Using Platform LSF™ HPC Features

Version 8
Release date: January 2011
Last modified: January 10, 2011
Support: support@platform.com
Comments to: doc@platform.com

Platform™

We'd like to hear from you You can help us make this document better by telling us what you think of the content, organization, and usefulness of the information. If you find an error, or just want to make a suggestion for improving this document, please address your comments to doc@platform.com.

Your comments should pertain only to Platform documentation. For product support, contact support@platform.com.

Document redistribution and translation

Trademarks LSF is a registered trademark of Platform Computing Inc. in the United States and in other jurisdictions.

PLATFORM COMPUTING, PLATFORM SYMPHONY, PLATFORM JOBSCHEDULER, PLATFORM ENTERPRISE GRID ORCHESTRATOR, PLATFORM EGO, PLATFORM VM ORCHESTRATOR, PLATFORM VMO, ACCELERATING INTELLIGENCE, and the PLATFORM and PLATFORM LSF logos are trademarks of Platform Computing Inc. in the United States and in other jurisdictions.

UNIX is a registered trademark of The Open Group in the United States and in other jurisdictions.

Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.

Microsoft is either a registered trademark or a trademark of Microsoft Corporation in the United States and/or other countries.

Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

Macrovision, Globetrotter, and FLEX*lm* are registered trademarks or trademarks of Macrovision Corporation in the United States of America and/or other countries.

Topspin is a registered trademark of Topspin Communications, Inc.

Intel, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Other products or services mentioned in this document are identified by the trademarks or service marks of their respective owners.

Third Party License Agreements

http://www.platform.com/legal-notices/third-party-license-agreements

Contents

CHAPTER

1

About Platform LSF HPC Features

Contents

What Are Platform LSF HPC Features?

Platform LSF™ HPC features maximize the performance of High Performance Computing (HPC) clusters.

Platform LSF is the industry standard workload management software product, it provides load sharing in a distributed system and batch scheduling for compute-intensive jobs. The HPC features provide support for:

- Dynamic resource discovery and allocation (resource reservation) for parallel batch job execution
- Full job-level control of the distributed processes to ensure no processes will become un-managed. This effectively reduces the possibility of one parallel job causing severe disruption to an organization's computer service
- The standard MPI interface
- Heterogeneous resource-based batch job scheduling including job-level resource usage enforcement

Advanced HPC scheduling policies

Platform LSF HPC features enhance the job management capability of your cluster through advanced scheduling policies such as:

- Policy-based job preemption
- Advance reservation
- Memory and processor reservation
- Memory and processor backfill
- Cluster-wide resource allocation limits
- User and project-based fairshare scheduling
- Topology-aware scheduling

LSF daemons Run on every node to collect resource information such as processor load, memory availability, interconnect states, and other host-specific as well as cluster-wide resources. These agents coordinate to create a single system image of the cluster.

HPC workload scheduler Supports advanced HPC scheduling policies that match user demand with resource supply.

Job-level runtime resource management Control sequential and parallel jobs (terminate, suspend, resume, send signals) running on the same host and across hosts. Configure and monitor job-level and system-wide CPU, memory, swap, and other runtime resource usage limits.

Application integration support

Packaged application integrations and tailored HPC configurations make Platform LSF ideal for Industrial Manufacturing, Life Sciences, Government and Research sites using large-scale modeling and simulation parallel applications involving large amounts of data. Platform LSF helps Computer-Aided Engineering (CAE) users reduce the cost of manufacturing, and increase engineer productivity and the quality of results.

Platform LSF is integrated to work out of the box with many HPC applications, such as LSTC LS-Dyna, FLUENT, ANSYS, MSC Nastran, Gaussian, Lion Bioscience SRS, and NCBI BLAST.

Parallel application support

Platform LSF supports jobs using the following parallel job launchers:

POE The IBM Parallel Operating Environment (POE) interfaces with the Resource Manager to allow users to run parallel jobs requiring dedicated access to the high performance switch.

The LSF integration for IBM High-Performance Switch (HPS) systems provides support for submitting POE jobs from AIX hosts to run on IBM HPS hosts.

OpenMP Platform LSF provides the ability to start parallel jobs that use OpenMP to communicate between process on shared-memory machines and MPI to communicate across networked and non-shared memory machines.

PVM Parallel Virtual Machine (PVM) is a parallel programming system distributed by Oak Ridge National Laboratory. PVM programs are controlled by the PVM hosts file, which contains host names and other information.

MPI The Message Passing Interface (MPI) is a portable library that supports parallel programming. LSF supports several MPI implementations, includding MPICH, a joint implementation of MPI by Argonne National Laboratory and Mississippi State University. LSF also supports MPICH-P4, MPICH-GM, LAM/MPI, Intel® MPI, IBM Message Passing Library (MPL) communication protocols, as well as SGI and HP-UX vendor MPI integrations.

blaunch distributed application framework

Most MPI implementations and many distributed applications use `rsh` and `ssh` as their task launching mechanism. The `blaunch` command provides a drop-in replacement for `rsh` and `ssh` as a transparent method for launching parallel and distributed applications within LSF.

Similar to the LSF `lsrun` command, `blaunch` transparently connects directly to the RES/SBD on the remote host, and subsequently creates and tracks the remote tasks, and provides the connection back to LSF. There no need to insert `pam`, taskstarter into the `rsh` or `ssh` calling sequence, or configure any wrapper scripts.

`blaunch` supports the following core command line options as `rsh` and `ssh`:

- `rsh` *`host_name command`*
- `ssh [`*`user_name`*`@]`*`host_name command`*

All other `rsh` and `ssh` options are silently ignored.

Important: You cannot run blaunch directly from the LSF command line.

`blaunch` only works within an LSF job; it can only be used to launch tasks on remote hosts that are part of a job allocation. It cannot be used as a standalone command. On success `blaunch` exits with 0.

Windows `blaunch` is supported on Windows 2000 or later with the following exceptions:

- Only the following signals are supported: SIGKILL, SIGSTOP, SIGCONT.
- The `-n` option is not supported.
- CMD.exe /C <user command line> is used as an intermediate command shell when `-no-shell` is not specified

- CMD.exe /C is not used when `-no-shell` is specified.

See"blaunch Distributed Application Framework" on page 14 for more information.

PAM

The Parallel Application Manager (PAM) is the point of control for LSF HPC features. PAM interfaces the user application with LSF. For all parallel application processes (tasks), PAM:

- Monitors and forwards control signals to parallel tasks
- Monitors resource usage while the user application is running
- Passes job-level resource limits to `sbatchd` for enforcement
- Collects resource usage information and exit status upon termination

See the *Platform LSF Command Reference* for more information about PAM.

Resizable jobs

Jobs running in HPC system integrations (psets, cpusets, etc.) cannot be resized.

Resource requirements

Jobs running in HPC system integrations (psets, cpusets, etc.) cannot have compound resource requirements.

Jobs running in HPC system integrations (psets, cpusets, etc.) cannot have resource requirements with compute unit strings (`cu`[...]).

When compound resource requirements are used at any level, an `esub` can create job-level resource requirements which overwrite most application-level and queue-level resource requirements. `-R` merge rules are explained in detail in *Administering Platform LSF*.

HPC Components

HPC components take full advantage of the resources of LSF for resource selection and batch job process invocation and control.

User requests	Batch job submission to LSF using the `bsub` command.
mbatchd	Master Batch Daemon (MBD) is the policy center for LSF. It maintains information about batch jobs, hosts, users, and queues. All of this information is used in scheduling batch jobs to hosts.
LIM	Load Information Manager is a daemon process running on each execution host. LIM monitors the load on its host and exchanges this information with the master LIM. For batch submission the master LIM provides this information to `mbatchd`. The master LIM resides on one execution host and collects information from the LIMs on all other hosts in the cluster. If the master LIM becomes unavailable, another host will automatically take over.
mpirun.lsf	Reads the environment variable LSF_PJL_TYPE, and generates the appropriate command line to invoke the PJL. The `esub` programs provided in LSF_SERVERDIR set this variable to the proper type.
sbatchd	Slave Batch Daemons (SBDs) are batch job execution agents residing on the execution hosts. `sbatchd` receives jobs from `mbatchd` in the form of a job specification and starts RES to run the job according the specification. `sbatchd` reports the batch job status to `mbatchd` whenever job state changes.
blaunch	The `blaunch` command provides a drop-in replacement for `rsh` and `ssh` as a transparent method for launching parallel and distributed applications within LSF.
PAM	Parallel Application Manager is the point of control for LSF HPC features. PAM interfaces the user application with the LSF system.
RES	Remote Execution Servers reside on each execution host. RES manages all remote tasks and forwards signals, standard I/O, resources consumption data, and parallel job information between PAM and the tasks.
PJL	Parallel Job Launcher is any executable script or binary capable of starting parallel tasks on all hosts assigned for a parallel job (for example, `mpirun`, `poe`, `prun`.)
TS	`TaskStarter` is an executable responsible for starting a task on the local host and reporting the process ID and host name to the PAM. `TaskStarter` is located in LSF_BINDIR.
Application task	The individual process of a parallel application
First execution host	The host name at the top of the execution host list as determined by LSF. Starts PAM.
Execution hosts	The most suitable hosts to execute the batch job as determined by LSF
esub.*pjl_type*	LSF provides a generic `esub` to handle job submission requirements of your HPC applications. Use the `-a` option of `bsub` to specify the application you are running. For example, to submit a job to LAM/MPI: `bsub -a lammpi bsub_options mpirun.lsf myjob`

The method name `lammpi`, uses the `esub` for LAM/MPI jobs (`LSF_SERVERDIR/esub.lammpi`), which sets the environment variable LSF_PJL_TYPE=lammpi. The job launcher, `mpirun.lsf` reads the environment variable LSF_PJL_TYPE=lammpi, and generates the appropriate command line to invoke LAM/MPI as the PJL to start the job.

CHAPTER

2

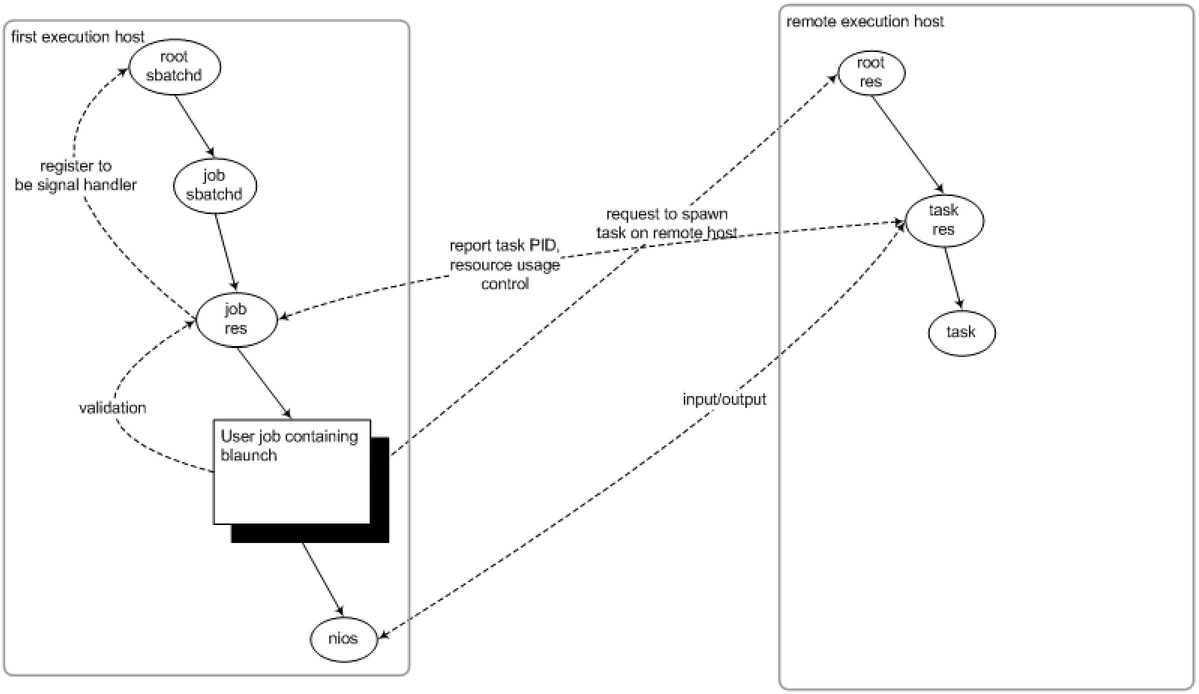
Running Parallel Jobs

Contents

blaunch Distributed Application Framework

Most MPI implementations and many distributed applications use `rsh` and `ssh` as their task launching mechanism. The `blaunch` command provides a drop-in replacement for `rsh` and `ssh` as a transparent method for launching parallel and distributed applications within LSF.

The following figure illustrates `blaunch` processing:


first execution host
root sbatchd
register to be signal handler
job sbatchd
job res
validation
User job containing blaunch
nios
report task PID, resource usage control
request to spawn task on remote host
input/output
remote execution host
root res
task res
task

About the blaunch command

Similar to the LSF `lsrun` command, `blaunch` transparently connects directly to the RES/SBD on the remote host, and subsequently creates and tracks the remote tasks, and provides the connection back to LSF. There no need to insert `pam`, taskstarter into the `rsh` or `ssh` calling sequence, or configure any wrapper scripts.

`blaunch` supports the following core command line options as `rsh` and `ssh`:

- `rsh host_name command`
- `ssh host_name command`

Whereas the host name value for `rsh` and `ssh` can only be a single host name, you can use the `-z` option to specify a space-delimited list of hosts where tasks are started in parallel. All other `rsh` and `ssh` options are silently ignored.

Important: You cannot run blaunch directly from the command line as a standalone command.

`blaunch` only works within an LSF job; it can only be used to launch tasks on remote hosts that are part of a job allocation. On success, `blaunch` exits with 0.

Windows: `blaunch` is supported on Windows 2000 or later with the following exceptions:

- Only the following signals are supported: SIGKILL, SIGSTOP, SIGCONT.

- The -n option is not supported.
- CMD.EXE /C <user command line> is used as intermediate command shell when:
  - -no-shell is not specified
- CMD.EXE /C is not used when -no-shell is specified.
- Windows Vista User Account Control must be configured correctly to run jobs.

See the *Platform LSF Command Reference* for more information about the blaunch command.

LSF APIs for the blaunch distributed application framework

LSF provides the following APIs for programming your own applications to use the blaunch distributed application framework:

- `lsb_launch()`—a synchronous API call to allow source level integration with vendor MPI implementations. This API will launch the specified command (`argv`) on the remote nodes in parallel. LSF must be installed before integrating your MPI implementation with `lsb_launch()`. The `lsb_launch()` API requires the full set of `liblsf.so`, `libbat.so` (or `liblsf.a`, `libbat.a`).
- `lsb_getalloc()`—allocates memory for a host list to be used for launching parallel tasks through `blaunch` and the `lsb_lanuch()` API. It is the responsibility of the caller to free the host list when it is no longer needed. On success, the host list will be a list of strings. Before freeing host list, the individual elements must be freed. An application using the `lsb_getalloc()` API is assumed to be part of an LSF job, and that LSB_MCPU_HOSTS is set in the environment.

See the *Platform LSF API Reference* for more information about these APIs.

The blaunch job environment

`blaunch` determines from the job environment what job it is running under, and what the allocation for the job is. These can be determined by examining the environment variables LSB_JOBID, LSB_JOBINDEX, and LSB_MCPU_HOSTS. If any of these variables do not exist, `blaunch` exits with a non-zero value. Similarly, if `blaunch` is used to start a task on a host not listed in LSB_MCPU_HOSTS, the command exits with a non-zero value.

The job submission script contains the `blaunch` command in place of `rsh` or `ssh`. The `blaunch` command does sanity checking of the environment to check for LSB_JOBID and LSB_MCPU_HOSTS. The `blaunch` command contacts the job RES to validate the information determined from the job environment. When the job RES receives the validation request from `blaunch`, it registers with the root `sbatchd` to handle signals for the job.

The job RES periodically requests resource usage for the remote tasks. This message also acts as a heartbeat for the job. If a resource usage request is not made within a certain period of time it is assumed the job is gone and that the remote tasks should be shut down. This timeout is configurable in an application profile in `lsb.applications`.

The `blaunch` command also honors the parameters LSB_CMD_LOG_MASK, LSB_DEBUG_CMD, and LSB_CMD_LOGDIR when defined in `lsf.conf` or as environment variables. The environment variables take precedence over the values in `lsf.conf`.

To ensure that no other users can run jobs on hosts allocated to tasks launched by `blaunch` set LSF_DISABLE_LSRUN=Y in `lsf.conf`. When LSF_DISABLE_LSRUN=Y is defined, RES refuses remote connections from `lsrun` and `lsgrun` unless the user is either an LSF administrator or root. LSF_ROOT_REX must be defined for remote execution by root. Other remote execution commands, such as `ch` and `lsmake` are not affected.

Temporary directory for tasks launched by blaunch

By default, LSF creates a temporary directory for a job only on the first execution host. If LSF_TMPDIR is set in `lsf.conf`, the path of the job temporary directory on the first execution host is set to `LSF_TMPDIR/`*`job_ID`*`.tmpdir`.

If LSB_SET_TMPDIR= Y, the environment variable TMPDIR will be set equal to the path specified by LSF_TMPDIR. This value for TMPDIR overrides any value that might be set in the submission environment.

Tasks launched through the `blaunch` distributed application framework make use of the LSF temporary directory specified by LSF_TMPDIR:

- When the environment variable TMPDIR is set on the first execution host, the `blaunch` framework propagates this environment variable to all execution hosts when launching remote tasks
- The job RES or the task RES creates the directory specified by TMPDIR if it does not already exist before starting the job
- The directory created by the job RES or task RES has permission 0700 and is owned by the execution user
- If the TMPDIR directory was created by the task RES, LSF deletes the temporary directory and its contents when the task is complete
- If the TMPDIR directory was created by the job RES, LSF will delete the temporary directory and its contents when the job is done
- If the TMPDIR directory is on a shared file system, it is assumed to be shared by all the hosts allocated to the `blaunch` job, so LSF does not remove TMPDIR directories created by the job RES or task RES

Automatic generation of the job host file

LSF automatically places the allocated hosts for a job into the $LSB_HOSTS and $LSB_MCPU_HOSTS environment variables. Since most MPI implementations and parallel applications expect to read the allocated hosts from a file, LSF creates a host file in the the default job output directory `$HOME/.lsbatch` on the execution host before the job runs, and deletes it after the job has finished running. The name of the host file created has the format:

`.lsb.<`*`jobID`*`>.hostfile`

The host file contains one host per line. For example, if `LSB_MCPU_HOSTS="hostA 2 hostB 2 hostC 1"`, the host file contains:

```
hostA
```

```
hostA
hostB
hostB
hostC
```

LSF publishes the full path to the host file by setting the environment variable LSB_DJOB_HOSTFILE.

Configuring application profiles for the blaunch framework

Handle remote task exit

You can configure an application profile in `lsb.applications` to control the behavior of a parallel or distributed application when a remote task exits. Specify a value for RTASK_GONE_ACTION in the application profile to define what the LSF does when a remote task exits.

The default behavior is:

When ...	LSF ...
Task exits with zero value	Does nothing
Task exits with non-zero value	Does nothing
Task crashes	Shuts down the entire job

RTASK_GONE_ACTION has the following syntax:

`RTASK_GONE_ACTION="``[`**`KILLJOB_TASKDONE`**` | `**`KILLJOB_TASKEXIT`**`]`
`[`**`IGNORE_TASKCRASH`**`]`**`"`**

Where:

- IGNORE_TASKCRASH

  A remote task crashes. LSF does nothing. The job continues to launch the next task.
- KILLJOB_TASKDONE

  A remote task exits with zero value. LSF terminates all tasks in the job.
- KILLJOB_TASKEXIT

  A remote task exits with non-zero value. LSF terminates all tasks in the job.

For example:

```
RTASK_GONE_ACTION="IGNORE_TASKCRASH KILLJOB_TASKEXIT"
```

RTASK_GONE_ACTION only applies to the `blaunch` distributed application framework.

When defined in an application profile, the LSB_DJOB_RTASK_GONE_ACTION variable is set when running `bsub -app` for the specified application.

You can also use the environment variable LSB_DJOB_RTASK_GONE_ACTION to override the value set in the application profile.

Handle communication failure

By default, LSF shuts down the entire job if connection is lost with the task RES, validation timeout, or heartbeat timeout. You can configure an application profile in `lsb.applications` so only the current tasks are shut down, not the entire job.

Use DJOB_COMMFAIL_ACTION="KILL_TASKS" to define the behavior of LSF when it detects a communication failure between itself and one or more tasks. If not defined, LSF terminates all tasks, and shuts down the job. If set to KILL_TASKS, LSF tries to kill all the current tasks of a parallel or distributed job associated with the communication failure.

DJOB_COMMFAIL_ACTION only applies to the `blaunch` distributed application framework.

When defined in an application profile, the LSB_DJOB_COMMFAIL_ACTION environment variable is set when running `bsub -app` for the specified application.

Set up job launching environment

LSF can run an appropriate script that is responsible for setup and cleanup of the job launching environment. You can specify the name of the appropriate script in an application profile in `lsb.applications`.

Use DJOB_ENV_SCRIPT to define the path to a script that sets the environment for the parallel or distributed job launcher. The script runs as the user, and is part of the job. DJOB_ENV_SCRIPT only applies to the `blaunch` distributed application framework.

If a full path is specified, LSF uses the path name for the execution. If a full path is not specified, LSF looks for it in LSF_BINDIR.

The specified script must support a `setup` argument and a `cleanup` argument. LSF invokes the script with the `setup` argument before launching the actual job to set up the environment, and with `cleanup` argument after the job is finished.

LSF assumes that if setup cannot be performed, the environment to run the job does not exist. If the script returns a non-zero value at setup, an error is printed to `stderr` of the job, and the job exits.

Regardless of the return value of the script at cleanup, the real job exit value is used. If the return value of the script is non-zero, an error message is printed to `stderr` of the job.

When defined in an application profile, the LSB_DJOB_ENV_SCRIPT variable is set when running `bsub -app` for the specified application.

For example, if `DJOB_ENV_SCRIPT=mpich.script`, LSF runs

```
$LSF_BINDIR/mpich.script setup
```

to set up the environment to run an MPICH job. After the job completes, LSF runs

```
$LSF_BINDIR/mpich.script cleanup
```

On cleanup, the `mpich.script` file could, for example, remove any temporary files and release resources used by the job. Changes to the LSB_DJOB_ENV_SCRIPT environment variable made by the script are visible to the job.

Update job heartbeat and resource usage

Use DJOB_HB_INTERVAL in an application profile in `lsb.applications` to configure an interval in seconds used to update the heartbeat between LSF and the tasks of a parallel or distributed job. DJOB_HB_INTERVAL only applies to the `blaunch` distributed application framework.

When DJOB_HB_INTERVAL is specified, the interval is scaled according to the number of tasks in the job:

```
max(DJOB_HB_INTERVAL, 10) + host_factor
```

where

host_factor = 0.01 * *number of hosts allocated for the job*

When defined in an application profile, the LSB_DJOB_HB_INTERVAL variable is set in the parallel or distributed job environment. You should not manually change the value of LSB_DJOB_HB_INTERVAL.

By default, the interval is equal to SBD_SLEEP_TIME in `lsb.params`, where the default value of SBD_SLEEP_TIME is 30 seconds.

Update job heartbeat and resource usage

Use DJOB_RU_INTERVAL in an application profile in `lsb.applications` to configure an interval in seconds used to update the resource usage for the tasks of a parallel or distributed job. DJOB_RU_INTERVAL only applies to the `blaunch` distributed application framework.

When DJOB_RU_INTERVAL is specified, the interval is scaled according to the number of tasks in the job:

`max(DJOB_RU_INTERVAL, 10) + host_factor`

where

host_factor = 0.01 * *number of hosts allocated for the job*

When defined in an application profile, the LSB_DJOB_RU_INTERVAL variable is set in parallel or distributed job environment. You should not manually change the value of LSB_DJOB_RU_INTERVAL.

By default, the interval is equal to SBD_SLEEP_TIME in `lsb.params`, where the default value of SBD_SLEEP_TIME is 30 seconds.

How blaunch supports task geometry and process group files

The current support for task geometry in LSF requires the user submitting a job to specify the wanted task geometry by setting the environment variable LSB_PJL_TASK_GEOMETRY in their submission environment before job submission. LSF checks for LSB_PJL_TASK_GEOMETRY and modifies LSB_MCPU_HOSTS appropriately

The environment variable LSB_PJL_TASK_GEOMETRY is checked for all parallel jobs. If LSB_PJL_TASK_GEOMETRY is set users submit a parallel job (a job that requests more than 1 slot), LSF attempts to shape LSB_MCPU_HOSTS accordingly.

Resource collection for all commands in a job script

Parallel and distributed jobs are typically launched with a job script. If your job script runs multiple commands, you can ensure that resource usage is collected correctly for all commands in a job script by configuring LSF_HPC_EXTENSIONS=CUMULATIVE_RUSAGE in `lsf.conf`. Resource usage is collected for jobs in the job script, rather than being overwritten when each command is executed.

Resizable jobs and blaunch

Because a resizable job can be resized any time, the `blaunch` framework is aware of the newly added resources (hosts) or released resources. When a validation request comes with those additional resources, the `blaunch` framework accepts the request and launches the remote tasks accordingly. When part of an allocation is released, the `blaunch` framework makes sure no remote tasks are running on those released resources, by terminating remote tasks on the released hosts if any. Any further validation requests with those released resources are rejected.

The `blaunch` framework provides the following functionality for resizable jobs:

- The `blaunch` command and `lsb_getalloc()` API call accesses up to date resource allocation through the LSB_DJOB_HOSTFILE environment variable
- Validation request (to launch remote tasks) with the additional resources succeeds
- Validation request (to launch remote tasks) with the released resources fails
- Remote tasks on the released resources are terminated and the `blaunch` framework terminates tasks on a host when the host has been completely removed from the allocation.
- When releasing resources, LSF allows a configurable grace period (DJOB_RESIZE_ GRACE_PERIOD in `lsb.applications`) for tasks to clean up and exit themselves. By default, there is no grace period.
- When remote tasks are launched on new additional hosts but the notification command fails, those remote tasks are terminated.

Submitting jobs with blaunch

Use `bsub` to call `blaunch`, or to invoke an execution script that calls `blaunch`. The `blaunch` command assumes that `bsub -n` implies one task per job slot.

- Submit a job:

  ```
  bsub -n 4 blaunch myjob
  ```

- Submit a job to launch tasks on a specific host:

  ```
  bsub -n 4 blaunch hostA myjob
  ```

- Submit a job with a host list:

  ```
  bsub -n 4 blaunch -z "hostA hostB" myjob
  ```

- Submit a job with a host file:

  ```
  bsub -n 4 blaunch -u ./hostfile myjob
  ```

- Submit a job to an application profile

  ```
  bsub -n 4 -app djob blaunch myjob
  ```

Example execution scripts

Launching MPICH-P4 tasks

To launch an MPICH-P4 tasks through LSF using the `blaunch` framework, substitute the path to `rsh` or `ssh` with the path to `blaunch`. For example:

Sample `mpirun` script changes:

```
...
# Set default variables
AUTOMOUNTFIX="sed -e s@/tmp_mnt/@/@g"
DEFAULT_DEVICE=ch_p4
RSHCOMMAND="$LSF_BINDIR/blaunch"
SYNCLOC=/bin/sync
CC="cc"
...
```

You must also set special arguments for the ch_p4 device:

```
#! /bin/sh
#
# mpirun.ch_p4.args
#
# Special args for the ch_p4 device
setrshcmd="yes"
givenPGFile=0
case $arg in
...
```

Sample job submission script:

```
#! /bin/sh
#
# job script for MPICH-P4
#
#BSUB -n 2
#BSUB -R'span[ptile=1]'
#BSUB -o %J.out
#BSUB -e %J.err
NUMPROC=`wc -l $LSB_DJOB_HOSTFILE|cut -f 1 -d ' '`
mpirun -n $NUMPROC -machinefile $LSB_DJOB_HOSTFILE ./myjob
```

Launching ANSYS jobs

To launch an ANSYS job through LSF using the `blaunch` framework, substitute the path to `rsh` or `ssh` with the path to `blaunch`. For example:

```
#BSUB -o stdout.txt
#BSUB -e stderr.txt
# Note: This case statement should be used to set up any
# environment variables needed to run the different versions
# of Ansys. All versions in this case statement that have the
# string "version list entry" on the same line will appear as
# choices in the Ansys service submission page.

case $VERSION in
 10.0)  #version list entry
        export ANSYS_DIR=/usr/share/app/ansys_inc/v100/Ansys
        export ANSYSLMD_LICENSE_FILE=1051@licserver.company.com
       export MPI_REMSH=/opt/lsf/bin/blaunch
        program=${ANSYS_DIR}/bin/ansys100
        ;;
  *)
        echo "Invalid version ($VERSION) specified"
        exit 1
        ;;
esac

if [ -z "$JOBNAME" ]; then
    export JOBNAME=ANSYS-$$
fi

if [ $CPUS -eq 1 ]; then
```

```
    ${program} -p ansys -j $JOBNAME -s read -l en-us -b -i $INPUT $OPTS
else
    if [ $MEMORY_ARCH = "Distributed" ] Then
      HOSTLIST=`echo $LSB_HOSTS | sed s/" "/":1:"/g` ${program} -j $JOBNAME -p
ansys -pp -dis -machines \
    ${HOSTLIST}:1 -i $INPUT $OPTS
    else
       ${program} -j $JOBNAME -p ansys -pp -dis -np $CPUS \
    -i $INPUT $OPTS
    fi
fi
```

OpenMP Jobs

Platform LSF provides the ability to start parallel jobs that use OpenMP to communicate between process on shared-memory machines and MPI to communicate across networked and non-shared memory machines.

This implementation allows you to specify the number of machines and to reserve an equal number of processors per machine. When the job is dispatched, PAM only starts one process per machine.

OpenMP specification The OpenMP specifications are owned and managed by the OpenMP Architecture Review Board. See `www.openmp.org` for detailed information.

OpenMP esub

An `esub` for OpenMP jobs, `esub.openmp`, is installed with Platform LSF. The OpenMP `esub` sets environment variable LSF_PAM_HOSTLIST_USE=unique, and starts PAM.

Use `bsub -a openmp` to submit OpenMP jobs.

Submitting OpenMP jobs

To run an OpenMP job with MPI on multiple hosts, specify the number of processors and the number of processes per machine. For example, to reserve 32 processors and run 4 processes per machine:

```
bsub -a openmp -n 32 -R "span[ptile=4]" myOpenMPJob
```

`myOpenMPJob` runs across 8 machines (4/32=8) and PAM starts 1 MPI process per machine.

To run a parallel OpenMP job on a single host, specify the number of processors:

```
bsub -a openmp -n 4 -R "span[hosts=1]" myOpenMPJob
```

PVM Jobs

Parallel Virtual Machine (PVM) is a parallel programming system distributed by Oak Ridge National Laboratory. PVM programs are controlled by the PVM hosts file, which contains host names and other information.

PVM esub

An `esub` for PVM jobs, `esub.pvm`, is installed with Platform LSF. The PVM `esub` calls the `pvmjob` script.

Use `bsub -a pvm` to submit PVM jobs.

pvmjob script

The `pvmjob` shell script is invoked by `esub.pvm` to run PVM programs as parallel LSF jobs. The `pvmjob` script reads the LSF environment variables, sets up the PVM hosts file and then runs the PVM job. If your PVM job needs special options in the hosts file, you can modify the `pvmjob` script.

Example

For example, if the command line to run your PVM job is:

```
myjob data1 -o out1
```

the following command submits this job to run on 10 processors:

```
bsub -a pvm -n 10 myjob data1 -o out1
```

Other parallel programming packages can be supported in the same way.

SGI Vendor MPI Support

Compiling and linking your MPI program

You must use the SGI C compiler (`cc` by default). You cannot use `mpicc` to build your programs.

For example, use one of the following compilation commands to build the program `mpi_sgi`:

- On IRIX/TRIX:

  ```
  cc -g -64 -o mpi_sgi mpi_sgi.c -lmpi
  f90 -g -64 -o mpi_sgi mpi_sgi.c -lmpi
  cc -g -n32 -mips3 -o mpi_sgi mpi_sgi.c -lmpi
  ```

- On Altix:

  ```
  efc -g -o mpi_sgi mpi_sgi.f -lmpi
  ecc -g -o mpi_sgi mpi_sgi.c -lmpi
  gcc -g -o mpi_sgi mpi_sgi.c -lmpi
  ```

System requirements

SGI MPI has the following system requirements:

- Your SGI systems must be running IRIX 6.5.24 or higher, or SGI Alitx ProPack 3.0 or higher, with the latest operating system patches applied. Use the `uname` command to determine your system configuration. For example:

  ```
  uname -aR
  IRIX64 hostA 6.5 6.5.17f 07121148 IP27
  ```

- SGI MPI version:
  - On IRIX/TRIX: SGI MPI 3.2.04 (MPT 1.3.0.3) released December 7 1999 or later with the latest patches applied
  - On Altix: MPT 1.8.1 or later and SGI Array Services 3.6 or later

Use the one of the following commands to determine your installation:

- On IRIX/TRIX:

  ```
  versions mpt mpi sma
  ```

- On Altix:

  ```
  rpm -qa | grep sgi-mpt
  rpm -qa | grep sgi-array
  ```

Configuring LSF to work with SGI MPI

To use 32-bit or 64-bit SGI MPI with Platform LSF, set the following parameters in `lsf.conf`:

- Set LSF_VPLUGIN to the full path to the MPI library `libxmpi.so`.

  For example:
  - On SGI IRIX: `LSF_VPLUGIN="/usr/lib32/libxmpi.so"`
  - On SGI Altix: `LSF_VPLUGIN="/usr/lib/libxmpi.so"`

You can specify multiple paths for LSF_VPLUGIN, separated by colons (`:`). For example, the following configures both `/usr/lib32/libxmpi.so` for SGI IRIX, and `/usr/lib/libxmpi.so` for SGI IRIX:

```
LSF_VPLUGIN="/usr/lib32/libxmpi.so:/usr/lib/libxmpi.so"
```

- LSF_PAM_USE_ASH=Y enables LSF to use the SGI Array Session Handler (ASH) to propagate signals to the parallel jobs.

  See the SGI system documentation and the `array_session`(5) man page for more information about array sessions.

libxmpi.so file permission

For PAM to access the `libxmpi.so` library, the file permission mode must be 755 (`-rwxr-xr-x`).

Array services authentication (Altix only)

For PAM jobs on Altix, the SGI Array Services daemon `arrayd` must be running and AUTHENTICATION must be set to NONE in the SGI array services authentication file `/usr/lib/array/arrayd.auth` (comment out the AUTHENTICATION NOREMOTE method and uncomment the AUTHENTICATION NONE method).

To run a mulithost MPI applications, you must also enable `rsh` without password prompt between hosts:

- The remote host must defined in the `arrayd` configuration.
- Configure `.rhosts` so that `rsh` does not require a password.

The pam command

The `pam` command invokes the Platform Parallel Application Manager (PAM) to run parallel batch jobs in LSF. It uses the `mpirun` library and SGI array services to spawn the child processes needed for the parallel tasks that make up your MPI application. It starts these tasks on the systems allocated by LSF. The allocation includes the number of execution hosts needed, and the number of child processes needed on each host.

Using the pam -mpi option

The `-mpi` option on the `bsub` and `pam` command line is equivalent to `mpirun` in the SGI environment.

Using the pam -auto_place option

The `-auto_place` option on the `pam` command line tells the `mpirun` library to launch the MPI application according to the resources allocated by LSF.

Using the pam -n option

The `-n` option on the `pam` command line notifies PAM to wait for `-n` number of TaskStarter to return.

You can use both `bsub -n` and `pam -n` in the same job submission. The number specified in the `pam -n` option should be less than or equal to the number specified by `bsub -n`. If the number of tasks specified with `pam -n` is greater than the number specified by `bsub -n`, the `pam -n` is ignored.

For example, you can specify:

```
bsub -n 5 pam -n 2 a.out
```

Here, the job requests 5 processors, but PAM only starts 2 parallel tasks.

