TAFlow Timing Analysis Toolset
Install guide

João Lourenço, Tiago Carvalho, Luis Miguel Pinho

V0.5, July 2023

Index

1 Prerequisites and Initial notes

Before you start the configuration, we recommend having these requisites:

- Ubuntu (version 20.04 or higher)
- Clang (version 13 or higher)
- Python (version 3.8 or higher)

The usage of these auxiliary tools and our tools is not recommended in other Operative Systems, including WSL (Windows Subsystem for Linux), due to previous attempts to install/use them were not successful.

During this guide are expected some errors to occur, like needed library missing. To the errors that appeared previously to us, it will be given a solution to solve them.

In case you get an error that doesn't appear here, please inform us.

Much of this guide is sourced from the material provided by BSC-CNS. If you wish to read the original material, click here.

2 Setup

Before you start installing the auxiliary tools, it is recommended that you setup your environment by defining a directory for the source code and other for the tools itself. The following example and throughout the guide the environment variables and the paths used were selected due to other project, you are free to change them to your liking, just keep in mind to change them in the necessary steps of the guide.

```
export AMPERE_PATH=/opt/ampere

export AMPERE_SRC_PATH=/home/${USER}/ampere

export PATH=${AMPERE_PATH}/llvm/bin:${AMPERE_PATH}/extrae/bin:${AMPERE_PATH}/papi/bin:${PATH}

export LD_LIBRARY_PATH=${AMPERE_PATH}/llvm/lib:${LD_LIBRARY_PATH}
```

3 Extrae

3.1 PAPI

Before installing Extrae, we recommend installing the tool PAPI:

```
cd ${AMPERE_SRC_PATH}

git clone https://github.com/icl-utk-edu/papi.git

cd papi/src

./configure --prefix=${AMPERE_PATH}/papi

make -j 8

sudo make install
```

Note:

If you check the original source, the second step to install PAPI will have a different repository, but the one given by them is no longer available, it only has a README file asking you to use the repository given in this guide.

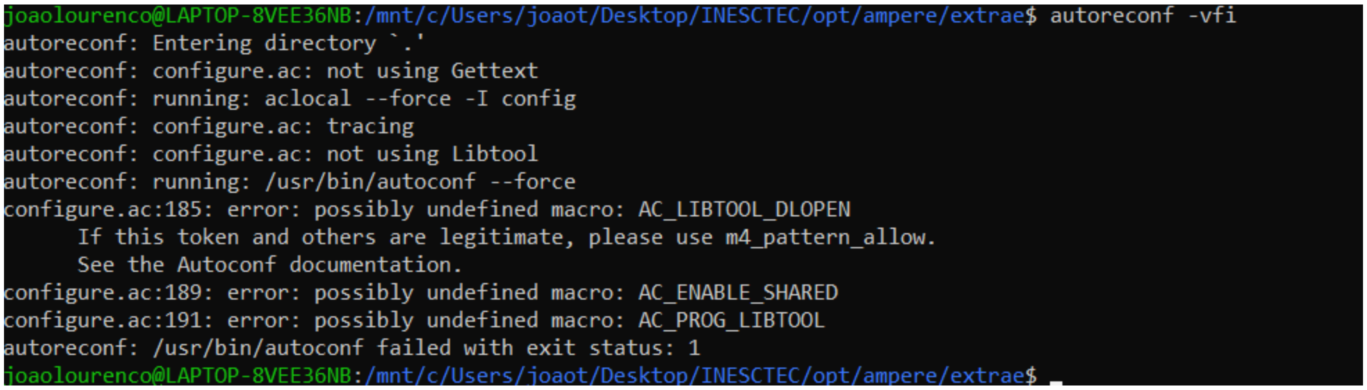
3.2 Extrae

After following the previous, you can now install Extrae:

```
cd ${AMPERE_SRC_PATH}

git clone https://gitlab.bsc.es/ampere-sw/wp2/extrae.git

cd extrae

autoreconf -vfi

./configure --prefix=${AMPERE_PATH}/extrae --without-mpi --without-
dyninst --without-unwind --with-papi=${AMPERE_PATH}/papi --enable-
openmp-intel

make -j4

sudo make install
```

Notes:

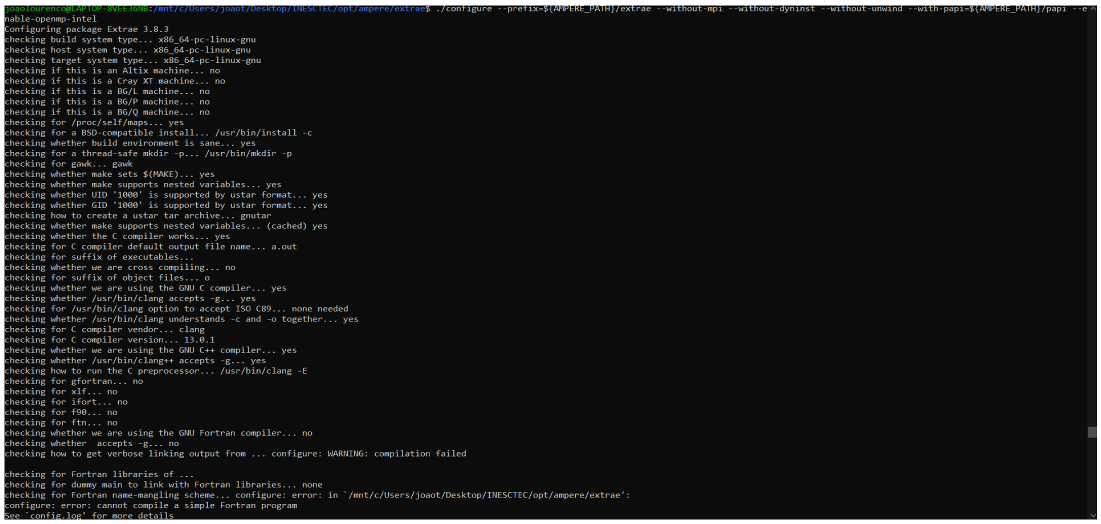
- Exit status 1 after executing “autoreconf -vfi”:



```
joaolourenco@LAPTOP-8VEE36NB:/mnt/c/Users/joaot/Desktop/INESCTEC/opt/ampere/extrae$ autoreconf -vfi
autoreconf: Entering directory `.'
autoreconf: configure.ac: not using Gettext
autoreconf: running: aclocal --force -I config
autoreconf: configure.ac: tracing
autoreconf: configure.ac: not using Libtool
autoreconf: running: /usr/bin/autoconf --force
configure.ac:185: error: possibly undefined macro: AC_LIBTOOL_DLOPEN
      If this token and others are legitimate, please use m4_pattern_allow.
      See the Autoconf documentation.
configure.ac:189: error: possibly undefined macro: AC_ENABLE_SHARED
configure.ac:191: error: possibly undefined macro: AC_PROG_LIBTOOL
autoreconf: /usr/bin/autoconf failed with exit status: 1
joaolourenco@LAPTOP-8VEE36NB:/mnt/c/Users/joaot/Desktop/INESCTEC/opt/ampere/extrae$ _
```

