Universiteit van Amsterdam
BSc Informatica
Dr Clemens Grelck

Concurrency en
Parallel Programmeren

Assignment 1

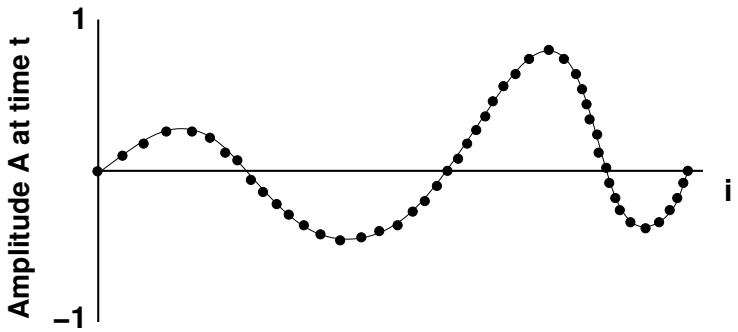
Parallel Programming for Shared Memory Systems

Assignment 1.1: Wave equation simulation with PThreads
(Code: 20%, Report: 20%)

Consider the following 1-dimensional wave equation:

$$A_{i,t+1} \;=\; 2 \times A_{i,t} - A_{i,t-1} \;+\; c \times (A_{i-1,t} \;-\; (2 \times A_{i,t} - A_{i+1,t}))$$

The wave equation describes the movement of a wave in a time- and space-discretized way. Space discretization here means that we represent the wave amplitude not as a continuous function, but as a vector of values. Time discretization means that we simulate continuous motion as a sequence of equi-distant time steps. Thus, the above formula defines the wave's amplitude $A_{i,t+1}$ at location i and time step $t+1$ as a recurrence relation of the current (t) and previous ($t-1$) amplitude at the same location (i) and the current amplitude at the neighbouring locations to the left ($i-1$) and to the right ($i+1$). The amplitude at locations with the least and the greatest index shall be fixed to zero, as illustrated below. In the above wave equation c is a constant that defines the spatial impact, i.e. the weight of the left and right neighbours' values on the new value. Its concrete value is irrelevant from the perspective of parallel program organization; let us work with 0.15.



Write a multithreaded C program that uses multiple cores to simulate the above wave equation in parallel. The program shall be parameterized over the number of discrete amplitude points and the number of discrete time steps to be simulated, as well as the number of parallel threads to be used. Use three equally sized buffers to store the three generations of the wave needed simultaneously. Rotate the buffers after each time step.

Two initial generations of the wave shall be read in from file before starting the simulation, and the final wave shall be written to file after completing the simulation. Parallelize your program by creating the given number of additional threads and let all threads collaboratively simulate each time step. Divide the work appropriately among the threads. A high resolution timer shall be

used to measure the time needed for simulation, excluding setup, input and output times. Print the measured execution time and the normalized time, i.e. measured time divided by the number of amplitude points and time steps simulated, to the standard output stream.

On Blackboard we provide an implementation framework that takes care of parameter interpretation and timing as well as input and output of wave data in order to liberate you from writing a lot of boiler plate code irrelevant to parallel computing. The use of this framework is mandatory; details are provided during the lab.

Run experiments with different problem sizes, i.e. number of amplitude points being 10^3, 10^4, 10^5, 10^6, 10^7. Adjust the number of time steps to yield a simulation time that allows for reliable timing of parallel execution without excessively waiting for results, i.e. roughly between 10 and 100 seconds. Report your results as speedup graphs: sequential execution time divided by parallel execution time using 1, 2, 4, 6 and 8 threads, assuming an 8-core machine for experimentation.

Assignment 1.2: Wave equation simulation with OpenMP

(Code: 15%, Report: 15%)

Reconsider the 1-dimensional wave equation studied in assignment 1.1. Parallelize your sequential simulation code using OpenMP and repeat the experiments of assignment 1.1. Report your results and compare them with those obtained by your PThread-based code developed for assignment 1.1.

Experiment with different scheduling techniques and block sizes using the number of threads that achieved the highest performance in your previous experiments. Report your results and explain why some schedulings perform better than others.

