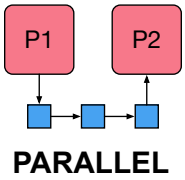

CISC 372: Parallel Computing

University of Delaware, Spring 2022

HW 6

This assignment is due at ***noon*** *on* ***Tuesday, Nov. 15****. Topics: OpenMP.*

Instructions

Copy directory `372-2022F/hw/hw6` to your personal repository `372-USER/hw/hw6` and add it to version control. For example, from within `372-USER/hw`, type

```
cp -r ../../372-2022F/hw/hw6 .
svn add hw6
```

Then change into `hw6`: `cd hw6`.

1. Problem 1: Correct or not?

Here are 10 little OpenMP programs. Examine each one closely. Assume the requested number of threads is always granted. Decide whether the program is correct or incorrect. If correct, determine all possible outputs. If incorrect, explain. You may also try compiling and executing the programs, if you think it will help.

Put you answers in a `README.txt` file using the following format:

```
1. CORRECT.  Outputs: 123, 234, 456, 789
2. INCORRECT.  Reason: data race on z at "z=x+y"
...
```

1.1. Program.

```
#include <stdio.h>
int main() {
#pragma omp parallel num_threads(3)
  {
    printf("hi");
  }
}
```

1.2. **Program.**

```
#include <stdio.h>
#include <omp.h>
int main() {
  int x;
#pragma omp parallel private(x) default(none) num_threads(3)
  {
    x = omp_get_thread_num();
    printf("%d", x);
  }
}
```

1.3. **Program.**

```
#include <stdio.h>
int main() {
  int x = 1;
#pragma omp parallel shared(x) default(none) num_threads(3)
  {
    x=0;
    printf("%d", x);
  }
  printf("%d\n", x);
}
```

1.4. **Program.**

```
#include <stdio.h>
#include <omp.h>
int main() {
  int x = 1;
#pragma omp parallel firstprivate(x) default(none) num_threads(2)
  {
    x += omp_get_thread_num();
    printf("%d", x);
  }
  printf("%d\n", x);
}
```

1.5. **Program.**

```
#include <stdio.h>
#include <omp.h>
int main() {
  int x = 1;
#pragma omp parallel shared(x) default(none) num_threads(3)
  {
    int n = omp_get_num_threads();
    for (int i=0; i<n; i++) {
      if (i == omp_get_thread_num())
        x += i;
#pragma omp barrier
    }
  }
  printf("%d\n", x);
}
```

1.6. **Program.**

```
#include <stdio.h>
#define n 5
int a[n];
int main() {
#pragma omp parallel shared(a) default(none) num_threads(3)
  for (int i=0; i<n; i++)
    a[i] = i;
  for (int i=0; i<n; i++)
    printf("%d", a[i]);
  printf("\n");
}
```

1.7. **Program.**

```
#include <stdio.h>
#include <omp.h>
int x = 0;
int main() {
#pragma omp parallel shared(x) default(none) num_threads(4)
  {
    int tid = omp_get_thread_num();
#pragma omp critical (AAA)
    {
      x += tid;
    }
  }
  printf("%d\n", x);
}
```

1.8. **Program.**

```
#include <stdio.h>
#include <omp.h>
#define n 5
int a[n], b[n];
int main() {
  for (int i=0; i<n; i++)
    a[i] = i;
#pragma omp parallel for shared(a,b) default(none) num_threads(3)
  for (int i=0; i<n-1; i++)
    b[i] = a[i+1] - a[i];
  for (int i=0; i<n-1; i++)
    printf("%d", b[i]);
  printf("\n");
}
```

1.9. **Program.**

```
#include <stdio.h>
#define n 3
int main() {
  int a[n][n], b[n][n], c[n][n];
#pragma omp parallel for collapse(2)
  for (int i=0; i<n; i++)
    for (int j=0; j<n; j++) {
      a[i][j] = b[i][j] = 1;
      c[i][j] = 0;
    }
#pragma omp parallel for collapse(3)
  for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
      for (int k=0; k<n; k++)
        c[i][j] += a[i][k]*b[k][j];
  for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++)
      printf("%d", c[i][j]);
    printf("\n");
  }
}
```

