# EAR 6.0

User guide

EAR team


2025-11-07

# Contents

# Introduction

EAR 6.0 is a system software for energy management, accounting and optimization for super computers. Main EAR services are:

1. Application energy optimization. An **easy-to-use** and **lightweight** optimization service to automatically select the optimal CPU, memory and GPU frequency according to the application and the node characteristics. This service is offered by EAR core components: The EAR library and EAR Node Manager. EARL is a runtime library automatically loaded with the applications and it offers application metrics monitoring and it can select the frequencies based on the application behaviour on the fly. The Library is loaded automatically through the EAR Loader (EARLO) and it can be easily integrated with different system batch schedulers (e.g., SLURM). The EARL provides deep application accounting(both power/energy and performance) and energy optimization in a completely transparent and dynamic way.
2. Job and Node monitoring: A complete **energy and performance accounting and monitoring system** . Node and application monitoring are also provided by the EAR core components (EAR library and EAR node manager). These two components are the data provides and information is reported to the DB using the EAR DB manager (EARDBD). The EARDBD is a distributed service offering buffering and aggregation of data, minimizing the number of connections with the DB server. EAR includes several report plugins for both relational (MariaDB and PostgreSQL) and non-relational Databases such as EXAMON. EAR commands for data reporting are only based on relational DBs.
3. Cluster power management (powercap). A **cluster energy manager** to monitor and control the energy consumed in the system through the EAR Global Manager (EARGMD) and EARD. EAR support a powerful and flexible configuration where different architectures with different power limits can be configured in the same cluster. CPU and CPU+GPU nodes are supported.

Visit the architecture page for a detailed description of each of these components. The user guide contains information about how to use EAR as an end user in a production environment. The admin guide has all the information related to the installation and setting up, as well as all core components details. Moreover, you can find a list of tutorials about how to use EAR.

## License



Contact: ear-support@bsc.es.

# User guide

## Job monitoring and optimization with EAR

EAR was first designed to be usable 100% transparently by users, which means that you can run your applications enabling/disabling/tuning EAR with the less effort to for changing your workflow, e.g., submission scripts. This is achieved by providing integrations (e.g., plug-ins, hooks) with system batch schedulers, which do all the effort to set-up EAR at job submission. **SLURM is the main batch scheduler fully compatible with EAR** thanks to EAR's SLURM SPANK plug-in.

With EAR's SLURM plug-in, running an application with EAR is as easy as submitting a job with either `srun`, `sbatch` or `mpirun`.

EAR is also compatible with **PBSPro**, thorugh the EAR PBRPro Hook.

The EAR Library (EARL) is automatically loaded with some applications when EAR is enabled by default.

Check with the `ear-info` command if EARL is `on`/`off` by default. If it's `off`, use `--ear=on` option offered by EAR SLURM plug-in to enable it. For other schedulers, a simple prolog/epilog command can be created to provide transparent job submission with EAR and default configuration. The EAR development team had worked also with OAR and PBSPro batch schedulers, but currently there is not any official stable nor supported feature.

## Use cases

Since EAR was targetting computational applications, some applications are automatically loaded and others are not, avoiding running EAR with, for example, bash processes. The following list summarizes the application use cases where the EARL can be loaded transparently with them:

- MPI applications: IntelMPI, OpenMPI, Fujitsu and CRAY versions.
- Non-MPI applications: OpenMP, CUDA, MKL and OneAPI.
- Python applications.

Other use cases not listed here might still be supported. See the dedicated section.

### MPI applications

EARL is automatically loaded with MPI applications when it is enabled by default (check `ear-info`). EAR supports the utilization of both `mpirun`/`mpiexec` and `srun` commands.

When using `sbatch`/`srun` or `salloc`, Intel MPI and OpenMPI are fully supported. When using specific MPI flavour commands to start applications (e.g., `mpirun`, `mpiexec.hydra`), there are some key points which you must take account. See next sections for examples and more details.

Review SLURM's MPI Users Guide, read your cluster documentation or ask your system administrator to see how SLURM is integrated with the MPI Library in your system.

**Hybrid MPI + (OpenMP, CUDA, MKL) applications** EARL automatically supports this use case. `mpirun`/`mpiexec` and `srun` are supported in the same manner as explained above.

**Python and Julia MPI applications** EARL cannot detect automatically MPI symbols when some of these languages are used. On that case, an environment variable is provided to give EARL a hint of the MPI flavour being used.

Export `EAR_LOAD_MPI_VERSION` environment with the value from the following table depending on the MPI implementation you are loading:

| MPI flavour | Value |
|---|---|
| Intel MPI | *intel* |
| Open MPI | *open mpi* or *ompi* |
| MVAPICH | *mvapich* |
| Fujitsu MPI | *fujitsu mpi* |
| Cray MPICH | *cray mpich* |

#### Running MPI applications on SLURM systems

**Using `srun` command** Running MPI applications with EARL on SLURM systems using `srun` command is the most straightforward way to start using EAR. All jobs are monitored by EAR and the Library is loaded by default depending on the cluster configuration.

Even though it is automatic, there are few flags than can be selected at job submission. They are provided by EAR's SLURM SPANK plug-in. When using SLURM commands for job submission, both Intel and OpenMPI implementations are supported.

**There is no need to load the EAR module** for running a job with `srun` and get EARL loaded. Review SLURM's MPI Users Guide, read your cluster documentation or ask your system administrator to see how SLURM is integrated with the MPI Library in your system.

**Using `mpirun`/`mpiexec` command** To provide an automatic loading of EARL, the only requirement from the MPI library is to be coordinated with the scheduler. Review SLURM's MPI Users Guide, read your cluster documentation or ask your system administrator to see how SLURM is integrated with the MPI Library in your system.

Intel MPI

Recent versions of Intel MPI offers two environment variables that can be used to guarantee the correct scheduler integrations:

- `I_MPI_HYDRA_BOOTSTRAP` sets the bootstrap server. It must be set to *slurm*.
- `I_MPI_HYDRA_BOOTSTRAP_EXEC_EXTRA_ARGS` sets additional arguments for the bootstrap server. These arguments are passed to SLURM, and they can be all the same as EAR's SPANK plug-in provides.

You can read here the Intel environment variables guide.

OpenMPI

For joining OpenMPI and EAR it is highly recommended to use SLURM's `srun` command. When using `mpirun`, as OpenMPI is not fully coordinated with the scheduler, EARL is not automatilly loaded on all nodes. Therefore EARL will be disabled and only basic energy metrics. will be reported. To provide support for this workflow, EAR provides `erun` command. Read the corresponding examples section for more information about how to use this command.

MPI4PY

To use MPI with Python applications, the EAR Loader cannot automatically detect symbols to classify the application as Intel or OpenMPI. In order to specify it, the user has to define the `EAR_LOAD_MPI_VERSION` environment variable with the values specified in the table explained above.

It is recommended to add in Python modules to make it easy for final users. Ask your system administrator or check your cluster documentation.

MPI.jl

According to the documentation, the basic Julia wrapper for MPI is inspired by mpi4py. Check its section for running this kind of use case.

### Non-MPI applications

**Python** Since version 4.1 EAR automatically executes the Library with Python applications, so no further action is needed. You must run the application with `srun` command to pass through the EAR's

SLURM SPANK plug-in in order to enable/disable/tuning EAR. See EAR submission flags provided by EAR SLURM integration.

**OpenMP, CUDA, Intel MKL and OneAPI** To load EARL automatically with non-MPI applications it is required to have it compiled with dynamic symbols and also it must be executed with `srun` command. For example, for CUDA applications the `--cudart=shared` option must be used at compile time. EARL is loaded for OpenMP, MKL and CUDA programming models when symbols are dynamically detected.

#### Other application types or frameworks

For other programming models or sequential apps not supported by default, EARL can be forced to be loaded by setting `EAR_LOADER_APPLICATION` environment variable, which must be defined with the executable name. For example:

```
1 #!/bin/bash
2
3 export EAR_LOADER_APPLICATION=my_app
4 srun my_app
```

#### Using EARL inside Singularity containers

Apptainer (formerly Singularity) is an open source technology for containerization. It is widely used in HPC contexts because the level of virtualization it offers enables the access to local services. It allows for greater reproducibility, making the programs less dependent on the environment they are being run on.

An example singularity command could look something like this:

```
1 singularity exec $IMAGE program
```

where `IMAGE` is an environment variable that contains the path of the Singularity container, and `program` is the executable to be run in the image.

In order to be able to use EAR inside the container two actions are needed:

- Binding EAR paths to make them visible in the container.
- Exporting some environment variables to the execution environment to make them available during the execution.

To bind folders there are two options: (1) using the environment variable `SINGULARITY_BIND` /`APPTAINER_BIND` or (2) using the `-B` flag when running the container. 1 is a comma separated string of pairs of paths `[path_1][[:path_2][:perms]]` such that *path_1* in local will be mapped into *path_2* in the image with the permissions set in *perms*, which can be *r* or *rw*. Specifying *path_2* and *perm* is optional. If they are not specified *path_1* will be bound in the same location.

To make EAR working the following paths should be added to the binding configuration:

- `$EAR_INSTALL_PATH`,`$EAR_INSTALL_PATH/bin`,`$EAR_INSTALL_PATH/lib`, `$EAR_TMP`

You should have an EAR module to have the above environment variables. Contact your system administrator for more information.

Once paths are deployed, to execute (for example) an OpenMPI application inside a Singularity/Apptainer enabling the EAR Library just the following is needed:

```
module load ear

mpirun -np <# processes> singularity exec $IMAGE erun --ear=on --
    program="program args"
```

A more complete example would look something like this:

```
export IMAGE=[path_to_image]/ubuntu_ompi.sif
export BENCH_PATH=[path_to_benchmark]
export APPTAINER_BIND="$EAR_INSTALL_PATH:$EAR_INSTALL_PATH:ro,$EAR_TMP:
    $EAR_TMP:rw"
export APPTAINERENV_EAR_REPORT_ADD=sysfs.so

mpirun -np 64 singularity exec $IMAGE $EAR_INSTALL_PATH/bin/erun \
    --ear=on --ear-verbose=1 \
    --program=$BENCH_PATH/bt-mz.D.64
```

Note that the example exports `APPTAINERENV_EAR_REPORT_ADD` to set the environment variable `EAR_REPORT_ADD` to load `sysfs` report plug-in. See next section about report plug-ins.

#### Using EARL through the COMPSs Framework

COMP Superscalar (COMPSs) is a task-based programming model which aims to ease the development of applications for distributed infrastructures, such as large High-Performance clusters (HPC), clouds and container managed clusters. COMPSs provides a programming interface for the development of the applications and a runtime system that exploits the inherent parallelism of applications at execution time. **Since version 5.0 EAR supports monitoring and optimization of workflows** and the COMPSs Framework includes the integration with EAR. Check out the dedicated section from the

official COMPSs documentation for more information about how to measure the energy consumption of your workflows.

