ECE 565
Fall 2019

Assignment #3

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;
}
```

2. **Parallelization.** Consider the following code. In part (a) you will express this code as DOPIPE parallelism, if necessary using $post_{ID}(\ldots)$ and $wait_{ID}(\ldots)$ to provide any synchronization, where ID is just a label to identify different pairs of post and wait synchronization. You may simply write ‘S1’ and ‘S2’ in your code, so you do not need to write out these full lines of code.

```
...
for (i=1; i<N; i++) {
   for (j=2; j<N-4; j=j+2) { // note iterator
        S1: a[i][j] = a[i-1][j-2] + b[i][j+4];
        S2: b[i][j] = b[i][j] + 1;
   }
}
...
```

(a) Exploit DOPIPE parallelism to the fullest and show the code which would execute in parallel on two processors.

(b) Show a mathematical expression to express the execution time of the code loops for both a serial execution of the loops, and for the DOPIPE version of the loops executing on two processors. You may base your expression on variables $T_{S1}$ and $T_{S2}$ to represent the execution time of statements S1 and S2, and N which determines how many times each loop is executed.

3. **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) Is each diagonal or anti-diagonal of the traversal 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.

4. **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/CORAL-benchmarks/Summaries/MiniFE_Summary_v2.0.pdf
https://asc.llnl.gov/CORAL-benchmarks/Throughput/MiniFE_ref_2.0-rc3.tar.gz

(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