Concurrency Control

Monitors

The next approach to parallelism we shall look at is to have constructs as part of the language

Concurrency Control

Monitors

The next approach to parallelism we shall look at is to have constructs as part of the language

For example, a *monitor* is a language construct that combines mutual exclusion and synchronisation in a way that can be easier to use than the concurrency primitives

Concurrency Control

Monitors

The next approach to parallelism we shall look at is to have constructs as part of the language

For example, a *monitor* is a language construct that combines mutual exclusion and synchronisation in a way that can be easier to use than the concurrency primitives

```
monitor Name
   local variable declarations
   func fun1(args)  body
   func fun2(args)  body
   ...
end
```

Concurrency Control

Monitors

The next approach to parallelism we shall look at is to have constructs as part of the language

For example, a *monitor* is a language construct that combines mutual exclusion and synchronisation in a way that can be easier to use than the concurrency primitives

```
monitor Name
    local variable declarations
    func fun1(args)  body
    func fun2(args)  body
    ...
end
```

The actual syntax will vary by language

Concurrency Control

Monitors

Mutual exclusion is enforced by

> *only one thread at a time may be executing any function inside a given monitor*

Concurrency Control

Monitors

Mutual exclusion is enforced by

> *only one thread at a time may be executing any function inside a given monitor*

So, if one thread is executing `fun1` and another thread tries to execute `fun2`, it will have to wait until the first thread exits the monitor

Concurrency Control

Monitors

So there is mutual exclusion on the local variables and within the dynamic scope of the functions in the monitor, i.e., mutual exclusion continues even if `fun1` calls a function defined outside the monitor

Concurrency Control

Monitors

So there is mutual exclusion on the local variables and within the dynamic scope of the functions in the monitor, i.e., mutual exclusion continues even if `fun1` calls a function defined outside the monitor

The mutual exclusion finishes when the thread of control exits the (top level) monitor function

Concurrency Control

Monitors

So there is mutual exclusion on the local variables and within the dynamic scope of the functions in the monitor, i.e., mutual exclusion continues even if `fun1` calls a function defined outside the monitor

The mutual exclusion finishes when the thread of control exits the (top level) monitor function

Clearly, monitors will be implemented with locks, but this conveniently hidden from the programmer using them

Concurrency Control

Monitors

Synchronisation is provided by the use of condition variables

Concurrency Control

Monitors

Synchronisation is provided by the use of condition variables

`wait(c);` and `signal(c);`

Concurrency Control

Monitors

Synchronisation is provided by the use of condition variables

`wait(c);` and `signal(c);`

The associated lock is the monitor mutual exclusion lock, and is implicit

Concurrency Control

Monitors

Synchronisation is provided by the use of condition variables

`wait(c);` and `signal(c);`

The associated lock is the monitor mutual exclusion lock, and is implicit

Just like the POSIX version, `wait()` will drop the monitor lock to allow other threads access; and try to regain it when it resumes

Concurrency Control

Monitors

We can easily implement a lock using a monitor:

```
monitor Lock
  int flag = 0;
  condition c;
  lock() { while (flag == 1) wait(c); flag = 1; }
  unlock() { flag = 0; signal(c); }
end
```

Concurrency Control

Monitors

We can easily implement a lock using a monitor:

```
monitor Lock
  int flag = 0;
  condition c;
  lock() { while (flag == 1) wait(c); flag = 1; }
  unlock() { flag = 0; signal(c); }
end
```

The monitor lock provides the atomicity we need in the definition of `lock`

Concurrency Control

Monitors

Monitors help with management of mutual exclusion, but the usual nesting deadlock is still possible. For monitors `m1` and `m2`:

```
monitor m1
  fun1() { ... fun2() ...}
  ...
end
```

```
monitor m2
  fun2() { ... fun1() ... }
  ...
end
```

1

`fun1` in monitor `m1` calls
`fun2` in monitor `m2` (waits)

2

`fun2` in monitor `m2` calls
`fun1` in monitor `m1` (waits)

Concurrency Control

Monitors

Modularity might even encourage this error, though monitors are high enough level to be easy to analyse automatically so there are source code tools to spot this

Concurrency Control

Monitors

Modularity might even encourage this error, though monitors are high enough level to be easy to analyse automatically so there are source code tools to spot this

They require careful use and are not a universal solution!

