Thread Level Parallelism (TLP)

Overview

• Goal: Higher performance through parallelism

• Job-level (process-level) parallelism
  ❖ High throughput for independent jobs

• Application-level parallelism
  ❖ Single program run on multiple processors → speedup

• Each core can operate concurrently and in parallel
• Multiple threads may operate in a time sliced fashion on a single core
Thread Level Parallelism (TLP)

- Multiple threads of execution
- Exploit ILP in each thread
- Exploit concurrent execution across threads

Instruction and Data Streams

- Taxonomy due to M. Flynn

<table>
<thead>
<tr>
<th>Data Streams</th>
<th>Single</th>
<th>Multiple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Streams</td>
<td>Single</td>
<td>Multiple</td>
</tr>
<tr>
<td><strong>SISD:</strong> Intel Pentium 4</td>
<td><strong>SIMD:</strong> SSE instructions of x86</td>
<td></td>
</tr>
<tr>
<td>Multiple</td>
<td><strong>MISD:</strong> No examples today</td>
<td><strong>MIMD:</strong> Intel Xeon e5345</td>
</tr>
</tbody>
</table>

Example: Multithreading (MT) in a single address space
Recall The Executable Format

Object file ready to be linked and loaded

header
text
static data
reloc
symbol table
debug

Linker

Loader

An executable instance or Process

Static Libraries

What does a loader do?

Process

- A process is a running program with state
  - Stack, memory, open files
  - PC, registers

- The operating system keeps tracks of the state of all processors
  - E.g., for scheduling processes

- There many processes for the same application
  - E.g., web browser

- Operating systems class for details
Process Level Parallelism

- Parallel processes and throughput computing
- Each process itself does not run any faster

From Processes to Threads

- Switching processes on a core is expensive
  - A lot of state information to be managed
- If I want concurrency, launching a process is expensive
- How about splitting up a single process into parallel computations?
  - Lightweight processes or threads!
Thread Parallel Execution

A Thread

- A separate, *concurrently executable* instruction stream within a process
  - Minimum amount state to execute on a core
    - Program counter, registers, stack
    - Remaining state shared with the parent process
      - Memory and files

- Support for creating threads
- Support for merging/terminating threads
- Support for synchronization between threads
  - In accesses to shared data

Our datapath so far!
A Simple Example

// Illustrate using CreateThread in gthreads
// George F. Riley, Georgia Tech, ECSE383, Spring 2011
#include "gthreads.h"  // Must be included to use the gthreads library

void BubbleSort(int d[], int startingPoint, int length)
   { // This is the thread starting point
      // This is where, in this example, the sorting of array d will be done
      EndThread(); // Call this just before exiting
   }

const int nThreads = 4; // Number of threads desired
const int maxSize = 512000; // Largest sort size

int main()
   {
      int d[maxSize];  // Array to be sorted
      int start = 0;   // Starting point of sub-array
      int length = (omitted); // Length of sub-array
      for (int k = 0; k < nThreads; ++k)
         { // Create each of the four sorting threads
            createThread(BubbleSort, d, start, length);
            // Need more code here. (omitted)
         }
      // At this point all threads are created
      WaitAllThreads();  // This waits until all child threads are done
      // Perform the merge procedure to merge the separate sub-arrays
   }

Thread Execution: Basics

Thread #1
- PC, registers, stack pointer
- Stack
- Heap
- Static data
- funcA()
- funcB()

Thread #2
- PC, registers, stack pointer
- Stack

Threads Execution on a Single Core

- Hardware threads
  - Each thread has its own hardware state

- Switching between threads on each cycle to share the core pipeline – why?

Thread #1

```
lw $t0, label($0)
lw $t1, label1($0)
and $t2, $t0, $t1
andi $t3, $t1, 0xffff
srl $t2, $t2, 12
```

```
    lw
    lw
    ……
```

Interleaved execution

Thread #2

```
lw $t3, 0($t0)
add $t2, $t2, $t3
addi $t0, $t0, 4
addi $t1, $t1, -1
bne $t1, $zero, loop
```

```
lw
lw
lw
lw
lw
lw
```

Improve utilization!

No pipeline stall on load-to-use hazard!

An Example Datapath

From Poonacha Kongetira, Microarchitecture of the UltraSPARC T1 CPU
Conventional Multithreading

- Zero-overhead context switch
- Duplicated contexts for threads

Execution Model: Multithreading

- **Fine-grain** multithreading
  - Switch threads after each cycle
  - Interleave instruction execution

- **Coarse-grain** multithreading
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but does not hide short stalls (e.g., data hazards)
  - If one thread stalls (e.g., I/O), others are executed

_Courtesy H. H. Lee_
Software Threads

Thread #1
- PC, registers, stack pointer
- Stack

Thread #2
- PC, registers, stack pointer
- Stack

Need to save and restore thread state

Static data
- funcA()
- funcB()

Create_thread(funcB)
Create_thread(funcA)

WaitAllThreads()

Thread Synchronization

Process
- thread
- Share data?

