159.341 Programming Languages, Algorithms & Concurrency

Concurrent Programming - Parallel Programming

Daniel Playne
d.p.playne@massey.ac.nz

Parallel Programming

The final step in developing a concurrent/parallel program is to actually write it.

- Hardware Classification
- Types of Parallelism
- Algorithm Design
- Paradigm
- ***Program***

Parallel Programming

There are a great many languages, libraries and APIs that can be used to develop parallel or concurrent programs.

Each of these generally fall into one of two categories:

- ***Implicit Parallelism*** - the programmer does not explicitly specify the parallelism but allows a compiler/run-time to exploit it.
- ***Explicit Parallelism*** - source code explicitly defines how to parallelise the computation using constructs/directives etc

Implicit Parallelism

Implicit parallelism is exploited by a compiler/interpreter/run-time to automatically identify parallelism within a program and exploit it.

Languages that support *implicit parallelism* do not require specific directives, operators or functions to specify operations that can be computed in parallel.

However, such languages often use some unusual constructs to allow the compiler to detect opportunities for parallelism.

Automatic Parallelism

The *holy grail* of implicit parallelism is an automatically parallelising compiler that can take an arbitrary program and produce an efficient parallel executable.

However, despite decades of work there has been limited progress towards this achieving this goal.

There are some major challenges involved in developing an automatically parallelising compiler.

Automatic Parallelism

The first is detecting dependencies between instructions, whenever the result of on instruction depends on the output of another then there is a dependency between them.

```
1.   A := B + C
2.   D := C + D
3.   E := C + 1
4.   F := A + D
5.   C := G + B
```

Instructions 1, 2 and 3 could be performed in parallel as their result does not depend on the result of the others.

However, 4 must be executed after 1 and 2 to produce the correct result, likewise 5 must be executed after 2 and 3 to ensure their results are not affected.

Automatic Parallelism

Detecting dependencies between instructions and identifying instructions that can be executed efficiently in parallel is a hard problem.

An even harder problem is to consider that many sequential algorithms do not parallelise well and may require a different algorithm more suited to parallelism.

Developing a compiler that can detect the use of such algorithms and substitute them for parallel alternatives is troublesome to say the least.

Explicit Parallelism

For this reason, most parallel programs are written using ***explicit parallelism*** where the programmer is responsible for identifying parallelism in their program.

These languages/libraries/APIs are often developed for certain target architecture:

- Shared Memory
- Distributed Memory

Threading Libraries

We have already looked at two libraries for writing parallel programs - POSIX Threads and Windows API threads.

These are both an example of an ***explicit threading*** library where the programmer exploits parallelism by *explicitly* creating, destroying and controlling threads.

Communication between threads can be performed through access to shared memory and requires explicit synchronisation.

Threading Libraries

Some languages define threading as part of the language.

- Java
- Python
- C++ (since C++11)
- etc.

Generally more portable than using libraries that may or may not be supported on certain operating systems.

Threading Libraries

Explicit threading libraries are a fairly *low-level* approach to multi-threaded programming.

Exposing the lowest level operations gives the programmer more control over the execution of the threads.

The downside of course is that the programmer has to correctly program all these details.

Threading Libraries

Explicit threading libraries can provide very good performance when implemented properly.

They also come with the risk that the programmer either makes mistakes or certain features become too difficult to develop.

e.g. load-balancing gets left out because it became too much work to write.

OpenMP

An alternative to an explicit threading library is to use an API such as OpenMP (the MP stands for *multiprocessing*).

In OpenMP the programmer does not explicitly create threads and specify their behaviour.

Instead the programmer can use code directives to specify blocks of code that can be run in parallel.

It is up to the compiler and the run-time system to decide exactly how these blocks are executed by different threads.

OpenMP

Blocks of code can be marked as suitable for computing in parallel using `#pragma` directives such as the following.

```
#pragma parallel for
for(int i = 0; i < N; ++i) {
   A[i] = B[i] + C[i];
}
```

This directive denotes that the `for` loop can be computed in parallel by multiple threads.

OpenMP

There are some advantages to the approach used by OpenMP.

OpenMP was designed to allow programmers to *incrementally* parallelise their code. It is relatively simple to add directives to certain sections of an existing program.

Whereas using an explicit threading library may require the significant parts of the code to be rewritten (or the entire thing).

OpenMP

OpenMP programs still define a correct sequential program.

While it may not be the most efficient implementation, the directives an OpenMP program can be ignored and still yield a correct program.

Programs written with explicit threading require threads to be supported.

OpenMP

The downside of this approach is that the programmer must defer some decisions on thread behaviour to the compiler / run-time system.

This may have a performance implications and also means that it can be more difficult to specify low-level thread interactions.

Explicit threading libraries are generally more work but also provide the flexibility to specify whatever thread behaviour we want.

Threading Building Blocks

Another high-level approach to developing multi-threaded programs is to use a library such as Threading Building Blocks or TBB.

This library provides programmers with a number of high-level *algorithms* that specify parallel tasks.

- `parallel_for`
- `parallel_reduce`
- `parallel_pipeline`
- `parallel_sort`

Threading Building Blocks

TBB uses a *task-based* programming approach where parallel programs are specified in terms of tasks rather than threads.

Defining programs in terms of tasks has some advantages:

- Better matching of parallelism to available resources
- Faster task startup/shutdown
- Improved load-balancing

Threading Building Blocks

TBB has a *task scheduler* which maps tasks to actual worker threads.

Executing tasks in this manner allows load-balancing to be implemented and helps avoid *over*/*undersubscription* of resources.

However, it does rely on a well-implemented *task scheduler* (in some use-cases/architectures the TBB scheduler has been shown to account for over 40% of the runtime).

Threading Building Blocks

Like OpenMP, Threading Building Blocks allows programmers to express tasks that may be executed in parallel.

Depending on the exact details of other tasks, threads and the scheduler two parallel tasks may actually be executed by the same thread sequentially.

Java Concurrency

Java has provided support for concurrency since version 5.0.

There are two main ways of implementing concurrency in Java:

- Direct thread control and management
- Using an *executor* to manage threads and execute tasks.

Java Concurrency

Java provides a `Runnable` interface with a function called `run()` which can be used by any class that executes in a different thread.

```
public class MyRunnable implements Runnable {
   public void run() {
      ...
   }
}
```

Java Concurrency

A `Runnable` object can either be executed by directly creating a new thread or submitted to an `Executor`.

```
// Runnable Object
Runnable r = new MyRunnable(...);

// Explicitly create new thread
(new Thread(r)).start();

// Use Executor
Executor e = new ...
e.execute(r);
```

The `Executor` object may create a new thread to execute `r`, it may use an existing worker thread or even execute it all in the calling thread.

Synchronisation Primitives

Multi-threading libraries/APIs will provide some form of synchronisation primitives that can be used to synchronise threads and protect critical sections.

These primitives may be as simple as `mutexes` and `semaphores` or more complex or higher-level constructs.

We'll look at these in greater detail when we look at the different APIs.

Data Structures

Similar to synchronisation primitives, multi-threading libraries may provide thread-safe data structures.

These are data-structures that are designed to be accessed by multiple threads without being internally corrupted.

Incorrect access to a non thread-safe data structure may result in incorrect results or completely corrupt the data structure.

Summary

- Explicit/Implicit Parallelism
- Explicit Threading Libraries
- Parallelism Exposing Libraries
- Synchronisation Primitives
- Concurrent Data-Structures