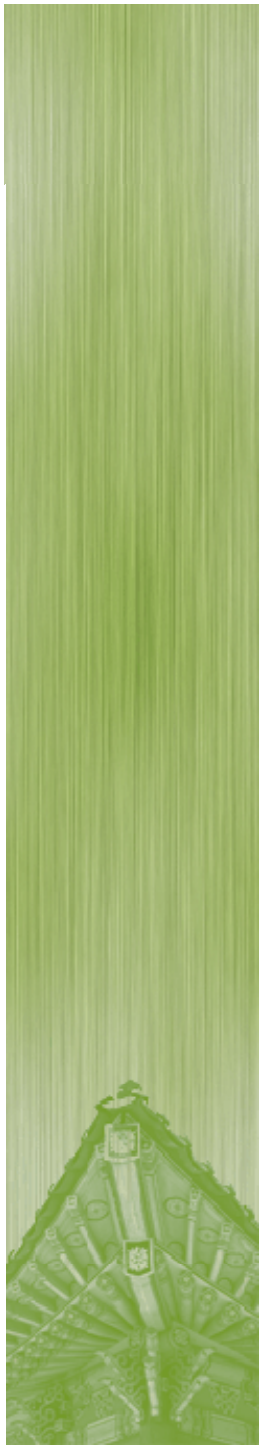
Final Exam

- 12:00 – 13:20, December 14 (Monday), 2009
- #330110 (odd student id)
- #330118 (even student id)

- Scope:
  - Everything
- Closed-book exam
- Final exam scores will be posted in the lecture homepage



Parallel Programming

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

SKK UNIVERSITY

Challenges

- **Difficult to write parallel programs**
  - Most programmers think sequentially
  - Performance vs. correctness tradeoffs
  - Missing good parallel abstractions

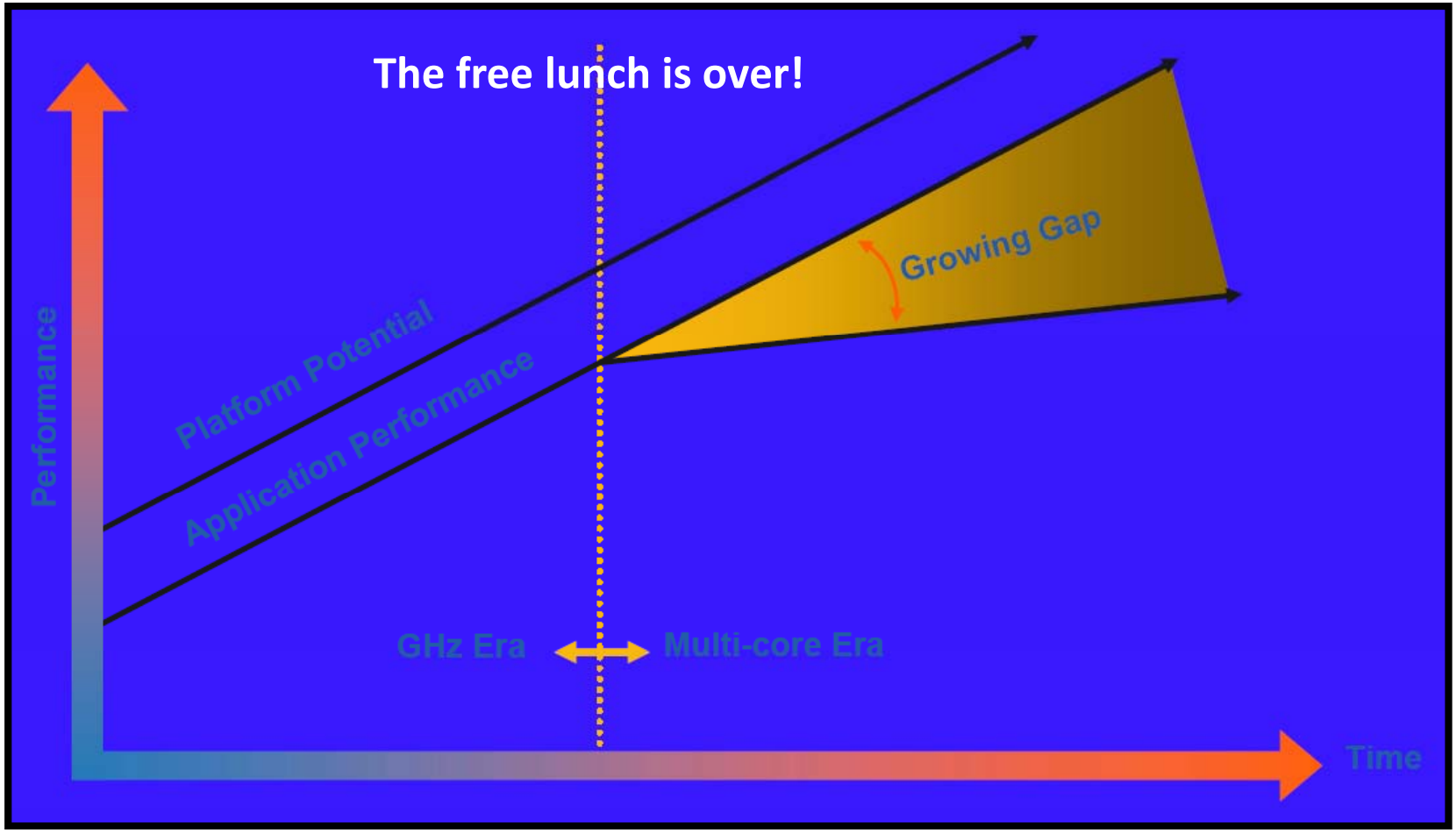
- **Automatic parallelization by compilers**
  - Works with some applications:

    loop parallelism, reduction
  - Unclear how we can apply to other complex applications

Concurrent vs. Parallel

- **Concurrent program**
  - A program containing two or more processes (threads)
- **Parallel program**
  - A concurrent program in which each process (thread) executes on its own processor, and hence the processes (threads) execute in parallel
- **Concurrent programming has been around for a while, so why do we bother?**
  - Logical parallelism (GUIs, asynchronous events, …) vs. Physical parallelism (for performance)

Think Parallel or Perish



Parallel Prog. Models (1)

- **Shared address space programming model**
  - Single address space for all CPUs
  - Communication through regular load/store (implicit)
  - Synchronization using locks and barriers (explicit)
  - Ease of programming
  - Complex hardware for cache coherence

  - Thread APIs (Pthreads, …)
  - OpenMP
  - Intel TBB (Thread Building Blocks)

Parallel Prog. Models (2)

- **Message passing programming model**
  - Private address space per CPU
  - Communication through message send/receive over network interface (explicit)
  - Synchronization using blocking messages (implicit)
  - Need to program explicit communication
  - Simple hardware
    - No cache coherence supporting hardware

  - RPC (Remote Procedure Calls)
  - PVM (obsolete)
  - MPI (de factor standard)

Parallel Prog. Models (3)

