NAME

orterun, mpirun, mpiexec – Execute serial and parallel jobs in Open MPI. oshrun, shmemrun – Execute serial and parallel jobs in Open SHMEM.

Note: *mpirun*, *mpiexec*, and *orterun* are all synonyms for each other as well as *oshrun*, *shmemrun* in case Open SHMEM is installed. Using any of the names will produce the same behavior.

SYNOPSIS

Single Process Multiple Data (SPMD) Model:

mpirun [options] **<program>** [<args>]

Multiple Instruction Multiple Data (MIMD) Model:

mpirun [global_options]
[local_options1] **<program1>** [<args1>] :
[local_options2] **<program2>** [<args2>] :
... :
[local_optionsN] **<programN>** [<argsN>]

Note that in both models, invoking *mpirun* via an absolute path name is equivalent to specifying the *--prefix* option with a *<dir>* value equivalent to the directory where *mpirun* resides, minus its last subdirectory. For example:

% /usr/local/bin/mpirun ...

is equivalent to

% mpirun --prefix /usr/local

QUICK SUMMARY

If you are simply looking for how to run an MPI application, you probably want to use a command line of the following form:

% mpirun [-np X] [--hostfile <filename>] <program>

This will run X copies of *<program>* in your current run-time environment (if running under a supported resource manager, Open MPI's *mpirun* will usually automatically use the corresponding resource manager process starter, as opposed to, for example, *rsh* or *ssh*, which require the use of a hostfile, or will default to running all X copies on the localhost), scheduling (by default) in a round-robin fashion by CPU slot. See the rest of this page for more details.

Please note that mpirun automatically binds processes as of the start of the v1.8 series. Three binding patterns are used in the absence of any further directives:

Bind to core: when the number of processes is <= 2

Bind to socket: when the number of processes is > 2

Bind to none: when oversubscribed

If your application uses threads, then you probably want to ensure that you are either not bound at all (by specifying --bind-to none), or bound to multiple cores using an appropriate binding level or specific number of processing elements per application process.

OPTIONS

mpirun will send the name of the directory where it was invoked on the local node to each of the remote nodes, and attempt to change to that directory. See the "Current Working Directory" section below for further details.

<program>
The program executable. This is identified as the first non-recognized argument to mpirun.

<args> Pass these run-time arguments to every new process. These must always be the last arguments to *mpirun*. If an app context file is used, *<args>* will be ignored.

-h, **--help** Display help for this command

-q, **--quiet** Suppress informative messages from orterun during application execution.

-v, **--verbose**
Be verbose

-V, **--version**
Print version number. If no other arguments are given, this will also cause orterun to exit.

-N <num>
Launch num processes per node on all allocated nodes (synonym for npernode).

-display-map, **--display-map**
Display a table showing the mapped location of each process prior to launch.

-display-allocation, **--display-allocation**
Display the detected resource allocation.

-output-proctable, **--output-proctable**
Output the debugger proctable after launch.

-dvm, **--dvm**
Create a persistent distributed virtual machine (DVM).

-max-vm-size, **--max-vm-size** <size>
Number of processes to run.

-novm, **--novm**
Execute without creating an allocation-spanning virtual machine (only start daemons on nodes hosting application procs).

-hnp, **--hnp** <arg0>
Specify the URI of the Head Node Process (HNP), or the name of the file (specified as file:filename) that contains that info.

Use one of the following options to specify which hosts (nodes) of the cluster to run on. Note that as of the start of the v1.8 release, mpirun will launch a daemon onto each host in the allocation (as modified by the following options) at the very beginning of execution, regardless of whether or not application processes will eventually be mapped to execute there. This is done to allow collection of hardware topology information from the remote nodes, thus allowing us to map processes against known topology. However, it is a change from the behavior in prior releases where daemons were only launched after mapping was complete, and thus only occurred on nodes where application processes would actually be executing.

-H, **-host**, **--host** <host1,host2,...,hostN>
List of hosts on which to invoke processes.

-hostfile, **--hostfile** <hostfile>
Provide a hostfile to use.

-default-hostfile, **--default-hostfile** <hostfile>
Provide a default hostfile.

-machinefile, **--machinefile** <machinefile>
Synonym for *-hostfile*.

-cpu-set, **--cpu-set** <list>
Restrict launched processes to the specified logical cpus on each node (comma-separated list). Note that the binding options will still apply within the specified envelope - e.g., you can elect to bind each process to only one cpu within the specified cpu set.

The following options specify the number of processes to launch. Note that none of the options imply a particular binding policy - e.g., requesting N processes for each socket does not imply that the processes will be bound to the socket.

-c, **-n**, **--n**, **-np** <#>
Run this many copies of the program on the given nodes. This option indicates that the specified file is an executable program and not an application context. If no value is provided for the number of copies to execute (i.e., neither the "-np" nor its synonyms are provided on the command line), Open MPI will automatically execute a copy of the program on each process slot (see below for description of a "process slot"). This feature, however, can only be used in the SPMD model and will return an error (without beginning execution of the application) otherwise.

—map-by ppr:N:<object>
Launch N times the number of objects of the specified type on each node.

