Parallel Processing and Software Performance

Lukáš Marek
DISTRIBUTED SYSTEMS RESEARCH GROUP
http://dsg.mff.cuni.cz

CHARLES UNIVERSITY PRAGUE
Faculty of Mathematics and Physics
Introduction

- Benchmarking in parallel environment
- Shared hardware resources
  - CPU
    - Cache coherency, cache bandwidth, cache sharing
  - Main memory bandwidth
  - (HDD)
  - Many topics still left to explore
    - Hyper-threading, NUMA, HDD (SSD), networking, ...

Motivation

• Explore scenarios where parallel execution influences performance
  ▪ Cover all possible cases
  ▪ Design various benchmarks with possibility to run them in parallel with a different configuration
  ▪ Multi-platform (hardware) as much as possible
How we did it - framework

• Framework – RIP
  ▪ C++, R
  ▪ Parallel execution of workloads (benchmarks)
  ▪ Designing new benchmarks as simple as possible
    • example
  ▪ Automated benchmarking with various parameters
  ▪ Simple composition of benchmarks
• Automated graph creation
• Artificial workloads
  ▪ Designed to create highest (worst-case) overhead
  ▪ Effect isolation

• One observing workload (benchmark)
  ▪ Does all the measurement

• One or more load generating (interfering) workloads (benchmarks)
  ▪ Can also measure to confirm results
How we did it – experiments

- Data collected
  - 10 runs
  - RDTSC
    - 10 samples
    - Average over 100 cycles – because of overhead
  - Performance counters
    - 3 samples
    - Average over 1000 cycles – because of overhead
• Intel Core 2 Quad – dual CPU
  ▪ Differences between CPU architectures
    • Core i7 – CPU interconnect, integrated memory controller, NUMA, hyper-threading, ...
  ▪ Differences between vendors
• 8 GB unified access main memory
  ▪ AMD uses NUMA architecture
Experiments

- CPU
  - Cache coherency
    - Ping-pong
    - Producer – consumer
  - Cache sharing
    - Bandwidth
    - Thrashing – capacity competition
- Memory
  - Bandwidth
    - Single-pointer
    - Multi-pointer
Produce – consumer

- Cache coherency
- Transport data from one core to another
  - Write data on first core
  - Read data on second core
- Ex: producer – consumer
Producer – consumer (clean data)

- cores in a same package with the shared L2 cache
- cores in a same package without the shared L2 cache
- cores in a different package

Duration of reading one cache line [cycles – transferred mem Avg]

Transferred memory size [bytes]
Producer – consumer (modified data)

- Cores in a same package with the shared L2 cache
- Cores in a same package without the shared L2 cache
- Cores in a different package

Duration of reading one cache line [cycles – transferred mem Avg]

Transferred memory size [bytes]
Producer – consumer memory transfers
If we transfer data from one core to another:

- data transfer is relatively cheap for cores that share the L2 cache
- data transfer to the core that does not share with the source the L2 cache is 5 times slower than to the core which does
- modified data can be transported in some cases more than two times faster than clean data
Cache bandwidth

• Shared cache
• Competition for bandwidth to the shared cache
• Two threads accessing memory
  ▪ Interfering workload inserts NOP instructions between accesses to simulate intensity
  ▪ Interfering workload hits or misses the cache
  ▪ Linear reads
• Ex: two running processes (threads) access memory
Cache bandwidth – interfering cache hit
Cache bandwidth – interf. cache miss

Duration of one access [cycles – 100 Avg]

Number of NOP instructions (thrasher)
When two cores compete for the bandwidth to the shared L2 cache:
  - if both cores are hitting the L2 cache the overhead is only about 12%
  - if one of the two cores starts missing the L2 cache, the access time for core that is still hitting the L2 cache can be twice as high than in the case of the single-core access

Multi-pointers – maybe unrealistic stressing
Cache thrashing

• Shared cache
• Competition for space in the shared cache
• Two threads accessing memory
  ▪ Observing workload has fixed size memory
  ▪ Interfering workload increases memory and uses NOPs for access intensity regulations
  ▪ Interfering workload modifies cache lines to distinguish whose cache line was evicted
• Ex: two running processes (threads) access memory
Cache thrashing – thrasher 4MB

L2 modified cache lines evicted per one access [count – 1000 Avg]

Number of NOP instructions (thrasher)

