Date: June 22, 2018
Version: 1.0

MATE USER'S GUIDE

CAOS (UAB)
June 22, 2018

COMPUTER ARCHITECTURES
AND OPERATING SYSTEMS DEPARTMENT
AUTONOMOUS UNIVERSITY OF BARCELONA

Email: gr.hpca4se@uab.cat
Web: http://grupsderecerca.uab.cat/hpca4se/en
https://github.com/HPCA4SE-UAB

Index

1 Introduction

The aim of this document is to explain the essential steps to configure, install and execute MATE [2, 3, 4]. MATE needs two programs, the Analyzer and AC, to work in conjunction with the DMLib shared library. In figure 1 we can see how they interact with each other:

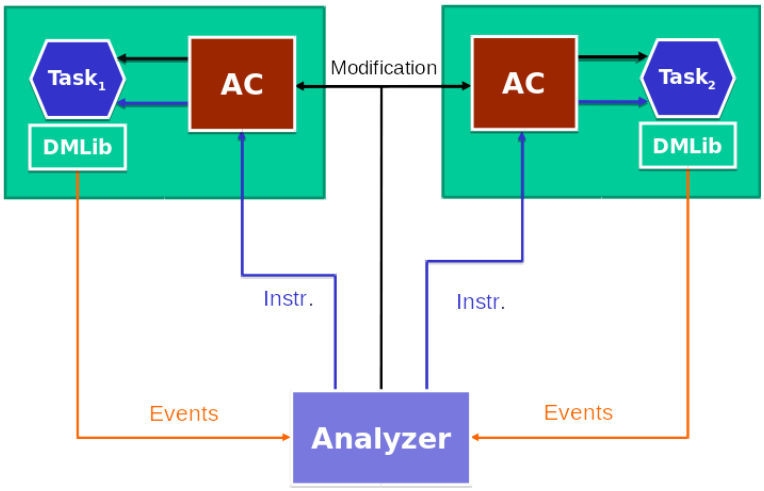

Figure 1: MATE structure

Thus, MATE is composed of the following modules which cooperate to control and improve the application's performance:

- The **Application Controller** (AC) is a daemon that controls the execution and the dynamic instrumentation of each individual MPI task.
- The **Analyzer** is a centralized process that carries out the application performance analysis, and decides on the monitoring and tuning. It automatically detects existing performance problems on the fly and requests appropriate changes to improve the application's performance.
- The **Dynamic Monitoring Library** (DMLib) is a shared library that is dynamically loaded by the AC in the application tasks to facilitate collecting data and delivering it to the Analyzer

The knowledge required to perform analysis and tuning is encapsulated in MATE in a piece of software called a *tunlet* which will be explained more in detail in section 5. This document is composed of 4 main sections: MATE installation, configuration files, a full execution example and the steps to create a tunlet.

2 MATE installation

2.1 Dependencies and build

First off, MATE needs some dependencies to be installed before building it. Those are the following:

- **Dyninst** - The most critical dependency for MATE is Paradyn's Dyninst API. This library is responsible for inserting the instrumentation to the application that will be monitored. MATE is programmed and tested to work with versions >7.0.1, being v9.3.2 the latest version tested with. Dyninst can be downloaded from `https://github.com/dyninst/dyninst` [1].
- **MPI** - MATE is currently implemented for MPI applications and therefore, it requires some installed version. The latest MPI installation and version tested with MATE was OpenMPI v1.10.2 .
- **PAPI** - The latest tested version of PAPI along with MATE was v5.6.1. It can be downloaded from `http://icl.cs.utk.edu/papi/`
- **Other dependencies** - MATE also needs the following libraries:
  - **Boost** - The latest tested version of Boost was v1.62.0 . It can be downloaded from `https://www.boost.org/`
  - **libelf & libdwarf** - These two libraries can be downloaded from `https://sourceforge.net/p/elftoolchain/wiki/Home/`
  - **libiberty** - It is usually installed along with gcc and it can be downloaded from `https://gcc.gnu.org/onlinedocs/libiberty/`