Examples

Running a job

To run a job and have LSF select the host, the command:

```
mpirun -np 4 a.out
```

is entered as:

```
bsub -n 4 pam -mpi -auto_place a.out
```

Running a job on a single host

To run a single-host job and have LSF select the host, the command:

```
mpirun -np 4 a.out
```

is entered as:

```
bsub -n 4 -R "span[hosts=1]" pam -mpi -auto_place a.out
```

Running a job on multiple hosts

To run a multihost job (5 processors per host) and have LSF select the hosts, the following command:

```
mpirun hosta -np 5 a.out: hostb -np 5 a.out
```

is entered as:

```
bsub -n 10 -R "span[ptile=5]" pam -mpi -auto_place a.out
```

For a complete list of `mpirun` options and environment variable controls refer to the SGI `mpirun` man page.

Limitations

- SBD and MBD take a few seconds to get the process IDs and process group IDs of the PAM jobs from the SGI MPI components, If you use `bstop`, `bresume`, or `bkill` before this happens, uncontrolled MPI child processes may be left running.
- A single MPI job cannot run on a heterogeneous architecture. The entire job must run on systems of a single architecture.

HP Vendor MPI Support

When you use `mpirun` in stand-alone mode, you specify host names to be used by the MPI job.

Automatic Platform MPI library configuration

During installation, `lsfinstall` sets LSF_VPLUGIN in `lsf.conf` to the full path to the MPI library `libmpirm.sl`. For example:

```
LSF_VPLUGIN="/opt/mpi/lib/pa1.1/libmpirm.sl"
```

On Linux

On Linux hosts running Platform MPI, you must manually set the full path to the vendor MPI library `libmpirm.so`.

For example, if Platform MPI is installed in `/opt/hpmpi`:

```
LSF_VPLUGIN="/opt/hpmpi/lib/linux_ia32/libmpirm.so"
```

The pam command

The `pam` command invokes the Platform Parallel Application Manager (PAM) to run parallel batch jobs in LSF. It uses the `mpirun` library to spawn the child processes needed for the parallel tasks that make up your MPI application. It starts these tasks on the systems allocated by LSF. The allocation includes the number of execution hosts needed, and the number of child processes needed on each host.

Automatic host allocation by LSF

Using the pam -mpi option

To achieve better resource utilization, you can have LSF manage the allocation of hosts, coordinating the start-up phase with `mpirun`.

This is done by preceding the regular `mpirun` command with:

```
bsub pam -mpi
```

The `-mpi` option on the `bsub` and `pam` command line is equivalent to `mpirun` in the Platform MPI environment. The `-mpi` option must be the first option of the `pam` command.

How to run Platform MPI jobs

1. Add the Platform MPI command `mpirun` in the *$PATH* environment variable.
2. Set the *MPI_ROOT* environment variable to point to the Platform MPI installation directory.
3. Set LSF_VPLUGIN in `lsf.conf` or in your environment.
4. Submit thte job with `-lsb_hosts` option: **`bsub -I -n 3 pam -mpi mpirun -lsb_hosts myjob`**

Running a job on a single host

For example, to run a single-host job and have LSF select the host, the command:

```
mpirun -np 14 a.out
```

is entered as:

```
bsub pam -mpi mpirun -np 14 a.out
```

Running a job on multiple hosts

For example, to run a multi-host job and have LSF select the hosts, the command:

```
mpirun -f appfile
```

is entered as:

```
bsub -n 8 -R "span[ptile=4]" pam -mpi mpirun -f appfile
```

where `appfile` contains the following entries:

```
-h host1 -np 4 a.out
-h host2 -np 4 b.out
```

In this example `host1` and `host2` are used in place of actual host names and refer to the actual hosts that LSF allocates to the job.

LSF Generic Parallel Job Launcher Framework

Any parallel execution environment (for example a vendor MPI, or an MPI package like MPICH-GM, MPICH-P4, or LAM/MPI) can be made compatible with LSF using the generic parallel job launcher (PJL) framework.

Vendor MPIs for SGI MPI and Platform MPI are already integrated with Platform LSF.

The generic PJL integration is a framework that allows you to integrate any vendor's parallel job launcher with Platform LSF. PAM does not launch the parallel jobs directly, but manages the job to monitor job resource usage and provide job control over the parallel tasks.

System requirements

- Vendor parallel package is installed and operating properly
- LSF cluster is installed and operating properly

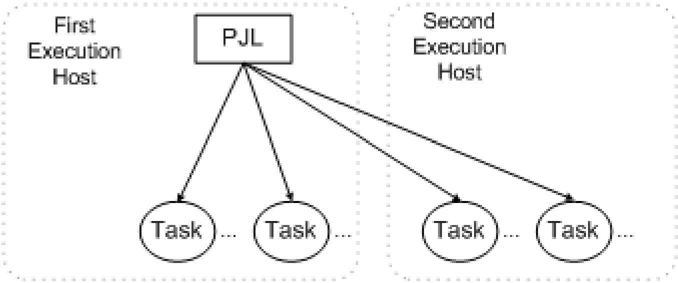
How the Generic PJL Framework Works

Terminology

First execution host	The host name at the top of the execution host list as determined by LSF. Starts PAM.
Execution hosts	The most suitable hosts to execute the batch job as determined by LSF
task	A process that runs on a host; the individual process of a parallel application
parallel job	A parallel job consists of multiple tasks that could be executed on different hosts.
PJL	(Parallel Job Launcher) Any executable script or binary capable of starting parallel tasks on all hosts assigned for a parallel job (for example, `mpirun`.)
sbatchd	Slave Batch Daemons (SBDs) are batch job execution agents residing on the execution hosts. `sbatchd` receives jobs from `mbatchd` in the form of a job specification and starts RES to run the job according the specification. `sbatchd` reports the batch job status to `mbatchd` whenever job state changes.
mpirun.lsf	Reads the environment variable LSF_PJL_TYPE, and generates the appropriate `pam` command line to invoke the PJL. The esub programs provided in LSF_SERVERDIR set this variable to the proper type.
TS	(TaskStarter) An executable responsible for starting a parallel task on a host and reporting the process ID and host name to PAM. TS is located in LSF_BINDIR.
PAM	(Parallel Application Manager) The supervisor of any parallel LSF job. PAM allows LSF to collect resources used by the job and perform job control. PAM starts the PJL and maintains connection with RES on all execution hosts. It collects resource usage, updates the resource usage of tasks and its own PID and PGID to `sbatchd`. It propagates signals to all process groups and individual tasks, and cleans up tasks as needed.
PJL wrapper	A script that starts the PJL. The wrapper is typically used to set up the environment for the parallel job and invokes the PJL.
RES	(Remote Execution Server) An LSF daemon running on each server host. Accepts remote execution requests to provide transparent and secure remote execution of jobs and tasks. RES manages all remote tasks and forwards signals, standard I/O, resources consumption data, and parallel job information between PAM and the tasks.

Architecture

Running a parallel job using a non-integrated PJL



Without the generic PJL framework, the PJL starts tasks directly on each host, and manages the job.

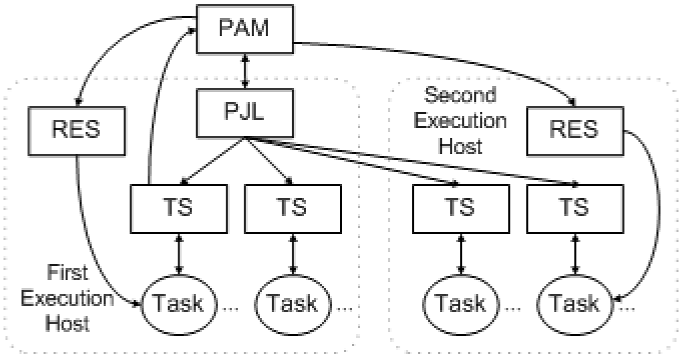
Even if the MPI job was submitted through LSF, LSF never receives information about the individual tasks. LSF is not able to track job resource usage or provide job control.

If you simply replace PAM with a parallel job launcher that is not integrated with LSF, LSF loses control of the process and is not able to monitor job resource usage or provide job control. LSF never receives information about the individual tasks.

Using the generic PJL framework

PAM is the resource manager for the job. The key step in the integration is to place TS in the job startup hierarchy, just before the task starts. TS must be the parent process of each task in order to collect the task process ID (PID) and pass it to PAM.

The following figure illustrates the relationship between PAM, PJL, PJL wrapper, TS, and the parallel job tasks.



1. Instead of starting the PJL directly, PAM starts the specified PJL wrapper on a single host.
2. The PJL wrapper starts the PJL (for example, `mpirun`).
3. Instead of starting tasks directly, PJL starts TS on each host selected to run the parallel job.
4. TS starts the task.

Each TS reports its task PID and host name back to PAM. Now PAM can perform job control and resource usage collection through RES.

TaskStarter also collects the exit status of the task and reports it to PAM. When PJL exits, PAM exits with the same termination status as the PJL.

Customize mpirun.lsf

If you choose to customize `mpirun.lsf` and your job scripts call `mpirun.lsf` more than once, make use of the the environment variables that call a custom command, script, or binary when needed:

- $MPIRUN_LSF_PRE_EXEC: Runs before calling pam..PJL_wrapper.
- $MPIRUN_LSF_POST_EXEC: Runs after calling pam..PJL_wrapper.

These environment variables are run as users.

Integration methods

There are 2 ways to integrate the PJL.

Method 1

In this method, PAM rewrites the PJL command line to insert TS in the correct position, and set callback information for TS to communicate with PAM.

Use this method when:

- You always use the same number of PJL arguments
- The job in the PJL command line is the executable application that starts the parallel tasks

For details, see "Integration Method 1" on page 37

Method 2

In this method, you rewrite or wrap the PJL to include TS and callback information for TS to communicate with PAM. This method of integration is the most flexible, but may be more difficult to implement.

Use this method when:

- The number of PJL arguments is uncertain
- Parallel tasks have a complex startup sequence
- The job in the PJL command line could be a script instead of the executable application that starts the parallel tasks

For details, see "Integration Method 2" on page 39.

Error handling

1. If PAM cannot start PJL, no tasks are started and PAM exits.
2. If PAM does not receive all the TS registration messages (host name and PID) within a the timeout specified by LSF_HPC_PJL_LOADENV_TIMEOUT in `lsf.conf`, it assumes that the job can not be executed. It kills the PJL, kills all the tasks that have been successfully started (if any), and exits. The default for LSF_HPC_PJL_LOADENV_TIMEOUT is 300 seconds.
3. If TS cannot start the task, it reports this to PAM and exits. If all tasks report, PAM checks to make sure all tasks have started. If any task does not start, PAM kills the PJL, sends a message to kill all the remote tasks that have been successfully started, and exit.
4. If TS terminates before it can report the exit status of the task to PAM, PAM never succeeds in receiving all the exit status. It then exits when the PJL exits.
5. If the PJL exits before all TS have registered the exit status of the tasks, then PAM assumes the parallel job is completed, and communicates with RES, which signals the tasks.

Using the pam -n option (SGI MPI only)

The `-n` option on the `pam` command line specifies the number of tasks that PAM should start.

You can use both `bsub -n` and `pam -n` in the same job submission. The number specified in the `pam -n` option should be less than or equal to the number specified by `bsub -n`. If the number of task specified with `pam -n` is greater than the number specified by `bsub -n`, the `pam -n` is ignored.

For example, you can specify:

```
bsub -n 5 pam -n 2 -mpi a.out
```

Here, 5 processors are reserved for the job, but PAM only starts 2 parallel tasks.

Custom job controls for parallel jobs

As with sequential LSF jobs, you can use the JOB_CONTROLS parameter in the queue (`lsb.queues`) to configure custom job controls for your parallel jobs.

If the custom job control contains ...	Platform LSF ...
A signal name (for example, SIGSTOP or SIGTSTP)	Propagates the signal to the PAM PGID and all parallel tasks
A `/bin/sh` command line or script	Sets all job environment variables for the command action. Sets the following additional environment variables: ◆ LSB_JOBPGIDS—a list of current process group IDs of the job ◆ LSB_JOBPIDS—a list of current process IDs of the job ◆ LSB_PAMPID—the PAM process ID ◆ LSB_JOBRES_PID—the process ID of RES for the job For the SUSPEND action command, sets the following environment variables: ◆ LSB_SUSP_REASONS—an integer representing a bitmap of suspending reasons as defined in lsbatch.h. The suspending reason can allow the command to take different actions based on the reason for suspending the job. ◆ LSB_SUSP_SUBREASONS—an integer representing the load index that caused the job to be suspended. When the suspending reason SUSP_LOAD_REASON (suspended by load) is set in LSB_SUSP_REASONS, LSB_SUSP_SUBREASONS set to one of the load index values defined in lsf.h.

Using the LSB_JOBRES_PID and LSB_PAMPID environment variables

How to use these two variables in your job control scripts:

- If `pam` and the job RES are in same process group, use LSB_JOBRES_PID. Here is an example of JOB_CONTROL defined in the queue:

```
JOB_CONTROLS = TERMINATE[kill -CONT -$LSB_JOBRES_PID; kill -TERM
-$LSB_JOBRES_PID]
```

- If `pam` and the job RES are in different process groups (for example, `pam` is started by a wrapper, which could set its own PGID). Use both LSB_JOBRES_PID and LSB_PAMPID to make sure your parallel jobs are cleaned up.

```
JOB_CONTROLS = TERMINATE[kill -CONT -$LSB_JOBRES_PID -$LSB_PAMPID; kill -TERM
-$LSB_JOBRES_PID -$LSB_PAMPID]
```

LSB_PAM_PID may not be available when job first starts. It take some time for pam to register back its PID to `sbatchd`.

For more information See the *Platform LSF Configuration Reference* for information about JOB_CONTROLS in the `lsb.queues` file.

See *Administering Platform LSF* for information about configuring job controls.

Sample job termination script for queue job control

By default, LSF sends a SIGUSR2 signal to terminate a job that has reached its run limit or deadline. Some applications do not respond to the SIGUSR2 signal (for example, LAM/MPI), so jobs may not exit immediately when a job run limit is reached. You should configure your queues with a custom job termination action specified by the JOB_CONTROLS parameter.

Sample script Use the following sample job termination control script for the TERMINATE job control in the `hpc_linux` queue for LAM/MPI jobs:

```
#!/bin/sh

#JOB_CONTROL_LOG=job.control.log.$LSB_BATCH_JID
JOB_CONTROL_LOG=/dev/null

kill -CONT -$LSB_JOBRES_PID >>$JOB_CONTROL_LOG 2>&1

if [ "$LSB_PAM_PID" != "" -a "$LSB_PAM_PID" != "0" ]; then
    kill -TERM $LSB_PAM_PID >>$JOB_CONTROL_LOG 2>&1

    MACHINETYPE=`uname -a | cut -d" " -f 5`
    while [ "$LSB_PAM_PID" != "0" -a "$LSB_PAM_PID" != "" ] # pam is running
    do
        if [ "$MACHINETYPE" = "CRAY" ]; then
            PIDS=`(ps -ef; ps auxww) 2>/dev/null | egrep ".*[/\[ \t]pam[]
\t]*$"| sed -n "/grep/d;s/^ *[^ \t]* *\([0-9]*\).*/\1/p" | sort -u`
        else
            PIDS=`(ps -ef; ps auxww) 2>/dev/null | egrep " pam |/pam |
pam$|/pam$"| sed -n "/grep/d;s/^ *[^ \t]* *\([0-9]*\).*/\1/p" | sort -u`
        fi

        echo PIDS=$PIDS >> $JOB_CONTROL_LOG
        if [ "$PIDS" = "" ]; then # no pam is running
            break;
        fi
        foundPamPid="N"
        for apid in $PIDS
        do
            if [ "$apid" = "$LSB_PAM_PID" ]; then
                # pam is running
                foundPamPid="Y"
                break
            fi
        done

        if [ "$foundPamPid" == "N" ]; then
            break # pam has exited
        fi
        sleep 2
    done
fi

# User other terminate signals if SIGTERM is
# caught and ignored by your application.
kill -TERM -$LSB_JOBRES_PID >>$JOB_CONTROL_LOG 2>&1
exit 0
```

To configure the script in the hpc_linux queue

1. Create a job control script named `job_terminate_control.sh`.
2. Make the script executable:

   ```
   chmod +x job_terminate_control.sh
   ```

3. Edit the hpc_linux queue in `lsb.queues` to configure your `job_terminate_control.sh` script as the TERMINATE action in the JOB_CONTROLS parameter. For example:

   ```
   Begin Queue
   QUEUE_NAME    = hpc_linux_tv
   PRIORITY      = 30
   NICE          = 20
   # ...
   JOB_CONTROLS = TERMINATE[kill -CONT -$LSB_JOBRES_PID; kill
   -TERM -$LSB_JOBRES_PID]
   JOB_CONTROLS = TERMINATE [/path/job_terminate_control.sh]
   TERMINATE_WHEN = LOAD PREEMPT WINDOW
   RERUNNABLE = NO
   INTERACTIVE = NO
   DESCRIPTION   = Platform LSF TotalView Debug queue.
   End Queue
   ```

4. Reconfigure your cluster to make the change take effect:

   ```
   # badmin mbdrestart
   ```

Integration Method 1

When to use this integration method

In this method, PAM rewrites the PJL command line to insert TS in the correct position, and set callback information for TS to communicate with PAM.

Use this method when:

- You always use the same number of PJL arguments
- The job in the PJL command line is the executable application that starts the parallel tasks

Using pam to call the PJL

Submit jobs using `pam` in the following format:

```
pam [other_pam_options] -g num_args pjl [pjl_options] job [job_options]
```

The command line includes:

- The `pam` command and its options (*other_pam_options*)
- the `pam -g num_args` option
- The parallel job launcher or PJL wrapper (*pjl*) and its options (*pjl_options*)
- The job to run (*job*) and its options (*job_options*)

pam options

The `-g` option is required to use the generic PJL framework. You must specify all the other `pam` options before `-g`.

num_args specifies how many space-separated arguments in the command line are related to the PJL, including the PJL itself (after that, the rest of the command line is assumed to be related to the binary application that launches the parallel tasks).

For example:

- A PJL named `no_arg_pjl` takes no options, so `-g 1` is required after the other `pam` options:

  ```
  pam [pam_options] -g 1 no_arg_pjl job [job_options]
  ```

- A PJL is named `3_arg_pjl` and takes the options `-a`, `-b`, and `group_name`, so The option `-g 4` is required after the other `pam` options:

```
pam [pam_options] -g 4 3_arg_pjl -a -b group_name job [job_options]
```

How PAM inserts TaskStarter

Before the PJL is started, PAM automatically modifies the command line and inserts the TS, the host and port for TS to contact PAM, and the LSF_ENVDIR in the correct position before the actual job.

TS is placed between the PJL and the parallel application. In this way, the TS starts each task, and LSF can monitor resource usage and control the task.

For example, if your LSF directory is `/usr/share/lsf` and you input:

```
pam [pam_options] -g 3 my_pjl -b group_name job [job_options]
```

PAM automatically modifies the PJL command line to:

```
my_pjl -b group_name /usr/share/lsf/TaskStarter -p host_name:port_number
-c /user/share/lsf/conf job [job_options] [pjl_options]
```

For more detailed examples See “Example Integration: LAM/MPI” on page 47

Integration Method 2

When to use this integration method

In this method, you rewrite or wrap the PJL to include TS and callback information for TS to communicate with PAM. This method of integration is the most flexible, but may be more difficult to implement.

Use this method when:

- The number of PJL arguments varies
- Parallel tasks have a complex startup sequence
- The job in the PJL command line could be a script instead of the executable application that starts the parallel tasks

Using pam to call the PJL

Submit jobs using `pam` in the following format:

```
pam [other_pam_options] -g pjl_wrap [pjl_wrap_options] job [job_options]
```

The command line includes:

- The PJL wrapper script (*pjl_wrap*) and its options (*pjl_wrap_options*). This wrapper script must insert TS in the correct position before the actual job command.
- The job to run (*job*) and its options (*job_options*)

  The job could be a wrapper script that starts the application that starts the parallel tasks, or it could be the executable application itself

pam options The `-g` option is required to use the generic PJL framework. You must specify all the other `pam` options before `-g`.

Placing TaskStarter in your code

Each end job task must be started by the binary TaskStarter that is provided by Platform Computing.

When you use this method, PAM does not insert TS for you. You must modify your code to use TS and the LSF_TS_OPTIONS environment variable. LSF_TS_OPTIONS is created by PAM on the first execution host and contains the callback information for TS to contact PAM.

You must insert TS and the PAM callback information directly in front of the executable application that starts the parallel tasks.

To place TS and its options, you can modify either the PJL wrapper or the job script, depending on your implementation. If the package requires the path, specify the full path to `TaskStarter`.

Example

This example modifies the PJL wrapper. The job script includes both the PJL wrapper and the job itself.

Before Without the integration, your job submission command line is:

```
bsub -n 2 jobscript
```

Your job script is:

```
#!/bin/sh
if [ -n "$ENV1" ]; then
  pjl -opt1 job1
else
  pjl -opt2 -opt3 job2
fi
```

After After the integration, your job submission command line includes the `pam` command:

```
bsub -n 2 pam -g new_jobscript
```

Your new job script inserts TS and LSF_TS_OPTIONS before the jobs:

```
#!/bin/sh
if [ -n "$ENV1" ]; then
  pjl -opt1 usr/share/lsf/TaskStarter $LSF_TS_OPTIONS job1
else
  pjl -opt2 -opt3 usr/share/lsf/TaskStarter $LSF_TS_OPTIONS
job2
fi
```

For more detailed examples See "Example Integration: LAM/MPI" on page 47

Tuning PAM Scalability and Fault Tolerance

To improve performance and scalability for large parallel jobs, tune the following parameters.

Parameters for PAM (lsf.conf)

For better performance, you can adjust the following parameters in `lsf.conf`. The user's environment can override these.

LSF_HPC_PJL_LOADENV_TIMEOUT

Timeout value in seconds for PJL to load or unload the environment. For example, the time needed for IBM POE to load or unload adapter windows.

At job startup, the PJL times out if the first task fails to register within the specified timeout value. At job shutdown, the PJL times out if it fails to exit after the last Taskstarter termination report within the specified timeout value.

Default: LSF_HPC_PJL_LOADENV_TIMEOUT=300

LSF_PAM_RUSAGE_UPD_FACTOR

This factor adjusts the update interval according to the following calculation:

RUSAGE_UPDATE_INTERVAL + *num_tasks* * 1 * LSF_PAM_RUSAGE_UPD_FACTOR.

PAM updates resource usage for each task for every SBD_SLEEP_TIME + *num_tasks* * 1 seconds (by default, SBD_SLEEP_TIME=15). For large parallel jobs, this interval is too long. As the number of parallel tasks increases, LSF_PAM_RUSAGE_UPD_FACTOR causes more frequent updates.

Default: LSF_PAM_RUSAGE_UPD_FACTOR=0.01

Running Jobs with Task Geometry

Specifying task geometry allows you to group tasks of a parallel job step to run together on the same node. Task geometry allows for flexibility in how tasks are grouped for execution on system nodes. You cannot specify the particular nodes that these groups run on; the scheduler decides which nodes run the specified groupings.

Task geometry is supported for all Platform LSF MPI integrations including IBM POE, LAM/MPI, MPICH-GM, MPICH-P4, and Intel® MPI.

Use the LSB_PJL_TASK_GEOMETRY environment variable to specify task geometry for your jobs. LSB_PJL_TASK_GEOMETRY overrides any process group or command file placement options.

The environment variable LSB_PJL_TASK_GEOMETRY is checked for all parallel jobs. If LSB_PJL_TASK_GEOMETRY is set users submit a parallel job (a job that requests more than 1 slot), LSF attempts to shape LSB_MCPU_HOSTS accordingly.

The `mpirun.lsf` script sets the LSB_MCPU_HOSTS environment variable in the job according to the task geometry specification. The PJL wrapper script controls the actual PJL to start tasks based on the new LSB_MCPU_HOSTS and task geometry.

Syntax

`setenv LSB_PJL_TASK_GEOMETRY "{(`*`task_ID`*`,...) ...}"`

For example, to submit a job to spawn 8 tasks and span 4 nodes, specify:

```
setenv LSB_PJL_TASK_GEOMETRY "{(2,5,7)(0,6)(1,3)(4)}"
```

- Tasks 2,5, and 7 run on one node
- Tasks 0 and 6 run on another node
- Tasks 1 and 3 run on a third node
- Task 4 runs on one node alone

Each *task_ID* number corresponds to a task ID in a job, each set of parenthesis contains the task IDs assigned to one node. Tasks can appear in any order, but the entire range of tasks specified must begin with 0, and must include all task ID numbers; you cannot skip a task ID number. Use braces to enclose the entire task geometry specification, and use parentheses to enclose groups of nodes. Use commas to separate task IDs.

For example.

```
setenv LSB_PJL_TASK_GEOMETRY "{(1)(2)}"
```

is incorrect because it does not start from task 0.

```
setenv LSB_PJL_TASK_GEOMETRY "{(0)(3)}"
```

is incorrect because it does not specify task 1and 2.

LSB_PJL_TASK_GEOMETRY cannot request more hosts than specified by the `bsub -n` option.

For example:

```
setenv LSB_PJL_TASK_GEOMETRY "{(0)(1)(2)}"
```

specifies three nodes, one task per node. A correct job submission must request at least 3 hosts:

```
bsub -n 3 -R "span[ptile=1]" -I -a mpich_gm mpirun.lsf my_job
Job <564> is submitted to queue <hpc_linux>.
<<Waiting for dispatch ...>>
<<Starting on hostA>>
...
```

Planning your task geometry specification

You should plan their task geometry in advance and specify the job resource requirements for LSF to select hosts appropriately.

Use `bsub -n` and `-R "span[ptile=]"` to make sure LSF selects appropriate hosts to run the job, so that:

- The correct number of nodes is specified
- All exceution hosts have the same number of available slots
- The `ptile` value is the maximum number of CPUs required on one node by task geometry specifications.

LSB_PJL_TASK_GEOMETRY only guarantees the geometry but does not guarantee the host order. You must make sure each host selected by LSF can run any group of tasks specified in LSB_PJL_TASK_GEOMETRY.

You can also use `bsub -x` to run jobs exclusively on a host. No other jobs share the node once this job is scheduled.

Usage notes and limitations

- MPICH-P4 jobs:

  MPICH-P4 `mpirun` requires the first task to run on local node OR all tasks to run on remote node (`-nolocal`). If the LSB_PJL_TASK_GEOMETRY environment variable is set, `mpirun.lsf` makes sure the task group that contains task 0 in LSB_PJL_TASK_GEOMETRY runs on the first node.
- LAM/MPI jobs:

  You should not specify `mpirun n` manually on command line; you should use LSB_PJL_TASK_GEOMETRY for consistency with other Platform LSF MPI integrations. LSB_PJL_TASK_GEOMETRY overrides the `mpirun n` option.
- OpenMPI jobs:

  Each thread of an OpenMPI job is counted as a task. For example, task geometry specification is:

  ```
  setenv LSB_PJL_TASK_GEOMETRY "{(1), (2,3,4) (0,5)}"
  ```

  and task 5 is an `openmp` job that spawns 3 threads. From this specification, the job spans 3 nodes, and maximum number of CPUs required is 4 (because `(0,5)` requires 4 cpus). The job should be submitted as:

  ```
  bsub -n 12 -R "span[ptile=4]" -a openmp mpirun.lsf myjob
  ```

Examples

For the following task geometry:

```
setenv LSB_PJL_TASK_GEOMETRY "{(2,5,7)(0,6)(1,3)(4)}"
```

The job submission should look like:

```
bsub -n 12 -R "span[ptile=3]" -a poe mpirun.lsf myjob
```

If task 6 is an OpenMP job that spawns 4 threads, the job submission is:

```
bsub -n 20 -R "span[ptile=5]" -a poe mpirun.lsf myjob
```

Do not use -a openmp or set LSF_PAM_HOSTLIST_USE for OpenMP jobs.

A POE job has three tasks: `task0`, `task1`, and `task2`, and

Task `task2` spawns 3 threads. The tasks `task0` and `task1` run on one node and `task2` runs on the other node. The job submission is:

```
bsub -a poe -n 6 -R "span[ptile=3]" mpirun.lsf -cmdfile
mycmdfile
```

where `mycmdfile` contains:

```
task0
task1
task2
```

The order of the tasks in the task geometry specification must match the order of tasks in `mycmdfile`:

```
setenv LSB_PJL_TASK_GEOMETRY "{(0,1)(2)}"
```

If the order of tasks in `mycmdfile` changes, you must change the task geometry specification accordingly.

For example, if `mycmdfile` contains:

```
task0
task2
task1
```

the task geometry must be changed to:

```
setenv LSB_PJL_TASK_GEOMETRY "{(0,2)(1)}"
```

Enforcing Resource Usage Limits for Parallel Tasks

A typical Platform LSF parallel job launches its tasks across multiple hosts. By default you can enforce limits on the total resources used by all the tasks in the job. Because PAM only reports the sum of parallel task resource usage, LSF does not enforce resource usage limits on individual tasks in a parallel job.

For example, resource usage limits cannot control allocated memory of a single task of a parallel job to prevent it from allocating memory and bringing down the entire system. For some jobs, the total resource usage may be exceed a configured resource usage limit even if no single task does, and the job is terminated when it does not need to be.

Attempting to limit individual tasks by setting a system-level swap hard limit (RLIMIT_AS) in the system limit configuration file (`/etc/security/limits.conf`) is not satisfactory, because it only prevents tasks running on that host from allocating more memory than they should; other tasks in the job can continue to run, with unpredictable results.

By default, custom job controls (JOB_CONTROL in `lsb.queues`) apply only to the entire job, not individual parallel tasks.

Enabling resource usage limit enforcement for parallel tasks

Use the LSF_HPC_EXTENSIONS options TASK_SWAPLIMIT and TASK_MEMLIMIT in `lsf.conf` to enable resource usage limit enforcement and job control for parallel tasks. When TASK_SWAPLIMIT or TASK_MEMLIMIT is set in LSF_HPC_EXTENSIONS, LSF terminates the entire parallel job if any single task exceeds the limit setting for memory and swap limits.

Other resource usage limits (CPU limit, process limit, run limit, and so on) continue to be enforced for the entire job, not for individual tasks.

For more information For detailed information about resource usage limits in LSF, see the “Runtime Resource Usage Limits” chapter in *Administering Platform LSF*.

Assumptions and behavior

- To enforce resource usage limits by parallel task, you must use the LSF generic PJL framework (PAM/TS) to launch your parallel jobs.
- This feature only affects parallel jobs monitored by PAM. It has no effect on other LSF jobs.
- LSF_HPC_EXTENSIONS=TASK_SWAPLIMIT overrides the default behavior of swap limits (`bsub -v`, `bmod -v`, or SWAPLIMIT in `lsb.queues`).
- LSF_HPC_EXTENSIONS=TASK_MEMLIMIT overrides the default behavior of memory limits (`bsub -M`, `bmod -M`, or MEMLIMIT in `lsb.queues`).
- LSF_HPC_EXTENSIONS=TASK_MEMLIMIT overrides LSB_MEMLIMIT_ENFORCE=Y or LSB_JOB_MEMLIMIT=Y in `lsf.conf`
- When a parallel job is terminated because of task limit enforcement, LSF sets a value in the LSB_JOBEXIT_INFO environment variable for any post-execution programs:
  - LSB_JOBEXIT_INFO=SIGNAL -29 SIG_TERM_SWAPLIMIT
  - LSB_JOBEXIT_INFO=SIGNAL -25 SIG_TERM_MEMLIMIT

- When a parallel job is terminated because of task limit enforcement, LSF logs the job termination reason in `lsb.acct` file:
  - TERM_SWAP for swap limit
  - TERM_MEMLIMIT for memory limit

  and `bacct` displays the termination reason.

