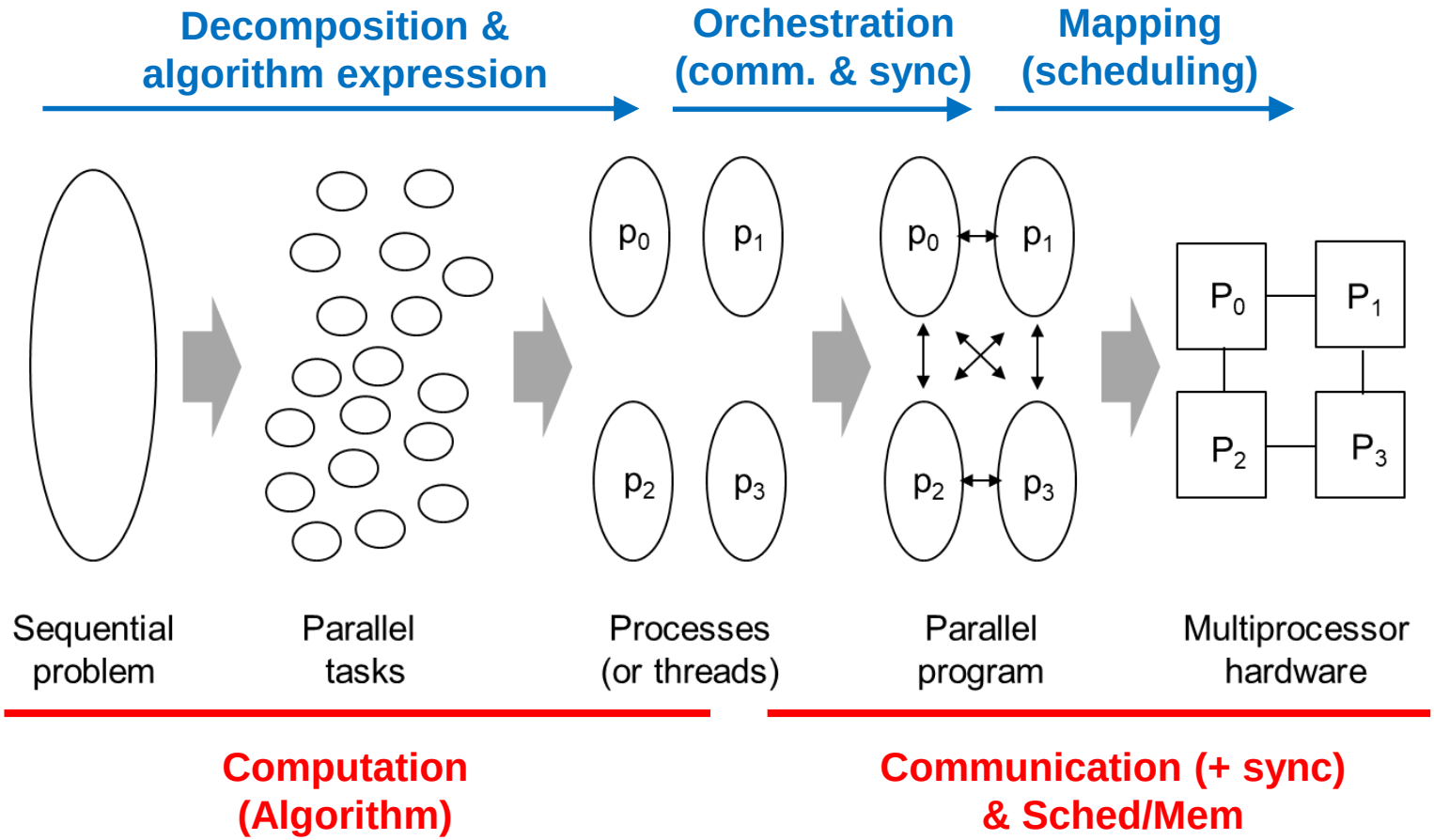
Multicore Performance #3
- Computation and Communication -

Minsoo Ryu

Department of Computer Science and Engineering
Hanyang University

Creating a Multicore Program



Multicore Computing Challenges

- It is not easy to develop an efficient multicore program

- Three key aspects of multicore computing
  - Computation — 2 순위 고려사항
  - Communication — 1 순위 고려사항
  - Synchronization — 3 순위 고려사항

- Some Challenges
  - Complex algorithms, multicore programming, memory issues, communication costs, …

Outline

Parallel Algorithms

- The cost or complexity of serial algorithms is estimated in terms of the space (memory) and time (processor cycles) that they take

- Parallel algorithms must take into account
  - Parallelizability
  - Communication between different processors
  - Load balancing across processors

- Let's consider two quicksort algorithms
  - Sequential quicksort
  - Parallel quicksort

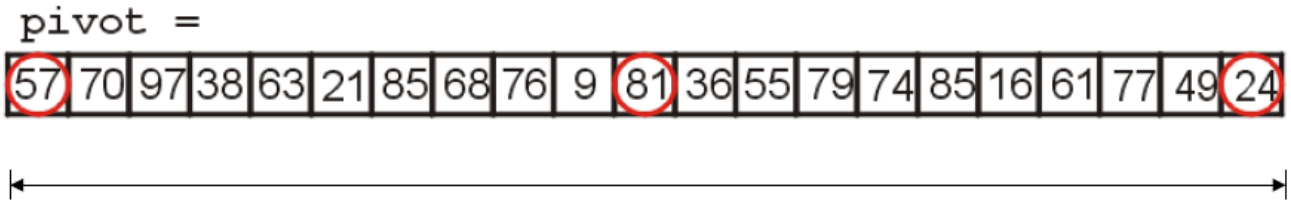
Sequential Quicksort

- **Invented by Tony Hoare**
  - Published in Comm. Of the ACM, 1961
  - 1980 Turing Award winner
  - Generally recognized as the fastest sorting algorithm, in the average case

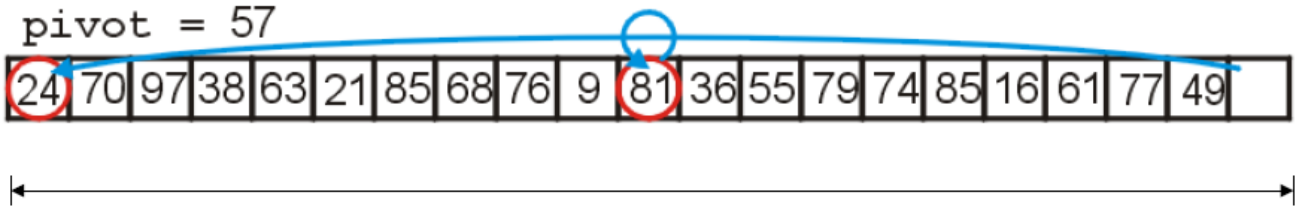
- **The quicksort algorithm**
  - A divide and conquer algorithm which relies on a partition operation
  - To partition an array an element called a pivot is selected. All elements smaller than the pivot are moved before it and all greater elements are moved after it
  - The lesser and greater sublists are then recursively sorted

Example of Sequential Quicksort

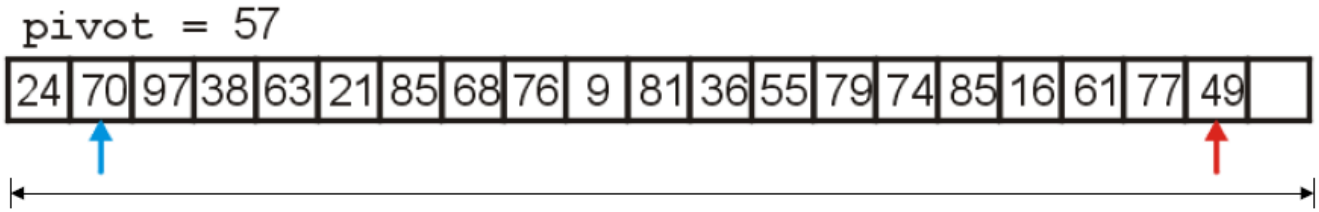
- Examine the first, middle, and last entries



- Select 57 to be our pivot and move 24 into the first location

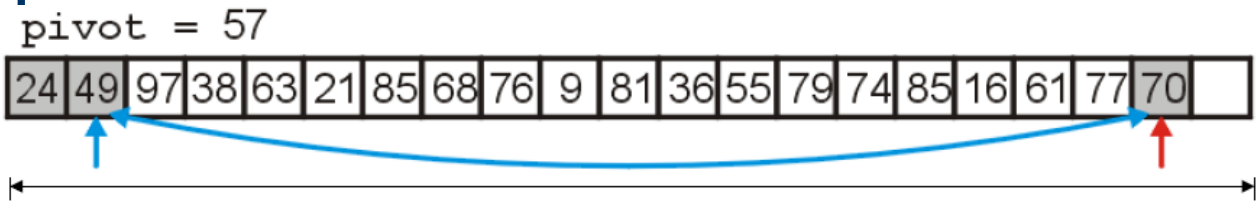


- From both ends, search forward until we find 70 > 57 and search backward until we find 49 < 57

