November 7, 2017

DVA336 Parallel Systems

# Assignment 1

*Introduction to Shared Memory Programming*

## Introduction

The purpose of this lab is to get acquainted with shared memory programming using Pthreads (POSIX threads) and OpenMP.

You can compile all the C source files found in `src/` at once, with the command make, unless stated otherwise below. You could also compile an individual `target` with the command make target. Study the Makefile (`src/Makefile`) to see how the targets are compiled.

The Pthread `man`-pages or e.g. this tutorial hold more information about the Pthreads API. The OpenMP specification manual (available here) or e.g. this tutorial hold more information about the OpenMP API.

## Preparation

Read Chapter 7 *Programming Shared Address Space Platforms* in the main textbook *Introduction to Parallel Computing, 2nd Edition*, by Grama et al. and/or have a look at the tutorials listed above.

## Examination

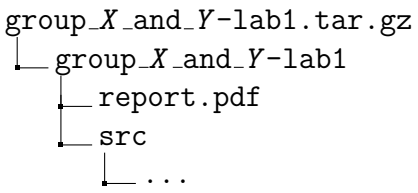
You should solve and hand in a solution for this lab in groups of **two**. Of course, it is required that each member of the group can explain the details of the group's solution individually. Therefore, you are encouraged to work on solving the different exercises on your own, and then discuss different alternatives together with your partner. This will help you get a deeper understanding of what you are doing and why you are doing it. Furthermore, having solved the exercises on your own will be to your advantage on the written exam.

Send in your gzipped tar archive-file by e-mail to the course assistant. The archive file should be called `group_X_and_Y-lab1.tar.gz`. This archive should contain *only* one folder, called `group_X_and_Y-lab1`. This folder should in turn contain

a report, called `report.pdf`, and a folder called `src`. The file `report.pdf` should contain answers to, results from, and reasoning about the questions in this lab-pm (explain the reasons for the phenomena that occur). The `src` folder should contain your modified source code and all files needed to compile your programs; you should not need to modify the provided `Makefile`. In the file/folder names above, the name *X*`_and_`*Y* should identify you, i.e., *X* and *Y* should be replaced with 10-character strings that clearly identifies you as the two students of the group. Example: Katy Andersson, born March 11, 1991, and Larry David, born July 2, 1947 name their file

```
group_19910311KA_and_19470702LD-lab1.tar.qz
```

The structure and naming convention of the archive file is depicted below.




Assuming you have a folder called `group_`*X*`_and_`*Y*`-lab1`, you can create the archive-file with the following command.

```
$ tar -czf group_X_and_Y-lab1.tar.gz  group_X_and_Y-lab1
```

# Hello World!

In the files `src/hellopt.c` and `src/hellomp.c` there are two threaded versions of a "Hello World!" program, implemented using Pthreads and OpenMP respectively.

1. Study and run the programs, then modify them so that each thread prints "`Hello World! ID,TOTAL`", where `ID` is the thread ID (which belongs to the interval `[0, ..., TOTAL-1]`) and `TOTAL` is the total number of threads.

   *Hint Pthreads*: Give argument to the executed function.

2. When is "`Goodbye World...`" printed?

3. In `src/hellopt.c`, make the threads detached (instead of joinable) and run the program 10 times more. When is "`Goodbye World...`" printed now? Why is this so?

4. In `src/hellomp.c`, add `nowait` to the `single` construct and run the program again. When is "`Goodbye World...`" printed now? Why did the threads synchronize before?

5. In `src/hellopt.c`, comment the `pthread_exit()` call in `main()` (also keep the threads detached) and run the program a couple of times more. What happens now?

## Global & Local Data – Critical Sections

In the POSIX thread model, variables are declared as global or local in the same way as for an ordinary sequential program. Global variables are declared in the global region of the code (outside any function definition) and are accessible by all threads. Local variables are declared within the function(s) executed by the threads and are accessible only by one thread (within the specific instance of the function, just like in single-threaded C programs).

1. In `src/globalpt.c`, we have a program which initializes an array of `n` elements to `[0,1, ..., n-1]`, using Pthreads. Study the program, run it and make a note of the execution time. Is the output always "`Correct`"?
2. Obviously, there is a race condition on the globally declared `put_index`. Use a mutex to protect `put_index` and run the program again. Make sure that the output is "`Correct`" and make a note of the execution time.

