2. Decomposing the Problem

Parallel Programming

Dr Hamidreza Khaleghzadeh
School of Computing
University of Portsmouth

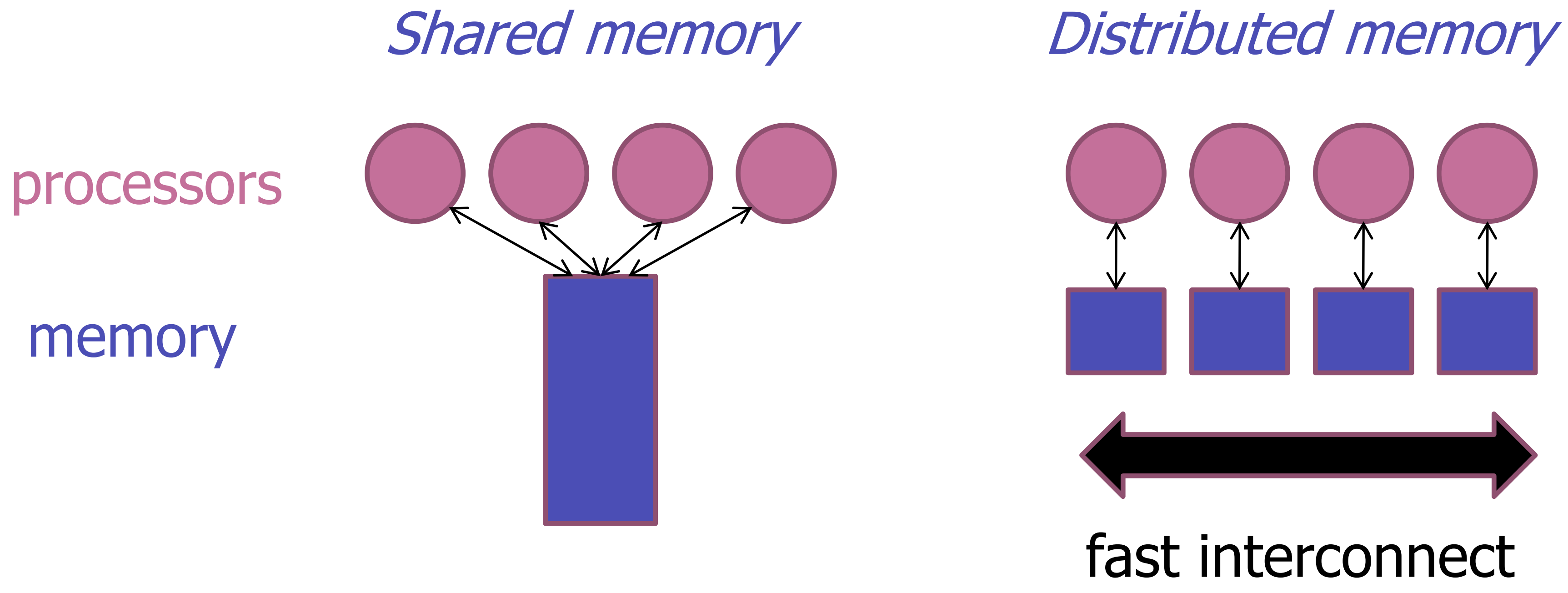
Goals

- Start with a general overview of programming frameworks for parallel computing, contrasting *shared memory* and *distributed memory*.
- Then, with reference to a simple example, discuss issues of problem decomposition and load balancing.

PARALLEL PROGRAMMING

Parallel Computer Organization

- Traditional dichotomy in parallel computing – shared vs distributed memory (*SMP* † vs *cluster*):



†*Symmetric Multi-Processor*.

SMP Programming Frameworks

- Shared memory (cooperating *threads*):
  - POSIX or Java Threads
  - OpenMP
  - Cilk
  - Intel Threaded Building Blocks
  - etc, etc

Threads

- In C/C++ can use the *POSIX threads*† (*pthreads*) libraries to achieve concurrency.
- In Java, use native Thread class for same effect – we are using this approach in early labs.
- On *multicore* system, threads automatically scheduled on different cores to give parallel execution.

† There is a decent tutorial at http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html, but note pthreads is not limited to Linux – also works on Windows, for example!

OpenMP

- *OpenMP*[†] (which rather misleadingly styles itself an "API") is a set of *compiler directives* and supporting libraries for exploiting *shared memory* computers.
- Defined for C/C++ (or Fortran)
- Parallelizing a loop in OpenMP *may* be as simple as this:

```
#pragma omp parallel for
for(i = 0 ; i < N ; i++) {
    ...
}
```

† www.openmp.org

Cluster Programming Frameworks

- Distributed memory (cooperating *processes*)
  - MPI (PVM, etc)
  - Co-array Fortran, UPC, etc
  - Global Array Toolkit (etc)
  - etc, etc

