Manual for the lab course exercises of Parallel algorithms and parallel computers (in4026)

Ana Varbanescu, Tamás Vinkó, Nitin Chiluka, Jie Shen, Henk Sips

version: April 7, 2013

Contents

1 Introduction

This is the manual for the lab course exercises accompanying the course Parallel Algorithms and Parallel Computers (code in4026). This manual describes the requirements of the practical exercises and gives some information on the two used languages/libraries: the POSIX Threads (pthreads) and Open-MP. In this manual we assume your knowledge of the language C is reasonable (otherwise see [2]), and that you have some experience using a UNIX or LINUX operating system. The knowledge required for these exercises can be found in [3]. Additionally we advise to take a look at some of the algorithms in [1].

1.1 Purpose

The purpose of the assignments is to achieve some insights and some experience in designing and implementing (architecture-independent) parallel algorithms. For these assignments, the correct design, and implementation of the algorithm is as important as the performance of the resulting program.

1.2 Enrolment

The lab course assistant of the course is:

- Jie Shen
  room: HB 07.280
  telephone: (0)15 27 2783830

To enrol yourself to the lab course, please send an email to the address:

`in4026@st.ewi.tudelft.nl`

Please mention your name, email-address, study and student number.

Both permanent and actual information on this assignment can be found on the blackboard site `http://blackboard.tudelft.nl`.

1.3 Assignment

The assignment contains three exercises, each of them worth 10 points. In each exercise your tasks are the following (with indication about how many points you can collect by solving them):

1. provide a sequential implementation,
2. provide a parallel implementation based on OpenMP and derived directly from your sequential solution (3 points),
3. provide a parallel implementation using POSIX Threads, also based on the same algorithm as your sequential solution (4 points),
4. and write a short report on the experiments with these codes (3 points).

We assume you require approximately 60 hours to complete all these exercises. **You have to solve your assignment on your own.** One correct and complete solution (including all three implementations) will be awarded 10 points. To pass the lab course, you need to accumulate 20 points. Note that you have to write at least one report, thus you cannot pass the lab course with only programming.

The exercises you have to solve require the design, implementation and efficiency analysis of parallel programs. The solutions of the exercises have to be sent to the official email address of the labcourse:

`in4026@st.ewi.tudelft.nl`

before the deadline announced on the course website and in the emails sent out during the semester (see `blackboard.tudelft.nl`).

A complete solution of an exercise contains:

1. the source codes of the problem (together with all additional required files - e.g., `Makefile`),
2. at least one data set (embedded in the code, generated by the code, or included in an additional file) to test the correctness of the code; keep the size of these correctness tests within readable limits (below 50 elements).
3. a brief report explaining the design and implementation of your algorithms, as well as a set of comprehensive experimental results, clearly analyzed. The design and implementation explanations should not exceed 2 pages.

A solution will usually be reviewed within two weeks. You will receive an email to inform you that your exercise is correct or that you should improve your program or your report at some points. You will get additional time to make these modifications.

If you have not finished your exercises before the final deadline, you should restart your work next year. However, the completed/graded exercises from this year can be re-used next year (e.g., if you submitted B and C and you got 9 and 7 points, respectively, you may choose to only do exercise A next year, do A and improve C, or do all A,B, and C).

1.4 Requirements

The requirements of the report and the program you have to construct for each of the three exercises are the following:

1. As an **introduction**, include the problem requirements and briefly explain your algorithm.
2. Give an **algorithm proof**: show why the algorithm will give the desired results. For example, if the exercise is: calcultate $n!$, and your algorithm is: fac(0)=1 and fac(n)=$n \cdot$ fac($n-1$) you should include a proof by induction.
3. Give an **analysis of the time-complexity** of your algorithm related to the size of the problem (usually N).
4. **Implement** first a sequential algorithm. Next, add not more than 3 OpenMP pragmas to parallelize the time consuming sections/loops of the code. Further, implement the algorithm using PThreads. Explain the transition from pseudocode (or algorithm) to the implementation, focusing especially on work and data distribution.
5. **Test your program** varying both the problem size (at least 7 different problem sizes) and the number of threads (at least 1, 2, 4, 8, and 16 threads - P). Measure the execution time, T, for each of the test cases (be sure to run the application multiple times - i.e. 10, 100, or even 1000 times, and average the results). Present your results in the following graphs: $T(N, P$ fixed), $T(P, N$ fixed), Speed-up(N, P Fixed), Speed-up(P, N fixed). Analyze the results, focusing on the difference (if any) between the measured and expected performance. See some more guidelines and requirements in subsection 1.5.
6. Include a **Conclusions section** to briefly explain what have you done and, more important, what have you learned from the given exercise and its parallelization. Comment on efficiency and usability of the given implementation language/library for the given application and, if possible, generalize.