- **Parallel programming models vs. Parallel computer architectures**

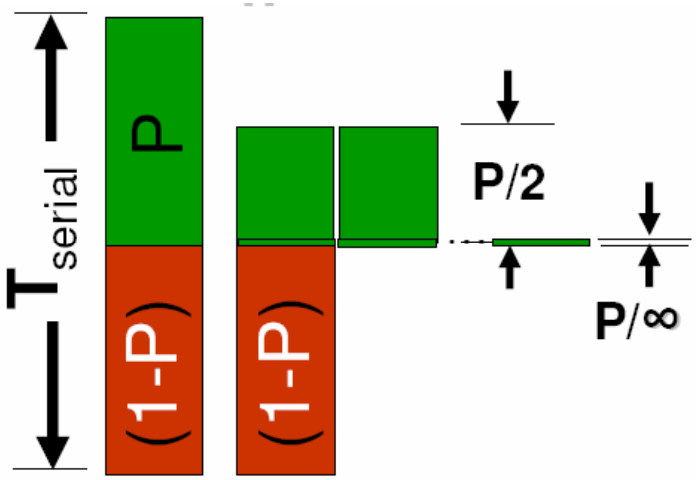
		Parallel Architectures	
		Shared-memory	**Distributed-memory**
Parallel Prog. Models	**Single Address Space**	Pthreads OpenMP	(CC) NUMA Software DSM
	Message Passing	Multi-processes MPI	MPI

Speedup (1)

- **Amdahl's Law**

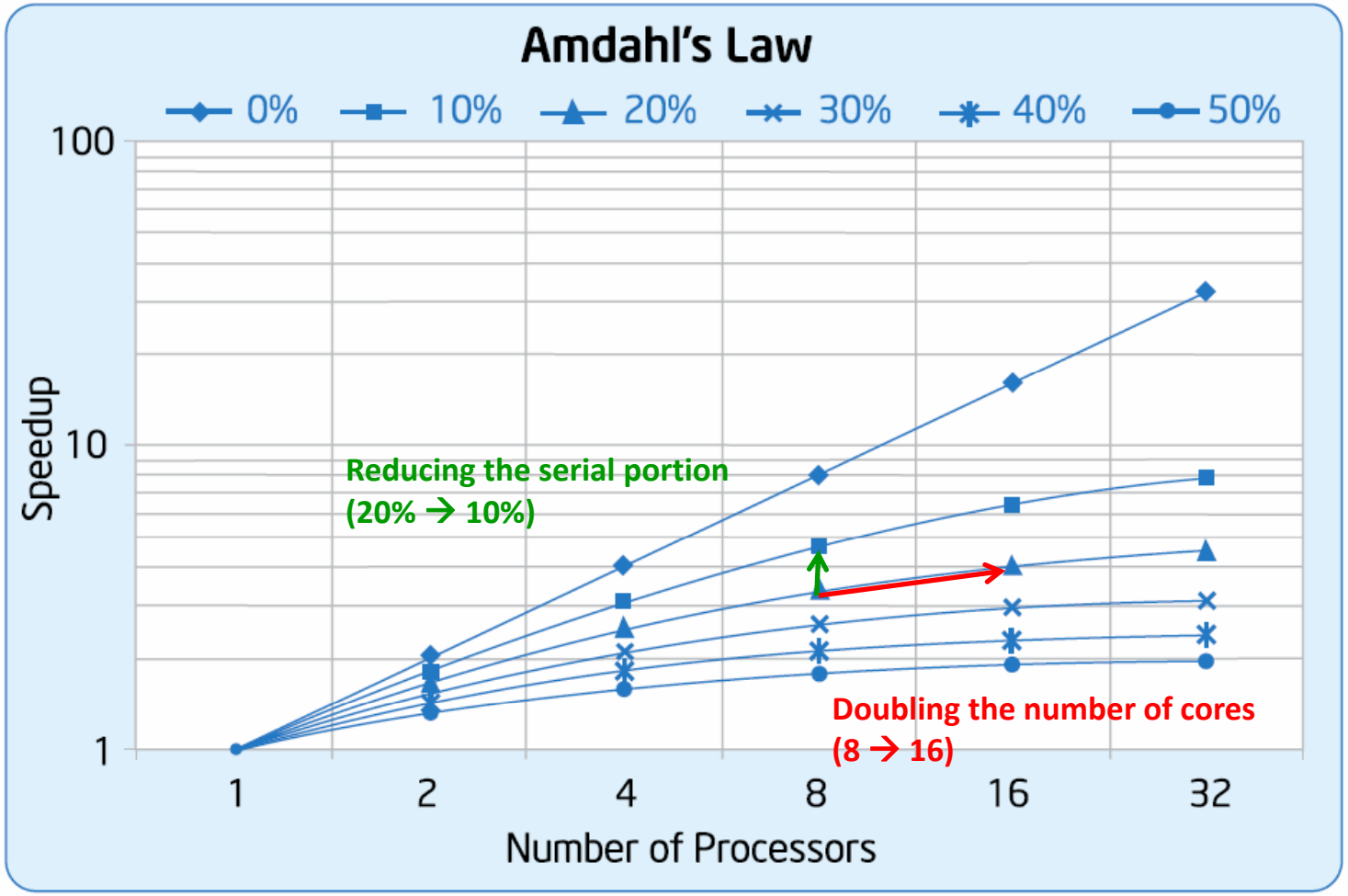
$$Speedup = \frac{1}{(1-P) + P/n}$$

  - $S = (1- P)$: the time spent executing the serial portion
    n : the number of processor cores



- A theoretical basis by which the speedup of parallel computations can be estimated.
- The theoretical upper limit of speedup is limited by the serial portion of the code.

Speedup (2)


Amdahl's Law
0%
10%
20%
30%
40%
50%
100
10
1
Speedup
Reducing the serial portion
(20% → 10%)
Doubling the number of cores
(8 → 16)
1
2
4
8
16
32
Number of Processors

Speedup (3)

- **Speedup in reality**

$$Practical\ Speedup = \frac{1}{(1-P) + P/n + H(n)}$$

  - $H(n)$: overhead
    - Thread management & scheduling
    - Communication & synchronization
    - Load balancing
    - Extra computation
    - Operating system overhead
- **For better speedup,**
  - Parallelize more (increase P)
  - Parallelize effectively (reduce $H(n)$)

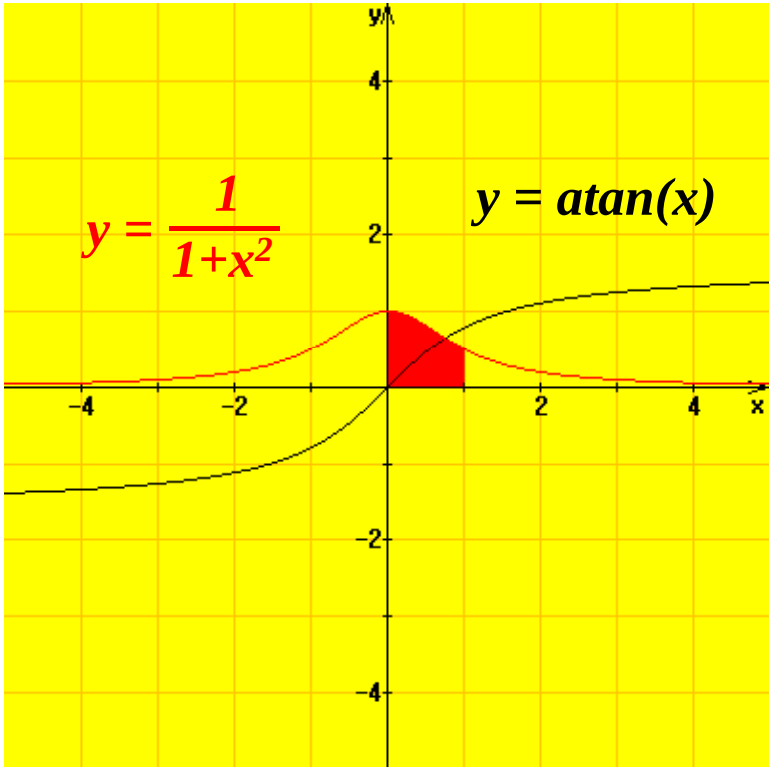
Calculating Pi (1)