Example Integration: LAM/MPI

The script `lammpirun_wrapper` is the PJL wrapper. Use either "Integration Method 1" on page 37 or "Integration Method 2" on page 39 to call this script:

`pam [`*`other_pam_options`*`] -g `*`num_args`*` lammpirun_wrapper `*`job`*` [`*`job_options`*`]`

`pam [`*`other_pam_options`*`] -g lammpirun_wrapper `*`job`*` [`*`job_options`*`]`

Example script

```
#!/bin/sh
#
# ------------------------------------------------------
# Source the LSF environment. Optional.
# ------------------------------------------------------
. ${LSF_ENVDIR}/lsf.conf

# ------------------------------------------------------
# Set up the variable LSF_TS representing the TaskStarter.
# ------------------------------------------------------
LSF_TS="$LSF_BINDIR/TaskStarter"

# ------------------------------------------------------------------------
# Define the function to handle external signals:
# - display the signal received and the shutdown action to the user
# - log the signal received and the daemon shutdown action
# - exit gracefully by shutting down the daemon
# - set the exit code to 1
# -------------------------------------------------------------------------
#
lammpirun_exit()
{
   trap '' 1 2 3 15
   echo "Signal Received, Terminating the job<${TMP_JOBID}> and run lamhalt
..."
   echo "Signal Received, Terminating the job<${TMP_JOBID}> and run lamhalt
..." >>$LOGFILE
   $LAMHALT_CMD >>$LOGFILE 2>&1
   exit 1
} #lammpirun_exit

#-----------------------------------
# Name: who_am_i
# Synopsis: who_am_i
# Environment Variables:
# Description:
#       It returns the name of the current user.
# Return Value:
#       User name.
#-----------------------------------
who_am_i()
{
if  [ `uname` = ConvexOS ] ; then
```

```
    _my_name=`whoami | sed -e "s/[      ]//g"`
else
    _my_name=`id | sed -e 's/[^(]*(\([^)]*\)).*/\1/' | sed -e "s/[      ]//g"`
fi

echo $_my_name
} # who_am_i

#
#  -----------------------------------------------------
# Set up the script's log file:
# - create and set the variable LOGDIR to represent the log file directory
# - fill in your own choice of directory LOGDIR
# - the log directory you choose must be accessible by the user from all hosts
# - create a log file with a unique name, based on the job ID
# - if the log directory is not specified, the log file is /dev/null
# - the first entry logs the file creation date and file name
# - we create and set a second variable DISPLAY_JOBID to format the job
#   ID properly for writing to the log file
#  ----------------------------------------------------
#
#
# Please specify your own LOGDIR,
# Your LOGDIR must be accessible by the user from all hosts.
#
LOGDIR=""

TMP_JOBID=""
if [ -z "$LSB_JOBINDEX" -o "$LSB_JOBINDEX" = "0" ]; then
    TMP_JOBID="$LSB_JOBID"
    DISPLAY_JOBID="$LSB_JOBID"
else
    TMP_JOBID="$LSB_JOBID"_"$LSB_JOBINDEX"
    DISPLAY_JOBID="$LSB_JOBID[$LSB_JOBINDEX]"
fi

if [ -z "$LOGDIR" ]; then
    LOGFILE="/dev/null"
else
    LOGFILE="${LOGDIR}/lammpirun_wrapper.job${TMP_JOBID}.log"
fi



#
# -----------------------------------------------------
# Create and set variables to represent the commands used in the script:
#  - to modify this script to use different commands, edit this section
# ----------------------------------------------------
#
TPING_CMD="tping"
LAMMPIRUN_CMD="mpirun"
LAMBOOT_CMD="lamboot"
```

```
LAMHALT_CMD="lamhalt"

#
# -----------------------------------------------------
# Define an exit value to rerun the script if it fails
# - create and set the variable EXIT_VALUE to represent the requeue exit value
# - we assume you have enabled job requeue in LSF
# - we assume 66 is one of the job requeue values you specified in LSF
# ----------------------------------------------------
#
# EXIT_VALUE should not be set to 0
EXIT_VALUE="66"

#
# -----------------------------------------------------
# Write the first entry to the script's log file
# - date of creationg
# - name of log file
# ----------------------------------------------------
#
my_name=`who_am_i`
echo "`date` $my_name" >>$LOGFILE

# ----------------------------------------------------
# Use the signal handling function to handle specific external signals.
# ----------------------------------------------------
#
trap lammpirun_exit 1 2 3 15

#
# -----------------------------------------------------
# Set up a hosts file in the specific format required by LAM MPI:
# - remove any old hosts file
# - create a new hosts file with a unique name using the LSF job ID
# - write a comment at the start of the hosts file
# - if the hosts file was not created properly, display an error to
#   the user and exit
# - define the variables HOST, NUM_PROC, FLAG, and TOTAL_CPUS to
#   help with parsing the host information
# - LSF's selected hosts are described in LSB_MCPU_HOSTS environment variable
# - parse LSB_MCPU_HOSTS into the components
# - write the new hosts file using this information
# - write a comment at the end of the hosts file
# - log the contents of the new hosts file to the script log file
# ----------------------------------------------------
#
LAMHOST_FILE=".lsf_${TMP_JOBID}_lammpi.hosts"

if [ -d "$HOME" ]; then
    LAMHOST_FILE="$HOME/$LAMHOST_FILE"
fi
```

```
#
#
# start a new host file from scratch
rm -f $LAMHOST_FILE
echo "# LAMMPI host file created by LSF on `date`" >> $LAMHOST_FILE

# check if we were able to start writing the conf file
if [ -f $LAMHOST_FILE ]; then
    :
else
    echo "$0: can't create $LAMHOST_FILE"
    exit 1
fi

HOST=""
NUM_PROC=""
FLAG=""
TOTAL_CPUS=0
for TOKEN in $LSB_MCPU_HOSTS
do
    if [ -z "$FLAG" ]; then
        HOST="$TOKEN"
        FLAG="0"
    else
        NUM_PROC="$TOKEN"
        TOTAL_CPUS=`expr $TOTAL_CPUS + $NUM_PROC`
        FLAG="1"
    fi

    if [ "$FLAG" = "1" ]; then
        _x=0
        while [ $_x -lt $NUM_PROC ]
        do
            echo "$HOST" >>$LAMHOST_FILE
            _x=`expr $_x + 1`
        done

        # get ready for the next host
        FLAG=""
        HOST=""
        NUM_PROC=""
    fi
done

# last thing added to LAMHOST_FILE
echo "# end of LAMHOST file" >> $LAMHOST_FILE

echo "Your lamboot hostfile looks like:" >> $LOGFILE
cat $LAMHOST_FILE >> $LOGFILE
```

```
#  -----------------------------------------------------
#  Process the command line:
# - extract [mpiopts] from the command line
# - extract jobname [jobopts] from the command line
#  -----------------------------------------------------
ARG0=`$LAMMPIRUN_CMD -h 2>&1 | \
      egrep '^[[:space:]]+-[[:alpha:][:digit:]-]+[[:space:]][[:space:]]' | \
      awk '{printf "%s ", $1}'`
# get -ton,t and -w / nw options
TMPARG=`$LAMMPIRUN_CMD -h 2>&1 | \
      egrep '^[[:space:]]+-[[:alpha:]_-]+[[:space:]]*(,|/)[[:space:]]-
[[:alpha:]]*' |
      sed 's/,/ /'| sed 's/\// /' | \
      awk '{printf "%s %s ", $1, $2}'`
ARG0="$ARG0 $TMPARG"

ARG1=`$LAMMPIRUN_CMD -h 2>&1 | \
      egrep '^[[:space:]]+-[[:alpha:]_-
]+[[:space:]]+<[[:alpha:][:space:]_]+>[[:space:]]' | \
      awk '{printf "%s ", $1}'`

while [ $# -gt 0 ]
do
     MPIRunOpt="0"

     #single-valued options
     for option in $ARG1
     do
         if [ "$option" = "$1" ]; then
            MPIRunOpt="1"
      case "$1" in
          -np|-c)
         shift
         shift
         ;;
          *)
         LAMMPI_OPTS="$LAMMPI_OPTS $1" #get option name
         shift
         LAMMPI_OPTS="$LAMMPI_OPTS $1" #get option value
         shift
         ;;
      esac
            break
         fi
     done

     if [ $MPIRunOpt = "1" ]; then
        :
     else
        #Non-valued options
        for option in $ARG0
        do
```

```
            if [ $option = "$1" ]; then
               MPIRunOpt="1"
         case "$1" in
         -v)
             shift
        ;;
         *)
        LAMMPI_OPTS="$LAMMPI_OPTS $1"
        shift
        ;;
         esac
         break
            fi
        done
     fi

     if [ $MPIRunOpt = "1" ]; then
        :
     else
        JOB_CMDLN="$*"
        break
     fi

done

# -------------------------------------------------------------------------------
# Set up the CMD_LINE variable representing the integrated section of the
# command line:
# - LSF_TS, script variable representing the TaskStarter binary.
#   TaskStarter must start each and every job task process.
# - LSF_TS_OPTIONS, LSF environment variable containing all necessary
#   information for TaskStarter to callback to LSF's Parallel Application
#   Manager.
# - JOB_CMDLN, script variable containing the job and job options
#--------------------------------------------------------------------------------
if [ -z "$LSF_TS_OPTIONS" ]
then
    echo CMD_LINE="$JOB_CMDLN" >> $LOGFILE
    CMD_LINE="$JOB_CMDLN "
else
    echo CMD_LINE="$LSF_TS $LSF_TS_OPTIONS $JOB_CMDLN" >> $LOGFILE
    CMD_LINE="$LSF_TS $LSF_TS_OPTIONS $JOB_CMDLN "
fi

#
# -----------------------------------------------------
# Pre-execution steps required by LAMMPI:
# - define the variable LAM_MPI_SOCKET_SUFFIX using the LSF
#   job ID and export it
# - run lamboot command and log the action
# - append the hosts file to the script log file
# - run tping command and log the action and output
# - capture the result of tping and test for success before proceeding
# - exits with the "requeue" exit value if pre-execution setup failed
# ----------------------------------------------------
#

LAM_MPI_SOCKET_SUFFIX="${LSB_JOBID}_${LSB_JOBINDEX}"
export LAM_MPI_SOCKET_SUFFIX

echo $LAMBOOT_CMD $LAMHOST_FILE >>$LOGFILE
$LAMBOOT_CMD $LAMHOST_FILE >>$LOGFILE 2>&1
echo $TPING_CMD h -c 1 >>$LOGFILE
$TPING_CMD N -c 1 >>$LOGFILE 2>&1
EXIT_VALUE="$?"

if [ "$EXIT_VALUE" = "0" ]; then
#
# -----------------------------------------------------
# Run the parallel job launcher:
# - log  the action
# - trap the exit value
# ----------------------------------------------------
#
    #call mpirun -np # a.out
    echo "Your command line looks like:" >> $LOGFILE
    echo $LAMMPIRUN_CMD $LAMMPI_OPTS -v C $CMD_LINE >> $LOGFILE
    $LAMMPIRUN_CMD $LAMMPI_OPTS -v C $CMD_LINE
    EXIT_VALUE=$?
#
# -----------------------------------------------------
#  Post-execution steps required by LAMMPI:
# - run lamhalt
# - log the action
# ----------------------------------------------------
#
    echo $LAMHALT_CMD >>$LOGFILE
    $LAMHALT_CMD >>$LOGFILE 2>&1
fi

#
# -----------------------------------------------------
# Clean up after running this script:
# - delete the hosts file we created
# - log the end of the job
# - log the exit value of the job
# ----------------------------------------------------
#
# cleanup temp and conf file then exit
rm -f $LAMHOST_FILE
echo "Job<${DISPLAY_JOBID}> exits with exit value $EXIT_VALUE." >>$LOGFILE 2>&1
# To support multiple jobs inside one job script
# Sleep one sec to allow next lamd start up, otherwise tping will return error
sleep 1
```

```
exit $EXIT_VALUE
#
# -----------------------------------------------------
# End the script.
# ----------------------------------------------------
#
```

Tips for Writing PJL Wrapper Scripts

A wrapper script is often used to call the PJL. We assume the PJL is not integrated with LSF, so if PAM was to start the PJL directly, the PJL would not automatically use the hosts that LSF selected, or allow LSF to collect resource information.

The wrapper script can set up the environment before starting the actual job.

Script log file The script should create and use its own log file, for troubleshooting purposes. For example, it should log a message each time it runs a command, and it should also log the result of the command. The first entry might record the successful creation of the log file itself.

Command aliases Set up aliases for the commands used in the script, and identify the full path to the command. Use the alias throughout the script, instead of calling the command directly. This makes it simple to change the path or the command at a later time, by editing just one line.

Signal handling If the script is interrupted or terminated before it finishes, it should exit gracefully and undo any work it started. This might include closing files it was using, removing files it created, shutting down daemons it started, and recording the signal event in the log file for troubleshooting purposes.

Requeue exit value In LSF, job requeue is an optional feature that depends on the job's exit value. PAM exits with the same exit value as PJL, or its wrapper script. Some or all errors in the script can specify a special exit value that causes LSF to requeue the job.

Redirect screen output Use `/dev/null` to redirect any screen output to a null file.

Access LSF configuration Set LSF_ENVDIR and source the `lsf.conf` file. This gives you access to LSF configuration settings.

Construct host file The hosts LSF has selected to run the job are described by the environment variable LSB_MCPU_HOSTS. This environment variable specifies a list, in quotes, consisting of one or more host names paired with the number of processors to use on that host:

"*host_name number_processors host_name number_processors* ..."

Parse this variable into the components and create a host file in the specific format required by the vendor PJL. In this way, the hosts LSF has chosen are passed to the PJL.

Vendor-specific pre-exec Depending on the vendor, the PJL may require some special pre-execution work, such as initializing environment variables or starting daemons. You should log each pre-exec task in the log file, and also check the result and handle errors if a required task failed.

Double-check external resource If an external resource is used to identify MPI-enabled hosts, LSF has selected hosts based on the availability of that resource. However, there is some time delay between LSF scheduling the job and the script starting the PJL. It's a good idea to make the script verify that required resources are still available on the selected hosts (and exit if the hosts are no longer able to execute the parallel job). Do this immediately before starting the PJL.

PJL The most important function of the wrapper script is to start the PJL and have it execute the parallel job on the hosts selected by LSF. Normally, you use a version of the `mpirun` command.

Vendor-specific post-exec Depending on the vendor, the PJL may require some special post-execution work, such as stopping daemons. You should log each post-exec task in the log file, and also check the result and handle errors if any task failed.

Script post-exec The script should exit gracefully. This might include closing files it used, removing files it created, shutting down daemons it started, and recording each action in the log file for troubleshooting purposes.

Other Integration Options

Once the PJL integration is successful, you might be interested in the following LSF features.

For more information about these features, see the LSF documentation.

Using a job starter

A job starter is a wrapper script that can set up the environment before starting the actual job.

Using external resources

You may need to identify MPI-enabled hosts

If all hosts in the LSF cluster can be used run the parallel jobs, with no restrictions, you don't need to differentiate between regular hosts and MPI-enabled hosts.

You can use an external resource to identify suitable hosts for running your parallel jobs.

To identify MPI-enabled hosts, you can configure a static Boolean resource in LSF.

For some integrations, to make sure the parallel jobs are sent to suitable hosts, you must track a dynamic resource (such as free ports). You can use an LSF ELIM to report the availability of these. See *Administering Platform LSF* for information about writing ELIMs.

Named hosts

- If you create a dedicated LSF queue to manage the parallel jobs, make sure the queue's host list includes only MPI-enabled hosts.
- The `bsub` option `-m host_name` allows you to specify hosts by name. All the hosts you name are used to run the parallel job.
- The `bsub` option `-R res_req` allows you to specify any LSF resource requirements, including a list of hosts; in this case, you specify that the hosts selected must have one of the names in your host list.

Using esub

An `esub` program can contain logic that modifies a job before submitting it to LSF. The `esub` can be used to simplify the user input. An example is the LAM/MPI integration in the Platform open source FTP directory.

CHAPTER

3

Using Platform LSF with HP-UX Processor Sets

LSF makes use of HP-UX processor sets (psets) to create an efficient execution environment that allows a mix of users and jobs to coexist in the HP Superdome cell-based architecture.

Contents

About HP-UX Psets

HP-UX processor sets (*psets*) are available as an optional software product for HP-UX 11i Superdome multiprocessor systems. A pset is a set of active processors group for the exclusive access of the application assigned to the set. A pset manages processor resources among applications and users.

The operating system restricts applications to run only on the processors in their assigned psets. Processes bound to a pset can only run on the CPUs belonging to that pset, so applications assigned to different psets do not contend for processor resources.

A newly created pset initially has no processors assigned to it.

Dynamic application binding

Each running application in the system is bound to some pset, which defines the processors that the application can run on.

Scheduling allocation domain

A pset defines a scheduling allocation domain that restricts applications to run only on the processors in its assigned pset.

System default pset

At system startup, the HP-UX system is automatically configured with one system default pset to which all enabled processors are assigned. Processor 0 is always assigned to the default pset. All users in the system can access the default pset.

For more information

See the HP-UX 11i system administration documentation for information about defining and managing psets.

How LSF uses psets

Processor isolation

On HP-UX 11i Superdome multiprocessor systems, psets can be created and deallocated dynamically out of available machine resources. The pset provides processor isolation, so that a job requiring a specific number of CPUs only run on those CPUs.

Processor distance

Processor distance is a value used to measure how fast the process running on one processor access local memory of another processor. The bigger the value is, the slower memory access is. For example, the processor distance of two processes within one cell is less than that of two processes between cells.

When creating a pset for the job, LSF uses a best-fit algorithm for pset allocation to choose processors as close as possible to each other. LSF attempts to choose the set of processors with the smallest processor distance.

Pset creation and deallocation

LSF makes use of HP-UX processor sets (psets) to create an efficient execution environment that allows a mix of users and jobs to coexist in the HP Superdome cell-based architecture.

When a job is submitted, LSF:

- Chooses the best CPUs based on job resource requirements (number of processors requested and pset topology)
- Creates a pset for the job. The operating system assigns a unique pset identifier (pset ID) to it.

  LSF has no control over the pset ID assigned to a newly created pset.

- Places the job processes in the pset when the job starts running

After the job finishes, LSF destroys the pset. If no host meets the CPU requirements, the job remains pending until processors become available to allocate the pset.

CPU 0 in the default pset 0 is always considered last for a job, and cannot be taken out of pset 0, since all system processes are running on it. LSF cannot create a pset with CPU 0; it only uses the default pset if it cannot create a pset without CPU 0.

LSF topology adapter for psets (RLA)

RLA runs on each HP-UX11i host. It is started and monitored by `sbatchd`. RLA provides services for external clients, including pset scheduler plugin and `sbatchd` to:

- Allocate and deallocate job psets
- Get the job pset ID
- Suspend a pset when job is suspended, and reassign all CPUs within pset back to pset 0
- Resume a pset, and before a job is resumed, assign all original CPUs back to the job pset
- Get pset topology information, cells, CPUs, and processor distance between cells.
- Get updated free CPU map
- Get job resource map

RLA maintains a status file in the directory defined by LSB_RLA_WORKDIR in `lsf.conf`, which keeps track of job pset allocation information. When RLA starts, it reads the status file and recovers the current status.

Assumptions and limitations

Account mapping

User-level account and system account mapping are not supported. If a user account does not exist on the remote host, LSF cannot create a pset for it.

Resizable jobs

Jobs running in a pset cannot be resized.

Resource reservation

By default, job start time is not accurately predicted for pset jobs with topology options, so the forecast start time shown by `bjobs -l` is optimistic. LSF may incorrectly indicate that the job can start at a certain time, when it actually cannot start until some time after the indicated time.

For a more accuration start-time estimate, you should configure time-based slot reservation. With time-based reservation, a set of pending jobs will get future allocation and estimated start time.

See *Administering Platform LSF* for more information about time-based slot reservation.

Chunk jobs

Jobs submitted to a chunk job queue are not chunked together, but run outside of a pset as a normal LSF job.

Preemption

- When LSF selects pset jobs to preempt, specialized preemption preferences, such as MINI_JOB and LEAST_RUN_TIME in the PREEMPT_FOR parameter in `lsb.params`, and others are ignored when slot preemption is required.
- Preemptable queue preference is not supported.

Suspending and resuming jobs

When a job is suspended with `bstop`, all CPUs in the pset are released and reassigned back to the default pset (pset 0). Before resuming the job LSF reallocates the pset and rebinds all job processes to the job pset.

Pre-execution and post-execution

Job pre-execution programs run within the job pset, since they are part of the job. Post-execution programs run outside of the job pset.

Configuring LSF with HP-UX Psets

Automatic configuration at installation

lsb.modules During installation, `lsfinstall` adds the `schmod_pset` external scheduler plugin module name to the PluginModule section of `lsb.modules`:

```
Begin PluginModule
SCH_PLUGIN                  RB_PLUGIN             SCH_DISABLE_PHASES
schmod_default                 ()                        ()
schmod_fcfs                    ()                        ()
schmod_fairshare               ()                        ()
schmod_limit                   ()                        ()
schmod_preemption              ()                        ()
...
schmod_pset                    ()                        ()
End PluginModule
```

The schmod_pset plugin name must be configured after the standard LSF plugin names in the PluginModule list.

See the *Platform LSF Configuration Reference* for more information about `lsb.modules`.

lsf.conf During installation, `lsfinstall` sets the following parameters in `lsf.conf`:

- On HP-UX hosts, sets the full path to the HP vendor MPI library `libmpirm.sl`.

  `LSF_VPLUGIN="/opt/mpi/lib/pa1.1/libmpirm.sl"`
- On Linux hosts running Platform MPI, sets the full path to the HP vendor MPI library `libmpirm.so`.

  For example, if Platform MPI is installed in `/opt/hpmpi`:

  `LSF_VPLUGIN="/opt/hpmpi/lib/linux_ia32/libmpirm.so"`
- LSF_ENABLE_EXTSCHEDULER=Y

  LSF uses an external scheduler for pset allocation.
- LSB_RLA_PORT=*port_number*

  Where *port_number* is the TCP port used for communication between the LSF topology adapter (RLA) and `sbatchd`.

  The default port number is 6883.
- LSB_SHORT_HOSTLIST=1

  Displays an abbreviated list of hosts in `bjobs` and `bhist` for a parallel job where multiple processes of a job are running on a host. Multiple processes are displayed in the following format:

  `processes``*hostA`

lsf.shared During installation, the Boolean resource `pset` is defined in `lsf.shared`:

```
Begin Resource
RESOURCENAME     TYPE      INTERVAL  INCREASING   DESCRIPTION
...
pset             Boolean   ()        ()           (PSET)
...
End Resource
```

You should add the pset resource name under the RESOURCES column of the Host section of `lsf.cluster.`*`cluster_name`*. Hosts without the pset resource specified are not considered for scheduling pset jobs.

lsb.hosts

For each pset host, `lsfinstall` enables "!" in the MXJ column of the HOSTS section of `lsb.hosts` for the HPPA11 host type.

For example:

```
Begin Host
HOST_NAME MXJ    r1m      pg     ls     tmp   DISPATCH_WINDOW   # Keywords
#hostA     () 3.5/4.5    15/    12/15   0       ()              # Example
default    !    ()       ()     ()      ()      ()
HPPA11     !    ()       ()     ()      ()      ()              #pset host
End Host
```

lsf.cluster.cluster_name

For each pset host, `hostsetup` adds the `pset` Boolean resource to the HOST section of `lsf.cluster.`*`cluster_name`*.

Configuring default and mandatory pset options

Use the DEFAULT_EXTSCHED and MANDATORY_EXTSCHED queue paramters in `lsb.queues` to configure default and mandatory pset options.

DEFAULT_EXTSCHED=PSET[*topology*]

where *topology* is:

`[`**`CELLS=`**`*num_cells*` | **`PTILE=`**`*cpus_per_cell*] [`**`;CELL_LIST=`***`cell_list`*`]`

Specifies default pset topology scheduling options for the queue.

`-extsched` options on the `bsub` command override any conflicting queue-level options set by DEFAULT_EXTSCHED.

For example, if the queue specifies:

```
DEFAULT_EXTSCHED=PSET[PTILE=2]
```

and a job is submitted with no topology requirements requesting 6 CPUs (`bsub -n 6`), a pset is allocated using 3 cells with 2 CPUs in each cell.

If the job is submitted:

```
bsub -n 6 -ext "PSET[PTILE=3]" myjob
```

The pset option in the command overrides the DEFAULT_EXTSCHED, so a pset is allocated using 2 cells with 3 CPUs in each cell.

MANDATORY_EXTSCHED=PSET[*topology*]

Specifies mandatory pset topology scheduling options for the queue.

MANDATORY_EXTSCHED options override any conflicting job-level options set by `-extsched` options on the `bsub` command.

For example, if the queue specifies:

```
MANDATORY_EXTSCHED=PSET[CELLS=2]
```

and a job is submitted with no topology requirements requesting 6 CPUs (`bsub n 6`), a pset is allocated using 2 cells with 3 CPUs in each cell.

If the job is submitted:

```
bsub -n 6 -ext "PSET[CELLS=3]" myjob
```

MANDATORY_EXTSCHED overrides the pset option in the command, so a pset is allocated using 2 cells with 3 CPUs in each cell.

Use the CELL_LIST option in MANDATORY_EXTSCHED to restrict the cells available for allocation to pset jobs. For example, if the queue specifies:

```
MANDATORY_EXTSCHED=PSET[CELL_LIST=1-7]
```

job psets can only use cells 1 to 7; cell 0 is not used for pset jobs.

Using LSF with HP-UX Psets

Specifying pset topology options

To specify processor topology scheduling policy options for pset jobs, use:

- The `-extsched` option of `bsub`.

  You can abbreviate the -extsched option to -ext.

- DEFAULT_EXTSCHED or MANDATORY_EXTSCHED, or both, in the queue definition (`lsb.queues`).

If LSB_PSET_BIND_DEFAULT is set in `lsf.conf`, and no pset options are specified for the job, LSF binds the job to the default pset 0. If LSB_PSET_BIND_DEFAULT is not set, LSF must still attach the job to a pset, and so binds the job to the same pset being used by the LSF daemons.

For more information about job operations, see *Administering Platform LSF*.

For more information about `bsub`, see the *Platform LSF Command Reference*.

Syntax

`-ext``[`**`sched`**`]` **`"PSET[`***`topology`***`]"`**

where *topology* is:

`[`**`CELLS=`***`num_cells`* `|` **`PTILE=`***`cpus_per_cell`*`][`**`;CELL_LIST=`***`cell_list`*`]`

- **`CELLS`**=*num_cells*

  Defines the exact number of cells the LSF job requires. For example, if `CELLS=4`, and the job requests 6 processors (`bsub -n 6`) on a 4-CPU/cell HP Superdome system with no other jobs running, the pset uses 4 cells, and the allocation is 2, 2, 1, 1 on each cell. If LSF cannot satisfy the CELLS request, the job remains pending.

  If CELLS is greater than 1 and you specify minimum and maximum processors (for example, `bsub -n 2,8`), only the minimum is used.

  To enforce job processes to run within one cell, use `"PSET[CELLS=1]"`.

- **`PTILE=`***cpus_per_cell*

  Defines the exact number of processors allocated on each cell up to the maximum for the system. For example, if `PTILE=2`, and the job requests 6 processors (`bsub -n 6`) on a 4-CPU/cell HP Superdome system with no other jobs running, the pset spreads across 3 cells instead of 2 cells, and the allocation is 2, 2, 2 on each cell.

  The value for `-n` and the PTILE value must be divisible by the same number. If LSF cannot satisfy the PTILE request, the job remains pending. For example:

  ```
  bsub -n 5 -ext "PSET[PTILE=3] ...
  ```

  is incorrect.

  To enforce jobs to run on the cells that no others jobs are running on, use `"PSET[PTILE=4]"` on 4 CPU/cell system.

You can specify either one CELLS or one PTILE option in the same PSET[] option, not both.

- **`CELL_LIST=`***min_cell_ID*[**`-`***max_cell_ID*][**`,`** *min_cell_ID*[**`-`***max_cell_ID*] ...]

The LSF job uses only cells specified in the specified cell list to allocate the pset. For example, if `CELL_LIST=1,2`, and the job requests 8 processors (`bsub -n 8`) on a 4-CPU/cell HP Superdome system with no other jobs running, the pset uses cells 1 and 2, and the allocation is 4 CPUs on each cell. If LSF cannot satisfy the CELL_LIST request, the job remains pending.

If CELL_LIST is defined in DEFAULT_EXTSCHED in the queue, and you do not want to specify a cell list for your job, use the CELL_LIST keyword with no value. For example, if `DEFAULT_EXTSCHED=PSET[CELL_LIST=1-8]`, and you do not want to specify a cell list, use `-ext "PSET[CELL_LIST=]"`.

Priority of topology scheduling options

The options set by `-extsched` can be combined with the queue-level MANDATORY_EXTSCHED or DEFAULT_EXTSCHED parameters. If `-extsched` and MANDATORY_EXTSCHED set the same option, the MANDATORY_EXTSCHED setting is used. If `-extsched` and DEFAULT_EXTSCHED set the same options, the `-extsched` setting is used.

topology scheduling options are applied in the following priority order of level from highest to lowest:

1. Queue-level MANDATORY_EXTSCHED options override ...
2. Job level `-ext` options, which override ...
3. Queue-level DEFAULT_EXTSCHED options

For example, if the queue specifies:

```
DEFAULT_EXTSCHED=PSET[CELLS=2]
```

and the job is submitted with:

```
bsub -n 4 -ext "PSET[PTILE=1]" myjob
```

The pset option in the job submission overrides the DEFAULT_EXTSCHED, so the job will run in a pset allocated using 4 cells, honoring the job-level PTILE option.

If the queue specifies:

MANDATORY_EXTSCHED=PSET[CELLS=2]

and the job is submitted with:

```
bsub -n 4 -ext "PSET[PTILE=1]" myjob
```

The job will run on 2 cells honoring the cells option in MANDATORY_EXTSCHED.

Partitioning the system for specific jobs (CELL_LIST)

Use the `bsub -ext "PSET[CELL_LIST=`*`cell_list`*`]"` option to partition a large Superdome machine. Instead of allocating CPUs from the entire machine, LSF creates a pset containing only the cells specified in the cell list.

Non-existent cells are ignored during scheduling, but the job can be dispatched as long as enough cells are available to satisfy the job requirements. For example, in a cluster with both 32-CPU and 64-CPU machines and a cell list specification CELL_LIST=1-15, jobs can use cells 1-7 on the 32-CPU machine, and cells 1-15 on the 64-CPU machine.

CELL_LIST and CELLS

You can use CELL_LIST with the PSET[CELLS=*num_cells*] option. The number of requested cells in the cell list must be less than or equal to the number of cells in the CELLS option; otherwise, the job remains pending.

CELL_LIST and PTILE You can use CELL_LIST with the PSET[PTILE=*cpus_per_cell*] option. The PTILE option allows the job pset to spread across several cells. The number of required cells equals the number of requested processors divided by the PTILE value. The resulting number of cells must be less than or equal to the number of cells in the cell list; otherwise, the job remains pending.

For example, the following is a correct specification:

```
bsub -n 8 -ext "PSET[PTILE=2;CELL_LIST=1-4]" myjob
```

The job requests 8 CPUs spread over 4 cells (8/2=4), which is equal to the 4 cells requested in the CELL_LIST option.

Viewing pset allocations for jobs

bjobs -l After a pset job starts to run, use `bjobs -l` to display the job pset ID. For example, if LSF creates pset 23 on `hostA` for job 329, `bjobs` shows:

```
bjobs -l 329

Job <329>, User <user1>, Project <default>, Status <RUN>, Queue <normal>, Ext
                     sched <PSET[]>, Command <sleep 60>
Thu Jan 22 12:04:31 2010: Submitted from host <hostA>, CWD <$HOME>, 2
Processors
                     Requested;
Thu Jan 22 12:04:38 2010: Started on 2 Hosts/Processors <2*hostA>, Execution
Home
                     </home/user1>, Execution CWD </home/user1>;
Thu Jan 22 12:04:38 2010: psetid=hostA:23;
Thu Jan 22 12:04:39 2010: Resource usage collected.
                     MEM: 1 Mbytes;  SWAP: 2 Mbytes;  NTHREAD: 1
                     PGID: 18440;  PIDs: 18440


 SCHEDULING PARAMETERS:
           r15s   r1m  r15m   ut      pg    io   ls    it    tmp    swp    mem
 loadSched   -     -     -     -       -     -    -     -     -      -      -
 loadStop    -     -     -     -       -     -    -     -     -      -      -

 EXTERNAL MESSAGES:
 MSG_ID FROM       POST_TIME      MESSAGE                          ATTACHMENT
 0        -            -             -                                   -
 1      user1      Jan 22 12:04   PSET[]
```

The pset ID string for bjobs does not change after the job is dispatched.

bhist Use `bhist` to display historical information about pset jobs:

```
bhist -l 329

Job <329>, User <user1>, Project <default>, Extsched <PSET[]>, Command <sleep
                     60>
Thu Jan 22 12:04:31 2010: Submitted from host <hostA>, to Queue <normal>, CWD
<$H
                     OME>, 2 Processors Requested;
```

```
Thu Jan 22 12:04:38 2010: Dispatched to 2 Hosts/Processors <2*hostA>;
Thu Jan 22 12:04:38 2010: psetid=hostA:23;
Thu Jan 22 12:04:39 2010: Starting (Pid 18440);
Thu Jan 22 12:04:39 2010: Running with execution home </home/user1>, Execution
CWD
                     </home/user1>, Execution Pid <18440>;
Thu Jan 22 12:05:39 2010: Done successfully. The CPU time used is 0.1 seconds;
Thu Jan 22 12:05:40 2010: Post job process done successfully;

Summary of time in seconds spent in various states by  Thu Jan 22 12:05:40
  PEND     PSUSP    RUN      USUSP    SSUSP    UNKWN    TOTAL
  7        0        61       0        0        0        68
```

bacct Use bacct to display accounting information about pset jobs:

```
bacct -l 329
Accounting information about jobs that are:
  - submitted by all users.
  - accounted on all projects.
  - completed normally or exited
  - executed on all hosts.
  - submitted to all queues.
  - accounted on all service classes.
------------------------------------------------------------------------------

Job <331>, User <user1>, Project <default>, Status <DONE>, Queue <normal>, Co
                     mmand <sleep 60>
Thu Jan 22 18:23:14 2010: Submitted from host <hostA>, CWD <$HOME>;
Thu Jan 22 18:23:23 2010: Dispatched to <hostA>;
Thu Jan 22 18:23:23 2010: psetid=hostA:23;
Thu Jan 22 18:24:24 2010: Completed <done>.

Accounting information about this job:
     CPU_T     WAIT     TURNAROUND   STATUS     HOG_FACTOR    MEM    SWAP
      0.12        9             70     done         0.0017     1M      2M
------------------------------------------------------------------------------

SUMMARY:      ( time unit: second )
 Total number of done jobs:       1      Total number of exited jobs:     0
 Total CPU time consumed:       0.1      Average CPU time consumed:     0.1
 Maximum CPU time of a job:     0.1      Minimum CPU time of a job:     0.1
 Total wait time in queues:     9.0
 Average wait time in queue:    9.0
 Maximum wait time in queue:    9.0      Minimum wait time in queue:    9.0
 Average turnaround time:        70 (seconds/job)
 Maximum turnaround time:        70      Minimum turnaround time:        70
 Average hog factor of a job:  0.00 ( cpu time / turnaround time )
 Maximum hog factor of a job:  0.00      Minimum hog factor of a job:  0.00
```

Examples

The following examples assume a 4-CPU/cell HP Superdome system with no other jobs running:

- Submit a pset job without topology requirement:

  ```
  bsub -n 8 -ext "PSET[]" myjob
  ```

  A pset containing 8 cpus is created for the job. According to default scheduler policy, these 8 cpus will come from 2 cells on a single host.

- Submit a pset job specifying 1 CPU per cell:

  ```
  bsub -n 6 -ext "PSET[PTILE=1]" myjob
  ```

  A pset containing 6 processors is created for the job. The allocation uses 6 cells with 1 processor per cell.

- Submit a pset job specifying 4 cells:

  ```
  bsub -n 6 -ext "PSET[CELLS=4]" myjob
  ```

  A pset containing 6 processors is created for the job. The allocation uses 4 cells: 2 cells with 2 processors and 2 cells with 1 processor.

- Submit a pset job with a range of CPUs and 3 CPUs per cell:

  ```
  bsub -n 7,10 -ext "PSET[PTILE=3]" myjob
  ```

  A pset containing 9 processors is created for the job. The allocation uses 3 cells, with 3 CPUs each.

- Submit a pset job with a range of CPUs and 4 cells:

  ```
  bsub -n 7,10 -ext "PSET[CELLS=4]" myjob
  ```

  A pset containing 7 processors is created for the job. The allocation uses 4 cells, 3 cells with 2 CPUs and 1 cell with 1 CPU:

- Submit a pset job with a range of CPUs and 1 cell:

  ```
  bsub -n 2,4 -ext "PSET[CELLS=1]" myjob
  ```

  A pset containing 4 processors is created for the job. The allocation uses 1 cell with 4 CPUs.

- Submit a pset job requiring cells 1 and 2 with 4 CPUs per cell:

  ```
  bsub -n 8 -ext"PSET[PTILE=4;CELL_LIST=1,2]" myjob
  ```

  A pset containing 8 processors is created for the job. The allocation uses cells 1 and 2, each with 4 CPUs.

- Submit a pset job requiring a specific range of 6 cells:

  ```
  bsub -n 16 -ext "PSET[CELL_LIST=4-9]" myjob
  ```

  A pset containing 16 processors is created for the job. The allocation uses cells between 4 and 9.

- Submit a pset job requiring processors from two ranges of cells, separated by a comma:

  ```
  bsub -n 16 -ext "PSET[CELL_LIST=1-5,8-15]" myjob
  ```

  A pset containing 16 processors is created for the job. The allocation uses processors from cells 1 through 5 and cells 8 through 15.

CHAPTER

4

Using Platform LSF with IBM POE

Contents

Running IBM POE Jobs

The IBM Parallel Operating Environment (POE) interfaces with the Resource Manager to allow users to run parallel jobs requiring dedicated access to the high performance switch.

The LSF integration for IBM High-Performance Switch (HPS) systems provides support for submitting POE jobs from AIX hosts to run on IBM HPS hosts.

An IBM HPS system consists of multiple nodes running AIX. The system can be configured with a high-performance switch to allow high bandwidth and low latency communication between the nodes. The allocation of the switch to jobs as well as the division of nodes into pools is controlled by the HPS Resource Manager.

Run `chown` to change the owner of nrt_api to root, and then use `chmod` to set `setuid` bit (chmod u+s).

hpc_ibm queue for POE jobs

During installation, `lsfinstall` configures a queue in `lsb.queues` named `hpc_ibm` for running POE jobs. It defines requeue exit values to enable requeuing of POE jobs if some users submit jobs requiring exclusive access to the node.

The `poejob` script will exit with 133 if it is necessary to requeue the job. Other types of jobs should not be submitted to the same queue. Otherwise, they will get requeued if they happen to exit with 133.

```
Begin Queue
QUEUE_NAME    = hpc_ibm
PRIORITY      = 30
NICE          = 20
...
RES_REQ = select[ poe > 0 ]
REQUEUE_EXIT_VALUES = 133 134 135
...
DESCRIPTION   = This queue is to run POE jobs ONLY.
End Queue
```

Configuring LSF to run POE jobs

Ensure that the HPS node names are the same as their host names. That is, `st_status` should return the same names for the nodes that `lsload` returns.

To set up POE jobs

"1. Configure per-slot resource reservation (lsb.resources)".

"2. Optional. Enable exclusive mode (lsb.queues)".

"3. Optional. Define resource management pools (rmpool) and node locking queue threshold".

"4. Optional. Define system partitions (spname)".

"5. Allocate switch adapter specific resources".

"6. Optional. Tune PAM parameters".

"7. Reconfigure to apply the changes".

1. Configure per-slot resource reservation (lsb.resources)

To support the IBM HPS architecture, LSF must reserve resources based on job slots. During installation, `lsfinstall` configures the ReservationUsage section in `lsb.resources` to reserve HPS resources on a per-slot basis.

Resource usage defined in the ReservationUsage section overrides the cluster-wide RESOURCE_RESERVE_PER_SLOT parameter defined in `lsb.params` if it also exists.

```
Begin ReservationUsage
RESOURCE            METHOD
adapter_windows     PER_SLOT
ntbl_windows        PER_SLOT
csss                PER_SLOT
css0                PER_SLOT
End ReservationUsage
```

2. Optional. Enable exclusive mode (lsb.queues)

To support the MP_ADAPTER_USE and `-adapter_use` POE job options, you must enable the LSF exclusive mode for each queue. To enable exclusive mode, edit `lsb.queues` and set EXCLUSIVE=Y:

```
Begin Queue
...
EXCLUSIVE=Y
...
End Queue
```

3. Optional. Define resource management pools (rmpool) and node locking queue threshold

If you schedule jobs based on resource management pools, you must configure `rmpools` as a static resource in LSF. Resource management pools are collections of SP2 nodes that together contain all available SP2 nodes without any overlap.

