Parallel Image Analysis

Lab session 4:
Distributed memory parallelization using MPI

In this lab you will be learning how to work with compiling and running MPI programs at UPPMAX. You will use the Kalkyl cluster. The example programs used can be found here:
`http://www.cb.uu.se/~cris/ParallelImageAnalysis/lab4.zip`

The ZIP file contains four directories:

```
example1
example2
example3
example4
```

The first three examples contain small MPI programs that you will compile and run. The fourth example contains a serial program, an image filtering program similar to the one used in previous labs. There, your lab exercise will be to parallelize the filtering program using MPI.

1 Compiling and running MPI programs

In this lab you will use the OpenMPI implementation on the Kalkyl cluster att UPPMAX, together with the GNU compiler gcc. Load the modules like this:

```
module load gcc openmpi
```

You will then get a message like this:

```
mod: loaded OpenMPI 1.4.3, compiled with gcc4.4 (found in /opt/openmpi/1.4.3gcc4.4/)
```

Each of the example programs has a Makefile, so to compile each program just execute "make" in that directory.

You can execute MPI programs either by using `mpirun` directly on the login node or by submitting a job through the queueing system. See the lecture notes for further instructions.

2 Exercises

2.1 Exercise 1: compiling and running a simple MPI program, without communication

The files needed for this exercise are in the directory "example1" included in the zip file.

The test program is called "mpitest1". It initializes MPI, gets information about the rank of the current process and the total number of processes, and calls `MPI_Get_processor_name` to get the name of the compute node where the process is running.

Start by looking through the source code and try to understand what the program is doing.

Compile the program:

```
make
```

Verify that the executable file "mpitest1" was generated.

Then run the program on the login node, using for example 3 processes:

```
mpirun -np 3 mpitest1
```

After running the program, look at the files that were generated by the program. Each process should have created its own file, and in the file you can read information about which computer that process was running on. Now, since you ran on the login node, all processes are expected to run on the same node, so you should see the name of the login node (kalkyl1 or kalkyl4) in all of the files.

Nest, try running the program by submitting a job to the queueing system. See the lecture notes or the Kalkyl user guide for instructions on how to write a job script, then submit the job script using the command sbatch. Submit a few such jobs asking for different numbers of nodes/processes and verify by looking in the output files that the program was really run on several different nodes.

2.2 Exercise 2: compiling and running a simple MPI program, with communication

The files needed for this exercise are in the directory "example2" included in the zip file.

The test program is called "mpitest2". It is supposed to be run using only two processes, and performs simple point-to-point communication calls between those two processes.

Start by looking through the source code and try to understand what the program is doing. Then compile and run the program:

```
make
mpirun -np 2 mpitest2
```

As in the first example, each process generates an output file. Look in the output files to see the reports of what each process was doing.

Try changing the program in some way, run it again and check that it behaves as you expected. For example, try changing the tag in a send call so that it does not match the tag in the corresponding receive call. What happens then?

2.3 Exercise 3: compiling and running an MPI program and measuring speedup

The files needed for this exercise are in the directory "example3" included in the zip file.

The test program is called "mpitest3". It starts by generating a list of random numbers, and then uses a parallel sort algorithm to sort the numbers in ascending order. The program takes one input argument, giving the number of items (numbers) to sort. The larger the input argument, the longer time it will take to sort the list.

First compile the program and run it on the login node for a small number of processes and a short list to sort. For example, using 4 processes and a list length of 20:

```
make
mpirun -np 4 mpitest3 20
```

Verify that the output from the program looks OK.

Next, run the program using 1, 2, 4, 8, 16, 32, and 64 processes by submitting jobs to the queueing system. Choose a list length such that it takes around 30 seconds to run the program with only one process (maybe a list length of 100 million), and see how much faster the program becomes when using different numbers of processes.

Note that it takes some time for the program to generate the list of random numbers (see the output line "Setting up list of xxx random numbers took yyy wall seconds"). Therefore, to measure only the time for the sort operation you should look at the timing reported at the output line "Parallel sort done! Took yyy wall seconds".

What is the best speedup you can achieve?

2.4 Exercise 4: parallelizing an image filtering program using MPI

The files needed for this exercise are in the directory "example4" included in the zip file.

This time, the given example source code is a serial program. The program is very similar to the filtering program used in Lab session 2 of this course, but slightly modified to make it easier to parallelize using MPI. This version of the program should produce exactly the same results as the filtering program used in Lab session 2.

Start by compiling and running the serial program. Then, parallelize the program using MPI.

Hint: the function `filter_gauss_one_direction` is called for independent parts of the image, so one approach to parallelize is to make different MPI processes call `filter_gauss_one_direction` for different parts of the image.

Important information

We have several nodes on Kalkyl reserved during each of the lab sessions. To use the reserved nodes (and not have to wait in the queue with all the people doing actual work), add `--reservation=g2011040` to your `sbatch` and `interactive` commands. Thus, to submit a job to the queueing system:

```
sbatch --reservation=g2011040 yourjobscript
```

The project number for this course is `g2011040`. This goes after the `-A` option of the `sbatch` and `interactive` commands.