EARL loading is **only available** using `enqueue_compss` and with Python applications. The command has the flag `--ear` which you can set either a boolean (i.e., *true* or *false*) or a string value. The latter can be any of the job submission flags. See the example provided by the COMPSs documentation.

## Retrieving EAR data

As a job accounting and monitoring tool, EARL collects some metrics that you can get to see or know your applications workload. The Library is doted with several modules and options to be able to provide different kind of information.

As a very simple hint of your application's workload, you can enable EARL verbosity (i.e., `--ear-verbose=1`) to get loop data at runtime. **The information is shown at *stderr* by default.** Read how to set up verbosity at submission time and verbosity environment variables provided for a more advanced tuning of this EAR feature.

### Post-mortem application data

To get offline EAR job data you can use the **eacct** command, a tool to provide the monitored job data stored in the EAR Database. You can request information in different ways. Thus, you can read either per-node or aggregated job datas averaged along the execution time or get metrics collected at runtime. See eacct usage examples for a better overview of what `eacct` provides.

### Runtime report plug-ins

There is another way to get runtime and aggregated data during runtime without the need of calling `eacct` after the job completion. EAR implements a reporting system mechanism which let developers to add new report plug-ins, so there is an unlimited set of ways to report EAR collected data. EAR releases come with a fully supported report plug-in (i.e., *csv_ts.so*) which provides the same runtime and aggregated data reported to the Database in CSV files, directly while the job is running. You can load this plug-in in two ways:

1. By setting `--ear-user-db` flag at submission time.
2. Loading directly the report plug-in through an environment variable: `export EAR_REPORT_ADD=csv_ts.so`.

Read report plug-ins dedicated section for more information.

#### Other EARL events

You can also request EAR to report **events** to the Database. They show more details about EARL internal state and can be provided with `eacct` command. See how to enable EAR events reporting and which kind of events EAR is reporting.

#### MPI stats

If your application applies, you can request EAR to report at the end of the execution a summary about its MPI behaviour. The information is provided along two files and is the aggregated data of each process of the application.

#### Paraver traces

Finally, EARL can provide runtime data in the Paraver trace format. Paraver is a flexible performance analysis tool maintained by the *Barcelona Supercomputing Center*'s tools team. This tool provides an easy way to visualize runtime data, computing derived metrics and to provide histograms for better of your application behaviour. See on the environment variables page how to generate Paraver traces.

Another way to see runtime information with Paraver is to use the open source tool **ear-job-visualization**, a CLI program written in Python which gets CSV files generated by `--ear-user-db` flag and converts its data to the Paraver trace format. EAR metrics are reported as trace events. Node information is stored as Paraver task information. Node GPU data is stored as Paraver thread information.

You can find in the tool's repository two examples of Paraver Config Files, one for basic CPU metrics and other adding GPU metrics.

#### Data visualization with Grafana

EAR data can be visualized with Grafana dashboards in two different ways: Using grafana with SQL queries (depending on your Data Center configuration) and visualizing data collected with `eacct` and loading locally. The second option will be explained since you might expect to not having access to the EAR Database.

Once you have your own Grafana instance running, you need to install *csv-datasource*:

```
1 bin/grafana-cli plugins install marcusolsson-csv-datasource (You can
    first check if it's already available by testing the available Data
    sources)
```

Enable the CSV plug-in by creating a `custom.ini` file in the conf directory with the following content:

```
[plugin.marcusolsson-csv-datasource]
allow_local_mode = true
```

Once you have a local server running on your PC or laptop, open your web browser and connect to Grafana at the URL: `http://localhost:3000/login`. Next steps are:

**Create the Data source**

In the left menu, select Configuration/Data source/Add data source. Select CSV data source from the list of options. You need to create a new data source for each CSV file you are going to visualize. For each one, select *Local*. Note that the path must to be a **public directory**.

**Import the Dashboard**

Go to the left menu, *Dashboard*, and select the *Import* option. This option allows you uploading or selecting a json file with pre-specified graphs, tables, etc. Graphs are associated with data sources, so you may need to change the Data Source name in the json file to match the one you've created on Grafana. The json file is here, and below you can see the Data Source names expected. There is a configuration for two data sources: *EAR_loops* for visualizing CSV files containing EAR loop signatures (e.g., `eacct [-j <job_id>[.<step_id>]] -r -c <filename>`) and *EAR_app* for visualizing application signatures (e.g., `eacct [-j <job_id>[.<step_id>]] -l -c <filename>`).

```
{
  "__inputs": [
    {
      "name": "DS_EAR_LOOPS",
      "label": "EAR_loops",
      "description": "",
      "type": "datasource",
      "pluginId": "marcusolsson-csv-datasource",
      "pluginName": "CSV"
    },
    {
      "name": "DS_EAR_APPS",
      "label": "EAR_apps",
      "description": "",
      "type": "datasource",
      "pluginId": "marcusolsson-csv-datasource",
      "pluginName": "CSV"
    }
  ],
```

Import the JSON file to create the visualization dashboards and refresh the URL the browser page. Below you can see an example of what you will see.

**Figure 1:** EAR Grafana Dashboard example

## EAR job submission flags

The following EAR options can be specified when running `srun` and/or `sbatch`, and are supported with `srun`/`sbatch`/`salloc`:

| Option | Description |
|---|---|
| **--ear**=[on\|off] | Enables/disables EAR library loading with this job. |
| **--ear-user-db**=*<filename>* | Asks the EAR Library to generate a set of CSV files with EARL metrics. |
| **--ear-verbose**=[0\|1] | Specifies the level of verbosity; the default is 0. |

When using `ear-user-db` flag, one file per node is generated with the average node metrics (node signature) and one file with multiple lines per node is generated with runtime collected metrics (loops node signatures). Read `eacct`'s section in the commands page to know which metrics are reported, as data generated by this flag is the same as the reported (and retrieved later by the command) to the Database.

Verbose messages are placed by default at *stderr*. For jobs with multiple nodes, `ear-verbose` option can result in lots of messages mixed at *stderr*. We recommend to split up SLURM's output (or error) file per-node. You can read SLURM's filename pattern specification for more information.

If you still need to have job output and EAR output separated, you can set `EARL_VERBOSE_PATH` environment variable and one file per node will be generated only with EAR output. The environemnt variable must be set with the path (a directory) where you want the output files to be generated, it will be automatically created if needed.

> You can always check the available EAR submission flags provided by EAR's SLURM SPANK plug-in by typing `srun --help`.

#### CPU frequency selection

The EAR configuration file supports the specification of *EAR authorized users*, who can ask for a more privileged submission options. The most relevant ones are the possibility to ask for a specific optimisation policy and a specific CPU frequency.

> Contact with the sys admin or helpdesk team to become an authorized user.

- The `--ear-policy=policy_name` flag asks for *policy_name* policy. Type `srun --help` to see policies currently installed in your system.
- The `--ear-cpufreq=value` (*value* must be given in kHz) asks for a specific CPU frequency.

#### GPU frequency selection

EAR **version 3.4 and upwards** supports GPU monitoring for NVIDIA devices from the point of view of the application and node monitoring. GPU frequency optimization is not supported yet. **Authorized** users can ask for a specific GPU frequency by setting the `SLURM_EAR_GPU_DEF_FREQ` environment variable, giving the desired GPU frequency expressed in kHz. Only one frequency for all GPUs is now supported.

> Contact with sys admin or helpdesk team to become an authorized user.

To see the list of available frequencies of the GPU you will work on, you can type the following command:

```
1 nvidia-smi -q -d SUPPORTED_CLOCKS
```

### Examples

#### `srun` examples

Having an MPI application asking for one node and 24 tasks, the following is a simple case of job submission. If EARL is turned on by default, no extra options are needed to load it. To check if it is on

by default, load the EAR module and execute the `ear-info` command. EAR verbose is set to 0 by default, i.e., no EAR messages.

```
srun -J test -N 1 -n 24 --tasks-per-node=24 application
```

The following executes the application showing EAR messages, including EAR configuration and node signature in *stderr*.

```
srun --ear-verbose=1 -J test -N 1 -n 24 --tasks-per-node=24 application
```

EARL verbose messages are generated in the standard error. For jobs using more than 2 or 3 nodes messages can be overwritten. If the user wants to have EARL messages in a file the `SLURM_EARL_VERBOSE_PATH` environment variable must be set with a folder name. One file per node will be generated with EARL messages.

```
export SLURM_EARL_VERBOSE_PATH=logs
srun --ear-verbose=1 -J test -N 1 -n 24 --tasks-per-node=24 application
```

The following asks for EARL metrics to be stored in csv file after the application execution. Two files per node will be generated: one with the average/global signature and another with loop signatures. The format of output files is *<filename>.<nodename>*.time.csv for the global signature and *<filename>.<nodename>*.time.loops.csv for loop signatures.

```
srun -J test -N 1 -n 24 --tasks-per-node=24 --ear-user-db=filename
    application
```

For EAR *authorized users*, the following executes the application with a CPU frequency of 2.0GHz:

```
srun --ear-cpufreq=2000000 --ear-policy=monitoring --ear-verbose=1 -J
    test -N 1 -n 24 --tasks-per-node=24 application
```

For `--ear-cpufreq` to have any effect, you must specify the `--ear-policy` option even if you want to run your application with the default policy.

#### `sbatch` + EARL + `srun`

When using `sbatch` EAR options can be specified in the same way. If more than one `srun` is included in the job submission, EAR options can be inherited from `sbatch` to the different `srun` instances or they can be specifically modified on each individual `srun`.

The following example will execute twice the application. Both instances will have the verbosity set to 1. As the job is asking for 10 nodes, we have set the `SLURM_EARL_VERBOSE_PATH` environment variable set to the *ear_log* folder. Moreover, the second step will create a set of csv files placed in the

*ear_metrics* folder. The nodename, Job Id and Step Id are part of the filename for a better identification.

```
#!/bin/bash
#SBATCH -N 1
#SBATCH -e test.%j.err
#SBATCH -o test.%j.out
#SBATCH --ntasks=24
#SBATCH --tasks-per-node=24
#SBATCH --cpus-per-task=1
#SBATCH --ear-verbose=1

export SLURM_EARL_VERBOSE_PATH=ear_logs

srun application

mkdir ear_metrics
srun --ear-user-db=ear_metrics/app_metrics application
```

#### EARL + `mpirun`

**Intel MPI** When running EAR with `mpirun` rather than `srun`, we have to specify the utilization of `srun` as bootstrap. Version 2019 and newer offers two environment variables for bootstrap server specification and arguments.

```
export I_MPI_HYDRA_BOOTSTRAP=slurm
export I_MPI_HYDRA_BOOTSTRAP_EXEC_EXTRA_ARGS="--ear-policy=monitoring
    --ear-verbose=1"
mpiexec.hydra -n 10 application
```

**OpenMPI** Bootstrap is an Intel(R) MPI option but not an OpenMPI option. For OpenMPI `srun` must be used for an automatic EAR support. In case OpenMPI with `mpirun` is needed, EAR offers the `erun` command, which is a program that simulates all the SLURM and EAR SLURM Plug-in pipeline. You can launch `erun` with the `--program` option to specify the application name and arguments.

```
mpirun -n 4 /path/to/erun --program="hostname --alias"
```

In this example, `mpirun` would run 4 erun processes. Then, `erun` will launch the application `hostname` with its alias parameter. You can use as many parameters as you want but the semicolons have to cover all of them in case there are more than just the program name.