If “autoreconf -vfi” gives this error, try running "automake --add-missing". The previous command might need the libtool, in case you don’t have it run “sudo apt-get install libtool”. After you are sure that the libtool is installed run "automake --add-missing", run “autoreconf -vfi” again after and you can continue to the following steps.

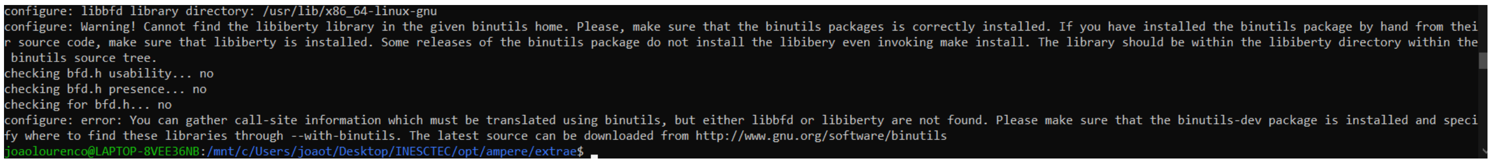
- Executing the step “./configure {...} intel ” says a Fortran compiler was not found:



```
joaolourenco@LAPTOP-8VEE36NB:/mnt/c/Users/joaot/Desktop/INESCTEC/opt/ampere/extrae$ ./configure --prefix=${AMPERE_PATH}/extrae --without-mpi --without-dyninst --without-unwind --with-papi=${AMPERE_PATH}/papi --e
nable-openmp-intel
Configuring package Extrae 3.8.3
checking build system type... x86_64-pc-linux-gnu
checking host system type... x86_64-pc-linux-gnu
checking target system type... x86_64-pc-linux-gnu
checking if this is an Altix machine... no
checking if this is a Cray XT machine... no
checking if this is a BG/L machine... no
checking if this is a BG/P machine... no
checking if this is a BG/Q machine... no
checking for /proc/self/maps... yes
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /usr/bin/mkdir -p
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking whether make supports nested variables... yes
checking whether UID '1000' is supported by ustar format... yes
checking whether GID '1000' is supported by ustar format... yes
checking how to create a ustar tar archive... gnutar
checking whether make supports nested variables... (cached) yes
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether /usr/bin/clang accepts -g... yes
checking for /usr/bin/clang option to accept ISO C89... none needed
checking whether /usr/bin/clang understands -c and -o together... yes
checking for C compiler vendor... clang
checking for C compiler version... 13.0.1
checking whether we are using the GNU C++ compiler... yes
checking whether /usr/bin/clang++ accepts -g... yes
checking how to run the C preprocessor... /usr/bin/clang -E
checking for gfortran... no
checking for xlf... no
checking for ifort... no
checking for f90... no
checking for ftn... no
checking whether we are using the GNU Fortran compiler... no
checking whether  accepts -g... no
checking how to get verbose linking output from ... configure: WARNING: compilation failed

checking for Fortran libraries of ...
checking for dummy main to link with Fortran libraries... none
checking for Fortran name-mangling scheme... configure: error: in `/mnt/c/Users/joaot/Desktop/INESCTEC/opt/ampere/extrae':
configure: error: cannot compile a simple Fortran program
See `config.log' for more details
```

Run the command “sudo apt install gfortran” and repeat the step you were in.

- binutils and libiberty missing:



```
configure: libbfd library directory: /usr/lib/x86_64-linux-gnu
configure: Warning! Cannot find the libiberty library in the given binutils home. Please, make sure that the binutils packages is correctly installed. If you have installed the binutils package by hand from thei
r source code, make sure that libiberty is installed. Some releases of the binutils package do not install the libibery even invoking make install. The library should be within the libiberty directory within the
 binutils source tree.
checking bfd.h usability... no
checking bfd.h presence... no
checking for bfd.h... no
configure: error: You can gather call-site information which must be translated using binutils, but either libbfd or libiberty are not found. Please make sure that the binutils-dev package is installed and speci
fy where to find these libraries through --with-binutils. The latest source can be downloaded from http://www.gnu.org/software/binutils
joaolourenco@LAPTOP-8VEE36NB:/mnt/c/Users/joaot/Desktop/INESCTEC/opt/ampere/extrae$
```

Run the commands “sudo apt install binutils-dev” and “sudo apt install libiberty-dev”. After that repeat the step you were in.

- “omp.h” not found error:

If you get this error, run the command “sudo apt-get install libomp-dev” and run again the step you were in.

