Introduction and Motivation

“To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.”

E. Dijkstra, 1972 Turing Award Lecture

The First Software Crisis

Programming in assembly language is complex and programs are non-portable

Solution:

High-level languages such as Fortran
Programmers could view the computer as a “Fortran Virtual Machine”

Net Result:

Programmers are oblivious about the processor
Programmers haven’t needed to know anything about the processors to get good speedups – Moore’s Law
Programs are oblivious about the processor
A Fortran program written in 1970 still works and with increased performance in 2009, even though the processors have changed dramatically

The Second Software Crisis

Large software systems developed by hundreds of programmers and requiring millions of lines of code were difficult to build and maintain

Solution:

Software engineering principles
Structured Programming, followed by Object-Oriented Programming
The Third Crisis

Software developers have abstracted away the processor.
They assume that Moore’s law will always provide performance gains.
However, “The Free Lunch is Over!”


Why is the Free Lunch Over?

Clock speeds are no longer increasing.
Mostly because of heat/power issues.
Moore’s Law is now manifesting itself in multi-core machines.
Multi-core is here; many-core is coming.
Software will need to be multi-core aware in order to make any performance gains.

What can we do about it?

We need to introduce our students to parallelism early in the curriculum.
- New programming models, with appropriate languages and compilers will need to be developed.
- That’s a lot of work!
This tutorial will introduce one technique for introducing multicore programming incrementally.

What IS OpenMP?

- An accepted standard developed in 1997 by a group of industry specialists.
- Consists of a small set of compiler directives, augmented with a small set of library routines and environment variables using the base languages Fortran and C/C++.
- Several OpenMP compilers are available.
  - We will use Visual Studio in this tutorial, since it supports OpenMP and it is widely available in the academic environment.

OpenMP

- For C/C++, the OpenMP directives are contained in #pragma statements.
  - The OpenMP #pragma statements have the format:
    
    #pragma omp directive_name ...

  - where omp is the OpenMP keyword.
  - May be additional parameters (clauses) after the directive name for different options.
  - Some directives require code to specified in a structured block (a statement or statements) that follows the directive and then the directive and structured block form a "construct".

OpenMP (cont'd)

- OpenMP uses a thread-based “fork-join” model.
- Initially, a single thread is executed by a master thread.
- Parallel regions (sections of code) can be executed by multiple threads (a team of threads).
- The parallel directive creates a team of threads with a specified block of code executed by the multiple threads in parallel. The exact number of threads in the team determined by one of several ways.
- Other directives used within a parallel construct to specify parallel for loops and different blocks of code for threads.
A Quick Introduction to Shared Memory Machines

The Shared-Memory Model

What Is a Thread?
- Main thread executes program's "main" function
- Main thread may create other threads to execute other functions
- Threads have own program counter, copy of CPU registers, and stack of activation records
- Threads share process's data, code, address space, and other resources
- Threads have lower overhead than processes

Another Way of Looking at It

Why This View Matters

What Are Threads Good For?
- Making programs easier to understand
- Overlapping computation and I/O
- Improving responsiveness of GUIs
- Improving performance through parallel execution
Fork/Join Programming Model

- When program begins execution, only master thread active
- Master thread executes sequential portions of program
- For parallel portions of program, master thread forks (creates or awakens) additional threads
- At join (end of parallel section of code), extra threads are suspended or die

Relating Fork/Join to Code

More General Threads Model

- When program begins execution, only one user thread, called the main thread, is active
- The main thread can create other threads, which execute other functions
- Created threads can also create additional threads
- How this is done varies according to programming language or API

Fork/Join versus General Model

- You can implement fork/join in the general model
- Hence fork/join a special case of general model
  - More structured
  - More easily optimized
- General model
  - More flexible
  - Better support for functional decompositions

Incremental Parallelization

- Sequential program a special case of threaded program
- Programmers can add parallelism incrementally
- Profile program execution
- Repeat
  - Choose best opportunity for parallelization
  - Transform sequential code into parallel code
- Until further improvements not worth the effort
Utility of Threads

- Threads are flexible enough to implement
  - Domain decomposition
  - Functional decomposition
  - Pipelining
Domain Decomposition

• Sequential Code:
  int a[1000], i;
  for (i = 0; i < 1000; i++)
    a[i] = foo(i);
• Thread 0:
  for (i = 0; i < 500; i++)
    a[i] = foo(i);
• Thread 1:
  for (i = 500; i < 1000; i++)
    a[i] = foo(i);

Functional Decomposition

int e;
main ()
{
  int x[10], j, k, m;  j = f(x, k);  m = g(x, k);
}
int f(int *x, int k)
{
  int a;  a = e * x[k] * x[k];  return a;
}
int g(int *x, int k)
{
  int a;  a = k - 1;  a = e / x[k];  return a;
}

Voluntary int e;
Static variable: Shared

main ()
{
  int x[10], j, k, m;  j = f(x, k);  m = g(x, k);
}
int f(int *, int k)
{
  int a;  a = e * x[k] * x[k];  return a;
}
int g(int *, int k)
{
  int a;  a = k - 1;  a = e / x[k];  return a;
}
### Functional Decomposition

```c
volatile int e;

main () {
    int x[10], j, k, m;
    j = f(x, k);  m = g(x, k);
}

int f(int *x, int k) {
    int a;
    a = e * x[k] * x[k];
    return a;
}

int g(int *x, int k) {
    int a;
    k = k-1;
    a = e / x[k];
    return a;
}
```

### Shared and Private Variables

- **Shared variables**
  - Static variables
  - Heap variables
  - Contents of run-time stack at time of call
- **Private variables**
  - Loop index variables
  - Run-time stack of functions invoked by thread

### The Shared-Memory Model

- CPU
  - Private Memory
- Shared Memory

### The Threads Model

- Thread
  - Private Variables
  - Shared Variables