LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

lrz

Technische Universität München

Praktikum im WS 2023/2024
Evaluierung moderner HPC-Architekturen und -Beschleuniger (LMU)
Evaluation of Modern Architectures and Accelerators (TUM)

Sergej Breiter, MSc., Minh Thanh Chung, MSc., Amir Raoofy, MSc.,
Dr. Karl Fürlinger, Dr. Josef Weidendorfer

Assignment 07 – Due: 7.12.2023, 16:00

Latencies to Memory and between Cores

In this assignment, we look at various properties of the memory hierarchy and connectivity of CPU cores regarding latencies involved.

1. **Linked List Traversal**

   Measurements in this task only use one thread - no parallelization is requested.

   List data structures are essential for a lot of algorithms. To allow for fast insertion and removal of elements of the list, often, so-called linked lists are used. After lots of insertions and removals, a traversal of a linked list results in almost random accesses to memory. We emulate this access pattern by traversing the elements of an array with N elements in a pseudo-random order. We emulate the pseudo-random accesses by accessing the $(k \cdot i \bmod N)$-th element of A for the i-th access, with the ratio of k and N being close to the golden ratio ($\frac{1+\sqrt{5}}{2}$). We ensure that all elements of A are visited by asserting that k and N are coprime integers. Also we choose a power-of-2 for N as this allows a fast version of the modulo operator by masking out the lower $log_2(N)$ bits of N through an AND operation.

   We compare two code variants of exactly the same traversal, with the same memory accesses being done, and same computations done. The only difference is that the first version calculates the indexes during traversal from i, while the second one gets the index of the next element to access from the previously accessed element. Elements store a next-index value and another double value to be summed up. As we want exactly the same accesses in both traversals, we need to use the accessed index in both cases by some dummy calculation. To avoid that the compiler optimizes out this calculation, the final result is stored to a location outside of the traversal function. The initialization is shown in in Fig. 1.

   The source snippet in Fig. 2 provides the first version of our emulated traversal, with the indexes to be accessed being calculated. The source snippet in Fig. 3 shows the second variant

```
// basic elements of the linked list
struct entry { double v; int next; };

void init(int N, int k, struct entry* A)
{
  int mask = N-1; // N is power-of-2
  for( int i=0; i<N; ++i ) {
    A[i].v    = (double) i;
    A[i].next = (k*(i+1)) & mask;
  }
}
```

Figure 1: Initialization of list (task 1).

```
double sum_indexcalc(int N, int k, int REP, struct entry* A, double *psum,
    int* pdummy)
{
  auto t0 = std::chrono::high_resolution_clock::now();
  int mask = N-1; // N is power-of-2
  int dummy = 0;
  double sum = 0.0;
  for( int r=0; r<REP; ++r ) {
    int next = 0;
    for( int i=0; i<N; ++i ) {
      sum   += A[next].v;
      dummy |= A[next].next;
      next   = (k*(i+1)) & mask;
  }
  auto t1 = std::chrono::high_resolution_clock::now();
  *psum = mysum; *pdummy = dummy;

  // return average time per element access in nanoseconds
}
```

Figure 2: Code snippet with index calculation during traversal.

with the index of the element to be accessed next being loaded from the previously accessed element.

For the k used to approximate the golden ratio, make sure that k and N actually are co-prime. You might want to implement an assertion scheme for verifying that all elements are accessed in both of your functions. (e.g., by first setting $v = 0$ for all entries, then set $v = 1$ in the traversal, and check that v of all entries are as expected).

(a) Implement the full code to run traversals of `sum_indexcalc()` and `sum_indexload()`. Measure the performance of both variants by plotting the average time required for traversing one element of the list, with array size N set to $\{2^{10}, 2^{11}, ...2^{30}\}$ and k on the one hand set to 1 and on the other hand set to the value such that k/N is near the golden ratio. Make sure to bind the thread during the runs to a fixed core. Do the measurement on all systems in BEAST.

(b) Try to explain the results of your experiments. Especially, what is the influence of the size N, setting of k, and the difference of the two code variants `sum_indexcalc()` and `sum_indexload()`?

(c) For random access traversal with a large enough N, the first access into a list element

```
double sum_indexload(int N, int k, int REP, struct entry* A, double *psum,
    int* pdummy)
{
  auto t0 = std::chrono::high_resolution_clock::now();
  int mask = N-1; // N is power-of-2
  int dummy = 0;
  double sum = 0.0;
  for( int r=0; r<REP; ++r ) {
    int next = 0;
    for( int i=0; i<N; ++i ) {
      sum   += A[next].v;
      dummy |= (k*(i+1)) & mask;
      next   = A[next].next;
  }
  auto t1 = std::chrono::high_resolution_clock::now();
  *psum = mysum; *pdummy = dummy;

 // return average time per element access in nanoseconds
}
```

Figure 3: Code snippet using indexes pre-calculated before traversal.

should be a cache miss. With this assumption, what is the utilized memory bandwidth? How large is it for the various systems, for each of the variants?

