A Programmer’s View of Shared and Distributed Memory Architectures

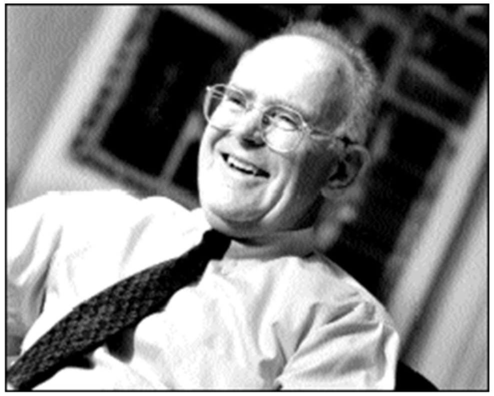
Overview

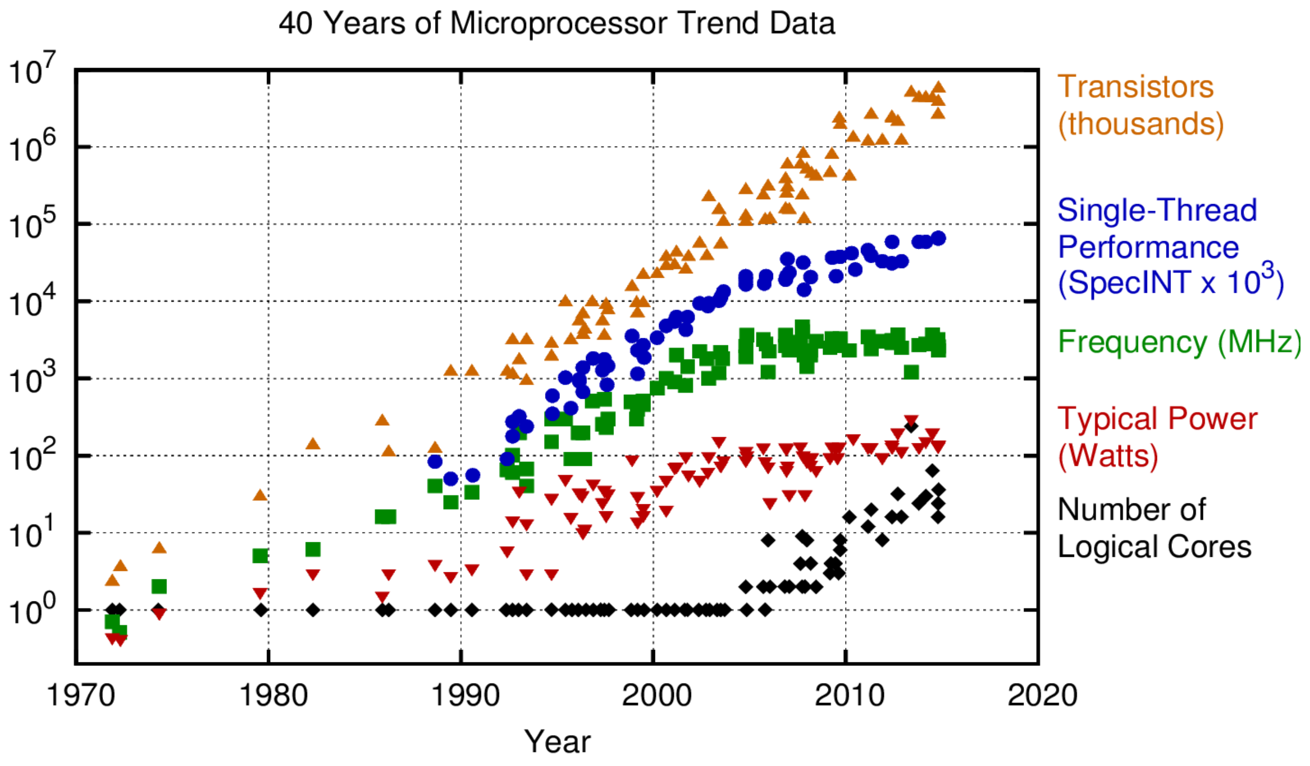
- Shared-memory
  - Architecture: chip has some number of cores (e.g., Intel Skylake has up to 18 cores depending on the model) with common memory
  - Application program is decomposed into a number of threads, which run on these cores; data structures are in common memory
  - Threads communicate by reading and writing memory locations
  - Programming systems: pThreads, OpenMP, Intel TBB

- Distributed-memory
  - Architecture: network of machines (Stampede II: 4,200 KNL hosts) with no common memory
  - Application program and data structures are partitioned into processes, which run on machines
  - Processes communicate by sending and receiving messages
  - Programming: MPI communication library

Shared-memory Architectures for Programmers

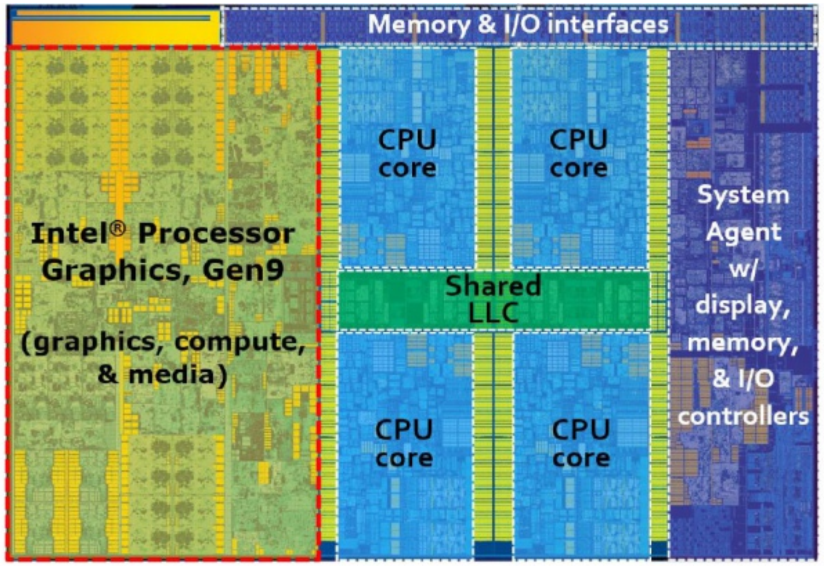
Moore's Law


Gordon Moore (Intel)



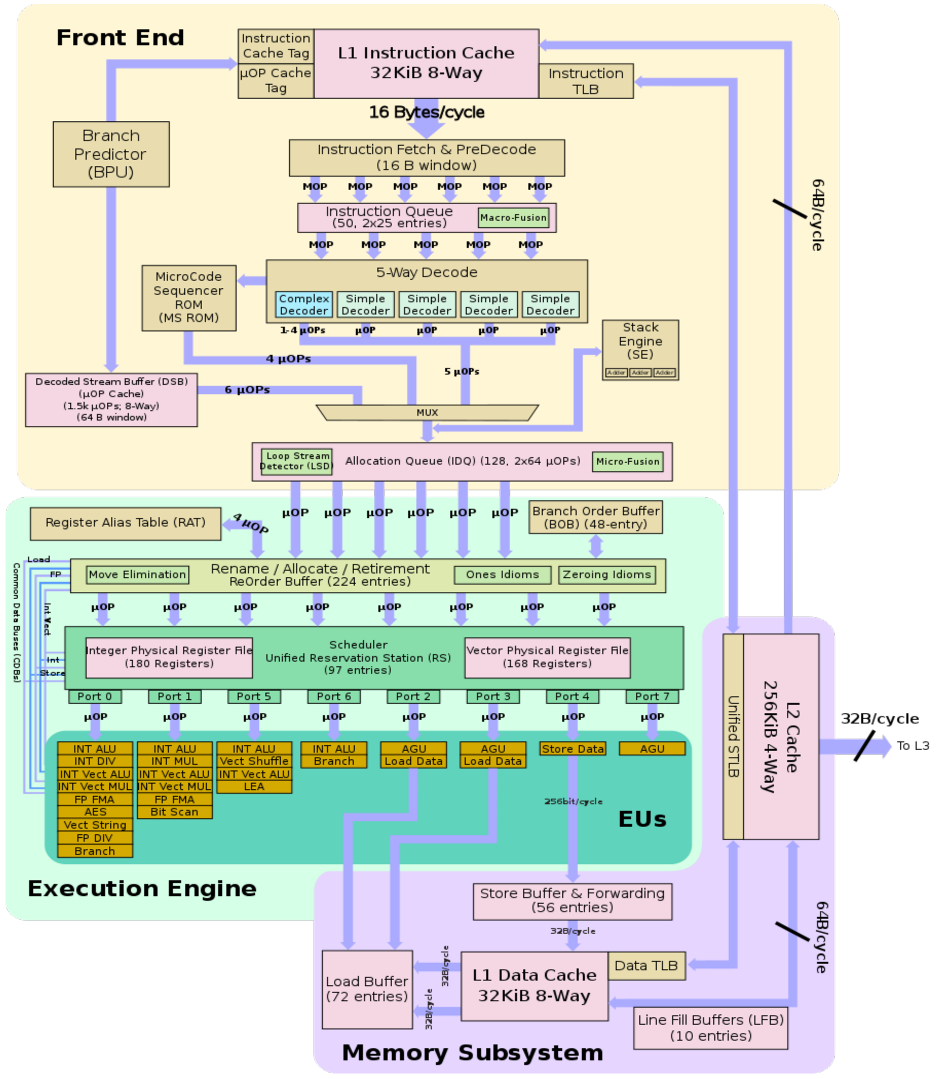
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Intel Skylake chip