For example, to configure 2 resource management pools, `p1` and `p2`, made up of 6 SP2 nodes (`sp2n1`, `sp2n1`, `sp2n3`, ..., `sp2n6`):

1 Edit `lsf.shared` and add an external resource called `pool`. For example:

```
Begin Resource
RESOURCENAME TYPE    INTERVAL INCREASING DESCRIPTION
...
pool         Numeric ()       ()         (sp2 resource mgmt
pool)
lock
Numeric 60       Y          (IBM SP Node lock status)
...
End Resource
```

`pool` represents the resource management pool the node is in, and and `lock` indicates whether the switch is locked.

2 Edit `lsf.cluster.`*`cluster_name`* and allocate the `pool` resource. For example:

```
Begin ResourceMap
RESOURCENAME  LOCATION
...
pool          (p1@[sp2n1 sp2n2 sp2n3] p2@[sp2n4 sp2n5
sp2n6])
...
End ResourceMap
```

3 Edit `lsb.queues` and a threshold for the `lock` index in the `hpc_ibm` queue:

```
Begin Queue
NAME=hpc_ibm
...
lock=0
...
End Queue
```

The scheduling threshold on the `lock` index prevents dispatching to nodes which are being used in exclusive mode by other jobs.

4. Optional. Define system partitions (spname)

If you schedule jobs based on system partition names, you must configure the static resource `spname`. System partitions are collections of HPS nodes that together contain all available HPS nodes without any overlap. For example, to configure two system partition names, `spp1` and `spp2`, made up of 6 SP2 nodes (`sp2n1`, `sp2n1`, `sp2n3`, ..., `sp2n6`):

1 Edit `lsf.shared` and add an external resource called `spname`. For example:

```
Begin Resource
RESOURCENAME TYPE    INTERVAL INCREASING DESCRIPTION
...
spname       String  ()       ()         (sp2 sys partition
name)
...
End Resource
```

2 Edit `lsf.cluster.`*`cluster_name`* and allocate the spname resource. For example:

```
Begin ResourceMap
RESOURCENAME  LOCATION
...
spname        (spp1@[sp2n1 sp2n3 sp2n5] spp2@[sp2n2 sp2n4
sp2n6])
...
End ResourceMap
```

5. Allocate switch adapter specific resources

If you use a switch adapter, you must define specific resources in LSF. During installation, `lsfinstall` defines the following external resources in `lsf.shared`:

- `adapter_windows`—number of free adapter windows on IBM SP Switch2 systems
- `ntbl_windows`—number of free network table windows on IBM HPS systems
- `css0`—number of free adapter windows on on css0 on IBM SP Switch2 systems
- `csss`—number of free adapter windows on on csss on IBM SP Switch2 systems

- `dedicated_tasks`—number of of running dedicated tasks
- `ip_tasks`—number of of running IP (Internet Protocol communication subsystem) tasks
- `us_tasks`—number of of running US (User Space communication subsystem) tasks

These resources are updated through `elim.hpc`.

```
Begin Resource
RESOURCENAME     TYPE    INTERVAL INCREASING DESCRIPTION
...
adapter_windows Numeric  30       N     (free adapter windows on css0 on IBM SP)
ntbl_windows    Numeric  30       N     (free ntbl windows on IBM HPS)
poe             Numeric  30       N     (poe availability)
css0            Numeric  30       N     (free adapter windows on css0 on IBM SP)
csss            Numeric  30       N     (free adapter windows on csss on IBM SP)
dedicated_tasks Numeric  ()       Y     (running dedicated tasks)
ip_tasks        Numeric  ()       Y     (running IP tasks)
us_tasks        Numeric  ()       Y     (running US tasks)
...
End Resource
```

You must edit `lsf.cluster.`*`cluster_name`* and allocate the external resources. For example, to configure a switch adapter for six SP2 nodes (`sp2n1`, `sp2n1`, `sp2n3`, ..., `sp2n6`):

```
Begin ResourceMap

RESOURCENAME      LOCATION
...
adapter_windows   [default]
ntbl_windows      [default]
css0              [default]
csss              [default]
dedicated_tasks   (0@[default])
ip_tasks          (0@[default])
us_tasks          (0@[default])
...
End ResourceMap
```

The `adapter_windows` and `ntbl_windows` resources are required for all POE jobs.

The other three resources are only required when you run IP and US jobs at the same time.

6. Optional. Tune PAM parameters

To improve performance and scalability for large POE jobs, tune the following `lsf.conf` parameters. The user's environment can override these.

- LSF_HPC_PJL_LOADENV_TIMEOUT

  Timeout value in seconds for PJL to load or unload the environment. For example, the time needed for IBM POE to load or unload adapter windows.

  At job startup, the PJL times out if the first task fails to register within the specified timeout value. At job shutdown, the PJL times out if it fails to exit after the last Taskstarter termination report within the specified timeout value.

Default: LSF_HPC_PJL_LOADENV_TIMEOUT=300

◆ LSF_PAM_RUSAGE_UPD_FACTOR

This factor adjusts the update interval according to the following calculation:

RUSAGE_UPDATE_INTERVAL + num_tasks * 1 * LSF_PAM_RUSAGE_UPD_FACTOR.

PAM updates resource usage for each task for every SBD_SLEEP_TIME + num_tasks * 1 seconds (by default, SBD_SLEEP_TIME=15). For large parallel jobs, this interval is too long. As the number of parallel tasks increases, LSF_PAM_RUSAGE_UPD_FACTOR causes more frequent updates.

Default: LSF_PAM_RUSAGE_UPD_FACTOR=0.01For large clusters

7. Reconfigure to apply the changes

1 Run `badmin ckconfig` to check the configuration changes.

If any errors are reported, fix the problem and check the configuration again.

2 Reconfigure the cluster:

```
badmin reconfig
Checking configuration files ...
No errors found.
Do you want to reconfigure? [y/n] y
Reconfiguration initiated
```

LSF checks for any configuration errors. If no fatal errors are found, you are asked to confirm reconfiguration. If fatal errors are found, reconfiguration is aborted.

POE ELIM (elim.hpc)

An external LIM (ELIM) for POE jobs is supplied with LSF.

On IBM HPS systems, ELIM uses the `st_status` or `ntbl_status` command to collect information from the Resource Manager.

PATH variable in elim

The ELIM searches the following path for the `poe` and `st_status` commands:

```
PATH="/usr/bin:/bin:/usr/local/bin:/local/bin:/sbin:/usr/sbin:/usr/ucb:/usr/sbi
n:
/usr/bsd:${PATH}"
```

If these commands are installed in a different directory, you must modify the PATH variable in `LSF_SERVERDIR/elim.hpc` to point to the correct directory.

POE esub (esub.poe)

The `esub` for POE jobs, `esub.poe`, is installed by `lsfinstall`. It is invoked using the `-a poe` option of `bsub`. By default, the POE `esub` sets the environment variable LSF_PJL_TYPE=poe. The job launcher, `mpirun.lsf` reads the environment variable LSF_PJL_TYPE=poe, and generates the appropriate `pam` command line to invoke POE to start the job.

LSF options

The value of the `bsub -n` option overrides the POE `-procs` option. If no `-n` is used, the `esub` sets default values with the variables LSB_SUB_NUM_PROCESSORS=1 and LSB_SUB_MAX_NUM_PROCESSORS=1.

POE options

If you specify `-euilib us` (US mode), then `-euidevice` must be `css0` or `csss` (the HPS for interprocess communications.)

The `-euidevice sn_all` option is supported. The `-euidevice sn_single` option is ignored. POE jobs submitted with `-euidevice sn_single` use `-euidevice sn_all`.

POE PJL wrapper (poejob)

The POE PJL (Parallel Job Launcher) wrapper, `poejob`, parses the POE job options, and filters out those that have been set by LSF.

Submitting POE jobs

Use `bsub` to submit POE jobs, including parameters required for the application and POE. PAM launches POE and collects resource usage for all running tasks in the parallel job.

Syntax

```
bsub -a poe [bsub_options] mpirun.lsf program_name [program_options]
[poe_options]
```

where:

-a poe

Invokes `esub.poe`.

Examples

Running US jobs

To submit an POE job in US mode, and runs on six processors:

```
bsub -a poe -n 6 mpirun.lsf my_prog -euilib us -euidevice css0
```

Running IP jobs

To run POE jobs in IP mode, MP_EUILIB (or `-euilib`) must be set to IP (Internet Protocol communication subsystem). For example:

```
bsub -a poe -n 6 mpirun.lsf my_prog -euilib ip ...
```

POE -procs option

The POE -procs option is ignored by esub.poe. Use the bsub -n option to specify the number of processors required for the job. The default if -n is not specified is 1.

Submitting POE jobs with a job script

A wrapper script is often used to call the POE script. You can submit a job using a job script as an embedded script or directly as a job, for example:

```
bsub -a -n 4 poe < embedded_jobscript
bsub -a -n 4 poe jobscript
```

For information on generic PJL wrapper script components, see Chapter 2, “Running Parallel Jobs”.

See *Administering Platform LSF* for information about submitting jobs with job scripts.

IBM SP Switch2 support

The SP Switch2 switch should be correctly installed and operational. By default, LSF only supports homogeneous clusters of IBM SP PSSP 3.4 or PSSP 3.5 SP Swich2 systems.

To verify the version of PSSP, run:

```
lslpp -l | grep ssp.basic
```

Output should look something like:

```
lslpp -l | grep ssp.basic
ssp.basic      3.2.0.9  COMMITTED  SP System Support Package
ssp.basic      3.2.0.9  COMMITTED  SP System Support Package
```

To verify the switch type, run:

```
SDRGetObjects Adapter css_type
```

Switch type	Value
SP_Switch_Adapter	2
SP_Switch_MX_Adapter	3
SP_Switch_MX2_Adapter	3
SP_Switch2_Adapter	5

`SP_Switch2_Adapter` indicates that you are using SP Switch2.

Use these values to configure the `device_type` variable in the script `LSF_BINDIR/poejob`. The default for `device_type` is 3.

IBM High Performance Switch (HPS) support

Running US jobs

Tasks of a parallel job running in US mode use the IBM pSeries High Performance Switch (HPS) exclusively for communication. HPS resources are referred to as *network table windows*. For US jobs to run, network table windows must be allocated ahead of the actual application startup.

You can run US jobs through LSF control (Load Leveler (LL) is not used). Job execution for US jobs has two stages:

1. Load HPS network table windows using `ntbl_api` HPS support via The AIX Switch Network Interface (SNI)
2. Optional. Start the application using the POE wrapper `poe_w` command

Running IP jobs

IP jobs do not require loading of network table windows. You just start `poe` or `poe_w` with the proper host name list file supplied.

How jobs start

Starting a parallel job on a pSeries HPS system is similar to starting jobs on an SP Switch2 system:

1. Load a table file to connect network table windows allocated to a task
2. Launch the task over network table windows connected
3. Unload the same table file to disconnect the network table window allocated to the task

Migrating IBM Load Leveler Job Scripts to Use LSF Options

You can integrate LSF with your POE jobs by modifying your job scripts to convert POE Load Leveler options to LSF options. After modifying your job scripts, your LSF job submission will be equivalent to a POE job submission:

`bsub < jobscript` becomes equivalent to `Llsubmit jobCmdFile`

The following POE options are handled differently when converting to LSF options:

- US (User Space) options
- IP (Internet Protocol) options
- `-nodes` combinations
- Other Load Leveler directives

US options

Use the following combinations of US options as a guideline for converting them to LSF options.

-cpu_use unique

-adapter_use dedicated	-adapter_use shared
bsub -a poe -R "select[adapter_windows>0 && us_tasks==0] rusage[adapter_windows=1: us_tasks=1: dedicated_tasks=1]	bsub -a poe -R "select[adapter_windows>0 && dedicated_tasks==0]rusage[adapter_windows=1: us_tasks=1]" ◆ Set MXJ to ! for the hosts on which these jobs will run ◆ The slots can only run these jobs

-cpu_use multiple

-adapter_use dedicated	-adapter_use shared
bsub -a poe -R "select[adapter_windows>0 && us_tasks=0] rusage[adapter_windows=1: us_tasks=1: dedicated_tasks=1]"	bsub -a poe -R "select[adapter_windows>0 && dedicated_tasks==0]"Rusage[adapter_windows=1:us_tasks=1]" ◆ Set MXJ ( ) for the hosts on which these jobs will run ◆ The hosts can only run these jobs

IP options

For IP jobs that do not use a switch, `adapter_use` does not apply. Use the following combinations of IP options as a guideline for converting them to LSF options.

-cpu_use unique

bsub -R "rusage[ip_tasks=1]"	◆ Set MXJ to ! for the hosts on which these jobs will run ◆ The slots can only run these jobs

-cpu_use multiple

bsub -R "rusage[ip_tasks=1]"	◆ Set MXJ ( ) for the hosts on which these jobs will run ◆ The hosts can only run these jobs

-nodes combinations

-nodes	-tasks_per_nodes -nodes combination	-nodes -procs
Cannot convert to LSF. You must use span[host=1]	bsub -n a*b -R "span[ptile=b]" ◆ Only use if the poe options are: poe -nodes a -tasks_per_nodes b -nodes b	bsub -n a*b -R "span[ptile=b]" ◆ Only use if the poe options are: poe -nodes a -tasks_per_nodes b -procs a*b

Load Leveler directives

Load Leveler job commands are handled as follows:

- Ignored by LSF
- Converted to `bsub` options (or queue options in `lsb.queues`)
- Require special handling in your job script

Load Leveler Command	Ignored	bsub option	Special Handling
account_no	Y		Use LSF accounting.
arguments	Y		Place job arguments in the job command line
blocking		bsub -n with span[ptile]	
all checkpoint commands	Y		
class		bsub -P or -J	
comment	Y		
core_limit		bsub -C	
cpu_limit		bsub -c or -n	
data_limit		bsub -D	
dependency		bsub -w	
environment			Set in job script or in esub.poe
error		bsub -e	
executable	Y		Enter the job name in the job script
file_limit		bsub -F	
group	Y		
hold		bsub -H	
image_size		bsub -v or -M	
initialdir	Y		The working directory is the current directory
input		bsub -i	
job_cpu_limit		bsub -c	
job_name		bsub -J	
job_type	Y		Handled by esub.poe
max_processors		bsub -n min, max	
min_processors		bsub -n min, max	
network		bsub -R	
node combinations		See "-nodes combinations" on page 80	
notification			Set in lsf.conf

Load Leveler Command	Ignored	bsub option	Special Handling
notify_user			Set in lsf.conf
output		bsub -o	
parallel_path	Y		
preferences		bsub -R "select[...]	
queue		bsub -q	
requirements		bsub -R and -m	
resources		bsub -R	Set rusage for each task according to the Load Leveler equivalent
rss_limit		bsub -M	
shell	Y		
stack_limit		bsub -S	
startdate		bsub -b	
step_name	Y		
task_geometry			Use the LSB_PJL_TASK_GEOMETRY environment variable to specify task geometry for your jobs. LSB_PJL_TASK_GEOMETRY overrides any mpirun n option.
total_tasks		bsub -n	
user_priority		bsub -sp	
wall_clock_limit		bsub -W	

Simple job script modifications

The following example shows how to convert the POE options in a Load Leveler command file to LSF options in your job scripts for a non-shared US or IP job.

Assumptions

- Only one job at a time can run on a non-shared node
- An IP job can share a node with a dedicated US job (`-adapter_use` is dedicated)
- The POE job always runs one task per CPU, so the `-cpu_use` option is not used

Example Load Leveler command file

This example uses following POE job script to run an executable named `mypoejob`:

```
#!/bin/csh
#@ shell = /bin/csh
#@ environment = ENVIRONMENT=BATCH; COPY_ALL;\
#  MP_EUILIB=us; MP_STDOUTMODE=ordered; MP_INFOLEVEL=0;
#@ network.MPI = switch,dedicated,US
#@ job_type = parallel
#@ job_name = batch-test
#@ output = $(job_name).log
#@ error  = $(job_name).log
#@ account_no = USER1
#@ node = 2
#@ tasks_per_node = 8
#@ node_usage = not_shared
#@ wall_clock_limit = 1:00:00
#@ class = batch
#@ notification = never
#@ queue
# --------------------------------------------
# Copy required workfiles to $WORKDIR, which is set
# to /scr/$user under the large GPFS work filesystem,
# named /scr.
cp ~/TESTS/mpihello $WORKDIR/mpihello

# Change directory to $WORKDIR
cd $WORKDIR

# Execute program mypoejob
poe mypoejob
poe $WORKDIR/mpihello

# Copy output data from $WORKDIR to appropriate archive FS,
# since we are currently running within a volatile
# "scratch" filesystem.

# Clean unneeded files from $WORKDIR after job ends.
rm -f $WORKDIR/mpihello
echo "Job completed at: `date`"
```

To convert POE options in a Load Leveler command file to LSF options

1. Make sure the queue `hpc_ibm` is available in `lsb.queues`.
2. Set the EXCLUSIVE parameter of the queue:

   ```
   EXCLUSIVE=Y
   ```
3. Create the job script for the LSF job. For example:

   ```
   #!/bin/csh
   # mypoe_jobscript
   # Start script ---------
   #BSUB -a poe
   #BSUB -n 16
   #BSUB -x
   #BSUB -o batch_test.%J_%I.out
   #BSUB -e batch_test.%J_%I.err
   #BSUB -W 60
   #BSUB -J batch_test
   #BSUB -q hpc_ibm
   setenv ENVIRONMENT BATCH
   setenv MP_EUILIB=us
   # Copy required workfiles to $WORKDIR, which is set
   # to /scr/$user under the large GPFS work filesystem,
   # named /scr.
   cp ~/TESTS/mpihello $WORKDIR/mpihello

   # Change directory to $WORKDIR
   cd $WORKDIR

   # Execute program mypoejob
   mpirun.lsf mypoejob -euilib us
   mpirun.lsf $WORKDIR/mpihello -euilib us
   ```

```
# Copy output data from $WORKDIR to appropriate archive FS,
# since we are currently running within a volatile
# "scratch" filesystem.

# Clean unneeded files from $WORKDIR after job ends.
rm -f $WORKDIR/mpihello
echo "Job completed at: `date`"
# End script ---------
```

4. Submit the job script as a redirected job, specifying the appropriate resource requirement string:

```
bsub -R "select[adapter_windows>0] rusage[adapter_windows=1] span[ptile=8]" <
mypoe_jobscript
```

Comparing some of the converted options

POE	LSF
#@ environment = ENVIRONMENT=BATCH; MP_EUILIB=us	setenv ENVIRONMENT BATCH setenv MP_EUILIB=us
#@wall_clock_limit = 1:00:00	#BSUB - W 60
#@ output = $(job_name).log	#BSUB -o batch_test.%J_%I.out
#@ error = $(job_name).log	#BSUB -e batch_test.%J_%I.err
#@node =2 #@tasks_per_node =8	#BSUB -n 16 -R "span[ptile=8]"
# Execute programs: poe mypoejob poe $WORKDIR/mpihello	#Execute programs: mpirun.lsf mypoejob -euilib us mpirun.lsf $WORKDIR/mpihello -euilib us

Submitting the job

Compare the job script submission with the equivalent job submitted with all the LSF options on the command line:

```
bsub -x -a poe -q hpc_ibm -n 16 -R "select[adapter_windows>0]
rusage[adapter_windows=1] span[ptile=8]" mpirun.lsf mypoejob -euilib us
```

To submit the same job as an IP job, substitute `ip` for `us`, and remove the `select` and `rusage` statements:

```
bsub -x -a poe -q hpc_ibm -n 16 -R "span[ptile=8]" mpirun.lsf mypoejob
-euilib ip
```

To submit the job as a shared US or IP job, remove the `bsub -x` option from the job script or command line. This allows other jobs to run on the host your job is running on:

```
bsub -a poe -q hpc_ibm -n 16 -R "span[ptile=8]" mpirun.lsf mypoejob -euilib us
```

or

```
bsub -a poe -q hpc_ibm -n 16 -R "span[ptile=8]" mpirun.lsf mypoejob -euilib ip
```

Advanced job script modifications

If your environment runs any of the following:

- A mix of IP and US jobs,
- A combinations of dedicated and shared `-adapter_use`
- Unique and multiple `-cpu_use`

your job scripts must use the `us_tasks` and `dedicated_tasks` LSF resources.

The following examples show how to convert the POE options in a Load Leveler command file to LSF options in your job scripts for several kinds of jobs.

-adapter_use dedicated and -cpu_use unique

- This example uses following POE job script:

```
#!/bin/csh
#@ shell = /bin/csh
#@ environment = ENVIRONMENT=BATCH; COPY_ALL;\
#  MP_EUILIB=us; MP_STDOUTMODE=ordered; MP_INFOLEVEL=0;
#@ network.MPI = switch,dedicated,US
#@ job_type = parallel
#@ job_name = batch-test
#@ output = $(job_name).log
#@ error  = $(job_name).log
#@ account_no = USER1
#@ node = 2
#@ tasks_per_node = 8
#@ node_usage = not_shared
#@ wall_clock_limit = 1:00:00
#@ class = batch
#@ notification = never
#@ queue
# ---------------------------------------------
# Copy required workfiles to $WORKDIR, which is set
# to /scr/$user under the large GPFS work filesystem,
# named /scr.
cp ~/TESTS/mpihello $WORKDIR/mpihello

# Change directory to $WORKDIR
cd $WORKDIR

# Execute program(s)
poe mypoejob
poe $WORKDIR/mpihello

# Copy output data from $WORKDIR to appropriate archive FS,
# since we are currently running within a volatile
# "scratch" filesystem.

# Clean unneeded files from $WORKDIR after job ends.
rm -f $WORKDIR/mpihello
echo "Job completed at: `date`"
```

- The job script for the LSF job is:

```
#!/bin/csh
# mypoe_jobscript
#BSUB -a poe
#BSUB -n 16
#BSUB -x
#BSUB -o batch_test.%J_%I.out
#BSUB -e batch_test.%J_%I.err
#BSUB -W 60
#BSUB -J batch_test
```

```
#BSUB -q hpc_ibm
setenv ENVIRONMENT BATCH
setenv MP_EUILIB us
# Copy required workfiles to $WORKDIR, which is set
# to /scr/$user under the large GPFS work filesystem,
# named /scr.
cp ~/TESTS/mpihello $WORKDIR/mpihello

# Change directory to $WORKDIR
cd $WORKDIR

# Execute program(s)
mpirun.lsf mypoejob -euilib us
mpirun.lsf $WORKDIR/mpihello -euilib us
# Copy output data from $WORKDIR to appropriate archive FS,
# since we are currently running within a volatile
# "scratch" filesystem.

# Clean unneeded files from $WORKDIR after job ends.
rm -f $WORKDIR/mpihello
echo "Job completed at: `date`"
# End of script ---------
```

Submitting the job

- Submit the job script as a redirected job, specifying the appropriate resource requirement string:

```
bsub -R "select[adapter_windows>0] rusage[adapter_windows=1] span[ptile=8]" <
mypoe_jobscript
```

- Submit `mypoejob` as a single exclusive job:

```
bsub -x -a poe -q hpc_ibm -n 16 -R "select[adapter_windows>0]
rusage[adapter_windows=1] span[ptile=8]" mpirun.lsf mypoejob -euilib us
```

Controlling Allocation and User Authentication for IBM POE Jobs

About POE authentication

Establishing authentication for POE jobs means ensuring that users are permitted to run parallel jobs on the nodes they intend to use. POE supports two types of user authentication:

- AIX authentication (the default)

  Uses `/etc/hosts.equiv` or `$HOME/.rhosts`
- DFS/DCE authentication

When interactive remote login to HPS execution nodes is not allowed, you can still run parallel jobs under Parallel Environment (PE) through LSF. PE jobs under LSF on the system with restricted access to the execution nodes uses two wrapper programs to allow user authentication:

- `poe_w`—wrapper for the `poe` driver program
- `pmd_w`—wrapper for `pmd` (PE Partition Manager Daemon)

Enabling user authentication for POE jobs

To enable user authentication through the `poe_w` and `pmd_w` wrappers, you must set LSF_HPC_EXTENSIONS="LSB_POE_AUTHENTICATION" in `/etc/lsf.conf`.

Enforcing node and CPU allocation for POE jobs

To enable POE Allocation control, use LSF_HPC_EXTENSIONS="LSB_POE_ALLOCATION" in `/etc/lsf.conf`. `poe_w` enforces the LSF allocation decision from `mbatchd`.

For US jobs, `swtbl_api` and `ntbl_api` validates network table windows data files with `mbatchd`. For IP and US jobs, `poe_wrapper` validates the POE host file with the information from `mbatchd`. If the information does not match with the information from `mbatchd`, the job is terminated.

When LSF_HPC_EXTENSIONS="LSB_POE_ALLOCATION" is set:

- `poe_w` parses the POE host file and validates its contents with information from `mbatchd`.
- `ntbl_api` and `swtbl_api` parse the network table and switch table data files and validate their contents with information from `mbatchd`.

Validation rules

- Host names from data files must match host names as allocated by LSF
- The number of tasks per node cannot exceed the number of tasks per node as allocated by LSF
- Total number of tasks cannot exceed the total number of tasks requested at job submission (`bsub -n`)

Configuring POE allocation and authentication support

Configure services

1. Register `pmv4lsf` (`pmv3lsf`) service with `inetd`:
   a. Add the following line to `/etc/inetd.conf`:

```
pmv4lsf   stream  tcp    nowait  root   /etc/pmdv4lsf pmdv4lsf
```

   b. Make a symbolic link from pmd_w to `/etc/pmdv4lsf`.

   For example:

```
# ln -s $LSF_BINDIR/pmd_w /etc/pmdv4lsf
```

   Both $LSF_BINDIR and /etc must be owned by root for the symbolic link to work. Symbolic links are not allowed under /etc on some AIX 5.3 systems, so you may need to copy $LSF_BINDIR/pmd_w to /etc/pmdv4lsf:

```
cp -f $LSF_BINDIR/pmd_w /etc/pmdv4lsf
```

   c. Add `pmv4lsf` to `/etc/services`.

   For example:

```
pmv4lsf           6128/tcp   #pmd wrapper
```

2. Add `poelsf` service to `/etc/services`.

   The port defined for this service will be used by pmd_w and poe_w for communication with each other.

```
poelsf      6129/tcp   #pmd_w - poe_w communication port
```

3. Run one of the following commands to restart `inetd`:

```
# refresh -s inetd
# kill -1 "inetd_pid"
```

Configure parameters

1. Create `/etc/lsf.conf` file if does not exist already and add the following parameter:

```
LSF_HPC_EXTENSIONS="LSB_POE_ALLOCATION LSB_POE_AUTHENTICATION"
```

2. (Optional) Two optional parameters can be added to the `lsf.conf` file:
   - LSF_POE_TIMEOUT_BIND—time in seconds for poe_w to keep trying to set up a server socket to listen on.

     Default: 120 seconds.
   - LSF_POE_TIMEOUT_SELECT—time in seconds for poe_w to wait for connections from pmd_w.

     Default: 160 seconds.

Both LSF_POE_TIMEOUT_BIND and LSF_POE_TIMEOUT_SELECT can also be set as environment variables for poe_w to read.

Example job scripts

For IP jobs For the following job script:

```
# mypoe_jobscript
#!/bin/sh
#BSUB -o out.%J
#BSUB -n 2
#BSUB -m "hostA"
#BSUB -a poe

export MP_EUILIB=ip

mpirun.lsf ./hmpis
```

Submit the job script as a redirected job, specifying the appropriate resource requirement string:

```
bsub -R "select[poe>0]" < mypoe_jobscript
```

For US jobs: For the following job script:

```
# mypoe_jobscript
#!/bin/sh
#BSUB -o out.%J
#BSUB -n 2
#BSUB -m "hostA"
#BSUB -a poe

export MP_EUILIB=us

mpirun.lsf ./hmpis
```

Submit the job script as a redirected job, specifying the appropriate resource requirement string:

```
bsub -R "select[ntbl_windows>0] rusage[ntbl_windows=1] span[ptile=1]" <
mypoe_jobscript
```

Limitations

- POE authentication for LSF jobs is supported on PE 3.x or PE 4.x. It is assumed that only one `pmd` version is installed on each node in the default location:

  `/usr/lpp/ppe.poe/bin/pmdv3` for PE 3.x

  or

  `/usr/lpp/ppe.poe/bin/pmdv4` for PE 4.x

  If both `pmdv3` and `pmdv4` are available in `/usr/lpp/ppe.poe/bin`, `pmd_w` launches `pmdv3`.

Submitting IBM POE Jobs over InfiniBand

Platform LSF installation adds a shared nrt_windows resource to run and monitor POE jobs over the InfiniBand interconnect.

lsb.shared

```
Begin Resource
RESOURCENAME     TYPE    INTERVAL INCREASING DESCRIPTION
...
poe             Numeric  30       N    (poe availability)
dedicated_tasks Numeric  ()       Y    (running dedicated
tasks)
ip_tasks        Numeric  ()       Y    (running IP tasks)
us_tasks        Numeric  ()       Y    (running US tasks)
nrt_windows     Numeric  30       N    (free nrt windows on
IBM poe over IB)
...
End Resource
```

lsf.cluster.cluster_name

```
Begin ResourceMap
RESOURCENAME        LOCATION
poe                 [default]
nrt_windows     [default]
dedicated_tasks     (0@[default])
ip_tasks     (0@[default])
us_tasks     (0@[default])
End ResourceMap
```

Job Submission

Run `bsub -a poe` to submit an IP mode job:

```
bsub -a poe mpirun.lsf job job_options -euilib ip poe_options
```

Run bsub -a poe to submit a US mode job:

```
bsub -a poe mpirun.lsf job job_options -euilib us poe_options
```

If some of the AIX hosts do not have InfiniBand support (for example, hosts that still use HPS), you must explicitly tell LSF to exclude those hosts:

```
bsub -a poe -R "select[nrt_windows>0]" mpirun.lsf job job_options poe_options
```

Job monitoring

Run `lsload` to display the `nrt_windows` and `poe` resources:

```
lsload -l
HOST_NAME  status r15s  r1m  r15m ut  pg  io ls it tmp   swp mem nrt_windows poe
hostA      ok     0.0   0.0  0.0  1% 8.1  4 1  0  1008M 4090M 6976M 128.0   1.0
hostB      ok     0.0   0.0  0.0  0% 0.7  1 0  0  1006M 4092M 7004M 128.0   1.0
```

CHAPTER

5

Using Platform LSF with SGI Cpusets

Platform LSF makes use of SGI cpusets to enforce processor limits for LSF jobs. When a job is submitted, LSF creates a cpuset and attaches it to the job before the job starts running, After the job finishes, LSF deallocates the cpuset. If no host meets the CPU requirements, the job remains pending until processors become available to allocate the cpuset.

Contents

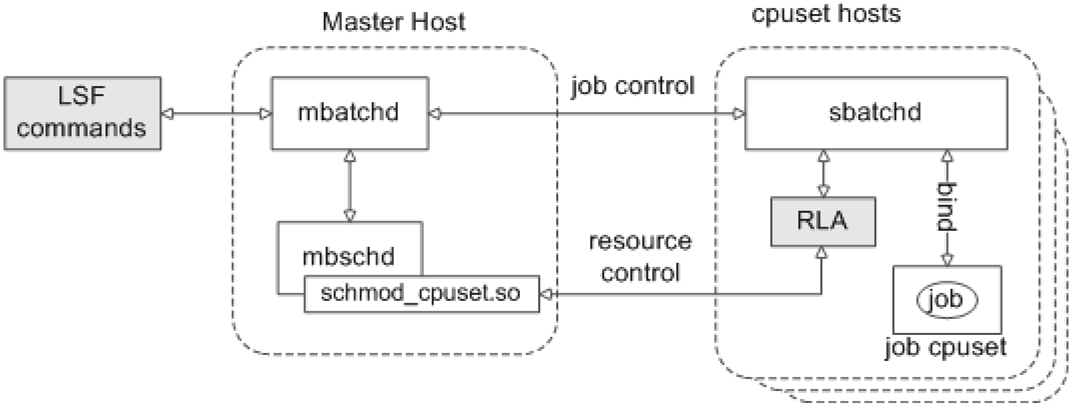
About SGI cpusets

An SGI cpuset is a named set of CPUs. The processes attached to a cpuset can only run on the CPUs belonging to that cpuset.

Dynamic cpusets Jobs are attached to a cpuset dynamically created by LSF. The cpuset is deleted when the job finishes or exits. If not specified, the default cpuset type is dynamic.

Static cpusets Jobs are attached to a static cpuset specified by users at job submission. This cpuset is *not* deleted when the job finishes or exits. Specifying a cpuset name at job submission implies that the cpuset type is static. If the static cpuset does not exist, the job will remain pending until LSF detects a static cpuset with the specified name.

System architecture



How LSF uses cpusets

CPU containment and reservation On systems running IRIX 6.5.24 and up or SGI Altix or AMD64 (x86-64) ProPack 3.0 and up, cpusets can be created and deallocated dynamically out of available machine resources. Not only does the cpuset provide containment, so that a job requiring a specific number of CPUs will only run on those CPUs, but also reservation, so that the required number of CPUs are guaranteed to be available only for the job they are allocated to.

Cpuset creation and deallocation LSF can be configured to make use of SGI cpusets to enforce processor limits for LSF jobs. When a job is submitted, LSF creates a cpuset and attaches it to the job when the job is scheduled. After the job finishes, LSF deallocates the cpuset. If no host meets the CPU requirements, the job remains pending until processors become available to allocate the cpuset.

Assumptions and limitations

- When LSF selects cpuset jobs to preempt, MINI_JOB and LEAST_RUN_TIME are ignored in the PREEMPT_FOR parameter in `lsb.params`

- When using cpusets, LSF schedules jobs based on the number of slots assigned to the hosts instead of the number of CPUs. The `lsb.params` parameter setting PARALLEL_SCHED_BY_SLOTS=N has no effect.
- Preemptable queue preference is not supported
- Before upgrading from a previous version, clusters must be drained of all running jobs (especially cpuset hosts)
- The new cpuset integration cannot coexist with the old integration within the same cluster
- Under the MultiCluster lease model, both clusters must use the same version of the cpuset integration
- LSF supports up to ProPack 6.0.
- LSF will not create a cpuset on hosts of different ProPack versions.

Backfill and slot reservation

Since backfill and slot reservation are based on an entire host, they may not work correctly if your cluster contains hosts that use both static and dynamic cpusets or multiple static cpusets.

Chunk jobs

Jobs submitted to a chunk job queue are not chunked together, but run as individual LSF jobs inside a dynamic cpuset.

Preemption

- When LSF selects cpuset jobs to preempt, specialized preemption preferences, such as MINI_JOB and LEAST_RUN_TIME in the PREEMPT_FOR parameter in `lsb.params`, and others are ignored when slot preemption is required.
- Preemptable queue preference is not supported.

Pre-execution and post-execution

Job pre-execution programs run within the job cpuset, since they are part of the job. By default, post-execution programs run outside of the job cpuset.

If JOB_INCLUDE_POSTPROC=Y is specified in `lsb.applications`, post-execution processing is not attached to the job cpuset, and Platform LSF does not release the cpuset until post-execution processing has finished.

Suspended jobs

Jobs suspended (for example, with `bstop`) will release their cpusets.

Cpuset memory options

- **SGI Altix Linux ProPack versions 4 and lower** do not support memory migration; you must define RESUME_OPTION=ORIG_CPUS to force LSF to recreate the original cpuset when LSF resumes a job.
- **SGI Altix Linux ProPack 5** supports memory migration and does not require additional configuration to enable this feature. If you submit and then suspend a job using a dynamic cpuset, LSF will create a new dynamic cpuset when the job resumes. The memory pages for the job are migrated to the new cpuset as required.
- **SGI Altix Linux ProPack 3** only supports CPUSET_OPTIONS=CPUSET_MEMORY_LOCAL. If the cpuset job runs on an Altix host, other cpuset attributes are ignored.
- **SGI Altix Linux ProPack 4 and ProPack 5** do not support CPUSET_OPTIONS=CPUSET_MEMORY_MANDATORY or CPUSET_OPTIONS=CPUSET_CPU_EXCLUSIVE attributes. If the cpuset job runs on an Altix host, the cpusets created on the Altix system will have their memory usage restricted to the memory nodes containing the CPUs assigned to the cpuset. The CPUSET_MEMORY_MANDATORY and CPUSET_CPU_EXCLUSIVE attributes are ignored.

Static cpusets

- **SGI Altix Linux ProPack 4 and ProPack 5** static cpuset definitions must include both the cpus and the memory nodes on which the cpus reside. The memory node assignments should be non-exclusive, which allows other cpusets to use the same nodes. With non-exclusive assignment of memory nodes, the allocation of cpus will succeed even if the cpuset definition does not correctly map cpus to memory nodes.