1.5 Testing environment

To make your final tests for the reports of your implementations, it is requested to use a computer with 8 cores. If you do not have access to this kind of computer, there will be some of these reserved in the common computer room on the 9th floor of the EWI building. Details will be sent out later to all the enrolled students, and will be put online on the information board of the course.

Note that to implement and pre-test the codes, one can use even a single core machine, the 8 core computer is required only for the final tests.

Moreover, if you happen to have access to an 8 core computer with UNIX, Pthreads and OpenMP installed, you can do the final testing on that computer and report the results (with mentioning the computer you use).

1.6 General remarks

Please pay attention to the following points:

- Include your name and student number in the report.
- Use well-written English in your report.
- Every design and implementation step should directly follow from the previous steps.
- In your design and your implementation you should use existing rules of thumb, such as useful comments, clear function and variable names, functions and main not too extensive, etc.
- If you need to make certain assessments on the given input, clearly specify them (e.g., if you need that the size of the vector is a power of 2) and then you do not have to check them in the code; a brief explanation on why these limitations are needed is appreciated.
- If you have interesting design/implementation ideas or optimization ideas, be sure to emphasize them.
- If you think of eventual additional optimizations that may be done, but you have no time to actually implement them, be sure to explain them.

2 POSIX Threads

A thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system. For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c-1995 standard. Also referred as Pthreads, POSIX has became the standard threads API, supported by most vendors in one way or another.

2.1 API variables and functions

For this course, the most important variables and functions of the pthreads API are:

`pthread_t` is used to identify a thread.

`pthread_attr_t` is used to identify a thread attribute object.

`pthread_create (thread, attr, start_routine, arg)` create a new thread, with attributes specified by `attr`, within a process. If the attributes specified by `attr` are modified later, the thread's attributes shall not be affected.

`pthread_exit (status)` function shall terminate the calling thread and make the value status available to any successful join with the terminating thread.

`pthread_attr_init|destory (attr)` function shall initialize/destroy a thread attributes object.

`pthread_join (threadID, status)` function shall suspend execution of the calling thread until the target thread terminates, unless the target thread has already terminated. On return from a successful `pthread_join()` call with a non-NULL `status` argument, the value passed to `pthread_exit()` by the terminating thread shall be made available in the location referenced by `status`.

`pthread_barrier_t` is used to identify a barrier.

`pthread_barrierattr_t` is used to define a barrier attributes object.

`pthread_barrier_init (barrier, attr, count)` function shall allocate any resources required to use the barrier referenced by `barrier` and shall initialize the barrier with attributes referenced by `attr`. The `count` argument specifies the number of threads that must call `pthread_barrier_wait()` before any of them successfully return from the call.

`pthread_barrier_wait(barrier)` function shall synchronize participating threads at the barrier referenced by `barrier`. The calling thread shall block until the required number of threads have called `pthread_barrier_wait()` specifying the barrier.

`pthread_barrier_destroy(barrier)` function shall destroy the barrier referenced by `barrier` and release any resources used by the barrier.

2.2 General remarks

For all files using pthread routines use the header `<pthread.h>`. When compiling you need to specify the `libpthread.a` library (on some systems this is included in the standard C library, so this is redundant), i.e., `gcc -lpthread` in case of using UNIX standard GNU C compiler.

