ParaStation V5

# ParaStation MPI

—

# User's Guide

Release 5.0.7-4
Published August 2011

ParTec CLUSTER COMPETENCE CENTER

# *ParaStation MPI* User's Guide

Release 5.0.7-4
Copyright © 2002-2011 ParTec Cluster Competence Center GmbH
August 2011
Printed 9 August 2011, 16:26

ParTec Cluster Competence Center GmbH
Possartstr. 20
D-81679 München
Phone +49-89-99809-0
Fax +49-89-99809-555

http://www.par-tec.com
`<info@par-tec.com>`

The *ParaStation MPI Administrator's Guide* and *ParaStation MPI User's Guide* provide detailed information about *ParaStation MPI*. Installation and configuration of *ParaStation MPI* including process management and communication libraries are described in-depth by the *ParaStation MPI Administrator's Guide*. Usage of MPI and related tools are described by the *ParaStation MPI User's Guide*.

Though it may seem hard to believe, this manual might contain errors. We welcome any reports on errors or problems that are found. We also would appreciate suggestions on improving this book. Please direct all comments and problems to `<support@par-tec.com>`.

The most up-to-date version of this document is available at http://docs.par-tec.com.

> Share your knowledge with others. It's a way to achieve immortality.
>
> —Dalai Lama

# Table of Contents

# List of Figures

# Chapter 1. Preface

## 1.1. About this document

This book discusses the user interface and the related utilities of *ParaStation MPI*. This includes an adapted development framework for MPI applications based on MPIch. Furthermore *ParaStation MPI*'s process spawning mechanism is described and the potential ways to steer it are presented.

For a detailed discussion of the installation and configuration of *ParaStation MPI* and the administrators utilities in the standard distribution take a look at the *ParaStation MPI Administrator's Guide*. The *ParaStation MPI* programming interfaces documentation can be found in the API reference.

This document describes version 5.0 of the *ParaStation MPI* software. Previous versions of *ParaStation MPI* are no longer covered by this document. Information about this outdated versions can be found in previous versions of this document.

The most up-to-date version of this document is available at http://docs.par-tec.com.

## 1.2. This book's audience

This book is targeted to users compiling or running parallel applications on Linux HPC cluster within a MPI environment using *ParaStation MPI*.

The users should be familiar with the concepts of MPI, how to compile, link and run an application.

## 1.3. *ParaStation MPI* overview

*ParaStation MPI* is an integrated cluster management and communication solution. It combines unique features only found in *ParaStation MPI* with common techniques, widely used in high performance computing, to deliver an integrated, easy to use and reliable compute cluster environment.

The version 5 of *ParaStation MPI* supports various communication technologies as interconnect network. It comes with an optimized communication protocol for Ethernet that enables Gigabit Ethernet to play a new role in the market of high throughput, low latency communication. Beside InfiniBand and Myrinet, it also supports the upcoming 10G Ethernet networks.

Like previous versions, *ParaStation MPI* includes an integrated cluster administration and management environment. Using communicating daemon processes on each cluster node, an effective resource management and single point of administration is implemented. This results in a single system image view of the cluster.

From the user's point of view this cluster management leads to an easier and more effective usage of the cluster. Important features like load balancing, job control and input/output management, common in classical supercomputers, but rarely found in compute clusters, are implemented by *ParaStation MPI* thus being now also available on clusters.

## 1.4. The history of *ParaStation MPI*

The fundamentals of the *ParaStation* software were laid in 1995, when the *ParaStation* communication hardware and software system was presented. It was developed at the chair of Professor Tichy at computer science department of Karlsruhe University.

When in 1998 *ParaStation2* was presented, it was a pure software project. The communication platform used then was Myrinet, a Gigabit interconnect developed by Myricom. The development of *ParaStation2* still took place at the University of Karlsruhe.

*ParaStation* became commercial in 1999 when ParTec AG was founded. This spin-off from the University of Karlsruhe owned all rights and patents connected with the *ParaStation* software. ParTec promoted the further development and improvement of the software. This included the support of a broader basis of supported processor types, communication interconnect and operating systems.

Version 3 of the *ParaStation* software for Myrinet was a rewrite from scratch fully in the responsibility of ParTec. All the know-how and experiences achieved from the former versions of the software were incorporated into this version. It was presented in 2001 and was a major breakthrough with respect to throughput, latency and stability of the software. Nevertheless it was enhanced constantly with regard to performance, stability and usability.

In 2002 the *ParaStation FE* software was presented opening the *ParaStation* software environment towards Ethernet communication hardware. This first step in the direction of independence from the underlying communication hardware brought the convenient *ParaStation* management facility to Beowulf clusters for the first time. Furthermore the suboptimal communication performance for large packets gained from the MPIch/P4 implementation of the MPI message passing interface, the de facto standard on Beowulf clusters, was improved to the limits that may to be expected from the physical circumstances.

With *ParaStation4* presented in 2003 the software became really communication platform independent. With this version of the software even Gigabit Ethernet became a serious alternative as a cluster interconnect due to the throughput and latency that could be achieved.

In the middle of 2004, all rights on *ParaStation* where transferred from ParTec AG to the ParTec Cluster Competence Center GmbH. This new company takes a much more service-oriented approach to the customer. The main goal was to deliver integrated and complete software stacks for LINUX-based compute clusters by selecting state-of-the-art software components and driving software development efforts in areas where real added value can be provided. The ParTec Cluster Competence Center GmbH continued to develop and support the *ParaStation* product as an important part of it's portfolio.

At the end of 2007, *ParaStation MPI* was released supporting MPI2 and even more interconnects and especially protocols, like DAPL. *ParaStation MPI* is backward compatible to the previous *ParaStation4* version. At the same time, it was renamed to *ParaStation MPI*, as *ParaStation5* now includes more components beside a MPI.

# Chapter 2. Overview

Besides the brief overview on *ParaStation MPI* given within the introduction this chapter will give a more detailed insight concerning the building blocks forming the *ParaStation MPI* system and the main architecture of *ParaStation MPI*.

As already mentioned *ParaStation MPI* consists of the modules

- high performance communication subsystem and
- the cluster management facility.

The communication subsystem of *ParaStation MPI* is composed of a couple of libraries and kernel modules. Applications that want to benefit from the *ParaStation MPI* communication system have to be build against these libraries, except when the TCP bypass is used. Furthermore the cluster nodes running this application have to have the kernel module(s) loaded and the shared versions of the communication libraries loaded, if used.

The management part of *ParaStation MPI* is implemented in a daemon process, running on each of the cluster nodes. All these daemons constantly gather and interchange information in order to get a unique global view of the cluster. Applications that want to profit from this view to the cluster have to talk to this daemons. This is usually done via an interface implemented in another library. Thus parallel applications have to be linked against this library, too.

Both parts of *ParaStation MPI* will be discussed in detail within the next sections of this chapter.

In order to preserve the high communication bandwidth of the fast network to the applications, a strict separation between the application related communication traffic and the network traffic caused by administration tasks is made. Therefore, the *ParaStation MPI* communication subsystem is only used for the application traffic. The concept of splitting the two different types of communication is shown in Figure 2.1, “The *ParaStation MPI* network setup.”.

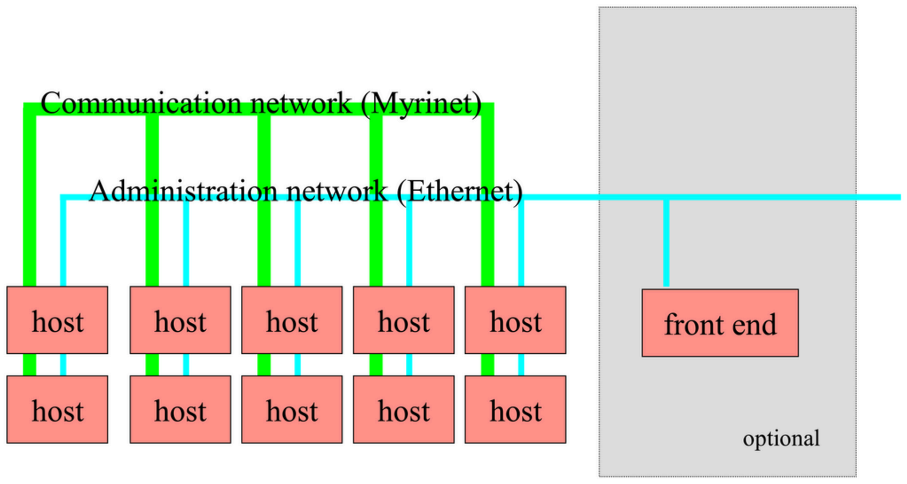
Communication network (Myrinet)
Administration network (Ethernet)
host
host
host
host
host
host
host
host
host
host
front end
optional

Figure 2.1. The *ParaStation MPI* network setup.

Often both types of communication share the same physical network. Especially for Ethernet based clusters, this is a common architecture. Albeit this may cause performance problems for some parallel applications.

As depicted in Figure 2.1, “The *ParaStation MPI* network setup.” it is possible to have an optional frontend machine not connected by the application network to the cluster, all the same being fully integrated into the *ParaStation MPI* management system. Thus, it is possible to start parallel applications on this machine, making it unnecessary for the user to login or even being able to login to every cluster node. Furthermore, the frontend machine might act as an `home`-directory and compile server for the users.

## 2.1. The communication subsystem

The *ParaStation MPI* high speed communication subsystem supports different communication paths and interconnect technologies. Depending on the physically available network(s) and the actual configuration, these interconnects are automatically selected by the communication library.

### 2.1.1. Communication paths

*ParaStation MPI* currently supports a variety of communication paths to transfer application data. Not all paths are always available, depending on the physical network(s) and system architecture (uni-processor or SMP) installed and the actual configuration.

The following list shows all currently available interconnects and protocols, supported by *ParaStation MPI*. The list also defines the order used to select a transport by the process management.

Shared Memory
: If two or more processes of a parallel task run on the same physical node, shared memory will be used for communication between these processes. This typically happens on SMP nodes, where one process per CPU is spawned.

InfiniBand using verbs
: If InfiniBand is available on this cluster, *ParaStation MPI* may make use of a verbs driver.

10G Ethernet or InfiniBand using DAPL
: Especially for 10G Ethernet, this interface should be prefered against p4sock or TCP, if available. The DAPL interface may also be used for InfiniBand.

Myrinet using GM
: This interconnect is only available, if Myrinet and GM is installed on this cluster.

Gigabit Ethernet using *ParaStation MPI* protocol p4sock
: This is the most effective and therefore preferred protocol for Gigabit Ethernet. For 10G Ethernet use p4sock only, if DAPL is not available. It is based on the *ParaStation* specific protocol `p4sock`.

Ethernet using TCP
: This is the default communication path. In fact, all connections providing TCP/IP can be used, independent of the underlying network.

Independent of the underlying transport networks and protocols in use, *ParaStation MPI* uses reliable communication.

While spawning processes on a cluster, *ParaStation MPI* will decide which interconnects can be used for communication. Using environment variables, the usage of particular interconnects and/or protocols may be controlled by the user. See ps_environment(7) for details.

### 2.1.2. Communication interfaces

*ParaStation MPI* provides several communication interfaces, suitable for different levels of functionality and environments.

