Chip Multiprocessors
COMP35112

Lecture 3 - Parallel Programming using Data Sharing

Graham Riley
Threads

- A thread is a flow of control executing a program
- A process can consist of one or more threads
- All threads in the same process have the same address space
- Can create threads in:
  - Java – covered in COMP25111
  - C (or Fortran) – using Pthreads library
Java (revision)

- Two ways of defining a Thread
  - Class inherits from `java.lang.Thread`
  - Class implements `java.langRunnable`
- In both cases, a `run()` method defines what the thread does when it starts running
- `Thread.start()` gets it going
- Can use `Thread.join()` to wait for it to complete
class ThreadEg extends Thread {
    int me;

    ThreadEg(int me) {
        this.me = me;
    }

    public void run() {
        System.out.println("Thread " + me + " running");
    }
}
class Demo {
    public static void main(String[] args) {
        final int NOWORKERS = 5;
        ThreadEg[] threads = new ThreadEg[NOWORKERS];
        for (int i = 0; i < NOWORKERS; i++)
            threads[i] = new ThreadEg(i);
        for (int i = 0; i < NOWORKERS; i++)
            threads[i].start();
        for (int i = 0; i < NOWORKERS; i++)
            try {
                threads[i].join();
            } catch (InterruptedException e) {
                /* do nothing */
            }
        System.out.println("All done");
    }
}
Pthreads Library (via C)

- Pthreads – a.k.a. POSIX threads
- Use `pthread_create()` to make a thread
  - It executes the function given as a parameter to the call
  - With one argument also given as a parameter to the call
- Also have `pthread_join()`
- Google “pthreads” for lots of documentation!
Pthreads Example - 1

#define null 0
#define NOWORKERS 5

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *proc_life(void *dummy) {
    int me = (int)dummy ;
    printf("Thread %i running\n", me) ;
}
Pthreads Example - 2

```c
pthread_t *initThreads(int n) {
    pthread_t *pid ;
    int i;
    int code ;
    pid = malloc(n * sizeof(pthread_t)) ;
    for (i = 0 ; i < n ; i++) {
        code = pthread_create(pid + i, null, proc_life, (void*)i);
    }
    return pid ;
}
```
int main(void) {
    pthread_t *workers;
    int i;
    workers = initThreads(NOWORKERS);
    for (i = 0; i < NOWORKERS; i++) {
        pthread_join(workers[i], null);
    }
    printf("All done\n");
}
Pthreads: Compile and Run

- Compile by:
  
  \texttt{gcc \textasciitilde o threadEg threadEg.c \textasciitilde lpthread}

- Output (from this and Java version) when run is:

  Thread 0 running
  Thread 1 running
  Thread 2 running
  Thread 3 running
  Thread 4 running
  All done
Data Parallelism

- The easiest form of parallelism to find
- Exploited in vector and array (SIMD) processors
  - E.g. CPUs (SSE, AVX); GPGPUs
- Common in computational science applications
- Divide computation into (nearly) equal sized chunks, each working on part of whole
- Works best when there are no data dependencies between these chunks
Matrix multiply of $n \times n$ matrices can easily be done as

- $n^2$ parallel threads (1 per result element), or
- $n$ parallel threads (1 per row/column of result)
- $p$ parallel threads – each computing $q$ rows/columns of the result, where $p \times q = n$
- etc.

Remember $c(i,j) = \sum a(i,k) \times b(k,j)$

How can the programmer choose which to use?

How do they “tell the system”?
N×N Parallel Matrix Multiplication in Fortran-like notation

DOALL I = 1,N
  DO J = 1,N
    C(I,J) = 0
    DO K = 1,N
      C(I,J) = C(I,J) + A(I,K)*B(K,J)
    END DO
  END DO
END DO

END DO ALL
Explicit vs. Implicit Parallelism

- **Explicit** parallelism is when the programmer spells out what should be done in parallel and what should be done in sequence
  - This does not necessarily mean creating threads, etc. There may be some higher level notation (e.g. OpenMP)
- **Implicit** parallelism is when the system is supposed to work this out for itself
  - Some languages (e.g. functional languages) have different views of computation which allow a default assumption of parallelism (no updatable shared state between functions)
Example Code for Implicit Parallelism

- Some languages (e.g. Matlab, C++ and Java - via libraries, Fortran after f90) allow expressions on arrays:
  \[ A = B + C \]

- with no side effects:
  \[ y = f(x) + g(z) \]

- or even:
  \[ p = h(f(x), g(z)) \]
Automatic Parallelisation

- In an ideal world, a compiler could take an ordinary sequential program and derive the parallelism automatically.

- Manufacturers of parallel machines (pre-multicore) invested considerably in such technology.

- Can do quite well if the programs are simple enough – but dependency analysis can be very hard.

- Must be conservative – i.e. if you cannot be certain that parallel version computes correct result, don’t do it!
Example Problem for Parallelisation

```c
for (int i = 0 ; i < n-3 ; i++) {
    a[i] = a[i+3] + b[i] ;
} // can parallelise by making new version of array a
    // i.e. new_a[i] = a[i+3] + b[i]
for (int i = 5 ; i < n ; i++) {
    a[i] += a[i-5] * 2 ;
} // now that trick doesn’t work! Can instead limit
    // the parallelism to 5 ...
for (int i = 0 ; i < n ; i++) {
    a[i] = a[i + j] + 1 ;
} // is j +ve or –ve?
```
for (int i = 0 ; i < n-3 ; i++)
a[i] = a[i+3] + b[i] ;

<table>
<thead>
<tr>
<th>i=0</th>
<th>a[0] = a[3] + b[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>etc.</td>
<td></td>
</tr>
</tbody>
</table>

Write-after-Read dependency – WAR

A *false* dependency
for (int i = 5 ; i < n ; i++)
a[i] += a[i-5] * 2;

<table>
<thead>
<tr>
<th>i</th>
<th>a[i] equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>a[12] = a[12] + a[7]*2</td>
</tr>
<tr>
<td>etc.</td>
<td></td>
</tr>
</tbody>
</table>

Read-after-Write dependency – RAW

A *true* dependency
Shared Memory

- Everything in this lecture has been on the basis that threads share memory.
- In Java (and Pthreads) the normal language scope rules apply. So threads can declare variables local (and thus private) to themselves – but shared stuff is accessible everywhere.
- If you don’t have shared memory, threads have to communicate via messages – more like a distributed system. We will talk about MPI in another lecture.
Next Lecture

- Anyway, multicore computers share memory, don’t they?
- Next lecture will discuss why this isn’t as simple as it sounds!