Parallel Architecture/Programming

Hung-Wei Tseng
Von Neumann architecture

CPU is a dominant factor of performance since we heavily rely on it to execute programs.

By pointing “PC” to different part of your memory, we can perform different functions!

By pointing “PC” to different part of your memory, we can perform different functions!
CPU performance scales well before 2002

52%/year
The slowdown of CPU scaling

- Using PCIe P2P to eliminate CPU/DRAM from the data plane
- Reduce the CPU load
- Avoid redundant data copies in DRAM
- Improve the latency

1.5x in 67 months

5x in 67 months
Intel P4 (2000) 1 core
AMD Athlon 64 X2 (2005) 2 cores
Intel Nahalem (2010) 4 cores

SPARC T3 (2010) 16 cores
Nvidia Tegra 3 (2011) 5 cores
AMD Zambezi (2011) 16 cores
Die photo of a CMP processor
Memory hierarchy on CMP

- Each processor has its own local cache
Comparing SMT and CMP

• Assuming both application X and application Y have similar instruction combination, say 60% ALU, 20% load/store, and 20% branches. Consider two processors:

P1: CMP with a 2-issue pipeline on each core. Each core has a private L1 32KB D-cache

P2: SMT with a 4-issue pipeline. 64KB L1 D-cache

Which one do you think is better?
A. P1
B. P2
Parallelism
Parallelism in modern computers

• Instruction-level parallelism
• Data-level parallelism
• Thread-level parallelism
Processing models

- **SISD** — single instruction stream, single data
  - Pipelining instructions within a single program
  - Superscalar
- **SIMD** — single instruction stream, multiple data
  - Vector instructions
  - GPUs
- **MIMD** — multiple instruction stream (e.g. multiple threads, multiple processes), multiple data
  - Multicore processors
  - Multiple processors
  - Simultaneous multithreading
Processing models

- SISD — single instruction stream, single data
  - Pipelining instructions within a single program
  - Superscalar
- SIMD — single instruction stream, multiple data
  - Vector instructions
  - GPUs
- MIMD — multiple instruction stream (e.g. multiple threads, multiple processes), multiple data
  - Multicore processors
  - Multiple processors
  - Simultaneous multithreading

We will focus on this today
• How can we create programs to utilize these cores?

A. Parallel programming is easy, programmers will just write parallel programs from now on.

B. Parallel programming was hard, but architects have generally solved this problem in the 10 years since we saw the problem

C. You don’t need to write parallel code, Intel’s new compilers know how to extract thread level parallelism

D. Intel (and everyone else) is just building the chips, it’s on you to figure out how to use them.
Speedup an application with multi-threaded programming models
Parallel programming

- To exploit parallelism you need to break your computation into multiple “processes” or multiple “threads”
- Processes (in OS/software systems)
  - Separate programs actually running (not sitting idle) on your computer at the same time.
  - Each process will have its own virtual memory space and you need explicitly exchange data using inter-process communication APIs
- Threads (in OS/software systems)
  - Independent portions of your program that can run in parallel
  - All threads share the same virtual memory space
- We will refer to these collectively as “threads”
  - A typical user system might have 1-8 actively running threads.
  - Servers can have more if needed (the sysadmins will hopefully configure it that way)
Multi-processed model

- You can use fork() to create a child process
- You need to use files/sockets/MPI to exchange data
Multi-processed model

- You can use fork() to create a child process
- You need to use files/sockets/MPI to exchange data
Multi-threaded model

- Process
- Thread
- Stack
- Physical memory
- Code
- Static Data
- Data
- Heap
- Physical memory

Memory Addresses:
- Code: 0x000000000000
- Stack: 0xFFFFFFFFFFFF
- Heap: 0xFFFFFFFFFFFF

Thread:
- Thread 1
- Thread 2
- Thread 3

Diagram showing the relationship between processes, threads, memory regions, and physical memory.
Multi-threaded model

