University of Amsterdam
BSc Informatica
Dr Clemens Grelck

Concurrency en
Parallel Programmeren
2012/2013

Assignment 1

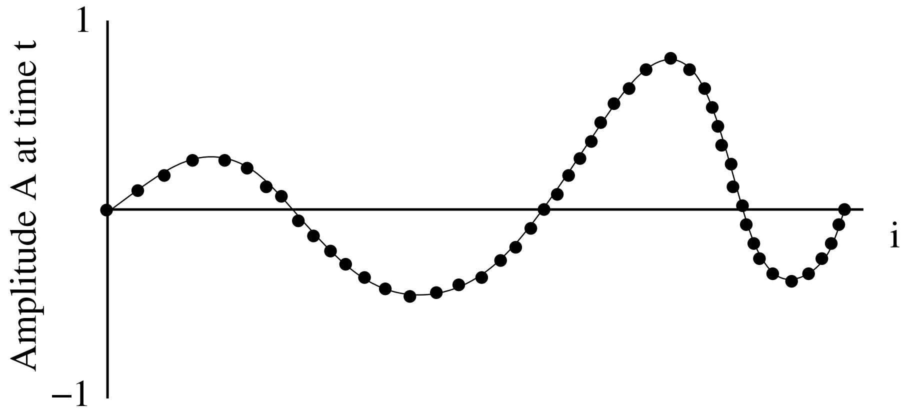
Multithreaded Programming with Posix Threads

Assignment 1.1: Wave equation simulation

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 organisation; let us work with 0.2.



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 dicrete time steps to be simulated, as well as the number of parallel threads to be used. Use three equal-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. Parallelise 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.

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 and 6 threads, assuming a 6-core machine for experimentation.

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 given in the first lab session.

Assignment 1.2: Sieve of Eratosthenes

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.

Assignment due date: November 8, 2012, 12:00