Supercomputing

Hands-On 3: Portable Parallel Programming with MPI

2022

Contents

Introduction

This Hands-on comprises 3 sessions. Next table shows the documents and files needed to develop each one of the exercises.

Session 1	Basic Operations	`operations.c`
Session 2	Algebraic Function	`function.c`
Session 3	Tridiagonal Matrix	`tridiagonal.c`

Please remind that in order to compile MPI programs, we should include the proper compilation option, such as:

```
$ mpicc code.c -o object
```

To execute an MPI program with several threads, you can use the following example (changing the number of processes accordingly):

```
$ mpirun -np 4 ./object
```

1 Basic Operations

The Algorithm 1 below solves the multiplication, addition and subtraction of the elements of a vector of integers. The variable array is the vector on which the operations will be performed. Then, modify the program to run in parallel using MPI. Present the primitives used. The idea is made the following MPI version with only 4 processes running. In the version, each process does a function: 1 add, 1 subtract and 1 multiplies. The other process is responsible for telling each of the other 3 its function, and when finished printing the results.

```
#include <stdio.h>
#define SIZE 12

int main (int argc, char **argv){

int i, sum = 0, subtraction = 0, mult = 1;
int array[SIZE];

for(i = 0; i < SIZE; i++)
 array[i] = i + 1;

for(i = 0; i < SIZE; i++)
 printf("array[%d] = %d\n", x, array[x]);

for(i = 0; i < SIZE; i++) {
 sum = sum + array[i];
 subtraction = subtraction - array[i];
 mult = mult * array[i];
}

printf("Sum = %d\n", sum);
printf("Subtraction = %d\n", subtraction);
printf("Multiply = %d\n", mult);

return 0;

}
```

Figure 1: Sequential code for the computation of the basic operations.

1.1 Initialize the MPI

The next step will be to modify the code in the file `sequential-basic-operations.c` to link the command with the MPI library by including `mpi.h` simultaneously with the variables. And initialize the MPI with the commands: `MPI_Init, MPI_Comm_size, MPI_Comm_rank, MPI_Status`.

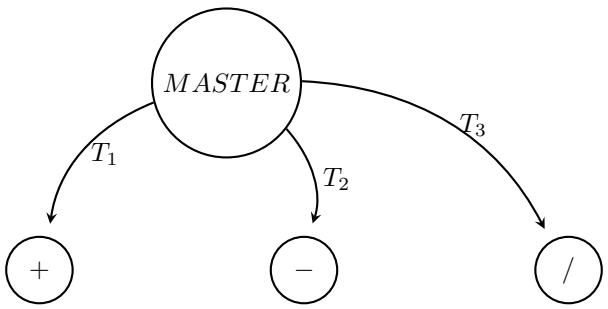
```
#include <mpi.h>
...
int  array[SIZE];
char operations[] = {'+', '-', '*'};
char operationsRec;
int  numberOfProcessors, id, to, from, tag = 1000;
int  i, result, value;
...
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numberOfProcessors);
MPI_Comm_rank(MPI_COMM_WORLD, &id);
MPI_Status status;
...
```

Figure 2: Parallel code for MPI initialize.

1.2 Master send data for workers

```
...
switch(id){
case 0:
  for(i = 0; i < SIZE; i++){
        array[i] = i + 1;
        printf("%d\t", i, array[i]);
  }
  printf("\n");

  for(to = 1; to < numberOfProcessors; to++) {
        MPI_Send(&array, SIZE, MPI_INT, to, tag, MPI_COMM_WORLD);
        MPI_Send(&operations[to-1], 1, MPI_CHAR, to, tag, MPI_COMM_WORLD);
  }
...
```

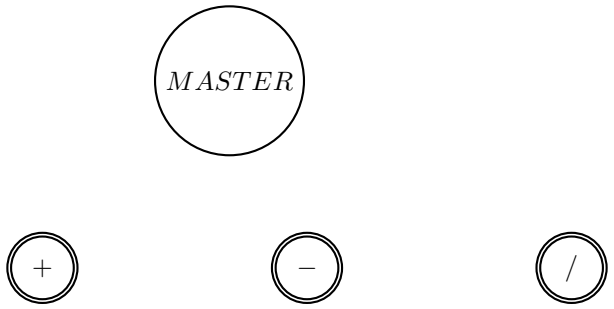
Figure 3: Parallel code for the computation of the basic operations.



