Memory Organisation Cartography & Analysis

David Beniamine, Youenn Corre, Damien Dosimont, Guillaume Huard

Memory Organisation Cartography & Analysis

David Beniamine, Youenn Corre, Damien Dosimont, Guillaume Huard

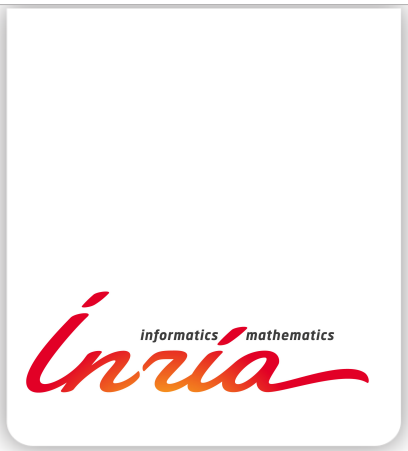
RESEARCH REPORT

N° 8694

March 2015

Project-Teams MOAIS

ISSN 0249-6399 ISRN INRIA/RR--8694--FR+ENG



Memory Organisation Cartography & Analysis

David Beniamine, Youenn Corre, Damien Dosimont, Guillaume Huard

Project-Teams MOAIS

Research Report n° 8694 — March 2015 — 16 pages

Abstract: Although performance analysis is one of the most important phase of High Performance Computing application development, analysis tools are complex to use. Most of the time, they rely on performance counters which are hard to understand for the final user. Therefore only few advanced users are able to do an efficient and precise analysis.
Moreover these counters focus on the processor while most of the performance issues are due to a bad memory usage.
In this report, we present MOCA a new kind of analysis tool which provides an overview of the memory access over time. We present MOCA's trace visualization through Ocelotl, a tool designed to provided an aggregated view of data while loosing as few information as possible. Finally, we explain how using these tools, the user can identify memory usage patterns, execution phases and how to interpret them.

Key-words: Memory Profiling, Memory analysis, HPC, MOCA, Cartography, Ocelotl, Virtual memory

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

MOCA un nouvel outil d'analyse de performances

Résumé : Bien que l'analyse de performances soit une phase importante du developpement d'applications de calcul haute performance, les outils d'analyses existants sont complexes à utiliser. Ils se basent la plupart du temps sur des compteurs de performances difficiles à interpréter pour l'utilisateur. De ce fait, seuls quelques utilisateurs aguerris sont capables de les utiliser correctement pour mener à bien une telle analyse.

De plus ces compteurs se focalisent sur les performances de processeur alors que la plupart des problèmes de performances proviennent d'une mauvaise utilisation de la mémoire.

Dans ce rapport, nous présentons MOCA un nouvel outil d'analyse qui se concentre sur la mémoire. Nous proposons une visualization des traces produites par MOCA via l'outil Ocelotl qui permet d'obtenir une vue aggregée en minimisant la perte d'informations. Finallement nous expliquant, comment ces deux outils permettent à l'utilisateur de voir les schémas d'utilisation de la mémoire, les phases d'éxecution et comment les interpréter.

Mots-clés : Profiling mémoire, Analyse mémoire, Calcul hautes performances, Cartography mémoire, Ocelotl, MOCA, Mémoire virtuelle

Contents

1 Introduction

Performance analysis is an important phase of HPC applications development. There are many tools to do this kind of analysis, however they are not trivial to use. Profiler like Oprofile [13] can be used to identify parts of the code on which we spend most of the time (hotspots). However this does not tell us if there is an issue and what kind of issue we are facing. To do so, one can looks at performance counters using PAPI [19] or Likwid [18]. These counters are based on CPU register, they give information such as the number of cycles stalled waiting for memory I/O, the number of cache faults, or idle cycles etc. They can help to understand the nature of the problem. Nevertheless, these values are hard to understand and, to interpret them, we need to combined and correlate them. Moreover, with complex applications, these tools provide too many data to be easily interpreted.

Some advanced tools, such as Vtunes [16], HPCToolkit [1] or Paraver [15], based on profiling and performance counters have been developed to help the user analysing these kind of data, yet they are complex to use and they still require the analyst to go through a lot of data to spot and understand the performance issues.

Furthermore, all these tools focus on the CPU while most performance issues come from bad memory use. Indeed the gap between memory and CPUs frequencies is not negligible: we need more than one hundred CPU cycles to do one memory access and it is still growing. To compensate this gap, complex caching mechanisms are used to benefit from these mechanisms one needs to think in terms of memory access patterns. Moreover with *Non Uniform Memory Access*, each CPU have some affinity with a range of memory addresses, and the programmer have to take this into account to write efficient code [10].

There are not many tools able to profile the memory usage as such analysis can be costly. MemProf [12] is one of these tools, it is designed to detect *NUMA* remote access patterns. It provides interesting information, however it is based on an AMD technology called Instruction Based Sampling [11] and is therefore limited to a restricted family of AMD processors.