4 BSC's LLVM

Before installing BSC'S LLVM, you will need the ninja build system (in case you don't have it) and define the environment variables for C and C++ compilers, we remind you that the paths used may differ from yours:

```
sudo apt-get install ninja-build

export CC=/usr/bin/clang

export CXX=/usr/bin/clang++
```

We can now proceed to install BSC'S LLVM:

```
cd ${AMPERE_SRC_PATH}

git clone -b omp-taskgraph --depth 1 https://gitlab.bsc.es/ampere-sw/wp2/llvm.git

cd llvm

mkdir build

cd build

cmake -DLLVM_ENABLE_PROJECTS="clang;openmp" -DCMAKE_BUILD_TYPE=MinSizeRel -G "Ninja" -
DCMAKE_INSTALL_PREFIX=${AMPERE_PATH}/llvm -DLIBOMP_TASKGRAPH=1 ../llvm

ninja install
```

5 Bash Setup

After installing every tool, it will be convenient to define in the shell script some environment variables that are useful, so they are defined automatically every time a new terminal is used.
Open a new terminal in your code editor and type the following command:

```
code ~/.bashrc
```

After running, a file should pop up. Scroll it all the way down and add the following code (remember to change the first 2 paths if needed):

```
export AMPERE_PATH=/opt/ampere
export AMPERE_SRC_PATH=/home/${USER}/ampere
export EXTRAE_HOME=${AMPERE_PATH}/extrae
export PAPI_PATH=${AMPERE_PATH}/papi

export PATH=${AMPERE_PATH}/llvm/bin:${EXTRAE_HOME}/bin:${PAPI_PATH}/bin:${PATH}
export LD_LIBRARY_PATH=${AMPERE_PATH}/llvm/lib:${LD_LIBRARY_PATH}

export LD_LIBRARY_PATH=${AMPERE_PATH}/papi/lib:${LD_LIBRARY_PATH}

export CC=${AMPERE_PATH}/llvm/bin/clang
export CXX=${AMPERE_PATH}/llvm/bin/clang++

export OMP_PATH=${AMPERE_PATH}/llvm/lib

export LD_LIBRARY_PATH=${EXTRAE_HOME}/lib:${LD_LIBRARY_PATH}
export EXTRAE_SKIP_AUTO_LIBRARY_INITIALIZE=1

export OMP_NUM_THREADS=8
export EXTRAE_CONFIG_FILE=./extrae.xml
```

6 Our tools

The following chapter is for those who which to use and understand better the tools we provide besides the previous auxiliar ones, explaining the correct way to use it depending on your purpose. If your goal was only to install the previous tools, you can skip this.

6.1 Module

Installation

If your goal is to use as a module, the easiest way to use is by adding it to the dependencies of your python project, which can be done by directly specifying the git repository as a dependency:

```
$ pip install git+https://gitlab.bsc.es/ampere-sw/WP3/time-predictability.git
```

The project is automatically built, including the command line tools as explained in the following section. Therefore, this approach works for both python module and for the command line tools.

Usage

Assuming that this project is already in the dependencies of your project, at least two things are necessary: the meta-parallel module that contains classes for the TDG structure, and the module(s) of the tool(s) to be used.

For the meta-parallel module, you need to import the classes to be used. The most important one, and potentially the only one you might need to use, is the TDG class. This class expects a tdg.json and builds an instance of TDG, which will be used by all the other modules as an input:

```
from time_predictability.meta_parallel.tdg import TDG
(...)
tdg = TDG.read_json(input_tdg)
```

Then you need to import the tool to be used and follow the instructions in the subsections below.

Time analysis

The time analysis module is used to populate a TDG with metrics related to timing analysis of the parallel graph. The first step to use this module is to import it. E.g.:

```
from time_predictability.time_analysis.analysis import TDGAnalysis
```

Then, one can use a class that calculates a set of metrics and automatically annotates the TDG. The simplest use of this class is as follow:

```
analysis = TDGAnalysis(tdg,num_threads)
analysis.analyse()
output_json = analysis.tdg_to_json()
```

An instance has to be created with the TDG as input and a number of threads. This number of threads represent the number of threads available in the target system, in order calculate some of the metrics (e.g. the WCRT). Then, the "analyse" function will calculate the metrics that are active by default. The last line regenerates the json that includes the calculated metrics. Note that during the execution of the "analyse" function is invoked, the tdg object is immediately annotated with the metrics, so it can be used in later steps.

The following table contains a list of the available metrics, a brief description, and if they are active by default:

Key	Name	Description	Level	Active
Wcet	Worst case execution time	Max execution time observed	Task	X
Pmc	Performance counter metrics	Metrics related to each pmc read	Task	X
Volume	Volume	Total volume of the TDG	TDG	X
Cpl	Critical path length	Cost of the critical path	TDG	X
Max_par	Maximum parallelism	Maximum possible level of parallelism	TDG	X
Depth	Depth	Maximum depth of the TDG	TDG	X
Makespan	Makespan	Execution time from the source task to the sink task. Requires all task to be annotated with ´static_thread´	TDG	
Wcrt	Worst case response time	Calculates an upper bound of the excepted worst execution time of the TDG. It presents the results for tied and untied tasks. If the calculated metric is worst than the volume, then the volume is given.	TDG	X

To activate and deactivate metrics, one can create a list with a set of 'keys' (from the table) and use the following functions over the TDGAnalysis instance:

```
to_include = ['makespan']
analysis.include(to_include)
to_exclude = ['wcrt','depth']
analysis.exclude(to_exclude)
```

Note that by including a list of metrics does not deactivate the others. If one desires, for instance, to only include one metric, let's say the 'wcet' of each task, then the function 'exclude_all' can be used for this end, as in the example below:

```
analysis.exclude_all()
to_include = ['wcet']
analysis.include(to_include) #only wcet will be active
```

There is also a feature that allows one to remove the "results" property (that comes with the TDG) to clean up the output TDG:

```
(...)
analysis.analyse()
analysis.clear_results()
output_json = analysis.tdg_to_json()
```

This example will generate the json containing the metrics, but without the results.