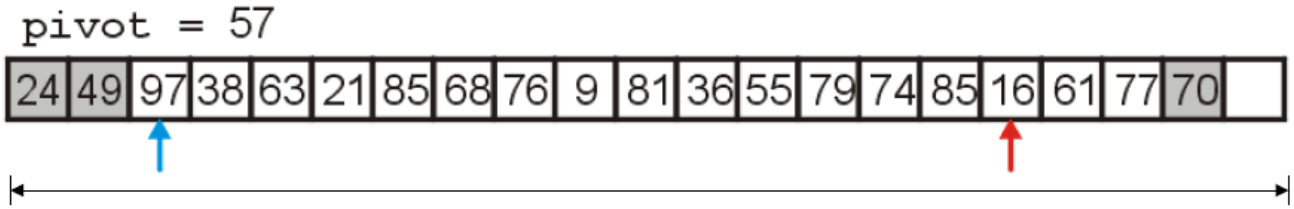


Example of Sequential Quicksort

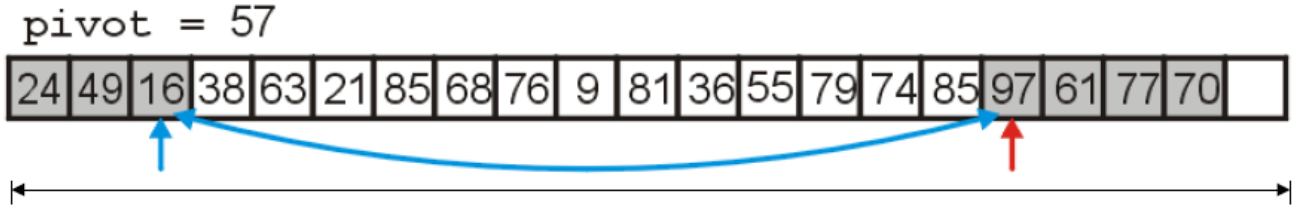
- Swap 70 and 49



- Search forward until we find 97 > 57 and search backward until we find 16 < 57

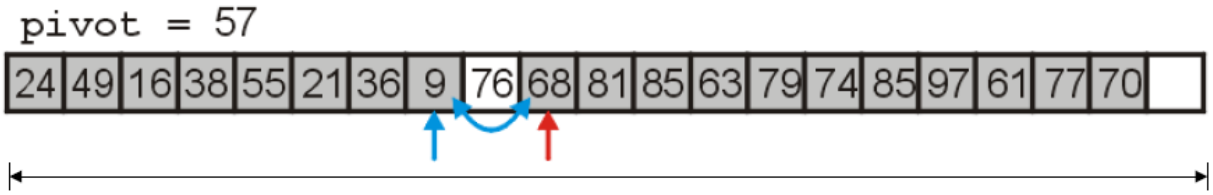


- Swap 16 and 97

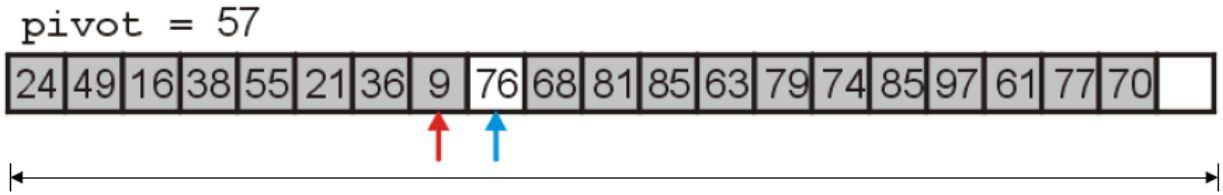


Example of Sequential Quicksort

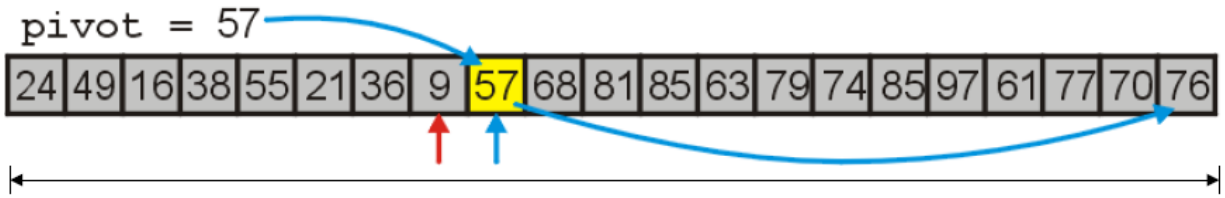
- Repeat searching and swapping, and swap 68 and 9



- Search forward and backward
  - The indices are out of order, so we stop

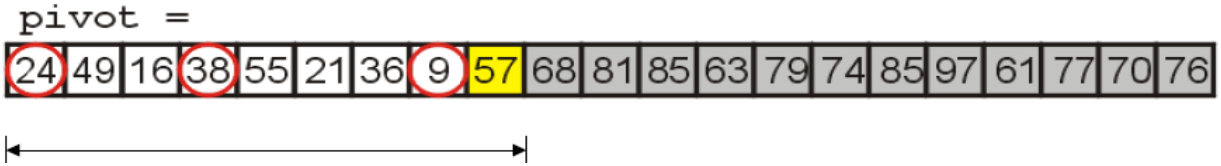


- Move the larger item to the vacancy at the end of the array
  - Fill the empty location with the pivot, 57

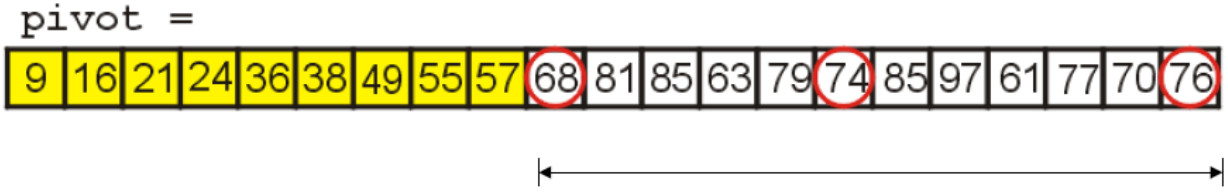


Example of Sequential Quicksort

- Now recursively call quick sort on the first and second half of the list
  - All entries < 57 are sorted



  - All entries >= 57 are be sorted



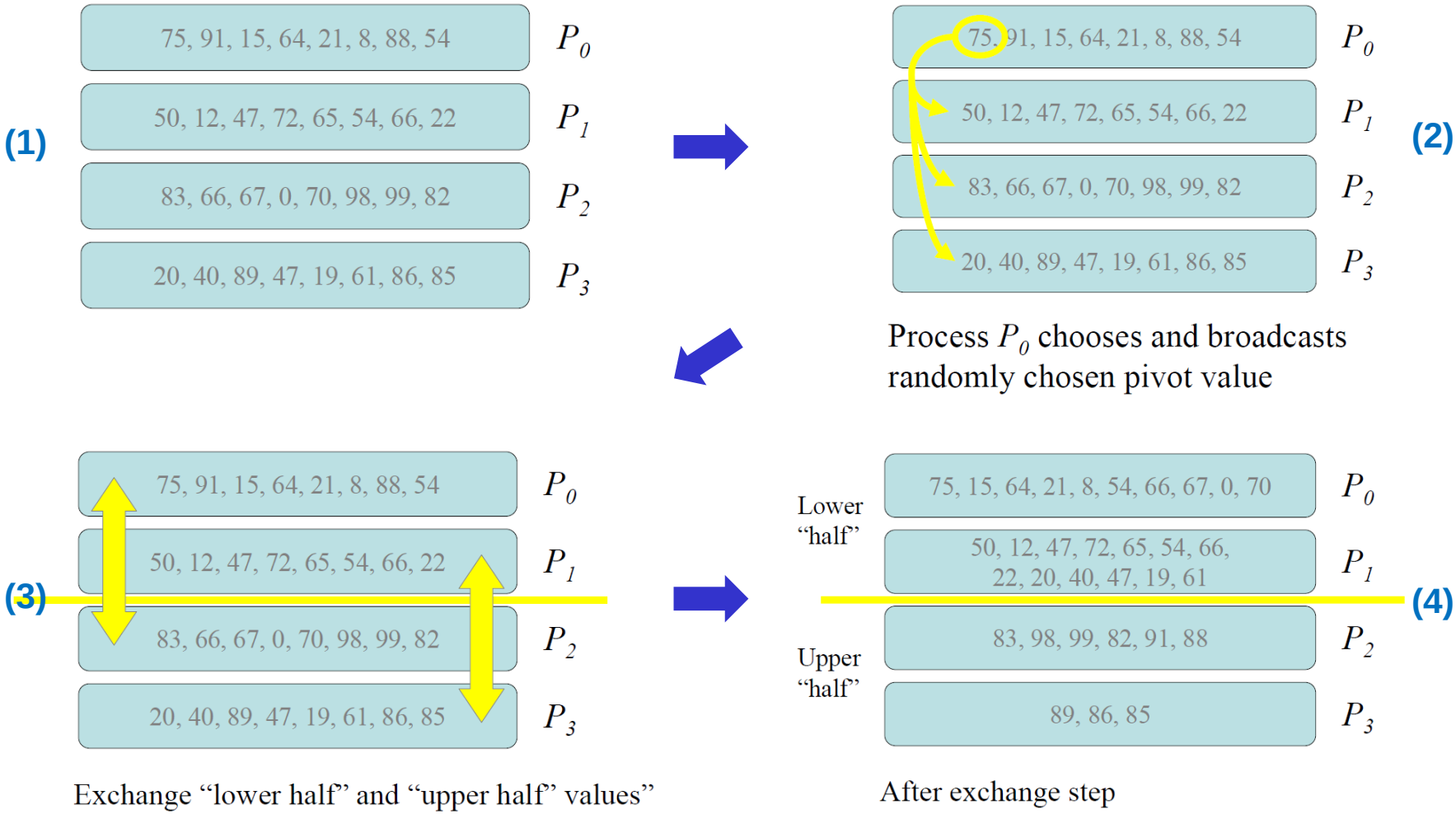
  - Finally, we arrive at an ordered list

