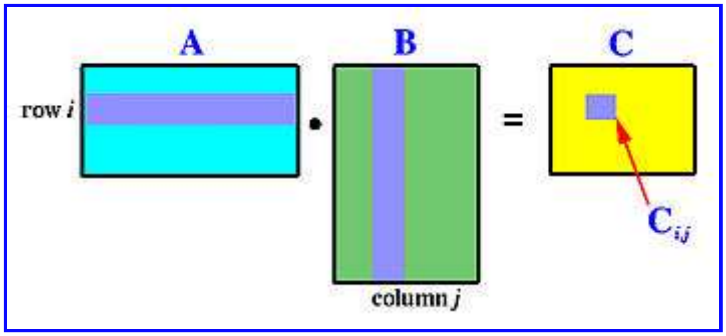
Programming Assignment VI

Due on Friday, December 14, 2018 @ 11pm

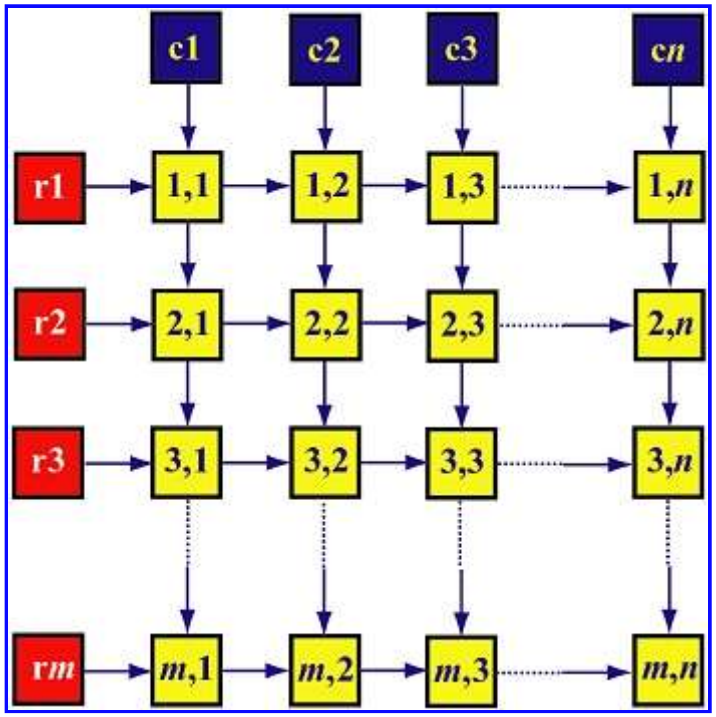
70 points

Matrix Multiplication, Again!

We all know that the product of matrix $\mathbf{A}=[a_{i,j}]_{m\times h}$ and matrix $\mathbf{B}=[b_{i,j}]_{h\times n}$ is a matrix $\mathbf{C}=[c_{i,j}]_{m\times n}$, where $c_{i,j} = a_{i,1}b_{1,j} + a_{i,2}b_{2,j} + a_{i,3}b_{3,j} + ... + a_{i,h}b_{h,j}$. In other words, $c_{i,j}$ is the inner (or dot) product of the i-th row of **A** and the j-th column of **B**. This concept is shown in the diagram below:



With synchronous channels, we have yet another interesting way to compute the product of two matrices. We may use $m\times n$ processors (simulated by $m\times n$ threads) organized in a grid of m rows and n columns. The processor $P_{i,j}$ on row i and column j has two synchronous channels connected to the processor to its right $P_{i,j+1}$ and to the processor below $P_{i+1,j}$. However, the processors on the last row have no channels going down, and the processors on the last column have no channel to the right. This is shown below.



In addition to this processor grid, we also have two arrays of processors (also simulated by threads). Processors r_i ($1 <= i <= m$) are connected to the processors on the first column, and processors c_j ($1 <= j <= n$) are connected to the processors on the first row. The job for processor r_i is sending the elements on row i of matrix **A** to its neighbor in the order of $a_{i,1}$, $a_{i,2}$, $a_{i,3}$, ..., $a_{i,h}$. The job for processor c_j is sending the elements on column j of matrix **B** in the order of $b_{1,j}$, $b_{2,j}$, $b_{3,j}$, ..., $b_{h,j}$.

