Energy Aware Runtime (EAR) documentation

Installation guide

This document is part of the Energy Aware Runtime (EAR) framework. It has been created in the context of the BSC-Lenovo Cooperation project.

Contact

BSC Julita Corbalan julita.corbalan@bsc.es

Lenovo Luigi Brochard lbrochard@lenovo.com

1 Energy Aware Runtime installation overview

The installation of the Energy Aware Runtime (EAR) framework has several steps:

- Downloading and installing the required packages:
  - **MPI**: Message Passing Interface.
  - **PAPI**: Performance Application Programming Interface.
  - **GSL**: GNU Scientific Library.
  - **CPUPower**: Linux kernel tools to examine and tune frequency and power saving related features of your processor.
  - **FreeIPMI**: provides the capability of controlling IPMI (Intelligent Platform Management Interface), and it's used to read hardware sensors.
  - **SLURM**: SLURM is an open source cluster manager and job scheduler, if you want to use EAR SLURM plugin, must be installed before.

- Compiling and installing:
  - Autotools provides an automatic dependency detection and generation of makefiles. Given a default packages installation paths, '*./configure*', '*make*' and '*make install*' will be enough for the EAR installation.

- Learning phase:
  EAR uses a hardware characterization (Hardware Signature) computed prior the application execution, also called Learning phase. This hardware characterization is a matrix of per-node coefficients, based in a node performance and power metrics obtained through a benchmarking tools:
  - These benchmarking tools are also called kernels, are included in EAR package.
  - The kernels have to be compiled to use all node cores and to consume between 1 and 2 minutes of execution time.
  - Are provided scripts to automatically compile and execute these kernels. Section 5 describes how to configure these files to adapt the kernels sizes to the node and the desired execution time.
  - Also is includes a script to automatically execute all the kernels in a list of nodes with a range of frequencies. Once the kernels are executed, sequentially in all nodes in isolation, the coefficients are computed. EAR distribution includes files to automatically execute the learning phase in SLURM systems (section 5.2), using the EAR slurm plugin, and in systems without scheduling support for EAR (section 5.3).

- When EAR is going to be used in a SLURM system, the plugin will decrease the complexity of all those steps.

2 Installing and configuring EAR components

2.1 Pre-requisites

2.1.1 PAPI (with RAPL)

The Performance Application Programming Interface (PAPI) provides an interface for use of the performance hardware counters found in most major microprocessors. PAPI enables software engineers to see, in near real time, the relation between software performance and processor events. PAPI has to be installed also with the component RAPL, given its power measurement capabilities.

Download PAPI files through the following links and install its headers and libraries:
Download: http://icl.utk.edu/papi/software/index.html
Website: http://icl.utk.edu/papi/
RAPL: http://icl.cs.utk.edu/projects/papi/wiki/PAPITopics:RAPL_Access

2.1.2 GSL

The GNU Scientific Library (GSL) is a numerical library for C and C++ programmers. It is free software under the GNU General Public License. The library provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting.

Download GSL files through the following links and install its headers and libraries:
Download: http://ftp.rediris.es/mirror/GNU/gsl/
Website: https://www.gnu.org/software/gsl/

2.1.3 CPUPower

CPUPower is a set of userspace utilities designed to assist with CPU frequency scaling and is highly recommended because it provides useful command-line utilities and a systemd service to change the governor at boot.

To download and install check your Linux distribution package distribution manager or download Linux kernel code and compile this tool located in the following path: */tools/power/cpupower*.
More info: https://wiki.archlinux.org/index.php/CPU_frequency_scaling

2.1.4 Free IPMI

FreeIPMI library provides the capability of controlling IPMI (Intelligent Platform Management Interface), which it's an independent system running in motherboards. It offers multiple functionalities to system administrators, but in this case, is used to read hardware sensors.

Download FreeIPMI files through the following links and install its headers and libraries:
Download: https://www.gnu.org/software/freeipmi/download.html
Website: https://www.gnu.org/software/freeipmi/

2.1.5 MPI

MPI is required for the operation of the library, because the EAR library overlaps MPI symbols but also for the compilation of the stress tests used to train the EAR prediction model. Given that MPI is just a standard, the information of the MPI distribution installed in your cluster depends on you.