-npersocket, **--npersocket** <#persocket>
On each node, launch this many processes times the number of processor sockets on the node. The *-npersocket* option also turns on the *-bind-to-socket* option. (deprecated in favor of --map-by ppr:n:socket)

-npernode, **--npernode** <#pernode>
On each node, launch this many processes. (deprecated in favor of --map-by ppr:n:node)

-pernode, **--pernode**
On each node, launch one process -- equivalent to *-npernode* 1. (deprecated in favor of --map-by ppr:1:node)

To map processes:

--map-by <foo>
Map to the specified object, defaults to *socket*. Supported options include slot, hwthread, core, L1cache, L2cache, L3cache, socket, numa, board, node, sequential, distance, and ppr. Any object can include modifiers by adding a : and any combination of PE=n (bind n processing elements to each proc), SPAN (load balance the processes across the allocation), OVERSUBSCRIBE (allow more processes on a node than processing elements), and NOOVERSUBSCRIBE. This includes PPR, where the pattern would be terminated by another colon to separate it from the modifiers.

-bycore, **--bycore**
Map processes by core (deprecated in favor of --map-by core)

-byslot, **--byslot**
Map and rank processes round-robin by slot.

-nolocal, **--nolocal**
Do not run any copies of the launched application on the same node as orterun is running. This option will override listing the localhost with **--host** or any other host-specifying mechanism.

-nooversubscribe, **--nooversubscribe**
Do not oversubscribe any nodes; error (without starting any processes) if the requested number of processes would cause oversubscription. This option implicitly sets "max_slots" equal to the "slots" value for each node. (Enabled by default).

-oversubscribe, **--oversubscribe**
Nodes are allowed to be oversubscribed, even on a managed system, and overloading of processing elements.

-bynode, **--bynode**
Launch processes one per node, cycling by node in a round-robin fashion. This spreads processes evenly among nodes and assigns MPI_COMM_WORLD ranks in a round-robin, "by node" manner.

-cpu-list, **--cpu-list** <cpus>
List of processor IDs to bind processes to [default=NULL].

To order processes' ranks in MPI_COMM_WORLD:

--rank-by <foo>
Rank in round-robin fashion according to the specified object, defaults to *slot*. Supported options include slot, hwthread, core, L1cache, L2cache, L3cache, socket, numa, board, and node.

For process binding:

--bind-to <foo>
Bind processes to the specified object, defaults to *core*. Supported options include slot, hwthread, core, l1cache, l2cache, l3cache, socket, numa, board, and none.

-cpus-per-proc, **--cpus-per-proc** <#perproc>
Bind each process to the specified number of cpus. (deprecated in favor of --map-by <obj>:PE=n)

-cpus-per-rank, **--cpus-per-rank** <#perrank>
Alias for *-cpus-per-proc*. (deprecated in favor of --map-by <obj>:PE=n)

-bind-to-core, **--bind-to-core**
Bind processes to cores (deprecated in favor of --bind-to core)

-bind-to-socket, **--bind-to-socket**
Bind processes to processor sockets (deprecated in favor of --bind-to socket)

-report-bindings, **--report-bindings**
Report any bindings for launched processes.

For rankfiles:

-rf, **--rankfile** <rankfile>
Provide a rankfile file.

To manage standard I/O:

-output-filename, **--output-filename** <filename>
Redirect the stdout, stderr, and stddiag of all processes to a process-unique version of the specified filename. Any directories in the filename will automatically be created. Each output file will consist of filename.id, where the id will be the processes' rank in MPI_COMM_WORLD, left-filled with zero's for correct ordering in listings. A relative path value will be converted to an absolute path based on the cwd where mpirun is executed. Note that this *will not* work on environments where the file system on compute nodes differs from that where mpirun is executed.

-stdin, **--stdin** <rank>
The MPI_COMM_WORLD rank of the process that is to receive stdin. The default is to forward stdin to MPI_COMM_WORLD rank 0, but this option can be used to forward stdin to any process. It is also acceptable to specify *none*, indicating that no processes are to receive stdin.

-merge-stderr-to-stdout, **--merge-stderr-to-stdout**
Merge stderr to stdout for each process.

-tag-output, **--tag-output**
Tag each line of output to stdout, stderr, and stddiag with **[jobid, MCW_rank]<stdxxx>** indicating the process jobid and MPI_COMM_WORLD rank of the process that generated the output, and the channel which generated it.

-timestamp-output, **--timestamp-output**
Timestamp each line of output to stdout, stderr, and stddiag.

-xml, **--xml**
Provide all output to stdout, stderr, and stddiag in an xml format.

-xml-file, **--xml-file** <filename>
Provide all output in XML format to the specified file.

-xterm, **--xterm** <ranks>
Display the output from the processes identified by their MPI_COMM_WORLD ranks in separate xterm windows. The ranks are specified as a comma-separated list of ranges, with a -1 indicating all. A separate window will be created for each specified process. **Note:** xterm will normally terminate the window upon termination of the process running within it. However, by adding a "!" to the end of the list of specified ranks, the proper options will be provided to ensure that xterm keeps the window open *after* the process terminates, thus allowing you to see the process' output. Each xterm window will subsequently need to be manually closed. **Note:** In some environments, xterm may require that the executable be in the user's path, or be specified in absolute or relative terms. Thus, it may be necessary to specify a local executable as "./foo" instead of just "foo". If xterm fails to find the executable, mpirun will hang, but still respond correctly to a ctrl-c. If this happens, please check that the executable is being specified correctly and try again.

To manage files and runtime environment:

-path, **--path** <path>
<path> that will be used when attempting to locate the requested executables. This is used prior to using the local PATH setting.

--prefix <dir>
Prefix directory that will be used to set the *PATH* and *LD_LIBRARY_PATH* on the remote node before invoking Open MPI or the target process. See the "Remote Execution" section, below.

--noprefix
Disable the automatic --prefix behavior

-s, **--preload-binary**
Copy the specified executable(s) to remote machines prior to starting remote processes. The executables will be copied to the Open MPI session directory and will be deleted upon completion of the job.

