UNIVERSITÉ GRENOBLE - ALPES

NANO2017 DEMA
SP 1—Interactive Debugging

Livrable D4 :
Version 2.0 du debugger de performance

Kevin Pouget, Jean-François Méhaut
UGA-LIG/INRIA CORSE
January 30, 2017

Contents

Chapter 1

Introduction

In this report, we present the second part of the performance debugger development, done during the second semester of 2016. It continues the work started in the first semester and presented in the NANO2017/DEMA Deliverable D2.

In this second stage of the work, we focused on the multithreaded aspect of modern applications. We paid a particular attention to the OpenMP programming environment [2], which was already the target of the previous deliverables of the project: in the Deliverable D1 and D3, we studied how GDB, the free debugger of the GNU project [4], would be extended to support OpenMP programming model. The target of these deliverables was functional debugging, with the improvement of the control of the multithread execution, as the ease of understanding of the current application state.

In this deliverable, we pursued the work started in Deliverable D2 on the design of a interactive profiling tool based on a source level debugger. We extended our tool and combined the part aware of the OpenMP programming model with the part dealing with the interactive profiling.

In the following of this report, we first detail the features we implemented for OpenMP profiling (Chapter 2). Then, we introduce some additional features that were developed to help developers while doing performance debugging of a Numa machine (Chapter 3). Finally, we present an illustrated methodology for the the debugging of a Numa-related performance problem (Chapter 4).

Chapter 2

OpenMP Profiling

In the deliverable D2, we improved GDB to support interactive execution profiling. In this first part of the work, we only considered plain C codes, and function-level profilings. In the second part, we turn towards more realistic applications and focus on OpenMP programming environment.

OpenMP programming was already the topic of concern of Deliverables D1 and D3, where we extended mcGDB, our programming-model-centric debugger, to improve the functional debugging of OpenMP applications.

and worked on programming-model centric performance debugging for OpenMP.

In this chapter, we introduce how we we join these two aspects by interconnecting the capture module of our model-centric debugger (D1/D3) and the interactive profiler built as part of D2. Then we detail the new fine-grained profiling capabilities we developed.

2.1. HOOKING THE MODEL-CENTRIC DEBUGGER

As part of the D1/D3 deliverables and the previous work on model-centric debugging [5], we built an extensible debugger environment to hook OpenMP operations. Figure 2.1.1 gives an overview of this organization:

capture — this module is dependent of the OpenMP implementation. It is in charge of capturing the operations and updating the OpenMP framework state, usually through breakpoints and readings of the OpenMP function parameters. It passes this information to the **representation** module.

representation — this implementation-independent module keeps track of the state of the OpenMP execution. Its updates are triggered by the **capture** module.

interaction — this module is in charge of the interactions with the user, either through GDB command-line or a custom graphical interface (*eg*, Temanejo).

Additionally, as we presented in D3, Section 7.2 *Aspect-Oriented Programming for **interaction** Modules*, the **representation** module can be hooked (with *aspects*)

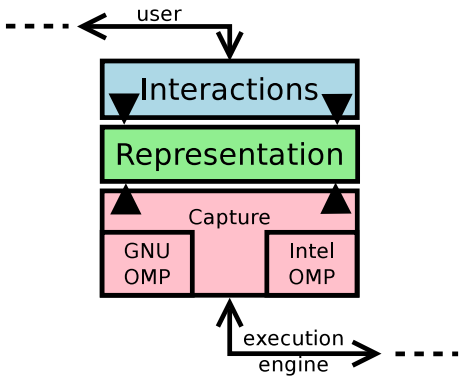

Figure 2.1.1: Organization of the model-centric debugger OpenMP module

to developed more reactive interaction modules. This is the feature we used to interconnect our profiling mechanisms with OpenMP events.

As part of the case-study described in Chapter 4, we put a particular attention to the profiling of OpenMP static loops. These loops are executed in parallel by the workers of the current team. The loop iteration space is distributed over the workers in a static manner, defined by the OpenMP specification. The `pragma` for this construct is structured as follows:

```
#pragma omp for
for (<loop counter start>; <end-of-loop condition>; <increment>)
  <loop body block>
```

We first updated the Intel OpenMP **capture** module to capture OpenMP parallel static loops (see the source code in `model/task/environment/openmp/capture/iomp/kmpc_for_static.py`).

In this module, we set breakpoints on the Intel OpenMP [1] functions in charge of the static loop. When these breakpoints are triggered, we parse the functions parameters to find out the loop upper and lower bounds, its increment and the source-code location. We also capture the OpenMP thread iteration space.

This information is passed to the **representation** module's `ForLoopJob` class, which represents an OpenMP parallel loop. Its main methods are the following:

- `__init__(self, loc, lower, upper, incr)`
- `start_work_on(self, worker, lower, upper)`
- `stop_work_of(self, worker)`

They are called by the **capture** module to instantiate the loop and indicate when the different threads start and stop the loop execution.