`erun` will simulate on the remote node both the local and remote pipelines for all created processes. It has an internal system to avoid repeating functions that are executed just one time per job or node, like SLURM does with its plugins.

**IMPORTANT NOTE** If you are going to launch `n` applications with `erun` command through a sbatch job, you must set the environment variable `SLURM_STEP_ID` to values from `0` to `n-1` before each `mpirun` call. By this way `erun` will inform the EARD the correct step ID to be then stored to the Database.

## EAR job Accounting (`eacct`)

The `eacct` command shows accounting information stored in the EAR DB for jobs (and steps) IDs. The command uses EAR's configuration file to determine if the user running it is privileged or not, as **non-privileged users can only access their information**. It provides the following options.

### Usage examples

The basic usage of `eacct` retrieves the last 20 applications (by default) of the user executing it. If a user is **privileged**, they may see all users applications. The default behaviour shows data from each job-step, aggregating the values from each node in said job-step. If using SLURM as a job manager, a *sb* (sbatch) job-step is created with the data from the entire execution. A specific job may be specified with `-j` option.

```
1 [user@host EAR]$ eacct -j 175966
2     JOB-STEP USER       APPLICATION      POLICY NODES AVG/DEF/IMC(GHz)
          TIME(s)    POWER(W) GBS     CPI    ENERGY(J)    GFLOPS/W IO(MBs)
          MPI%  G-POW (T/U)   G-FREQ  G-UTIL(G/MEM)
3   175966-sb   user       afid             NP     2     2.97/3.00/---
       3660.00    381.51   ---     ---   2792619      ---      ---     ---
          ---           ---     ---
4   175966-2    user       afid             MO     2     2.97/3.00/2.39
       1205.26    413.02   146.21  1.04  995590       0.1164   0.0
       21.0  ---           ---     ---
5   175966-1    user       afid             MT     2     2.62/2.60/2.37
       1234.41    369.90   142.63  1.02  913221       0.1265   0.0
       19.7  ---           ---     ---
6   175966-0    user       afid             ME     2     2.71/3.00/2.19
       1203.33    364.60   146.23  1.07  877479       0.1310   0.0
       17.9  ---           ---     ---
```

The command shows a pre-selected set of columns, read `eacct`'s section on the EAR commands page.

For node-specific information, the `-l` (i.e., long) option provides detailed accounting of each individual node: In addition, `eacct` shows an additional column: `VPI (%)` (See the example below). The VPI is meaning the percentage of AVX512 instructions over the total number of instructions.

```
[user@host EAR]$ eacct -j 175966 -l
     JOB-STEP    NODE ID    USER ID    APPLICATION      AVG-F/IMC-F
        TIME(s)    POWER(s)   GBS        CPI        ENERGY(J)  IO(MBS)
        MPI%  VPI(%)  G-POW(T/U)    G-FREQ G-UTIL(G/M)
  175966-sb      cmp2506    user       afid              2.97/---
     3660.00    388.79     ---        ---        1422970    ---     ---
        ---     ---     ---       ---
  175966-sb      cmp2507    user       afid              2.97/---
     3660.00    374.22     ---        ---        1369649    ---     ---
        ---     ---     ---       ---
  175966-2       cmp2506    user       afid              2.97/2.39
     1205.27    423.81     146.06     1.03       510807     0.0
     21.2  0.23    ---     ---       ---
  175966-2       cmp2507    user       afid              2.97/2.39
     1205.26    402.22     146.35     1.05       484783     0.0
     20.7  0.01    ---     ---       ---
  175966-1       cmp2506    user       afid              2.58/2.38
     1234.46    374.14     142.51     1.02       461859     0.0
     19.4  0.00    ---     ---       ---
  175966-1       cmp2507    user       afid              2.67/2.37
     1234.35    365.67     142.75     1.03       451362     0.0
     20.0  0.01    ---     ---       ---
  175966-0       cmp2506    user       afid              2.71/2.19
     1203.32    371.76     146.25     1.08       447351     0.0
     17.9  0.01    ---     ---       ---
  175966-0       cmp2507    user       afid              2.71/2.19
     1203.35    357.44     146.21     1.05       430128     0.0
     17.9  0.01    ---     ---       ---
```

If EARL was loaded during an application execution, runtime data (i.e., EAR loops) may be retrieved by using `-r` flag. You can still filter the output by Job (and Step) ID.

Finally, to easily transfer `eacct`'s output, `-c` option saves the requested data in CSV format. Both aggregated and detailed accountings are available, as well as filtering. When using along with `-l` or `-r` options, all metrics stored in the EAR Database are given. Please, read the commands section page to see which of them are available.

```
[user@host EAR]$ eacct -j 175966.1 -r
    JOB-STEP      NODE ID    ITER.  POWER(W) GBS      CPI      GFLOPS/W
        TIME(s)  AVG_F IMC_F IO(MBS) MPI%  G-POWER(T/U) G-FREQ   G-UTIL
       (G/MEM)
 175966-1         cmp2506    21     360.6    115.8    0.838    0.086
       1.001    2.58  2.30  0.0      11.6  0.0   /   0.0  0.00     0%/0%
 175966-1         cmp2507    21     333.7    118.4    0.849    0.081
       1.001    2.58  2.32  0.0      12.0  0.0   /   0.0  0.00     0%/0%
 175966-1         cmp2506    31     388.6    142.3    1.010    0.121
       1.113    2.58  2.38  0.0      19.7  0.0   /   0.0  0.00     0%/0%
 175966-1         cmp2507    31     362.8    142.8    1.035    0.130
       1.113    2.59  2.37  0.0      19.5  0.0   /   0.0  0.00     0%/0%
```

```
7  175966-1          cmp2506    41      383.3    143.2    1.034    0.124
          1.114    2.58  2.38  0.0      19.6  0.0   /   0.0  0.00      0%/0%
```

```
1 [user@host EAR]$ eacct -j 175966 -c test.csv
2  Successfully written applications to csv. Only applications with EARL
      will have its information properly written.
3
4 [user@host EAR]$ eacct -j 175966.1 -c -l test.csv
5  Successfully written applications to csv. Only applications with EARL
      will have its information properly written.
```

### Job energy optimization: EARL policies

The core component of EAR at the user's job level is the EAR Library (EARL). The Library deals with job monitoring and is the component which implements and applies optimization policies based on monitored workload.

**We highly recommend you** to read EARL documentation and also how energy policies work in order to better understand what is doing the Library internally, so you can explore easily all features (e.g., tunning variables, collecting data) EAR offers to the end-user so you will have more knowledge about how much resources your application consumes and how to correlate with its computational characteristics.

## EAR commands

EAR offers a set of commands which help both users and administrators to interact with different components:

- Commands to retrieve data stored in the DB: eacct and ereport.
- Commands to control and temporary modify cluster settings: econtrol.
- Commands to create/update/clean the DB: edb_create, edb_clean_pm and edb_clean_apps.
- A command to load transparently the EAR Library on systems where the batch scheduler has not a plug-in nor some use case isn't supported (e.g., running an OpenMPI application on SLURM systems through the `mpirun` command): erun.
- A command to show current EAR installation information: ear-info.

Commands belonging to the first three categories read the EAR configuration file (`ear.conf`) to determine whether the user is authorized, as some of them has some features (or the wall command) only available that set of users. Root is a special case, it doesn't need to be included in the list of authorized users. Some options are disabled when the user is not authorized.

> **NOTE** EAR module must be loaded in your environment in order to use EAR commands.

### EAR job Accounting (eacct)

The `eacct` is a simple command to see a jobs' energy accounting information. It can also retrieve EARL events that occurred on a job execution.

The command uses the EAR Configuration file to determine whether the user running it is authorized, as **non-privileged users can only access their information**. It provides the following options.

```
    -h      displays this message
    -v      displays current EAR version
    -u      specifies the user whose applications will be retrieved.
       Only available to privileged users. [default: all users]
    -j      specifies the job id and step id to retrieve with the
       format [jobid.stepid] or the format [jobid1,jobid2,...,jobid_n].
            A user can only retrieve its own jobs unless said user is
               privileged. [default: all jobs]
    -a      specifies the application names that will be retrieved. [
       default: all app_ids]
    -c      specifies the file where the output will be stored in CSV
       format. [default: no file]
    -t      specifies the energy_tag of the jobs that will be retrieved
       . [default: all tags].
    -s      specifies the minimum start time of the jobs that will be
       retrieved in YYYY-MM-DD. [default: no filter].
    -e      specifies the maximum end time of the jobs that will be
       retrieved in YYYY-MM-DD. [default: no filter].
    -l      shows the information for each node for each job instead of
        the global statistics for said job.
    -x      shows the last EAR events. Nodes, job ids, and step ids can
        be specified as if it were showing job information.
    -m      prints power signatures regardless of whether mpi
       signatures are available or not.
    -r      shows the EAR loop signatures. Nodes, job ids, and step ids
        can be specified as if were showing job information.
    -o      modifies the -r option to also show the corresponding jobs.
        Should be used with -j.
    -n      specifies the number of jobs to be shown, starting from the
        most recent one. [default: 20][to get all jobs use -n all]
    -f      specifies the file where the user-database can be found. If
        this option is used, the information will be read from the file
        and not the database.
    -b      verbose mode for debugging purposes
```

The basic usage of `eacct` retrieves the last 20 applications (by default) of the user executing it. If a user is **privileged**, they may see all users applications. The default behaviour shows data from each job-step, aggregating the values from each node in said job-step. If using SLURM as a job manager, a *sb*

(sbatch) job-step is created with the data from the entire execution. A specific job may be specified with `-j` option.