Concurrency Control

Java Monitors

Monitors clearly fit well with object oriented languages: for example, Java implements monitors on a per-object level:

```
class foo {
  private int n = 0;
  public synchronized int inc() { n++; }
  public synchronized int dec() { n--; }
  ...
}
```

Methods with the `synchronized` keyword are within a per-object monitor, i.e., one per instance of `foo`

Concurrency Control

Java Monitors

Only one of `inc` and `dec` can be executing on a given instance of `foo` at a time

Concurrency Control

Java Monitors

Only one of `inc` and `dec` can be executing on a given instance of `foo` at a time

Condition variables: `wait()`, `notify()` and `notifyAll()`

Concurrency Control

Java Monitors

Only one of `inc` and `dec` can be executing on a given instance of `foo` at a time

Condition variables: `wait()`, `notify()` and `notifyAll()`

Class methods (`static`) can be synchronised, too, locking the class but not its instances

Concurrency Control

Monitors

Monitors are fairly easy to use, but are somewhat large grained: the whole of each monitor, for example *all* methods marked `synchronized` in a Java object

```
class foo {
  private int n = 0, m = 0;
  public synchronized int incn() { n++; }
  public synchronized int decn() { n--; }
  public synchronized int incm() { m++; }
  public synchronized int decm() { m--; }
}
```

Concurrency Control

Monitors

To have separate locks on some of the methods requires code refactoring (or see below): You can do this, but this is driving the code towards complexity

Concurrency Control

Monitors

To have separate locks on some of the methods requires code refactoring (or see below): You can do this, but this is driving the code towards complexity

Similarly, it is a bit fiddly to decide on what functionality goes into which monitor: if you are not careful you end up with all your code in one big monitor—sequential!

Concurrency Control

Monitors

Exercise What about the following?

```
class foo {
  private int n = 0, m = 0;
  public synchronized int incn() { n++; }
  public synchronized int decn() { n--; }
  public synchronized int incm() { m++; }
  public synchronized int decm() { m--; }
  public synchronized int swap() { int s = m; m = n; n = s; }
}
```

Concurrency Control

Java Monitors

Java recognises that monitors are sometimes too large, so it allows synchronising of *statements* (rather than whole methods) as a way of providing finer gain control

```
public class locket {
  private Object nlock = new Object();
  private int n = 0;
  public void inc() {
    synchronized(nlock) { n++; }
  }
  public void dec() {
    synchronized(nlock) { n--; }
  }
}
```

Concurrency Control

Java Monitors

`synchronized` takes an arbitrary object as argument

Concurrency Control

Java Monitors

`synchronized` takes an arbitrary object as argument

A class can have as many of these as it likes in addition to the implicit one provided by the class monitor

Concurrency Control

Java Monitors

`synchronized` takes an arbitrary object as argument

A class can have as many of these as it likes in addition to the implicit one provided by the class monitor

This is fine, but we have just reinvented mutexes!

Concurrency Control

Java Monitors

`synchronized` takes an arbitrary object as argument

A class can have as many of these as it likes in addition to the implicit one provided by the class monitor

This is fine, but we have just reinvented mutexes!

But in a more convenient form: you can't forget to lock or unlock these

Concurrency Control

Java Monitors

Incidentally, Java also has a library of *atomic* datatypes, e.g., `AtomicInteger` with a few methods, that does the obvious thing

Concurrency Control

Java Monitors

Incidentally, Java also has a library of *atomic* datatypes, e.g., `AtomicInteger` with a few methods, that does the obvious thing

But these are tiresome to use as Java does not have operator overloading, like C++: thus `n.incrementAndGet()` rather than overloading ++ and using the simpler `++n`

Concurrency Control

Conditional Critical Regions

Exercise A similar, but simpler, kind of idea is *conditional critical regions*, where a semaphore is associated with blocks of code (the critical regions)

```
let s = Semaphore::new(1);
...
region s {
  // critical region
  ...
  await <some condition>
  ...
}
```

```
region s {
   ...
   <set condition>
   ...
}
```

Read about this (e.g., in Ada).

