

1 Instructions

When you are finished submit all your work through the MyClasses page for this class. Create a directory called Homework01, put each programming exercise into its own subdirectory of this directory, zip the entire Homework01 directory up into the file Homework01.zip, and then submit this zip file to Homework #1. Make sure all the code and document files are included in the zip file.

2 Programming Exercises

1. Suppose we toss darts randomly at a square dartboard whose bullseye is at the origin and whose sides are two feet in length. Suppose also that there's a circle inscribed in the square dartboard. The radius of the circle is 1 foot, and hence its area is π square feet. If the points that are hit by the darts are uniformly distributed (and we always hit the square), then the number of darts that hit inside the circle should approximately satisfy the equation

$$\frac{\text{Number in the circle}}{\text{Total number of tosses}} = \frac{\pi}{4}$$

since the ratio of the area of the circle to the area of the square is $\frac{\pi}{4}$. We can use this formula to estimate the value of π with a random number generator.

This is called a “Monte Carlo” method, since it uses randomness (the dart tosses). Write an OpenMP program that uses a Monte Carlo method to estimate π . Read in the total number of tosses before forking any threads. Use a clause to find the total number of darts hitting inside the circle. Print the result after joining all the threads. You may want to use unsigned long or unsigned long long for the number of hits in the circle and the number of tosses, since both may have to be very large to get a reasonable estimate of π .

Some specifics:

- (a) As with most of the class examples have the program compute and time a serial version and then compute and time a parallel version.
- (b) Have the program display the values, errors, and times of each run. Also have it print out the speedup and efficiency of the run.
- (c) Experiment with the different clause options and loop scheduling to get the best time out of your parallel code.
- (d) Run your program on the machines in the Linux lab and in a separate document answer the following questions.
 - i. Which clauses did you use and why? What other clauses did you try and why do you feel they did not run as well?
 - ii. What loop scheduling gave the best times? Why do you think this is the case.

- iii. From your runs on the Linux lab machines, what was your maximum speedup and how many threads did you use to get it?
- iv. From your runs on the Linux lab machines, what was your maximum efficiency and how many threads did you use to get it?

You do not want to use the `rand` function for this exercise. First, it is not thread-safe and second it runs very slowly. Here are some alternatives to `rand` to generate suitable random numbers for this exercise. Each run at different speeds so when you choose one, use it for both the serial version and the parallel version.

- (a) There is a thread-safe version of `rand`. The command is `rand_r`. The `_r` is supposed to suggest that the function is re-entrant, which is sometimes used as a synonym for thread-safe. The syntax of the call is

```
n = rand_r(&seed)
```

where `seed` is an unsigned int that just keeps track of the current “seed” of the random number generator. Note that `seed` and `n` are (in general) not the same numbers. You should use `n` for your random number and just let the function update `seed`. You will want to set the seed before the first call to `rand_r` with something like `time(0)`, but then just let `rand_r` deal with it. There needs to be different seed value for each thread you create, that is what will keep the random numbers separate. Feel free to look up the function to get some more details.

- (b) C++ also has a more sophisticated random number generator. This approach has the advantage of being able to create a different generator for each thread instead of using a global method like `rand` and `rand_r`. First include the `random` library. Then for each generator you want, you create a random device, link it to a generator, and create a generator distribution type. After it is created you can get your sequence of random numbers by calling the distribution on the generator. An example of a standard Mersenne twister engine producing a random double between -1 and 1 is below.

```
#include <random>

...

int main() {
    ...

    random_device rd;
    mt19937 gen(rd());
    uniform_real_distribution<> rng(-1.0, 1.0);

    ...

    double x = rng(gen);

    ...
}
```

- (c) You could always create one yourself. Below is an example of a linear congruential random number generator I have used for non-class purposes. Again, a different algorithm than the other two methods but has the advantage of being able to create a different generator per thread and uses a larger range of random values (unsigned long). The first block of code is LCRandGen.h and the second is LCRandGen.cpp. Feel free to use this if you wish.

```

#ifndef LCRANDGEN_H_
#define LCRANDGEN_H_

/*
 * 64 bit linear congruential pseudo-random number generator.
 */

#include <limits>

class LCRandGen {
protected:
    unsigned long m;
    unsigned long a;
    unsigned long val;

    void next();
    void init();

public:
    LCRandGen(unsigned long s = 0);

    void setSeed(unsigned long);
    void setMA(unsigned long, unsigned long);
    void setMAS(unsigned long, unsigned long, unsigned long);
    unsigned long rand();
    unsigned long rand(unsigned long, unsigned long);
    double frand();
    double frand(double, double);
    unsigned long max();
};

#endif /* LCRANDGEN_H_ */

