

# Image sharpening exercise

---

Running a simple parallel program



# Reusing this material



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

[http://creativecommons.org/licenses/by-nc-sa/4.0/deed.en\\_US](http://creativecommons.org/licenses/by-nc-sa/4.0/deed.en_US)

This means you are free to copy and redistribute the material and adapt and build on the material under the following terms: You must give appropriate credit, provide a link to the license and indicate if changes were made. If you adapt or build on the material you must distribute your work under the same license as the original.

Acknowledge EPCC as follows: “© EPCC, The University of Edinburgh, [www.epcc.ed.ac.uk](http://www.epcc.ed.ac.uk)”

Note that this presentation contains images owned by others. Please seek their permission before reusing these images.

# Aims (i)

- To familiarise yourself with running parallel programs
- To run a real parallel code (that does file I/O)
  - On different numbers of cores
  - Measure the time taken
  - Observe increase in performance (Amdahl's law? – see later)
- Acknowledgements
  - Algorithm, diagrams and images taken from:
  - *Hypermedia Image Processing Reference*, Bob Fisher, Simon Perkins, Ashley Walker and Erik Wolfart, Department of Artificial Intelligence, University of Edinburgh (1994)

## Aims (ii)

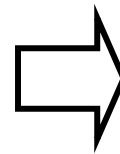
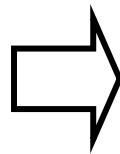
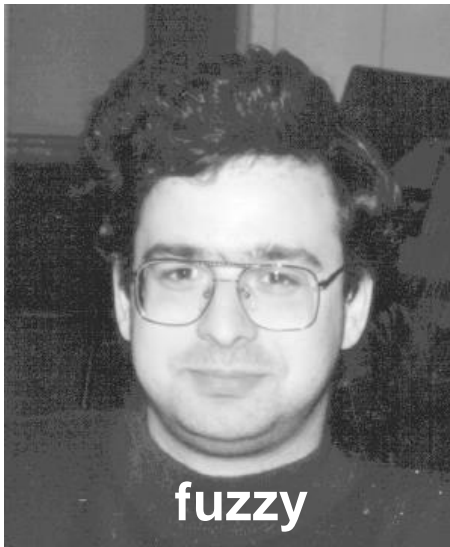
- To get you running on the machine
- To sort out all the practical details
  - usernames
  - passwords
  - graphics
  - transferring files
  - using the batch system
  - idiosyncrasies of your Windows / Mac / Linux laptop
- Please ask for assistance if you need it!
  - Demonstrators are here to help with all aspects of course