Go to `https://github.com/HPCA4SE-UAB/MATE` to download MATE once all dependencies have been cleared. The building of MATE is similar to any UNIX-like system installation:

```
$ ./configure --with-dyninst=<DYNINST_DIR> --prefix=<PREFIX>
$ make && make install
```

For details on the features that can be input to the configure step, see the annex in section 6 or do:

```
$ ./configure --help
```

Additionaly, **make** has four targets explained below:

- **make** - Compiles the application and puts the binaries of Analyzer and AC in their corresponding folders.
- **make install** - Installs MATE in the prefix chosen (by default /lib and /bin).
- **make clean** - Deletes the compiled files generated in the make step.
- **make doc** - Creates the documentation of MATE in ./doc using Doxygen .

The tree structure that MATE uses can be seen in Figure 2.

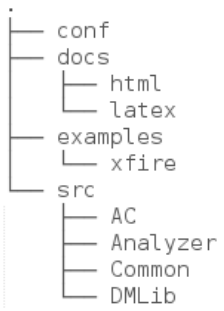

Figure 2: MATE folder tree

3 Configuration files

After the installation process and depending on the application to monitor, you must take care of MATE's configuration files. This step is necessary to make MATE work in conjunction with the MPI application. There are four configuration files that have to be modified. Examples of these can be found in the root directory under the *conf/* folder:

- **AC.ini:** configures the **Application Controller**,
- **Analyzer.ini:** configures the **Analyzer**,
- **Tunlet.ini:** configures the **tunlet** and
- **DMLib.ini:** configures the DMLib's *syslog* options.

In general, a MATE configuration file is composed of several *sections* with their own parameters and should look like the following:

```
[section_name]
parameter_identifier = # user_provided_value
...
[another_section_name]
parameter_identifier = # user_provided_value
...
```

The details for each of these files are shown in the following subsections.

3.1 Application controller - AC.ini

The AC.ini file is where the AC configuration is placed. This file requires the sections: **AC**, **Analyzer** and **Syslog** detailed below:

[AC]

The **AC** section specifies parameters associated to the application controller executable:

```
[AC]
ACPath = # Application Controller path
DMLib =  # libDMLib.so library path
PTPAcceptorPort = # Analyzer waiting event port
```

[Analyzer]

The **Analyzer** section is where the Analyzer parameters are specified:

```
[Analyzer]
Host = # Host where the Analyzer is executing
Port = # Analyzer port
```

[Syslog]

The **Syslog** section is where the debug parameters have to be specified:

```
[Syslog]
MasterSwitch = # true/false
LogLevel =
    # 0 - To show all debug messages in log file.
    # 1 - To show INFO messages.
    # 2 - To show no messages.
StdErrLogLevel = # 0/ 1/ 2 the same as LogLevel,
# but the message will be printed in terminal
StdErr =  # true/false to activate the information output
LogFile = # log file name
LogPath = # log file path
AppendMachineName = # true/false
AppendFile = # true/false
Prefix = # prefix to use when printing information
```

3.2 Analyzer - Analyzer.ini

The Analyzer configuration file requires the sections: **Analyzer**, **EventCollector** and **Syslog**:

[Analyzer]

This section only requires the boolean parameter *DisableTuningActions* to be set. As its name suggests, it will disable the tuning action if set to *true*:

```
[Analyzer]
DisableTuningActions = # true/false
```

[EventCollector]

This section defines the Analyzer port to use (where it receives the application metrics from libDMLib.so):

```
[EventCollector]
Port = # Analyzer port where it will get the app metrics
       # from libDMLib.so
```

[Syslog]

This section contains the same parameters as the previous **Syslog** section from the Application Controller and should be filled with the values relative to the Analyzer.