The job assigned to each processor $P_{i,j}$ is very simple:

- Initially, $P_{i,j}$ clears a local variable to zero.
- For each iteration, it carries out the following steps:
  1. Receives a number from its top neighbor and a number from its left neighbor
  2. Sends the number from the top (*resp.*, left) neighbor down (*resp.*, right).
  3. Multiplies these two numbers.
  4. Adds the result to its local variable.
- Repeat the above until there is no further data from the two incoming channels. Once this happens, the value stored in the local variable of processor $P_{i,j}$ is $c_{i,j}$.

To make sure every processor knows the end of data has reached, after sending out the last data of a row or a column, processor r_i or c_j sends out a special mark **EOD** (*i.e.*, end-of-data). Therefore, when processor $P_{i,j}$ receives **EOD**'s from its top and left neighbors, it knows that the computation completes and the result is in its local variable. Each processor must also pass the **EOD** to its neighbors!

Your job is to write a program using **ThreadMentor** to simulate this matrix multiplication algorithm.

Program Specifications

Write a program using C++ and **ThreadMentor** to do the following:

- The main program reads in matrices **A** and **B**, and constructs all channels and creates all threads. All values are non-negative integers. **All channels must be synchronous.**
- Thread r_i only "sees" the i-th row of matrix **A**, thread c_j only "sees" the j-th column of matrix **B**, and thread $P_{i,j}$ does not "see" anything in matrices **A** and **B**.
- Start all threads and allow thread r_i (*resp.*, c_j) to send row i of matrix **A** (*resp.*, column j of matrix **B**) into the corresponding channel.
- After all values have been sent, a **EOD** is sent to the channel and the sender terminates. Since all values of the involved matrices are non-negative, you may use a negative value (*e.g.*, -1) for **EOD**.
- After thread $P_{i,j}$ receives the **EOD**'s from its top and left neighbors, it saves the computed result into row i and column j of matrix **C**, which is a global array. **Note that this is the only occasion in which thread $P_{i,j}$ can access the global array C.**
- After all entries of matrix **C** are computed, the main program prints the result and terminates.

Except for locking `stdout`, you can only use channels. Otherwise, you receive no point.

Input and Output

The input to your program is in a file with the following format:

```
l   m   <---------------------- # of rows and columns of matrix A
a11 a12 a13 ... a1m <--------- row 1 of A
a21 a22 a23 ... a2m <--------- row 2 of A
```

```
 ................
al1 al2 al3 ... alm <--------- row l of A
u   v   <-------------------- # of rows and columns of matrix B
b11 b12 b13 ... b1v <--------- row 1 of B
b21 b22 b23 ... b2v <--------- row 2 of B
 ................
bu1 bu2 bu3 ... buv <--------- row u of B
```

Note that the matrices can be multiplied only if $m = u$ holds. You may assume that l, m, u and v are positive integers less than or equal to 8. But, there is no guarantee that m is equal to u. Your program should report this problem and stop if m is not equal to u. The input values for the matrices are always non-negative integers.

The following is in file `data.in1`. Click **here** to download a copy.

```
3  4
1  2  3  4
5  6  7  8
9 10 11 12
4  2
8  7
6  5
4  3
2  1
```

The result is shown below:

$$\begin{bmatrix} 40 & 30 \\ 120 & 94 \\ 200 & 158 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \cdot \begin{bmatrix} 8 & 7 \\ 6 & 5 \\ 4 & 3 \\ 2 & 1 \end{bmatrix}$$