Parallel Quicksort

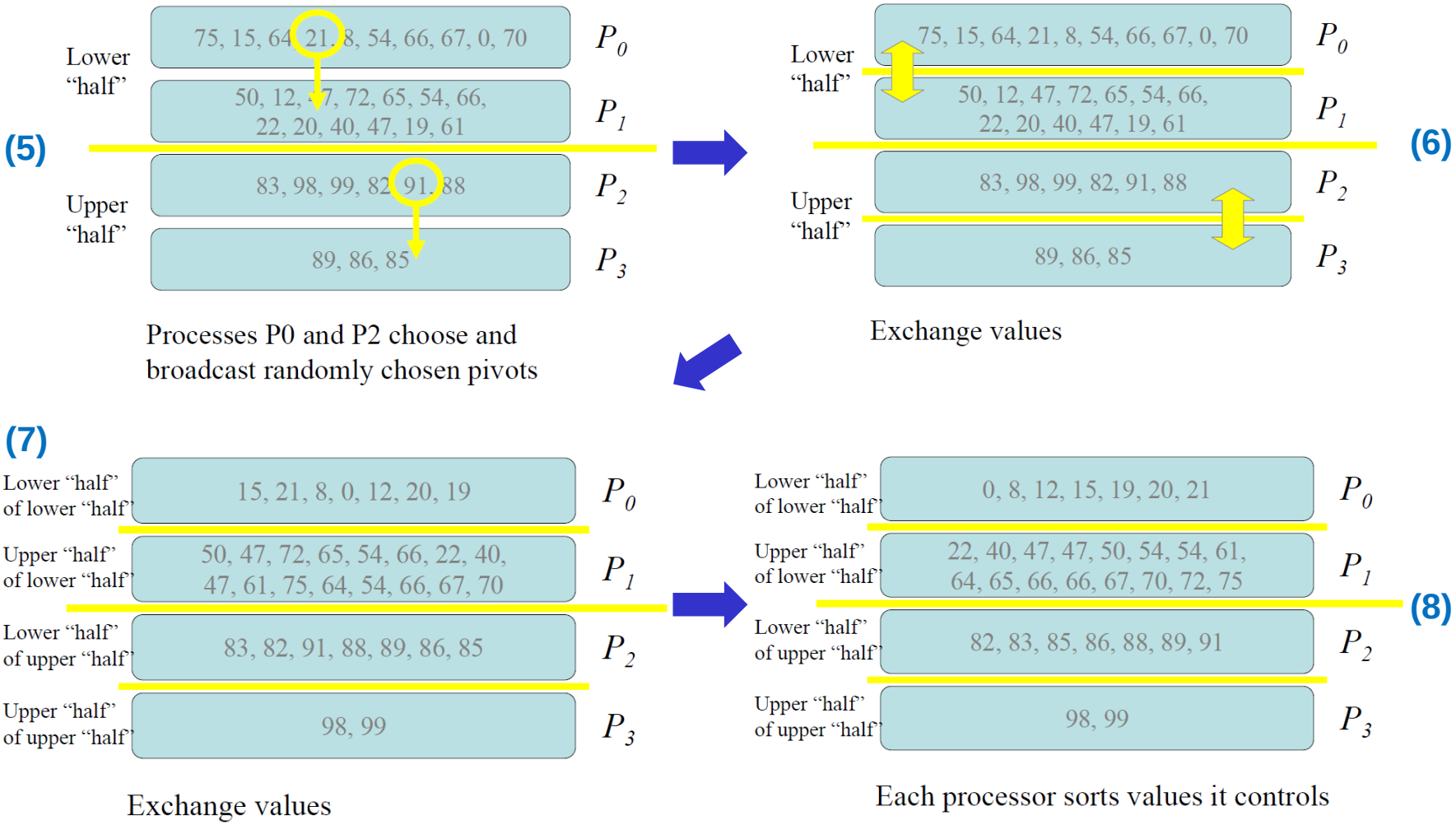
Algorithm

- **Each process holds a segment of the unsorted list**
  - The unsorted list is evenly distributed among the processes
- **Processes exchange their data**
- **The list segment stored on each process is sorted**
  - The last element on process i's list is smaller than the first element on process i + 1's list

Example of Parallel Quicksort



Example of Parallel Quicksort



Performance Issues with Parallel Quicksort

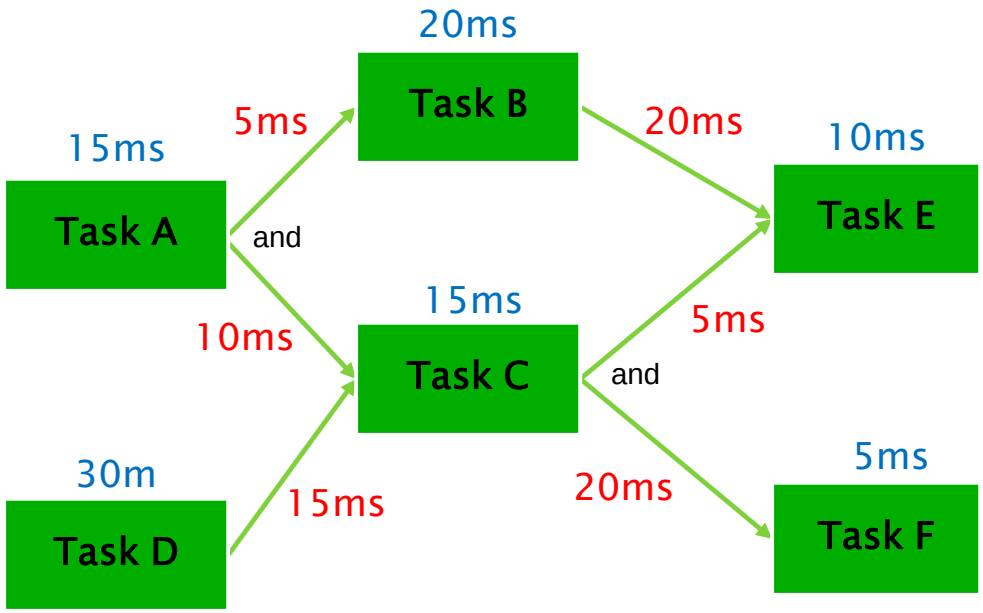
- **Performance concerns**
  - **The parallel quicksort algorithm is likely to do a poor job of load balancing**
    - If the pivot value is not the median value, we will not divide the list into two equal sublists
    - Finding the median value is prohibitively expensive on a parallel computer
  - **The communication cost for data exchange across processors could be too high**

- **There are other quicksort algorithms**
  - **Hyperquicksort**
  - **Parallel sorting by regular sampling**

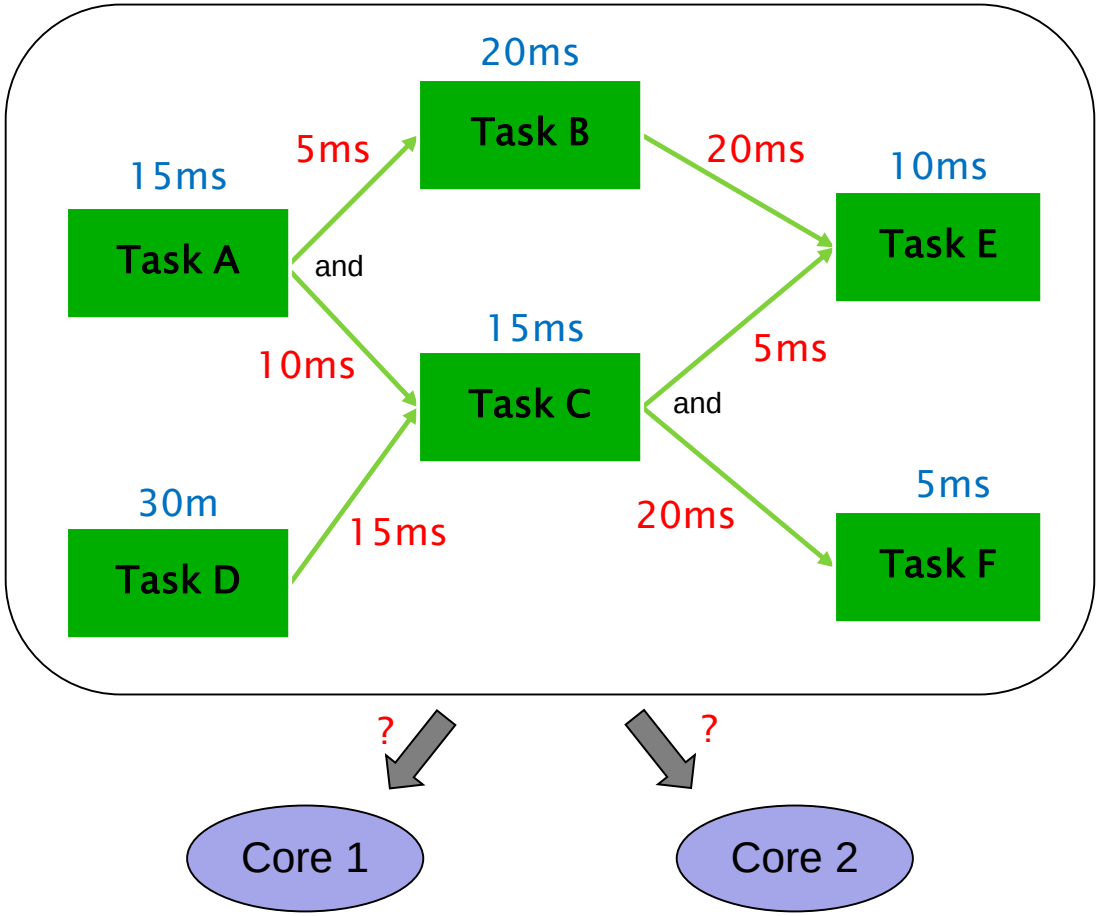
Communication Cost

Task Graph (Directed Acyclic Graph)

