Lecture 25 (Mon & Wed 12/01 & 03/2008)

- HW #4 (optional) - Due Fri Dec 5 during class
- Lab #4 Hardware - Due Fri Dec 5 at 5pm

Today: Parallelism!

Pipelining vs. Parallel processing

- In both cases, multiple "things" processed by multiple "functional units"
  Pipelining: each thing is broken into a sequence of pieces, where each piece is handled by a different (specialized) functional unit
  Parallel processing: each thing is processed entirely by a single functional unit

- We will briefly introduce the key ideas behind parallel processing
  - instruction level parallelism
  - thread-level parallelism

Exploiting Parallelism

- Of the computing problems for which performance is important, many have inherent parallelism

- Best example: computer games
  - Graphics, physics, sound, AI etc. can be done separately
  - Furthermore, there is often parallelism within each of these:
    - Each pixel on the screen’s color can be computed independently
    - Non-contacting objects can be updated/simulated independently
    - Artificial intelligence of non-human entities done independently

- Another example: Google queries
  - Every query is independent
  - Google is read-only!

Parallelism at the Instruction Level

```
add $2 <- $3, $4
or $2 <- $2, $4
lw $6 <- 0($4)
add $7 <- $6, 0x5
sub $8 <- $8, $4
```

Dependences?
- RAW
- WAR

When can we reorder instructions?
- Surprisingly: Multiple instructions executing in parallel at "same" stage
**Data Dependences**

**Flow dependence - RAW.** Read-After-Write. A “true” dependence. Read a value after it has been written into a variable.

**Anti-dependence - WAR.** Write-After-Read. Write a new value into a variable after the old value has been read.

**Output dependence - WAW.** Write-After-Write. Write a new value into a variable and then later on write another value into the same variable.

---

**Exploiting Parallelism at the Data Level**

- Consider adding together two arrays:

  ```c
  void array_add(int A[], int B[], int C[], int length) {
    int i;
    for (i = 0; i < length; ++i) {
      C[i] = A[i] + B[i];
    }
  }
  ```

  Operating on one element at a time

---

**Exploiting Parallelism at the Data Level**

- Consider adding together two arrays:

  ```c
  void array_add(int A[], int B[], int C[], int length) {
    int i;
    for (i = 0; i < length; ++i) {
      C[i] = A[i] + B[i];
    }
  }
  ```

  Operating on one element at a time
Exploiting Parallelism at the Data Level (SIMD)

- Consider adding together two arrays:

  ```c
  void array_add(int A[], int B[], int C[], int length) {
      int i;
      for (i = 0 ; i < length ; ++i) {
          C[i] = A[i] + B[i];
      }
  }
  ```

  Operate on MULTIPLE elements

Single Instruction, Multiple Data (SIMD)

Intel SSE/SSE2 as an example of SIMD

- Added new 128 bit registers (XMM0 - XMM7), each can store:
  - 4 single precision FP values (SSE) 4 * 32b
  - 2 double precision FP values (SSE2) 2 * 64b
  - 16 byte values (SSE2) 16 * 8b
  - 8 word values (SSE2) 8 * 16b
  - 4 double word values (SSE2) 4 * 32b
  - 1 128-bit integer value (SSE2) 1 * 128b

Is it always that easy?

- Not always... a more challenging example:

  ```c
  unsigned sum_array(unsigned *array, int length) {
      int total = 0;
      for (int i = 0 ; i < length ; ++i) {
          total += array[i];
      }
      return total;
  }
  ```

  Is there parallelism here?