--preload-files <files>
Preload the comma separated list of files to the current working directory of the remote machines where processes will be launched prior to starting those processes.

-set-cwd-to-session-dir, **--set-cwd-to-session-dir**
Set the working directory of the started processes to their session directory.

-wd <dir>
Synonym for *-wdir*.

-wdir <dir>
Change to the directory <dir> before the user's program executes. See the "Current Working Directory" section for notes on relative paths. **Note:** If the *-wdir* option appears both on the command line and in an application context, the context will take precedence over the command line. Thus, if the path to the desired wdir is different on the backend nodes, then it must be specified as an absolute path that is correct for the backend node.

-x <env>
Export the specified environment variables to the remote nodes before executing the program. Only one environment variable can be specified per *-x* option. Existing environment variables can be specified or new variable names specified with corresponding values. For example:
% mpirun -x DISPLAY -x OFILE=/tmp/out ...

The parser for the *-x* option is not very sophisticated; it does not even understand quoted values. Users are advised to set variables in the environment, and then use *-x* to export (not define) them.

Setting MCA parameters:

-gmca, **--gmca** <key> <value>
Pass global MCA parameters that are applicable to all contexts. *<key>* is the parameter name; *<value>* is the parameter value.

-mca, **--mca** <key> <value>
Send arguments to various MCA modules. See the "MCA" section, below.

-am <arg0>
Aggregate MCA parameter set file list.

-tune, **--tune** <tune_file>
Specify a tune file to set arguments for various MCA modules and environment variables. See the "Setting MCA parameters and environment variables from file" section, below.

For debugging:

-debug, **--debug**
Invoke the user-level debugger indicated by the *orte_base_user_debugger* MCA parameter.

--get-stack-traces
When paired with the **--timeout** option, *mpirun* will obtain and print out stack traces from all launched processes that are still alive when the timeout expires. Note that obtaining stack traces can take a little time and produce a lot of output, especially for large process-count jobs.

-debugger, **--debugger** <args>
Sequence of debuggers to search for when *--debug* is used (i.e. a synonym for *orte_base_user_debugger* MCA parameter).

--timeout <seconds>
The maximum number of seconds that *mpirun* (also known as *mpiexec*, *oshrun*, *orterun*, etc.) will run. After this many seconds, *mpirun* will abort the launched job and exit with a non-zero exit status. Using **--timeout** can be also useful when combined with the **--get-stack-traces** option.

-tv, **--tv**
Launch processes under the TotalView debugger. Deprecated backwards compatibility flag. Synonym for *--debug*.

There are also other options:

--allow-run-as-root
Allow *mpirun* to run when executed by the root user (*mpirun* defaults to aborting when launched as the root user).

--app <appfile>
Provide an appfile, ignoring all other command line options.

-cf, **--cartofile** <cartofile>
Provide a cartography file.

-continuous, **--continuous**
Job is to run until explicitly terminated.

-disable-recovery, **--disable-recovery**
Disable recovery (resets all recovery options to off).

-do-not-launch, **--do-not-launch**
Perform all necessary operations to prepare to launch the application, but do not actually launch it.

-do-not-resolve, **--do-not-resolve**
Do not attempt to resolve interfaces.

-enable-recovery, **--enable-recovery**
Enable recovery from process failure [Default = disabled].

-index-argv-by-rank, **--index-argv-by-rank**
Uniquely index argv[0] for each process using its rank.

-leave-session-attached, **--leave-session-attached**
Do not detach OmpiRTE daemons used by this application. This allows error messages from the daemons as well as the underlying environment (e.g., when failing to launch a daemon) to be output.

-max-restarts, **--max-restarts** <num>
Max number of times to restart a failed process.

-ompi-server, **--ompi-server** <uri or file>
Specify the URI of the Open MPI server (or the mpirun to be used as the server), the name of the file (specified as file:filename) that contains that info, or the PID (specified as pid:#) of the mpirun to be used as the server. The Open MPI server is used to support multi-application data exchange via the MPI-2 MPI_Publish_name and MPI_Lookup_name functions.

-personality, **--personality** <list>
Comma-separated list of programming model, languages, and containers being used (default="ompi").

--ppr <list>
Comma-separated list of number of processes on a given resource type [default: none].

-report-child-jobs-separately, **--report-child-jobs-separately**
Return the exit status of the primary job only.

-report-events, **--report-events** <URI>
Report events to a tool listening at the specified URI.

-report-pid, **--report-pid** <channel>
Print out mpirun's PID during startup. The channel must be either a '-' to indicate that the pid is to be output to stdout, a '+' to indicate that the pid is to be output to stderr, or a filename to which the pid is to be written.

-report-uri, **--report-uri** <channel>
Print out mpirun's URI during startup. The channel must be either a '-' to indicate that the URI is to be output to stdout, a '+' to indicate that the URI is to be output to stderr, or a filename to which the URI is to be written.

-show-progress, **--show-progress**
Output a brief periodic report on launch progress.

-terminate, **--terminate**
Terminate the DVM.

-use-hwthread-cpus, **--use-hwthread-cpus**
Use hardware threads as independent cpus.

-use-regexp, **--use-regexp**
Use regular expressions for launch.

The following options are useful for developers; they are not generally useful to most ORTE and/or MPI users:

-d, **--debug-devel**
Enable debugging of the OmpiRTE (the run-time layer in Open MPI). This is not generally useful for most users.

--debug-daemons
Enable debugging of any OmpiRTE daemons used by this application.

--debug-daemons-file
Enable debugging of any OmpiRTE daemons used by this application, storing output in files.

-display-devel-allocation, **--display-devel-allocation**
Display a detailed list of the allocation being used by this job.