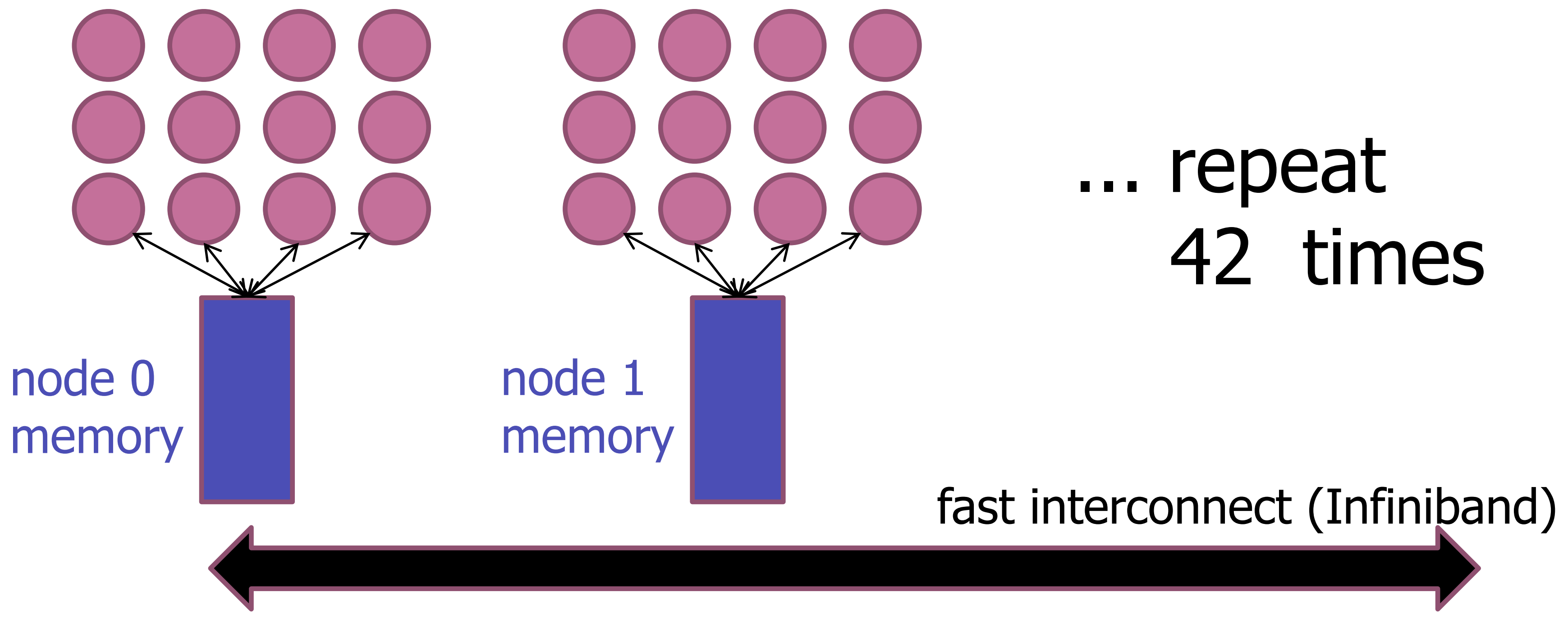
MPI

- *MPI*[†] (simply *Message Passing Interface*) is a standard programming interface for sending and receiving messages in C or Fortran programs.
- Beyond this, provides most popular framework for *Single Program Multiple Data* (SPMD) computing on large scale distributed memory parallel computers – i.e. *clusters*.
- In labs we will use a Java version of MPI called *MPJ Express*.

[†] http://www-unix.mcs.anl.gov/mpi

Real World – Hybrids

- Clusters of shared memory nodes – typical of large modern systems
- e.g. Portsmouth SCIAMA supercomputer



Programming Hybrid Machines

- *Can* be programmed using a combination of shared memory framework (e.g. threads or OpenMP) *within* nodes, plus distributed memory framework (e.g. MPI) *between* nodes.
- Sometimes easier to adopt the lowest common denominator – use MPI (say) across *all* cores. But efficiency lower.

SMPs – Pros and Cons

- Parallel programming is generally easier for shared memory systems – also called *Symmetric Multi-Processors* (SMPs).
  - No need to split up data structures into separate pieces held in memory of different nodes
  - No need for explicit communication operations between nodes – inter-node communication happens implicitly through memory access (usually *synchronization* operations will still be required).
- Commodity multicore microprocessors are generally SMPs.
  - But the *largest* parallel systems are *not*.

A Large SMP†

- COSMOS supercomputer in Cambridge for astrophysics, launched (by Stephen Hawking) 2012
- One of the biggest *shared memory* computers in Europe. SGI Altix UV2000 built from Intel Xeon processors with total *1856* cores.
- But recall *largest* supercomputers have *millions* of cores!

† https://safe.epcc.ed.ac.uk/diracwiki/index.php/Cambridge_COSMOS_SHARED_MEMORY_Service

Early Lectures

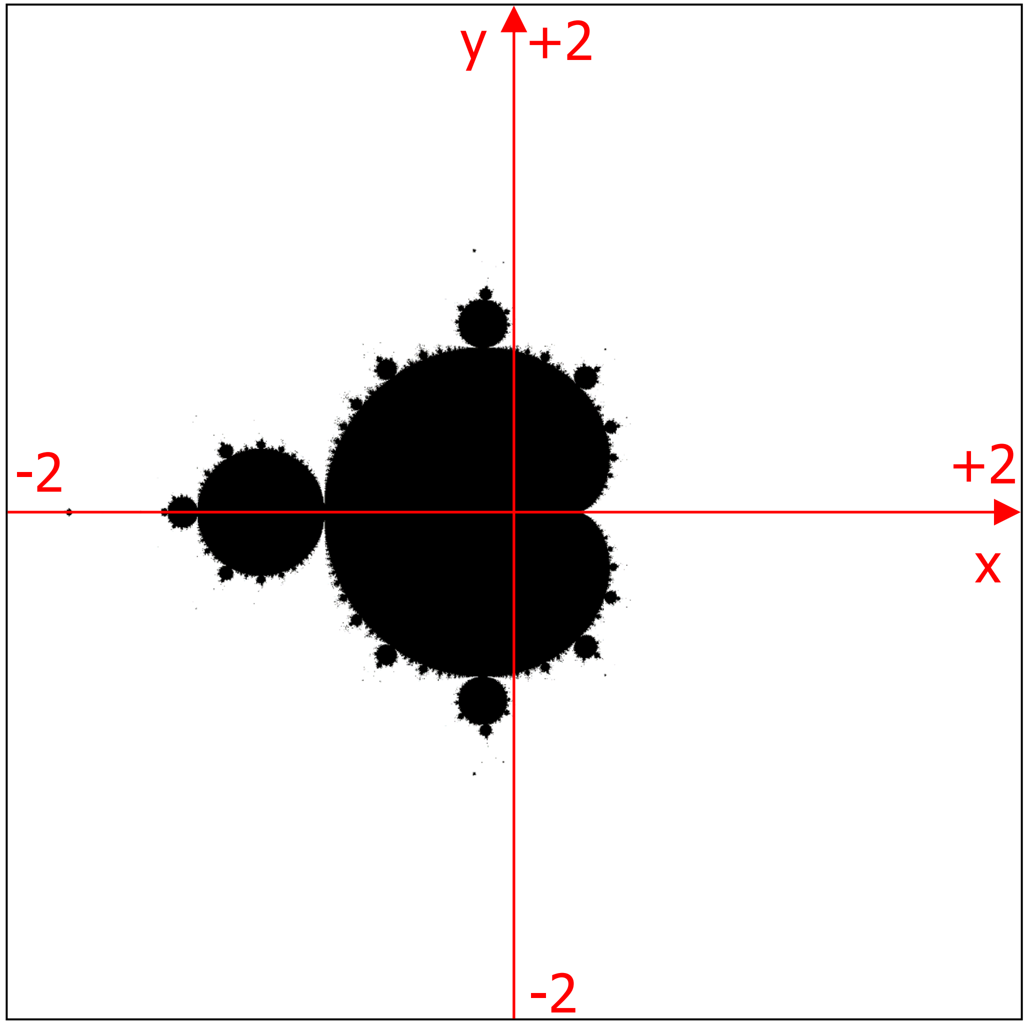
- Shared memory systems are ultimately less scalable than distributed memory.
- But, because easier, we begin our exploration of parallel programming assuming shared memory systems.
- Later lectures will explore parallel programming for clusters.