- $\tan \frac{\pi}{4} = 1 \Rightarrow \text{atan}(1) = \frac{\pi}{4}$

- $4 \int_0^1 \frac{1}{1+x^2} \, dx$

  $= 4 \, (\text{atan}(1) - \text{atan}(0))$

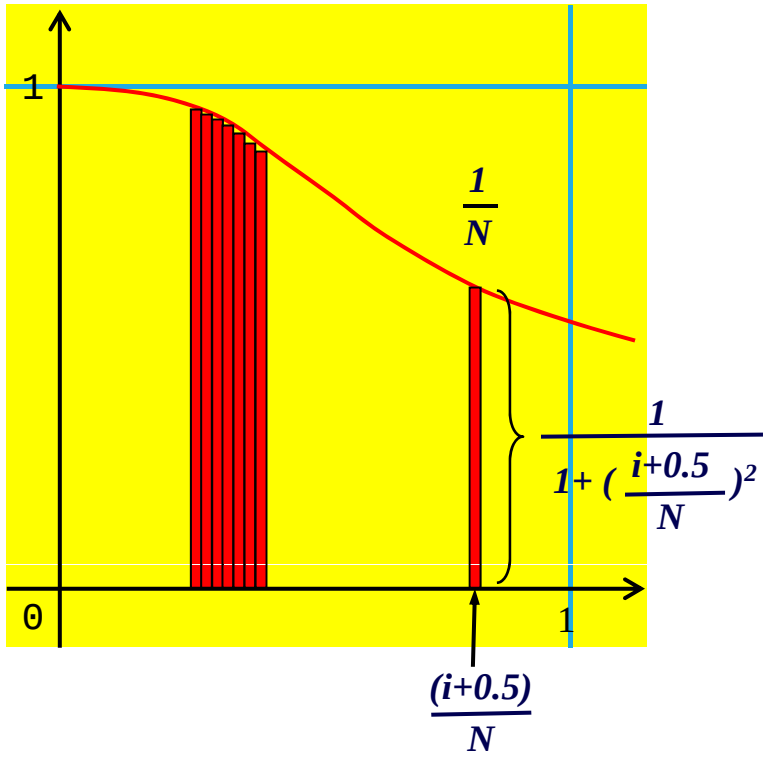
  $= 4 \, (\frac{\pi}{4} - 0) = \pi$



Calculating Pi (2)

- Pi: Sequential version

```
#define N       20000000
#define STEP    (1.0 / (double) N)

double compute ()
{
    int i;
    double x;
    double sum = 0.0;

    for (i = 0; i < N; i++)
    {
        x = (i + 0.5) * STEP;
        sum += 4.0 / (1.0 + x * x);
    }
    return sum * STEP;
}
```



Threads API (1)

- **Threads are the natural unit of execution for parallel programs on shared-memory hardware**
  - Windows threading API
  - POSIX threads (Pthreads)
- **Threads have their own**
  - Execution context
  - Stack
- **Threads share**
  - Address space (code, data, heap)
  - Resources (PID, open files, etc.)

Threads API (2)

- Pi: Pthreads version

```
#include <pthread.h>
#define N       20000000
#define STEP    (1.0 / (double) N)
#define NTHREADS 8

double pi = 0.0;
pthread_mutex_t lock =
    PTHREAD_MUTEX_INITIALIZER;
int main()
{
  int i;
  pthread_t tid[NTHREADS];
  for (i = 0; i < NTHREADS; i++)
    pthread_create (&tid[i], NULL,
          compute, (void *) i);
  for (i = 0; i < NTHREADS; i++)
    pthread_join (tid[i], NULL);
}
```

```
void *compute (void *arg)
{
  int i;
  double x;
  double sum = 0.0;
  int id = (int) arg;

  for (i=id; i<N; i += NTHREADS)
  {
    x = (i + 0.5) * STEP;
    sum += 4.0 / (1.0 + x * x);
  }
  pthread_mutex_lock (&lock);
  pi += (sum * STEP);
  pthread_mutex_unlock (&lock);
}
```

Threads API (3)

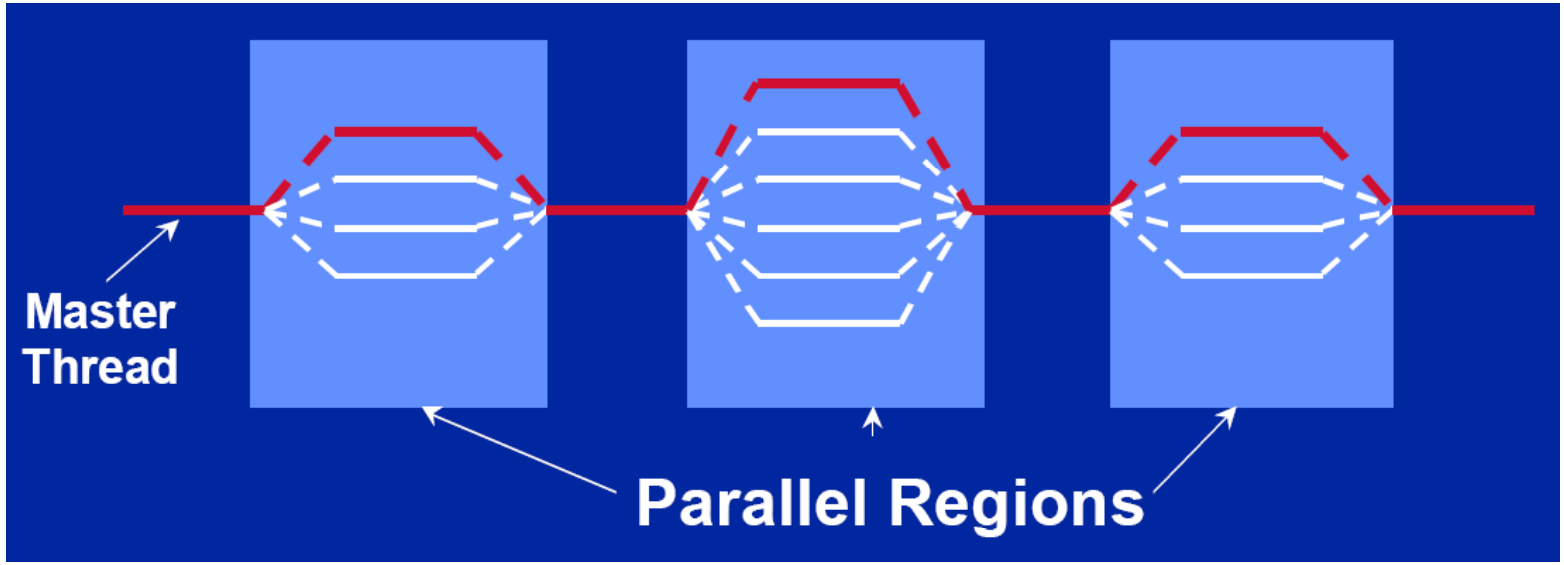
- **Pros**
  - Flexible and widely available
  - The thread library gives you detailed control over the threads
  - Performance can be good
- **Cons**
  - YOU have to take detailed control over the threads.
  - Relatively low level of abstraction
  - No easy code migration path from sequential program
  - Lack of structure means error-prone

