DLB User Guide

Release 1.2

Barcelona Supercomputing Center

January 11, 2017

CONTENTS

1 Introduction to DLB **1**
1.1 LeWI: Lend When Idle . . . 1

2 Technical Requierements **3**
2.1 Programming models . . . 3
2.2 Shared Memory between processes . . . 3
2.3 Preload mechanism for MPI applications . . . 3
2.4 Parallel regions in OpenMP . . . 3
2.5 Non-busy waiting for MPI calls . . . 3

3 How to install DLB **5**
3.1 Build requirements . . . 5
3.2 Installation steps . . . 5
3.3 DLB configure flags . . . 5

4 How to run with DLB **7**
4.1 Examples by Programming Model . . . 7

5 Public API **9**
5.1 Basic set of DLB API . . . 9
5.2 Advanced set of DLB API . . . 10

6 FAQ: Frequently Asked Questions **13**
6.1 Does my application need to meet any requirements to run with DLB? . . . 13
6.2 Mercurium may have DLB support with the `--dlb` flag. What does it do? Should I use it? . . . 13
6.3 Which policy should I use for DLB? . . . 13
6.4 DLB fails registering a mask. What does it mean? . . . 14
6.5 I'm running a hybrid MPI + OpenMP application but DLB doesn't seem to have any impact . . . 14
6.6 I'm running a hybrid application, 1 thread per process, and DLB still does nothing . . . 14
6.7 I'm running a well allocated hybrid MPI + OpenMP but DLB still doesn't do anything. . . . 14
6.8 Can I see DLB events in a Paraver trace? . . . 15
6.9 Can I disable DLB tracing in an OmpSs execution? . . . 15

7 Content of the Package **17**
7.1 Structure . . . 17
7.2 Binaries . . . 17
7.3 Libraries . . . 17
7.4 Examples . . . 18

Index **19**

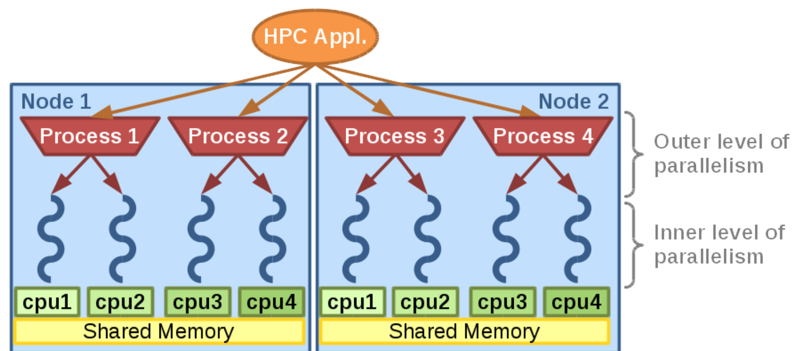
CHAPTER

ONE

INTRODUCTION TO DLB

The DLB library aims to improve the load balance of HPC hybrid applications (i.e., two levels of parallelism).

In the following picture we can see the structure of a typical HPC hybrid application: one application that will run processes on several nodes, and each process will spawn several threads.



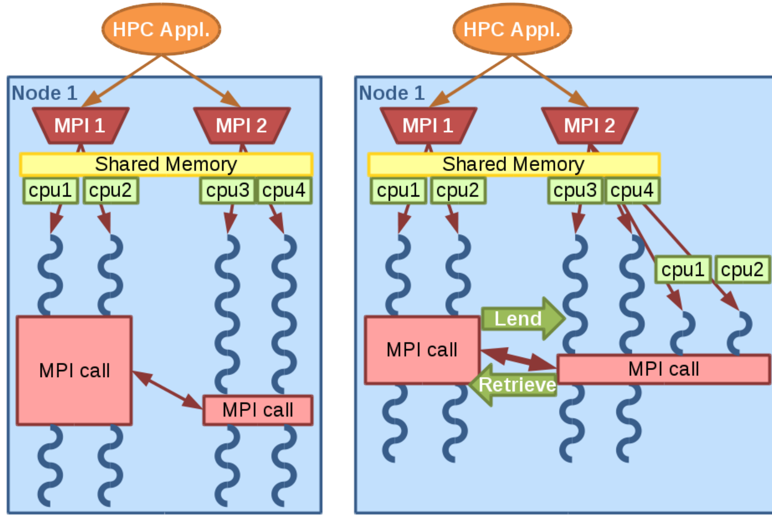
The DLB library will improve the load balance of the outer level of parallelism (e.g MPI) by redistributing the computational resources at the inner level of parallelism (e.g. OpenMP). This readjustment of resources will be done dynamically at runtime.