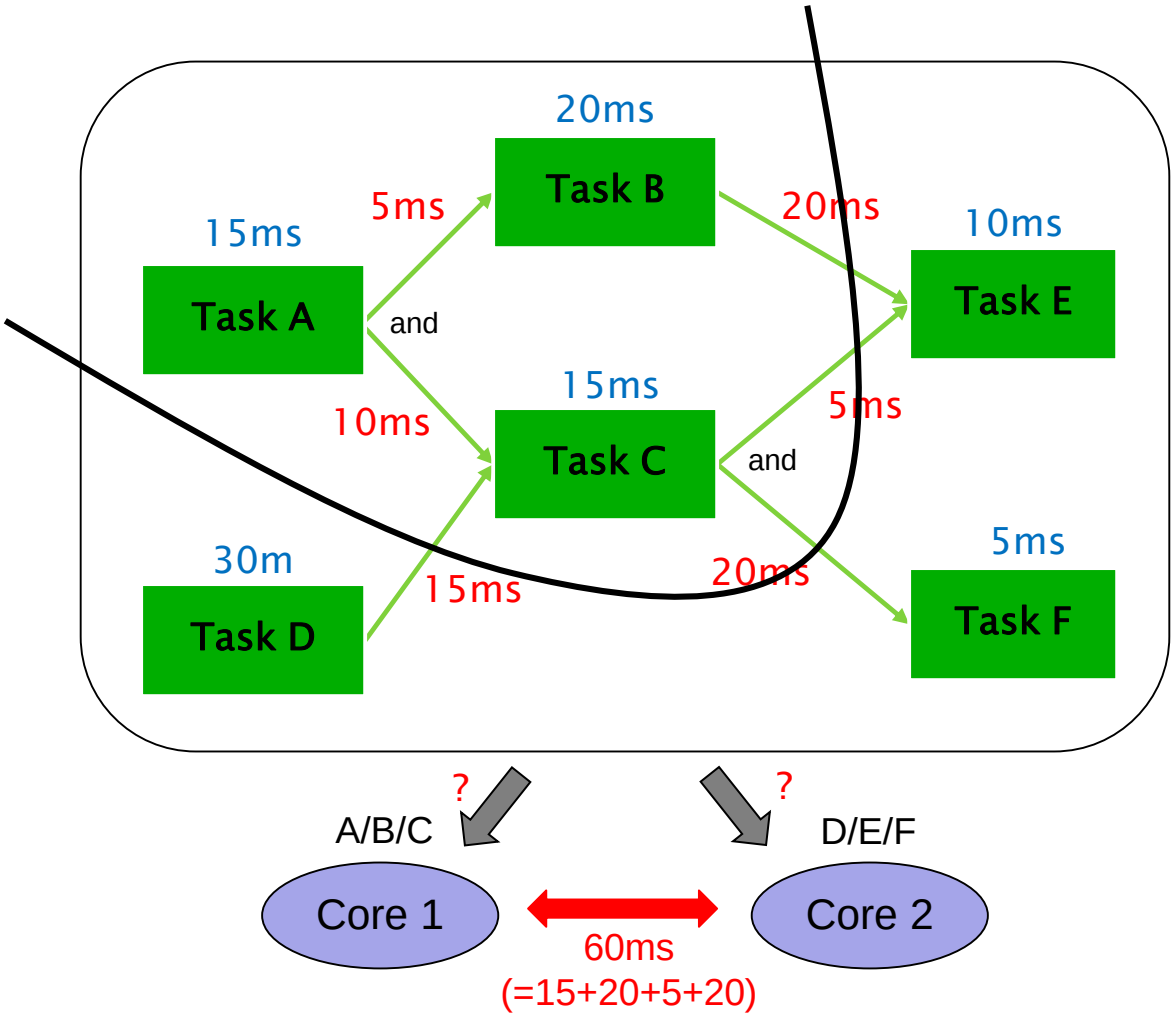
- Task graph
  - One task cannot start until some other tasks finish
  - The output of one task is the input to another task