**PSPort**
: The **PSPort** interface is the native low-level communication interface provided by *ParaStation MPI*. Any communication can be done using this interface, although programs using it will not be portable. Thus it is recommended to use a standard interface as MPI discussed below.

  The main features of the **PSPort** interface are:

  - Fragmentation/Defragmentation of large messages.

- Buffering of asynchronously received messages.
- Provision of ports.
- Selective receive.
- Thread save.
- Control Data possible in every message.

The PSPort interface is encapsulated in the library `libpsport.a`.

**TCP bypass**

This Linux kernel extension redirects network traffic within a cluster from the TCP layer to the *ParaStation MPI* p4sock protocol layer. Due to the very small overhead of this protocol, this bypass functionality increases performance and lowers latencies seen by the application.

The application does not recognize this redirection of network packets and needn't be modified in any way. The user may control the usage of the bypass by setting the LD_PRELOAD environment variable.

**MPI**

This is an implementation of the MPI Message Passing Interface. It is assumed as the standard interface in order to write parallel applications.

Its key features are:

- Based on MPIch2.
- Synchronous and asynchronous communication.
- Zero copy communication.
- Implements the whole standard of MPI-2.
- Support for MPI-IO to a PVFS parallel filesystem.

**RMI**

This interface enables even Java to profit from the high-performance communication provided by *ParaStation MPI*.

This interface is mainly a research project and thus not included within the standard distribution. The corresponding parts of *ParaStation MPI* may be obtained by request from ParTec. Please contact `<support@par-tec.com>`.

The main features of the **RMI** interface are:

- Enable Remote Method Invocation over all supported interconnects.
- Serialization of objects is provided.

## 2.2. The cluster management facility

In addition to the pure communication tasks that build the primary requirements of parallel application running on a cluster, further assistance of the runtime environment is needed in order to startup and control these applications.

The main requirements demanded from a cluster management framework are:

**Node management**

The management system should automatically detect usable nodes. The ability to administrate this nodes from a central point of administration has to be provided.

**Fault tolerance**

If some nodes are temporarily unavailable due to hardware failure, maintenance or any other reason, the management system must not fail. This situation has to be gracefully treated and normal work with the remaining nodes has to be provided.

**Process management**

The management system has to provide a transparent mechanism to start parallel tasks. Load balancing has to be considered within this process. Further requirements to special resources only provided by some nodes have to be taken into account. The actual behavior should be widely configurable.

**Job control & Error management**

If some or all processes forming a parallel task fail unexpectedly, the management system has to reset left over processes and clean up the accessed node in order to keep the cluster in a usable state.

*ParaStation MPI* fulfills all this requirements by providing a framework for cluster-wide job and resource management. The backbone of the *ParaStation MPI* management facility is build by the *ParaStation MPI* daemons psid(8) running on every node of the cluster, communicating underneath each other constantly. Thus the management system is not located on a single node but distributed throughout the whole cluster. This design prevents the creation of single points of failure.

The main tasks of the *ParaStation MPI* daemons are:

- recovery and distribution of local information like the local load values, the state of the locally controlled processes or the condition of local communication interface,
- status control of the cluster by receiving the information from all other daemons,
- initialization and the status control of the local communication layer,
- startup of local processes on request of local or remote processes,
- control of the locally started processes,
- transmission of input and output of the locally controlled processes,
- forwarding of signals to the locally controlled processes and
- provision of an interface to the *ParaStation MPI* cluster management environment.

### 2.2.1. Node management

In order to provide a fault tolerant node management, *ParaStation MPI* uses the concept of virtual nodes in contrast to the classical approach of handling statical node lists. The idea behind this concept is not to manage nodes by lists of hostnames provided by the user, but to provide a pool of virtual nodes and request nodes from this list at startup time. These virtual nodes of course represent the physical nodes available. Users normally do not request for special nodes but for a set of virtual nodes.

The main advantage of this concept becomes clear if one or more nodes are temporarily unavailable or even down. While the concept of static nodelists requires the user to change this lists manually, within the setup chosen by *ParaStation MPI* the pool becomes smaller because of the lacking nodes. On the other hand the user does not have to change the syntax or any configuration in order to get a required number of nodes, at least as long as enough nodes are available.

Albeit the request of selected nodes is not necessary due to the virtual node concept, it is still possible. This behavior is quite useful in the case where a virtual partitioning of the cluster controlled by an external batch system is desired.

Normally the user will request for any N nodes. *ParaStation MPI* then decides which nodes are available and fit best to the requirements of the user. These nodes will be given to the user and the parallel task will be spawned throughout these nodes.

Further request for virtual nodes posted by any other user will recognize the processes of this parallel task and take them into account when processes have to be spawned, too.

The details of process distribution within the startup of parallel tasks will be discussed in detail in Chapter 3, *Starting up programs*

### 2.2.2. Fault tolerance

Within *ParaStation MPI* fault tolerance is given in many ways. First of all the concept of virtual nodes provides stability against node failure in the sense that the cluster as a whole will work even if several nodes are temporarily unavailable.

In addition, *ParaStation MPI* offers mechanisms in order to increase the reliability and stability of the cluster system. E.g. if a node fails while it is utilized by parallel tasks, this task will be shut down in a controlled manner. Thus the remaining nodes that where used by this task will be cleaned up and released for further jobs.

The distributed concept of *ParaStation MPI*'s management facilities makes the administration of the cluster feasible even if some nodes are down. Furthermore it prevents the emerge of single point of failure that would lead to an unusable cluster due to a local failure on just one node.

### 2.2.3. Process management

The management facility of *ParaStation MPI* offers a complete process management system. *ParaStation MPI* recognizes and controls dependencies between processes building a parallel task on various nodes of the cluster.

The process management includes creation of processes on remote nodes, control of the I/O channels of these remotely started processes and the management of signals across node boundaries.

In contrast to the spawning mechanism used by many other cluster management systems, i.e. the spawning via a rsh/ssh mechanism, the startup of remote processes via *ParaStation MPI* is very fast since the *ParaStation MPI* daemon psid(8) is constantly in a standby mode in order to start these processes. No further login or authentication overhead is necessary.

Since *ParaStation MPI* knows about the dependencies between the processes building a parallel task it is able to take them into account. The processes are no longer independent but form a task in the same sense as the nodes no longer are independent computers but form the cluster as a unique system. The fact that *ParaStation MPI* handles distributed processes as a unit, plays an important rule especially in the context of job control and error management discussed within the next section.

Furthermore *ParaStation MPI* takes care that output produced by remote processes is forwarded to the intended destination. This is usually the controlling `tty` of the I/O handling process, i.e. the process that was initially started in order to bring up the parallel task, but might also be a normal file the output is redirected to. Input directed to the parallel task is forwarded, too. The default is to send it to the process with rank 0, but it might be addressed to any process of the parallel task.

Last but not least *ParaStation MPI* handles the propagation of signals. This means that signals send to the I/O handling process will be forwarded to all processes of the task.

### 2.2.4. Job control & error management

Beside the job control already discussed in the previous section, error management plays an important role in order to run a cluster without much administrative engagement on a day to day basis. It is not acceptable to have to clean up remaining processes of crashed parallel task by hand, to lose output due to erroneous processes or to leave the user without meaningful error messages after a parallel task has failed.

*ParaStation MPI* supports the administrator and the end user within this complex. Since the management facility controls all processes that were notified towards *ParaStation MPI*, it is capable to take actions in the case that one of the processes fails.

If an unexpected failure of a process is recognized, all processes within the corresponding parallel task will be notified. The parent process of the parallel task will be notified in any case, all other processes will be signaled only on request.

Furthermore all necessary measures will be taken in order to clean up the resources that were allocated by the crashed process.

Besides the fact that all output that was “on the line” when the failure took place will reach its final destination, the user will get feedback about the kind of error that crashed the process. This feedback is usually of the form “Got `sigxxx` on node `nodenum`”.

### 2.2.5. Integration into existing environments

Beside the inherent management capabilities of *ParaStation MPI* it is prepared for easy interaction with more evolved batch systems as e.g. LSF, OpenPBS or PBS PRO. This enables a *ParaStation MPI* cluster to get embedded into an existing environment or even to build a node within a Grid.

The integration of a *ParaStation MPI* cluster into an existing environment relieves the end user from learning just another runtime environment. Furthermore it makes the use of a cluster more stream lined with an existing site policy concerning the use of supercomputing facilities.

# Chapter 3. Starting up programs

*ParaStation MPI*'s management and task handling facilities solve all of the nasty problems seen on a compute cluster while handling parallel applications. *ParaStation MPI* implements a fast and robust startup mechanism for parallel tasks. In addition the *ParaStation MPI* daemons running on each node of the cluster control the various remotely started processes forming the Parallel Task in a combined effort. They also clean up the whole task if one of the processes dies unexpectedly.

Within this chapter it is assumed that an executable ready to run on a *ParaStation MPI* cluster does already exist. It might have been built by the end-user using the MPI framework as described in Chapter 4, *Using MPI*, have been provided by an independent software vendor or can be a native *ParaStation MPI* application only making use of the low-level communication and management functions as described in Chapter 5, *Building native ParaStation MPI applications*. Anyhow, it has to make use of the *ParaStation MPI* management library as described in the API reference.

This chapter describes how to start a *ParaStation MPI* aware executable, how the processes will be distributed within the cluster, how to control the distribution strategy used by *ParaStation MPI* and how the mechanism of input and output redirection works.

## 3.1. Spawning processes

Starting a parallel task consisting of N processes is as simple as executing:

```
 $ mpiexec -np N /some/path/to/program [args]
```

Given the default settings of *ParaStation MPI* the `N` instances of `program` will be distributed to N free processors in the cluster. For more details, refer to mpiexec(1).

The command `/some/path/to/program` has to exist and must be executable for the user on each node.

Likewise the current working directory must be accessible on each node. Otherwise, the user's `HOME` will be used as current working directory, and a warning will be issued. This may lead to unexpected results.

Let's have a brief look behind the scenes to understands what is happening while starting a parallel task, i.e. when distributing the processes forming the task within the cluster.

The locally started process (which is the process actually created when **mpiexec** is executed) connects to the local *ParaStation MPI* daemon psid(8), issuing a spawn request. This request, including the number of processes and other necessary information, is forwarded to the *master node*, which will provide a temporary node list. The local process will actually startup the remote processes using the psid on the particular node and will afterwards convert to an I/O handling process, the so called *ParaStation MPI* Logger process. The I/O handling mechanisms will be discussed in detail in Section 3.3, “Redirecting standard input and output”

Of course the distribution of the processes building the parallel task can be controlled in order to match the site's policy or tailor the process placement for a given algorithm. Because of the huge number of different types of applications running on a cluster *ParaStation MPI* provides far reaching features to control the distribution and spawning process. The manual pages for process_placement(7) and mpiexec(1) will discuss this in detail.

## 3.2. Placing processes onto nodes

The placement of processes for a parallel task within *ParaStation MPI* can be biased by various environment variables. Setting this variables is optional, if not set, the default behavior will make sure, that the usability of the cluster will not be jeopardized.

The user can predefine lists of nodes, can request exclusive use or overbooking of nodes, may change sorting criteria for node lists, or even may control process neighborhood on SMP nodes. Consequently, it is possible to precisely adjust the process distribution within the cluster to algorithms implemented and thus gain a maximum of performance out of the parallel task.