Physical memory

0x000000000000

0xFFFFFFFFFFFF

Stack

Core

Code

Static Data

Heap

Data

0xFFFFFFFFFFFFFFF
POSIX threads

- All threads within the same process share the same memory space
- You may use pthread_create to spawn a thread

```c
/* Do matrix multiplication */
for(i = 0 ; i < NUM_OF_THREADS ; i++)
{
    tids[i] = i;
    pthread_create(&thread[i], NULL, threaded_blockmm, &tids[i]);
}
for(i = 0 ; i < NUM_OF_THREADS ; i++)
    pthread_join(thread[i], NULL);
```

Spawn a thread

Synchronize and wait for thread to terminate
for(i = 0; i < ARRAY_SIZE; i+=4) {
    va = _mm_load_ps(&a[i]);
    vb = _mm_load_ps(&b[i]);
    vt = _mm_add_ps(va, vb);
    _mm_store_ps(&c[i],vt);
}

NUM_OF_THREADS = ARRAY_SIZE/4;
thread = (pthread_t *)malloc(sizeof(pthread_t)*NUM_OF_THREADS);
tids = (int *)malloc(sizeof(pthread_t)*NUM_OF_THREADS);

for(i = 0 ; i < NUM_OF_THREADS ; i++)
{
    tids[i] = i;
    pthread_create(&thread[i], NULL, threaded_vadd, &tids[i]);
}
for(i = 0 ; i < NUM_OF_THREADS ; i++)
    pthread_join(thread[i], NULL);

void *threaded_vadd(void *thread_id)
{
    int tid = *(int *)thread_id;
    int i = tid * 4;
    va = _mm_load_ps(&a[i]);
    vb = _mm_load_ps(&b[i]);
    vt = _mm_add_ps(va, vb);
    _mm_store_ps(&c[i],vt);
    return NULL;
}
Cache and shared memory model
Supporting POSIX threads

• Provide a single memory space that all processors can share
• All threads within the same program shares the same address space.
• Threads communicate with each other using shared variables in memory
• Provide the same memory abstraction as single-thread programming
Cache on Multiprocessor

• Coherency
  • Guarantees all processors see the same value for a variable/memory address in the system when the processors need the value at the same time
    • What value should be seen

• Consistency
  • All threads see the change of data in the same order
    • When the memory operation should be done
Simple cache coherency protocol

• Snooping protocol
  • Each processor broadcasts / listens to cache misses
• State associate with each block (cacheline)
  • Invalid
    • The data in the current block is invalid
  • Shared
    • The processor can read the data
    • The data may also exist on other processors
  • Exclusive
    • The processor has full permission on the data
    • The processor is the only one that has up-to-date data
Accessing the cache

The position of the requesting word in a cache block

Hit: The data was found in the cache
Miss: The data was not found in the cache
Simple cache coherency protocol

- Invalid
  - read/write miss (bus)
  - write miss (processor)
- Shared
  - read miss (processor)
  - write miss (bus)
- Exclusive
  - write hit
  - write miss (bus)
  - write request (processor)
  - read miss (bus)
  - write back data

read miss/hit
Cache coherency practice

- What happens when core 0 modifies 0x1000?, which belongs to the same cache block as 0x1000?
Cache coherency practice

• Then, what happens when core 2 reads 0x1000?
Assuming that we are running the following code on a CMP with a cache coherency protocol, how many of the following outputs are possible? (a is initialized to 0 as assume we will output more than 10 numbers)

\[
\begin{array}{l}
\text{thread 1} & \text{thread 2} \\
\text{while(1)} & \text{while(1)} \\
\quad \text{printf("%d ",a);} & \quad \text{a++;} \\
\end{array}
\]

① 0 1 2 3 4 5 6 7 8 9  
② 1 2 5 9 3 6 8 10 12 13  
③ 1 1 1 1 1 1 1 1 64 100  
④ 1 1 1 1 1 1 1 1 1 100  
A. 0  
B. 1  
C. 2  
D. 3  
E. 4
It’s show time!