This dynamism allows DLB to react to different sources of imbalance: Algorithm, data, hardware architecture and resource availability among others.

1.1 LeWI: Lend When Idle

The main load balancing algorithm used in DLB is called LeWI (Lend When Idle). The idea of the algorithm is to use the computational resources that are not being used for useful computation to speed up processes in the same computational node.

To achieve this DLB will lend the cpus of a process waiting in a blocking MPI call to another process running in the same node.



a) Unbalanced MPI application   b) Unbalanced MPI application balanced with LeWI

CHAPTER

TWO

TECHNICAL REQUIEREMENTS

2.1 Programming models

The currently supported programming models or runtimes by DLB are the following:

- MPI + OpenMP
- MPI + OmpSs
- MPI + SMPSs (not maintained)
- Nanos++ (Multiple Applications)

2.2 Shared Memory between processes

DLB can balance processes running on the same node and sharing memory. DLB is based on shared memory and needs shared memory between all the processes sharing resources.

2.3 Preload mechanism for MPI applications

When using MPI applications we need the preload mechanism (in general available in any Linux system). This is only necessary when using the MPI interception, it is not necessary when calling DLB with the public API from the application or a runtime.

2.4 Parallel regions in OpenMP

In OpenMP applications we need parallelism to open and close (parallel region) to be able to change the number of threads. This is because the OpenMP programming model only allows to change the number of threads outside a parallel region.

We also need to add a call to the DLB API to update the resources used before each parallel. This limitation is not present when using the Nanos++ runtime.

2.5 Non-busy waiting for MPI calls

When using MPI applications we need the MPI blocking calls not to be busy waiting. However, DLB offers a mode where one CPU is reserved to wait for the MPI blocking call and is not lent to other processes.

If the MPI library does not offer a Non-busy waiting mode, or we do not want ot use it of any other reason, We can tell DLB to use a non blocking mode with the environment variable `LB_LEND_MODE=1CPU`.

CHAPTER

THREE

HOW TO INSTALL DLB

3.1 Build requirements

- A supported platform running Linux (i386, x86-64, ARM, PowerPC or IA64).
- GNU C/C++ compiler versions 4.4 or better.
- Python 2.4 or better

3.2 Installation steps

1. Get the latest DLB *tarball* from https://pm.bsc.es/dlb. Unpack the file and enter the new directory:

```
$ tar xzf dlb-x.y.tar.gz
$ cd dlb-x.y/
```

2. Configure it, with optionally some of the *DLB configure flags*:

```
$ ./configure --prefix=$DLB_PREFIX
```

3. Build and install:

```
$ make
$ make install
```

3.3 DLB configure flags

By default, the *autotools scripts* will build four versions of the library, the combination of the performance and debug versions with the instrumentation option. The basic library (performance, no-instrumentation) cannot be disabled but the other three can be freely disabled using the following flags.

--disable-debug Disable Debug library.

--disable-instrumentation Disable Instrumentation library.

--disable-instrumentation-debug Disable Instrumentation-Debug library.

DLB library has two optional dependencies. MPI allows DLB to automatically detect some patterns about the load balance of the application. When MPI support is detected, another set of libraries `libdlb_mpi*` and `libdlb_mpif*` are built; refer to *Preload mechanism for MPI applications* for more details. The other optional dependency is HWLOC that allows DLB library to get some knowledge about the hardware details of the compute node. If HWLOC is not found, hardware detection will fall back to some OS utilities.

--with-mpi=<mpi_prefix> Specify where to find the MPI libraries and include files.

--with-hwloc=<hwloc_prefix> Specify where to find the HWLOC libraries.

Some of the load balancing policies rely on the number of CPUs in a compute node. In those cases where this number cannot be determined at run-time or cases where the number of CPUs differs between run-time and compile-time, the user may overwrite the value using the following flag.

--with-cpus-per-node=<N> Overwrite value of CPUs per node detected at configure time.

