Department of Computer Science, ETH Zürich

Supporting Parallelism in Operating Systems & Programming Languages

Fall 2013

Exercise 1: Warm-ups

1 Parallel generation of pseudo-random numbers

Write a multi-threaded program where each thread generates a predetermined number of pseudo-random numbers N. The number of threads created T and N should be configurable at run-time (e.g., by using command-line arguments). The default value for T should number of CPUs as seen by the OS.[1] The program should print execution time for each thread, and the total execution time for all threads.

You will need to:

- Create threads using the `pthread` library
- Programatically find the number of CPUs available in the system
- Measure execution time
- Be careful of compiler optimizations

For bonus points:

- Create your own functions for measuring CPU cycles using the `rdtsc x86` instruction.

For your report:

- **Q1.1:** Can you determine the performance of a parallel program using only the execution time of each of its threads in the general case?
- Execute your program on a machine with multiple cores and construct graphs about your program's performance. Discuss the results.

2 False sharing

The purpose of this exercise is to measure the effect of false sharing. To do that, create a program that spawns a number of threads, where each thread performs an addition to a memory location a number of times (e.g., 10^9). Allow the user to set the following parameters: (i) the number of threads, (ii) the placement of the

[1] In fact, in this document we will use the term CPU for an execution unit perceived by the OS as a CPU. An SMT thread of execution, for example, falls into this category.

threads on the CPUs and (iii) whether the threads will operate on memory locations residing on the same cache-line or not.[2]

You will need to:

- Affine threads to specific CPUs
- Be careful of compiler optimizations
- Verify that the addition results are valid.

For bonus points:

- Find the cache-line size of your machine.

For your report:

- Get results from executing your program on different machines (e.g., you can use your laptop and the `ikq`, `bach03/04` machines) and present them using one or more graphs. Discuss the results.

3 NUMA allocation

The purpose of this exercise is the measure the performance difference of accessing memory on a local NUMA node against accessing memory on a remote NUMA node. To that effect, write a program that allocates memory on numa node N and measures the latency of reading this memory from CPU c, where N and c are parameters specified as command line arguments.

You will need to:

- Affine threads to specific CPUs
- Allocate memory from specific memory nodes
- Discover the NUMA topology of the machine
- Be careful of compiler optimizations
- Consider cache effects

For bonus points:

- Write a program that measures memory bandwidth instead of latency.
- Write a program that tries to discover the NUMA topology of a machine using only measurements of memory operations performance. Try to minimize the number of measurements required to make a decision.

For your report:

- Execute your program on a bach machine (`bach03/04`). Present and discuss your results.

[2] You can assume a cache-line size of 64-bytes.

4 Dates & Deliverables

07.10.2013: Discussion/questions
11.10.2013: Final version of the code & final report