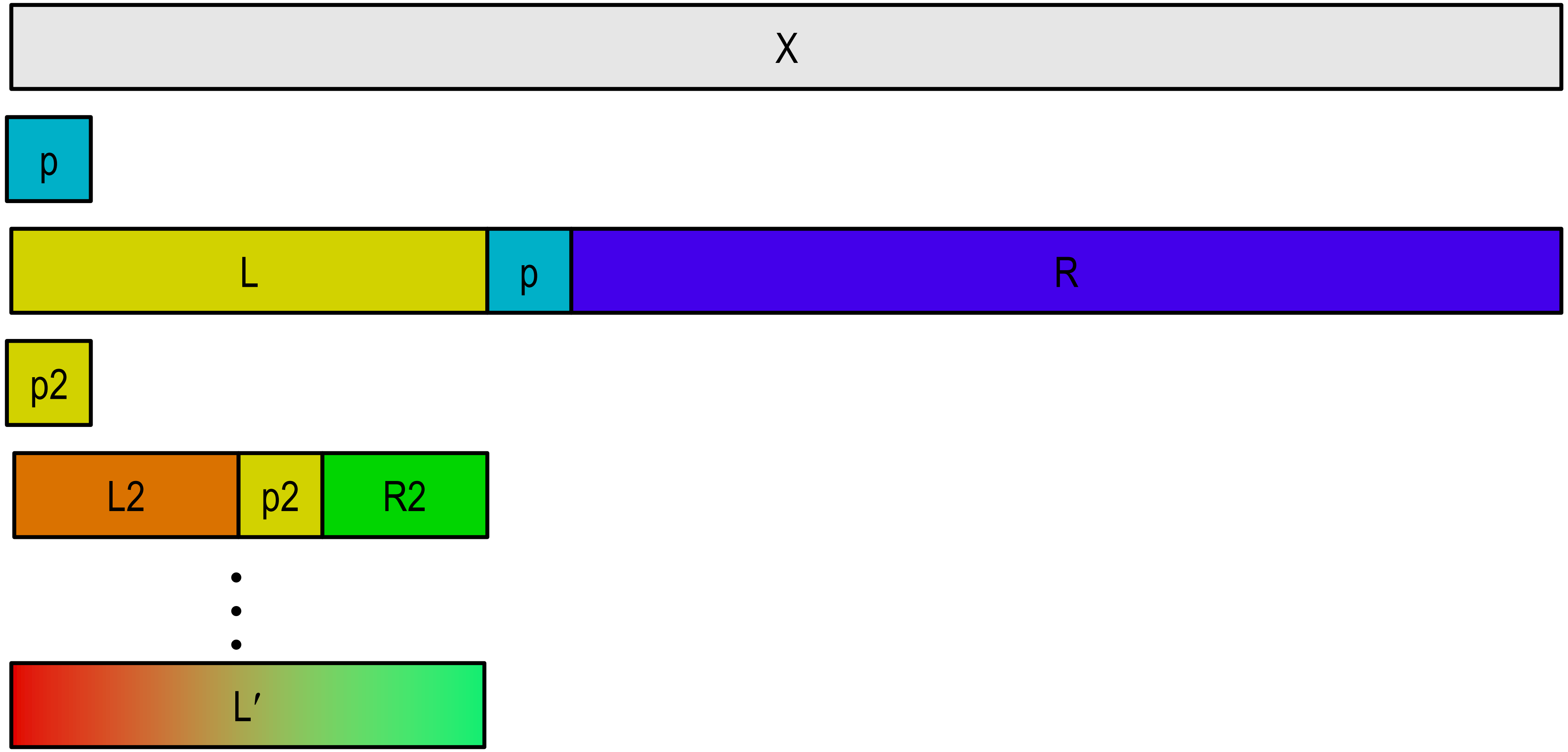
- Gene Amdahl (Nov. 16, 1922 – Nov. 10, 2015)

- **Captures the difficulty of using parallelism to speed things up.**
- **Overall problem**
  - T Total sequential time required
  - p Fraction of total that can be sped up ($0 \leq p \leq 1$)
  - k Speedup factor
- **Resulting Performance**
  - $T_k = pT/k + (1-p)T$
    - Portion which can be sped up runs k times faster
    - Portion which cannot be sped up stays the same
  - Least possible running time:
    - $k = \infty$
    - $T_\infty = (1-p)T$