CHAPTER

FOUR

HOW TO RUN WITH DLB

DLB library is originally designed to be run on applications using a Shared Memory Programming Model (OpenMP or OmpSs), although it is not a hard requirement, it is very advisable in order to exploit the thread management of the underlying Programming Model runtime.

4.1 Examples by Programming Model

4.1.1 OmpSs

If you are running an OmpSs application you just need to set two parameters. First, append `--thread-manager=dlb` to the Nanos++ `NX_ARGS` environment variable. With this option, the Nanos++ runtime relies on DLB to take every decision about thread management. Second, set the DLB variable `LB_POLICY="auto_LeWI_mask"`, which is the LeWI policy for autonomous threads. This DLB policy needs a highly malleable runtime, as Nanos++ is.

You don't need to do any extra steps if your application doesn't use the DLB API, as all the communication with DLB is handled by the Nanos++ runtime. Otherwise, just add the flag `--dlb` to the Mercurium compiler to automatically add the required compile and link flags:

```
$ smpfc --ompss [--dlb] foo.c -o foo
$ export NX_ARGS+=" --thread-manager=dlb"
$ export LB_POLICY="auto_LeWI_mask"
$ ./foo
```

4.1.2 MPI + OmpSs

In the same way, you can run MPI + OmpSs applications with DLB:

```
$ OMPI_CC="smpfc --ompss [--dlb]" mpicc foo.c -o foo
$ export NX_ARGS="--thread-manager=dlb"
$ export LB_POLICY="auto_LeWI_mask"
$ mpirun -n 2 ./foo
```

However, MPI applications with only one running thread may become blocked due to the main thread entering a synchronization MPI blocking point. DLB library can intercept MPI calls to overload the CPU in these cases:

```
$ export LB_LEND_MODE="BLOCK"
$ mpirun -n 2 -x LD_PRELOAD=${DLB_PREFIX}/lib/libdlb_mpi.so ./foo
```

4.1.3 OpenMP

OpenMP is not as malleable as OmpSs so the ideal DLB policy is the simple `LeWI`, without individual thread management nor CPU mask support. Also, unless you are using Nanos++ as an OpenMP runtime, you will need to manually call the API functions in your code, thus you will need to pass the compile and linker flags to your compiler:

```
$ gcc -fopenmp foo.c -o foo -I${DLB_PREFIX}/include \
        -L${DLB_PREFIX}/lib -ldlb -Wl,-rpath,${DLB_PREFIX}/lib
$ export LB_POLICY="LeWI"
$ ./foo
```

4.1.4 MPI + OpenMP

In the case of MPI + OpenMP applications, you can let all the DLB management to the MPI interception. Thus, allowing you to run DLB applications without modifying your binary. Simply, preload an MPI version of the library and DLB will balance the resource of each process during the MPI blocking calls:

```
$ mpicc -fopenmp foo.c -o foo
$ export LB_POLICY="LeWI"
$ mpirun -n 2 -x LD_PRELOAD=${DLB_PREFIX}/lib/libdlb_mpi.so ./foo
```

CHAPTER

FIVE

PUBLIC API

The DLB API can be divided into:

Basic set The basic set is very simple and reduced and oriented to application developers. The different functions will be explained in detail in section *Basic set of DLB API*.

Advanced set The advanced set is oriented to programming model runtimes but can be used by applications also. The advanced functions will be explained in detail in section *Advanced set of DLB API*.

MPI API This is a specific API for MPI. We offer an MPI interface that will be called by Extrae if we are tracing the application or internally in the MPI intercept API. All the calls of this API are of the form shown below, and thus not documented.

- DLB_<mpi_call_name>_enter(...)
- DLB_<mpi_call_name>_leave(...)