Memory & I/O interfaces
CPU core
CPU core
Intel® Processor Graphics, Gen9 (graphics, compute, & media)
Shared LLC
System Agent w/ display, memory, & I/O controllers
CPU core
CPU core

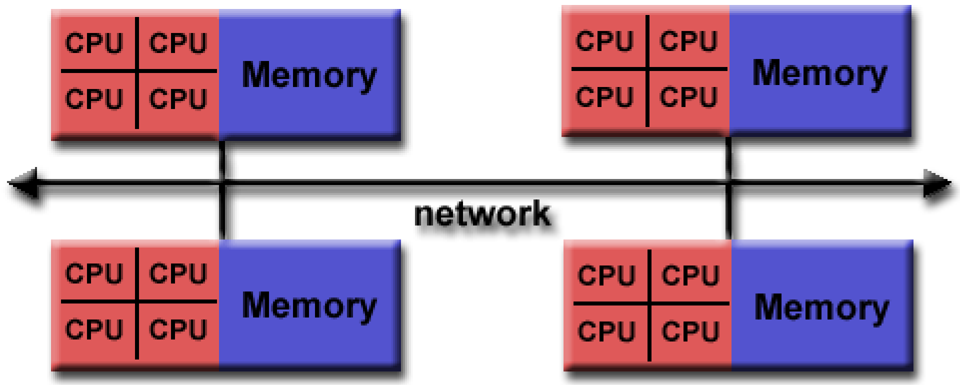
Chip



Front End
Instruction Cache Tag
µOP Cache Tag
L1 Instruction Cache 32KiB 8-Way
Instruction TLB
16 Bytes/cycle
Branch Predictor (BPU)
Instruction Fetch & PreDecode (16 B window)
MOP
Instruction Queue (50, 2x25 entries)
Macro-Fusion
MicroCode Sequencer ROM (MS ROM)
5-Way Decode
Complex Decoder
Simple Decoder
1-4 µOPs
µOP
4 µOPs
5 µOPs
Stack Engine (SE)
Decoded Stream Buffer (DSB) (µOP Cache) (1.5k µOPs; 8-Way) (64 B window)
6 µOPs
MUX
Loop Stream Detector (LSD)
Allocation Queue (IDQ) (128, 2x64 µOPs)
Micro-Fusion
64B/cycle
Register Alias Table (RAT)
4 µOP
Branch Order Buffer (BOB) (48-entry)
Move Elimination
Rename / Allocate / Retirement ReOrder Buffer (224 entries)
Ones Idioms
Zeroing Idioms
Common Data Buses (CDBs)
Load
FP
Int Vect
Int
Store
Integer Physical Register File (180 Registers)
Scheduler Unified Reservation Station (RS) (97 entries)
Vector Physical Register File (168 Registers)
Port 0
Port 1
Port 5
Port 6
Port 2
Port 3
Port 4
Port 7
INT ALU
INT DIV
INT Vect ALU
INT Vect MUL
FP FMA
AES
Vect String
FP DIV
Branch
INT MUL
Bit Scan
Vect Shuffle
LEA
AGU
Load Data
Store Data
256bit/cycle
EUs
Execution Engine
Unified STLB
L2 Cache 256KiB 4-Way
32B/cycle
To L3
Store Buffer & Forwarding (56 entries)
32B/cycle
Load Buffer (72 entries)
L1 Data Cache 32KiB 8-Way
Data TLB
Line Fill Buffers (LFB) (10 entries)
Memory Subsystem

Block diagram of each core[5]

Shared-memory m/c: cartoon picture



- Several multi-core chips connected by bus or network
- Single-address space for all cores but non-uniform memory access times

Typical latency numbers

From: *Latency numbers every HPC programmer should know*

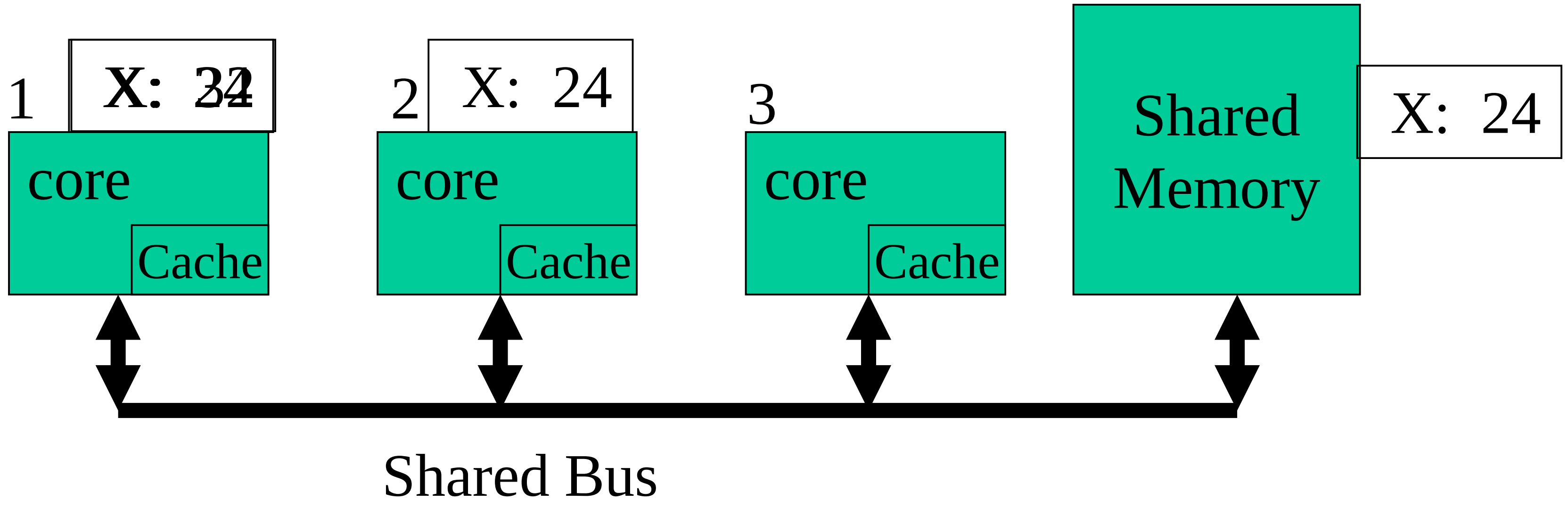
L1 cache reference/hit	1.5 ns	4 cycles
Floating-point add/mult/FMA operation	1.5 ns	4 cycles
L2 cache reference/hit	5 ns	12 ~ 17 cycles
L3 cache hit	16-40 ns	40-300 cycles
256MB main memory reference "Broadwell" E5-2690v4	75-120 ns	TinyMemBench on
Send 4KB message between hosts	1-10 µs	MPICH on 10-100Gbps
Read 1MB sequentially from disk ~200MB/sec hard disk (seek time would be additional latency)	5,000,000 ns	5 ms
Random Disk Access (seek+rotation)	10,000,000 ns	10 ms
Send packet CA->Netherlands->CA	150,000,000 ns	150 ms

Locality is important.