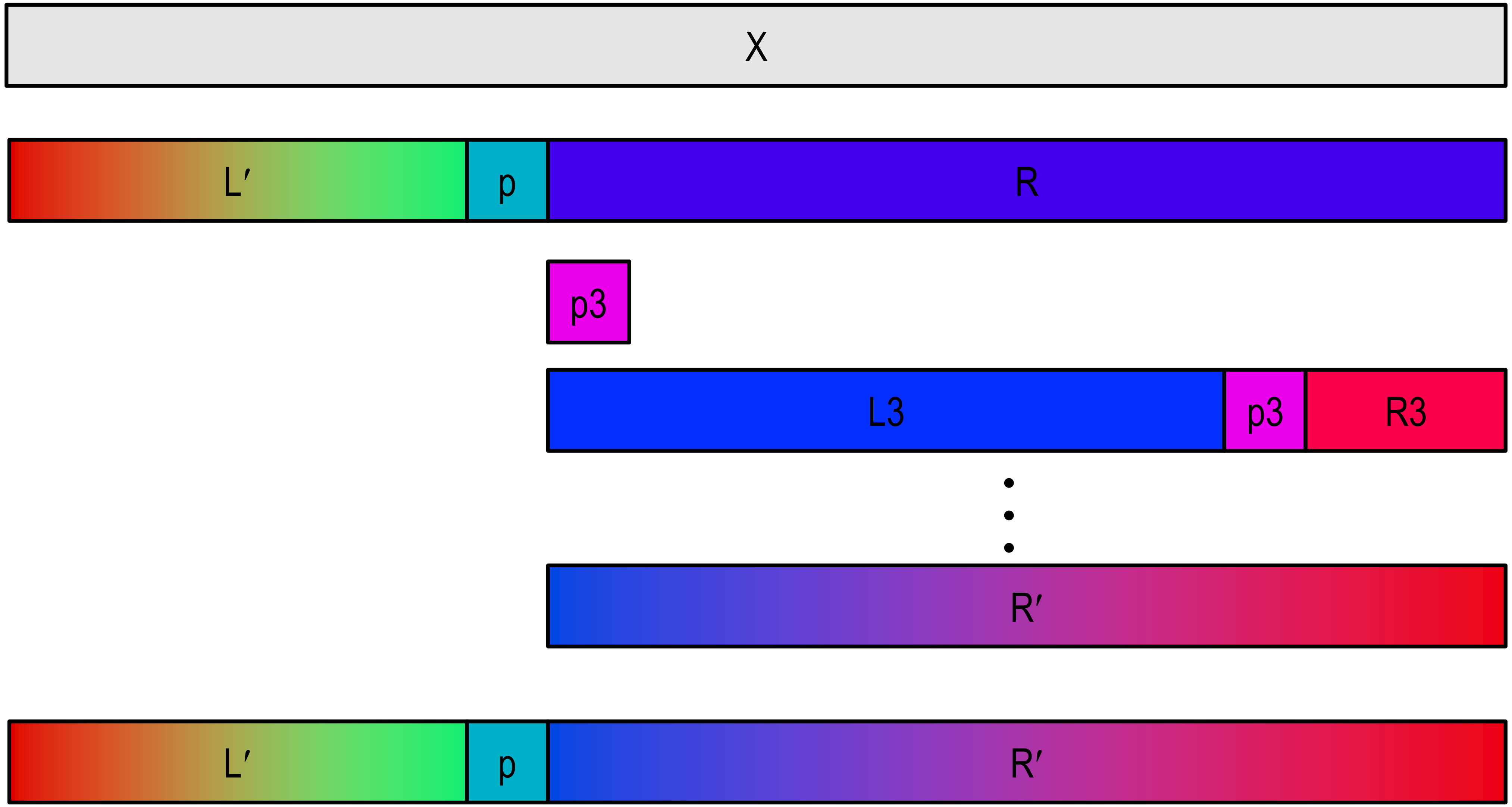
A More Substantial Example: Sort

- **Sort set of N random numbers**
- **Multiple possible algorithms**
  - Use parallel version of quicksort
- **Sequential quicksort of set of values X**
  - Choose "pivot" p from X
  - Rearrange X into
    - L: Values $\leq$ p
    - R: Values $\geq$ p
  - Recursively sort L to get L′
  - Recursively sort R to get R′
  - Return L′ : p : R′

