UNIVERSITÉ GRENOBLE - ALPES

# NANO2017 DEMA
# SP 1—Interactive Debugging

# Délivrable D3 :
# Intégration d'OpenMP 4.0

Kevin Pouget, Jean-François Méhaut
UJF-LIG/CORSE
January 19, 2016

# *Contents*


**1 Introduction** **5**

**2 Task Debugging in Cooperation with Temanejo** **7**
2.1 OpenMP Task Programming Model . . . . . . . . . . . . . . . . . . . . . . 7
2.2 Temanejo Task Debugger . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.2.1 Property Mechanism . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.2.2 Task Execution Control . . . . . . . . . . . . . . . . . . . . . . . . 9
2.2.3 Ayudame Helper Library . . . . . . . . . . . . . . . . . . . . . . . 9
2.3 Interactions between Temanejo and mcGDB . . . . . . . . . . . . . . . . . 10
2.3.1 Ayudame preload . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.3.2 mcGDB commands . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.4 mcGDB+Temanejo Task Debugging . . . . . . . . . . . . . . . . . . . . . 12
2.4.1 Task State and Properties . . . . . . . . . . . . . . . . . . . . . . 12
2.4.2 Task Execution Control . . . . . . . . . . . . . . . . . . . . . . . . 14

**3 Discussions about mcGDB Implementation** **15**
3.1 Supporting Multiple OpenMP Environment . . . . . . . . . . . . . . . . 15
3.2 Aspect-Oriented Programming for `interaction` Modules . . . . . . . . 16
3.3 Performance Micro-Benchmarking . . . . . . . . . . . . . . . . . . . . . 18
3.3.1 Native GDB with Python Code . . . . . . . . . . . . . . . . . . . . 18
3.3.2 mcGDB OpenMP . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

**4 Conclusion** **23**

**References** **25**

**A Appendix** **27**
A.1 Access to Source-Code . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
A.1.1 Download . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
A.1.2 Installation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
A.1.3 Compile `libmcgdb-omp` . . . . . . . . . . . . . . . . . . . . . . . 30
A.1.4 OpenMP environment . . . . . . . . . . . . . . . . . . . . . . . . . 30
A.1.5 Test, Benchmark and Documentation . . . . . . . . . . . . . . . . 31

A.1.6 Ayudame/Temanejo . . . . . . . . . . . . . . . . . . . . . . . . . . 32
A.2 OpenMP Task Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

# Chapter 1

# Introduction

In this document, we detail the work we carried out for the deliverable D3 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.

In this deliverable, we extend programming-model centric debugging and MCGDB

[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.

to encompass OpenMP task-based programming model (Chapter 2). We also discuss in Chapter 3 complementary aspects of mcGDB implementation.
The procedure to retrieve the deliverable source code is described in Annex A.1.

# *Chapter 2*

## *Task Debugging in Cooperation with Temanejo*

Temanejo [5] is a task-graph debugging tool for OMPss [3] programming environment (part of the BSC STARSS family ). It offers a visual representation of the application task graph, as well as different properties of the tasks and data dependencies. It also allows blocking and unblocking tasks, and stepping the application execution task-by-task.

The design of this tool is close to what we advocate with programming-model interactive debugging. However it currently misses one aspect that is crucial from our point-of-view: it does not offer source-level debugging capabilities.

Indeed, Temanejo works in cooperation with a helper library, Ayudame, that is running within the task-based runtime. This design helps them capturing and controlling the programming-model abstract machine, however it does not support any kind of source-level introspection or interactivity.

After a fruitful meeting with Jose Gracia and Mathias Nachtmann from HLRS Stuttgart, Germany, we decided to start a cooperation between both tools, so that mcGDB would offer source-level (language—through GDB—and model) interactive debugging, and Temanejo the visualization engine and part of the model-level interactivity.

In the following, we introduce OpenMP 4.0 task support, which is the target of this part of the work. Then we describe Temanejo debugging capabilities, and finally we detail the interactions that occur between mcGDB and Temanejo.

### 2.1. OpenMP Task Programming Model

OpenMP 4.0 support for task programming is based on data dependencies. Before this version, OpenMP only supported independent tasks, with no scheduling constraints. We discussed briefly the mcGDB support for such simple tasks in Sections 2.2.2 and 3.2.4.

OpenMP 4.0 [2] introduced the ability to specify input and output dependencies when creating tasks. The dependencies are memory locations that are respectively read or written by the task. Output dependencies are considered ready when their generating tasks finish their execution. Respectively, input dependencies blocks the

task execution their writer task have completed their execution.