Parallelism Languages

The logical approach to parallel programming is to use a language that was designed from the start to support parallelism

Parallelism Languages

The logical approach to parallel programming is to use a language that was designed from the start to support parallelism

There have been very many attempts at creating new languages with explicit support for parallelism

Parallelism Languages

The logical approach to parallel programming is to use a language that was designed from the start to support parallelism

There have been very many attempts at creating new languages with explicit support for parallelism

For example, Occam, Strand, Erlang, Linda, SALSA, SISAL, Parlog, Charm++, NESL, Go, Rust as just a few from a huge list

Parallelism Languages

The logical approach to parallel programming is to use a language that was designed from the start to support parallelism

There have been very many attempts at creating new languages with explicit support for parallelism

For example, Occam, Strand, Erlang, Linda, SALSA, SISAL, Parlog, Charm++, NESL, Go, Rust as just a few from a huge list

We should have time to look at one or more of these towards the end of the Unit

Parallelism Languages

The logical approach to parallel programming is to use a language that was designed from the start to support parallelism

There have been very many attempts at creating new languages with explicit support for parallelism

For example, Occam, Strand, Erlang, Linda, SALSA, SISAL, Parlog, Charm++, NESL, Go, Rust as just a few from a huge list

We should have time to look at one or more of these towards the end of the Unit

Some of these languages are quite difficult to learn and use effectively

Language Modification

A conservative approach to getting these kinds of parallel support is to take an existing language, like C, and tweak *the language* to add parallelism

Language Modification

A conservative approach to getting these kinds of parallel support is to take an existing language, like C, and tweak *the language* to add parallelism

Then, so the theory goes, you can tap into the existing expertise in that language and extend it to parallel systems

Language Modification

A conservative approach to getting these kinds of parallel support is to take an existing language, like C, and tweak *the language* to add parallelism

Then, so the theory goes, you can tap into the existing expertise in that language and extend it to parallel systems

This is true to a certain extent, but it still tries to layer parallel ideas over a sequential foundation

Language Modification

A conservative approach to getting these kinds of parallel support is to take an existing language, like C, and tweak *the language* to add parallelism

Then, so the theory goes, you can tap into the existing expertise in that language and extend it to parallel systems

This is true to a certain extent, but it still tries to layer parallel ideas over a sequential foundation

Parallelism should not be an afterthought, but should really be part of the foundation

Language Modification

The main example we shall be looking at is *OpenMP* (Open MultiProcessing)

Language Modification

The main example we shall be looking at is *OpenMP* (Open MultiProcessing)

This takes C (or C++) and add some new constructs to notate parallel execution

Language Modification

The main example we shall be looking at is *OpenMP* (Open MultiProcessing)

This takes C (or C++) and add some new constructs to notate parallel execution

By hiding the low-level primitive locking and synchronisation they aim to provide an easier way of writing parallel programs

Language Modification

The main example we shall be looking at is *OpenMP* (Open MultiProcessing)

This takes C (or C++) and add some new constructs to notate parallel execution

By hiding the low-level primitive locking and synchronisation they aim to provide an easier way of writing parallel programs

And minimise the kinds of errors the primitives invoke

OpenMP

OpenMP fits nicely into the superstep model of computation

OpenMP

OpenMP fits nicely into the superstep model of computation

While you shall *not* be using OpenMP for the coursework, some of you might want to use it for your FY Project