2.1.6 SLURM (required for EAR SLURM plugin)

SLURM is an open source cluster manager and job scheduler for large and small Linux clusters. It requires no kernel modifications for its operation and is relatively self-contained.

Download SLURM files through the following links and install its headers and libraries:
Download: https://slurm.schedmd.com/download.html
Website: https://slurm.schedmd.com/

3 Compiling and installing

Briefly, the shell commands '*./configure; make; make install*' should configure, build, and install this package.

The '*configure*' shell script attempts to guess the correct values for a different set of hardware and operative systems. It generates a 'Makefile' in each directory of the package and other required files containing tests or variable definitions. Also it creates a shell script '*config.status*' used to recreate the current configuration in case of need, and also a file '*config.log*' containing the configuration process output.

3.1 Configure and compiling options

It could be useful to run '*./configure --help*' for listing the options details.

Configure is based on shell variables which initial value could be given by setting variables in the command line or in the environment. You can see a table with the most influential variables below.

Variable	**Description**
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	C/C++ preprocessor flags, e.g. -I<include dir> if you have headers in a nonstandard directory <include dir>.

This is an example of '*CC*', '*CFLAGS*' and '*LIBS*' overwriting:

./configure CC=c99 CFLAGS=-g LIBS=-lposix

Another set of specific options were included to help configure to find some required packages like PAPI, SLURM, etc.

Option	**Description**
--with-papi=<path>	Specifies the path to PAPI installation
--with-gsl=<path>	Specifies the path to GSL installation
--with-cpufreq=<path>	Specifies the path to CPUFreq installation

--with-slurm=<path>	Specifies the path to SLURM installation
--with-freeipmi=<path>	Specifies the path to FreeIPMI installation

This is an example of '*CC*' overwriting and PAPI path specification:

./configure CC=c99 --with-papi=/path_to_papi/PAPI

If unusual procedures must be done to compile the package, please try to figure out how 'configure' could check whether to do them and contact the team to be considered for the next release. In the meantime, you can use overwrite shell variables or export them to the environment (e.g. LD_LIBRARY) or specify the paths of external requirements.

You can remove the binaries and object files from the source folder by typing '*make clean*'.

3.2 Installing options, folders, and files

Install EAR files by typing '*make install*' if you are comfortable with default installation path (*/usr/local*). If not, there is a possibility to establish root and inner folders in a different path by adding to '*./configure*' some of these arguments (only prefix is working by now):

Argument	**Description**
--prefix=<path>	Sets the root install path. [Default: /usr/local]
--bindir=<path>	Sets the binaries install path. [Default: PREFIX/bin]
--libdir=<path>	Sets the libraries install path. [Default PREFIX/lib]
--sysconfdir=<path>	Sets the configuration files path. [Default PREFIX/etc]

This is an example of installing EAR in a custom path, but '*etc*' in a separate folder:

./configure --prefix=/hpc/base/ctt/packages/ear --sysconfdir=/hpc/base/ctt/configs/

Once installed, the root folder which will be referenced by the environment variable *EAR_INSTALL_PATH*, contains the following internal directories (provisional folders could be found here, just ignore them):

Path	**Description**
/lib	Libraries.
/bin	Binaries like the privileged daemon, and other tools.
/bin/kernels	Kernel (or benchmarks).
/etc	Configuration file examples and scripts.

4 Configuration

4.1 General configuration and environment module

A module file is created in autotools folder. This file is a python script which defines the variable 'EAR_INSTALL_PATH', which references to the EAR installation path and is needed for the correct EAR operation. You have to install it using GNU Modules program. After that, you can use the command '*module load*' to load it to the environment. Check its name and version by typing '*module avail*' command.

4.2 SLURM plugin configuration

Already compiled the plugin using make, next to EAR and other tools, SLURM must be aware of the presence of the compiled plugin. To do that you have to modify the SLURM's plugin configuration file called '*plugstack.conf*'. The file location depends on your cluster SLURM installation details.

Once found, edit '*plugstack.conf*' and add a new row as the supplied example (figure 1).

```
#required?      plugin                                  args
required        /usr/local/lib/slurm/ear_slurm_plugin.so        ear_conf_file=/usr/local/etc/slurm/ear.conf
```

Figure 1. plugstack.conf

The *required* field means the failure of any of the plugin's functions will cause slurmd to terminate. The *plugin* field just points to the plugin.