In addition, the administrator may define partitions reserved for users or group of users.

The manual page process_placement(7) will discuss all this parameters in detail.

## 3.3. Redirecting standard input and output

The redirection of standard input, output, and error output (file descriptor 0, 1 and 2) is done automatically without any further user interaction. In particular, the following actions are performed:

- The originally started process, which is the root process (first) of the parallel task, converts to a so called *ParaStation MPI* Logger process after spawning all child processes. It receives output from the spawned processes, or more precisely from the controlling Forwarder processes, and dispense it to its final destination. This might be the (virtual) terminal from which the root process was started or the file to which the output of the root process is redirected.
- The child processes are started by the *ParaStation MPI* daemon psid(8) on request of the root process of the parallel task via a so called *ParaStation MPI* Forwarder process. These forwarder processes control the spawned child process and handle all input/output produced. This output then is forwarded to the Logger process on the starting node which will send it to its final destination.

Input data is handled in a quite similar way. The *ParaStation MPI* Logger process reads the input from the (virtual) terminal or the file from which the input is redirected and sends it to a forwarder or a group of forwarders. The default action is to send it to the forwarder of the process with rank 0. This behavior might be modified as discussed below.

The redirection of input and output can be configured by a number of environment variables:

PSI_INPUTDEST

Forward all input to the process with rank *`rank`* or a list of processes with ranks *`ranklist`* defined by this variable.

The rank of a process is unique within a parallel task. The rank of a process can be determined via the PSE_getRank(3) library function. The rank of a process is identical to the MPI rank within the `MPI_COMM_WORLD` context which can be identified using the MPI_Comm_rank(3) library call.

The default is to forward all input to the process with rank 0.

PSI_SOURCEPRINTF

If this environment variable is defined, each fraction of output is prepended with a tag providing the rank of the process that was producing it. If the *ParaStation MPI* Logger is put into the verbose mode using the `PSI_LOGGERDEBUG` environment variable, also the length of the output fraction to print is shown.

PSI_NOMSGLOGGERDONE

Define this environment variable in order to suppress the message

```
PSIlogger: done
```

produced by the *ParaStation MPI* Logger process before exiting. This happens when all *ParaStation MPI* Forwarder have closed their connections to the logger.

PSI_LOGGERDEBUG

Turn on verbose mode for the *ParaStation MPI* Logger process. This will produce messages about connecting and detaching forwarders, received output, sent input and received signals.

This variable is intended for internal use only.

PSI_FORWARDERDEBUG

Turn on verbose mode for the *ParaStation MPI* Forwarder process. This will print information about received input, output and signals.

This variable is also intended for internal use only.

## 3.4. Spawning the environment

Another important task while spawning parallel applications in a cluster is to setup a proper environment for the newly created processes on each node.

*ParaStation MPI* by default exports only a limited set of environment variables to newly spawned processes, like `HOME`, `USER`, `SHELL` or `TERM`.

In addition, each currently defined environment variable can be exported to spawned processes by adding the variable name to the special *ParaStation MPI* variable `PSI_EXPORTS`. E.g., within a Bourne shell (or look alike), the commands

```
$ PSI_EXPORTS=Variable1,Variable2,...
$ export PSI_EXPORTS
```

will export the variables `Variable1` and `Variable2` to all processes. Accordingly, for a **csh** based environment, the command

```
$ setenv PSI_EXPORTS Variable1,Variable2,...
```

will export this variables to all subsequent parallel tasks.

Beside these variables, additional variables might be inherently set for remote processes by the inetd(8) , e.g. `PATH` or `HOSTNAME`.

Beside defining environment variables, **mpiexec** may be used to export variables. E.g.

```
$ mpiexec -E var content ...
```

may be used to propagate the variable *`var`* to all processes. Refer to mpiexec(1) for more details.

For a complete list of environment variables automatically exported by *ParaStation MPI*, refer to ps_environment(7).

## 3.5. Starting up serial tasks

*ParaStation MPI* is capable of starting serial tasks, therefore tasks not parallelized with MPI, within the cluster. All actions described in section Section 3.2, “Placing processes onto nodes”, Section 3.3, “Redirecting standard input and output” and Section 3.4, “Spawning the environment” are also considered for this type of tasks.

In order to start the serial program *`program`* somewhere within the cluster, simply execute

```
$ mpiexec -np 1 program [args]
```

where *`args`* are the arguments that shall be passed to *`program`*.

Depending on the settings of the environment variables as discussed in the previous sections, *`program`* will be started on a distinct node of the cluster. Parts of the current environment will eventually be passed to this node, too. Also the input and output is forwarded correctly to and from the remotely started process.

Starting serial processes using the *ParaStation MPI* management facilities offers a couple of advantages:

- Load balancing performed by *ParaStation MPI* towards serial processes results in a much better usage of the whole cluster in contrast of distributing the jobs manually.
- The user does not have to deal with the question, which node should be used for a particular job. *ParaStation MPI* takes care about deciding where (and when) to run the task.
- The possibility of starting serial jobs from the frontend machine without having to log on to the node that actually runs the jobs enables the system operator to disallow the users to log on the nodes in general. This enables much better control over the cluster and increases the security of the system.

## 3.6. Starting up administrative tasks

*ParaStation MPI* is also able to start up serial tasks, which are not counted within the process management. Those tasks are called administrative tasks. Thus, an arbitrarily number of tasks can be placed on a particular node, even if there are already parallel tasks spawned.

In order to start an administrative task on a node, execute

```
 # pssh -h nodename program args
```

This will run the command **program** on the node `nodename` providing it the arguments [`args`].

By default, only root is allowed to run this type of tasks, as this may burden unexpected load onto a compute node. To enable other users, there id must be added to the `adminuser` list or the user's group must be added to the `admingroup` list. For details, refer to psiadmin(8) and parastation.conf(5).

Currently, there is no way to change the user id except for root. There is no way to provide a password. The data is not encrypted while transfered across the network.

In addition to **pssh**, the **mpiexec** command may be used to run one or more administrative tasks in parallel. Use the option `--admin` to do so:

```
 # mpiexec -np 1 --admin --hosts nodename program args
```

The `nodename` may be a particular hostname or a (command separated) list of hosts.

## 3.7. Using the *ParaStation MPI* queuing facility

*ParaStation MPI* is able to queue task start request if the required resources are currently in use. This queuing facility is disabled by default. It can be enabled by each user independently. All requests of all users are held within one queue and managed on a first-come-first-serve strategy [1] .

If queuing is not enabled, start up requests, which cannot immediately be satisfied due to resource constrains, e.g. too few CPUs are currently available, will terminate giving the error message

```
 PSI: PSI_createPartition: Resource temporarily unavailable.
```

To enable the queuing facility, an environment variable called `PSI_WAIT` must be defined within the user's environment:

```
 # export PSI_WAIT=""
```

The actual value of `PSI_WAIT` is not considered, only the existence of this variable is checked.

[1] For more sophisticated queuing features, batch systems can be integrated with *ParaStation MPI*. For more details see *ParaStation MPI Administrator's Guide*.

For task startup and process distribution, more environment variables are taken into account, e.g. *`PSI_OVERBOOK`*. See also Section 3.2, “Placing processes onto nodes”, process_placement(7) and ps_environment(5) .

## 3.8. Suspending *ParaStation MPI* tasks

Parallel Tasks started by *ParaStation MPI* can be suspended by sending the system signal `SIGTSTP` to the *ParaStation MPI* Logger process. The signal will be forwarded to all processes of the parallel task and will by default stop the processes. To continue, the `SIGCONT` must be sent to the *ParaStation MPI* Logger process. This signal will also be forwarded to all processes of the task.

The application has to be prepared to handle interrupted system calls properly.

Depending on the transport protocol in use, tasks can be suspended only for a limited period time. If using TCP (HwType ethernet), connections may timeout and after sending the `SIGCONT` signal, the processes will receive I/O errors for this sockets. Using the *ParaStation MPI* protocol *p4sock* will solve this problem, as this protocol does not use any timeout features.

Suspending a task using the signal `SIGTSTP` will also trigger the *ParaStation MPI* queuing facility (see Section 3.7, “Using the *ParaStation MPI* queuing facility”). Depending of the global setting of `freeOnSuspend`, CPUs will be reused for newly spawned processes. Refer to parastation.conf(5).

# Chapter 4. Using MPI

This chapter explains how to compile and link MPI applications using the adapted version of MPIch2 coming with *ParaStation MPI*. Within this chapter it is assumed that this version of MPI is installed in the default location, i.e. `/opt/parastation/mpi2`. If this is not the case please refer to the section “Installing MPI” within the *ParaStation MPI Administrator's Guide* in order to setup things correctly.

Furthermore it is assumed that the directory `/opt/parastation/mpi2/bin` is set within the `PATH` environment variable. This directory holds all MPI2 related commands, like **mpicc**.

There are different versions of the *ParaStation MPI* environment available, depending on the hardware architecture and supported compilers. For x86, versions for GNU, Intel and Portland Group compilers are available. Likewise, for x86_64, versions for the GNU and Intel compiler are available. The libraries support all available languages and language options for the selected compiler, e.g. Fortran, Fortran90, C or C++. The different versions of the MPI library will be installed in parallel, thus it is possible to compile and run applications using different compilers at the same node. For information about the directory structure used by *ParaStation MPI*, refer to *ParaStation MPI Administrator's Guide* .

This chapter is not intended to be a tutorial on how to use the MPI message passing interface. It is assumed that the reader knows how to write MPI applications and how to compile programs in general on the platform *ParaStation MPI* is running on. If you don't know about MPI yet the MPI homepage is a good starting point to learn more about the Message Passing Interface.

## 4.1. Building MPI applications

Within this section it is assumed to have a working C source code stored in `prog.c`, which uses MPI to exchange messages between the different processes of the parallel task. In order to create an executable from this source code, all that has to be done is:

```
$ mpicc prog.c -o prog
```

This will create the executable `prog` from the source code. Notice that the header file `mpi.h` and the libraries needed to link the programs are found automatically. Furthermore you don't have to deal with the names of the libraries needed for linking. The actual MPI library and further libraries containing the low level communication calls are also linked automatically.

If your code is scattered to various source files, building is as easy as in the example above. First compile all the source files using

```
$ mpicc -c *.c
```

and afterwards link everything together:

```
$ mpicc *.o -o prog
```

Compiling of Fortran 77, Fortran 90 or C++ codes is very similar. Just replace the command **mpicc** by **mpif77**, **mpif90** and **mpiCC** respectively.

## 4.2. Starting MPI applications

In order to start a MPI application build as described in the last section, mpiexec(1) has to be used. E.g. if you want to start the executable **prog** with a maximum rank of 8, the command

```
$ mpiexec -np 8 prog [args]
```

has to be executed. The argument(s) *args* are passed to each instance of the executable **prog**. A detailed description of the mpiexec(1) command and its options can be found within the corresponding manual page.

All details discussed in Chapter 3, *Starting up programs* about spawning and input/output redirection also applies to MPI applications linked against *ParaStation MPI*. I.e. all the environment variables discussed in ps_environment(7) can be used to control the distribution of processes within the cluster.

## 4.3. Selecting Compiler Alternatives for MPI Programs

The choice of compiler will effect application performance, as will the optimization flags used during compilation. The *ParaStation MPI* environment has simplified the selection of compilers by dispensing with the normal MPI environment variables used in MPICH distributions.

