NUMAPROF, A NUMA Memory Profiler

Sébastien Valat[1(✉)] and Othman Bouizi[2(✉)]

[1] CERN, Meyrin, Switzerland
sebastien.valat@cern.ch
[2] INTEL, Meudon, France
othman.bouizi@intel.com

Abstract. The number of cores in HPC systems and servers increased a lot for the last few years. In order to also increase the available memory bandwidth and capacity, most systems became NUMA (Non-Uniform Memory Access) meaning each processor has its own memory and can share it. Although the access to the remote memory is transparent for the developer, it comes with a lower bandwidth and a higher latency. It might heavily impact the performance of the application if it happens too often. Handling this memory locality in multi-threaded applications is a challenging task. In order to help the developer, we developed NUMAPROF, a memory profiling tool pinpointing the local and remote memory accesses onto the source code with the same approach as MALT, a memory allocation profiling tool. The paper offers a full review of the capacity of NUMAPROF on mainstream HPC workloads. In addition to the dedicated interface, the tool also provides hints about unpinned memory accesses (unpinned thread or unpinned page) which can help the developer find portion of codes not safely handling the NUMA binding. The tool also provides dedicated metrics to track access to MCDRAM of the Intel Xeon Phi codenamed Knight's Landing. To operate, the tool instruments the application by using Pin, a parallel binary instrumentation framework from Intel. NUMAPROF also has the particularity of using the OS memory mapping without relying on hardware counters or OS simulation. It permits understanding what really happened on the system without requiring dedicated hardware support.

Keywords: NUMA · Memory · Profiler · Instrumentation · Pin
Access · Remote · MCDRAM · KNL

1 Introduction

In the late 2000s, the number of cores in servers and HPC systems increased a lot, with commonly at least two CPUs per server. There is now up to 72 cores in one CPU if we consider the up-to-date Intel® Xeon Phi™ codenamed Knight's Landing (KNL). In order to avoid hitting the memory wall [17], most of the current architectures became NUMA (Non-Uniform Memory Access) meaning

that each processor has its own memory. This way we can easily increase the memory bandwidth of the system to feed the cores. This architecture now applies inside the socket itself like for the Intel KNL which can be configured to run as up to 4 NUMA domains or looking at some AMD® processors (*e.g.* Ryzen™ 1950X).

Nowadays, all the commonly available NUMA architectures are ccNUMA (Cache Coherent NUMA) meaning that the synchronization between the NUMA domains is automatically handled by the architecture without requiring any intervention from the developer. This is certainly one of the reasons for the wide adoption of this kind of architecture as all existing programs can run out of the box on them. Even if it is simple to make an application running on NUMA machines, it is hard to extract the full performance of the machine because of the data remote access costs.

There are now some studies about using OpenMP on multiple NUMA domains [15], some are also explicitly tuning the OpenMP runtime for this usage [5]. Using multi-threaded applications over NUMA domains fully rely on the developer to well place the data in memory to efficiently exploit the architecture and ideally limit the remote memory accesses. Strictly avoiding them is not always a good thing; it is sometimes better to spread the data over multiple NUMA domains to get more memory bandwidth. This is where the knowledge of the developer is needed and the automation becomes limited.

As we will show in the next section, on most of the operating systems, the semantic to set up the memory placement is by default implicit. It relies on the first touch policy which can lead to many mistakes. Without any profiling tools, the developer must rely on his confidence and on global timings measurement to believe that he made the placement right. This is where we present the NUMAPROF tool. It provides a profiling backend associated with a graphical interface annotating the source code with various metrics. Such a tool can also be useful for developers working on NUMA-aware runtimes like MPC [10].

Section 2 will first look on the related work by listing the available NUMA profiling tools. Section 3 will contrast our contribution. As our tool mainly targets the Linux operating system we will describe the available API in Sect. 4. Section 5 will provide the technical details about NUMAPROF. We will lastly provide an analysis example by using NUMAPROF on an application and show some findings.

2   Related Work

Some NUMA profiling tools already exist and we can find several papers on this topic. We first found NUMATOP [19] which is the "simplest" one. It is an application similar to `top` which counts the number of local and remote accesses of all running applications and displays the result into the terminal. The tool relies on hardware counters to determine the local and remote accesses. It is interesting for looking for the processes placement when running multiple applications on the same node and check their binding. As a limitation, it does not provide any details on the source of the remote accesses inside the application.