Processor
- Cache
- Processor
- Cache
- Processor
- Cache
- Processor

Interconnection Network
- Memory
- I/O
Thread Interactions

- What about shared data?
  - Need synchronization support

- Several different types of synchronization: we will look at one in detail
  - We are specifically interested in the exposure in the ISA

Example: Communicating Threads

The Producer calls

```c
while (1) {
    while (count == BUFFER_SIZE) ; // do nothing
    // add an item to the buffer
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
}
```

**Thread 1**
Example: Communicating Threads

The Consumer calls

```c
while (1) {
    while (count == 0)
        ; // do nothing
    // remove an item from the buffer
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
}
```

Thread 2

Uniprocessor Implementation

- `count++` could be implemented as
  ```c
  register1 = count;
  register1 = register1 + 1;
  count = register1;
  ```

- `count--` could be implemented as
  ```c
  register2 = count;
  register2 = register2 - 1;
  count = register2;
  ```

- Consider this execution interleaving:
  ```c
  S0: producer execute register1 = count {register1 = 5}
  S1: producer execute register1 = register1 + 1 {register1 = 6}
  S2: consumer execute register2 = count {register2 = 5}
  S3: consumer execute register2 = register2 - 1 {register2 = 4}
  S4: producer execute count = register1 {count = 6 }
  S5: consumer execute count = register2 {count = 4}
  ```
Synchronization

- We need to prevent certain instruction interleaving
  - Or at least be able to detect violations!

- Some sequence of operations (instructions) must happen atomically
  - E.g., `register1 = count;
    register1 = register1 + 1;
    count = register1;`
  - atomic operations/instructions

- Serializing access to shared resources is a basic requirement of concurrent computation
  - What are critical sections?

Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don’t synchronize
    - Result depends of order of accesses

- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write

- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions
Implementing an Atomic Operation

// lock object is shared by all threads

while (lock.getAndSet(true)) \rightarrow \textit{Atomic}
    Thread.yield();

Update count;

lock.set(false) \rightarrow \textit{Atomic}

---

Synchronization in MIPS

- Load linked: \texttt{ll rt, offset(rs)}
- Store conditional: \texttt{sc rt, offset(rs)}
  - Succeeds if location not changed since the \texttt{ll}
    - Returns 1 in rt
  - Fails if location is changed
    - Returns 0 in rt
- Example: atomic swap (to test/set lock variable)
  
  \texttt{try: add $t0,$zero,$s4 ; copy exchange value}
  \texttt{\quad ll $t1,0($s1) ; load linked}
  \texttt{\quad sc $t0,0($s1) ; store conditional}
  \texttt{\quad beq $t0,$zero,try ; branch store fails}
  \texttt{\quad add $s4,$zero,$t1 ; put load value in $s4}
Other Synchronization Primitives

- **test&set(lock)**
  - Atomically read and set a lock variable

- **swap r1, r2, [r0]**
  - With 1/0 values this functions as a lock variable

  ....and a few others

---

Simultaneous Multithreading

- In *multiple-issue dynamically scheduled processors*
  - Instruction-level parallelism across threads
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available

- Example: Intel Pentium-4 HT
  - Two threads: duplicated registers, shared function units and caches
  - Known as *Hyperthreading* in Intel terminology
Hyper-threading

- Implementation of Hyper-threading adds less than 5% to the chip area
- Principle: share major logic components (functional units) and improve utilization
- Architecture State: All core pipeline resources needed for executing a thread

Multithreading with ILP: Examples
How Much Parallelism in a Thread?

Performance

Made sense to go Superscalar/OOO: good ROI

Very little gain for substantial effort

Source: G. Loh

Commodity Multicore Processor

Coherent Shared Memory Programming Model

From www.zdnet.com
Parallel Programming

- Parallel software is the problem
- Need to get significant performance improvement
  - Otherwise, just use a faster uniprocessor, since it’s easier!
- Difficulties
  - Partitioning
  - Coordination
  - Communications overhead
Amdahl’s Law

- Sequential part can limit speedup

- Example: 100 processors, 90× speedup?
  - $T_{new} = T_{parallelizable}/100 + T_{sequential}$
  - $Speedup = \frac{1}{(1-F_{parallelizable}) + F_{parallelizable}/100} = 90$
  - Solving: $F_{parallelizable} = 0.999$

- Need sequential part to be 0.1% of original time

Scaling Example

- Workload: sum of 10 scalars, and $10 \times 10$ matrix sum
  - Speed up from 10 to 100 processors

- Single processor: Time = $(10 + 100) \times t_{add}$

- 10 processors
  - Time = $10 \times t_{add} + 100/10 \times t_{add} = 20 \times t_{add}$
  - Speedup = $110/20 = 5.5$ (55% of potential)

