Engenharia de Sistemas de Computação
Performance Optimisation

Class 2 – Profiling

Vítor Oliveira (vitor.s.p.oliveira@gmail.com)

13 Abril 2021

1 Conhecer a infrastrutura

Para medir o desempenho, tal como muitas outras tarefas na engenharia do desempenho, dependem da infraestrutura em particular onde a aplicação irá ser executada.

Conhecer a infraestrutura

Servidores típicos

- Na medição do desempenho esse conhecimento permite conhecer os limites da infraestrutura que fazem sentido de forma criar expectativas corretas para a execução.
- As máquinas servidoras mais utilizadas atualmente (SMP) possuem dois encaixes de processadores, com um número alargados de núcleos de computação por cada um dos processadores. Algumas máquinas possuem um número maior de encaixes, mas são menos habituais porque têm um custo por processador bastante superior.
- A esses dois sockets estão ligados os módulos de memórias e os dispositivos externos através do barramento PCI, como as placas controladoras de rede, placas controladores de gráficos e os controladores de armazenamento, aos quais estão ligados os discos (HDD e SSD) ou possuem eles próprios armazenamento (NVME).

Conhecer a infraestrutura

Servidores típicos a curto/médio prazo

- A tendência nos servidores é de aumentar o número de núcleos por socket significativamente;
- Partindo dos típicos 8 a 32 núcleos por processador, temos já processadores comuns de 128 núcleos por processador (AMD);
- Para interligar estes núcleos é necessário construir barramentos de comunicação internos complexos, e muitas vezes heterogéneos, tal como cada núcleo em si mesmo pode ter características de desempenho um pouco diferentes dos seus gémeos;
- Estes núcleos competem no acesso à memória, aos periféricos e também à corrente elétrica, o que torna o comportamento final difícil de prever e ligeiramente diferente de processador para processador (mesmo sendo aparentemente iguais);
- Na competição pelo corrente elétrica, os fabricantes têm vindo a ser cada vez mais agressivos com estratégias que visam manter o TDP dentro dos limites do CPU, o que introduz variações muito rápidas e habituais da frequência de cada núcleo.

Conhecer a infraestrutura

Servidores típicos a curto/médio prazo

- A utilização de co-processadores como GPUs, DSPs e FPGA é cada vez mais comum, o eu introduz espaços de endereçamento diferentes na mesma máquina, tal como já acontece na computação paralela entre máquinas;
- Os dispositivos de memória persistente estão também a alterar significativamente o modo como as aplicações que têm estado exploram o hardware:
  - usando-os como memória RAM económica de grande capacidade e/ou,
  - usando-os como dispositivos de armazenamento de extremamente baixa latência;
- As generalização das redes de alto débito e RDMA tem vindo a permitir comunicar entre nós diferentes a velocidades mais altas e sem envolvimento e comutação de contexto pelo OS.
- ...

Conhecer a infraestrutura

A utilização de memória cache é crítica

- A memória cache é essencial para os CPUs terem um bom desempenho porque o acesso a memória fora da cache incorre numa grande penalização de desempenho;
- Tipicamente cada linha de cache (a unidade mínima que o processador lê e escreve na memória) é de 64 bytes (há alguns casos de 128 bytes) o que pode dar origem a false-sharing entre processadores/núcleos;
- Mover processos entre processadores invalida parte das caches pelo que o SO tende a manter os processos a correr nos mesmo cores, a menos que outros fiquem livres durante algum tempo;

Conhecer a infraestrutura

A memória cache é muito limitada

- Se a aplicação tiver acesso a memória dispersa estas caches tornam-se menos eficientes porque há pouco reuso da cache;
- Da mesma forma, se o código da aplicação for muito grande a cache de instruções irá também ser menos eficiente;
- Isto tem implicações nas estratégias dos compiladores, que se usarem “inline” mais agressivamente vão colocar em causa a a reutilização da cache de instruções;
- No entanto, o compilador não pode prever a carga real que a aplicação irá suportar, pelo que as sua heurísticas poderão falhar mais frequentemente para alguns casos;
- Para aumentar a utilização da cache de instruções as aplicações podem ser compiladas em dois passos com ”profile-guided optimizations” e a aplicação de carga real entre as fases para instruir o compilador.

