计算机组成与系统结构 Computer Organization & System Architecture

Huang Kejie (黄科杰) 百人计划研究员

Office: 玉泉校区老生仪楼 304

Email address: huangkejie@zju.edu.cn

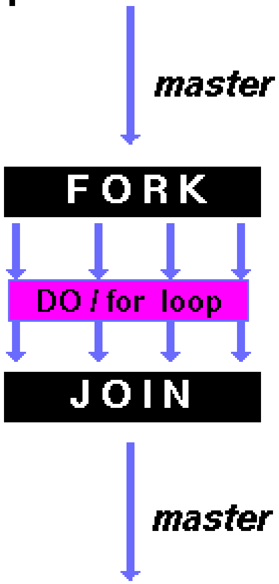
HP: 17706443800

OpenMP

- C extension: no new language to learn
- Multi-threaded, shared-memory parallelism
  - Compiler Directives, **`#pragma`**
  - Runtime Library Routines, **`#include <omp.h>`**
- **`#pragma`**
  - Ignored by compilers unaware of OpenMP
  - Same source for multiple architectures
    - E.g., same program for 1 & 16 cores
- Only works with shared memory

OpenMP Parallel `for` *pragma*

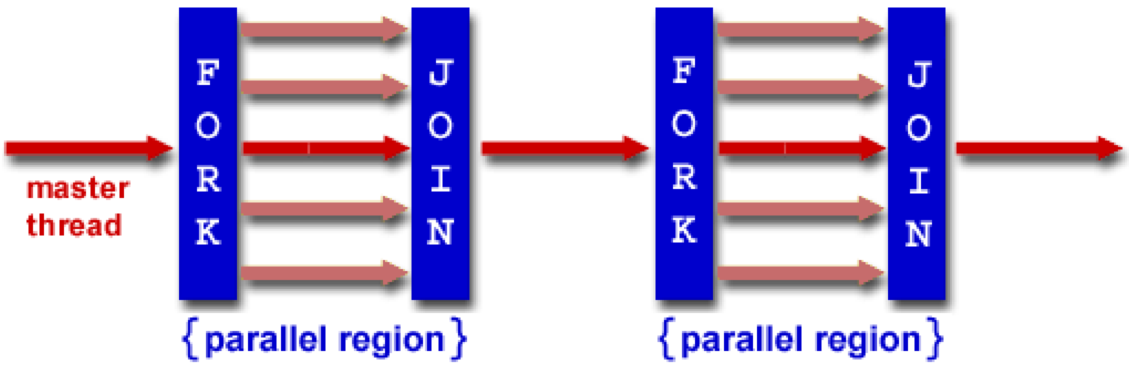
```
#pragma omp parallel for
  for (i=0; i<max; i++) zero[i] = 0;
```



- Master thread creates additional threads, each with a separate execution context
- All variables declared outside for loop are shared by default, except for loop index which is implicitly *private* per thread
- Implicit “barrier” synchronization at end of for loop
- Divide index regions sequentially per thread
  - Thread 0 gets 0, 1, …, (max/n)-1;
  - Thread 1 gets max/n, max/n+1, …, 2*(max/n)-1
  - Why?

OpenMP Programming Model

- Fork - Join Model:



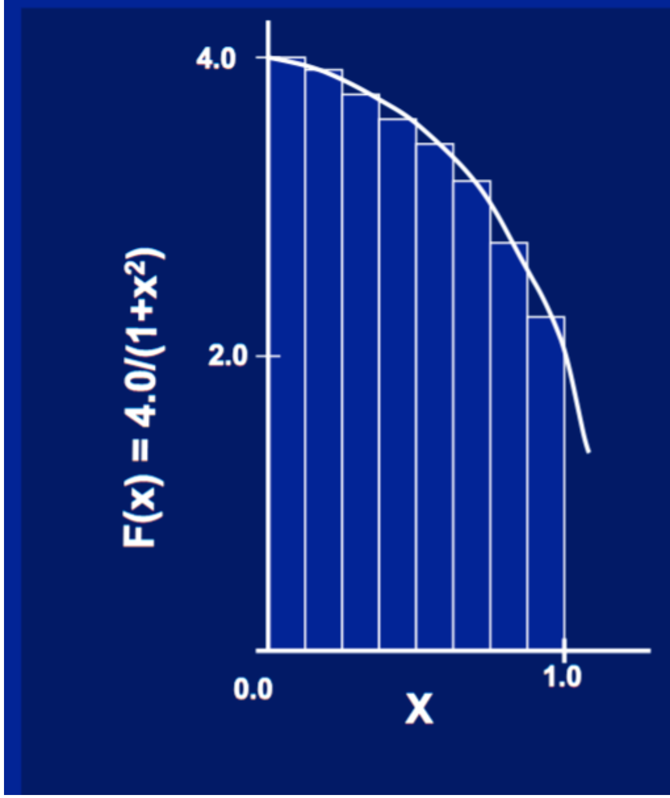
- OpenMP programs begin as single process (master thread)
  – Sequential execution
- When parallel region is encountered
  – Master thread “forks” into team of parallel threads
  – Executed simultaneously
  – At end of parallel region, parallel threads ”join”, leaving only master thread
- Process repeats for each parallel region
  – Amdahl’s Law?

What Kind of Threads?

- OpenMP threads are operating system (software) threads
- OS will multiplex requested OpenMP threads onto available hardware threads
- Hopefully each gets a real hardware thread to run on, so no OS-level time-multiplexing
- But other tasks on machine compete for hardware threads!

Example 2: Computing π

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

http://openmp.org/mp-documents/omp-hands-on-SC08.pdf

Sequential π

```c
#include <stdio.h>

void main () {
    const long num_steps = 10;
    double step = 1.0/((double)num_steps);
    double sum = 0.0;
    for (int i=0; i<num_steps; i++) {
        double x = (i+0.5) *step;
        sum += 4.0*step/(1.0+x*x);
    }
    printf ("pi = %6.12f\n", sum);
}
```

```
pi = 3.142425985001
```

- Resembles π, but not very accurate
- Let's increase **`num_steps`** and parallelize

Parallelize (1) …

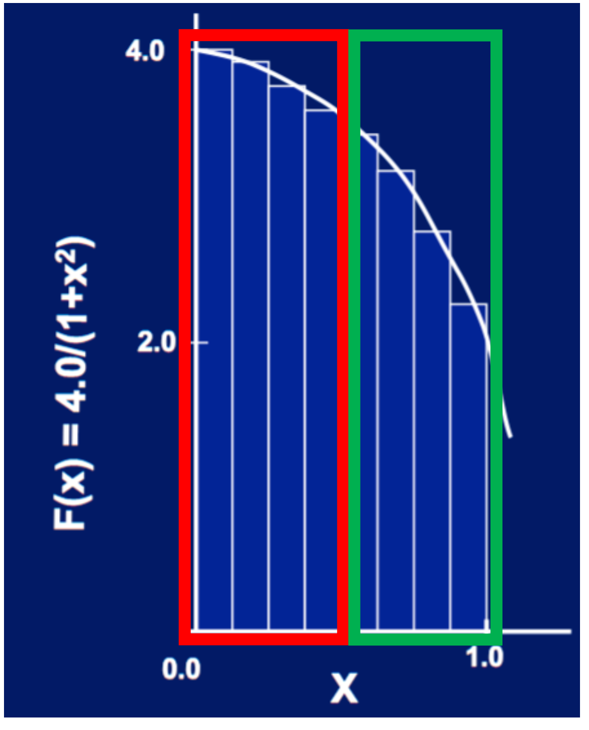
```c
#include <stdio.h>

void main () {
    const long num_steps = 10;
    double step = 1.0/((double)num_steps);
    double sum = 0.0;
#pragma parallel for
    for (int i=0; i<num_steps; i++) {
        double x = (i+0.5) *step;
        sum += 4.0*step/(1.0+x*x);
    }
    printf ("pi = %6.12f\n", sum);
}
```

- Problem: each threads needs access to the shared variable sum

- Code runs sequentially …

Parallelize (2) …



sum[0] sum[1]

- Compute
  - `sum[0]` and `sum[1]`
- in parallel

- Compute
  - `sum = sum[0] + sum[1]`
- sequentially

Parallel π

```c
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 4;
    const long num_steps = 10;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
            printf("i =%3d,  id =%3d\n", i, id);
        }
    }
    double pi = 0;
    for (int i=0; i<NUM_THREADS; i++) pi += sum[i];
    printf ("pi = %6.12f\n", pi);
}
```