SNPERF [11] provides charts with memory accesses on each NUMA domain over time within a given process. It relies on hardware counters of the Origin 2000^{TM} architecture. This is useful to check if the memory bandwidth is balanced over the nodes. This tool is not available on the web and is hardware dependent on an uncommon architecture.

Memphis [9] is a tool making deep instrumentation using AMD Opteron^{TM} hardware counters. It uses event sampling to report the domain access count onto source code lines. For this property, it looks like what we want. But it is now unavailable and its code is dependent on a specific old hardware. From the paper it also looks not providing a graphical interface to exploit the profiles it generates.

For another approach, there is MemProf [7] which, this time, focuses on access patterns. The tool tracks each thread memory access flow by using a kernel module and tracking threads binding to detect specific bad patterns in the application. Again, it relies on specific hardware feature from AMD®: Instruction Based Sampling (IBS) [6].

One can think about a tool based on Valgrind [13] and we can find such a tool: NUMAgrind [18]. This time, the tool is based on an architecture simulation, so, not relying on specific hardware counters. It simulates all the cache details and also the page affinity. This approach is interesting because it allows simulating any kind of architecture with the drawback of neglecting the effective operating system memory mappings. Last, it relies on Valgrind which does not allow to run threads in parallel, making the overhead really big for a large number of cores. This is an issue to simulate something on the KNL, for example. The tool is not available any more on the Internet.

Still looking for hardware simulation we can also find SIMT [14] which fully simulates the architecture details. This tool simulates the interconnect protocol and can be used for architecture design but can also give some hints about applications on those architectures.

Lastly, we found HPCToolkit [8] which is really close to what we want to provide with NUMAPROF and is also open source. It uses hardware counters and events sampling to track the application. It then provides a nice reporting by annotating the source code and the call stacks in a graphical interface. Interestingly, it provides all the usage information of a variable in one go: allocation site, first touch site and all access site. It also provides a summary of the access pattern over threads so the developer can look at how the memory accesses are distributed and might find some reordering or packing to improve data locality. This is something we do not have in NUMAPROF. Due to its usage of hardware counters it has a low overhead but becomes dependent on specific hardware. It has the advantage of providing a kind of cost estimation due to the sampling approach which is something we do not yet have in NUMAPROF. The NUMA part of the tool described in the paper is not available in the public release.

Really close to our approach for the backend implementation we also found Tabarnac [4] which also instrument the memory accesses. It then generate a static web page as output with limited charts.

We finally took inspiration and codes from MALT [16], a malloc tracker, for its web-based interface providing global metrics and source annotation. Web GUI are interesting as it can be easily forwarded via the ssh-port forward, avoiding the latency of rendering when forwarding a full X application. It also helps to quickly implement the interface and open possibility to work with multiple people connected onto the same interface.

3   Contribution

Looking on the available tool we notice a few things. Many of them rely on dedicated hardware counters which make them quickly outdated and not being able to be used on up-to-date architectures. This way, they also, most of the time, rely on sampling which might miss some details of the code.

On the other side we have tools working on hardware emulation which is nice as allowing simulation of any kind of hardware on our workstation. But it has the disadvantage to miss the real memory mapping built by the operating system we run on. Also a lot of them are not yet/anymore available.

Last, all the listed tools, except HPCToolkit and Tabarnac (non interactive), provide raw text output which might be hard to interpret. To be useful, the tool needs to come up with an attractive graphical interface annotating the source code with the extracted metrics.

In order to overcome these issues, we want to build NUMAPROF following those rules:

– We do not rely on hardware counters.
– We keep track of the real mapping of the operating system.
– We must run in parallel.
– We provide a graphical interface to look on the profile annotating the code lines.
– We want to make it Open-Source to be available to the community, available at [3].

With NUMAPROF we allow checking what the operating system does for the mapping. This can be useful to validate codes but also runtimes which are supposed to help handling multithreading. Being independent on hardware counters ensure a better support over time by not being dependent on a specific architecture. On this aspect we also provide dedicated efforts on handling the huge pages semantic when considering the first touch measurements which is explicitly handled by none of the listed tools.

