Parallel architectures (I)

Hung-Wei Tseng
Von Neumann architecture

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

CPU is a dominant factor of performance since we heavily rely on it to execute programs!
CPU performance scales well before 2002

52%/year
The slowdown of CPU scaling

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
Outline

• Different forms of parallelism
• Brief introduction to parallel computing
• Chip multiprocessors
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
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)
- You can use fork() to create a child process
- You need to use files/sockets/MPI to exchange data
Multi-threaded model

Process

Thread
0x000000000000

Thread

Code

Static Data

Heap

Data

Stack

Physical memory

0xFFFFFFFFFFFFFFF

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 a for thread to terminate
Demo — vector add

```c
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;
}
```
Chip Multiprocessor
• You can use fork() to create a child process
• You need to use files/sockets/MPI to exchange data
• You can use `fork()` to create a child process
• You need to use files/sockets/MPI to exchange data
Multi-threaded model

Physical memory

Thread
0x000000000000

Thread

Thread

Thread

Process

Code

Static Data

Data

Heap

Stack

Physical memory

0xFFFFFFFFFFFF

0xFFFFFFFFFFFFFFF
Multi-threaded model

Physical memory

Core

0x0000000000000000

Code

Static Data

Data

Heap

Stack

0xFFFFFFFFFFFF

0xFFFFFFFFFFFFFF
Simple idea...

- Connecting all processor and shared memory to a bus.
- Processor speed will be slow b/c all devices on a bus must run at the same speed
Memory hierarchy on CMP

- Each processor has its own local cache.
Die photo of a CMP processor
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

memory address: 0x8 0 0 0 0 1 5 8

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
- Shared
- Exclusive

- read/write miss (bus)
- read miss (processor)
- write miss (bus)
- write request (processor)
- write hit
- write request (processor)
- write back data
- 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?
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>
thread 1

```c
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;
}
```

thread 2

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

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
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!
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!
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