Trial Run

```
i =  1,  id =  1
i =  0,  id =  0
i =  2,  id =  2
i =  3,  id =  3
i =  5,  id =  1
i =  4,  id =  0
i =  6,  id =  2
i =  7,  id =  3
i =  9,  id =  1
i =  8,  id =  0
pi = 3.142425985001
```

Parallel π

```c
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 4;
    const long num_steps = 1000000;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
            // printf("i =%3d,  id =%3d\n", i, id);
        }
    }
    double pi = 0;
    for (int i=0; i<NUM_THREADS; i++) pi += sum[i];
    printf ("pi = %6.12f\n", pi);
}
```

Scale up: num_steps = 10^6

pi = 3.141592653590

You verify how many digits are correct …

Can We Parallelize Computing **sum**?

```
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 1000;
    const long num_steps = 100000;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    double pi = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
        }
        pi += sum[id];
    }
    printf ("pi = %6.12f\n", pi);
}
```

Always looking for ways to beat Amdahl's Law …

Summation inside parallel section

- Insignificant speedup in this example, but …
- **pi = 3.138450662641**
- Wrong! And value changes between runs?!
- What's going on?

What Kind of Threads?

- What are the possible values of `*(x1)` after executing this code by two concurrent threads?

```
# *(x1) = 100
lw   x2,0(x1)
addi x2,x2,1
sw   x2,0(x1)
```

Answer	`*(x1)`
RED	100 or 101
GREEN	101
ORANGE	101 or 102
YELLOW	100 or 101 or 102

What's Going On?

```c
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 1000;
    const long num_steps = 100000;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    double pi = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
        }
        pi += sum[id];
    }
    printf ("pi = %6.12f\n", pi);
}
```

- Operation is really
  `pi = pi + sum[id]`
- What if >1 threads reads current (same) value of **`pi`**, computes the sum, stores the result back to **`pi`**?
- Each processor reads same intermediate value of **`pi`**!
- Result depends on who gets there when
  - A "race" → result is **<u>not deterministic</u>**

OpenMP Reduction

```
double avg, sum=0.0, A[MAX]; int i;
#pragma omp parallel for private ( sum )
for (i = 0; i <= MAX ; i++)
    sum += A[i];
avg = sum/MAX;  // bug
```

- *Problem is that we really want sum over all threads!*
- Reduction: specifies that 1 or more variables that are private to each thread are subject of reduction operation at end of parallel region: **reduction(operation:var)** where
  - Operation: operator to perform on the variables (var) at the end of the parallel region
  - Var: One or more variables on which to perform scalar reduction.

```
double avg, sum=0.0, A[MAX]; int i;
#pragma omp for reduction(+ : sum)
for (i = 0; i <= MAX ; i++)
    sum += A[i];
avg = sum/MAX;
```

Calculating π Original Version

```
#include <omp.h>
#define NUM_THREADS 4
static long num_steps = 100000; double step;

void main () {
  int i;     double  x, pi, sum[NUM_THREADS];
  step = 1.0/(double) num_steps;
  #pragma omp parallel private ( i, x )
  {
    int id = omp_get_thread_num();
    for (i=id, sum[id]=0.0; i<num_steps; i=i+NUM_THREADS)
    {
      x = (i+0.5)*step;
      sum[id] += 4.0/(1.0+x*x);
    }
  }
  for(i=1; i<NUM_THREADS; i++)
    sum[0] += sum[i];  pi = sum[0];
  printf ("pi = %6.12f\n", pi);
}
```

Version 2: parallel for, reduction

```
#include <omp.h>
#include <stdio.h>
/static long num_steps = 100000;
double step;
void main ()
{     int i;     double x, pi, sum = 0.0;
      step = 1.0/(double) num_steps;
#pragma omp parallel for private(x) reduction(+:sum)
      for (i=1; i<= num_steps; i++){
            x = (i-0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
      }
      pi = sum;
   printf ("pi = %6.8f\n", pi);
}
```

Synchronization

- Problem:
  - Limit access to shared resource to 1 actor at a time
  - E.g. only 1 person permitted to edit a file at a time
    - otherwise changes by several people get all mixed up

- Solution:

- Take turns:
  - Only one person get's the microphone & talks at a time
  - Also good practice for classrooms, btw …

Data Races and Synchronization

- Two memory accesses form a *data race* if from different threads access same location, at least one is a write, and they occur one after another

- If there is a data race, result of program varies depending on chance (which thread first?)