We first need to restructure the code

```c
unsigned sum_array2(unsigned *array, int length) {
    unsigned total, i;
    unsigned temp[4] = {0, 0, 0, 0};
    for (i = 0 ; i < length & ~0x3 ; i += 4) {
        temp[0] += array[i];
        temp[1] += array[i+1];
        temp[2] += array[i+2];
        temp[3] += array[i+3];
    }
    for (; i < length ; ++ i) {
        total += array[i];
    }
    return total;
}
```
Then we can write SIMD code for the hot part

```c
unsigned sum_array2(unsigned *array, int length) {
    unsigned total, i;
    unsigned temp[4] = {0, 0, 0, 0};
    for (i = 0 ; i < length & ~0x3 ; i += 4) {
        temp[0] += array[i];
        temp[1] += array[i+1];
        temp[2] += array[i+2];
        temp[3] += array[i+3];
    }
    for (; i < length ; ++i) {
        total += array[i];
    }
    return total;
}
```

Thread level parallelism: Multi-Core Processors
- Two (or more) complete processors, fabricated on the same silicon chip
- Execute instructions from two (or more) programs/threads at same time

**Multi-Cores are Everywhere**
- **Intel Core Duo** in new Macs: 2 x86 processors on same chip
- **XBox360**: 3 PowerPC cores
- **Sony Playstation 3**: Cell processor, an asymmetric multi-core with 9 cores (1 general-purpose, 8 special purpose SIMD processors)

Why Multi-cores Now?
- Number of transistors we can put on a chip growing exponentially...
... and performance growing too...

- But power is growing even faster!!
  - Power has become limiting factor in current chips

What is a Thread?

- What does Shared Memory imply?
- Machine model

As programmers, do we care?

- What happens if we run a program on a multi-core?

```c
void array_add(int A[], int B[], int C[], int length) {
    int i;
    for (i = 0 ; i < length ; ++i) {
        C[i] = A[i] + B[i];
    }
}
```

What if we want a program to run on both processors?

- We have to explicitly tell the machine exactly how to do this
  - This is called parallel programming or concurrent programming
- There are many parallel/concurrent programming models
  - We will look at a relatively simple one: fork-join parallelism
  - In CSE 451, you learn about threads and explicit synchronization

```c
def fork_join_parallelism()
    fork_thread()
    join_threads()
```
Fork/Join Logical Example

1. Fork \( N-1 \) threads
2. Break work into \( N \) pieces (and do it)
3. Join \((N-1)\) threads

```c
void array_add(int A[], int B[], int C[], int length) {
    cpu_num = fork(N-1);
    int i;
    for (i = cpu_num; i < length; i += N) {
        C[i] = A[i] + B[i];
    }
    join();
}
```

How good is this with caches?

How does this help performance?

- Parallel speedup measures improvement from parallelization:
  
  \[
  \text{speedup}(p) = \frac{\text{time for best serial version}}{\text{time for version with } p \text{ processors}}
  \]

- What can we realistically expect?

Reason #1: Amdahl’s Law

- In general, the whole computation is not (easily) parallelizable

```
\text{New Execution Time} = \frac{1-s}{p} + s
```

- For example, consider a program that, when executing on one processor, spends 10% of its time in a non-parallelizable region. How much faster will this program run on a 3-processor system?

```
\text{New Execution Time} = \frac{.9T}{3} + .1T = \frac{1-s}{p} + s
```

- What is the maximum speedup from parallelization?
Reason #2: Overhead

```c
void array_add(int A[], int B[], int C[], int length) {
    cpu_num = fork(N-1);
    int i;
    for (i = cpu_num ; i < length ; i += N) {
        C[i] = A[i] + B[i];
    }
    join();
}
```

— Forking and joining is not instantaneous
  • Involves communicating between processors
  • May involve calls into the operating system
    — Depends on the implementation

\[ \text{New Execution Time} = \frac{1-s}{P} + s + \text{overhead(P)} \]

Programming Explicit Thread-level Parallelism

- As noted previously, the programmer must specify how to parallelize
- But, want path of least effort

- Division of labor between the Human and the Compiler
  - Humans: good at expressing parallelism, bad at bookkeeping
  - Compilers: bad at finding parallelism, good at bookkeeping

- Want a way to take serial code and say “Do this in parallel!” without:
  - Having to manage the synchronization between processors
  - Having to know a priori how many processors the system has
  - Deciding exactly which processor does what
  - Replicate the private state of each thread

- OpenMP: an industry standard set of compiler extensions
  - Works very well for programs with structured parallelism.