-display-devel-map, **--display-devel-map**
Display a more detailed table showing the mapped location of each process prior to launch.

-display-diffable-map, **--display-diffable-map**
Display a diffable process map just before launch.

-display-topo, **--display-topo**
Display the topology as part of the process map just before launch.

-launch-agent, **--launch-agent**
Name of the executable that is to be used to start processes on the remote nodes. The default is "orted". This option can be used to test new daemon concepts, or to pass options back to the daemons without having mpirun itself see them. For example, specifying a launch agent of orted -mca odls_base_verbose 5 allows the developer to ask the orted for debugging output without clutter from mpirun itself.

--report-state-on-timeout
When paired with the **--timeout** command line option, report the run-time subsystem state of each process when the timeout expires.

There may be other options listed with *mpirun --help*.

Environment Variables

MPIEXEC_TIMEOUT
Synonym for the **--timeout** command line option.

DESCRIPTION

One invocation of *mpirun* starts an MPI application running under Open MPI. If the application is single process multiple data (SPMD), the application can be specified on the *mpirun* command line.

If the application is multiple instruction multiple data (MIMD), comprising of multiple programs, the set of programs and argument can be specified in one of two ways: Extended Command Line Arguments, and Application Context.

An application context describes the MIMD program set including all arguments in a separate file. This file essentially contains multiple *mpirun* command lines, less the command name itself. The ability to specify different options for different instantiations of a program is another reason to use an application context.

Extended command line arguments allow for the description of the application layout on the command line using colons (*:*) to separate the specification of programs and arguments. Some options are globally set across all specified programs (e.g. --hostfile), while others are specific to a single program (e.g. -np).

Specifying Host Nodes

Host nodes can be identified on the *mpirun* command line with the *-host* option or in a hostfile.

For example,

mpirun -H aa,aa,bb ./a.out
launches two processes on node aa and one on bb.

Or, consider the hostfile

```
% cat myhostfile
aa slots=2
bb slots=2
cc slots=2
```

Here, we list both the host names (aa, bb, and cc) but also how many "slots" there are for each. Slots indicate how many processes can potentially execute on a node. For best performance, the number of slots may

be chosen to be the number of cores on the node or the number of processor sockets. If the hostfile does not provide slots information, Open MPI will attempt to discover the number of cores (or hwthreads, if the use-hwthreads-as-cpus option is set) and set the number of slots to that value. This default behavior also occurs when specifying the *-host* option with a single hostname. Thus, the command

mpirun -H aa ./a.out
launches a number of processes equal to the number of cores on node aa.

mpirun -hostfile myhostfile ./a.out
will launch two processes on each of the three nodes.

mpirun -hostfile myhostfile -host aa ./a.out
will launch two processes, both on node aa.

mpirun -hostfile myhostfile -host dd ./a.out
will find no hosts to run on and abort with an error. That is, the specified host dd is not in the specified hostfile.

When running under resource managers (e.g., SLURM, Torque, etc.), Open MPI will obtain both the hostnames and the number of slots directly from the resource manger.

Specifying Number of Processes

As we have just seen, the number of processes to run can be set using the hostfile. Other mechanisms exist.

The number of processes launched can be specified as a multiple of the number of nodes or processor sockets available. For example,

mpirun -H aa,bb -npersocket 2 ./a.out
launches processes 0-3 on node aa and process 4-7 on node bb, where aa and bb are both dual-socket nodes. The *-npersocket* option also turns on the *-bind-to-socket* option, which is discussed in a later section.

mpirun -H aa,bb -npernode 2 ./a.out
launches processes 0-1 on node aa and processes 2-3 on node bb.

mpirun -H aa,bb -npernode 1 ./a.out
launches one process per host node.

mpirun -H aa,bb -pernode ./a.out
is the same as *-npernode* 1.

Another alternative is to specify the number of processes with the *-np* option. Consider now the hostfile

```
% cat myhostfile
aa slots=4
bb slots=4
cc slots=4
```

Now,

mpirun -hostfile myhostfile -np 6 ./a.out
will launch processes 0-3 on node aa and processes 4-5 on node bb. The remaining slots in the hostfile will not be used since the *-np* option indicated that only 6 processes should be launched.

Mapping Processes to Nodes: Using Policies

The examples above illustrate the default mapping of process processes to nodes. This mapping can also be controlled with various *mpirun* options that describe mapping policies.

Consider the same hostfile as above, again with *-np* 6:

	node aa	node bb	node cc
mpirun	0 1 2 3	4 5	

```
mpirun --map-by node    0 3         1 4         2 5

mpirun -nolocal                     0 1 2 3     4 5
```

The *--map-by node* option will load balance the processes across the available nodes, numbering each process in a round-robin fashion.

The *-nolocal* option prevents any processes from being mapped onto the local host (in this case node aa). While *mpirun* typically consumes few system resources, *-nolocal* can be helpful for launching very large jobs where *mpirun* may actually need to use noticeable amounts of memory and/or processing time.

Just as *-np* can specify fewer processes than there are slots, it can also oversubscribe the slots. For example, with the same hostfile:

mpirun -hostfile myhostfile -np 14 ./a.out
: will launch processes 0-3 on node aa, 4-7 on bb, and 8-11 on cc. It will then add the remaining two processes to whichever nodes it chooses.

One can also specify limits to oversubscription. For example, with the same hostfile:

mpirun -hostfile myhostfile -np 14 -nooversubscribe ./a.out
: will produce an error since *-nooversubscribe* prevents oversubscription.