OpenMP

Here is a simple loop

```
for (i = 0; i < 10; i++) {
  sq[i] = n + i*i;
}
```

OpenMP

Here is a simple loop

```
for (i = 0; i < 10; i++) {
  sq[i] = n + i*i;
}
```

With OpenMP annotation

OpenMP

Here is a simple loop

```
for (i = 0; i < 10; i++) {
  sq[i] = n + i*i;
}
```

With OpenMP annotation

```
#pragma omp parallel for
for (i = 0; i < 10; i++) {
  sq[i] = n + i*i;
}
```

OpenMP

Here is a simple loop

```
for (i = 0; i < 10; i++) {
  sq[i] = n + i*i;
}
```

With OpenMP annotation

```
#pragma omp parallel for
for (i = 0; i < 10; i++) {
  sq[i] = n + i*i;
}
```

The `#pragma omp` indicates that we want the loop to be run in parallel

OpenMP

Here is a simple loop

```
for (i = 0; i < 10; i++) {
  sq[i] = n + i*i;
}
```

With OpenMP annotation

```
#pragma omp parallel for
for (i = 0; i < 10; i++) {
  sq[i] = n + i*i;
}
```

The `#pragma omp` indicates that we want the loop to be run in parallel

`#pragma` is a general C mechanism, not limited to OpenMP

OpenMP

When this is run, the loop is split into some number of chunks, running on some number of threads

OpenMP

When this is run, the loop is split into some number of chunks, running on some number of threads

The OpenMP runtime system determines the number of chunks and number of threads

OpenMP

When this is run, the loop is split into some number of chunks, running on some number of threads

The OpenMP runtime system determines the number of chunks and number of threads

That is, it makes a choice when the code is run

OpenMP

When this is run, the loop is split into some number of chunks, running on some number of threads

The OpenMP runtime system determines the number of chunks and number of threads

That is, it makes a choice when the code is run

And the numbers of chunks and threads may differ on different runs

OpenMP

Typically the number of chunks is the same as the number of threads, which is the same as the number of processors in the system, but it need not be

OpenMP

Typically the number of chunks is the same as the number of threads, which is the same as the number of processors in the system, but it need not be

And each chunk typically iterates close to

$$\frac{\text{size of loop}}{\text{number of chunks}}$$

times

OpenMP

Also important is that the runtime creates parallel code with a *private* version of `i` per thread

OpenMP

Also important is that the runtime creates parallel code with a *private* version of `i` per thread

Each thread wants its `i` to range, in parallel, over different values, e.g., 0–2, 3–5, 6–8, 9

OpenMP

Also important is that the runtime creates parallel code with a *private* version of `i` per thread

Each thread wants its `i` to range, in parallel, over different values, e.g., 0–2, 3–5, 6–8, 9

Or maybe 0–2, 3–5, 6–7, 8–9; or something else

OpenMP

Also important is that the runtime creates parallel code with a *private* version of `i` per thread

Each thread wants its `i` to range, in parallel, over different values, e.g., 0–2, 3–5, 6–8, 9

Or maybe 0–2, 3–5, 6–7, 8–9; or something else

The runtime decides, and potentially might choose a different split in different runs

OpenMP

Also important is that the runtime creates parallel code with a *private* version of `i` per thread

Each thread wants its `i` to range, in parallel, over different values, e.g., 0–2, 3–5, 6–8, 9

Or maybe 0–2, 3–5, 6–7, 8–9; or something else

The runtime decides, and potentially might choose a different split in different runs

The `parallel for` construct knows the loop variable must be private

OpenMP

Also important is that the runtime creates parallel code with a *private* version of `i` per thread

Each thread wants its `i` to range, in parallel, over different values, e.g., 0–2, 3–5, 6–8, 9

Or maybe 0–2, 3–5, 6–7, 8–9; or something else

The runtime decides, and potentially might choose a different split in different runs

The `parallel for` construct knows the loop variable must be private

But the variables `n` and `sq` are *shared* across the threads

OpenMP

Note:

OpenMP

Note:

- we do not give a number of threads

OpenMP

Note:

- we do not give a number of threads
- the creation and destruction of threads is all hidden from us: it may create and destroy threads on each occurrence of a `#pragma omp`; or it may use a thread pool

OpenMP

Note:

- we do not give a number of threads
- the creation and destruction of threads is all hidden from us: it may create and destroy threads on each occurrence of a `#pragma omp`; or it may use a thread pool
- the compiler determines we need a per-thread variable `i`

OpenMP

Note:

- we do not give a number of threads
- the creation and destruction of threads is all hidden from us: it may create and destroy threads on each occurrence of a `#pragma omp`; or it may use a thread pool
- the compiler determines we need a per-thread variable `i`
- by using the construct we are assuring the compiler that it *is* safe to do the loop in parallel and there are no data (or other) races.
  If the loop was
  ```
  av[i] = av[i] + av[i-1];
  ```
  it would blindly do this in parallel

OpenMP

Note:

- we do not give a number of threads
- the creation and destruction of threads is all hidden from us: it may create and destroy threads on each occurrence of a `#pragma omp`; or it may use a thread pool
- the compiler determines we need a per-thread variable `i`
- by using the construct we are assuring the compiler that it *is* safe to do the loop in parallel and there are no data (or other) races.
  If the loop was
  ```
  av[i] = av[i] + av[i-1];
  ```
  it would blindly do this in parallel
- so OpenMP provides a simple *mechanism*, but no *analysis*

OpenMP

Exercise Convince yourself why the following is wrong:

Convert

```
for (i = 0; i < 10; i++) {
  av[i] = av[i] + av[i-1];
}
```

to

```
#pragma omp parallel for
for (i = 0; i < 10; i++) {
  av[i] = av[i] + av[i-1];
}
```

OpenMP

Another example:

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[])
{
#pragma omp parallel
  printf("Hello world, I am thread %d\n",
         omp_get_thread_num());
  return 0;
}
```

OpenMP

Another example:

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[])
{
#pragma omp parallel
  printf("Hello world, I am thread %d\n",
         omp_get_thread_num());
  return 0;
}
```

Guesses for the output?

OpenMP

Running on an 8 core machine:

```
Hello world, I am thread 0
Hello world, I am thread 6
Hello world, I am thread 5
Hello world, I am thread 4
Hello world, I am thread 3
Hello world, I am thread 1
Hello world, I am thread 7
Hello world, I am thread 2
```

OpenMP

Note:

OpenMP

Note:

- the `printf`s are in no particular order; running the same code again gives a different order output

OpenMP

Note:

- the `printf`s are in no particular order; running the same code again gives a different order output
- the `printf`s are separate, the outputs are not mixed. This is because this implementation has internal locks on output streams

OpenMP

Note:

- the `printfs` are in no particular order; running the same code again gives a different order output
- the `printfs` are separate, the outputs are not mixed. This is because this implementation has internal locks on output streams
- We see all of the `printfs`: OpenMP has an implicit barrier at the end of each construct (superstep). This means the main thread (or rather, the `pragma parallel`) waits for all threads to finish before moving on and executing the next line (`return` in this example)

OpenMP

There are several OpenMP pragmas

OpenMP

There are several OpenMP pragmas

```
#pragma omp parallel for
for (...) { }
```

OpenMP

There are several OpenMP pragmas

```
#pragma omp parallel for
for (...) { }
```

The loop variable is made private per-thread; by default all other variables are shared between the threads

OpenMP

```
#pragma omp parallel sections
{
#pragma omp section
  {
    printf("Hello world, I am thread %d\n",
           omp_get_thread_num());
  }
#pragma omp section
  {
    printf("hi there, I am thread %d\n",
           omp_get_thread_num());
  }
}
```

This executes on (maybe) just two threads, one thread per section

OpenMP

The sections need not contain similar code

OpenMP

The sections need not contain similar code

Exercise But ideally should contain codes that take roughly the same time to execute. Why?