Below table shows some examples of `eacct` usage.

| Command line | Description |
|---|---|
| eacct | Shows last 20 jobs executed by the user. |
| eacct -j <JobID> | Shows data of the job <JobID>, one row for each step of the job. |
| eacct -j <JobID>.<StepID> | Shows data of the step <StepID> of job <JobID>. |
| eacct -j <JobIDx>,<JobIDy>,<JobIDz> | Shows data of jobs (one row per step) <JobIDx>,<JobIDy> and <JobIDz>. |

The command shows a pre-selected set of columns:

| Column field | Description |
|---|---|
| JOB-STEP | JobID and StepID reported. JobID-*sb* is shown for the sbatch step in SLURM systems. |
| USER | The username of the user who executed the job. |
| APPLICATION | Job's name or executable name if job name is not provided. |
| POLICY | Energy optimization policy name. *MO* means for monitoring, *ME* for min_energy, *MT* for min_time and *NP* is the job ran without EARL. |
| NODES | Number of nodes involved in the job run. |
| AVG/DEF/IMC(GHz) | Average CPU frequency, default frequency and average uncore frequency. Includes all the nodes for the step. In GHz. |
| TIME(s) | Average step execution time along all nodes, in seconds. |
| POWER(W) | Average node power along all the nodes, in Watts. |
| GBS | CPU main memory bandwidth (GB/second). Hint for CPU/Memory bound classification. |
| CPI | CPU Cycles per Instruction. Hint for CPU/Memory bound classification. |
| ENERGY(J) | Accumulated node energy. Includes all the nodes. In Joules. |
| GFLOPS/W | CPU+GPU GFlops per Watt. Hint for energy efficiency. The metric uses the number of operations, not instructions. |
| IO(MBS) | I/O (read and write) Mega Bytes per second. |

| Column field | Description |
|---|---|
| MPI% | Percentage of MPI time over the total execution time. It's the average including all the processes and nodes. |

If EAR supports GPU monitoring/optimisation, the following columns are added:

| Column field | Description |
|---|---|
| G-POW (T/U) | Average GPU power. Accumulated per node and averaged along involved nodes. *T* mean for total GPU power consumed (even the job is not using any or all of GPUs in one node). *U* means for only used GPUs on each node. |
| G-FREQ | Average GPU frequency. Per node and average of all the nodes. |
| G-UTIL(G/MEM) | GPU utilization and GPU memory utilization. |

For node-specific information, the `-l` (i.e., long) option provides detailed accounting of each individual node. In addition, `eacct` shows an additional column: `VPI(%)`. The VPI means the percentage of AVX512 instructions over the total number of instructions.

For runtime data (EAR loops) one may retrieve them with `-r`. Both Job and Step ID filtering works. To easily transfer command's output, `-c` option saves it in .csv format. Both aggregated and detailed accountings are available, as well as filtering:

| Command line | Description |
|---|---|
| eacct -j <JobID> -c test.csv | Adds to the file `test.csv` all metrics shown above for each step if the job <JobID>. |
| eacct -j <JobID>.<StepID> -l -c test.csv | Appends to the file `test.csv` all metrics in the EAR DB for each node involved in step <StepID> of job <JobID>. |

| Command line | Description |
|---|---|
| eacct -j <JobID>.<StepID> -r -c test.csv | Appends to the file `test.csv` all metrics in EAR DB for each loop of each node involved in step <StepID> of job <JobID>. |

When requesting long format (i.e., `-l` option) or runtime metrics (i.e., `-r` option) to be stored in a CSV file (i.e., `-c` option), header names change from the output shown when you don't request CSV format. Below table shows header names of CSV file storing long information about jobs. **Bold** fields indicate they are just filled when the EAR Library (EARL) is enabled. Format is the same used by the csv report plug-in.

| Field name | Description |
|---|---|
| JOBID | The JobID. |
| STEPID | The StepID. |
| **APPID** | The EARL application ID used to identify the workload. This value is useful to identify different applications executed in a workflow within the same job-step. |
| USERID | The username of the user who executed the job. |
| GROUPID | The group name of the user who executed the job. |
| ACCOUNTID | The account name of the user who executed the job. |
| JOBNAME | Job's name or executable name if job name is not provided. |
| ENERGY_TAG | The energy tag used if the user set one for its job step. |
| JOB_START_TIME | The Unix Epoch timestamp in seconds when the job-step started. |
| JOB_END_TIME | The Unix Epoch timestamp in seconds when the job-step ended. |
| **JOB_EARL_START_TIME** | The Unix Epoch timestamp in seconds when the EARL started monitoring the job-step, i.e., the application start time. |

| Field name | Description |
|---|---|
| **JOB_EARL_END_TIME** | The Unix Epoch timestamp in seconds when the EARL ended monitoring the job-step, i.e., the application end time. |
| POLICY | Energy optimization policy name used by the EARL. **MO** means for *monitoring-only*, **ME** for *min_energy*, **MT** for *min_time* and **NP** is the job ran without EARL. |
| **POLICY_TH** | The policy threshold used by the optimization policy used by EARL. |
| **JOB_NPROCS** | The number of processes involved in the application. |
| JOB_TYPE | The job type. |
| JOB_DEF_FREQ | The default frequency at which the job started. |
| EARL_ENABLED | Indicates whether the job-step ran with the EARL enabled. |
| **EAR_LEARNING** | Whether the application was run in the learning phase. |
| NODENAME | The node name the rest of the row information belongs to. |
| **AVG_CPUFREQ_KHZ** | Average CPU frequency of the job step executed in the node, expressed in kHz. This value is computed by the EARL. |
| **AVG_IMCFREQ_KHZ** | Average uncore frequency of the job step executed in the node, expressed in kHz. **Default data fabric frequency on AMD sockets**. This value is computed by the EARL. |
| **DEF_FREQ_KHZ** | The default frequency of the job step executed in the node, expressed in kHz. This value corresponds to the default frequency the EAR Library sets at the beginning, and it has the same value as *JOB_DEF_FREQ*. |
| **TIME_SEC** | Execution time period (in seconds) which comprises the application metrics reported by the EARL. |

| Field name | Description |
|---|---|
| **CPI** | CPU Cycles per Instruction. Hint for CPU/Memory bound classification. |
| **TPI** | Memory transactions per Instruction. Hint for CPU/Memory bound classification. |
| **MEM_GBS** | CPU main memory bandwidth (GB/second). Hint for CPU/Memory bound classification. |
| **IO_MBS** | I/O (read and write) Mega Bytes per second. |
| **PERC_MPI** | Percentage of *TIME_SEC* spent in MPI calls. |
| **DC_NODE_POWER_W** | Average node power along the time period, in Watts. This value differs from *NODEMGR_DC_NODE_POWER_W* in that it is computed and reported by the EARL. |
| **DRAM_POWER_W** | Average DRAM power along the time period, in Watts. **Not available on AMD sockets**. This value differs from *NODEMGR_DRAM_POWER_W* in that it is computed and reported by the EARL. |
| **PCK_POWER_W** | Average RAPL package power along the time period, in Watts. This value shows the aggregated power of all sockets in a package. This value differs from *NODEMGR_PCK_POWER_W* in that it is computed and reported by the EARL. |
| **CYCLES** | Total number of cycles retrieved along the time period. |
| **INSTRUCTIONS** | Total number of instructions retrieved along the time period. |
| **CPU_GFLOPS** | Total number of giga-Floating point operations per second along the time period. |
| **CPU_UTIL** | The CPU time of the application. |
| **L1_MISSES** | Total number of L1 cache misses along the time period. |
| **L2_MISSES** | Total number of L2 cache misses along the time period. |

| Field name | Description |
|---|---|
| **L3_MISSES** | Total number of L3/LLC cache misses along the time period. |
| **SPOPS_SINGLE** | Total number of single precision 64 bit floating point operations. |
| **SPOPS_128** | Total number of single precision 128 bit floating point operations. |
| **SPOPS_256** | Total number of single precision 256 bit floating point operations. |
| **SPOPS_512** | Total number of single precision 512 bit floating point operations. |
| **DPOPS_SINGLE** | Total number of double precision 64 bit floating point operations. |
| **DPOPS_128** | Total number of double precision 128 bit floating point operations. |
| **DPOPS_256** | Total number of double precision 256 bit floating point operations. |
| **DPOPS_512** | Total number of double precision 512 floating point 512 operations. |
| NODEMGR_DC_NODE_POWER_W | Average node power along the time period, in Watts. This value differs from *DC_NODE_POWER_W* in that it is computed and reported by the Node Manager (the EARD) independently on whether the EARL was enabled. |
| NODEMGR_DRAM_POWER_W | Average DRAM power along the time period, in Watts. **Not available on AMD sockets**. This value differs from *DRAM_POWER_W* in that it is computed and reported by the Node Manager (the EARD) independently on whether the EARL was enabled. |

| Field name | Description |
|---|---|
| NODEMGR_PCK_POWER_W | Average RAPL package power along the time period, in Watts. This value shows the aggregated power of all sockets in a package. This value differs from *PCK_POWER_W* in that it is computed and reported by the Node Manager (the EARD) independently on whether the EARL was enabled. |
| NODEMGR_MAX_DC_POWER_W | The peak DC node power computed by the Node Manager. |
| NODEMGR_MIN_DC_POWER_W | The minimum DC node power computed by the Node Manager. |
| NODEMGR_TIME_SEC | Execution time period (in seconds) which comprises the job-step metrics reported by the Node Manager. |
| NODEMGR_AVG_CPUFREQ_KHZ | The average CPU frequency computed by the Node Manager during the job-step execution time. |
| NODEMGR_DEF_FREQ_KHZ | The default frequency set by the Node Manager when the job-step began. |

If EARL supports GPU monitoring/optimisation, the following columns are added:

| Field name | Description |
|---|---|
| GPU*x*_POWER_W | Average GPU*x* power, in Watts. |
| GPU*x*_FREQ_KHZ | Average GPU*x* frequency, in kHz. |
| GPU*x*_MEM_FREQ_KHZ | Average GPu*x* memory frequency, in kHz. |
| GPU*x*_UTIL_PERC | Average percentage of GPU*x* utilization. |
| GPU*x*_MEM_UTIL_PERC | Average percentage of GPU*x* memory utilization. |
| GPU*x*_GFLOPS | GPU*x* GFLOPS. |
| GPU*x*_TEMP | AverageGPU*x* temperature. |
| GPU*x*_MEMTEMP | AverageGPU*x* memory temperature. |

For runtime metrics (i.e., `-r` option), *USERID*, *GROUPID*, *JOBNAME*, *USER_ACC*, *ENERGY_TAG* (as energy tags disable EARL), *POLICY* and *POLICY_TH* are not stored at the CSV file. However, the iteration time

(in seconds) is present on each loop as *ITER_TIME_SEC*, as well as a timestamp (i.e., *TIMESTAMP*) with the Unix Epoch elapsed time in seconds.

