University of Leeds

Lecture 13: Load balan

Previous lectures

Several times in this module we have mentioned the concept of **load b**

-
- Usually realised when **synchronisi**
- First encountered for the Mandelbrot set g 3].

- Important for parallel performance for shared and distributed memory CPU, and GPU.

Today's lecture

Today we will look at load balancing more closely, and how to redu

- tor, this
- Understand how heterogeneity in the problem results in poor load balancing.
- See how a **task scheduler** can imp **runtime**.

- Go through a concrete example of a **work pool**.

The Mandelbrot set (*c.f.* Lecture 3)

Code on Minerva: `Mandelbrot_MPI.c` plus makefile

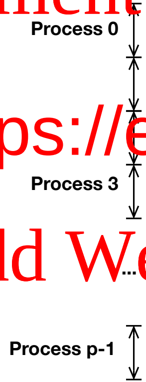
- Domain is $-2 \leq x \leq 2$ and -2 y 2.
- C
  it

- P
  the number of iterations.

- Here the **black** region corresponds to a **high number of iterations**.
- **No upper bound** - some points will iterate **indefinitely** if allowed.

Strip partitioning

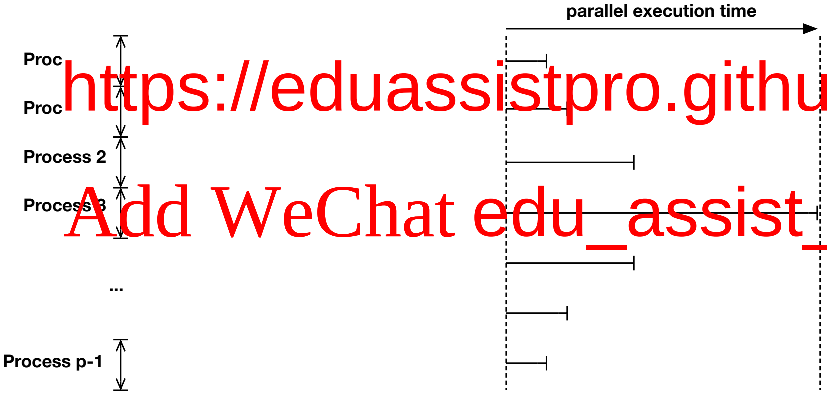
Partition the domain *[cf. last lecture]* into **horizontal strips**[1].



[1] Equivalent results for partitioning into vertical strips, or blocks.

Load imbalance

Because some pixels take longer to calculate the colour than others, the load is **unevenly distributed** across the processes:



Load balancing

- Parallel execution time determined by the **last** processing unit to finish.
- r at least

-

Definition

The goal of **load balancing** is for each **processing unit** (thread or process) to perform a **similar volume of computations**, and therefore finish at roughly the same time.

Up until now most problems we have encountered have been **naturally load balanced**.

For ex gning
each p
good l

- Each unit performs n/p additions

Note that the Mandelbrot set is a **ma**
parallel problem (since there are no data dependencies).

- *Still* a challenge to attain good performance.

Static load balancing

Definition

Som

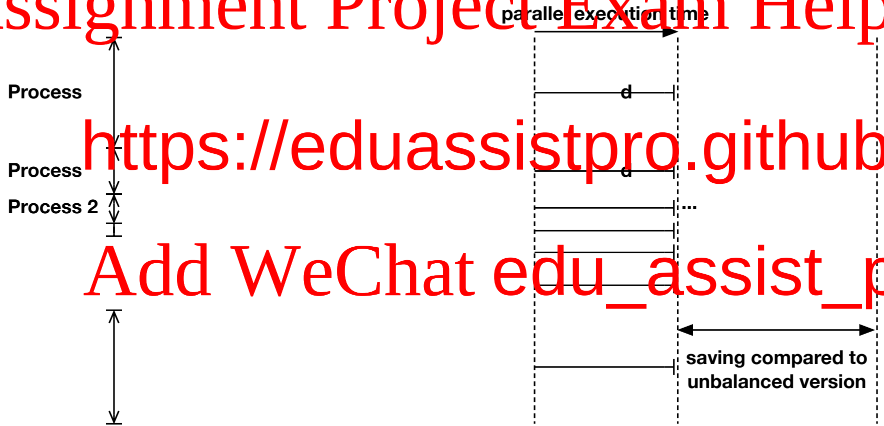
a .