The creation of dependent tasks looks as follows:

```
#pragma omp task depend(in: i,j) depend(out:i,j)
       update_1(&i, &j);
#pragma omp task depend(in: i)   depend(out:i)
       update_2a(&i);
#pragma omp task depend(in: j)   depend(out:j)
       update_2b(&j);
```

Here, functions `update_2a` and `update_2b` will not be executed until `update_1` has completed, however they may be executed in parallel. In Appendix A.2 we reproduced the sample code we use to test Temanejo and mcGDB task support.

### 2.2. TEMANEJO TASK DEBUGGER

Temanejo's main window is presented in Fig. 2.2.1. The graph plotted correspond to the execution of the source-code in Appendix A.2.

We can see that the tasks (the nodes) have different colors. As per the legend on the left-hand side, this corresponds to their debugging state: created, running or finished. Node margin color and shape have their default value. The data dependencies (the links between the nodes) are colored according to the name of the variable on which the dependency is set. The arcs are (implicitly) oriented, the upper node has an *output* dependency on the variable, the lower one has an *input* dependency.

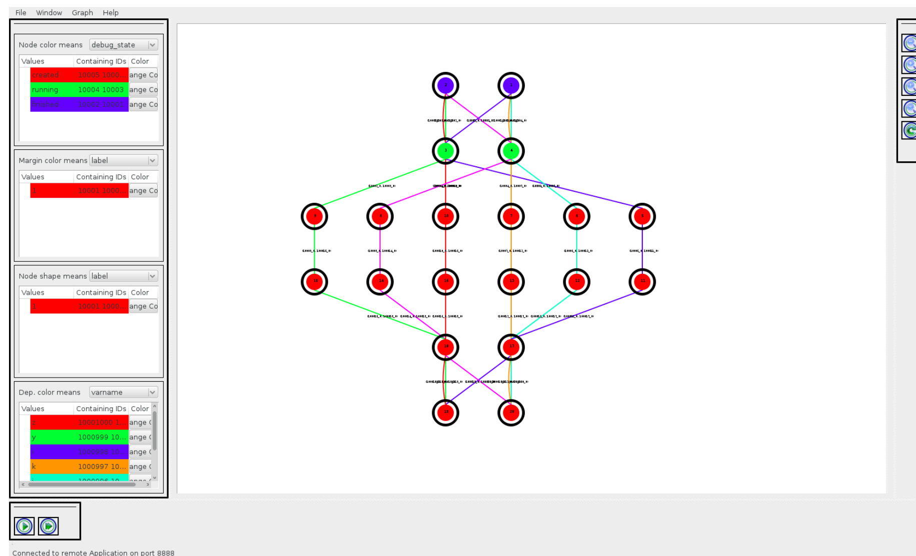


Figure 2.2.1: Temanejo task-graph visualization

### 2.2.1 Property Mechanism

Since the refactoring of its version 2.0, Temanejo is agnostic of the underlying runtime. To that purpose, the information about the execution state was abstracted away into a property-based system. Tasks and dependencies must have a unique identifier, and dependencies must have inbound and outbound tasks. Then, any other information related to a task or a dependency is stored as a key-value property.

This abstract mechanism is very convenient for our purpose, as for instance the variable name (`varname`) of dependency was not available in Temanejo native execution. But once mcGDB could capture this information, it was straightforward to inform Temanejo about it.

These properties are listed on Temanejo side-bar. Task properties allow users to control the inner and outer colors and the shape of the graph nodes. Dependency properties allow users to control the color of the graph edges.

### 2.2.2 Task Execution Control

Natively, Temanejo lets users block the execution of a specific task, with a left click on a node. Internally, Temanejo transmits the order to Ayudame, which blocks (with an infinite loop) the task execution until a release order. Temanejo can also "step forward" the execution, that is, let $n$ tasks start their execution. The value of $n$ is 1 for a single step and 10 for a fast-forward.

### 2.2.3 Ayudame Helper Library

One strength of Temanejo native operation model is the Ayudame library, loaded inside OMPss runtime. Ayudame is the operational counter-part of Temanejo: it collects the different information directly from the runtime and transmits them to Temanejo. It also receives Temanejo requests (only task blocking so far) and implement them.

Ayudame itself is runtime independent: its interface is generic, so it can be interconnected to any task runtime. But its packaging also provides a OMPss/Mercurium compiler instrumentation library that tells the compiler where and how to insert the relevant calls to Ayudame inside OMPss/Nanox runtime.

As of today, there is only a weak collaboration between Ayudame and mcGDB: we implemented an OMP `capture` back-end that relies on Ayudame to capture and control OMPss tasks. As OMPss is not a target runtime for the DEMA project, we only developed this support as a proof of concept. This preliminary support shows that, for task debugging, we can defer the actual instrumentation part to Ayudame. If tomorrow Ayudame supports new runtime systems, they will be supported for free in mcGDB[1].