Your program's output should look like the following, although it is impossible for you to generate an identical output due to the dynamic behavior of threaded programs. The output is obvious and does not require further elaboration. Note that r_i threads, c_j threads and $P_{i,j}$ threads start from columns 1, 4 and 7, respectively. **Your program should print an error message if the input matrices cannot be multiplied.**

```
                .........
Row thread r[3] started
                .........
   Column thread c[5] started
                .........
      Thread P[3,4] started
                .........
Row thread r[3] sent 5 to P[3,1]
                .........
      Thread P[2,4] received 3 from above and 2 from left
                .........
      Thread P[2,4] sent 3 to below and 2 to right
                .........
   Column thread c[2] sent 4 to P[1,4]
                .........
Row thread r[4] sent EOD to P[4,1] and terminated
                .........
      Thread P[3,2] received EOD, saved result 54 and terminated
                .........

*** From main ***
Matrix A: xx rows and yy columns
xx xx xx xx ..... xx
```

```
xx xx xx xx ..... xx
    ............
xx xx xx xx ..... xx

Matrix B: xx rows and yy columns
xx xx xx xx ..... xx
xx xx xx xx ..... xx
    ............
xx xx xx xx ..... xx

Matrix C = A*B: xx rows and yy columns
xx xx xx xx ..... xx
xx xx xx xx ..... xx
    ............
xx xx xx xx ..... xx
```

The following is in file `data.in2`. Click **here** to download a copy.

```
5  4
 1  3  5  7
 9 11 13 15
17 19 21 23
25 27 29 31
33 35 37 39
4  6
 1  2  3  4  5  6
 7  8  9 10 11 12
13 14 15 16 17 18
19 20 21 22 23 24
```

The result is shown below:

$$\begin{bmatrix} 220 & 236 & 252 & 268 & 284 & 300 \\ 540 & 588 & 636 & 684 & 732 & 780 \\ 860 & 940 & 1020 & 1100 & 1180 & 1260 \\ 1180 & 1292 & 1404 & 1516 & 1628 & 1740 \\ 1500 & 1644 & 1788 & 1932 & 2076 & 2220 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 5 & 7 \\ 9 & 11 & 13 & 15 \\ 17 & 19 & 21 & 23 \\ 25 & 27 & 29 & 31 \\ 33 & 35 & 37 & 39 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 & 17 & 18 \\ 19 & 20 & 21 & 22 & 23 & 24 \end{bmatrix}$$

Submission Guidelines

General Rules

1. All programs must be written in C++.
2. Use the `submit` command to submit your work. You may submit as many times as you want, but only the last on-time one will be graded.
3. Unix filename is case sensitive, `THREAD.cpp`, `Thread.CPP`, `thread.CPP`, etc are ***not*** the same.
4. We will use the following approach to compile and test your programs:

   ```
   make noVisual    <-- make your program
   ./prog6 < file   <-- test your program with an input file
   ```

   This procedure may be repeated a number of times with different input files to see if your program works correctly.
5. Your implementation should fulfill the program specification as stated. Any deviation from the specification will cause you to receive **zero** point.
6. A `README` file is always required.

7. **No late submission will be graded.**
8. **Programs submitted to wrong class and/or wrong section will not be graded.**

Compiling and Running Your Programs

This is about the way of compiling and running your program. If we cannot compile your program due to syntax errors, wrong file names, etc, **we cannot test your program, and, as a result, you receive 0 point**. If your program compiles successfully but fails to run, **we cannot test your program, and, again, you receive 0 point**. **Therefore, before submitting your work, make sure your program can compile and run properly.**

1. **Not-compile programs receive 0 point.** By **not-compile**, I mean any reason that could cause an unsuccessful compilation, including missing files, incorrect filenames, syntax errors in your programs, and so on. Double check your files before you submit, since I will **not** change your program. Note again: Unix filenames are ***case sensitive***.
2. **Compile-but-not-run programs receive 0 point. Compile-but-not-run** usually means you have attempted to solve the problem to some degree but you failed to make it working properly.
3. **A meaningless or vague program receives 0 point even though it compiles successfully.** This usually means your program does not solve the problem but serves as a placeholder or template just making it to compile and run.

Program Style and Documentation

This section is about program style and documentation.

1. For each file, the first piece should be a program header to identify yourself like this:

```
// ------------------------------------------------------
// NAME : John Smith                          User ID: xxxxxxxx
// DUE DATE : mm/dd/yyyy
// PROGRAM ASSIGNMENT #
// FILE NAME : xxxx.yyyy.zzzz (your unix file name)
// PROGRAM PURPOSE :
//    A couple of lines describing your program briefly
// ------------------------------------------------------
```

   Here, **User ID** is the one you use to login. It is ***not*** your social security number nor your M number.