Sequential Quicksort Visualized



Sequential Quicksort Visualized



Sequential Quicksort Code

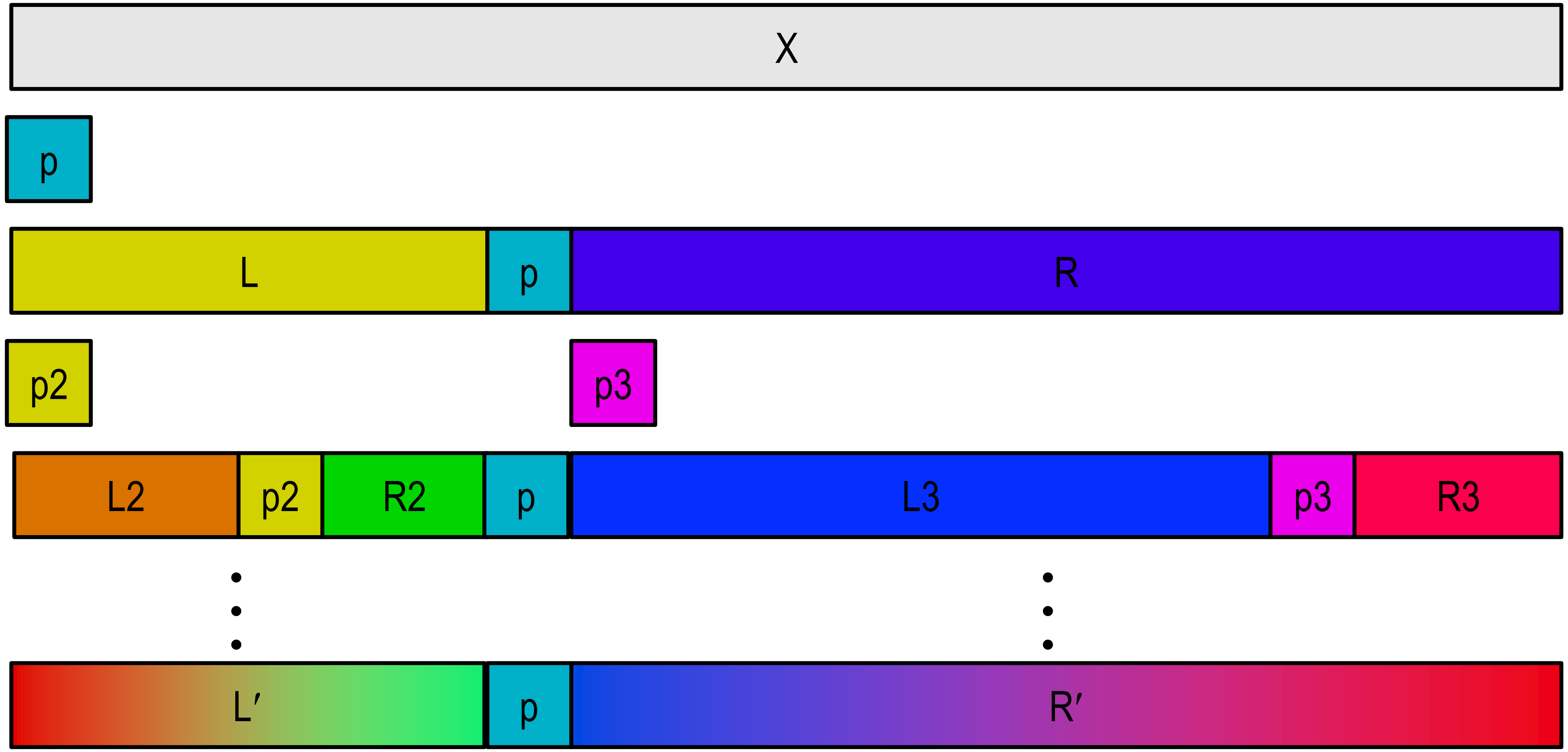
```
void qsort_serial(data_t *base, size_t nele) {
  if (nele <= 1)
    return;
  if (nele == 2) {
    if (base[0] > base[1])
      swap(base, base+1);
    return;
  }

  /* Partition returns index of pivot */
  size_t m = partition(base, nele);
  if (m > 1)
    qsort_serial(base, m);
  if (nele-1 > m+1)
    qsort_serial(base+m+1, nele-m-1);
}
```