- Avoid data races by synchronizing writing and reading to get *deterministic* behavior

- Synchronization done by user-level routines that rely on hardware synchronization instructions

Locks

- Computers use locks to control access to shared resources
  - Serves purpose of microphone in example
  - Also referred to as “semaphore”

- Usually implemented with a variable
  - `int lock;`
    - 0 for unlocked
    - 1 for locked

Synchronization with Locks

```
// wait for lock released
while (lock != 0) ;
// lock == 0 now (unlocked)

// set lock
lock = 1;

     // access shared resource ...
     // e.g. pi
     // sequential execution! (Amdahl ...)

// release lock
lock = 0;
```

Lock Synchronization

- Thread 1

```
while (lock != 0)
```

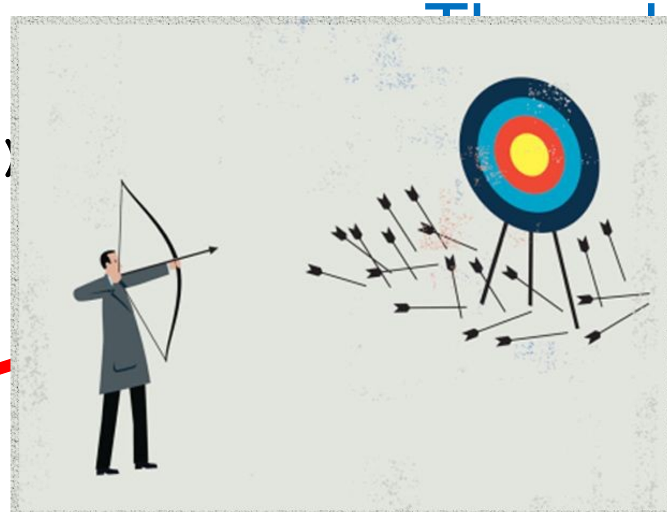
```
lock = 1;
```

```
// critic
```

```
lock = 0;
```

2

```
k != 0) ;
```



not set,
it
they
got and set the lock!

Try as you like, this problem has no solution, not even at the assembly level.

Unless we introduce new instructions, that is!

Hardware Synchronization

- Solution:
  - Atomic read/write
  - Read & write in single instruction
    - No other access permitted between read and write
  - Note:
    - Must use *shared memory* (multiprocessing)

- Common implementations:
  - Atomic swap of register ↔ memory
  - Pair of instructions for “linked” read and write
    - write fails if memory location has been “tampered” with after linked read

- RISCV has variations of both, but for simplicity we will focus on the former

RISCV Atomic Memory Operations (AMOs)

- AMOs atomically perform an operation on an operand in memory and set the destination register to the original memory value
- R-Type Instruction Format: **`Add, And, Or, Swap, Xor, Max, Max Unsigned, Min, Min Unsigned`**

31–27	26	25	24–20	19–15	14–12	11–7	6–0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7
operation	ordering		src	addr	width	dest	AMO

- Load from address in rs1 to "t"
- rd = "t", i.e., the value in memory
- Store at address in rs1 the calculation "t" <operation> rs2
- aq and rl insure in order execution

```
amoadd.w rd,rs2,(rs1):
  t = M[x[rs1]];
  x[rd] = t;
  M[x[rs1]] = t + x[rs2]
```

RISCV Critical Section

- Assume that the lock is in memory location stored in register a0
- The lock is “set” if it is 1; it is “free” if it is 0 (it’s initial value)

```
li t0, 1                    # Get 1 to set lock
Try:  amoswap.w.aq t1, t0, (a0) # t1 gets old lock value
                            # while we set it to 1
    bnez      t1, Try       # if it was already 1, another
                            # thread has the lock,
                        # so we need to try again
      … critical section goes here …
amoswap.w.rl x0, x0, (a0)       # store 0 in lock to release
```

