ECE 565
Fall 2021

Assignment #3

Submit instructions: Submit a file named **hw3_writeup.pdf**.

1. **Programming Models.** For the following code, identify variables that are read-only, read/write non-conflicting, and read-write conflicting, if we consider parallelizing the "for i" loop only or if we consider parallelizing the "for j" loop only. In the code, the function 'ceil(arg)' returns the integer ceiling of the passed argument.

```
float product;
int i, j, N, sum;
int data_array[N];
int data_gridX[N][N], data_gridY[N][N];
float measurement[N];

for (i=1; i<N; i++) {
   product = 1;
   for (j=1; j<N; j++) {
        product = product * sum * measurement[j]
        measurement[ceil(product)]++;
        if ((i-j) >= 0)
             data_gridX[i-j][j] = data_gridX[i-j][j] * 2;
        if ((j-i) >= 0)
             data_gridY[i][j-i] = data_gridY[i][j-i] / 4;
   }
   sum = sum + data_array[i] + product;
}
```

1) Considering parallelizing i-loop only:
   read-only: ***N, data_array***
   read/write non-conflicting: ***data_gridX, data_gridY***
   read-write conflicting: ***product, sum, i,j, measurement***

2) Considering parallelizing j-loop only:
   read-only: ***data_array, N, sum***
   read/write non-conflicting: ***i, data_gridX, data_gridY***
   read-write conflicting: ***product, measurement, j***

2. **Code Analysis for Parallel Task Identification.** For the code shown below, analyze the loop-carried data dependencies in order to answer each question about the presence of independent, parallel tasks. To help answer the questions, you may want to list out the data dependencies and/or draw the LDG. If you show your work (on the front or back of this page), and how you arrived at each answer, you may receive some partial credit if the final answer is not correct.

```
...
for (i=1; i<N; i++) {
   for (j=N-3; j>0; j--) {
        S1: b[i][j] = b[i][j+2] + a[i][j];
        S2: a[i][j] = a[i][j] + a[i-1][j] + a[i+1][j];
   }
}
...
```

(a) Is each "for i" loop iteration an independent parallel task (i.e. DOALL parallelism)?

(b) Is each "for j" loop iteration an independent parallel task?

(c) Assume that we change the loop structure to traverse across the diagonals or anti-diagonals (i.e. as in chart 9 of the shared memory programming slides), and then traverse each node within the diagonal or anti-diagonal in the inner loop. Is the update of each node along a diagonal an independent parallel task? Is the update of each node along an anti-diagonal an independent parallel task?

(d) Other than any identified in parts (a) – (c) are there any other parallel tasks that you can identify? If so, describe this available parallelism.

2)

Sol:

Dependencies:

Loop-independent:

S1[i,j] -> A S2[i,j] (1)

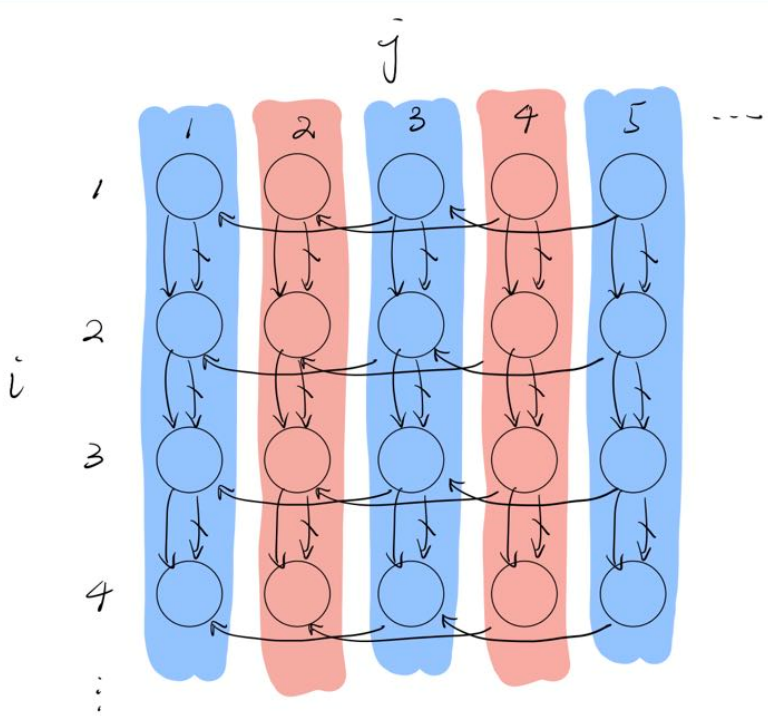
Loop-carried:

S1[i,j] -> T S1[i,j-2] (2)

S2[i,j] -> T S2[i+1,j] (3)

S2[i,j] -> A S2[i+1,j] (4)

LDG:



a)

No. It's not, since we have loop-carried dependencies (3) and (4).

b)

No. It's not, since we have loop-carried dependency (2).

c)

d)

Yes. As shown in the LDG, the odd j and even j can be grouped as two groups of tasks. And the two groups are independent to each other.

3. **Code Profiling & Performance Counters.** In this problem, you will work with a (most likely new to you) existing code base to identify the functions which dominate execution time, calculate theoretical possible speedups, and extract various metrics using performance counters.

To complete this problem, you may use your own personal machine of your choosing, or you may compile, build, run, and measure your code on one of two available server machines: **kraken.egr.duke.edu** or **leviathan.egr.duke.edu**. An individual account will be created for you, and your username and initial password will be emailed. The code that you will work with is a Finite Element mini-application, which assembles and solves a linear system of equations. You may download the code, and view additional documentation and information at these links:
https://asc.llnl.gov/sites/asc/files/2020-06/MiniFE_Summary_v2.0.pdf
https://asc.llnl.gov/sites/asc/files/2020-09/miniFE_openmp-2.0-rc3.zip

(a) **Performance profiling**: After downloading and unpackaging the code, prepare the makefile for compilation. We will use this makefile: makefile.gnu.serial. The "serial" indicates that we will only use a single-threaded run of the program. To 'CFLAGS' and 'CXXFLAGS' add the flags needed to enable use of 'gprof' (refer to the class notes and don't forget optimization!). Also add the following flag '-fno-inline' which prevents function inlining and will make this exercise more interesting. Compile the program with: 'make –f makefile.gnu.serial'.

Next, execute the program with the following input: ./miniFE.x -nx 40 -ny 80 -nz 160

Finally, collect & examine the flat profile using gprof. For each of the top 8 functions, report: 1) the function name, 2) the number of calls, and 3) the percentage of execution time for all calls to the function.

(b) **Amdahl's Law**: Based on the percentage of execution time identified for the top function from part (a), calculate the overall application speedup that would be possible if this function were optimized to produce a 5x speedup for that function alone.

(c) **Performance Counters**: Use the 'perf' tool to collect performance counter data over the execution of the miniFE application (remember to run with the program arguments as specified above). Recall from the lecture notes that you may use 'perf list' to show a list of events to collect.
Collect and report counts for the following events:

- Instructions
- CPU cycles (and also show IPC, instructions per cycle)
- Branch instructions
- Branches misses (mispredictions)
- Cache references
- L1 data cache load misses
- L1 instruction cache load misses
- LLC (last level cache) loads
- LLC (last level cache) load misses
- Data TLB load misses

4. **Performance Counters.** For this question, you will revisit your matrix multiplication code from Assignment #2 (question 4, part a). In Assignment #2, you measured the performance of your code across three different orderings of the loop nests: I-J-K, J-K-I, and I-K-J. The assumption was that performance differed due to different memory access patterns with different cache miss rates. In this problem, you will use 'perf' as in question 3 to study the performance across these 3 loop nest orderings in more detail.

First, re-run your program across different loop nest orderings on the machine where you are using 'perf', and record you new performance results. Then, run your program across different loop nest orderings and use 'perf' to see if the miss rates of various levels of cache explain your performance results. Show your results and analysis in your writeup.