- **Sort nele elements starting at base**
  - Recursively sort L or R if has more than one element

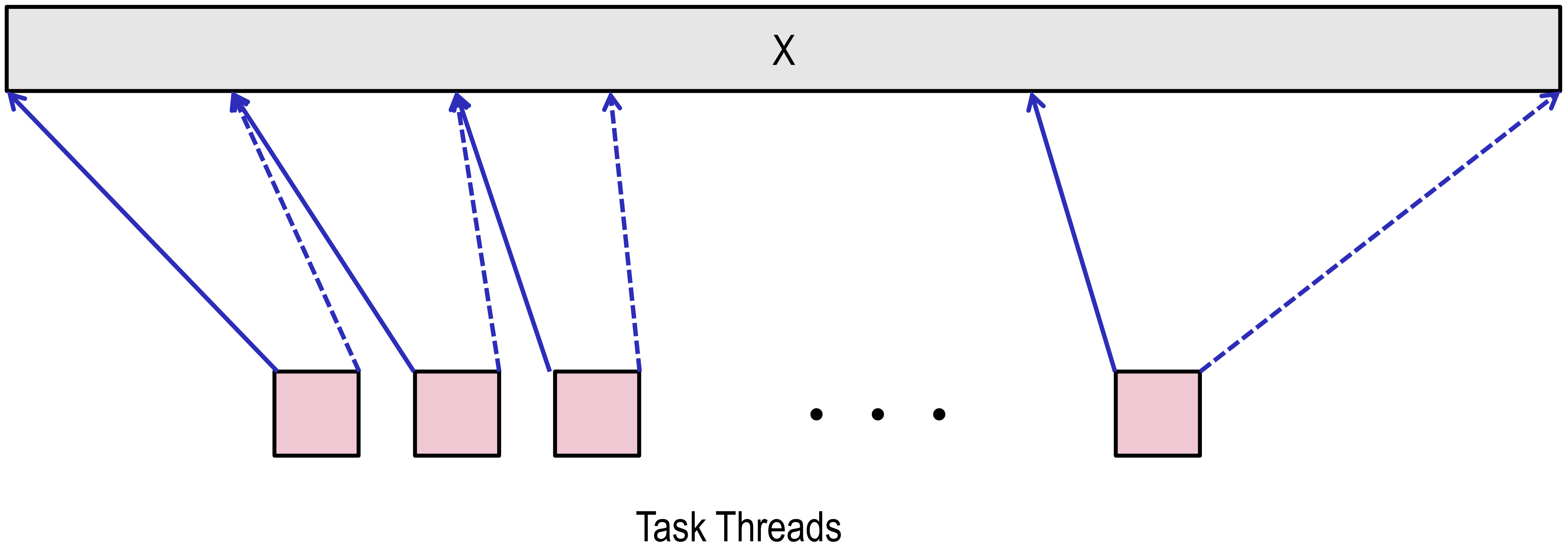
Parallel Quicksort

- **Parallel quicksort of set of values X**
  - If N ≤ Nthresh, do sequential quicksort
  - Else
    - Choose "pivot" p from X
    - Rearrange X into
      - L: Values ≤ p
      - R: Values ≥ p
    - Recursively spawn separate threads
      - Sort L to get L′
      - Sort R to get R′
    - Return L′ : p : R′