To implement the OpenMP loop profiling, we wrote aspects for these functions, that trigger the beginning and end of a profiling region (see `model/task/`

`environment/openmp/interaction/loop.py`). This fine-grained loop profiling is further described in the following section.

The hooking mechanism we have presented in this section could be easily extended to other aspects of OpenMP. For instance, the debugger is already aware of the OpenMP tasks, so implementing task-level profiling would be trivial.

2.2. FINE-GRAINED OPENMP STATIC LOOP PROFILING

In the previous section, we have presented how we hooked mcGDB breakpoints with the interactive profiler. To control this profiling, we implemented a single command-line function:

```
(gdb) omp loop profile iterations
```

which installs the loop-specific breakpoints and activates a profiling flag. After that and when the execution encounters an OpenMP static loop, the debugger callbacks trigger the beginning of a profiling region. At the end of the loop-chunk execution, another callback stops it.

In the work of this deliverable, we mainly used the `perf stat` profiling module. Modern versions of `perf` have the ability to profile the execution thread by thread, however the version installed on our NUMA machine (discussed in the following chapter — Chapter 3) is rather old and does not support it. So we chose to force a single-threaded execution during the profiling period, with GDB's `scheduler-locking` parameter: at the beginning of the loop-chunk execution, we activate the scheduler-locking and start the profiling, and at the end we disable it.

Neither GDB nor the scheduler-locking alter the CPU mapping, so this profiling generates accurate measurements. The drawback, however, is the time overhead in the full-loop profiling.

In Deliverable 2, Section 11.2 *Presenting the Measurements*, we detailed how these profiling measures could be plotted. The illustration used in this section (Figure 2.2.1) corresponds to a loop profiling. It is important to note that in this plotting, a *column* corresponds to one loop-chunk execution profile. The recording order is not deterministic. It reflects the reality of the execution, but does not really have sense *per-se*. Hence, we recommend the use of the sorting modifier (`<`) to force a more meaningful order.

During OpenMP loop profiling, the interactive profiler can also integrate OpenMP-specific information in the profiling results. As an example, we added the thread loop counter start (`omp_loop_start`) and length (`omp_loop_len`). The start counter can be used to sort the measurements (`<` modifier), and the length counter can be used to plot only the records with the same chunk size (`@2` modifiers for length 2):

```
omp_loop_start | 98 293 387 290 203 385 194 299 435 56 ... <
omp_loop_len | 3 3 2 3 3 2 3 3 2 3 2 2 3 3 3 3 2 3 3   ... @2
instructions | 118342470 118337189 78900017 118343626  ...

| Rendering chart plot of "omp_loop_start (sorted), omp_loop_len,
| instructions" into /tmp/chart-20170117-155044.png
```

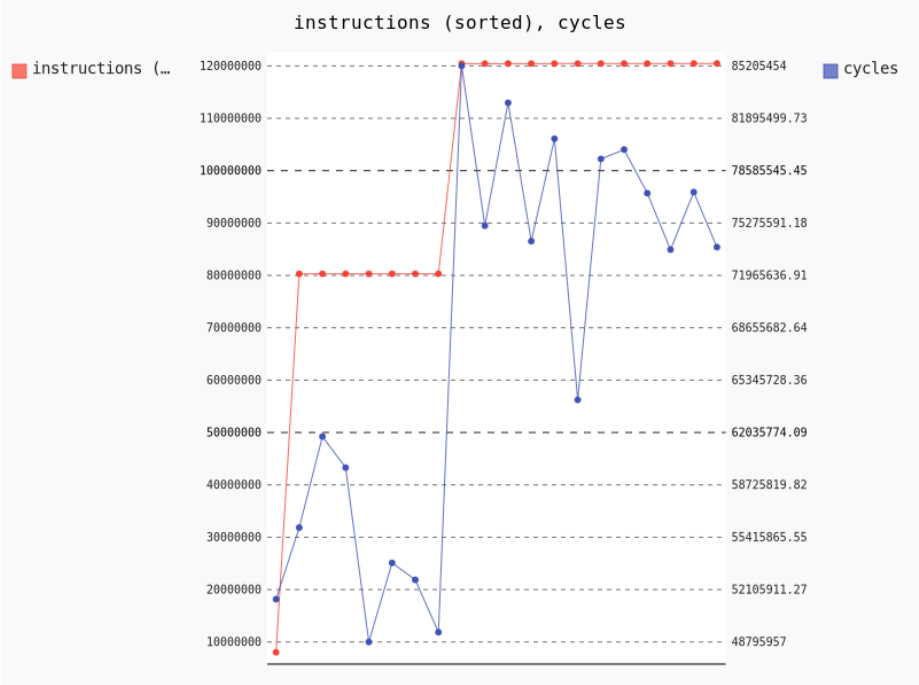

Figure 2.2.1: Example of a multi-profile plot, showing the instruction count and number of CPU cycles, sorted over the number of instructions.

