UNIVERSITÉ GRENOBLE - ALPES

NANO2017 DEMA
SP 1—Interactive Debugging

Délivrable D1 :
Intégration d'OpenMP 3.0

Kevin Pouget, Jean-François Méhaut
Équipe LIG/INRIA CORSE
January 7, 2016

Contents

Chapter 1

Introduction

In this document, we detail the work we carried out for the deliverable D1 of NANO 2017 DEMA / Interactive Debugging sub-project.

OpenMP [2] is the specification[1] of a runtime environment for shared memory parallel programming, based on the fork-join programming model. This specification is used more and more often to exploit the computing power of multi-core processors.

OpenMP provides an advanced methodology to develop parallel application, however it does not provide any help for the debugging part of the development process. Worth, it even confuses tools such as source-level interactive debuggers. Indeed, these tools often work natively only at binary and language level and miss in important part of the high-level execution semantic. In the case of OpenMP, the confusion is actually one step above, because it relies on compiler transformations. This means that the code executed around OpenMP pragmas is not strictly equivalent to the one written in the application source files.

Our prior work [7] introduced the concept of "programming-model centric" source-level interactive debugging as an extension of the traditional language-level interactive debugging. The idea was to integrate into debuggers the notion of "programming models", as abstract machines running over the physical ones. These abstract machines, implemented by runtime libraries and programming frameworks, provide the high-level primitives required for the implementation of today's parallel applications.

The idea of programming model is to implement in debuggers functionalities related to these abstract machines. In particular, they should 1/ provide a structural representation of the architecture, 2/ monitor the dynamic behaviors such as communications, and 3/ help users interacting with the abstract machine.

We developed a proof-of-concept, `mcGDB`, as a Python extension of GDB, the debugger of the GNU project.

[1] In the rest of this report, we use OpenMP to refer to any runtime environment implementing the standard. For particular cases, we explicitly use the environment name.

In this deliverable, we extend programming-model centric debugging and MCGDB to encompass OpenMP fork-join programming model. The contribution is divided into two aspects, presented in the following chapters of this document:

1. new functionalities to control OpenMP fork-join parallel programming
2. a sequence diagram representation of the OpenMP execution

The procedure to retrieve mcGDB source code, part of this deliverable, is described in Annex A.1.

Chapter 2

Representing and Controlling Fork-Join Applications

The first difficulty that developers face when using an interactive debugger to study OpenMP applications is that the tools are not aware of the fork-join paradigm used in the execution. Instead, they only show the application as a multi-threaded process that, in the middle of runtime-specific functions, executes the application source code. This execution representation can be called *multi-sequential*, as, from the debugger point-of-view, there is no interaction between the different threads. Our goal in the part of the work was to shift this representation towards a more appropriate fork-join representation.

2.1. Internal Representation of OpenMP Executions Model

The first step of the design of a programming-model centric debugger consists in the capture and interpretation of execution events, in order to build an internal representation of the state of the application. This internal representation will later be used by all the user-facing functionalities of the model-centric debugger.

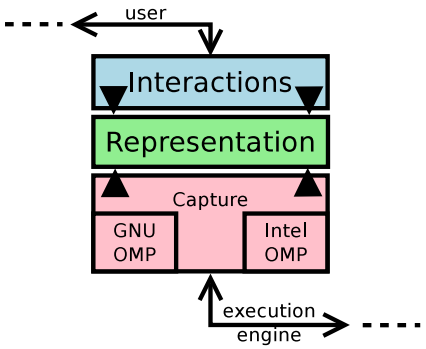

Figure 2.1.1: mcGDB internal organization

2.1.1 `representation` classes

The classes from the `representation` module represent the core of the OpenMP debugger support. Events coming from the `capture` module will call them all along the application execution, and commands from the `interaction` modules will rely on them to provide model-level functionalities to the user interface(s). Fig 2.1.1 pictures this organization. Package `capture` is introduced in the following subsection and in Delivrable 3, Section 2.1 (Aspect-Oriented Programming); and `interaction` modules are described in the two subsequent chapters and in Delivrable 3, Section 2.2 (Multi-runtime support).

The documentation of these classes can be found in the website of `mcGDB` online documentation, as well as a link to the corresponding source code. As an example, the classes reflecting OpenMP `single` zones and barriers look as follows:

```
class SingleJob(Job):
    def __init__(self, worker, parallel_job): ...
    def enter(self, inside, worker): ...
    def finished(self): ...
    def completed(self): ...

class Barrier(Job):
   def __init__(self, parallel_job, worker, single=None): ...
   def reach_barrier(self, worker, location):
      if self.single and self.single.visitor is worker:
          self.single.finished()
   def leave_barrier(self, worker):
   def completed(self):
      if self.single:
         self.single.completed()
      ...
```

Inside these classes, only a minimal tracking is done: we only store the information required to tell which worker did the single zone job, which workers reached it, if they all left, etc.

