Introduction to Parallel Programming with MPI and OpenMP Lab

Stéphane Zuckerman

Preliminaries

Where to find the course slides: Go to `https://perso-etis.ensea.fr/zuckerman/teaching.html`. The slides for both MPI and OpenMP are available. There is also a link entitled "Additional source files" (accessible at the address: `https://perso-etis.ensea.fr/zuckerman/TP_ParallelProg`). You can build the example program by calling `make`:

```
~/Programming/C/filtre_image$ make mrproper
rm -f *.o
rm -f convolution_filter
~/Programming/C/filtre_image$ make
gcc -g -Wall -Wextra -pedantic -Wimplicit-fallthrough=0 -O3 -std=c11 -fopenmp -o main.o -c ma
gcc -g -Wall -Wextra -pedantic -Wimplicit-fallthrough=0 -O3 -std=c11 -fopenmp -o img_convolut
gcc -g -Wall -Wextra -pedantic -Wimplicit-fallthrough=0 -O3 -std=c11 -fopenmp -o img_mgmt.o -
gcc -g -Wall -Wextra -pedantic -Wimplicit-fallthrough=0 -O3 -std=c11 -fopenmp -o convolution_
```

Building an MPI program.

```
mpicc $CFLAGS -o $PROGRAM source1.c source2.c # etc.
```

... Where `$PROGRAM` is the desired name for the resulting application, and `$CFLAGS` contains the usual compilation flags. I suggest you use the following:

```
mpicc -Wall -Wextra -pedantic -g -std=c11 -o $PROGRAM source1.c source2.c # etc.
```

Running an MPI program.

```
mpirun -oversubscribe -np $NPROCS $PROGRAM
```

... where `$NPROCS` is the number of desired MPI processes for this computation. Note: `oversubscribe` allows you to create more MPI processes than available cores. It is usually not a good idea, as it creates more virtual tasks than hardware processors are able to run simultaneously.

1 MPI

1.1 Basic MPI Programming

1. Build an MPI program where each process sends a "token" to an MPI process where its rank is one above. Each process must print its rank number before passing the token to the next process. If the process with the highest rank receives the token, it sends it back to process zero, and the program ends.

2. Implement a dot product (see explanations below) using only `MPI_Send` and `MPI_Recv`.

3. Implement a matrix-vector product (we provide the sequential code to compute it at the end of this handout). Use `MPI_Send` and `MPI_Receive` to distribute data from the "master" process (rank 0) to the other processes. The master process (rank 0) is in charge of initializing all the data beforehand.

1.2 Intermediate MPI Programming

1. Copy your dot product program in a new file and modify it to use `MPI_Reduce`.

2. Copy your matrix-vector program in a new file, and modify it so that the `b` vector is broadcast to all MPI processes using the `MPI_Bcast` function call.

3. Copy the previous program into a new file, and modify it so that instead of using `MPI_Send` and `MPI_Recv`, your program makes use of `MPI_Scatter` and `MPI_Gather`. Note: make sure the number of rows in matrix `A` is divisible by the number of MPI processes involved in the computation.

4. Optional (if you're done with everything else, including the OpenMP exercises below): modify your program so that it can use matrices which do not feature a number of rows divisible by the number of MPI processes (hint: use `MPI_Scatterv` and `MPI_Gatherv`).

2 OpenMP

1. Build an OpenMP program where each thread sends a "token" to another OpenMP thread where its ID is one above. Each thread must print its rank number before passing the token to the next thread. If the thread with the highest rank receives the token, it sends it back to thread zero, and the program ends.

2. Implement three variants of the dot product (see explanations below):

   (a) Using the `critical` directive

   (b) Using the `atomic` directive

   (c) Using the `reduction` clause (see `http://www.openmp.org` to get the latest specification and see how to use `reduce`).

3. Implement a matrix-vector product (we provide the sequential code to compute it at the end of this handout). Use `omp parallel` (with the `default(none)` clause) and `omp for` to distribute data across the threads. The various data structures must be initialized in the sequential part of the program.

3 Stencil Codes

Stencil codes are used in many scientific computations, *e.g.*, to compute partial differential equations (such as heat diffusion on a plate using Laplacians, or computational fluid dynamics), or perform image processing (where one pixel is modified according to its neighbors). Usually the computation stops when the solution converges toward a good-enough precision, or it has reached a specific number of computation steps. We show the code for a 5-point 2D stencil below.

```
static inline void swap(void** p1, void** p2) {
    void* tmp = *p1;
    *p1 = *p2;
    *p2 = tmp;
}

void
FivePointStencil(int tsteps, int n_rows, int n_cols,
                 double *p_old, double *p_new)
{
    double (*m_old)[n_cols] = (double (*)[n_cols]) p_old,
           (*m_new)[n_cols] = (double (*)[n_cols]) p_new;

    while (tsteps-- > 0) {
        for (int i = 1; i < n_rows-1; ++i)
            for (int j = 1; j < n_cols-1; ++j)
                m_new[i][j] = ( m_old[i-1][j] + m_old[i+1][j]
                              + m_old[i][j-1] + m_old[i][j+1] )
                            / 4;
        swap(m_old,m_new);
    }
}
```

Adapt the sequential code to make it work in OpenMP then MPI.

Appendix

Dot Product

The dot product (also called the *scalar product*) takes two vectors $V1_n$ and $V2_n$, and returns the sum of the products of corresponding elements in $V1$ and $V2$:

$$s = V1 \cdot V2 = \sum_{i=1}^{n} v_{1_i} v_{2_i}$$

The corresponding code is given below.

```
double dotproduct(double *v1, double *v2, int n)
{
    double sum = 0.0;
    for (int i = 0; i < n; ++i)
        sum += v1[i]double * v2[i];
    return sum;
}
```

Matrix-Vector Product

A matrix-vector multiplication is defined as the product of a matrix $A_{M,N}$ with a column vector b_N. The result is stored in a row vector c_M: $c_M = A_{M,N} \times b_M$. The sequential code to compute such a product is given below. Each computed element of c is the result of a dot product between a row in A and the column vector b.

```
#include <string.h>
void matvec(int n_rows, int n_cols, int max_rows, int max_cols,
```

```
            double A[max_rows][max_cols],
            double b[max_cols], double c[max_rows])
{
    memset(c,0,sizeof(double)*n_rows);
    for (int i = 0; i < n_rows; ++i)
        for (int j = 0; j < n_cols; ++j)
            c[i] += A[i][j] * b[j];
}
```