OpenMP (1)

- **What is OpenMP?**
  - A set of compiler directives and library routines for portable parallel programming on shared memory systems
    - Fortran 77, Fortran 90, C, and C++
    - Multi-vendor support, for both Unix and Windows
  - Standardizes loop-level (data) parallelism
  - Combines serial and parallel code in single source
  - Incremental approach to parallelization
  - http://www.openmp.org, http://www.compunity.org
    - Standard documents, tutorials, sample codes

OpenMP (2)

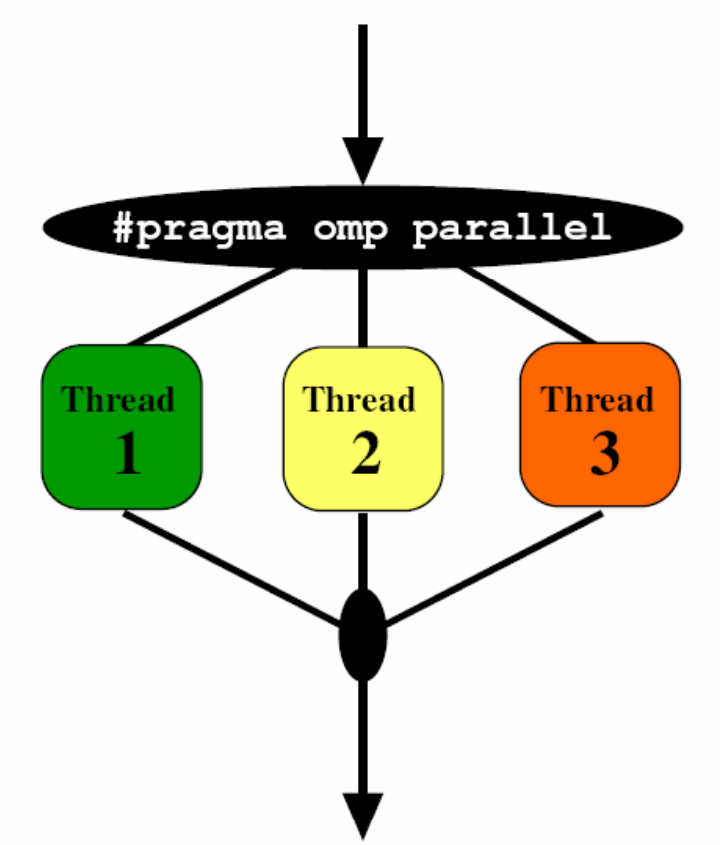
- **Fork-join model**
  - Master thread spawns a team of threads as needed
  - Synchronize when leaving parallel region (join)
  - Only master executes sequential part



OpenMP (3)

- **Parallel construct**

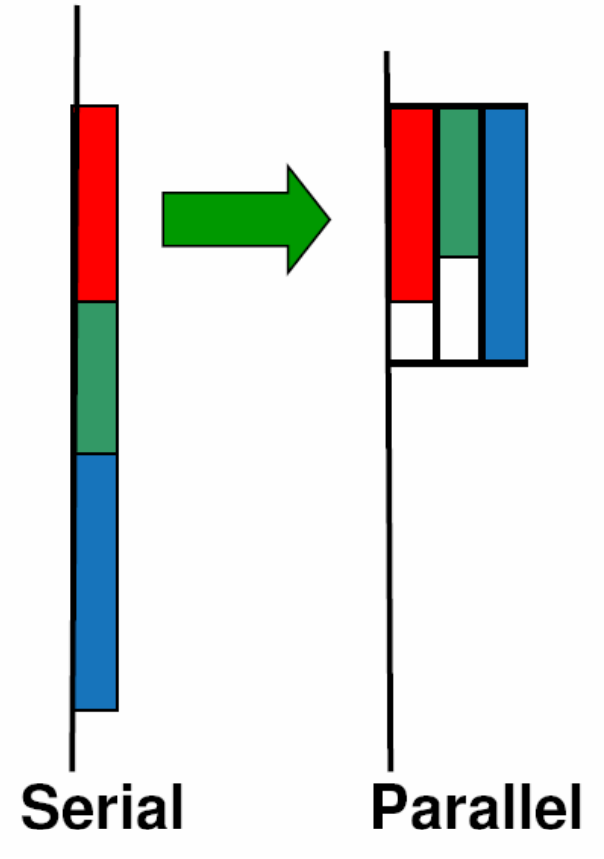
```
#pragma omp parallel
{
        task();
}
```



OpenMP (4)

- Work-sharing construct: sections

```
#pragma omp parallel sections
{
#pragma omp section
        phase1();
#pragma omp section
        phase2();
#pragma omp section
        phase3();
}
```

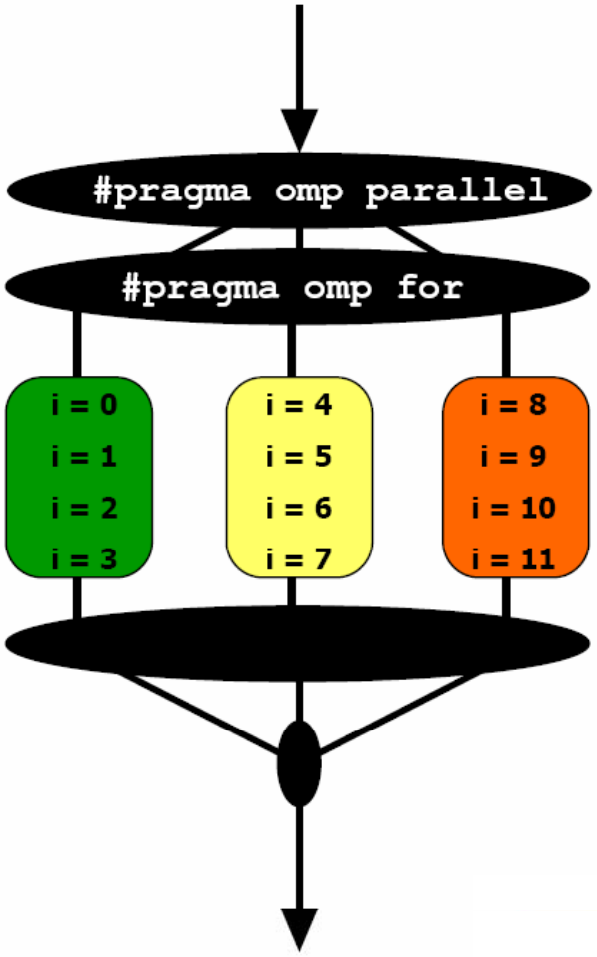


OpenMP (5)

- Work-sharing construct: loop

```
#pragma omp parallel for
  for (i = 0; i < 12; i++)
    c[i] = a[i] + b[i];
```

```
#pragma omp parallel
{
  task ();
#pragma omp for
  for (i = 0; i < 12; i++)
    c[i] = a[i] + b[i];
}
```