```

```

#include "LCRandGen.h"

LCRandGen::LCRandGen(unsigned long s) {
    // From BSD, glibc used by gcc.
    // m = 1103515245;
    // a = 12345;

    // From Numerical Recipes
    // m = 1664525;
    // a = 1013904223;

    // From Musl
    // m = 6364136223846793005;
    // a = 1;

    // From MMIX by Donald Knuth
    m = 6364136223846793005;
    a = 1442695040888963407;
    val = s;
}

```

```

        init();
    }

    /*
     * Since the some of the above constants assume a modulus of 2^(31) - 1 we
     * apply the
     * next function several times to avoid small numbers at the start.
     */
    void LCRandGen::init() {
        for (int i = 0; i < 10; i++)
            next();
    }

    void LCRandGen::next() {
        val = val * m + a;
    }

    void LCRandGen::setSeed(unsigned long s) {
        val = s;
        init();
    }

    void LCRandGen::setMA(unsigned long mv, unsigned long av) {
        m = mv;
        a = av;
        init();
    }

    void LCRandGen::setMAS(unsigned long mv, unsigned long av, unsigned long s)
    {
        m = mv;
        a = av;
        val = s;
        init();
    }

    unsigned long LCRandGen::rand() {
        next();
        return val;
    }

    unsigned long LCRandGen::rand(unsigned long min, unsigned long max) {
        next();
        return val % (max - min) + min;
    }

    double LCRandGen::frand() {
        next();
        return static_cast<double>(val) / std::numeric_limits<unsigned long>::
            max();
    }

    double LCRandGen::frand(double min, double max) {
        return frand() * (max - min) + min;
    }

    unsigned long LCRandGen::max() {
        return std::numeric_limits<unsigned long>::max();
    }

```

2. In the exercise section of the Pacheco and Malensek book (p. 419) they define the counting sort as

Algorithm 1 Counting Sort Algorithm

```

1: procedure COUNTSORT( $A, n$ )                                ▷ Array  $A$  of size  $n$ .
2:   Create array  $temp$  same size as  $A$ .
3:   for  $i$  from 0 to  $n - 1$  do                                ▷ Inclusive
4:      $count \leftarrow 0$ 
5:     for  $j$  from 0 to  $n - 1$  do                                ▷ Inclusive
6:       if  $A[j] < A[i]$  then
7:         Increment  $count$ 
8:       else if  $A[j] = A[i]$  and  $j < i$  then
9:         Increment  $count$ 
10:      end if
11:    end for
12:     $temp[count] \leftarrow A[i]$ 
13:  end for
14:  Copy  $temp$  to  $A$  and delete  $temp$ .
15: end procedure

```

In their text you can read the description and their code. They are using C and you can substitute the C++ equivalencies if you wish. Write a program that does both a serial and a parallel implementation of this sort and test it. Some specifics:

- (a) Let the user input the size of the array, and populate the array with random integers.
- (b) Have the program sort the array of integers and time a serial version and then sort and time a parallel version. Make sure that you use the same initial array for both algorithms. You don't want to sort an already sorted array.
- (c) Include a function that will test if an array is sorted. After you sort the arrays always send them through this test do you know if the algorithms and your parallelizations are working.
- (d) Have the program display the times of each run. Also have it print out the speedup and efficiency of the run.
- (e) Experiment with the different clause options and loop scheduling to get the best time out of your parallel code.
- (f) Run your program on the machines in the Linux lab and in a separate document answer the following questions.
 - i. Which loops and which clauses did you use and why? What other clauses and loops did you try and why do you feel they did not run as well?
 - ii. What loop scheduling gave the best times? Why do you think this is the case.
 - iii. From your runs on the Linux lab machines, what was your maximum speedup and how many threads did you use to get it?
 - iv. From your runs on the Linux lab machines, what was your maximum efficiency and how many threads did you use to get it?