5.1 Basic set of DLB API

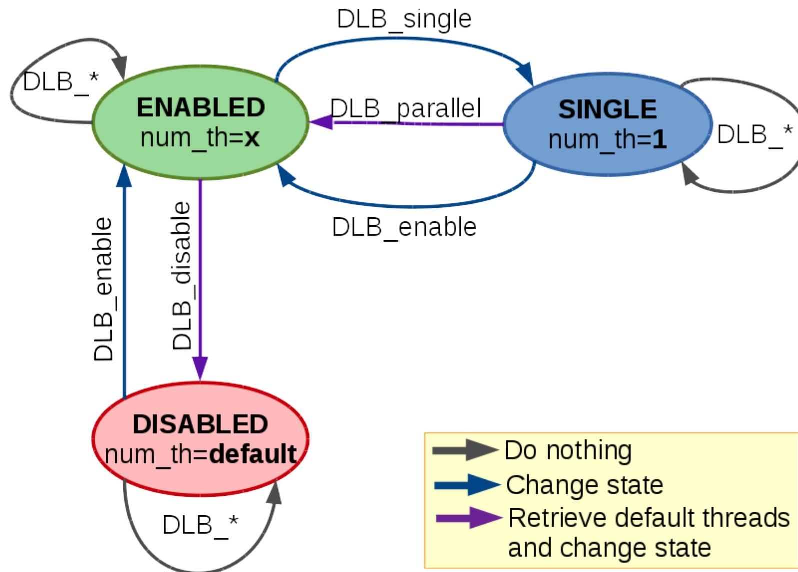
This API is intended to give hints from the application to DLB. With this hints DLB is able to make a more efficient use of resources.

`void DLB_disable(void)`
Will disable any DLB action. And reset the resources of the process to its default. While DLB is disabled there will not be any movement of threads for this process. Useful to limit parts of the code were DLB will not be helpful, by disabling DLB we avoid introducing any overhead.

`void DLB_enable(void)`
Will enable DLB. DLB is enabled by default when the execution starts. And if it was not previously disable it will not have any effect. Useful to finish parts of the code were we disabled DLB temporally.

`void DLB_single(void)`
Will lend all the threads of the process except one. Useful to mark parts of the code that are serial. The remaining threads can be used by some other process. All the DLB functions will be disabled except lending the thread when entering an MPI call, exiting and MPI call, DLB_parallel and DLB_enable.

`void DLB_parallel(void)`
Will claim the default threads and enable all the DLB functions. Useful when exiting a serial section of code.



We can summarize the behavior of these functions with the states graph shown in the figure. We can consider three states for DLB (for each process) *Enabled*, *Disabled* and *Single*.

- *Enabled* would be the default state, where DLB will react to any API call.
- The *Disabled* state will not allow any change in the number of threads (only a call to `DLB_enable` will have effect). The number of threads of the process in *Disabled* state will be the default.
- The *Single* state will only react at `DLB_enable` or `DLB_parallel` API calls. The number of threads of the process in the *Single* state will be 1.

5.2 Advanced set of DLB API

The advanced set of calls is designed to be used by runtimes, either in the outer level or the inner level of parallelism. But advanced users can also use them from applications.

void DLB_Init (void)
Initialize the DLB library and all its internal data structures. Must be called once and only one by each process in the DLB system.

void DLB_Finalize (void)
Finalize the DLB library and clean up all its data structures. Must be called by each process before exiting the system.

void DLB_reset (void)
Reset the number of threads of this process to its default.

void DLB_UpdateResources (void)
Check the state of the system to update your resources. You can obtain more resources in case there are available CPUs.

void DLB_UpdateResources_max (int max_resources)
Check the state of the system to update your resources. You can obtain more resources in case there are available CPUs. The maximum number of resources that you can get is `max_resources`.

void DLB_ReturnClaimedCpus (void)
Check if any of the resources you are using have been claimed by its owner and return it if necessary.

void DLB_Lend(void)
Lend all your resources to the system. Except in case you are using the *1CPU* block mode you will lend all the resources except one CPU.

void DLB_Retrieve(void)
Retrieve all your default resources previously lent.

int DLB_ReleaseCpu(int cpu)
Lend this CPU to the system. The return value is 1 if the operation was successful and 0 otherwise.

int DLB_ReturnClaimedCpu(int cpu)
Return this CPU to the system in case it was claimed by its owner. The return value is 1 if the CPU was returned to its owner and 0 otherwise.

void DLB_ClaimCpus(int cpus)
Claim as many CPUs as the parameter `cpus` indicates. You can only claim your CPUs. Therefore if you are claiming more CPUs than the ones that you have lent, you will only obtain as many CPUs as you have lent.

void DLB_AcquireCpu(int cpu)
Notify the system that you are going to use this CPU. The system will try to adjust himself to this requirement, This function may leave the system in an unstable state. Avoid using it.

void DLB_AcquireCpus(dlb_cpu_set_t mask)
Same as `DLB_AcquireCpu`, but with a set of CPUs.