Architecture/software boundary

- Cache coherence
  - interaction between caching and program semantics
- Atomic instructions
  - interaction between threads: synchronization
- Memory consistency model
  - interaction between instruction reordering and program semantics

Cache coherence problem



- Core 1 loads X: obtains 24 from memory and caches it
- Core 2 loads X: obtains 24 from memory and caches it
- Core 1 stores 32 to X: its locally cached copy is updated
- Core 3 loads X: what value should it get?
  - memory and core 2 think it is 24
  - core 1 thinks it is 32
- Illusion that there is a single variable X is broken

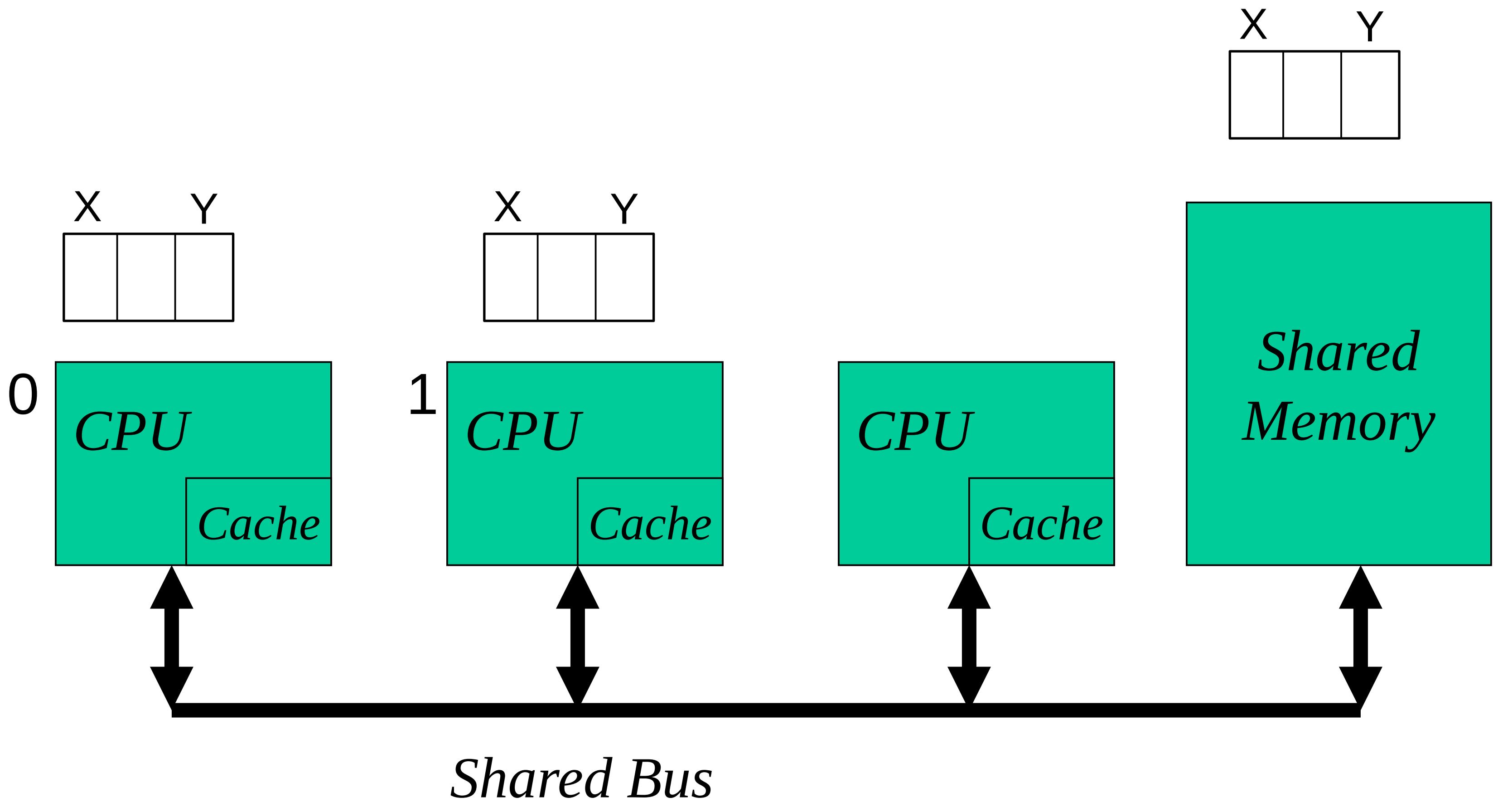
One solution

- Exclusive caching: ensure that at most one cache can have a given line at any time
- Implementation: snoopy caches
  - cache on each core ‘snoops’ (*i.e.* watches) for activity concerned with lines it has cached
  - load/store cache hit: perform operation just as in sequential machines
  - load/store cache miss:
    - perform bus cycle to obtain line
    - if some other cache has line, line is transferred to this cache and marked invalid in other cache
    - otherwise line is obtained from memory

Better solution: write-invalidate protocol

- Exclusive caching is too draconian
  - even read-only data cannot be in multiple caches
  - data written in one round that is read-only in next round cannot be in multiple caches
- Write-invalidate protocol
  - line can reside in several caches if all cores are reading from it
  - if a core wants to write to that line, line is invalidated from all other caches
- One implementation: MESI protocol

False-sharing



- Core 0 reads and writes X
- Core 1 reads and writes Y
- No true sharing, but if X and Y are on the same line, there will be a lot of invalidation misses

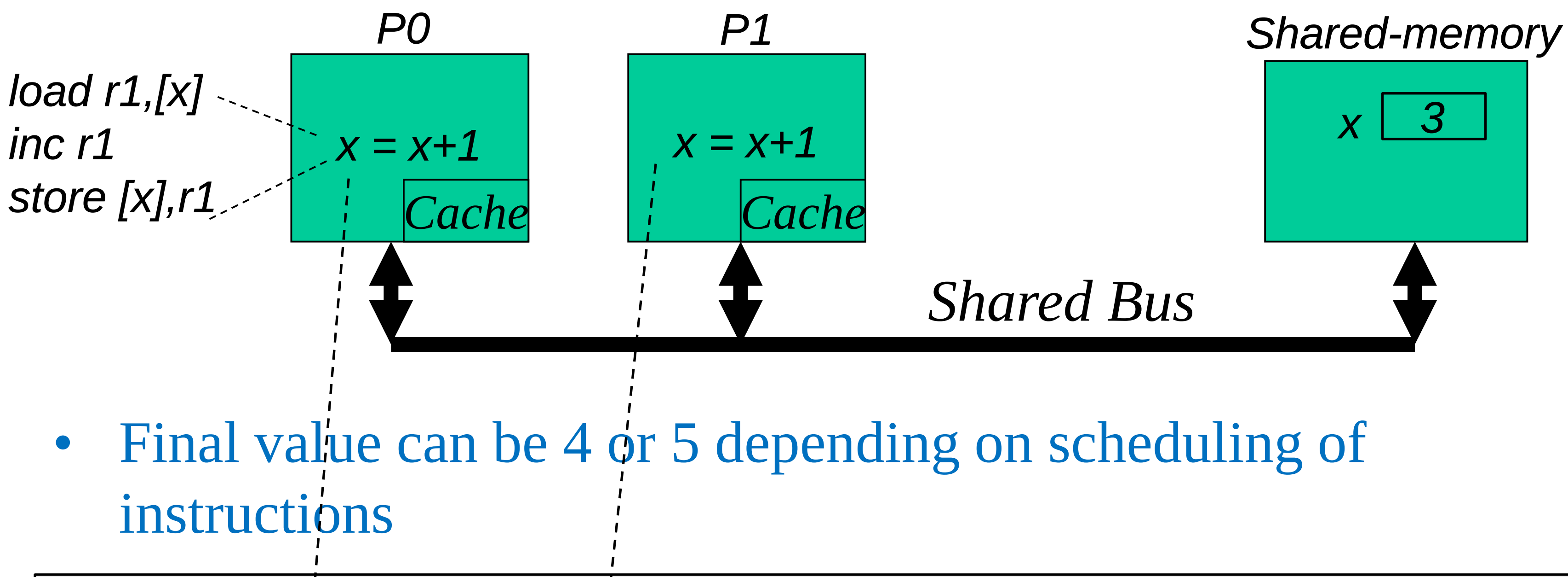
Summary

- Solution to cache-coherence:
  - snoopy caches and write-invalidate protocol
- True-sharing
  - a variable or array element is read and written by two or more cores repeatedly
- False-sharing
  - two or more cores read and write distinct variables or array elements that happen to be in the same cache line
- Sharing results in "ping-ponging" of cache lines between cores due to invalidations
  - reduces performance
  - to improve performance, try to reduce sharing of cache lines between cores