Mapping Simulator

This tool allows the simulation of OpenMP thread mapping algorithms over a target TDG, outputting the expected mapping of that algorithm, and the corresponding makespan. It uses the timing analysis tool to extract the WCET of each task and simulates the execution of the TDG, where the task to thread mapping is responsibility of the selected algorithm.

The most important imports for this tool are the following:

```
from time_predictability.simulation.scheduler import MasterQueue
from time_predictability.simulation.simulator import Simulation, Simulator
```

The first one is to build and instantiate the mapping algorithm, based on set of heuristics. The second ones are the simulator itself and the class that will contain the result of the simulation.

To build the algorithm, one can use the function 'get_algorithm_by_name', which requires the name of the heuristic for the task2thread mapping, the name of the heuristic for the queue, a boolean that says if this algorithm is to be used with a single queue (false) or a queue-per-thread approach, and the number of threads available for the simulation. The function returns a lambda that, when invoked, creates an instance of the algorithm.

For task2thread, currently the tool contains the following algorithms:

Key	Description	Queue per thread
BestFit	Selects the queue with least execution time	Yes
BFS	Breadth-first schedule (OpenMP)	No
SEQR	Sequential Code, replicas in different threads	Yes

For the allocation queue, currently the tool contains the following algorithms:

Key	Description
BestFit	Selects the task with lower WCET
FIFO	First-in-first out queue

Then, we can use the Simulator class to create a simulation with an input TDG and the mapping, algorithm, and use the "simulate" method to execute the simulation. The following shows an example of how to use the simulator with the BestFit algorithm for task2thread, with a FIFO queue:

```
useMultiQueues = True
master = MasterQueue.get_algorithm_by_name('BestFit', 'FIFO', useMultiQueues, num_threads)
simulator:Simulator = Simulator(tdg, master(), num_threads)
simulation:Simulation = simulator.simulate()
```

After the simulation executes, a 'Simulation' class instance is returned, which contains the mapping computed during the simulation, organized both by task and by thread. In sequence, this mapping can be used over the TDG to statically define the threads for each task. The following is an example of using these two functionalities:

```
static_map = simulation.get_mapping()
makespan = simulation.makespan
tdg.set_static_threads(static_map,makespan)
```

Each task in the TDG has now the static_thread property defined with the corresponding 'thread' in the static_map. We can now save the TDG with the given mapping in a file:

```
output_tdg = 'output_tdg.json'
out_json = tdg.to_json(put_results=False)
with open(output_tdg, 'w') as outfile:
        json.dump(out_json, outfile, indent=2)
```

The 'tdg.to_json' method converts the TDG again to JSON format (a string), where the 'put_results' argument is simply used to remove the results that might exist inside the task.

Mapping Exploration

The mapping exploration API uses the two previous APIs as its basis and builds an exploration environment to try and find the best task to thread mapping, based on a list of algorithms to make the mapping. To start, two classes have to be imported:

```
from time_predictability.mapping_exploration.explorer import ExplorationConfig,
MappingExploration
```

The exploration requires two inputs: the target TDG and an exploration configuration. The configuration defines the number of available threads, the deadline, and the list of algorithms to use. The following is an example of creating the exploration with three algorithms and the execution of the exploration:

```
tdg:TDG = TDG.read_json(input_file)
num_threads = 4
deadline = 2000000
exp_config:ExplorationConfig =
ExplorationConfig(num_threads,deadline)
exp_config.add_algorithm('BestFit','BestFit',true)
exp_config.add_algorithm('BestFit','FIFO',true)
exp_config.add_algorithm('BFS','FIFO',false)

explorer:MappingExploration = MappingExploration(tdg,exp_config)
success = explorer.explore()
if success:
        explorer.apply_map()
        out_json = tdg.to_json(put_results=(not clear_results))
        with open(output_file, 'w') as outfile:
                json.dump(out_json, outfile,indent=2)
```

It is possible to see in the example that almost at the end we request the exploration with the 'explore' method, which returns a boolean. This boolean tells if the exploration was successful or not, i.e., if the exploration was able to find a mapping within the deadline. If so, by using the "apply_map" method it is possible to apply the best static mapping, obtained from the exploration, in the TDG, defining for each task its 'static_thread' property. The rest of the code is common procedure to convert the TDG (now annotated with a static mapping) into a JSON-format string and output it to a file.

6.2 Command line tool

Installation

The first step is to clone this project. Then, one can either create a virtual environment (to keep the project contents contained in a controlled environment) or simply jump this step to the next one and install the command line tools global to the system.

Suggestion

To install this tool we suggest that a python virtual environment is used to keep the dependencies of the project localized. Inside the project folder run the following commands:

If in windows:

```
> python3 -m venv <env_name>
> <env_name>\Scripts\activate
```

In in Linux:

```
$ python3 -m venv <env_name>
$ source <env_name>/bin/activate
```

This will then put your command line with the format:

```
(env_name) \my\current\dir>

Or

(env_name) /my/current/dir$
```

This means that all the Note that whenever you open a new terminal you will have to run again the command **source <name_for_the_env_folder/bin/activate** to again enter the environment variable. In the examples below a virtual environment named '.venv' is used. Therefore, every time an example of the command line appears with **(.venv) /a/path$** means that we are inside that environment.

Now we have to install the tools, including its dependencies. Since we are inside a virtual environment, the dependencies will only be installed in this environment, and not globally, thus not polluting the system.