[1] Again, this only applies to the task debugging support, not the rest of what what presented earlier.

### 2.3. INTERACTIONS BETWEEN TEMANEJO AND MCGDB

Fig. 2.3.1 presents an overview of the cooperation architecture between mcGDB and Temanejo. In this workflow, mcGDB and this application can be executed in a remote system, such as an embedded board or a cluster front-end, whereas Temanejo GUI remains on the workstation. Temanejo opens a network sockets, and on request, mcGDB connects to it and feeds Temanejo with the graph structure and sequence diagram updates. mcGDB also listen on this socket for Temanejo requests.

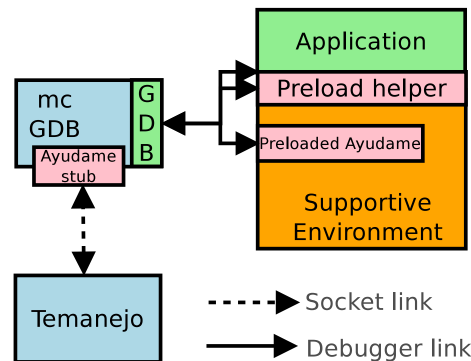


Figure 2.3.1: Cooperation between mcGDB and Temanejo/Ayudame

## 2.3.1 Ayudame preload

In Fig. 2.3.1, we can see an Ayudame stub within mcGDB. This stub is Ayudame `communication` module, whose core is written in C++ but also exported in Python through a `swig` binding. This module allows us to communicate with Temanejo through an implementation-independent interface, which is better for the portability over future Temanejo/Ayudame releases.

Originally, we intended to connect Ayudame with the Ayudame-stub running in mcGDB, to capture Ayudame execution knowledge: mcGDB was supposed to act as a proxy between Ayudame and Temanejo, and capture on-the-fly the information required to build its internal representation.

This communication link has been dismissed because of its inefficiency. Indeed, the socket link is by nature asynchronous, and in our context, the execution of mcGDB code *blocks* the application execution, and inversely. Hence, that induces a significant delay between the happening of event and its handling into mcGDB. This delay is not acceptable for us, as it means that the debugger representation is not always in sync with the execution state. To counter this problem, we deactivated the socket communication in Ayudame and replaced it with a non-operation. And with a breakpoint on Ayudame event handler (`ayu_event` function), we capture synchronously the information that would have been transmitted over the socket link.

McGDB can also use Ayudame preloaded library blocking capabilities: during

the initialization of the library, we capture the setup of the blocking and unblocking functions (they are part of the TCA interface, that abstracts away these runtime-dependent functionalities). By calling these functions from mcGDB, we let Ayudame decide how to implement the task blocking and unblocking.

One benefit of this design is that the runtime-dependent function could mark the task as "non-schedulable" *within* the task scheduler, and hence the task blocking would not affect the execution parallelism. This is however, as of today, not the current implementation, which relies on an infinite loop. This is equivalent in terms of result to our implementation—introduced in Section 2.1.3#Thread blocked function.

### 2.3.2 mcGDB commands

We introduced different commands in mcGDB to control and interact with Temanejo:

**omp temanejo init [+host=localhost] +port=[8888]** Setup the socket link communication between mcGDB and Temanejo. The optional arguments indicate how to reach Temanejo server. If Temanejo is not listening yet on the port, the Ayudame stub will loop forever until the socket is available.

**Note:** Before the `init` command is ran, mcGDB records the operations required for Temanejo visualization. Hence, event if the link is setup in the middle of the execution, Temanejo will still receive and replay all the events that occurred beforehand, leading to an up-to-date and accurate representation[2].

**Interaction from Temanejo to mcGDB** When mcGDB connects to Temanejo, it currently displays the following warning:

```
Please note that Temanejo cannot control *GDB*, only mcGDB.
Manually run ‘omp temanejo jobs run all‘ when needed.
```

The problem behind this warning is that GDB is not multi-thread safe, though we can (and must) use Python threads, for instance to listen to Temanejo requests. These threads can access and modify the Python environment, however *they must not* access `gdb.*` package. This package is mostly implemented in C and directly accesses GDB internal structures. In practise, any call to this package, or objects created from it, leads to a segmentation fault.

Hence, a Temanejo request that only affect mcGDB state can be executed automatically. However, a request that involves GDB must be stacked. It will be either automatically executed (by the main thread) upon execution resume (with an event callback from GDB), or manually with `omp temanejo jobs run all`.