Although we could have used Valgrind to make the binary instrumentation, it would lead to large overhead when running on nodes like the Intel KNL handling up to 288 threads. Hence, we proceed by using Pin [12]. This tool does an on-the-fly binary instrumentation providing services very similar to Valgrind but with multi-threading support and being a little bit faster. It is also easy to use, just as Valgrind, wrapping the command line we want to run and making an on

the fly instrumentation. NUMAPROF instruments all the memory accesses and check the NUMA distance of each access to build the profile.

In addition to the local and remote metric, the tool also adds a new metric: unpinned. This permit to track cases where the running thread is not bound to a specific domain and making "random" accesses generating "random" placement of the memory. Compared to other tools we also searched to provide information on the source lines making the first touch. Hence, we pinpoint the source of the remote accesses. Lastly, we also provide access counter on memory allocation site to detect which allocated chunks lead to incorrect accesses.

The tool also provide dedicated metrics to handle the case of the MCDRAM from Intel Knight-Landing helping profiling on this architecture.

Our tool currently does not provide caches simulation so we report raw accesses not considering cache effects. This might be done in a second step to get more meaningful metrics when considering spin locks which in practice stay in the cache not generating effective remote accesses (eg. in the OpenMP runtime).

4   NUMA Linux API

This section will describe the Linux NUMA API to understand what we want to observe and which operations we need to intercept to track the NUMA state of the application.

We first remember that most operating systems allocate a virtual segment when an application makes a big allocation. It then fills it, on the fly, with physical pages when the application starts to really access it. This is called the first touch policy. Operating systems like Linux decide during this first access which NUMA page to map, by default by looking at the current thread location. This way the operating system tries to automatically fit the data accesses by considering the memory usage will later be done in the same way. This approach is nice but leads in some ways to problems because it is implicit and many non-expert developers might not know what they are really doing. We will show that our tool is specifically tuned to track those problems.

Hopefully, the user can change the first touch policy of a segment by using the `membind` system call. It setups a strict binding (`MPOL_BIND`) of a segment to a specific NUMA domain or forces page interleaving (`MPOL_INTERLEAVE`). It can lastly set a preferred domain (`MPOL_PREFERRED`) which will be chosen and neglected if there is no free pages any more on this domain. This `membind` call is specific to Linux. In last resort, the user can also provide a detailed mapping by using the `move_pages` system call providing the mapping for each page of the targeted segment.

By default, the threads are spawned randomly on a core and can move depending on decisions from the operating system scheduler. But, as the memory is linked to the process, the OS has no knowledge of the link between data and threads. When a thread moves to another NUMA domain, there is no information permitting to also move the related data. The only available real solution is to bind the threads on a specific core, to place the data accordingly and keep this state for the whole execution.

This thread mapping can be handled via the `sched_setaffinity` system call which is also not POSIX. This call takes a mask as parameters to allow the thread to run on the given cores (or CPU threads if using hyper-threading). This mapping is automatically done when the user sets the `OMP_PROC_BIND` environment variable for OpenMP.

The user can also handle the thread and memory binding by selecting its behaviour through the command wrapper `numactl`.

Again, we see with this interface that nothing links the threads to the data so the placement policy is implicit meaning that nothing prevents the user from mistakes and nothing can notify the user if he does the thing wrong. This is where tools like NUMAPROF can be useful. Also, all the aforementioned interfaces are Linux dependent so our tool will target only this operating system.

5   Implementation Details

This section will give details about the NUMAPROF implementation going from the backend up to the graphical interface.

5.1   Metrics

As many other tools, NUMAPROF provides the local/remote access metric. But it also track the unpinned accesses, meaning the thread is not bound to a specific NUMA domain and can move on other nodes. It also tracks if a page has been first-touched by such a thread, meaning the page is "randomly" placed on the machine generating later random accesses.

NUMAPROF also finds the code location where the page placement was made, which happened at the first touch. Lastly, it counts the access metrics on the allocation site, meaning we can quickly know which allocated segments are concerned by remote memory accesses.