- 100 processors
  - Time = $10 \times t_{add} + 100/100 \times t_{add} = 11 \times t_{add}$
  - Speedup = $110/11 = 10$ (10% of potential)

- Idealized model
  - Assumes load can be balanced across processors
Scaling Example (cont)

- What if matrix size is $100 \times 100$?
- Single processor: $\text{Time} = (10 + 10000) \times t_{\text{add}}$
- 10 processors
  - $\text{Time} = 10 \times t_{\text{add}} + 10000/10 \times t_{\text{add}} = 1010 \times t_{\text{add}}$
  - $\text{Speedup} = 10010/1010 = 9.9$ (99% of potential)
- 100 processors
  - $\text{Time} = 10 \times t_{\text{add}} + 10000/100 \times t_{\text{add}} = 110 \times t_{\text{add}}$
  - $\text{Speedup} = 10010/110 = 91$ (91% of potential)
- Idealized model
  - Assuming load balanced

Strong vs Weak Scaling

- **Strong scaling**: problem size fixed
  - As in example
- **Weak scaling**: problem size proportional to number of processors
  - 10 processors, $10 \times 10$ matrix
    - $\text{Time} = 20 \times t_{\text{add}}$
  - 100 processors, $32 \times 32$ matrix
    - $\text{Time} = 10 \times t_{\text{add}} + 10000/100 \times t_{\text{add}} = 20 \times t_{\text{add}}$
  - Constant performance in this example
  - For a fixed size system grow the number of processors to improve performance
### Programming Model: Message Passing

- Each processor has private physical address space
- Hardware sends/receives messages between processors

![Diagram showing processors, caches, memory, and interconnection network.]

### Parallelism

- Write message passing programs
- Explicit send and receive of data
  - Rather than accessing data in shared memory

![Diagram showing send() and receive() calls between two processes.]

---

(39) Programming Model: Message Passing

(40) Parallelism
High Performance Computing

- The dominant programming model is message passing
- Scales well but requires programmer effort
- Science problems have fit this model well to date

A Simple MPI Program

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>

int main(argc, argv)
    int argc; char * argv[];
{
    int myid, numprocs;
    int tag, source, destination, count;
    int buffer;
    MPI_Status status;

    MPI_Init(& argc, & argv);
    MPI_Comm_size(MPI_COMM_WORLD,& numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,& myid);
    tag=1234;
    source=0; destination=1; count=1;
    if(myid == source){
        buffer=5678;
        MPI_Send(& buffer,count,MPI_INT,destination,tag,MPI_COMM_WORLD);
        printf("processor %d sent %d\n",myid,buffer); }
    if(myid == destination){
        MPI_Recv(& buffer,count,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
        printf("processor %d got %d\n",myid,buffer);
    }
    MPI_Finalize();
}
```

The Message Passing Interface (MPI) Library

From http://geco.mines.edu/workshop/class2/examples/mpi/c_ex01.c
A Simple MPI Program

```c
#include "mpi.h"
#include <stdio.h>
#include <math.h>

int main(int argc, char *argv[])
{
    int n, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    while (1) {
        if (myid == 0) {
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d", &n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0) break;
        else {
            h = 1.0 / (double) n;
            sum = 0.0;
            for (i = myid + 1; i <= n; i += numprocs) {
                x = h * ((double) i - 0.5);
                sum += (4.0 / (1.0 + x*x));
            }
            mypi = h * sum;
            MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
            if (myid == 0) {
                printf("pi is approximately %.16f, Error is %.16f
", pi, fabs(pi - PI25DT));
            }
        }
    }
    MPI_Finalize();
    return 0;
}
```

Loosely Coupled Clusters

- Network of independent computers
  - Each has private memory and OS
  - Connected using I/O system
    - E.g., Ethernet/switch, Internet
- Suitable for applications with independent tasks
  - Web servers, databases, simulations, ...
- High availability, scalable, affordable
- Problems
  - Administration cost (prefer virtual machines)
  - Low interconnect bandwidth
    - c.f. processor/memory bandwidth on an SMP
Grid Computing

- Separate computers interconnected by long-haul networks
  - E.g., Internet connections
  - Work units farmed out, results sent back

- Can make use of idle time on PCs
  - E.g., SETI@home, World Community Grid

Study Guide

- What is the difference between hardware MT and software MT

- Distinguish between TLP and ILP

- Given two threads of computation, the MIPS pipeline, fine grained MT, show the state of the pipeline after 7 cycles

- How many threads do you need with fine grain FT before you branch penalties are no longer a problem?

- With coarse grain MT on a datapath with full forwarding, can you still have load-to-use hazards?
Study Guide (cont.)

- Name two differences between the coherent shared memory and message passing architectures

Glossary

- Atomic operations
- Coarse grain MT
- Fine grained MT
- Grid computing
- Hyperthreading
- Multithreading
- Message Passing Interface

- Simultaneous MT
- Strong scaling
- Swap instruction
- Test & set operation
- Thread Level Parallelism (TLP)
- Weak scaling