Homework 1: Parallel and Distributed Computing

DUE: See exact time on blackboard

ICSI 520

This homework has three parts. Part 1: Optimize a given problem without threads. Parts 2, 3: Introduction to pThreads

Part 1

Your task is to optimize matrix multiplication (matmul) code to run fast(er) on a single processor core of Ulabany's RIT cluster.

We consider a special case of matmul:

C := C + A*B

where A, B, and C are n x n matrices. This can be performed using $2n^3$ floating point operations (n^3 adds, n^3 multiplies), as in the following pseudocode:

```
for i = 1 to n
  for j = 1 to n
    for k = 1 to n
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
    end
  end
end
```

Instructions

- Your grade will mostly depend on two factors:
  - Performance sustained by your code on the cluster,
  - Explanations of the performance features you observed (including what didn't work)
- I will not grade submissions that do fewer computations than the $2n^3$ algorithm
- Your code must use double-precision to represent real numbers
- Use gcc as your compiler
- Besides compiler intrinsic functions and built-ins, your code must only call into the C standard library.
- The matrices are all stored in column-major order
- Submission will include the following
  - The naïve implementation of above algorithm
  - The optimized implementation
  - Document describing your observations, speedup and the optimization technique used
  - Follow other submission instructions at the end

Part 2

In this part you will learn how threading helps speed up computations by leveraging multicore systems. You will use pThreads.

You will do a serial and a parallel implementation and show the speedup.

- Write the following programs in C
    - A serial program
        - Write a function void SumUpto(int number). This function sums all the numbers from 0 to "*number*" and prints the sum on stdout
        - Call this function from main. The value for "*number"* should be passed in as a command line parameter.
        - Once this works, enclose the function call in a for loop (0 to *p)*
            - The for loop upper limit (*p)* parameter should be passed in as a command line parameter (if no parameter is specified, use default as 2)
        - Time the execution (start of main to end of main) – time in microseconds.
            - See timing example on blackboard
    - A parallel program using pThreads
        - Write the thread function void * SumUpto(void *arg). Note this is same as above. You can pass in the "number" as the thread argument (*arg*)
        - Create threads in main using the *SumUpto* function
            - Use command line argument to determine how many threads *(p)* to create. If no argument is specified, use 2
            - The value for "*number"* should also be passed in as a command line parameter.
        - Measure the execution time and compare to corresponding serial implementation using the table below

Fill the following table and submit it on blackboard along with your code.

Number (summing upto this number)	#Times executed (# of threads) *p*	Serial execution time (microseconds)	Parallel execution time (microseconds)
1000	2		
10000	2		
100000	2		
1000000	2		
1000	8		
10000	8		
100000	8		
1000000	8		
1000	16		
10000	16		
100000	16		
1000000	16		
1000	36		
10000	36		
100000	36		
1000000	36		

-

Part 3

Given is a matrix of NxN integers (you create this). The matrix is initialized to random values between 1 and 10 inclusive. The problem is to calculate the distribution of values, i.e., the number of elements that are 1, the number that are 2, etc.

Divide the work up among P worker processes (threads). The matrix is shared by the workers.

You may **NOT** assume that N is a multiple of P. Divide the matrix into *almost* P equal-size strips of rows. Each worker will ONLY calculate the distribution of its strip(s). When all workers have completed their tasks, **only one** worker should accumulate and print the distributions.

Use the C rand number function with seed to generate psuedo-random numbers.

The values of N and P should be read as command-line arguments. Write the results to standard output.

Run experiments for N between 64 - 1000 and P between, 2 to 64. (NOTE: for serial code, P =1)

N and P are command line args

Describe your results. What do you observe? Why?
Create tabular results (can add more rows if needed)

N	P	Serial execution time (microseconds)	Parallel execution time (microseconds)

Grading

Serial Implementation	20%
Parallel Implementation	20%
Correct execution Parallel (speedup achievements)	35%
Correct serial execution	20%
Code clarity	10%
Observations	5%

Observations can be submitted as a text file or as comments on blackboard. These are your observations on the results you achieve.

Code clarity includes

- Use of variable names
- Function naming
- File names
- Formatting
- Comments where needed
- Makefile

Late submissions do not get ANY points for bonus (if applicable) plus they are subject to at least 20% grade deduction as per syllabus.

Reminder: If one or any of your binaries don't execute on student.rit.albany.edu, you get 0 points for this and any other homework where that happens.

Submission:

Your submission should include the following:

1. **Makefile** that builds all required binaries:
    a. Part_*N*_*type*.out where *N* is the homework part and *type* can be serial/parallel
2. Serial code source file(s) for each part of the homework
3. Parallel code source file(s) for each part of the homework
4. Document (word or excel) detailing the tabular results

All of the above should be zipped up into one file and uploaded on blackboard

If you are doing the bonus, add a comment in your code about it. (if applicable)

Naming convention of your zipped file – If you do not follow this, you get zero points:

FirstName_LastName_Homework_<n>.zip

Where n is the homework number.