Assignment 1.3: Sieve of Eratosthenes

(Code: 20%, Report: 10%)

The Sieve of Eratosthenes is an ancient algorithm to compute prime numbers, attributed to the Greek mathematician Eratosthenes of Cyrene (276 BC – 194 BC). Write a multithreaded C program that prints the unbounded list of prime numbers to the standard output stream following the approach of Eratosthenes as detailed below.

The (multithreaded) sieve consists of one generator thread that generates the (a-priori) unbounded sequence of natural numbers starting with the number two (by definition the least prime number) and a pipeline of filter threads that each filters out multiples of a certain prime number from the sequence of prime number candidates (initially all natural numbers). The threads shall communicate with each other solely by means of bounded queues, as introduced in the lecture.

The generator thread has a single output queue to which it sends the natural numbers generated; it is a *producer*. Each filter thread is connected to two distinct queues: an inbound queue where it receives natural numbers as prime number candidates and an outbound queue to which it sends all received numbers that are not multiples of the filter thread's filter number and, hence, remain prime number candidates. Consequently, filter threads are both *consumers* and *producers*.

Upon creation a filter thread receives the address of its input queue as an argument. The first number it receives from the input queue is a prime number, which it prints to the standard output stream. Any subsequently received number that is a multiple of the initial prime number is discarded. All other numbers received on the input queue are forwarded to the output queue.

The first time this happens the filter thread instantiates a new output queue and creates a further filter thread, which uses the new queue as input queue.

The Sieve of Eratosthenes is meant to generate the infinite list of prime numbers. Thus, it is not needed to terminate threads or gracefully shutdown the communication links between threads. Use CTRL-C to terminate program execution when you have seen enough prime numbers. Obviously this interpretation of the Sieve of Eratosthenes is not the most efficient way of computing prime numbers; it is rather meant as a programming exercise to study pipeline parallelism and event handling in multithreaded programming.

Report the time it takes to determine the first 100, 1000, 5000 prime numbers using all available cores on the experimental system.

Restrictions on using PThreads and OpenMP:

You are only permitted to use those features that appear somewhere on the slides!

Instructions for submission:

For each assignment we expect two kinds of deliverables:

a) source code in the form of a tar-archive of all relevant files that make up the solution of a programming exercise with an adequate amount of comments ready to be compiled; and

b) a report in the form of a pdf file that explains the developed solution at a higher level of abstraction, illustrates and discusses the outcomes of experiments and draws conclusions from the observations made.

Please, submit one archive with all code files and separately one pdf-file with your report for the entire assignment.

Instructions for writing reports:

Your report is supposed to explain your solution to someone who is familiar with programming, in our case in C, including the parallel programming paradigms used throughout the course. Briefly summarize the assignment as an introduction. Normally defining the research question would be major part of such a report, but as this is given as an assignment, do not waste time to rephrase the assignment in all details.

Then describe your solution at a high level of abstraction, i.e. do not copy the whole program code into the report. It is, however, often relevant and interesting to copy some lines of code that either contain the key solution to the given problem or that you find for whatever reason interesting to talk about. If needed such code snippets can also be simplified and beautified to improve readability.

In this part of the report you should also describe why you came up with your solution, what alternatives you considered and rejected for what reasons, etc. Anything you find remarkable or super smart about your solution should be elaborated on here. Convince the reader (us) that your solution is the best thing since sliced bread.

If you feel that your solution is not the best thing since sliced bread, also discuss that. Explain potential shortcomings and failures; explain why you couldn't solve them (e.g. submission deadline

was 5 minutes ahead when you figured it out) and sketch out what you think would be a way forward. This can be the basis for a good report, even if the programming exercise did not work out that well for you.

Various assignments ask you to run certain experiments. Reporting on the outcome of these experiments is a critical part of your report. Whenever possible, present your findings in a graphical way and discuss them in the text. If you feel more or different experiments could be interesting, run them as well and report on them.

Try to draw conclusions from the experiments. Why does this code perform better than that code. Why does this code scale to 4 cores, but not to 8, etc. All these are interesting and relevant questions as well as an opportunity for you to demonstrate your knowledge.

We do not prescribe minimum or maximum page numbers, but it should be clear by now that writing the report is a significant part of each assignment series. Take this into account when planning your time. Your job is not done when your code compiles without errors!

Code due date: See Blackboard
Report due date: See Blackboard