- ▲ - 128 KiB memspeed memory size
- ▼ - 256 KiB memspeed memory size
- ◇ - 512 KiB memspeed memory size
- ◇ - 1 MiB memspeed memory size
- + - 2 MiB memspeed memory size
- ◇ - 4 MiB memspeed memory size
- ◇ - 8 MiB memspeed memory size
When two cores compete for the space in the shared L2 cache:

- a competition for the cache space starts when a sum of the allocated memory blocks is over 2 MiB
- a core with smaller size of data could be also evicted if it does not access the data fast enough
Eviction behavior in physically mapped cache

virtual → physical → physically mapped cache
Eviction behavior in physically mapped cache II
Memory bandwidth

• Sharing a memory bus and a memory controller
  ▪ Memory blocks with the same size accessed from two cores
Memory bandwidth – SP read linear

- single core variant
- cores in a same package with the shared L2 cache
- cores in a same package without the shared L2 cache
- cores in a different package

Duration of one access [cycles – 100 Avg]

Accessed memory block [bytes – size]
Memory bandwidth – SP write linear

- single core variant
- cores in a same package with the shared L2 cache
- cores in a same package without the shared L2 cache
- cores in a different package

Duration of one access [cycles – 100 Avg]

Accessed memory block [bytes – size]
Memory bandwidth – SP read random

- Single core variant
- Cores in a same package with the shared L2 cache
- Cores in a same package without the shared L2 cache
- Cores in a different package

Duration of one access [cycles – 100 Avg]

Accessed memory block [bytes – size]
Memory bandwidth – page walks

- single core variant
- cores in a same package with the shared L2 cache
- cores in a same package without the shared L2 cache
- cores in a different package

Accessed memory block [bytes – size] vs Duration of page-walks per one access [clocks – 1000 Avg]
When two cores access the memory:

- with more shared components (like the memory bus or the shared L2 cache) an access slows down
- when using the linear access, the overhead can be almost 80% in comparison with the single-core access
- the reading linear access is almost two times faster than the writing linear access
- the linear access can be more than two times faster than the random access
Memory bandwidth – MP read linear

- one pointer
- two pointers
- four pointers
- eight pointers

Duration of one access [cycles – 100 Avg]

Accessed memory block [bytes – size]

Lukáš Marek
Memory bandwidth – MP read random

Duration of one access [cycles – 100 Avg]

Accessed memory block [bytes – size]
Memory bandwidth – MP diff cores

- single core variant
- cores in a same package with the shared L2 cache
- cores in a same package without the shared L2 cache
- cores in a different package

Duration of one access [cycles – 100 Avg]

Accessed memory block [bytes – size]
Memory bandwidth – MP L2 prefetch

- single core variant
- cores in a same package with the shared L2 cache
- cores in a same package without the shared L2 cache
- cores in a different package
Memory bandwidth – MP summary

- When two cores access the memory with high intensity using multiple pointers:
  - multiple pointers in random access can improve the throughput more than 300%
  - an intensive access into the shared L2 cache can disable prefetching and improve the throughput
Other experiments

• Memory saturation
  ▪ Small number of tested configurations
  ▪ Two cores 90% throughput as four cores

• HDD
  ▪ Too many configurations – HDD 7,2K, HDD 15K, SSD, RAID, SATA, PATA, SCSI, ...
  ▪ Too many layers – OS, File system, user libraries, ...
  ▪ Some measures done with little (no) success

• AMD – NUMA, ...
• Cache coherency
• Read or write to the shared variable
  ▪ Write is done using “lock inc”
• Ex: synchronization primitives

Ping-pong

Core 1  Core 2

L1  L1
Ping-pong – read x write

- • cores in a same package with the shared L2 cache
- ■ cores in a same package without the shared L2 cache
- • cores in a different package

Duration of access [cycles – 100 Avg]

Variable is shared [0 – false, 1 – true]
Ping-pong – write x read

- • cores in a same package with the shared L2 cache
- ■ cores in a same package without the shared L2 cache
- • cores in a different package

Duration of access [cycles – 100 Avg]

Variable is shared [0 – false, 1 – true]
• When we have a shared variable which is often modified:
  ▪ the average read access can be 20 times higher than in the case of non-shared variable
  ▪ the write access can be on average 5 times slower than the non-shared access
  ▪ cores further apart from each other can do more accesses in a row before the other core requests the memory again
  ▪ Partially also valid for false sharing (lock instruction)
Questions?