To summarize, we provide the listed metric for each access call site and allocation site:

first-touch. Counts the number of first touch on the given location.
unpinned first-touch. Counts the number of unpinned first touch on the given location.
local. Counts the local memory accesses when the thread is on the same NUMA domain as the page.
remote. Counts the remote memory accesses when the thread is on a different NUMA domain than the page.
unpinned-page. Counts the cases of a thread bound to a NUMA domain but the page is not.
unpinned-thread. Counts the cases of a thread not bound accessing a bound page.
unpinned-both. Counts the cases of a non-bound thread accessing a non bound page.

MCDRAM. Counts accesses to local MCDRAM memory. MCDRAM is considered when the accessed NUMA domain has no cores.

MCDRAM has two sub-counters which are local and remote to consider the case of the KNL which can be configured with 4 NUMA domains each having its local MCDRAM.

5.2   Thread Tracking

In order to build our metrics we first need to track the thread placement. For this, we need to check the thread location at spawning time. Thanks to Pin we can intercept the spawning of a thread and place our handler on it. From there we extract the thread binding by using the `sched_getaffinity` system call. From the cores/threads affinity, we extract the NUMA affinity of the thread and keep track of this state. A thread which can move onto more than one NUMA domain will be considered as unpinned. If the thread is not bound, we pursue without needing to check on every access where he is running to determine the local or remote memory access. We just track its memory accesses as unpinned.

We lastly need to track the thread movement if there are changes during the run. In the Linux OS this can happen on the call of `sched_setaffinity` which we intercept thanks to Pin. We do not manage a possible change from outside the process. But, in this case there is nothing to do except to provide a function to be called by the user.

Threads can also set up a memory policy (`set_mempolicy`) possibly assigning a remote memory domain to the process. We intercept this policy and consider the thread bound if it is restricted to one domain.

All the thread movement (which are rare) will be logged in the profile so the user can check if it matches his expectation. This is also a new feature from NUMAPROF compared to existing tools.

5.3   Memory Access Tracking

As said, thanks to Pin, we probe all the memory accesses and check if they are remote or local. We already know where the thread is, thanks to the tracking described in the previous section. We now need to know where the page is. On Linux this can be done by using the `move_pages` system call which is normally used to move the pages. If we do not provide a mapping in parameters, the call returns the current location of the pages. Notice we can request a list of pages in one call.

To limit the overhead we cannot make a system call for every memory access so we need to cache the information. As a workaround, we build a shadow page table rebuilding the same structure used in the kernel. Then we can easily detect the first access (first touch) and use the cached value on next accesses. We do not remove entries in this table so we can use it in a lock-free manner having to take locks only when we add new entries in the tree. This permit to maintain scalability.

The page table entries are allocated when intercepting calls to `mmap`. To track the state of the pages we also need to track calls to `munmap` (resp. `mremap`) to flag the related pages as released (resp. to move them). While flagging the page as free, we do not remove the tree structures if empty to keep the algorithm read lock-free.

We also implemented a dedicated support for huge pages. At first touch we request the binding with `move_pages` for all the pages of the given huge page. If all the pages are available on the same NUMA domain, we consider it is a huge page, so, accounting the first touch only once for the huge page.

One last thing, to improve the performance of the tool we do not probe the local stack memory accesses, as we know they are local. This provides in practice a speedup of a factor 2, greatly reducing the high overhead of the tool.

5.4 Instruction and Allocation Counters

We report the metrics globally, per thread, per call site (instruction) and per allocation site. For the call site, we maintain an `std::map` indexed by the instruction memory address. This structure is not lock-free so we need to take locks on every access. In order to make the profiler scalable we cache the entries of this tree into each thread. In this way, each thread can quickly update the counters in the cache and then sometimes (when there are too many entries in the cache) flush it into the global tree by taking locks.

For the allocations we again need a way to quickly find in a lock-free manner the pointer to the counters. For this, we extend the shadow page table by making an entry for every eight addressed bytes. This pointer is used to point the corresponding allocation site. We used the same approach as the instruction with a cache in the thread to store local copies of the counters and then flush the cache when it becomes too big. This limits the number of atomic operations. Notice we can up to double the memory consumption by using this approach. To limit this effect we use two storage methods. For allocation fully using a page, we use only one pointer and allocate the list of eight bytes entries only if the page contains small allocations.

