Parallel Programming Concepts WS 2011 / 2012

Assignment 1

(Submission deadline: Dec 4th 2011, 23:59 CET)

General Rules

Each of your code solutions has to fulfill the described application functionality (‚correctness') regardless of the underlying hardware. In order to test that, we will run your software on a FutureSOC lab machine with > 12 cores. A solution is correct if your program runs with the (directly or indirectly) configured number of threads and terminates without errors. After finishing, your program must terminate with exit code 0.

Two out of three tasks must be solved correctly in order to pass the assignment. Your submitted solution must contain **only the source code files, a Makefile and ready-to-execute binaries** for Windows 64-bit or Linux 2.6 64-bit. GUI applications are not allowed. Programs expecting any kind of keyboard input at any point in the execution are not allowed. Documentation should be done inside the source code. Students can submit solutions either **alone or as team of max. 2 persons**.

Please submit your solution **as one ZIP file** before the given deadline at https://www.dcl.hpi.uni-potsdam.de/parProg/. Please note that the submission system demands a certificate-based authentication. Incorrect submissions (missing source code, missing inline documentation, missing binaries, non-functional binaries) are rejected, with one possibility for re-submission.

Assignment 1

The first assignment covers the usage of basic synchronization primitives in a thread-based shared memory system. You have to solve the given programming exercises in C / C++ either based on the Windows Thread API[1] or the POSIX pthread API[1]. Additional threading libraries are not allowed for this assignment.

Task 1.1

Implement a program that sums up a range of numbers in parallel. The general algorithmic problem is called „parallel reduction“ or „prefix scan“. An appropriate parallelization strategy can be found in the Internet or in literature, if necessary.

Your application should accept three parameters: The number of threads to use, the start index and the end index (64bit numbers) of the range to compute. For example, the command line

```
./parsum 30 1 10000000000
```

should lead to a parallel summation of the numbers 1,2,...,10000000000, based on 30 threads running in parallel. Your program should **only** output the result and terminate. The solution is correct if a true parallelized computation takes place (no Gauss please), the solution scales based on the number of threads, and if the application produces correct results for all input. We will evaluate your solution with different thread counts / summation ranges. The result with lowest average runtime will be shown in the lecture.

Task 1.2

Implement the dining philosophers problem with a freely chosen deadlock-free solution strategy[2]. Your application should accept two parameters, the number of philosophers resp. forks (min. 3) and the maximum run time in seconds. Each philosopher should be represented by a thread. You are free to map also other stake holders (e.g. waiters) or resources (e.g. forks) to threads if necessary.

After the given run time is exceeded, the program must terminate with exit code 0 and output the number of „feedings“ per philosopher.

---

[1] `man pthread`

[2] http://en.wikipedia.org/wiki/Dining_philosophers_problem

Task 1.3

Implement an agent communication ring based on the CSP channel concept.

The first step is to realize helper functions for CSP-like channel functionality, e.g. with the following interface:

```
// creates a CSP channel, returns a handle
int createChannel(int maxMsgSize)
// sends data to the given channel with rendezvous behavior
void sendTo(int chan, char* data, int msgSize)
// receives data from the given channel with rendezvous behavior
void recvFrom(int chan, char** data, int* msgSize)
```

The helper functions must be based on shared memory interaction and must implement rendezvous behavior.

In the second step, create an executable application based on your channel helper functions. The application should accept the number of concurrent threads and the number of ‚rounds' as command line argument. The threads should form a one-way communication ring, where each thread receives a value from a channel to his „left neighbor", and sends this value through another channel to his „right neighbor". One dedicated master thread first sends a random value, and then performs the receive operation. The random value has to travel the given number of rounds through the communication ring until the application terminates. Every time the value passes the master thread, it should print out an according information.