In the following chapter, we detail how we introduce NUMA-specific debugging and profiling capabilities, before going more in detail into a performance debugging case-study in the last chapter.

Chapter 3

Numa-aware Profiling and Debugging

In the previous chapter, we have shown how to interactively profile OpenMP loops. However, this is still too limited to perform the real-life performance debugging case-study of Chapter 4.

In this chapter, we put the focus on NUMA-aware profiling and debugging. Our developments were done on the NUMA machine `idchire` owned by the Laboratoire d'Informatique de Grenoble. This machine has 24 Intel Xeon E5-4640 processors of 8 cores (for a total of 192 cores). Each processor has 32GB of local (fast) memory; it can access the remaining 724GB, but slower. Typical problems that occur on such machines is frequent memory accesses on remote memory slots. In these cases, developers should consider moving the data on the processor using it ... but it is not always feasible.

In the following, we introduce the additional features we implemented in our interactive profiling and debugging tool. These features aim at providing developers with a better understanding of the current state of the NUMA execution (Section 3.1), additional profiling information (Section 3.2) and possibilities to alter the state of the NUMA machine to test debugging hypotheses (Section 3.3).

3.1. NUMA-State Understanding

Natively, GDB does not provide any information related to the hardware topology of the machine. But for NUMA architectures, it is important to know where threads are running, and where memory cells are physically stored. If both are located on the same processor node, then the data accesses will be efficient, otherwise they will be costly.

```
(gdb) numa current_node
(gdb) numa current_core
```

These commands indicateon which core and node the current thread is running. This information is computed with the help of the kernel, which exposes in `/proc/$PID/task/$LWP_ID/status` the cores on which the thread (LWP) can run (field `cpus_allowed_list`). If the thread has been pined to a given core (by OpenMP

for instance), a single core id will be written. If the thread is not pined, it will be a range.

Once the core id is known, the `/proc/cpuinfo` kernel file indicates the mapping between the core (field `processor`) and its processor node (field "physical id").

An alternative way to find out the node and core id is to call functions `sched_get cpu` (`libc`) and `numa_node_of_cpu` (`libnuma`) within the process address-space, in the right thread. It gives the same answers, however this second solution is more intrusive and may cause unexpected behaviors in GDB usage (`Control`$\hat{\text{C}}$ may interrupt these internal functions calls, instead of the normal process execution), so it is not recommended.

```
(gdb) numa pagemap
```

Once the current thread has been located on the machine, developers will want to locate their memory cells. This information is also exported by the kernel, though the `/sys/devices/system` files. It is however harder to retrieve, so we reused a tool called `pagemap`[1], developed earlier in the team.

`Pagemap` takes a PID and a memory address as parameter, and returns the processor node on which the memory page is located. This is exactly the information we wanted to provide.

We wrote a GDB command wrapper around this program, and added GDB's ability to compute a memory address with the language syntax:

```
(gdb) numa pagemap &my_var
| Address 0x7fffffffda14 is located on node N18
```

3.2. NEW PROFILING COUNTERS

Similarly to what we described for OpenMP profiling counters in Section 2.2, we integrated NUMA information in the profiler counters.

At the beginning of the profiling regions, the commands `numa current_node` and `numa current_core` are called to record which core executes the region. These counters are then provided as profiler measurements, like the other ones.

The core id information can helps understanding NUMA-related performance problems, as we describe in the following chapter. It is also useful to specify the order of the region plots, with the < modifier (see Deliverable D2, Section 11.2). For instance in OpenMP loop profiling, the order of the profiles is set by the earliest chunk-execution starts. This order has no particular meaning, it is system and implementation dependent. Thus, the core id offers a better ordering.

We also experimented an address-to-processor lookup (`numa pagemap`) at the beginning of OpenMP loops (by recording the location of the array cells accessed during the chunk execution), but in its naive form it adds a significant time overhead. More efficient implementations could be designed in the future, such as not launching the `pagemap` process at each lookup, or even re-implementing it inside GDB/Python.

[1] https://forge.imag.fr/projects/pagemap/

3.3. DEBUGGING HYPOTHESES TESTING

Finally, in order to let developers test debugging hypotheses on-the-fly, we developed new debugging commands that change the property of the NUMA execution. They all rely on the `move_pages (2)` function of the `libnuma`. We wrote a lightweight wrapper around this function to drive it from GDB/Python, see `model/numa/move_page.c`.

`numa spread_pages`

This command looks up up the current boundaries of process' heap (found in `/proc/$PID/maps`) and spreads it across all the machine's nodes, page per page, on a round-robin fashion.