2.1.2 `capture` module

The `capture` module is the interface between the OpenMP implementation-generic code and the different OpenMP runtime implementations. As of today, this module is implemented for GNU GOMP [5], Intel OpenMP [1] and partially for OMPSS [4].

To continue the previous example with the `single` zone and barriers, in GOMP we use the following functions to update the internal representation:

```
# capture the semantic of ‘bool GOMP_single_start(void)‘
# (returns 1 if thread is allowed inside)

class GOMP_single_start_Breakpoint(OmpFunctionBreakpoint):
```

```
    def __init__(self):
        OmpFunctionBreakpoint.__init__(self, "GOMP_single_start")

    def prepare_after (self, data):
        ret = int(my_archi.return_value(my_archi.INT))

        # gives current single zone or creates it if first
        single = SingleJob.get_single_zone(current_worker())

        inside = ret == 1
        single.enter(inside, current_worker())

# capture the semantic of ‘GOMP_barrier (void)‘
# returns after the barrier has completed

class GOMP_barrier_Breakpoint(OmpFunctionBreakpoint):

    def __init__(self):
        OmpFunctionBreakpoint.__init__(self, "GOMP_barrier")

    def prepare_before (self):
        barrier = Barrier.get_barrier(current_worker())

        barrier.reach_barrier(current_worker(), fname_lineno())

        ...

    def prepare_after (self, data):
        ...
        barrier.leave_barrier(current_worker())
```

In Intel OpenMP and OMPSS, *for these two functions*, the code is identical: only the symbol name differs. The `capture` code of task or parallel zone creation varies more from one runtime to another.

2.1.3 `libmcgdb`: a Preloaded Helper for the `capture` Module

There are multiple ways to implement the information capture required for model-centric debugging. Our choice is to use debugger breakpoints and memory inspection to gather all the information we need. However, one limit of this approach is that it is hard to control *precisely and independently* the thread executions. For instance, “stop

all the threads at that location" is hard to implement in GDB and Python GDB[1,2].

We describe later the command `omp start` (Section 2.2.2), that continues the execution until all the threads are at the beginning of a parallel zone. This command precisely requires such a control of GDB.

To solve this problem, we preload a shared library into the application address-space. This library has to implement the OpenMP runtime *internal* API, that is unfortunately not necessarily public or with long term stability. The functions implementing this API are inserted between the application and the OpenMP runtime library by the dynamically linker (with the `LD_PRELOAD` environment variable). They usually only perform simple operations for the debugger (emit event or block threads with infinite loops), and then forward the function call to the real implementation.

As an example, the interception function used to implement `omp start` looks as follows:

```
void
GOMP_parallel (void *(*fn) (void *), void *arg,
               unsigned num_threads, unsigned int flags) {
  struct trampoline_data *data;

  init_gomp_preload();

  data = malloc(sizeof(*data));
  data->routine = fn;
  data->arg = arg;

  real_GOMP_parallel (GOMP_parallel_trampoline, data,
                      num_threads, flags);
  free(data);
}
```

Function `GOMP_parallel` is called by the application when a `#pragma omp parallel` is reached. It initializes the preloaded library, then overrides the function arguments to divert the worker executions towards `GOMP_parallel_trampoline`:

```
void
GOMP_parallel_trampoline(void *arg) {
  struct trampoline_data *data = (struct trampoline_data *) arg;

  thread_debug.can_run = 1;
  thread_debug.initialized = 1;

  mcgdb_thread_can_run(&mcgdb_can_pass_parallel);
```

[1] This is different from "stop each time a thread passes at that location", which is a standard GDB operation.

[2] Non-stop debugging [8] mode of GDB may help in that purpose, but its Python support is not mature enough to support our model-centric debugging framework.

```
  /* Do not add anything here, mcGDB steps into data->routine(). */

  data->routine(data->arg);
}
```

In this second function, we initialize thread-specific data, then we test if the thread is allowed to enter the parallel zone:

```
void
mcgdb_thread_can_run(volatile int *can_pass) {
  while (!thread_debug.can_run);

  while (can_pass != NULL && ! *can_pass);
}
```

Function `mcgdb_thread_can_run` is watched by `mcGDB`. At the beginning of the command `omp start`, `mcgdb_can_pass_parallel` is set to `0`, and hence the threads will infinite loop in that function. Each time a thread enters `mcgdb_thread_can_run`, a counter is increased in mcGDB. When all the threads are busy waiting, `mcGDB` switches `mcgdb_can_pass_parallel` to `1`, but *does not* continue the execution. Instead, it instructs each thread to sequentially perform a `finish` (finish the current function, `mcgdb_thread_can_run`) and a `step` (step inside the next function), which leads them to the first line of parallel zone (the function behind `data->routine(data->arg)`).