PROBLEM DECOMPOSITION

First Example

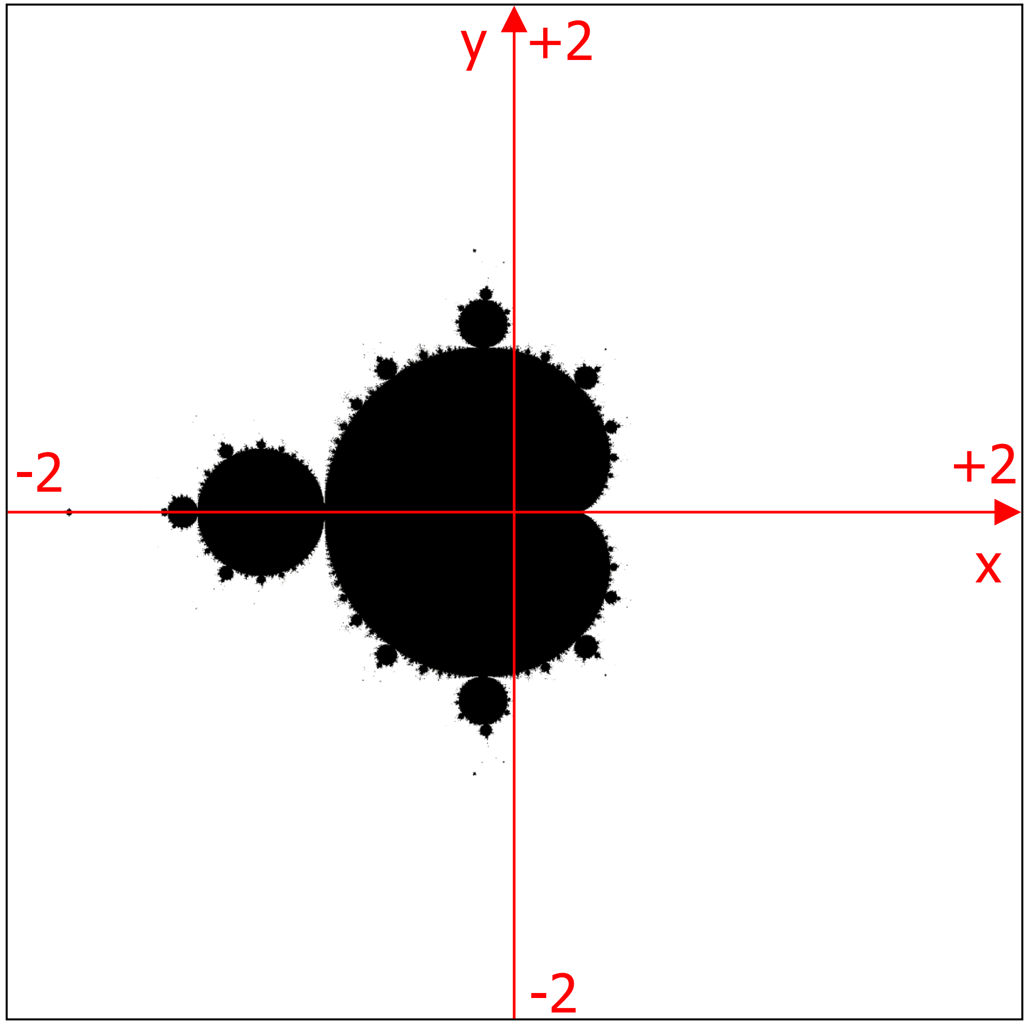
- In the labs you should already have started looking at parallelizing the *Mandelbrot Set*.
- Good starting example because fun and easy to parallelize ("embarrassingly parallel").
- But also illustrates significant issues about *problem decomposition* - always important in parallel programming.

Mandelbrot Set



Mathematically, the set is the *black area* within part of the *x*, *y* (or complex number) plane with $-2 \le x, y \le 2$

Mandelbrot Set (Maths)



Define complex:

