Saint Petersburg National Research University of Information Technologies, Mechanics and Optics (ITMO University)
Faculty of Informational Technologies and Programming

REPORT

about laboratory work № 1

« MatrixMatrix Multiplication in OpenMP»

Student
Nerinovsky Arseny **_M41331_**
(Surname, initials) Group

Saint-Petersburg, 2018

Report

June 29, 2019

1 GOAL OF LABORATORY WORK

The Goal was to show how matrix multiplication can be speeded with the same precision and results up with the help of Multithreaded architecture.

2 TASK DEFINITION

Matrix Multiplication is done by a single threaded architecture and using OpenMP which supports multithread architecture. We first calculated the time taken by the single thread architecture about how much time it takes for a single threaded application to calculate the matrix multiplication of two matrix of large size (test size = 1000 x 1000). And then we implement the same with the help of multithreaded system and we drew the results.

3 BRIEF THEORY

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran on most platforms, instruction set architectures and operating systems, including Solaris, AIX, HP-UX, Linux, macOS, and Windows. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior OpenMP is managed by the nonprofit technology consortium OpenMP Architecture Review Board (or OpenMP ARB), jointly defined by a group of major computer hardware and software vendors, including AMD, IBM, Intel, Cray, HP, Fujitsu, Nvidia, NEC, Red Hat, Texas Instruments, Oracle Corporation, and more OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer. An application built with the hybrid model of parallel programming can run on a computer cluster using both OpenMP and Message Passing Interface (MPI), such that OpenMP is used for parallelism within a (multi-core) node while MPI is used for parallelism between nodes. have also been efforts to run OpenMP on software distributed shared memory systems, to translate OpenMP into MPI and to extend OpenMP for non-shared memory systems.

4 ALGORITHM (METHOD) of IMPLEMENTATION

[23]:
```
%cat mm.c
```

```
#include <omp.h>

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

#ifndef M
# define M 10
#endif

#ifndef N
# define N 14
#endif

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

        float **A, **B, **C;

        A = (float**)malloc(sizeof(float*)*M);
        for(int i=0; i < M; ++i)
                A[i] = (float*)malloc(sizeof(float)*N);

        B = malloc(sizeof(float*)*N);
        for(int i=0; i < N; ++i)
                B[i] = malloc(sizeof(float)*M);

        C = malloc(sizeof(float*)*M);
        for(int i=0; i < M; ++i)
                C[i] = malloc(sizeof(float)*M);


        #pragma omp parallel for
        for(int i=0; i < M; ++i) {
                for(int j=0; j < N; ++j) {
                        A[i][j] = i*j + 1; //rand();
                        B[j][i] = i*100 + j; // rand();
                }
        }

        #pragma omp parallel for
        for(int i=0; i < M; ++i) {
                for(int j=0; j < M; ++j) {
                        C[j][i] = 0;
```

```
                }
        }



#define LOOP \
for(int i=0; i < M; ++i) { \
        for(int j=0; j < M; ++j) { \
                for(int k=0; k < N; ++k) { \
                        C[i][j] += A[i][k] * B[k][j]; \
                } \
        }       \
}
        #pragma omp parallel
        {

#if defined _DYNAMIC
        //printf("# DYNAMIC SCHEDULE\n");
        #pragma omp for schedule(dynamic, CHUNK)
        LOOP
#elif defined _STATIC
        //printf("# STATIC SCHEDULE\n");
        #pragma omp for schedule(static,CHUNK)
        LOOP
#elif defined _GUIDED
        // printf("# GUIDED SCHEDULE\n");
        #pragma omp for schedule(guided)
        LOOP
#else
        //printf("# NO SCHEDULE\n");
        #pragma omp for
        LOOP
#endif

        }

        //printf("C: ");
        //print_raveled_mx(C, M, M);

        return 0;
```

5 RESULT AND EXPERIMENTS

Task:

1. Implement Matrix Multiplication:

1. Using SPMD Pattern
2. Using OpenMP loop directives with different schedule (type[, chunk]) configurations.

2. Do speedup tests:
    1. with different amount of data
    2. with different number of threads
3. Write a report that should include: task definition, brief theory, implementation description, test results (tables, graphics...)