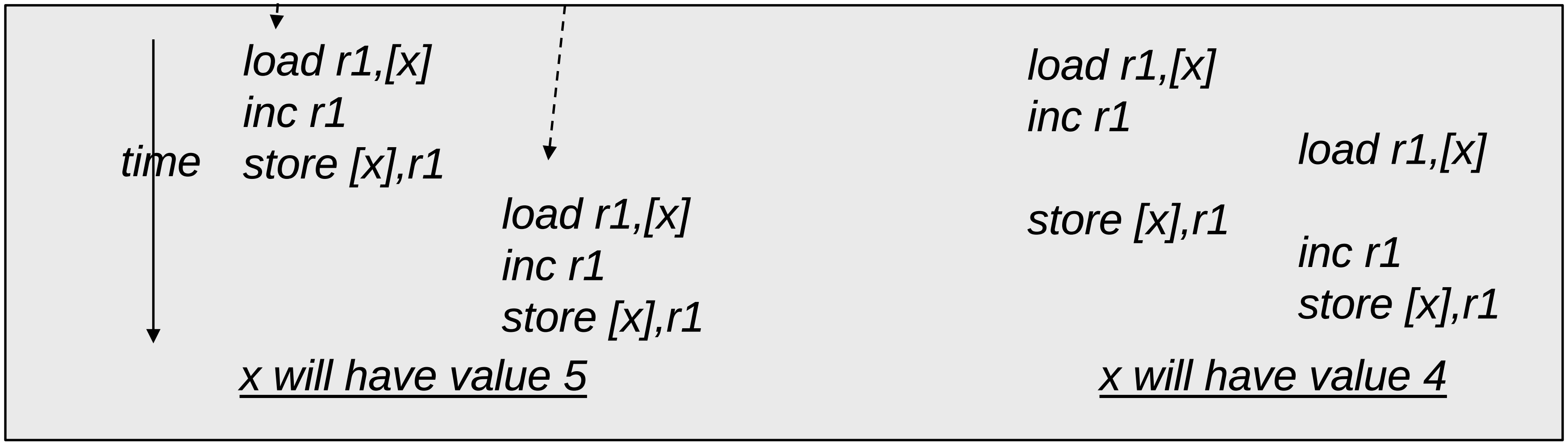
Atomic instructions

- Example: sum all the elements of an array
  - core 0 adds up first half, core 1 adds up second half
  - each core adds its contribution to variable sum
- Problem: unless cores are synchronized, you get a *data-race*
    - result of read/modify/write may not be what you expect
    - final value can depend on how code is compiled and on scheduling of instructions from threads
- General problem:
  - read/modify/write must be performed atomically on a collection of variables or data structure elements

Data-race illustration



- Final value can be 4 or 5 depending on scheduling of instructions



```
time    load r1,[x]
        inc r1
        store [x],r1
                        load r1,[x]
                        inc r1
                        store [x],r1
        x will have value 5
```

```
load r1,[x]
inc r1
                load r1,[x]
store [x],r1
                inc r1
                store [x],r1
x will have value 4
```

Solution

- Architecture provides atomic instructions
  - small collection of read/modify/write instructions operating on ints, doubles, etc.
  - execute as though all other threads were suspended during execution of atomic instruction
  - examples:
    - swap(addr, reg)
      - swap value in memory at address addr with value in register reg
    - atomic add(reg,addr)
- Easy to modify MESI protocol to implement atomic instructions
  - like write but line is pinned in cache until instruction completes
  - no other core can steal line until instruction completes

Limitations of atomic instructions

- Atomic instructions give you atomicity for read/modify/write on data types like ints, floats, doubles (fit in cache line)
- Do not solve atomicity problem for updates to large amounts of data like arrays or structs
- Hardware solution: transactional memory
  - jury is still out about whether this is useful
- Software solution: locks
  - pThreads library: mutex-locks and spin-locks
  - implementation of locks uses atomic instructions

pThreads library: low-level shared-memory programming

Threads

- Software analog of cores
  - Each thread has its own PC, SP, registers, and stack
  - All threads share heap and globals
- Runtime system handles mapping of threads to cores
  - if there are more threads than cores, runtime system will time-slice threads on cores
  - HPC applications: usually #threads = #cores
    - portability: number of threads is usually a runtime parameter
- Threads have two kinds of names
  - pThread name: opaque handle used by pThreads library (like social security number for people)
  - short name: usually an integer 0,1,2…(like first names for people) and used in application program to tell threads what to do or where to write their results

Thread Basics: Creation and Termination

- Program begins execution with main thread
- Creating threads:

```
int pthread_create (
    pthread_t *thread_handle,
    const pthread_attr_t *attribute,
    void * (*thread_function)(void *),
    void *arg);
```

- Type (void *) is C notation for “raw address” (can point to anything)
- Thread is created and starts to execute thread_function with parameter arg, which specifies short name and other data to be passed to thread
- Thread handle: opaque handle for thread

Terminating threads

- Thread terminated when:
    - o it returns from its starting routine, or
    - o it makes a call to pthread_exit()
- Main thread
    - – exits with pthread_exit(): other threads will continue to execute
    - – otherwise other threads automatically terminated
- Cleanup:
    - – pthread_exit() routine does not close files
    - – any files opened inside the thread will remain open after the thread is terminated.

Example

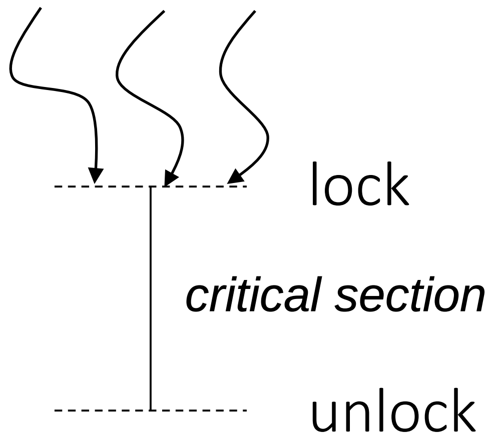
```
#include <pthread.h>
 #include <stdio.h>
 #include <stdlib.h>
 #define NUM_THREADS 5
 int threadArg[NUM_THREADS];//parameters for threads
 pthread_t handles[NUM_THREADS]; //store opaque handles for threads

 void *PrintHello(void *threadIdPtr) {
    int shortId = * (int *)threadIdPtr;
    printf("\n%d: Hello World!\n", shortId);
    pthread_exit(NULL);
  }

 int main(int argc, char *argv[]) {
    for(int t=0;t<NUM_THREADS;t++){
       printf("Creating thread %d\n", t);
       threadArg[t] = t;
       pthread_create(&handles[t], NULL, PrintHello, &threadArg[t]);
    }
    pthread_exit(NULL);
 }
```

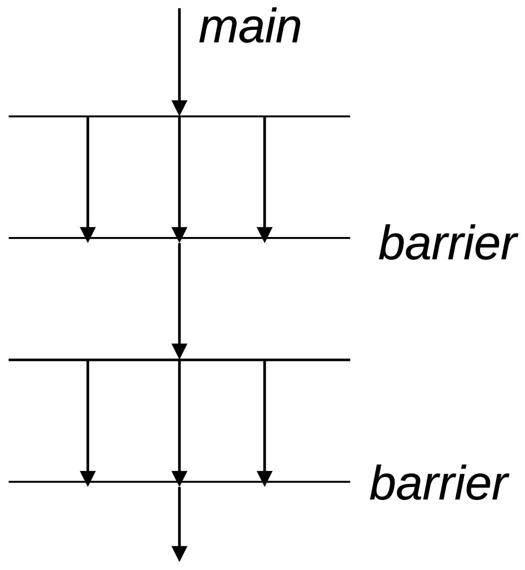
Output

```
Creating thread 0
Creating thread 1

0: Hello World!

1: Hello World!
Creating thread 2
Creating thread 3

2: Hello World!

3: Hello World!
Creating thread 4

4: Hello World!
```

Synchronization

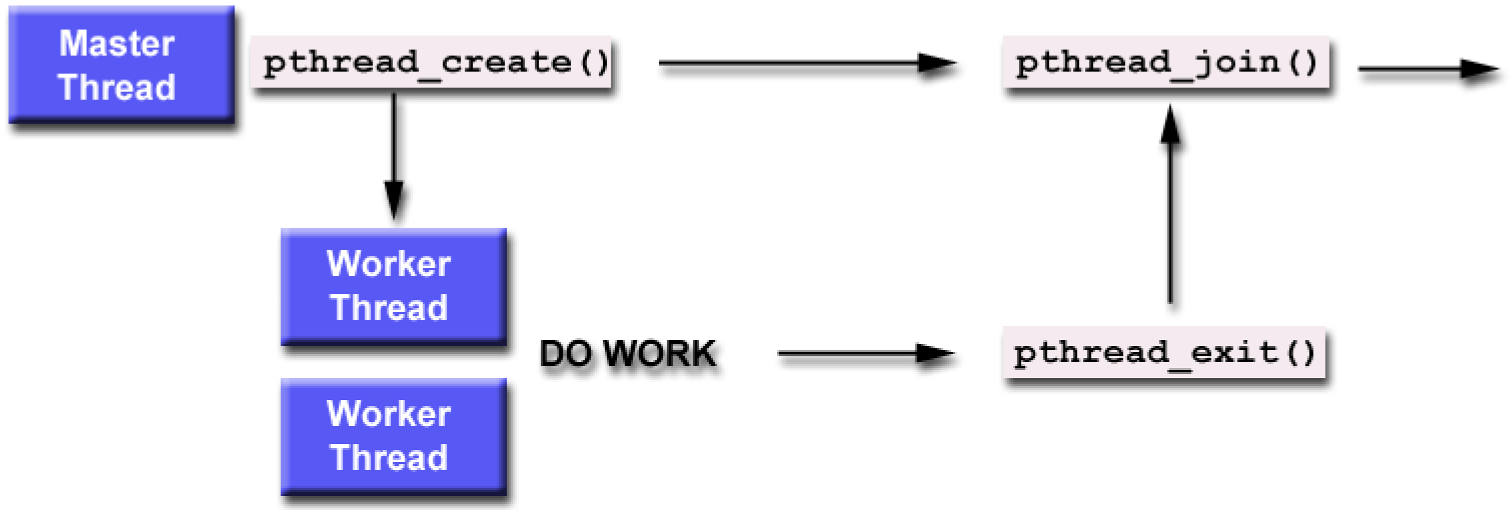
- Join:
  - block thread until some other thread terminates
- Lock:
  - used to ensure mutual exclusion: only one thread at a time can
    - access some data
    - execute some piece of code (critical section)
  - two kinds: mutexes and spin-locks
- Barrier:
  - all threads must reach barrier before any thread can move ahead