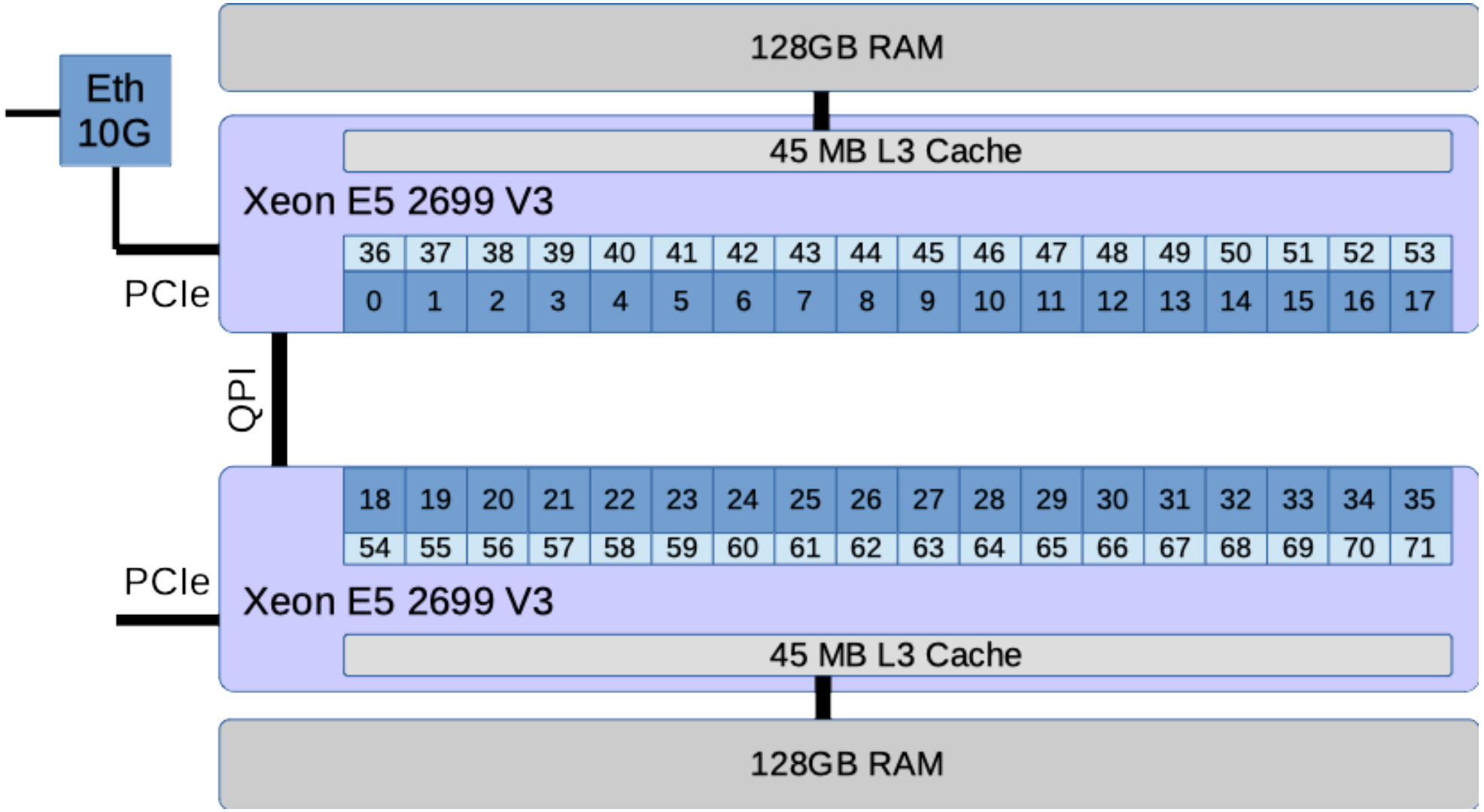
Conhecer a infraestrutura

Nota: os servidores não “são como” os portáteis e vice-versa...

- Os portáteis não são representativos do desempenho que pode ser obtido nos servidores típicos:
  - Os portáteis estão normalmente otimizados para o baixo consumo enquanto os servidores estão otimizados para alto desempenho;
  - A memória máxima de um servidor moderno ultrapassa 1TB;
  - Os servidores possuem tipicamente na ordem das dezenas por processador;
  - Etc.
- Não se pode sentir tentado a partir do principio que os dados obtidos num sistema pode ser extrapolado facilmente para outro, porque tal é enganador.