1.10. **Program.**

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#define n 5
int *p, *q;
int main() {
#pragma omp parallel sections
  {
#pragma omp section
    {
      p = malloc(n*sizeof(int));
      assert(p);
      for (int i=0; i<n; i++) p[i] = i;
    }
#pragma omp section
    {
      q = malloc(n*sizeof(int));
      assert(q);
      for (int i=0; i<n; i++) q[i] = i;
    }
  }
#pragma omp parallel for shared(p,q) default(none)
  for (int i=0; i<n; i++)
    p[i] += q[i];
  for (int i=0; i<n; i++)
    printf("%d", p[i]);
  printf("\n");
#pragma omp parallel sections
  {
#pragma omp section
    {
      free(p);
    }
#pragma omp section
    {
      free(q);
    }
  }
}
```

2. Problem 2: Wavemaker 1d

Work in directory `wavemaker1d`.

Look at the sequential program `wavemaker1d.c`. This simulation is based on the 1-dimensional wave equation. Imagine a string that is stretched taught in the horizontal direction, and is fixed at both ends, like a piano string or guitar string. When you pluck the string, the pieces of the string will move in the vertical direction according to certain physical principles, creating a wave motion.

The mathematical model records the vertical position (amplitude) of the string at each point. This data is represented as an array of doubles of length `nx`. It then iterates over time, updating this array according to the discretized 1d wave equation

```
u_next[i] = 2.0*u_curr[i] - u_prev[i] +
  k*(u_curr[i+1] + u_curr[i-1] - 2.0*u_curr[i]);
```

The basic structure of the program is very similar to 1d-diffusion. The main difference is that the update in the wave case requires knowledge of **the previous two time steps**, while the diffusion update requires knowledge of only the previous time step. Hence `wavemaker1d.c` maintains 3 copies of the data: one for the "next" time step, one for the "current" time step, and one for the "previous" time step. At the end of each time step, the three pointers are swapped cyclically, to avoid the performance hit of copying large amounts of data.

The result could be displayed with a heat map (like diffusion), but for waves it looks better to see the amplitude displayed along the y-axis, in order to simulate what an actual string looks like as the waves travel through it. This is accomplished using `ANIM_Create_graph`. Read the documentation for this function in `anim.h`, and make sure you understand it.

Run the sequential program for different values. You will see it begins with a particular initial configuration which simulates an initial "pluck" on one side. The `Makefile` provides a couple of tests, but you should run it yourself with various values. Look at the generated movies.

Your job is to use OpenMP to parallelize the program. Call your program `wavemaker1d_omp.c` and use any OpenMP constructs you like. Your program must be correct (equivalent to the original) and efficient. It should not take any additional arguments; the number of threads to use can be obtained from the environment variable `OMP_NUM_THREADS`. It should print the number of threads at the end of the first line of output, e.g.:

```
wavemaker1d: nx=20000 ny=100 width=10 k=0.005 nstep=1000 wstep=10 nthreads=16
```

A `Makefile` is provided in this directory; it compares the results with the sequential for correctness checking.

Your performance goal is to get the best possible speedup on Bridges-2 for the following parameters:

```
./wavemaker1d_omp.exec 20000000 10000 1000 0.005 1000 1000 out.anim
```

(Note: do not attempt to convert this ANIM file to GIF or MP4.)

The first thing to do is establish your baseline. Run the original sequential program on Bridges-2 on the parameters above and note the time. Create a a subdirectory of `wavemaker1d` named `bridges2` and within that subdirectory add two files, `wave_seq.sh` and `wave_seq.out` which are the bash script and the output for the sequential run.

Then add files `wave_par.sh` and `wave_par.out` for your best parallel run, also in subdirectory `bridges2`.

Finally, add a `README.txt` in `wavemaker1d` with the following data and format:

```
Sequential time (s) :  xxx.xxx
Parallel time (s)   :  xxx.xxx
Speedup             :   xx.x
```

3. Problem 3: Parallelize Megaloop

Work in directory `megaloop`. See the program `megaloop.c`. Make sure you can compile and execute it. The program takes 3 command line arguments (all natural numbers).

Parallelize this program using OpenMP directives. Call your parallel program `megaloop_omp.c`. Follow these rules:

(1) Change the `main` function only. Do not change any other part of the code.
(2) The only changes you can make are to add OpenMP directives, the characters `{` and `}`, white space, and comments. No other changes are allowed. Do not change the order of statements or anything else.
(3) The goal is to maximize concurrency: i.e., express as much parallelism as possible. If there are two statements that can safely execute in parallel, express that fact. The goal is not necessarily to get the best (smallest) time, although a better time is not a bad guide.
(4) The parallel version should be functionally equivalent to the sequential version, i.e., they should print the same result if given the same inputs. The only exception to this is some possible difference due to roundoff error.
(5) All `parallel` directives should use `default(none)`.
(6) You can and should test your program.

Note: a correct solution that obtains some parallelism is better than an incorrect solution that has a lot of parallelism.

Note: the goal is to maximize concurrency, i.e., the amount of computation that can happen at the same time as other computation. This is an abstract concept; we are not actually going to time any execution. If there are two statements which can be safely executed by two different threads, then they should be. If there is a barrier which is not necessary, it should be removed. And so on.