Join

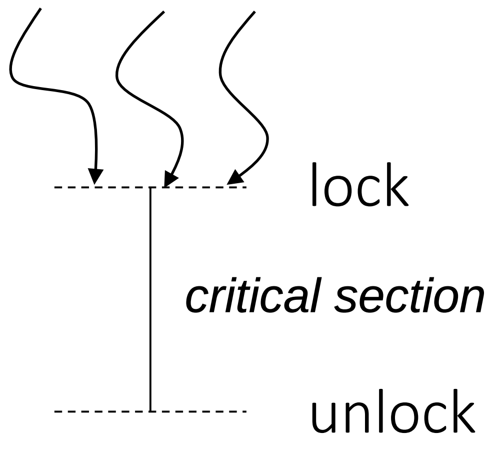
pthread_join (threadid,status)



- The pthread_join() function blocks the calling thread until the specified thread terminates.
- The programmer can obtain the target thread's termination return status if it was specified in the target thread's call to pthread_exit().

Critical section in code

- Portion of code that should be executed by only thread at a time
- Implementation: bracket critical section with lock/unlock
- Can be used to implement atomic updates to anything
- Coarse-grain locking
  - not the right solution for parallelism but it is a start



Mutex-locks

- Lock is implemented by
  - variable with two states: *available* or *not_available*
  - queue that can hold ids of threads waiting for the lock
- Lock acquire:
  - If lock is *available,* it is changed to *not_available,* and control returns to application program
  - If lock is *not_available*, thread is queued up at the lock, and control returns to application program only when lock is acquired by that thread
  - Key invariant: once a thread tries to acquire lock, control returns to thread only after lock has been awarded to that thread
- Lock release:
  - next thread in queue is informed it has acquired lock
- Fairness: thread that wants lock gets it even if other threads want to acquire lock unbounded number of times

Pthreads API

- Type
```
        pthread_mutex_t
```
- Lock initialization
```
        int pthread_mutex_init(
                pthread_mutex_t *mutex_lock,
                const pthread_mutexattr_t *lock_attr);
```
- Acquiring lock
```
    int pthread_mutex_lock(
                 pthread_mutex_t *mutex_lock);
```
- Releasing lock
```
        int pthread_mutex_unlock (
                 pthread_mutex_t *mutex_lock);
```

Spin-locks/trylocks

- Another kind of lock: spin-lock, trylock
- Lock acquire is different from mutex: if lock is available, acquire it; otherwise return a "busy" error code (EBUSY)

```
int pthread_mutex_trylock(
        pthread_mutex_t *mutex_lock);
```

- Faster than `pthread_mutex_lock` on typical systems when there is no contention since it does not have to deal with queues associated with locks

Implementing locks using swap

- Recall: swap(addr,reg)
  - swap contents of address and register atomically
- Spin-lock using swap (test-and-set spin-lock)
  - variable L has 0/1 for unlocked/locked
  - lock code:

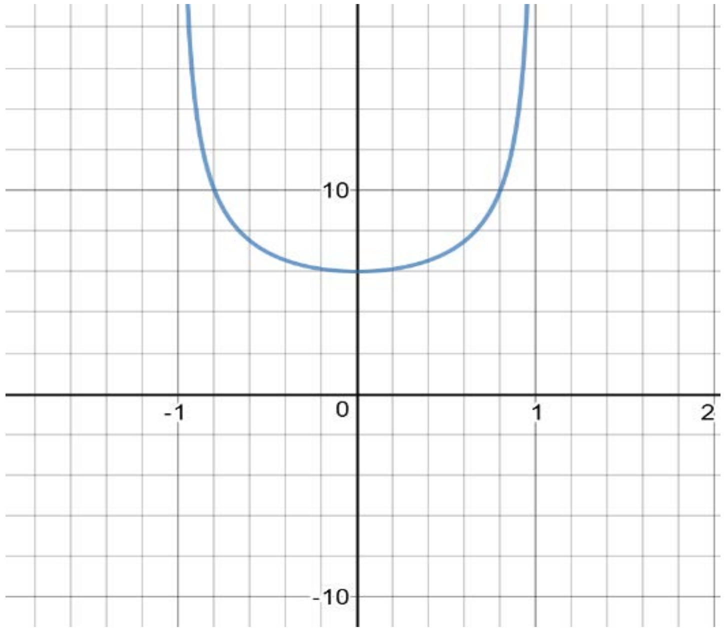
```
rx ← 1;
swap(L,rx);
return rx; //if returned value = 0 you have lock else not
```

  - unlock

```
L ← 0;
```

- More efficient implementation
  - test-and-test-set spin-lock

Application: numerical integration



$$f(x) = \frac{6}{\sqrt{1-x^2}}$$

- Estimate value of π using numerical integration $\int_0^{1/2} f(x)dx = \pi$
- Divide interval [0,1/2) into steps of equal size h and compute

$$\sum_{i=0}^{\frac{1}{2h}-1} f(i*h)*h$$

Abstraction

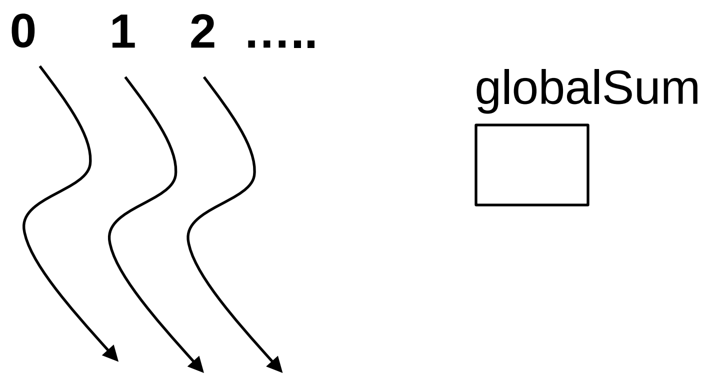
$$sum = \Sigma_{i=1}^{n} f(i)$$

- Parallelism:
  - map: function evaluations *f(i)* can be done in parallel
  - reduce: if addition is associative, *f(i)* values can be summed in parallel in O(log(n)) steps
    - we will not worry about exploiting this parallelism
- We will write several pThreads programs to illustrate the concepts we have studied

Solution (I)

- Distribution of work
  - round-robin with p threads
  - thread t computes values for i = t,t+p,t+2p..
- Single global variable globalSum
- Whenever thread computes a value, it adds it to global variable
- Preventing data races
  - use a mutex-lock

$$\sum_{i=0}^{\frac{1}{2h}-1} f(i*h)*h$$



Code

```
#include <pthread.h>
#include <stdlib.h>
#include <math.h>
#include <stdio.h>

#define MAX_THREADS 512

pthread_t handles[MAX_THREADS];
int threadArg[MAX_THREADS];
double globalSum = 0.0;
pthread_mutex_t globalSum_lock;

void *compute_pi (void *);

int numPoints;
int numThreads;
double step;

double f(double x) {
  return (6.0/sqrt(1-x*x));
}
```

```
int main(int argc, char *argv[]) {

  pthread_attr_t attr;
  pthread_attr_init (&attr);

  numPoints = 100000000;
  step = 0.5/numPoints;
  numThreads = atoi(argv[1]); //number of threads is an input

  //create threads and initialize sum array
  for (int i=0; i< numThreads; i++) {
    threadArg[i] = i;
    pthread_create(& handles[i],&attr,compute_pi,& threadArg[i]);
  }

  //join with threads and add their contributions from sum array
  for (int i=0; i< numThreads; i++) {
    pthread_join(handles[i], NULL);
  }
  printf("%f\n", globalSum);
  return 0;

}
```

```
void *compute_pi (void *threadIdPtr) {
  int myId = *(int *)threadIdPtr;

  for (int i = myId; i < numPoints; i+=numThreads) {
     double x = step * ((double) i);  // next x
     double value = step*f(x);
     pthread_mutex_lock(&globalSum_lock);
         globalSum = globalSum + value;  // Add to globalSum
     pthread_mutex_unlock(&globalSum_lock);
   }
```