A fix for GDB would not be overly hard to implement, but it would imply a C patch telling `readline` library to periodically jump into Python environment and execute the stacked functions (that is equivalent to our `run all` code).

[2]The current implementation of this recording is simplistic, a proxy records the function calls to the communicator-to-be object, and replays them when the communicator is actually instantiated. We did not study to cost of this recording, which may have to be optimized and/or deactivable.

**omp temanejo jobs** List the jobs currently stacked

**omp temanejo jobs run all|<id>** Run all or one job

**omp temanejo jobs cancel <id>** Cancels a job

**Debugging**

**omp temanejo debug [on|off]** If `on`, print on the console the messages sent to Temanejo.

**omp temanejo debug nop [on|off]** If `on`, message are not actually sent to Temanejo.

**omp temanejo debug userdata <data>** Send to Temanejo a `userdata` message with payload `MCGDB#<data>`.

**omp temanejo finish** Close the connection to Temanejo.

**Ayudame preloaded library**

**omp ayudame preload** updates GDB's environment to preload Ayudame library in the application. This command has to be run *before* the beginning of the application execution.

## 2.4. MCGDB+TEMANEJO TASK DEBUGGING

In this section, we introduce the task debugging support we designed in the context of the cooperation between mcGDB and Temanejo.

### 2.4.1 Task State and Properties

**Description**

**info task [<id>*]** Print the list of the tasks created by the application. If `<id>*` is provided, only include task with these ids.

- **[key]=[value]** Filter on the properties to print. Key must start with `key` and value must start with `value`.
- **+src** Print the source-code of the tasks.
- **+deps** Print the dependencies of the tasks (name, address and IDs of the dependent tasks)
- **+sched** Indicate if the task is not schedulable (see next subsection #Artificial Debugger Lock Detection).
- **+internal** Also print internal properties (starting with `__`).
- **+none** Include properties whose value is `None`.

In Temanejo, task and dependency properties are displayed through node and edge color, shape and outline, as shown in Fig. 2.2.1.

**Properties set by mcGDB**

**Task properties:**

**debug_state** State of the task, according to mcGDB. It can be `created`, `running`, `finished`, but also `blocked by the debugger` or `blocked by dependency` (see Section 2.4.1#Debugging-lock Detection).

**executed_by** Identifier of the worker that executed this task.

**running_on** Identifier of the worker that executes the task. If the task has completed, this property is set to `None`.

**src_lines** Format: `<filename:start:stop>`. Name of the file that defines the task, and first and last line number of its scope.

**Dependency properties:**

**varname** Name of the variable that hold the data dependency.

**address** Address of the data dependency.

**Artificial Debugger Lock Detection**

Blocking tasks from the debugger (see Section 2.4.2#Task blocking) leads sooner or later the execution into a *artificial* deadlock situation. This a not a real application deadlock, as it is introduced by the debugger, but nonetheless the execution is not able to make any further progress. Hence, the debugger can help the user by noticing this situation and breaking the execution.

We implemented such a mechanism in mcGDB, thanks to its task-graph knowledge: a task can only run if all of its ancestors have completed their execution. Hence, blocking a task will prevent the execution of all of its descendants.

To take that into account, when the user blocks a task, we change its state (`debug_state`) to `blocked by the debugger`, and set the state of its descendants to `blocked by dependency`. In addition to the ordinary state `created`, `running` and `finished`, we can decide if the task is schedulable or not (see `info task +sched` to query it). If no task is schedulable (and they are not all finished), then we have an artificial deadlock. To automatically stop the debugger when this happens, each time a task becomes unschedulable, we check if there is one schedulable. If not, we break the execution.

A particular situation that did not happen when testing on a desktop computer, but did occurred on Juno board LITTLE/slow cores (see Section 3.1) is when the graph is not immediately fully constructed. In this case, if the user blocks a task, and later a new task is connected to its descendants, then the state of this task must be changed to `blocked by dependency`. Otherwise, the debugger will see this task as schedulable and will not detect that the artificial deadlock situation occured.

#### 2.4.2 Task Execution Control

**Catchpoints** These commands are used to stop the execution at different points related to task creations and executions.

**omp next task** Continues the execution until the next task creation.

**omp task break all|next [n]|<id>*** Catchpoint on the beginning of the execution of all the tasks, the next one, the $n^{th}$ next one, or only for tasks with IDs in `<id>*`

In Temanejo:

**omp task break next** is equivalent to the button `Next`

**omp task break next 10** is equivalent to the button `Fast-forward`.

**omp task break <id>** is equivalent to a right-click on a task, then `break`.

**Task blocking** This command blocks the execution of the tasks. The actual implementation of this command varies between the runtime. As of today, it means that upon the beginning of task execution, the underlying thread will start an infinite loop (see Section 2.1.3#Thread blocked function).

