CS69001 Computing Laboratory – I

Assignment No: C3

Date: 29–October–2018

Let A, B be $N \times N$ matrices. The $N \times N$ product matrix $C = AB$ is to be computed. For a reason to be clear soon, N is assumed to be even. We also assume that array indexing is zero-based. Let us use lower-case letters and subscripts ij for denoting the (i, j)-th elements of the matrices. For example, a_{ij} is the (i, j)-th element of A for $i, j \in \{0, 1, 2, \ldots, N-1\}$. In the **`main()`** function, the initial process creates spaces for A, B, C in the shared memory. You may fix the dimensions N and M beforehand (no need to take user input).

```
#define N 1000
#define M N/2
```

Part 1: Sequential Matrix Multiplication

Use the formula

$$c_{ij} = \sum_{k=0}^{N-1} a_{ik} b_{kj}$$

for computing the product. Write a function **`matmul1(C,A,B)`** to implement this standard algorithm.

Part 2: Parallel Matrix Multiplication by Method 1

Let $M = N/2$. Break the matrices into $M \times M$ blocks.

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}, \quad B = \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}, \quad C = \begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix}.$$

For each $I, J, K \in \{0, 1\}$, define the block product

$$D_{IJK} = A_{IK} B_{KJ}.$$

For each $I, J \in \{0, 1\}$, we then have

$$C_{IJ} = D_{IJ0} + D_{IJ1}.$$

Write a function **`matmul2(C,A,B)`** to compute $C = AB$ using this block decomposition of the matrices.

Let P denote the initial process. This process creates eight child processes $P_0, P_1, P_2, \ldots, P_7$. Each child tag $t \in \{0, 1, 2, \ldots, 7\}$ can be identified as a three-bit integer IJK. The first task of the child process P_t is to compute the block product D_{IJK} in a local (non-shared) $M \times M$ array.

Each child process then tries to modify the relevant block in the product matrix C. For $I, J \in \{0, 1\}$, let $s = 2I + J$. Only the child processes P_{2s} and P_{2s+1} attempt to modify C_{IJ}. These processes have computed the local block products D_{IJ0} and D_{IJ1}, respectively. The process which first accesses the block C_{IJ} *copies* its local block product to C_{IJ}. The second process to access C_{IJ} *adds* its local block product to C_{IJ}. After this modification (copy/add), each child process terminates.

Evidently, P_{2s} and P_{2s+1} needs mutual exclusion to modify C_{IJ}. Moreover, each of these processes must know whether it has to copy or add its block product. Use a shared variable to store the status of the block C_{IJ} (not written by any process, copied by only one process, contributed by two processes). Since this is a write-enabled shared variable, its access too needs to be performed in a mutually exclusive fashion. Use appropriate semaphores to enforce mutual exclusion and synchronization.

The initial process P waits until all the eight child processes terminate. This synchronization can be achieved by the **`wait()`** system call.

Part 3: Parallel Matrix Multiplication by Method 2

Use the same block decomposition of Part 2. The initial process P again forks eight new child processes $Q_0, Q_1, Q_2, \ldots, Q_7$. Let $t = 4I + 2J + K$, where $I, J, K \in \{0, 1\}$. The task of the child process Q_t is to compute the block product D_{IJK} in the *shared* memory. Allocate different shared memory segments to the eight block products, so these products may be computed in parallel without a need of mutual exclusion.

Let again $s = 2I + J$. The processes P_{2s} and P_{2s+1} compute the shared block products D_{IJ0} and D_{IJ1}, respectively. The process (among these two) to finish first terminates immediately. The process to finish second, computes the block $C_{IJ} = D_{IJ0} + D_{IJ1}$, and then terminates. Since the block products D_{IJK} now reside in shared memory, any child process can access the block products computed by other child processes.

The synchronization between the two processes P_{2s} and P_{2s+1} is to be implemented using a semaphore. As in Part 2, the initial process P **`wait()`**'s until all the eight child processes $Q_0, Q_1, Q_2, \ldots, Q_7$ terminate. Write a function **`matmul3(C,A,B)`** to carry out the product in the manner described in this part.