By last, the argument '*ear_conf_file*' points to the EAR library, daemon and plugin configuration file. It contains all the environment variables, and this is an advantage respect to generic configuration mechanism because allows an administrator to provide a new default values, saving the user to write all variable exports for each computing job (in case the user wants the same default values). You can see an example of '*ear.conf*' in the figure 2:

```
EAR=0
EAR_DB_PATHNAME/etc/DBS/db
EAR_COEFF_DB_PATHNAME=/etc/COEFFICIENTS/coeff.
EAR_DYNAIS_LEVELS=4
EAR_DYNAIS_WINDOW_SIZE=200
EAR_LEARNING_PHASE=0
EAR_MIN_PERFORMANCE_EFFICIENCY_GAIN=0.75
EAR_PERFORMANCE_PENALTY=0.2
EAR_POWER_POLICY=MONITORING_ONLY
EAR_PERFORMANCE_ACCURACY=100000
EAR_RESET_FREQ=0
EAR_TURBO=0
EAR_TMP=/tmp
EAR_GUI_PATH=/tmp
EAR_VERBOSE=0
```

Figure 2. EAR configuration file for SLURM.

For a complete list of the EAR environment variables, please head up to the section 8 of the Reference Manual. Both files are present in EAR '*/etc*' installation folder, you just have to overwrite the new values.

Generic configuration still valid in the same way and still have the priority over any default environment variables: an user could overwrite the configuration file values by exporting the variables to its current environment. Finally the EAR *srun* values (check '*srun --help*') will prevail over the rest, but these variables are properly commented in the user guide.

For these configuration changes to take effect, remember reboot the cluster nodes or reload SLURM's daemon.

4.3 Generic configuration

The configuration of the EAR library and tools behavior is based in environment variables. These variables defines binary paths, saving paths, power policies, thresholds and names. You can find a list and description of these variables in the attached document "Energy Aware Runtime: reference document".

Before launching application you will have to export those environment variables which you want to overwrite its default value; for example the location of the installed EAR library which will be overlapped using LD_PRELOAD, a path for saving temporary files, the policy to be used and the application name.

All files, including libraries, benchmarks (also called kernels) and scripts can be found in the path pointed by the environment variable *EAR_INSTALL_PATH*, which is loaded by the EAR environment module. So when referring a library or script, they are is placed in that folder.

5 Executing the learning phase

The learning phase is an installation step in which the hardware is characterized in form of trained coefficients (also called Hardware Signature). In that case, the characterization is made for each cluster computing node by getting metrics during the execution of some benchmarks.

This coefficients allows EAR model to predict performance per frequency depending on the running program context, on the fly, and thus dynamically adjusting each node frequency to this optimal during application execution therefore saving energy.

There are two ways of complete a learning phase. One is using SLURM on clusters managed by this queue manager. Other is manually, for systems where other queue managers are used or maybe there aren't.

5.1 Kernels

Kernels are the stressing programs which learning phase will execute to get the node coefficients. The learning phase runs a copy of each benchmark in all single nodes, but runs it on all nodes simultaneously. In this way every node is isolated with its own copy of the benchmark, and is stressing its cores or memory bandwidth to the maximum.

You can see a summary of these kernels in the table below:

File	**Reference**
dgemm	https://software.intel.com/en-us/mkl-tutorial-c-multiplying-matrices-using-dgemm
stream	http://www.cs.virginia.edu/stream/ref.html
bt-mz	https://www.nas.nasa.gov/publications/npb.html
sp-mz	
lu-mz	
ep	
lu	
ua	

5.2 Coefficients, databases and summaries

Three types of files will be generated during the learning phase or any job submission.

The first type, the coefficient files, contains these values per node and per frequency, meaning that there as many files as nodes and frequencies (nodes in the cluster * number of *P_STATEs*). The coefficients are the final files generated by the learning phase. Once stored, these files remains untouched and must be write protected against users.

In case a node doesn't find its coefficients files, it will search in the location pointed by the environment variable plus the word default and node available frequencies suffixes: '$*EAR_COEFF_DB_PATHNAME_default{frequency}*'.