Conhecer a infraestrutura

Exemplo 1



2 Conhecer a infrastrutura: hands-on

Utilizar diversos fontes de informação para investigar as características do sistema de computação.

Conhecer a infraestrutura: hands-on

Informação sobre a máquinas em concreto

- numactl --hardware
- lscpu (cat /proc/cpuinfo)
- lspci
- fdisk –l (root)
- ifconfig –a
- lsof
- top (then type ‘1’ to see the cpus)
- i7z (old, https://github.com/ajaiantilal/i7z)

Conhecer a infraestrutura: hands-on

Informação sobre a máquinas em concreto

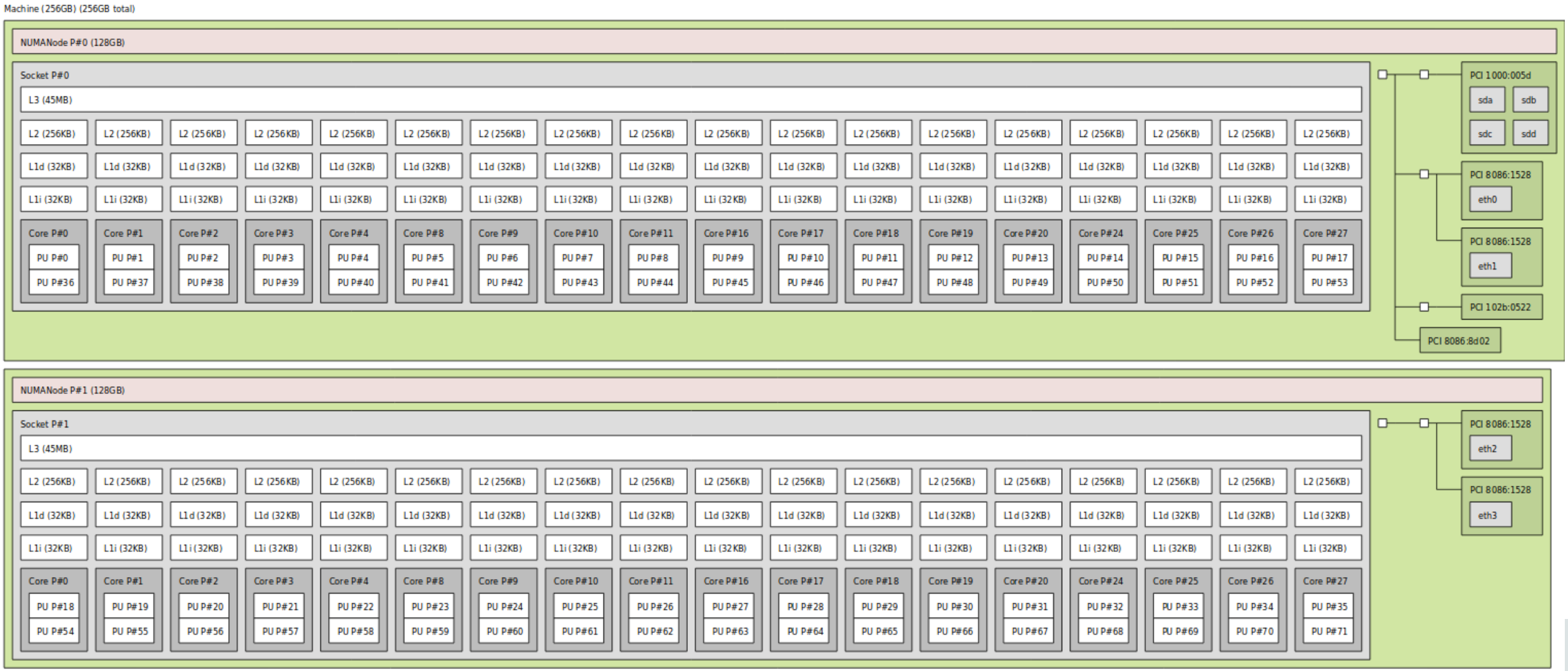
```
$ numactl --hardware


available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53
node 0 size: 128805 MB
node 0 free: 10766 MB
node 1 cpus: 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69
70 71
node 1 size: 129018 MB
node 1 free: 11306 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10

$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 63
model name      : Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
stepping        : 2
microcode       : 0x38
cpu MHz         : 1200.042
cache size      : 46080 KB
physical id     : 0
siblings        : 36
core id         : 0
cpu cores       : 18
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 15
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr
sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology
nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 fma cx16 xtpr pdcm pcid
dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm ida arat epb pln
pts dtherm tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid cqm xsaveopt
cqm_llc cqm_occup_llc
bugs            :
bogomips        : 4589.51
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
```