The *main*() function

- Create shared-memory segments A, B, C_1, C_2, C_3, each capable of storing an $N \times N$ integer matrix.
- Populate A and B with random integers in the range $[-9, 9]$.
- Call **`matmul1(C1,A,B)`** to compute the product $C_1 = AB$. Report the time taken by this function.
- Call **`matmul2(C2,A,B)`** to compute $C_2 = AB$ by the method of Part 2. Report the time taken by this function. Also, verify that C_1 and C_2 are the same matrix.
- Call **`matmul2(C3,A,B)`** to compute $C_3 = AB$ by the method of Part 3. Report the time taken by this function. Also, verify that C_1 and C_3 are the same matrix.
- The above calls are to be made by the initial process P. Appropriate child processes come to being by the calls **`matmul2()`** and **`matmul3()`**. The child processes would also print the times taken by them (from creation to termination).
- P should delete all shared-memory segments and semaphores when these are no longer used.

Sample output

```
+++ Sequential matrix multiplication
    Time taken = 2.172489619 sec

+++ Parallel matrix multiplication by Method 1
    Time taken by Process 0 = 0.668225744 sec
    Time taken by Process 1 = 0.941227939 sec
    Time taken by Process 6 = 1.127982911 sec
    Time taken by Process 7 = 1.161493170 sec
    Time taken by Process 4 = 1.173572038 sec
    Time taken by Process 5 = 1.153494400 sec
    Time taken by Process 2 = 1.184520792 sec
    Time taken by Process 3 = 1.218318732 sec
    Time taken = 1.219022799 sec

+++ Parallel matrix multiplication by Method 2
    Time taken by Process 0 = 0.962716714 sec
    Time taken by Process 1 = 1.015974284 sec
    Time taken by Process 5 = 1.031559906 sec
    Time taken by Process 7 = 1.106795162 sec
    Time taken by Process 4 = 1.137290960 sec
    Time taken by Process 6 = 1.147446384 sec
    Time taken by Process 3 = 1.142069608 sec
    Time taken by Process 2 = 1.151665685 sec
    Time taken = 1.178885296 sec
```

Submit a single C/C++ source file.

How to measure time

There are many ways actually. And all depends on what time you are interested in. And in what resolution. Whatever you want, do **#include <time.h>** (the usual header file declaring time-related functions).

In a time-shared system, it is natural to ask, for how much time your processes individually run. Here follows a way that gives you that to a reasonable accuracy. If you are running a single-process program with no other compute-intensive process(es) running, this time tallies closely with the delay you experience.

```
clock_t c1, c2;
double CPU_time_taken;

c1 = clock();
/* Insert the code, of which you want the measure the time */
c2 = clock();

CPU_time_taken = (double)(c2 - c1) / (double)CLOCKS_PER_SEC;

printf("CPU time taken = %lf seconds\n", CPU_time_taken);
```

For this assignment, however, there is a problem with the CPU-usage time. The initial process *P* does only managerial activity, and eats up very little CPU time. Irrespective of the delay you see in the running of your program, the **CPU_time_taken** by *P* is reported as quite low. In this situation, you would like to register the calendar time spent. In Unix, the calendar time is measured by the number of seconds that elapsed from Jan 01, 1970, 00:00:00 hr UTC (surprisingly, this is a time that never existed!!!). You may use the system call **time(NULL)** to obtain these many seconds. Now, here is a technology that lets you capture the calendar time with an approximate resolution of nano-seconds.

```
struct timespec ts1, ts2;
double cal_time_taken;

clock_gettime(CLOCK_REALTIME, &ts1);
/* Your timely code here */
clock_gettime(CLOCK_REALTIME, &ts2);

cal_time_taken = 1000000000. * (double)(ts2.tv_sec - ts1.tv_sec);
cal_time_taken += (double)(ts2.tv_nsec - ts1.tv_nsec);
cal_time_taken /= 1000000000.;

printf("calendar time taken = %.9lf seconds\n", cal_time_taken);
```