**omp task block <id>*** Blocks the tasks with IDs in `<id>*`.

**omp task unblock <id>*** Unblocks the tasks with IDs in `<id>*`.

In Temanejo:

**omp task block <id>** is equivalent to a right-click on a task, then `block`.

**omp task unblock <id>** is equivalent to a right-click on a blocked task, then `unblock`.

# *Chapter 3*

## *Discussions about mcGDB Implementation*

In this last chapter of the report, we go though mcGDB implementation and experimentation details. We first discuss the portability question, regarding the OpenMP runtime but also the CPU architecture. Then, we present an implementation technique we used to decouple the core module (`representation`) from the user-facing interface (`interaction` package). We finally discuss the micro-benchmarking we carried out to measure the execution overhead introduced by mcGDB.

### 3.1. Supporting Multiple OpenMP Environment

One goal of mcGDB framework and architecture is to facilitate the porting from one model/environment to another. In the case of OpenMP, this means porting from one runtime implementation to another.

**Multi-runtime support** To highlight this ability, we developed our debugger support for both GNU GOMP [4] and Intel OpenMP [1]. As of today, both runtime have the same support in mcGDB. We also have a preliminary support for OMPss [3] and Ayudame targets, but further engineering work would be required to finalize their support.

The distinction between the runtime implementations are concealed in the `capture` package. To support a new implementation, one have to figure out how to capture the right execution events (new parallel zone, beginning of a task execution, etc.) and call the `representation` API accordingly. The `libmcgdb` library would have to be updated as well.

The ongoing work on OMPT and OMPD APIs [6] (OpenMP Trace and Debugging APIs, respectively) appears as a promising opportunity to have a full implementation-independent mcGDB support for OpenMP. These standardized interfaces would allow tracers and debuggers to receive notifications upon various OpenMP execution events. These events are very close to what is required for model-centric debugging, which means that mcGDB could catch them to build its internal representation. Hence, it would become implementation agnostic and would support any compliant OpenMP implementation.

**Multi-architecture support** As part of the *embedded* aspect of DEMA, we tested our debugger support on an ARM Juno board, featuring a big.`little` processor.

Since its early developments, mcGDB implementation is target independent. The few aspects that are architecture dependent (such as data types and parameter accesses) are abstracted away behind a generic interface (`mcgdb.toolbox.target.my_archi`), plus GDB internal mechanisms (frame specifications, type printing, etc.). Hence, the main part of the ARM porting consisted in adding a simple architecture support for ARM `aarch64`.

Another problem showed up during the tests, due to the fact that `arm-gdb` cannot modify global variables. This is a debugger implementation limitation unrelated to mcGDB. It finally did not affect our support, as these global variables (in `libmcgdb`) were not actually used, but only implemented prospectively (to let the threads know their GDB thread id).

### 3.2. ASPECT-ORIENTED PROGRAMMING FOR `interaction` MODULES

As we mentioned in the previous section, the design of mcGDB puts an important focus on modularity: it should easily support new architecture and runtime implementation. This this modularity also concerns the user-interaction package. Tomorrow, our command-line functionalities may be replaced by a graphical version, or simply removed. Hence, these functionalities must be decoupled as much as possible from the core modules.

To support this modularity, we introduced in mcGDB an aspect-oriented programming interface. This interface is specific to our needs, and hence we wrote it from scratch.

#### Description

Aspect-oriented programming is a programming technique that allows adding behavior (code) to an existing function without modifying the function itself. Here is an example is pseudo-code:

```
function test()
  // do something
end test;


@aspect.before(test)
function before_test()
  // do something before test
end before_test;

@aspect.after(test)
function after_test()
  // do something after test
end after_test;
```

Function `test` is the main function (in mcGDB, it is part of the `representation` module). Aspect functions `before_test` and `after_test` are respectively executed before and after function `test` (in mcGDB they are in the `interaction` package).

#### Implementation in mcGDB

In mcGDB, aspect-oriented support is implemented in module `mcgdb.toolbox.aspect`. It relies on Python (2 and 3) introspection capabilities.

