Homework 3 (due November 19, 2019)

Warm up question – do not submit the result: Compile and execute the "compute pi" program on "linux.cs.pitt.pitt.edu" (see file pi.c and the notes below). Measure the execution times for *P* =1, 2, 4, 8, 16 and 32 threads with *N*= 10000, 100000, 1000000 and 10000000 points and observe the difference in speedups.

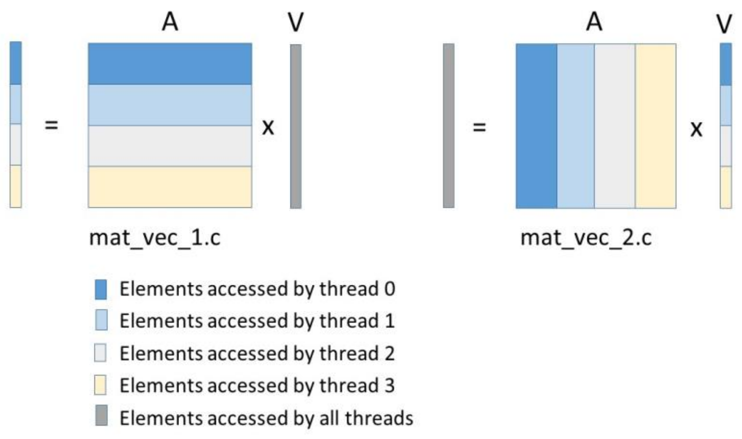
Notes:

1) linux.cs.pitt.edu is a dual processor with 10 cores per processor.
2) Once on linux.cs.pitt.edu you can copy pi.c (or mat_vec.c) using
```
cp  /afs/cs.pitt.edu/public/courses/1541/pi.c   .
```
3) To compile a c program (for example pi.c) use
```
gcc  -D_REENTRANT -lpthread -lm -o pi pi.c
```
4) For each specific values of P and N, you should repeat the execution multiple times and take the average (excluding outliers). The execution time may vary from one run to the next due to interference from OS and/or other executing processes.

Questions 1: Write two parallel versions of the matrix/vector multiplication program, mat_vec.c, that multiplies an *NxN* matrix, A, by an *N*-dimensional vector, V. Your programs should accept a second command line argument, *P*, which indicates the number of execution threads.

- In your first version, mat_vec_1.c, each thread computes *N/P* rows of the result vector
- In your second version, mat_vec_2.c, all threads collaboratively compute the elements of the result vector, with each thread computing the product of *N/P* columns of A with *N/P* elements of V and accumulating the result in the product vector. Mutual exclusion should be used when accumulating the results.

If *N* is not a multiple of *P*, then you may assign fewer than *N/P* rows (or columns) to the last thread. The following figure illustrates the partitioning the computation among 4 threads in each of mat_vec_1.c and mat_vec_2.c.


A
V
=
x
mat_vec_1.c
A
V
=
x
mat_vec_2.c
Elements accessed by thread 0
Elements accessed by thread 1
Elements accessed by thread 2
Elements accessed by thread 3
Elements accessed by all threads

Measure and report the execution times and speed ups of both versions on linux.cs.pitt.edu with *P* = 1, 2, 4, 8, 16 and 32 and *N* = 4000 and 8000, 12000 (run each experiment a few times and exclude any outlier caused by interference from OS or other processes). Comment on your results. In addition to including the results in your homework, you should email the source files to the TA before class time on the due date.

FASTEST IMPLEMENTATION COMPETITION: the efficiency of mat_vec.2.c depends on the granularity of the critical sections used to enforce mutual exclusion. The fastest 5 implementations will get 50% additional credits for this question.

Questions 2 (Amdahl Law): In this exercise, we are considering enhancing a machine by adding vector hardware to it. When a computation is run in vector mode on the vector hardware, it is 12 times faster than the normal mode of execution. We call the percentage of time that could be spent using vector mode, the *percentage of vectorization (can be denoted by PV).*

a) Draw a graph that plots the speedup as a function of the percentage of the computation performed in vector mode.
b) What percentage of vectorization is needed to achieve a speedup of 2?
c) What percentage of vectorization is needed to achieve one-half the maximum possible speedup attainable from using vector mode?
d) Suppose you have measured the typical percentage of vectorization to be 70%. The hardware design group estimates that it can speed up the vector hardware to 20 (instead of 12) with significant additional investment. You wonder if the compiler group could increase the percentage of vectorization, instead. What percentage of vectorization would the compiler group need to achieve in order to equal the improvement proposed by the hardware group?

Questions 3: Consider the execution of 11 instructions, A1, …, A11 of a thread A, and 8 instructions B1, …, B8 of a second thread, B, on a single 2-issue pipeline, where in each cycle at most one R-type instruction and one M-type (Memory type) instruction can be issued but instructions cannot be issued out of order (similar to the superscalar that you implemented in Project 1). Moreover, due to data dependencies and constrains on forwarding in the architecture, a scheduler (instruction dispatcher) has to observe the following:

A5 should be issued at least 2 cycle after A3 is issued
A6 should be issued at least 2 cycle after A4 is issued
A9 should be issued at least 4 cycle after A8 is issued
B4 should be issued at least 2 cycle after B3 is issued
B5 should be issued at least 2 cycle after B4 is issued
B6 should be issued at least 1 cycle after B5 is issued

Which leads to the following schedules for each thread when no multithreading is used:

cycle	R-type issued	M-type issued
1	A1	A2
2	A3	
3	A4	
4		A5
5	A6	A7
6	A8	
7		
8		
9		
10	A9	
11	A10	A11

cycle	R-type issued	M-type issued
1		B1
2	B3	B2
3		
4		B4
5		
6	B5	
7		B6
8	B8	B7
9		
10		
11		

Show using three similar tables the schedule if

a) Coarse grain multithreading is used (switch threads after an idle cycle or after the executing thread terminates)
b) Fine grained multithreading is used. Consider two variations:
    b1) instructions from A and B are issued strictly in alternate cycles
    b2) threads issue instructions in alternate cycles, but the order is changed if no instructions can be issued from one thread while some instructions can be issued from the other.
c) Simultaneous multithreading is used (with priority given to thread A -- that is, if two instructions Ai and Bk, that are of the same type, can be issued in the same cycle, Ai is scheduled)