Cruz et al. have developed SPCD [6, 5, 7], it's a tool based on a Linux kernel module able to detect inter-threads memory communications. They gather some useful information and their analysis technique is portable and efficient. However instead of displaying these informations to the programmer, they use it directly inside a runtime to bind threads which communicate a lot on the same processor.

In a previous study, we have presented HeapInfo [2], a Valgrind tool which intercepts memory access, and provides a memory cartography to the user. Although this tool provides a new form of analysis, Valgrind is based on instrumentation and not designed for HPC applications, therefore, HeapInfo was too slow to be usable. Moreover Heapinfo traces visualisation was based on pdf files generated by R, those files were too heavy to be correctly displayed by pdf viewers, and it was almost impossible to zoom on different parts of the cartography.

In this report, we present *Memory Organisation Cartography & Analysis*, a new tool based on a kernel module which provides the same kind of analysis and even more than HeapInfo with an overhead of factor ≤ 1.6. We will first present *MOCA*'s design and we will discuss its performances in section 2. Then, in section 3, we show how to visualise and interpret *MOCA*'s output through a study case.

2 MOCA

In this section we present Moca design, how to use it and we provide a small experimental validation.

2.1 Design

HeapInfo's main drawback was the cost of binary instrumentation. Although there are some faster instrumentation tools than Valgrind, such as DYNINST[3] or PIN [4]. We decided to avoid instrumentation to reduce as much as possible the analysis overhead.

MOCA is based on a Linux kernel module which periodically records some memory access by intercepting page faults (this mechanism is detailed in 2.1.1). Memory accesses are recorded independently for each *task* (Linux internal representation of threads and processes) preserving the analysed application's parallelism. All the access that are produced by one thread during one period are grouped into chunks. Therefore we know for each recorded access when did it occurs and which thread is responsible of it.

As we should not be able to keep all the recorded accesses on memory for complex application, we need to flush data during the execution. To do so, a user thread wakes up periodically and reads *MOCA*'s */proc* files. These are virtual files and reading them triggers a callback inside *MOCA* which flushes all data.

2.1.1 Implementation

To record memory access, our module intercepts page faults by using a Linux `jprobe`. As page faults do not occurs at each memory access, we periodically invalid the recorded addresses to trigger some false page faults. This technique was introduced by [8] and is represented by figure 1.

The page fault handler is *MOCA*'s central component, it needs to be efficient to limit the analysis overhead. Therefore we use hash maps to store each recorded informations such as monitored tasks, recorded access, and false page faults.

To trigger a false page fault, we need to remove the `PRESENT` flags from a *Page Table Entry*. This is easy to do, nevertheless, it is way more complex to know if we are responsible for a page fault or not. We propose two implementations to answer this question. The first one adds each *PTE* to a hash map while clearing the `PRESENT` flags. This method implies a bit of synchronisation, but is always reliable. Moreover only one thread can add false page faults while many can test if a page fault is true or not, therefore we can use an efficient reader/writer synchronisation mechanism. The second method is a hack: a *PTE* present in the page table but without the `PRESENT` flags can only happen if a page have been swapped out. Therefore, if we are sure that no page will ever be swapped out in our application, we can avoid using the hash map. Obviously this hack can only be used if we have way more memory than what our application needs as the module needs to store a lot of data in memory. If the application did swap, Linux will kill the process as soon as it tries to access a swapped page. Therefore we recommend to use the first mechanism unless you really know what you are doing. The cost of these implementations will be discussed in section 2.2.

When the monitoring thread wakes up, it ends the current chunk of each task, and mark all accessed page as absent to trigger a page fault on the next access.