For the Mandelbrot set example, we could assign s to regions where the calculations should be

- Should improve load balancing.

However, an **exact** expression is not available. Therefore any such **heuristic** can only achieve **approximate** load balancing.

Static load balancing *(ideal case)*



Dynamic load balancing

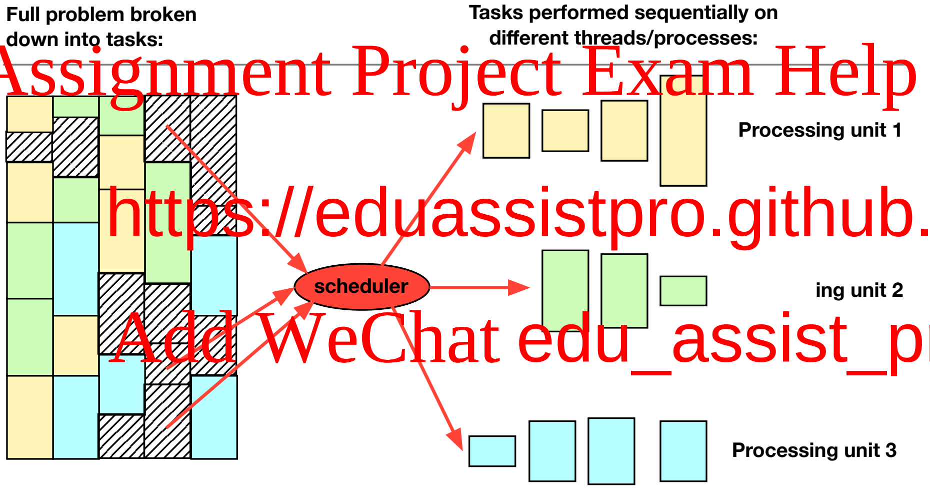
Definition

D … *riori*

Basic idea:

1. Break the problem down into small

2. Each processing unit performs **on**
3. When it is complete it starts is assigned another task
4. epeat 3 until all tasks are complete


Full problem broken down into tasks:
Tasks performed sequentially on different threads/processes:
scheduler
Processing unit 1
ing unit 2
Processing unit 3

Functional or task parallelism

Up to now we have largely considered parallelising the same operation to a (large) data set.

-
-

Now we are parallelising a number of **t**

- Called **task parallelism** or **functi**
- Be warned that these terms are sometimes used to refer to slightly different concepts.
- More on task parallelism in Lecture 19.

Work pools

The **scheduler** assigns tasks to processing units **at runtime**.

To understand the role of a scheduler, we will look at a s
scheduler implemented in MPI - a **cen**

- One process (usually rank 0) performs the sc
  the **main** process[1].
- Remaining processes action the tasks - the **workers**[1].

[1] You may see 'master' (for main) and 'slaves' (for workers) in the literature.

Worker pseudocode

Function `workerProcess()` in `Mandelbrot_MPI.c`

```
initialise();  // Including MPI_Init().

while( true )
{
  // Wa
  MP

  // Is this a termination request?
  if( message==TERMINATE ) break;

  // Else perform calculation and send back to rank 0.
  result = actionTask( message );
  MPI_Send( result, ... );
}

finalise();  // Including MPI_Finalize().
```

Main process pseudocode (1)

Function mainProcess() in Mandelbrot_MPI.c


```
initialiseAndOpenWindow();

// Initi
int numA

// Send in
for( p=1; p<numProcs; p++ )
{
  MPI_Send(task,...,p,...);
  numActive++;
}
```


For this Mandelbrot example, each `task` is a **row of pixel colours to be calculated**.