`numa spread_3D_matrix mat sz_x sz_y sz_z sz_elt`

This command was tailored to the hypothesis we wanted to test in Chapter 4, but its design can inspire more general-purpose commands. It takes as parameters a 3-dimensional matrix `mat`, the size of the three dimensions `x, y` and `z`, as well as the size of an element (eg, `sizeof(double)`.

It then traverses the matrix and spreads its pages across the NUMA nodes. The spread is done according to the OpenMP static loop distribution scheme: the cells accessed by threads 1-7 are moved to node 1, those of threads 8-15 go to node 2, *etc.*

In the following chapter, we present the case-study of a performance debugging, assisted by an interactive debugger. In this debugging workflow, we illustrate the usage of the different commands we presented in this chapter and in the previous one.

Chapter 4

An Illustrated Interactive Performance Debugging Workflow

In this chapter, we present and illustrate an the workflow of an interactive performance debugging, based on the tools and ideas developed as part of this deliverable. This case-study is inspired from the work of Drebes *et al.* [3], carried out as part of NANO2017/DEMA sub-project 2.

The case-study is based on the OpenMP implementation of MG class C [6], from the NAS Parallel Benchmarks specification suite. MG runs five kernels on a "Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive". The C class corresponds to "standard test problems".

As for the previous chapter, the study was done on the NUMA machine `idchire` owned by the Laboratoire d'Informatique de Grenoble. This machine has 24 Intel Xeon E5-4640 processors of 8 cores (for a total of 192 cores). Each processor has a 32GB of local (fast) memory; it can access the remaining 724GB, but slower.

4.1. ATTESTING THE PERFORMANCE PROBLEM

The initial step of the debugging consists in attesting the performance problem. To that purpose, we first run `MG.C` with Aftermath tracer and visualize the output. Figure 4.1.1 shows Aftermath OpenMP overview, where the green strips corresponds to OpenMP parallel for loops. As the pattern is repeated on all the loop executions, we take one randomly. I corresponds to the first lines of function `resid`:

```
#pragma omp for
    for (i3 = 1; i3 < n3-1; i3++) {
        for (i2 = 1; i2 < n2-1; i2++) {
            for (i1 = 0; i1 < n1; i1++) {
                u1[i1] = u[i3][i2-1][i1] + u[i3][i2+1][i1]
                       + u[i3-1][i2][i1] + u[i3+1][i2][i1];
                u2[i1] = u[i3-1][i2-1][i1] + u[i3-1][i2+1][i1]
                       + u[i3+1][i2-1][i1] + u[i3+1][i2+1][i1];
            }
```

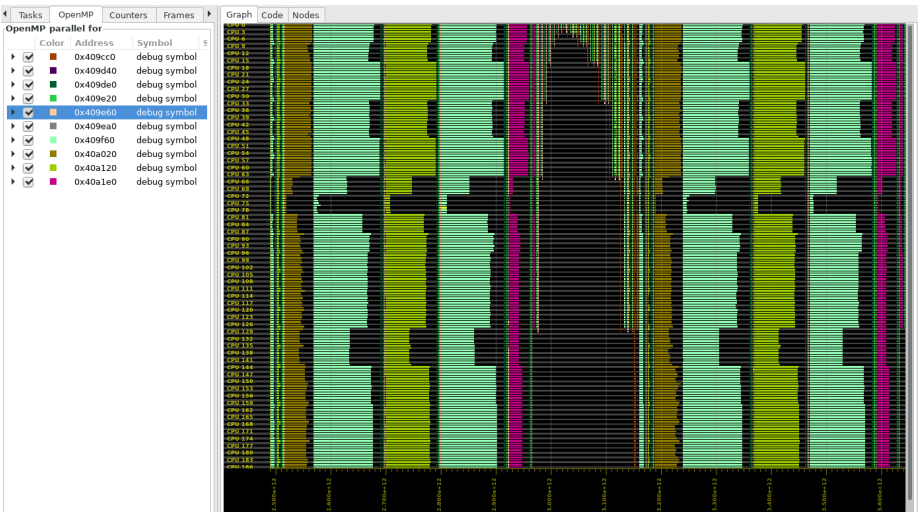

Figure 4.1.1: Attesting the Performance Problem with Aftermath: each strip (static parallel loops) should have the same time length.

```
            for (i1 = 1; i1 < n1-1; i1++) {
                r[i3][i2][i1] = v[i3][i2][i1]
                    - a[0] * u[i3][i2][i1]
                    - a[2] * (u2[i1] + u1[i1-1] + u1[i1+1])
                    - a[3] * (u2[i1-1] + u2[i1+1]);
            }
        }
    }
```

A quick look at this function suggests that all the iterations should take the same time, as the loop code does not dependent of the data content (there is no flow-control statement).

