Using Abel

Abel

- Abel: UiO's Linux cluster
- Number of compute nodes: 650+
- Each compute node has two 2.6GHz Xeon E5-2670 8-core CPUs
- Total number of CPU cores: 10,000+
- Inter-connect: InfiniBand
- No. 368 on the TOP500 list (Nov 2014)
  `http://www.top500.org/system/177801`

More about Abel

- Operating system: Linux, 64 bit Centos 6
- Access to Abel (using your UiO username/password)
  `ssh -X username@abel.uio.no`
- Software available on Abel, please check
  `www.uio.no/english/services/it/research/hpc/abel/help/software/`

The `module` command

On Abel the `module` command is used to set up environments for compilers, MPI versions and some installed application software

More info:

`www.uio.no/english/services/it/research/hpc/abel/help/user-guide/modules.ht`

Examples:

```
module load intel
module unload intel
```

MPI installations

There are two MPI installations on Abel:

- OpenMPI (compiled with four different compilers)
  `module load openmpi.gnu` or
  `module load openmpi.intel` or
  `module load openmpi.open64` or
  `module load openmpi.pgi`
- Intel's MPI (compiled with two different compilers)
  `module load intelmpi.gnu` or
  `module load intelmpi.intel`

Compiling MPI code

- First, remember to load MPI by, for example,
  `module load openmpi.gnu`
- Then, the `mpicc` (or `mpicxx`) command can be used to compile MPI code

Job script

Remember: you shouldn't use Abel in an interactive mode!

You must write a job script for running a compiled code, for example

```
#!/bin/bash
#SBATCH --job-name=YourJobname
#SBATCH --account=ln0001k
# Wall clock limit:
#SBATCH --time='00:05:00'
# Number of MPI processes:
#SBATCH --ntasks=4
# Max memory usage per MPI process:
#SBATCH --mem-per-cpu=100m

mpirun ./a.out
```

You must use `ln0001k` as the project account

`www.uio.no/english/services/it/research/hpc/abel/help/user-guide/`

Queue system

- SLURM is the queue system on Abel
- Basic commands
  `sbatch jobscript` – submitting a job
  `squeue -u username` or `squeue -j jobID` – inspecting job(s)
  `scancel jobID` – terminating job
- For more info about SLURM
  `https://computing.llnl.gov/linux/slurm/documentation.html`

Using OpenMPI on your own laptop

OpenMPI

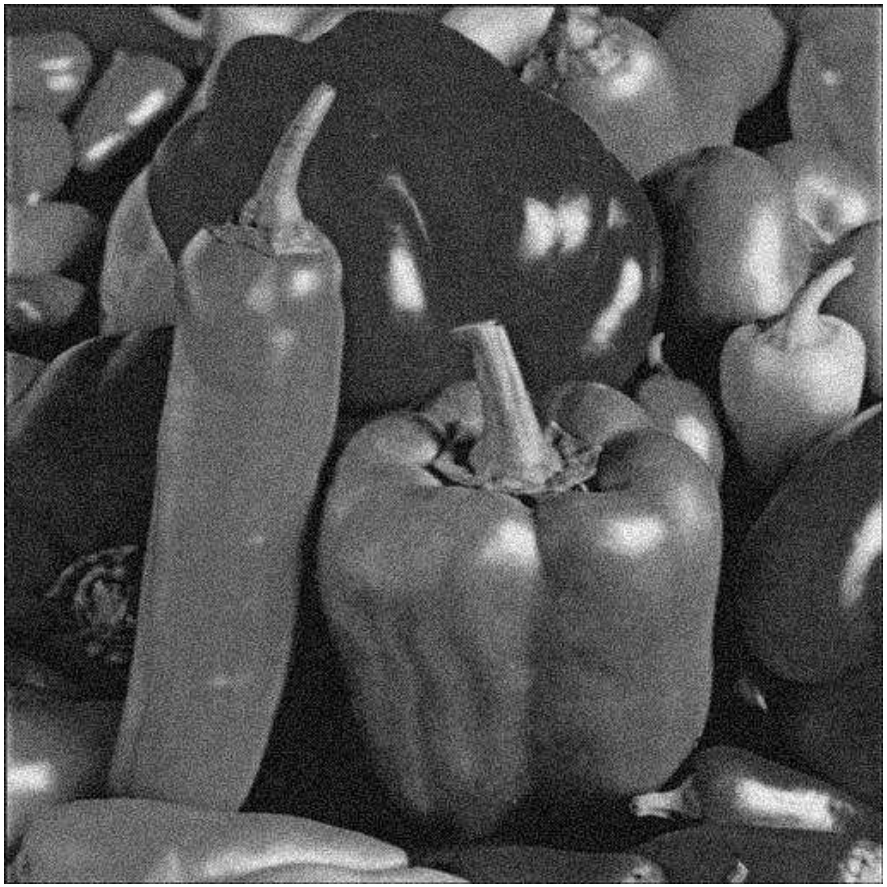
- OpenMPI (`http://www.open-mpi.org`) is an open source MPI implementaion
- Source code for download:
  `http://www.open-mpi.org/software/ompi/`
- OpenMPI can be installed on laptops using Linux or OS X
  You can turn your own laptop into an MPI-enabled parallel computer!