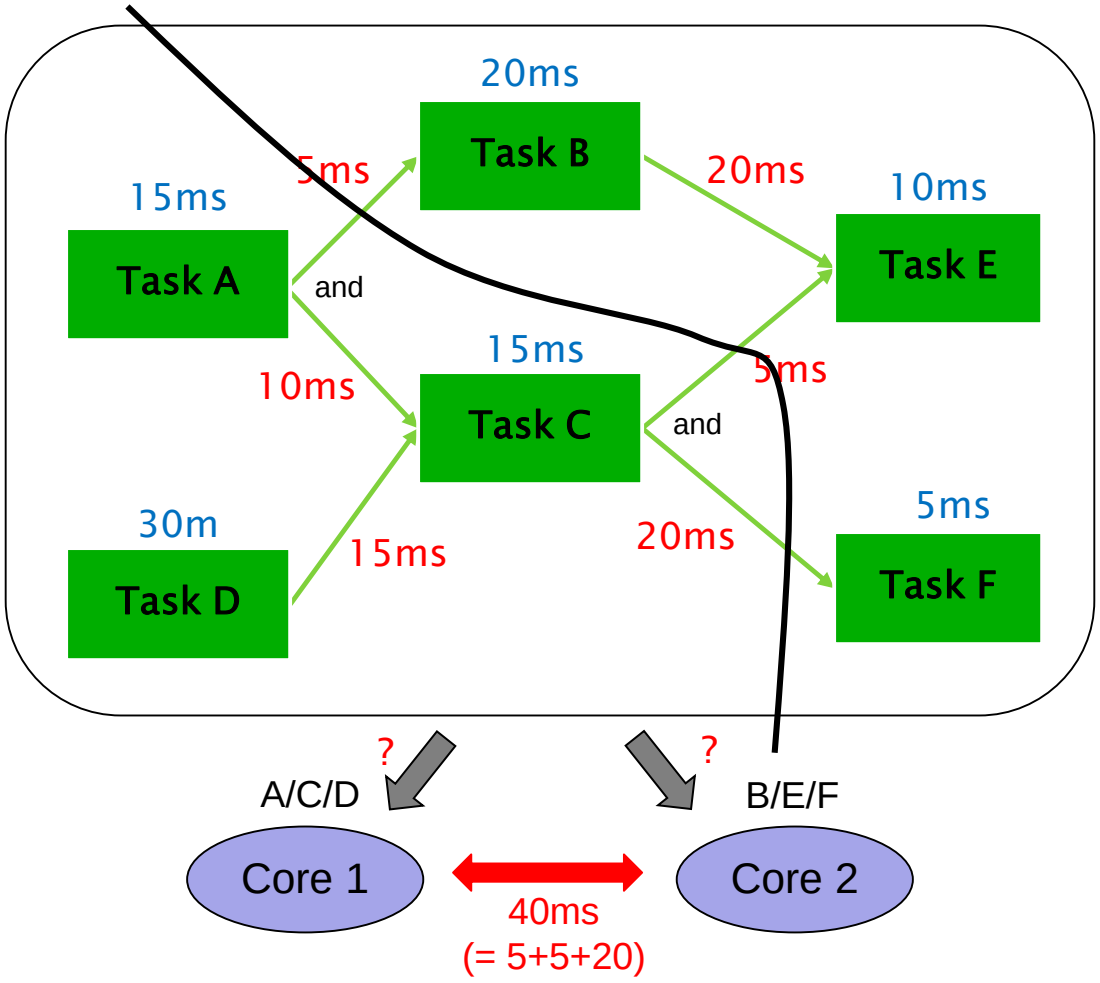
Task Allocation and Communication Cost



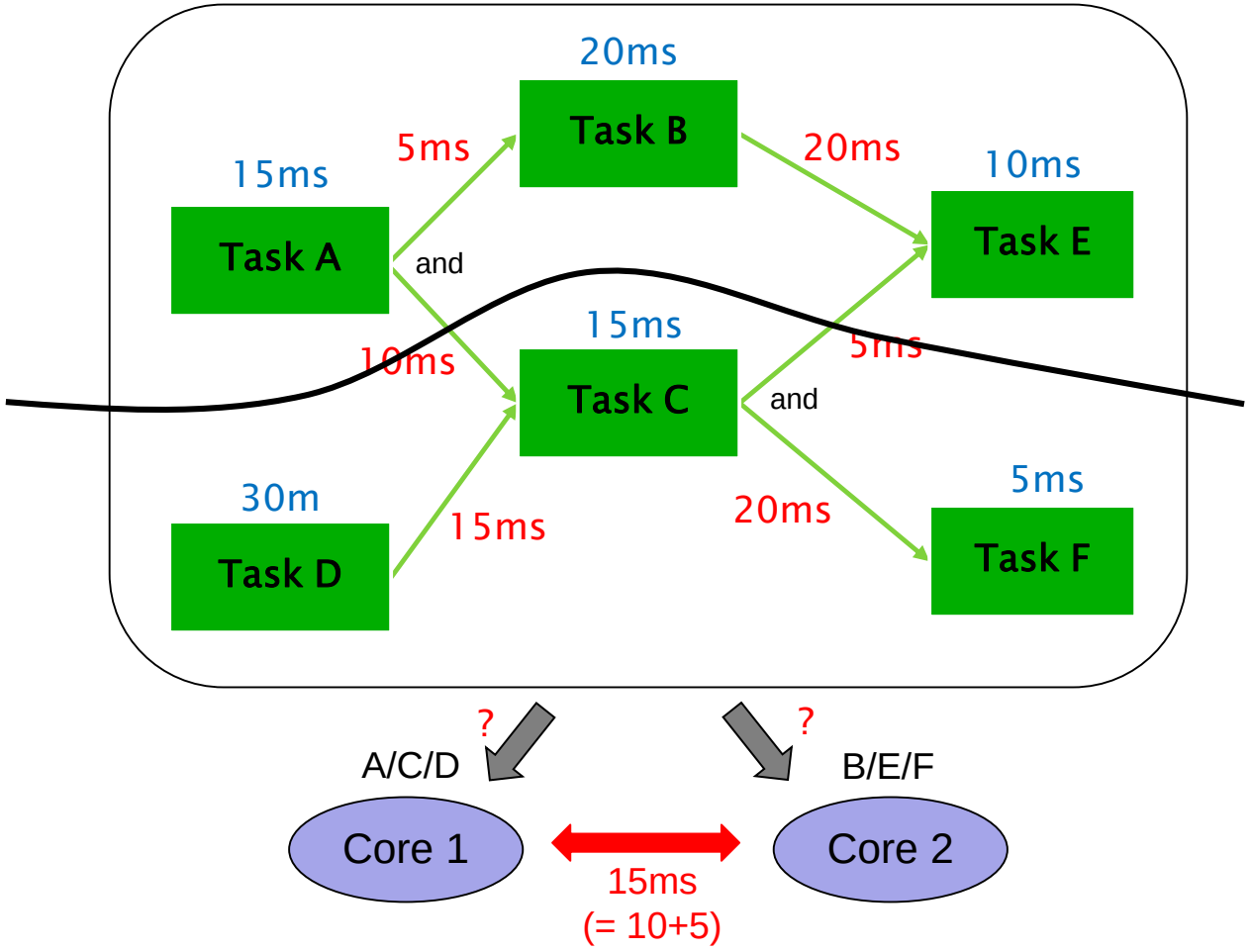
Task Allocation and Communication Cost



Task Allocation and Communication Cost



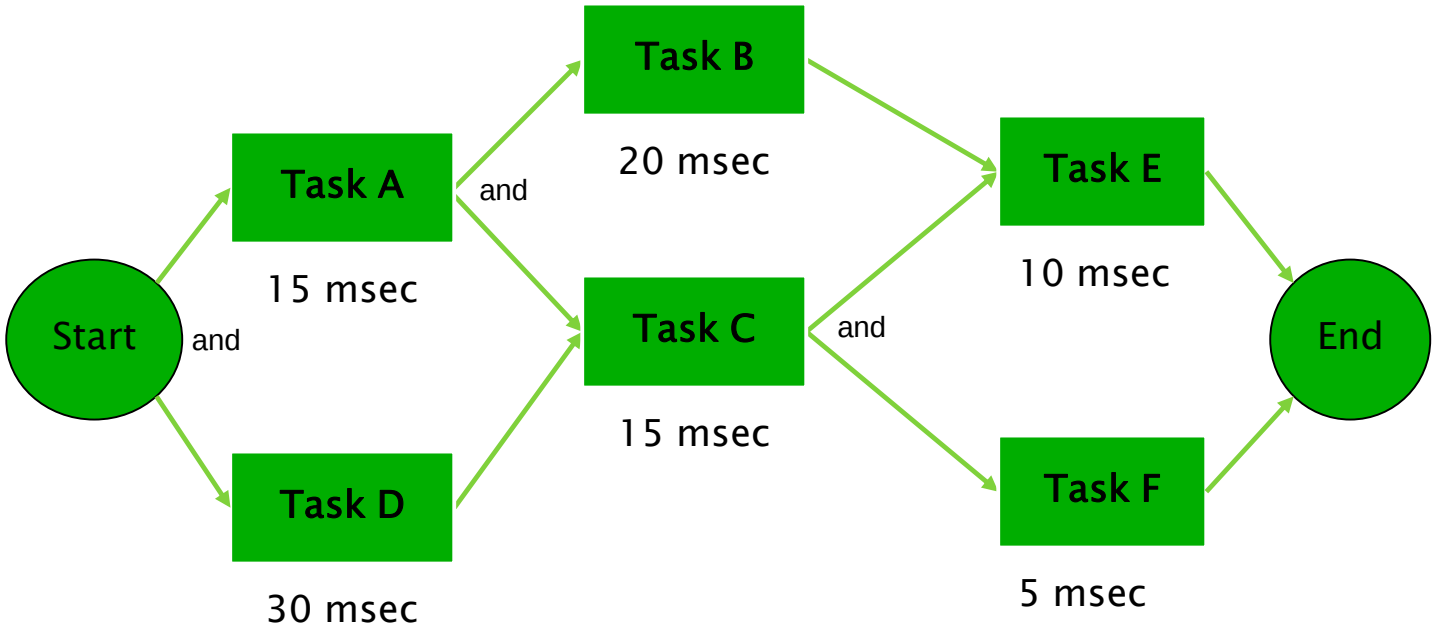
Task Allocation and Communication Cost



Latency and Throughput

- **If tasks are periodic, what are the latency and throughput?**

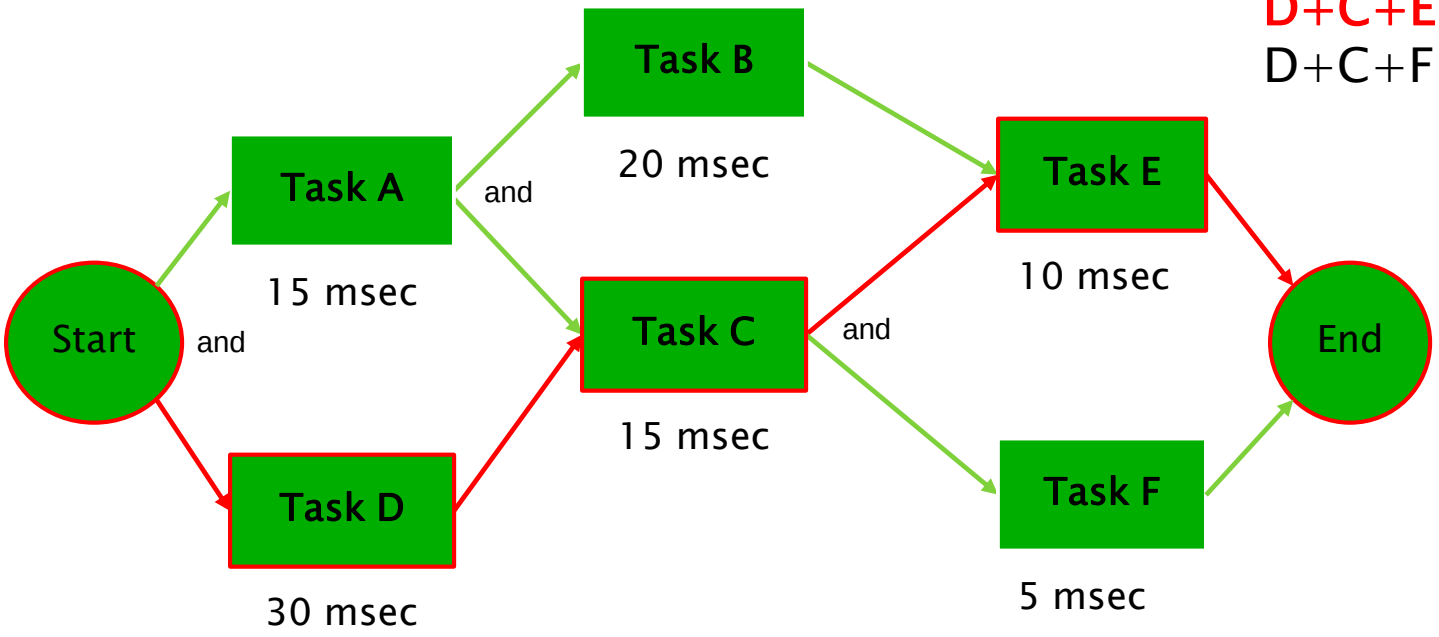
비디오 영상이 들어오는 상황 가정.
latency : input --> output 걸리는 시간
thruoughput : 단위시간당 처리량