Lock Synchronization

Broken Synchronization

```
while (lock != 0) ;

lock = 1;

// critical section

lock = 0;
```

Fix (lock is at location (a0))

```
li              t0, 1
Try  amoswap.w.aq t1, t0, (a0)
      bnez        t1, Try
Locked:

     # critical section

Unlock:
     amoswap.w.rl x0, x0, (a0)
```

OpenMP Locks

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(void) {
    omp_lock_t lock;
    omp_init_lock(&lock);

#pragma omp parallel
    {
        int id = omp_get_thread_num();

        // parallel section
        // ...

        omp_set_lock(&lock);
        // start sequential section
        // ...
        printf("id = %d\n", id);

        // end sequential section
        omp_unset_lock(&lock);

        // parallel section
        // ...

    }
    omp_destroy_lock(&lock);
}
```

Synchronization in OpenMP

- Typically are used in libraries of higher level parallel programming constructs

- E.g. OpenMP offers `$pragmas` for common cases:
  - critical
  - atomic
  - barrier
  - ordered

- OpenMP offers many more features
  - See online documentation
  - Or tutorial at
    - http://openmp.org/mp-documents/omp-hands-on-SC08.pdf

OpenMP Critical Section

```
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 1000;
    const long num_steps = 100000;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    double pi = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
        }
#pragma omp critical
        pi += sum[id];
    }
    printf ("pi = %6.12f\n", pi);
}
```

The Trouble with Locks …

- … is ***dead-locks***

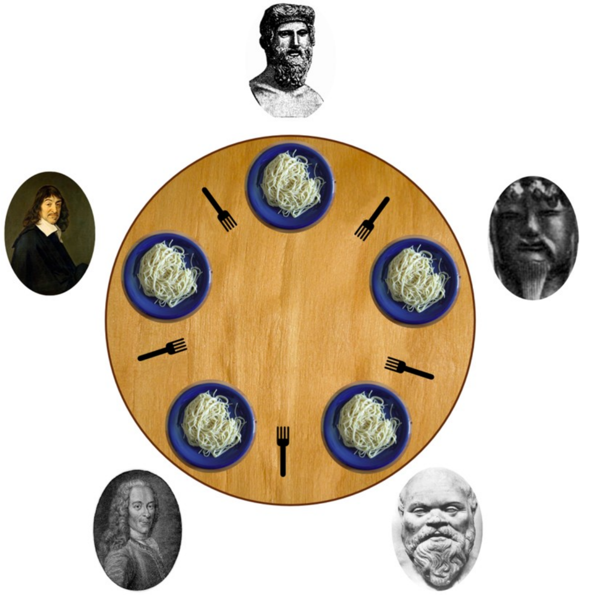
- Consider 2 cooks sharing a kitchen
  - Each cooks a meal that requires salt and pepper (locks)
  - Cook 1 grabs salt
  - Cook 2 grabs pepper
  - Cook 1 notices s/he needs pepper
    - it's not there, so s/he waits
  - Cook 2 realizes s/he needs salt
    - it's not there, so s/he waits

- A not so common cause of cook starvation
  - But deadlocks are possible in parallel programs
  - Very difficult to debug
    - **`malloc/free`** is easy …

Deadlock

- Deadlock: a system state in which no progress is possible

- Dining Philosopher's Problem:
  - Think until the left fork is available; when it is, pick it up
  - Think until the right fork is available; when it is, pick it up
  - When both forks are held, eat for a fixed amount of time
  - Then, put the right fork down
  - Then, put the left fork down
  - Repeat from the beginning



- Solution?

OpenMP Timing

- Elapsed wall clock time:

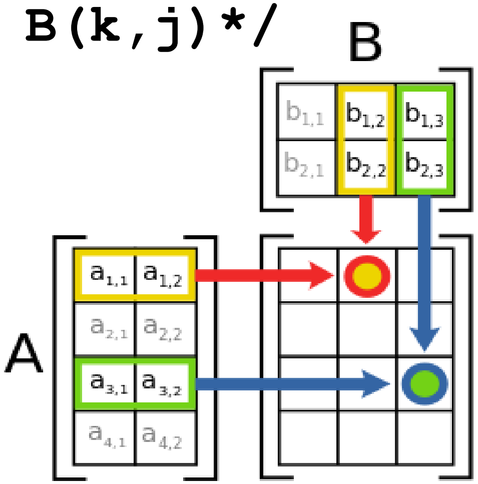
```
double omp_get_wtime(void);
```

  - Returns elapsed wall clock time in seconds
  - Time is measured per thread, no guarantee can be made that two distinct threads measure the same time
  - Time is measured from “some time in the past,” so subtract results of two calls to `omp_get_wtime` to get elapsed time

Matrix Multiply in OpenMP