3.3 Tunlet - Tunlet.ini

In the Tunlet configuration file (Tunlet.ini), the functions to monitor and its associated features are indicated in the **Functions** section. The fields *function1*, *function2*, ..., *functionN* are necessary to follow the pattern.

```
[Functions]
function1 = # name of one function to monitor
function2 = # name of another function to monitor
...
```

Each of the functions defined should have its corresponding section such that:

```
[name_of_function1]
entry = # FuncEntry/FuncExit
event = # iterStart/iterEnd ...
...
[name_of_function2]
entry = ...
```

The parameters can have the following values:

entry *FuncEntry / FuncExit* this corresponds to where the function will be monitored, FuncEntry if in function entry instance, FuncExit if in function exit instance.

event Name of the event that will be generated. Each event is some significant activity that will be performed during the execution of the app and that collects all the necessary data during run-time.

Depending on the information that is desired to obtain from each function, one can specify the following values:

- **Source:** Depending on the element to monitor, the *source* parameter can have the following values:

Source	Monitoring element
FuncParamValue	Function parameter.
VarValue	Function variable.
FuncReturnValue	Function return value.
ConstValue	Constant value.
FuncParamPointerValue	Function pointer parameter.
SendMessageSize	The size of the message to send.**This attribute can only be inserted if the entry specified is *FuncEntry*.**
RecvMessageSize	The size of the message to receive. **This attribute can only be inserted if the entry specified is *FuncExit*.**
SenderTid	The identifier of the sender. **This attribute can only be inserted if the entry specified is *FuncExit*.**

- **type:** The source type can be *Integer*, *Short*, *Float*, *Double*, *Char* or *String*.
- **id:** Since *source* is a function parameter, then *id* is the identifier of the parameters. For instance, if the function f is called by $f(a, b, c)$, the id corresponding to a is 0, b is 1 and c is 2. In any other case, the *id* will correspond to the name of the variable.

3.4 DMLib - DMLib.ini:

This configuration file only contains a **Syslog** section which will have the same parameters as the other ones. The values need to be changed to those relative to the DMLib log files. An example of a DMLib.ini file can be seen in section 6.3.4.

4 Full execution example with XFire

For this full execution demo, the configuration files that are used can be found in section 6. The demo is done with the XFire application. Its files and installation steps can be found in *examples/xfire*. The steps to execute XFire along with MATE are:

1. **Install XFire:** To install XFire, go to *examples/xfire/src/* from MATE's root directory and do:

   ```
   $ ./configure && make
   ```

   This will create a *xfire* executable in XFire's root directory.

2. **Configure Analyzer.ini:** The *Analyzer.ini* config file is in the root directory of XFire and can be seen in section 6.3.1. The default parameters and values already in *Analyzer.ini* don't need to be changed for the example to work.

3. **Configure AC.ini:** *AC.ini* is also provided in the root directory of XFire and can be seen in section 6.3.2. The parameter *ACPath* and *DMLib* must be changed to point to where the *AC* executable and *libDMLib.so* library are respectively (in absolute path). The *Host* parameter represents the name or ip of the host where the *Analyzer* will be run (e.g. *localhost* if in the same machine).

4. **Configure Tunlet.ini:** The *Tunlet.ini* can be seen in section 6.3.3. This is where the events are defined and will be added as instrumentation to the program. Those do not need to be changed for the example to work.

5. **Execute the Analyzer:** The general way of calling the Analyzer is by doing:

   ```
   $ Analyzer -config </Analyzer/config/file/path> <mpi_app>
   ```

   To run the Analyzer example, inside XFire's root dir, do:

   ```
   $ Analyzer -config Analyzer.ini xfire
   ```

6. **Execute the Application controller:** The general way of calling the AC is by doing:

   ```
   $ mpirun -np <n_processes> AC <mpi_app> <app_params>
   ```

   To run the AC with the XFire app, inside XFire's root dir, do:

   ```
   $ mpirun -np 4 AC xfire demo
   ```

   Where *demo* is the parameter that tells XFire to run an example.