Worst Case Latency

➢ Find the critical path

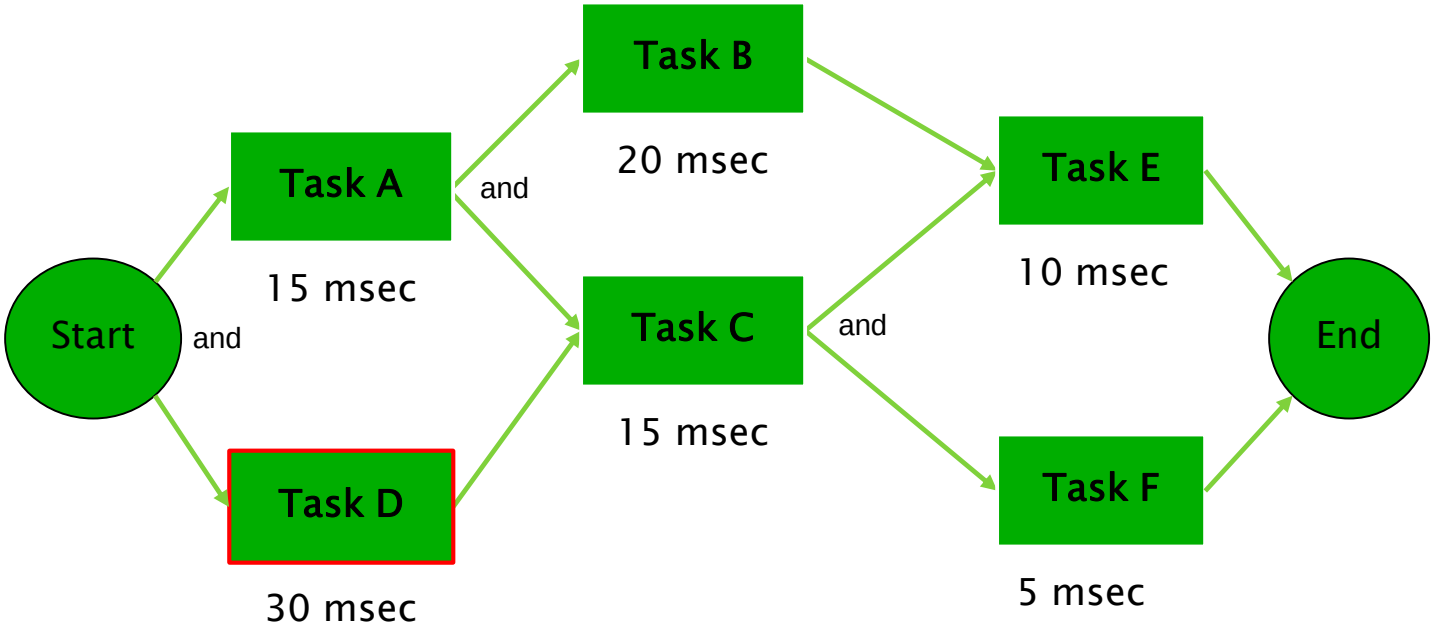
A+B+E = 45
A+C+E = 40
A+C+F = 35
D+C+E = 55
D+C+F = 50



Maximum Throughput

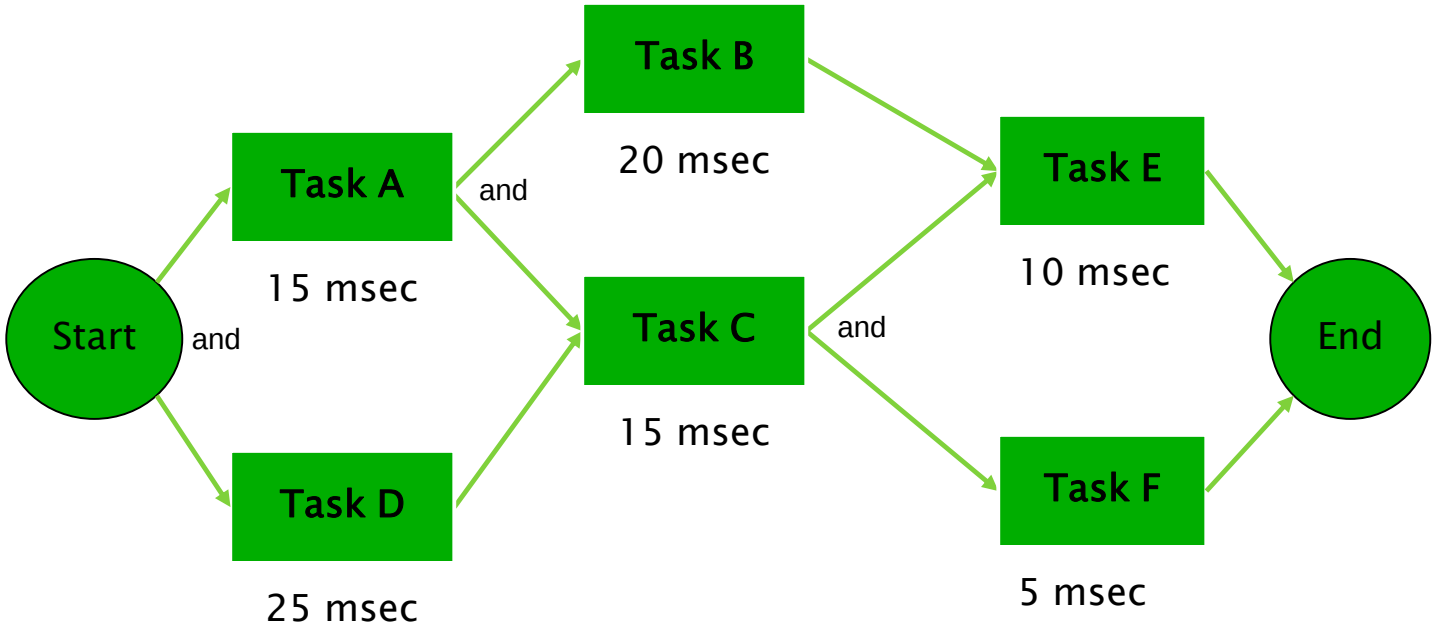
- Find the slowest stage

Max(A, B, C, D, E, F) = 30 (= D)



Bottleneck Identification and Optimization

- Thread D is the common bottleneck for both latency and throughput
- Reduce the execution time of D to 25ms (← 30ms)
  - Worst case latency: 50 msec (← 55 msec)
  - Throughput: 25 msec (← 30 msec)

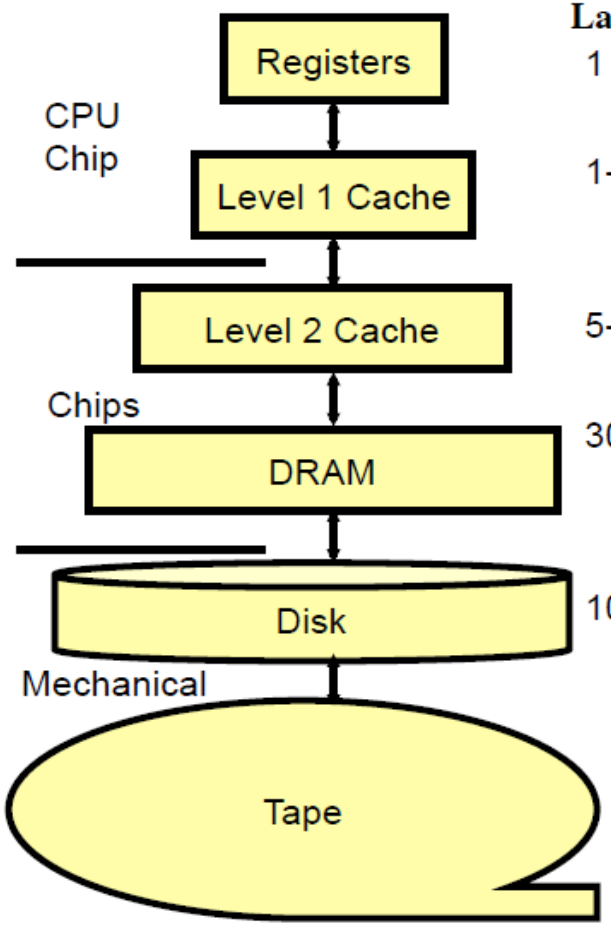


Memory and Caching

Memory Performance

- **DRAM Memory**
  - **30-50 nanoseconds to get data from memory into CPU**
  - **CPU can do 2-4 operations every 300 picoseconds**
    - 100+ times slower than CPU!
- **Hard drive**
  - **A typical 7200 RPM desktop HDD has a "disk-to-buffer" data transfer rate up to 1030 Mbit/s**
- **Solution**
  - **Rely on caching based on the memory hierarchy**
  - **If data fits in cache, you get Gigaflops performance per processor**