Parallel Quicksort Visualized

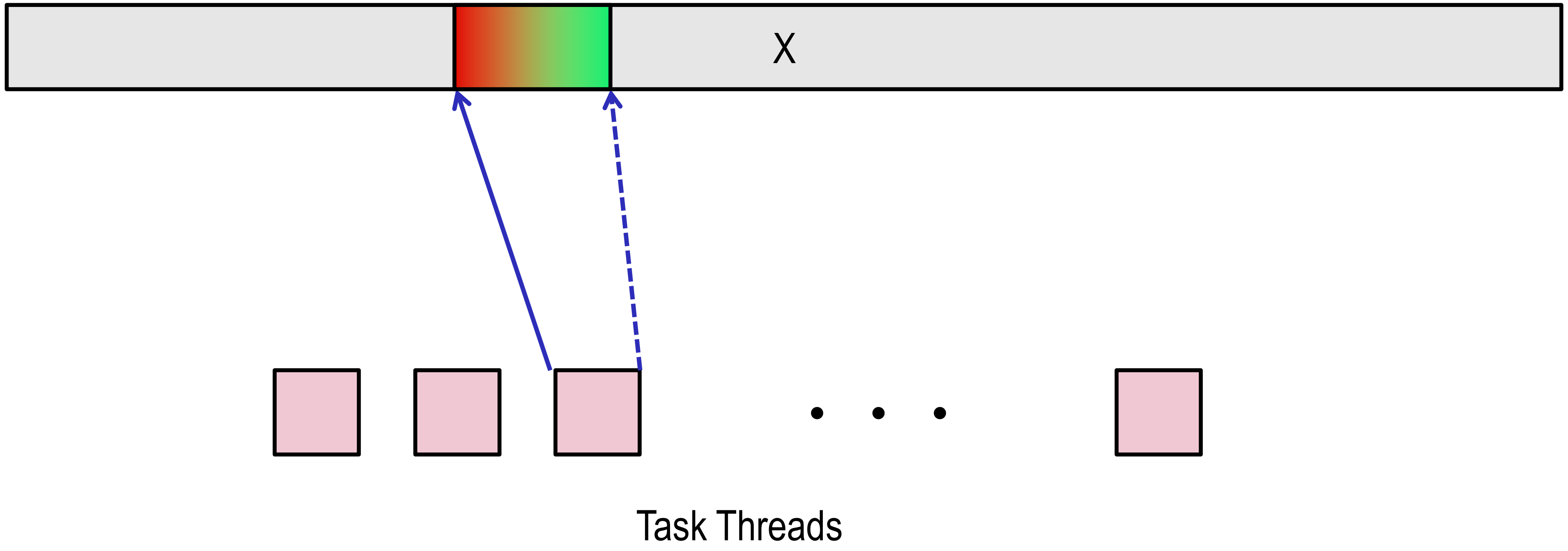


Thread Structure: Sorting Tasks



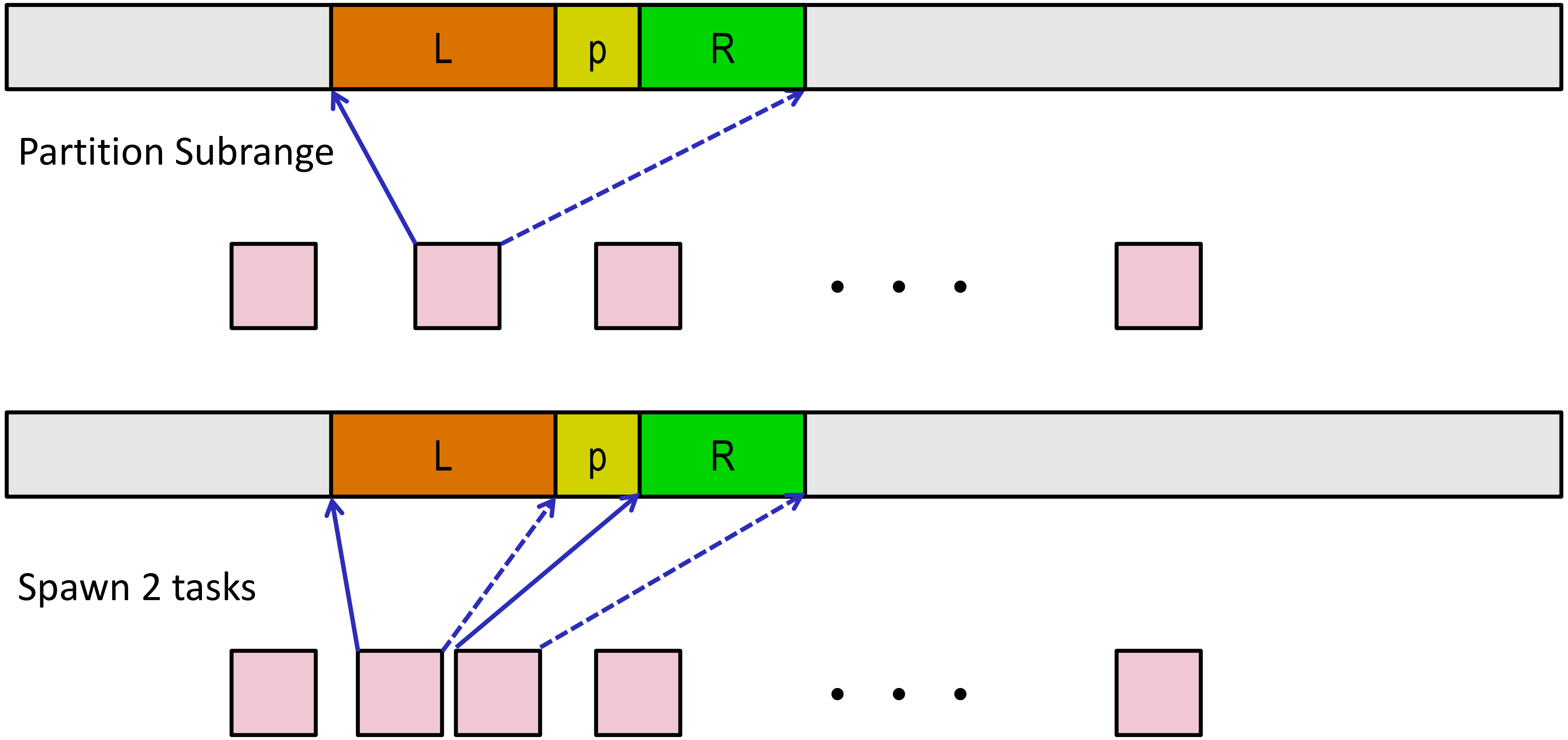
- **Task: Sort subrange of data**
  - Specify as:
    - **`base`**: Starting address
    - **`nele`**: Number of elements in subrange
- **Run as separate thread**

Small Sort Task Operation



- **Sort subrange using serial quicksort**

Large Sort Task Operation



Top-Level Function (Simplified)

```
void tqsort(data_t *base, size_t nele) {
    init_task(nele);
    global_base = base;
    global_end = global_base + nele - 1;
    task_queue_ptr tq = new_task_queue();
    tqsort_helper(base, nele, tq);
    join_tasks(tq);
    free_task_queue(tq);
}
```

- **Sets up data structures**
- **Calls recursive sort routine**
- **Keeps joining threads until none left**
- **Frees data structures**

Recursive sort routine (Simplified)

```
/* Multi-threaded quicksort */
static void tqsort_helper(data_t *base, size_t nele,
                          task_queue_ptr tq) {
    if (nele <= nele_max_sort_serial) {
        /* Use sequential sort */
        qsort_serial(base, nele);
        return;
    }
    sort_task_t *t = new_task(base, nele, tq);
    spawn_task(tq, sort_thread, (void *) t);
}
```

