XJCO3221 Parallel Computation

Peter Jimack

University of Leeds

Lecture 13: Load balancing

Previous lectures

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

- Poor load balancing results in processing units spending time **idle**.
- Usually realised when **synchronising** threads or processes.
- First encountered for the Mandelbrot set generator *[Lecture 3]*.
- Important for parallel performance for **all** architectures — shared and distributed memory CPU, and GPU.

Today's lecture

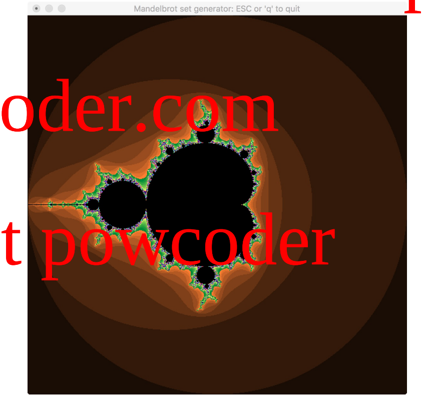
Today we will look at load balancing more closely, and how to reduce its performance penalty:

- Return to the example of the **Mandelbrot set** generator, this time in MPI.
- Understand how heterogeneity in the problem results in poor load balancing.
- See how a **task scheduler** can improve load balancing **at runtime**.
- Go through a concrete example of a **work pool**.

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

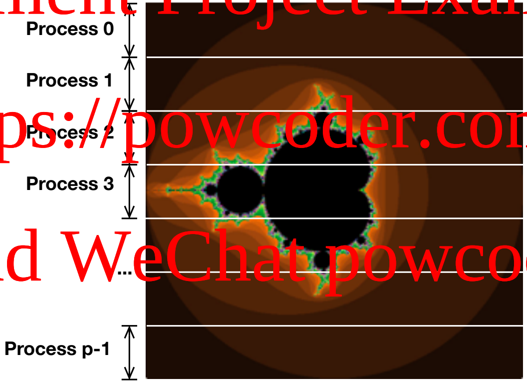
Code on Minerva: `Mandelbrot_MPI.c` plus makefile

- Domain is $-2 \le x \le 2$ and $-2 \le y \le 2$.
- Calculation performed **iteratively** for **each** (x, y).
- Pixel coloured according to 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

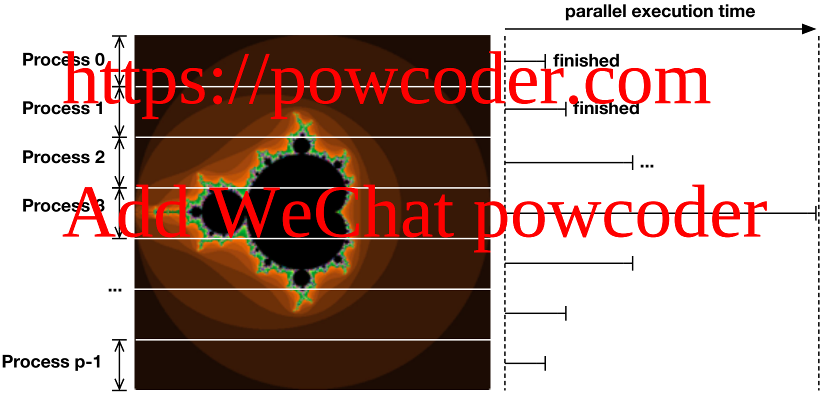
Partition the domain *[cf. last lecture]* into **horizontal strips**[1]:



[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.
- Poor load balancing results in significant **idle time** for at least one process/thread.
- Inefficient use of available resources.

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 example, for vector addition between two n-vectors, assigning each processing unit to equal numbers of vector elements results in good load balancing.

- Each unit performs n/p additions.

Note that the Mandelbrot set is a **map**, *i.e.* an **embarrassingly parallel problem** (since there are no data dependencies).

- *Still* a challenge to attain good performance.

Static load balancing

Definition

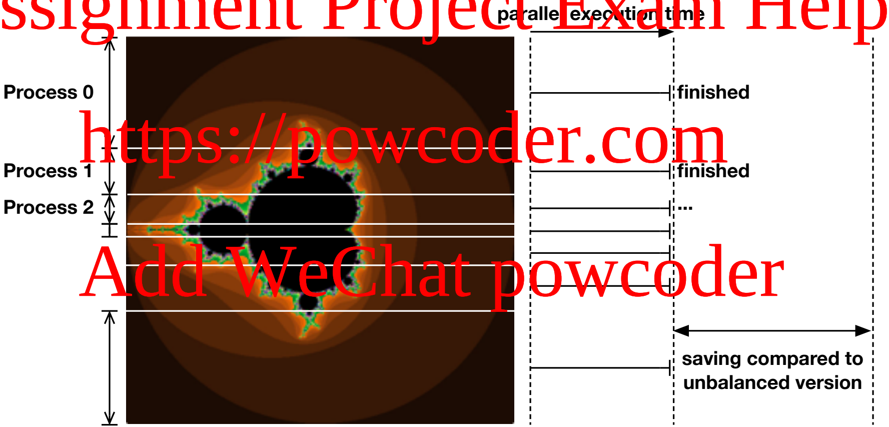
Sometimes it is possible to determine (approximately) equal loads **at compile time**. This is known as **static load balancing**.

For the Mandelbrot set example, we could assign **larger** domains to regions where the calculations should be **fast**.