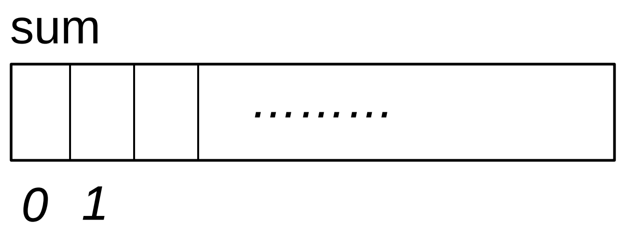
Performance

- Computation of each value added to globalSum takes little time
  - lock/add/unlock will be serial bottleneck
- We can replace critical section by atomic add
  - but atomic adds must be done serially, so serial bottleneck is still there
- In both solutions, you will also have a lot of cache line ping-ponging

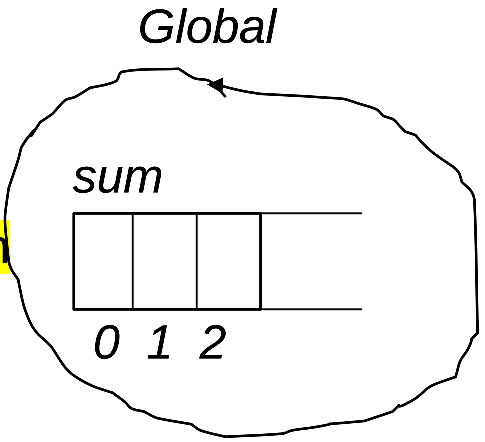
Solution (II)

- To avoid synchronization, create a global array sum
- Thread t
  - adds each value into sum[i] where sum is a global array
- Main thread joins with each worker thread and reads its contribution from sum array
- Main thread prints answer after joining with all worker threads

$$\sum_{i=0}^{\frac{1}{2h}-1} f(i*h)*h$$



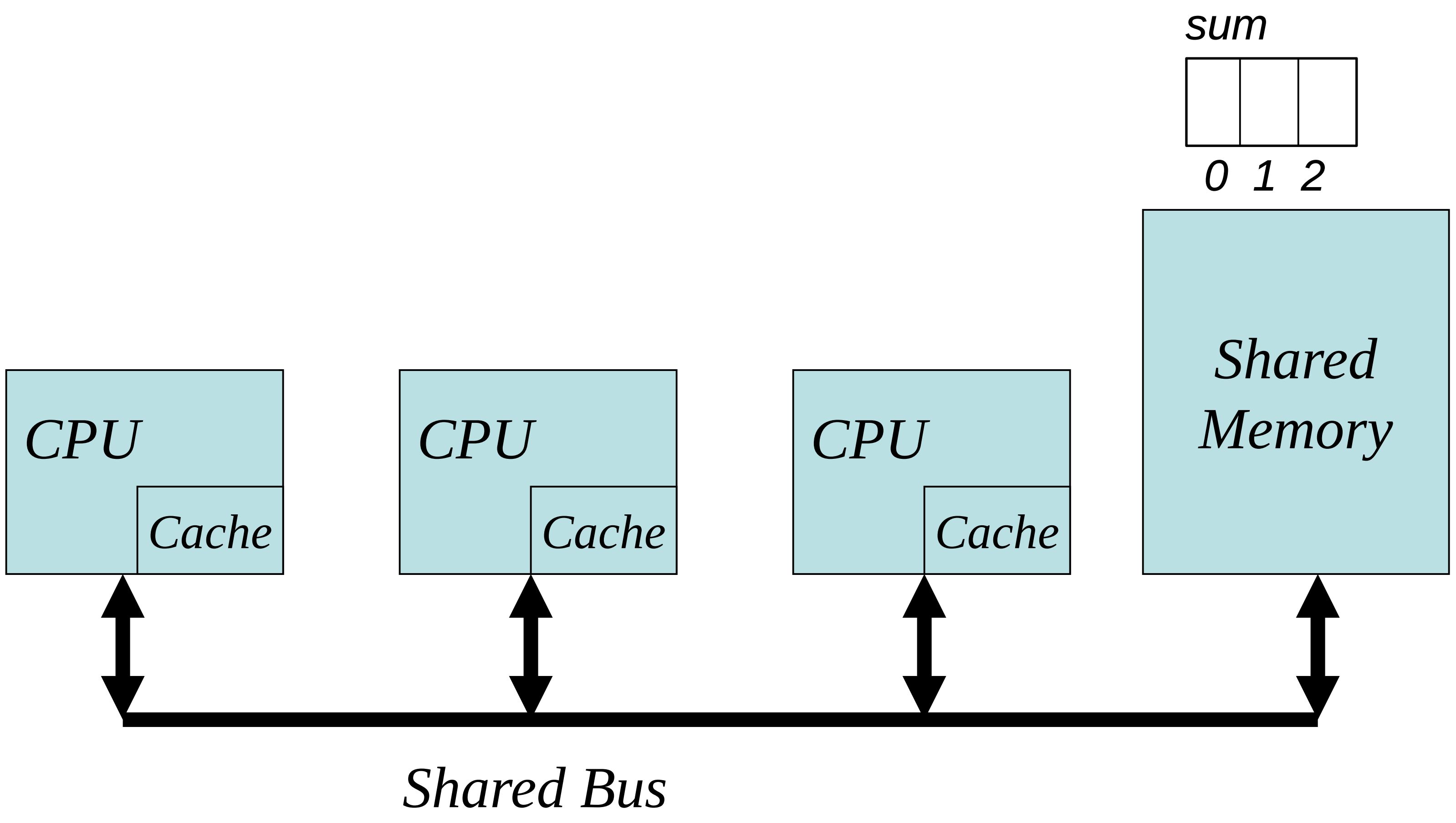
```
void *compute_pi (void *threadIdPtr) {
  int myId = *(int *)threadIdPtr;
 for (int i = myId; i < numPoints; i+=numThreads) {
     double x = step * ((double) i);  // next x
     sum[myId] = sum[myId] + step*f(x);  // Add to local sum
  }
}
```



Code for main thread must add up values in sum array.

```
………
 for (int i=0; i< numThreads; i++) {
    pthread_join(handles[i], NULL);
    pi += sum[i];
  }
……..
```

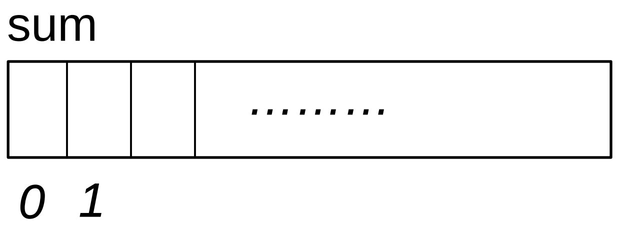
Problem: false-sharing



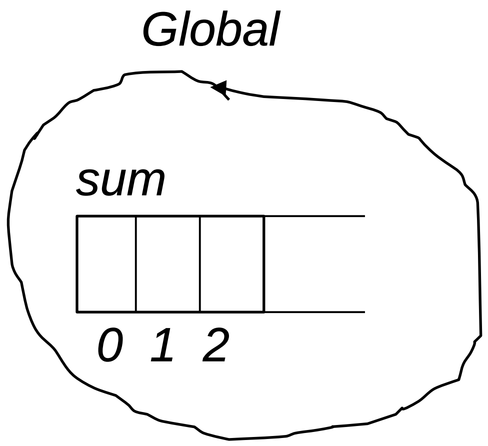
Solution (III)

- Thread t
  - computes values for i = t, t+P,t+2P,…
  - adds each value into a local variable of thread
  - when it is done, it writes the final value into sum[i]
- Main thread joins with each worker thread and reads its contribution from sum array
- Main thread prints answer after joining with all worker threads