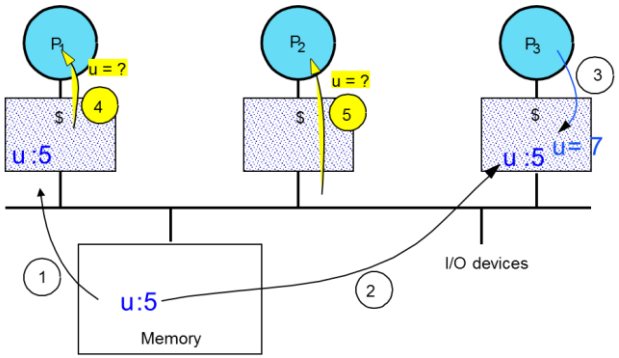
Memory Hierarchy

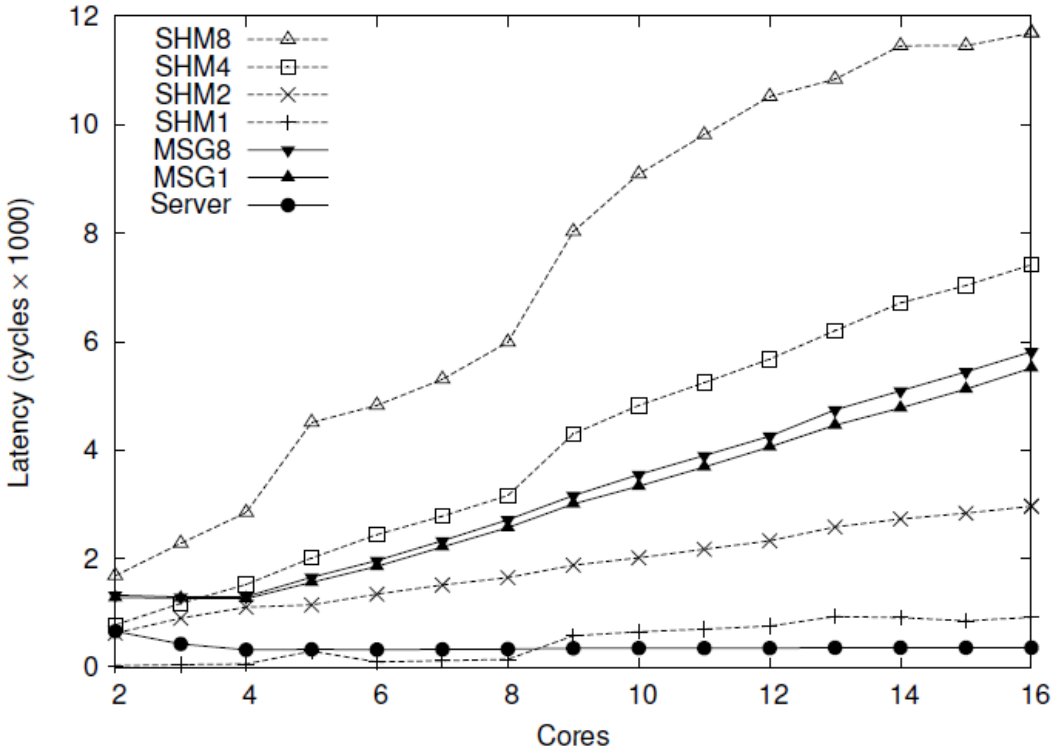


	Latency	Bandwidth	
Registers	1 cyc	3-10 words/cycle < 1KB	compiler managed
Level 1 Cache	1-3cy	1-2 words/cycle 32KB -1MB	hardware managed
Level 2 Cache	5-10cy	1 word/cycle 1MB - 4MB	hardware managed
DRAM	30-100cy	0.5 words/cycle 64MB - 4GB	OS managed
Disk	10^6-10^7cy	0.01 words/cycle 4GB+	OS managed
Tape			

Shared Memory and Message Passing

- **Shared memory vs. MSG passing (Microsoft, 2009)**
  - Shared memory updates lead to large cache miss costs due to the cache coherence mechanism
  - MSG passing costs much less than shared memory





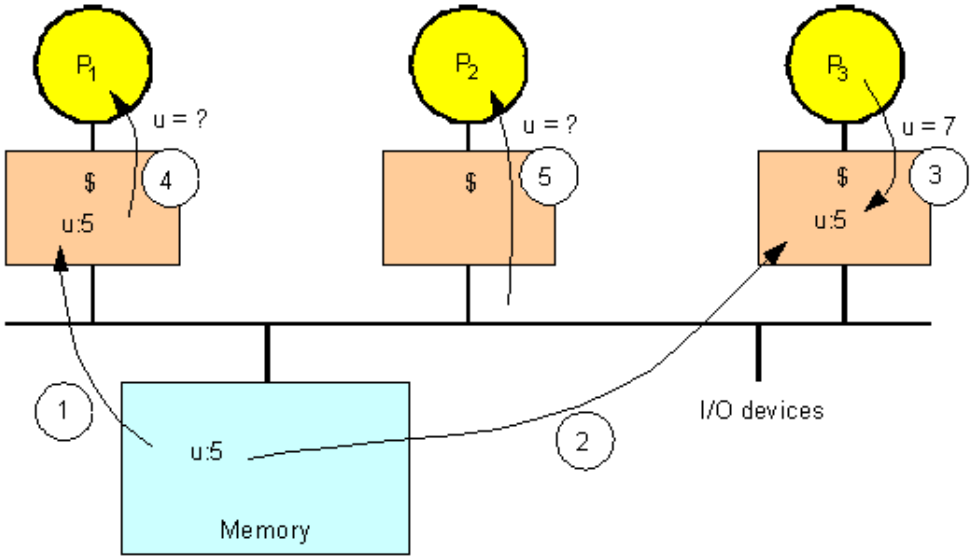
Shared Memory and Message Passing

- **SHM updates**
  - **A processing core updates a shared memory variable**
  - **Other processing cores stall due to cache coherence**

- **MSG passing**
  - **Only a server processor accesses the memory variable**
  - **Other processors send/receive values to/from the server**
    - There is no cache coherence and no processor stall
    - Each processor may experience some queueing delay
      - ✓Grows linearly with the number of clients
      - ✓Does not cause processor stall
    - Memory update cost at the server is low and constant

Cache Coherence

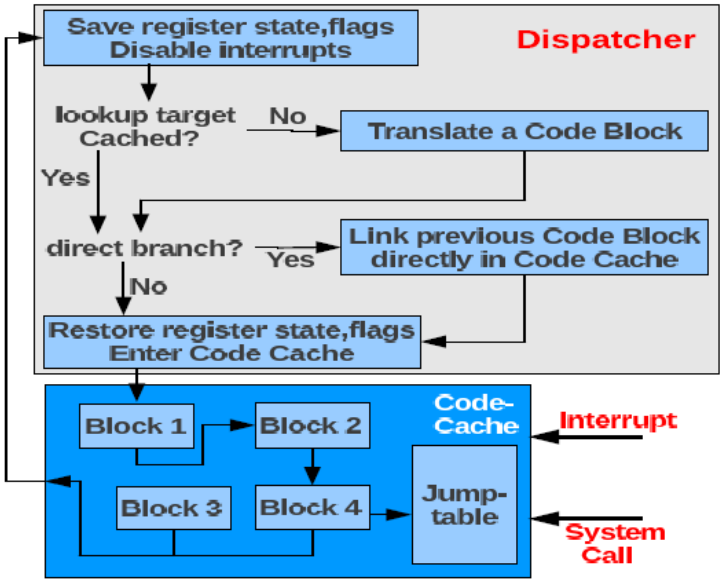
- Cache coherence is the consistency of shared resource data stored in multiple local caches
- Two cache coherency protocols
  - Snoopy protocol
  - Directory-based protocol



User-Level Caching of Data and Code

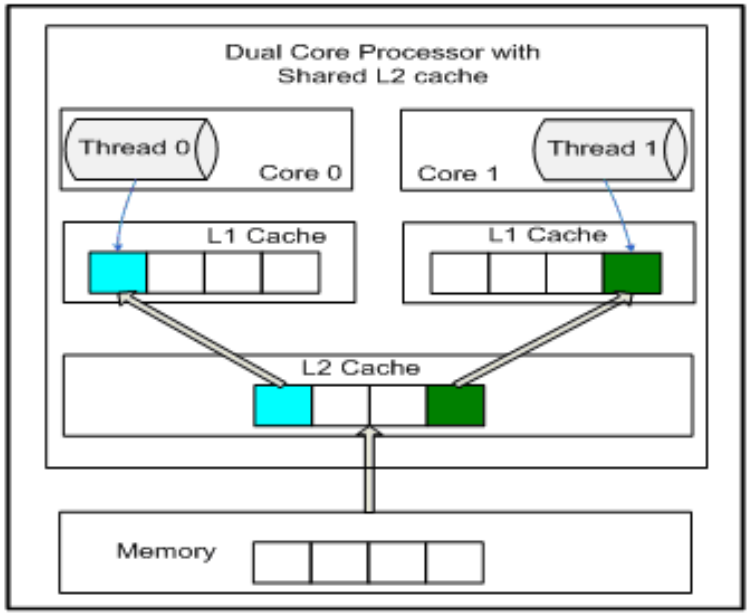
- **Data caching**
  - **Maintains a free list for each type of user-defined data structure**
  - **Allocate one from the free list, use it, and return it back to the list**
    - The slab allocator in the Linux kernel
  - **Data content can also be cached for possible reuse**

- **Code caching**
  - **Cache translated code and reuse it**
  - **One application is the binary translation in full virtualization**



False Sharing with Hardware Cache

- **False sharing**
  - Multiple threads are accessing items of data held on a single cache line
  - The cache line is constantly being bounced between processors due to the cache coherency mechanism



- **Solution**
  - It is easy to solve false sharing by padding the accessed structures so that the variable used by each thread resides on a separate cache line