$$c = x + iy$$

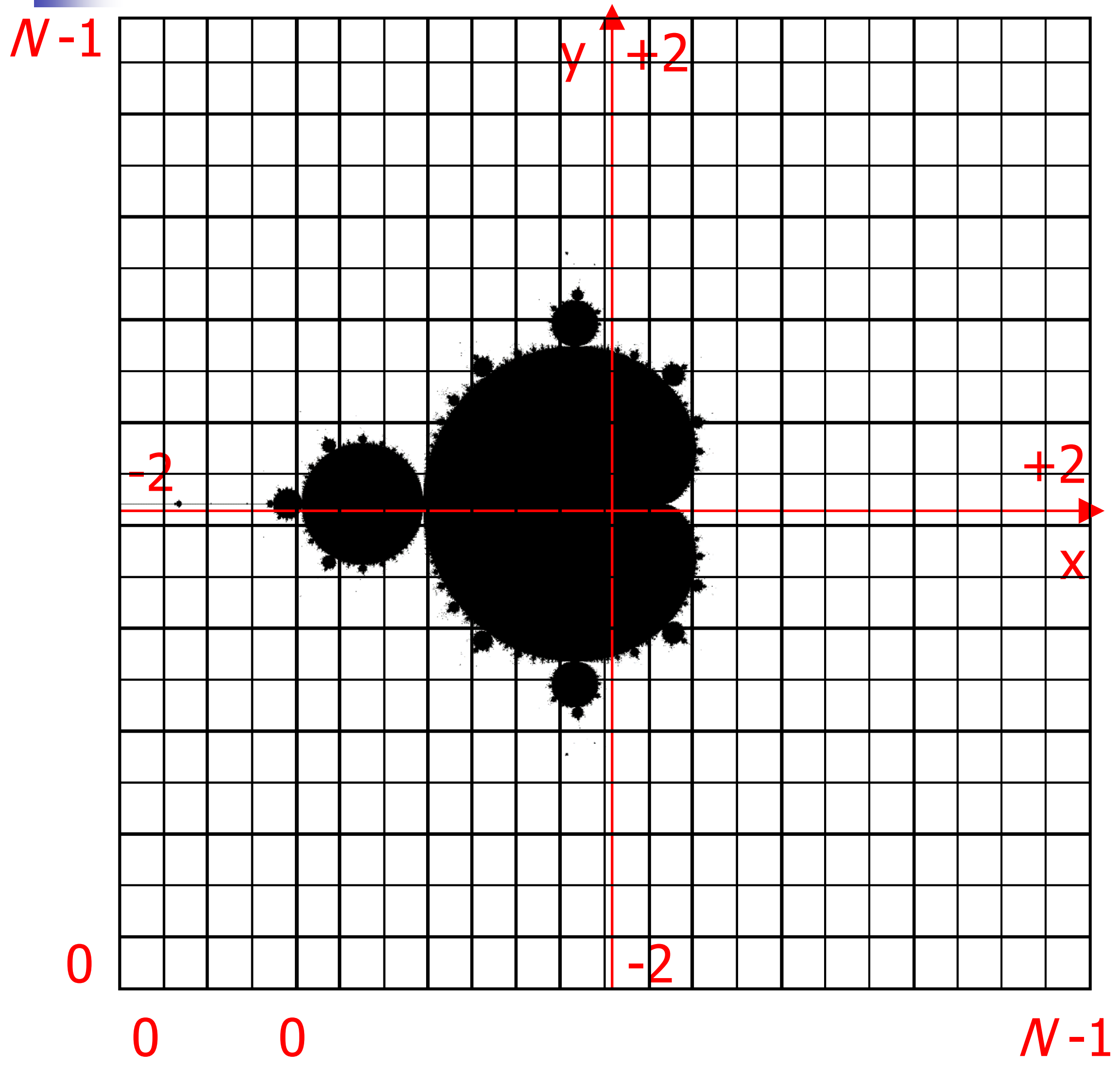
and recurrence:

$$z_0 = c$$

$$z_{k+1} = c + z_k^2$$

If $\mathrm{abs}(z_k) < 2$ for all $k < \infty$, then point x,y is in the set (black area).

Mandelbrot Set (Numeric)



Consider only x, y values in N by N grid ("pixels").

If abs(z_k) < 2 for all k < CUTOFF, then point x,y is in the set (black area), where CUTOFF is some sufficiently large number.

Pseudocode

```
for(int i = 0 ; i < N ; i++) {
   for(int j = 0 ; j < N ; j++) {
       double x = step * i - 2.0 ;    // -2 ≤ x ≤ 2
       double y = step * j - 2.0 ;    // -2 ≤ y ≤ 2

       complex c = (x, y), z = c ;

       int k = 0 ;
       while (k < CUTOFF && abs(z) < 2.0) {
           z = c + z * z;
           k++ ;
       }
       set [i] [j] = k ;
   }
}
```

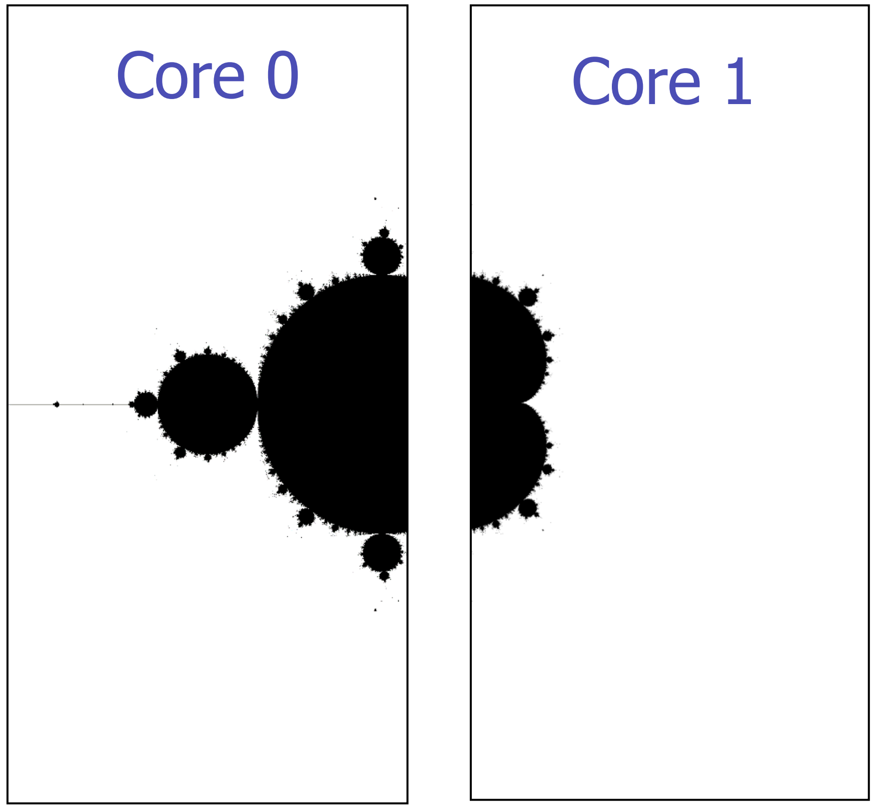
Loop Structure

- Don't worry about details of *z* recurrence here, but *do* look at general loop structure.
- Mathematically, *innermost while loop* repeats *forever* if we are in the black region; in practice, stop the loop after some number **`CUTOFF`** of iterations.
- *Upshot*: *inside or near* the black area there are many more iterations - and *more computational work* - than for typical points in the white area (where the while loop may end after a few iterations).