OpenMP

```
#pragma omp parallel
{
#pragma omp for
#pragma omp sections
#pragma omp barrier
#pragma omp masked
#pragma omp critical
...
}
```

A general parallel section that contains more specific ways of parallelising

OpenMP

`barrier` is an explicit barrier

OpenMP

`barrier` is an explicit barrier

`masked` marks code that will only be executed by threads that match the mask

OpenMP

`barrier` is an explicit barrier

`masked` marks code that will only be executed by threads that match the mask

`critical` marks a critical region that will be executed by exactly one thread at a time (a monitor or mutex)

OpenMP

```
#include <stdio.h>

int count = 0;

void inc() {
#pragma omp critical
  count++;
}

int main(int argc, char* argv[])
{
#pragma omp parallel
  inc();

  printf("count = %d\n", count);
  return 0;
}
```

Prints the number of threads (bad code!)

OpenMP

Each parallel pragma can take extra arguments for fine control:

```
#pragma omp parallel for [shared(vars), private(vars),
firstprivate(vars),lastprivate(vars),
default(shared|none), reduction(op:vars), copyin(vars),
if(expr), ordered, schedule(type[,chunkSize])]
```

OpenMP

Each parallel pragma can take extra arguments for fine control:

```
#pragma omp parallel for [shared(vars), private(vars),
firstprivate(vars),lastprivate(vars),
default(shared|none), reduction(op:vars), copyin(vars),
if(expr), ordered, schedule(type[,chunkSize])]
```

- `shared` a list of variables that are shared between the threads (default: all variables except the loop variable)

OpenMP

Each parallel pragma can take extra arguments for fine control:

```
#pragma omp parallel for [shared(vars), private(vars),
firstprivate(vars),lastprivate(vars),
default(shared|none), reduction(op:vars), copyin(vars),
if(expr), ordered, schedule(type[,chunkSize])]
```

- `shared` a list of variables that are shared between the threads (default: all variables except the loop variable)
- `private` a list of variables that are private to each thread; default for a loop variable

OpenMP

Each parallel pragma can take extra arguments for fine control:

```
#pragma omp parallel for [shared(vars), private(vars),
firstprivate(vars),lastprivate(vars),
default(shared|none), reduction(op:vars), copyin(vars),
if(expr), ordered, schedule(type[,chunkSize])]
```

- `shared` a list of variables that are shared between the threads (default: all variables except the loop variable)
- `private` a list of variables that are private to each thread; default for a loop variable
- `nowait` remove the implicit barrier at the end of the section

OpenMP

Each parallel pragma can take extra arguments for fine control:

```
#pragma omp parallel for [shared(vars), private(vars),
firstprivate(vars),lastprivate(vars),
default(shared|none), reduction(op:vars), copyin(vars),
if(expr), ordered, schedule(type[,chunkSize])]
```

- `shared` a list of variables that are shared between the threads (default: all variables except the loop variable)
- `private` a list of variables that are private to each thread; default for a loop variable
- `nowait` remove the implicit barrier at the end of the section
- `reduction(op:vars)` private variables that are *reduced* using the `op` at the end

OpenMP

```
int i;
#pragma omp parallel reduction(+:i)
  i = omp_get_thread_num();
printf("i = %d\n", i);
```

Each thread gets its own private `i`; at the end of the section all copies are reduced to the single value of `i` by +

OpenMP

```
int i;
#pragma omp parallel reduction(+:i)
  i = omp_get_thread_num();
printf("i = %d\n", i);
```

Each thread gets its own private `i`; at the end of the section all copies are reduced to the single value of `i` by +

So, maybe, $0 + 6 + 5 + 4 + 3 + 1 + 7 + 2 = 28$

OpenMP

```
int i;
#pragma omp parallel reduction(+:i)
  i = omp_get_thread_num();
printf("i = %d\n", i);
```

Each thread gets its own private `i`; at the end of the section all copies are reduced to the single value of `i` by +

So, maybe, $0 + 6 + 5 + 4 + 3 + 1 + 7 + 2 = 28$

Reductions turn out to be commonly needed in parallel programs