$$\sum_{i=0}^{\frac{1}{2h}-1} f(i*h)*h$$



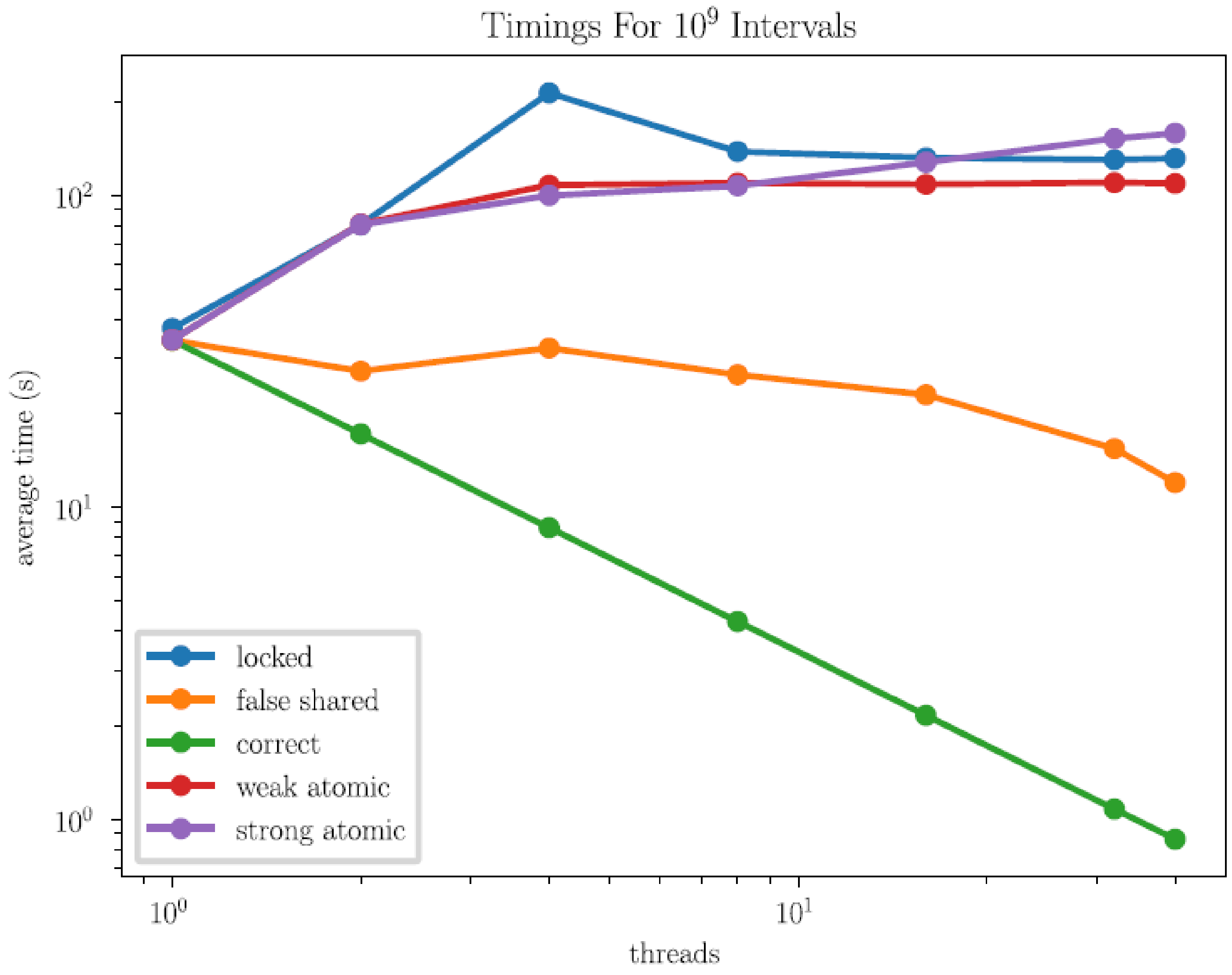
```
void *compute_pi (void *threadIdPtr) {
  int myId = *(int *)threadIdPtr;

 double mySum =0.0;
 for (int i = myId; i < numPoints; i+=numThreads) {
     double x = step * ((double) i);  // next x
     mySum = mySum + step*f(x);  // Add to local sum
  }
  sum[myId] = mySum; //write to global sum array
}
```



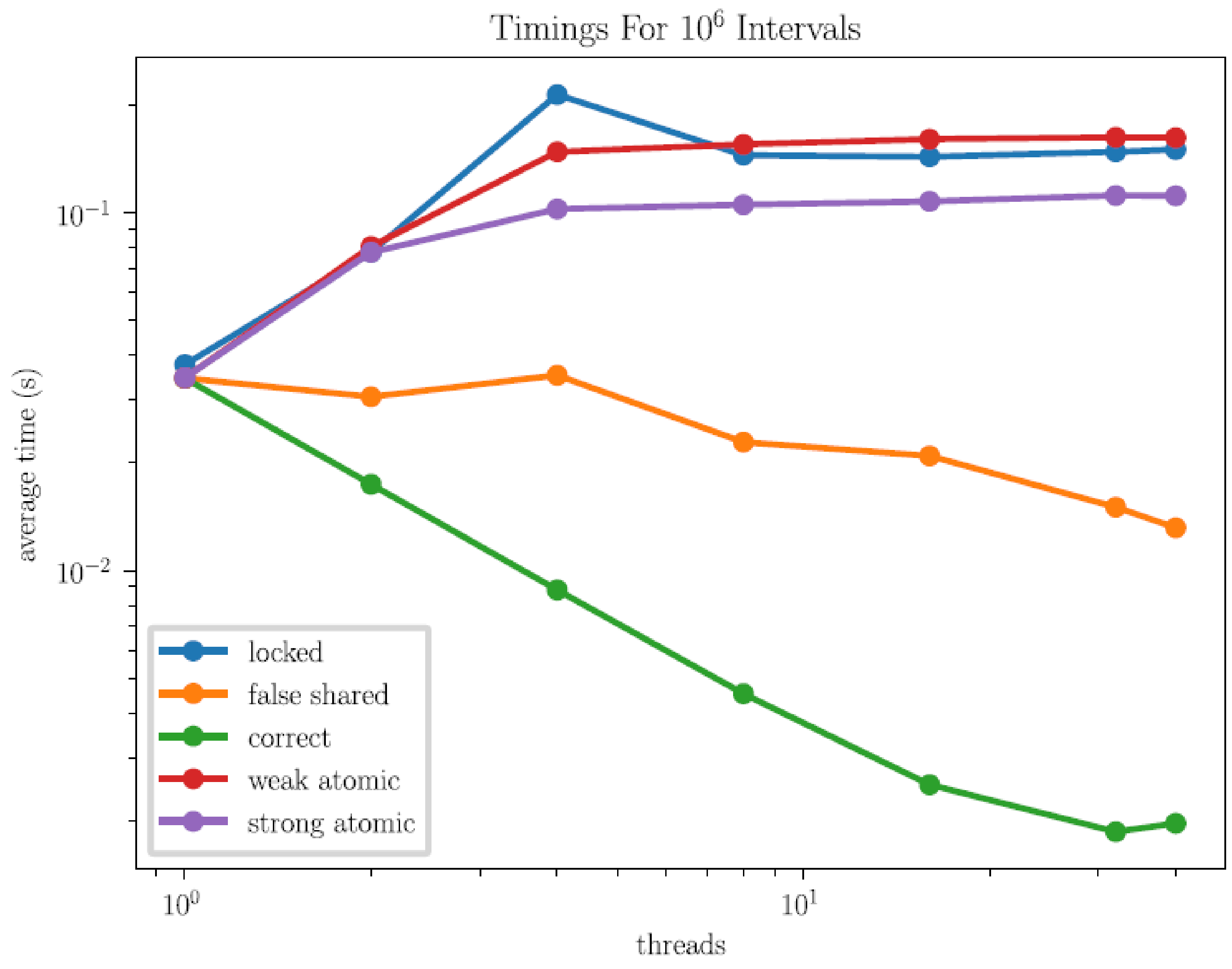
Numerical Integration Versions

- We saw three versions of program to compute pi
  - Version 1: summation in global variable
  - Version 2: summation in sum array
  - Version 3: local summation + update sum array
- Which version will perform best?
  - Version 1: true-sharing leads to many coherence misses + serialization in global variable updates
  - Version 2: false-sharing leads to many coherence misses

Performance



Performance



Summary

- Architecture
  - cache coherence
  - atomic instructions
  - memory consistency model
- The POSIX Thread API
  - creating and destroying threads
  - synchronization
    - join
    - mutual exclusion: locks and spin-locks
    - intrinsics for atomic instructions
    - barrier
- Performance:
  - minimize false and true sharing
  - keep critical sections small

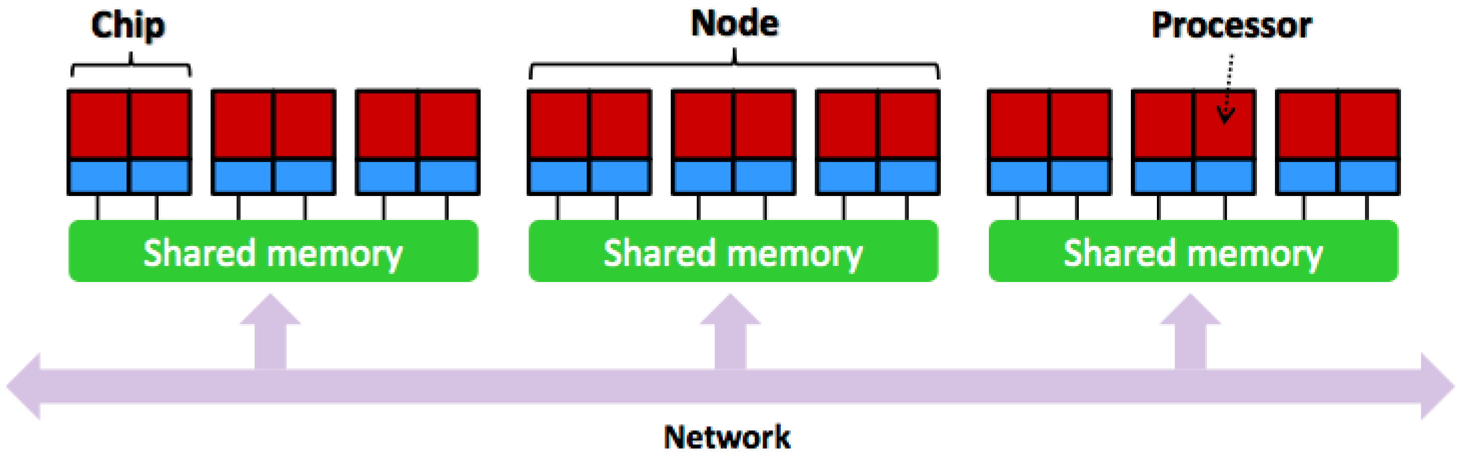
Distributed-memory programming

Clusters and data-centers

TACC Stampede 2 cluster

- 4,200 Intel Knights Landing nodes, each with 68 cores
- 1,736 Intel Xeon Skylake nodes, each with 48 cores
- 100 Gb/sec Intel Omni-Path network with a fat tree topology employing six core switches