```
// C[M][N] = A[M][P] × B[P][N]
start_time = omp_get_wtime();
#pragma omp parallel for private(tmp, j, k)
  for (i=0; i<M; i++){
    for (j=0; j<N; j++){
      tmp = 0.0;
      for (k=0; k<P; k++){
        /* C(i,j) = sum(over k) A(i,k) * B(k,j)*/
        tmp += A[i][k] * B[k][j];
      }
      C[i][j] = tmp;
    }
  }
run_time = omp_get_wtime() - start_time;
```

Outer loop spread across N threads; inner loops inside a single thread



Matrix Multiply in Open MP

- More performance optimizations available:
  - Higher *compiler optimization* (-O2, -O3) to reduce number of instructions executed
  - *Cache blocking* to improve memory performance
  - Using SIMD AVX instructions to raise floating point computation rate (DLP)

And, in Conclusion, …

- Sequential software execution speed is limited
- Parallel processing is the only path to higher performance
  - SIMD: instruction level parallelism
    - Implemented in all high performance CPUs today (x86, ARM, …)
    - Partially supported by compilers
  - MIMD: thread level parallelism
    - Multicore processors
    - Supported by Operating Systems (OS)
    - Requires programmer intervention to exploit at single program level
      - E.g. OpenMP
  - SIMD & MIMD for maximum performance
- Synchronization
  - Requires hardware support: specialized assembly instructions
  - Typically use higher-level support
  - Beware of deadlocks

Peer Instruction: Why Multicore?

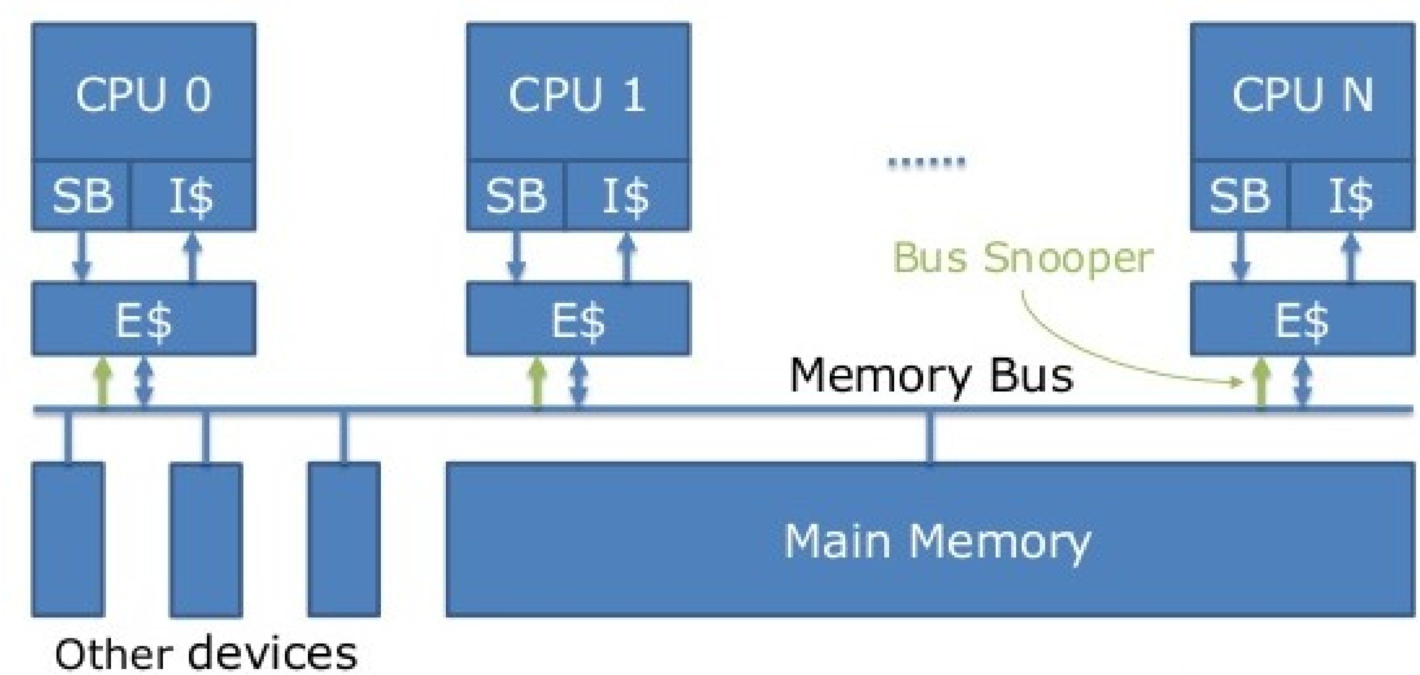
The switch in ~ 2005 from one processor per chip to multiple processors per chip happened because:

I. The “power wall” meant that no longer get speed via higher clock rates and higher power per chip
