Universität Heidelberg
Institut für Technische Informatik

Prof. Dr. Robert Strzodka
Steffen Lammel

Advanced Parallel Computing
Term 2015 (Summer)

Exercise 3

- **Return electronically (MOODLE) until Tuesday, 12.05.2015 14:00**
- **Include name on the top sheet. Stitch several sheets together.**
- **A maximum of two students is allowed to work jointly on the exercises.**

3.1 Reading

Read the following two papers and provide reviews as explained in the first lecture (see slides):

- David Wood and Mark Hill, Cost-Effective Parallel Computing, IEEE Computer, 1995.
- P. Stenstrom, A Survey of Cache Coherence Schemes for Multiprocessor, IEEE Computer, 1990.

(25 points)

3.2 Shared Counter

Implement a Pthread program which performs the following calculation: a counter (initialized to zero) has to be incremented **C** times, so that at the end of the program has the value **C**. A configurable number of threads (**N** threads in total) shall cooperatively increment the counter. Use a mutex to ensure mutual exclusion when incrementing the counter. Before the threads start incrementing the counter, all threads have to synchronize using a barrier. This avoids the visibility of start-up effects and maximizes contention. **N** and **C** should be parameters of your program.

In pseudo-code, each thread should do the following:

```
Barrier();
for ( i = 0 to (C/N)-1 ) {
     Mutex_Lock ();
     inc (counter);
     Mutex_Unlock();
}
```

Develop a suitable program, first without the mutex. Depending on a sufficient large C, data races should result in a mismatch between counter and **C** at the end of the program. Reproduce this behavior. Provide an interpretation why this is dependent on the value of **C**.

Now, implement the program using a mutex to protect the critical section. For a varying **C**, ensure that after execution the counter matches **C**, i.e. there are no race conditions anymore.

Conduct your experiments on one of the creek nodes.

(10 points)

3.3 Shared counter revisited

Start with the program from exercise 2.3. Implement two alternative counter update methods and answer the questions:

1. Use an atomic operation to increment the counter. Obviously, now no locking is required anymore (why?). For gcc, the following intrinsic can be used:
   `__sync_add_and_fetch(&var,1);`
   In this example, variable `var` is atomically incremented. Feel free to use any other atomic intrinsic.

2. Now, use the atomic operation (or another suitable one) to implement your own locking mechanism. I.e., implement a `lock_rmw(*lock)` and an `unlock_rmw(*lock)` function, by relying on atomic operations. Why are atomic operations required in this case?

Develop your programs and perform initial testing on one of the **lsra** nodes. Validate the correctness of these two new programs with the same methodology as in exercise 2.3. I.e., for a varying **C** and **N**, ensure that after execution the counter matches **C**, i.e. there are no race conditions anymore.

(25 points)

Prof. Dr. Robert Strzodka
Steffen Lammel

3.4 Shared counter performance analysis

Now, measure the overall execution time using suitable functions (for instance *clock_gettime* or *gettimeofday*). Report the overall execution time and the derived number of updates per second for a varying number of threads (1-48) and sufficiently large number of updates (providing stable results). For this experiment, use the computer **moore** (48 cores in total). As **moore** is often heavily used, please ensure that you only use it for performance experiments.

	PTHREAD MUTEX		**ATOMIC INCREMENT**		**LOCK_RMW**	
Thread count	**Execution time**	**Updates per second**	**Execution time**	**Updates per second**	**Execution time**	**Updates per second**
1						
2						
4						
8						
12						
16						
24						
32						
40						
48						

Run this experiment with all three implementations. Report results in the table above, include appropriate units (ns, us, ms, s). Include a graphical representation here (varying number of threads on x-axis with updates per second on y-axis). Interpret your results!

(15 points)

Total: 75 points

General notes:

- Measure the overall execution time using suitable functions (for instance *clock_gettime* or *gettimeofday*).
- For experiments, please use the computer **moore** (48 cores in total). As **moore** often is heavily used, please ensure that you only use it for experiments.