int DLB_CheckCpuAvailability(int cpu)
This function returns 1 if your CPU is available to be used, 0 otherwise. Only available for policies with autonomous threads.

int DLB_Is_auto(void)
Return 1 if the policy allows autonomous threads 0 otherwise.

void DLB_Update(void)
Update the status of 'Statistics' and 'DROM' modules, like updating the process statistics or check if some other process has signaled a new process mask.

void DLB_NotifyProcessMaskChange(void)
Notify DLB that the process affinity mask has been changed. DLB will then query the runtime to obtain the current mask.

void DLB_NotifyProcessMaskChangeTo(const dlb_cpu_set_t mask)
Notify DLB that the process affinity mask has been changed.

void DLB_PrintShmem(void)
Print the data stored in the Shared Memory

int DLB_SetVariable(const char *variable, const char *value)
Change the value of a DLB internal variable

int DLB_GetVariable(const char *variable, char *value);
Get DLB internal variable

void DLB_PrintVariables(void);
Print DLB internal variables

CHAPTER SIX

FAQ: FREQUENTLY ASKED QUESTIONS

- *Does my application need to meet any requirements to run with DLB?*
- *Mercurium may have DLB support with the `--dlb` flag. What does it do? Should I use it?*
- *Which policy should I use for DLB?*
- *DLB fails registering a mask. What does it mean?*
- *I'm running a hybrid MPI + OpenMP application but DLB doesn't seem to have any impact*
- *I'm running a hybrid application, 1 thread per process, and DLB still does nothing*
- *I'm running a well allocated hybrid MPI + OpenMP but DLB still doesn't do anything.*
- *Can I see DLB events in a Paraver trace?*
- *Can I disable DLB tracing in an OmpSs execution?*

6.1 Does my application need to meet any requirements to run with DLB?

Can the number of threads of your application be modified at any time? It is not rare to manually allocate some private storage for each thread in OpenMP applications and then each thread in a parallel access this provate storage though the thread id. This kind of methodology can break the execution if the number os threads is modified and the local storage is not adjusted, or at least considered.

6.2 Mercurium may have DLB support with the `--dlb` flag. What does it do? Should I use it?

If Mercurium was configured with DLB support, it will accept the `--dlb` option flag to automatically include the DLB headers and link with the correspondig library. If your application does not use any DLB API function you don't need to use this flag.

6.3 Which policy should I use for DLB?

In some cases each policy should be considered for study, but as a general rule, use `LB_POLICY=LeWI` in OpenMP applications and `LB_POLICY=auto_LeWI_mask` in OmpSs applications.

6.4 DLB fails registering a mask. What does it mean?

When executing your application with DLB you may encounter the following error:

```
DLB PANIC[hostname:pid:tid]: Error trying to register CPU mask: [ 0 1 2 3 ]
```

A process registering into DLB will register its CPU affinity mask as owned CPUs. DLB can move the ownership of registered CPUs once the execution starts but it will fail with a panic error if a new process tries to register a CPU already owned by other process.

This typically ocurrs if you run two applications without specifying the process mask, or in case of MPI, if the `mpiexec` is not executed with the right option flags. If the former case you would need to run the applications using the `taskset` command), if the latter every MPI implementation has different options so you will need to check the appropriate documentation.

6.5 I'm running a hybrid MPI + OpenMP application but DLB doesn't seem to have any impact

Did you place your process and threads in a way they can help each other? DLB aware applications need to be placed or distributed in a way such that another process in the same node can benefit from the serial parts of the application.

For instance, in a cluster of 4 CPUs per node you may submit a hybrid job of X MPI process and 4 OpenMP threads per process. That means that each node would only contain one process, so there will never be resource sharing within the node. Now, if you submit another configuration with either 2 or 1 threads per process, each node will contain 2 or 4 DLB process that will share resources when needed.

6.6 I'm running a hybrid application, 1 thread per process, and DLB still does nothing

By default DLB can reduce the number of threads of a process up to a minumum of 1. If the application is not MPI that's enough because we assume that at least there is always serial code to do.

But, in MPI applications, we may reach a point where the serial code is only an `MPI_Barrier` where the process is only wasting CPU cycles waiting for other processes to synchronize. If you configure DLB to intercept MPI calls, this CPU can be used instead for helping other processess in the same node.