- Demo!

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>while(1)</td>
<td>while(1)</td>
</tr>
<tr>
<td>printf(&quot;%d &quot;,a);</td>
<td>a++;</td>
</tr>
</tbody>
</table>

Assuming that we are running the following code on a CMP with a cache coherency protocol, how many of the following outputs are possible? (a is initialized to 0)

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>while(1)</td>
<td>while(1)</td>
</tr>
<tr>
<td>printf(&quot;%d &quot;, a);</td>
<td>a++;</td>
</tr>
</tbody>
</table>

① 0 1 2 3 4 5 6 7 8 9
② 1 2 5 9 3 6 8 10 12 13
③ 1 1 1 1 1 1 1 1 64 100
④ 1 1 1 1 1 1 1 1 1 100

A. 0
B. 1
C. 2
D. 3
E. 4
int loop;
int main()
{
    pthread_t thread;
    loop = 1;

    pthread_create(&thread, NULL, modifyloop, NULL);
    while(loop == 1)
    {
        continue;
    }
    pthread_join(thread, NULL);
    fprintf(stderr,"User input: %d\n", loop);
    return 0;
}

void* modifyloop(void *x)
{
    sleep(1);
    printf("Please input a number:\n");
    scanf(\"%d\",&loop);
    return NULL;
}
```c
volatile int loop;

int main()
{
    pthread_t thread;
    loop = 1;

    pthread_create(&thread, NULL,
                  modifyloop, NULL);
    while(loop == 1)
    {
        continue;
    }
    pthread_join(thread, NULL);
    fprintf(stderr,"User input: %d\n", loop);
    return 0;
}

void* modifyloop(void *x)
{
    sleep(1);
    printf("Please input a number:\n");
    scanf("%d", &loop);
    return NULL;
}
```

Observer

prevents the compiler from putting loop in the register
• Comparing implementations of thread_vadd — L and R, please identify which one will be performing better and why

A. L is better, because the cache miss rate is lower
B. R is better, because the cache miss rate is lower
C. L is better, because the instruction count is lower
D. R is better, because the instruction count is lower
E. Both are about the same
void *threaded_vadd(void *thread_id) {
    __m128 va, vb, vt;
    int tid = *(int *)thread_id;
    int i = tid * 4;
    for(i = tid * 4; i < ARRAY_SIZE; i+=4*NUM_OF_THREADS) {
        va = _mm_load_ps(&a[i]);
        vb = _mm_load_ps(&b[i]);
        vt = _mm_add_ps(va, vb);
        _mm_store_ps(&c[i],vt);
    }
    return NULL;
}

void *threaded_vadd(void *thread_id) {
    __m128 va, vb, vt;
    int tid = *(int *)thread_id;
    int i = tid * 4;
    for(i = tid*(ARRAY_SIZE/NUM_OF_THREADS); i < (tid+1)*(ARRAY_SIZE/NUM_OF_THREADS); i+=4) {
        va = _mm_load_ps(&a[i]);
        vb = _mm_load_ps(&b[i]);
        vt = _mm_add_ps(va, vb);
        _mm_store_ps(&c[i],vt);
    }
    return NULL;
}
Cache coherency practice

- Assume $a[0]$’s address is 0x1000
- Now, core 1 updates $a[4]$-$a[7]$ (address: 0x1010-0x101F, which belongs the same block as 0x1000?)
- Then, if Core 0 accesses $a[0]$-$a[3]$, starting from 0x1000, it will be a miss!
Comparing implementations of thread_vadd — L and R, please identify which one will be performing better and why

**Version L**
```c
void *threaded_vadd(void *thread_id)
{
    __m128 va, vb, vt;
    int tid = *(int *)thread_id;
    int i = tid * 4;
    for(i = tid * 4; i < ARRAY_SIZE; i+=4*NUM_OF_THREADS)
    {
        va = _mm_load_ps(&a[i]);
        vb = _mm_load_ps(&b[i]);
        vt = _mm_add_ps(va, vb);
        _mm_store_ps(&c[i],vt);
    }
    return NULL;
}
```