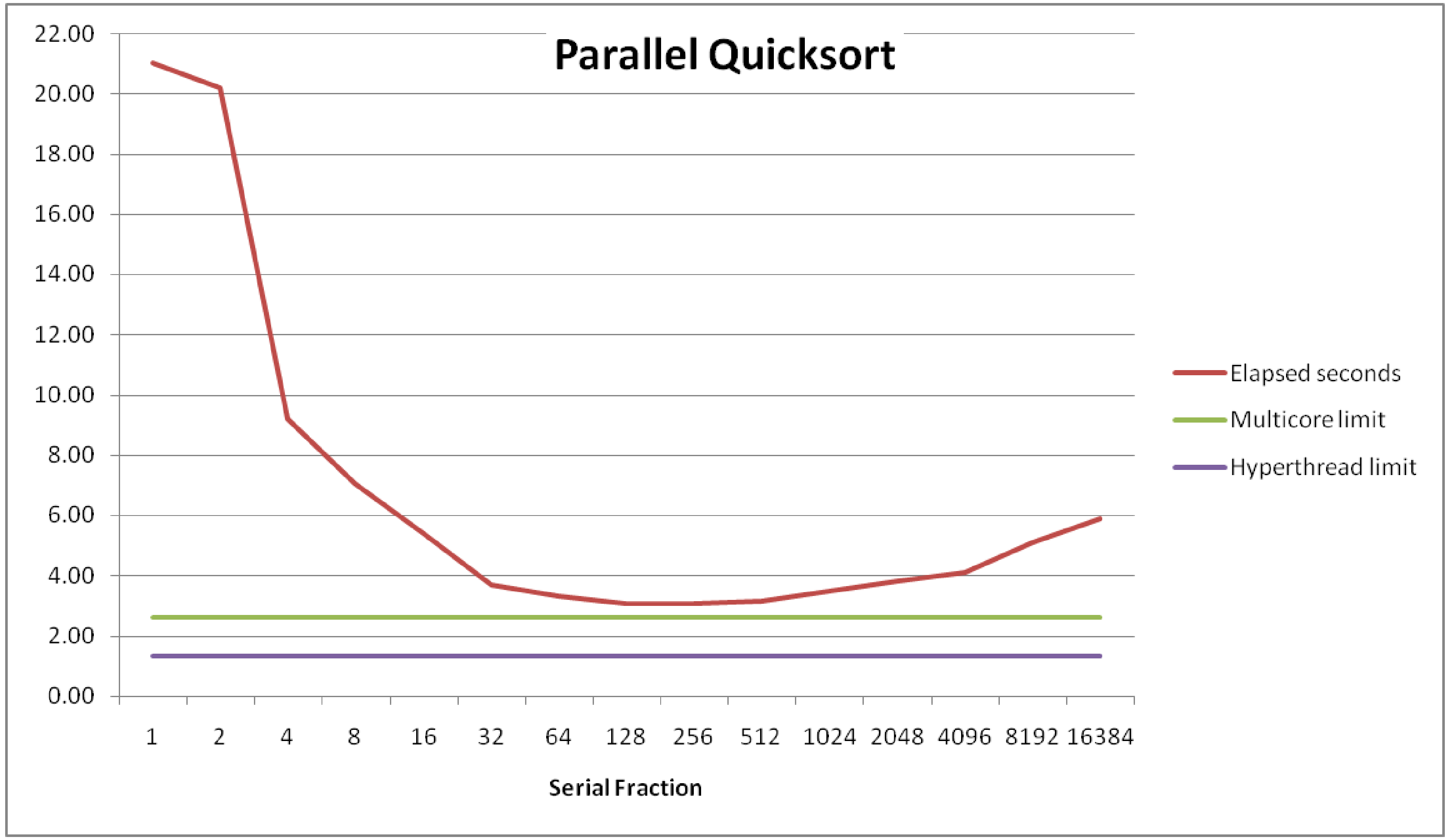
- **Small partition: Sort serially**
- **Large partition: Spawn new sort task**

Sort task thread (Simplified)

```
/* Thread routine for many-threaded quicksort */
static void *sort_thread(void *vargp) {
    sort_task_t *t = (sort_task_t *) vargp;
    data_t *base = t->base;
    size_t nele = t->nele;
    task_queue_ptr tq = t->tq;
    free(vargp);
    size_t m = partition(base, nele);
    if (m > 1)
        tqsort_helper(base, m, tq);
    if (nele-1 > m+1)
        tqsort_helper(base+m+1, nele-m-1, tq);
    return NULL;
}
```