### EAR system energy Report (ereport)

The ereport command creates reports from the energy accounting data from nodes stored in the EAR DB. It is intended to use for energy consumption analysis over a set period of time, with some additional (optional) criteria such as node name or username.

```
Usage: ereport [options]
Options are as follows:
        -s start_time            indicates the start of the period from which the energy consumed will be computed. Format: YYYY-MM-DD. Default: end_time minus insertion time*2.
        -e end_time              indicates the end of the period from which the energy consumed will be computed. Format: YYYY-MM-DD. Default: current time.
        -n node_name |all        indicates from which node the energy will be computed. Default: none (all nodes computed)
                                         'all' option shows all users individually, not aggregated.
        -u user_name |all        requests the energy consumed by a user in the selected period of time. Default: none (all users computed).
                                         'all' option shows all users individually, not aggregated.
        -t energy_tag|all        requests the energy consumed by energy tag in the selected period of time. Default: none (all tags computed).
                                         'all' option shows all tags individually, not aggregated.
        -i eardbd_name|all       indicates from which eardbd (island) the energy will be computed. Default: none (all islands computed)
                                         'all' option shows all eardbds individually, not aggregated.
        -g                       shows the contents of EAR's database Global_energy table. The default option will show the records for the two previous T2 periods of EARGM.
                                         This option can only be modified with -s, not -e
        -x                       shows the daemon events from -s to -e. If no time frame is specified, it shows the last 20 events.
        -v                       shows current EAR version.
```

```
17          -h                          shows this message.
```

### Examples

The following example uses the 'all' nodes option to display information for each node, as well as a start_time so it will give the accumulated energy from that moment until the current time.

```
1  [user@host EAR]$ ereport -n all -s 2018-09-18
2       Energy (J)        Node      Avg. Power (W)
3       20668697          node1        146
4       20305667          node2        144
5       20435720          node3        145
6       20050422          node4        142
7       20384664          node5        144
8       20432626          node6        145
9       18029624          node7        128
```

This example filters by EARDBD host (one per island typically) instead:

```
1  [user@host EAR]$ ereport -s 2019-05-19 -i all
2       Energy (J)         Node
3       9356791387         island1
4      30475201705         island2
5      37814151095         island3
6      28573716711         island4
7      29700149501         island5
8      26342209716         island6
```

And to see the state of the cluster's energy budget (set by the sysadmin) you can use the following:

```
1  [user@host EAR]$ ereport -g
2  Energy%  Warning lvl            Timestamp       INC th      p_state
      ENERGY T1    ENERGY T2      TIME T1      TIME T2        LIMIT
      POLICY
3  111.486          100  2019-05-22 10:31:34            0          100
                893      1011400       907200          600       604800
      EnergyBudget
4  111.492          100  2019-05-22 10:21:34            0          100
                859      1011456       907200          600       604800
      EnergyBudget
5  111.501          100  2019-05-22 10:11:34            0          100
                862      1011533       907200          600       604800
      EnergyBudget
6  111.514          100  2019-05-22 10:01:34            0          100
                842      1011658       907200          600       604800
      EnergyBudget
7  111.532          100  2019-05-22 09:51:34            0          100
                828      1011817       907200          600       604800
      EnergyBudget
```

```
8  111.554            0  2019-05-22 09:41:34            0            0
              837       1012019       907200          600       604800
      EnergyBudget
```

### EAR Control (econtrol)

The `econtrol` command modifies cluster settings (temporally) related to power policy settings. These options are sent to all the nodes in the cluster.

> **NOTE** Any changes done with `econtrol` will not be reflected in `ear.conf` and thus will be lost when reloading the system.

```
1  Usage: econtrol [options]
2          --status                                  ->requests the current
              status for all nodes. The ones responding show the current
3                                                       power, IP
                                                           address and
                                                           policy
                                                           configuration
                                                           . A list
                                                           with the
                                                           ones not
4                                                       responding is
                                                           provided
                                                           with their
                                                           hostnames
                                                           and IP
                                                           address.
5                                                       --status=
                                                           node_name
                                                           retrieves
                                                           the status
                                                           of that node

                                                           individually
                                                           .
6          --type          [status_type]           ->specifies what type
              of status will be requested: hardware,
7                                                       policy, full (
                                                           hardware+
                                                           policy),
                                                           app_node,
                                                           app_master,
                                                           eardbd,
                                                           eargm or
                                                           power. [
                                                           default:
                                                           hardware]
```

```
8         --power                                   ->requests the current
              power for the cluster.
9                                                       --power=
                                                            node_name
                                                            retrieves
                                                            the current
                                                            power of
                                                            that node
                                                            individually
                                                            .
10        --set-freq      [newfreq]                 ->sets the frequency of
               all nodes to the requested one
11        --set-def-freq  [newfreq]  [pol_name]     ->sets the default
              frequency for the selected policy
12        --set-max-freq  [newfreq]                 ->sets the maximum
              frequency
13        --set-powercap  [new_cap]                 ->sets the powercap of
              all nodes to the given value. A node can be specified
14                                                       after the
                                                            value to
                                                            only target
                                                             said node.
15        --hosts         [hostlist]                ->sends the command
              only to the specified hosts. Only works with status,
              power_status,
16                                                       --power and --
                                                            set-
                                                            powercap
17        --restore-conf                            ->restores the
              configuration for all nodes
18        --active-only                             ->supresses inactive
              nodes from the output in hardware status.
19        --health-check                            ->checks all EARDs and
              EARDBDs for errors and prints all that are unresponsive.
20        --domain [domain:target]        ->sends the requested command
              to the requested targets, effectively filtering
21                          which nodes receive the message. Available
                               domains are: tag, node, subcluster/
                               eargmid, island.
22
23        --mail [address]                          ->sends the output of
              the program to address.
24        --ping                                    ->pings all nodes to
              check whether the nodes are up or not. Additionally,
25                                                      --ping=
                                                            node_name
                                                            pings that
                                                            node
                                                            individually
                                                            .
26        --version                                 ->displays current EAR
```

```
                version.
27          --help                                  ->displays this message
                .
```

`econtrol`'s status is a useful tool to monitor the nodes in a cluster. The most basic usage is the hardware status (default type) which shows basic information of all the nodes.

```
1  [user@login]$ econtrol --status
2  hostname      power   temp    freq    job_id  stepid
3     node2        278    66C    2.59      6878       0
4     node3        274    57C    2.59      6878       0
5     node4         52    31C    1.69         0       0
6
7  INACTIVE NODES
8  node1   192.0.0.1
```

The application status type can be used to retrieve all currently running jobs in the cluster. `app_master` gives a summary of all the running applications while `app_node` gives detailed information of each node currently running a job.

```
1  [user@login]$ econtrol --status --type=app_master
2  Job-Step    Nodes   DC power      CPI      GBS   Gflops     Time Avg
      Freq
3    6878-0        2     280.13     0.37    24.39   137.57    54.00
        2.59
4
5  [user@login]$ econtrol --status --type=app_node
6  Node id     Job-Step   M-Rank   DC power      CPI      GBS   Gflops
         Time Avg Freq
7    node2       6878-0        0     280.13     0.37    24.39   137.57
        56.00     2.59
8    node3       6878-0        1     245.44     0.37    24.29   136.40
        56.00     2.59
```

A list of nodes can be specified to only target those for the commands:

```
1  [user@login]$ econtrol --status --hosts node2,node3
2  hostname      power   temp    freq    job_id  stepid
3     node2        278    66C    2.59      6878       0
4     node3        274    57C    2.59      6878       0
5
6  [user@login]$ econtrol --status --hosts island[0,1]node[2-3]
7  hostname      power   temp    freq    job_id  stepid
8  island0node2    278    66C    2.59         0       0
9  island0node3    274    57C    2.59         0       0
10 island1node2    273    56C    2.59         0       0
11 island1node3    272    57C    2.59         0       0
```

If only one node is targeted for a status, one may do:

```
[user@login]$ econtrol --status=island0node2
hostname      power   temp    freq    job_id  stepid
island0node2    278    66C    2.59         0       0
```

For any other command type (including status):

```
[user@login]$ econtrol --status --hosts island0node2
hostname      power   temp    freq    job_id  stepid
island0node2    278    66C    2.59         0       0

[user@login]$ econtrol --restore-conf --hosts island0node2

[user@login]$ econtrol --status --domain node:island0node2
hostname      power   temp    freq    job_id  stepid
island0node2    278    66C    2.59         0       0
```

Furthermore, the domain option may be used to filter out any nodes not belonging to a specified domain.

To only send a command to a single island (as defined in ear.conf):

```
[user@login]$ econtrol --restore-conf --domain island:2
```

To send to all the nodes that belong to a tag (as defined in ear.conf), regardless of their island or any other configuration:

```
[user@login]$ econtrol --set-powercap 400 --domain tag:cpu_only
```

One can also do use a double filter:

```
[user@login]$ econtrol --set-freq 3100000 --domain tag:cpu_only island
    :3
```

**NOTE**: Using domain filtering with a hostlist specification is not supported and may cause some errors.

### Database commands

#### edb_create

Creates the EAR DB used for accounting and for the global energy control. Requires root access to the MySQL server. It reads the `ear.conf` to get connection details (server IP and port), DB name (which may or may not have been previously created) and EAR's default users (which will be created or altered to have the necessary privileges on EAR's database).

```
Usage:edb_create [options]
        -p       Specify the password for MySQL's root user.
        -o       Outputs the commands that would run.
        -r       Runs the program. If '-o' this option will be
           overriden.
        -h       Shows this message.
```

#### edb_clean_pm

Cleans periodic metrics from the database. Used to reduce the size of EAR's database, it will remove every Periodic_metrics entry older than num_days:

```
Usage:./src/commands/edb_clean_pm [options]
    -d num_days     REQUIRED: Specify how many days will be kept in
       database. (default: 0 days).
    -p          Specify the password for MySQL's root user.
    -o          Print the query instead of running it (default: off).
    -r          Execute the query (default: on).
    -h          Display this message.
    -v          Show current EAR version.
```

#### edb_clean_apps

Removes applications from the database. It is intended to remove old applications to speed up queries and free up space. It can also be used to remove specific applications from database. It removes ALL the information related to those jobs (the following tables will be modified for each job: Loops, if they exist; GPU_signatures, if they exist; Signatures, if they exist; Power signatures, Applications, and Jobs).

It is recommended to run the application with the -o option first to ensure that the queries that will be executed are correct.

```
Usage:edb_clean_apps [-j/-d] [options]
    -p      The program will request the database user's password.
    -u user     Database user to execute the operation (it needs DELETE
        privileges). [default: root]
    -j jobid.stepid Job id and step id to delete. If no step_id is
       introduced, every step within the job will be deleted
    -d ndays    Days to preserve. It will delete any jobs older than
       ndays.
    -o      Prints out the queries that would be executed. Exclusive
       with -r. [default:on]
    -r      Runs the queries that would be executed. Exclusive with -o.
        [default:off]
```

```
    -l      Deletes Loops and its Signatures. [default:off]
    -a      Deletes Applications and related tables. [default:off]
    -h      Displays this message
```

### erun

`erun` is a program that simulates all the SLURM and EAR SLURM Plug-in pipeline. It was designed to provide compatibility between MPI implementations not fully compatible with SLURM SPANK plug-in mechanism (e.g., OpenMPI), which is used to set up EAR at job submission. You can launch `erun` with the `--program` option to specify the application name and arguments. See the usage below:

```
erun --help

This is the list of ERUN parameters:
Usage: ./erun [OPTIONS]

Options:
    --job-id=<arg>  Set the JOB_ID.
    --nodes=<arg>   Sets the number of nodes.
    --program=<arg> Sets the program to run.
    --clean     Removes the internal files.

SLURM options:
...
```

The syntax to run an MPI application with `erun` has the form `mpirun -n <X> erun --program ='my_app arg1 arg2 .. argN'`. Therefore, `mpirun` will run *X* erun processes. Then, `erun` will launch the application `my_app` with the arguments passed, if specified. You can use as many parameters as you want but the semicolons have to cover all of them in case there are more than just the program name.