PAM jobs on IRIX

PAM on IRIX cannot launch parallel processes within cpusets.

Array services authentication (Altix only)

For PAM jobs on Altix, the SGI Array Services daemon `arrayd` must be running and AUTHENTICATION must be set to NONE in the SGI array services authentication file `/usr/lib/array/arrayd.auth` (comment out the AUTHENTICATION NOREMOTE method and uncomment the AUTHENTICATION NONE method).

To run a mulithost MPI applications, you must also enable `rsh` without password prompt between hosts:

- The remote host must defined in the `arrayd` configuration.
- Configure `.rhosts` so that `rsh` does not require a password.

For more information about SGI Array Services, see “SGI Job Container and Process Aggregate Support” on page 116.

For more information about PAM jobs, see “SGI Vendor MPI Support” on page 25.

Forcing a cpuset job to run

The administrator must use `brun -c` to force a cpuset job to run. If job is forced to run on non-cpuset hosts, or if any host in the host list specified with `-m` is not a cpuset host, `-extsched` cpuset options are ignored and the job runs with no cpusets allocated.

If the job is forced to run on a cpuset host:

- For dynamic cpusets: LSF allocates a dynamic cpuset without any cpuset options and runs the job inside the dynamic cpuset
- For static cpusets: LSF runs the job in static cpuset. If the specific static cpuset does not exsit, the job is requeued.

Resizable jobs

Jobs running in a cpuset cannot be resized.

Configuring LSF with SGI Cpusets

Automatic configuration at installation and upgrade

lsb.modules During installation and upgrade, `lsfinstall` adds the `schmod_cpuset` external scheduler plugin module name to the PluginModule section of `lsb.modules`:

```
Begin PluginModule
SCH_PLUGIN              RB_PLUGIN           SCH_DISABLE_PHASES
schmod_default              ()                        ()
schmod_cpuset               ()                        ()
End PluginModule
```

The schmod_cpuset plugin name must be configured after the standard LSF plugin names in the PluginModule list.

For upgrade, `lsfinstall` comments out the `schmod_topology` external scheduler plugin name in the PluginModule section of `lsb.modules`

lsf.conf During installation and upgrade, `lsfinstall` sets the following parameters in `lsf.conf`:

- LSF_ENABLE_EXTSCHEDULER=Y

  LSF uses an external scheduler for cpuset allocation.
- LSB_CPUSET_BESTCPUS=Y

  LSF schedules jobs based on the shortest CPU radius in the processor topology using a best-fit algorithm for cpuset allocation.

  LSF_IRIX_BESTCPUS is obsolete.
- LSB_SHORT_HOSTLIST=1

  Displays an abbreviated list of hosts in `bjobs` and `bhist` for a parallel job where multiple processes of a job are running on a host. Multiple processes are displayed in the following format:

  `processes*hostA`

For upgrade, `lsfinstall` comments out the following obsolete parameters in `lsf.conf`, and sets the corresponding RLA configuration:

- LSF_TOPD_PORT=*port_number*, replaced by LSB_RLA_PORT=*port_number*, using the same value as LSF_TOPD_PORT.

  Where *port_number* is the TCP port used for communication between the LSF topology adapter (RLA) and `sbatchd`.

  The default port number is 6883.
- LSF_TOPD_WORKDIR=*directory* parameter, replaced by LSB_RLA_WORKDIR=*directory* parameter, using the same value as LSF_TOPD_WORKDIR

  Where *directory* is the location of the status files for RLA. Allows RLA to recover its original state when it restarts. When RLA first starts, it creates the directory defined by LSB_RLA_WORKDIR if it does not exist, then creates subdirectories for each host.

lsf.shared During installation and upgrade, `lsfinstall` defines the cpuset Boolean resource in `lsf.shared`:

```
Begin Resource
RESOURCENAME    TYPE      INTERVAL  INCREASING   DESCRIPTION
...
cpuset          Boolean   ()        ()           (cpuset host)
...
End Resource
```

You should add the cpuset resource name under the RESOURCES column of the Host section of `lsf.cluster.`*`cluster_name`*. Hosts without the cpuset resource specified are not considered for scheduling cpuset jobs.

---

lsf.cluster.cluster_name

For each cpuset host, `hostsetup` adds the `cpuset` Boolean resource to the HOST section of `lsf.cluster.`*`cluster_name`*.

For more information See the *Platform LSF Configuration Reference* for information about the `lsb.modules`, `lsf.conf`, `lsf.shared`, and `lsf.cluster.`*`cluster_name`* files.

Optional configuration

lsb.queues

- In some pre-defined LSF queues, such as `normal`, the default MEMLIMIT is set to 5000 (5 MB). However, if ULDB is enabled (LSF_ULDB_DOMAIN is defined), the MEMLIMIT should be set greater than 8000.
- MANDATORY_EXTSCHED=CPUSET[*cpuset_options*]

  Sets required cpuset properties for the queue. MANDATORY_EXTSCHED options override `-extsched` options used at job submission.
- DEFAULT_EXTSCHED=CPUSET[*cpuset_options*]

  Sets default cpuset properties for the queue if the `-extsched` option is not used at job submission. `-extsched` options override the options set in DEFAULT_EXTSCHED.

  See "Specifying cpuset properties for jobs" on page 102 for more information about external scheduler options for setting cpuset properties.

lsf.conf

- LSB_RLA_UPDATE=*seconds*

  Specifies how often the LSF scheduler refreshes cpuset information from RLA.

  The default is 600 seconds.
- LSB_RLA_WORKDIR=*directory* parameter, where *directory* is the location of the status files for RLA. Allows RLA to recover its original state when it restarts. When RLA first starts, it creates the directory defined by LSB_RLA_WORKDIR if it does not exist, then creates subdirectories for each host.

  You should avoid using `/tmp` or any other directory that is automatically cleaned up by the system. Unless your installation has restrictions on the LSB_SHAREDIR directory, you should use the default:

  `LSB_SHAREDIR/`*`cluster_name`*`/rla_workdir`

You should not use a CXFS file system for LSB_RLA_WORKDIR.

- LSF_PIM_SLEEPTIME_UPDATE=Y

  On Altix hosts, use this parameter to improve job throughput and reduce a job's start time if there are many jobs running simultaneously on a host. This parameter reduces communication traffic between `sbatchd` and PIM on the same host.

  When this parameter is defined:

  - `sbatchd` does not query PIM immediately as it needs information—it will only query PIM every LSF_PIM_SLEEPTIME seconds.
  - `sbatchd` may be intermittently unable to retrieve process information for jobs whose run time is smaller than LSF_PIM_SLEEPTIME.
  - It may take longer to view resource usage with `bjobs -l`.

Increase file descriptor limit for MPI jobs (Altix only)

By default, Linux sets the maximum file descriptor limit to 1024. This value is too small for jobs using more than 200 processes. To avoid MPI job failure, specify a larger file descriptor limit. For example:

```
# /etc/init.d/lsf stop
# ulimit -n 16384
# /etc/init.d/lsf start
```

Any host with more than 200 CPUs should start the LSF daemons with the larger file descriptor limit. SGI Altix already starts the `arrayd` daemon with the same `ulimit` specifier, so that MPI jobs run without LSF can start as well.

For more information

See the *Platform LSF Configuration Reference* for information about the `lsb.queues` and `lsf.conf` files.

Resources for dynamic and static cpusets

If your environment uses both static and dynamic cpusets or you have more than one static cpuset configured, you must configure decreasing numeric resources to represent the cpuset count, and use `-R "rusage"` in job submission. This allows preemption, and also lets you control number of jobs running on static and dynamic cpusets or on each static cpuset.

Configuring cpuset resources

1. Edit `lsf.shared` and configure resources for cpusets and configure resources for static cpusets and non-static cpusets. For example:

```
Begin Resource
RESOURCENAME  TYPE    INTERVAL INCREASING  DESCRIPTION  # Keywords
   ...
   dcpus       Numeric ()       N
   scpus       Numeric ()       N
End Resource
```

Where:

- `dcpus` is the number CPUs outside static cpusets (that is the total number of CPUs minus the number of CPUs in static cpusets).
- `scpus` is the number of CPUs in static cpusets. For static cpusets, configure a separate resource for each static cpuset. You should use the cpuset name as the resource name.

The names `dcpus` and `scpus` can be any name you like.

2 Edit `lsf.cluster.`*`cluster_name`* to map the resources to hosts. For example:

```
Begin ResourceMap
RESOURCENAME        LOCATION
dcpus               (4@[hosta]) # total cpus - cpus in static cpusets
scpus               (8@[hostc]) # static cpusets
End ResourceMap
```

- For dynamic cpuset resources, the value of the resource should be the number of free CPUs on the host; that is, the number of CPUs *outside* of any static cpusets on the host.
- For static cpuset resources, the number of the resource should be the number of CPUs in the static cpuset.

3 Edit `lsb.params` and configure your cpuset resources as preemptable. For example:

```
Begin Parameters
...
PREEMPTABLE_RESOURCES = scpus dcpus
End Parameters
```

4 Edit `lsb.hosts` and set MXJ greater than or equal to the total number of CPUs in static and dynamic cpusets you have configured resources for.

Viewing your cpuset resources

Use the following commands to verify your configuration:

```
bhosts -s
RESOURCE                 TOTAL       RESERVED       LOCATION
dcpus                      4.0            0.0       hosta
scpus                      8.0            0.0       hosta
lshosts -s
RESOURCE                                VALUE       LOCATION
dcpus                                       4       hosta
scpus                                       8       hosta
bhosts
HOST_NAME        STATUS     JL/U    MAX  NJOBS  RUN  SSUSP  USUSP  RSV
hosta            ok            -      -      1    1      0      0    0
```

Using preemption

To use preemption on systems running IRIX or TRIX versions earlier than 6.5.24, use `cpusetscript` as the job suspend action in `lsb.queues`:

```
Begin Queue
...
JOB_CONTROLS = SUSPEND[cpusetscript]
...
End Queue
```

To enable checkpointing before the job is migrated by the `cpusetscript`, specify the CHKPNT=*chkpnt_dir* parameter in the configuration of the preemptable queue.

Submitting jobs

You must use `-R "rusage"` in job submission. This allows preemption, and also lets you control number of jobs running on static and dynamic cpusets or on each static cpuset.

Configuring default and mandatory cpuset options

Use the DEFAULT_EXTSCHED and MANDATORY_EXTSCHED queue paramters in `lsb.queues` to configure default and mandatory cpuset options.

Use keywords SGI_CPUSET[] or CPUSET[] to identify the external scheduler parameters. The keyword SGI_CPUSET[] is deprecated. The keyword CPUSET[] is preferred.

DEFAULT_EXTSCHED=[SGI_]CPUSET[*cpuset_options*]

Specifies default cpuset external scheduling options for the queue.

`-extsched` options on the `bsub` command are merged with DEFAULT_EXTSCHED options, and `-extsched` options override any conflicting queue-level options set by DEFAULT_EXTSCHED.

For example, if the queue specifies:

```
DEFAULT_EXTSCHED=CPUSET[CPUSET_OPTIONS=CPUSET_CPU_EXCLUSIVE]
```

and a job is submitted with:

```
-extsched "CPUSET[CPUSET_TYPE=dynamic;CPU_LIST=1,5,7-12;
CPUSET_OPTIONS=CPUSET_MEMORY_LOCAL]"
```

LSF uses the resulting external scheduler options for scheduling:

```
CPUSET[CPUSET_TYPE=dynamic;CPU_LIST=1, 5, 7-12;
CPUSET_OPTIONS=CPUSET_CPU_EXCLUSIVE CPUSET_MEMORY_LOCAL]
```

DEFAULT_EXTSCHED can be used in combination with MANDATORY_EXTSCHED in the same queue. For example, if the job specifies:

```
-extsched "CPUSET[CPU_LIST=1,5,7-12;MAX_CPU_PER_NODE=4]"
```

and the queue specifies:

```
Begin Queue
...
DEFAULT_EXTSCHED=CPUSET[CPUSET_OPTIONS=CPUSET_CPU_EXCLUSIVE]
MANDATORY_EXTSCHED=CPUSET[CPUSET_TYPE=dynamic;MAX_CPU_PER_NODE=2]
...
End Queue
```

LSF uses the resulting external scheduler options for scheduling:

```
CPUSET[CPUSET_TYPE=dynamic;MAX_CPU_PER_NODE=2;CPU_LIST=1, 5,
7-12;CPUSET_OPTIONS=CPUSET_CPU_EXCLUSIVE]
```

If cpuset options are set in DEFAULT_EXTSCHED, and you do not want to specify values for these options, use the keyword with no value in the `-extsched` option of `bsub`. For example, if `DEFAULT_EXTSCHED=CPUSET[MAX_RADIUS=2]`, and you do not want to specify any radius option at all, use `-extsched "CPUSET[MAX_RADIUS=]"`.

See "Specifying cpuset properties for jobs" on page 102 for more information about external scheduling options.

MANDATORY_EXTSCHED=[SGI_]CPUSET[*cpuset_options*]

Specifies mandatory cpuset external scheduling options for the queue.

`-extsched` options on the `bsub` command are merged with MANDATORY_EXTSCHED options, and MANDATORY_EXTSCHED options override any conflicting job-level options set by `-extsched`.

For example, if the queue specifies:

```
MANDATORY_EXTSCHED=CPUSET[CPUSET_TYPE=dynamic;MAX_CPU_PER_NODE=2]
```

and a job is submitted with:

```
-extsched "CPUSET[MAX_CPU_PER_NODE=4;CPU_LIST=1,5,7-12;]"
```

LSF uses the resulting external scheduler options for scheduling:

```
CPUSET[CPUSET_TYPE=dynamic;MAX_CPU_PER_NODE=2;CPU_LIST=1, 5, 7-12]
```

MANDATORY_EXTSCHED can be used in combination with DEFAULT_EXTSCHED in the same queue. For example, if the job specifies:

```
-extsched "CPUSET[CPU_LIST=1,5,7-12;MAX_CPU_PER_NODE=4]"
```

and the queue specifies:

```
Begin Queue
...
DEFAULT_EXTSCHED=CPUSET[CPUSET_OPTIONS=CPUSET_CPU_EXCLUSIVE]
MANDATORY_EXTSCHED=CPUSET[CPUSET_TYPE=dynamic;MAX_CPU_PER_NODE=2]
...
End Queue
```

LSF uses the resulting external scheduler options for scheduling:

```
CPUSET[CPUSET_TYPE=dynamic;MAX_CPU_PER_NODE=2;CPU_LIST=1, 5,
7-12;CPUSET_OPTIONS=CPUSET_CPU_EXCLUSIVE]
```

If you want to prevent users from setting certain cpuset options in the `-extsched` option of `bsub`, use the keyword with no value. For example, if the job is submitted with `-extsched "CPUSET[MAX_RADIUS=2]"`, use `MANDATORY_EXTSCHED=CPUSET[MAX_RADIUS=]` to override this setting.

See "Specifying cpuset properties for jobs" on page 102 for more information about external scheduling options.

Priority of topology scheduling options

The options set by `-extsched` can be combined with the queue-level MANDATORY_EXTSCHED or DEFAULT_EXTSCHED parameters. If `-extsched` and MANDATORY_EXTSCHED set the same option, the MANDATORY_EXTSCHED setting is used. If `-extsched` and DEFAULT_EXTSCHED set the same options, the `-extsched` setting is used.

topology scheduling options are applied in the following priority order of level from highest to lowest:

1. Queue-level MANDATORY_EXTSCHED options override ...
2. Job level `-ext` options, which override ...
3. Queue-level DEFAULT_EXTSCHED options

For example, if the queue specifies:

```
DEFAULT_EXTSCHED=CPUSET[MAX_CPU_PER_NODE=2]
```

and the job is submitted with:

```
bsub -n 4 -ext "CPUSET[MAX_CPU_PER_NODE=1]" myjob
```

The cpuset option in the job submission overrides the DEFAULT_EXTSCHED, so the job will run in a cpuset allocated with a maximum of 1 CPU per node, honoring the job-level MAX_CPU_PER_NODE option.

If the queue specifies:

```
MANDATORY_EXTSCHED=CPUSET[MAX_CPU_PER_NODE=2]
```

and the job is submitted with:

```
bsub -n 4 -ext "CPUSET[MAX_CPU_PER_NODE=1]" myjob
```

The job will run in a cpuset allocated with a maximum of 2 CPUs per node, honoring the MAX_CPU_PER_NODE option in the queue.

Using LSF with SGI Cpusets

Specifying cpuset properties for jobs

To specify cpuset properties for LSF jobs, use:

- The `-extsched` option of `bsub`.
- DEFAULT_EXTSCHED or MANDATORY_EXTSCHED, or both, in the queue definition (`lsb.queues`).

If a job is submitted with the `-extsched` option, LSF submits jobs with hold, then resumes the job before dispatching it to give time for LSF to attach the `-extsched` options. The job starts on the first execution host.

For more information about job operations, see *Administering Platform LSF*.

For more information about `bsub`, see the *Platform LSF Command Reference*.

Syntax

`-ext[sched] "[SGI_]CPUSET[cpuset_options]"`

Specifies a list of CPUs and cpuset attributes used by LSF to allocate a cpuset for the job.

You can abbreviate the -extsched option to -ext. Use keywords SGI_CPUSET[] or CPUSET[] to identify the external scheduler parameters. The keyword SGI_CPUSET[] is deprecated. The keyword CPUSET[] is preferred.

where *cpuset_options* are:

- `CPUSET_TYPE=static |dynamic | none;`

  Specifies the type of cpuset to be allocated.

  If you specify `none`, no cpuset is allocated and you cannot specify any other cpuset options, and the job runs outside of any cpuset.
- `CPUSET_NAME=name;`

  name is the name of a static cpuset. If you specify CPUSET_TYPE=static, you must provide a cpuset name. If you specify a cpuset name, but specify CPUSET_TYPE that is not static, the job is rejected.

Options valid only for dynamic cpusets

- `MAX_RADIUS=radius;`

  radius is the maximum cpuset radius the job can accept. If the radius requirement cannot be satisfied the job remains pending. `MAX_RADIUS` implies that the job cannot span multiple hosts. LSF puts each cpuset host into its own group to enforce this when MAX_RADIUS is specified.
- `RESUME_OPTION=ORIG_CPUS;`

  Specifies how LSF should recreate a cpuset when a job is resumed.

  By default, LSF tries to create the original cpuset when a job resumes. If this fails, LSF tries to create a new cpuset based on the default memory option.

  - ORIG_CPUS specifies that the job must be run on the original cpuset when it resumes. If this fails, the job remains suspended.

  Because memory migration is not supported on Altix for ProPack versions 4 or lower, you must define RESUME_OPTION=ORIG_CPUS to force LSF to recreate the original cpuset when LSF resumes a job.

- `CPU_LIST=`*`cpu_ID_list`*`;`

  cpu_ID_list is a list of CPU IDs separated by commas. The CPU ID is a positive integer or a range of integers. If incorrect CPU IDs are specified, the job remains pending until the specified CPUs are available.

  You must specify at least as many CPU IDs as the number of CPUs the job requires (`bsub -n`). If you specify more CPU IDs than the job requests, LSF selects the best CPUs from the list.
- `CPUSET_OPTIONS=`*`option_list`*`;`

  option_list is a list of cpuset attributes joined by a pipe (|). If incorrect cpuset attributes are specified, the job is rejected. See "Cpuset attributes" on page 104 for supported cpuset options.
- `MAX_CPU_PER_NODE=max_num_cpus;`

  max_num_cpus is the maximum number of CPUs on any one node that will be used by this job. Cannot be used with the NODE_EX option.
- `MEM_LIST=`*`mem_node_list`*`;`

  (Altix ProPack 4 and ProPack 5) *mem_node_list* is a list of memory node IDs separated by commas. The memory node ID is a positive integer or a range of integers. For example:

  ```
  "CPUSET[MEM_LIST=0,1-2]"
  ```

  Incorrect memory node IDs or unavailable memory nodes are ignored when LSF allocates the cpuset.
- `NODE_EX=Y | N;`

  Allocates whole nodes for the cpuset job. This option cannot be used with the MAX_CPU_PER_NODE option.

When a job is submitted using `-extsched`, LSF creates a cpuset with the specified CPUs and cpuset attributes and attaches it to the processes of the job. The job is then scheduled and dispatched.

Running jobs on specific CPUs

The CPUs available for your jobs may have specific features you need to take advantage of (for example, some CPUs may have more memory, others have a faster processor). You can partition your machines to use specific CPUs for your jobs, but the cpusets for your jobs cannot cross hosts, and you must run multiple operating systems

You can create static cpusets with the particular CPUs your jobs need, but you cannot control the specific CPUs in the cpuset that the job actually uses.

A better solution is to use the CPU_LIST external scheduler option to request specific CPUs for your jobs. LSF can choose the best set of CPUs from the CPU list to create a cpuset for the job. The best cpuset is the one with the smallest CPU radius that meets the CPU requirements of the job. CPU radius is determined by the processor topology of the system and is expressed in terms of the number of router hops between CPUs.

CPU_LIST requirements

To make job submission easier, you should define queues with the specific CPU_LIST requirements. Set CPU_LIST in MANDATORY_EXTSCHED or DEFAULT_EXTSCHED option in your queue definitions in lsb.queues.

span[ptile] resource requirement

CPU_LIST is interpreted as a list of *possible* CPU selections, not a strict requirement. For example, if you subit a job with the the `-R "span[ptile]"` option:

```
bsub -R "span[ptile=1]" -ext "CPUSET[CPU_LIST=1,3]" -n2 ...
```

the following combination of CPUs is possible:

CPUs on host 1	CPUs on host 2
1	1
1	3
3	1
3	3

Cpuset attributes

The following cpuset attributes are supported in the list of cpuset options specified by CPUSET_OPTIONS:

- CPUSET_CPU_EXCLUSIVE—defines a restricted cpuset
- CPUSET_MEMORY_LOCAL—threads assigned to the cpuset attempt to assign memory only from nodes within the cpuset. Overrides the MEM_LIST cpuset option.
- CPUSET_MEMORY_EXCLUSIVE—threads not assigned to the cpuset do not use memory from within the cpuset unless no memory outside the cpuset is available
- CPUSET_MEMORY_KERNEL_AVOID—kernel attempts to avoid allocating memory from nodes contained in this cpuset
- CPUSET_MEMORY_MANDATORY—kernel limits all memory allocations to nodes contained in this cpuset
- CPUSET_POLICY_PAGE—Causes the kernel to page user pages to the swap file to free physical memory on the nodes contained in this cpuset. This is the default policy if no other policy is specified. Requires CPUSET_MEMORY_MANDATORY.
- CPUSET_POLICY_KILL—The kernel attempts to free as much space as possible from kernel heaps, but will not page user pages to the swap file. Requires CPUSET_MEMORY_MANDATORY.

See the SGI resource administration documentation and the man pages for the `cpuset` command for information about these cpuset attributes.

SGI Altix

- **SGI Altix Linux ProPack versions 4 and lower** do not support memory migration; you must define RESUME_OPTION=ORIG_CPUS to force LSF to recreate the original cpuset when LSF resumes a job.
- **SGI Altix Linux ProPack 5** supports memory migration and does not require additional configuration to enable this feature. If you submit and then suspend a job using a dynamic cpuset, LSF will create a new dynamic cpuset when the job resumes. The memory pages for the job are migrated to the new cpuset as required.
- **SGI Altix Linux ProPack 3** only supports CPUSET_OPTIONS=CPUSET_MEMORY_LOCAL. If the cpuset job runs on an Altix host, other cpuset attributes are ignored.

- **SGI Altix Linux ProPack 4 and ProPack 5** do not support CPUSET_OPTIONS=CPUSET_MEMORY_MANDATORY or CPUSET_OPTIONS=CPUSET_CPU_EXCLUSIVE attributes. If the cpuset job runs on an Altix host, the cpusets created on the Altix system will have their memory usage restricted to the memory nodes containing the CPUs assigned to the cpuset. The CPUSET_MEMORY_MANDATORY and CPUSET_CPU_EXCLUSIVE attributes are ignored.

Restrictions on CPUSET_MEMORY_MANDATORY

- CPUSET_OPTIONS=CPUSET_MEMORY_MANDATORY implies node-level allocation
- CPUSET_OPTIONS=CPUSET_MEMORY_MANDATORY cannot be used together with MAX_CPU_PER_NODE=*max_num_cpus*

Restrictions on CPUSET_CPU_EXCLUSIVE

The scheduler will not use CPU 0 when determining an allocation on IRIX or TRIX. You must not include CPU 0 in the list of CPUs specified by CPU_LIST.

MPI_DSM_MUSTRUN environment variable

You should not use the MPI_DSM_MUSTRUN=ON environment variable. If a job is suspended through preemption, LSF can ensure that cpusets are recreated with the same CPUs, but it cannot ensure that a certain task will run on a specific CPU. Jobs running with MPI_DSM_MUSTRUN cannot migrate to a different part of the machine. MPI_DSM_MUSTRUN also interferes with job checkpointing.

Including memory nodes in the allocation (Altix ProPack4 and Propack 5)

When you specify a list of memory node IDs with the cpuset external scheduler option MEM_LIST, LSF creates a cpuset for the job that includes the memory nodes specified by MEM_LIST in addition to the local memory attached to the CPUs allocated for the cpuset. For example, if `"CPUSET[MEM_LIST=30-40]"`, and a 2-CPU parallel job is scheduled to run on CPU 0-1 (physically located on node 0), the job is able to use memory on node 0 and nodes 30-40.

Unavailable memory nodes listed in MEM_LIST are ignored when LSF allocates the cpuset. For example, a 4-CPU job across two hosts (hostA and hostB) that specifies MEM_LIST=1 allocates 2 CPUs on each host. The job is scheduled as follows:

- CPU 0 and CPU 1 (memory=node 0, node 1) on hostA
- CPU 0 and CPU 1 (memory=node 0, node 1) on hostB

If hostB only has 2 CPUs, only node 0 is available, and the job will only use the memory on node 0.

MEM_LIST is only available for dynamic cpuset jobs at both the queue level and the command level.

CPUSET_MEMORY_LOCAL

When both MEM_LIST and CPUSET_OPTIONS=CPUSET_MEMORY_LOCAL are both specified for the job, the root cpuset nodes are included as the memory nodes for the cpuset. MEM_LIST is ignored, and CPUSET_MEMORY_LOCAL overrides MEM_LIST.

CPU radius and processor topology

If LSB_CPUSET_BESTCPUS is set in `lsf.conf`, LSF can choose the best set of CPUs that can create a cpuset. The best cpuset is the one with the smallest CPU radius that meets the CPU requirements of the job. CPU radius is determined by the processor topology of the system and is expressed in terms of the number of router hops between CPUs.

For better performance, CPUs connected by metarouters are given a relatively high weights so that they are the last to be allocated

Best-fit and first-fit CPU list

By default, LSB_CPUSET_BESTCPUS=Y is set in `lsf.conf`. LSF applies a best-fit algorithm to select the best CPUs available for the cpuset.

Example For example, the following command creates an exclusive cpuset with the 8 best CPUs if available:

```
bsub -n 8 -extsched "CPUSET[CPUSET_OPTIONS=CPUSET_CPU_EXCLUSIVE]" myjob
```

If LSB_CPUSET_BESTCPUS is not set in `lsf.conf`, LSF builds a CPU list on a first-fit basis; in this example, the first 8 available CPUs are used.

Maximum radius for dynamic cpusets

Use the MAX_RADIUS cpuset external scheduler option to specify the maximum radius for dynamic cpuset allocation. If LSF cannot allocate a cpuset with radius less than or equal to MAX_RADIUS, the job remains pending.

`MAX_RADIUS` implies that the job cannot span multiple hosts. LSF puts each cpuset host into its own group to enforce this when MAX_RADIUS is specified.

How the best CPUs are selected

CPU_LIST	MAX_RADIUS	LSB_CPUSET_BESTCPUS	Algorithm used	Applied to
specified	specified or not specified	N	first fit	cpus in CPU_LIST
not specified	specified or not specified	N	first fit	all cpus in system
specified	specified	Y	max radius	cpus in CPU_LIST
not specified	specified	Y	max radius	all cpus in system
specified	not specified	Y	best fit	cpus in CPU_LIST
not specified	not specified	Y	best fit	all cpus in system

Allocating cpusets on multiple hosts (Altix only)

On SGI Altix systems, if a single host cannot satisfy the cpuset requirements for the job, LSF will try to allocate cpusets on multiple hosts, and the parallel job will be launched within the cpuset.

If you define the external scheduler option `CPUSET[CPUSET_TYPE=none]`, no cpusets are allocated and the job is dispatched and run outside of any cpuset.

Spanning multiple hosts is not supported on TRIX. Platform HPC creates cpusets on a single host (or on the first host in the allocation.)

LSB_HOST_CPUSETS environment variable

After dynamic cpusets are allocated and before the job starts running LSF sets the LSB_HOST_CPUSETS environment variable. LSB_HOST_CPUSETS has the following format:

```
number_hosts host1_name cpuset1_name host2_name
cpuset2_name ...
```

For example, if `hostA` and `hostB` have 2 CPUs, and `hostC` has 4 CPUs, cpuset `1-0` is created on `hostA`, `hostB` and `hostC`, and LSB_HOST_CPUSETS set to:

```
3 hostA 1-0 hostB 1-0 hostC 1-0
```

LSB_HOST_CPUSETS is only set for jobs that allocate dynamic cpusets.

LSB_CPUSET_DEDICATED environment variable

When a static or dynamic cpuset is allocated, LSF sets the LSB_CPUSET_DEDICATED environment variable. For `CPUSET_TYPE=none`, LSB_CPUSET_DEDICATED is not set.

The LSB_CPUSET_DEDICATED variable is set by LSF as follows:

- For `CPUSET_TYPE=dynamic` cpusets, LSB_CPUSET_DEDICATED=YES.

  This implies MPI_DISTRIBUTE=ON to get good NUMA placement in MPI jobs. The cpusets assigned to this job are not intended to be shared with other jobs or other users.
- For `CPUSET_TYPE=static` cpusets, LSB_CPUSET_DEDICATED=NO.

  Static cpusets are typically used to run a number of jobs concurrently. The cpusets assigned to this job are intended to be shared with other jobs, or it is unknown whether the cpusets assigned are intended to be shared.

How cpuset jobs are suspended and resumed

When a cpuset job is suspended (for example, with bstop), job processes are moved out of the cpuset and the job cpuset is destroyed. LSF keeps track of which processes belong to the cpuset, and attempts to recreate a job cpuset when a job is resumed, and binds the job processes to the cpuset.

When a job is resumed, regardless of how it was suspended, the RESUME_OPTION is honored. If RESUME_OPTION=ORIG_CPUS then LSF first tries to get the original CPUs from the same nodes as the original cpuset in order to use the same memory. If this does not get enough CPUs to resume the job, LSF tries to get any CPUs in an effort to get the job resumed.

SGI Altix Linux ProPack 5 supports memory migration and does not require additional configuration to enable this feature. If you submit and then suspend a job using a dynamic cpuset, LSF will create a new dynamic cpuset when the job resumes. The memory pages for the job are migrated to the new cpuset as required.

Example Assume a host with 2 nodes, 2 CPUs per node (total of 4 CPUs)

Node	CPUs	
0	0	1
1	2	3

When a job running within a cpuset that contains cpu 1 is suspended:

1. The job processes are detached from the cpuset and suspended
2. The cpuset is destroyed

When the job is resumed:

1. A cpuset with the same name is recreated
2. The processes are resumed and attached to the cpuset

The RESUME_OPTION parameter determines which CPUs are used to recreate the cpuset:

- If RESUME_OPTION=ORIG_CPUS, only CPUs from the same nodes originally used are selected.
- If RESUME_OPTION is not ORIG_CPUS LSF will first attempt to use cpus from the original nodes to minimize memory latency. If this is not possible, any free CPUs from the host will be considered.

If the job originally had a cpuset containing cpu 1, the possibilities when the job is resumed are:

RESUME_OPTION	Eligible CPUs			
ORIG_CPUS	0	1		
not ORIG_CPUS	0	1	2	3

Viewing cpuset information for your jobs

bacct, bjobs, bhist The `bacct -l`, `bjobs -l`, and `bhist -l` commands display the following information for jobs:

- CPUSET_TYPE=static | dynamic | none
- NHOSTS=*number*
- HOST=*host_name*
- CPUSET_NAME=*cpuset_name*
- NCPUS=*num_cpus*—the number of actual CPUs in the cpuset; can be greater than the number of slots

```
bjobs -l 221

Job <221>, User <user1>, Project <default>, Status <DONE>, Queue <normal>, Com
                     mand <myjob>
Thu Dec 15 14:19:54 2009: Submitted from host <hostA>, CWD <$HOME
                     >, 2 Processors Requested;
Thu Dec 15 14:19:57 2009: Started on 2 Hosts/Processors <2*hostA>
                     , Execution Home </home/user1>, Execution CWD
                     </home/user1>;
Thu Dec 15 14:19:57 2009: CPUSET_TYPE=dynamic;NHOSTS=1;HOST=hostA;CPUSET_NAME=
                     /reg62@221;NCPUS=2;
Thu Dec 15 14:20:03 2009: Done successfully. The CPU time used is 0.0 seconds.

 SCHEDULING PARAMETERS:
           r15s   r1m  r15m   ut      pg    io   ls    it    tmp    swp   mem
 loadSched   -     -     -     -       -     -    -     -     -      -     -
 loadStop    -     -     -     -       -     -    -     -     -      -     -

 EXTERNAL MESSAGES:
 MSG_ID FROM       POST_TIME      MESSAGE              ATTACHMENT
 0        -           -              -                     -
 1        -           -              -                     -
 2      root       Dec 15 14:19   JID=0x118f; ASH=0x0      N

bhist -l 221
Job <221>, User <user1>, Project <default>, Command <myjob>
Thu Dec 15 14:19:54 2009: Submitted from host <hostA>, to Queue <
                     normal>, CWD <$HOME>, 2 Processors Requested;
Thu Dec 15 14:19:57 2009: Dispatched to 2 Hosts/Processors <2*hostA>;
Thu Dec 15 14:19:57 2009: CPUSET_TYPE=dynamic;NHOSTS=1;HOST=hostA
                     ;CPUSET_NAME=/reg62@221;NCPUS=2;
Thu Dec 15 14:19:57 2009: Starting (Pid 4495);
Thu Dec 15 14:19:57 2009: External Message "JID=0x118f; ASH=0x0" was posted
from "ro
                     ot" to message box 2;
Thu Dec 15 14:20:01 2009: Running with execution home </home/user1>, Execution
CWD
                     </home/user1>, Execution Pid <4495>;
Thu Dec 15 14:20:01 2009: Done successfully. The CPU time used is 0.0 seconds;
Thu Dec 15 14:20:03 2009: Post job process done successfully;

Summary of time in seconds spent in various states by  Thu Dec 15 14:20:03
  PEND     PSUSP    RUN      USUSP    SSUSP    UNKWN    TOTAL
  3        0        4        0        0        0        7

bacct -l 221
Accounting information about jobs that are:
  - submitted by all users.
  - accounted on all projects.
  - completed normally or exited
  - executed on all hosts.
  - submitted to all queues.
  - accounted on all service classes.
------------------------------------------------------------------------------

Job <221>, User <user1>, Project <default>, Status <DONE>, Queue <normal>, Com
                     mand <myjob>
Thu Dec 15 14:19:54 2009: Submitted from host <hostA>, CWD <$HOME>;
Thu Dec 15 14:19:57 2009: Dispatched to 2 Hosts/Processors <2*hostA>;
Thu Dec 15 14:19:57 2009: CPUSET_TYPE=dynamic;NHOSTS=1;HOST=hostA;CPUSET_NAME=
                     /reg62@221;NCPUS=2;
Thu Dec 15 14:20:01 2009: Completed <done>.
```

```
Accounting information about this job:
     CPU_T     WAIT     TURNAROUND   STATUS     HOG_FACTOR    MEM    SWAP
      0.03        3              7     done         0.0042     0K      0K
------------------------------------------------------------------------------

SUMMARY:      ( time unit: second )
 Total number of done jobs:       1      Total number of exited jobs:     0
 Total CPU time consumed:       0.0      Average CPU time consumed:     0.0
 Maximum CPU time of a job:     0.0      Minimum CPU time of a job:     0.0
 Total wait time in queues:     3.0
 Average wait time in queue:    3.0
 Maximum wait time in queue:    3.0      Minimum wait time in queue:    3.0
 Average turnaround time:         7 (seconds/job)
 Maximum turnaround time:         7      Minimum turnaround time:         7
 Average hog factor of a job:  0.00 ( cpu time / turnaround time )
 Maximum hog factor of a job:  0.00      Minimum hog factor of a job:  0.00
```

brlainfo Use `brlainfo` to display topology information for a cpuset host. It displays

- Cpuset host name
- Cpuset host type
- Total number of CPUs
- Free CPUs
- Total number of nodes
- Free CPUs per node
- Available CPUs with a given radius
- List of static cpusets

```
brlainfo
HOSTNAME            CPUSET_OS   NCPUS  NFREECPUS NNODES  NCPU/NODE NSTATIC_CPUSETS
hostA               SGI_TRIX    2      2         1       2         0
hostB               PROPACK_4   4      4         2       2         0
hostC               PROPACK_4   4      3         2       2         0
```

```
brlainfo -l
HOST: hostC
CPUSET_OS    NCPUS  NFREECPUS NNODES  NCPU/NODE NSTATIC_CPUSETS
PROPACK_4    4      3         2       2         0
FREE CPU LIST: 0-2
NFREECPUS ON EACH NODE: 2/0,1/1
STATIC CPUSETS: NO STATIC CPUSETS
CPU_RADIUS: 2,3,3,3,3,3,3,3
```

Examples

- Specify a dynamic cpuset:

```
bsub -n 8 -extsched "CPUSET[CPUSET_TYPE=dynamic;CPU_LIST=1, 5, 7-12;]" myjob
```

If CPUSET_TYPE is not specified, the default cpuset type is dynamic:

```
bsub -R "span[hosts=1]" -n 8 -extsched "CPUSET[CPU_LIST=1, 5, 7-12;]" myjob
```

Jobs are attached to a cpuset dynamically created by LSF. The cpuset is deleted when the job finishes or exits.