OpenMP (6)

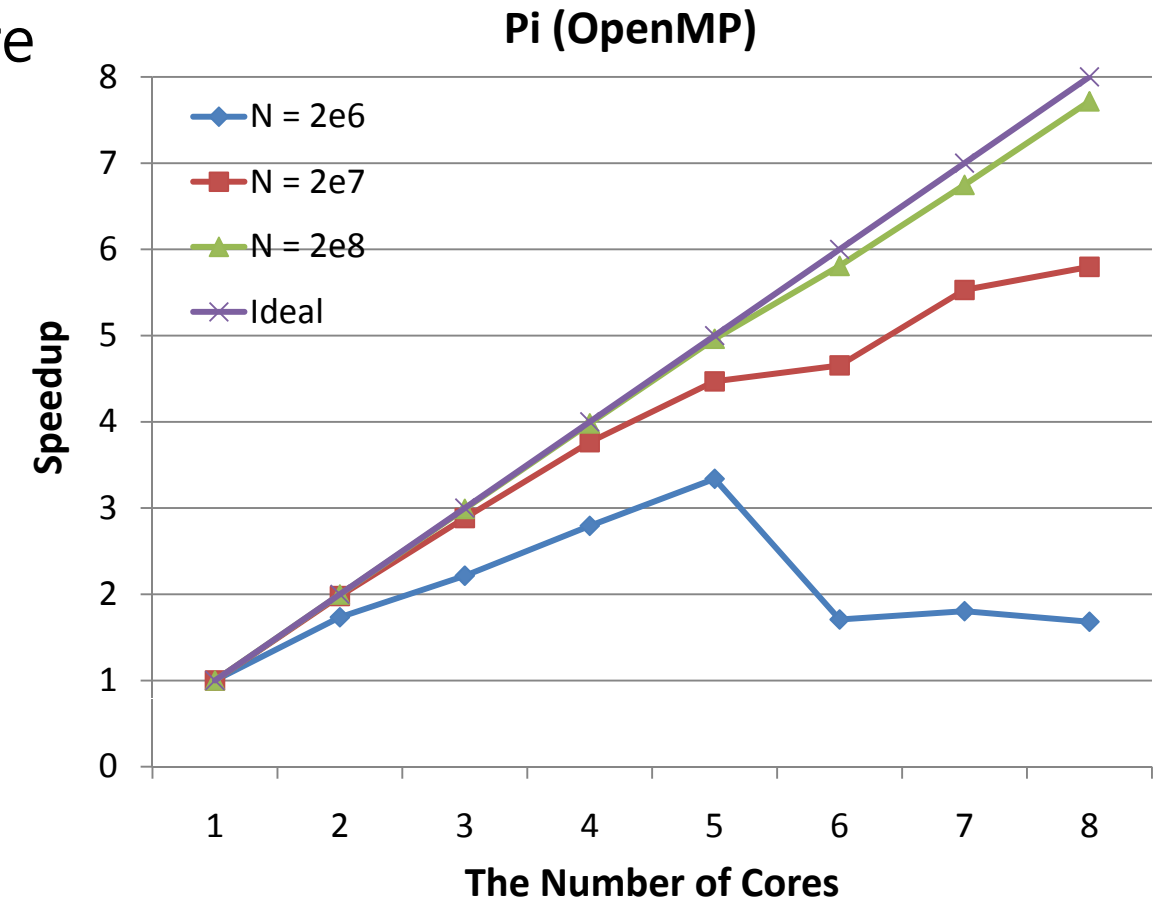
- Pi: OpenMP version

```
#ifdef _OPENMP
#include <omp.h>
#endif

#define N     20000000
#define STEP  (1.0/(double) N)

int main()
{
  double pi;

  pi = compute();
}
```

```
double compute ()
{
  int i;
  double x;
  double sum = 0.0;

#pragma omp parallel for
private(x) reduction(+:sum)
  for (i = 0; i < N; i ++)
  {
    x = (i + 0.5) * STEP;
    sum += 4.0 / (1.0 + x * x);
  }
  return sum * STEP;
}
```

OpenMP (7)

- **Environment**
  - Dual Quad-core Xeon (8 cores)
  - 2.4GHz
  - Intel OpenMP Compiler



OpenMP (8)

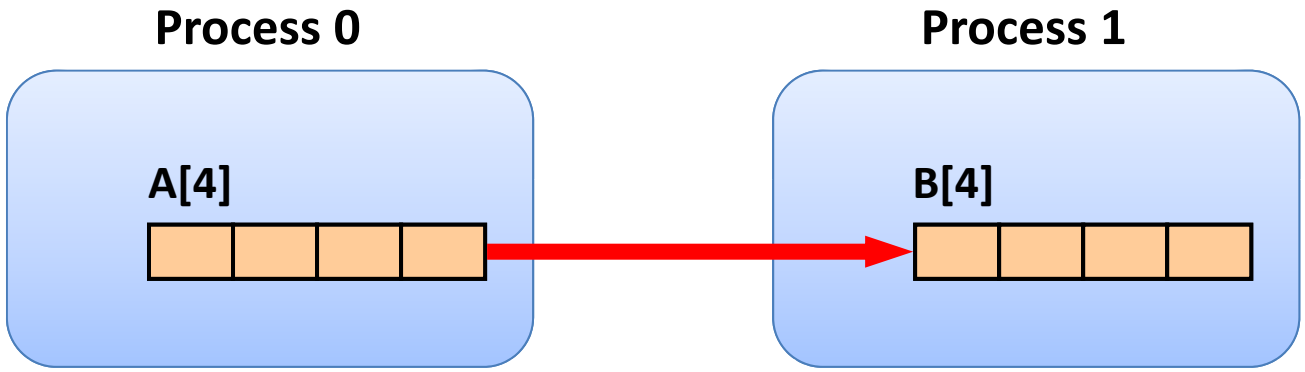
- **Pros**
  - Simple and portable
  - Single source code for both serial and parallel version
  - Incremental parallelization
- **Cons**
  - Primarily for bounded loops over built-in types
    - Only for Fortran-style C/C++ programs
    - Pointer-chasing loops?
  - Performance may be degraded due to
    - Synchronization overhead
    - Load imbalancing
    - Thread management overhead

MPI (1)

- **Message Passing Interface**
  - Library standard defined by a committee of vendors, implementers, and parallel programmers
  - Used to create parallel programs based on message passing
  - Available on almost all parallel machines in C/Fortran
  - Two phases
    - MPI 1: Traditional message-passing
    - MPI 2: Remote memory, parallel I/O, dynamic processes
  - Tasks are typically created when the jobs are launched
  - http://www.mpi-forum.org

MPI (2)

- **Point-to-point communication**
  - Sending and receiving messages

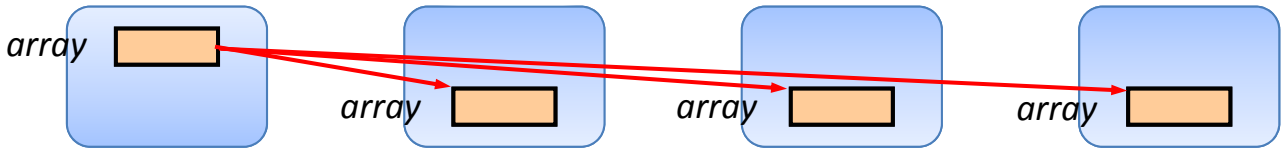



```
MPI_Recv (B, 4, MPI_INT, MPI_ANY_SOURCE,
          MPI_ANY_TAG, MPI_COMM_WORLD);
```