We can confirm this assumption with a loop-chunk profiling in mcGDB, by profiling the instruction count of each loop chunk execution:

```
(gdb) mcgdb load_model_by_name om
(gdb) tb resid
(gdb) run
...
(gdb) omp loop profile iterations
(gdb) set profile-perf-counters instructions
(gdb) continue
... ^C
(gdb) profile graph plot-all all
(gdb) profile graph offline
numa core | .......  <
```

```
omp_loop_len | ....
instructions | .... y2
```

Figure 4.1.2 shows the plot generated with these commands. The profiling was manually interrupted after 51 loop-chunk executions. In the plot, the profilings are sorted according to their `numa node` value. We can see that this serie is increasing, but not regularly. This is because not all the profiles have been recorded yet. The serie `omp loop len` indicates the length of the OpenMP chunk executed in the profile. It is either 3 or 2. Finally, the `instruction` serie indicates the number of instructions executed during the chunk. We can see two levels, corresponding exactly to the loop lengths.

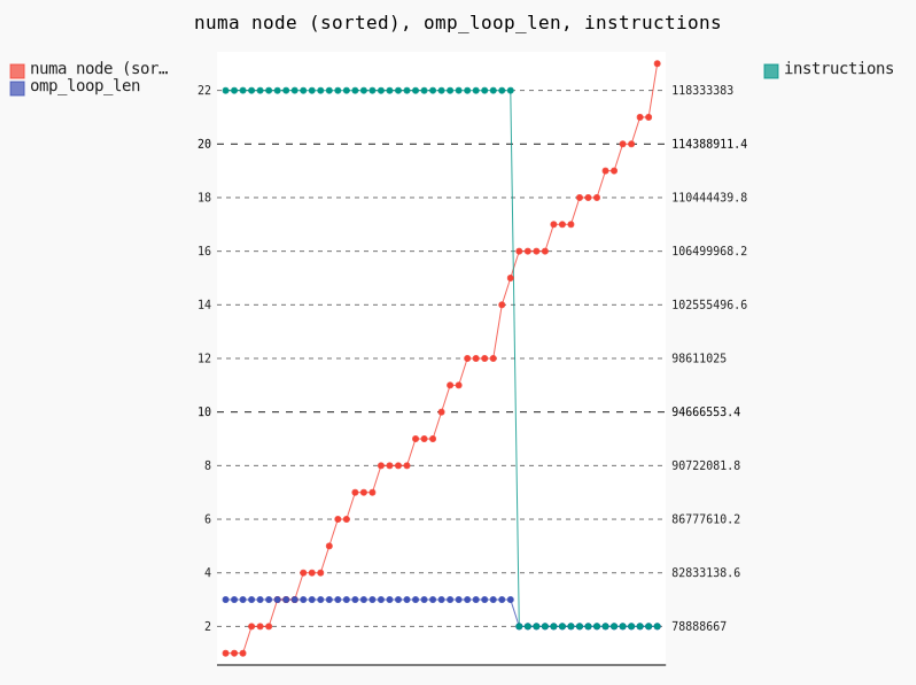

Figure 4.1.2: Attesting the Performance Problem with mcGDB: for a given loop-chunk length, all the executions have the same instruction count

At this stage, developers know that there is a data access problem.

Pinpointing the NUMA data-access problem

To better understand the problem, developers can compute the instruction-per-cycle (IPC) level of the loop execution, by dividing the instruction count by the cycle count:

```
(gdb) omp loop profile iterations
(gdb) set profile-perf-counters instructions,cycles
```

```
(gdb) continue
... ^C
(gdb) profile graph plot-all all
(gdb) profile graph offline
numa node | .......  y# <
instructions | .... /
cycles | ..........
```