Threads run simultaneously. Even on a single-processor machine (where the kernel uses time slicing to simulate true multitasking) we can, from a programmer's perspective, imagine threads as executing at the same time. Each thread is uniquely identified by a thread ID of type `pthread_t`. All pthread functions return 0 on success and a non-zero error code on failure. Note that when a multi-threaded program starts executing, it has one thread running, which executes the `main()` function of the program. This is already a full-fledged thread, with its own thread ID. In order to create a new thread, the program should use the already mentioned `pthread_create()` function. When threads are created they begin executing a function, whose parameter is passed during creation.

In pthreads all arguments are passed as a void pointer. If the parameters are passed by reference to the thread, then they will be considered as shared. If multiple arguments need to be passed then a way is that one creates a structure with all the desired arguments and pass an element of that structure as a void pointer.

A typical mistake: in `pthread_create()` function the fourth parameter, `arg` is a pointer to data, and a common mistake is to pass this parameter and then overwrite it when calling a new thread. You must make sure a thread will no longer access this data before overwriting it with a new value to be passed to another thread.

Finally, some important remarks:

- Any data whose address may be determined by a thread are accessible to all threads in the process. This does not mean that a thread cannot have private data.
- A function is thread-safe if it may be safely invoked concurrently by multiple threads. Functions which use static objects are not thread safe. For example `rand()` is *not* thread-safe function.
- Writing multithreaded programs require more careful thought more difficult to debug than single threaded programs.

2.3 Further reading and references

There are several online tutorials and documentation on POSIX Threads. For convenience, we list some of them here, which might be helpful for the coding of the three exercises.

- POSIX Threads Programming,
  `https://computing.llnl.gov/tutorials/pthreads/`
- For a short introduction on the pthread synchronization primitives (barrier, mutexes and semaphores) see: PThreads primer,
  `http://pages.cs.wisc.edu/ travitch/pthreads_primer.html`
- pthread Tutorial, especially the Section 6 (Rules for Multithreaded Programming) is recommended,
  `http://www.multicoreinfo.com/research/misc/Pthread-Tutorial-Peter.pdf`
- POSIX threads explained,
  `http://www.ibm.com/developerworks/library/l-posix1.html`
- glibc reference on pthreads,
  `http://fria.fri.uniza.sk/doc/glibc-doc-reference/html/POSIX-Threads.html`

3 Open-MP

OpenMP differs from pthreads in several significant ways. Where pthreads is implemented purely as a library, OpenMP is implemented as a combination of library calls and a set of compiler directives or pragmas. These directives instruct the compiler to create threads, perform synchronization operations and manage shared memory. Also, OpenMP is a platform-independent API and hence works on Unix and Windows platforms.

For the exercises of this course the most important feature of OpenMP is the for cycle parallelization, which is discussed briefly in the next subsection.

3.1 Loop parallelization

The following example[1] is a simple vector-add program. Arrays `a, b, c`, and variable `n` will be shared by all threads. Variable `i` will be private to each thread; each thread will have its own unique copy. The iterations of the loop will be distributed dynamically in `chunk` sized pieces. Threads will not synchronize upon completing their individual pieces of work (see the keyword `nowait`).

```
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000

main ()
{
   int i, chunk;
   float a[N], b[N], c[N];

   /* Some initializations */
   for (i=0; i < N; i++)
      a[i] = b[i] = i * 1.0;

   chunk = CHUNKSIZE;

   #pragma omp parallel shared(a,b,c,chunk) private(i)
   {
      #pragma omp for schedule(dynamic,chunk) nowait
      for (i=0; i < N; i++)
         c[i] = a[i] + b[i];
   }  /* end of parallel section */
}
```

[1]taken from `https://computing.llnl.gov/tutorials/openMP`.

3.2 Further reading and references

Some of the online references for OpenMP are listed below:

- OpenMP Tutorial,
  `https://computing.llnl.gov/tutorials/openMP/`
- OpenMP: An API for Writing Portable SMP Application Software,
  `www.openmp.org/presentations/sc99/sc99_tutorial.pdf`
- Brief explanation on OpenMP,
  `http://cobweb.ecn.purdue.edu/~eigenman/ECE563/Handouts/ECE563-OpenMP.pdf`

References

[1] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[2] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Bell Telephone Laboratories, Inc., 1978.

[3] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Programming*. The benjamin/Cummngs Publishing Company, Inc., 1994.