The second type are database files which stores metrics about the previously executed applications as historic in binary form, meaning that new data is stored by the learning phase, but also by any normal user job submission. The coefficients files are generated from these database files.

The third type of files are plaintext summaries, a record of the launched applications that can be consulted by users or administrators. Are generated, like the database files, on every user job submission.

The paths of this type of saved data are controlled by the environment variables *EAR_DB_PATHNAME*, *EAR_COEFF_DB_PATHNAME* and *EAR_USER_DB_PATHNAME.*

5.3 Executing the learning phase using SLURM

Executing the learning phase with SLURM reduces considerably the steps and configuration of any execution, including the learning phase.

It is important to be sure that your SLURM installation is in the best conditions and your MPI processes are binding correctly to a CPU. Coefficients are a frail thing, and have to be as accurate as possible in order to be precise in frequency predictions to save the maximum energy possible (take a look to SLURM's CPU management and TaskPlugin web sections).

5.3.1 Compiling, installing and testing kernels

Kernels must be compiled for a specific architecture since they must use all the cores in the node. Moreover, to guarantee a good performance accuracy, they must run for a minimum execution time. To get that time that you have to execute a test phase prior to learning phase.

It is provided a script to compile these kernels and also to test its execution time in '$EAR_INSTALL_PATH/*etc/learning_phase_compile.sh*'. You have to edit some parameters of this script as you can see in the following example (figure 3).


```
# Edit architecture values
export CORES=28
export SOCKETS=2
export CORES_PER_SOCKET=14

# Update the paths
export EAR_SRC_PATH=$HOME/git/EAR
```

Figure 3. learning_phase_compile.sh editable region.

The parameters to update are:

- **CORES**: the total number of cores in a single computing node.
- **SOCKETS**: the total number of sockets in a single computing node.
- **CORES_PER_SOCKET**: the total number of cores per socket in a single computing node.
- **EAR_SRC_PATH**: the path to the place where you unzipped EAR package.

Once the parameters are correctly edited, compile and launch the script using the argument '*compile*', in the node where you want to compile. You could also compile in remote node using '*srun*' like the following example:

'*srun -N1 -n1 -w compile_node learning_phase_compile.sh compile*'.

Once compiled, execute the test in a computing node, this time using '*srun*' because a lot of processes will be created. To do that, launch the same script with its argument '*test*':

'*srun -N1 -n1 --exclusive learning_phase_compile.sh test'*

Remember to allocate the whole node with the argument '*--exclusive*'. If you want to launch SLURM through your MPI run program, take a look to this example:

'*mpirun -n 1 -host node1 -bootstrap slurm -bootstrap-exec /path_to_slurm/srun -bootstrap-exec-args="--exclusive" learning_phase_compile.sh test*'

In this example, the bootstrap option was used, next to the reference to our custom SLURM installation.

Once executed, inspect the test information file you wrote in the location pointed by the environment variable **EAR_USER_DB_PATHNAME** in your '*ear.conf*' file. You can see a sample of this file and a discordant kernel execution time in figure 4.

```
USERNAME;JOB_ID;NODENAME;APPNAME;FREQ;TIME;CPI;TPI...
xuser;838;xnode;bt-mz.C.28;2600000;35.460251;0.401421;1.560220...
xuser;838;xnode;sp-mz.C.28;2600000;52.380336;0.500832;4.614459...
xuser;838;xnode;stream_mpi;2600000;83.232500;2.714464;87.458571...
xuser;838;xnode;ep.D.28;2600000;307.452702;1.647134;0.020879;0.029462...
```

Figure 4. $EAR_USER_DB_PATHNAME file sample.

If the time (in seconds) is between 60 and 120, the compilation, installation and test are done and you can proceed to the execute the learning phase. In case some benchmarks are not between these times, it is recommended to compile using different parameters.

If one of these benchmarks is *lu-mz, bt-mz, sp-mz, lu, ep* or *ua*, you could easily alter its behavior by changing its class letter in the same test script. For example if you want to increase its execution time of a kernel compiled with class letter C, switch it by D. Or if you want to decrease the execution time of a kernel compiled with class letter B, switch the letter by A. Then compile and execute again.