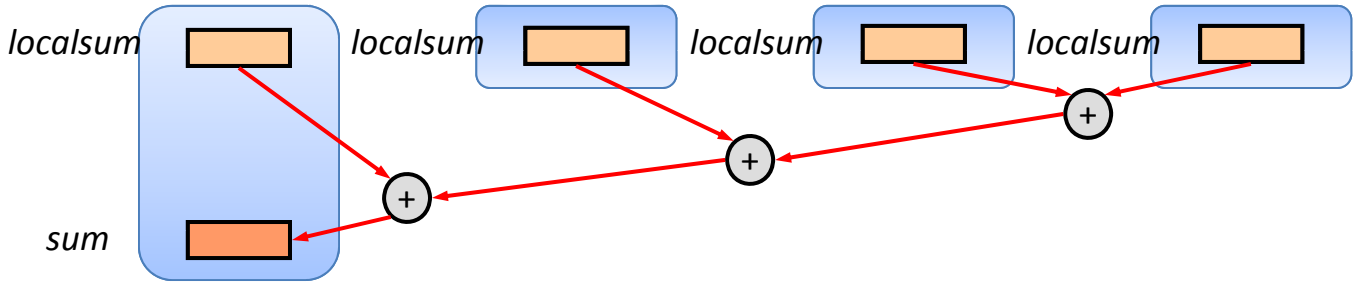
```
MPI_Send (A, 4, MPI_INT, 1, 1000,
          MPI_COMM_WORLD);
```

MPI (3)

- **Collective communication**
  - Broadcast
    - `MPI_Bcast (array, 100, MPI_INT, 0, comm);`



  - Reduce
    - `MPI_Reduce (&localsum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, comm);`



MPI (4)

- Pi: MPI version

```
#include “mpi.h”
#include <stdio.h>
#define N     20000000
#define STEP  (1.0 / (double) N)
#define f(x)  (4.0 / (1.0 + (x)*(x))
int main (int argc, char *argv[]) {
   int i, numprocs, myid, start, end, n;
   double sum = 0.0, mypi, pi;
   MPI_Init (&argc, &argv);
   MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
   MPI_Comm_rank (MPI_COMM_WORLD, &myid);
   start = (N / numprocs) * myid + 1;
   end = (myid==numprocs-1)? n : (n/numprocs) * (myid+1);
   for (i = start; i <= end; i++)
       sum += f(STEP * ((double) i + 0.5));
   myip = STEP * sum;
   MPI_Reduce (&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
   MPI_Finalize ();
}
```

MPI (5)

- **Pros**
  - Runs on either shared or distributed memory architectures
  - Can be used on a wider range of problems
  - Distributed memory computers are less expensive than large shared memory computers
- **Cons**
  - Requires more programming changes to go from serial to parallel version
  - Can be harder to debug
  - Performance is limited by the communication network between the nodes

Parallel Benchmarks (1)

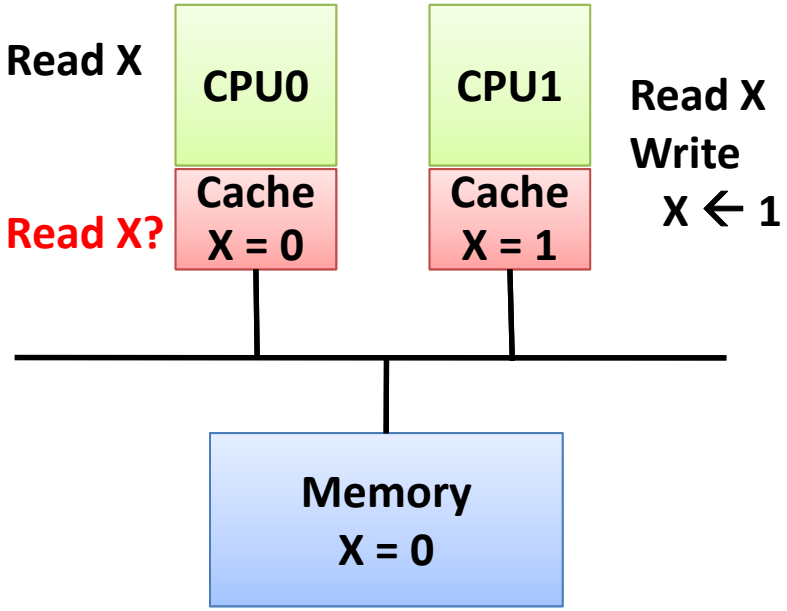
- **Linpack**
  - Matrix linear algebra
  - Basis for measuring "Top500 Supercomputing sites" (http://www.top500.org)
- **SPECrate**
  - Parallel run of SPEC CPU programs
  - Job-level (or task-level) parallelism
- **SPLASH**
  - Stanford Parallel Applications for Shared Memory
  - Mix of kernels and applications
  - Strong scaling: keep the problem size fixed

Parallel Benchmarks (2)

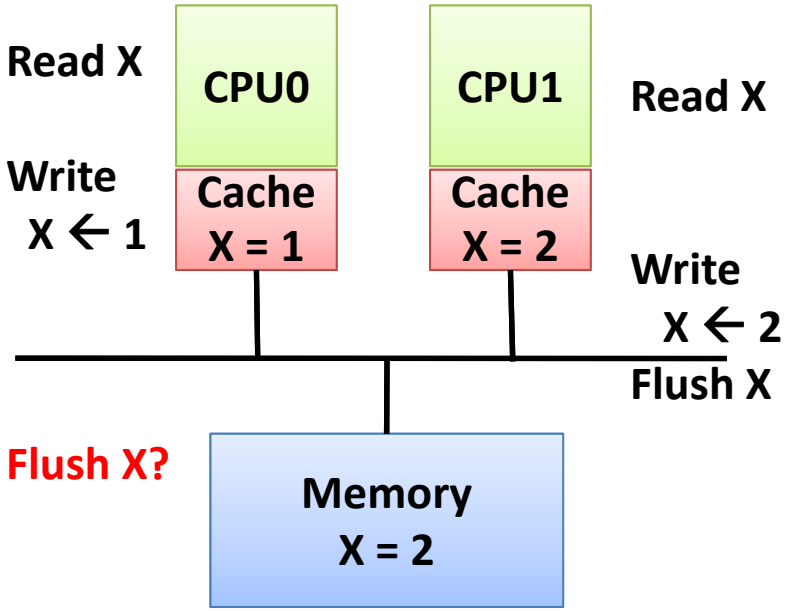
- **NAS parallel benchmark suite**
  - By NASA Advanced Supercomputing (NAS)
  - Computational fluid dynamics kernels
- **PARSEC suite**
  - Princeton Application Repository for Shared Memory Computers
  - Multithreaded applications using Pthreads and OpenMP

Cache Coherence (1)

- Examples



Invalidate all copies before allowing a write to proceed



Disallow more than one modified copy

Cache Coherence (2)

- **Cache coherence protocol**
  - A set of rules to maintain the consistency of data stored in the local caches as well as in main memory
  - Ensures multiple read-only copies and exclusive modified copy
  - Types
    - Snooping-based (or "Snoopy")
    - Directory-based
  - Write strategies
    - Invalidation-based
    - Update-based