- Specify a list of CPUs for an exclusive cpuset:

```
bsub -n 8 -extsched "CPUSET[CPU_LIST=1, 5, 7-12;
CPUSET_OPTIONS=CPUSET_CPU_EXCLUSIVE|CPUSET_MEMORY_LOCAL]" myjob
```

The job `myjob` will succeed if CPUs 1, 5, 7, 8, 9, 10, 11, and 12 are available.

- Specify a static cpuset:

```
bsub -n 8 -extsched "CPUSET[CPUSET_TYPE=static; CPUSET_NAME=MYSET]" myjob
```

Specifying a cpuset name implies that the cpuset type is static:

```
bsub -n 8 -extsched "CPUSET[CPUSET_NAME=MYSET]" myjob
```

Jobs are attached to a static cpuset specified by users at job submission. This cpuset is *not* deleted when the job finishes or exits.

- Run a job without using any cpuset:

```
bsub -n 8 -extsched "CPUSET[CPUSET_TYPE=none]" myjob
```

Using preemption

- Jobs requesting static cpusets:

```
bsub -n 4 -q low rusage[scpus=4]" -extsched "CPUSET[CPUSET_NAME=MYSET]"
sleep 1000
```

```
bsub -n 4 -q low rusage[scpus=4]" -extsched "CPUSET[CPUSET_NAME=MYSET]"
sleep 1000
```

After these two jobs start running, submit a job to a high priority queue:

```
bsub -n 4 -q high rusage[scpus=4]" -extsched "CPUSET[CPUSET_NAME=MYSET]"
sleep 1000
```

The most recent job running on the low priority queue (job 102) is preempted by the job submitted to the high priority queue (job 103):

```
bjobs
JOBID   USER    STAT  QUEUE      FROM_HOST   EXEC_HOST  JOB_NAME SUBMIT_TIME
103     user1   RUN   high       hosta       4*hosta    *eep 1000 Jan 22 08:24
101     user1   RUN   low        hosta       4*hosta    *eep 1000 Jan 22 08:23
102     user1   SSUSP low        hosta       4*hosta    *eep 1000 Jan 22 08:23
```

```
bhosts -s
RESOURCE                 TOTAL      RESERVED
LOCATION
dcpus                      4.0           0.0       hosta
scpus                      0.0           8.0       hosta
```

- Jobs request dynamic cpusets:

```
bsub -q high rusage[dcpus=1]" -n 3 -extsched "CPUSET[CPU_LIST=1,2,3]" sleep
1000
```

```
bhosts -s
RESOURCE                 TOTAL      RESERVED
LOCATION
dcpus                      3.0           1.0       hosta
scpus                      8.0           0.0       hosta
```

Using SGI Comprehensive System Accounting facility (CSA)

The SGI Comprehensive System Accounting facility (CSA) provides data for collecting per-process resource usage, monitoring disk usage, and chargeback to specific login accounts. If is enabled on your system, LSF writes records for LSF jobs to CSA.

SGI CSA writes an accounting record for each process in the `pacct` file, which is usually located in the `/var/adm/acct/day` directory. SGI system administrators then use the `csabuild` command to organize and present the records on a job by job basis.

For each job running on the SGI system, LSF writes an accounting record to CSA when the job starts and when the job finishes. LSF daemon accounting in CSA starts and stops with the LSF daemon.

See the SGI resource administration documentation for information about CSA.

Setting up SGI CSA

1. Set the following parameters in `/etc/csa.conf` to `on`:
    - CSA_START
    - WKMG_START
2. Run the `csaswitch` command to turn on the configuration changes in `/etc/csa.conf`.

See the SGI resource administration documentation for information about the `csaswitch` command.

Information written to the pacct file

LSF writes the following records to the `pacct` file when a job starts and when it exits:

- Job record type (job start or job exit)
- Current system clock time
- Service provider (LSF)
- Submission time of the job (at job start only)
- User ID of the job owner
- Array Session Handle (ASH) of the job (not available on Altix)
- SGI job container ID (PAGG job ID on Altix)
- SGI project ID (not available on Altix)
- LSF job name if it exists
- Submission host name
- LSF queue name
- LSF external job ID
- LSF job array index
- LSF job exit code (at job exit only)
- NCPUS—number of CPUs the LSF job has been using

Viewing LSF job information recorded in CSA

Use the SGI `csaedit` command to see the ASCII content of the `pacct` file. For example:

```
# csaedit -P /var/csa/day/pacct -A
```

For each LSF job, you should see two lines similar to the following:

```
--------------------------------------------------------------------------------
---------
37   Raw-Workld-Mgmt  user1    0x19ac91ee000064f2 0x0000000000000000       0
REQID=1771  ARRAYID=0  PROV=LSF  START=Jun  4 15:52:01  ENTER=Jun  4 15:51:49
TYPE=INIT  SUBTYPE=START  MACH=hostA  REQ=myjob  QUE=normal
…
39   Raw-Workld-Mgmt  user1    0x19ac91ee000064f2 0x0000000000000000       0
REQID=1771  ARRAYID=0  PROV=LSF  START=Jun  4 16:09:14  TYPE=TERM  SUBTYPE=EXIT
MACH=hostA  REQ=myjob  QUE=normal--
--------------------------------------------------------------------------------
---------
```

The REQID is the LSF job ID (1771).

See the SGI resource administration documentation for information about the `csaedit` command.

Using SGI User Limits Database (ULDB—IRIX only)

The SGI user limits database (ULDB) allows user-specific limits for jobs. If no ULDB is defined, job limits are the same for all jobs. If you use ULDB, you can configures LSF so that jobs submitted to a host with the SGI job limits package installed are subject to the job limits configured in the ULDB.

Set LSF_ULDB_DOMAIN=*domain_name* in `lsf.conf` to specify the name of the LSF domain in the ULDB domain directive. A domain definition of name *domain_name* must be configured in the `jlimit.in` input file.

The ULDB contains job limit information that system administrators use to control access to a host on a per user basis. The job limits in the ULDB override the system default values for both job limits and process limits. When a ULDB domain is configured, the limits will be enforced as SGI job limits.

If the ULDB domain specified in LSF_ULDB_DOMAIN is not valid or does not exist, LSF uses the limits defined in the domain named `batch`. If the `batch` domain does not exist, then the system default limits are set.

When an LSF job is submitted, an SGI job is created, and the job limits in the ULDB are applied.

Next, LSF resource usage limits are enforced for the SGI job under which the LSF job is running. LSF limits override the corresponding SGI job limits. The ULDB limits are used for any LSF limits that are not defined. If the job reaches the SGI job limits, the action defined in the SGI system is used.

SGI job limits in the ULDB apply only to batch jobs.

You can also define resource limits (rlimits) in the ULDB domain. One advantage to defining rlimits in ULDB as opposed to in LSF is that rlimits can be defined per user and per domain in ULDB, whereas in LSF, limits are enforced per queue or per job.

See the SGI resource administration documentation for information about configuring ULDB domains in the `jlimit.in` file.

SGI Altix

SGI ULDB is not supported on Altix systems, so no process aggregate (PAGG) job-level resource limits are enforced for jobs running on Altix. Other operating system and LSF resource usage limits are still enforced.

LSF resource usage limits controlled by ULDB job limits

- PROCESSLIMIT—Corresponds to SGI JLIMIT_NUMPROC; `fork`(2) fails, but the existing processes continue to run
- MEMLIMIT—Corresponds to JLIMIT_RSS; Resident pages above the limit become prime swap candidates
- DATALIMIT—Corresponds to LIMIT_DATA; malloc(3) calls in the job fail with errno set to ENOMEM
- CPULIMIT—Corresponds to JLIMIT_CPU; a SIGXCPU signal is sent to the job, then after the grace period expires, SIGINT, SIGTERM, and SIGKILL are sent
- FILELIMIT—No corresponding limit; use process limit RLIMIT_FSIZE
- STACKLIMIT—No corresponding limit; use process limit RLIMIT_STACK
- CORELIMIT—No corresponding limit; use process limit RLIMIT_CORE

- SWAPLIMIT—Corresponds to JLIMIT_VMEM; use process limit RLIMIT_VMEM

Increasing the default MEMLIMIT for ULDB

In some pre-defined LSF queues, such as `normal`, the default MEMLIMIT is set to 5000 (5 MB). However, if ULDB is enabled (LSF_ULDB_DOMAIN is defined) the MEMLIMIT should be set greater than 8000 in `lsb.queues`.

Example ULDB domain configuration

The following steps enable the ULDB domain `LSF` for user `user1`:

1. Define the LSF_ULDB_DOMAIN parameter in `lsf.conf`:

```
...
LSF_ULDB_DOMAIN=LSF
...
```

Note You can set the LSF_ULDB_DOMAIN to include more than one domain. For example:
`LSF_ULDB_DOMAIN="lsf:batch:system"`

2. Configure the domain directive `LSF` in the `jlimit.in` file:

```
domain <LSF> {                              # domain for LSF
        jlimit_numproc_cur = unlimited
        jlimit_numproc_max = unlimited      # JLIMIT_NUMPROC
        jlimit_nofile_cur = unlimited
        jlimit_nofile_max = unlimited       # JLIMIT_NOFILE
        jlimit_rss_cur = unlimited
        jlimit_rss_max = unlimited          # JLIMIT_RSS
        jlimit_vmem_cur = 128M
        jlimit_vmem_max = 256M              # JLIMIT_VMEM
        jlimit_data_cur = unlimited
        jlimit_data_max =unlimited          # JLIMIT_DATA
        jlimit_cpu_cur = 80
        jlimit_cpu_max = 160                # JLIMIT_CPU
}
```

3. Configure the user limit directive for `user1` in the `jlimit.in` file:

```
user user1 {
        LSF {
           jlimit_data_cur = 128M
           jlimit_data_max = 256M
         }
}
```

4. Use the IRIX `genlimits` command to create the user limits database:

```
genlimits -l -v
```

SGI Job Container and Process Aggregate Support

An SGI job contains all processes created in a login session, including array sessions and session leaders. Job limits set in ULDB are applied to SGI jobs either at creation time or through the lifetime of the job. Job limits can also be reset on a job during its lifetime.

SGI IRIX job containers

If SGI Job Limits is installed, LSF creates a job container when starting a job, uses the job container to signal all processes in the job, and uses the SGI job ID to collect job resource usage for a job.

If LSF_ULDB_DOMAIN is defined in `lsf.conf`, ULDB job limits are applied to the job.

The SGI job ID is also used for kernel-level checkpointing.

SGI Altix Process Aggregates (PAGG)

Similar to an SGI job container, a process aggregate (PAGG) is a collection of processes. A child process in a PAGG inherits membership, or attachment, to the same process aggregate containers as the parent process. When a process inherits membership, the process aggregate containers are updates for the new process member. When a process exits, the process leaves the set of process members and the aggregate containers are updated again.

SGI Altix

Since SGI ULDB is not supported on Altix systems, no PAGG job-level resource limits are enforced for jobs running on Altix. Other operating system level and LSF resource limits are still enforced.

Viewing SGI job ID and Array Session Handle (ASH)

Use `bjobs` and `bhist` to display SGI job ID and Array Session Handle.

SGI Altix

On Altix systems, the array session handle is not available. It is displayed as ASH=0x0.

```
bjobs -l 640
Job <640>, User <user1>, Project <default>, Status <RUN>, Queue <normal>,
                     Command <pam -mpi -auto_place myjob>
Tue Jan 20 12:37:18 2009: Submitted from host <hostA>, CWD <$HOME>, 2
Processors Re
                     quested;
Tue Jan 20 12:37:29 2009: Started on 2 Hosts/Processors <2*hostA>,
                     Execution Home </home/user1>, Execution CWD </home/user1>;
Tue Jan 20 12:37:29 2009: CPUSET_TYPE=dynamic;NHOSTS=1;ALLOCINFO=hostA 640-0;
Tue Jan 20 12:38:22 2009: Resource usage collected.
                     MEM: 1 Mbytes;  SWAP: 5 Mbytes;  NTHREAD: 1
                     PGID: 5020232;  PIDs: 5020232


 SCHEDULING PARAMETERS:
           r15s   r1m  r15m   ut      pg    io   ls    it    tmp    swp    mem
 loadSched   -     -     -     -       -     -    -     -     -      -      -
 loadStop    -     -     -     -       -     -    -     -     -      -      -
 EXTERNAL MESSAGES:
 MSG_ID FROM       POST_TIME      MESSAGE                            ATTACHMENT

 0         -            -                        -                        -
 1         -            -                        -                        -
 2      root       Jan 20 12:41   JID=0x2bc0000000001f7a; ASH=0x2bc0f     N
```

```
bhist -l 640
Job <640>, User <user1>, Project <default>, Command
                     <pam -mpi -auto_place myjob>
Sat Oct 19 14:52:14 2009: Submitted from host <hostA>, to Queue <normal>, CWD
                     <$HOME>, Requested Resources <unclas>;
Sat Oct 19 14:52:22 2009: Dispatched to <hostA>;
Sat Oct 19 14:52:22 2009: CPUSET_TYPE=none;NHOSTS=1;ALLOCINFO=hostA;
Sat Oct 19 14:52:23 2009: Starting (Pid 5020232);
Sat Oct 19 14:52:23 2009: Running with execution home </home/user1>, Execution
CWD
                     </home/user1>, Execution Pid <5020232>;
Sat Oct 19 14:53:22 2009: External Message "JID=0x2bc0000000001f7a;
ASH=0x2bc0f" was
                     posted from "root" to message box 2;

Summary of time in seconds spent in various states by  Sat Oct 19 14:54:00
  PEND     PSUSP    RUN      USUSP    SSUSP    UNKWN    TOTAL
  8        0        98       0        0        0        106
```

CHAPTER

6

Using Platform LSF with LAM/MPI

Contents

About Platform LSF and LAM/MPI

LAM (Local Area Multicomputer) is an MPI programming environment and development system for heterogeneous computers on a network. With LAM, a dedicated cluster or an existing network computing infrastructure can act as one parallel computer solving one problem.

System requirements

- LAM/MPI version 6.5.7 or higher

Assumptions

- LAM/MPI is installed and configured correctly
- The user's current working directory is part of a shared file system reachable by all hosts

Glossary

LAM	(Local Area Multicomputer) An MPI programming environment and development system for heterogeneous computers on a network.
MPI	(Message Passing Interface) A message passing standard. It defines a message passing API useful for parallel and distributed applications.
PAM	(Parallel Application Manager) The supervisor of any parallel job.
PJL	(Parallel Job Launcher) Any executable script or binary capable of starting parallel tasks on all hosts assigned for a parallel job.
RES	(Remote Execution Server) An LSF daemon residing on each host. It monitors and manages all LSF tasks on the host.
TS	(TaskStarter) An executable responsible for starting a task on the local host and reporting the process ID and host name to the PAM.

Files installed by lsfinstall

During installation, `lsfinstall` copies these files to the following directories:

These files...	Are installed to...
`TaskStarter`	`LSF_BINDIR`
`pam`	`LSF_BINDIR`
`esub.lammpi`	`LSF_SERVERDIR`
`lammpirun_wrapper`	`LSF_BINDIR`
`mpirun.lsf`	`LSF_BINDIR`
`pjllib.sh`	`LSF_BINDIR`

Resources and parameters configured by lsfinstall

- External resources in `lsf.shared`:

```
Begin Resource
RESOURCE_NAME  TYPE    INTERVAL  INCREASING     DESCRIPTION
...
lammpi         Boolean    ()         ()          (LAM MPI)
...
End Resources
```

The `lammpi` Boolean resource is used for mapping hosts with LAM/MPI available.

You should add the `lammpi` resource name under the RESOURCES column of the Host section of `lsf.cluster.`*`cluster_name`*.

- Parameter to `lsf.conf`:

  `LSB_SUB_COMMANDNAME=y`

Configuring LSF to work with LAM/MPI

System setup

1. For troubleshooting LAM/MPI jobs, edit the `LSF_BINDIR/lammpirun_wrapper` script, and specify a log directory that all users can write to. For example:

   ```
   LOGDIR="/mylogs"
   ```

   Do not use LSF_LOGDIR for this log directory.

2. Add the LAM/MPI home directory to your path. The LAM/MPI home directory is the directory that you specified as the prefix during LAM/MPI installation.
3. Add the path to the LAM/MPI commands to the $PATH variable in your shell startup files (`$HOME/.cshrc` or `$HOME/.profile`).
4. Edit `lsf.cluster.`*`cluster_name`* and add the `lammpi` resource for each host with LAM/MPI available. For example:

   ```
   Begin    Host
   HOSTNAME  model   type    server r1m   mem   swp   RESOURCES
   ...
   hosta       !       !         1   3.5   ()   ()     (lammpi)
   ...
   End      Host
   ```

Submitting LAM/MPI Jobs

bsub command

Use `bsub` to submit LAM/MPI jobs:

`bsub -a lammpi -n` *`number_cpus`* `[`**`-q`** *`queue_name`*`]` **`mpirun.lsf`**
`[`**`-pam "`***`pam_options`***`"`**`]` `[`*`mpi_options`*`]` *`job`* `[`*`job_options`*`]`

- **`-a lammpi`** tells esub the job is a LAM/MPI job and invokes `esub.lammpi`.
- **`-n`** *`number_cpus`* specifies the number of processors required to run the job
- **`-q`** *`queue_name`* specifies a LAM/MPI queue that is configured to use the custom termination action. If no queue is specified, the `hpc_linux` queue is used.
- **`mpirun.lsf`** reads the environment variable LSF_PJL_TYPE=lammpi set by `esub.lammpi`, and generates the appropriate `pam` command line to invoke LAM/MPI as the PJL

Examples

- `% bsub -a lammpi -n 3 -q hpc_linux mpirun.lsf /examples/cpi`
  A job named `cpi` is submitted to the `hpc_linux` queue. It will be dispatched and run on 3 CPUs in parallel.
- `% bsub -a lammpi -n 3 -R "select[mem>100] rusage[mem=100:duration=5]" -q hpc_linux mpirun.lsf /examples/cpi`
  A job named `cpi` is submitted to the `hpc_linux` queue. It will be dispatched and run on 3 CPUs in parallel. Memory is reserved for 5 minutes.

Submitting a job with a job script

A wrapper script is often used to call the LAM/MPI script. You can submit a job using a job script as an embedded script or directly as a job, for example:

```
% bsub -a lammpi -n 4 < embedded_jobscript
```

```
% bsub -a lammpi -n 4 jobscript
```

Your job script must use `mpirun.lsf` in place of the `mpirun` command.

For information on generic PJL wrapper script components, see Chapter 2, “Running Parallel Jobs”.

See *Administering Platform LSF* for information about submitting jobs with job scripts.

Job placement with LAM/MPI jobs

The `mpirun -np` option is ignored. You should use the LSB_PJL_TASK_GEOMETRY environment variable for consistency with other Platform LSF MPI integrations. LSB_PJL_TASK_GEOMETRY overrides the `mpirun -np` option.

The environment variable LSB_PJL_TASK_GEOMETRY is checked for all parallel jobs. If LSB_PJL_TASK_GEOMETRY is set users submit a parallel job (a job that requests more than 1 slot), LSF attempts to shape LSB_MCPU_HOSTS accordingly.

Log files

For troubleshooting LAM/MPI jobs, define LOGDIR in the `LSF_BINDIR/lammpirun_wrapper` script. Log files (`lammpirun_wrapper.job[`*`job_ID`*`].log`) are written to the LOGDIR directory. If LOGDIR is not defined, log messages are written to `/dev/null`.

For example, the log file for the job with job ID 123 is:

```
lammpirun_wrapper.job123.log
```

CHAPTER

7

Using Platform LSF with MPICH-GM

Contents

About Platform LSF and MPICH-GM

MPICH is a freely available, portable implementation of the MPI Standard for message-passing libraries, developed jointly with Mississippi State University. MPICH is designed to provide high performance, portability, and a convenient programming environment.

MPICH-GM is used with high performance Myrinet networks. Myrinet is a high-speed network which allows OS-bypass communications in large clusters. MPICH-GM integrates with Platform LSF so users can run parallel jobs on hosts with at least one free port.

Requirements

- MPICH version 1.2.6 or later

  You should upgrade all your hosts to the same version of MPICH-GM.

- GM versions 1.5.1, and 1.6.3 or later

Assumptions

- MPICH-GM is installed and configured correctly
- The user's current working directory is part of a shared file system reachable by all hosts

Glossary

MPI	(Message Passing Interface) A message passing standard. It defines a message passing API useful for parallel and distributed applications.
MPICH	A portable implementation of the MPI standard.
GM	A message based communication system developed for Myrinet.
MPICH-GM	An MPI implementation based on MPICH for Myrinet.
PAM	(Parallel Application Manager) The supervisor of any parallel job.
PJL	(Parallel Job Launcher) Any executable script or binary capable of starting parallel tasks on all hosts assigned for a parallel job.
RES	(Remote Execution Server) An LSF daemon residing on each host. It monitors and manages all LSF tasks on the host.
TS	(TaskStarter) An executable responsible for starting a task on the local host and reporting the process ID and host name to the PAM.

For more information

- See the Myricom Web site at `www.myrinet.com` for software distribution and documentation on Myrinet clusters.
- See the Mathematics and Computer Science Division (MCS) of Argonne National Laboratory (ANL) MPICH Web page at `www-unix.mcs.anl.gov/mpi/mpich/` for more information about MPICH.

Files installed by lsfinstall

During installation, `lsfinstall` copies these files to the following directories:

These files...	Are installed to...
`TaskStarter`	`LSF_BINDIR`
`pam`	`LSF_BINDIR`
`esub.mpich_gm`	`LSF_SERVERDIR`
`gmmpirun_wrapper`	`LSF_BINDIR`
`mpirun.lsf`	`LSF_BINDIR`
`pjllib.sh`	`LSF_BINDIR`

Resources and parameters configured by lsfinstall

- External resources in `lsf.shared`:

```
Begin Resource
RESOURCE_NAME   TYPE     INTERVAL   INCREASING     DESCRIPTION
...
mpich_gm        Boolean     ()          ()        (MPICH GM MPI)
...
End Resources
```

The `mpich_gm` Boolean resource is used for mapping hosts with MPICH-GM available.

You should add the `mpich_gm` resource name under the RESOURCES column of the Host section of `lsf.cluster.`*`cluster_name`*.

- Parameter to `lsf.conf`:

```
LSB_SUB_COMMANDNAME=y
```

Configuring LSF to Work with MPICH-GM

Configure GM port resources (optional)

If there are more processors on a node than there are available GM ports, you should configure the external static resource name `gm_ports` to limit the number of jobs that can launch on that node.

lsf.shared

Add the external static resource `gm_ports` in `lsf.shared` to keep track of the number of free Myrinet ports available on a host:

```
Begin Resource
RESOURCENAME  TYPE      INTERVAL INCREASING  RELEASE   DESCRIPTION
...
gm_ports      Numeric   ()       N           N   (number of free myrinet ports)
...
End Resource
```

lsf.cluster.cluster_name

Edit the resource map in `lsf.cluster.`*`cluster_name`* to configure hosts in the cluster able to collect `gm_ports`. For example, the following configures 13 GM ports available for GM 2.0 and 5 GM ports are available for mGM 1.x.

```
Begin ResourceMap
RESOURCENAME        LOCATION
...
gm_ports            13@[default]
...
End ResourceMap
```

lsb.resources

Configure the `gm_ports` resource as PER_SLOT in a ReservationUsage section in `lsb.resources`:

```
Begin ReservationUsage
RESOURCE      METHOD
...
gm_port       PER_SLOT
...
End ReservationUsage
```

gmmpirun_wrapper script

Modify the `gmmpirun_wrapper` script in `LSF_BINDIR` so that the `mpirun.ch_gm` command in the scripts point to:

MPIRUN_CMD="/*path*/mpirun.ch_gm"

where *path* is the path to the directory where the `mpirun.ch_gm` command is stored.

lsf.conf (optional)

LSF_STRIP_DOMAIN

If the `gm_board_info` command returns host names that include domain names you cannot define LSF_STRIP_DOMAIN in `lsf.conf`. If the `gm_board_info` command returns host names without domain names, but LSF commands return host names that include domain names, you must define LSF_STRIP_DOMAIN in `lsf.conf`.

Performance tuning

To improve performance and scalability for large parallel jobs, tune the following parameters as described in "Tuning PAM Scalability and Fault Tolerance" on page 41:

- LSF_HPC_PJL_LOADENV_TIMEOUT
- LSF_PAM_RUSAGE_UPD_FACTOR

The user's environment can override these.

Submitting MPICH-GM Jobs

bsub command

Use `bsub` to submit MPICH-GM jobs.

`bsub -a mpich_gm -n` *`number_cpus`* **`mpirun.lsf`**
`[`**`-pam`** `"`*`pam_options`*`"] [`*`mpi_options`*`]` *`job`* `[`*`job_options`*`]`

- **`-a mpich_gm`** tells `esub` the job is an MPICH-GM job and invokes `esub.mpich_gm`.
- **`-n`** *`number_cpus`* specifies the number of processors required to run the job
- **`mpirun.lsf`** reads the environment variable LSF_PJL_TYPE=mpich_gm set by `esub.mpich_gm`, and generates the appropriate `pam` command line to invoke MPICH-GM as the PJL

For example:

```
% bsub -a mpich_gm -n 3 mpirun.lsf /examples/cpi
```

A job named `cpi` will be dispatched and run on 3 CPUs in parallel.

To limit the number of jobs using GM ports, specify a resource requirement in your job submission:

`-R "rusage[gm_ports=1]`

Submitting a job with a job script

You can use a wrapper script to call the MPICH-GM job launcher. You can submit a job using a job script as an embedded script or directly as a job, for example:

```
% bsub -a mpich_gm -n 4 < embedded_jobscript
```

```
% bsub -a mpich_gm -n 4 jobscript
```

Your job script must use `mpirun.lsf` in place of the `mpirun` command.

For information on generic PJL wrapper script components, see Chapter 2, “Running Parallel Jobs”.

See *Administering Platform LSF* for information about submitting jobs with job scripts.

Using AFS with MPICH-GM

Complete the following steps only if you are planning to use AFS with MPICH-GM.

The MPICH-GM package contains an `esub.afs` file which combines the `esub` for MPICH-GM and the `esub` for AFS so that MPICH-GM and AFS can work together.

Steps

1. Install and configure LSF for AFS.
2. Edit `mpirun.ch_gm`. The location of this script is defined with the MPIRUN_CMD parameter in the script `LSF_BINDIR/gmmpirun_wrapper`.
3. Replace the following line:

   ```
   exec($rsh,'-n',$_,$cmd_ln);
   ```

   with:

   ```
   exec($lsrun,'-m',$_,'/bin/sh','-c',"$cmd_ln < /dev/null");
   ```

4. Add the following line to `mpirun.ch_gm` before the line `$rsh="rsh";` replacing $LSF_BINDIR by the actual path:

   ```
   $lsrun=”$LSF_BINDIR/lsrun”;
   $rsh="rsh";
   ```

   For example:

   ```
   $lsrun="/usr/local/lsf/7.0/linux2.4-glibc2.1-
   x86/bin/lsrun";
   ```

5. Comment out the following line:

   ```
   #$rsh="rsh";
   ```

6. Replace the following line:

   ```
   exec($rsh,$_,$cmdline);
   ```

   with:

   ```
   exec($lsrun,'-m',$_,'/bin/sh','-c',$cmdline);
   ```

7. Replace the following line:

   ```
   exec($rsh,'-n',$_,$cmdline);
   ```

   with:

   ```
   exec($lsrun,'-m',$_,'/bin/sh','-c',"$cmdline</dev/null");
   ```

8. Replace the following line:

   ```
   die "$rsh $_ $argv{$lnode}->[0]:$!\n"
   ```

   with:

   ```
   die "$lsrun -m $_ sh -c $argv{$lnode}->[0]:$!\n"
   ```

9. Save the `mpirun.ch_gm` file.

CHAPTER

8

Using Platform LSF with MPICH-P4

Contents

About Platform LSF and MPICH-P4

MPICH is a freely available, portable implementation of the MPI Standard for message-passing libraries, developed jointly with Mississippi State University. MPICH is designed to provide high performance, portability, and a convenient programming environment.

MPICH-P4 is an MPICH implementation for the ch_p4 device, which supports SMP nodes, MPMD programs, and heterogeneous collections of systems.

Requirements

- MPICH version 1.2.5 or later

You should upgrade all your hosts to the same version of MPICH-P4.

Assumptions and limitations

- MPICH-P4 is installed and configured correctly
- The user's current working directory is part of a shared file system reachable by all hosts
- The directory specified by the MPICH_HOME variable is accessible by the same path on all hosts
- Process group files are not supported. The `mpich.ch_p4 p4pg` option is ignored.

Glossary

MPI	(Message Passing Interface) A message passing standard. It defines a message passing API useful for parallel and distributed applications.
MPICH	A portable implementation of the MPI standard.
MPICH-P4	An MPI implementation based on MPICH for the chp4 device.
PAM	(Parallel Application Manager) The supervisor of any parallel job.
PJL	(Parallel Job Launcher) Any executable script or binary capable of starting parallel tasks on all hosts assigned for a parallel job.
RES	(Remote Execution Server) An LSF daemon residing on each host. It monitors and manages all LSF tasks on the host.
TS	(TaskStarter) An executable responsible for starting a task on the local host and reporting the process ID and host name to the PAM.

For more information

- See the Mathematics and Computer Science Division (MCS) of Argonne National Laboratory (ANL) MPICH Web page at `www-unix.mcs.anl.gov/mpi/mpich/` for more information about MPICH and MPICH-P4.

Files installed by lsfinstall

During installation, `lsfinstall` copies these files to the following directories:

These files...	Are installed to...
`TaskStarter`	`LSF_BINDIR`
`pam`	`LSF_BINDIR`
`esub.mpichp4`	`LSF_SERVERDIR`
`mpichp4_wrapper`	`LSF_BINDIR`
`mpirun.lsf`	`LSF_BINDIR`
`pjllib.sh`	`LSF_BINDIR`

Resources and parameters configured by lsfinstall

- External resources in `lsf.shared`:

  ```
  Begin Resource
  RESOURCE_NAME  TYPE    INTERVAL  INCREASING     DESCRIPTION
  ...
  mpichp4       Boolean    ()          ()         (MPICH P4 MPI)
  ...
  End Resources
  ```

  The `mpichp4` Boolean resource is used for mapping hosts with MPICH-P4 available.

  You should add the `mpichp4` resource name under the RESOURCES column of the Host section of `lsf.cluster.`*`cluster_name`*.

- Parameter to `lsf.conf`:

  ```
  LSB_SUB_COMMANDNAME=y
  ```

Configuring LSF to Work with MPICH-P4

mpichp4_wrapper script

Modify the `mpichp4_wrapper` script in `LSF_BINDIR` to set MPICH_HOME. The default is:

```
MPICH_HOME="/opt/mpich-1.2.5.2-ch_p4/"
```

Submitting MPICH-P4 Jobs

bsub command

Use `bsub` to submit MPICH-P4 jobs.

```
bsub -a mpichp4 -n number_cpus mpirun.lsf
[-pam "pam_options"] [mpi_options] job [job_options]
```

- **`-a mpichp4`** tells `esub` the job is an MPICH-P4 job and invokes `esub.mpichp4`.
- **`-n`** *`number_cpus`* specifies the number of processors required to run the job
- **`mpirun.lsf`** reads the environment variable LSF_PJL_TYPE=mpichp4 set by `esub.mpichp4`, and generates the appropriate `pam` command line to invoke MPICH-P4 as the PJL

For example:

```
% bsub -a mpichp4 -n 3 mpirun.lsf /examples/cpi
```

A job named `cpi` will be dispatched and run on 3 CPUs in parallel.

P4 secure-server jobs

1. To start the P4 secure-server, run the following command:

   ```
   % $MPICH_HOME/bin/serv_p4 -o -p port
   ```

   where *port* is the port number of the MPICH-P4 secure server.
2. Submit your job with the `-p4ssport` option using the following syntax:

```
bsub -a mpichp4 -n number_cpus mpirun.lsf [-pam "pam_options"] [mpi_options]
-p4ssport port job [job_options]
```

where *port* is the port number of the MPICH-P4 secure server.

You must specify full path for the job.

See the MPICH-P4 documentation for more information about the `p4ssport` secure server `mpirun.ch_p4` command option.

Task geometry with MPICH-P4 jobs

MPICH-P4 `mpirun` requires the first task to run on local node OR all tasks to run on remote node (`-nolocal`). If the LSB_PJL_TASK_GEOMETRY environment variable is set, `mpirun.lsf` makes sure the task group that contains task 0 in LSB_PJL_TASK_GEOMETRY runs on the first node.

The environment variable LSB_PJL_TASK_GEOMETRY is checked for all parallel jobs. If LSB_PJL_TASK_GEOMETRY is set users submit a parallel job (a job that requests more than 1 slot), LSF attempts to shape LSB_MCPU_HOSTS accordingly.

Submitting a job with a job script

You can submit a job using a job script as an embedded script or directly as a job, for example:

```
% bsub -a mpichp4 -n 4 < embedded_jobscript
```

```
% bsub -a mpichp4 -n 4 jobscript
```

Your job script must use `mpirun.lsf` in place of the `mpirun` command.

For information on generic PJL wrapper script components, see Chapter 2, “Running Parallel Jobs”.

See *Administering Platform LSF* for information about submitting jobs with job scripts.

CHAPTER

9

Using Platform LSF with MPICH2

Contents

About Platform LSF and MPICH2

MPICH is a freely available, portable implementation of the MPI Standard for message-passing libraries, developed jointly with Mississippi State University. MPICH is designed to provide a high performance, portable, and convenient programming environment. MPICH2 implements both MPI-1 and MPI-2.

The `mpiexec` command of MPICH2 spawns all tasks, while LSF retains full control over the tasks spawned. Specifically, LSF collects rusage information, performs job control (signal), and cleans up after the job is finished. Jobs run within LSF allocation, controlled by LSF.

Requirements

- ☐ MPICH2 version 1.0.4 or later

You should upgrade all your hosts to the same version of MPICH2.

Assumptions and limitations

- MPICH2 is installed and configured correctly
- The user's current working directory is part of a shared file system reachable by all hosts
- Currently, `mpiexec -file filename` (XML job description) is not supported.

Glossary

MPI	(Message Passing Interface) A message passing standard. It defines a message passing API useful for parallel and distributed applications.
MPICH	A portable implementation of the MPI standard.
MPICH2	An MPI implementation that implements both MPI-1 and MPI-2.
PAM	(Parallel Application Manager) The supervisor of any parallel job.
PJL	(Parallel Job Launcher) Any executable script or binary capable of starting parallel tasks on all hosts assigned for a parallel job.
RES	(Remote Execution Server) An LSF daemon residing on each host. It monitors and manages all LSF tasks on the host.
TS	(TaskStarter) An executable responsible for starting a task on the local host and reporting the process ID and host name to the PAM.