After the execution of this algorithm, all the workers are stopped at the beginning of the parallel zone.

Thread blocked function We also use `libmcgdb` to implement a "thread blocker" function. This function is rather simple, it only consists in an infinite loop:

```
void
__thread_blocker(void) {
  /* This thread/task is blocked by mcGDB.
   * Do not try to exit manually this function,
   * it would compromize your execution.  */
  while(1);
}
```

During the library initialization, mcGDB looks up the address of the symbol `__thread_blocker`. Then, for instance to implement task blocking (see Delivrable 3, Section 1.4.2 #Task blocking), mcGDB set the thread IP register (instruction pointer) to this address (and an offset in x86). As a result, the thread executes the infinite loop, and its normal execution is stopped. Upon unblock request, mcGDB restores the original IP value, and the thread continues its normal execution.

2.2.1 Execution representation

In order to understand the current state of the application, developers must have a straightforward way to see the location of the different workers[3]. We implemented different commands in that regard that present a more suitable (tree-shaped) stacktrace, and overviews of workers position and OpenMP tasks. More advanced execute representation engines (assisted with visualization tools) are presented in the subsequent chapters.

Tree-shaped backtraces

An OpenMP backtrace (also called stacktrace or callstack) representation is not linear (multi-linear), as sequential (multithreaded) application backtraces, but rather tree-like. This representation has already been applied to SPMD applications (like MPI), where it is straightforward to implement: all the MPI processes start from the *same* `main` function, then at some points the execution flows diverge, based on their process rank or data they received.

OpenMP tree backtraces are harder to implement, because the application's threads/workers do not *really* share their backtrace. The connection is only *logical*: within the OpenMP runtime, threads/workers wait for work to be executed, for instance a parallel zone, then they jump into the application code and carry out the task.

Here is an example running the code of Annex A.2, with first our tree callstack (`omp where`), then GDB multi-sequential callstacks (`thread apply all where`).

```
(gdb) omp where
main () at parallel-demo.c:5
  #pragma omp parallel
    #parallel zone #1 of main
    + at parallel-demo.c:13              [Thread 2]
    + at parallel-demo.c:10              [Thread 3,4]
       #pragma omp single_start          [Thread 1]

(gdb) thread apply all where

Thread 4 (Thread 0x7ffff61ed700 (LWP 16923)):
#1  #parallel zone #1 of main () at parallel-demo.c:10

Thread 3 (Thread 0x7ffff69ee700 (LWP 16922)):
#0  #parallel zone #1 of main () at parallel-demo.c:10
```

[3] Workers are the support of fork-join code execution. They are usually implemented with long-life threads. In this report we distinguish one from the other in so that workers only execute "application tasks" and "disappear" afterwards. Threads, as seen by GDB (and the OS) remain all the time.

```
Thread 2 (Thread 0x7ffff71ef700 (LWP 16921)):
#0  #parallel zone #1 of main () at parallel-demo.c:13

Thread 1 (Thread 0x7ffff7fc3780 (LWP 16917)):
#0  #pragma omp single_start ()
#1  #parallel zone #1 of main () at parallel-demo.c:10
#3  #pragma omp parallel ()
#5  main () at parallel-demo.c:5
```

We can also notice in this output that 1/ some frames were omitted and 2/ OpenMP pragma appears in the callstack. This cleanup and rewriting was done with GDB/Python's ability to filter and decorate frames. With the help of mcGDB OpenMP knowledge, we can enrich the stack trace with model-level information, as in `Thread 1 Frame 1`. For this thread , the "language-level" callstack looks as follows:

```
(gdb) where no-filter
#0  GOMP_single_start () at /build/gcc-5.2.0/libgomp/single.c:40
#1  in main._omp_fn.0 () at parallel-demo.c:10
#2  in GOMP_parallel_trampoline (...) at gomp_preload.c:62
#3  in GOMP_parallel (...) at .../libgomp/parallel.c:168
#4  in GOMP_parallel (...) at gomp_preload.c:75
#5  in main () at parallel-demo.c:5
```

We can also notice with GDB `Frame 2 and 4` that `libmcgdb` frames (source file `gomp_preload.c`) where elided from the stack output.