After a while, the created events will start their execution and the log files will record the outputs of the AC, Analyzer and DMLib in the specified paths. Those were explicitly input in AC.ini, Analyzer.ini and DMLib.ini with the parameters *LogFile* and *LogPath*.

5 Tunlet creation and execution

To support the analysis of different problems, MATE includes a catalog of tuning techniques where each one solves a particular problem. Each tuning technique provides information about measure points, performance model (analytical model or set of rules) and tuning action/points/synchronization. Such knowledge is provided to MATE via specific libraries called *tunlets*. Each tunlet implements the logic to overcome a particular performance problem by encapsulating knowledge about it in several terms that define the information required for the monitoring, analysis and tuning phases. The tunlet is thought to detect and resolve situations (events) which are desired to be controlled in your application.

5.1 Steps to create a tunlet

1. **Creation of the tunlet**: For a faster and easier tunlet creation, use the tunlet example *MyTunlet.h* and *MyTunlet.cpp* inside the *src/Analyzer/* folder in the root directory of MATE. These and the rest of the files can be found in *examples/simple-tunlet/src/*.
   The steps to create a tunlet are detailed below:

   (a) **Create the events for the situations to detect:** Each event has to be given a name. Their definition is done in *MyTunlet.cpp* and *MyTunlet.h* by means of an *enumerator*, so, these will have to be defined as follows:

   In *MyTunlet.h:*

```
enum EventsEnum
{
  idReplaceFunction,
  idSetVariableValue,
  ...
};
```

   Then, in *MyTunlet.cpp* we will only need to create the corresponding map:

```
std::map<std::string, EventsEnum> EventMap = \
boost::assign::map_list_of ("ReplaceFunction", idReplaceFunction
    ) ("SetVariableValue", idSetVariableValue)...;
```

   (b) **Capture the events:** In *MyTunlet.cpp*, the function *MyTunlet::CreateEvent*, inherited from the *EventHandler* class, creates the new events using the previously defined map *EventMap*:

```
void MyTunlet::CreateEvent(Task & t)
{
  // Create the event
  EventsEnum idEvent;

  // Read the event name from Tunlet.ini
  idEvent = (EventsEnum) EventMap \
                          [_cfg.GetStringValue(func,"event")];

  // Create the new event
  Event endEvent(idEvent,func, ipFuncExit);
  endEvent.SetEventHandler(*this);
  t.AddEvent(endEvent);

  // These final lines have to be included
  std::string semaphoreFunc = "MonitorSignal";
  Event semaphoreEvent(0,semaphoreFunc, ipFuncExit);
  semaphoreEvent.SetEventHandler(*this);
  t.AddEvent(semaphoreEvent);
}
```

   (c) **Create the actions for each event:** All events are handled in the function *MyTunlet::HandleEvent* also inherited from *EventHandler* and therefore, there's where each specific action must be taken depending on what the event is. In

the example below, for *idReplaceFunction*, the action to perform is to change the function *function_to_replace()* for *new_function()*.

```
void MyTunlet::HandleEvent (EventRecord const & r)
{
   switch (r.GetEventId())
   {
      case idReplaceFunction:
      {
         _app->GetMasterTask()->ReplaceFunction("
            function_to_replace", "new_function", 0);
         break;
      }

      case idSetVariableValue: { ... break; }
   }
}
```

(d) **Finish the tunlet:** Apart from the above requirements, the following methods need to be defined in *MyTunlet.cpp*:

- *Initialize(Model::Application & app)*
- *BeforeAppStart()*
- *Destroy()*

Then from the *EventHandler* class:

- *HandleEvent*
- *CreateEvent*

And from the *TaskHandler*:

- *TaskStarted*
- *TaskTerminated*

Generally, these methods can be implemented by default instructions, unless requiring specific configurations. These default instructions are shown in *MyTunlet.cpp*.