It is important to link applications codes with MPI libraries and other ancillary libraries (such as MKL math libraries) compiled with the same compiler as that used for the main application. This ensures compatibility of shared objects and exposes the user to runtime libraries with the best set of optimization parameters for that particular compiler. *ParaStation MPI* achieves this requirement using wrapper scripts designed for use with each compiler.

### 4.3.1. Compiler & Linker Wrapper scripts

The following wrapper scripts are available in *ParaStation MPI* releases:

| Wrapper path | Compiler Type |
|---|---|
| `${MPI_HOME}/bin/mpicc` | **C** compiler |
| `${MPI_HOME}/bin/mpicxx` | **C++** compiler |
| `${MPI_HOME}/bin/mpif77` | **FORTRAN 77** compiler |
| `${MPI_HOME}/bin/mpif90` | **FORTRAN 90** compiler |

Table 4.1. Compiler wrappers for building MPI codes

With `${MPI_HOME}`:

| **`${MPI_HOME}`** | Compiler suite | commands |
|---|---|---|
| `/opt/parastation/mpi2` | GNU Compiler | **gcc**, **g++**, **gfortran** |
| `/opt/parastation/mpi2-intel` | Intel® Compiler Suite | **icc**, **icpc**, **ifort** |
| `/opt/parastation/mpi2-pgi` | PGI: Portland Group | **pgcc**, **pgCC**, **pgf77**, **pgf90** |

Table 4.2. MPI home directories for different compilers

Users will generally compile their application code using **make** and select the appropriate wrapper script in a **Makefile** as follows:

```
#
# Makefile
#

MPI_HOME="/opt/parastation/mpi2"

code: code.c
 $(MPI_HOME)/bin/mpicc -o code code.c
```

Users who wish to compile applications on the command line should add the directory containing the wrapper script of the selected compiler to their PATH. Use of command **which mpicc** will verify the PATH environment is correctly set (for the C compiler in this case). [1]

[1] Administrators should consider Environment Management Software for this (e.g. Modules: http://modules.sourceforge.net/)

# Chapter 5. Building native *ParaStation MPI* applications

This chapter describes briefly how to write native *ParaStation MPI* applications, i.e. applications that make use of the *ParaStation MPI* low level communication libraries and the *ParaStation MPI* management facilities without utilizing the standard MPI libraries.

In general it is not necessary to write native *ParaStation MPI* applications. Since the improvement in communication performance from circumventing the high level MPI library is negligible, in almost any scenario the utilization of MPI is the recommended way. Additionally MPI guarantees the portability of the programs to be developed.

Nevertheless the source code of a simple application comes with the standard *ParaStation MPI* software distribution. The file is named `native.c` and is located in the `/opt/parastation/doc/examples` directory. Furthermore a `Makefile` can be found in the same directory.

Taking a brief look at `native.c` it has to be noticed that the startup of the application and the necessary communication channels is quite complex. In the case of using MPI all this functionality is hidden within the simple MPI_Init(3) function. Thus the programmer does not have to take care about the details of the startup process. This is a further argument towards using MPI.

As already mentioned the usage of the *ParaStation MPI* API is only discussed briefly. A detailed description of the various function calls forming the API may be found in the API reference.

## 5.1. Using the management facility

The *ParaStation MPI* management system is used by native application in order to startup the processes of the parallel task and to assure the cleaning up by the *ParaStation MPI* daemons in the case of failure of one or more processes within the task.

In order to contact the local *ParaStation MPI* daemon and to initialize the *ParaStation MPI* library utilized to contact the *ParaStation MPI* management facility, PSE_initialize(3) has to be called. The rank of the actual process then is determined using the PSE_getRank(3) function.

Based on the rank the decision is made which function is inherited by the actual process. Three cases have to be distinguished:

rank = -1
: The process is the very first process started within the parallel task. Thus it is the root of all further processes. When the startup phase is finished this process will become a *ParaStation MPI* logger process and handle the standard I/O of the parallel task.

rank = 0
: This is the so called master process of the parallel task. It is spawned by the root process. Its function is firstly to spawn all further processes with rank > 0 and then to act as a normal compute process within the parallel task.

rank > 0
: These are normal compute processes started by the master process.

These three kinds of processes will be discussed within the next sections.

### 5.1.1. The root process

Both functions this process has to fulfill are handled almost completely within one function call. Thus nearly the entire functionality is hidden from the user's point of view.

But first PSE_setHWType(3) has to be called in order to specify the type of communication hardware the parallel task should utilize. This is a kind of pre-configuration of the following function call in order to spawn further processes.

All the rest that has to be done by this process is hidden within the PSE_spawnMaster(3) function: The master process is spawned and the process is converted into a *ParaStation MPI* logger process. This function usually never returns.

### 5.1.2. The master process

The first action the master process, as any other compute process, has to achieve is to register to its parent process via PSE_registerToParent(3). This is done in order to be noticed if the parent process exits unexpectedly. The usual behavior is to receive a SIGTERM signal in the case that this happens.

After it is registered to its parent process further processes, in this context called client processes, are going to be spawned. The target is to obtain the requested number of processes within the parallel task. This is done by calling the PSE_spawnTasks(3) function. At this point it has to be remarked that usually information has to be passed from the master process to its clients. This is due to enable the clients to connect back to the master in order to establish the connections used for communication. Thus the node and port number of the master process' communication interface has to be passed to the PSE_spawnTasks(3) function as well.

How to get an interface to the low level communication protocols provided by *ParaStation MPI* and to fetch information about this interface will be discussed within the next section.

### 5.1.3. Client processes

The client processes spawned by the master process have to register to their parent process, too. Thus they call PSE_registerToParent(3) as one of the first actions undertaken after the rank is determined.

For the further operation of the client processes it is necessary to get the information passed by the master process to them. This can be reached by using PSE_getMasterNode(3) and PSE_getMasterPort(3) respectively.

All further actions attempted by the master process on the one side and the client processes on the other side within the startup phase are concerning the establishment of the connections in order to do communication. These will be discussed within the next section.

After the startup phase all processes of the parallel task will reach a mode of normal operation, which usually will cover the actual application code. Within the application two kinds of exit mechanisms might wanted to be used. On the one hand an error detected within one process should result in the end of the whole parallel task. This can be reached by using the PSE_abort(3) function. On the other hand one of the processes might have finished all its tasks and want to exit without disturbing the other processes within the parallel task. In order to do so, PSE_finalize(3) has to be called. Afterwards the process might exit without shutting down all other processes.

## 5.2. Using the PSPort interface

In order to actually utilize the low level *ParaStation MPI* communication interfaces the **PSPort** library has to be used. The **PSPort** library supplies fast and reliable point to point connections to the user.

To make use of these connections first of all a port has to be opened. All further communication operations will act towards this port.

Furthermore information about all the ports of the other processes building the parallel task has to be gathered in order to be able to send data to this processes. Within the example program this problem is solved in the following way:

1. Initialize the **PSPort** interface using the PSP_Init(3) function within every process.

2. Open a port using the PSP_OpenPort(3) function call. The local port number and the *ParaStation MPI* node ID are determined via PSP_GetPortNo(3) and PSP_GetNodeID(3) respectively. This has to be done within every process, too.
3. The master process, i.e. the process with rank 0, which has spawned all other processes, is going to receive information concerning the local node ID and port number from the client processes. Therefore it has passed its own node ID and port number to the clients through the PSE_spawnTasks(3) function as discussed within the last section.
4. The client processes send the information concerning their local ports, i.e. port number and node ID, to the master process. Therefore they have to determine the node ID and port number of the master process before. This is done by using the PSE_getMasterNode(3) and PSE_getMasterPort(3) function calls, respectively, as discussed above.
5. When the master process has completed to receive the information from the client processes it redistributes the collected port and node numbers to all processes.

Now each process has information about the port number of all other processes. These information can then be used in order to do communication operations on a point-to-point basis between all the processes forming the parallel task.

The actual communication operations utilizing the low level **PSPort** interface are initiated using the PSP_ISend(3) and PSP_IReceive(3) functions. All communication within **PSPort** is nonblocking. Therefore each communication operation has to be tested upon completion. This is done via the PSP_Wait(3) function.

# Chapter 6. *ParaStation MPI* variables & application performance

## 6.1. Scope and audience

This chapter is intended for use by scientists and engineers who wish to explore the performance of their application codes using parameters that modify the behavior of the Massage-Passing Interface (MPI) and/or the placement of processes within a distributed memory cluster. This document details the parameters available to the user's environment that will impact application performance for the *ParaStation MPI* release of the Message-Passing Interface (MPI).

Any MPI and pscom variables found to be universally beneficial with respect to application performance have already been selected as defaults in the *ParaStation MPI* environment. The relative benefit of employing any of the parameters detailed herein is dependent upon specific application characteristics. In particular, the degree of inter-node message passing, message size and extent, frequency and type of collective operations used will all vary from one application to the next.

Prior to modifying any of the variables detailed in this chapter, the user is encouraged to make efforts to understand the communication characteristics of their code and how they relate to the physical hardware and interconnect topology.

## 6.2. *ParaStation MPI* run-time tuning parameters

This section describes those environment variables that can be used to modify the behavior of MPI at run-time. This section does not include those environment that influence the spawning of processes and the placement of jobs. Those details are included in Chapter 3, *Starting up programs* and the ps_environment(7) manual page.

### 6.2.1. General scaling tuning variables

A couple of variables are available to tune *ParaStation MPI* for large systems and jobs.

`PSP_TCP_BACKLOG`
: This variable defines the maximum backlog to the `listen()` call. For pscom versions below 5.0.34, this should be set to the number of cores within the system. For newer pscom versions, the default should be sufficient for all current systems.

  The actual value of `listen()` is also limited by the system to the value of `net.core.somaxconn`, which should also be set by the administrator to at least the number of cores within the system. See sysctl(8) for details.

`PMI_BARRIER_TMOUT` , `PMI_BARRIER_ROUND`
: These variables define the initial and total timeout for PMI barrier during job startup. See ps_environment(5) for details.

  The default values for both variables should be sufficient, even for very large systems. In case of appropriate warnings, they might be adjusted to circumvent network flaws.

### 6.2.2. InfiniBand specific tuning variables

Since reliable InfiniBand connections require a lot of memory for every connection, these communication path do not scale well with the number of connections. In that scenario, the memory footprint on each node is impacted significantly because every node (in the entire job) needs to store, in memory, N connection contexts, and allocate N sets of send/receive buffers. N is the total number of MPI ranks for that entire user job.

A better understanding of the memory footprint issues can be gained by using real life examples. Consider a system in which a single user is permitted to submit a single MPI job of 512 compute nodes - with each node having 24GB of available memory. If each of those compute nodes has 8 cores (2 quad core sockets) - then that would allow 4095 single threaded MPI tasks within a single user job.

By default, each InfiniBand connection in ParaStation MPI consumes 0.55 MB of memory for context information and send/receive queue buffering. As such, each core (or MPI rank) would need to reserve 4096 * 0.55MB of memory. That equates to 2GB of memory occupied just by the MPI layer per core. However, since there is only 3GB of memory available per core, this memory footprint is unacceptable. The next section details environment variables that can alleviate this problem.