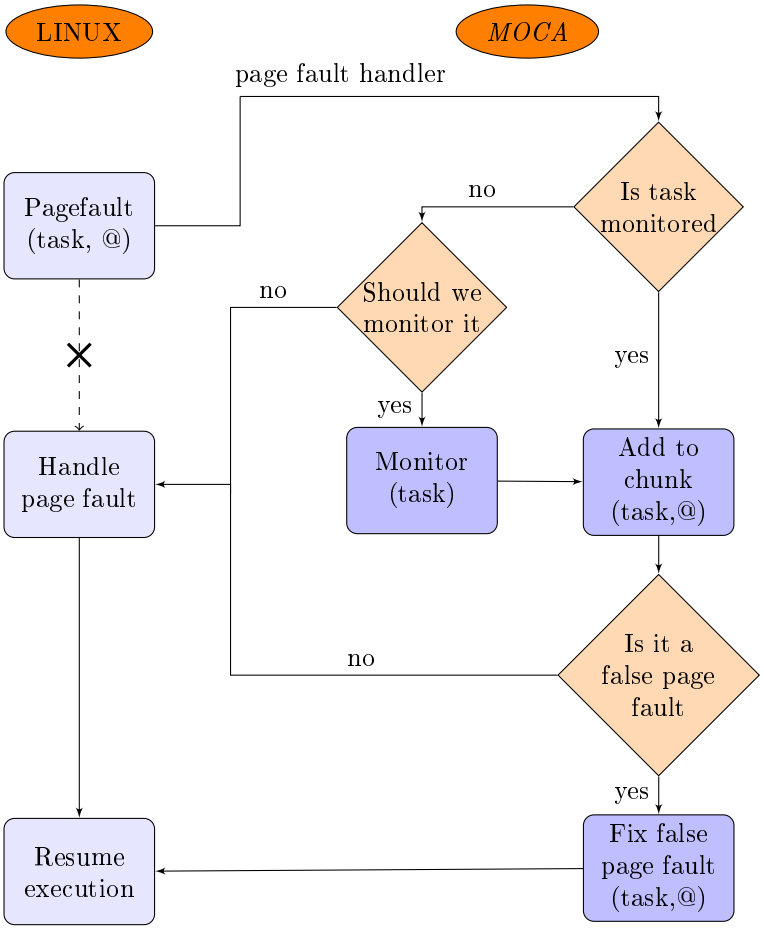


Figure 1: Flow chart of *MOCA*'s page faults

2.1.2 Usage

MOCA kernel module is not designed to be loaded manually, a script is provided to launch it. One can monitor an application using the command:

```
moca -c "my_command" -a "my_arguments"
```

This script makes sure that *MOCA* is correctly loaded and ready to monitor an application before starting it. The main idea of this script is described in the algorithm 1

MOCA will create a directory containing the application output, its output and the trace files. By default *MOCA* traces are saved in files named `Moca-taskX`, where X is the ID of the task owning the file (the ids starts at 0 and are attributed by creation order).

All of theses files starts with a Line giving the internalID and the system processID:

```
T internalId ProcessId
```

The Task0's first line ends with the page_size of the machine.

The second line is always the beginning of a chunk:

```
C id N start end cpumask
```

Algorithm 1 *MOCA*'s launcher algorithm

```
if pid = 0 then
    kill SIGSTP $$                                  # Child process
    exec cmd args
else
    modprobe moca main_pid=$pid                     # Parent process
    kill SIGCONT $pid
end if
```

These lines contains the chunk id which is unique for a task, the number N of accesses in the chunk, the chunk's beginning and end timestamp, and a bitmask telling which processors have accessed to this chunk.

Each chunk line is followed by N access lines:

```
A @Virt @Phy countread countwrite cpumas
```

An access line corresponds to one page, it gives the virtual and physical addresses of the page, the number of read and writes observed and a cpumask telling which processors are responsible of this access. *Please note that currently the physical page addresses detection is experimental and may not be reliable.*

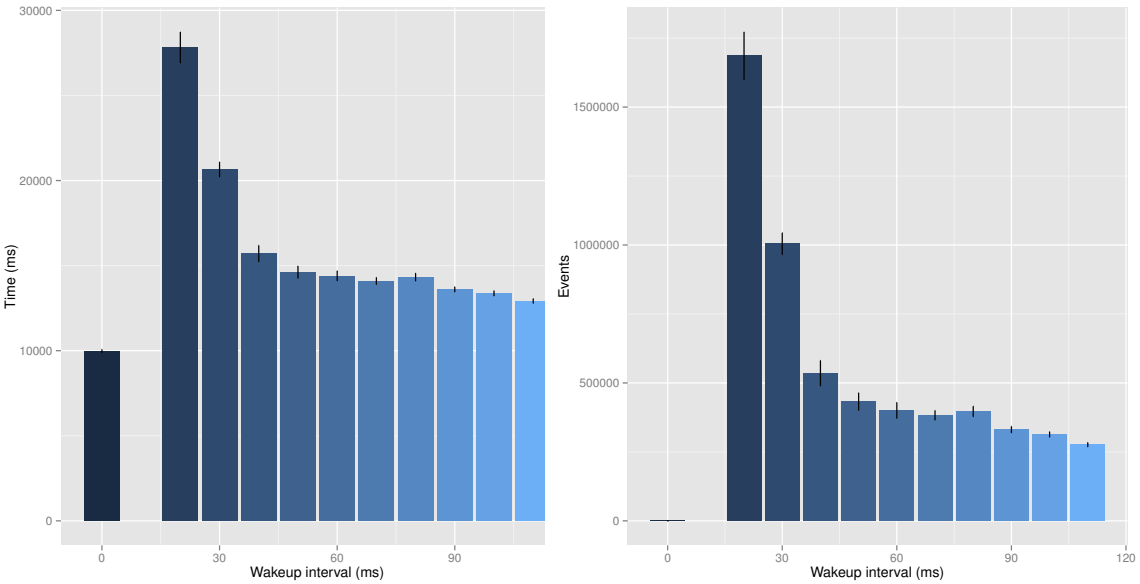
MOCA provides many parameters to do fine tuning aiming at speeding up the analysis, among them: the monitor thread's wakeup interval (its impact is discussed in 2.2), the number of chunks stored in memory and the number of addresses that we can store in a chunk.