II. There was no other performance option but replacing one inefficient processor with multiple efficient processors
III. OpenMP was a breakthrough in ~2000 that made parallel programming easy

RED	I only
GREEN	II only
ORANGE	I & II only
YELLOW	I, II, & III only

(Chip) Multicore Multiprocessor

- SMP: (Shared Memory) Symmetric Multiprocessor
  - Two or more identical CPUs/Cores
  - Single shared *coherent* memory

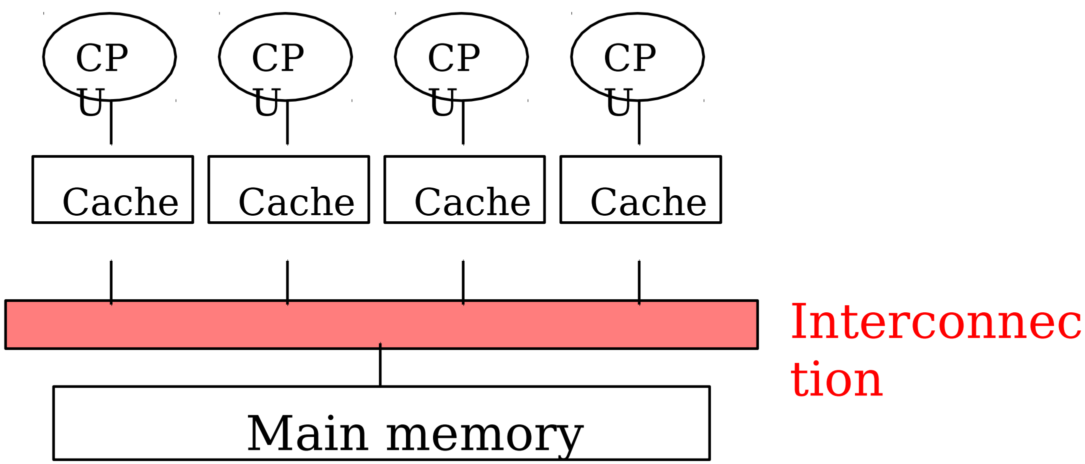


Multiprocessor Key Questions

- Q1 – How do they share data?
- Q2 – How do they coordinate?
- Q3 – How many processors can be supported?

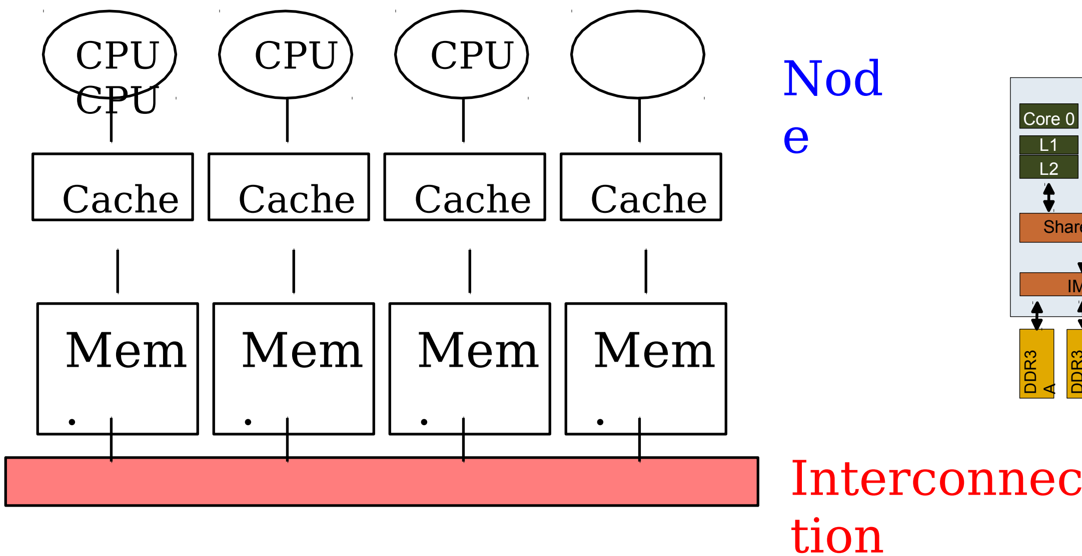
Taxonomy of Parallel Computers

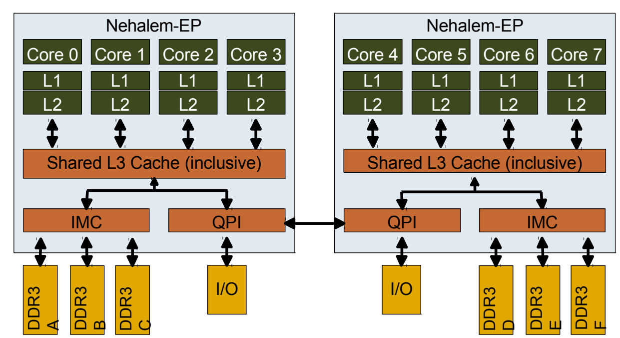
- According to physical organization of processors and memory:
  - Physically centralized memory, uniform memory access (UMA)
    - All memory is allocated at same distance from all processors
    - Also called symmetric multiprocessors (SMP)
    - Memory bandwidth is fixed and must accommodate all processors → does not scale to large number of processors
    - Used in CMPs today (single-socket ones)



Taxonomy of Parallel Computers

- According to physical organization of processors and memory:
  - Physically distributed memory, non-uniform memory access (NUMA)
    - A portion of memory is allocated with each processor (node)
    - Accessing local memory is much faster than remote memory
    - If most accesses are to local memory than overall memory bandwidth increases linearly with the number of processors
    - Used in multi-socket CMPs E.g Intel Nehalem





Taxonomy of Parallel Computers

- According to memory communication model