Parallel Decomposition

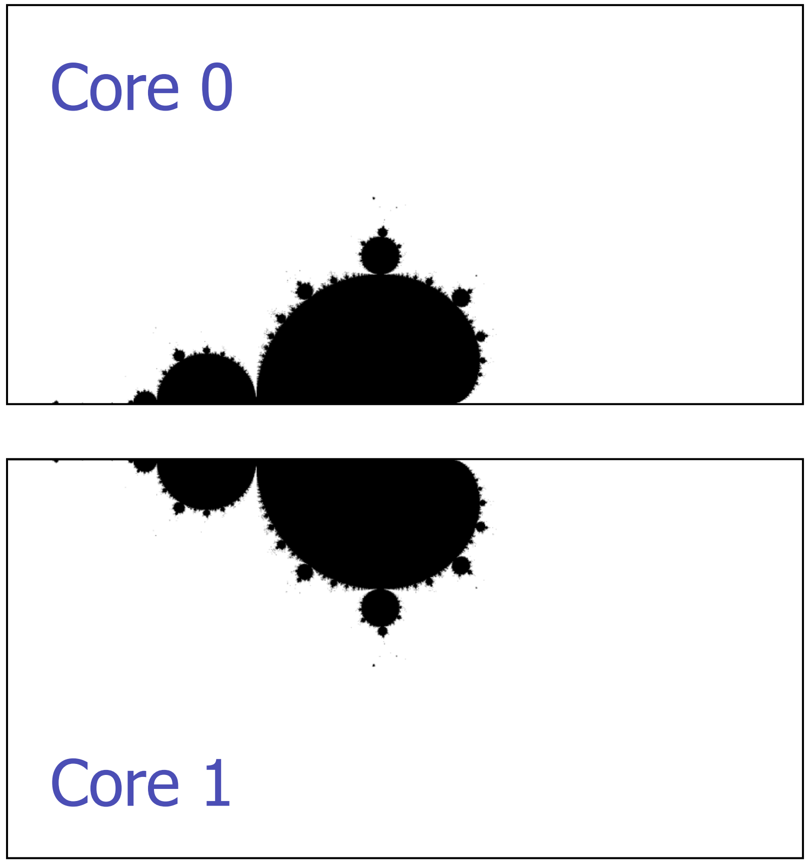
- If we want to calculate the set on, say, two cores, an obvious thing is to break the x,y plane into two halves – we may either do this vertically or horizontally.

Divide the *i* -loop



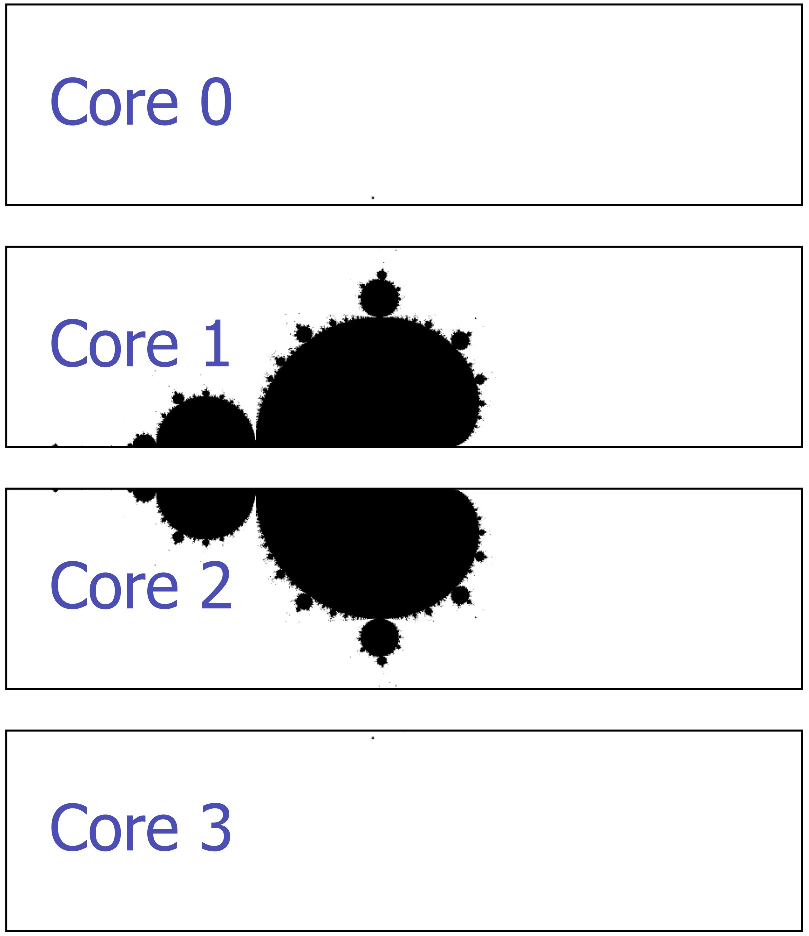
- Poor *load balancing*; core 0 has more work (more black!) than core 1.
- Work not evenly divided between cores, so imperfect parallel speed up.

Divide the *j* -loop



- Perfect *load balancing* because of symmetry of set.
- Parallel speed-up factor may be close to 2.