- 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)*



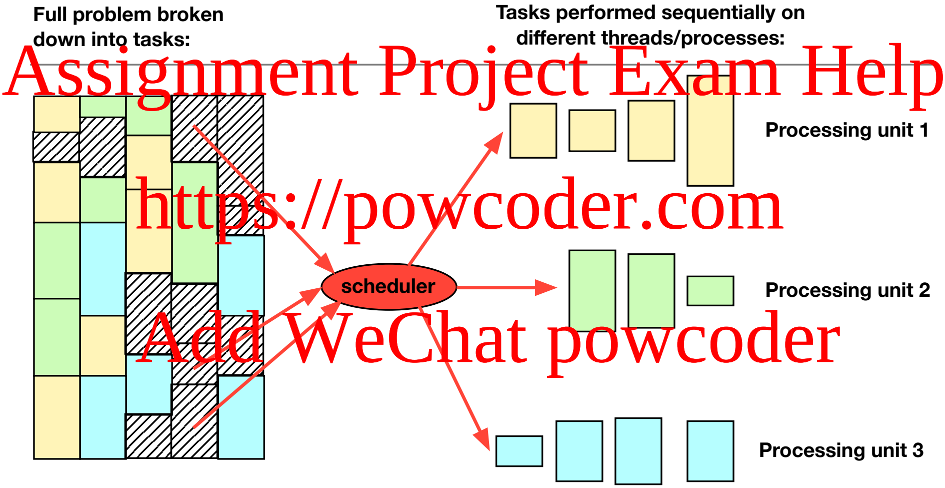
Dynamic load balancing

Definition

Dynamic load balancing is performed **at runtime**. No *a priori* knowledge of computation times is required.

Basic idea:

1. Break the problem down into small **independent tasks**.
2. Each processing unit performs **one** task at a time.
3. When it is complete, it starts/is assigned another task.
4. Repeat 3 until all tasks are complete.


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

- Known as **data parallelism**.
- When performed in a loop, can also be termed **loop parallelism**.

Now we are parallelising a number of **tasks**.

- Called **task parallelism** or **functional parallelism**.
- 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**.

- Often part of the parallel/concurrency runtime system.
- Introduces a (small) overhead.
- Various algorithms exist for efficient task scheduling.

To understand the role of a scheduler, we will look at a simple scheduler implemented in MPI - a **centralised work pool**.

- One process (usually rank 0) performs the scheduling - this is 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 )
{
  // Wait for message from the main (rank 0).
  MPI_Recv( message, ... );

  // 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();

// Initialise variable that tracks progress.
int numActive=0;

// Send initial request to each worker.
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.
  MPI_Recv(result,...,MPI_ANY_SOURCE,...,&status);
  numActive--;

  // Send request IMMEDIATELY to the SAME worker.
  if( !finished )
  {
    MPI_Send(task,...,status.MPI_SOURCE,...);
    numActive++;
  }

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

Main process pseudocode (3)

Function idle() in Mandelbrot_MPI.c

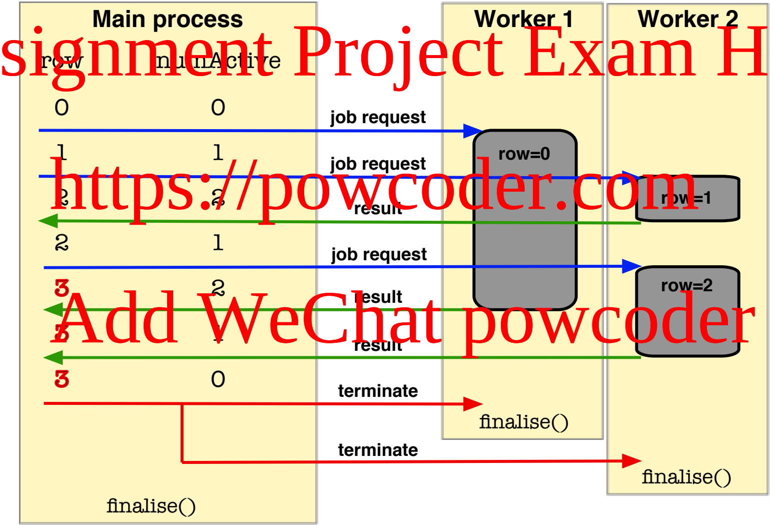
```
// Tell all workers to terminate.
for( p=1; p<numProcs; p++ )
  MPI_Send(TERMINATE,...);

finaliseAndCloseWindow();
```

- MPI_ANY_SOURCE in place of source in MPI_Recv() receives a message from **any** process.

- Used status.MPI_SOURCE to recover the rank of the sending 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 common approach is to use **deques** or *double ended queues*:

- Each processing unit maintains its own **deque** of tasks.
- Performs tasks sequentially, starting from the front.
- Once the dequeue is empty 'steals' a task from the **back** of a randomly selected 'victim' (**work stealing**).

[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; i<n; i++ ) { ... }
```

This breaks down the loop into ‘chunks’ of size chunk at runtime.

Can also be used for **static** scheduling:

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

There is also a **guided** option that decreases the chunk size exponentially **at runtime** to the final value chunk:

```
#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.

- SIMD (Single Instruction Multiple Data) **software** . . .
- . . . on MIMD (Multiple Instruction Multiple Data) **hardware**.

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

- The **main** process perform entirely different calculations to **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	Point-to-point communication	Blocking send and receives; buffering; deadlock for cyclic communication.
10	Data reorganisation	Scatter and gather; t_{comm}; collective communication in MPI.
11	Reduction	Binary trees; OpenMP and MPI.
12	Asynchronous communication	Non-blocking send and receives; domain partitioning and ghost cells.
13	Load balancing	Task parallelism; schedulers; work pools.

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