  - Shared address or shared memory
    - Processes in different processors can use the same virtual address space
    - Any processor can directly access memory in another processor node
    - Communication is done through shared memory variables
    - Explicit synchronization with locks and critical sections
    - Arguably easier to program??
  - Distributed address or message passing
    - Processes in different processors use different virtual address spaces
    - Each processor can only directly access memory in its own node
    - Communication is done through explicit messages
    - Synchronization is implicit in the messages
    - Arguably harder to program??
    - Some standard message passing libraries (e.g., MPI)

Shared Memory vs. Message Passing

- Shared memory

Producer (p1)	Consumer (p2)
flag = 0;	flag = 0;
…	…
a = 10;	while (!flag) {}
flag = 1;	x = a * y;

- Message passing

Producer (p1)	Consumer (p2)
…	…
a = 10;	receive(p1, b, label);
send(p2, a, label);	x = b * y;

Shared Memory Multiprocessor (SMP)

- Q1 – Single address space shared by all processors/cores

- Q2 – Processors coordinate/communicate through shared variables in memory (via loads and stores)
  - Use of shared data must be coordinated via synchronization primitives (locks) that allow access to data to only one processor at a time

- All multicore computers today are SMP

Multiprocessor Caches

- Memory is a performance bottleneck even with one processor
- Use caches to reduce bandwidth demands on main memory
- Each core has a local private cache holding data it has accessed recently
- Only cache misses have to access the shared common memory