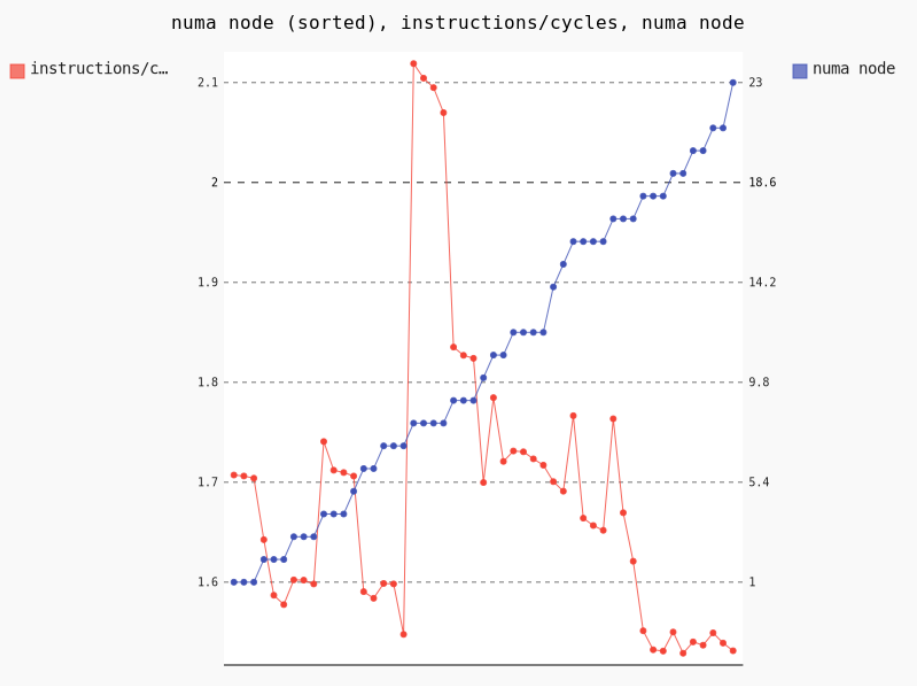

Figure 4.1.3: Pinpointing the Performance Problem with mcGDB: the IPC varies according to the NUMA node.

Figure 4.1.3 shows the plot generated with these commands. We can see that the IPC levels are often correlated to the NUMA node that ran the loop-chuck.

Plotting the `node-misses` against the `cycles` (Figure 4.1.4) finally confirms that the problem is caused by the location of the data on the NUMA nodes: the node getting low cycles consumption also has very low node-misses. That suggests that some data are located on its memory bank, and the other nodes have to access it remotely.

At this stage, developers know that there is a problem with the data access, related to the NUMA architecture. We make the assumption that they are aware of the first-touch policy of NUMA machine, that stores the memory pages on the memory bank of the core that touches it first (*ie*, writes on it).

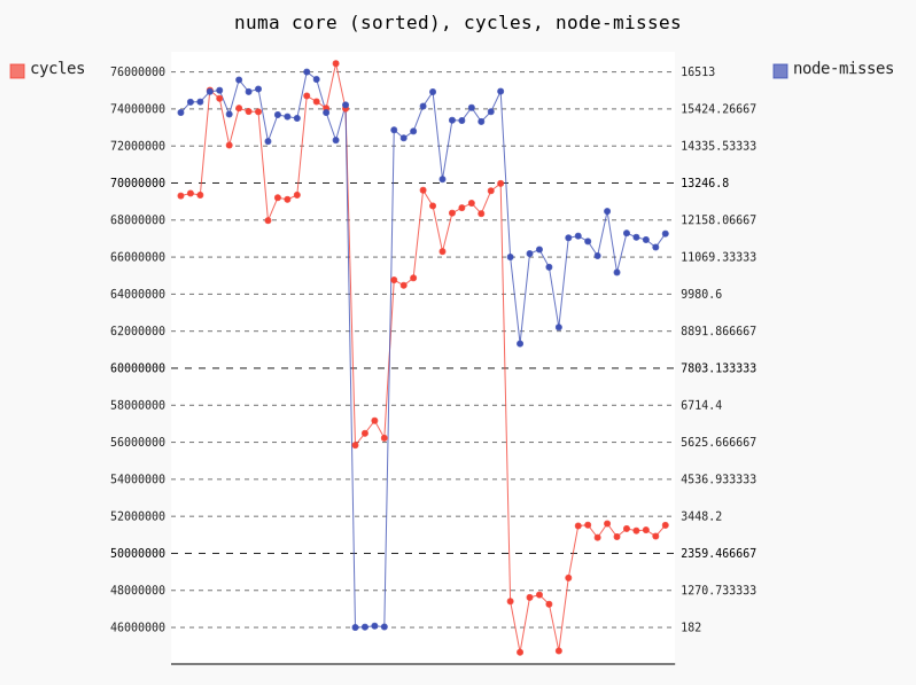

Figure 4.1.4: Pinpointing the Performance Problem with mcGDB: the node miss counter is low on a single node.

To understand the reason of the performance issue, the developers will continue their investigation by studying the code allocating and initializing the buffers used in this loop.

4.2. FINDING THE PERFORMANCE PROBLEM

The allocation of the memory buffers is located in the sequential part of the code. It consists in arrays of 3D matrices, made of indirection arrays:

```
u = (double ****) malloc((lt+1)*sizeof(double ***));
for (l = lt; l >=1; l--) {
    u[l] = (double ***) malloc(m3[l]*sizeof(double **));
    for (k = 0; k < m3[l]; k++) {
        u[l][k] = (double **) malloc(m2[l]*sizeof(double *));
        for (j = 0; j < m2[l]; j++) {
            u[l][k][j] = (double *) malloc(m1[l]*sizeof(double));
        }
    }
}
v = (double ***) malloc(m3[lt]*sizeof(double **));
```

```
    for (k = 0; k < m3[lt]; k++) {
        v[k] = (double **) malloc(m2[lt]*sizeof(double *));
        for (j = 0; j < m2[lt]; j++) {
            v[k][j] = (double *) malloc(m1[lt]*sizeof(double));
        }
    }
    r = (double ****) malloc((lt+1)*sizeof(double ***));
    for (l = lt; l >=1; l--) {
        r[l] = (double ***) malloc(m3[l]*sizeof(double **));
        for (k = 0; k < m3[l]; k++) {
            r[l][k] = (double **) malloc(m2[l]*sizeof(double *));
            for (j = 0; j < m2[l]; j++) {
                r[l][k][j] = (double *) malloc(m1[l]*sizeof(double));
            }
        }
    }
```

We assume that the application developers do not have an advanced knowledge of memory allocators, and hence they assume (incorrectly) that this code does not touch the memory pages.