### 6.2.3. MPI tuning for connected service types - addressing scalability problems with InfiniBand

*ParaStation MPI* uses by default the Reliable Connected (RC) InfiniBand service. *ParaStation MPI* also assumes that the majority of jobs include some all-to-all operations. Therefore all connections will be established inside MPI_Init() to be prepared for this type of communication. Unfortunately this has the scalability drawback described in the previous section.

However, informed users who have an understanding of their application's communication requirements can influence the connection behavior of the InfiniBand fabric to achieve scalability improvements. In particular, the memory footprint left by the implementation of connected service types can be significantly reduced using parameters detailed below.

`PSP_ONDEMAND=[0 | 1]`

**mpiexec -ondemand** is the equivalent command line argument that has the same effect as `PSP_ONDEMAND=1`.

Users who have examined their application code and have determined that it doesn't use all-to-all communication patterns can use **"on-demand"** connections. Using this setting, connections are only established when they are required by the application. If two ranks don't communicate with each other directly, then no connection will be generated. This mechanism will save host memory by not reserving QP buffers between ranks that never communicate but are nonetheless part of the same job.

If your application is known to use all-to-all communication patterns, the `PSP_ONDEMAND` variable will NOT reduce memory usage and should not be used!

Incorrect use of this flag can result in application failure due to out-of-memory errors. This may arise when late all-to-all communication QPs are established after the application itself has consumed much of the host memory. Users are therefore advised to use this environment variable with caution.

If your applications use just the following communication patterns:

- Nearest neighbor communication
- Scatter/Gather and All-Reduce, Alltoall collectives based on binary trees

It is appropriate to set the `PSP_ONDEMAND=1` provided the communication patterns used in a users MPI code are limited to the above types.

In applications where all-to-all communication is required, there are still some tuning parameters available to improve scaling behavior. Again, these parameters focus on the reduction of the memory footprint caused by excessive numbers of private send/receive message buffers.

`PSP_OPENIB_SENDQ_SIZE=`*`num`* `/ PSP_OPENIB_RECVQ_SIZE=`*`num`*

The default buffer settings for the each send/receive queue (for which there is 1 per connection) is to allocate sixteen 16kb buffers. The size of the buffer itself is not tunable (set at 16kb), but the number of buffers per connection (for the send and receive queues respectively) is a tunable parameter.

Examples of setting the number of the send/receive buffers per connection are shown below:

```
PSP_OPENIB_SENDQ_SIZE=3
```

```
PSP_OPENIB_RECVQ_SIZE=3
```

Using the above parameters, the size of the send/rec queue buffers are reduced from 0.55 MB/connection (when both values are set to the default of 16) to 0.14 MB/connection. Reducing either of the above parameters to a value of less that 3 will cause deadlocks.

Users should also be aware that reducing the number of buffers will effect the message issue rate and overall throughput of messages within their application. Users are encouraged to test various values of send/receive size with a view to determining, by experiment, which values produce the best all-round performance.

# Reference Pages

# mpiexec

mpiexec — run a *ParaStation MPI* program

## Synopsis

**mpiexec** { --np *num* | -np *num* | -n *num* } [ -e | --exports=*envlist* ] [ -x | --envall ] [ -E | --env=*name value* ] [ -b | --bnr ] [ -a | --jobalias=*alias* ] [ options ] program [arg]...

**mpiexec** { -A | --admin } [ -L | --login=*name* ] {[ -N | --nodes=*nodelist* ] | [ -H | --hosts=*hostlist* ] | [ -f | --hostfile=*file* ] | [ --machinefile=*file* ]} program [arg]...

**mpiexec** [ -V | --version ] [ -? | --help ] [ --usage ] [ --extendedhelp ] [ --extendedusage ] [ --debughelp ] [ --debugusage ] [ --commhelp ] [ --commusage ]

## Description

The **mpiexec** command is the typical way to start parallel or serial jobs. It hides the differences of starting jobs of various implementations of the Message Passing Interface, version 2, from the user. Within the *ParaStation MPI* implementation of this command, the startup of parallel jobs is handled as described by the process_placement(7) manual page. The process spawning may also be steered by environment variables which are described in detail within ps_environment(7).

This version of **mpiexec** supports the `Process Manager Interface (PMI)` protocol. Therefore, this version of **mpiexec** also supports many other implementations of MPI2, like MPICH2, MVAPICH2 or Intel MPI.

The command **mpiexec** is typically used like

```
mpiexec -np num prog [args]
```

This will start up the program **prog** *num* times in parallel forming a parallel job. *Args* are optional arguments which will be passed to each task. **Prog** is not necessarily required to use MPI calls to transfer data.

To run a serial job, aka a job consisting only of a single task, use a task count of `1`, e.g.

```
mpiexec -np 1 prog [args]
```

## Options

### General options

`-n` *num* , `-np` *num* , `--np` *num* , `--np=`*num*
: Specify the number of processes to start.

`-e` , `--exports=`*envlist*
: Name or comma-separated list of environment variable(s) exported to all processes.

`-x` , `--envall`
: Export all environment variables to all processes.

`-E` , `--env` *name value*
: Export the variable *name* with the content *value*.

`-b` , `--bnr`
: Enable *ParaStation4* compatibility mode.

`-a` , `--jobalias=`*name*
: Assign an alias to the job. This name is currently only used for accounting purposes.

## Options controlling process placement

`-N , --nodes=`*`list`*
: Comma- or space-separated list of nodes IDs to use, e.g. *`3-5,7,11-17`*.

`-H , --hosts=`*`list`*
: Comma- or space-separated list of hosts to use, e.g. *`host1,host2`*.

`-f , --hostfile=`*`filename`*
: Hostfile to use.

`--machinefile=`*`filename`*
: Machinefile to use. Equal to `--hostfile`.

`-o , --overbook`
: Allow overbooking.

`-f , --loopnodesfirst`
: Place consecutive processes on different nodes, if possible. If not set, as many as possible consecutive processes will be placed within a SMP node to allow local communication using shared memory.

`-E , --exclusive`
: Do not allow any other processes on used nodes.

`-s , --sort=`*`criteria`*
: Sorting criteria to be used when selecting nodes: *`proc`*, *`load`*, *`proc+load`* or *`none`*.

`-d , --wdir=`*`dir`*
: Working directory to start the processes.

`-P , --path=`*`pathlist`*
: Set the PATH list.

## Communication options

`-c , --discom=`*`connection`*
: Disable a particular communication architecture: *`SHM`*, *`TCP`*, *`P4S`*, *`GM`*, *`MVAPI`*, *`OPENIB`* or *`ELAN`*. *`Connection`* may also be a comma- or space-separated list to disable multiple architectures at once. This option is typically used for testing and debugging purposes only.

`-t , --network=`*`string`*
: Space-separated list of networks used for MPI communication.

`-y , --schedyield`
: Use sched yield system call.

`-r , --retry=`*`num`*
: Number of connection retries.

## I/O forwarding options

`-s , --inputdest=`*`ranklist`*
: Define the rank the standard input is forwarded to. *`Rank`* may also be a comma-separated list of ranks, or *`all`* to send the input to all processes. For example, the option `--inputdest=`*`0-3,5,8-15`* will forward standard input to all but ranks 4, 6 and 7.

`-l , --sourceprintf`
: Print output-source info.

`-T , --timestamp`
: Print time stamps.

`-R , --rusage`
: Print elapsed sys/user time.

`-m , --merge`
: Merge identical output lines from all processes.

### Privileged options

`-A` , `--admin`
: Start processes as administrative tasks. These tasks are not counted within the ParaStation resource management and may be run in parallel to compute tasks. Only privileged users are allowed to run administrative tasks.

`-L` , `--login=`*`name`*
: Remote user used to execute command. For administrative tasks only.

### Other options

`--gdb`
: Run processes under control of **gdb**.

`-u` , `--usize=`*`size`*
: Set the universe size.

`--extendedhelp`
: Print extended help.

`--debughelp`
: Print debug help.

`--comhelp`
: Print communication debug help.

### Extended options

`--plugindir=`*`dir`*
: Directory to search plugins. TODO

`--sndbuf=`*`size`*
: Define the TCP sendbuffer size.

`--rcvbuf=`*`size`*
: Define the TCP receivebuffer size.

`--delay`
: Don't use the NODELAY option for TCP sockets.

`--mergedepth=`*`num`*
: Number of lines to search for identical output. Defaults to 300.

`--mergetimeout=`*`secs`*
: Timeout in seconds to wait for identical output. Defaults to 2secs.

`--pmitimeout=`*`secs`*
: Timeout in seconds for all clients to join the first PMI barrier. Use *`-1`* to disable. Defaults to *`60sec + np * 0.5µsec`*.

`--pmiovertcp`
: Connect to the PMI client using a TCP/IP socket.

`--pmioverunix`
: Connect to the PMI client using a UNIX domain socket (default).

`--pmidisable`
: Disable PMI interface.

### Debugging options

`--pscomdb`
: Enable `libpscom` debugging.

`--psidb`
: Enable **psid** debugging.

`--pmidb`
: Enable `PMI` debugging.

`--pmidbclient`
: Enable `PMI` client debugging.

`--pmidbkvs`
: Enable `PMI` key-value-space debugging.

`--show`
: Show command for remote execution but don't run it.

`--loggerrawmode`
: Set raw mode of the logger.

`--sigquit`
: Output debug information on signal SIGQUIT.

### Compatibility options

`-bnr`
: Enable *ParaStation4* compatibility mode. Same as `--bnr`.

`-machinefile=`*`filename`*
: Machinefile to use. Same as `--hostfile`.

`-1`
: Override default of trying first (ignored).

`-ifhn=`*`string`*
: Space-separated list of networks enabled.

`-file=`*`string`*
: File with additional information (ignored).

`-tv=`*`string`*
: Run procs under totalview (ignored).

`-tvsu=`*`string`*
: Totalview startup only (ignored).

`-gdb`
: Run processes under control of gdb. Same as `--gdb`.

`-gdba`
: Attach to debug processes with gdb (ignored).

`-ecfn`
: Output xml exit codes filename (ignored).

`-dir=`*`dir`* , `-wdir=`*`dir`*
: Working directory to start the processes. Same as `--wdir`.

`-umask=`*`mask`*
: Umask for remote process (ignored).

`-path=`*`pathlist`*
: Set the PATH list.

`-host=`*`host`*
: Host to start on (ignored).

`-soft=`*`num`*
: Giving hints instead of a precise number for the number of processes (ignored).

`-arch=`*`arch`*
: Arch type to start on (ignored).

`-envall`
: Export all environment variables to all processes. Same as `--envall`.

`-envnone`
: Export no environment variables to all processes.

`-envlist=`*`envlist`*
: Name or comma-separated list of environment variable(s) exported to all processes. Same as `--exports`.

`-env=`*`name value`*
: Export the variable *`name`* with the content *`value`*. Same as `--env`.

`-usize=`*`size`*
: Set the universe size. Same as `--usize`.

### Global compatibility options

`-gnp` *`num`* , `-gn` *`num`*
: Specify the number of processes to start. Same as `-np` or `-n`.

`-gdir=`*`dir`* , `-gwdir=`*`dir`*
: Working directory to start the processes. Same as `--wdir`.

`-gumask=`*`mask`*
: Umask for remote process (ignored).

