Reverse Engineering MareNostrum 5 System Characteristics

Manuel Joey Becklas and Juan José Olivera

1 Introduction

Modern computer organization and architecture provide useful abstractions that hide many hardware details from programmers. In most cases, these abstractions enable software portability and simplify development. However, for domains such as high-performance computing (HPC) or safety-critical systems, a deeper understanding of the underlying architecture can be essential to maximize performance, ensure scalability or diagnose performance bottlenecks.

Microbenchmarking is one of the main techniques used to explore and characterize low-level architectural behaviors. By carefully designing experiments that isolate specific aspects of the hardware, they reveal detailed properties of the memory hierarchy, processor interconnects, bandwidth capabilities, etc. These insights can be highly valuable both for application tuning and system-level performance modeling.

In this project we perform a microbenchmarking study focused mainly on memory aspects of the MareNostrum 5 supercomputing system. Our syudy analyzes several key aspects of memory behavior on a modern multi-socket, multi-core architecture. We begin by describing the system characteristics based on publicly available documentation. Then we present a series of targeted experiments covering:

- **Memory Peak Bandwidth (PBW):** evaluating the maximum attainable data transfer rates.
- **Cache Latencies and Sizes:** measuring memory access latency to estimate cache sizes.
- **Cache Coherence Effects:** investigating the performance impact of shared data access across multiple cores and sockets, exposing effects of the coherence protocols.

The goal of this work is to document and better understand the low-level memory characteristics and behavior of the system using hands-on experiments, while also demonstrating the application of microbenchmarking methodology techniques and principles acquired throughout the project.

2 System Overview

MareNostrum 5 is a supercomputer composed of multiple clusters of computing nodes equipped with high-end processors. The system is organized into several partitions according to workload types. Among these, the two primary partitions are: **General Purpose Partition (GPP)** 6,408 nodes oriented towards traditional CPU-based HPC workloads, and the **Accelerated Partition (ACC)** which has 1,120 nodes targeting GPU-accelerated workloads. In this work, we focus exclusively on CPU microbenchmarking; therefore, all experiments were performed on the **General Purpose Partition (GPP)** nodes.

Each GPP node contains the following aspects relevant for this work:

- Two sockets, each populated with an Intel Xeon Platinum 8480+ processor.
- Each processor has 56 physical cores that operate at base frequence of 2GHz.
- Each socket has 8x 16GB DDR5 DIMM's operating at 4800MHz.
- The total memory per node is 256GB (128GB per socket, 2GB per core).
- Memory is organized under a Non-Uniform Memory Access (NUMA) architecture, where each CPU socket manages its own directly attached-memory.
- Both sockets are connected through Intel Ultra Path Interconnect (UPI) links, supporting inter-socket coherence.

The overall NUMA topology of the node is shown in Figure 1. For futher specs information consult MareNostrum 5 official documentation [[5]].

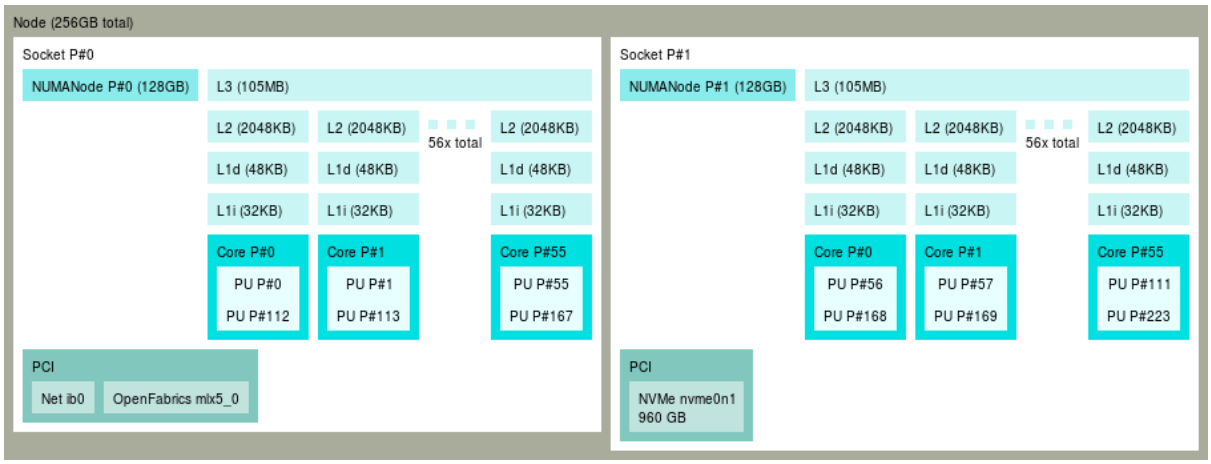

Figure 1: lstopo image of a GPP node in MareNostrum 5.

2.1 High-precision cycle timer

In order to take high-precision cycle measurements, we follow [4]. On Intel architectures, one can take fine-grain time measurements by reading the time stamp counter. But simply using the RDTSC instruction is not sufficient due to out of order execution. Simply inserting a serializing instruction such as CPUID before RDTSC comes with it's problems as well, because the CPUID instruction has non-deterministic latency. Hence, while the above works fine for the *START* time stamp, a better solution is to use the RDTSC**P** instruction *followed* by CPUID for the *STOP* time stamp. This way, the CPUID latency is never measured. Note that there is still an overhead taking the time stamp, but now this overhead is a lot more deterministic.

Because of lack of administrator rights on MareNostrum, we cannot run the benchmarks in kernel mode from within a kernel module as in [4]. The main consequence of this is that we cannot disable preemptions and interrupts.[1] However, as we run our benchmarks on an exclusive node on a HPC system, OS involvement should be minimal, so hopefully this will not have a noticeable impact.[2]

The real problem with the approach is that the CPUs on MareNostrum do run at a variable frequency. Hence, while the fixed frequency time stamp counter can be reliably converted to a

[1] Pinning of processes/threads is possible from user mode.
[2] In case of doubt, running additional trials ought make up for it.

unit of time such as nanoseconds, the obtained measurements cannot be used to determine the number of CPU cycles.
Therefore, in the end, this specific method plays a lesser role in most of our experiments. When only time (as opposed to cycles) is needed and the surpassed interval is large enough, the overhead of any timer because negligible and as such the choice thereof does not matter.