- v. Update the program to include functions for the insertion sort, and for the sort function in the algorithm library.
 - vi. Time the 4 sorts, counting serial, counting parallel, insertion, and sort, and compare these.
3. Another sort that is along the same lines as the counting sort above has an implementation of the following. With the implementation below it only sorts arrays of positive integers. It can be generalized to other data types if we use key-value pairs and an integer key. In the code, *A* is the array to be sorted, *sz* is its size and *max* is the largest element in the array. The *max* parameter can easily be removed if we simply found the *max* before we created the counts array. This would only be an $O(n)$ operation that was probably done anyway before the call to this function. As you can see, this is not an in-place sorting algorithm, neither was the counting sort from Pacheco and Malensek.

```

1 void intsort(int *A, int sz, int max) {
2     int *counts = new int[max + 1]();
3     int *temp = new int[sz];
4
5     for (int i = 0; i < sz; i++)
6         counts[A[i]]++;
7
8     for (int i = 1; i < max + 1; i++)
9         counts[i] += counts[i - 1];
10
11    for (int i = 0; i < sz; i++)
12        temp[--counts[A[i]]] = A[i];
13
14    copy(temp, temp + sz, A);
15    delete[] temp;
16    delete[] counts;
17 }

```

Write a program that incorporates this function and a parallel version of the function. Specifically,

- (a) Let the user input the size of the array, and populate the array with random positive integers, (≥ 1).
- (b) Write a parallel version of this function. When attacking this determine which loops can be parallelized, how much can you put in a single parallel region and where do you need to join. Do not change the algorithm for this part.
- (c) Have the program sort the array of integers and time a serial version and then sort and time a parallel version. Make sure that you use the same initial array for both algorithms. You don't want to sort an already sorted array.
- (d) Include a function that will test if an array is sorted. After you sort the arrays always send them through this test do you know if the algorithms and your parallelizations are working.
- (e) Have the program display the times of each run. Also have it print out the speedup and efficiency of the run.

- (f) Experiment with the different clause options and loop scheduling to get the best time out of your parallel code.
 - (g) Run your program on the machines in the Linux lab and in a separate document answer the following questions.
 - i. Which loops and which clauses did you use and why? What other clauses and loops did you try and why do you feel they did not run as well?
 - ii. What loop scheduling gave the best times? Why do you think this is the case.
 - iii. From your runs on the Linux lab machines, what was your maximum speedup and how many threads did you use to get it?
 - iv. From your runs on the Linux lab machines, what was your maximum efficiency and how many threads did you use to get it?
 - v. Update the program to include the `sort` function in the algorithm library.
 - vi. Time the 3 sorts, serial, parallel, and sort, and compare these.
4. Take the `intsort` function from the last exercise and trace through the algorithm by hand with small examples, say arrays of size 7 or 8 and entries between 1 and 10. Specifically, start with an unsorted array, look at the arrays `temp` and `counts` that are produced, compare these arrays with the sorted version of the input array, look at the arrays entries and the indexes of those entries. See if you can do a revision of the algorithm that can be parallelized further than you did in the previous exercise. Hint, you can. Incorporate this new version into your program from the last exercise but also keep the previous version, we will compare them. As before,
- (a) Let the user input the size of the array, and populate the array with random positive integers, (≥ 1).
 - (b) Incorporate this new version into your program keeping the serial and previous parallel versions.
 - (c) Have the program sort the array of integers and time the serial version and then sort and time both parallel versions. Make sure that you use the same initial array for both algorithms. You don't want to sort an already sorted array. Also time the `sort` function from the algorithm library.
 - (d) Include a function that will test if an array is sorted. After you sort the arrays always send them through this test do you know if the algorithms and your parallelizations are working.
 - (e) Have the program display the times of each run. Also have it print out the speedup and efficiency of the run.
 - (f) Experiment with the different clause options and loop scheduling to get the best time out of your parallel code.
 - (g) Run your program on the machines in the Linux lab and in a separate document answer the following questions.
 - i. What from your new algorithm could be parallelized further than the previous one? And why?

- ii. What loop scheduling gave the best times? Why do you think this is the case.
- iii. From your runs on the Linux lab machines, what was your maximum speedup and how many threads did you use to get it?
- iv. From your runs on the Linux lab machines, what was your maximum efficiency and how many threads did you use to get it?
- v. Time the 4 sorts, serial, parallel, updated parallel, and sort, and compare these.