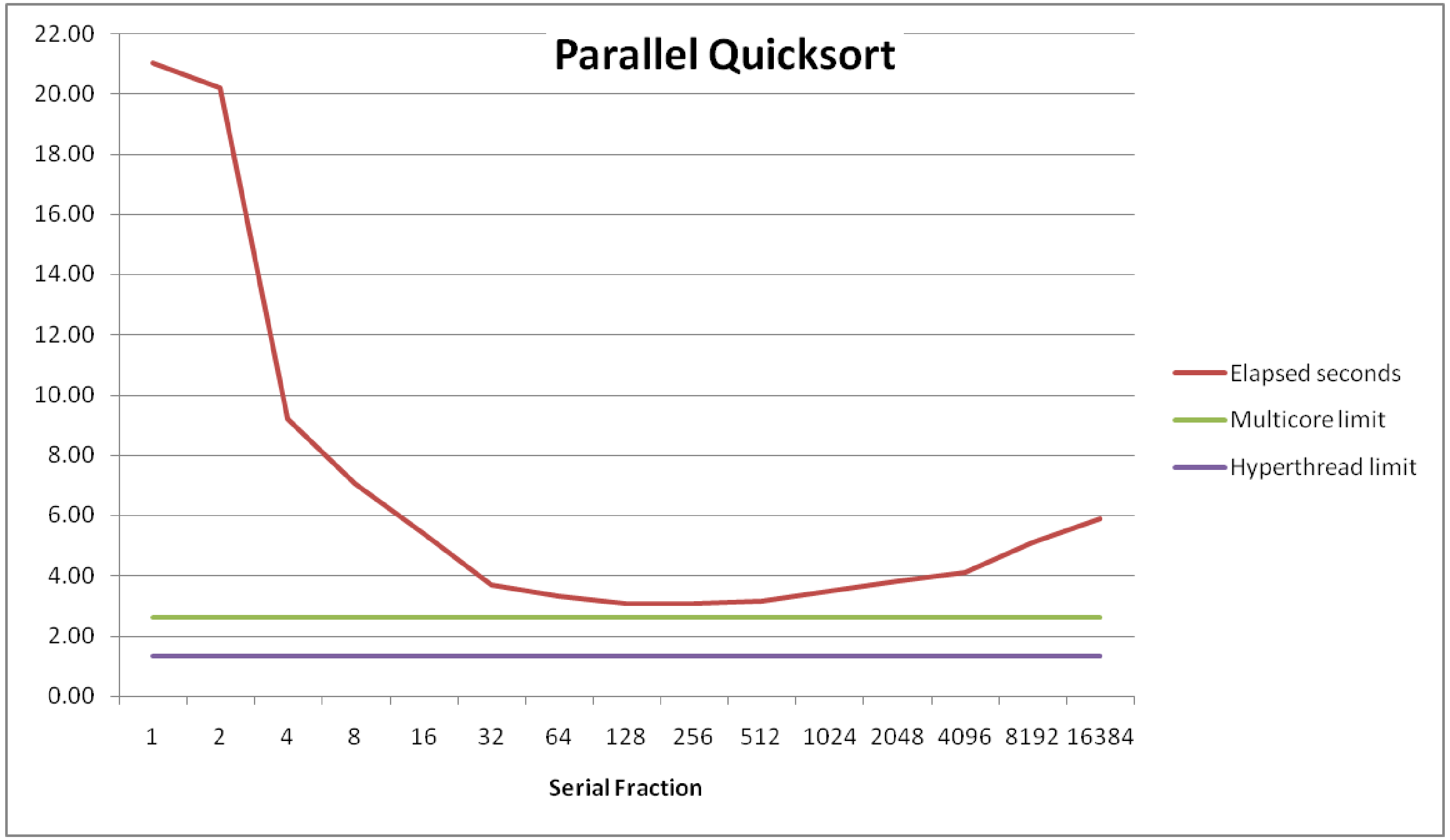
- **Get task parameters**
- **Perform partitioning step**
- **Call recursive sort routine on each partition**

Parallel Quicksort Performance



- **Serial fraction: Fraction of input at which do serial sort**
- **Sort 2^{27} (134,217,728) random values**
- **Best speedup = 6.84X**

Parallel Quicksort Performance



- **Good performance over wide range of fraction values**
  - F too small: Not enough parallelism
  - F too large: Thread overhead + run out of thread memory

Lessons Learned

- **Must have parallelization strategy**
  - Partition into K independent parts
  - Divide-and-conquer
- **Inner loops must be synchronization free**
  - Synchronization operations very expensive
- **Beware of Amdahl's Law**
  - Serial code can become bottleneck
- **You can do it!**
  - Achieving modest levels of parallelism is not difficult
  - Set up experimental framework and test multiple strategies

18-600 Foundations of Computer Systems

Lecture 27: "Future of Computing Systems"

John P. Shen & Zhiyi Yu
December 7, 2016

Next Time ...

Electrical & Computer ENGINEERING