2. **Modification of Ctrl.cpp:**[1] The class *Controller* is responsible for calling the desired functions from our tunlet via its method *Controller::Run*, but first, we must place the necessary headers, including our tunlet:

```
#include "Ctrl.h"
#include "Config.h"
#include "DTAPI.h"
#include "Syslog.h"
#include "MyTunlet.h"
#include <unistd.h>
```

Then, we will define the constructor where we will read the configuration file with the *ConfigHelper* object.

[1]The whole code is included in section 6.2.2 and in *examples/simple-tunlet/*

```
7  Controller::Controller(CommandLine & cmdLine, std::string const &
       cfgFile)
8      : _cmdLine(cmdLine)
9  {
10     // Read configuration from file
11     _cfg = ConfigHelper::ReadFromFile(cfgFile);
12
13     // Configure log
14     Syslog::Configure(_cfg);
15 }
```

Now we can define the *Controller::Run* method that will call our functions from the tunlet. First we have to create the application, a pointer to the DTLibrary (by passing the configuration to it) and an object for our tunlet:

```
17 void Controller::Run (ShutDownManager *sdm)
18 {
19     DTLibrary * lib = DTLibraryFactory::CreateLibrary (_cfg);
20     MyTunlet mt;
21
22      // Create the application model
23      Model::Application & app = lib->CreateApplication(
24        _cmdLine.GetAppPath(),
25        _cmdLine.GetAppArgc(),
26        _cmdLine.GetAppArgv());
27
28     // Set the reference to the app in the sdm so it can stop the ACs
29     sdm->setApp(app);
30
31     // Initialize the tunlet
32     mt.Initialize(app);
```

Then start the tunlet and the app:

```
34      // Start app
35     mt.BeforeAppStart();
36     app.Start();
```

And create the main loops that will take care of the event processing:

```
37     while (app.GetStatus() == stStarting && !sdm->isFinished()) {
38          int nEvents = app.ProcessEvents (true);// Blocking
39        if (nEvents > 0)
40           Syslog::Debug("[Ctrl] Processed %d events", nEvents);
41        sleep (1);
42     }
43
44     // Do nothing if the user stopped the application
45     if (!sdm->isFinished()) {
46        Syslog::Debug("[Ctrl] App is running...");
47        Syslog::Debug("[Ctrl] Waiting for events");
48     }
49
50     while (app.GetStatus () == stRunning && !sdm->isFinished()) {
51        int nEvents = app.ProcessEvents (true); // Blocking
52        if (nEvents > 0)
53           Syslog::Debug("[Ctrl] Processed %d events", nEvents);
```

```
54          sleep (1);
55      }
```

Finally destroy both the tunlet and the library.

```
56
57      Syslog::Debug ("[Ctrl] Application has finished");
58
59      // Destroy the tunlet
60      mt.Destroy ();
61
62      // Cleanup
63      DTLibraryFactory::DestroyLibrary(lib);
64  }
```

Once the tunlet is created and *Ctrl.cpp* modified, the *Analyzer* has to be recompiled and reinstalled. To do so, include the object *MyTunlet.o* in the Makefile inside *Analyzer/* and execute the following in MATE's *root* folder:

```
$ make && make install
```

3. **Configure Analyzer.ini, AC.ini, Tunlet.ini and DMLib.ini:** Once the Analyzer is ready, we must adapt the files Analyzer.ini, AC.ini, Tunlet.ini and DMLib.ini. To find the ones used in this example, go to *examples/simple-tunlet/* or section 6 of this document. Also, templates of these can be found in *conf/*. They have to be copied into the root directory of the MPI application that we will execute.