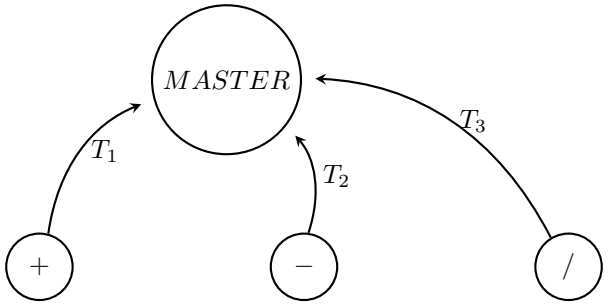
1.3 Workers makes partial processing

```
default:
        MPI_Recv(&array, SIZE, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
        MPI_Recv(&operationsRec, 1, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);

        switch (operationsRec) {
                case '+':
                        value = 0;
                        for(i = 0; i < SIZE; i++)
                          value += array[i];
                        break;
                case '-':
                        value = 0;
                        for(i = 0; i < SIZE; i++)
                          value -= array[i];
                        break;
                case '*':
                         value = 1;
                         for(i = 0; i < SIZE; i++)
                          value *= array[i];
                         break;
        }
```

Figure 4: The shell script for automatizing the execution code.



1.4 Workers send partial solution to master



```
/*Return to the Master*/
MPI_Send(&value, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
MPI_Send(&operationsRec, 1, MPI_CHAR, 0, tag, MPI_COMM_WORLD);
```

Figure 5: The shell script for automatizing the execution code.

1.5 Master computes the final solution

```
...
for(to = 1; to < numberOfProcessors; to++) {
  MPI_Recv(&result, 1, MPI_INT, to, tag, MPI_COMM_WORLD, &status);
  MPI_Recv(&operationsRec, 1, MPI_CHAR, to, tag, MPI_COMM_WORLD, &status);
  printf ("(%c) = %d\n", operationsRec, result);
}
...
}
```

Figure 6: The shell script for automatizing the execution code.

2 Algebraic Function

The idea of this Hands-On is to make an algorithm that uses the `MPI_Recv` and `MPI_Send` routines in the Master-Worker Paradigm in such a way that in the sequential code:

The Master-Worker Paradigm in such a way that in the scheme:

1. Master:
   - Create the processes;
   - Shows the format of the function $f(x) = a * x^3 + b * x^2 + c * x + d$;

     Note: Asks for the value of x;

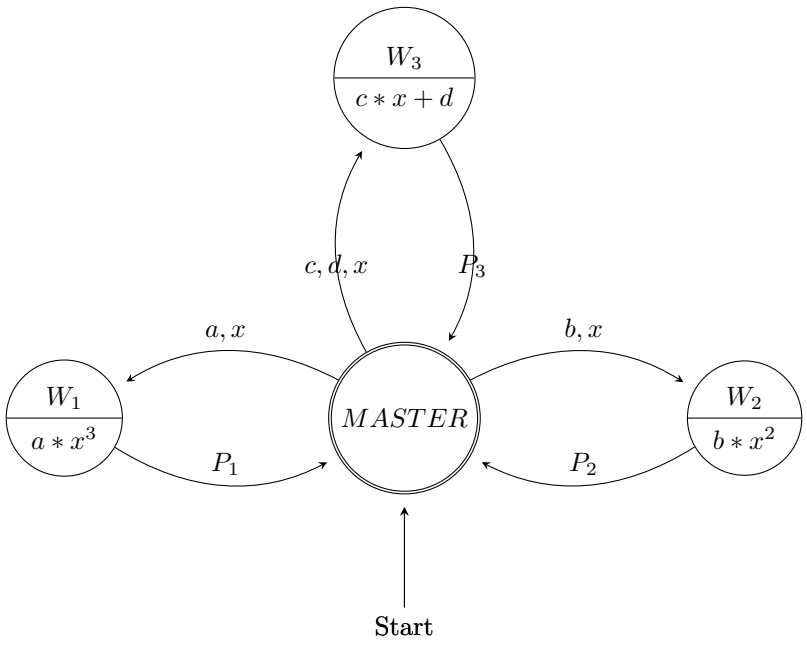
     Note: Ends the values of a, b, c and x to the workers.
2. Worker:
   - Calculate the function and return value to the master;

