DLB User Guide

Release 1.1

Barcelona Supercomputing Center

December 23, 2016

CHAPTER

ONE

INTRODUCTION TO DLB

1.1 First presentation

blablablabla blablablabla

1.2 Second presentation

blablablabla blablablabla

1.2.1 MPI interception

1.2.2 Statistics

1.2.3 Dynamic Resource Ownership Manager

CHAPTER

TWO

HOW TO INSTALL DLB

2.1 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

2.2 Installation steps

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

```
$ tar xzf dlb-x.y.tar.z
$ 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
```

2.3 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 *MPI interception* 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. If the MPI installation does not follow the standard installation tree like `prefix/include` and `prefix/lib`, you can also specify each of the paths through the options `--with-mpi-headers` and `--with-mpi-libs`.

--with-hwloc=<hwloc_prefix> Specify where to find the HWLOC libraries. As in the previous option, you can use `--with-hwloc-headers` and `--with-hwloc-libs`.

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

THREE

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.

3.1 Environment Variables

DLB library can be completely configured at run-time using Environment Variables.

Module configuration Env. Variables

- LB_POLICY = [policy]
- LB_STATISTICS = [0, 1]
- LB_DROM = [0, 1]

MPI Env. Variables

- LB_JUST_BARRIER = [0, 1]
- LB_LEND_MODE = [BLOCK, 1CPU]

Verbose Env. Variables

- LB_VERBOSE = [api|microlb|shmem|mpi_api|mpi_intercept|stats|drom]
- LB_VERBOSE_FORMAT = [node|pid|mpinode|mpirank|thread]

Tracing Env. Variables

- LB_TRACE_ENABLED = [0, 1]
- LB_TRACE_COUNTERS = [0, 1]

Misc Env. Variables

- LB_MASK => (CPU range) DLB Mask for LeWI_mask policies
- LB_GREEDY => (bool) Greedy option for LeWI
- LB_SHM_KEY = [key]

FIXME / Currently disabled

- LB_BIND => (bool) Bind option for LeWI, currently disabled
- LB_THREAD_DISTRIBUTION
- LB_AGGRESSIVE_INIT => (bool) Currently aggressive init is managed by the runtime

- LB_PRIORIZE_LOCALITY => (bool) prioritize (fixme variable typo) locality when acquiring CPUs, currently only implemented in the shmem_bitset, which is not used anymore

3.2 DLB 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

3.3 Examples by Programming Model

3.3.1 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
```

3.3.2 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
```

3.3.3 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
```

3.3.4 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
```

3.4 Running with the script

TBD. Ticket #33

CHAPTER

FOUR

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*.

Statistics This set of functions allows the user to obtain some statistics about CPU usage. For a more detailed description see *Statistics*. These functions are described in section *Statistics Interface*.

Dynamic Resource Ownership Manager With these functions, the user can manage from an external process the CPU ownership of each DLB running process. For a more detailed description see *Dynamic Resource Ownership Manager*. These functions are described in section *Dynamic Resource Manager Interface*.

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. There is one API function that is aimed to be called by the user, explained in section *MPI Interface*.

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

4.1 Basic set of DLB API

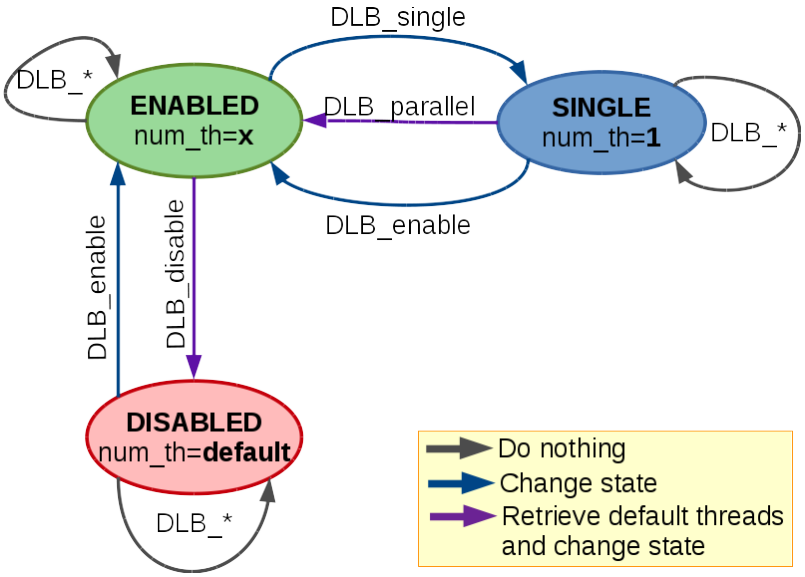
This API is intended to give hints from the application to DLB. With this hints DLB is able to make a more efficient use of resources.