3 Tools and Methodology

To perform the experiments, multiple benchmarking tools and utilities were employed. The development, selection and execution of these tools followed commonly accepted microbenchmarking principles, with the goal of obtaining reliable and meaningful measurements while minimizing noise or external interference. The following practices were systematically applied throughout the study:

- Multiple repetitions of each measurement to reduce variability, avoid cold-start effects and improve statistical significance.
- Exclusive node allocation during experiments to eliminate interference from other jobs or background processes.
- Minimal benchmark code, designed to isolate the behavior of interest and reduce interaction of other processor system components.
- Stressing the relevant subsystems to saturate bandwidth or expose architectural limits.
- Access to full source code for all tools, allowing implementation review to understand, verify and modify as needed for specific experiments.
- Preference for open-sourced or trusted benchmark implementations developed by domain experts.

Throughout the project, we made use of reference implementations developed by experts to ensure correctness and to serve as a foundation for improving our own implementations and deepening our understanding of microbenchmarking techniques. All compiled code was done with *gcc -O3* compiler and base flag.

The following tools and codes were used during the experimental execution:

- **Likwid-bench:** a benchmarking application together with a framework to enable rapid prototyping of multi-threaded assembly kernels
- **Multithreaded Peak BandWidth (mpbw.cpp):** a C++ program written to measure the bandwidth by reading data from a large buffer in memory in a multithreaded fashion to fully saturate the memory channels [[2]].
- **STREAM (stream.c)** an open-source benchmark to measuring memory performance by means of kernels operations of different characteristics (write/read ratio) [[3]].
- **numactl (Linux utility):** used to control NUMA placement policies and modify memory allocation and thread binding.
- **Clamchowder MemoryLatency (MemoryLatency.c)**: Cache latency measuring program by pointer-chaising technique [[1]].
- **Cache shared memory codes**
- **Plotting Utility (plot.py):** simple python program for generating the plots from captured data.

Add c file names and description

4 Experiments

Using the existing infrastructure tools in MareNostrum 5, we compiled the benchmark programs and used the installed CLI tools to run the following experiments.

4.1 Memory Peak Bandwidth (PBW)

Based on the information given in Section 2, we can quickly calculate the theoretical peak memory bandwidth

$$4800 \text{ MT/s} \cdot 64 \text{ bits/T} \cdot 16 \text{ channels} = 600 \text{ GB/s}.$$

However, this theoretical peak assumes that requests to the RAM chip are issued without gaps and does not take other limitations, such as the number of concurrent outstanding misses supported by MSHRs, memory controller limitations, queuing overhead, and so on.

In order to measure practical peak[3] bandwidth, (with high degree of confidence), we use *likwid-bench*. In Figure 2, we can see the bandwidth obtained for different *likwid* benchmarks. `Copy` copies one array into another, `load` loads each array element into registers, `store` initializes an array, and `stream` stores the element-wise scaled sum of two arrays in a third array (two loads, one store). The `mem` indicates that non-temporal stores are used in order to avoid extra transfers from memory through the cache hierarchy for stores, and `avx512` indicates that 512-bit wide AVX instructions are used as opposed to scalars.

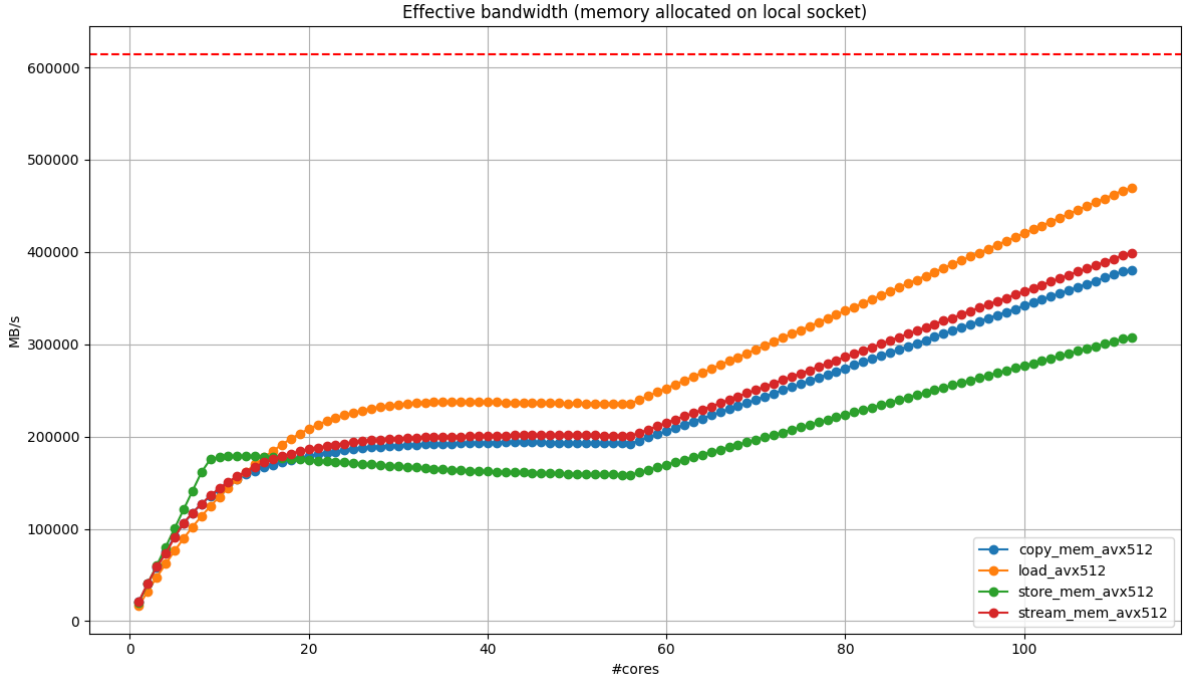

Figure 2: Peak bandwidth for contiguous memory access patterns with different load/store ratios, as number of cores increases up to one complete node (112 cores).