To maintain this page table pointers we need to track the call to `malloc`, `free`, `calloc`, `realloc`...

5.5 Scalability

One can check the scalability of the tool on the Hydro [2] application running on an Intel KNL. The Fig. 1 shows that the tool scales up to 64 threads with an overhead of a factor 27x. There is then a slight increase with an overhead of 60x on 256 threads. In any case this is far better than what we observe with the Valgrind's memcheck tool due to the serialization of the threads.

5.6 Graphical Interface

NUMAPROF takes back the idea from MALT by providing a dynamic web-based interface. It provides global metrics summarizing the application like the
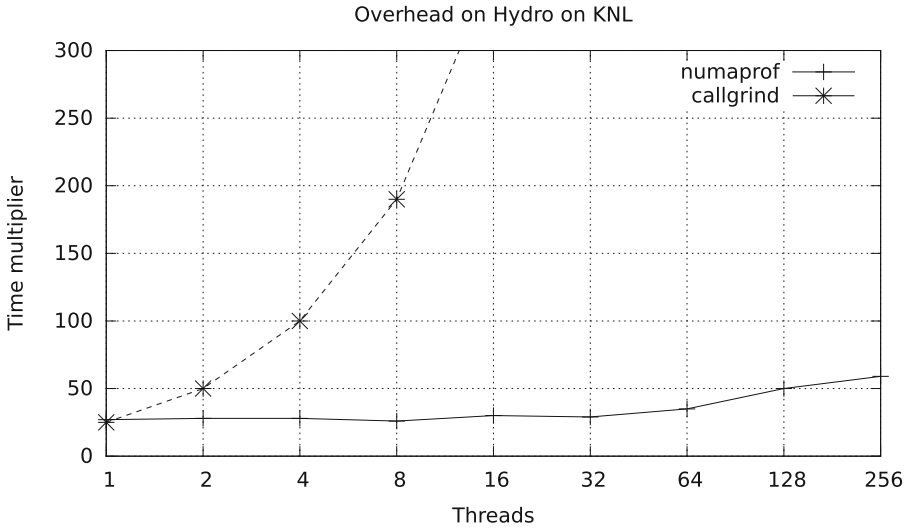
Fig. 1. NUMAPROF and Valgrind overhead on HydroC code on KNL.

global local/remote/unpinned metrics and an access matrix. The access matrix is provided to quickly check if the memory accesses are on the diagonal or form a line pointing intensive accesses to one NUMA domain. Second, the interface provides the per thread metrics to quickly see how the first touch and accesses are balanced. It also provides the pinning log and access matrix for each thread. Finally, and most important feature, the GUI provides annotation of the source code to project the counters onto the source lines as shown in Fig. 2. This is the core part of the tool.

6 Use Case Example: Hydro

We tested the tool onto some applications: AMG2013, HACC, Cloverleaf and it mostly shows that those applications were well optimized making only local memory accesses. On huge pages they showed some remote accesses when the array splitting not to match with 2 MB limits of huge pages but we observed by testing that this does not impact too much the application performance.

We then tested the Hydro [2] application (commit d1303337624) which is less tuned. On this application NUMAPROF observed interesting things. Firstly, in August 2017 we observed that when running the application on an Intel KNL was not allocating memory on the MCDRAM (in FLAT mode) when it should. Searching deeper on the issues pointed a bug form the kernel which ignored the `MPOL_PREFERRED` semantic when the huge pages are enabled. This bug was known by Red Hat [1] and has been fixed end of 2017. Moving to the right policy makes the application going down from 42.8 s to 28.0 s.
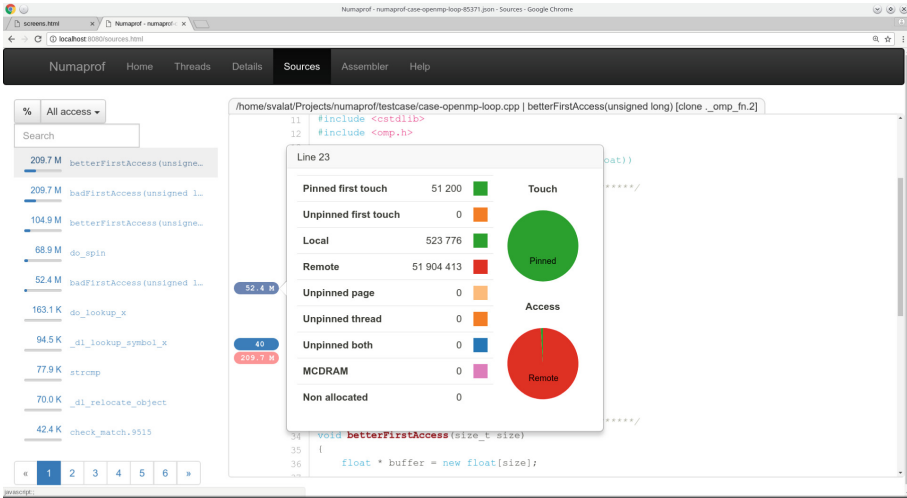
Fig. 2. NUMAPROF graphical interface, source code annotation.