In mcGDB OpenMP `representation` module, where the core functionalities are, we define the classes that can receive aspects by setting `mcgdb.toolbox.aspect.Tracker` as a super-class (Python support multiple inheritance so it does not alter the original design):

```
class Job(aspect.Tracker): ...
class ParallelJob(Job): ...
  def start_working(self, worker): ...
  def stop_working(self, worker): ...
```

In mcGDB OpenMP `interaction` modules, the aspects are defined with the following syntax:

```
def step_aspects(Tracks):
  @Tracks(representation.ParallelJob)
  class ParallelJobTracker:
      def __init__(this): ...
      def start_working(this): ...
      def stop_working(this): ...

aspect.register("step", step_aspects)
```

By default, aspects are called `after` the main function execution, except if the aspect function defines a `before` keyword. Setting before *and* after aspects may not be possible in the current implementation.

Inside the aspect function, the first argument (named `this` in the example, to contrast with Python usual `self`) holds the information about the main function:

**this.self** holds a reference to the actual class instance (a `ParallelJob` instance here)

**this.args.*** contains the named arguments of the actual function (for instance `this.args.worker` in `start_working` function), *except for* the class constructor `__init__` were we cannot have access to the name of the arguments.

**this.meth_args** contains the ordered list of the function arguments (useful only in the class constructor)

We believe that this implementation technique is particularly convenient for the implementation of the `interaction` modules, as it leads to cleanly decoupled user-facing functionalities. Adding a new functionality does not impact the rest of the implementation, and hence it can be activated or deactivated at will.

### 3.3. PERFORMANCE MICRO-BENCHMARKING

As part of our developments for the DEMA project, we wrote a micro-benchmark and non-regression testing framework. We conducted preliminary experimentation to measure the impact of GDB breakpoints and mcGDB event capture (i.e., breakpoints and Python code). The results are presented in Fig. 3.3.1.

To measure the debugger intrusion in the execution time, we first wrote a simple C code that loops over a given task (in its algorithmic definition). This task is a function call for GDB testing, and OpenMP parallel zone for mcGDB OpenMP testing. The C code measures the time taken for the loop execution, and divides it by the loop counter to get the average time of single iteration.

Second, we implemented an automatic GDB execution framework, controlled by Python scripts. These scripts set breakpoints, watchpoints, etc. according to the requirements, then run the code and parse the execution time computer by the C code.

The average of a single iteration is what is plotted in Fig.3.3.1. The loop counter was set to 1000, and the experiment was repeated 10 times in a row.

### 3.3.1 Native GDB with Python Code

The chart in Fig.3.3.1#Native GDB and Python presents the time it takes to carry out some of GDB basic operations:

**Nominal time** Time of the `usleep(t)` syscall. The sleep time `t` is always excluded.

**Breakpoint command** Time of an internal GDB breakpoint set through the command-line interface (`break <loc>; command silent continue;`).

**HW Watchpoint command** Like above, but with a memory breakpoint.

**Python Breakpoint parameters ii** Internal breakpoint with a Python callback. The last digits indicate (as a binary flag—01 for 1, 10 for 2, 11 for 1 and 2) what parameters were read:

- 1 is a function parameter: `int(gdb.parse_and_eval("it")`
- 2 is a local variable of the caller function: `int(gdb.newest_frame().older().read_var("i").`

**Function Breakpoint** This is a basic mcGDB breakpoint, with some more Python code involved that Python Breakpoint parameters 00.

**Finish Breakpoint** This is another basic mcGDB breakpoint that stops before and after a function call.[1]

From this chart, we can draw the following conclusions:

[1] Note that this is not a `gdb.FinishBreakpoint`, which currently takes a lot of time, `120.000us`. We will have to investigate what happens.