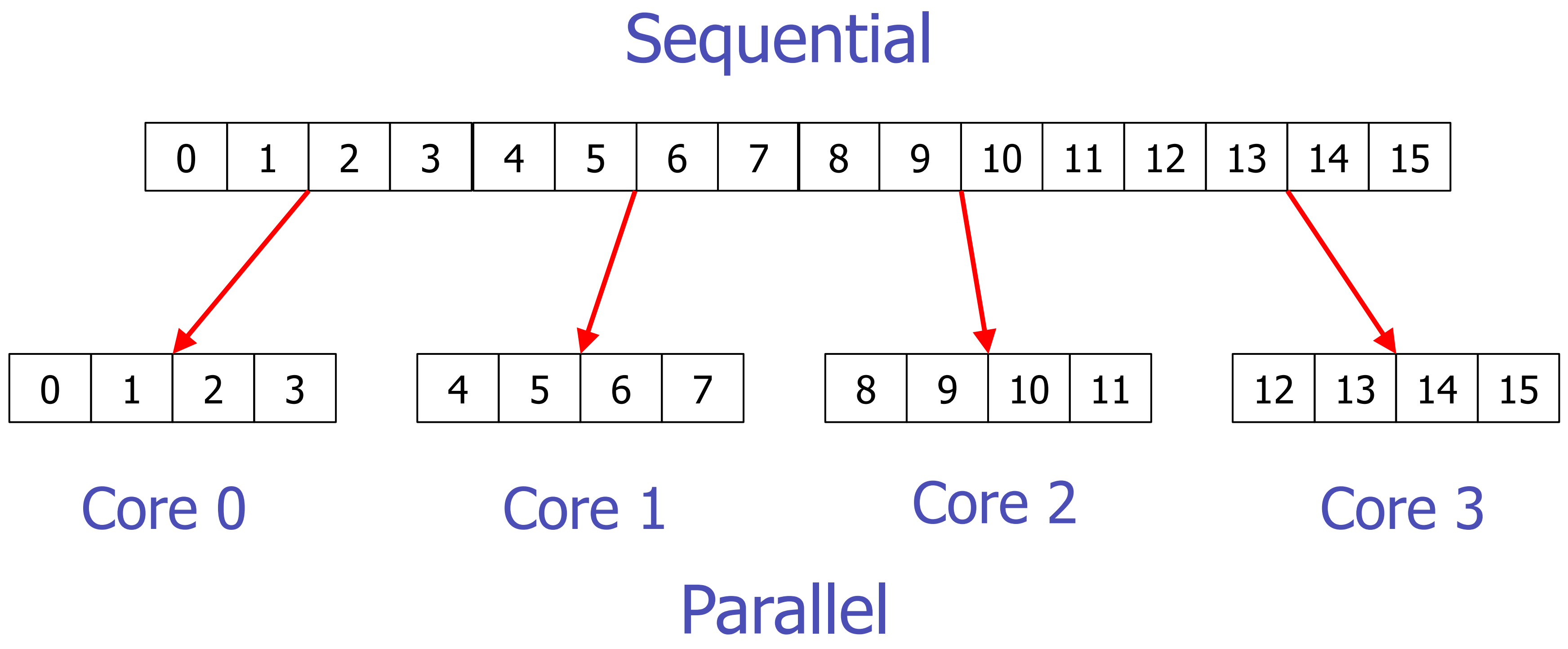
But…



- Disastrous *load balancing* – cores 0 and 3 only have about a tenth of the work of cores 1 and 2.
- Parallel speed-up on four cores not much better than on two cores!

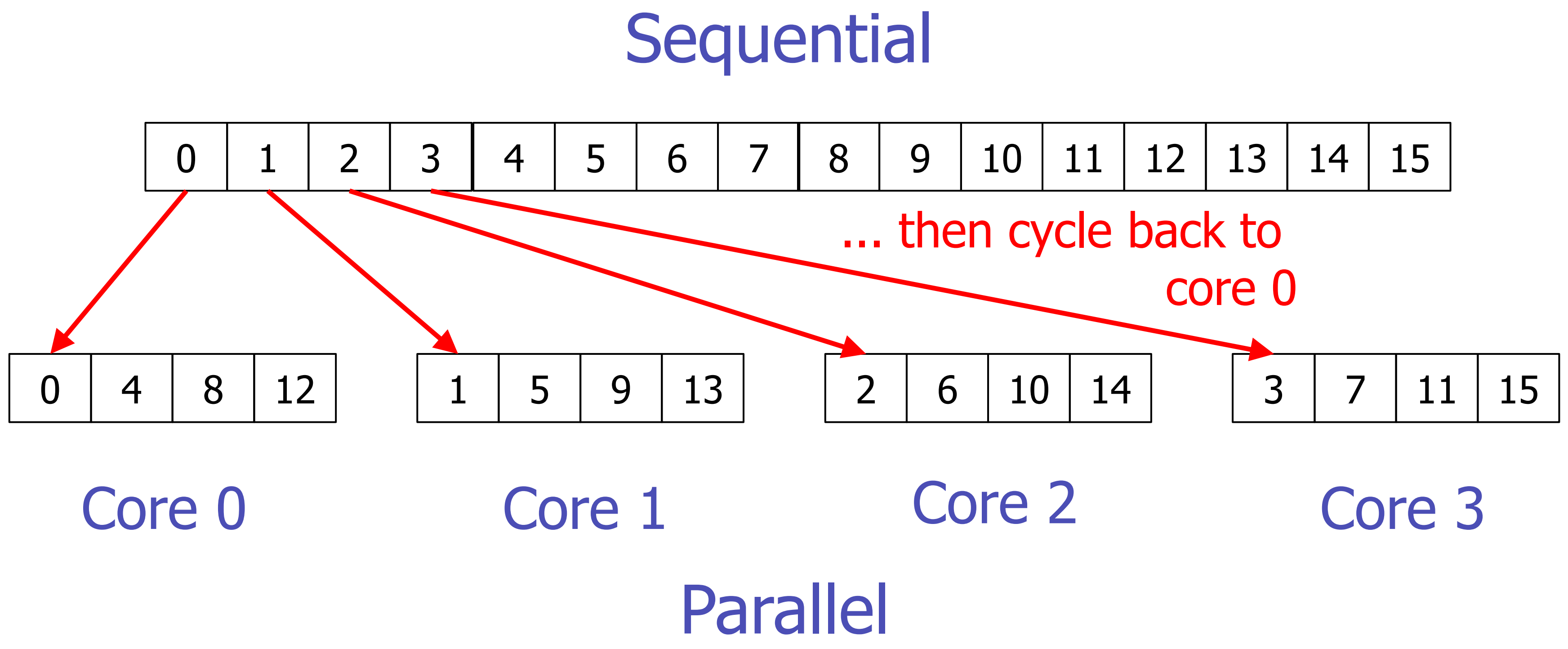
Workload Distribution Formats

- Both horizontal and vertical decompositions are using a *block-wise* distributions of *index space*:

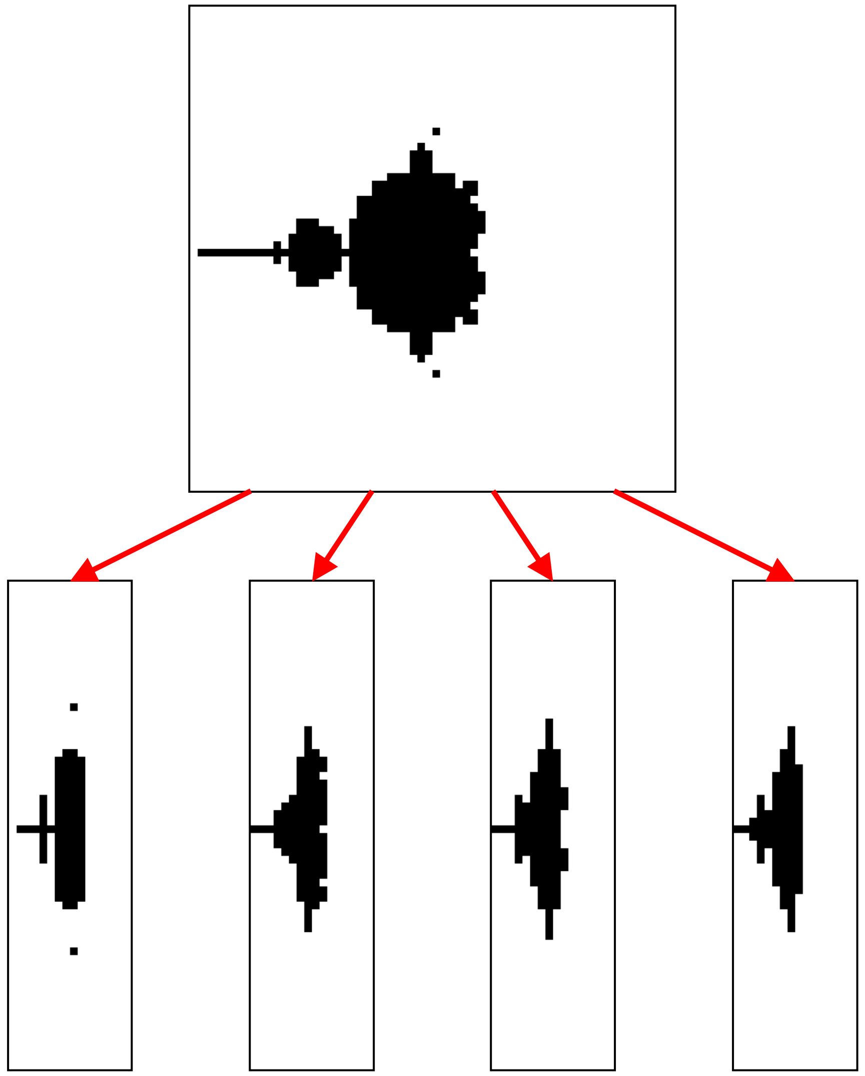


"Cyclic" Distribution

- An alternative distributions of *index space* from original problem:



Mandelbrot with Cyclic Decomposition



- Much more even break down of set.
- Good load balance.

Programming Considerations

- Assume original loop is

```
for (int i = 0 ; i < N ; i++) {...}
```

- Block-wise decomposition, each thread does[†]:

```
for (int i = me*N/P; i < (me+1)*N/P ; i++) {...}
```

- Cyclic decomposition, each thread does

```
for (int i = me ; i < N ; i+=P) {...}
```

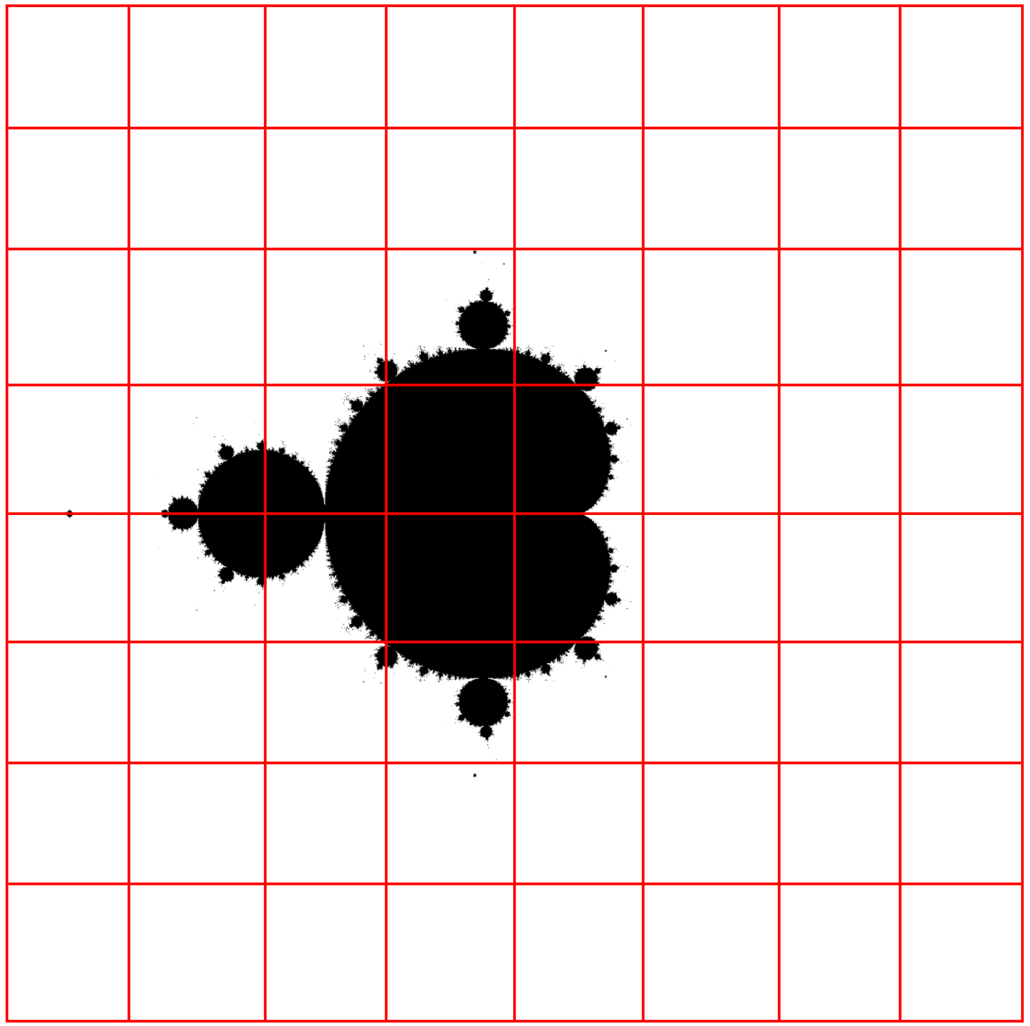
- Here *P* is number of threads, and *me* is current thread identity.

