Supercomputing

Hands-On 1: Portable Parallel Programming with OpenMP

Contents

1 Matrix Multiple Benchmark **1**
- 1.1 Entering time measurement metrics 3
- 1.2 Inserting the OpenMP directive 3
- 1.3 Performance Analysis 4

2 Asynchronous Task **4**

Introduction

This Hands-on comprises 2 sessions. Next table shows the documents and files needed to develop each one of the exercises.

Session 1	Matrix Multiple	`mm.c`
Session 2	Asynchronous Task	`asyncTaskOpenMP.c`

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

```
$ gcc mm.c -o mm -fopenmp
```

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

```
$ OMP_NUM_THREADS=4 ./mm 100
```

1 Matrix Multiple Benchmark

The definite algebraic operation of the matrix can be defined as:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + ... + a_{in} b_{nj}.$$

where i is summed over for all possible values of j and k and the notation above uses the summation convention. The sequential code of the program is available in the file `mm.c`. Figure 1 shows an extract of such code. In particular, we can see the algebraic operation include a loop that implements the summatory of the above definition.

```
#include <stdio.h>
#include <stdlib.h>

void initializeMatrix(int *matrix, int size)
{
  for(int i = 0; i < size; i++)
    for(int j = 0; j < size; j++)
      matrix[i * size + j] = rand() % (10 - 1) * 1;
}

void printMatrix(int *matrix, int size)
{
  for(int i = 0; i < size; i++)
  {
    for(int j = 0; j < size; j++)
      printf("%d\t", matrix[i * size + j]);
    printf("\n");
  }
  printf("\n");
}

int main (int argc, char **argv)
{
 int size = atoi(argv[1]);
 int i, j, k;

 int  *A = (int *) malloc (sizeof(int)*size*size);
 int  *B = (int *) malloc (sizeof(int)*size*size);
 int  *C = (int *) malloc (sizeof(int)*size*size);

 initializeMatrix(A, size);
 initializeMatrix(B, size);

 for(i = 0; i < size; i++)
  for(j = 0; j < size; j++)
    for(k = 0; k < size; k++)
       C[i * size + j] += A[i * size + k] * B[k * size + j];

 printMatrix(A,size);
 printMatrix(B,size);
 printMatrix(C,size);

 free(A);
 free(B);
 free(C);

 return 0;
}
```

Figure 1: Sequential code for the computation of the matrix multiple.

First, compile and execute the program. At run time, an argument can be used to pass the size problem for the algorithm. For example, to use the size problem for a matrix order 100 you can use:

```
$ ./mm 100
```

1.1 Entering time measurement metrics

The next step will be to modify the code in the file `mm.c` to enter time measurement metrics of the matrix multiply in parallel using OpenMP. A first approach could be to use the command `omp_get_wtime()` for the get the initial and final time. You will need to link the command with the OpenMP library by including `omp.h`.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
...
double t1, t2;
...
t1 = omp_get_wtime();
for(i = 0; i < n; i++)
  for(j = 0; j < n; j++)
    for(k = 0; k < n; k++)
        C[i * n + j] += A[i * n + k] * B[k * n + j];
t2 = omp_get_wtime();

printf("%d\t%f\n",n, t2-t1);
...
```

Figure 2: Sequential code for the computation of the matrix multiply inserting the ime measurement metrics.

1.2 Inserting the OpenMP directive

The next step will be to modify the code in the file `mm.c` to perform the computation of the integral in parallel using OpenMP. A first approach could be to use the directive `parallel for` considering the variables i, j and k are `private`. After this change, you can compile and execute the program.

```
...
t1 = omp_get_wtime();
#pragma omp parallel for private(i, j, k)
  for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
       for(k = 0; k < n; k++)
          C[i * n + j] += A[i * n + k] * B[k * n + j];
t2 = omp_get_wtime();

printf("%d\t%f\n",n, t2-t1);
...
```

Figure 3: Parallel code for the computation of the matrix multiply.

1.3 Performance Analysis

The last step will be to create the shell script file to measure the perform the computation of the matrix multiply in parallel using OpenMP. The shell script is available in the file `script.sh`.

```
#!/bin/sh
for i in 100 200 300 400 500 600 700 800 900 1000
do
 printf "\033[1D$i :"
 OMP_NUM_THREADS=$1  ./mm  "$i"
done
```

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

Compile and execute the script. At run time, an argument can be used to select the number of the threads. For example, to use the first variant you can use for 4 threads:

```
$ bash script.sh 4
```

Through experiments, the following questions should be answered:

- What is the behavior of the execution time and the speedup varying the size of the problem? (Show the solution with tabular and graphical values).
- What is the optimal number of threads for the best parallel solution?

2 Asynchronous Task

Asynchronous programming is a set of techniques for implementing expensive operations that run concurrently with the rest of the program. One domain where asynchronous programming is often used is in programs with a graphical user interface: it is often unacceptable when the user interface freezes while performing a costly operation. Also, asynchronous operations are essential for parallel applications that need to run multiple tasks simultaneously. The following is a code (`asyncTaskOpenMP.c`) that represents a task being done asynchronously. Before understanding the code, compile and run it as follows:

```
$ gcc asyncTaskOpenMP.c -o asyncTaskOpenMP -fopenmp
$ ./asyncTaskOpenMP  10  2
```

From the previous action answer the following questions:

- What does the code do from the compilation and execution of the previous code?
- How would it be possible to extend the code so that the five threads perform asynchronous tasks?

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define SIZE_MATRIX 10

int main(int argc, char **argv)
{
  int n = atoi(argv[1]);
  int block_size = atoi(argv[2]);
  int matrix[SIZE_MATRIX][SIZE_MATRIX], k1 = 10, k2 = 20;
  int i, j, row, column;

  for(i = 0; i < n; i++)
  {
    for(j = 0; j < n; j++)
    {
      matrix[i][j] = 5;
      printf("%d\t", matrix[i][j]);
    }
    printf("\n");
  }

  printf("\n\n");

  omp_set_num_threads(5);

  #pragma omp parallel private (row, column)
  {
    int id = omp_get_thread_num();

    if(id == 0)
    {
      for(row = 0; row < n; row++)
        for(column = 0; column < block_size; column++)
          matrix[row][column] *= k1;
    }
    if(id == 1)
    {
      for(row = 0; row < n; row++)
        for(column = block_size; column < 2 * block_size; column++)
          matrix[row][column] *= k2;
    }
  }

  for(i = 0; i < n; i++)
  {
    for(j = 0; j < n; j++)
      printf("%d\t", matrix[i][j]);
    printf("\n");
  }
  return 0;
}
```

Figure 5: Parallel code for the asynchronous computation using OpenMP.

References

[1] M. Boratto. *Hands-On Supercomputing with Parallel Computing*. Available: `https://github.com/muriloboratto/Hands-On-Supercomputing-with-Parallel-Computing`. 2022.

[2] B. Chapman, G. Jost and R. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007, USA.