Worker position overview

To give a quick overview of the state of the OpenMP abstract machine, mcGDB command `info workers` shows the current position of OpenMP workers, in terms of parallel zones:

```
(gdb) info workers
> Worker #1: ParallelJob #1 > CriticalJob #1
  Worker #2: ParallelJob #1 > Barrier #1
  Worker #3: ParallelJob #1
  Worker #4: ParallelJob #1 > Barrier #1
```

Task overview

This command lists the OpenMP tasks instantiated in OpenMP abstract machine. We comeback with further details and explanation on OpenMP task support in Deliverable 3, Chapter 1.

```
(gdb) info task 3 +src +deps
   #3 TaskJob #3
```

```
debug_state: created
sources: minimal_omp_threads.c:42-43
Input dependencies:
 i (0x7fffffffe370) from TaskJob #1 (not ready)
 j (0x7fffffffe360) from TaskJob #2 (not ready)
 k (0x7fffffffe35c) from TaskJob #2 (not ready)
Output dependencies:
 i (0x7fffffffe370) to TaskJob #5
 j (0x7fffffffe360) to TaskJob #9
 k (0x7fffffffe35c) to TaskJob #10
----------------------------------------
42 #pragma omp task depend(in:i,y,z) depend(out:i,y,z)
43         foo1(&i, &y, &z);
----------------------------------------
```

We can see in this output that `Task 3` was just created. It has three input and output dependencies, and it corresponds to two lines from the application source code.

Execution visualization

Visualization is an important aspect of our work, as it greatly helps understanding the current state of the execution. We detail two visualization engines in the next parts, one for OpenMP parallel zones (Chapter 3), and the other more focused on OpenMP 4.0 tasks (Delivrable 3, Chapter 1).

2.2.2 Execution control

In this subsection, we detail the different commands we implemented in mcGDB to support OpenMP execution control. Theses commands illustrate the capabilities of mcGDB/OMP, but they could be extended and/or combined in a later effort to tackle more efficiently user needs.

General commands

omp start Continues the execution until the beginning of the first parallel zone. This command relies on `libmcgdb_omp`.

omp next <zone> Continues the execution until the next given OpenMP zone. `<zone>` can be `single, critical, task, sections, barrier` or `master`.

omp step Continues the execution until one thread starts working on a new zone.

omp all_out Continues the execution until all the threads are right after the current zone. This command relies on `libmcgdb_omp`.

omp schedule all|single Alias of GDB's `scheduler-locking` parameter. With value `single/on`, only the *current* thread is allowed to run. With value `all/off`, all

the threads can run. The default (`all/off`) behavior may appear confusing in some situations, in particular during GDB `next` command. While the current thread tries to reach the next source-code line, the other threads (briefly) continue their execution. They may hence hit breakpoints or output text messages. The `single/on` behavior can also introduce artificial locks, for instance if the thread waits on a busy resource.

Zone-specific commands

Sections

omp sections new Catchpoint on the beginning of section zones.

omp sections step-by-step Catchpoint on sections' execution, to execution each section one by one. Activates GDB's scheduler-locking for the zone and deactivates it afterwards.

omp sections finish Continues the execution until the end of the section zone.

Critical sections

omp critical next Continues the execution until the next thread enters the critical zone, to execute it step by step.

Barriers

omp barrier pass Continues the execution until all the threads are right after the current barrier. This command relies on `libmcgdb_omp`.

Tasks

omp task break [all|next] Catchpoint on the beginning of the execution of all the tasks, or just the next one.

In the following part, we introduce a sequence diagram representation of the OpenMP execution, that aims at helping user to have a better and quicker understanding of the current state of the application execution.

Chapter 3

Execution Representation with Sequence Diagrams

When debugging an application, it is important that developers have a quick and accurate understanding of the current state of the execution. Indeed, if each time they pause the application execution, they waste seconds in understanding the current state, they may easily loose track of their debugging guideline. They should rather have a straightforward way to review it, such as what the callstack provides for simple sequential applications.

In OpenMP, two difficulties are combined: first, the callstack is messy because of the entanglement between application frames, compiler-outlined frames and OpenMP runtime frames. Second, all the threads are supposed to execute more or less the same code (SPMD paradigm), but this is not reflected in the usual display of the callstack.

In this chapter, we introduce a visual representation of the current location of OpenMP workers, within the OpenMP zones. This representation is inspired from UML sequence diagrams [3]. These diagrams show the interaction over the time between different actors. The semantic of our diagram is distinct from the UML's, but the idea and general shapes are shared.

3.1. Diagram Semantic

An actor corresponds to an OpenMP worker. Its background color indicates that ...

- **Orange** the worker is currently selected in GDB command-line interface,
- **Red** the worker is selected and GDB scheduler-locking is enabled (see Section 2.2.2).
- **White** the worker is not selected.

A box corresponds to an OpenMP zone. It covers the workers bound to that zone[1,2].

[1] The diagram shows the *current* state of the execution, and its artifacts. Hence, a worker may be *about* to enter the zone, but not yet in. That may appear counter-intuitive.