The initialization of the values is done by the same loops doing the computation, so developers assume (correctly) that the pages are touched by the same core that will access it later.

For the developers, these two assumptions are contradictory and must be experimentally verified. One way to do it is to measure the *system page faults* (such faults are triggered, among other reasons, when a page is touched for the first time): the allocation is not supposed to do many page faults, whereas the initialization should.

In this code, it is easy to locate the beginning and end of the memory allocation, and the end of the initialization (they are in sequence). So we used these code lines to delimit two manual profiling regions:

```
(gdb) start
(gdb) until <beginning_of_allocation>
# Stopped at <beginning_of_allocation>
(gdb) profile manual start
(gdb) until <end_of_allocation>
# Stopped at <end_of_allocation>
(gdb) profile manual stop
| min_flt:                      873,602
(gdb) profile manual start
(gdb) until <end_of_initialization>
# Stopped at <end_of_initialization>
(gdb) profile manual stop
| min_flt:                       15,612
```

This profiling contradicts the developers expectations: the memory allocation did an order of magnitude more page faults that the initialization (873,602 against

15,612).

At this stage, developers now know that the problem comes from the memory allocation: they thought that it was not touched, thus not bound to a particular NUMA node during this phase. But the evidences show that this is not true.

In practise, the standard `glibc malloc` implementation writes meta-information ahead of the pointer it returns. This is fine in most architectures, but not on NUMA systems.

4.3. FIXING THE PROBLEM

Developers have understood that there is a problem with the memory allocator of their application. It touches the memory pages, thus forcing the storage on the node running the main thread.

To ensure that the page distribution is indeed the problem, the developers can run our heap spread command `numa spread_pages`. This command spread the memory pages corresponding to the process' heap all over the machine's nodes, in a round-robin fashion:

```
(gdb) start
(gdb) until <end_of_allocation>
(gdb) profile spread_pages
| Process heap goes from 0x60c000 to 0x670000 (=400Ki B)
(gdb) continue # or detach
# performance improved from 49.76s to 4.45s
```

This command improves the computation time by a factor of 10, from 49.76*s* to 4.45*s*, just be redistributing the process heap space over all the NUMA nodes.

It confirms that a NUMA-aware memory allocator would improve the overall performance.

In this chapter, we have shown an interactive performance debugging workflow, that illustrates the usage and efficiency of the tool developed as part of Deliverables 2 and 4. A conference paper will be prepared in the forthcoming months to publish and spread these results.

Chapter 5

Conclusion

In this document, we described the work done as part of the Nano2017/Dema Deliverable 4. We detailed how we improved our interactive OpenMP debugger and joined two aspects: the functional model-centric debugger (Deliverables D1 and D3) and the interactive performance profiler started earlier this year (Deliverable D2).

We introduced how to interactively profile OpenMP loops, and test NUMA performance debugging hypothesis. We also describe and illustrated a performance debugging workflow, done in collaboration with Aftermath, the target tool of the Sub-project 2.

As part of the Nano2017/Dema sub-project 1, we have demonstrated a novel approach for OpenMP interactive debugging. This approach relies on the abstraction of OpenMP's programming model, and hence could be ported easily to other similar development environments.

With our illustrated workflow, we demonstrated how interactive debuggers can help performance debugging, a domain they not usually used for. We showed that their interactivity allows developers to test and verify debugging hypothesis on-the-fly, without the need to recompile the source-code, and sometimes even by changing dynamically it behavior, or some operating system properties.

We are preparing a conference paper out of the results of this deliverable and those of the Sub-project 2. The paper should be submitted on an international workshop by the end of February 2017.

References

[1] Intel OpenMP Runtime. https://www.openmprtl.org/.

[2] OpenMP 4.0 standard. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[3] Andi Drebes, Jean-Baptiste Bréjon, Antoniu Pop, Karine Heydemann, and Albert Cohen. Language-Centric Performance Analysis of OpenMP Programs with Aftermath. In *International Workshop on OpenMP (IWOMP)*, pages 237–250, October 2016.

[4] Gnu Project. GDB, The GNU Debugger. http://www.gnu.org/software/gdb/, 1986-2013.