We can notice a few things from these results. Firstly, the achieved bandwidth depends on the load to store ratio. It is higher for loads than for stores. This can be explained by the fact that DRAM timings for writes are generally longer than for reads. However, as expected, even the achieved bandwidth for pure loads comes short of the theoretical bandwidth calculated

[3] referring to a contiguous access pattern

above. The peak bandwidth with 112 cores is 480GB/s which is around 80% of the theoretical 600GB/s.

Furthermore, the achieved bandwidth when only few cores are involved is way below. This demonstrates that the CPU is designed with multi core performance in mind. A single core cannot issue enough concurrent memory accesses. In the multi-core case, cores will make memory accesses concurrently, and "fairness" also becomes important. Dedicating extra hardware to exploit achievable single core memory bandwidth would be wasteful, or even counterproductive. Nonetheless, if all cores access memory concurrently, the bandwidth saturates, and may even slightly deteriorate.

The straight line when scaling with more than 56 cores is due to the fact that, in our benchmarks, when increasing the thread count, we first fill one complete socket, and then the second one. So for example with 60 threads involved, 56 run on (the 56 cores of) socket 0 and 4 on socket 1. Therefore, when increasing beyond 56 cores, the performance on socket 0 remains constant. The socket determines overall execution time, as its threads have lower bandwidth/core. Thus, a linear increase in threads on socket 1 results in a linear increase in measured bandwidth. When all 112 cores across the two sockets are used, the load imbalance disappears and we get twice the bandwidth compared to 1 socket.

4.2 Local vs Remote Memory

The above peak bandwidth assumed that all threads access memory on the local socket. But what if threads access memory on the opposite socket? Similarly as above, we can calculate a theoretical peak based on the spec sheet. The Intel Xeon Platinum 8480+ processor supports up to 4 UPI links, each has a duplex bandwidth of 16 GT/s, i.e. 32 GB/s. That amounts to a total peak bandwidth of

$$32 \text{ GB/s/link} \cdot 4 \text{ links} = 128 \text{ GB/s}$$

For testing the impact of accessing remote memory we can use the Linux command utility *numactl*. This command let's us change the operating system NUMA policy to control in what socket (NUMA node) are threads and memory located. However, we need to use a different simpler benchmark to fully demonstrate the effect exerted by the command.

The program *mpbw.cpp* that also measures peak memory bandwidth is used in this test. Similarly to *likwid-bench*, it will gradually increase the number of cores used but only while performing read operations and up to 56 cores. For this reason, it produces a bandwidth curve in Figure 3 similar to the first half of "*load_avx512*" in Figure 2.

We first run it with two NUMA modes, (1) all threads and memory is bound to Socket 0 (as previous tests) and, (2) all threads are bound to Socket 0 while memory is allocated in Socket 1. From the results it is possible to observe the expected behavior: accessing remote memory, even it is via a high performance technology like UPI, it drastically shrinks the maximum bandwidth. In the optimal configuration, PBW reaches 220GB/s while in the second configuration it reaches 126GB/s.

Finally, driven by curiosity we apply a third possible NUMA configuration: *interleave=all*. To our knowledge, this would allocate processes and memory in both sockets following a round robin fashion. The results of this experiment are shown in Figure 4.

Without adding more cores (still 56) the achieved PBW rises up to 360GB/s. At first this appears counterintuitive, as we have not scaled the active number of resources in the computing node. However, we have to keep in mind that the PBW improvement decays rapidly as contention in a Socket's memory controller is developed. When the cores, half the cores are distributed in each socket, we can add up almost linearly each socket's PBW. This is a surprising

and practical insight for improving applications provided we don't care leaving half of the cores in a node idle. One scenario where this is useful would be when we know that half the cores would be memory bound while the rest does not stress much memory.

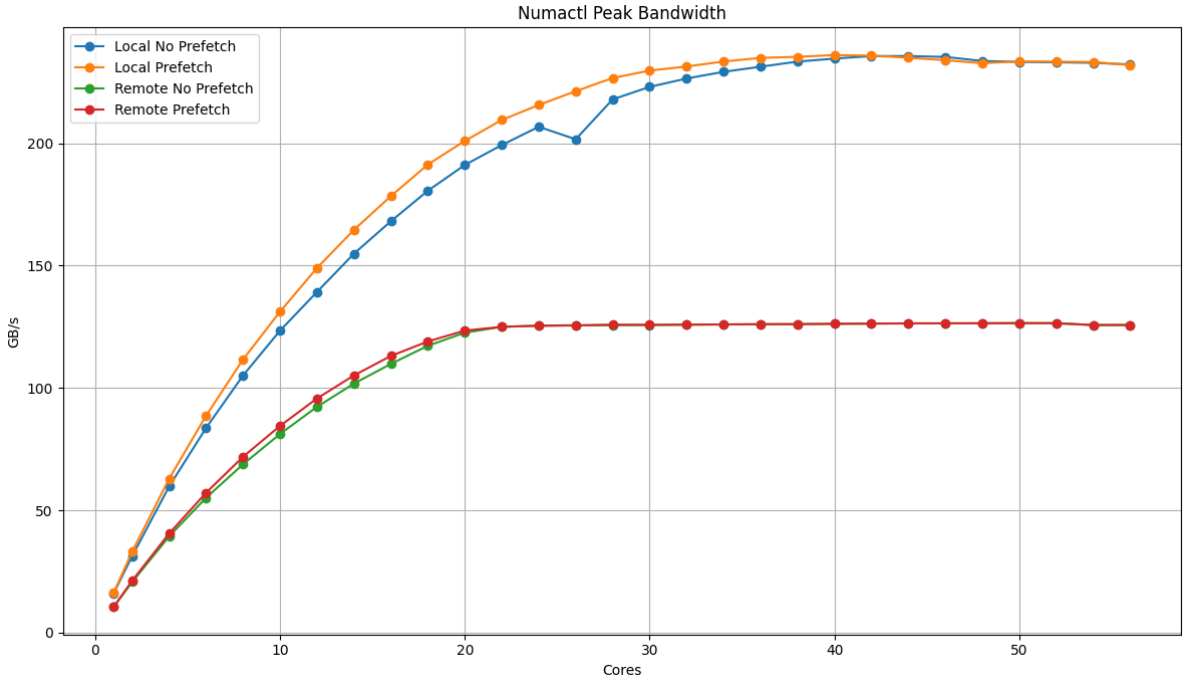

Figure 3: Cache Latency as array size increases.