Mandatory assignment 1

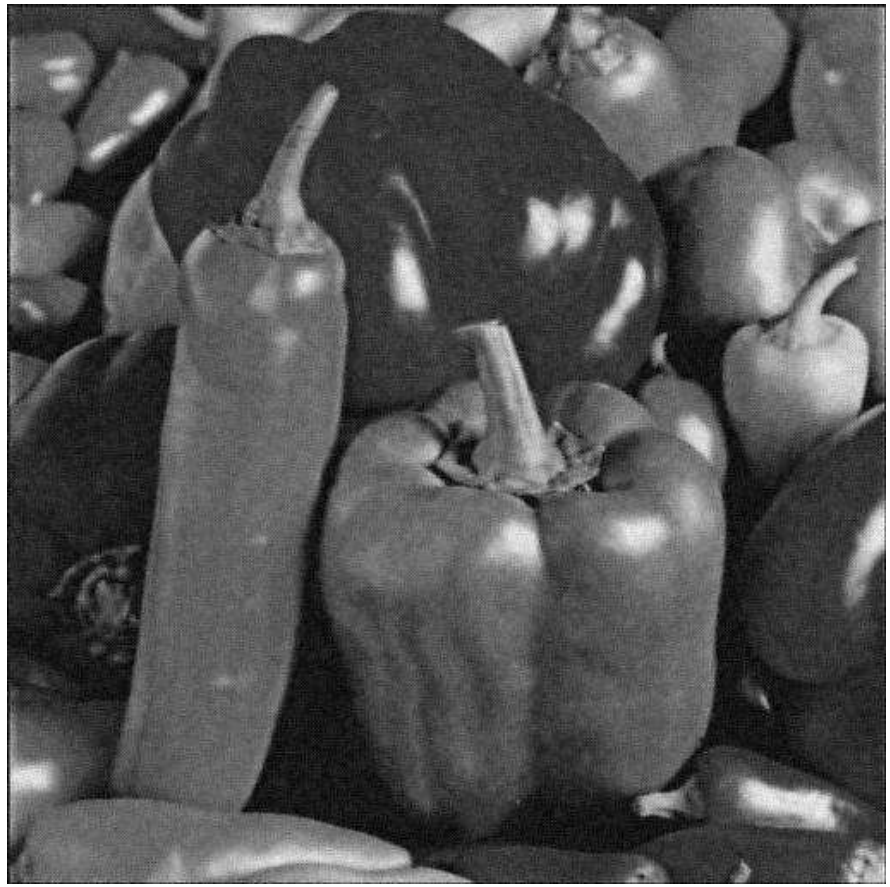
Objectives

- Translation of simple mathematical formulas to a working code
- Compilation of existing C source codes into an external library
- Implementation a simple denoising algorithm
- Parallelization of the denoising algorithm via MPI programming

