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.
A more conservative approach is to take an existing language, like C, and tweak *the language* to add parallelism.
A more conservative approach 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 more conservative approach 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 does try to layer parallel ideas over a sequential foundation.
A more conservative approach 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 does try to layer parallel ideas over a sequential foundation.

Parallelism should not be an afterthought, but should really be part of the foundation.
The main examples we shall be looking at are OpenMP (Open MultiProcessing) and (briefly) Cilk Plus
The main examples we shall be looking at are *OpenMP* (Open MultiProcessing) and (briefly) *Cilk Plus*

These take C (or C++) and add some new constructs to notate parallel execution
The main examples we shall be looking at are *OpenMP* (Open MultiProcessing) and (briefly) *Cilk Plus*

These take 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 examples we shall be looking at are *OpenMP* (Open MultiProcessing) and (briefly) *Cilk Plus*

These take 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
Firstly OpenMP

This 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.
Firstly OpenMP

This fits nicely into the superstep model of computation
Firstly OpenMP

This 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
Here is a simple loop

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

Here is a simple loop

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

With OpenMP annotation

```c
#pragma omp parallel for
for (i = 0; i < 10; i++) {
    sq[i] = n + i*i;
}
```
Here is a simple loop

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

With OpenMP annotation

```c
#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

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

With OpenMP annotation

```c
#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.
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.

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

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.

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

\[
\frac{\text{size of loop}}{\text{number of chunks}} \text{ times}
\]
Also important is that the runtime creates parallel code with a *private* version of \( i \) per thread.
Also important is that the runtime creates parallel code with a \textit{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.
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
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 *parallel for* construct knows the loop variable must be *private*
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 parallel for construct knows the loop variable must be private

But the variable $n$ is shared across the threads
OpenMP

Note:

• we do not give a number of threads
• the creation and destruction of threads is all hidden from us
• 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
\[ \text{av}[i] = \text{av}[i] + \text{av}[i-1]; \]

it would blindly do this in parallel
• so OpenMP provides a simple mechanism, but no analysis
OpenMP

Note:

- we do not give a number of threads
Note:

- we do not give a number of threads
- the creation and destruction of threads is all hidden from us
Note:

- we do not give a number of threads
- the creation and destruction of threads is all hidden from us
- the compiler determines we need a per-thread variable $i$
Note:

- we do not give a number of threads
- the creation and destruction of threads is all hidden from us
- 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

```c
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
- 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
  \[ \text{av}[i] = \text{av}[i] + \text{av}[i-1]; \]
  it would blindly do this in parallel
- so OpenMP provides a simple *mechanism*, but no *analysis*
```c
#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;
}
```
```c
#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?
```
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
Note:

- The `printf` statements are in no particular order; running the same code again gives a different output.
- The `printf` statements are separate, the outputs are not mixed. This is because this implementation of `printf` has an internal lock.
- We see all of the `printf` statements: 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 (e.g., `return` in this example).
OpenMP

Note:

- the printf statements are in no particular order; running the same code again gives a different output.
Note:

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

- The `printf`s are in no particular order; running the same code again gives a different output.
- The `printf`s are separate, the outputs are not mixed. This is because this implementation of `printf` has an internal lock.
- We see all of the `printf`s: 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
There are several OpenMP pragmas

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

The loop variable is private; by default all other variables are shared between the threads
#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
The sections need not contain similar code
The sections need not contain similar code

Exercise. But ideally should contain codes that take roughly the same time to execute. Why?
#pragma omp parallel
{
    #pragma omp for
    #pragma omp sections
    #pragma omp barrier
    #pragma omp master
    #pragma omp critical
    ...
}

A general parallel section that contains more specific ways of parallelising
barrier is an explicit barrier
barrier is an explicit barrier

master marks code that will be executed exactly once
OpenMP

barrier is an explicit barrier

master marks code that will be executed exactly once

critical marks a critical region that will be executed by exactly one thread at a time (a monitor or mutex)
```c
#include <stdio.h>

int count = 0;

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

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

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

Prints the number of threads
```
OpenMP

Each parallel pragma can take extra arguments for fine control:

```c
#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:

```c
#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:

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

Each thread gets its own private \textit{i}; at the end of the section all copies are reduced to the single value of \textit{i} by +
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, $0 + 6 + 5 + 4 + 3 + 1 + 7 + 2 = 28$
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
OpenMP

There are several useful functions

- `int omp_get_num_threads(void)` returns the number of threads in this parallel region.
- `int omp_get_thread_num(void)` returns a per-thread unique number.
- `int omp_get_max_threads(void)` returns the maximum number of threads available (often defaults to the number of cores).
- `void omp_set_num_threads(int)` sets the number of threads OpenMP can use.
- `int omp_get_num_procs(void)` returns the number of processors in this system.
OpenMP

There are several useful functions

- `int omp_get_num_threads(void)` returns the number of threads in this parallel region
- `int omp_get_thread_num(void)` returns a per-thread unique number
- `int omp_get_max_threads(void)` the maximum number of threads available (often defaults to the number of cores)
- `void omp_set_num_threads(int)` set the number of threads OpenMP can use
- `int omp_get_num_procs(void)` number of processors in this system
There are several useful functions

- `int omp_get_num_threads(void)` returns the number of threads in this parallel region
- `int omp_get_thread_num(void)` returns a per-thread unique number
OpenMP

There are several useful functions

- `int omp_get_num_threads(void)` returns the number of threads in this parallel region
- `int omp_get_thread_num(void)` returns a per-thread unique number
- `int omp_get_max_threads(void)` the maximum number of threads available (often defaults to the number of cores)
There are several useful functions

- `int omp_get_num_threads(void)` returns the number of threads in this parallel region
- `int omp_get_thread_num(void)` returns a per-thread unique number
- `int omp_get_max_threads(void)` the maximum number of threads available (often defaults to the number of cores)
- `void omp_set_num_threads(int)` set the number of threads OpenMP can use
There are several useful functions

- `int omp_get_num_threads(void)` returns the number of threads in this parallel region
- `int omp_get_thread_num(void)` returns a per-thread unique number
- `int omp_get_max_threads(void)` the maximum number of threads available (often defaults to the number of cores)
- `void omp_set_num_threads(int)` set the number of threads OpenMP can use
- `int omp_get_num_procs(void)` number of processors in this system
OpenMP

And lots more functionality
And lots more functionality

For example, setting the environment variable
OMP_NUM_THREADS before running the program sets the default
number of threads
And lots more functionality

For example, setting the environment variable `OMP_NUM_THREADS` before running the program sets the default number of threads

```
OMP_NUM_THREADS=7 ./prog
```
OpenMP

And lots more functionality

For example, setting the environment variable
OMP_NUM_THREADS before running the program sets the default number of threads

OMP_NUM_THREADS=7 ./prog

OpenMP is widely supported. For example, to compile under GCC:
cc -fopenmp -Wall -o prog prog.c
OpenMP is clearly naturally associated with shared memory
OpenMP is clearly naturally associated with shared memory.

There is a distributed memory version from Intel, called *Cluster OpenMP*. 
OpenMP is clearly naturally associated with shared memory.

There is a distributed memory version from Intel, called *Cluster OpenMP*.

There is an undercurrent of “if your program doesn’t work well on normal OpenMP, then it won’t work well on Cluster OpenMP.”
OpenMP is an evolving standard that is easy to use; you can modify existing programs incrementally. It hides messy threads fiddling. Unlike pthreads, it needs compiler support. If you pass control of the parallelism to a compiler, you need that compiler to be good at it. It still allows trivially buggy programs.
OpenMP

- is an evolving standard
OpenMP

- is an evolving standard
- is easy to use; you can modify existing programs incrementally
OpenMP

- is an evolving standard
- is easy to use; you can modify existing programs incrementally
- hides messy threads fiddling
OpenMP

- is an evolving standard
- is easy to use; you can modify existing programs incrementally
- hides messy threads fiddling
- needs compiler support, unlike pthreads
OpenMP

- is an evolving standard
- is easy to use; you can modify existing programs incrementally
- hides messy threads fiddling
- needs compiler support, unlike pthreads
- is dependent on good implementation of the compiler: if you pass control of the parallelism to a compiler you need that compiler to be good at it
OpenMP

• is an evolving standard
• is easy to use; you can modify existing programs incrementally
• hides messy threads fiddling
• needs compiler support, unlike pthreads
• is dependent on good implementation of the compiler: if you pass control of the parallelism to a compiler you need that compiler to be good at it
• still allows trivially buggy programs