[†]Formulas here get slightly more complicated if *P* doesn't divide *N* exactly.

Is Cyclic Decomposition Just Better?

- Not always, by any means.
- Even on shared memory computers, it sometimes leads to poor *cache* utilization, because of the stride in the local loop (i += P).
- On distributed memory computers, as we will see later, it often leads to *excessive communication requirements*.
- But for problems where *load balancing* is the main problem it can be very useful.

Other Approaches to Load-Balancing



- Divide the work into *many small chunks* – in this case many small tiles of Mandelbrot
- Need *many more tasks* than number of available *cores*.

Dynamic Load Balancing

- Each core grabs one tile, more or less randomly, and works on that.
- If it finishes before all tiles have been processed, it grabs another tile, and so.
- Keeps busy.
- Obviously more complex to program, but will discuss in more detail later.

ASIDE: PARALLEL SPEEDUP AND PARALLEL EFFICIENCY

Parallel Speedup

- One way of expressing the performance of a program is in *GigaFLOPS*. For parallel programs, another way is in terms of *speedup*.
- Suppose the execution time of the sequential version of the program is T_S, and the execution of a parallel version of the same program running on P cores is T_P.
- We define the *parallel speedup* as:

$$T_S / T_P$$

- What is the *ideal* parallel speedup on P cores?

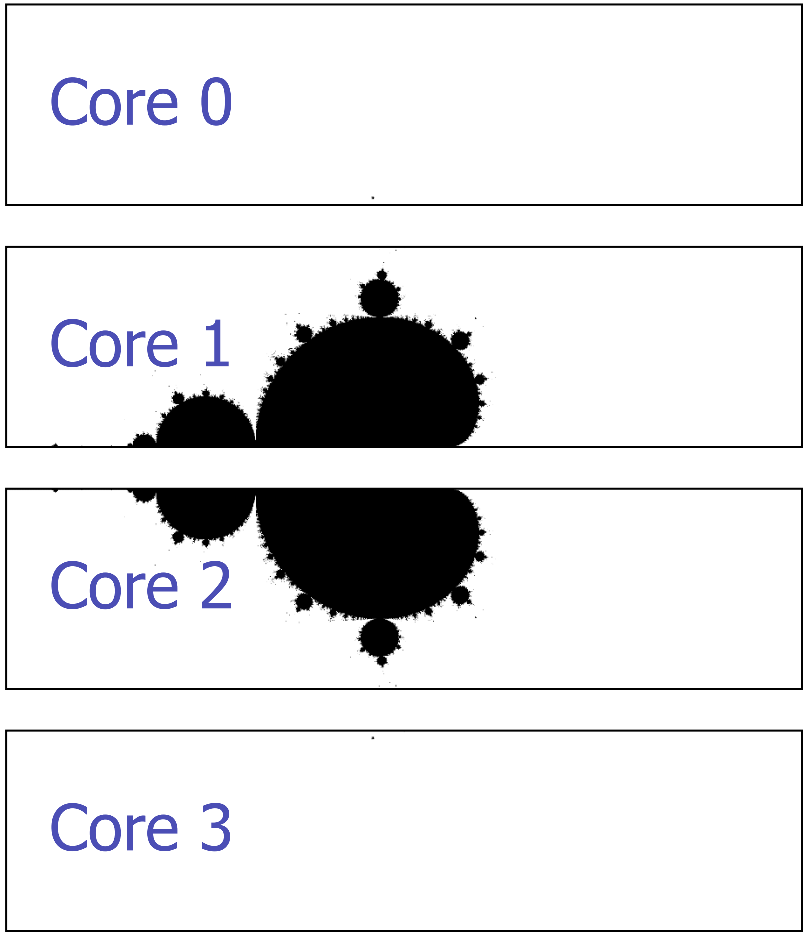
Parallel Efficiency

- *Parallel efficiency* is the ratio of the *actual* speedup of a parallel program to the ideal speedup:

$$(T_S / T_P) / P$$

  The efficiency is normally a number between 0 and 1.
- It measures how effectively we are making use of the P available cores.
- Parallel efficiency can fall due to poor load balancing, or other overheads of parallelism.

An Example from Above



- Assume the only inefficiency is due to load balancing...
- ... and cores 0 to 3 respectively do:

  5%, 45%, 45%, 5%

  of total work.

Speedup and Efficiency for Example

- Time for individual cores to complete work is:

$$0.05T_S, 0.45T_S, 0.45T_S, 0.05T_S$$

and parallel program as a whole finishes in *longest* of these times:

$$T_P = 0.45T_S$$

- *Parallel speedup* is:

$$T_S / T_P = T_S / (0.45T_S) = 1/0.45 \approx 2.22$$

- *Parallel efficiency* is

$$(T_S / T_P) / P = 2.22 / 4 \approx 0.55$$

or 55%.

ASIDE: BENCHMARKING PROTOCOL

Reproducibility of Timings

- Benchmarking parallel and sequential programs will be vital throughout the labs.
- For various reasons the time a program takes to execute is usually not completely reproducible – it varies from run to run.
- Where high quality timing results are needed it is advised to run the program several times (e.g. 4, 10, or more times) and record *all* the times.
  - To start with, this gives you a feeling for how variable the timing results are.
  - But then what should you record as the final result?

Summary Timings

- Many people's intuition is to take the *average* of all the runs as the best estimate of the time.
- I recommend that instead you take the *minimum* time of all the runs as the final result.
  - This may sound like cheating. But usually most variation from the minimum time is due to events external to the program itself (OS threads etc).
  - Using the minimum time usually leads to smoother and more reproducible graphs, etc.

Summary

- Next week: Topics in *shared memory parallel programming*.

Further Reading

- See links embedded in these slides.