[2] There is an unsolved problem with that representation, for instance if Worker 1 and 3 are in the zone, but not Worker 2. Currently, the box would incorrectly cover Worker 2.

A red-cross arrow indicates the current location of the worker[3].

An activity box indicates that the worker did execute application code inside the zone. The box superposition indicates the zone nesting level.

A self-referencing arrow indicates the beginning or end of a particular task. That task can be an OpenMP task, but also a section, a critical zone, etc.

A traversing arrow indicates an execution flow transfer, for instance within critical regions.

A box-long double strike indicates a barrier. If workers are blocked inside that barrier, they will appear between two double strikes (`<Barrier>..</Barrier>`). Likewise, if a worker executes a task during its wait, the task will be plotted within the barrier.

3.2. DIAGRAM EXAMPLES

The subsections below present example of sequence diagrams for different OpenMP zones. They correspond to the execution of the code in Annex A.2.

3.2.1 Parallel Zone

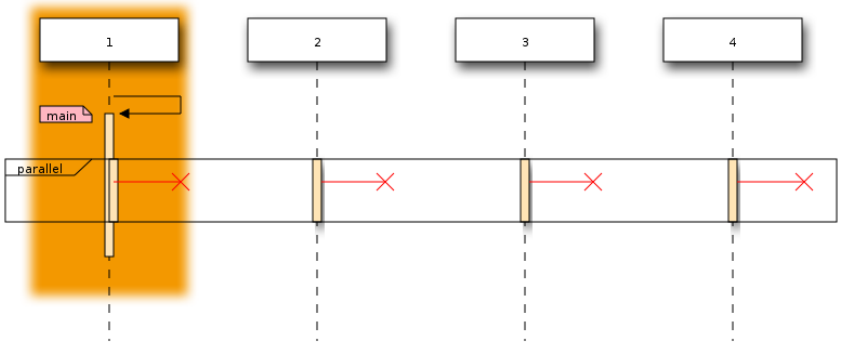

Figure 3.2.1: mcGDB sequence diagram of a parallel zone

[3] There is an unsolved problem with that representation, if Worker 1 is in a barrier, and Worker 2 is *before* the barrier, Worker 2 position may be plotted *after* the barrier.

