INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

Spring Semester, 2020
Introduction to Parallel Scientific Computing (CSE504)

HW II
January 25, 2020

Due: **12.02.20** **Instructor: Dr. Pawan Kumar** Maximum Marks: 20

Instructions:

This assignment needs to be done in C/C++. The goal of this assignment is to introduce you to basic code profiling and cache optimizations. Assignments submitted after the deadline won't be considered for grading. It is a good idea to type the code rather than copy pasting it. Write answers to your question in some text editor.

1. **(Profiling of a sequential code)** **[6]**

Before a program (which typically is not known in all its details) is parallelized, it is common practice to use parformance analysis tools in order to identify the most compute intensive parts of the program code. A simple tool for this purpose is the program gprof, which measures for each procedure (or method) P in the program how often it is called and how much time is required to process P. In doing so, the tool differentiates between

1. the inclusive time of P, which also includes the processing time of the procedures called by P (children), and
2. the exclusive time (or self time) of P, which does not include the processing time of these calls.

When, e.g., the procedure

```
void funcA() {
for (int i=0; i<N; i++) { ... }
funcB();
}
```

is invoked, the exclusive time of funcA is only the runtime of the for loop, while the inclusive time also includes the runtime of funcB. In this exercise, you should familiarize yourself with `gprof`.

- First, compile example.cpp with instrumentation for `gprof`:

```
g++ -pg -g -o example example.cpp
```

- Start the program with `./example`. During the execution, the gathered profiling data is written to a file `gmon.out`.
- A readable form of this data is then obtained using the command

```
gprof ./example
```

Thoroughly look at the output, in particular, the included explanations, and answer the following questions:

1. Which function requires the most computing time (exclusive) and would therefore be the first candidate for parallelization/optimization?
   (a) What speedup and what efficiency would you expect in the best case, if only this function is parallelized?
   (b) How often is the function called?
   (c) What other functions does the function call?
2. Which function is most frequently called and from which other functions?
3. How is the computation time of the function `func5` composed?
4. Attach the screenshot of the command line `gprof` profile output for verification.

2. (**Cache Effect**) [2]

Consider the following piece of C-code

```
#include <stdio.h>
#include <stdlib.h>
#define n 10*1024*1024

struct DATA
{
  int a;
  int b;
  int c;
  int d;
};

struct DATA pMyData[n];

int main()
{

  for (long i=0; i<10*1024*1024; i++)
    {
      pMyData[i].a = pMyData[i].b;
    }

  return 0;

}
```

Save this file as co1a.c. Next consider the following code

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

```
#include <stdlib.h>
#define n 10*1024*1024

struct DATA
{
  int a;
  int b;
};

struct DATA pMyData[n];

int main()
{

  for (long i=0; i<10*1024*1024; i++)
    {
      pMyData[i].a = pMyData[i].b;
    }

  return 0;

}
```

Save this file as co1b.c. Compile these codes using compiler flags `-c99 -O0`, and compare the times using `time` command. The `real` time produced by the `time` command is the wallclock time. Run both these codes 5 times, and take the average. Which code is faster and why? You may convince your answer by showing a sample memory-cache mapping diagram.

3. **(Cache Effect)** [2]

Consider the following C-code

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

#define nRows 1024*8
#define nCols 1024*8

char MyData[nRows*nCols];

int main()
{

  for (long x=0; x < nRows; x++)
    for (long y=0; y < nCols; y++)
    {
      MyData[x+y*nRows]++;
    }
  return 0;

}
```

Save this code as co2a.c. Now consider the following code

```
#include <stdio.h>
#include <stdlib.h>
#define nRows 1024*8
#define nCols 1024*8

char MyData[nRows*nCols];

int main()
{
 for (long y=0; y < nCols; y++)
  for (long x=0; x < nRows; x++)
    {
      MyData[x+y*nRows]++;
    }

  return 0;