Just run the following command and all dependencies are installed, and the executables are created and added to this environment.

```
$ python setup.py build
```

Usage

Time analysis

To execute as a command line, after installation, one can run the analysis-tool without argument to obtain the command options:

```
(.venv) my/work/place$ time-analysis
usage: [options] <input tdg> <output tdg>
     ---------------------------------------------------------------------------
    | option | argument             | description                                  |
     --------|--------------------|---------------------------------------------
    | -h     | n/a                  | show this message                            |
    | -t     | <num_threads>        | specify max number of threads                |
    | -l     | n/a                  | list metrics                                 |
    | -x     | <metric(,metric)*>   | set metrics that should not be calculated    |
    | -s     | <metric(,metric)*>   | set metrics that should be calculated        |
    | -a     | <metric(,metric)*>   | append metrics that should be calculated     |
    | -c     | n/a                  | remove 'results' property from tasks         |
     ---------------------------------------------------------------------------
```

The tool requires an input tdg.json file, the output file location, and a set of options that define the number of threads (which defaults to 1) and the metrics to calculate. Many of the metrics are already active. To see the available metrics, and if they are active, the -l option can be used to print. To control which ones are active or inactive three options can be used: -x, -s and -a, with a list of keys (from the table) separated by a comma. The -x option deactivates the given list of metrics, -a activate the listed metrics (but does nothing with the others), and -s activates ONLY the ones specified in the given list.

The -c options makes that the output json, regardless of the calculated metrics, does not include the results in each task. It serves as a way to have a clean TDG, where only the metrics are listed.

Mapping Simulator

To execute as a command line, after installation, one can run the map-exploration without argument to obtain the command options:

```
(.venv) my/work/place$ map-simulator
usage: [options] <input tdg> <output tdg>
     ----------------------------------------------------------------------------------
    | option | argument            | description                                      |
     --------|---------------------|---------------------------------------------------
    | -h     | n/a                 | show this message                                |
    | -l     | n/a                 | list heuristics                                  |
    | -n     | <num_threads>       | specify number of threads available (def: 1)     |
    | -t     | <task2thread>       | specifies the scheduling algorithm (def: BFS)    |
    | -q     | <allocation_queue>  | list metrics that should be calculated (def: FIFO) |
    | -m     | n/a                 | use one queue per thread (instead of single queue) |
    | -c     | n/a                 | remove 'results' property from tasks             |
     ----------------------------------------------------------------------------------
```

The tool requires an input tdg.json file, the output file location, and a set of options that define the number of threads (which defaults to 1) and the heuristics to use for each phase of the mapping algorithm. To see the available heuristics one can use the -l option to print them. Use -t to specify the heuristic for the task to thread mapping phase and -q to specify the heuristic for the queue. The -m option is used to specify that the algorithm should work with multiple queues, one per queue (if not used then just one queue is used for all the system). Take into account that some of the algorithms do not work with single/multiple algorithms. Please see the table above to see if the algorithm uses multiple queues or not.

The -c options makes that the output json, regardless of the calculated metrics, does not include the results in each task. It serves as a way to have a clean TDG, where only the metrics are listed.

Mapping Exploration

To execute as a command line, after installation, one can run the map-exploration tool without argument to obtain the command options:

```
(.venv) my/work/place$ map-exploration
usage: [options] <config.json>
     -=================================== Options ===========================-
    | option | argument  | description                                      |
     --------|-----------|--------------------------------------------------
    | -h     | n/a       | show this message                                |
    | -l     | n/a       | list heuristics                                  |
    | -c     | n/a       | remove 'results' property from tasks             |
     -======================================================================-


     -====================== Configuration File Properties ===================================-
    | property             | argument       | description                                     |
     ---------------------|---------------|-------------------------------------------------
    | input                | <tdg.json>     | input tdg json file                             |
    | output               | <tdg.json>     | output tdg json file                            |
    | num_threads          | int            | number of threads for the simulator             |
    | deadline             | int            | maximum makespan the TDG can have               |
    | map_algorithms       | [algorithms+]  | list of mapping algorithm specifications        |
     -=======================================================================================-
     -=========================== For each algorithm ==========================================-
    | property             | argument       | description                                     |
     ---------------------|---------------|-------------------------------------------------
    | task2thread          | <heuristic>    | one of the available heuristics for task2thread |
    | queue                | <heuristic>    | one of the heuristics for allocation queue      |
    | queue_per_thread     | true|false     | specifies if the algorithm has one queue per thread|
     -=======================================================================================-
```

The tool requires as input a configuration file which contains the parameters necessary for the exploration, more specifically: the input and output TDG files, number of threads, the deadline, and a list of algorithms to use for the exploration. Regarding the algorithms, they can be a combination of the heuristics defined in this table. The -l option can also be used to see these heuristics. A configuration file looks something like the following example:

```
{
    "input": "examples/fixtures/tdgs/heat/heat_extrae_freq_2188.json",
    "output": "results/heat_extrae_freq_2188.json",
    "num_threads": 4,
    "deadline": 280000000,
    "map_algorithms":[
        {
            "task2thread": "BestFit",
            "queue": "BestFit",
            "queue_per_thread": true
        },
        {
            "task2thread": "BestFit",
            "queue": "FIFO",
            "queue_per_thread": true
        },
        {
            "task2thread": "BFS",
            "queue": "FIFO",
            "queue_per_thread": false
        },
        {
            "task2thread": "SEQR",
            "queue": "FIFO",
            "queue_per_thread": true
        }
    ]
}
```

The tool is then going to process the input TDG, experiment with the different algorithms and, at the end, output a TDG file with the best static mapping found. If no static mapping provides a makespan less than the given deadline, then the output TDG is not generated, as no fitting mapping was found with the provided parameters. The following is an example of running the tool with the previous configuration file:

```
(.venv) my/work/place$ map-exploration -c
examples/fixtures/exp_configs/heat/heat_freq_2188.json
        algorithm BestFit(BestFit) is within deadline:  278122897 <= 280000000
        algorithm BestFit(FIFO) is within deadline:  276962046 <= 280000000
        algorithm BFS(FIFO) is within deadline:  271898301 <= 280000000
        algorithm SEQR(FIFO) is not within deadline:  992947740 > 280000000
The best mapping found is BFS+FIFO with a makespan of 271898301
```

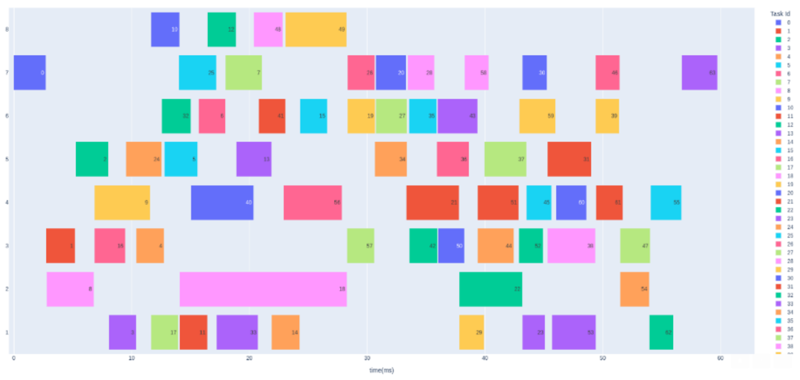
6.3 Graphics generation

Our tools also enable to visualize any time the data that as introduced and is being worked on, which provides an easier way to analyse it and compare to previous results.