3.2.2 Single Zone

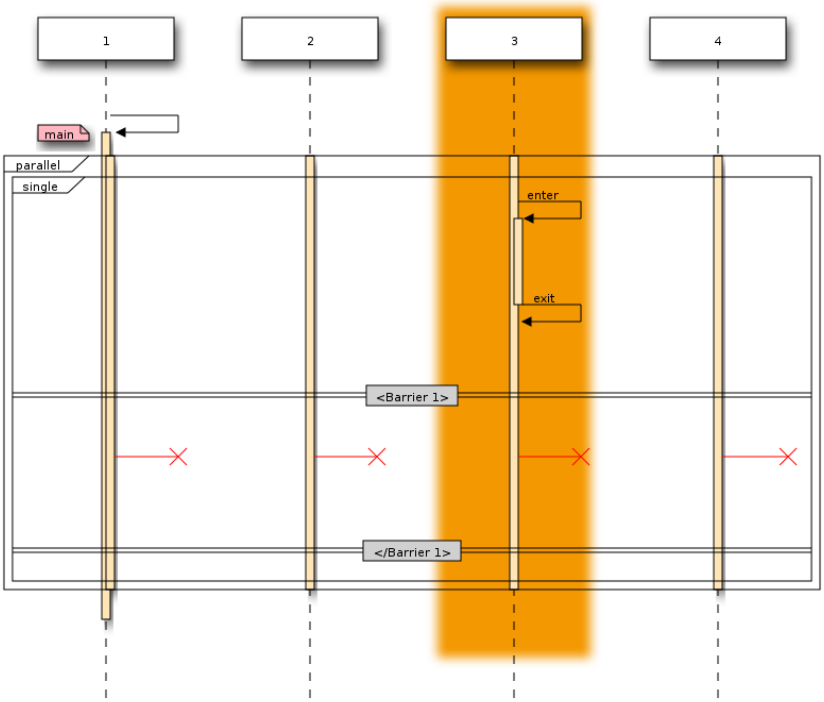

Figure 3.2.2: mcGDB sequence diagram of a single zone and its implicit barrier

3.2.3 Critical Zone

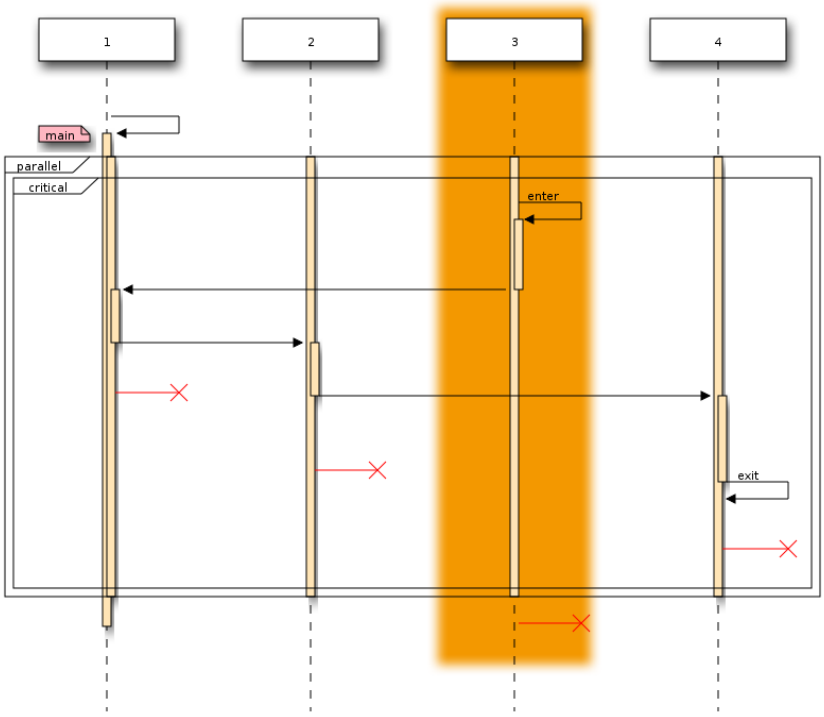

Figure 3.2.3: mcGDB sequence diagram of a critical zone

3.2.4 Task Execution

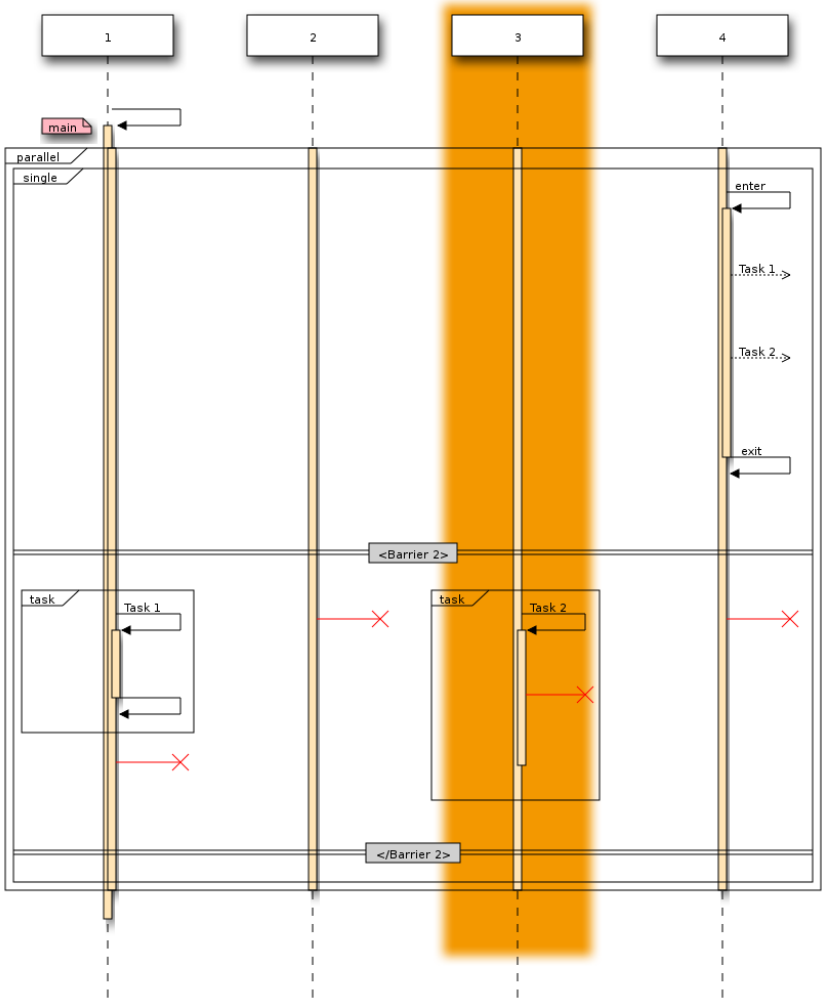

Figure 3.2.4: mcGDB sequence diagram of two task executions

3.3. IMPLEMENTATION AND CONNECTION WITH MCGDB

Our OpenMP sequence diagram visualization engine is based on `seqdiag` [6], an open source tool published under Licence Apache 2.0. It takes as input a text file describing the diagram to plot. This idea is similar and inspired by GraphViz DOT

language.