   *Hint*: Remember to correctly *allocate*, *initialize* and *destroy* the mutex.
3. Instead of protecting the shared (global) variable with a mutex, we can statically divide the array into sub-arrays and let each sub-array be initialized by one thread, using a thread-local index variable. Implement this solution; make sure that the output is always "`Correct`" and make a note of the execution time.

   *Hint*: Make the thread ID (`i`) available to `body()` and use this to decide which elements should be initialized by the thread.
4. Explain the difference in the way that the two approaches (mutex and local index) offer parallel efficiency. Compare to the first (unprotected) approach.

By default, all variables in an OpenMP program are globally accessible by all threads. Some of the variables can optionally be made local to each thread by using the `private` clause.

5. In `src/globalmp.c` there is a sequential version of the program found in `src/globalpt.c` (which of course does not suffer from the race condition in `src/globalpt.c`). Run this program and make a note of the execution time.
6. Parallelize the code using the `for` directive (note that `i` inherently becomes `private` when using the `for` directive). Run the program and compare the execution time to that of the sequential version of the code.

   *Hint*: Remember to include `omp.h`.

7. In `src/privatemp.c`, we have a simple loop construct. Notice how the `private` clause is used to make `tmp` private to each thread. Run the program.

8. As you might notice, `tmp` is not initialized (it does not keep its "global" value) in the threads. This issue can be solved by using `firstprivate`. Make this change and run the program. What is the output?

9. As you might notice, `tmp` does not change in the "global" scope (sequential region) after the parallel region. We can store the value of `tmp` in the last iteration of the loop in the "global" `tmp` by adding the clause `lastprivate` to the pragma. Make this change and run the program. What is the output?

10. In `src/criticalmp.c`, we have a race condition on the shared variable `globsum`. If you run the program 20 times, you will probably not be able to produce the same result all times. Fix this problem by using the `critical` or `atomic` construct. (You will see a better solution in the last exercise of this lab, "The Scalar Product".) Run the program. What is the final value of `globsum`?

# Explicit Synchronization

The POSIX threads API supplies condition variables (`pthread_cond_t`) and barriers (`pthread_barrier_t`) to the programmer as a means of synchronizing threads. You will see how to use condition variables in lab 2.

1. Compile and run the program in `src/synchpt.c`. When is "`Done`" printed?

2. Make the threads synchronize before "`Done`" is printed, using a barrier. Can the printing of "`Thread with interval...`" and "`Done`" be interleaved?

In OpenMP, synchronization is implicitly performed at the end of the `parallel`, `do/for`, `sections` and `single` constructs. You have already seen an example of how to override this in the `single` with `nowait` exercise. Explicit synchronization can also be performed using the `barrier` construct.

3. In `src/synchmp.c`, we have an OpenMP parallel program, similar to the one in the previous exercise. When is "`Done`" printed?

4. Again make the threads synchronize before "`Done`" is printed, using a barrier. Can the printing of "`Thread with interval...`" and "`Done`" be interleaved?

# Auxiliary OpenMP Constructs

1. In `src/orderedmp.c`, there is a program consisting of a loop that is split between 4 threads which print the loop index `i`. Run the program a couple of times. Note that the printing of the indices is done interleavingly.

2. Now change the program so that the index `i` is printed in an ordered fashion (i.e. starting with `i=0` and ending with `i=loop_limit-1`).

   *Hint*: Use the `ordered` construct.

3. Copy the file `src/orderedmp.c` into a new file, called `src/mastermp.c`.

4. Modify the code in `src/mastermp.c` so that (only) the master thread prints "`Thread thread_number is the master`" immediately after the ordered loop. You could also play with the `nowait` clause on the `for` construct to allow the master's number to be printed earlier. Compile the program with "`make mastermp`" and run it. Which thread is the master thread?

The `sections` construct can be used to achieve a noniterative worksharing. Each specified section is executed once by one of the available threads. As you might imagine, this style of execution is easily performed using Pthreads as well – simply assign different functions to the different threads.

5. In the file `sectionsmp.c`, four functions are called sequentially. Run the program a couple of times and make a note of the execution patterns (in which order and by which threads are the functions executed).

6. Make each of the function calls be made within a `section` construct. Compile and run the program a couple of times. How does the execution pattern change? Why?