You could see where you have to edit in the following picture (figure 5).

```
# Compiling or executing the different kernels
learning-phase lu-mpi C
learning-phase ep D
learning-phase bt-mz C
learning-phase sp-mz C
learning-phase lu-mz C
learning-phase ua C
learning-phase dgemm
learning-phase stream
```

Figure 5. Bottom region of learning_phase_compile.sh script.

As you can see, there are no class letters for *dgemm* or *stream* kernels. *Stream* is a well known benchmark and there is no need to manual modification because varies its behavior itself. For *dgemm* or for a class letter benchmark which doesn't fit in your goals, it's recommended to do a manual kernel modification.

5.3.2 Manual kernel modification

Depending on the kernel, modify its key file inside 'ear_learning_phase/kernels' path of the uncompressed EAR package:

- bt-mz, sp-mz and lu-mz: '*NPB3.3.1-MZ/NPB3.3-MZ-MPI/sys/setparams.c*'
- lu and ep: '*NPB3.3.1/NPB3.3-MPI/sys/setparams.c*'
- ua: 'NPB3.3.1/NPB3.3-OMP/sys/setparams.c'
- stream: '*STREAM/stream_mpi.f*'
- dgemm: '*DGEMM/mkl_fortran_samples/matrix_multiplication/src/dgemm_example.f*'

As you can see, *lu-mz, bt-mz, sp-mz, lu, ep* or *ua* shares a file called '*setparams.c*'. If you want to add or subtract load, locate a function called '*write_sp_info*' in this file, the class letter you want to use it and modify the number of iterations variable '*niter*'. You can see a sample of this code in figure 6.


```
void write_sp_info(FILE *fp, int nprocs,
{
  int gx_size, gy_size, gz_size, niter, x
  long max_xysize, max_xybcsize;
  int max_numzones, max_lsize, nprocs2;
  char *dt, *ratio, *int_type;

  int_type="integer";
  if      (class == 'S')
  {gx_size = 24; gy_size=24; gz_size=6;
   x_zones = y_zones = 2;
   dt = "0.015d0";   niter = 100;}
  else if (class == 'W')
  {gx_size = 64; gy_size=64; gz_size=8;
   x_zones = y_zones = 4;
   dt = "0.0015d0";  niter = 400;}
  else if (class == 'A')
  {gx_size = 128; gy_size=128; gz_size=16
   x_zones = y_zones = 4;
   dt = "0.0015d0";  niter = 400;}
  else if (class == 'B')
```

Figure 6. write_sp_info function in setparams.c.

For dgemm, you have to edit the file '*dgemm_example.f*'. Take a look to PARAMETER variable definition in the first line, which sets the size of the computing matrix. Increase or decrease that values equally depending if you want to add or subtract computing time. You can see a sample of this code in figure 7.

```
DOUBLE PRECISION ALPHA, BETA
INTEGER          M, P, N, I, J
PARAMETER        (M=16384, P=16384, N=16384)
DOUBLE PRECISION A(M,P), B(P,N), C(M,N)
```

Figure 7. Head of dgemm_example.f.

5.3.3 Executing the learning phase

Once the kernels have been compiled and installed, you are ready to execute the Learning phase using the script '*etc/learning_phase_execute.sh*'. This script will launch a set of all kernels in all nodes in isolation, stressing the machine and saving report files and databases and other data where you pointed in the EAR SLURM's configuration file '*ear.conf*'

As the compiling phase, a minimum script edition has to be made. You can see the parameters to edit in figure 8.

```
# Edit architecture values
export CORES=28
export SOCKETS=2
export CORES_PER_SOCKET=14

# Edit learning phase parameters
export EAR_MIN_P_STATE=1
export EAR_MAX_P_STATE=6
```

Figure 8. learning_phase_execute.sh editable region.

The parameters to update are:

- **CORES**: the total number of cores in a single computing node.
- **SOCKETS**: the total number of sockets in a single computing node.
- **CORES_PER_SOCKET**: the total number of cores per socket in a single computing node.
- **EAR_MIN_P_STATE**: defines the maximum frequency to set during the learning phase. The default value is 1, meaning that the nominal frequency will be the maximum frequency that your cluster nodes will set. In the current version of EAR turbo support is not included
- **EAR_MAX_P_STATE**: defines the minimum frequency to test during the learning phase. If 6 is set and EAR_MIN_P_STATE is 1, it means that 6 frequencies will be set during the learning phase, from 1 to 6. This set of frequencies have to match with the set of frequencies that your cluster nodes are able to set during computing time.