For such purpose, there are available the following:

```
def results_map_view():
    args = sys.argv[1:]

    if len(args) < 2: #3
        print(results_map_view_message)
        return
    input_tdg = args[0]
    range_str = args[1]
    # output_png = args[2]
    with open(input_tdg,'r') as json_file:
        tdg_results:Dict = json.load(json_file)

    app_name,tdgs = next( item for item in tdg_results.items() if
item[0] != END_TO_END_DEADLINE)
    # print(app_name,tdgs)

    range_params = range_str.split(":")
    range_init = int(range_params[0])
    range_fini = range_init+1
    range_step = 1
    if len(range_params) > 1:
        range_fini = int(range_params[1])
        if len(range_params) > 2:
            range_step = int(range_params[2])
    results_range = range(range_init,range_fini,range_step)
    for tdg in tdgs:
        MappingViewer.show(tdg[NODES], specific_range=results_range)
```

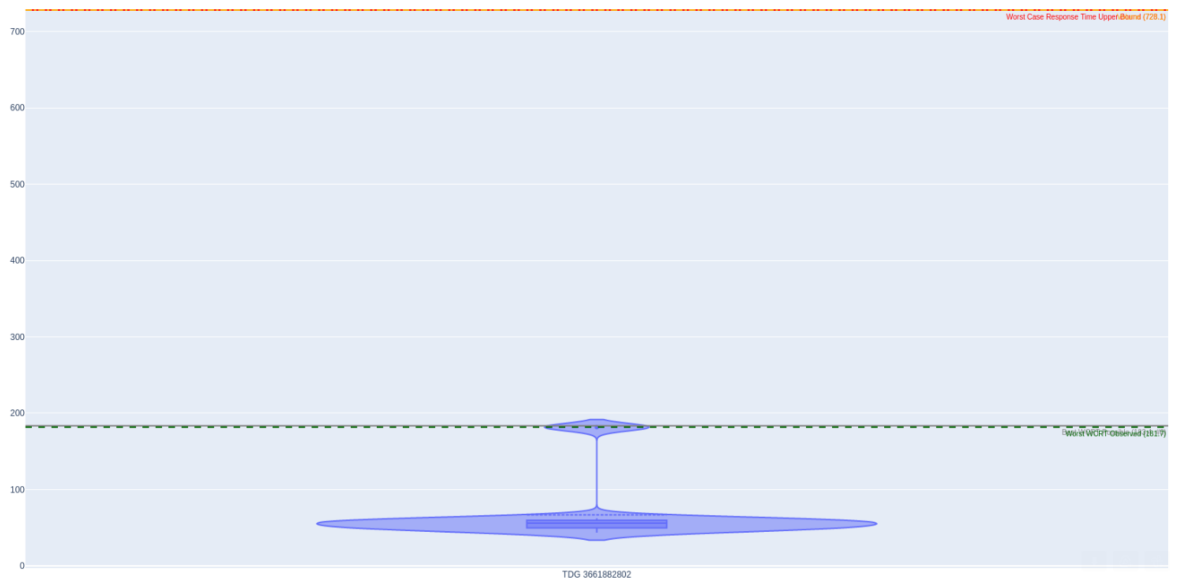
Any time you want, you can use the command **results-map-view <input.json> <range as: init[:fini[:step]]>** to visualize the mapping of the results obtained by running the application. This tool works without the execution of any time analysis or map exploration, using the values given by Extrae:



This tool can also be executed together with the previous ones, allowing a more fluid usage of the tools we provide.