[5] Kevin Pouget. *Programming-Model Centric Debugging for Multicore Embedded Systems*. PhD thesis, Université de Grenoble, École Doctorale MSTII, feb 2014.

[6] University of Versailles Saint Quentin en Yvlines . NAS Parallel Benchmarks 3.0, Unofficial OpenMP C Version. https://github.com/benchmark-subsetting/NPB3.0-omp-C/blob/master/MG/mg.c (commit 9681631), 2014-11-08.

Chapter A

Appendix

A.1. Access to the Source Code

A.1.1 Download

mcgdb

- `http://dema.gforge.inria.fr/delivrable/2017-01_mcgdb/mcgdb.tgz`
- `git+ssh://USER@scm.gforge.inria.fr/gitroot/dema/mcgdb.git`

Requirements

```
pip install colorlog pysigset enum34 pyparsing
```

- Profiling
  - Linux Perf (mandatory)
    `https://perf.wiki.kernel.org/`
- Logging
  - Colorlog (recommended)
    `https://pypi.python.org/pypi/colorlog`
- GDB internal thread safety:
  - Pysigset (recommended)
    `https://pypi.python.org/pypi/pysigset/`
- Task/OpenMP
  - Enum34 (Python2 only)
    `https://pypi.python.org/pypi/enum34`

  - pyparsing
    `https://pypi.python.org/pypi/pyparsing`

- Documentation
  - Rendering
    * Sphinx
      `https://pypi.python.org/pypi/Sphinx`
    * Sphinx RTD theme (optional)
      `https://pypi.python.org/pypi/sphinx_rtd_theme`

A.1.2 Installation

Our developments were done with GDB 7.12 and Python 3.5. GDB supports Python 2 and Python 3, and our support should work with both versions.

Load mcGDB from GDB

Put in `.gdbinit`:

```
python
sys.path.append("/path/to/Python")
try:
  import mcgdb
  #mcgdb.initialize()
  mcgdb.initialize_by_name()
except Exception as e:
  import traceback
  print ("Couldn't load Model-Centric Debugging: {}".format(e))
  traceback.print_exc()
end
```

Put in your `$PATH`:

```
ln -s $(which gdb) mcgdb
ln -s mcgdb mcgdb-omp
```

Then load your binary with `mcgdb-omp`

Convenience with GDB/mcGDB

Add these lines to your `.gdbinit`:

```
## almost mandatory:

set height 0
set width 0
```

```
## for convenience:

set breakpoint pending on
set print pretty
set confirm off

# for debugging

set python print-stack full
```

A.1.3 Compile `libmcgdb-omp`

```
cd $MCGDB_PATH
cd model/task/environment/openmp/capture/preload
make # generates __binaries__/libmcgdb_omp.preload.so
```

A.1.4 Compile `libmcgdb_perf_stat.preload.so`

```
cd $MCGDB_PATH
cd model/profiling/
make # generates __binaries__/libmcgdb_perf_stat.preload.so
```

A.1.5 OpenMP environment

Our OpenMP profiling support works with Intel OpenMP.

Intel OpenMP

Intel OpenMP should be compiled with debugging symbols (and optionally OMPT support). Here is the procedure:

```
mkdir -p intel_omp/{build,install}
cd intel_omp
INTEL_OMP_HOME=$(pwd)
# url checked 17/12/2015
wget https://www.openmprtl.org/sites/default/files/libomp_20150701_oss.tgz
tar xvf libomp_20150701_oss.tgz

cd build
cmake -DCMAKE_C_FLAGS="-g -O0"                              \
      -DCMAKE_INSTALL_PREFIX:PATH=$INTEL_OMP_HOME/install \
      -DLIBOMP_OMPT_SUPPORT=true                            \
      $INTEL_OMP_HOME/libomp_oss/

# -- LIBOMP: OpenMP Version       -- 41
# -- LIBOMP: OMPT-support         -- true
# -- LIBOMP: Build                -- 20150701
```

```
# -- LIBOMP: Use predefined linker flags      -- true

make && make install

export LD_LIBRARY_PATH=$INTEL_OMP_HOME/install/lib

# compile OMP application
path/to/clang -fopenmp -g $FILENAME

# check that $INTEL_OMP_HOME/install/lib/libiomp5.so is actually used
ldd a.out | grep libiomp5.so

# tested with clang 3.5.0
clang --version
# clang version 3.5.0
# (https://github.com/clang-omp/clang.git a5dbd16db2515a5b2fa82c7dd416d370968646b1)
# (https://github.com/clang-omp/llvm 1c313aa94183e765c450be6bda3913e22abc3073)
# Target: x86_64-unknown-linux-gnu
```

A.2. RECORDING OF THE INTERACTIVE DEBUGGING

A terminal-based recording of the interactive debugging presented in Chapter 4 is available along with the source-code:

- `http://dema.gforge.inria.fr/delivrable/2017-01_mcgdb/video`