OpenMP

```c
void array_add(int A[], int B[], int C[], int length) {
    int i;
    for (i =0 ; i < length ; i += 1) {
        // Without OpenMP
        C[i] = A[i] + B[i];
    }
}
```

```c
#include <omp.h>
main () { int nthreads, tid;
    /* Fork a team of threads giving them their own copies of
       variables */
    #pragma omp parallel private(tid)
    /* Obtain and print thread id */
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);
    /* Only master thread does this */
    if (tid == 0) {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }
    /* All threads join master thread and terminate */
}
```

- OpenMP figures out how many threads are available, forks (if necessary),
  divides the work among them, and then joins after the loop.

OpenMP “hello world” Example
Performance Optimization

- Until you are an expert, first write a working version of the program
- Then, and only then, begin tuning, first collecting data, and iterate
  - Otherwise, you will likely optimize what doesn’t matter

“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.” -- Sir Tony Hoare

Summary so Far

- Multi-core is having more than one processor on the same chip.
  - Soon most PCs/servers and game consoles will be multi-core
  - Results from Moore’s law and power constraint
- Exploiting multi-core requires parallel programming
  - Automatically extracting parallelism too hard for compiler, in general.
  - But, can have compiler do much of the bookkeeping for us
  - OpenMP
- Fork-Join model of parallelism
  - At parallel region, fork a bunch of threads, do the work in parallel, and then join, continuing with just one thread
  - Expect a speedup of less than P on P processors
    - Amdahl’s Law: speedup limited by serial portion of program
    - Overhead: forking and joining are not free

Approaches to Parallelism

- Parallel Algorithms
- Parallel Language
- Message passing (low-level)
- Parallelizing compilers
Parallel Languages

- **Fortran 90** - Array language. Triplet notation for array sections. Operations and intrinsic functions possible on array sections.

- **High Performance Fortran (HPF)** - Similar to Fortran 90, but includes data layout specifications to help the compiler generate efficient code.

- **ZPL** - Array-based language at UW. Compiles into C code (highly portable).

- **C*** - C extended for parallelism

Object-Oriented

- concurrent Smalltalk,

- threads in Java, Ada, thread libraries for use in C/C++

Functional

- NESL, Multiplisp

- Id & Sisal (more dataflow)

Distributed Memory Architecture

- Each Processor has direct access only to its local memory

- Processors are connected via high-speed interconnect

- Data structures must be distributed

- Data exchange is done via explicit processor-to-processor communication: send/receive messages

- Example Programming Model: Widely used standard: MPI

Message Passing Interface

MPI is not a language but rather a collection of subroutines and their arguments.

**MPI provides:**

- Point-to-point communication

- Collective operations
  - Barrier synchronization
  - gather/scatter operations
  - Broadcast, reductions

- Different communication modes
  - Synchronous/asynchronous
  - Blocking/non-blocking
  - Buffered/unbuffered

- C/C++ and Fortran bindings

http://www.mpi-forum.org

Shared Memory Architecture

- Processors have direct access to global memory and I/O through bus or fast switching network

- Cache Coherency Protocol guarantees consistency of memory and I/O accesses

- Each processor also has its own memory (cache)

- Data structures are shared in global address space

- Concurrent access to shared memory must be coordinated

- Example Programming Model: OpenMP
OpenMP

- OpenMP: portable shared memory parallelism
- Higher-level API for writing portable multithreaded applications
- Provides a set of compiler directives and library routines for parallel application programmers
- API bindings for Fortran, C, and C++

http://www.OpenMP.org

Parallelizing Compilers

Automatically transform a sequential program into a parallel program.
1. Identify loops whose iterations can be executed in parallel.
2. Often done in stages.

Q: Which loops can be run in parallel?
Q: How should we distribute the work/data?

Writing OpenMP Applications

- Program is built with OpenMP-enabled compiler flags
- Programmer explicitly adds OpenMP pragmas
- Fine tuning using OpenMP Profiling and Performance Analysis Tools

Parallelizing Compiler
Programmer
Inserts OpenMP directives
OpenMP program
Performance tuning