2.2 Performances

In this section, we evaluate the impact of *MOCA*'s on the execution time of a parallel matrix multiplication. All experiments ran on a virtual machine made with kameleon[17], its recipe is available here `http://moais.imag.fr/membres/david.beniamine/kameleon/deb_virt_dev_RR_MOCA.tar.gz`. The host machine is composed of a 6 core Intel Xeon E5-1607 with $16G$ of RAM, its operating system is a Debian Wheezy. The VM consists of a Debian Jessie running on 2 cores identical to the host and with $4G$ of RAM. Each point is the result of 30 different runs.

In our first experiment, we try to evaluate the impact of the wakeup interval, as explain earlier it should be one of the most important settings. Figure 2a presents the effect of this setting on the execution time. A wakeup interval of 0 ms means an execution without monitoring. We can see that when the interval is too small (≤ 30 ms), the execution time grows exponentially. This can be explained by the fact that every 50 ms, Linux scheduler wakes up and chooses the next process to execute, therefore if we monitor the application more often than that, we don't give enough time to the application to actually do its computations. Nevertheless, when the interval is ≥ 40 ms, the overhead goes down and stabilises between 1.6 to 1.3 times the original execution time. Although this overhead is not negligible, it is a reasonable cost for obtaining memory access information, and it is a great improvement compared to *HeapInfo*.

Figure 2b shows the number of events (memory access) captured for each value of the wakeup interval. For small wakeup intervals, as we spend most of the time in our tools, almost each memory access triggers a false page fault. Therefore the number of captured events is high for these values($\geq 10^6$). This also explains why the monitoring takes so much time for wakeup intervals ≤ 30 ms. On the other hand, when the wakeup interval is too high (≥ 80 ms), we miss a lot of events.



(a) Execution time.

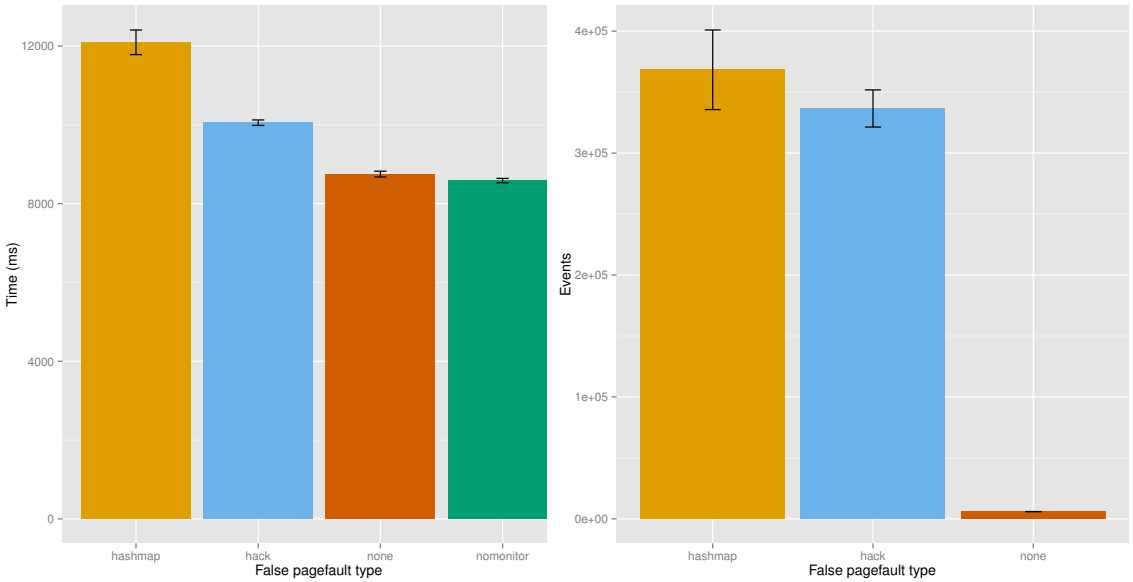
(b) Number of captured events

Figure 2: Impact of the monitor thread's wakeup interval on a parallel matrix multiplication of 1000^2 doubles. A wakeup interval of 0 *ms* means no monitoring.

The two plots of figure 2 show clearly that we can reach a nice trade-off between performances and precision of the trace with a wakeup interval between 40 and 60 *ms*. As this interval is synchronised with the Linux scheduler interval, we should obtain similar results with other applications.

The second experiment shows the impact of the false page faults. We compare the two page faults mechanisms previously presented (*hashmap* and *hack*) with a normal run out of *MOCA* (*nomonitor*) and a monitoring run without false page faults (*none*). Figure 3b shows the number of events captured with both implementations and without using false page faults. It clearly shows that the mechanism is required for a good analysis. The number of event captured by the two implementations is comparable.