7. Enable nested parallelism by calling "`omp_set_nested(1)`"; i.e. change the current "`omp_set_nested(0)`" statement. Compile and run the program. How does the execution pattern change? Why? What is achieved by using nested parallelism?

   *Hint*: See the OpenMP specification for complimentary information on `nested` parallelism.

# The Scalar Product

Now you will write a parallel program that calculates the scalar product of two vectors, $A = [a_1, a_2, \ldots, a_i, \ldots, a_n]$ and $B = [b_1, b_2, \ldots, b_i, \ldots, b_n]$. Recall that the scalar (or dot) product is calculated as:

$$A \cdot B = \sum_{i=1}^{n} a_i b_i$$

In our case the value of $n$ is predefined to 1000000 elements.

In the files `src/scalarpt.c` and `src/scalarmp.c`, there are skeleton programs which you should use to implement a Pthreads and an OpenMP version respectively.

1. Implement a sequential version of the program in `src/scalarmp.c`. Compile and run your solution. What is the execution time?
2. What is the scalar product?
3. In `src/scalarpt.c`, parallelize your algorithm using Pthreads. Run the program using as many threads as you have cores in the CPU (use at least a quad-core processor). Measure the execution time. What is the achieved speed-up? Could you improve your solution regarding unnecessarily introduced delays (due to e.g. synchronization between the threads)?

   *Hint*: Remember that mutual exclusion constraints on shared variables introduce delays.

   *Hint*: The local result of a thread can be returned to the join-function in the `status` variable.
4. What is the scalar product?
5. In `src/scalarmp.c`, parallelize your algorithm using OpenMP. Remember that you could e.g. use `critical` to protect shared variables. Here you should instead use `reduction` to allow the compiler to optimize your code in a very scalable way (the given operation is performed in a tree-like fashion). Consult the OpenMP specification for more information regarding the `reduction` clause. Run the program and make a note of the execution time. How many threads did you use? Compare this solution regarding the achieved speed-up to the Pthread case.
6. What is the scalar product?

# Calculation of $\pi$

The value of $\pi$ can be numerically calculated using the following formula:

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

By using the midpoint rule, we can approximate the above integral as follows:

$$\pi = \int_0^1 \frac{4}{1+x^2} dx = [4 \arctan(x)]_0^1 \approx h \sum_{i=1}^{n} \frac{4}{1+x_i^2} = h \sum_{i=1}^{n} \frac{4}{1+((i-1/2)h)^2}$$

Here, $n$ is the number of intervals to use (the higher the $n$, the better the approximation), $h$ is the length of one interval (i.e. $h = 1/n$, and is thus comparable to $dx$ in the integral when $n$ approaches infinity).

1. Implement a sequential version of the algorithm in `src/pimp.c`. Compile and run the program (make sure that the output of your solution is correct). Make a note of the execution time.

2. In `src/pipt.c`, implement a parallel version of the algorithm using Pthreads. Compile and run your solution for all numbers of threads in the interval $[1, K]$, where $K$ is the number of physical cores in the computer you use. Make sure that the output of your solution is correct. Make notes of the execution times.

   *Hint*: Remember the difference in performance between handling data globally and locally.

3. In `src/pimp.c`, implement a parallel version of your solution using OpenMP. Compile and run your solution for all numbers of threads in the interval $[1, K]$. Again, make sure that the output of your solution is correct, and make notes of the execution times.

   *Hint*: Remember to perform the summation of the thread-individual results in an efficient manner.

4. Visualize the achieved speed-ups for the two parallelizations. Compare them to each other. A close to linear speed-up should be achievable since this problem is perfectly parallel. Could your solutions be improved?

**NOTE:** When submitting your solutions to the course assistant, remember to follow the directions given in the section Examination!

# Additional Exercises (Optional)

In this section you will learn how to parallelize some more algorithms. Evaluate the performance of your solutions by varying the number of used threads $k$ in the interval $[1, K]$, where $K$ is the number of physical cores available. Preferably, use a computer with $K \geq 8$.