Conhecer a infraestrutura: hands-on

hwloc (https://www.open-mpi.org/projects/hwloc/)



Conhecer a infraestrutura: hands-on

Benchmarks the capacidades

- cpu (sysbench)
- network (iperf3)
- storage (fio, iozone)
- ...

3 Demo applications

Different applications explore the computing infrastructure in different ways, we will focus on two small demos to explore the hardware in different ways.

Demo applications

- A histogram which increments one or more integers from one or more threads;
- A message passer between one or more pairs of threads

Extract the shared training.tgz and compile the applications in:
https://drive.google.com/file/d/1-SaqYN525xt7f7Djp3hThvs7Z5lY8kop/view?usp=sharing

```
$ tar zxvf training.tgz
$ cd training
$ gcc -o histogram histogram.c -pthread -lrt -DNUMA -lnuma -lm -lrt –openmp
$ gcc -o messages messages.c -pthread -lrt -DNUMA -lnuma -lm -lrt --openmp
```

Demo applications

Histogram

- Goal: have a set of threads increment one or more counters concurrently, with and without guards to assure correctness, and check the gain from running in parallel in several different scenarios

- Command line:
  - $ histogram --[atomic,mutex,nobarrier,omp-atomic,omp-critical] --threads=<n> --size=<n> --points=<n> --cpu=<c>

- Each thread can increment one or more points individually, or share the counter with other threads, according to the size parameter

Demo applications

Histogram

- Test using –atomic,--mutex and –nobarrier
- Test using 1, 2, 4, 8, 16 threads
- Test using size=1 and size = 1000
- Test using cores [0,1] [0,36] [0,71] [0,0]

- Example
- $ ./histogram --atomic --threads=2 --size=1 --points=1 --cpu=0 –cpu=36
- synctest: running 125000000 iteractions with 2 threads, using atomic locking on 2 cpus
- waiting for 2 threads: ..
- Test ended; verification = 250000000 ok, 2.10s

Demo applications

Histogram

- experiments
  - $ ./histogram --nobarrier -t 1 --size=1 -c0 (3.35)
  - $ ./histogram --nobarrier -t 2 --size=1 -c0 -c3 (2.44)
  - $ ./histogram --mutex -t 2 --size=1 -c0 -c3 (42.67)
  - $ ./histogram --atomic -t 2 --size=1 -c0 -c3 (7.72)
  - $ ./histogram --atomic -t 2 --size=1 (7.80)
  - $ ./histogram --atomic -t 4 --size=1 -c0 -c1 -c2 -c3 (7.66)
  - $ ./histogram --atomic -t 4 --size=40 -c0 -c1 -c2 -c3 (1.39)
- in a second terminal window, execute
  - $ top (then ‘1’ in the UI)
  - $ sudo perf top

Demo applications

Message passing

- Goal: have a pair of threads exchange a message (synchronization point) and check the gain from running in parallel in several different scenarios

- Command line:
  - $ messages –[test1,test2,test3,test4,test5] threads=<n> --cpu=<c>
  - Where
  - Test1 = mutex_lock + cond_wait/signal
  - Test2 = mutex_lock busy wait
  - Test3 = __sync_bool_compare_and_swap + yield
  - Test4 = __sync_bool_compare_and_swap + no yield
  - Test5 = mutex_lock busy wait + yield

