# CSC/ECE 506: Architecture of Parallel Computers
# Problem Set 1
# Due: Tuesday, February 6, 2024

**Problem 1.** (*20 points)* Consider an algorithm that runs for 240 seconds on a uniprocessor system with 60% of the work being parallelizable.

(a) With a multiprocessor system, how many processors would be needed to execute the algorithm in a total execution time of 1/2 the time of the uniprocessor system?

(b) What is the efficiency of the system in part (a)?

(c) Can any multiprocessor system execute the algorithm in 1/3 the time of the uniprocessor system? Explain.

(d) How many processors are needed to execute the algorithm in 108 seconds?

For parts (e), (f), and (g), assume that the algorithm is now improved so that 75% of the previously serial work is now parallelizable.

(e) What fraction of the total work is now parallelizable? What fraction is serial?

(f) How many processors will it take to execute the algorithm in 1/4 the time of the uniprocessor system after this improvement?

(g) What is the maximum speedup in this new system?

**Problem 2.** (*20 points*) (*20 points*) Consider the program below. Assume all statements (S1 and S2) take 1 unit of time to execute. Assume N = 500.

```
for (int i=0; i<N; i++){
      S1: x[i] = w[i] * y[i];
      S2: z[i+1] = z[i] / x[i] – k;
      S3: u[i] = w[i] * v[i];
}
```

Assume the synchronization overhead is negligible.

(a) (*5 points*) Parallelize the above code using DOPIPE. Write the parallelized code.

(b) (*4 points*) How many units of time would it take for the parallelized program to run, and what would its speedup be using

i. 500 processors.
ii. 100 processors.
iii. 2 processors.

(c) (*5 points*) Parallelize the code using DOACROSS. Write the parallelized code.

(d) (*4 points*) How many units of time would this program take to run and what is its speedup if the number of processors is

i. 500 processors
ii. 100 processors
iii. 2 processors

(e) (*2 points*) Which of the two techniques discussed between DOACROSS and DOPIPE are preferable in a system with a very limited number of processors?

**Problem 3.** *(20 points)*Consider this C program:

```
1:  for (i = 1; i <= 1024; i++) {
2:              sum[i] = 0;
3:              for (j = 1; j <= i; j++)
4:                      sum[i] = sum[i] + i;
5:  }
```

In this code, each of the statements 2 and 4 take two machine cycles. Ignore the overhead due to loop control (statements 1, 3, and 5), all other system overhead (IPC, stalling, etc.) and resource conflicts.

(a) What is the total execution time of the program on a uniprocessor system?

(b) Think about how a parallel version of this program could be executed on a shared memory multiprocessor consisting of 32 processors.

Using a simple partitioning scheme that divides *i*-loop iterations among 32 processors (i.e. processor 1 executes iterations *i* = 1 to 32, processor 2 executes iterations *i* = 33 to 64, and so on), determine the total execution time and the speedup compared to the answer in part (a). Note that this leads to an unbalanced computational workload among the processors.

(c) Modify the partitioning scheme to facilitate a balanced parallel execution of all the computational workload over 32 processors. What are the minimum execution time and speedup factor over the uniprocessor execution resulting from the balanced parallel execution on 32 processors?

**Problem 4.** *(15 points)* Here is an algorithm commonly used in image-processing applications.

Consider an image *f* with width=ImageWidth and height=ImageHeight. *f* is a 2D grid of pixels. *k* is a kernel of width=2*w*+1 and height=2*h*+1 where (2*w*+1) < ImageWidth and (2*h*+1) < ImageHeight. The image *f* is processed using the kernel *k* to produce a new image *g* as shown:

```
for y = 0 to ImageHeight do
        for x = 0 to ImageWidth do
                sum = 0
                for i= -h to h do
                        for j = –w to w do
                                sum = sum + k[j,i] * f[x – j, y – i]
                        end for
                end for
                g[x y] = sum
        end for
end for
```

(a). Identify the read-only, R/W non-conflicting and R/W conflicting variables, if the **for** *y* loop is parallelized.

| Read only | R/W non-conflicting | R/W conflicting |
|---|---|---|

(b). Identify the read-only, R/W non-conflicting and R/W conflicting variables, if (only) the **for** *i* loop is parallelized. Assume that the **for** *i* tasks for the previous value of *x* must complete before the **for** *i* tasks of the current value of *x* are started.

| Read only | R/W non-conflicting | R/W conflicting |
|---|---|---|

(c). Identify the read-only, R/W non-conflicting and R/W conflicting variables, if the **for** *i* loop is parallelized. Assume that the **for** *i* tasks for the previous value of *x* do not have to complete before the **for** *i* tasks of the current value of *x* are started.

| Read only | R/W non-conflicting | R/W conflicting |
|---|---|---|