## Matrix Multiplication

As you know, multiplication of the two matrices $A$ (of size $n \times p$) and $B$ (of size $p \times m$) is performed by multiplying row $i$ in $A$ with column $j$ in $B$, element-wise. The multiplied values are then added to create the corresponding value (element $i$,$j$) in the result matrix. I.e., if the result matrix is called $C$ (of size $n \times m$), and the multiplication $C = AB$ is performed, then we calculate the element $c_{ij}$ as:

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}$$

1. In the file `src/matmulpt.c`, there is a naïve sequential implementation of the matrix multiplication problem. Compile and run the program. Make a note of the execution time.
2. Change the order of the loops so that you exploit the spatial locality of the caches in a better way. Measure the execution time of your solution. What is the speed-up compared to the naïve solution?
3. Also implement cache blocking on top of the changed loop-order in order to exploit temporal locality as well. Measure the execution time of your solution for some different block sizes (e.g. $\{4, 8, 16, 32, 64, 128\}$). What is the speed-up compared to
   (a) the naïve solution?
   (b) the previous solution?
4. Finally, parallelize the algorithm from the previous step (2 or 3) using Pthreads. Measure the execution time of your solution for $[1, K]$ threads. What is the speed-up compared to
   (a) the naïve solution?
   (b) the previous solution?
5. Visualize the achieved speed-ups.
6. Where did the largest speed-up occur? Why?

## LU-Factorization

As the final exercise of this section, you will parallelize an implementation of the LU-factorization algorithm using OpenMP.

1. Compile and run the sequential version of the LU-factorization algorithm which is available in `src/lump.c`. Make a note of the execution time and the error.
2. Parallelize the relevant part of the code (the part on which time is measured) using OpenMP directives.
3. Compile and run your code using as many threads as there are physical cores on the computer you use. Make sure that the error is the same as the one printed by the sequential version of the program.

   *Hint*: Some parts of the code cannot be parallelized in an easy way. Why is that? Do *not* try to improve on the algorithm! Just parallelize the code given in `src/lump.c`.
4. What is your achieved speed-up? Is this what you expected? Why/Why not? What part of the code cannot be parallelized?

## Dining Philosophers

In this exercise, you will learn how to use some Pthread primitives to achieve a "predictable" parallel system. In the file `src/diningpt.c` there is an implementation of the classical dining philosophers problem. There are $n$ philosophers that alternatingly eat and think. When eating, the philosophers compete for the available forks. There are $n$ forks and each philosopher needs 2 forks when eating, so obviously, all philosophers cannot eat at once.

1. Study the program. What are the potential problems?
2. Compile and run it. What happens?
3. Replace the `char`-variable locking scheme with a solution based on Pthread mutexes. Note that you are not allowed to remove the call to `sleep()` inbetween the picking up of the two forks!

   *Hint*: Think carefully about how to solve this problem before starting to implement it! Remember the four conditions that must be fulfilled in order for a deadlock to occur.
4. Is deadlock and starvation avoided by your solution? Try to make use of as much parallelism as possible ($\lfloor n/2 \rfloor$, not neighboring, philosophers should be able to eat at any given point in time, if they want to)!

## A Reader/Writer System with a Buffer

In this exercise you will implement a reader/writer system where the communication is performed via a circular buffer. Study the given source `src/cliserpt.c`.

1. In the function `reader_body()`, add code that reads from the buffer if it is not empty. If the buffer is empty, the readers should wait on a condition variable that should be used to signal that the buffer is non-empty (preferably call the condition variable `non_empty`). Ensure mutual exclusion on the buffer-handling code by using a mutex.

   *Hint*: Remember that a thread can wake up from waiting on a condition variable (due to external events) even if the condition variable has not been signaled. To ensure the correct behavior, use a `while`-loop that checks your non-empty condition to encapsulate the condition wait instruction.

   *Hint*: To statically initialize a mutex or condition variable, the macros `PTHREAD_MUTEX_INITIALIZER` and `PTHREAD_COND_INITIALIZER` can be used respectively. Do not forget to destroy all mutexes and condition variables at the end of `main()` though.

2. In the function `writer_body()`, add code that writes a random integer to the buffer if it is not full. If the buffer is full, the writers should wait on a condition variable that should be used to signal that the buffer is non-full (preferably call the condition variable `non_full`). Ensure mutual exclusion on the buffer-handling code by using the same mutex as for the readers.

3. Compile and run your solution. Is there any correspondence between the returned values from the writers and readers?

4. Are you guaranteed to find every value returned by the writers among the values returned by the readers? Why/why not?

## A note on OpenMP

You could solve the two previous exercises using OpenMP. Take a minute or two to think about why Pthreads was chosen for these exercises.