Limits to oversubscription can also be specified in the hostfile itself:

```
% cat myhostfile
aa slots=4 max_slots=4
bb         max_slots=4
cc slots=4
```

The *max_slots* field specifies such a limit. When it does, the *slots* value defaults to the limit. Now:

mpirun -hostfile myhostfile -np 14 ./a.out
: causes the first 12 processes to be launched as before, but the remaining two processes will be forced onto node cc. The other two nodes are protected by the hostfile against oversubscription by this job.

Using the *--nooversubscribe* option can be helpful since Open MPI currently does not get "max_slots" values from the resource manager.

Of course, *-np* can also be used with the *-H* or *-host* option. For example,

mpirun -H aa,bb -np 8 ./a.out
: launches 8 processes. Since only two hosts are specified, after the first two processes are mapped, one to aa and one to bb, the remaining processes oversubscribe the specified hosts.

And here is a MIMD example:

mpirun -H aa -np 1 hostname : -H bb,cc -np 2 uptime
: will launch process 0 running *hostname* on node aa and processes 1 and 2 each running *uptime* on nodes bb and cc, respectively.

Mapping, Ranking, and Binding: Oh My!

Open MPI employs a three-phase procedure for assigning process locations and ranks:

mapping Assigns a default location to each process

ranking Assigns an MPI_COMM_WORLD rank value to each process

binding Constrains each process to run on specific processors

The *mapping* step is used to assign a default location to each process based on the mapper being employed. Mapping by slot, node, and sequentially results in the assignment of the processes to the node level. In contrast, mapping by object, allows the mapper to assign the process to an actual object on each node.

Note: the location assigned to the process is independent of where it will be bound - the assignment is used solely as input to the binding algorithm.

The mapping of process processes to nodes can be defined not just with general policies but also, if

necessary, using arbitrary mappings that cannot be described by a simple policy. One can use the "sequential mapper," which reads the hostfile line by line, assigning processes to nodes in whatever order the hostfile specifies. Use the *-mca rmaps seq* option. For example, using the same hostfile as before:

mpirun -hostfile myhostfile -mca rmaps seq ./a.out

will launch three processes, one on each of nodes aa, bb, and cc, respectively. The slot counts don't matter; one process is launched per line on whatever node is listed on the line.

Another way to specify arbitrary mappings is with a rankfile, which gives you detailed control over process binding as well. Rankfiles are discussed below.

The second phase focuses on the *ranking* of the process within the job's MPI_COMM_WORLD. Open MPI separates this from the mapping procedure to allow more flexibility in the relative placement of MPI processes. This is best illustrated by considering the following two cases where we used the —map-by ppr:2:socket option:

	node aa	node bb
rank-by core	0 1 ! 2 3	4 5 ! 6 7
rank-by socket	0 2 ! 1 3	4 6 ! 5 7
rank-by socket:span	0 4 ! 1 5	2 6 ! 3 7

Ranking by core and by slot provide the identical result - a simple progression of MPI_COMM_WORLD ranks across each node. Ranking by socket does a round-robin ranking within each node until all processes have been assigned an MCW rank, and then progresses to the next node. Adding the *span* modifier to the ranking directive causes the ranking algorithm to treat the entire allocation as a single entity - thus, the MCW ranks are assigned across all sockets before circling back around to the beginning.

The *binding* phase actually binds each process to a given set of processors. This can improve performance if the operating system is placing processes suboptimally. For example, it might oversubscribe some multicore processor sockets, leaving other sockets idle; this can lead processes to contend unnecessarily for common resources. Or, it might spread processes out too widely; this can be suboptimal if application performance is sensitive to interprocess communication costs. Binding can also keep the operating system from migrating processes excessively, regardless of how optimally those processes were placed to begin with.

The processors to be used for binding can be identified in terms of topological groupings - e.g., binding to an l3cache will bind each process to all processors within the scope of a single L3 cache within their assigned location. Thus, if a process is assigned by the mapper to a certain socket, then a *—bind-to l3cache* directive will cause the process to be bound to the processors that share a single L3 cache within that socket.

To help balance loads, the binding directive uses a round-robin method when binding to levels lower than used in the mapper. For example, consider the case where a job is mapped to the socket level, and then bound to core. Each socket will have multiple cores, so if multiple processes are mapped to a given socket, the binding algorithm will assign each process located to a socket to a unique core in a round-robin manner.

Alternatively, processes mapped by l2cache and then bound to socket will simply be bound to all the processors in the socket where they are located. In this manner, users can exert detailed control over relative MCW rank location and binding.

Finally, *--report-bindings* can be used to report bindings.

As an example, consider a node with two processor sockets, each comprising four cores. We run *mpirun* with *-np 4 --report-bindings* and the following additional options:

```
% mpirun ... --map-by core --bind-to core
[...] ... binding child [...,0] to cpus 0001
[...] ... binding child [...,1] to cpus 0002
```

```
[...] ... binding child [...,2] to cpus 0004
[...] ... binding child [...,3] to cpus 0008

% mpirun ... --map-by socket --bind-to socket
[...] ... binding child [...,0] to socket 0 cpus 000f
[...] ... binding child [...,1] to socket 1 cpus 00f0
[...] ... binding child [...,2] to socket 0 cpus 000f
[...] ... binding child [...,3] to socket 1 cpus 00f0

% mpirun ... --map-by core:PE=2 --bind-to core
[...] ... binding child [...,0] to cpus 0003
[...] ... binding child [...,1] to cpus 000c
[...] ... binding child [...,2] to cpus 0030
[...] ... binding child [...,3] to cpus 00c0

% mpirun ... --bind-to none
```

Here, *--report-bindings* shows the binding of each process as a mask. In the first case, the processes bind to successive cores as indicated by the masks 0001, 0002, 0004, and 0008. In the second case, processes bind to all cores on successive sockets as indicated by the masks 000f and 00f0. The processes cycle through the processor sockets in a round-robin fashion as many times as are needed. In the third case, the masks show us that 2 cores have been bound per process. In the fourth case, binding is turned off and no bindings are reported.