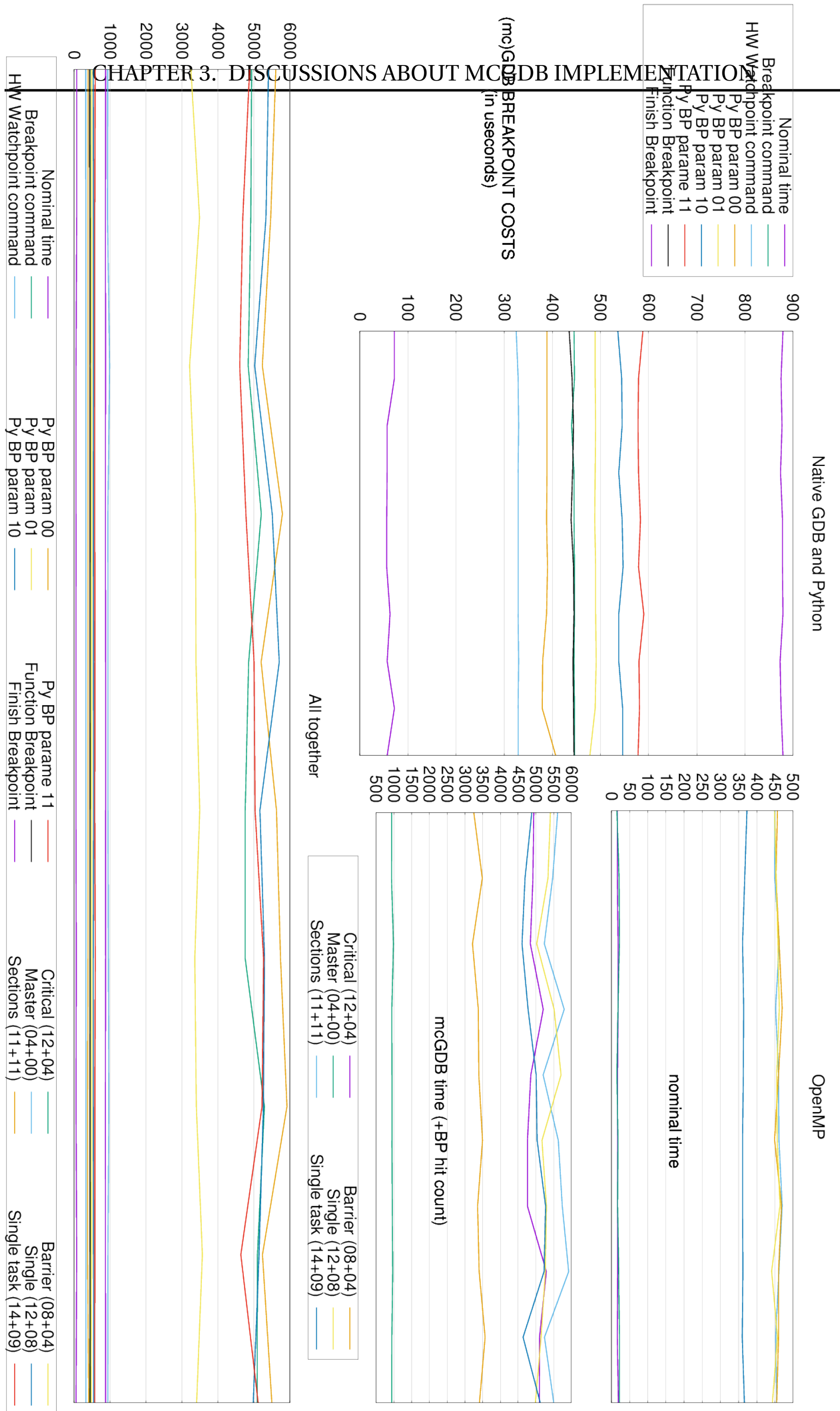


Figure 3.3.1: Micro-benchmarking of debugger intrusion in the execution time.

- Hardware watchpoints are faster than breakpoints. This makes sense as for breakpoints, GDB needs to switch to the debuggee context multiple times to single-step over the breakpointed instruction, whereas this is (certainly) automatic with hardware-assisted watchpoints.
- Python breakpoints are faster than GDB command-line breakpoints. This may be due to a repeated parsing of the breakpoint `command`.
- Reading values from the debuggee's current frame takes $100us$. Reading it from the frame above adds $50us$, which maybe the time required to query the OS about the stack registers value. $100us$ is also close to the difference of time between a breakpoint and a watchpoint.
- A mcGDB Python `FunctionBreakpoint` takes as much time as GDB breakpoint command. The additional time compared with the simple python breakpoint (parameters 00) certainly comes from Python interpretation cost.
- A mcGDB `FinishBreakpoint` involves 2 breakpoints, so it is logical that it takes twice as much time as a `FunctionBreakpoint`.

### 3.3.2 mcGDB OpenMP

The charts in Fig.3.3.1#OpenMP present the time it takes to pass the different OpenMP constructs. The chart #OpenMP/Nominal time is the time without debugger instruction, and the chart #OpenMP/mcGDB is the time with mcGDB event capture. The experimentation ran on a quad-core processor, so with 4 OpenMP threads/workers.

- The sections zone has three sections.
- The line single task stands for a task spawn from within a single construct.
- The figure indicated in the legend is the number of breakpoints that where hit (before+after). We did not include in that count the $16+5$ breakpoints required to handle the new threads and parallel zones.

From this chart, we can draw the following conclusions:

- The `master` construct is the fastest to pass, but also the simplest: the master threads (`get_id() == 0`) executes the block, the other continues.
- The `barrier` construct is also fast. There are 4 breakpoints that come from the preloaded library. They could have been disabled to improve the performance, as they are only useful in interactive mode. The $4+4$ other breakpoints correspond to the hit of the barrier function and its return.
- The construct `critical` and `single` (because of the `barrier`) also have these spurious stops in the current benchmark.
- We cannot explain why the `single+task` construct is faster than the `single` construct. Maybe a different handling inside OpenMP. The 3 additional breakpoints correspond to the task creation (1) and execution $(1+1)$.