`erun` will simulate on the remote node both the local and remote pipelines for all created processes. It has an internal system to avoid repeating functions that are executed just one time per job or node, like SLURM does with its plugins.

**IMPORTANT NOTE** If you are going to launch `n` applications with `erun` command through a sbatch job, you must set the environment variable `SLURM_STEP_ID` to values from `0` to `n-1` before each `mpirun` call. By this way `erun` will inform the EARD the correct step ID to be stored then to the Database.

The `--job-id` and `--nodes` parameters create the environment variables that SLURM would have created automatically, because it is possible that your application make use of them. The `--clean` option removes the temporal files created to synchronize all ERUN processes.

Also you have to load the EAR environment module or define its environment variables in your environment or script:

| Variable | Parameter |
|---|---|
| EAR_INSTALL_PATH=<path> | prefix=<path> |
| EAR_TMP=<path> | localstatedir=<path> |
| EAR_ETC=<path> | sysconfdir=<path> |
| EAR_DEFAULT=<on/off> | default=<on/off> |

### ear-info

`ear-info` is a tool created to quickly view useful information about the current EAR installation of the system. It shows relevant details for both users and administrators, such as configuration defaults, installation paths, etc.

```
[user@hostname ~]$ ear-info -h
Usage: ear-info [options]
    --node-conf[=nodename]
    --help
```

The tool prints out information without giving it any argument. It shows a resume about EAR parameters set at compile time, as well as some installation dependent configuration: - The current EAR version. - The maximum number of CPUs/processors supported. - The maximum number of sockets supported. - Whether the current installation provides support for GPUs. - The default optimization policy. - Whether the EAR Library is enabled by default on job submission. - Information about EAR's Uncore Frequency Scaling policy (eUFS) configuration. - EAR's dynamic load balancing policy. - EAR's application phase classification. - EAR's MPI stats collection feature. - EAR data reporting mechanism configuration.

Below there is an example of the output:

```
EAR version 4.3
Max CPUS supported set to 256
Max sockets supported set to 4
EAR installed with GPU support MAX_GPUS 8
Default cluster policy is monitoring
EAR optimization  by default set to 0


Environment configuration section..............
                                    eUFS  1
                                eUFS limit 0.02
```

```
                    Load balanced enabled 1
                          Load Balance th 0.80
             Use turbo for critical path 1
                               Use turbo 0
                           Exclusive mode 0
                          Use EARL phases 1
                        Use energy models 1
     Max IMC frequency (0 = not defined) 0
     Min IMC frequency (0 = not defined) 0
 GPU frequency/pstate (0 = max GPU freq) 0
                          MPI optimization 0
                            MPI statistics 0
                               App. Tracer no trace
               App. Extra report plugins no extra plugins
            App. reporting loops to EARD 1
.........................................


HACK section.........................
                             Install path /hpc/base/ctt/packages/EAR/ear
              Energy optimization policy
                          GPU power policy /hpc/base/ctt/packages/EAR/ear
                             /lib/plugins/policies/gpu_monitoring.so
                          CPU power  model /hpc/base/ctt/packages/EAR/ear
                             /lib/plugins/policies/gpu_monitoring.so
                 CPU shared power  model /hpc/base/ctt/packages/EAR/ear
                    /lib/plugins/models/cpu_power_model_default.so
.........................................
```

EAR was designed to be installed on heterogeneous systems, so some configuration parameters that are applied to a set of nodes identified by different tags. The `--node-conf` flag can be used to request additional information about a specific node. Configuration related to EAR's power capping sub-system, default optimization policies configuration and other parameters associated with the node requested are retrieved. You can read the EAR configuration section for more details about how EAR uses tags to identify and configure different kind of nodes on a given heterogeneous system.

Contact with ear-support@bsc.es for more information about the nomenclature used by `ear-info`'s output.

## EAR environment variables

### Introduction

EAR offers some environment variables in order to provide users the opportunity to tune or request some of EAR features. They must be exported before the job submission, e.g., in the batch script.

The current EAR version has support for SLURM, PBS and OAR batch schedulers. In SLURM systems the scheduler may filter environment variables not prefixed with *SLURM_* character set (this happens when the batch script is submitted purging all environment variables to work in a clean environment). For that reason, the first design of EAR environment variables was to have variable names with the form *SLURM_*<variable_name>.

Now that EAR has support for other batch schedulers, and in order to maintain the coherency of environment variables names, below environment variables need the prefix of the scheduler used on the system the job is submitted on, plus an underscore. For example, in SLURM systems, the environment variable presented as `EAR_LOADER_APPLICATION` must be exported as `SLURM_EAR_LOADER_APPLICATION` in the submission batch script. In an OAR installed system, this variable would be exported as `OAR_EAR_LOADER_APPLICATION`. This design may only have a real effect on SLURM systems, but it makes it easier for the development team to provide support for multiple batch schedulers.

All examples showing the usage of below environment variables assume a system using SLURM.

## Loading EAR Library

### EAR_LOADER_APPLICATION

Rules the EAR Loader to load the EAR Library for a specific application that does not follow any of the current programming models (or maybe a sequential app) supported by EAR. Your system must have installed the non-MPI version of the Library (ask your system administrator).

The value of the environment variable must coincide with the job name of the application you want to launch with EAR. If you don't provide it, the EAR Loader will compare it against the executable name. For example:

```
1 #!/bin/bash
2
3 export EAR_LOADER_APPLICATION=my_job_name
4
5 srun --ntasks 1 --job-name=my_job_name ./my_exec_file
```

See the Use cases section to read more information about how to run jobs with EAR.

### EAR_LOAD_MPI_VERSION

Forces to load a specific MPI version of the EAR Library. This is needed, for example, when you want to load the EAR Library for Python + MPI applications, where the Loader is not able to detect the MPI

implementation the application is going to use. Accepted values are either *intel* or *open mpi*. The following example runs Tensorflow 1 benchmarks for several convulational neural networks with EAR. It can be downloaded from Tensorflow benchmarks repository.

```
#!/bin/bash

#SBATCH --job-name=TensorFlow
#SBATCH -N 8
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=18

## Specific modules here
## ...

export EAR_LOAD_MPI_VERSION="open mpi"

srun --ear-policy=min_time \
    python benchmarks/scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py \
    ... more application options
```

See the Use cases section to read more information about how to run jobs with EAR.

## Report plug-ins

### EAR_REPORT_ADD

Specify a report plug-in to be loaded. The value must be a shared object file, and it must be located at `$EAR_INSTALL_PATH/lib/plugins/report` or at the path from where the job was launched. Alternatively, you can provide the full path (absolute or relative) of the report plug-in.

```
#!/bin/bash

export EAR_REPORT_ADD=my_report_plugin.so:my_report_plugin2.so

srun -n 10 my_mpi_app
```

For more information, see Report plug-ins section.

### Extended GPU metrics

EAR library collects a set of extra GPU metrics on demand. To enable and report it two env var has to be defined `EAR_GPU_DCGMI_ENABLED` to enable them, the dcgmi.so report plugin has to be requested, and the `DCGMI_LOGS_PATH` has to be set to specify the path for csv generated. This last env var is

optional, if not set, the current working directory is used. By default, only few metrics are gathered, to collect all of them set the `EAR_DCGM_ALL_EVENTS` env var.

### Verbosity

#### EARL_VERBOSE_PATH

Specify a path to create a file (one per node involved in a job) where to print messages from the EAR Library. This is useful when you run a job in multiple nodes, as EAR verbose information for each of them can result in lots of messages mixed at stderr (EAR messages default channel). Also, there are applications that print information in both stdout and stderr, so maybe a user wants to have information separated.

If the path does not exist, EAR will create it. The format of generated files names is `earl_log.<node_rank>.<local_rank>.<job_step>.<job_id>`, where the *node_rank* is an integer set by EAR from 0 to *n_nodes - 1* involved in the job, and it indicates to which node the information belongs to. The local rank is an arbitrary rank set by EAR of a process in the node (from 0 to *n_procceses_in_node - 1*). It indicates which process is printing messages to the files, and it will be always the first one indexed, i.e., 0. Finally, the *job_step* and *job_id* are fields showing information about the job corresponding to the execution from where messages were generated.

```
1  #!/bin/bash
2
3  #SBATCH -j my_job_name
4  #SBATCH -N 2
5  #SBATCH -n 96
6
7  export EARL_VERBOSE_PATH=ear_logs_dir_name
8  export I_MPI_HYDRA_BOOTSTRAP=slurm
9  export I_MPI_HYDRA_BOOTSTRAP_EXEC_EXTRA_ARGS="--ear-verbose=1"
10
11 mpirun -np 96 -ppn 48 my_app
```

After the above job example completion, in the same directory where the application was submitted, there will be a directory called *ear_logs_dir_name* with two files, i.e., one for each node, called *earl_logs.0.0.*. and *earl_logs.1.0.*., respectively.

## Frequency management

### EAR_GPU_DEF_FREQ

Set a GPU frequency (in kHz) to be fixed while your job is running. The same frequency is set for all GPUs used by the job.

```
#!/bin/bash

#SBATCH -J gromacs-cuda
#SBATCH -N 1

export I_MPI_PIN=1
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi2.so

input_path=/hpc/appl/biology/GROMACS/examples
input_file=ion_channel.tpr
GROMACS_INPUT=$input_path/$input_file

export EAR_GPU_DEF_FREQ=1440000

srun --cpu-bind=core --ear-policy=min_energy gmx_mpi mdrun \
    -s $GROMACS_INPUT -noconfout -ntomp 1
```

### EAR_JOB_EXCLUSIVE_MODE

Indicate whether the job will run in a node exclusively (non-zero value). EAR will reduce the CPU frequency of those cores not used by the job. This feature explodes a very easy vector of power saving.

```
#!/bin/bash
#SBATCH -N 1
#SBATCH -n 64
#SBATCH --cpus-per-task=2
#SBATCH --exclusive

export EAR_JOB_EXCLUSIVE_MODE=1

srun -n 10 --ear=on ./mpi_mpi_app
```

### Controlling Uncore/Infinity Fabric frequency

EARL offers the possibility to control the Integrated Memory Controller (IMC) for Intel(R) architectures and Infinity Fabric (IF) for AMD architectures. On this page we will use the term *uncore* to refer both of

them. Environment variables related to uncore control covers policy specific settings or the chance for a user to fix it during an entire job.