Open MPI's support for process binding depends on the underlying operating system. Therefore, certain process binding options may not be available on every system.

Process binding can also be set with MCA parameters. Their usage is less convenient than that of *mpirun* options. On the other hand, MCA parameters can be set not only on the *mpirun* command line, but alternatively in a system or user mca-params.conf file or as environment variables, as described in the MCA section below. Some examples include:

mpirun option	MCA parameter key	value
--map-by core	rmaps_base_mapping_policy	core
--map-by socket	rmaps_base_mapping_policy	socket
--rank-by core	rmaps_base_ranking_policy	core
--bind-to core	hwloc_base_binding_policy	core
--bind-to socket	hwloc_base_binding_policy	socket
--bind-to none	hwloc_base_binding_policy	none

Rankfiles

Rankfiles are text files that specify detailed information about how individual processes should be mapped to nodes, and to which processor(s) they should be bound. Each line of a rankfile specifies the location of one process (for MPI jobs, the process' "rank" refers to its rank in MPI_COMM_WORLD). The general form of each line in the rankfile is:

```
rank <N>=<hostname> slot=<slot list>
```

For example:

```
$ cat myrankfile
rank 0=aa slot=1:0-2
rank 1=bb slot=0:0,1
rank 2=cc slot=1-2
$ mpirun -H aa,bb,cc,dd -rf myrankfile ./a.out
```

Means that

Rank 0 runs on node aa, bound to logical socket 1, cores 0-2.
Rank 1 runs on node bb, bound to logical socket 0, cores 0 and 1.
Rank 2 runs on node cc, bound to logical cores 1 and 2.

Rankfiles can alternatively be used to specify *physical* processor locations. In this case, the syntax is somewhat different. Sockets are no longer recognized, and the slot number given must be the number of the physical PU as most OS's do not assign a unique physical identifier to each core in the node. Thus, a proper physical rankfile looks something like the following:

$ cat myphysicalrankfile
rank 0=aa slot=1
rank 1=bb slot=8
rank 2=cc slot=6

This means that

Rank 0 will run on node aa, bound to the core that contains physical PU 1
Rank 1 will run on node bb, bound to the core that contains physical PU 8
Rank 2 will run on node cc, bound to the core that contains physical PU 6

Rankfiles are treated as *logical* by default, and the MCA parameter rmaps_rank_file_physical must be set to 1 to indicate that the rankfile is to be considered as *physical*.

The hostnames listed above are "absolute," meaning that actual resolveable hostnames are specified. However, hostnames can also be specified as "relative," meaning that they are specified in relation to an externally-specified list of hostnames (e.g., by mpirun's --host argument, a hostfile, or a job scheduler).

The "relative" specification is of the form "+n<X>", where X is an integer specifying the Xth hostname in the set of all available hostnames, indexed from 0. For example:

$ cat myrankfile
rank 0=+n0 slot=1:0-2
rank 1=+n1 slot=0:0,1
rank 2=+n2 slot=1-2
$ mpirun -H aa,bb,cc,dd -rf myrankfile ./a.out

Starting with Open MPI v1.7, all socket/core slot locations are be specified as *logical* indexes (the Open MPI v1.6 series used *physical* indexes). You can use tools such as HWLOC's "lstopo" to find the logical indexes of socket and cores.

Application Context or Executable Program?

To distinguish the two different forms, *mpirun* looks on the command line for *--app* option. If it is specified, then the file named on the command line is assumed to be an application context. If it is not specified, then the file is assumed to be an executable program.

Locating Files

If no relative or absolute path is specified for a file, Open MPI will first look for files by searching the directories specified by the *--path* option. If there is no *--path* option set or if the file is not found at the *--path* location, then Open MPI will search the user's PATH environment variable as defined on the source node(s).

If a relative directory is specified, it must be relative to the initial working directory determined by the specific starter used. For example when using the rsh or ssh starters, the initial directory is $HOME by default. Other starters may set the initial directory to the current working directory from the invocation of *mpirun*.

Current Working Directory

The *−wdir* mpirun option (and its synonym, *−wd*) allows the user to change to an arbitrary directory before the program is invoked. It can also be used in application context files to specify working directories on specific nodes and/or for specific applications.

If the *−wdir* option appears both in a context file and on the command line, the context file directory will

override the command line value.

If the *-wdir* option is specified, Open MPI will attempt to change to the specified directory on all of the remote nodes. If this fails, *mpirun* will abort.

If the *-wdir* option is **not** specified, Open MPI will send the directory name where *mpirun* was invoked to each of the remote nodes. The remote nodes will try to change to that directory. If they are unable (e.g., if the directory does not exist on that node), then Open MPI will use the default directory determined by the starter.

All directory changing occurs before the user's program is invoked; it does not wait until *MPI_INIT* is called.

Standard I/O

Open MPI directs UNIX standard input to /dev/null on all processes except the MPI_COMM_WORLD rank 0 process. The MPI_COMM_WORLD rank 0 process inherits standard input from *mpirun*. **Note:** The node that invoked *mpirun* need not be the same as the node where the MPI_COMM_WORLD rank 0 process resides. Open MPI handles the redirection of *mpirun*'s standard input to the rank 0 process.

Open MPI directs UNIX standard output and error from remote nodes to the node that invoked *mpirun* and prints it on the standard output/error of *mpirun*. Local processes inherit the standard output/error of *mpirun* and transfer to it directly.