   For each function in your program, include a simple description like this:

```
// ------------------------------------------------------
// FUNCTION  xxyyzz : (function name)
//     the purpose of this function
// PARAMETER USAGE :
//    a list of all parameters and their meaning
// FUNCTION CALLED :
//    a list of functions that are called by this one
// ------------------------------------------------------
```

2. Your programs must contain enough concise and to-the-point comments. Do not write a novel!
3. Your program should have good indentation.
4. Do not use global variables!

Program Specifications

Your program must follow exactly the requirements of this programming assignment. Otherwise, you receive 0 point even though your program runs and produces correct output. The following is a list of potential problems.

1. Your program does not use the indicated algorithms/methods to solve this problem.
2. Your program does not follow the structure given in the specification. For example, your program is not divided into functions and files, etc when the specification says you should.
3. Any other significant violation of the given program specification.
4. **Incorrect output format.** This will cost you some points depending on how serious the violations are. The grader will make a decision. Hence, carefully check your program output against the required one.
5. **Your program does not achieve the goal of maximum parallelism.**

Program Correctness

If your program compiles and runs, we will check its correctness. We normally run your program with two sets of input data, one posted on this programming assignment page (the public one) and the other prepared by the grader (the private one). You program must deliver correct results for both data sets. Depending on the seriousness of the problem(s), significant deduction may be applied. For example, if your program delivers all wrong results for the public data set, you receive 0 point for that component.

The `README` File

A file named `README` is required to answer the following questions:

- The logic of your program
- Why does your program work?
- The meaning, initial value and the use of each variable. Explain why their initial values and uses are correct. Justify your claim.
- Answer the following:
    1. Why does thread $P_{i,j}$ compute the inner (or dot) product of row i of matrix **A** and column j of matrix **B** correctly?
    2. Why are *synchronous* channels used rather than *asynchronous* channels? A rigorous reasoning is required.
    3. Thread $P_{i,j}$ terminates when it receives a **EOD** from its top and left neighbors. Does this thread termination follow a particular pattern? Draw a diagram of this termination pattern. **Note that your diagram must be readable when it is printed.**
    4. Can a thread terminate once it receives a **EOD** from its left (or top) neighbor and still deliver correct results. Explain your answer as clear as possible.

You should elaborate your answer and provide details. **When answering the above questions, make sure each answer starts with a new line and have the question number (*e.g.*, Question X:) clearly shown. Separate two answers with a blank line.**

Note that the file name has to be `README` rather than `readme` or `Readme`. Note also that there is ***no*** filename extension, which means filename such as `README.TXT` is ***NOT*** acceptable.

`README` must be a plain text file. We do not accept files produced by any word processor. Moreover, watch for very long lines. More precisely, limit the length of each line to no more than 80 characters with the Return/Enter key for line separation. Missing this file, submitting non-text file, file with long lines, or providing incorrect and/or vague answers will cost you many points. Suggestion: Use a Unix text editor to prepare your `README` rather than a word processor.

Final Notes

1. Your submission should include the following files:
    1. File `thread.h` that contains all class definitions of your threads.
    2. File `thread.cpp` contains all class implementations of your threads.
    3. File `thread-main.cpp` contains the main program.
    4. File `Makefile` is a makefile that compiles the above three files to an executable file `prog6`. **Note that without following this file structure your program is likely to fall into the compile-but-not-run category, and, as a result, you may receive a low grade. So, before submission, check if you have the proper file structure and correct makefile.** Note that your `Makefile` **should not** activate the visualization system.
    5. File `README`.
2. **Always start early, because I will not grant any extension if your home machine, network connection, your phone line or the department machines crash in the last minute.**
3. **Since the rules are all clearly stated, no leniency will be given and none of the above conditions is negotiable. So, if you have anything in doubt, please ask for clarification.**
4. **Click here to see how your program will be graded.**