Finally figure 3a shows the impact of the false page faults mechanism on the overall execution time. When we disable the false page faults, the execution time is comparable to a normal execution. However we saw with the previous figure that it is not a good idea to do so. As expected, the *hack* false page faults is the faster implementation as it does not require any synchronisation. Nevertheless, the *hashmap* method is only 1.2 times slower and is always reliable. Furthermore if we are monitoring an application slow enough to make this difference not negligible, it is highly probable that your application uses enough memory to swap when monitored. Therefore we recommend to use the *hashmap* mechanism except if you really know what you are doing.



(a) Execution time

(b) Number of captured events.

Figure 3: Impact of the false page faults mechanism on a parallel matrix multiplication of 1000^2 doubles.

3 A study case

In this section we will first explain how the visualisation of *MOCA* traces works, then we will present an example analysis.

3.1 Visualisation

One of Heapinfo's drawbacks was the visualisation: the cartography were outputted as pdf files. When the traces became too large, there were more information than pixels, the traces were messy and hard to interpret. Furthermore, it was almost impossible to zoom-in on different parts or to focus on some type of events.

MOCA's output is designed to be imported inside the trace visualisation framework *Framesoc* [14], and to be visualised using *Ocelotl*[9]. This tool uses an adaptive algorithm to aggregate data. This algorithm returns a set of results which provides a trade-off between the information loss and the reduction of the visualisation complexity. The user has then the possibility to move the cursor between a precise view of the trace or something more aggregated. Moreover it provides the ability to navigate through the trace, focus on one type of event or another.

We provide two different views of *MOCA*'s traces. The first, is a *memory gantt chart*, it shows memory usage of each threads, as we can see in figure 4. This view can be used to spot difference in the memory usage in terms of quantity or pattern. The second, quite similar to HeapInfo's cartography, displays the memory addresses accessed depending on the time, as we can see in figure 5. It allows the user to focus on the global memory pattern. We provide both views for physical and virtual addresses. Most of the time virtual addresses are recommended to have a better understanding of what the code is doing. However, working on physical addresses can be useful for *NUMA* machines to detect remote access. We will explain how to use these

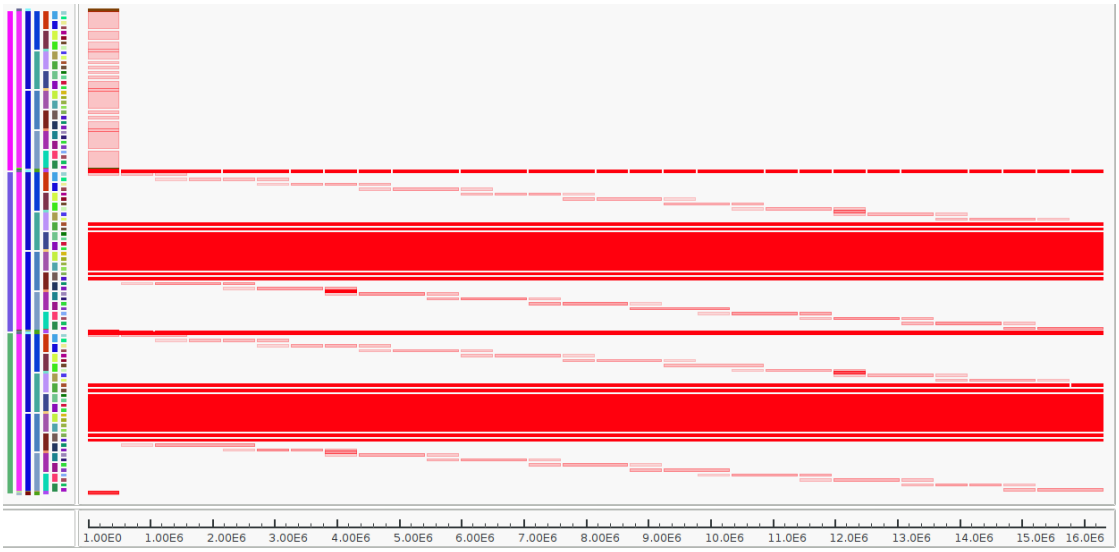


Figure 4: Memory-gantt view of a parallel matrix multiplication.

views in the next section.

On both figures, we can notice that the memory hierarchy (on the left) is a tree. For the first view, the top level correspond to the threads/ processes. In figure 4 we can clearly identify 3 threads. The second level is created by merging together successive set of addresses. Every other level is computed by cutting the previous into two or three parts. The last level corresponds to the pages address detected by *MOCA*. Although this hierarchy is a bit artificial it shows the different parts of the memory (stack, heap, library, etc.). Furthermore, it can be used to compute a pre aggregation which can speed *Ocelotl* up. For the cartography view, the tree is identical, except that the first level is the second one of the memory gantt view.

3.2 Example on a matrix multiplication

In this section, we show an example use of *MOCA*. The studied application is a parallel matrix multiplication done by 2 threads. The algorithm is quite naive: a master thread initializes the matrix then creates two threads. Each thread will compute half of the result matrix: the first will do the even index and the second the odds, which gives us the algorithm 2.