False Sharing and Its Solution

- Each thread accesses to the counter structure
  - A. The size of counter is 4 bytes
  - B. Each counter is located at 64-byte intervals

```
#define COUNT 100000000
volatile int go = 0;
volatile int counters[20];

void *spin( void *id )
{
  int myid = (int)id + 1;
  while( !go ) {}
  counters[myid] = 0;
  while ( counters[myid]++ < COUNT ) {}
}
```

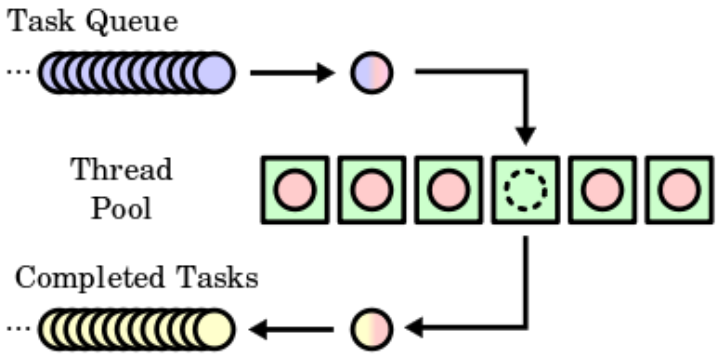
A. thread code with false sharing (64-byte cache line size)

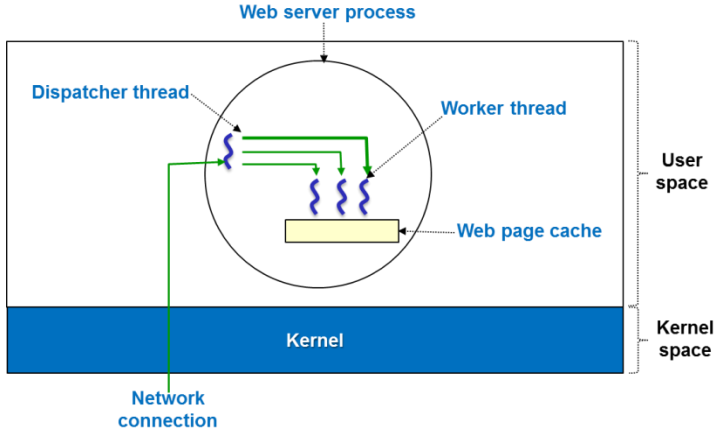
```
#define COUNT 100000000
volatile int go = 0;
volatile int counters[320];

void *spin( void *id )
{
  int myid = ( (int)id + 1) * 16;
  while( !go ) {}
  counters[myid] = 0;
  while ( counters[myid]++ < COUNT ) {}
}
```

B. thread code without false sharing (64-byte cache line size)

Thread Pooling (Thread Cache)

- The overhead of dynamic thread creation and termination is high
  - If the amount of work known in advance use static thread creation otherwise use dynamic thread creation
- Thread pool
  - Create a number of threads
  - Jobs are organized in a queue
  - A new job is dispatched to a free thread in the pool





Thread Stack Size

- Each thread requires a separate stack area
  - Reading the stack size for a POSIX thread

```
int main()
{
  size_t stacksize;
  pthread_attr_t attributes;
  pthread_attr_init( &attributes );
  pthread_attr_getstacksize( &attributes, &stacksize );
  printf( "Stack Size = %i\n", stacksize);
  pthread_attr_destroy( &attributes );
}
```

  - Running it on Ubuntu produces the result shown in Listing 5.11, indicating that the default stack size is 8MB

```
$ gcc stack.c -lpthread
$ ./a.out
Stack Size = 8388608
```

Thread Stack Size

- If each thread is allocated an 8MB stack, then there can be at most 512 threads on 32-bit HW
  - 512 * 8 MB = entire 4GB address space
  - The default size of stack depends on the underlying OS
- It is possible to change the default stack size by the following option
  - ulimit -s <stacksize>
  - Thus, it can be a good idea to assess how much memory is actually required for the stack of each child thread and limit the default stack size

Contention and Synchronization

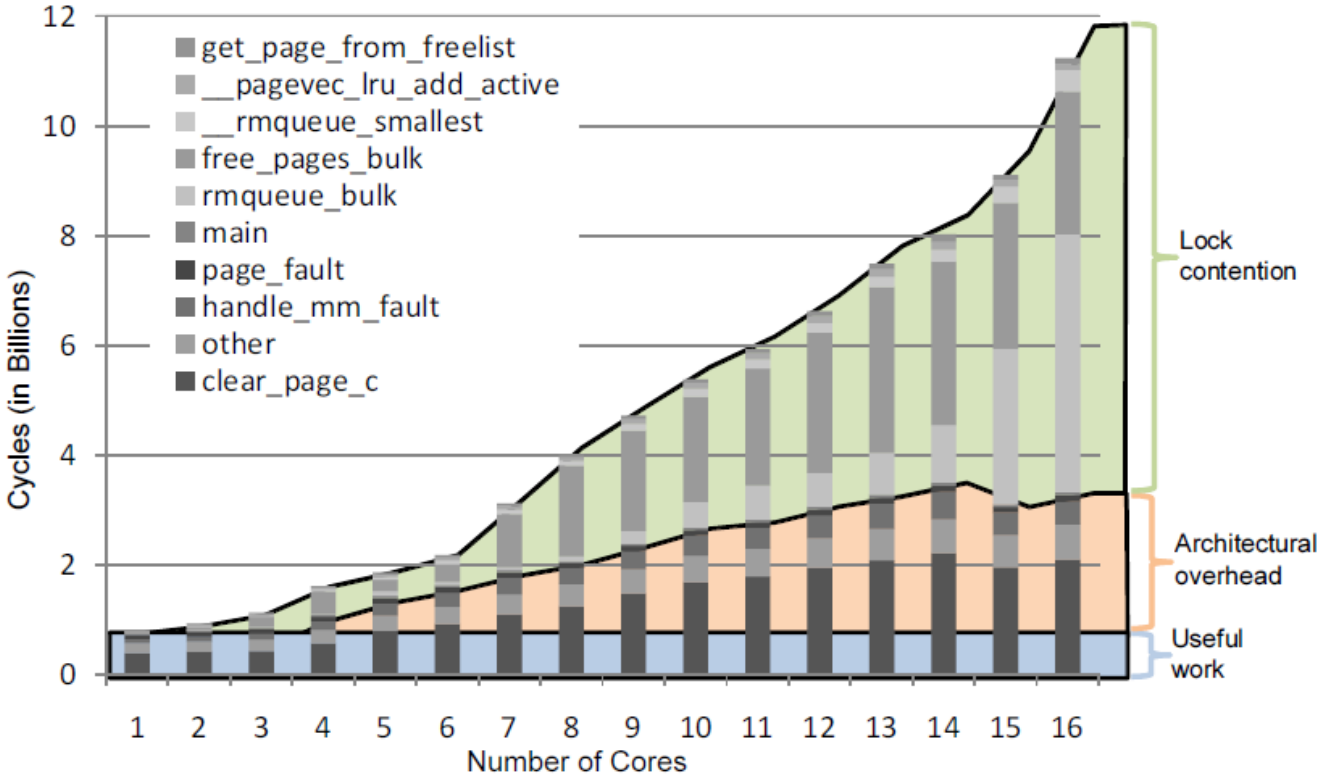
Lock Contention

- Lock contention between different threads cause significant performance degradation
  - Hot locks (hot spots)

- General solutions (to reduce the lock holding time)
  - Narrow the lock scope
  - Reduce lock granularity

Lock Contention

- **Memory allocation experiments (MIT, 2009)**
  - **16-core Intel system with Linux 2.6.24.7 kernel**
    - Xeon E7340 CPUs at 2.40GHz and 16GB of RAM



Lock Holder Preemption

- Lock holder preemption describes the situation when a CPU is preempted inside a critical section
  - Other threads waiting for the lock must wait indefinitely

- Well known problems
  - A VCPU (virtual CPU) is preempted inside the guest kernel while holding a spinlock (in OS virtualization)
  - Priority inversion problem (in real-time systems)

Shared Data Contention

- Contention on shared data may seriously limit performance as well as causing locking overhead
  - Only one thread can occupy the bus and access the memory at a time
- Two approaches
  - Data replication
  - Data partitioning

Data Replication and Consistency Control

- **Data replication is used for two purposes**
  - **To increase the reliability of a system**
    - The system can continue working after one replica crashes
  - **To improves the performance**
    - Processors can access its own copy simultaneously
    - Performs well especially when the data is mostly read

- **Consistency control**
  - **One issue of data replication is consistency problems**
  - **Whenever a copy is modified, modifications have to be carried out on all copies**
    - Analogous to the cache coherence problem

Data Partitioning

- **Data is divided across *n* processors, and threads are routed to the partitions that contain the data they need to access**

- **Data partitioning is often used for database systems on multicore HW**
  - **Some applications are "perfectly partitionable"**
    - Every transaction can be executed in its entirety at a single partition
  - **Some applications are "imperfectly partitionable"**
    - Many applications have some transactions that span multiple partitions
    - For these transactions, some form of concurrency control is needed

Rule of Thumb

- Always keep in mind
  - "Computation is FAST"
  - "Communication is SLOW"
- If you can "do extra work in an initialization step" to reduce the work done in each time step, it is generally well worth the effort
- Collect all of your "communication at one point" in your program


Q&A