Demo applications

Message passing

- Test using --test1, --test2, --test3, --test4, --test5
- Test using 1, 2, 4, 8, 16 threads
- Test using cores [0,1] [0,3] [0,0]
- Examples
  - $ ./messages -1 -c0 -c1
  - synctest: running 500000000 iteractions with 2 threads, using test1 on 2 cpus
  - waiting for 2 threads: Iter [1/15] took 1833465 ns, 1833465 cycles
  - Iter [2/15] took 1822293 ns, 1822293 cycles
  - …
  - Iter [15/15] took 1864726 ns, 1864726 cycles
  - Average time 1.842978 us, median time 1.837423 us

Demo applications

Message passing

- $ ./messages -1 -c0 -c1 (1.83, mutex+cond)
- $ ./messages -1 -c0 -c2 (2.59)
- $ ./messages -2 -c0 -c1 (0.46, mutex busy)
- $ ./messages -3 -c0 -c1 (0.14, CAS+yield)
- $ ./messages -4 -c0 -c1 (0.04, CAS+busy)
- $ ./messages -5 -c0 -c1 (0.18, mutex + yield)

Demo applications

Some take-aways

- Synchronization primitives
  - Synchronization, even when uncontended, takes a significant amount of time
  - But atomic operations provide a much smaller overhead then mutexes
  - That becomes more significant as the number of threads grows
- Locality
  - The CPU core where the threads competing for synchronization are running is very important for the throughput
  - That is mainly related to the cache access, but also to other factors
- Scheduling in and out
  - Having to re-schedule a task takes time to context and trashes the caches
  - Avoid synchronization whenever possible to avoid contention and wasted CPU time

4 Profiling

Profiling consists on analyzing the behavior of the systems and applications providing views of the way resources are being used.

Profiling

- In order to evaluate the behavior of an application it is useful to use profilers that quickly extract some information about the system and application behavior;
- There are many ways to profile, but care must be taken not to be deceived by profilers which by highlighting details that are irrelevant can hide the most relevant behavior.
- It usually counts the time in the CPU, but can include other kinds of information
- The call-stacks are extracted to assign time to the function and to its childs whenever a sample includes it

Profiling

- Profiling can be very intrusive and slow-down the application significantly (yes, I’m talking about you callgrind)
- Sampling can reduce the overhead at the cost of lower resolution (perf, oprofile, gperftools)
- To identify periods where the application is stopped waiting one needs to profile off-cpu time (gperftools, eBPF)

Profiling

CPU utilization

- The kind of profiling most commonly considered is the CPU utilization profiling.
- With a sampling profiler one can track the functions that took the longest to execute by checking how many samples contains that functions in the top of the call stack. The parent/child execution time is taken as those functions are on the call stack of any sample.
- While there are profilers that support wall-clock time profiling, that is not the common, the samples are taken at regular intervals only while they are running on the CPU, so they don't account for time spent being scheduled out or waiting on storage or network.
- Most importantly, they don't consider the time waiting on synchronization primitives, so they won't help when we have a slow system due to ineffective coordination or even contention on the locks.
- Nevertheless, they are good at showing how the CPU is used, so for functions that execute dense kernels, they are a good tool to identify those kernels and help optimize them.

Profiling

Evaluating CPU utilization

- For in depth analysis of the CPU profile the best tool is usually a call stack graph, which connects nodes using arcs that show the CPU utilization of each function and the functions that called.
- These charts are usually very detailed and take some time to interpret correctly.
- FlameGraphs have lately been used to allow a simpler first look at the profile. They allow a stacked view of functions with their width corresponding to their execution time compared to que total execution.
- The impact of functions that are called from multiple places get hidden in the FlameGraphs.
- The next flamegraphs show the profile of the full mysqld process executing a Sysbench Update Non-Index with 32 threads, focusing on some of the main functions related to the binlog.

Profiling

But there are many different kinds of profiles:

- CPU utilization
- Function profiling
- Lock profiling
- System calls
- Storage
- Network usage
- Memory accesses
- File-systems
- Off-cpu
- ...

Profiling

- All events
  - Code changes
  - Dynamic instrumentation
  - Hardware counters
- Sampling
  - Sampling metric (cpu usage, realtime)
  - Frequency
- Context:
  - Whole process
  - Single-thread
  - Specific resources

Profiling

Profiling problems:

- Callstack problems (frame pointer and unwind)
- Inlined code
- Cpu profiling based on cpu execution time or SIGALRM
- Interrupted system calls