}
```

Save this program as co2b.c. Compile these codes using compiler flags `-c99 -O0`, and compare the times using `time` command. Which of these codes run faster and why?

4. **(Cache Optimization: Cache Blocking)** [10]

Consider the following piece of C-code.

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

#define n 1000

int main()
{
    int i, j, k;
    double *A[n], *B[n], *C[n];

    for (i = 0; i < n; i++){
         A[i] = (double *) malloc(n * sizeof(double));
         B[i] = (double *) malloc(n * sizeof(double));
         C[i] = (double *) malloc(n * sizeof(double));
     }
    for (i = 0; i <  n; i++)
      for (j = 0; j < n; j++){
         A[i][j] = 1.0;
         B[i][j] = 1.0;
         C[i][j] = 1.0;
       }

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

    return 0;
}
```

We have following observations.

1. This program does the matrix-matrix multiplication: $C = A * B$.
2. In the inner k-loop, the access pattern for matrices C and A are row-wise, but for array B, it is column-wise, which will cause Cache misses.

In view of improving the performance, we recall that the matrix vector multiplication may be re-written as follows:

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} B^T_{jk} = A_{i1} B^T_{j1} + \cdots + A_{i(N-1)} B^T_{j(N-1)}.$$

This formulation of the matrix-matrix multiplication allows row-wise access pattern for the array B, but we need a transpose of matrix B. Modify the above code in the following way:

1. Do transpose of matrix B to obtain matrix BT.
2. Replace the following line `C[i][j] += A[i][k] * B[k][j]` in the `k`-loop by:
   `C[i][j] += A[i][k] * BT[j][k]`

Do the following tasks:

1. Compile both the codes using flags `-O0`. Compare the times of the new code with the old code using `time` shell command. Take average time of 5 runs for both.
2. When doing matrix transpose, there are many cache misses as well (due to column-wise access pattern for one of the matrix), why do you think the second code still runs faster?
3. Report caches misses using the `likwid` profiler by attaching a screenshot of the profiling results.
   (a) The link to likwid profiler is:
   `https://github.com/RRZE-HPC/likwid`
   (b) The documentation is here:
   `http://www.vi-hps.org/upload/material/tw09/vi-hps-tw09-VI-HPS\_likwid.pdf`

Question 4 continues on the next page. . .

(c) Another doc file is here:

`https : //arxiv.org/pdf/1004.4431.pdf`

Let us now see if this code could be improved further. What we have done so far is to avoid cache misses as much as possible. But we can infact organize our data in chunks which are of cache line sizes, this is called **Cache blocking**. In the first matrix-matrix multiplication code, replace the main matrix-matrix computation loop by the following code

```
int jj, kk;
double sum = 0.0;
for (jj = 0; jj < n; jj += SM)
  for (i = 0; i < n; i++)
    for(j = jj; j < min(jj + SM, n); j++)
      for (kk = 0; kk < n; kk += SM){
        for (i=0; i < n; i++){
          for (j = jj; j < min(jj + SM, n); j++){
            sum = 0.0;
            for (k = kk; k < min(kk + SM,n); k++){
              sum += A[i][k] * B[k][j];
            }
            C[i][j] += sum;
          }
        }
      }
```

Do the following:

1. Compile the code by adding the flag as follows:

   `gcc -DCLS = $(getconf LEVEL1_DCACHE_LINESIZE) − O0 − o mycode mycode.c`

   This will fetch the Cache line size in variable `CLS`, which will be used in the macro below.
2. Add the following macro:

   `#define SM (CLS/sizeof(double))`

   at the top after the include files.
3. Moreover to use min function, you need to add the following macro:
   `#define min(x, y) (((x) < (y)) ? (x) : (y))`

Now, answer the following:

1. What is the time taken by the new code compared to other two matrix-matrix codes above? Compare all three versions for $n = 1000, 2000$, and 4000.
2. Does the new code run faster? If yes, why? *Hint: provide arguments on how the cache lines are used.*
3. Attach the screenshot of the cache hits with `likwid` profiler for all the three codes.