

Pthreads programming

1 Compile and run Pthreads programs

Copy the tar file pthreads.tar from the course homepage. Study the Makefile to see how to compile Pthreads programs. Compile the program helloworld, make helloworld. Run the program by typing the program name ./helloworld. Study the program source file and compare with the output. Re-run the program on more than one thread. Modify the program to also print the thread id, i.e., pass the thread number to the HelloWorld function.

If you want to pass more than one argument you must use structs. For this purpose, study and run the program hello_arg2.c (make hello2). Note also how the struct is used to return values from the threads. Modify the program to pass different messages to the different threads, e.g., by using the messages array.

2 Global and local data

In the thread model all global data are accessible to all threads while data defined within the thread functions are local and only accessible to the calling thread. In the program data.c we have two global arrays and each thread is reading and writing to them. Run the program and study its output. Is the value of the sum variable what you expect, re-run the program several times and explain the results.

3 Thread management

In Pthreads the threads can be defined as joinable or detached. What is the difference of joinable and detached? Run the program join.c (make join) and study the program flow. What would happen if we set the attribute to detached and remove the pthread_join function? What would happen if we also remove pthread_exit call in end of main?

4 Mutex Variables

Mutex variables (or locks) are used for protecting shared (global) data when multiple writes occur. Study and run the program mutex.c (make mutex). What would happen if we don't use a mutex in the thread function, what will the value of sum be? Re-compile without the -O3 flag and re-run the program, what happens?

5 Condition Variables

A condition variable is used for synchronization of threads. It allows a thread to block (sleep) until a specified condition is reached. In the program `synch.c` a global barrier is implemented using a condition variable, study and run the program. What happens if we remove the barrier, how will the output change? Another way to implement a barrier would be to constantly read a global variable until it changes. This is implemented in `spinwait.c`. Why does this not work? How can we fix this? (*Hint*: What is the meaning of the keyword `volatile` in C?)

6 Examples

Below we have four different applications, a sorting algorithm, a numerical integration algorithm, a numerical PDE solver and the Gram-Schmidt orthogonalization process. We will look at different ways to parallelize these algorithms and compare their performance.

- The Enumeration-sort algorithm

For each element (j) count the number of elements (i) that are smaller ($a(i) < a(j)$). This gives the rank of $a(j)$, i.e. $anew(rank(j))=a(j)$.

Finding the rank of an element is an independent and perfectly parallel task, i.e., finding ranks of two different elements can be computed concurrently. This strategy is implemented in the program `enumsort_1.c`. Study, compile (make `enum1`) and run the program.

The overhead in creating a thread for finding the rank of one element is too high and the efficiency becomes poor. A better approach is to create tasks where the rank of several elements are found by one thread within the task. This is implemented in the program `enumsort_2.c`. Study, compile (make `enum2`) and run the program.

- The program `pi` computes π in parallel using numerical integration. A numerical way to compute π is the following:

$$\int_0^1 \frac{4}{1+x^2} dx = [4 \arctan(x)]_0^1 = \pi$$

If we use the midpoint rule, we can compute the above integral as follows (see Figure 1):

$$\int_0^1 \frac{4}{1+x^2} dx = h \sum_{i=1}^n \frac{4}{1+x_i^2} = h \sum_{i=1}^n \frac{4}{1+((i-1/2)h)^2}.$$

The trivial parallel implementation of the latter formula is that if we have p threads available, we slice the sum into p pieces, attach one interval to each of them to compute a partial sum, and then each thread adds its partial sum to a global sum. Parallelize the computations in program `pi.c` with appropriate Pthreads functions using global and local variables.

- Consider the program `matmul.c` (make `matmul`). Here, two matrices A and B are multiplied and the result is the matrix C. Find the parallelism in the computations and create parallel tasks for the threads.

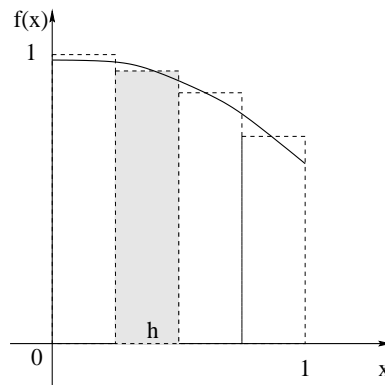


Figure 1: Numerical integration with the midpoint rule.

What is the maximal speedup of your parallelization on 1000x1000 matrices? For how small matrices is it worth doing the parallelization? (Remark, the code is not optimal due to cache performance.)

- The program `wave.cc` solves a PDE numerically using the finite difference method Leapfrog. The program is parallelized with Pthreads and uses two different ways of synchronization of the threads, a global barrier and non-synchronize check pointing. You can choose the way of synchronization by setting the parameter `BARRIER` in the file `wave.h` to either 1 (check point: “overlap”) or to 3 (global barrier: “synchronize”). The problem size and number of threads can be set in the file `wave.dat`. Log on to gullviva (`xrlogin gullviva`) and compare the different ways of synchronization for different problem settings, is there any difference and for what settings? Note, Gullviva has 32 cores compared to 8 cores on the login nodes. Compile by `make wave`.
- In the lectures we have discussed (or will discuss) different ways to parallelize the Gram-Schmidt orthogonalization process, a simple version using only `create/join` and a more advanced way with mutex variables to avoid global synchronization. The different parallelization strategies are implemented in the programs `gram2.c` and `gram3.c`, respectively. Study the programs and make sure you understand the implementations. Compile and run the different versions for different problem settings. The problem size is given at run time, e.g., give `./gram2 1000` for running with 1000x1000 vectors. The number of threads is set as a constant in the source files. Plot speedup curves for respective implementation.

7 Report

The laboration is a part of the examination. To pass you need to write a short informal report summarizing your results and answers together with your source code for the different tasks.