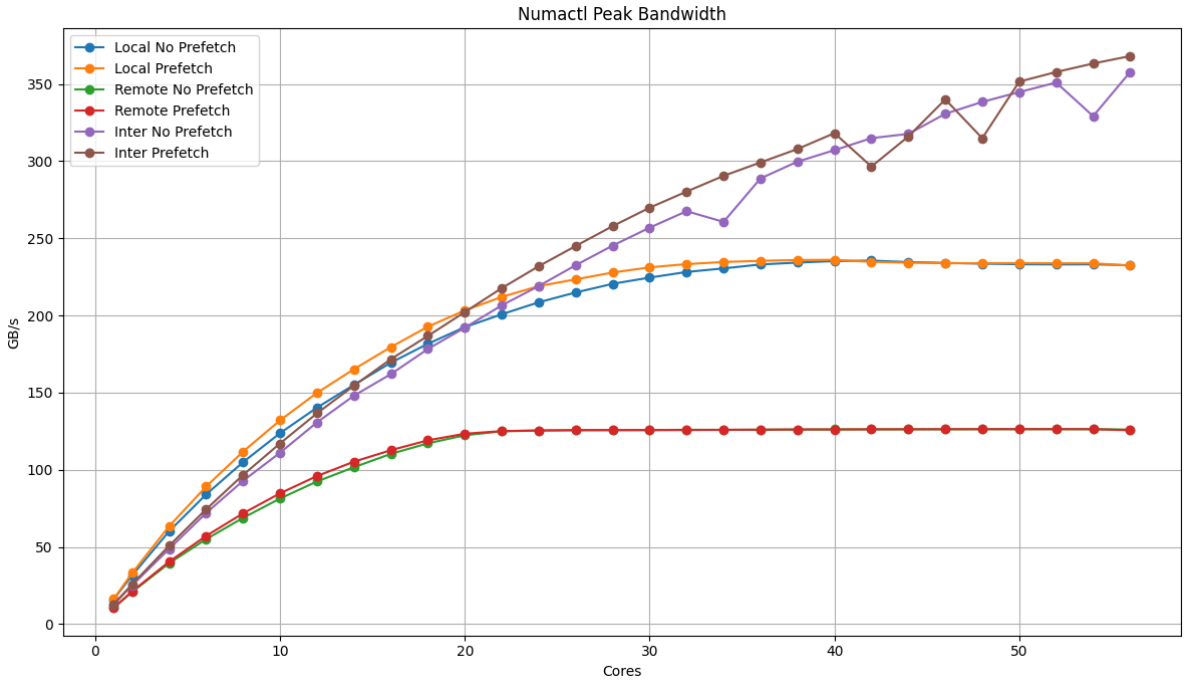

Figure 4: Cache Latency as array size increases.

4.3 Cache Latency and Cache Sizes

The idea behind measuring Cache Size is simple: measure the performance of accessing the elements of a small data buffer while the size increases progressively. At the beginning, the buffer should fully fit in L1 however, when the buffer surpasses the size of L1, it should available only in L2 and thus we will get an sharp performance penalty. The same thing will happen from L2 to L3 and beyond L3.

While the principle is simple, a couple of considerations should be made. First, as the array size will be small and we will measure mainly components in a single core (L1 and L2 caches), we should focus on access Latency rather than Bandwidth. In this work we measure by latency time (ns) but CPU cycles is also commonly used. Secondly, to measure full access latency, we have to avoid hitting items in the same cache-line or prefetched cache lines. In order to do it, it is common to use a random-cyclic pointer-chasing access pattern as described by [[6]]. Finally, due to the low latency and cycle period in the small array sizes, the number of repetitions is higher at the beginning and decays significantly for the largest array sizes.

Following these considerations, MemoryLatency.c measures the cache latency when raging the array size from 2 KB up to 1 GB and dynamically adjusting the number of repetitions of memory loads. The results are shown in Figure 5.

The results confirm the cache sizes. First, we can observe that the latency is almost zero at the beginning and then it ascends after 48 KB (L1 size). Then there is a big rise after 2 MB (L2 size) that stabilizes at 4 MB. Finally, we can see the steep rise after 105 MB (L3 size).