For more information

See the Mathematics and Computer Science Division (MCS) of Argonne National Laboratory (ANL) MPICH Web page at `www-unix.mcs.anl.gov/mpi/mpich/` for more information about MPICH and MPICH2.

Files installed by lsfinstall

During installation, `lsfinstall` copies these files to the following directories:

These files...	Are installed to...
`TaskStarter`	LSF_BINDIR
`pam`	LSF_BINDIR
`esub.mpich2`	LSF_SERVERDIR
mpich2_wrapper	LSF_BINDIR
`mpirun.lsf`	LSF_BINDIR
`pjllib.sh`	LSF_BINDIR

Resources and parameters configured by lsfinstall

- External resources in `lsf.shared`:

```
Begin Resource
RESOURCE_NAME  TYPE    INTERVAL  INCREASING     DESCRIPTION
...
mpich2        Boolean    ()          ()            (MPICH2 MPI)
...
End Resources
```

The `mpich2` Boolean resource is used for mapping hosts with MPICH2 available.

You should add the `mpich2` resource name under the RESOURCES column of the Host section of `lsf.cluster.`*`cluster_name`*.

- Parameter to `lsf.conf`:

```
LSB_SUB_COMMANDNAME=y
```

Configuring LSF to Work with MPICH2

1. Make sure MPICH2 commands are in the PATH environment variable. MPICH2 commands include `mpiexec`, `mpd`, `mpdboot`, `mpdtrace`, and `mpdexit`.

   For example:

```
[174]- which mpiexec /pcc/app/mpich2/kernel2.4-glibc2.3-x86/bin/mpiexec
```

2. Add an mpich2 boolean resource to the $LSF_ENVDIR/lsf.shared file.

   For example:

```
hmmer        Boolean     ()     ()     (hmmer availability)
lammpi       Boolean     ()     ()     (lam-mpi available host)
mpich2       Boolean     ()     ()     (mpich2 available host) <====
End Resource
```

3. Add mpich2 to each host that an mpich2 parallel job may run on.

   For example:

```
Begin  Host
HOSTNAME  model  type  server  r1m  mem  swp  RESOURCES   #Keywords
qat20       !      !        1   3.5  ()   ()   (mpich2)
qat21       !      !        1   3.5  ()   ()   (mpich2)
qat22       !      !        1   3.5  ()   ()   (mpich2)
End  Host
```

4. Run `lsadmin reconfig` and `badmin mbdrestart` as root.
5. Run `lshosts` to confirm that an mpich2 resource is configured on all hosts on which you would like to run mpich2 parallel jobs.

   For example:

```
[173]- lshosts
HOST_NAME       type    model  cpuf  ncpus  maxmem  maxswp  server  RESOURCES
qat20        LINUX86   PC1133  23.1      1    310M       -     Yes  (mpich2)
qat21.lsf.p LINUX86   PC1133  23.1      1    311M    635M     Yes  (mpich2)
qat22.lsf.p UNKNOWN UNKNOWN_   1.0      -       -       -     Yes  (mpich2)
```

6. Configure and start an MPD ring.
   a. If you want to start an MPD ring per job, this is the default and recommended mechanism, and you do not need to do any extra configuration.
   b. If you want to start an MPD ring for all users, use the `mpdboot` command as root on all machines.

      To check if mpdboot ran successfully, use the `mpdtrace` command

```
[root@qat20 test]# mpdtrace -l

qat20_37272
qat21_52535
```

      i. For MPICH2 1.0.3 only, add the following lines to $HOME/.mpd.conf for all users.

```
[61]- cat .mpd.conf
MPD_USE_ROOT_MPD=Y <==========
secretword=123579a
```

    ii Make sure $HOME/.mpd.conf has a permission mode of 600 after you finish the modification.
    iii Set LSF_START_MPD_RING=N in your job script or in the environment for all users.

  c If you want to start an MPD ring on all hosts, follow the steps described in the MPICH2 documentation to start an MPD ring across all LSF hosts for each user. The user MPD ring must be running all the time, and you must set LSF_START_MPD_RING=N in your job script or in the environment for all users.

Do not run `mpdallexit` or `mpdcleanup` to terminate the MPD ring.

---

7 Make sure LSF uses system host official names (/etc/hosts): this will prevent problems when you run the application.

    i Configure the $LSF_CONFDIRDIR/hosts file and the $LSF_ENVDIR/lsf.cluster.<*clustername*> file.

    For example:

```
172.25.238.91 scali scali.lsf.platform.com
172.25.238.96 scali1 scali1.lsf.plaform.com
```

    ii If the official host name returned to LSF is a short name, but LSF commands display host names that include domain names, you can use LSF_STRIP_DOMAIN in lsf.conf to display the short names.

8 Change the $LSF_BINDIR/mpich2_wrapper script to make sure MPI_TOPDIR= points to the MPICH2 install directory.

Building Parallel Jobs

1. Use `mpicc -o` to compile your source code.

   For example:

   ```
   [178]- which mpicc /pcc/app/mpich2/kernel2.4-glibc2.3-x86/bin/mpicc
   5:19pm Mon, Sep-19-2005 qat21:~/milkyway/bugfix/test
   [179]- mpicc -o hw.mpich2 hw.c 3.2
   ```

2. Make sure the compiled binary can run under the root MPD ring outside Platform LSF.

   For example:

   ```
   [180]- mpiexec -np 2 hw.mpich2
   ```

```
Process 0 is printing on qat21 (pid =16160):
Greetings from process 1 from qat20 pid 24787!
```

Submitting MPICH2 Jobs

bsub command

Use the `bsub` command to submit MPICH2 jobs.

1. Submit a job from the console command line:

```
bsub <bsub_options> -n <###> -a mpich2 mpirun.lsf <mpiexec_options> job
<job_options>
```

Note that -np options of mpiexec will be ignored.

For example:

```
bsub -I -n 8 -R "span[ptile=4]" -a mpich2 -W 2 mpirun.lsf -np 3 ./hw.mpich2
```

1. Submit a job using a script:

   `bsub < myjobscript.sh`

   where myjobscript.sh looks like:

```
#!/bin/sh
#BSUB -n 8
#BSUB -a mpich2
mpirun.lsf ./hw.mpich2
```

The `mpich2_wrapper` script supports almost all original `mpiexec` options except those that will affect job scheduling decisions, for example, `-np` (`-n`).

`-n` syntax is supported. If you use the `-n` option, you must either request enough CPUs when the job is submitted, or set the environment variable LSB_PJL_TASK_GEOMETRY. See "Running Jobs with Task Geometry" on page 42 for detailed usage of LSB_PJL_TASK_GEOMETRY.

Task geometry with MPICH2 jobs

MPICH2 `mpirun` requires the first task to run on the local node OR all tasks to run on a remote node (`-nolocal`). If the LSB_PJL_TASK_GEOMETRY environment variable is set, `mpirun.lsf` makes sure the task group that contains task 0 in LSB_PJL_TASK_GEOMETRY runs on the first node.

The environment variable LSB_PJL_TASK_GEOMETRY is checked for all parallel jobs. If LSB_PJL_TASK_GEOMETRY is set users submit a parallel job (a job that requests more than 1 slot), LSF attempts to shape LSB_MCPU_HOSTS accordingly.

CHAPTER

10

Using Platform LSF with MVAPICH

Contents

About Platform LSF and MVAPICH

MVAPICH is an open-source product developed in the Department of Computer and Information Science, The Ohio State University. MVAPICH is MPI-1 over VAPI for InfiniBand. It is an MPI-1 implementation on Verbs Level Interface (VAPI), developed by Mellanox Technologies. The implementation is based on MPICH and MVICH.

The LSF MVAPICH MPI integration is based on the LSF generic PJL framework. It supports the following MVAPICH variations:

- Generic MVAPICH (OSU)
- Cisco/Topspin® used in Platform OCS
- IBRIX™ roll used in Platform OCS

Requirements

☐ The latest release is MVAPICH 0.9.4 (includes MPICH 1.2.6). or later

You should upgrade all your hosts to the same version of MVAPICH.

Assumptions and limitations

- MVAPICH is installed and configured correctly
- The user's current working directory is part of a shared file system reachable by all hosts
- The directory specified by the MVAPICH_HOME variable is accessible by the same path on all hosts

Glossary

MPI	(Message Passing Interface) A message passing standard. It defines a message passing API useful for parallel and distributed applications.
MPICH	A portable implementation of the MPI standard.
PAM	(Parallel Application Manager) The supervisor of any parallel job.
PJL	(Parallel Job Launcher) Any executable script or binary capable of starting parallel tasks on all hosts assigned for a parallel job.
RES	(Remote Execution Server) An LSF daemon residing on each host. It monitors and manages all LSF tasks on the host.
TS	(TaskStarter) An executable responsible for starting a task on the local host and reporting the process ID and host name to the PAM.

For more information

- See the Mathematics and Computer Science Division (MCS) of Argonne National Laboratory (ANL) MPICH Web page at `www-unix.mcs.anl.gov/mpi/mpich/` for more information about MPICH.
- MVAPICH HOME: `nowlab.cis.ohio-state.edu/projects/mpi-iba/`
- ROCKS HOME: `www.rocksclusters.org/Rocks/`
- Topspin (now Cisco): `http://cisco.com/en/US/products/index.html`
- IBRIX roll: `http://www.rocksclusters.org/Rocks/`

Files installed by lsfinstall

During installation, `lsfinstall` copies these files to the following directories:

These files...	Are installed to...
`TaskStarter`	`LSF_BINDIR`
`pam`	`LSF_BINDIR`
`esub.mvapich`—sets the mode: rsh ssh or mpd	`LSF_SERVERDIR`
`mvapich_wrapper`	`LSF_BINDIR`
`mpirun.lsf`	`LSF_BINDIR`
`pjllib.sh`	`LSF_BINDIR`

Resources and parameters configured by lsfinstall

- External resources in `lsf.shared`:

  ```
  Begin Resource
  RESOURCE_NAME  TYPE    INTERVAL  INCREASING     DESCRIPTION
  ...
  mvapich       Boolean    ()         ()       (Infiniband MPI)
  ...
  End Resources
  ```

  The `mvapich` Boolean resource is used for mapping hosts with MVAPICH available.

  You should add the `mvapich` resource name under the RESOURCES column of the Host section of `lsf.cluster.`*`cluster_name`*.

- Parameter to `lsf.conf`:

  ```
  LSB_SUB_COMMANDNAME=y
  ```

Configuring LSF to Work with MVAPICH

esub.mvapich script

Modify the `esub.mvapich` in `LSF_SERVERDIR` to set MVAPICH_START_CMD.to one of `ssh`, `rsh`, or `mpd`. The default value is `ssh`.

mvapich_wrapper script

Modify the `mvapich_wrapper` script in `LSF_BINDIR` to set MVAPICH_HOME. The defaults are:

- Topspin/Cisco MPI: `MVAPICH_HOME="/usr/local/topspin`
- IBRIX Roll MPI: `MVAPICH_HOME="/opt/mpich/infiniband/gnu"`
- Generic MVAPICH: defined by your site. For example:
  `MVAPICH_HOME="/opt/mvapich"`

mpd command location

Make sure the mpirun_rsh/ssh/mpd command is under MVAPICH_HOME/bin.

Submitting MVAPICH Jobs

bsub command

Use `bsub -a mvapich` to submit jobs:

If the starting command is `mpd`, you must submit your MVAPICH jobs as exclusive jobs (`bsub -x`).

`bsub -a mvapich -n` *`number_cpus`* **`mpirun.lsf`**
`[`**`-pam "`***`pam_options`***`"`**`]` `[`*`mpi_options`*`]` *`job`* `[`*`job_options`*`]`

- **`-a mvapich`** tells `esub` the job is an MVAPICH job and invokes `esub.mvapich`.
- **`-n`** *`number_cpus`* specifies the number of processors required to run the job
- **`mpirun.lsf`** reads the environment variable LSF_PJL_TYPE=mvapich set by `esub.mvapich`, and generates the appropriate `pam` command line to invoke MVAPICH as the PJL

For example:

```
% bsub -a mvapich -n 3 mpirun.lsf /examples/cpi
```

A job named `cpi` will be dispatched and run on 3 CPUs in parallel.

Task geometry with MVAPICH jobs

MVAPICH supports the LSF task geometry feature except in MPD mode. When running in MPD mode, the order of the hosts specified in the machine file is not honored:

Submitting a job with a job script

A wrapper script is often used to call MVAPICH. You can submit a job using a job script as an embedded script or directly as a job, for example:

```
% bsub -a mvapich -n 4 < embedded_jobscript
```

```
% bsub -a mvapich -n 4 jobscript
```

Your job script must use `mpirun.lsf` in place of the `mpirun` command.

For more information

- See Chapter 2, "Running Parallel Jobs" for information about generic PJL wrapper script components
- See the *Platform LSF Command Reference* for information about the `bsub` command
- See *Administering Platform LSF* for information about submitting jobs with job scripts

CHAPTER

11

Using Platform LSF with Intel® MPI

Contents

About Platform LSF and the Intel® MPI Library

The Intel® MPI Library ("Intel MPI") is a high-performance message-passing library for developing applications that can run on multiple cluster interconnects chosen by the user at runtime. It supports TCP, shared memory, and high-speed interconnects like InfiniBand and Myrinet.

Intel MPI supports all MPI-1 features and many MPI-2 features, including file I/O, generalized requests, and preliminary thread support. it is based on the MPICH2 specification.

The LSF Intel® MPI integration is based on the LSF generic PJL framework. It supports the LSF task geometry feature.

Requirements

- Intel® MPI version 1.0.2 or later

You should upgrade all your hosts to the same version of Intel MPI.

Assumptions and limitations

- Intel MPI is installed and configured correctly
- When an Intel MPI job is killed, PAM reports exit status unknown
- When MPI tasks get killed, MPD automatically kills TaskStarter
- LSF host names must be the official host names recognized by the system

Glossary

MPD	Multi-Purpose Daemon (MPD) job startup mechanism
MPI	(Message Passing Interface) A message passing standard. It defines a message passing API useful for parallel and distributed applications.
MPICH	A portable implementation of the MPI standard.
MPICH2	An MPI implementation for platforms such as clusters, SMPs, and massively parallel processors.
PAM	(Parallel Application Manager) The supervisor of any parallel job.
PJL	(Parallel Job Launcher) Any executable script or binary capable of starting parallel tasks on all hosts assigned for a parallel job.
RES	(Remote Execution Server) An LSF daemon residing on each host. It monitors and manages all LSF tasks on the host.
TS	(TaskStarter) An executable responsible for starting a task on the local host and reporting the process ID and host name to the PAM.

For more information

- See the Mathematics and Computer Science Division (MCS) of Argonne National Laboratory (ANL) MPICH Web pages:
    - `www-unix.mcs.anl.gov/mpi/mpich/` for more information about MPICH.

  - `www-unix.mcs.anl.gov/mpi/mpich2/` for more information about MPICH2.
- See the Intel Software Network > Software Products > Cluster Tools > Intel MPI Library at `www.intel.com` for more information about the Intel MPI Library.
- See *Getting Started with the Intel® MPI Library* (`Getting_Started.pdf` in the Intel MPI installation documentation directory for more information about using the Intel MPI library and commands.

Files installed by lsfinstall

During installation, `lsfinstall` copies these files to the following directories

These files...	Are installed to...
`TaskStarter`	`LSF_BINDIR`
`pam`	`LSF_BINDIR`
`esub.intelmpi`	`LSF_SERVERDIR`
`intelmpi_wrapper`	`LSF_BINDIR`
`mpirun.lsf`	`LSF_BINDIR`
`pjllib.sh`	`LSF_BINDIR`

Resources and parameters configured by lsfinstall

- External resources in `lsf.shared`:

```
Begin Resource
RESOURCE_NAME  TYPE     INTERVAL  INCREASING     DESCRIPTION
...
intelmpi       Boolean     ()         ()            (Intel MPI)
...
End Resources
```

The `intelmpi` Boolean resource is used for mapping hosts with Intel MPI available.

You should add the `intelmpi` resource name under the RESOURCES column of the Host section of `lsf.cluster.`*`cluster_name`*.

- Parameter to `lsf.conf`:

```
LSB_SUB_COMMANDNAME=y
```

Configuring LSF to Work with Intel MPI

intelmpi_wrapper script

Modify the `intelmpi_wrapper` script in `LSF_BINDIR` to set MPI_TOPDIR The default value is:

```
MPI_TOPDIR="/opt/intel/mpi/2.0"
```

lsf.conf (optional)

To improve performance and scalability for large parallel jobs, tune the following parameters as described in "Tuning PAM Scalability and Fault Tolerance" on page 41:

- LSF_HPC_PJL_LOADENV_TIMEOUT
- LSF_PAM_RUSAGE_UPD_FACTOR

The user's environment can override these.

Working with the Multi-purpose Daemon (MPD)

The Intel® MPI Library ("Intel MPI") uses a Multi-Purpose Daemon (MPD) job startup mechanism. MPD daemons must be up and running on the hosts where an MPI job is supposed to start before `mpiexec` is started.

How Platform LSF manages MPD rings

LSF manages MPD rings for users automatically using `mpdboot` and `mpdtrace` commands.

Each MPI job running under LSF uses a uniquely labeled MPD ring. The ring is started by the `intelmpi_wrapper` during job launch and terminated by the `intelmpi_wrapper` after MPI application exits, either normally or abnormally. This allows coexistence of multiple MPI jobs belonging to different users as well as multiple jobs from the same user on the same set of hosts.

For more information

- See *Getting Started with the Intel® MPI Library* (`Getting_Started.pdf`) in the Intel MPI installation documentation directory for more information about using the Intel MPI library and commands
- See *Administering Platform LSF* for information about using job starters

Submitting Intel MPI Jobs

bsub command

Use `bsub -a intelmpi` to submit jobs.

If the starting command is `mpd`, you must submit your Intel MPI jobs as exclusive jobs (`bsub -x`).

bsub -a intelmpi -n *number_cpus* **mpirun.lsf**
[**-pam** "*pam_options*"] [*mpi_options*] *job* [*job_options*]

- **-a intelmpi** tells esub the job is an Intel MPI job and invokes `esub.intelmpi`.
- **-n** *number_cpus* specifies the number of processors required to run the job
- **mpirun.lsf** reads the environment variable LSF_PJL_TYPE=intelmpi set by `esub.intelmpi`, and generates the appropriate `pam` command line to invoke Intel MPI as the PJL

For example:

```
% bsub -a intelmpi -n 3 mpirun.lsf /examples/cpi
```

A job named `cpi` will be dispatched and run on 3 CPUs in parallel.

Task geometry with Intel MPI jobs

Intel MPI supports the LSF task geometry feature

Submitting a job with a job script

A wrapper script is often used to call Intel MPI. You can submit a job using a job script as an embedded script or directly as a job, for example:

```
% bsub -a intelmpi -n 4 < embedded_jobscript
```

```
% bsub -a intelmpi -n 4 jobscript
```

Your job script must use `mpirun.lsf` in place of the `mpirun` command.

Using Intel MPI configuration files (-configfile)

All `mpiexec -configfile` options are supported. `-configfile` should be the only option after the `mpiexec` command.

The placement options in the configuration file (`-gn`, `-gnp`, `-n`, `-np`, `-host`) must agree with the value of the LSB_MCPU_HOSTS and LSB_HOSTS environment variables.

mpiexec limitations

-file option is not supported

The `-file` option of `mpiexec` is not supported. You can use the `-configfile` option.

If you submit an Intel MPI job with `-file`, the `intelmpi_wrapper` will exit and fail the job. If you specify the log file for `intelmpi_wrapper`, an error message is appended to the log file:

Official host names

`mpiexec` requires host names as they are returned by the `hostname` command or the `gethostname()` system call. For example:

```
% hostname
hosta
% mpiexec -l -n 2 -host hosta.domain.com ./hmpi
mpdrun: unable to start all procs; may have invalid machine
names
    remaining specified hosts:
        hosta.domain.com

% mpiexec -l -n 2 -host hosta ./hmpi
0: myrank 0, n_processes 2
1: myrank 1, n_processes 2
0: From process 1: Slave process 1!
```

-genvlist option The `-genvlist` options does not work if the configuration file for `-configfile` has more than one entry.

For more information

- See Chapter 2, "Running Parallel Jobs" for information about generic PJL wrapper script components
- See the *Platform LSF Command Reference* for information about the `bsub` command
- See *Administering Platform LSF* for information about submitting jobs with job scripts

CHAPTER

12

Using Platform LSF with Open MPI

Contents

About Platform LSF and the Open MPI Library

The Open MPI Library is a high-performance message-passing library for developing applications that can run on multiple cluster interconnects chosen by the user at runtime. Open MPI supports all MPI-1 and MPI-2 features.

The LSF Open MPI integration is based on the LSF generic PJL framework. It supports the LSF task geometry feature.

Requirements

- Open MPI version 1.1 or later

You should upgrade all your hosts to the same version of Open MPI.

Assumptions and limitations

- Open MPI is installed and configured correctly
- The user-defined -app file option is not supported

Glossary

MPD	Multi-Purpose Daemon (MPD) job startup mechanism
MPI	(Message Passing Interface) A message passing standard. It defines a message passing API useful for parallel and distributed applications.
MPICH	A portable implementation of the MPI standard.
Open MPI	An MPI implementation for platforms such as clusters, SMPs, and massively parallel processors.
PAM	(Parallel Application Manager) The supervisor of any parallel job.
PJL	(Parallel Job Launcher) Any executable script or binary capable of starting parallel tasks on all hosts assigned for a parallel job.
RES	(Remote Execution Server) An LSF daemon residing on each host. It monitors and manages all LSF tasks on the host.
TS	(TaskStarter) An executable responsible for starting a task on the local host and reporting the process ID and host name to the PAM.

For more information

- See the Open MPI Project web page at `http://www.open-mpi.org/`

Files installed by lsfinstall

During installation, `lsfinstall` copies these files to the following directories

These files...	Are installed to...
`TaskStarter`	`LSF_BINDIR`
`pam`	`LSF_BINDIR`
`esub.openmpi`	`LSF_SERVERDIR`
`openmpi_wrapper`	`LSF_BINDIR`

These files...	Are installed to...
`mpirun.lsf`	`LSF_BINDIR`
`pjllib.sh`	`LSF_BINDIR`

Resources and parameters configured by lsfinstall

- External resources in `lsf.shared`:

```
Begin Resource
RESOURCE_NAME   TYPE    INTERVAL  INCREASING     DESCRIPTION
...
openmpi        Boolean    ()         ()           (Open MPI)
...
End Resources
```

The `openmpi` Boolean resource is used for mapping hosts with Open MPI available.

You should add the `openmpi` resource name under the RESOURCES column of the Host section of `lsf.cluster.`*`cluster_name`*.

- Parameter to `lsf.conf`:

```
LSB_SUB_COMMANDNAME=y
```

Configuring LSF to Work with Open MPI

- The `mpirun` command must be included in the `$PATH` environment variable on all LSF hosts.
- Make sure LSF uses system host official names (/etc/hosts): this will prevent problems when you run the application.
  - Configure the $LSF_CONFDIRDIR/hosts file and the $LSF_ENVDIR/lsf.cluster.<*clustername*> file.

    For example:

    ```
    172.25.238.91 scali scali.lsf.platform.com

    172.25.238.96 scali1 scali1.lsf.plaform.com
    ```
  - If the official host name returned to LSF is a short name, but LSF commands display host names that include domain names, you can use LSF_STRIP_DOMAIN in lsf.conf to display the short names.

No other configuration is required. Optionally, you can add the `openmpi` resource name under the RESOURCES column of the Host section of `lsf.cluster.`*`cluster_name`* to indicate the hosts in the cluster that have Open MPI installed and enabled.

Submitting Open MPI Jobs

bsub command

Use `bsub -a openmpi` to submit jobs.

For example:

```
bsub -a openmpi -n number_cpus mpirun.lsf a.out
```

- **`-a openmpi`** tells `esub` the job is an Open MPI job and invokes `esub.openmpi`.
- **`-n`** *`number_cpus`* specifies the number of processors required to run the job
- **`mpirun.lsf`** reads the environment variable LSF_PJL_TYPE=intelmpi set by `esub.openmpi`, and generates the appropriate `pam` command line to invoke Open MPI as the PJL

Task geometry with Open MPI jobs

Open MPI supports the LSF task geometry feature

Submitting a job with a job script

A wrapper script is often used to call Open MPI. You can submit a job using a job script as an embedded script or directly as a job, for example:

```
bsub -a < jobscript
```

Your job script must use `mpirun.lsf` in place of the `mpirun` command.

For more information

- See Chapter 2, "Running Parallel Jobs" for information about generic PJL wrapper script components
- See the *Platform LSF Command Reference* for information about the `bsub` command
- See *Administering Platform LSF* for information about submitting jobs with job scripts

CHAPTER

13

Using Platform LSF Parallel Application Integrations

Contents

Using LSF with ANSYS

LSF use supports various ANSYS solvers through a common integration console built-in to the ANSYS GUI. The only change the average ANSYS user sees is the addition of a **Run using LSF?** button on the standard ANSYS console.

Using ANSYS with LSF simplifies distribution of jobs, and improves throughput by removing the need for engineers to worry about when or where their jobs run. They simply request job execution and know that their job will be completed as fast as their environment will allow.

Requirements

- LSF HPC features enabled
- ANSYS version 5.6 or higher, available from Ansys Incorporated.

Configuring LSF for ANSYS

During installation, `lsfinstall` adds the Boolean resource `ansys` to the Resource section of `lsf.shared`.

Host configuration (optional)

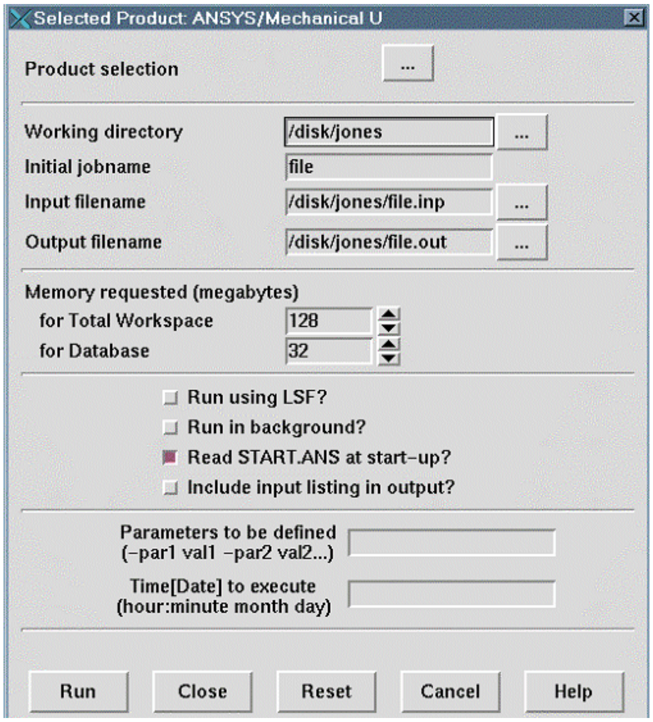
If only some of your hosts can accept ANSYS jobs, configure the Host section of `lsf.cluster.`*`cluster_name`* to identify those hosts.

Edit `LSF_ENVDIR/conf/lsf.cluster.cluster_name` file and add the `ansys` resource to the hosts that can run ANSYS jobs:

```
Begin Host
HOSTNAME    model   type  server  r1m   mem   swp  RESOURCES
...
hostA        !       !      1     3.5   ()    ()   ()
hostB        !       !      1     3.5   ()    ()   (ansys)
hostC        !       !      1     3.5   ()    ()   ()
...
End Host
```

Submitting jobs through ANSYS

To start a job, choose the **Batch** menu item. The following dialog is displayed:



Selected Product: ANSYS/Mechanical U

Product selection ...

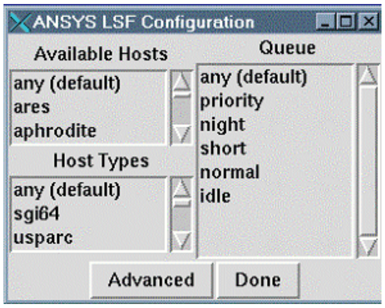
Working directory /disk/jones ...
Initial jobname file
Input filename /disk/jones/file.inp ...
Output filename /disk/jones/file.out ...

Memory requested (megabytes)
for Total Workspace 128
for Database 32

☐ Run using LSF?
☐ Run in background?
☒ Read START.ANS at start-up?
☐ Include input listing in output?

Parameters to be defined (-par1 val1 -par2 val2...)
Time[Date] to execute (hour:minute month day)

Initial Jobname	The name given to the job for easier recognition at runtime.
Input filename	Specifies the file of ANSYS commands you are submitting for batch execution. You can either type in the desired file name or click on the **...** button, to display a file selection dialog box.
Output filename	Specifies the file to which ANSYS directs text output by the program. If the file name already exists in the working directory, it will be overwritten when the batch job is started.
Memory requested	The memory requirements for the job.
Run using LSF?	Launches ANSYS LSF, a separately licensed product.
Run in background?	Runs the ANSYS job in background or in foreground mode.
Include input listing in output?	Includes or excludes the input file listing at the beginning of the output file.
Parameters to be defined	Additional ANSYS parameters
Time[Date] to execute	Specifies a start time and date to start the job. This option is active after **Run in background?** has been changed to Yes. To use this option, you must have permission to run the at command on UNIX systems.
Additional LSF configuration	You can also configure additional options to specify LSF job requirements such as queue, host, or desired host architecture:



Available Hosts	Allows users to specify a specific host to run the job on.
Queue	Allows users to specify which queue they desire instead of the default.
Host Types	Allows users to specify a specific architecture for their job.

Submitting jobs through the ANSYS command-line

Submitting a command line job requires extra parameters to run correctly through LSF.

Syntax

```
bsub -R ansys [bsub_options] ansys_command -b -p productvar
<input_name >&output_name
```

`-R`	Run the job on hosts with the Boolean resource `ansys` configured
`bsub_options`	Regular options to `bsub` that specify the job parameters
`ansys_command`	The ANSYS executable to be executed on the host (for example, `ansys57`)
`-b`	Run the job in ANSYS batch mode
`-p` *`productvar`*	ANSYS product to use with the job
`<input_name`	ANSYS input file. (You can also use the `bsub -i` option.)
`>&output_name`	ANSYS output file. (You can also use the `bsub -o` option.)

Using LSF with NCBI BLAST

LSF accepts jobs running NCBI BLAST (Basic Local Alignment Search Tool).

Requirements

- Platform LSF HPC features enabled
- BLAST, available from the National Center for Biotechnology Information (NCBI)

Configuring LSF for BLAST jobs

During installation, `lsfinstall` adds the Boolean resource `blast` to the Resource section of `lsf.shared`.

Host configuration (optional)

If only some of your hosts can accept BLAST jobs, configure the Host section of `lsf.cluster.`*`cluster_name`* to identify those hosts.

Edit `LSF_ENVDIR/conf/lsf.cluster.cluster_name` file and add the `blast` resource to the hosts that can run BLAST jobs:

```
Begin Host
HOSTNAME     model    type   server   r1m    mem    swp   RESOURCES
...
hostA         !        !       1      3.5    ()     ()    ()
hostB         !        !       1      3.5    ()     ()    (blast)
hostC         !        !       1      3.5    ()     ()    ()
...
End Host
```

Submitting BLAST jobs

Use BLAST parallel provided with LSF to submit BLAST jobs.

BLAST parallel is a PERL program that distributes BLAST searches across a cluster by splitting both the query file and the reference database and merging the result files after all BLAST jobs finish.

See the README in the `LSF_MISC/examples/blastparallel/` for information about installing, configuring, and using BLAST parallel.

Using LSF with FLUENT

LSF is integrated with FLUENT products from ANSYS Inc., allowing FLUENT jobs to take advantage of the checkpointing and migration features provided by LSF. This increases the efficiency of the software and means data is processed faster.

FLUENT 5 offers versions based on system vendors' parallel environments (usually MPI using the VMPI version of FLUENT 5.) Fluent also provides a parallel version of FLUENT 5 based on its own socket-based message passing library (the NET version).

This chapter assumes you are already familiar with using FLUENT software and checkpointing jobs in LSF.

See *Administering Platform LSF* for more information about checkpointing in LSF.

Requirements

- Platform LSF HPC features enabled
- FLUENT 5 or higher, available from ANSYS Inc.

Optional requirements

- Hardware vendor-supplied MPI environment for network computing to use the "vmpi" version of FLUENT 5.

Configuring LSF for FLUENT jobs

During installation, `lsfinstall` adds the Boolean resource `fluent` to the Resource section of `lsf.shared`.

LSF also installs the `echkpnt.fluent` and `erestart.fluent` files in LSF_SERVERDIR.

Host configuration (optional)

If only some of your hosts can accept FLUENT jobs, configure the Host section of `lsf.cluster.`*`cluster_name`* to identify those hosts.

Edit `LSF_ENVDIR/conf/lsf.cluster.cluster_name` file and add the `fluent` resource to the hosts that can run FLUENT jobs:

```
Begin Host
HOSTNAME    model   type   server  r1m   mem   swp   RESOURCES
...
hostA       !       !       1      3.5   ()    ()    ()
hostB       !       !       1      3.5   ()    ()    (fluent)
hostC       !       !       1      3.5   ()    ()    ()
...
End Host
```

Checkpointing in FLUENT

FLUENT 5 is integrated with LSF to use the LSF checkpointing capability. At the end of each iteration, FLUENT looks for the existence of a checkpoint file (`check`) or a checkpoint exit file (`exit`). If it detects the checkpoint file, it writes a case and data file, removes the checkpoint file, and continues iterating. If it detects a checkpoint exit file, it writes a case and data file, then exits.

Use the `bchkpnt` command to create the checkpoint and checkpoint exit files, which forces FLUENT to checkpoint, or checkpoint and exit itself. FLUENT also creates a journal file with instructions to read the checkpointed case and data files, and continue iterating. FLUENT uses this file when it is restarted with the `brestart` command.

echkpnt and erestart LSF installs `echkpnt.fluent` and `erestart.fluent`, which are special versions of `echkpnt` and `erestart` to allow checkpointing with FLUENT. Use `bsub -a fluent` to make sure your job uses these files.

Checkpoint directories

When you submit a checkpointing job, you specify a checkpoint directory.

Before the job starts running, LSF sets the environment variable LSB_CHKPNT_DIR. The value of LSB_CHKPNT_DIR is a subdirectory of the checkpoint directory specified in the command line. This subdirectory is identified by the job ID and only contains files related to the submitted job.

Checkpoint trigger files

When you checkpoint a FLUENT job, LSF creates a checkpoint trigger file (`check`) in the job subdirectory, which causes FLUENT to checkpoint and continue running. A special option is used to create a different trigger file (`exit`) to cause FLUENT to checkpoint and exit the job.

FLUENT uses the LSB_CHKPNT_DIR environment variable to determine the location of checkpoint trigger files. It checks the job subdirectory periodically while running the job. FLUENT does not perform any checkpointing unless it finds the LSF trigger file in the job subdirectory. FLUENT removes the trigger file after checkpointing the job.

Restarting jobs

If a job is restarted, LSF attempts to restart the job with the `-restart` option appended to the original FLUENT command. FLUENT uses the checkpointed data and case files to restart the process from that checkpoint, rather than repeating the entire process.

Each time a job is restarted, it is assigned a new job ID, and a new job subdirectory is created in the checkpoint directory. Files in the checkpoint directory are never deleted by LSF, but you may choose to remove old files once the FLUENT job is finished and the job history is no longer required.

Submitting FLUENT jobs

Use `bsub` to submit the job, including parameters required for checkpointing.

Syntax The syntax for the `bsub` command to submit a FLUENT job is:

`bsub` [**`-R fluent`**] **`-a fluent`** [**`-k`** *checkpoint_dir* | **`-k "`***checkpoint_dir* [*checkpoint_period*]**`"`** [*bsub options*] *FLUENT command* [*FLUENT options*] **`-lsf`**

`-R fluent` Optional. Specify the `fluent` shared resource if the FLUENT application is only installed on certain hosts in the cluster

`-a fluent` Use the `esub` for FLUENT jobs, which automatically sets the checkpoint method to `fluent` to use the checkpoint and restart programs for FLUENT jobs, `echkpnt.fluent` and `erestart.fluent`.

The checkpointing feature for FLUENT jobs requires all of the following parameters:

`-k` *checkpoint_dir*

Regular option to `bsub` that specifies the name of the checkpoint directory.

checkpoint_period

Regular option to `bsub` that specifies the time interval in minutes that LSF will automatically checkpoint jobs.