Thus it is possible to redirect standard I/O for Open MPI applications by using the typical shell redirection procedure on *mpirun*.

```
% mpirun -np 2 my_app < my_input > my_output
```

Note that in this example *only* the MPI_COMM_WORLD rank 0 process will receive the stream from *my_input* on stdin. The stdin on all the other nodes will be tied to /dev/null. However, the stdout from all nodes will be collected into the *my_output* file.

Signal Propagation

When orterun receives a SIGTERM and SIGINT, it will attempt to kill the entire job by sending all processes in the job a SIGTERM, waiting a small number of seconds, then sending all processes in the job a SIGKILL.

SIGUSR1 and SIGUSR2 signals received by orterun are propagated to all processes in the job.

A SIGTSTOP signal to mpirun will cause a SIGSTOP signal to be sent to all of the programs started by mpirun and likewise a SIGCONT signal to mpirun will cause a SIGCONT sent.

Other signals are not currently propagated by orterun.

Process Termination / Signal Handling

During the run of an MPI application, if any process dies abnormally (either exiting before invoking *MPI_FINALIZE*, or dying as the result of a signal), *mpirun* will print out an error message and kill the rest of the MPI application.

User signal handlers should probably avoid trying to cleanup MPI state (Open MPI is currently not async-signal-safe; see MPI_Init_thread(3) for details about *MPI_THREAD_MULTIPLE* and thread safety). For example, if a segmentation fault occurs in *MPI_SEND* (perhaps because a bad buffer was passed in) and a user signal handler is invoked, if this user handler attempts to invoke *MPI_FINALIZE*, Bad Things could happen since Open MPI was already "in" MPI when the error occurred. Since *mpirun* will notice that the process died due to a signal, it is probably not necessary (and safest) for the user to only clean up non-MPI state.

Process Environment

Processes in the MPI application inherit their environment from the Open RTE daemon upon the node on which they are running. The environment is typically inherited from the user's shell. On remote nodes, the exact environment is determined by the boot MCA module used. The *rsh* launch module, for example, uses either *rsh*/*ssh* to launch the Open RTE daemon on remote nodes, and typically executes one or more of the

user's shell-setup files before launching the Open RTE daemon. When running dynamically linked applications which require the *LD_LIBRARY_PATH* environment variable to be set, care must be taken to ensure that it is correctly set when booting Open MPI.

See the "Remote Execution" section for more details.

Remote Execution

Open MPI requires that the *PATH* environment variable be set to find executables on remote nodes (this is typically only necessary in *rsh*- or *ssh*-based environments -- batch/scheduled environments typically copy the current environment to the execution of remote jobs, so if the current environment has *PATH* and/or *LD_LIBRARY_PATH* set properly, the remote nodes will also have it set properly). If Open MPI was compiled with shared library support, it may also be necessary to have the *LD_LIBRARY_PATH* environment variable set on remote nodes as well (especially to find the shared libraries required to run user MPI applications).

However, it is not always desirable or possible to edit shell startup files to set *PATH* and/or *LD_LIBRARY_PATH*. The *--prefix* option is provided for some simple configurations where this is not possible.

The *--prefix* option takes a single argument: the base directory on the remote node where Open MPI is installed. Open MPI will use this directory to set the remote *PATH* and *LD_LIBRARY_PATH* before executing any Open MPI or user applications. This allows running Open MPI jobs without having pre-configured the *PATH* and *LD_LIBRARY_PATH* on the remote nodes.

Open MPI adds the basename of the current node's "bindir" (the directory where Open MPI's executables are installed) to the prefix and uses that to set the *PATH* on the remote node. Similarly, Open MPI adds the basename of the current node's "libdir" (the directory where Open MPI's libraries are installed) to the prefix and uses that to set the *LD_LIBRARY_PATH* on the remote node. For example:

Local bindir: /local/node/directory/bin

Local libdir: /local/node/directory/lib64

If the following command line is used:

% mpirun --prefix /remote/node/directory

Open MPI will add "/remote/node/directory/bin" to the *PATH* and "/remote/node/directory/lib64" to the *LD_LIBRARY_PATH* on the remote node before attempting to execute anything.

The *--prefix* option is not sufficient if the installation paths on the remote node are different than the local node (e.g., if "/lib" is used on the local node, but "/lib64" is used on the remote node), or if the installation paths are something other than a subdirectory under a common prefix.

Note that executing *mpirun* via an absolute pathname is equivalent to specifying *--prefix* without the last subdirectory in the absolute pathname to *mpirun*. For example:

% /usr/local/bin/mpirun ...

is equivalent to

% mpirun --prefix /usr/local

Exported Environment Variables

All environment variables that are named in the form OMPI_* will automatically be exported to new processes on the local and remote nodes. Environmental parameters can also be set/forwarded to the new processes using the MCA parameter *mca_base_env_list*. The *−x* option to *mpirun* has been deprecated, but the syntax of the MCA param follows that prior example. While the syntax of the *−x* option and MCA param allows the definition of new variables, note that the parser for these options are currently not very sophisticated - it does not even understand quoted values. Users are advised to set variables in the environment and use the option to export them; not to define them.

Setting MCA Parameters

The *-mca* switch allows the passing of parameters to various MCA (Modular Component Architecture) modules. MCA modules have direct impact on MPI programs because they allow tunable parameters to be set at run time (such as which BTL communication device driver to use, what parameters to pass to that BTL, etc.).