Starting the real analysis, we first looked at the access matrix from Fig. 3 showing a wrong placement of some segments. The test is done on an Intel KNL with MCDRAM activated which explains the two vertical lanes, the second one being MCDRAM accesses. We see that there are issues with the blocs allocated into the main memory but also a mis-distribution of the blocks into the MCDRAM shown by the vertical lines meaning all the threads accessed data on the same NUMA domain.
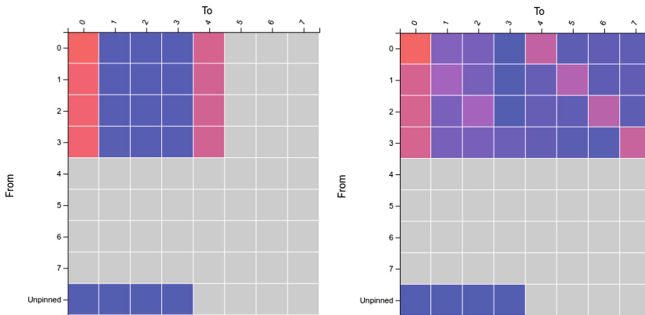

Fig. 3. HydroC access matrix before and after optimisation.

Looking at the annotated sources, we can search for remote memory accesses and more precisely on the related allocation call sites. We first found the allocation of the ThreadBuffers in the `Domain::setTiles` function which is not done in parallel. It maps all the allocated memory onto the first NUMA domain. This can be fixed by adding a `pragma omp parallel` section in place of the loop.

Second, we noticed remote memory accesses onto the Tile objects. Here there is a missing feature of NUMAPROF, we would like to have the call stack. Anyway, reading back the code leads to the call of `Tile::godunov()` which is called from `Domain::computeTimeStep()` in a loop indexed by i. If we look a little bit at the beginning of the loop, we now see that the index i is modified via the `m_mortonIdx` array. This is not done at the allocation time meaning the allocation and access orders mismatched. We can fix this by using the same index indirection in the allocation loop where we make the `new Tile`.

This slightly improves the access matrix making it more diagonal as shown by Fig. 3. This also translates onto performance by lowering the runtime to 22.9 s. This is a time reduction of 18% achieved without knowing the code in advance and by working for half an hour.

The tool still reports remote memory accesses. Most of them come from a spin lock into the OpenMP runtime. The rest mostly comes from access to global floating point constants which are stored in the data section of the binary. In other words, it is a false negative as those constant and spin locks will be in practice stored into the cache if accessed in a loop.

7   Conclusion

We showed the need to NUMA profiling tools and didn't find our wish in the existing ones. Hence, NUMAPROF was built in a simple way on top of the Pin instrumenter which quickly provided a useful tool coming with a nice web-based graphical interface. The source annotation and global metrics like access matrix had been useful to optimize the Hydro application gaining 18% of performance in half an hour without knowing the code before. Notice that the overhead of the tool was 60x on 256 threads.

Of course there is still a lot of work to be done. We mainly consider adding a cache simulator not to report too many memory access which are stored in the cache and do not slow down the application. Also we still have margins to improve the performance of the tool to reduce the overhead by mostly improving the internal caching mechanism and maybe packing the access and flushing them in groups. The tool is flexible enough that adding support of more memory levels like the 3D Xpoint should be easy.