A final edition has to be made, only if you changed the class letter of some kernel. If you did it, go down to the bottom of the file and replace the kernel class with its new letter.

Once the parameters are correctly edited, execute the learning phase launching using '*srun*' like the following example:

'*srun -N4 -n4 --exclusive -t 180 learning_phase_execute.sh*'

In the example, the script is launched in 4 nodes reserving them completely. A maximum time is set, but it depends on the set of frequencies set. Having more frequencies in that set, more steps would be added to the learning phase. Suppose that there are 8 programs that have a maximum duration of 16 minutes per frequency in that set.

You could use 'srun' through your MPI job launcher:

'*mpirun -n 3 -hosts node1,node2,node3 -bootstrap slurm -bootstrap-exec /path_to_slurm/srun -bootstrap-exec-args="--exclusive -t 180" learning_phase_execute.sh*'

Once the learning phase is completed the cluster users are ready to use EAR for their jobs. If you don't have interest in execute the learning phase manually, head directly to section 5.4 to get more information about learning phase generated files.

5.4 Executing the learning phase manually

Executing the learning phase manually adds extra steps and definitions compared to the SLURM's guide. It isn't difficult and it's steps will be progressively less as EAR is updated.

First of all, make sure that your PAPI, GSL, FreeIPMI and CPUPower libraries are visible by the dynamic loader, because the following scripts instructions will take your environment to run MPI.

5.4.1 Compiling, installing and testing kernels

Compiling, installing and testing learning phase kernels without SLURM, adds a few extra steps. But in fact is almost the same than the sections 5.2.1 and 5.2.2. So, read first both sections omitting any SLURM execution command or reference.

So, for compiling and installing you have to compile using the same script used in that section '*etc/learning_phase_compile.sh*', editing the same values of section 5.2.1 figure (FIGURE), but launching it with your MPI execution command instead of SLURM's '*srun*' in case you want to do it remotely.

Finally you have to test like the SLURM section explains, but a small edition has to be made before. Open the file '*etc/learning_phase_helper.sh*' and look to these lines of figure 9:

```
function launching
{
   ...
}

function launching_disabled
{
   export EAR_TMP=$HOME/tmp
   export EAR_COEFF_DB_PATHNAME=$EAR_TMP/coeff
   export EAR_USER_DB_PATHNAME=$EAR_TMP/sum
   export EAR_DB_PATHNAME=$EAR_TMP/db
   ...
   mpiexec.hydra -n $2 -genvall -genv                          \
      LD_PRELOAD=$EAR_INSTALL_PATH/ear/lib/libEAR.so \
      $BENCHS_BIN_PATH/$1
}
```

Figure 9. learning_phase_helper.sh editable region.

As you can see, the function '*launching*' is just the SLURM's execution command. So switch the word 'disabled' from '*launching_disabled*' adding it to the current one '*launching*' one. Also, replace path variables were you want to store the databases and if you are not using '*mpiexec.hydra*' replace this command by your MPI distribution execution command and it will be ready for the test in like this:

'*mpiexec.hydra -n 1 -host node1 learning_phase_compile.sh test*'

As you can see, the host was changed to a compute node, instead the previous compile node. Once executed, inspect the test information file like explains section 5.2.1 and perform, if needed, the changes that follows the sections 5.2.1 and 5.2.2.

5.4.2 Executing the learning phase

Executing the learning phase kernels without SLURM is almost the same than the section 5.2.3 and 5.2.2 just adding a small script edition. So, read first that section, perform all it's changes but omitting any SLURM reference or execution command.

As you previously switched the word 'disabled' from '*launching-disabled*' adding it to the current one '*launching*' one and maybe fixing the Intel's '*mpiexec.hydra*' command, no more editions will be necessary. Type a command like this to execute the learning phase by your MPI distribution:

'*mpiexec.hydra -n 2 -ppn 1 -hosts node1,node2 learning_phase_execute.sh*'

As you can see, host argument now includes two compute nodes. You can also create a file with the names of all nodes you want to get its coefficients (Hardware Signature) passing its path to argument '*-f*'.