**EAR_SET_IMCFREQ** Enables/disables EAR's eUFS feature. Type `ear-info` to see whehter eUFS is enabled by default.

You can control eUFS' maximum permitted time penalty by exporting `EAR_POLICY_IMC_TH`, which is a float indicating the threshold value that prevents the policy to reduce so much the uncore frequency, possible leading to considerable performance penalty.

Below example enables eUFS with a penalty threshold of 3.5%:

```
#!/bin/bash
...

export SLURM_EAR_SET_IMCFREQ=1
export SLURM_EAR_POLICY_IMC_TH=0.035
...

srun [...] my_app
```

**EAR_MAX_IMCFREQ and EAR_MIN_IMCFREQ** Set the maximum and minimum values (in kHz) at which *uncore* frequency should be. Two variables were designed because Intel(R) architectures let to set a range of frequencies that limits its internal UFS mechanism. If you set both variables with different values, the minimum one will be set.

Below example shows a job execution fixing the uncore frequency at 2.0GHz:

```
#!/bin/bash
...

export EAR_MAX_IMCFREQ=2000000
export EAR_MIN_IMCFREQ=2000000
...

srun [...] my_app
```

### Load Balancing

By default, EAR policies try to set the best CPU (and uncore, if enabled) frequency according to node grain metrics. This behaviour can be changed telling EAR to detect and deal with unbalanced workloads, i.e., there is no equity between processes regarding their MPI/computational activity.

When EAR detects such behaviour, policies slightly modify its way of CPU frequency selection by setting a different frequency for each process' cores according how far it is from the critical path. Please, contact with ear-support@bsc.es if you want more details about how it works.

> A correct CPU binding it's required to get the most benefit of this feature. Check the documentation of your application programming model/vendor/flavour or yur system batch scheduler.

**EAR_LOAD_BALANCE** Enables/Disables EAR's Load Balance strategy in energy policies. Type `ear-info` to see whether this feature is enabled by default.

Load unbalance detection algorithm is based on POP-CoE's Load Balance Efficiency metric, which is computed as the ratio between average useful computation time (across all processes) and maximum useful computation time (also across all processes). By default (if `EAR_LOAD_BALANCE` is enabled), a node load balance efficiency below **0.8** will trigger EAR's Load Balancing algorithm. This threshold value can be modified by setting `EAR_LOAD_BALANCE_TH` environment variable. For example, if you want to be more permissive with the application load balance and prevent per-process CPU frequency selection, you can increase the load balance threshold:

```
#!/bin/bash
...

export EAR_LOAD_BALANCE=1
export EAR_LOAD_BALANCE_TH=0.89
...

srun [...] my_app
```

#### Support for Intel(R) Speed Select Technology

Since version 4.2, EAR supports the interaction with Intel(R) Speed Select Technology (Intel(R) SST) which lets the user to have more fine grained control over per-CPU Turbo frequency. This feature opens a door to users for getting more control over the performance (also power consumption) across CPUs running their applications and jobs. It is available on selected SKUs of Intel(R) Xeon(R) Scalable processors. For more information about Intel(R) SST, below are listed useful links to official documentation:

- Intel(R) SST-CP
- Intel(R) SST-TF
- The Linux Kernel: Intel(R) Speed Select Technology User Guide

EAR offers two environment variables that let to specify a list of priorities (CLOS) in two different ways. The first one will set a CLOS for each task involved in the job. On the other hand, the second offered

variable will set a list of priorities per CPU involved in the job. Values must be within the range of available CLOS that Intel(R) SST provides you.

If some of the two supported environment variables are set, EAR will set-up all of its internals transparently if the architecture supports it. Also, it will restore configuration on the job ending. If Intel(R) SST is not supported, no effect will occur. If you enable EARL verbosity you will see the mapping of the CLOS set for each CPU in the node. Note that a `-1` value means that no change was done on the specific CPU.

**EAR_PRIO_TASKS** A list that specifies the CLOS that CPUs assigned to tasks must be set. This variable is useful because you can configure your application transparently without concerning about the affinity mask that the scheduler is assigning to your tasks. You can use this variable when you know (or guess) your application's tasks workload and you want to tune it by setting manually different Turbo priorities. Note that you still need to ensure that different tasks do not share CPUs.

For example, imagine you want to submit a job that runs a MPI application with 16 tasks, each one pinned on a single core, in a two-socket Intel(R) Xeon(R) Platinum 8352Y with 32 cores each, with Hyper-threading enabled, i.e., each task will run on two CPUs and 32 of the total 128 will be allocated by this application. Below could be a (simplified) batch script that submits this example:

```
#!/bin/bash

export EAR_PRIO_TASKS=0,0,0,0,1,1,1,1,2,2,2,2,3,3,3,3

srun --ntasks=16 --cpu-bind=core,verbose --ear-policy=monitoring --ear-
    cpufreq=2201000 --ear-verbose=1 bin/bt.C.x
```

The above script sets CLOS 0 to tasks 0 to 3, CLOS 1 to tasks 4 to 7, CLOS 2 to tasks 8 to 11 and CLOS 3 to tasks 12 to 15. The `srun` command binds each task to one core (through `--cpu-bind` flag), sets the turbo frequency and enables EAR verbosity. Below there is the output message shown by the batch scheduler (i.e., SLURM):

```
cpu-bind=MASK - ice2745, task  0  0 [23363]: mask 0x10000000000000001
    set
cpu-bind=MASK - ice2745, task  1  1 [23364]: mask 0
    x10000000000000000100000000 set
cpu-bind=MASK - ice2745, task  2  2 [23365]: mask 0x20000000000000002
    set
cpu-bind=MASK - ice2745, task  3  3 [23366]: mask 0
    x20000000000000000200000000 set
cpu-bind=MASK - ice2745, task  4  4 [23367]: mask 0x40000000000000004
    set
cpu-bind=MASK - ice2745, task  5  5 [23368]: mask 0
    x40000000000000000400000000 set
```

```
cpu-bind=MASK - ice2745, task  6  6 [23369]: mask 0x80000000000000008 set
cpu-bind=MASK - ice2745, task  7  7 [23370]: mask 0 x80000000000000000800000000 set
cpu-bind=MASK - ice2745, task  8  8 [23371]: mask 0x100000000000000010 set
cpu-bind=MASK - ice2745, task  9  9 [23372]: mask 0 x10000000000000001000000000 set
cpu-bind=MASK - ice2745, task 10 10 [23373]: mask 0x200000000000000020 set
cpu-bind=MASK - ice2745, task 11 11 [23374]: mask 0 x20000000000000002000000000 set
cpu-bind=MASK - ice2745, task 12 12 [23375]: mask 0x400000000000000040 set
cpu-bind=MASK - ice2745, task 13 13 [23376]: mask 0 x40000000000000004000000000 set
cpu-bind=MASK - ice2745, task 14 14 [23377]: mask 0x800000000000000080 set
cpu-bind=MASK - ice2745, task 15 15 [23378]: mask 0 x80000000000000008000000000 set
```

We can see here that SLURM spreaded out tasks accross the two sockets of the node, e.g., task 0 runs on CPUs 0 and 64, task 1 runs on CPUs 32 and 96. Below output shows how EAR sets and verboses CLOS list per CPU in the node. Following the same example, you can see that CPUs 0, 64, 32 and 96 have priority/CLOS 0. Note that those CPUs not involved in the job show a -1.

```
Setting user-provided CPU priorities...
PRIO0: MAX GHZ - 0.0 GHz (high)
PRIO1: MAX GHZ - 0.0 GHz (high)
PRIO2: MAX GHZ - 0.0 GHz (low)
PRIO3: MAX GHZ - 0.0 GHz (low)
[000, 0] [001, 0] [002, 1] [003, 1] [004, 2] [005, 2] [006, 3] [007, 3]
[008,-1] [009,-1] [010,-1] [011,-1] [012,-1] [013,-1] [014,-1] [015,-1]
[016,-1] [017,-1] [018,-1] [019,-1] [020,-1] [021,-1] [022,-1] [023,-1]
[024,-1] [025,-1] [026,-1] [027,-1] [028,-1] [029,-1] [030,-1] [031,-1]
[032, 0] [033, 0] [034, 1] [035, 1] [036, 2] [037, 2] [038, 3] [039, 3]
[040,-1] [041,-1] [042,-1] [043,-1] [044,-1] [045,-1] [046,-1] [047,-1]
[048,-1] [049,-1] [050,-1] [051,-1] [052,-1] [053,-1] [054,-1] [055,-1]
[056,-1] [057,-1] [058,-1] [059,-1] [060,-1] [061,-1] [062,-1] [063,-1]
[064, 0] [065, 0] [066, 1] [067, 1] [068, 2] [069, 2] [070, 3] [071, 3]
[072,-1] [073,-1] [074,-1] [075,-1] [076,-1] [077,-1] [078,-1] [079,-1]
[080,-1] [081,-1] [082,-1] [083,-1] [084,-1] [085,-1] [086,-1] [087,-1]
[088,-1] [089,-1] [090,-1] [091,-1] [092,-1] [093,-1] [094,-1] [095,-1]
[096, 0] [097, 0] [098, 1] [099, 1] [100, 2] [101, 2] [102, 3] [103, 3]
[104,-1] [105,-1] [106,-1] [107,-1] [108,-1] [109,-1] [110,-1] [111,-1]
[112,-1] [113,-1] [114,-1] [115,-1] [116,-1] [117,-1] [118,-1] [119,-1]
[120,-1] [121,-1] [122,-1] [123,-1] [124,-1] [125,-1] [126,-1] [127,-1]
```

**EAR_PRIO_CPUS** A list of priorities that should have the same length as the number of CPUs your job is using. This configuration lets to set up CPUs CLOS in a more low level way: **the *n-th* priority value of the list will set the priority of the *n-th* CPU your job is using.**

This way of configuring priorities rules the user to know exactly the affinity of its job's tasks before launching the application, so it becomes harder to use if your goal is the same as the one you can get by setting the above environment variable: task-focused CLOS setting. But it becomes more flexible when the user has more control over the affinity set to its application, because you can discriminate between different CPUs assigned to the same task. Moreover, this is the only way to set different priorities over different threads in no-MPI applications.

#### EAR_MIN_CPUFREQ

This variable can only be set by **authorized users**, and modifies the minimum CPU frequency the EAR Library can set. The EAR configuration file has a field called `cpu_max_pstate` which sets this limits on the tag it is configured. Authorized users can modify this limit at submission time by using this environment to test, for example, the best value for the `ear.conf` field.

#### Disabling EAR's affinity masks usage

For both Load Balancing and Intel(R) SST support, EAR uses processes' affinity mask read at the beginning of the job. If you are working on an application that changes (or may change) the affinty mask of tasks dynamically, this can lead some miss configuration not detected by EAR. To avoid any unexpected problem, **we highly recommend you** to export `EARL_NO_AFFINITY_MASK` environment variable **even you are not planning to work with some of the mentioned features**.