4. **Start the Analyzer and AC:** The last step to perform will be to start the Analyzer and AC as explained in section 4:

```
$ Analyzer -config Analyzer.ini <mpi_app>

$ mpirun -np <n_processes> AC <mpi_app> <app_params>
```

6 Annex

6.1 Configure options

Configure allows the user to customize the installation of MATE. If there are some packages installed in local, one can specify the following installation options to look for them:

General options	
--help	Display the help message
--prefix=<dir>	Specify the destination directory. By default /bin for the binaries AC and Analyzer and /lib for the the DM-Lib library
--with-papi	PAPI directory (must contain ./lib and ./include)
--with-papi-incdir	PAPI include directory
--with-papi-libdir	PAPI library directory
--with-libiberty	LIBIBERTY directory (must contain ./lib and ./include)
--with-libiberty-libdir	LIBIBERTY library directory
--with-libiberty-incdir	LIBIBERTY include directory
--with-libelf	LIBELF directory (must contain ./lib and ./include)
--with-libelf-incdir	LIBELF include directory
--with-libelf-libdir	LIBELF library directory
--with-libdwarf	LIBDWARF directory (must contain ./lib and ./include)
--with-libdwarf-incdir	LIBDWARF include directory
--with-libdwarf-libdir	LIBDWARF library directory
--with-dyninst	DYNINST directory (must contain ./lib and ./include
--with-dyninst-libdir	Directory to look for dyninst libraries
--with-dyninst-incdir	Directory to look for dyninst headers

Environment variables	
CC	C compiler command
CFLAGS	C compiler flags
LDFLAGS	Linker flags, e.g. -L<lib dir> if you have libraries in a nonstandard directory <lib dir>
LIBS	Libraries to pass to the linker, e.g. -l<library>
CPPFLAGS	(Objective) C/C++ preprocessor flags, e.g. -I<include dir> if you have headers in a nonstandard directory <include dir>
CXX	C++ compiler command
CXXFLAGS	C++ compiler flags

6.2 Simple tunlet example

6.2.1 MPI application

```c
#include <mpi.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int my_val = 1;

void hello_world(){
    printf("The replacing function has worked !!\n");
    // If we put these two lines, the Analyzer
    // will enter the function just once
    // MPI_Finalize();
    // exit(0);
}

void instr_function(int value_to_change){

    if (value_to_change == 1)
        printf("[INFO]: Value not changed: %d\n", value_to_change);
    else {
        printf("[INFO]: Changed value to %d!!\n", value_to_change);
    }
}

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
```

```
    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print of a hello world message from each process
    printf("Hello world from processor %s, rank %d out of %d processors\n",
           processor_name, world_rank, world_size);

    while(true){
        instr_function(my_val);
    }
    // Finalize the MPI environment.
    MPI_Finalize();
}
```

6.2.2 Ctrl.cpp

```
#include "Ctrl.h"
#include "Config.h"
#include "DTAPI.h"
#include "Syslog.h"
#include "MyTunlet.h"
#include <unistd.h>

Controller::Controller(CommandLine & cmdLine, std::string const & cfgFile)
   : _cmdLine(cmdLine)
{
   // Read configuration from file
   _cfg = ConfigHelper::ReadFromFile(cfgFile);

   // Configure log
   Syslog::Configure(_cfg);
}

void Controller::Run (ShutDownManager *sdm)
{
   DTLibrary * lib = DTLibraryFactory::CreateLibrary (_cfg);
   MyTunlet mt;

    // Create the application model
    Model::Application & app = lib->CreateApplication(
      _cmdLine.GetAppPath(),
      _cmdLine.GetAppArgc(),
      _cmdLine.GetAppArgv());

   // Set the reference to the app in the sdm so it can stop the ACs
   sdm->setApp(app);

   // Initialize the tunlet
   mt.Initialize(app);
```

```
    // Start app
    mt.BeforeAppStart();
    app.Start();

        while (app.GetStatus() == stStarting && !sdm->isFinished()) {
          int nEvents = app.ProcessEvents (true);// Blocking
        if (nEvents > 0)
            Syslog::Debug("[Ctrl] Processed %d events", nEvents);
        sleep (1);
    }

    // Do nothing if the user stopped the application
    if (!sdm->isFinished()) {
        Syslog::Debug("[Ctrl] App is running...");
        Syslog::Debug("[Ctrl] Waiting for events");
    }

    while (app.GetStatus () == stRunning && !sdm->isFinished()) {
        int nEvents = app.ProcessEvents (true); // Blocking
        if (nEvents > 0)
            Syslog::Debug("[Ctrl] Processed %d events", nEvents);
        sleep (1);
    }


    Syslog::Debug ("[Ctrl] Application has finished");

    // Destroy the tunlet
    mt.Destroy ();

    // Cleanup
    DTLibraryFactory::DestroyLibrary(lib);
}
```