As an example, here is a commented version of the text file that generated Fig. 3.2.2:

```
{
  1; 2; 3; 4;  # list of the application workers

  group { 3; } # worker to highlight

  1 -> 1 [leftnote="main"];

  parallel {
    4; 3; 2; 1; # workers inside the zone

    single {
      4; 3; 2; 1;

      3 -> 3 [label="enter"]; # beginning/end of
      3 <- 3 [label="exit"];  # single activity box

      === <Barrier 1> ===
      4 -> 4 [here]; 3 -> 3 [here];
      2 -> 2 [here]; 1 -> 1 [here];
      === </Barrier 1> ===
    }
  }
  1 <- 1 [narrow] # finish the 'main' activity box
}
```

We introduced important changes in `seqdiag` to support our vision of concurrent activity diagrams. In particular, it was mandatory for us that activities could occur in parallel, whereas UML semantic is more oriented towards sequential control transfers. Our semantic for parallel zone boxes also differs from UML, as we wanted workers activity box to start and stop automatically at the upper and lower boundary of the parallel box.

The diagram description is generated by mcGDB from its internal representation of the OpenMP execution events. Python module `openmp.interaction.sequence` is hooked with module `openmp.representation` through our aspect interface (discussed in Delivrable 3, Section 2.2). This design creates no dependency between the core module `openmp.representation` and the optional ones from `openmp.interaction.*`.

In `openmp.interaction.sequence`, we maintain another internal representation of the OpenMP execution, oriented towards the plotting of the diagram description. Each time the main representation is updated, the sequence diagram hooks are called, and the second representation is updated as needed.

In mcGDB command-line interface, the diagram generation is concealed behind one command and one parameter.

Parameter `omp-auto-sequence` indicates if the sequence diagram should be redrawn automatically or not. If set, *before displaying the promp* (and not more often[4]), mcGDB will check if the OpenMP internal representation has been updated (in practice, a flag is switched when an aspect hook is triggered) or if the GDB environment has changed (currently, the `scheduler-locking` value and selected thread) since the last plot, and redraw it if necessary.

Command `omp sequence` computes the description of the diagram, and, by default, saves it in `run` and generates `run.svg` and `run.png` images out of it. It accepts several options to changes its behavior:

--print or --show Implies **--no-gen**. Print to the screen the diagram description.

--no-gen Do not trigger the image generation.

--open Implies **--sync**. Open with `eog` (EyeOfGnome) the `png` image generated.

--sync Default. Block until images are generated.

--async Do not wait for image generation.

--no-write Do not save to file the diagram description.

--all Plot all the information captured. By default, only the zones containing a worker are plotted.

Temanejo Visualization interface While we developed the cooperation between Temanejo graphical interface and mcGDB (detailed in Delivrable 3, Chapter 1), we also extended Temanejo to support the visualization of our sequence diagram.

The communication between mcGDB and Temanejo is done through a network socket. Hence GDB can run on an embedded system or on a cluster front-end, while Temanejo remains on the development desktop.

As of today, Temanejo support for OpenMP sequence diagrams is rather primitive: it only supports display and automatic refresh. We can imagine, in future versions, some more cooperation between mcGDB and the diagram representation, such a displaying workers callstack on mouse hover, or switching the active thread by clicking on the actor box.

In the following chapter, we detail the work we did on for OpenMP task debugging and visualization. At the end of the chapter, we comeback on the possibilities and limitations of the cooperation between mcGDB and the of Temanejo task-graph and sequence diagram visualization engine.

[4] We believe that a strict update-on-change semantic is not useful for interactive debugging, as it would introduce an expensive computation cost *and slowdown* for no real benefits. Nonetheless, this feature would be trivial to implement.

Chapter 4

Conclusion

In this document, we detailed the deliverable D1 of Nano2017/DEMA sub- project 1 on Interactive Debugging. The source code corresponding to this deliverable is accessible with the procedure described in Annex A.1.

We introduced the new support of mcGDB for OpenMP 3.0 fork-join programming. This support is divided into two main aspects: 1/ representing and controlling the execution of fork-join based applications and 2/ representing the fork-join execution with sequence diagrams.

In the next months of the DEMA project, we will start the investigation on the possibilities of OpenMP application profiling controlled by an interactive model-centric debugger (Deliverables D2 qnd D4). We will continue and extend the work on mcGDB testing and benchmarking to validate its efficiency. We plan to the KaStORS OpenMP benchmark suite [9] to validate mcGDB implementation and measure its execution intrusion.

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] UML basics: The sequence diagram. http://www.ibm.com/developerworks/rational/library/3101.html.

[4] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesus Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02), 2011.