This methodology is not limited to x86 architecture. It can be applied to other architectures by adding support of DynamoRIO [20], which is similar to Pin. As a complement one can also consider using Valgrind.

The tool is available under open source license at http://memtt.github.io/.

References

1. Huge pages and preferred policy kernel bug. https://access.redhat.com/solutions/3155591
2. Hydro. https://github.com/HydroBench/Hydro
3. Numaprof. https://www.github.com/memtt/numaprof

4. Beniamine, D., Diener, M., Huard, G., Navaux, P.O.A.: TABARNAC: Tools for Analyzing Behavior of Applications Running on NUMA Architecture. Research Report 8774, Inria Grenoble Rhône-Alpes, Université de Grenoble, October 2015. https://hal.inria.fr/hal-01202105

5. Clet-Ortega, J., Carribault, P., Pérache, M.: Evaluation of OpenMP task scheduling algorithms for large NUMA architectures. In: Silva, F., Dutra, I., Santos Costa, V. (eds.) Euro-Par 2014. LNCS, vol. 8632, pp. 596–607. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09873-9_50

6. Drongowski, P.J.: Instruction-based sampling: A new performance analysis technique for amd family 10h processors (2007). http://developer.amd.com/Assets/AMD_IBS_paper_EN.pdf

7. Lachaize, R., Lepers, B., Quema, V.: MemProf: A memory profiler for NUMA multicore systems. In: Presented as Part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12), pp. 53–64. USENIX, Boston, MA (2012)

8. Liu, X., Mellor-Crummey, J.: A tool to analyze the performance of multithreaded programs on NUMA architectures. SIGPLAN Not. **49**(8), 259–272 (2014)

9. McCurdy, C., Vetter, J.: Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In: IEEE International Symposium on Performance Analysis of Systems Software (ISPASS), pp. 87–96 (2010)

10. Pérache, M., Jourdren, H., Namyst, R.: MPC: a unified parallel runtime for clusters of NUMA machines. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 78–88. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85451-7_9

11. Prestor, U.: Evaluating the memory performance of a ccNUMA system. http://www.cs.utah.edu/~uros/snperf/thesis.pdf

12. Roy, A., Hand, S., Harris, T.: Hybrid binary rewriting for memory access instrumentation. SIGPLAN Not. **46**(7), 227–238 (2011)

13. Seward, J., Nethercote, N.: Using valgrind to detect undefined value errors with bit-precision. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC 2005, pp. 2–2, USENIX Association, Berkeley, CA, USA (2005)

14. Tao, J., Schulz, M., Karl, W.: A simulation tool for evaluating shared memory systems. In: 36th Annual Simulation Symposium, 2003, pp. 335–342, March 2003

15. Terboven, C., an Mey, D., Schmidl, D., Jin, H., Reichstein, T.: Data and thread affinity in openmp programs. In: Proceedings of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem? MAW 2008, pp. 377–384, ACM, New York, NY, USA (2008)

16. Valat, S., Charif-Rubial, A.S., Jalby, W.: Malt: A malloc tracker. In: Proceedings of the 4th ACM SIGPLAN International Workshop on Software Engineering for Parallel Systems, SEPS 2017, pp. 1–10. ACM, New York, NY, USA (2017)

17. Wulf, W.A., McKee, S.A.: Hitting the memory wall: implications of the obvious. SIGARCH Comput. Archit. News **23**(1), 20–24 (1995)

18. Yang, R., Antony, J., Rendell, A., Robson, D., Strazdins, P.: Profiling directed NUMA optimization on Linux systems: a case study of the GAUSSIAN computational chemistry code. In: IEEE International Parallel Distributed Processing Symposium, pp. 1046–1057, May 2011

19. Yao, J.: Numatop: A tool for memory access locality characterization and analysis. https://01.org/sites/default/files/documentation/numatop_introduction_0.pdf

20. Zhao, Q., Rabbah, R., Amarasinghe, S., Rudolph, L., Wong, W.F.: Ubiquitous memory introspection. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO 2007, pp. 299–311. IEEE Computer Society, Washington, DC, USA (2007)