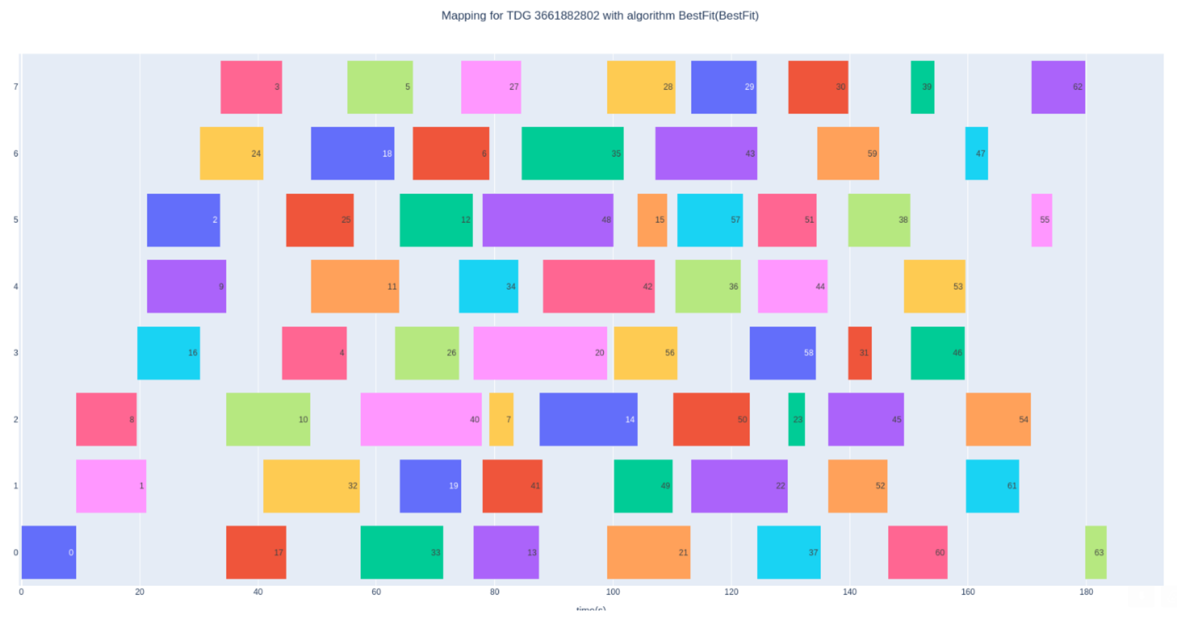
Time analysis

By running the time-analysis command with the flag "-g", you will be able to see the corresponding violin plot:



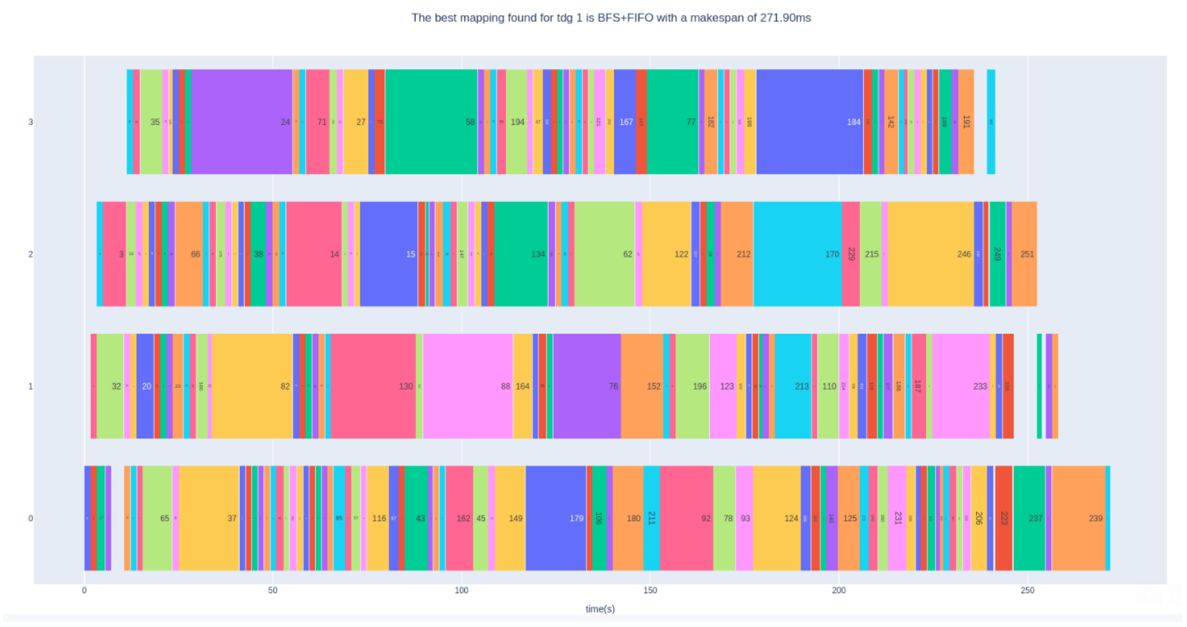
Map simulator

By running the map-simulator command with the flag "-g", you will be able to visualize the mapping:



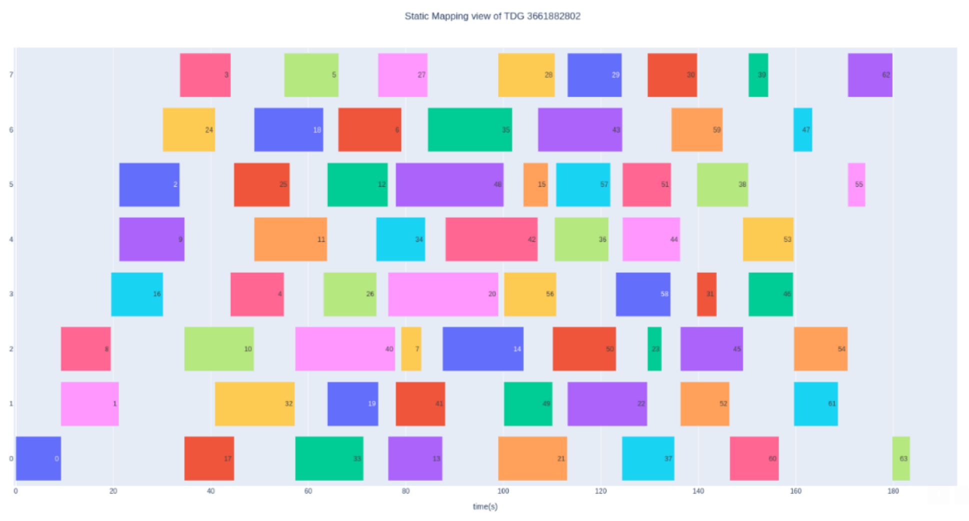
Map exploration

By running the mapping-exploration command with the "-g" flag, you will be able to see all defined static mappings for each TDG and it will also give you the chosen mapping, telling its respective makespan:



Static mapping

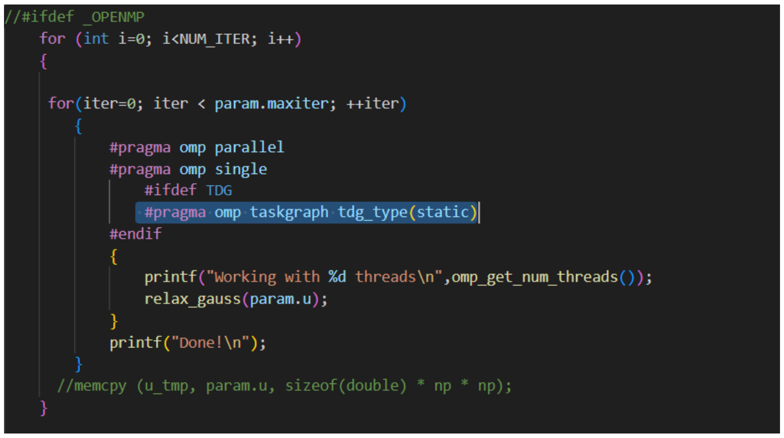
If you have any json file with static_mapping assigned, you can use **static-map-view** that will show you the expected execution of the corresponding TDG:



7 Workflow

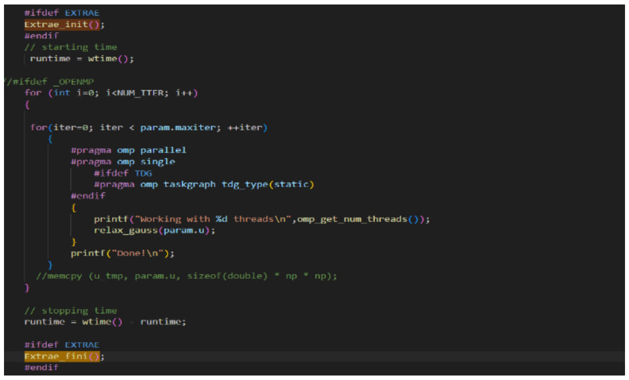
7.1 Pre requirements

1. Build task based OpenMP code.
   1.1. with #pragma omp taskgraph tdg_type(static) in parallel regions to generate dot/tdg files



```
//#ifdef _OPENMP
    for (int i=0; i<NUM_ITER; i++)
    {

     for(iter=0; iter < param.maxiter; ++iter)
        {
            #pragma omp parallel
            #pragma omp single
                #ifdef TDG
                #pragma omp taskgraph tdg_type(static)
            #endif
            {
                printf("Working with %d threads\n",omp_get_num_threads());
                relax_gauss(param.u);
            }
            printf("Done!\n");
        }
      //memcpy (u_tmp, param.u, sizeof(double) * np * np);
    }
```

2. Instrument code with Extrae (Extrae_init/Extrae_fini) and build/copy extrae.xml file



```
    #ifdef EXTRAE
    Extrae_init();
    #endif
    // starting time
     runtime = wtime();

//#ifdef _OPENMP
    for (int i=0; i<NUM_ITER; i++)
    {

     for(iter=0; iter < param.maxiter; ++iter)
        {
            #pragma omp parallel
            #pragma omp single
                #ifdef TDG
                #pragma omp taskgraph tdg_type(static)
            #endif
            {
                printf("Working with %d threads\n",omp_get_num_threads());
                relax_gauss(param.u);
            }
            printf("Done!\n");
        }
      //memcpy (u_tmp, param.u, sizeof(double) * np * np);
    }

    // stopping time
    runtime = wtime() - runtime;

    #ifdef EXTRAE
    Extrae_fini();
    #endif
```

3. Two possible ways to buid:
   3.1. normal build (dynamic mapping): builds .o files and .exe with dynamic mapping.
   3.2. static mapping: builds exe with precompiled .o files and a static mapping (in tdg.cpp file).

7.2 Profiling application

1. Compile code with dynamic mapping (i.e., without tdg.cpp file).
   1.1. important to have the .o file for later step (static mapping).
   1.2. this also generates dot files that will be used to generate json file.
2. Run dot2json command to convert dot files into a TDG.json file.
3. For X iterations:
   3.1. run application.
   3.2. execute the python script that copies prv contents into json file:

```
python3 ${AMPERE_SRC_PATH}/tdg-instrumentation-script/parsePrvAndTdg.py TDG.json *.prv *.pcf
```

3.2.1. This will populate the tdg.json file with new results.

4. The tool results-map-view can be used to visualize the mappings of the results.

7.3 Time analysis

1. Run the time-analysis tool to populate the json file with metrics.
   1.1. if ran with -g one can see a violin plot of the results.
   1.2. the -c option can be used to clear the results property of all nodes (to clean the json file).

7.4 Map exploration

1. Run the map-exploration tool to get the best static mapping.
   1.1. if ran with -g one can see the several mappings applied and the selected best static mapping.
2. One can also use the static-map-view to see the static-mapping defined in the TDG (will not work if any node has static_mapping = -1).
3. the json2cpp tool can be used to generate the cpp file with the static mapping, which can be used to recompile the code with a static mapping.

7.5 Analysis of the static mapping

1. recompile the code with the static mapping generated with the json2cpp tool.
2. run time-analysis again to see the new metrics.

8 Final notes

This guide is still in development, changes may occur to adapt to the feedback we receive, so anything that you think it could be improved or worth mentioning, please inform us.

9 Appendix

Setup.py – sets up the packages for time-predictability, defining the versions, the requirements and license information. Also specifies entry points for other available console scripts, which can be used as commands.

Cmd.py – contains different scripts, responsible for different tasks such as time analysis, mapping exploration and conversion from DOT files to JSON:

- Dot2json – This script is responsible for converting TDG represented in DOT files to the JSON format.
- Amxmi2json – This script is responsible for converting an AMXMI model to the JSON format.
- Time_analysis – Performs time analysis on a given TDG specified in JSON format, having various options to be specified (i.e. number of threads).
- Map_simulator – Simulates the map of tasks to threads of a TDG represented as a JSON file, allowing the usage of certain algorithms.
- Map_exploration – Performs mapping exploration for TDG represented in the JSON format, exploring different mapping options in order to find the best option that satisfies the deadline.

Analysis.py – Example of how to use the time analysis tool with default metrics.

Simulation.py – Example of how to use the simulation tool with an algorithm.

Exploration.py – Example of how to use the mapping exploration tool.

Analysis_complex.py - Second example of how to use the time analysis tool with default metrics.
license information