`-gpath=`*`pathlist`*
: Set the PATH list. Same as `-path`.

`-ghost=`*`host`*
: Host to start on (ignored).

`-gsoft=`*`num`*
: Giving hints instead of a precise number for the number of processes (ignored).

`-garch=`*`arch`*
: Arch type to start on (ignored).

`-genvall`
: Export all environment variables to all processes. Same as `--envall`.

`-genvnone`
: Export no environment variables to all processes. Same as `-envnone`.

`-genvlist=`*`envlist`*
: Name or comma-separated list of environment variable(s) exported to all processes. Same as `--exports`.

`-genv=`*`name value`*
: Export the variable *`name`* with the content *`value`*. Same as `--env`.

### Miscellaneous options

`-V --version`
: Show version and exit.

`-? --help`
: Show help message and exit.

`--usage`
: Show brief usage message and exit.

`--extendedusage`
: Show brief extended usage message and exit.

`--debugusage`
  Show brief debug usage message and exit.

## Extended description

The **mpiexec** command may be controlled in many ways. To do so, a lot of specialized options are implemented. This sections describes some of them and the underlying concepts.

### Option compatibility

The *ParaStation MPI* **mpiexec** command supports many options also found in other implementations, especially the MPICH2 version, to ensure compatibility on a command line level. For many of the long options, indicated by two dashes (`--`), versions with only one dash are implemented, e.g. `-envall` is equivalent to `--envall`.

The colon syntax (`:`) for defining local options, used for `MPI_COMM_SPAWN_MULTIPLE` calls, is currently not supported. In addition, configuration files are not supported (option `-configfile`).

### Signal handling

The **mpiexec** command will forward all possible signals to all tasks of the actual job. In particular, the signal `SIGTSTP` will be sent to all controlled processes and will therefore suspend the entire job immediately. See *ParaStation MPI User's Guide* for details about suspending jobs.

### Process Manager Interface (PMI)

This version of **mpiexec** supports the simple Process Manager Interface, version 1.1. Thus, it is able to startup and control any PMI compatible MPI application. The PMI protocol is the default interface for a **mpd** daemon. Supporting this protocol, the *ParaStation MPI* daemon psid(8) now is a complete replacement for **mpd**.

The PMI implementation is completely transparent to the user. *ParaStation MPI* will use this by default.

If at least one task uses the PMI protocol, the proper startup of all tasks is ensured by a global PMI barrier. This barrier is monitored by a timeout (see option `--pmitimeout`). If at least one task of the application fails to connect to the PMI within this timeout, the entire application will be terminated.

### Machine file format

Using the option `--hostfile` or `--machinefile`, the **mpiexec** command will read the list of nodes to be used from the specified file. One node name per line may be listed. Separated by a white space, the option `ifhn=`*`subnet`* may be added to each node defining the desired subnet for the MPI traffic for this particular node. *`Subnet`* may be a subnet or the IP address of a node's interface.

In case this subnet is not available or suitable, the communication subsystem may use a different subnet or interface for MPI communication.

### Merging output

To improve the readability of the output of parallel applications, identical output lines may be merged and printed only once using the `--merge` option. Output lines have to be delimited by a `\n`.

If the line merging option is enabled, the logger process buffers all lines output to stdout and stderr by each task to compare it against each other and therefore to identify identical lines. If an identical line for all tasks is found, it is written to stdout or stderr, respectively. The search depth may be set using the `--mergedepth` option.

Each output line is internally marked with a timestamp and monitored by a timeout. After this timeout, all identical lines read up to now are combined and written to stdout or stderr. The line order will be preserved. The timeout will improve the actual feedback to the user, even if one or more tasks will delay the output for a long period of time. The timeout may be set using the option `--mergetimeout`.

When merging output, each line written to stdout or stderr is prepended with a list of ranks enclosed in brackets at the beginning of the line, like *`[0-13,15]`*. With this example, an identical line was read from all but rank 14, assuming this is a job running on 16 CPUs. The output of rank 14 may be listed after a certain period of time, prepended with *`[14]`* or may be missing completely, this is up to the application.

While merging output lines, no output is ever suppressed! All lines will be output.

In general, output merging should not be used when saving binary output data. Especially, the characters `\n` and `\0` are swallowed.

## Spawning administrative tasks

In addition to parallel or serial compute jobs, the **mpiexec** command may also be used to spawn administrative tasks. This kind of tasks are not counted within the *ParaStation MPI* resource management and therefore may be run on nodes already in use.

To run an administrative task, use the option `--admin`. This will also enable the output merging capabilities, see option `--merge`. For example, the command

```
mpiexec --admin --hosts=node1,node2,node3,node4 date
```

will run the **date** command on the nodes 1 to 4. You have to supply a list of nodes using the `--hosts`, `--nodes` or `--hostfile` option. The number of processes will be automatically determined by the number of nodes.

Only members of the `adminuser` or `admingroup` list are allowed to run administrative tasks. Root is typically a member of the `adminuser` list.

## Debugging parallel applications

Parallel and serial jobs may be run under the control of **gdb** using the command line argument `--gdb`. By default, the standard input is redirected to each process (see option `--inputdest=all`) and the output of all processes is merged (`--merge`) to improve readability. The initial PMI timeout is also disabled (`--pmitimeout`). Each task is controlled by it's own **gdb** instance. All **gdb** instances will be controlled in parallel.

As with gdb, to actually run an application the **run** must be issued. All optional arguments from the **mpiexec** command line will be used by **gdb**. To supply arguments to the parallel tasks, use the gdb command **run *`args`*** or the **gdb** option *`-args`*.

To re-route the standard input to a particular task or a set of tasks, use the pseudo command **[*`rank`*]**, where *`rank`* is a single rank or a comma-separated list of ranks. Again, to redirect the input to all tasks, use the command **[`all`]**.

The **gdb** prompt will be automatically set to "`(gdb)\n`". The newline is required by the merging option. Therefore, the input will be read from the beginning of a new line.

Due to security reasons, terminated processes may not be restarted. Restart the entire job instead.

Signals are handled like expected: all catchable signals are forwarded by the logger task to all foreground processes controlled by the local forwarder(s). Within **gdb**, the resulting action of those forwarded signals depend on the actual signal handling within gdb.

### Running *ParaStation4* applications

The **mpiexec** command may also be used to run applications linked with the former version of *ParaStation MPI*. To do so, use the `--bnr` option. This will enable the startup mechanism used by *ParaStation4* and former versions.

## Errors

No known errors.

## See also

psmstart(1), ps_environment(7), process_placement(7), psid(8) and *ParaStation MPI User's Guide*.

# mpirun_chp4

mpirun_chp4 — run a MPIch/P4 MPI program on a *ParaStation MPI* cluster

## Synopsis

**mpirun_chp4** [-?Vv] -np *nodes* [[-nodes *nodelist*] | [-hosts *hostlist*] | [-hostfile *hostfile*]] [-sort {[proc] | [load] | [proc+load] | [none]} ] [-all-local] [-inputdest *dest*] [-sourceprintf] [-rusage] [-exports *envlist*] [-keep_pg] [-leave_pg] [--usage] program [arg]...

## Description

**mpirun_chp4** is a tool that enables MPIch/P4 programs to run on a *ParaStation MPI* cluster under control of the *ParaStation MPI* management facility. Within *ParaStation MPI* the startup of parallel jobs is handled as described within the process_placement(7) manual page. The spawning mechanism is either steered by environment variables, which are described in detail within ps_environment(7), or via options to the **mpirun_chp4** command. In fact these do nothing but setting the corresponding environment variables.

**mpirun_chp4** typically works like this:

```
  mpirun_chp4 -np num prog [args]
```

This will startup the parallel MPIch/P4 program **prog** on *num* nodes of the cluster. *args* are optional argument which will be passed to each instance of **prog**.

## Options

`-np` *nodes*
: Specify the number of processors to run on.

`-nodes` *nodelist*
: Define the nodes which will build the partition of the *ParaStation MPI* cluster used in order to spawn new processes.

  *nodelist* is a single character string containing a comma separated list of *ParaStation MPI* IDs. Depending on the existence of the environment variable `PSI_NODES_SORT` and the presence of the `-sort` option, the order of the nodes within *nodelist* might be relevant.

  If the number of spawned processes exceeds the number of nodes within the partition, some nodes may get more than one process.

  If any of the environment variables `PSI_NODES`, `PSI_HOSTS` or `PSI_HOSTFILE` is set, this option must not be given.

`-hosts` *hostlist*
: Define the nodes which will build the partition of the *ParaStation MPI* cluster used in order to spawn new processes.

  *hostlist* is a single character string containing a space separated list of hostnames. These have to be resolvable in order to get the corresponding *ParaStation MPI* IDs. Depending on the existence of the environment variable `PSI_NODES_SORT` and the presence of the `-sort` option, the order of the nodes within *hostlist* might be relevant.

  If the number of spawned processes exceeds the number of nodes within the partition, some nodes may get more than one process.

  If any of the environment variables `PSI_NODES`, `PSI_HOSTS` or `PSI_HOSTFILE` is set, this option must not be given.

`-hostfile` *`hostfile`*

Define the nodes which will build the partition of the *ParaStation MPI* cluster used in order to spawn new processes.

*`hostfile`* is the name of a file containing a list of hostnames. These have to be resolvable in order to get the corresponding *ParaStation MPI* IDs. The format of the file is one hostname per line. Depending on the existence of the environment variable `PSI_NODES_SORT` and the presence of the `-sort` option, the ordering of the nodes within the *`hostfile`* might be relevant.

If the number of spawned processes exceeds the number of nodes within the partition, some nodes may get more than one process.

If any of the environment variables `PSI_NODES`, `PSI_HOSTS` or `PSI_HOSTFILE` is set, this option must not be given.

`-sort` *`mode`*

Steer the sorting criterion which is used in order to bring the nodes within a partition in an appropriate order. This order will be used to spawn remote processes. The following values of *`mode`* are recognized:

proc

The nodes are sorted by the number of running *ParaStation MPI* processes before new processes are spawned. This is the default behavior.

load

The nodes are sorted by load before new processes are spawned. Therefore nodes with the least load are used first.

To be more specific, the load average over the last minute is used as the sorting criterion.

proc+load

The nodes are sorted corresponding to the sum of the 1 minute load and the number of running *ParaStation MPI* processes. This will lead to fair load-balancing even if processes are started without notification to the *ParaStation MPI* management facility.

none

No sorting of nodes before new processes are spawned. The nodes are used in a round robin fashion as they are set in the `PSI_NODES`, `PSI_HOSTS` or `PSI_HOSTFILE` environment variables or via the corresponding `-nodes`, `-hosts` or `-hostfile` options.

If the environment variables `PSI_NODES_SORT` is set, this option must not be given.

`-all-local`

Run all processes on the master node.

Keep in mind that the masternode is not necessarily the local machine but, depending on the *ParaStation MPI* configuration and the options and environment variables given, may be any machine within the *ParaStation MPI* cluster. Nevertheless all processes building the parallel MPIch/P4 task will run on the same node of the *ParaStation MPI* cluster.

`-inputdest` *`dest`*

Define the process which receives any input to the parallel task. *`dest`* is an integer number in the range from 0 to *`nodes`*-1, where *`nodes`* is set by the `-np` option.