Denoising



An image with noise



A denoised image

An image as a 2D array

An image can be thought as a 2D array, containing $m \times n$ pixels,

$$\mathbf{u} = \begin{bmatrix} u_{m-1,0} & u_{m-1,1} & \cdots & u_{m-1,n-1} \\ \vdots & \vdots & \vdots & \vdots \\ u_{1,0} & u_{1,1} & \cdots & u_{1,n-1} \\ u_{0,0} & u_{0,1} & \cdots & u_{0,n-1} \end{bmatrix}$$

A simple denoising algorithm

We can apply a few iterations of *isotropic diffusion*, where each iteration computes a new image $\bar{\mathbf{u}}$ as a "smoothed" version of $\mathbf{u}$. Each pixel of $\bar{\mathbf{u}}$ is calculated as

$$\bar{u}_{i,j} = u_{i,j} + \kappa \left(u_{i-1,j} + u_{i,j-1} - 4u_{i,j} + u_{i,j+1} + u_{i+1,j} \right)$$

κ is typically a small constant (such as 0.1)

Remarks

- The formula for $\bar{u}_{i,j}$ is applicable only for the interior pixels, that is, $1 \leq i \leq m-2$ and $1 \leq j \leq n-2$
- The boundary pixels of $\bar{\mathbf{u}}$ should simply copy the corresponding boundary pixels of $\mathbf{u}$
- Before going into a new iteration, we need to copy $\bar{\mathbf{u}}$ back to $\mathbf{u}$

Compiling an external C library

- We want to make use of an external C library for reading/writing JPEG images
- This external library exists as a set of header (`*.h`) files and C (`*.c`) files
  `http://heim.ifi.uio.no/xingca/inf-verk3830/simple-jpeg.ta`
- To prepare the external library
  - Compile all the `*.c` files into object (`*.o`) files
  - Group the resulting object files into a static library file
    `ar rcs libsimplejpeg.a *.o`

Reading/writing JPEG images

We can use two already implemented functions from the external C library (`libsimplejpeg.a`):

```
void import_JPEG_file (const char* filename,
                       unsigned char** image_chars,
                       int* image_height, int* image_width,
                       int* num_components);
void export_JPEG_file (const char* filename,
                       const unsigned char* image_chars,
                       int image_height, int image_width,
                       int num_components, int quality);
```

- A grey JPEG image is representated as a 1D array of values of type `unsigned char`
- We need to convert this 1D array of `unsigned char` values into a 2D array of floating-point values before doing numerical computations

Data structure for an image (suitable for denoising)

```
typedef struct
{
  float** image_data;  /* a 2D array of floats */
  int m;               /* # pixels in x-direction */
  int n;               /* # pixels in y-direction */
}
image;
```

Serial implementation

```
int main(int argc, char *argv[])
{
  int m, n, c, iters; float kappa;
  image u, u_bar;
  unsigned char *image_chars;

  /* ... */
  import_JPEG_file(input_jpeg_filename, &image_chars, &m, &n, &c);

  allocate_image (&u, m, n);
  allocate_image (&u_bar, m, n);
  convert_jpeg_to_image (image_chars, &u);

  iso_diffusion_denoising (&u, &u_bar, kappa, iters);

  convert_image_to_jpeg (&u_bar, image_chars);
  export_JPEG_file(output_jpeg_filename, image_chars, m, n, c, 75);

  deallocate_image (&u);
  deallocate_image (&u_bar);
```

Parallel implementation

- MPI programming
- Process 0 is responsible for reading the input noisy image
- Process 0 divides the input image into pieces, each assigned to one MPI process
- Denoising computation is done by all the MPI processes in parallel, with needed collaboration
- Finally, each MPI process sends its denoised piece back to process 0
- Process 0 “stiches” the pieces together as a whole image
- Process 0 writes the whole image back to file