# The image sharpening problem

Algorithm and implementation

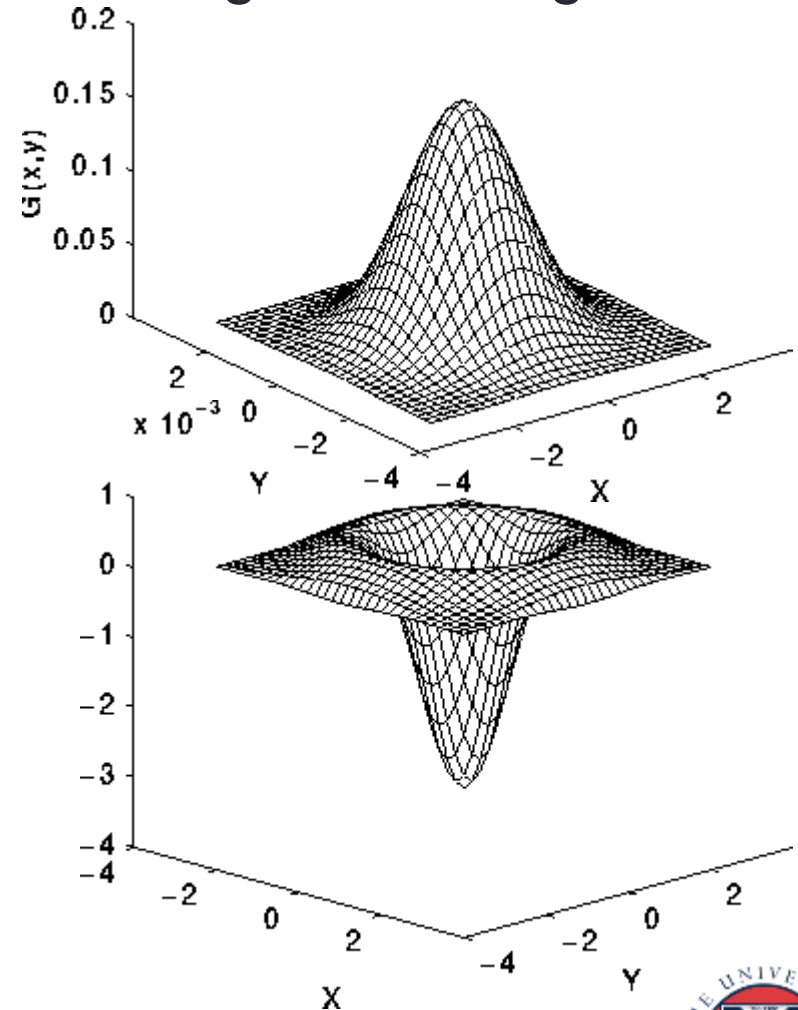
# Image sharpening

- Images can be fuzzy for two main reasons
  - random noise
  - blurring
- Aim to improve quality by
  - smoothing to remove noise
  - detecting edges
  - sharpening up the image with the edges



# Technicalities

- Each pixel replaced by a weighted average of its neighbours
  - weighted by a 2D Gaussian
  - averaged over a square region
- we will use:
  - Gaussian width of 1.4
  - a large square region
- then apply a Laplacian
  - this detects edges
  - a 2D second-derivative  $\nabla^2$
- Combine both operations
  - produces a single convolution filter



# Implementation

- For over every pixel in the image
  - loop over all pixels in a large area surrounding it
    - up to distanced  $d$  away in each direction:  $2d+1 \times 2d+1$  square
    - we use  $d = 8$ , i.e. a  $17 \times 17$  square
  - add in the value of the pixel weighted by a filter

$$edge(i, j) = \sum_{k=-d, d} \sum_{l=-d, d} image(i + k, j + l) \times filter(k, l)$$

- This gives the edges
  - add the edges back into the original image with some scaling factor
    - we use scale factor of 2.0
  - rescale the sharpened image so pixels lie in the range 0 - 255