Algorithm 2 Matrix multiplication thread algorithm

```
for l=0; l<sz; l++ do
  for c=myid(); c<sz; c+=NbThreads() do
    for k=0; k<sz; k++ do
      Res[l][c]=A[l][k]*B[k][c]
    end for
  end for
end for
```

If we go back to figure 4, we see clearly the master/slave behaviour of the threads. While the two slave threads have a similar memory access pattern, the master threads seems to do all its

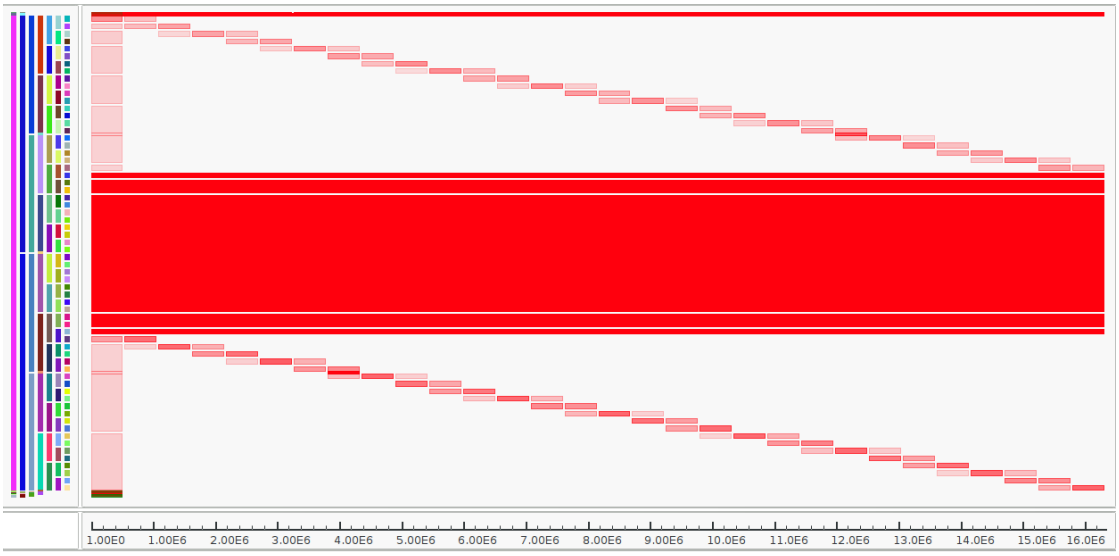


Figure 5: Cartography view of a parallel matrix multiplication

access at the beginning of the execution. Moreover it seems to access all the memory at once. We can also see that it does less access at a time as the blocks are lighter. From this picture, we can clearly identify two phases:

1. The three threads seem to use the memory, it is the initialisation phase.
2. Only two threads are awake, they follow the same access pattern: it is the computational phase.

Now let's zoom on the initialisation step. The result is shown in figure 6. We can clearly see that, contrary to what we thought on the last picture, during the initialisation phase, only the master thread is working. We can identify a three diagonal patterns happening at the same time, it correspond to the matrix initialisation. Moreover we see a few green access, while all the other are reds. Here a red access mean that it is on a page used by several threads, while green is for private data. Therefore we can see that the master thread also access to some private data. Among these data, we can find a pid array used to wait the end of the slave threads.

The cartography view of this initialisation phase, is almost identical except that we won't be able to know which thread is responsible for the accesses. However the global cartography view shown in figure 5 gives more information. In addition to the two execution phases, we can clearly identify three memory structures with different access patterns. Those structures correspond to the matrices. For the first and the third, we can see a regular diagonal access pattern which means that the matrix are accessed linearly. As most cache optimizations such as *prefetching* are designed for linear access, his is a good pattern.

By focusing on the middle of the execution and setting the aggregation to 0, we obtain the figure 7. We can't identify a clear pattern on the middle matrix, however, we see that at each time slots we access more a slightly different part of the matrix (the lighter a block is, the less access it contains). The access on this matrix seems dense and not designed to fit in a cache. Now, if we take a look at the algorithm 2, we can explain the density of the accesses. We go through all the matrix B while working on only one line of A. Moreover the two threads works on two different columns at the same time, while the work on the same lines of A and Res. Due