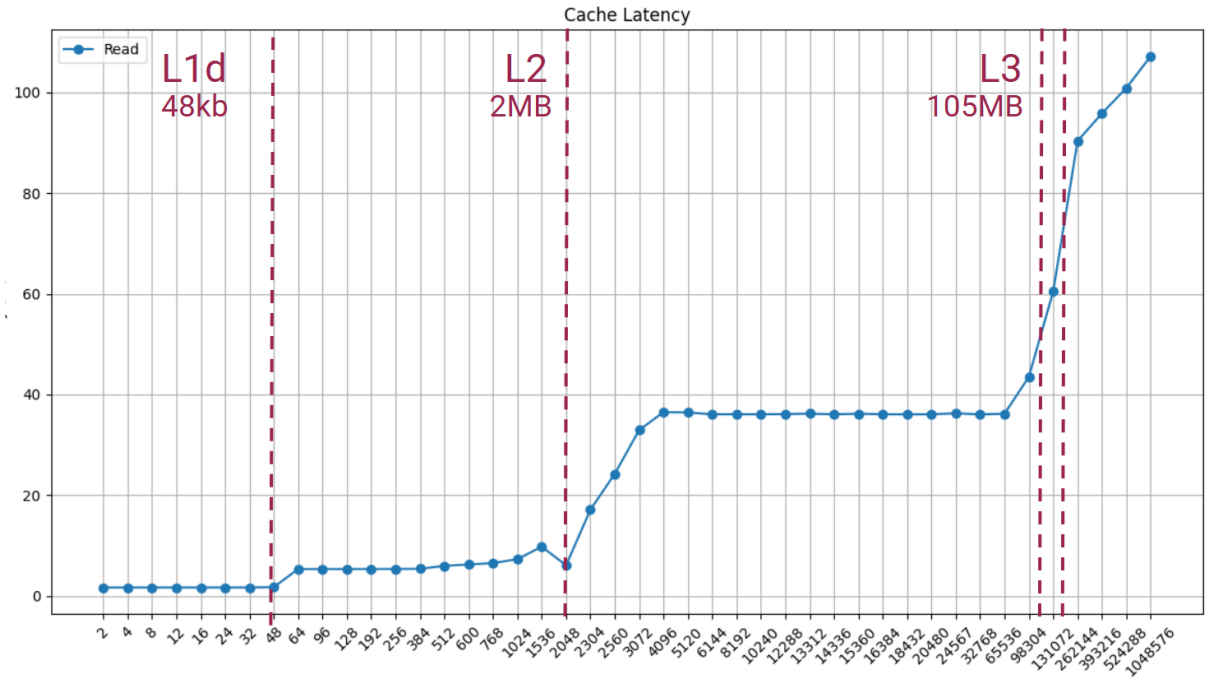

Figure 5: Cache Latency as array size increases.

Notice howeve that final L3 latency is not reached until 4MB. This probably happens due to the fact that the majority of the array fits in faster L2 at the start. However, at 4MB this trend stops, half of the array is in L2 and the other half in L3, so highest latency starts to dominate the memory loads.

Lastly, we compute average latencies by grouping measurements based on the cache sizes. Table 1 shows the average latency per cache and the relative slowdown (latency) compared to the

previous cache. L2 latency is comparable to L1 while having a much bigger cache size. However, L3 latency is an order of magnitude slower compared to L1.

Cache	Avg, Latency (ns)	Slowdown
L1	1.6803	1x
L2	6.1627	3.37x
L3	36.0355	5.85x

Table 1: Cache average latencies and relative slowdown to previous cache.

4.4 Effects of Different Cache Access Patterns

By varying the Cache Latency measuring code a couple other experiments are possible. The following results show a proof of concept, however, they have to be made more robust.

Varying Array Stride

During the pointer chaise, each accessed element is guaranteed to be located with an offset of 64 bytes at least, (the size of the cache line), to ensure each load always hits a new line. However, we can explore the effect of using bigger and lower offsets (strides) than the baseline (64 bytes) before the randomization to generate the random cycle path. We execute the same benchmark with strides 8, 16, 32, 64, 128 and 256 bytes and show the results in Figure 6.

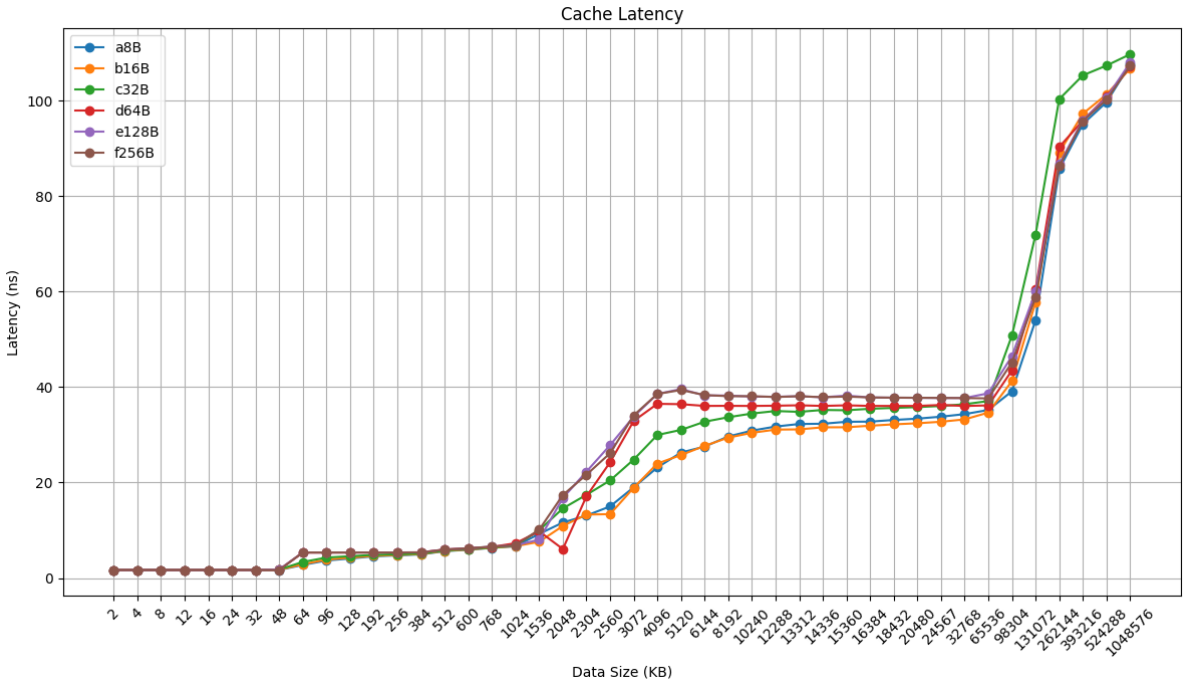

Figure 6: Cache Latency with different array strides.

The latency lines for the different strides are very similar for most of the array sizes. This is possibly due to the fact that the pointer-chase addresses are random, so even though some can be located in the same line, the probability is low. Despite of it, there is a visible effect when stepping into L3 which results in lower latency for smaller stride sizes.