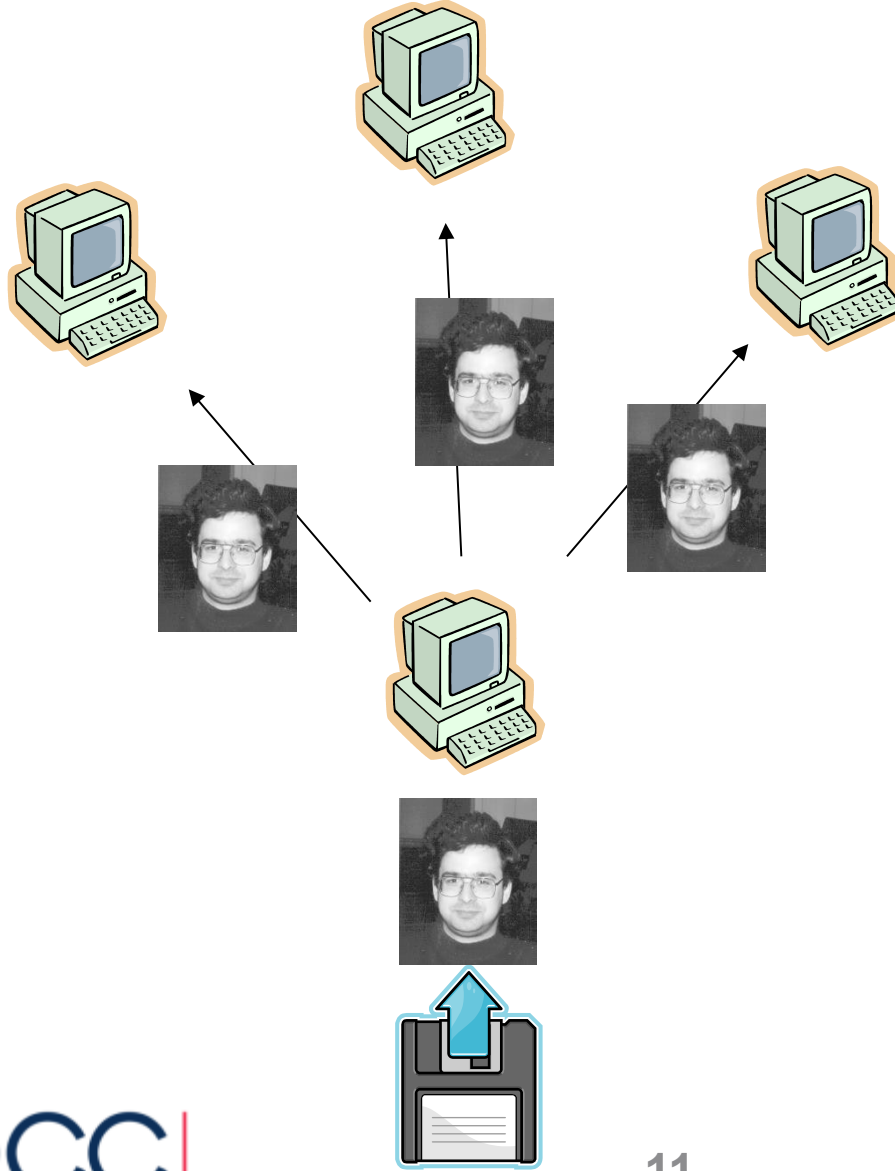
# Existing parallelisation

How the code takes advantage of multiple processors

# Parallelisation

- Each pixel can be processed independently
- A master process reads the image
- Broadcast the whole image to every process
- Each process computes edges for a subset of pixels:
  - scan the image line by line
  - with four processes, each process computes every fourth pixel
- Combine the edges back onto a master process
  - add back into original image and rescale
  - save to disk
- Reports two times:
  - calculation time for just computing edges on each process
  - overall time for the whole program including IO

# Parallelisation



|   |   |   |   |   |  |
|---|---|---|---|---|--|
|   |   |   |   |   |  |
| 1 | 2 | 3 | 4 | 1 |  |
| 2 | 3 | 4 | 1 | 2 |  |
| 3 |   |   |   |   |  |
|   |   |   |   |   |  |
|   |   |   |   |   |  |
|   |   |   |   |   |  |
|   |   |   |   |   |  |

# A number of implementations provided

- Supply a serial version for reference
- Parallelisation is achieved using message-passing model
- Implemented using MPI
  - the Message-Passing Interface
- Another version parallelised using shared-variables model
- Implemented using OpenMP
  - HPC standard for threaded programming
  - for interest - not critical to this exercise
- These concepts will be explained later in the course ...

# Miscellaneous notes

Extra stuff to help you with the practical

# PBS job submission scripts (ARCHER)

```
#PBS -N sharpen
```

**name for PBS  
batch job**

```
#PBS -l select=1
```

**how many *nodes*  
you want**

```
# now stuff that actually executes
```

```
...
```

```
aprun -n 4 ./sharpen
```

**program to run**

**parallel job launcher**

**how many *cores* to  
run on – remember  
24 cores per node!**

# PBS job submission scripts (Cirrus)

```
#PBS -N sharpen
```

name for PBS  
batch job

```
#PBS -l place=excl
```

exclusive access – no  
other users on node

```
#PBS -l select=1:ncpus=72
```

how many *nodes*  
you want

```
# now stuff that actually executes
```

```
...
```

```
mpiexec_mpt -n 4 -ppn 4 ./sharpen
```

program to run

number of Processes  
Per Node

parallel job launcher

how many *cores* to  
run on – remember  
36 cores per node!



# Compiling and Running

- We provide a tar file with code (C or Fortran) and image
  - copy tar file it to your local account
  - unpack it
  - compile it
  - run it on the back end using appropriate batch scripts
  - view the input and output images using `display` program
  - note the times for different numbers of processors
    - can you interpret them?