[8]:
```
#
# Helper functions
#

import subprocess
import os

def compile(*defs, **defskw):
    args = [f"-D{k}" for k in defs] + [f"-D{k}={v}" for k, v in defskw.items()]
    _cmd = 'gcc mm.c -o mm -fopenmp'.split() + args
    cmd = subprocess.run(_cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    # print(' '.join(_cmd))
    if(cmd.stdout): print('cmd.stdout', cmd.stdout)
    if(cmd.stderr): print('cmd.stderr', cmd.stderr)

def run(env=None):
    cmd = subprocess.run('./mm', stdout=subprocess.PIPE, stderr=subprocess.
 ↪PIPE, env=env)
```

5.1 Tests with diffeent amounts of threads

Lets test how the exection tim ewill change depending on the amount of threads

[24]:
```
compile(N=500, M=1000)


env = os.environ.copy()
env['OMP_NUM_THREADS'] = str(8)
print("Executing with 8 threads", end="\n\t")
%timeit run(env)
print()

env = os.environ.copy()
env['OMP_NUM_THREADS'] = str(4)
print("Executing with 4 threads", end="\n\t")
%timeit run(env)
print()
```

```
env = os.environ.copy()
env['OMP_NUM_THREADS'] = str(2)
print("Executing with 2 threads", end="\n\t")
%timeit run(env)
print()

env = os.environ.copy()
env['OMP_NUM_THREADS'] = str(1)
print("Executing with 1 thread", end="\n\t")
%timeit run(env)
print()
```

```
Executing with 8 threads
        1.08 s ś 144 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

Executing with 4 threads
        1.05 s ś 232 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

Executing with 2 threads
        1.63 s ś 49.9 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

Executing with 1 thread
        3.58 s ś 188 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)
```

5.2 Tests with diffent schedules

Let's analyze how different shedules will affect the program performance

[20]:
```
env = os.environ.copy()
env['OMP_NUM_THREADS'] = str(4)

compile('_STATIC', CHUNK=10, N=500, M=1000)


print("Executing with static schedule", end="\n\t")
%timeit run(env)
print()

compile('_DYNAMIC', CHUNK=10, N=500, M=1000)

print("Executing with dynamic schedule", end="\n\t")
%timeit run(env)
print()

compile('_GUIDED', N=500, M=1000)

print("Executing with guided schedule", end="\n\t")
```

```
%timeit run(env)
print()
```

```
Executing with static schedule
        1.04 s ś 216 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

Executing with dynamic schedule
        855 ms ś 18.5 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

Executing with guided schedule
        874 ms ś 64.2 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)
```

5.3 Tests with diffeent chunks

Lets see how the chunk size of a static and dynamic shedule affects the execution time

[12]:
```
env = os.environ.copy()
env['OMP_NUM_THREADS'] = str(4)


for ch in [2, 5, 10, 16, 20]:

    compile('_STATIC', CHUNK=ch, N=500, M=1000)

    print(f"Executing with static schedule with {ch} chunk size", end="\n\t")
    %timeit run(env)
    print()

    compile('_DYNAMIC', CHUNK=ch, N=500, M=1000)

    print(f"Executing with dynamic schedule with {ch} chunk size",end="\n\t")
    %timeit run(env)
    print()
```

```
Executing with static schedule with 2 chunk size
        958 ms ś 130 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

Executing with dynamic schedule with 2 chunk size
        883 ms ś 57.1 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

Executing with static schedule with 5 chunk size
        885 ms ś 34.5 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

Executing with dynamic schedule with 5 chunk size
        975 ms ś 23.5 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

Executing with static schedule with 10 chunk size
```

```
        1.03 s ś 34.4 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

Executing with dynamic schedule with 10 chunk size
        1.23 s ś 233 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

Executing with static schedule with 16 chunk size
        1.14 s ś 177 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

Executing with dynamic schedule with 16 chunk size
        975 ms ś 12.2 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

Executing with static schedule with 20 chunk size
        1.12 s ś 85.6 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

Executing with dynamic schedule with 20 chunk size
        1.07 s ś 91 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)
```

5.4 Tests with diffeent amounts of data

Finally lets test how the amount of data affects the execution speed

[13]:
```
env = os.environ.copy()
env['OMP_NUM_THREADS'] = str(4)
env = os.environ.copy()


compile(N=500, M=1000)
print(f"Executing with {500*1000} elements", end="\n\t")
%timeit run(env)
print()

print(f"Executing with {50*100} elements", end="\n\t")
compile(N=50, M=100)
%timeit run(env)
print()
```

```
Executing with 500000 elements
        1.15 s ś 96.5 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

Executing with 5000 elements
        5.2 ms ś 598 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)
```

6 CONCLUSION

As we can see Multi-Threading increased the speed of the running of the Application but too much multithreading might also degrade the running time and memory sharing sometimes is a

time consuming process too. So the tradeoff should be taken in mind. Overall It improves the working of the System.