Processor Parallel Cache Access

Each processor is capable of handling multiple memory streams in parallel, so if we generate independent pointer-chasing instructions, we can measure the level of optimization in simulta-

neous requests. This is done by using the same random-cyclic array but with different starting positions for each pointer-chase; scaling the amount of pointer chases requires minimal additional memory. For this test the Bandwidth (MB/s) is computed again as it is the proper measurement when aggregating the memory usage and up to 32 independent pointer-chases. The results are displayed in Figure 7 with additional division lines that separate the measurements based the cache they fit in.

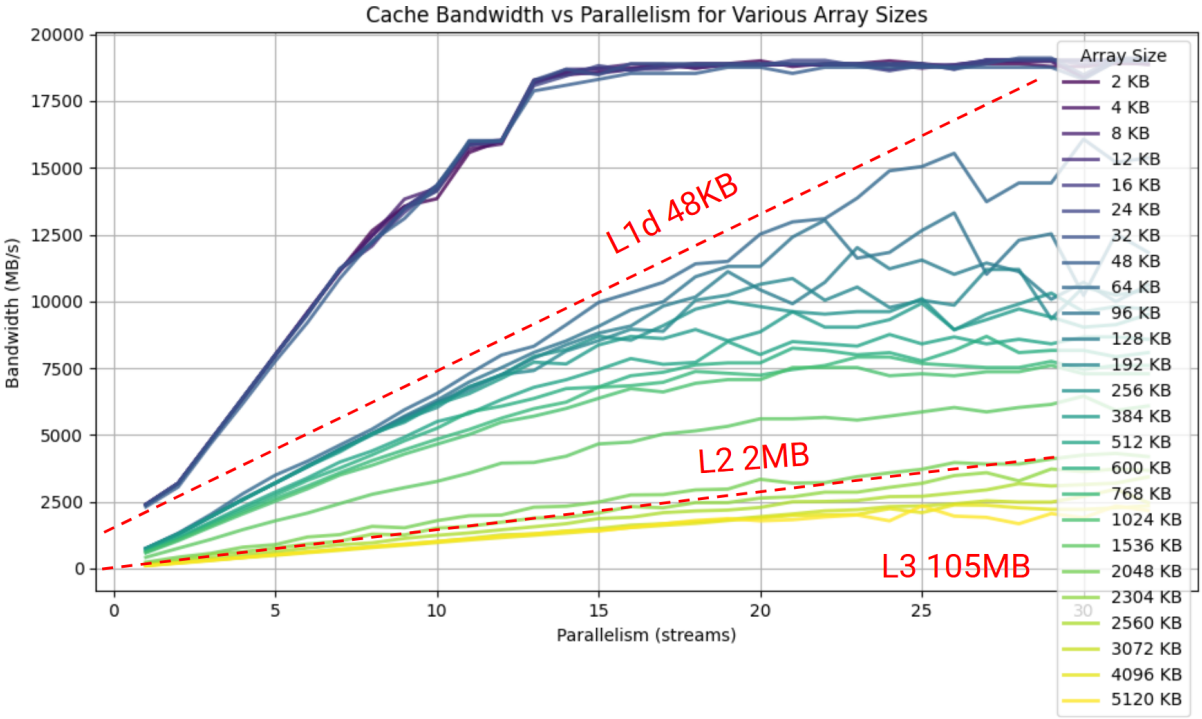

Figure 7: Bandwidth with different array strides and gradually increasing CPU cores.

For all the tests, increasing the number of independent memory accesses generally increases bandwidth as expected. Yet, there is a surprising pattern in the bandwidth variance depending on the cache the array fits in. For all the array sizes that fit in L1, we see that the achieved bandwidth is the same one across all and the highest. Furthermore, the distance of improvement in with respect the closest L2 measurement increases dramatically for the first 15 parallel streams. Afterwards, the bandwidth is saturated at 19GB/s.

When arrays fall into L3 we also see low variance and the worst bandwidths possible. However, arrays that fit in L2 have a high degree of variance ranging from 2.5GB/s to 15GB/s when 25 parallel streams are used.

4.5 Data Sharing Between Cores (Coherence)

Throughout this section, we will look at the effects when several cores access shared data. In particular, we compare what happens in the read-only, write-only, and update (read+write) cases. Each of the experiments has the same structure. After adequate initialization and warmup, an OpenMP parallel region is started with the desired number of threads. Each of the threads goes into a loop accessing the same shared variable a large number of times and subsequently completes[4]. The time is stopped before the creation and after completion of the parallel region. We choose the amount of work within the parallel region large enough such that the overhead of thread creation, final synchronization and destruction is negligible. Dividing

[4]OpenMP implicitly synchronizes before the threads actually complete

the walltime by the number of loop iterations gives a close estimate of the achieved throughput. Note that aesthetic reasons, we report the inverse of throughput.

Read-only The experiment's loop body consists of the single instruction[5]

```
movsd xmm0, QWORD PTR [%[x]].
```

As the number of cores increases, a very slight increase in memory access time gets visible when we zoom in, but this can be explained by the previously mentioned overhead of thread creation and destruction.