To use this future you need to preload the DLB MPI library and to set this environment variable:

```
export LB_MODE=BLOCKING
```

6.7 I'm running a well allocated hybrid MPI + OpenMP but DLB still doesn't do anything.

There could be several reasons as to why DLB could not help to improve the performance of an application.

Do you have enough parallel regions to enable the malleability of the number of threads at different points in your applications? Try to split you parallel region into smaller parallels.

Is your application very memory bandwith limited? Sometimes increasing the number of threads in some regions does not increase the performance if the parallel region is already limited by the memory bandwidth.

Could it be that your application does not suffer from load imbalance? Try our performance tools to check it out. (http://tools.bsc.es)

6.8 Can I see DLB events in a Paraver trace?

Yes, the effects of DLB are visible in any trace as it involves thread blocking and resuming but DLB can also emit some events. To do so it will depend on the programming model you want to trace.

For OmpSs programs: simply compile using the flag `--instrument` as you would usually do.

For OpenMP programs: You can link your application with the library `libdlb_instr.so` instead.

For MPI programs: Use the DLB version of any Extrae library (ending in -lb.so). If you don't find the library in the Extrae installation path, reconfigure it using the option `--with-load-balance=${DLB_PREFIX}`.

6.9 Can I disable DLB tracing in an OmpSs execution?

Yes, simply export the evironment variable `LB_TRACE_ENABLED=0`.

CHAPTER

SEVEN

CONTENT OF THE PACKAGE

7.1 Structure

```
dlb/
|-- bin
|-- include
|-- lib
`-- share
    |-- doc
    |   `-- dlb
    |       `-- examples
    |           |-- MPI+OMP
    |           |-- MPI+OmpSs
    |           `-- statistics
    `-- paraver_cfgs
        `-- DLB
```

7.2 Binaries

dlb Basic info, help and version

dlb_shm Utility to manage shared memory

dlb_taskset Utility to change the process mask of DLB processes

dlb_cpu_usage Python viewer if using stats

7.3 Libraries

DLB installs different versions of the library for different situations, but in general you should only focus on these ones:

libdlb.so Link against this library only if your application needs to call some DLB API function.

libdlb_mpi.so Link in the correct order or just preload it using `LD_PRELOAD` environment variable if you want DLB to intercetp MPI calls.

libdlb_mpif.so Same case as above, but to intercep MPI Fortran calls.

Remember that if the programming model already supports DLB (as in Nanos++), you don't need to link against any library.

7.4 Examples

Currently three examples are distributed with the source code and installed in the `${DLB_PREFIX}/share/doc/dlb/examples/` directory. Each example consists of a `README` file with a brief description and the steps to follow, a C source code file, a `Makefile` to compile the source code and a script `run.sh` to run the example.

Some Makefile variables have been filled at configure time. They should be enough to compile and link the examples with MPI support. Some Makefiles assume that Mercurium is configured in the `PATH`.

Note: In order to enable tracing you need an Extrae installation and to correctly set the `EXTRAE_HOME` environment variable.

Note: Some examples preload the DLB Extrae library (ending in trace-lb.so). If you don't have those libraries installed reconfigure your Extrae installation with the option `--with-load-balancing=${DLB_PREFIX}`

7.4.1 MPI + OpenMP

PILS is a synthetic MPI program with some predefined load balancing issues. Simply check the `Makefile` if everything is correct and run `make`. The `run.sh` script should also contain the MPI detected at configure time. Apart from that, two options can be modified at the top of the file, whether you want to enable *DLB* or to enable *TRACE* mode (or both).

A very similar example but just using OpenMP. Notable differences are the `-fopenmp` flag used in the `Makefile` that assumes a GNU-like flag. The `run.sh` script is also configured to allow two options, *DLB* and *TRACE*.

7.4.2 MPI + OmpSs

A very similar example but just using OmpSs. Make sure that Mercurium is in your `PATH` or modify the Makefile accordingly. Then, you can run it in the same way as the previous example.

7.4.3 Statistics

The last example consists of a PILS program designed to run for a long time, without DLB micro-load balancing, but with the Statistics module enabled. Check the `run.sh` script. The objective is to let the process run in background while you run one of the other two binaries provided. These two binaries `get_pid_list` and `get_cpu_usage` perform basic queries to the first PILS program and obtain some statistics about CPU usage.

INDEX

A

Advanced set, **9**

B

Basic set, **9**

M

MPI API, **9**