`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.

4.2 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

4.3 Statistics Interface

The next set of functions can be used only when the user has enabled the Statistics Module (see *Statistics*). With this interface the user can obtain different statistics about the CPU usage and their ownership.

void DLB_Stats_Init(void)
Initialize DLB Statistics Module

void DLB_Stats_Finalize(void)
Finalize DLB Statistics Module

int DLB_Stats_GetNumCpus(void)
Get the total number of available CPUs in the node

void DLB_Stats_GetPidList(int *pidlist,int *nelems,int max_len)
Get the PID's attached to this module

double DLB_Stats_GetCpuUsage(int pid)
Get the CPU Usage of the given PID

double DLB_Stats_GetCpuAvgUsage(int pid)
Get the CPU Average Usage of the given PID

void DLB_Stats_GetCpuUsageList(double *usagelist,int *nelems,int max_len)
Get the CPU usage of all the attached PIDs

void DLB_Stats_GetCpuAvgUsageList(double *avgusagelist,int *nelems,int max_len)
Get the CPU Average usage of all the attached PIDs

double DLB_Stats_GetNodeUsage(void)
Get the CPU Usage of all the DLB processes in the node

double DLB_Stats_GetNodeAvgUsage(void)
Get the number of CPUs assigned to a given process

int DLB_Stats_GetActiveCpus(int pid)
Get the number of CPUs assigned to a given process

void DLB_Stats_GetActiveCpusList(int *cpuslist,int *nelems,int max_len)
Get the number of CPUs assigned to each process

int DLB_Stats_GetLoadAvg(int pid, double *load)
Get the Load Average of a given process

float DLB_Stats_GetCpuStateIdle(int cpu)
Get the percentage of time that the CPU has been in state IDLE

float DLB_Stats_GetCpuStateOwned(int cpu)
Get the percentage of time that the CPU has been in state OWNED

float DLB_Stats_GetCpuStateGuested(int cpu)
Get the percentage of time that the CPU has been in state GUESTED

void DLB_Stats_PrintShmem(void)
Print the data stored in the Stats Shared Memory

4.4 Dynamic Resource Manager Interface

The next set of functions can be used when the user has enabled the Dynamic Resource Ownership Manager (DROM) Module (see *Dynamic Resource Ownership Manager*). With this interface the user can set or retrieve the process mask of each DLB process.

void DLB_Drom_Init(void)
Initialize DROM Module

void DLB_Drom_Finalize(void)
Finalize DROM Module

int DLB_Drom_GetNumCpus(void)
Get the total number of available CPUs in the node

void DLB_Drom_GetPidList(int *pidlist, int *nelems, int max_len)
Get the PID's attached to this module

int DLB_Drom_GetProcessMask(int pid, dlb_cpu_set_t mask)
Get the process mask of the given PID

int DLB_Drom_SetProcessMask(int pid, const dlb_cpu_set_t mask)
Set the process mask of the given PID

void DLB_Drom_PrintShmem(void)
Print the data stored in the Drom Shared Memory

4.5 MPI Interface

Unlike all the other MPI functions aimed to be called by Extrae, this one is specifically aimed to be used by the user. It is useful sometimes to block only a single node to synchronize the workload at a certain point while using the CPUs owned by the process to help other processes to reach this point.

void DLB_MPI_node_barrier(void)
Blocks until all processes in the same node have reached this routine.

CHAPTER

FIVE

EXAMPLES

Currently three examples are distributed with the source code, which are 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. The only prerequisite is to check the Makefile dependencies, like an MPI wrapper or the Mercurium compiler for the examples in OmpSs.

5.1 MPI + OmpSs

PILS is a synthetic MPI program with some predefined load balancing issues. Simply check the `Makefile` and modify it accordingly if you are using other MPI implementation than OpenMPI. Use the `I_MPI_CC` environment variable instead, or check your MPI documentation. the `run-sh` script is also written for an OpenMPI mpirun command. 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).

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

5.2 MPI + OpenMP

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*.

Note: The Extrae library used in this example when both options *DLB* and *TRACE* are enabled is a modified version which calls DLB hooks when an MPI call is captured. If you don't have this version of the library (ending in trace-lb.so), reconfigure your Extrae installation with the option `--with-load-balancing=${DLB_PREFIX}`

5.3 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.