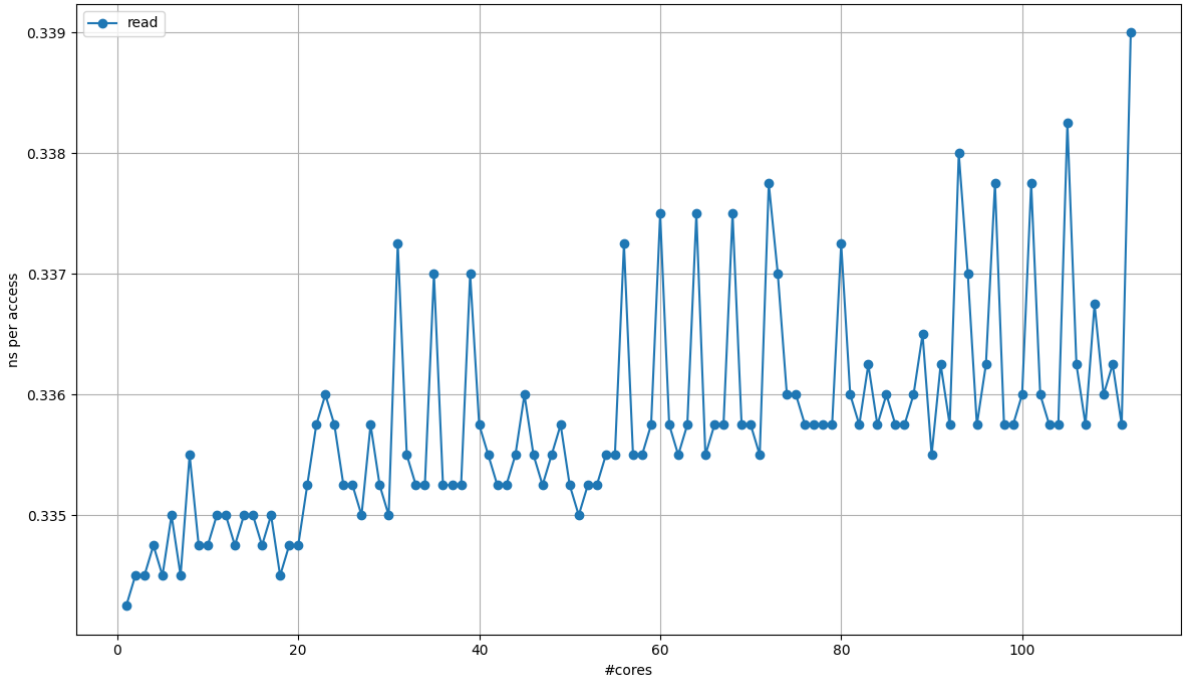

Figure 8: Average nanoseconds per read as the number of reading cores increases.

So we conclude that, irregardless the number of cores concurrently reading the data, the access time remains constant. This is expected, and confirms that the processor's coherence protocol has a logical state for clean, shared data, in which the data can be read without broadcasting the event to other cores. The average access time (as inverse of throughput) at around 0.33ns. This corresponds to the cycle time of the processor. Although the processor has more than one load port, the RAW dependency of incrementing the loop variable limits the number of executed loop iteration per cycle to 1. As the experiment only consists of one load instruction per loop iteration, its throughput is also limited to 1 load per cycle[6]. Running adjusted versions of the benchmarks with more than one instruction per loop iteration confirms this. For example, when replacing the loop body with

```
movsd xmm0, QWORD PTR [%[x]]
movsd xmm0, QWORD PTR [%[y]]
```

the average access time halves (i.e. the walltime does not change).

An experiment such as *read*, with an effectively 'empty' loop body can also be used to conclude

[5]GNU inline assembly syntax (masm dialect)

[6]We didn't notice this simple fact before the presentation, and so lacked a proper explanation for why the performance is limited

the processor frequency[7].

Write-only Here, we compare two experiments.

- In *write*, each processor writes some data to the same cache line analagous to the read experiment. The instruction in the loop body is

  ```
  movsd QWORD PTR [%[x]], xmm0.
  ```

- In *alternatingWrite*, each processor writes some data alternately to two one of two cache lines. The loop body consists of

  ```
  movsd QWORD PTR [%[x]], xmm0
  movsd QWORD PTR [%[y]], xmm0,
  ```

  where the variables `x` and `y` are on different cache lines.

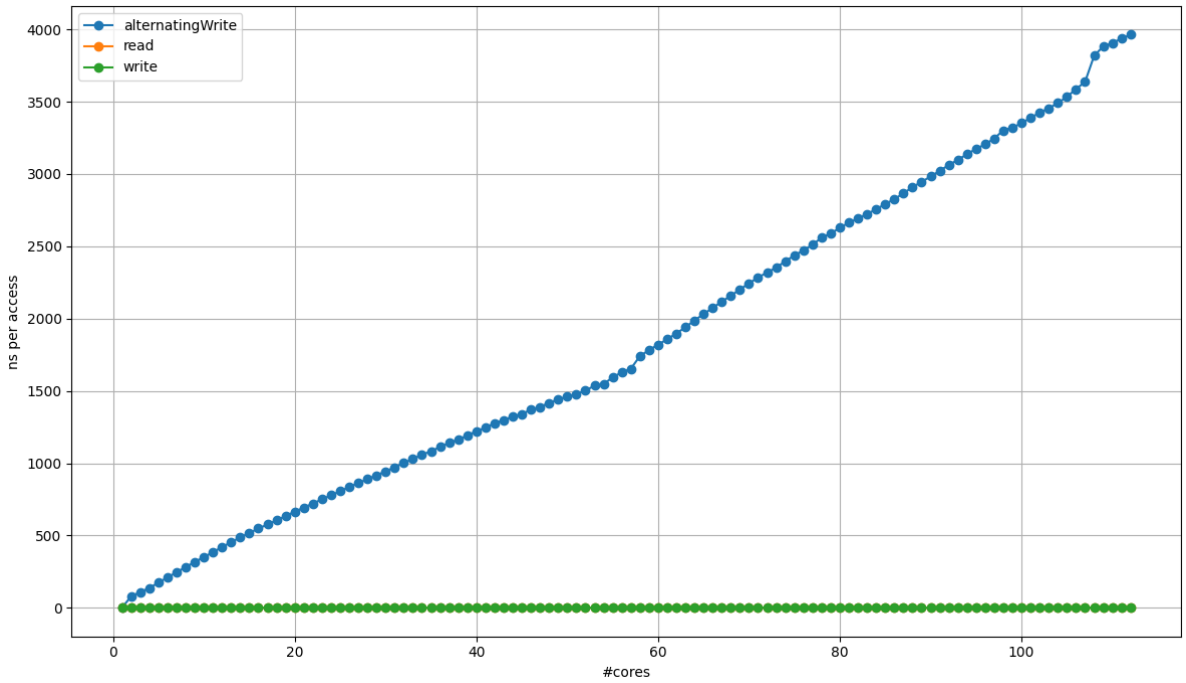

Figure 9: Average nanoseconds per write as the number of writing cores increases. In *write*, each processor writes data repeatedly to the same cache line. In *alternatingWrite*, each processer writes data repeatedly to one of two cachelines in an alternating fashion.