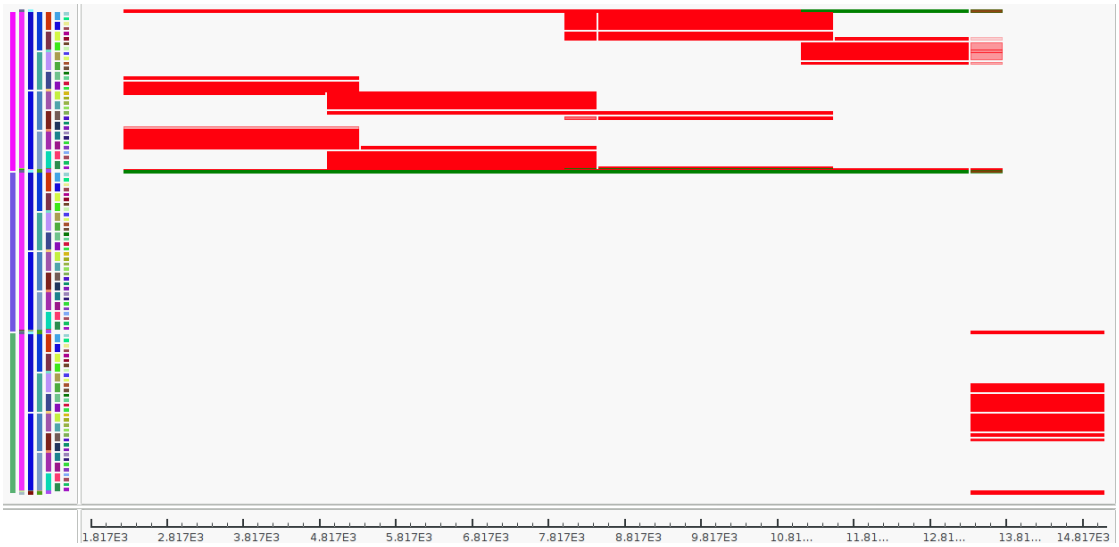


Figure 6: Memory-gantt view of a paralell matrix multiplication, initialisation

to the representation of 2D matrix in `C`, each access on B is separated from sz doubles. Hence *Ocelotl*groups almost everything in a huge chunks of access on all the matrices. To improve this application, we should try to work on small blocks of B. This is indeed the strategy used to compute efficient matrix multiplication. Although this example is quite simple, it shows how *MOCA*'s trace can easily highlight wrong memory access patterns.

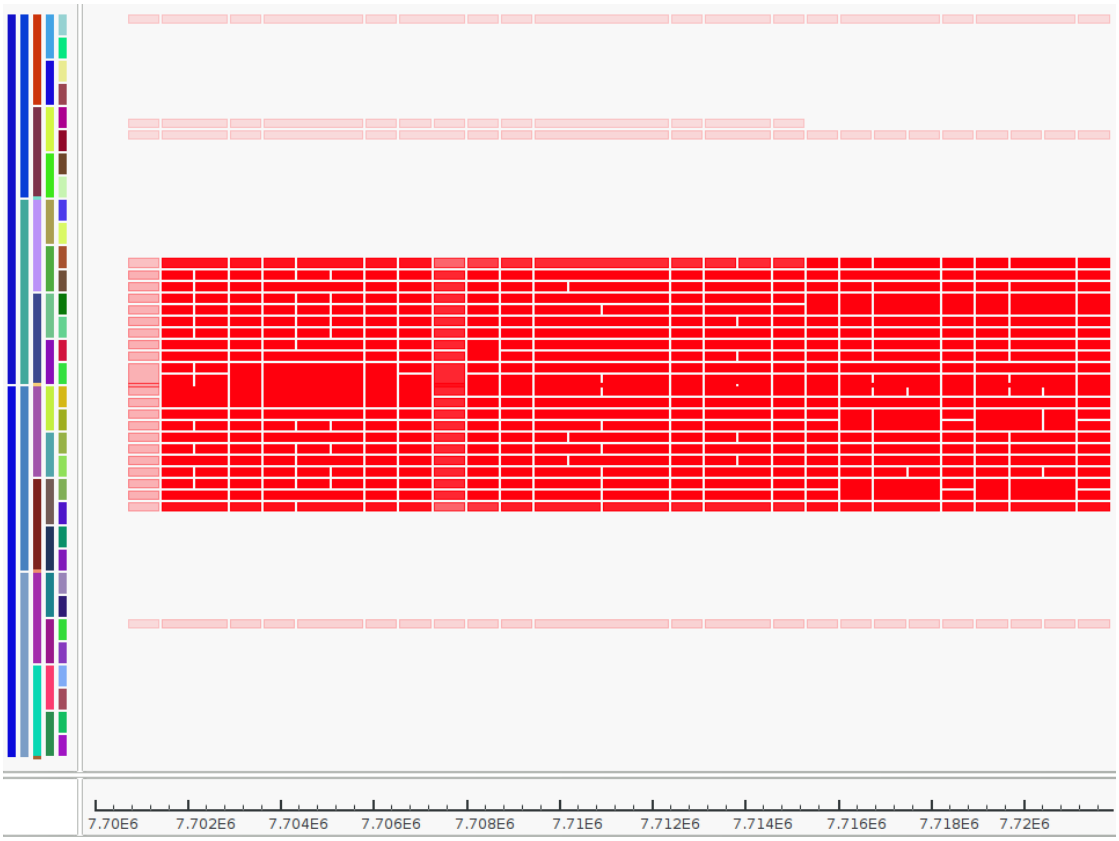


Figure 7: Cartography view of a parallel matrix multiplication, computing phase.

4 Conclusions

Performance analysis is one of the most important steps while writing high performance code. However it is a complex task and many analysis tools require a deep understanding of the machine to use them. Moreover while most performance issues come from bad memory usage, usual tools rely on CPU based metrics. They do so because analysing a dense memory traffic can be costly and provides huge traces potentially hard to understand.