     Note: In the end the Master shows the result of the function.

Below we can show the state diagram with the data movement between the Master and the Workers:

```
#include <stdio.h>

int main (int argc, char **argv){

double coef[4], total, x;
char c;

printf ("\nf(x) = a*x^3 + b*x^2 + c*x + d\n");

for(c = 'a'; c < 'e'; c++) {
 printf ("\nEnter the value of the 'constants' %c:\n", c);
 scanf ("%lf", &coef[c - 'a']);
}

printf("\nf(x)=%lf*x^3+%lf*x^2+%lf*x+%lf\n", coef[0], coef[1], coef[2], coef[3]);

printf("\nEnter the value of 'x':\n");
scanf("%lf", &x);

total = (coef[0]* x * x * x) + (coef[1]* x * x) + (coef[2]* x + coef[3]);

printf("\nf(%lf) = %lf*x^3 + %lf*x^2 + %lf*x + %lf = %lf\n",
x, coef[0], coef[1], coef[2], coef[3], total);

return 0;
}
```

Figure 7: The sequential code for the algebraic function.



3 Tridiagonal Matrix

The following algorithm initializes a number of processes to calculate the addition of constants (k_1, k_2, k_3) to an array structure. The values must be added respectively to the main **diagonal**, **superdiagonal** and **subdiagonal** elements of a symmetric tridiagonal matrix, as in the following example:

$$M = \begin{pmatrix} a & e & 0 & 0 \\ e & b & f & 0 \\ 0 & f & c & g \\ 0 & 0 & g & d \end{pmatrix} \Longrightarrow \begin{pmatrix} a+k_1 & e+k_3 & 0 & 0 \\ e+k_2 & b+k_1 & f+k_3 & 0 \\ 0 & f+k_2 & c+k_1 & g+k_3 \\ 0 & 0 & g+k_2 & d+k_1 \end{pmatrix}$$

```
#include <stdio.h>
#define ORDER 4

void printMatrix (int m[][ORDER]) {
  int i, j;
  for(i = 0; i < ORDER; i++) {
    printf ("| ");
    for (j = 0; j < ORDER; j++) {
      printf ("%3d ", m[i][j]);
    }
    printf ("|\n");
  }
  printf ("\n");
}

int main (int argc, char **argv){
  int k[3] = {100, 200, 300};
  int matrix[ORDER][ORDER], i, j;

  for(i = 0; i < ORDER; i++) {
    for(j = 0; j < ORDER; j++) {
      if( i == j )
        matrix[i][j] = i + j +1;
      else if(i == (j + 1)) {
        matrix[i][j] = i +  j + 1;
        matrix[j][i] = matrix[i][j];
      } else
           matrix[i][j] = 0;
     }
  }
  printMatrix(matrix);

  for(i = 0; i < ORDER; i++){
     matrix[i][i]     += k[0];    //main diagonal
     matrix[i + 1][i] += k[1];    //subdiagonal
     matrix[i][i + 1] += k[2];    //superdiagonal
  }
  printMatrix(matrix);

  return 0;
}
```

Figure 8: Sequential code for the computation of the tridiagonal matrix.

References

[1] F. Almeida, D. Giménez, J. M. Mantas and A. M. Vidal. *Introducción a la Programación Paralela*. Ed. Paraninfo, 2008, Spain.

[2] Forum, Message Passing Interface. *MPI: A Message-Passing Interface Standard*. University of Tennessee, 1994, USA.