The *-mca* switch takes two arguments: *<key>* and *<value>*. The *<key>* argument generally specifies which MCA module will receive the value. For example, the *<key>* "btl" is used to select which BTL to be used for transporting MPI messages. The *<value>* argument is the value that is passed. For example:

mpirun -mca btl tcp,self -np 1 foo
Tells Open MPI to use the "tcp" and "self" BTLs, and to run a single copy of "foo" an allocated node.

mpirun -mca btl self -np 1 foo
Tells Open MPI to use the "self" BTL, and to run a single copy of "foo" an allocated node.

The *-mca* switch can be used multiple times to specify different *<key>* and/or *<value>* arguments. If the same *<key>* is specified more than once, the *<value>*s are concatenated with a comma (",") separating them.

Note that the *-mca* switch is simply a shortcut for setting environment variables. The same effect may be accomplished by setting corresponding environment variables before running *mpirun*. The form of the environment variables that Open MPI sets is:

OMPI_MCA_<key>=<value>

Thus, the *-mca* switch overrides any previously set environment variables. The *-mca* settings similarly override MCA parameters set in the $OPAL_PREFIX/etc/openmpi-mca-params.conf or $HOME/.openmpi/mca-params.conf file.

Unknown *<key>* arguments are still set as environment variable -- they are not checked (by *mpirun*) for correctness. Illegal or incorrect *<value>* arguments may or may not be reported -- it depends on the specific MCA module.

To find the available component types under the MCA architecture, or to find the available parameters for a specific component, use the *ompi_info* command. See the *ompi_info(1)* man page for detailed information on the command.

Setting MCA parameters and environment variables from file.

The *-tune* command line option and its synonym *-mca mca_base_envar_file_prefix* allows a user to set mca parameters and environment variables with the syntax described below. This option requires a single file or list of files separated by "," to follow.

A valid line in the file may contain zero or many "-x", "-mca", or “--mca” arguments. The following patterns are supported: -mca var val -mca var "val" -x var=val -x var. If any argument is duplicated in the file, the last value read will be used.

MCA parameters and environment specified on the command line have higher precedence than variables specified in the file.

Running as root

The Open MPI team strongly advises against executing *mpirun* as the root user. MPI applications should be run as regular (non-root) users.

Reflecting this advice, mpirun will refuse to run as root by default. To override this default, you can add the *--allow-run-as-root* option to the *mpirun* command line.

Exit status

There is no standard definition for what *mpirun* should return as an exit status. After considerable discussion, we settled on the following method for assigning the *mpirun* exit status (note: in the following description, the "primary" job is the initial application started by mpirun - all jobs that are spawned by that job are designated "secondary" jobs):

- if all processes in the primary job normally terminate with exit status 0, we return 0
- if one or more processes in the primary job normally terminate with non-zero exit status, we return the exit status of the process with the lowest MPI_COMM_WORLD rank to have a non-zero status
- if all processes in the primary job normally terminate with exit status 0, and one or more processes in a secondary job normally terminate with non-zero exit status, we (a) return the exit status of the process with the lowest MPI_COMM_WORLD rank in the lowest jobid to have a non-zero status, and (b) output a message summarizing the exit status of the primary and all secondary jobs.
- if the cmd line option --report-child-jobs-separately is set, we will return -only- the exit status of the primary job. Any non-zero exit status in secondary jobs will be reported solely in a summary print statement.

By default, OMPI records and notes that MPI processes exited with non-zero termination status. This is generally not considered an "abnormal termination" - i.e., OMPI will not abort an MPI job if one or more processes return a non-zero status. Instead, the default behavior simply reports the number of processes terminating with non-zero status upon completion of the job.

However, in some cases it can be desirable to have the job abort when any process terminates with non-zero status. For example, a non-MPI job might detect a bad result from a calculation and want to abort, but doesn't want to generate a core file. Or an MPI job might continue past a call to MPI_Finalize, but indicate that all processes should abort due to some post-MPI result.

It is not anticipated that this situation will occur frequently. However, in the interest of serving the broader community, OMPI now has a means for allowing users to direct that jobs be aborted upon any process exiting with non-zero status. Setting the MCA parameter "orte_abort_on_non_zero_status" to 1 will cause OMPI to abort all processes once any process
exits with non-zero status.

Terminations caused in this manner will be reported on the console as an "abnormal termination", with the first process to so exit identified along with its exit status.

EXAMPLES

Be sure also to see the examples throughout the sections above.

mpirun -np 4 -mca btl ib,tcp,self prog1
: Run 4 copies of prog1 using the "ib", "tcp", and "self" BTL's for the transport of MPI messages.

mpirun -np 4 -mca btl tcp,sm,self
: --mca btl_tcp_if_include eth0 prog1
Run 4 copies of prog1 using the "tcp", "sm" and "self" BTLs for the transport of MPI messages, with TCP using only the eth0 interface to communicate. Note that other BTLs have similar if_include MCA parameters.

RETURN VALUE

mpirun returns 0 if all processes started by *mpirun* exit after calling MPI_FINALIZE. A non-zero value is returned if an internal error occurred in mpirun, or one or more processes exited before calling MPI_FINALIZE. If an internal error occurred in mpirun, the corresponding error code is returned. In the event that one or more processes exit before calling MPI_FINALIZE, the return value of the MPI_COMM_WORLD rank of the process that *mpirun* first notices died before calling MPI_FINALIZE will be returned. Note that, in general, this will be the first process that died but is not guaranteed to be so.

If the **--timeout** command line option is used and the timeout expires before the job completes (thereby forcing *mpirun* to kill the job) *mpirun* will return an exit status equivalent to the value of **ETIMEDOUT** (which is typically 110 on Linux and OS X systems).

SEE ALSO

MPI_Init_thread(3)