Cache Coherence (3)

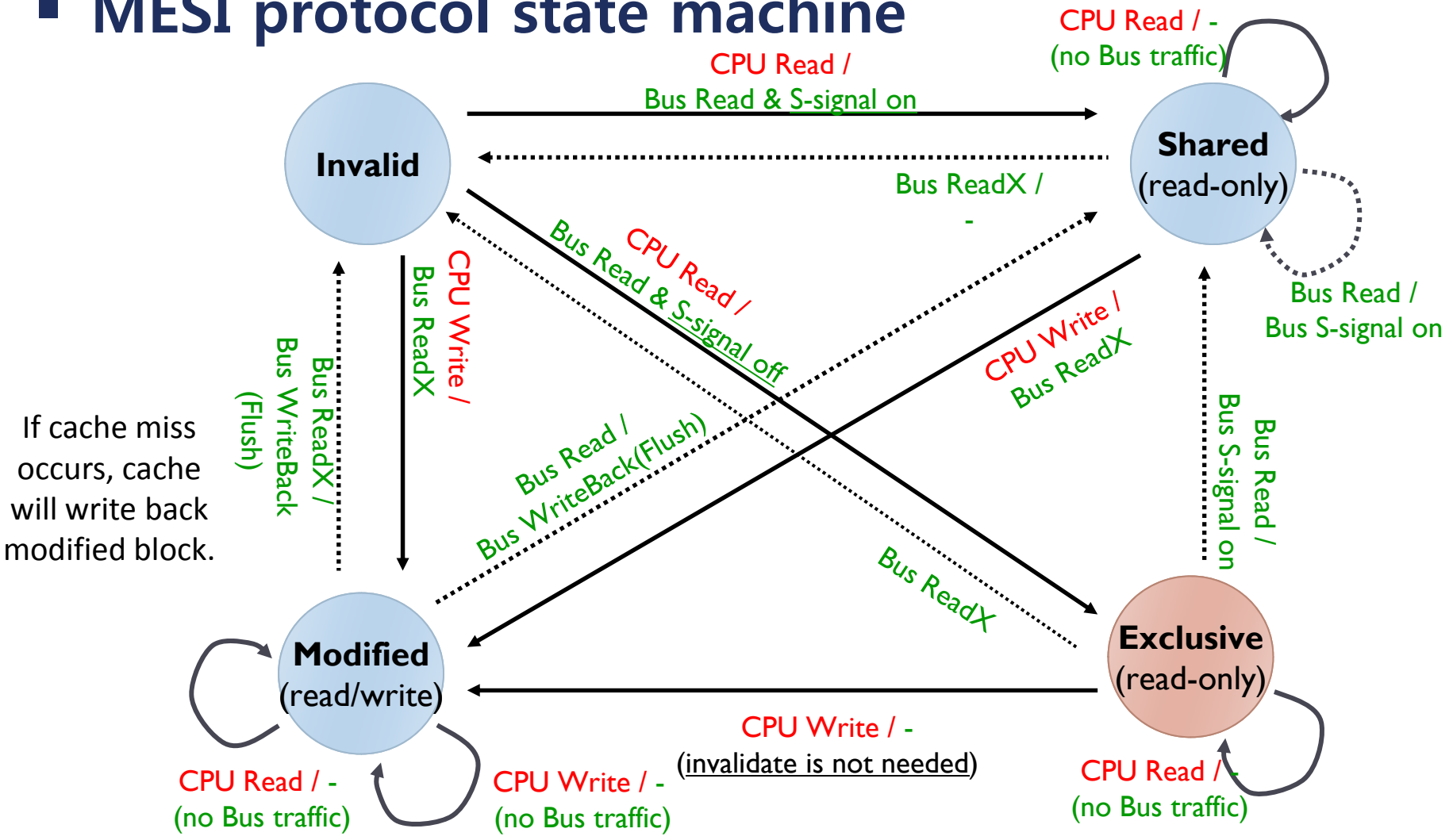
- **Snooping protocol**
  - All cache controllers monitor (or snoop) on the bus
    - Send all requests for data to all processors
    - Processors snoop to see if they have a shared block
    - Requires broadcast, since caching info resides at processors
    - Works well with bus (natural broadcast)
    - Dominates for small scale machines
  - Cache coherence unit
    - Cache block (line) is the unit of management
    - False sharing is possible: Two processors share the same cache line but not the actual word
    - Coherence miss: Invalidate can cause a miss for the data read before

Cache Coherence (4)

- **MESI protocol: invalidation-based**
  - Modified (M)
    - Cache has only one copy and copy is modified
    - Memory is not up-to-date
  - Exclusive (E)
    - Cache has only one copy and copy is unmodified (clean)
    - Memory is up-to-date
  - Shared (S)
    - Copies may exist in other caches and all are unmodified
    - Memory is up-to-date
  - Invalid (I)
    - Not in cache

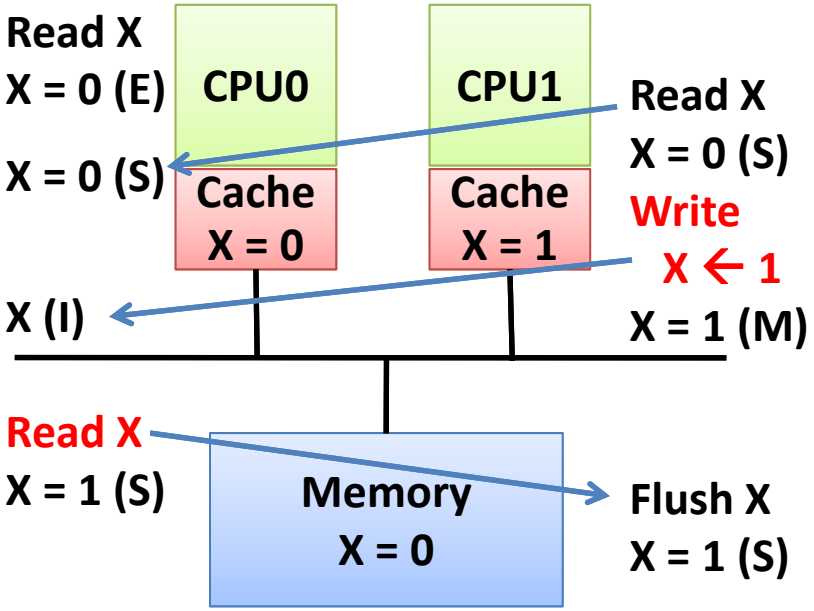
Cache Coherence (5)