The default is to send the input to the process with rank 0 within the parallel task.

`-sourceprintf`

If this option is enabled, the logger will give information about the source of the output produced, i.e. “[id]:” will be prepended to any line of output, where id is the rank of the printing process within the parallel task.

Usually the id coincides with the MPI-rank.

`-rusage`
: When this option is given, the logger will print a notice about the user and system time consumed by each process within the parallel task upon exit of this process.

`-exports` *`envlist`*
: Register a list of environment variables which should be exported to remote processes during spawning. Some environment variables (`HOME`, `USER`, `SHELL` and `TERM`) are exported by default.

  Furthermore `PWD` is set correctly for remote processes.

  *`envlist`* is a single character string containing a comma separated list of environment variables. Only the name of the environment variable has to be given.

  If the environment variable `PSI_EXPORTS` is set, *`envlist`* will be appended to this variable.

`-keep_pg` , `-leave_pg`
: Don't remove the process group file which steers the startup of the parallel MPIch/P4 task.

  This file will be constructed on the fly during startup by the **mpirun_chp4** program. The content of this file will depend on the configuration of the *ParaStation MPI* cluster, the options given to the **mpirun_chp4** and the environment variables set.

`-V` , `--version`
: Output version information and exit.

`-v` , `--verbose`
: Verbose execution with many message during startup of the parallel task.

`-?` , `--help`
: Show a help message.

`--usage`
: Display a brief usage message.

## Examples

In order to start the parallel MPIch/P4 program `prog1` on any 5 nodes within the *ParaStation MPI* cluster, execute:

```
mpirun_chp4 -np 5 prog1 -v
```

The option `-v` will be passed to any instance of `prog1` spawned.

If the parallel task should run on the nodes 5-9 of the cluster,

```
mpirun_chp4 -np 5 -nodes "5,6,7,8,9" prog1
```

has to be executed.

If the nodes should be sorted by load, use:

```
mpirun_chp4 -np 5 -nodes "5,6,7,8,9" -sort load prog1
```

In order to acquire information about the user and system time used by the spawned processes on the different nodes run:

```
mpirun_chp4 -np 5 -rusage prog1
```

## Errors

No known errors.

## See also

psmstart(1), ps_environment(7), process_placement(7)

# mpirun_chgm

mpirun_chgm — run a MPIch/GM MPI program on a *ParaStation MPI* cluster

## Synopsis

**mpirun_chgm** [-?Vv] -np *nodes* [[-nodes *nodelist*] | [-hosts *hostlist*] | [-hostfile *hostfile*]] [-sort {[proc] | [load] | [proc+load] | [none]} ] [-all-local] [-inputdest *dest*] [-sourceprintf] [-rusage] [-exports *envlist*] [[--gm-no-shmem] | [--gm-numa-shmem]] [--gm-wait=*sec*] [--gm-kill=*sec*] [--gm-eager=*size*] [--gm-recv=*mode*] [--usage] program [arg]...

## Description

**mpirun_chgm** is a tool that enables MPIch/GM programs to run on a *ParaStation MPI* cluster under control of the *ParaStation MPI* management facility. Within *ParaStation MPI* the startup of parallel jobs is handled as described within the process_placement(7) manual page. The spawning mechanism is either steered by environment variables, which are described in detail within ps_environment(7), or via options to the **mpirun_chgm** command. In fact these do nothing but setting the corresponding environment variables.

**mpirun_chgm** typically works like this:

```
mpirun_chgm -np num prog [args]
```

This will startup the parallel MPIch/GM program **prog** on *num* nodes of the cluster. *args* are optional argument which will be passed to each instance of **prog**.

## Options

`-np` *nodes*
: Specify the number of processors to run on.

`-nodes` *nodelist*
: Define the nodes which will build the partition of the *ParaStation MPI* cluster used in order to spawn new processes.

  *nodelist* is a single character string containing a comma separated list of *ParaStation MPI* IDs. Depending on the existence of the environment variable `PSI_NODES_SORT` and the presence of the `-sort` option, the order of the nodes within *nodelist* might be relevant.

  If the number of spawned processes exceeds the number of nodes within the partition, some nodes may get more than one process.

  If any of the environment variables `PSI_NODES`, `PSI_HOSTS` or `PSI_HOSTFILE` is set, this option must not be given.

`-hosts` *hostlist*
: Define the nodes which will build the partition of the *ParaStation MPI* cluster used in order to spawn new processes.

  *hostlist* is a single character string containing a space separated list of hostnames. These have to be resolvable in order to get the corresponding *ParaStation MPI* IDs. Depending on the existence of the environment variable `PSI_NODES_SORT` and the presence of the `-sort` option, the order of the nodes within *hostlist* might be relevant.

  If the number of spawned processes exceeds the number of nodes within the partition, some nodes may get more than one process.

  If any of the environment variables `PSI_NODES`, `PSI_HOSTS` or `PSI_HOSTFILE` is set, this option must not be given.

`-hostfile` *`hostfile`*
: Define the nodes which will build the partition of the *ParaStation MPI* cluster used in order to spawn new processes.

  *`hostfile`* is the name of a file containing a list of hostnames. These have to be resolvable in order to get the corresponding *ParaStation MPI* IDs. The format of the file is one hostname per line. Depending on the existence of the environment variable `PSI_NODES_SORT` and the presence of the `-sort` option, the ordering of the nodes within the *`hostfile`* might be relevant.

  If the number of spawned processes exceeds the number of nodes within the partition, some nodes may get more than one process.

  If any of the environment variables `PSI_NODES`, `PSI_HOSTS` or `PSI_HOSTFILE` is set, this option must not be given.

`-sort` *`mode`*
: Steer the sorting criterion which is used in order to bring the nodes within a partition in an appropriate order. This order will be used to spawn remote processes. The following values of *`mode`* are recognized:

  proc
  : The nodes are sorted by the number of running *ParaStation MPI* processes before new processes are spawned. This is the default behavior.

  load
  : The nodes are sorted by load before new processes are spawned. Therefore nodes with the least load are used first.

    To be more specific, the load average over the last minute is used as the sorting criterion.

  proc+load
  : The nodes are sorted corresponding to the sum of the 1 minute load and the number of running *ParaStation MPI* processes. This will lead to fair load-balancing even if processes are started without notification to the *ParaStation MPI* management facility.

  none
  : No sorting of nodes before new processes are spawned. The nodes are used in a round robin fashion as they are set in the `PSI_NODES`, `PSI_HOSTS` or `PSI_HOSTFILE` environment variables or via the corresponding `-nodes`, `-hosts` or `-hostfile` options.

  If the environment variables `PSI_NODES_SORT` is set, this option must not be given.

`-all-local`
: Run all processes on the master node.

  Keep in mind that the masternode is not necessarily the local machine but, depending on the *ParaStation MPI* configuration and the options and environment variables given, may be any machine within the *ParaStation MPI* cluster. Nevertheless all processes building the parallel MPIch/GM task will run on the same node of the *ParaStation MPI* cluster.

`-inputdest` *`dest`*
: Define the process which receives any input to the parallel task. *`dest`* is an integer number in the range from 0 to *`nodes`*-1, where *`nodes`* is set by the `-np` option.

  The default is to send the input to the process with rank 0 within the parallel task.

`-sourceprintf`
: If this option is enabled, the logger will give information about the source of the output produced, i.e. “[id]:” will be prepended to any line of output, where id is the rank of the printing process within the parallel task.

  Usually the id coincides with the MPI-rank.

`-rusage`
: When this option is given, the logger will print a notice about the user and system time consumed by each process within the parallel task upon exit of this process.

`-exports` *`envlist`*
: Register a list of environment variables which should be exported to remote processes during spawning. Some environment variables (`HOME`, `USER`, `SHELL` and `TERM`) are exported by default.

  Furthermore `PWD` is set correctly for remote processes.

  *`envlist`* is a single character string containing a comma separated list of environment variables. Only the name of the environment variable has to be given.

  If the environment variable `PSI_EXPORTS` is set, *`envlist`* will be appended to this variable.

`--gm-no-shmem`
: Disable the shared memory support (enabled by default).

`--gm-numa-shmem`
: Enable shared memory only for processes sharing the same Myrinet interface.

`--gm-wait=`*`sec`*
: Wait *`sec`* seconds between each spawning step.

  Usually the spawning of the client processes is done as fast as possible. Using this option a delay of *`sec`* seconds is introduced in between each spawning step.

`--gm-kill=`*`sec`*
: Kill all processes *`sec`* seconds after the first one exits.

  Usually the termination of a job is handled as follows: As long as a client process terminates normally, i.e. no signal was delivered to the process and the return value is 0, all other processes of the parallel task are not affected in any way. In any other case, i.e. if a client process exits with return value different from 0 or as the effect of a signal delivered to the process, all other processes will be killed immediately.

  Using this option changes the behavior of parallel tasks with processes terminating normally. Now all other processes will be killed *`sec`* seconds after the first process has terminated.

  Keep in mind that the implementation of MPI_finalize() within MPIch/GM blocks. This means no process of the parallel task will return from the MPI_finalize() before all processes have called this function. Thus this option only makes sense if a process is able to exit execution without calling MPI_finalize(). Following the MPI standard it is not recommended to do so.

`--gm-eager=`*`size`*
: Specifies the eager/rendezvous protocol threshold size.

  As within any state of the art communication library MPIch/GM implements two different protocols used depending on the message size being send. This option enables the possibility to modify the threshold size that determines which protocol to use.

  *`size`* is given in bytes.

`--gm-recv=`*`mode`*
: Specifies the receive mode of the GM communication library. Possible values for *`mode`* are `polling`, `blocking` or `hybrid`. The default is `polling`.

  For a detailed description of the different receive modes please refer to the GM documentation.

`-V , --version`
: Output version information and exit.

`-v , --verbose`
: Verbose execution with many message during startup of the parallel task.

`-?` , `--help`
: Show a help message.

`--usage`
: Display a brief usage message.

## Examples

In order to start the parallel MPIch/GM program `prog1` on any 5 nodes within the *ParaStation MPI* cluster, execute:

```
mpirun_chgm -np 5 prog1 -v
```

The option `-v` will be passed to any instance of `prog1` spawned.

If the parallel task should run on the nodes 5-9 of the cluster,

```
mpirun_chgm -np 5 -nodes "5,6,7,8,9" prog1
```

has to be executed.

If the nodes should be sorted by load, use:

```
mpirun_chgm -np 5 -nodes "5,6,7,8,9" -sort load prog1
```

In order to acquire information about the user and system time used by the spawned processes on the different nodes run:

```
mpirun_chgm -np 5 -rusage prog1
```

## Errors

No known errors.

## See also

psmstart(1), ps_environment(7), process_placement(7)

## mpirun_elan

mpirun_elan — run a QsNet MPI program on a *ParaStation MPI* cluster.

### Synopsis

**mpirun_elan** -np *num* [[--nodes=*nodelist*] | [--hosts=*hostlist*] | [--hostfile=*hostfile*]] [-sort={ proc | load | proc+load | none } ] [-inputdest=*dest*] [-sourceprintf] [-rusage] [[-e] | [--exports=*envlist*]] [[-k] | [--keep]] [-show] [ -v | --verbose ] [ -V | --version ] [ -? | --help ] [--usage] *command* [ *args* ]

### Description

**mpirun_elan** is a tool that enables programs linked with the Quadrics QsNet MPI library to run on a *ParaStation MPI* cluster under control of the *ParaStation MPI* process management facility. Within *ParaStation MPI* the startup of parallel jobs is handled as described within the process_placement(7) manual page. The spawning mechanism is steered by environment variables, which are described in detail within ps_environment(7).