In Figure 9, we can see the results compared to the read-only case. It is clear that when several cores write data to memory, some messages will need to be sent in order to at least inform the other cores that their data has become stale. So the fact that *write* scales just as well to many cores as *read* may come as a surprise initially. But it can be explained if the cores have write buffers that coalesce subsequent writes to the same cache line. As such, the result is an indication that this is indeed the case. Specifically, since we keep writing to the same variable and nothing else, each write can be coalesced with the previous, and it need not become globally visible anytime soon. Hence, the cache line itself may not be updated until the very end.

In order to comply with TSO consistency, however, the write buffer needs to be FIFO and cannot coalesce writes if they are not subsequent. The *alternatingWrite* experiment is specifically

[7] As far as variable frequency issues allows

designed to exploit that fact, in order to force each write to become globally visible relatively quickly (the write buffer cannot be infinite). Indeed, *alternatingWrite* shows performance degradation as the number of involved cores is increased. The degradation is approximately linear. That is, as the number of cores doubles, the average throughput achieved by a single core halves. This indicates that all write requests pass through a serialization point at which they are logically ordered. As such, the performance is limited by the throughput of the serialization point. It is rather impressive that the performance degradation is not super-linear.

Update Lastly, we consider the case where each core reads and writes to the shared memory location. Using

```
movsd xmm0, QWORD PTR [%[x]]
movsd QWORD PTR [%[x]], xmm0
```

in the loop body is affected by write buffer coalescing just as the write-only case. TSO consistency allows younger writes to become visible before older stores, so there is nothing preventing that optimization. The actual results are on the left in Figure 10. Notably, even the single core throughput is 8x lower than for *read*. At least in part, this may be explained by the RAW dependency between the two instructions. It also stands out that, although subtle, there is a noticeable performance degradation when adding more cores. Perhaps, the cores eagerly flush data from the write buffer to the cache.

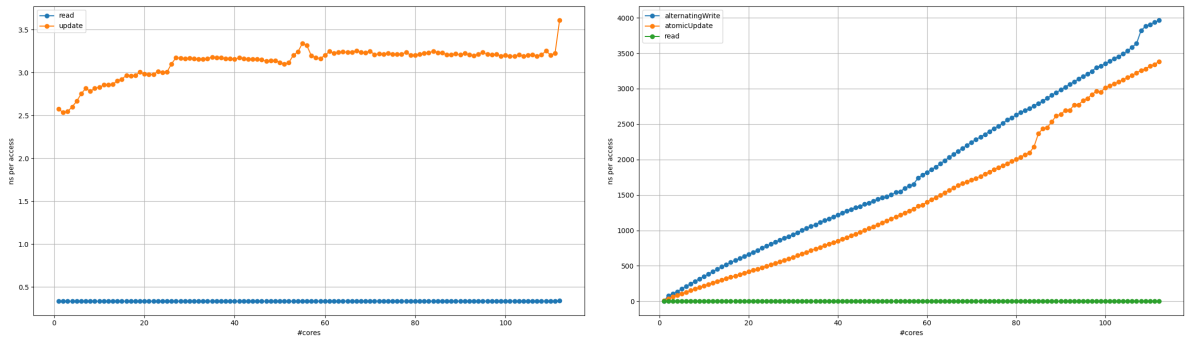

Figure 10: Average nanoseconds per update as the number of writing cores increases. Left: Non-atomic update. Right: Atomic update.

A practically more relevant type of update are atomic updates, such as

```
lock add QWORD PTR [%[x]], 1.
```

The experiment *atomicUpdate*'s loop body consists of exactly that. By definition, atomic operations need to execute atomically, so there is no possibility to coalesce writes here. In the right of Figure 10, we can see a linear performance degradation for *atomicUpdate* similarly to *write*. The constant factor actually is better for the former. It is possible that this is due to the lower initial performance in the single core case for atomic operations (∼6ns vs ∼0.33ns per access), perhaps leading to less congestion in the communication network.

5 Conclusions

In this project, we explored microbenchmarking techniques using a combination of self-developed code and existing reference benchmarks to investigate performance characteristics of the MareNostrum 5 GPP nodes. Several results aligned with expectations, confirming information available in hardware specifications previously well-know architectural behaviors. However, other findings provided valuable insights, such as the effects of NUMA interleaving on peak bandwidth,

variations in each cache's bandwidth, and observations related to lazy write coherence mechanisms.

These results serve as a useful reference for understanding the performance profile of MareNostrum 5 and for analyzing complex CPU behavior in multi-scoket, multi-core systems. Throughout the study, custom benchmarking code was iteratively refined by reviewing and incorporating ideas from open-source implementations, which further supported both the learning process and the reliability of the measurements.

While the experiments provide meaningful observations, there remains room for further improvement and extension. Potential future work may include expanding the analysis to additional architectural features, studying the convergence trend of memory access, or applying the methodology to other HPC systems.

References

[1] clamchowder. *Microbenchmarks*. URL: `https://github.com/clamchowder/Microbenchmarks`.

[2] Daniel Lemire. *Estimating your memory bandwidth*. URL: `https://lemire.me/blog/2024/01/13/estimating-your-memory-bandwidth/`.

[3] John D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Tech. rep. A continually updated technical report. http://www.cs.virginia.edu/stream/. Charlottesville, Virginia: University of Virginia, 1991-2007. URL: `http://www.cs.virginia.edu/stream/`.

[4] Gabriele Paoloni. "How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures". In: *Intel Corporation* 123.170 (2010).

[5] BSC Support. *MareNostrum 5 System Overview*. URL: `https://www.bsc.es/supportkc/docs/MareNostrum5/overview`.

[6] Henry Wong. *Intel Ivy Bridge Cache Replacement Policy*. Jan. 2013. URL: `http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/`.

A Project Repository

Project Repository: https://github.com/JJOL/Memory-Performance-Microbenchmarking