Cartoon picture of cluster



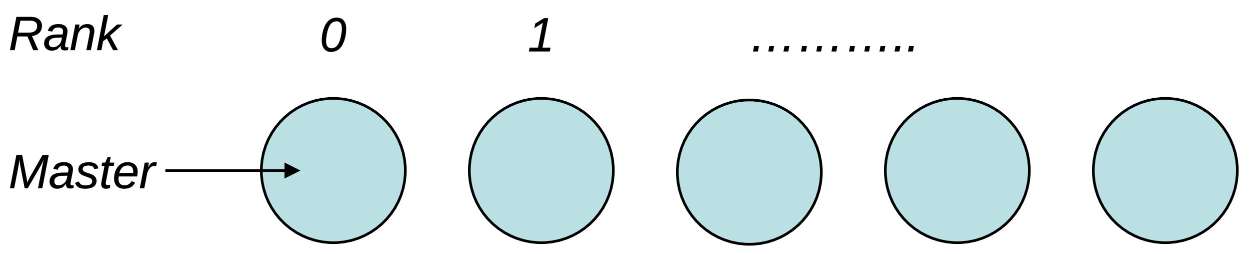
Typical latency numbers

From: *Latency numbers every HPC programmer should know*

L1 cache reference/hit	1.5 ns	4 cycles
Floating-point add/mult/FMA operation	1.5 ns	4 cycles
L2 cache reference/hit	5 ns	12 ~ 17 cycles
L3 cache hit	16-40 ns	40-300 cycles
256MB main memory reference "Broadwell" E5-2690v4	75-120 ns	TinyMemBench on
Send 4KB message between hosts	1-10 µs	MPICH on 10-100Gbps
Read 1MB sequentially from disk ~200MB/sec hard disk (seek time would be additional latency)	5,000,000 ns	5 ms
Random Disk Access (seek+rotation)	10,000,000 ns	10 ms
Send packet CA->Netherlands->CA	150,000,000 ns	150 ms

Locality is important.

Basic MPI constructs



Flat name space of processes
Rank:process ID

- MPI_COMM_SIZE
  - how many processes are there in the world?
- MPI_COMM_RANK
  - what is my logical number (rank) in this world?
- MPI_SEND (var, receiverRank)
  - specify data to be sent in message and who will receive it
  - data can be an entire array (with stride)
- MPI_RECV (var, senderRank)
  - whom to receive from and where to store received data

Programming Model

- Single Program Multiple Data (SPMD) model
- Program is executed by all processes
- Use conditionals to specify that only some processes should execute a statement
  - to execute only on master:

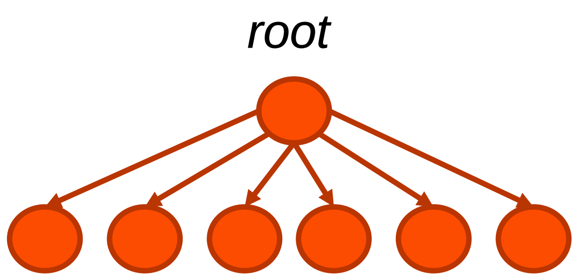
```
      if (rank == 0) then ...;
```

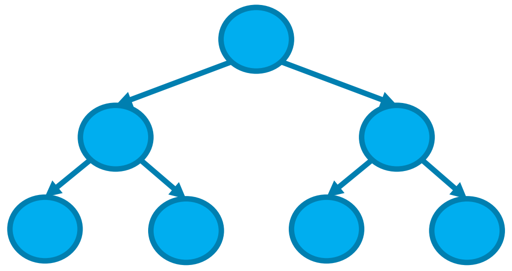
Hello World

```
/*The Parallel Hello World Program*/
#include <stdio.h>
#include <mpi.h>

main(int argc, char **argv)
{
   int myRank;

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

   printf("Hello World from Node %d\n", myRank);

   MPI_Finalize();
}
```

Collective communication

- Broadcast
  - some process (usually root) wants to send value to all other processes
- One solution:
  - use a loop with MPI_SEND
  - O(P) time but P is very big in clusters
- Better solution:
  - tree of processes
  - O(log(P)) time
- MPI_BCAST(var, rootRank)
- Similar collective for reductions
  - MPI_Reduce(var,result,MPI_SUM,rootRank)
  - result: variable on process rootRank that will contain the final result
  - var: contribution from this process
  - MPI_SUM: reduction operation is addition





Example: Pi in C

```
#include <mpi.h>
#include <math.h>
int main(int argc, char *argv[]) {
        double mypi = 0.0;
        [...snip...]

        MPI_Bcast(&num_segs, 1, MPI_INT, 0, MPI_COMM_WORLD);

        double width = 1.0 / (double) num_segs;
        for (int i = rank + 1; i <= n; i += size)
                mypi += width * sqrt(1 – (((double) i / num_segs) * ((double) i / num_segs));

        MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

        if (rank == 0)
                printf("pi is approximately %.16f, Error is %.16f\n",
                         4 * pi, fabs((4 * pi) - PI25DT));
        [...snip...]
}
```

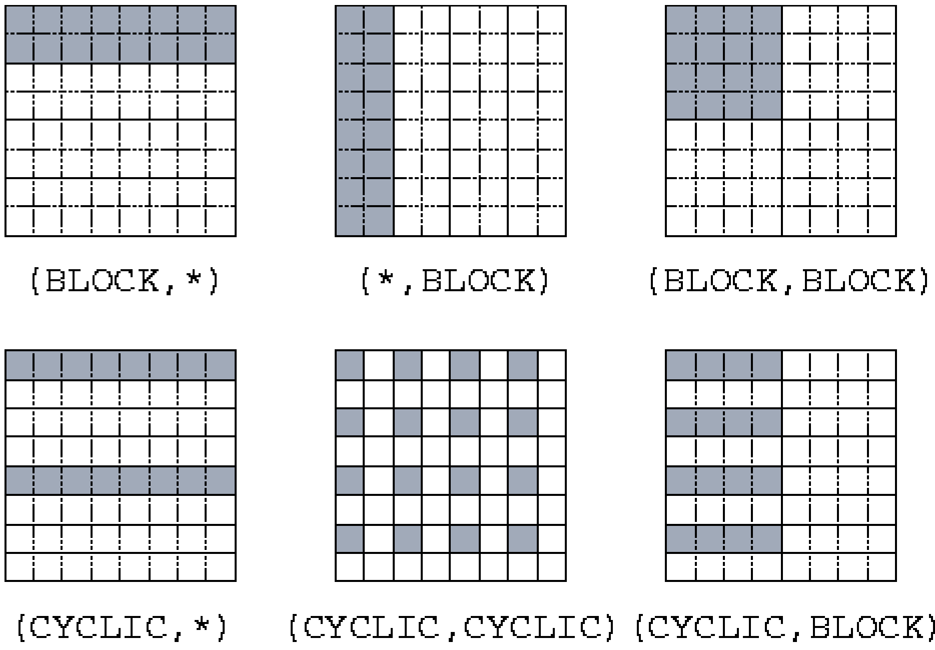
Tell all processes how many rectangles there are

Calculate my share of pi

Send the result to rank 0 **and** calculate the total at the same time

Data structures

- Since there is no global memory, data structures have to partitioned between processes
- No MPI support: entirely under the control of application program
- Common partitioning strategies for dense arrays
  - block row, block column, 2D blocks, etc.



Summary

- Low-level shared-memory and distributed-memory programming in pThreads and MPI can be very tedious
- Higher-level abstractions are essential for productivity
- Major problems
  - efficient implementation
  - performance modeling: changing a few lines in the code can change performance dramatically
- Lots of work left for Stephanies