The chart 3.3.1#All together combines the chars #Native GDB and Python and #OpenMP/mcGDB.

# *Chapter 4*

# *Conclusion*

In this document, we detailed the deliverable D3 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 task-based programming. This support consists of task-based execution representation and control improvements, in cooperation with Temanejo graphical debugger. We also discussed import implementation details of mcGDB, related to the support of multiple OpenMP environments and CPU architectures; the separation of cross-cutting concerns (user interaction and execution representing) through aspect-oriented programming, and the first steps of mcGDB micro-benchmarking.

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 and 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 [8] 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] 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.

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

[5] Rainer Keller, Steffen Brinkmann, José Gracia, and Christoph Niethammer. Temanejo: Debugging of thread-based task-parallel programs in starss. In *Tools for High Performance Computing 2011*. Springer Berlin Heidelberg, 2012.

[6] OpenMP Tools Working Group. OpenMP Technical Report 2 on the OMPT Interface. Technical report, OpenMP, 2014.

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

[8] 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 should 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 Ayudame/Temanejo

We use a development version of Ayudame/Temanejo, hence it build system is not finalized, and it does not support yet out-of-tree building. Temanejo only works with Python2, and furthermore, **Ayudame requires that GDB runs with Python2**. Otherwise, an `ImportError` will be raised, indicating that the symbol `PyInstance_Type` cannot be found in `$PYTHONPATH/ayudame/_ayu_socket.so`.

```
# install qt4/qtwebkit (qmake, libQtWebKit.so.4)
# install pyside (Resource Compiler for Qt version 4.8.7)
# install pyside-uic (PySide User Interface Compiler version 0.2.15,
#                     running on PySide 1.2.4.)
# install swig (3.0.7)
# install graphviz (2.38.0 (20140413.2041))
sudo pip2 install networkx pygraphviz

cd temanejo_mcgdb
TEMANEJO_HOME=$(pwd)
# install autoconf and automake
autoreconf -fiv

mkdir install
./configure --prefix=$(pwd)/install
cd ./Temanejo2/src/temanejo2/temanejo2/view/ \
   && pyside-uic mainwindow.ui -o mainwindow.py \
   && cd -
make && make install

# cd $TEMANEJO_HOME
cp -rv Temanejo2/src/temanejo2/resources/mcgdb-seqdiag/ \
       install/lib/python2.7/site-packages/temanejo2/resources/

cd $PYTHONPATH
ln -s $TEMANEJO_HOME/install/lib/python2.7/site-packages/ayudame
ln -s $TEMANEJO_HOME/install/lib/python2.7/site-packages/temanejo2

PATH=$TEMANEJO_HOME/install/bin/:$PATH

# and run Temanejo2
Temanejo2 -p 8888
```

### A.2. OpenMP Task Example

```
int main (void) {
  int i=0; int j=0; int k=0;
  int x=0; int y=0; int z=0;
#pragma omp parallel
    {
#pragma omp single
      {
#pragma omp task depend(in: i,j,k) depend(out:i,j,k)
        foo1(&i, &j, &k);
#pragma omp task depend(in:x,y,z) depend(out:x,y,z)
        foo1(&x, &y, &z);
#pragma omp task depend(in:i,y,z) depend(out:i,y,z)
        foo1(&i, &y, &z);
#pragma omp task depend(in:x,j,k) depend(out:x,j,k)
        foo1(&x, &j, &k);

        for ( i = 0; i < 2; ++i ){
#pragma omp task depend(in: i) depend(out: i)
          foo(&i);
#pragma omp task depend(in: j) depend(out: j)
          foo(&j);
#pragma omp task depend(in: k) depend(out: k)
          foo(&k);
#pragma omp task depend(in: x) depend(out: x)
          foo(&x);
#pragma omp task depend(in: y) depend(out: y)
          foo(&y);
#pragma omp task depend(in: z) depend(out: z)
          foo(&z);
        }
#pragma omp task depend(in:i,j,k) depend(out:i,j,k)
        foo1(&i, &j, &k);
#pragma omp task depend(in:x,y,z) depend(out:x,y,z)
        foo1(&x, &y, &z);
#pragma omp task depend(in:i,y,z) depend(out:i,y,z)
        foo1(&i, &y, &z);
#pragma omp task depend(in: x,j,k) depend(out: x,j,k)
        foo1(&x, &j, &k);
      }
    }
#pragma omp taskwait

    return 0;
}
```