6.3 XFire example

6.3.1 Analyzer.ini

```
[Syslog]
MasterSwitch=true
LogLevel=0
StdErr=true
StdErrLogLevel=0
LogPath=$HOME/mpi/log
LogFile=Analyzer.log
AppendMachineName=true
AppendFile=false
Prefix=Analyzer

[Analyzer]
DisableTuningActions=true
```

```
[EventCollector]
Port = 8800
```

6.3.2 AC.ini

```
[AC]
ACPath=</path/to/MATEs/AC>
DMLib=</path/to/MATEs/libDMLib.so>
PTPAcceptorPort=9900

[Analyzer]
Host=<host_name_or_ip>
Port=8800

[Syslog]
MasterSwitch=true
# LogLevel=0 writes all messages, LogLevel=1 writes
# only INFO messages
LogLevel=0
# Log only info level messages on std err
StdErrLogLevel=0
StdErr=true
LogFile=AC.log
LogPath=$HOME/mpi/log
AppendMachineName=true
AppendFile=false
Prefix=AC
```

6.3.3 Tunlet.ini

```
[global_sendreceive]
entry=FuncEntry
event=IterStart
source1=FuncParamValue
type1=Integer
id1=0
source2=VarValue
type2=Integer
id2=TheTotalWork
source3=VarValue
type3=Integer
id3=TheWorkSizeUnitBytes
source4=VarValue
type4=Integer
id4=NW

[Factoring_SetNumTuples]
```

```
entry=FuncEntry
event=NewBatch
source1=FuncParamValue
type1=Integer
id1=0
source2=FuncParamValue
type2=Integer
id2=1

[global_sendwork]
entry=FuncEntry
event=TupleStart
source1=FuncParamValue
type1=Integer
id1=0
source2=FuncParamValue
type2=Integer
id2=2
source3=FuncParamValue
type3=Integer
id3=4

[arcStepKernel]
entry=FuncEntry
event=CalcStart
source1=FuncParamValue
type1=Integer
id1=0
source2=FuncParamValue
type2=Integer
id2=1
```

6.3.4 DMLib.ini

```
[Syslog]
MasterSwitch=true
#LogLevel=0 writes all messages, LogLevel=1 writes only INFO messages
LogLevel=0
StdErr=true
StdErrLogLevel=1
LogPath=$HOME/mpi/log
LogFile=DMLib.log
AppendMachineName=true
AppendFile=false
Prefix=DMLib
```

References

[1] Dynamic instrumentation library. `http://www.paradyn.org/html/dyninst7.0-software.html`. Accessed: 2018-06-08.

[2] Morajko A., Caymes-Scutari P., Margalef T., and Luque E. Mate: Monitoring, analysis and tuning environment for parallel/distributed applications. *Concurrency and Computation: Practice and Experience*, 19(11):1517–1531, 2007.

[3] Anna Morajko, Oleg Morajko, Tomàs Margalef, and Emilio Luque. Mate: Dynamic performance tuning environment. In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, *Euro-Par 2004 Parallel Processing*, pages 98–107, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[4] Anna Sikora. *Dynamic Tuning of Parallel/Distributed Applications*. PhD thesis, Universitat Autònoma de Barcelona, Barcelona, 2004.