FLUENT command

Regular command used with FLUENT software.

`-lsf` Special option to the FLUENT command. Specifies that FLUENT is running under LSF, and causes FLUENT to check for trigger files in the checkpoint directory if the environment variable LSB_CHKPNT_DIR is set.

Examples

- Sequential FLUENT batch job

```
% bsub -a fluent fluent 3d -g -i journal_file -lsf
```

- Parallel FLUENT net version batch job on 4 CPUs

```
% bsub -a fluent -n 4 fluent 3d -t0 -pnet -g -i
journal_file -lsf
```

Note When using the net version of FLUENT 5, pam is not used to launch FLUENT, so the JOB_STARTER argument of the queue should not be set. Instead, LSF sets an environment variable to contain a list of hosts and FLUENT uses this list to launch itself.

Checkpointing, restarting, and migrating FLUENT jobs

Checkpointing **`bchkpnt`** `[`*`bchkpnt_options`*`] [`**`-k`**`] [`*`job_ID`*`]`

- **`-k`**

  Specifies checkpoint and exit. The job will be killed immediately after being checkpointed. When the job is restarted, it continues from the last checkpoint.

- *job_ID*

  Job ID of the FLUENT job. Specifies which job to checkpoint. Each time the job is migrated, the job is restarted and assigned a new job ID.

Restarting **`brestart`** `[`*`brestart options`*`]` *`checkpoint_directory`* `[`*`job_ID`*`]`

- *checkpoint_directory*

  Specifies the checkpoint directory, where the job subdirectory is located.

- *job_ID*

  Job ID of the FLUENT job, specifies which job to restart. At this point, the restarted job is assigned a new job ID, and the new job ID is used for checkpointing. The job ID changes each time the job is restarted.

Migrating **`bmig`** `[`*`bsub_options`*`] [`*`job_ID`*`]`

- *job_ID*

  Job ID of the FLUENT job, specifies which job to restart. At this point, the restarted job is assigned a new job ID, and the new job ID is used for checkpointing. The job ID changes each time the job is restarted.

Examples

- Sequential FLUENT batch job with checkpoint and restart

  ```
  % bsub -a fluent -k "/home/username 60" fluent 3d -g -i
  journal_file -lsf
  ```

  Submits a job that uses the checkpoint/restart method `echkpnt.fluent` and `erestart.fluent`, `/home/username` as the checkpoint directory, and a 60 minute duration between automatic checkpoints. FLUENT checks if there is a checkpoint trigger file `/home/username/exit` or `/home/username/check`.

  ```
  % bchkpnt job_ID
  ```

  `echkpnt` creates the checkpoint trigger file `/home/username/check` and waits until the file is removed and the checkpoint is successful. FLUENT writes a case and data file, and a restart journal file at the end of its current iteration. The files are saved in `/home/username/`*`job_ID`* and FLUENT continues to iterate.

  Use `bjobs` to verify that the job is still running after checkpoint.

  ```
  % bchkpnt -k job_ID
  ```

  `echkpnt` creates the checkpoint trigger file `/home/username/exit` and waits until the file is removed and the checkpoint is successful. FLUENT writes a case and data file, and a restart journal file at the end of its current iteration. The files are saved in `/home/username/`*`job_ID`* and FLUENT exits.

  Use `bjobs` to verify that the job is not running after checkpoint.

  ```
  % brestart /home/username/job_ID
  ```

  Starts a FLUENT job using the latest case and data files in `/home/username/`*`job_ID`*. The restart journal file `/home/username/`*`job_ID`*`/#restart.inp` is used to instruct FLUENT to read the latest case and data files and continue iterating.

- Parallel FLUENT VMPI version batch job with checkpoint and restart on 4 CPUs

  ```
  % bsub -a fluent -k "/home/username 60" -n 4 fluent 3d -t4
  -pvmpi -g -i journal_file -lsf
  ```

  ```
  % bchkpnt -k job_ID
  ```

  Forces FLUENT to write a case and data file, and a restart journal file at the end of its current iteration. The files are saved in `/home/username/`*`job_ID`* and FLUENT exits.

  ```
  % brestart /home/username/job_ID
  ```

  Starts a FLUENT job using the latest case and data files in `/home/username/`*`job_ID`*. The restart journal file `/home/username/`*`job_ID`*`/#restart.inp` is used to instruct FLUENT to read the latest case and data files and continue iterating.

  The parallel job is restarted using the same number of processors (4) requested in the original `bsub` submission.

  ```
  % bmig -m hostA 0
  ```

  All jobs on `hostA` are checkpointed and moved to another host.

Using LSF with Gaussian

Platform LSF accepts jobs running the Gaussian electronic structure modeling program.

Requirements

- Platform LSF HPC features enabled
- Gaussian 98, available from Gaussian, Inc.

Configuring LSF for Gaussian jobs

During installation, `lsfinstall` adds the Boolean resource `gaussian` to the Resource section of `lsf.shared`.

Host configuration (optional)

If only some of your hosts can accept Gaussian jobs, configure the Host section of `lsf.cluster.`*`cluster_name`* to identify those hosts.

Edit `LSF_ENVDIR/conf/lsf.cluster.cluster_name` file and add the `gaussian` resource to the hosts that can run Gaussian jobs:

```
Begin Host
HOSTNAME     model    type   server   r1m    mem    swp   RESOURCES
...
hostA         !        !       1      3.5    ()     ()    ()
hostB         !        !       1      3.5    ()     ()    (gaussian)
hostC         !        !       1      3.5    ()     ()    ()
...
End Host
```

Submitting Gaussian jobs

Use `bsub` to submit the job, including parameters required for Gaussian.

Using LSF with Lion Bioscience SRS

SRS is Lion Bioscience's Data Integration Platform, in which data is extracted by all other Lion Bioscience applications or third-party products. LSF works with the batch queue feature of SRS to provide load sharing and allow users to manage their running and completed jobs.

Requirements

- Platform LSF HPC features enabled
- SRS 6.1 and higher, available from Lion Bioscience

Configuring LSF for SRS jobs

During installation, `lsfinstall` adds the Boolean resource `lion` to the Resource section of `lsf.shared`.

Host configuration (optional)

If only some of your hosts can accept SRS jobs, configure the Host section of `lsf.cluster.`*`cluster_name`* to identify those hosts.

Edit `LSF_ENVDIR/conf/lsf.cluster.cluster_name` file and add the `lion` resource to the hosts that can run SRS jobs:

```
Begin Host
HOSTNAME     model    type   server   r1m    mem    swp   RESOURCES
...
hostA         !        !       1      3.5    ()     ()    ()
hostB         !        !       1      3.5    ()     ()    (lion)
hostC         !        !       1      3.5    ()     ()    ()
...
End Host
```

SRS batch queues

You must also configure SRS for batch queues. When SRS batch queueing is enabled, users select from the available batch queues displayed next to the application button in the page.

See the SRS administration manual for information about setting up a batch queue system. No additional configuration is required in LSF.

Submitting and monitoring SRS jobs

Submitting jobs

Use `bsub` to submit the job, including parameters required for SRS.

Monitoring jobs

As soon as the application is submitted, you can monitor the progress of the job. When applications are launched and batch queues are in use, an icon appears. The icon looks like a "new mail" icon in an email program when jobs are running, and looks like a "read mail" icon when all launched jobs are complete. You can click this icon at any time to:

- Check the status of running jobs
- See which jobs have completed
- Delete jobs
- Kill running jobs

You can also view the application results or launch another application against those results, using the results of the initial job as input for the next job.

See the *SRS Administrator's Manual* for more information.

Using LSF with LSTC LS-DYNA

LSF is integrated with products from Livermore Software Technology Corporation (LSTC). LS-DYNA jobs can use the checkpoint and restart features of LSF and take advantage of both SMP and distributed MPP parallel computation.

To submit LS-DYNA jobs through LSF, you only need to make sure that your jobs are checkpointable.

See *Administering Platform LSF* for more information about checkpointing in LSF.

Requirements

- Platform LSF HPC features enabled
- LS-DYNA version 960 and higher, available from LSTC

Optional requirements

- Hardware vendor-supplied MPI environment for network computing
- LSF MPI integration

Configuring LSF for LS-Dyna jobs

During installation, `lsfinstall` adds the Boolean resource `ls_dyna` to the Resource section of `lsf.shared`.

LSF also installs the `echkpnt.ls_dyna` and `erestart.ls_dyna` files in LSF_SERVERDIR.

Host configuration (optional)

If only some of your hosts can accept LS-DYNA jobs, configure the Host section of `lsf.cluster.`*`cluster_name`* to identify those hosts.

Edit `LSF_ENVDIR/conf/lsf.cluster.cluster_name` file and add the `ls_dyna` resource to the hosts that can run LS-DYNA jobs:

```
Begin Host
HOSTNAME    model    type   server   r1m    mem    swp   RESOURCES
...
hostA       !        !        1      3.5    ()     ()    ()
hostB       !        !        1      3.5    ()     ()    (ls_dyna)
hostC       !        !        1      3.5    ()     ()    ()
...
End Host
```

LS-DYNA integration with LSF checkpointing

LS-DYNA is integrated with LSF to use the LSF checkpointing capability. It uses application-level checkpointing, working with the functionality implemented by LS-DYNA. At the end of each time step, LS-DYNA looks for the existence of a checkpoint trigger file, named D3KIL.

LS-DYNA jobs always exit with 0 even when checkpointing. LSF will report that the job has finished when it has checkpointed.

Use the `bchkpnt` command to create the checkpoint trigger file, `D3KIL`, which LS-DYNA reads. The file forces LS-DYNA to checkpoint, or checkpoint and exit itself. The existence of a `D3KIL` file and the checkpoint information that LSF writes to the checkpoint directory specified for the job are all LSF needs to restart the job.

Checkpointing and tracking of resources of SMP jobs is supported.

With pam and Task Starter, you can track resources of MPP jobs, but cannot checkpoint. If you do not use pam and Task Starter, checkpointing of MPP jobs is supported, but tracking is not.

echkpnt and erestart

LSF installs `echkpnt.ls_dyna` and `erestart.ls_dyna`, which are special versions of `echkpnt` and `erestart` to allow checkpointing with LS-DYNA. Use `bsub -a ls_dyna` to make sure your job uses these files.

The method name `ls_dyna`, uses the `esub` for LS-DYNA jobs, which sets the checkpointing method `LSB_ECHKPNT_METHOD="ls_dyna"` to use `echkpnt.ls_dyna` and `erestart.ls_dyna`.

Checkpoint directories

When you submit a checkpointing job, you specify a checkpoint directory.

Before the job starts running, LSF sets the environment variable LSB_CHKPNT_DIR to a subdirectory of the checkpoint directory specified in the command line, or the CHKPNT parameter in lsb.queues. This subdirectory is identified by the job ID and only contains files related to the submitted job.

For checkpointing to work when running an LS-DYNA job from LSF, you must CD to the directory that LSF sets in `$LSB_CHKPNT_DIR` after submitting LS-DYNA jobs. You must change to this directory whether submitting a single job or multiple jobs. LS-DYNA puts all its output files in this directory.

Checkpoint trigger files

When you checkpoint a job, LSF creates a checkpoint trigger file named `D3KIL` in the working directory of the job.

The `D3KIL` file contains an entry depending on the desired checkpoint outcome:

- `sw1.` causes the job to checkpoint and exit. LS-DYNA writes to a restart data file `d3dump` and exits.
- `sw3.` causes the job to checkpoint and continue running. LS-Dyna writes to a restart data file `d3dump` and continues running until the next checkpoint.

The other possible LS-Dyna switch parameters are not relevant to LSF checkpointing.

LS-DYNA does not remove the `D3KIL` trigger file after checkpointing the job.

Restarting Jobs

If a job is restarted, LSF attempts to restart the job with the `-r` *`restart_file`* option used to replace any existing `-i` or `-r` options in the original LS-DYNA command. LS-DYNA uses the checkpointed data to restart the process from that checkpoint point, rather than starting the entire job from the beginning.

Each time a job is restarted, it is assigned a new job ID, and a new job subdirectory is created in the checkpoint directory. Files in the checkpoint directory are never deleted by LSF, but you may choose to remove old files once the LS-DYNA job is finished and the job history is no longer required.

Submitting LS-DYNA jobs

To submit DYNA jobs, redirect a job script to the standard input of `bsub`, including parameters required for checkpointing. With job scripts, you can manage two limitations of LS-DYNA job submissions:

- When LS-DYNA jobs are restarted from a checkpoint, the job will use the checkpoint environment instead of the job submission environment. You can restore your job submission environment if you submit your job with a job script that includes your environment settings.
- LS-DYNA jobs must run in the directory that LSF sets in the LSB_CHKPNT_DIR environment variable. This lets you submit multiple LS-DYNA jobs from the same directory but is also required if you are submitting one job. If you submit a job from a different directory, you must change to the `$LSB_CHKPNT_DIR` directory. You can do this if you submit your jobs with a job script.

> If you are running a single job or multiple jobs, all LS_DYNA jobs must run in the $LSB_CHKPT_DIR directory.

To submit LS-DYNA jobs with job submission scripts, embed the LS-DYNA job in the job script. Use the following format to run the script:

```
% bsub < jobscript
```

bsub syntax Inside your job scripts, the syntax for the `bsub` command to submit an LS-DYNA job is either of the following:

[**`-R ls_dyna`**] **`-k`** "*checkpoint_dir* **`method=ls_dyna`**" | **`-k`** "*checkpoint_dir* [*checkpoint_period*] **`method=ls_dyna`**" [*bsub_options*] *LS_DYNA_command* [*LS_DYNA_options*]

OR:

[**`-R ls_dyna`**] **`-a ls_dyna -k`** "*checkpoint_dir*" | **`-k`** "*checkpoint_dir* [*checkpoint_period*]" [*bsub options*] *LS_DYNA_command* [*LS_DYNA_options*]

`-R ls_dyna` Optional. Specify the `ls_dyna` shared resource if the LS-DYNA application is only installed on certain hosts in the cluster.

`method=ls_dyna` Mandatory. Use the `esub` for LS-DYNA jobs, which automatically sets the checkpoint method to `ls_dyna` to use the checkpoint and restart programs `echkpnt.ls_dyna` and `erestart.ls_dyna`. Alternatively, use `bsub -a` to specify the `ls_dyna` `esub`.

The checkpointing feature for LS-DYNA jobs requires all of the following parameters:

`-k` ***checkpoint_dir***

Mandatory. Regular option to `bsub` that specifies the name of the checkpoint directory. Specify the `ls_dyna` method here if you do not use the `bsub -a` option.

checkpoint_period

Regular option to `bsub` that specifies the time interval in minutes that LSF will automatically checkpoint jobs.

LS_DYNA_command

Regular LS-DYNA software command and options.

Preparing your job scripts

Environment variables Specify any environment variables required for your LS-DYNA jobs. For example:

```
LS_DYNA_ENV=VAL;export LS_DYNA_ENV
```

If you do not set your environment variables in the job script, then you must add some lines to the script to restore environment variables. For example:

```
if [ -f $LSB_CHKPNT_DIR/.envdump ]; then
.$LSB_CHKPNT_DIR/.envdump
fi
```

Change directory Ensure that your jobs run in the checkpoint directory set by LSF, by adding the following line after your bsub commands:

```
cd $LSB_CHKPNT_DIR
```

LS-DYNA command Write the LS-DYNA command you want to run. For example:

```
/usr/share/ls_dyna_path/ls960 endtime=2
i=/usr/share/ls_dyna_path/airbag.deploy.k ncpu=1
```

Example job scripts

All scripts must contain the `ls_dyna` method and the `cd` command to the checkpoint directory set by LSF.

- Job scripts with SMP LS-DYNA job embedded in the script. Environment variables are set in the script.

  ```
  % bsub < jobscript
  ```

  Example job submission script:

  ```
  #!/bin/sh
  #BSUB -J LS_DYNA
  #BSUB -k "/usr/share/checkpoint_dir method=ls_dyna"
  #BSUB -o "/usr/share/output/output.%J"
  cd $LSB_CHKPNT_DIR
  setenv LS_DYNA_VAR1 VAL1
  setenv LS_DYNA_VAR2 VAL2
  cp /usr/share/datapool/input.data /home/usr1/input.data
  /full_path/ls960 i=/home/usr1/input.data
  ```

- Job scripts with SMP LS-DYNA job embedded in the script. Environment variables are set in the script.

  ```
  % bsub < jobscript
  ```

  Example job submission script:

  ```
  #!/bin/sh
  #BSUB -J LS_DYNA
  #BSUB  -k "/usr/share/checkpoint_dir method=ls_dyna"
  cd $LSB_CHKPNT_DIR
  LS_DYNA_ENV=VAL;export LS_DYNA_ENV
  /usr/share/ls_dyna_path/ls960 endtime=2
  i=/usr/share/ls_dyna_path/airbag.deploy.k ncpu=1
  exit $?
  ```

- Job scripts with SMP LS-DYNA job embedded in the script. Environment variables are not set in the script, and the settings must be read from a hidden file, .envdump, which the `echkpnt.ls_dyna` program creates in the `$LSB_CHKPNT_DIR` directory. The script must source the `./`envdump file.

  ```
  % bsub < jobscript
  ```

Example job submission script:

```
#!/bin/sh
#BSUB -J LS_DYNA
#BSUB  -k "/usr/share/checkpoint_dir method=ls_dyna"
cd $LSB_CHKPNT_DIR
#after the first checkpoint
if [ -f $LSB_CHKPNT_DIR/.envdump ]; then
.$LSB_CHKPNT_DIR/.envdump
fi
/usr/share/ls_dyna_path/ls960 endtime=2
i=/usr/share/ls_dyna_path/airbag.deploy.k ncpu=1
exit $?
```

- Job script running MPP LS-DYNA job embedded in the script. Without PAM and TaskStarter, the job can be checkpointed, but not resource usage or job control are available.

  ```
  % bsub < jobscript
  ```

  `Example job submission script:`

  ```
  #!/bin/sh
  #BSUB -J LS_DYNA
  #BSUB -k "/usr/share/checkpoint_dir method=ls_dyna"
  #BSUB -o "/usr/share/output/output.%J"
  #BSUB -n 4
  cd $LSB_CHKPNT_DIR
  setenv ENV1 ENV1_VAL
  setenv ENV2 ENV2_VAL
  cp /usr/share/datapool/input.data /home/usr1/input.data
  mpirun /ls_dyna_mpp_path/mpp960 i=/home/usr1/input.data
  ```

- Job script with `lammpi` wrapper running MPP LS-DYNA job embedded in the script`.` PAM and TaskStarter ensures job control and resource usage information, but the job *cannot* be checkpointed.

  ```
  % bsub < jobscript
  ```

  Example job submission script:

  ```
  #!/bin/sh
  #BSUB -J LS_DYNA
  #BSUB -q priority
  #BSUB -n 1
  #BSUB -o /usr/share/output/output.%J
  #BSUB -k "/usr/share/checkpoint_dir method=ls_dyna"
  export PATH=/usr/share/jdk/bin:$PATH
  cd $LSB_CHKPNT_DIR
  pam -g 1 lammpirun_wrapper
  /usr/share/ls_dyna_mpp_path/mpp960
  i=/usr/share/DYNA/airbag.deploy.k
  ```

See *Administering Platform LSF* for information about submitting jobs with job scripts.

Checkpointing, restarting, and migrating LS-DYNA jobs

Checkpointing

`bchkpnt` `[`*`bchkpnt_options`*`] [`**`-k`**`] [`*`job_ID`*`]`

- **`-k`**

Specifies checkpoint and exit. The job will be killed immediately after being checkpointed. When the job is restarted, it continues from the last checkpoint.

- *job_ID*

  Job ID of the LS-DYNA job. Specifies which job to checkpoint. Each time the job is migrated, the job is restarted and assigned a new job ID.

See *Platform LSF Command Reference* for more information about `bchkpnt`.

Restarting

`brestart` `[`*`brestart_options`*`]` *`checkpoint_directory`* `[`*`job_ID`*`]`

- *checkpoint_directory*

  Specifies the checkpoint directory, where the job subdirectory is located. Each job is run in a unique directory.

  To change to the checkpoint directory for LSF to restart a job, place the following line in your job script before the LS-DYNA command is called:

  ```
  cd $LSB_CHKPNT_DIR
  ```

- *job_ID*

  Job ID of the LS-DYNA job, specifies which job to restart. After the job is restarted, it is assigned a new job ID, and the new job ID is used for checkpointing. A new job ID is assigned each time the job is restarted.

See *Platform LSF Command Reference* for more information about `brestart`.

Migrating

`bmig` `[`*`bsub_options`*`]` `[`*`job_ID`*`]`

- *job_ID*

  Job ID of the LS-DYNA job, specifies which job to migrate. After the job is migrated, it is restarted and assigned a new job ID. The new job ID is used for checkpointing. A new job ID is assigned each time the job is migrated.

See *Platform LSF Command Reference* for more information about `bmig`.

Using LSF with MSC Nastran

MSC Nastran Version 70.7.2 ("Nastran") runs in a Distributed Parallel mode, and automatically detects a job launched by LSF, and transparently accepts the execution host information from LSF.

The Nastran application checks if the LSB_HOSTS or LSB_MCPU_HOSTS environment variable is set in the execution environment. If either is set, Nastran uses the value of the environment variable to produce a list of execution nodes for the solver command line. Users can override the hosts chosen by LSF to specify their own host list.

Requirements

- Platform LSF HPC features enabled
- Nastran version 70.7.2 and higher, available from MSC Software

Configuring LSF for Nastran jobs

During installation, `lsfinstall` adds the Boolean resource `nastran` to the Resource section of `lsf.shared`.

No additional executable files are needed.

Host configuration (optional)

If only some of your hosts can accept Nastran jobs, configure the Host section of `lsf.cluster.`*`cluster_name`* to identify those hosts.

Edit `LSF_ENVDIR/conf/lsf.cluster.cluster_name` file and add the `nastran` resource to the hosts that can run Nastran jobs:

```
Begin Host
HOSTNAME    model    type   server   r1m    mem    swp   RESOURCES
...
hostA        !        !        1      3.5    ()     ()    ()
hostB        !        !        1      3.5    ()     ()    (nastran)
hostC        !        !        1      3.5    ()     ()    ()
...
End Host
```

Submitting Nastran jobs

Use `bsub` to submit the job, including parameters required for the Nastran command line.

Syntax

bsub -n *num_processors* [**-R nastran**] *bsub_options* *nastran_command*

- **-n** *num_processors*

  Number of processors required to run the job
- **-R nastran**

  Optional. Specify the `nastran` shared resource if the Nastran application is only installed on certain hosts in the cluster.

Nastran dmp variable

You must set the Nastran `dmp` variable to the same number as the number of processors the job is requesting (`-n` option of `bsub`).

Examples

- Parallel job through LSF requesting 4 processors:

  ```
  % bsub -n 4 -a nastran -R "nastran" nastran example dmp=4
  ```

Note that both the `bsub -n 4` and Nastran `dmp=4` options are used. The value for `-n` and `dmp` must be the same.

- Parallel job through LSF requesting 4 processors, no more than 1 processor per host:

  ```
  % bsub -n 4 -a nastran -R "nastran span[ptile=1]"
  nastran example dmp=4
  ```

Nastran on Linux using LAM/MPI

You must write a script that will pick up the LSB_HOSTS variable and provide the chosen hosts to the Nastran program. You can then submit the script using `bsub`:

```
bsub -a nastran lammpi -q hpc_linux -n 2 -o out -e err -R "span[ptile=1]"
lsf_nast
```

This will submit a 2-way job which will put its standard output in the file named `out` and standard error in a file named `err`. The `ptile=1` option tells LSF to choose at most 1 CPU per node chosen for the job.

Sample lsf_nast script

The following sample `lsf_nast` script only represents a starting point, but deals with the host specification for LAM/MPI. This script should be modified at your site before use.

```
#! /bin/sh
#
# lsf script to use with Nastran and LAM/MPI.
#
#
#Set information for Head node:
DAT=/home/user1/lsf/bc2.dat
#
#Set information for Cluster node:
TMPDIR=/home/user1/temp
#
LOG=${TMPDIR}/log
LSB_HOST_FILE=${TMPDIR}/lsb_hosts
:> ${LOG}
# The local host MUST be in the host file.
echo ${LSB_SUB_HOST} > ${LSB_HOST_FILE}
#
#
# Create the lam hosts file:
for HOST in $LSB_HOSTS
do
echo $HOST >> ${LSB_HOST_FILE}
done
#
cd ${TMPDIR}
rcp ${LSB_SUB_HOST}:${DAT} .
id
# recon -v ${LSB_HOST_FILE}
# cat ${LSB_HOST_FILE}
# pwd
recon -v ${LSB_HOST_FILE} >> ${LOG} 2>&1
```

```
lamboot -v ${LSB_HOST_FILE} >> ${LOG} 2>&1
NDMP=`sed -n -e '$=' ${LSB_HOST_FILE}`
HOST="n0"
(( i=1 ))
while (( i < $NDMP )) ; do
HOST="$HOST:n$i"
(( i += 1 ))
done
echo DAT=${DAT##*/}
pwd
nast707t2 ${DAT##*/} dmp=${NDMP} scr=yes bat=no hosts=$HOST >>
${LOG}
2>&1
wipe -v ${LSB_HOST_FILE} >> ${LOG} 2>&1
#
# Bring back files:
DATL=${DAT##*/}
rcp ${DATL%.dat}.log ${LSB_SUB_HOST}:${DAT%/*}
rcp ${DATL%.dat}.f04 ${LSB_SUB_HOST}:${DAT%/*}
rcp ${DATL%.dat}.f06 ${LSB_SUB_HOST}:${DAT%/*}
#
# End
```

CHAPTER

14

Using Platform LSF with the Etnus TotalView® Debugger

Contents

How LSF Works with TotalView

Platform LSF is integrated with Etnus TotalView® multiprocess debugger. You should already be familiar with using TotalView software and debugging parallel applications.

Debugging LSF jobs with TotalView

Etnus TotalView is a source-level and machine-level debugger for analyzing, debugging, and tuning multiprocessor or multithreaded programs. LSF works with TotalView two ways:

- Use LSF to start TotalView together with your job
- Start TotalView separately, submit your job through LSF and attach the processes of your job to TotalView for debugging

Once your job is running and its processes are attached to TotalView, you can debug your program as you normally would.

For more information See the *TotalView Users Guide* for information about using TotalView.

Installing LSF for TotalView

`lsfinstall` installs the application-specific `esub` program `esub.tvpoe` for debugging POE jobs in TotalView. It behaves like `esub.poe` and runs the `poejob` script, but it also sets the appropriate TotalView options and environment variables for POE jobs.

`lsfinstall` also configures `hpc_ibm_tv` queue for debugging POE jobs in `lsb.queues`. The queue is not rerunnable, does not allow interactive batch jobs (`bsub -I`), and specifies the following TERMINATE_WHEN action:

```
TERMINATE_WHEN=LOAD PREEMPT WINDOW
```

`lsfinstall` installs the following application-specific `esub` programs to use TotalView with LSF:

- Configures `hpc_linux_tv` queue for debugging LAM/MPI and MPICH-GM jobs in `lsb.queues.` The queue is not rerunnable, does not allow interactive batch jobs (`bsub -I`), and specifies the following TERMINATE_WHEN action:

  ```
  TERMINATE_WHEN=LOAD PREEMPT WINDOW
  ```

- `esub.tvlammpi`—for debugging LAM/MPI jobs in TotalView; behaves like `esub.lammpi`, but also sets the appropriate TotalView options and environment variables for LAM/MPI jobs, and sends the job to the `hpc_linux_tv` queue
- `esub.tvmpich_gm`—for debugging MPICH-GM jobs in TotalView; behaves like `esub,mpich_gm`, but also sets the appropriate TotalView options and environment variables for MPICH-GM jobs, and sends the job to the `hpc_linux_tv` queue

Environment variables for TotalView

On the submission host, make sure that:

- The path to the TotalView binary is in your $PATH environment variable
- $DISPLAY is set to `*console_name*:0.0`

Setting TotalView preferences

Before running and debugging jobs with TotalView, you should set the following options in your `$HOME/.preferences.tvd` file:

- `dset ignore_control_c {false}` to allow TotalView to respond to <CTRL-C>
- `dset ask_on_dlopen {false}` to tell TotalView not to prompt about stopping processes that use the `dlopen` system call

Limitations

While your job is running and you are using TotalView to debug it, you cannot use LSF job control commands:

- `bchkpnt` and `bmig` are not supported
- Default TotalView signal processing prevents `bstop` and `bresume` from suspending and resuming jobs, and `bkill` from terminating jobs
- `brequeue` causes TotalView to display all jobs in error status. Click and the jobs will rerun.
- Load thresholds and host dispatch windows do not affect jobs running in TotalView
- Preemption is not visible to TotalView
- Rerunning jobs within TotalView is not supported

Running Jobs for TotalView Debugging

Submit jobs two ways:

- Start a job and TotalView together through LSF
- Start TotalView and attach the LSF job

You must set the path to the TotalView binary in the $PATH environment variable on the submission host, and the $DISPLAY environment variable to `console_name:0.0`.

Compiling your program for debugging

Before using submitting your job in LSF for debugging in TotalView, compile your source code with the `-g` compiler option. This option generates the appropriate debugging information in the symbol table.

Any multiprocess programs that call `fork()`, `vfork()`, or `execve()` should be linked to the `dbfork` library.

See your compiler documentation and the *TotalView Users Guide* for more information about compiling programs for debugging.

Starting a job and TotalView together through LSF

Syntax **bsub -a tv***application* [*bsub_options*] **mpirun.lsf** *job* [*job_options*] [**-tvopt** *tv_options*]

-a tv*application* Specifies the application you want to run through LSF and debug in TotalView.

-tvopt *tv_options* Specifies options to be passed to TotalView. Use any valid TotalView command option, except `-a` (LSF uses this option internally). See the *TotalView Users Guide* for information about TotalView command options and setting up parallel debugging sessions.

Example To submit a POE job and run TotalView:

```
% bsub -a tvpoe -n 2 mpirun.lsf myjob -tvopt -no_ask_on_dlopen
```

The method name `tvpoe`, uses the special `esub` for debugging POE jobs with TotalView (`LSF_SERVERDIR/esub.tvpoe`). `-no_ask_on_dlopen` is a TotalView option that tells TotalView not to prompt about stopping processes that use the `dlopen` system call.

To submit a LAM/MPI job and run TotalView:

```
% bsub -a tvlammpi -n 2 mpirun.lsf myjob -tvopt -no_ask_on_dlopen
```

The method name `tvlammpi`, uses the special `esub` for debugging LAM/MPI jobs with TotalView (`LSF_SERVERDIR/esub.tvlammpi`). `-no_ask_on_dlopen` is a TotalView option that tells TotalView not to prompt about stopping processes that use the `dlopen` system call.

When the TotalView Root window opens:

1. TotalView automatically acquires the `pam` process and a Process window opens.
2. Click in the Process window to start debugging the process.

Depending on your TotalView preferences, you may see the Stop Before Going Parallel dialog. Click Yes. Use the Parallel page on the File > Preferences dialog to change the setting of When a job goes parallel or calls exec() radio buttons.

The process starts running and stops at the first breakpoint you set.

For MPICH-GM jobs, TotalView stops at two breakpoints: one in `pam`, and one in `MPI_init()`. Click to continue debugging.

3. Debug your job as you would normally in TotalView.

   When you are finished debugging your program, choose **File > Exit** to exit TotalView, and click Yes in the Exit dialog. As TotalView exits it kills the `pam` process. In a few moments, LSF detects that PAM has exited and your job exits as `Done successfully`.

Running TotalView and attaching a LSF job

Syntax **`bsub -a`** *application* [*bsub_options*] **`mpirun.lsf`** *job* [*job_options*]

`-a` *application* Specifies the application you want to run through LSF and debug in TotalView.

See the *TotalView Users Guide* for information about attaching jobs in TotalView and setting up parallel debugging sessions.

Example

1. Submit a job.

   For example:

```
% bsub -a poe -n 2 mpirun.lsf myjob
```

   The method name `poe`, uses the `esub` for running POE jobs (`LSF_SERVERDIR/esub.poe`).

```
% bsub -a mpich_gm -n 2 mpirun.lsf myjob
```

   The method name `mpich_gm`, uses the special `esub` for running MPICH-GM jobs (`LSF_SERVERDIR/esub.mpich_gm`).

2. Start TotalView on the execution host.

   For TotalView to load PAM, LSF_BINDIR must be in the $PATH environment variable on the execution host, or use **FIle > Search Path...** in TotalView to set the path to LSF_BINDIR.

   The TotalView Root window opens, and `pam` appears in the Unattached page of the TotalView Root window.

3. Double-click `pam` as the process to attach.

   A Process window opens. Your jobs move from the Unattached page to the Attached page.

   You should see all of your job processes in the Unattached page; you can select any process to attach, but to attach all parallel task on the local and remote hosts, you must attach to pam.

4. Click **Go** in the Process window?
5. Debug your job as you would normally in TotalView.

When you are finished debugging your program, choose **File > Exit** to exit TotalView, and click Yes in the Exit dialog. As TotalView exits it kills the `pam` process. In a few moments, LSF detects that PAM has exited and your job exits as `Done successfully`.

Viewing source code while debugging

Use **View > Lookup** Function to view the source code of your application while debugging. Enter `main` in the **Name** field and click **OK**. TotalView finds the source code for the `main()` function and displays it in the Source Pane.

See the *TotalView Users Guide* for information about displaying source code.

Controlling and Monitoring Jobs Being Debugged in TotalView

Controlling jobs

While your job is running and you are using TotalView to debug it, you cannot use LSF job control commands:

- `bchkpnt` and `bmig` are not supported
- Default TotalView signal processing prevents `bstop` and `bresume` from suspending and resuming jobs, and `bkill` from terminating jobs
- `brequeue` causes TotalView to display all jobs in error status. Click **Go** and the jobs will rerun.
- Job rerun within TotalView is not supported. Do not submit jobs for debugging to a rerunnable queue.

Monitoring jobs

Use `bjobs` to see the resource usage of jobs running under TotalView:

```
bsub -n 2 -a tvmpich_gm mpirun.lsf ./cpi -tvopt -no_ask_on_dlopen
Job <365> is submitted to queue <hpc_linux>.
bjobs -l 365

Job <365>, User <user1>, Project <default>, Status <DONE>, Queue
<hpc_linux>,
                     Command <totalview pam   -no_ask_on_dlopen -a -g 1
-tv gmmpirun_wrapper  ./cpi>
Fri Oct 11 15:46:47 2009: Submitted from host <hostA>, CWD <$HOME>, 2
Processors
                     Requested, Requested Resources <select[ (gm_ports >
0) ] rusage[gm_ports=1:duration=10]>;
Fri Oct 11 15:46:58 2009: Started on 2 Hosts/Processors <hostA> <hostB>,
Execution Home </home/user1>, Execution CWD
</home/user1>;
Fri Oct 11 15:53:07 2009: Done successfully. The CPU time used is 69.7 seconds.

 SCHEDULING PARAMETERS:
           r15s   r1m  r15m   ut      pg    io   ls    it    tmp    swp
mem
 loadSched   -     -     -     -       -     -    -     -     -      -
-
 loadStop    -     -     -     -       -     -    -     -     -      -
-

          adapter_windows
 loadSched      -           -               -
 loadStop       -           -               -

% bsub -a tvpoe -n 4 mpirun.lsf $JOB
Job <341> is submitted to queue <hpc_ibm>.
```

```
% bjobs -l 341
Job <341>, User <user1>, Project <default>, Status <DONE>, Queue <hpc_ibm>, Com
                     mand <totalview pam   -a -g 1 -tv poejob
/home/user1/cpi.poe >
Wed Oct 16 09:59:42 2009: Submitted from host <hostA>, CWD </home/user1,
                     4 Processors Requested;
Wed Oct 16 09:59:53 2009: Started on 4 Hosts/Processors <hostA>
                     <hostA> <hostA> <q
                     ataix05.lsf.platform.com>, Execution Home </home/user1>, E
                     xecution CWD </home/user1>;
Wed Oct 16 10:01:19 2009: Done successfully. The CPU time used is 97.0 seconds.

 SCHEDULING PARAMETERS:
           r15s   r1m  r15m   ut      pg    io   ls    it    tmp    swp    mem
 loadSched   -     -     -     -       -     -    -     -     -      -      -
 loadStop    -     -     -     -       -     -    -     -     -      -      -

          lammpi_load adapter_windows
 loadSched      -           -               -
 loadStop       -           -
```

Index