- Keep track with an incrementing variable `row`.

Main process pseudocode (2)

Function `idle()` in `Mandelbrot_MPI.c`


```
while( numActive>0 )
{
  // Get result from ANY worker process.
  MP
  nu

  // Se
  if( !finished )
  {
    MPI_Send(task,...,status.MPI_SOURCE,...)
    numActive++;
  }

  // Action the message.
  actionResult( result );
}
```

Main process pseudocode (3)

Function `idle()` in `Mandelbrot_MPI.c`

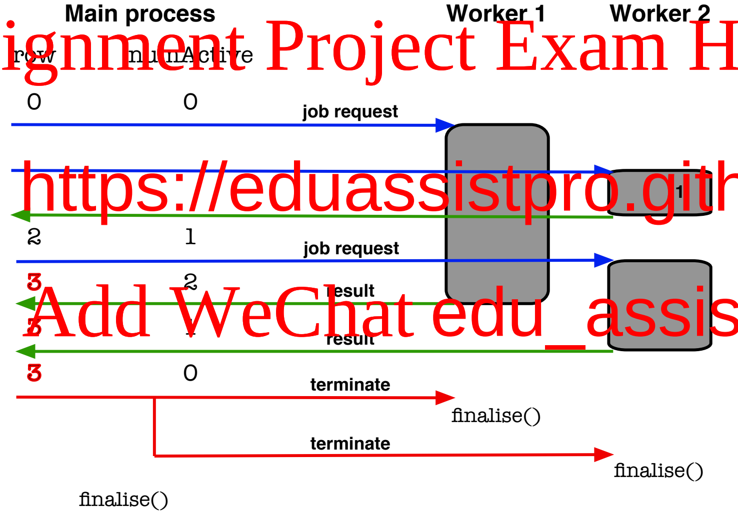

```
// Tell all workers to terminate.
for( p=1; p<numProcs; p++ )
  MP

final
```


- `MPI_ANY_SOURCE` in place of source in _ s a message from **any** process.

- Used `status.MPI_SOURCE` to recov process.
- Send next request **before** the (potentially slow) call to `actionResult()`.

Example: 3 rows and 2 workers



Modern schedulers

There are many more types of **work pool**, such as those with no 'main' process *(decentralised work pools)*[1].

A com ... ues:

-
- Performs tasks sequentially, starting fro
- Once the dequeue is empty, 'steals' a task fro ... of a randomly selected 'victim' (**work s**

[1]Wilkinson and Allen, *Parallel Programming* (Pearson, 2005).

OpenMP scheduler

OpenMP can also schedule loops using its `schedule` clause:

```
#pragma omp parallel for schedule(dynamic,chunk)
for( i=0
```

This b ntime.

Can al

```
#pragma omp parallel for schedule(static,chunk)
```


There is also a **guided** option that decrea exponentially **at runtime** to the final value

```
#pragma omp parallel for schedule(guided,chunk)
```

In all cases, `chunk` is optional and defaults to 1.

MIMD at last!

Up until today we have mostly performed the **same** calculations on each processing unit.

-
- **rdware**

Today is the first clear[1] example where w
MIMD pattern **in software**.

- The **main** process perform entirely diff
  workers *(division of labour)*.

[1]Ignoring trivial cases like *e.g.* rank 0 distributing global arrays.

Summary of distributed memory systems

Lec.	Content	Key points
8	Architectures and MPI	Clusters and supercomputers; interconnect network; starting with MPI.
9	Poin	
10	Da r sation	tive communication in MPI.
11	Reduction	Binary trees; OpenMP and MPI.
12	Asynchronous communication	Non-blockin partitioning a
13	Load balancing	Task parallelism; schedulers; work pools.

Next lecture we start looking at programming **general purpose graphics processing units** or GPGPUs.