Note: Since EAR version 5.0, EAR updates the process mask periodically (aprox 1 sec.) and always before applying the optimization policy.

### Workflow support

#### EAR_DISABLE_NODE_METRICS

By defining this environment variable, the user or workflow manager indicates EAR the current process must not be considered as power consumer, not affecting the CPU power models used to estimate the amount of power corresponding to each application sharing a node. This env variable target master-worker scenarios (or map-reduce) when one process is not doing computational work, just working as master creating and waiting for processes. By specifying this var, the EARL ignores the

affinity mas for this process and assumes its activity is not relevant for the whole power consumption. The value is not relevant, it only has to be defined. In a fork-join program (or similar, it has to be unset before the creation of the workers.

#### EAR_NTASK_WORK_SHARING

By defining this environment variable, the user indicates the library the set of processes sharing the node are in fact a single application (not MPI). This enables a synchronization at the beginning and all the processes with same jobid and stepid (or similar for other schedulers different than SLURM) works together. Only one of the will be selected as the master and will apply the energy policy. For GPU applications it's mandatory the process can access all the GPUs. Otherwise, it is not recommended and each process will apply its own optimization. The value is not relevant, it only has to be defined. **This feature is only supported on systems using SLURM**.

### Data gathering/reporting

#### EARL_REPORT_LOOPS

Since **version 4.3**, EAR can be configured to not report application loop signatures by default. This configuration satisfy a constraint for many HPC data centers where hundreds of jobs are launched daily, leading to too many loops reported and a quick EAR database size increase.

For those users which still want to get application loop data, this variable can be set to one (i.e., `export EARL_REPORT_LOOPS=1`) to force EAR report their application loop signatures. Therefore, users can get their loop data by calling `eacct -j <job_id> -r`.

#### EAR_GET_MPI_STATS

Use this variable to generate two files at the end of the job execution that will contain global, per process MPI information. You must specify the prefix (optionally with a path) of the filename. One file (*[path/]prefix.ear_mpi_stats.full_nodename.csv*) will contain a resume about MPI throughput (per-process), while the other one (*[path/]prefix.ear_mpi_calls_stats.full_nodename.csv*) will contain a more fine grained information about different MPI call types. Here is an example:

```
1 !#/bin/bash
2
3 #SBATCH -j mpi_job_name
4 #SBATCH -n 48
5
6 MPI_INFO_DST=$SLURM_JOBID-mpi_stats
```

```
7  mkdir $MPI_INFO_DST
8
9  export EAR_GET_MPI_STATS=$MPI_INFO_DST/$SLURM_JOB_NAME
10
11 srun -n 48 --ear=on ./mpi_app
```

At the end of the job, two files will be created at the directory named *-mpi_stats* located in the same directory where the application was submitted. They will be named *mpi_job_name.ear_mpi_stats.full_nodename.csv* and *mpi_job_name.ear_mpi_calls_stats.full_nodename.csv*. File pairs will be created for each node involved in the job.

Take into account that each process appends its own MPI statistics to files. This behavior does not guarantee that the header of files will be on the first line of them, as only one process writes it. You must move it at the top of each file manually before reading them with some tool you use to visualize and work with CSV files, e.g., spreadsheet, a R or Python package.

Below table shows fields available by **ear_mpi_stats** file:

| Field | Description |
|---|---|
| mrank | The EAR's internal node ID used to identify the node. |
| lrank | The EAR's internal rank ID used to identify the process. |
| total_mpi_calls | The total number of MPI calls. |
| exec_time | The execution time, in microseconds. |
| mpi_time | The time spent in MPI calls, in microseconds. |
| perc_mpi_time | The percentage of total execution time (i.e., *exec_time*) spent in MPI calls. |

Below table shows fields available by **ear_mpi_calls_stats** file:

| Field | Description |
|---|---|
| Master | The EAR's internal node ID used to identify the node. |
| Rank | The EAR's internal rank ID used to identify the process. |
| Total MPI calls | The total number of MPI calls. |
| MPI_time/Exec_time | The ration between time spent in MPI calls and the total execution time. |

| Field | Description |
|---|---|
| Exec_time | The execution time, in microseconds. |
| Sync_time | Time spent (in microseconds) in **blocking** synchronization calls, i.e., MPI_Wait, MPI_Waitall, MPI_Waitany, MPI_Waitsome and MPI_Barrier. |
| Block_time | Time spent in blocking calls, i.e., MPI_Allgather, MPI_Allgatherv, MPI_Allreduce, MPI_Alltoall, MPI_Alltoallv, MPI_Barrier, MPI_Bcast, MPI_Bsend, MPI_Cart_create, MPI_Gather, MPI_Gatherv, MPI_Recv, MPI_Reduce, MPI_Reduce_scatter, MPI_Rsend, MPI_Scan, MPI_Scatter, MPI_Scatterv, MPI_Send, MPI_Sendrecv, MPI_Sendrecv_replace, MPI_Ssend and all *Wait* calls of **Sync_time** field. |
| Collec_time | Time spent in **blocking** collective calls, i.e., MPI_Allreduce, MPI_Reduce and MPI_Reduce_scatter. |
| Total MPI sync calls | Total number of synchronization calls. |
| Total blocking calls | Total number of blocking calls. |
| Total collective calls | Total number of collective calls. |
| Gather | Total number of **blocking** Gather calls, i.e., MPI_Allgather, MPI_Allgatherv, MPI_Gather and MPI_Gatherv. |
| Reduce | Total number of **blocking** Reduce calls, i.e., MPI_Allreduce, MPI_Reduce and MPI_Reduce_scatter. |
| All2all | Total number of **blocking** All2all calls, i.e., MPI_Alltoall and MPI_Alltoallv. |
| Barrier | Total number of **blocking** Barrier calls, i.e., MPI_Barrier. |
| Bcast | Total number of **blocking** Bcast calls, i.e., MPI_Bcast. |
| Send | Total number of **blocking** Send calls, i.e., MPI_Bsend, MPI_Rsend, MPI_Send and MPI_Ssend. |
| Comm | Total number of **blocking** Comm calls, i.e., MPI_Cart_create. |
| Receive | Total number of **blocking** Receive calls, i.e., MPI_Recv. |
| Scan | Total number of **blocking** Scan calls, i.e., MPI_Scan. |
| Scatter | Total number of **blocking** Scatter calls, i.e., MPI_Scatter and MPI_Scatterv. |

| Field | Description |
|---|---|
| SendRecv | Total number of **blocking** SendRecv calls, i.e., MPI_Sendrecv, MPI_Sendrecv_replace. |
| Wait | Total number of **blocking** Wait calls, i.e., all MPI_Wait calls. |
| t_Gather | Time (in microseconds) spent in **blocking** Gather calls. |
| t_Reduce | Time (in microseconds) spent in **blocking** Reduce calls. |
| t_All2all | Time (in microseconds) spent in **blocking** All2all calls. |
| t_Barrier | Time (in microseconds) spent in **blocking** Barrier calls. |
| t_Bcast | Time (in microseconds) spent in **blocking** Bcast calls. |
| t_Send | Time (in microseconds) spent in **blocking** Send calls. |
| t_Comm | Time (in microseconds) spent in **blocking** Comm calls. |
| t_Receive | Time (in microseconds) spent in **blocking** Receive calls. |
| t_Scan | Time (in microseconds) spent in **blocking** Scan calls. |
| t_Scatter | Time (in microseconds) spent in **blocking** Scatter calls. |
| t_SendRecv | Time (in microseconds) spent in **blocking** SendRecv calls. |
| t_Wait | Time (in microseconds) spent in **blocking** Wait calls. |

#### EAR_TRACE_PLUGIN

EAR offers the chance to generate Paraver traces to visualize runtime metrics with the Paraver tool. Paraver is a visualization tool developed by CEPBA-Tools team and currently maintained by the Barcelona Supercomputing Center's tools team.

The EAR trace generation mechanism was designed to support different trace generation plug-ins although the Paraver trace plug-in is the only supported by now. You must set the value of this variable to `tracer_paraver.so` to load the tracer. This shared object comes with the official EAR distribution and it is located at `$EAR_INSTALL_PATH/lib/plugins/tracer`. Then you need to set the `EAR_TRACE_PATH` variable (see below) to specify the destination path of the generated Paraver traces.

#### EAR_TRACE_PATH

Specify the path where you want to store the trace files generated by the EAR Library. The path must be fully created. Otherwise, the Paraver tracer plug-in won't be loaded.

Here is an example of the usage of the above explained environment variables:

```
#!/bin/bash
...

export EAR_TRACE_PLUGIN=tracer_paraver.so
export EAR_TRACE_PATH=$(pwd)/traces
mkdir -p $EAR_TRACE_PATH

srun -n 10 --ear=on ./mpi_app
```

#### REPORT_EARL_EVENTS

Use this variable (i.e., `export REPORT_EARL_EVENTS=1`) to make EARL send internal events to the Database. These events are useful to have more information about Library's behaviour, like when DynAIS **(REFERENCE DYNAIS)** is turned off, the computational phase EAR is guessing the application is on or the status of the applied policy **(REF POLICIES)**. You can query job-specific events through `eacct -j <JobID> -x`, and you will get a table of all reported events:

| Field name | Description |
|---|---|
| Event_ID | Internal ID of the event stored at the Database. |
| Timestamp | *yyyy-mm-dd hh:mm:ss*. |
| Event_type | Which kind of event is it. Possible event types explained below. |
| Job_id | The JobID of the event. |
| Value | The value stored with the event. Categorical events explained below. |
| node_id | The node from where the event was reported. |

**Event types** Below are listed all kind of event types you can get when requesting job events. For categorical event values, the (value, category) mapping is explained.

- **policy_error** Reported when the policy couldn't select the optimal frequency.

- **dynais_off** Reported when DynAIS is turned off and the Library becomes in *periodic monitoring mode*.
- **earl_state** The internal EARL state. Possible values are:
  - **0** This is the initial state and stands for no iteration detected.
  - **1** EAR starts computing the signature
  - **2** EAR computes the local signature and executes the per-node policy.
  - **3** This state computes a new signature and evaluates the accuracy of the policy.
  - **4** Projection error.
  - **5** This is a transition state to recompute EARL timings just in case we need to adapt it because of the frequency selection.
  - **6** Signature has changed.
- **optim_accuracy** The internal optimization policy state. Possible values are:
  - **0** Policy not ready.
  - **1** Policy says all is ok.
  - **2** Policy says it'n not ok.
  - **3** Policy wants to try again to optimize.

> The above event types may be useful only for advanced users. Please, contact with ear-support@bsc.es if you want to know more about EARL internals.

- **energy_saving** Energy (in %) EAR is guessing the policy is saving.
- **power_saving** Power in (in %) EAR is guessing the policy is saving.
- **performance_penalty** Execution time (in %) EAR is guessing the policy is incrementing.