In this report, we have introduced *MOCA* a new kind of analysis tool which focuses on the memory. We showed that *MOCA* has an acceptable overhead, one can expect to do an analysis in 1.3 to 1.6 times the normal execution time. We have also evaluated the impact of the different components and settings of *MOCA* in terms of execution time and analysis precision.

We have shown through a study case how *MOCA* can be used to detect memory usage patterns and execution phases. Furthermore, we have shown how to identify potentially problematic parts of code.

The next step will be to monitor and optimise real applications on *NUMA* machines and to compare and/or combine our analysis with the result of classic tools.

References

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[2] David Beniamine. Cartographier la mémoire virtuelle d'une application de calcul scientifique. In *ComPAS'2013 / RenPar'21*, Grenoble, France, 2013.

[3] Andrew R. Bernat and Barton P. Miller. Anywhere, Any-time Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, pages 9–16, New York, NY, USA, 2011. ACM.

[4] Kristof Beyls and Erik D'Hollander. Reuse Distance as a Metric for Cache Behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*, volume 14, pages 617–622, 2001.

[5] Eduardo H. M. Cruz, Matthias Diener, Marco A. Z. Alves, and Philippe O. A. Navaux. Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols . *Journal of Parallel and Distributed Computing* , 74(3):2215–2228, 2014.

[6] M. Diener, E. H. M. Cruz, and P. O. A. Navaux. Using the Translation Lookaside Buffer to Map Threads in Parallel Applications Based on Shared Memory. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 532–543, May 2012.

[7] M. Diener, E. H. M. Cruz, and P. O. A. Navaux. Communication-Based Mapping Using Shared Pages. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 700–711, May 2013.

[8] M. Diener, F. L. Madruga, E. R. Rodrigues, M. A. Z. Alves, J. Schneider, P. O. A. Navaux, and H. U. Heiss. Evaluating Thread Placement Based on Memory Access Patterns for Multicore Processors. In *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*, pages 491–496, Sept 2010.

[9] Damien Dosimont, Lucas Mello Schnorr, Guillaume Huard, and Jean-Marc Vincent. A Trace Macroscopic Description based on Time Aggregation. Technical Report RR-8524, Apr 2014. Trace visualization; trace analysis; trace overview; time aggregation; parallel systems; embedded systems; information theory; scientific computation; multimedia application; debugging; optimization.

[10] Ulrich Drepper. *What every programmer should know about memory.* Red Hat, 2007.

[11] Paul J. Drongowski. Instruction-based sampling: A new performance analysis technique for AMD family 10h processors. Technical report, AMD CodeAnalyst Project, Boston Design center, November 2007.

[12] Renaud Lachaize, Baptiste Lepers, and Vivien Quema. MemProf: A Memory Profiler for NUMA Multicore Systems. In *USENIX 2012 Annual Technical Conference (USENIX ATC 12)*, pages 53–64, Boston, MA, 2012. USENIX.

[13] John Levon. Oprofile Manual. Victoria University of Manchester,, 2000.

[14] G. Pagano, D. Dosimont, G. Huard, V. Marangozova-Martin, and J. M. Vincent. Trace Management and Analysis for Embedded Systems. In *Embedded Multicore Socs (MCSoC), 2013 IEEE 7th International Symposium on*, pages 119–122, Sept 2013.

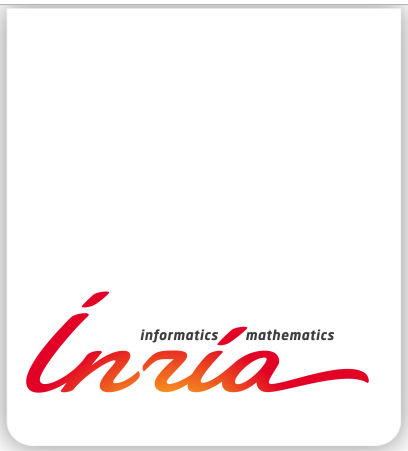
[15] V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVER: A Tool to Visualize and Analyze Parallel Code. In Patrick Nixon, editor, *Proceedings of WoTUG-18: Transputer and occam Developments*, pages 17–31, mar 1995.

[16] James Reinders. *VTune performance analyzer essentials*. Intel Press, 2005.

[17] Cristian Ruiz, Salem Harrache, Michael Mercier, and Olivier Richard. Reconstructable Software Appliances with Kameleon. *SIGOPS Oper. Syst. Rev.*, 49(1):80–89, jan 2015.

[18] J. Treibig, G. Hager, and G. Wellein. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010.

[19] V. M. Weaver, D. Terpstra, H. McCraw, M. Johnson, K. Kasichayanula, J. Ralph, J. Nelson, P. Mucci, T. Mohan, and S. Moore. PAPI 5: Measuring power, energy, and the cloud. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 124–125, April 2013.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399