[5] Free Software Foundation (FSF). GOMP – An OpenMP implementation for GCC. https://gcc.gnu.org/projects/gomp/.

[6] Takeshi KOMIYA. seqdiag - simple sequence-diagram image generator. http://blockdiag.com/en/seqdiag/.

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

[8] Nathan Sidwell, Vladimir Prus, Pedro Alves, Sandra Loosemore, and Jim Blandy. Non-stop multi-threaded debugging in gdb. In *GCC Developers' Summit*, 2008.

[9] Philippe Virouleau, Pierrick BRUNET, François Broquedis, Nathalie Furmento, Samuel Thibault, Olivier Aumage, and Thierry Gautier. Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite. In *10th International Workshop on OpenMP, IWOMP2014*, 10th International Workshop on OpenMP, IWOMP2014, Salvador, Brazil, France, September 2014. Springer.

Chapter A

Appendix

A.1. Access to Source-Code

A.1.1 Download

mcGDB

- `http://dema.gforge.inria.fr/delivrable/2015-12_mcgdb/mcgdb.tgz`
- `git+ssh://USER@scm.gforge.inria.fr/gitroot/dema/mcgdb.git`

OMP Seqdiag

- `http://dema.gforge.inria.fr/delivrable/2015-12_mcgdb/seqdiag.tgz`
- `git+ssh://USER@scm.gforge.inria.fr/gitroot/dema/seqdiag.git`
- Refer to upstream/readme to install
  - `http://blockdiag.com/en/seqdiag/`

Temanejo

- `http://dema.gforge.inria.fr/delivrable/2015-12_mcgdb/temanejo-mcgdb.tgz`
- `gitclonegit+ssh://USER@scm.gforge.inria.fr/gitroot/dema/temanejo_mcgdb.git`
- Refer to upstream/readme to install
  - `http://www.hlrs.de/organization/av/spmt/research/temanejo/`

Requirements

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

- 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`
  - Graph (one package–not currently in use)
    * Networkx (optional)
      `https://pypi.python.org/pypi/networkx/`
    * PyGraphViz (optional)
      `https://pypi.python.org/pypi/pygraphviz`
  - Sequence Diagram
    * Seqdiag (mcGDB version)

- Toolbox/Target

  - Access/ssh
    * Pushy
      `https://pypi.python.org/pypi/pushy`

- 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 7.10 and Python 2.7.10. GDB supports Python 2 and Python 3, and our support show work with both versions, except when communicating with Temanejo/Ayudame, which mandates Python2 usage. Python 3 usability was tested on version 3.4 and 3.5.

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: %s" % e)
  traceback.print_exc()
end
```

Put in your `$PATH`:

```
ln -s $(which gdb) mcgdb
ln -s mcgdb 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 OpenMP environment

Our OpenMP support works with GNU Gomp and Intel OpenMP.

GNU Gomp

Our GNU Gomp support was tested with a standard gcc 5.2.0 (archlinux x86 build), with comes with libgomp 1.0.0.

Intel OpenMP

Intel OpenMP should be compiled with debugging symbols (and 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.1.5 Test, Benchmark and Documentation

Test and benchmark mcGDB

With `sys.path` correctly configured, run:

```
import mcgdb
mcgdb.run_tests()
```

or from command-line:

```
python3 -c 'import mcgdb; mcgdb.run_tests()'
```

Generate mcGDB documentation

```
cd /path/to/mcgdb
cd documentation
make html
# or
make -f /path/to/mcgdb/documentation/Makefile html
```

A.1.6 OpenMP Sequence Diagram

Install `seqdiag` upstream version (for dependencies). It works with Python2 only.

```
sudo pip2 install seqdiag
sudo pip2 uninstall seqdiag
```

Then put in your `$PATH`:

```
ln -s seqdiag /path/to/python/seqdiag/seqdiag.py
```

A.2. OPENMP PARALLEL ZONE EXAMPLE

```
#include <omp.h>
#include <stdio.h>
int main() {
  printf("Beginning of main\n");

#pragma omp parallel
  {
    int id = omp_get_thread_num() + 1;

    printf("@%d in the parallel zone\n", id);

#pragma omp critical
    {
      printf("----------\n");
      printf("@%d in critical zone\n", id);
      printf("----------\n");
    }

#pragma omp sections
    {
#pragma omp section
      {
        printf("@%d in section 1!\n", id);
      }
#pragma omp section
      {
        printf("@%d in section 2!\n", id);
      }
#pragma omp section
      {
        printf("@%d in section 3!\n", id);
      }
    }
#pragma omp barrier
#pragma omp master
    {
      printf("@%d Master task...\n", id);
    }
#pragma omp barrier

    printf("@%d end if parallel zone!!!\n", id);
  }
  printf("End of main!\n");
}
```