- MESI protocol state machine

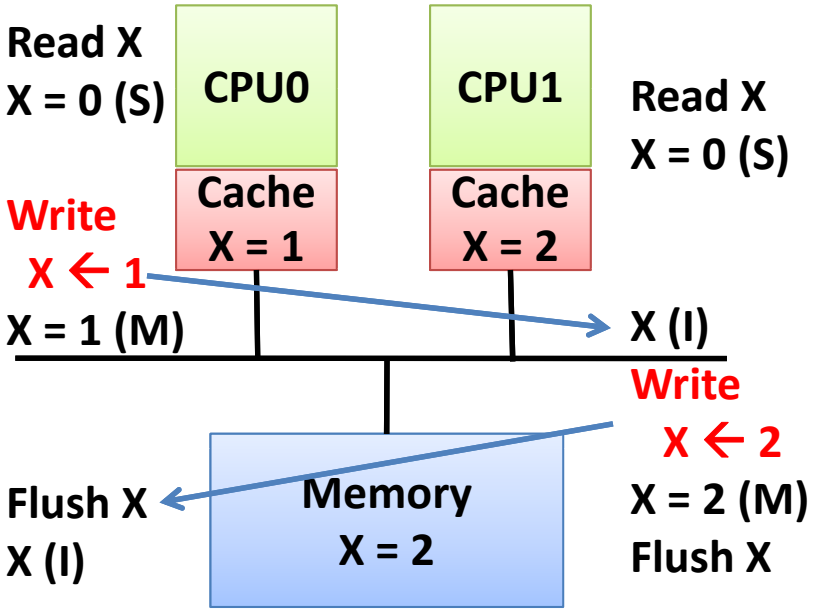


Cache Coherence (6)

- Examples revisited



Invalidate all copies before allowing a write to proceed

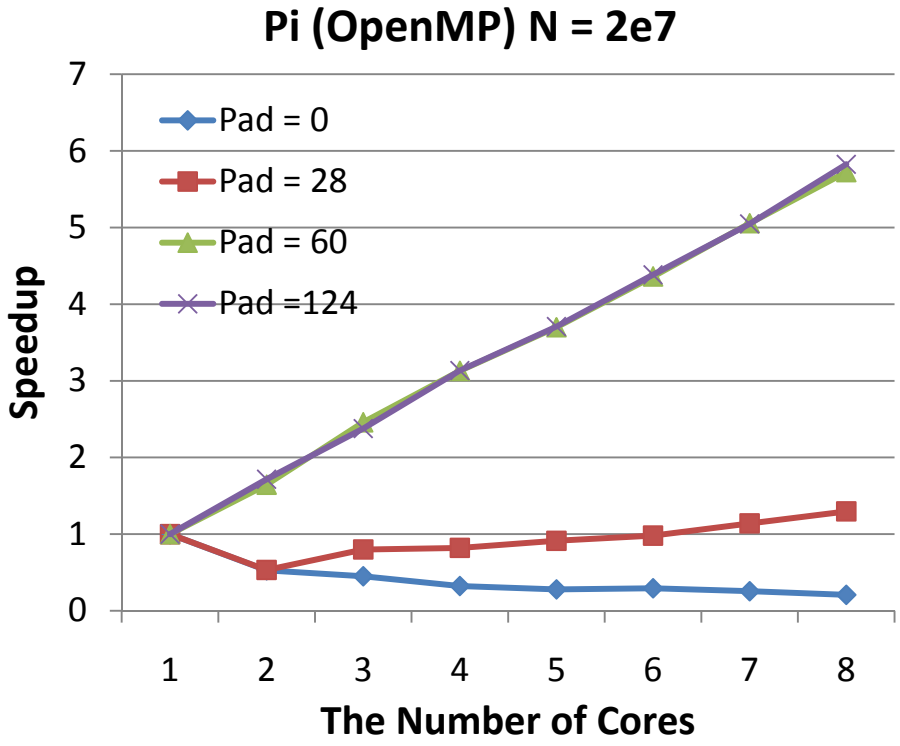


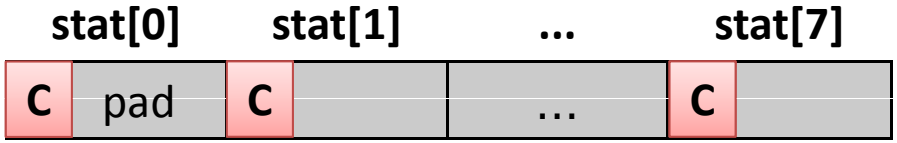
Disallow more than one modified copy

Cache Coherence (7)

- False sharing

```
struct thread_stat {
  int count;
  char pad[PADS];
} stat[8];

double compute ()  {
  int i, id;
  double x, sum = 0.0;

#pragma omp parallel for
private(x) reduction(+:sum)
  for (i = 0; i < N; i ++)  {
    int id=omp_get_thread_num();
    stat[id].count++;
    x = (i + 0.5) * STEP;
    sum += 4.0 / (1.0 + x * x);
  }
  return sum * STEP;
}
```

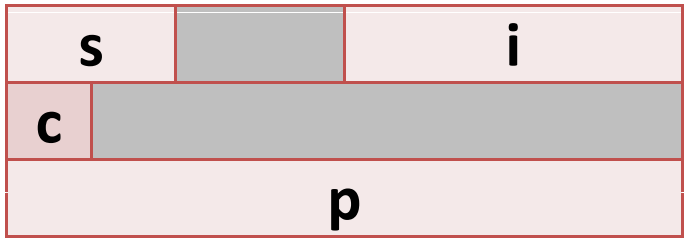




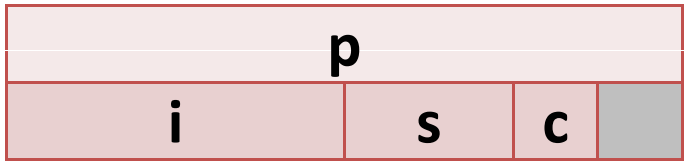
Cache-Aware Programming (1)

- **Pack data more tightly**
  - Usually, compilers do not pack structures

```
struct Loose {
    short s;
    int i;
    char c;
    Foo* p;
};
```




```
struct Tight {
    Foo* p;
    int i;
    short s;
    char c;
};
```



Cache-Aware Programming (2)

- **Work in the cache**
  - Reduce the working set
  - Divide a problem into smaller subproblems
  - Reorder steps in the code

- **Read-only data**
  - Separate read-only data from read-write data
  - Annotate constants with **`const`** keyword
  - Read-only data are stored in separate sections
  - If possible, separate read-mostly variables

Cache-Aware Programming (3)

- **Read-write data**
  - Group read-write variables which are used together into a structure
  - Move read-write variables which are often written to by different threads onto their own cache line
    - Reduce false sharing
    - May require padding
  - If a variable is used by multiple threads, but every use is independent, move it into Thread Local Storage (TLS)

Cache-Aware Programming (4)

- **Lock variables**
  - Keep the lock and the associated data on distinct cache line.
  - If a lock protects data that is frequently uncontended, try to keep the lock and the data on the same cache line.

Cache-Aware Programming (5)

- **Thread affinity**
  - Avoid moving a thread from one core to another
    - Reduce context switching
    - Reduce cache misses
    - Reduce TLB misses
  - Bind a process
    - `sched_setaffinity () / sched_getaffinity()`
  - Bind a thread
    - `pthread_setaffinity_np()`
    - `pthread_getaffinity_np()`
    - `pthread_attr_setaffinity_np()`
    - `pthread_attr_getaffinity_np()`

Summary: Why Hard?

- **Impediments to parallel computing**
  - Parallelization
    - Parallel algorithms which maximizes concurrency
    - Eliminating the serial portion as much as possible
    - Lack of standardized APIs, environments, and tools
  - Correct parallelization
    - Shared resource identification
    - Difficult to debug (data races, deadlock, …)
    - Memory consistency
  - Effective parallelization
    - Communication & synchronization overhead
    - Problem decomposition, granularity, load imbalance, affinity..
    - Architecture dependency