**Version R**
```c
void *threaded_vadd(void *thread_id)
{
    __m128 va, vb, vt;
    int tid = *(int *)thread_id;
    int i = tid * 4;
    for(i = tid*(ARRAY_SIZE/NUM_OF_THREADS); i < (tid+1)*(ARRAY_SIZE/NUM_OF_THREADS); i+=4)
    {
        va = _mm_load_ps(&a[i]);
        vb = _mm_load_ps(&b[i]);
        vt = _mm_add_ps(va, vb);
        _mm_store_ps(&c[i],vt);
    }
    return NULL;
}
```

A. L is better, because the cache miss rate is lower
B. R is better, because the cache miss rate is lower
C. L is better, because the instruction count is lower
D. R is better, because the instruction count is lower
E. Both are about the same

Performance comparison

- Comparing implementations of thread_vadd — L and R, please identify which one will be performing better and why
4C model

- 3Cs:
  - Compulsory, Conflict, Capacity
- Coherency miss:
  - A “block” invalidated because of the sharing among processors.
Types of coherence misses

- True sharing
  - Processor A modifies X, processor B also want to access X.

- False Sharing
  - Processor A modifies X, processor B also want to access Y. However, Y is invalidated because X and Y are in the same block!
Consider the given program. You can safely assume the caches are coherent. How many of the following outputs will you see?

① (0, 0)
② (0, 1)
③ (1, 0)
④ (1, 1)

A. 0
B. 1
C. 2
D. 3
E. 4

Again — how many values are possible?

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

volatile int a, b;
volatile int x, y;
volatile int f;

void* modifya(void *z) {
    a = 1;
    x = b;
    return NULL;
}

void* modifyb(void *z) {
    b = 1;
    y = a;
    return NULL;
}

int main() {
    int i;
    pthread_t thread[2];
    pthread_create(&thread[0], NULL, modifya, NULL);
    pthread_create(&thread[1], NULL, modifyb, NULL);
    pthread_join(thread[0], NULL);
    pthread_join(thread[1], NULL);
    fprintf(stderr, "(%d, %d)\n", x, y);
    return 0;
}
```
Why (0,0)?

- Processor/compiler may reorder your memory operations/instructions
  - Coherence protocol can only guarantee the update of the same memory address
  - Processor can serve memory requests without cache miss first
  - Compiler may store values in registers and perform memory operations later
- Each processor core may not run at the same speed (cache misses, branch mis-prediction, I/O, voltage scaling and etc..)
- Threads may not be executed/scheduled right after it’s spawned
Again — how many values are possible?

- Consider the given program. You can safely assume the caches are coherent. How many of the following outputs will you see?
  ① (0, 0)
  ② (0, 1)
  ③ (1, 0)
  ④ (1, 1)

A. 0  
B. 1  
C. 2  
D. 3  
E. 4
fence instructions

• x86 provides an “mfence” instruction to prevent reordering across the fence instruction

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=1; mfence x=b;</td>
<td>b=1; mfence</td>
</tr>
<tr>
<td>a=1 must occur/update before mfence</td>
<td>b=1 must occur/update before mfence</td>
</tr>
<tr>
<td>y=a;</td>
<td></td>
</tr>
</tbody>
</table>

You won’t see (0,0) at least...

• x86 only supports this kind of “relaxed consistency” model. You still have to be careful enough to make sure that your code behaves as you expected
Why is parallel programming hard?

• Processor behaviors are non-deterministic
  • You cannot predict which processor is going faster
  • You cannot predict when OS is going to schedule your thread
• Cache coherency only guarantees that everyone would eventually have a coherent view of data, but not when
• Cache consistency is hard to support