(d) The memory access latency is the elapsed time between a core triggering a load of a value from main memory and being able to use the loaded value. Assume that in contrast to main memory access latency, the second access into the same element always hits L1 and thus is neglectable. With this assumption, can you derive the memory access latency to local memory (i.e. within a NUMA domain) of the various systems from the results of the previous sub-task? Which value of N, k, and which variant can be used to derive the memory access latency?
Hint: the chosen setting must ensure that there is no prefetching happening and no memory parallelism possible (i.e. only one outstanding memory access at each point in time). Why is this important?

(e) Can you derive the access latency to other levels in the memory hierarchy (data caches) from the measurements of sub-task (a)? Summarize you results in a table as Table 1 shows. Mark with an "X" if you think this is not possible, and explain why.
Hint: you can use the "lscpu" command to obtain relevant information about the caches. Think about what is a reasonable value of N on each case.

	L1	L2	L3
milan	10ns / N_{L1}	20ns / N_{L2}	30ns / N_{L3}
ice	...	...	
cs	...		
thx			

Table 1: Example table for access latencies of caches.

(f) Compare the measured latencies on the A64FX system to the vendor-provided values. You can find them in the A64FX microarchitectural manual [1] (pages 64, 65, 68; load-to-use). Hint: the A64FX cores run at 1.8GHz.

[1] https://github.com/fujitsu/A64FX/blob/master/doc/A64FX_Microarchitecture_Manual_en_1.6.pdf

(g) Look up the manual page of "numactl". Use this tool to display the NUMA node distances on each system. What is the semantic of the reported values for the NUMA node distances? Which of the values corresponds to the measurement from sub-task (d)?
Hint: read this section [2] about the System Locality Information Table (SLIT).

(h) The tool `numactl` allows you to explicitly incorporate the memory allocation in a separate NUMA domain to enforce sequential traversals in a NUMA domain different from the one the code runs on. For each distinct NUMA distance, perform one measurement of the resulting **main memory** access latency using `numactl` and an appropriate value of N. Document how you performed the measurements and provide a table for the latency data similar to Table 1. Compare the NUMA node distance reported earlier by `numactl` to the values derived from your measurements.

2. **Core-to-core Latencies**

In this task, you will measure the best-case latency involved when cores want to talk to each other. This should reflect the hardware topology, so it is useful when you want to understand this from measurements. Furthermore, it gives insights into latencies to be expected when cores have to synchronize, e.g. within the runtime of OpenMP.

All communication between cores on modern multi-core systems works via shared memory. In the end, this is implemented by transactions of a cache coherence protocol, such as MOESI (AMD) or MESIF (Intel). Look up these protocols for more details. In this task you also will see how latencies change depending on the address used for communication.

Find the benchmark "c2c.c" in the sources for this assignment. It should compile on all systems with GCC or LLVM. It uses OpenMP to run a ping-pong test between given cores via writing values to a shared address and do busy-waiting to observe the address for changes on the other side. To get sensible data, you must bind threads to cores. The number of threads given triggers latency measurements between all pairs of threads.

(a) For each CPU multi-core system in BEAST (Milan, ThunderX2, IceLake, A64FX), find out the numbers given by Linux for the cores. Which OpenMP environment variable settings must be used to bind the range of physical cores to the corresponding number of threads? Give the answer using `OMP_PLACES` and `OMP_PROC_BIND`, and alternatively, `GOMP_CPU_AFFINITY`.

(b) For first tests, the benchmark may take long due to quadratic behavior in the number of cores. So, first run the test just for 2 threads, and set the environment variables to get measurements (1) between nearside cores within a socket, (2) between cores from different sockets, (3) if applicable: between cores on the same socket not sharing the last-level cache (this usually is L3, but L2 on A64FX). Document settings and results.

(c) Try to understand the benchmark. There is an option to get more verbose output, which shows results per memory address used, and also measurements for a "warmup phase". Why is the warmup phase before actual measurements useful? Find out the time interval used for the warmup phase. Is this enough for each of the systems in BEAST, or to high, such that measurements can be sped up?

(d) The benchmark uses multiple addresses in a range, 16 bytes apart, and measures the minimum, maximum, and average latency. Using verbosity mode, observe that these latencies may change depending on address used, also differently on each BEAST architecture. Try

[2] `https://uefi.org/htmlspecs/ACPI_Spec_6_4_html/06_Device_Configuration/Device_Configuration.html#system-locality-information-table`

to explain why this is happening. For this, it is useful to understand how the cache is implemented, over which the communication has to happen, such as an "partitioned L3 cache" for Intel.

(e) Now run the benchmark for all physical cores of the available architectures with default settings, and create as result figure a heat map of minimum latencies as 2D matrix. That is, cell (x,y) should show a color from a fitting range of colors for the minimum latency between cores x and y. Use black for any (x,x) in the matrix, as a core does not communicate with itself, and no measurement can be done for that. Explain how the architecture topology can be derived from this result. Hints: to reduce time needed for the full run, try to use a minimal warmup phase. Also, you can assume the matrix to be symmetrical for cutting runtime by half (modify the code accordingly). See python scripts in sources as ways to produce the 2d matrix heat map.

(f) The benchmark takes a lot of time also due to being sequential. Try to parallelize it to make it faster. How much faster do you get? Make another 2d heat map out of it. Are the results still the same? If not, try to explain what the problem might be for the benchmark to go wrong with your parallelization approach.