**mpirun_elan** typically works like this:

```
  mpirun_elan -np num prog [args]
```

This will startup the program **prog** *num* times in parallel forming a parallel job. *Args* are optional argument which will be passed to each instance of **prog**.

### Options

`-np num`
: Number of processes to create.

`-n, --nodes=nodeslist`
: List of node numbers to use.

`-h, --hosts=hosts`
: List of hosts to use.

`--hostfile=hostfile`
: Hostfile to use.

`-sort=criteria`
: Sorting criteria to use: *proc*, *load*, *proc+load* or *none*.

`-sourceprintf`
: Print output-source info.

`-rusage`
: Print resource usage when finished.

`-e, -exports=envlist`
: Environment to export to foreign nodes.

`-k, --keep`
: don't remove mpihosts file upon exit.

`-v, --verbose`
: Print diagnostic messages.

`-V, --version`
: Output version information and exit.

`-?, --help`
: Show this help messages.

`--usage`
: Display brief usage message.

## See also

psmstart(1), ps_environment(7), process_placement(7)

# mpirun-ipath-ps

mpirun-ipath-ps — run a InfiniPath™ MPI program on a *ParaStation MPI* cluster.

## Synopsis

**mpirun-ipath-ps** -np *nodes* [[-nodes *nodelist*] | [-hosts *hostlist*] | [-hostfile *hostfile*]] [-sort { proc | load | proc+load | none } ] [-exports=*envlist*] [-keep-mpihosts] [-keep-mpihosts] [ -v | --verbose ] [ -V | --version ] [ -n | -num-recv-bufs=*num* ] [ -N | -num-send-bufs=*num* ] [ -q | -quiescence-timeout=*num* ] [ -S | -short-len=*length* ] [ -L | -long-len=*length* ] [ -W | -rndv-window-size=*size* ] [ -c | -psc-spin-count=*count* ] [-stdin=*filename*] [-wdir=*path*] [-stdin-target=*rank*] [ -? | --help ] [--usage] *command* [ *args* ]

## Description

**mpirun-ipath-ps** is a tool that enables programs linked with the InfiniPath MPI library to run on a *ParaStation MPI* cluster under control of the *ParaStation MPI* management facility. Within *ParaStation MPI* the startup of parallel jobs is handled as described within the process_placement(7) manual page. The spawning mechanism is steered by environment variables, which are described in detail within ps_environment(7).

The command **mpirun-ipath-ps** is part of the `psipath.rpm` package. Please contact <`support@par-tec.com`> for details how to obtain this package.

**mpirun-ipath-ps** typically works like this:

```
mpirun-ipath-ps -np num prog [args]
```

This will startup the program **prog** *num* times in parallel forming a parallel job. *Args* are optional argument which will be passed to each instance of **prog**.

## Options

`-np` *nodes*
: Number of processes to create.

`-nodes=`*nodeslist*
: list of nodes to use.

`-hosts=`*hosts*
: list of hosts to use.

`-hostfile=`*hostfile*
: hostfile to use.

`-sort=`*criteria*
: Sorting criteria to use: *proc*, *load*, *proc+load* or *none*.

`-exports=`*envlist*
: Environment to export to foreign nodes.

`-keep-mpihosts`
: don't remove mpihosts file upon exit.

`-v, --verbose`
: Print diagnostic messages.

`-V, --version`
: Output version information and exit.

`-n, -num-recv-bufs=`*num*
: Number of receive buffers in runtime (deprecated and ignored).

`-N, -num-send-bufs=`*`num`*
: Number of send buffers in runtime.

`-q, -quiescence-timeout=`*`secs`*
: Wait time in seconds for quiescence on the nodes. Useful for detecting deadlocks. Value of 0 disables quiescence detection.

`-S, -short-len=`*`len`*
: Message length in bytes below which short message protocol is to be used (deprecated and ignored).

`-L, -long-len=`*`len`*
: Message length in bytes above which rendezvous protocol is to be used.

`-W, -rndv-window-size=`*`size`*
: Window size in bytes to use for native rendezvous.

`-c, -psc-spin-count=`*`count`*
: Number of times to loop for packets before yielding.

`-stdin=`*`filename`*
: Filename that should be fed as stdin to the node program.

`-wdir=`*`path`*
: Sets the working directory for the node program.

`-stdin-target=`*`rank`*
: Process rank that should receive the stdin file specified via -stdin option. Specify -1 if every process needs stdin.

`-?, --help`
: Show this help messages.

`--usage`
: Display brief usage message.

## See also

psmstart(1), ps_environment(7), process_placement(7)

# mpirun_openib

mpirun_openib — run a MVAPIch MPI program on a *ParaStation MPI* cluster.

## Synopsis

**mpirun_openib** -np *`nodes`* [[-nodes *`nodelist`*] | [-hosts *`hostlist`*] | [-hostfile *`hostfile`*]] [-sort { proc | load | proc+load | none } ] [-exports=*`envlist`*] [ -v | --verbose ] [ -V | --version ] [ -? | --help ] [--usage] *`command`* `[` *`args`* `]`

## Description

**mpirun_openib** is a tool that enables programs linked with the MVAPIch MPI library to run on a *ParaStation MPI* cluster under control of the *ParaStation MPI* management facility. Within *ParaStation MPI* the startup of parallel jobs is handled as described within the process_placement(7) manual page. The spawning mechanism is steered by environment variables, which are described in detail within ps_environment(7).

**mpirun_openib** typically works like this:

```
mpirun_openib -np num prog [args]
```

This will startup the program **prog** *`num`* times in parallel forming a parallel job. *`Args`* are optional argument which will be passed to each instance of **prog**.

## Options

`-np` *`nodes`*
: Number of processes to create.

`-nodes=`*`nodeslist`*
: list of nodes to use.

`-hosts=`*`hosts`*
: list of hosts to use.

`-hostfile=`*`hostfile`*
: hostfile to use.

`-sort=`*`criteria`*
: Sorting criterion to use: *`proc`*, *`load`*, *`proc+load`* or *`none`*.

`-exports=`*`envlist`*
: Environment to export to foreign nodes.

`-v, --verbose`
: Print diagnostic messages.

`-V, --version`
: Output version information and exit.

`-?, --help`
: Show this help messages.

`--usage`
: Display brief usage message.

## See also

psmstart(1), ps_environment(7), process_placement(7)

# psh

psh — run a command on or copy a file to multiple nodes.

## Synopsis

**psh** [ OPTIONS ] [ *`command`* [ *`args`* ] ]

**psh** [ OPTIONS ] [ -s ] [ *`files`* | *`directories`* ]

## Description

**psh** runs a command on all or a group of nodes. The output of each command is parsed and common parts are only shown once.

With the option `-s` or `-sync`, one or more file(s) and/or entire directories (including subdirectories) are synchronized (copied) to the selected nodes.

## Global options

`-c, -configfile` *`file`*
: Define a group and a configuration file to use. The file is looked up in the current directory.

`-g, -group` *`group`*
: Use nodes and other parameters in group *`group`*, defined in the configuration file `~/.psh.d/`*`group`*. This is short for of `-c ~/.psh.d/`*`group`*.

`-n, -node` *`nodelist`*
: Add nodes *`nodelist`* to the list of nodes.

`-x, -xnode` *`nodelist`*
: Exclude nodes *`nodelist`* from the list of nodes.

`-rcmd` *`command`*
: Use command *`command`* to start up a remote command. *`Command`* could be **pssh** (default), **rsh** or **ssh**.

`-diff` *`command`*
: Use command *`command`* to parse and compare the command output. Default is `diff -q`.

`-autocd` *`path`*
: Change to directory *`path`* on each node before running the command or copying the file. Default is `${PWD/${HOME}/~}`. Use an empty path (`""`) to disable changing to a directory first.

`-v`
: Be verbose.

`-h`
: Show a help message.

Without option `-g` or option `-c`, a default file named like the current hostname, the current hostname without trailing numerical characters ([0-9]*), or *`default`* (in this order) within `~/.psh.d/` is used. If none of this files is found, an empty template file `~/.psh.d/default` is created. This file may be modified to fit your needs!

*`Nodelist`* is a comma separated list of node names. The list can be abbreviated to *`node-[from-to]`*, e.g. *`node-[003-115,001]`* will be expanded to *`node-001`*, *`node-003`*, *`node-004`*, ..., *`node-115`*.

## Examples

This example shows a configuration file `~/.psh.d/rack1` defining nodes `node-01` up to `node-16` within rack number 1:

```
## psh defaults for rack 1
## uncomment all options you need...

## default nodelist:
# node nodename[01-16]
node node-[01-16]

## default remote command:
# rcmd ssh -x

## don't start more than 1 remote shell at the same time?
# max 1

## be always verbose ?
# verbose
```

To see the uptime of all nodes in rack 1, execute

```
# psh -g rack1 uptime
```

In order to compare the current date on all nodes, run `date` on all nodes:

```
# psh date
=== node-[01-04,06-16]
Mon Jul 18 12:45:17 CEST 2005
=== node-05
Mon Jul 18 12:45:18 CEST 2005
```

To calculate and compare the md5sum of all Linux kernels on all but node `node-03`, execute:

```
# psh -x node-03 md5sum /boot/vmlinux
=== node-[01,02,04,06-16]
c9c0728c41ca03a9ee9982869e28216e  /boot/vmlinuz
=== node-05
ff62b57a078f1f8244796c5609a053e8  /boot/vmlinuz
```

To copy the file `/etc/hosts` to all nodes, execute:

```
# psh -sync /etc/hosts
Prepare files for distribution:
/tmp/pshout.1235 /etc/hosts
Continue (y/n)?[n]
```

(Presuming a suitable configuration file within `~/.psh.d` is found.)

The local file `/etc/hosts` is also replaced by a copy of this file.

To copy the file `/etc/hosts` to all but the local node, execute:

```
# psh -s -x `hostname` /etc/hosts
Prepare files for distribution:
/tmp/pshout.1236 /etc/hosts
Continue (y/n)?[n]
```

## Files

`~/.psh.d/*`
: Configuration files for different groups.

## See also

pssh(8) and psmstart(8).

## psmstart

psmstart — start a program on a remote node

### Synopsis

**psmstart** program [arg]...

### Description

Start **program** on a remote node using the *ParaStation MPI* daemon psid(8). All arguments given at the command line will be passed to the remotely started **program**.

All output produced by **program** will be forwarded to the local terminal. All input **program** is expecting will be read from the local terminal. If the remote program catches a signal, a message will be printed.

The node, where the command is run on, will be automatically selected by *ParaStation MPI*, see process_placement(7) for details.

### Implementation

If **psmstart** is called, **program** is started remotely using the *ParaStation MPI* daemon psid(8). Within the startup process the local process will become a I/O handling process - marked as a logger process within psiadmin(8) - forwarding all the input needed and output produced by the remote process, respectively.

### See also

psid(8), psiadmin(8), pssh(8), ps_environment(7), process_placement(7)

## pssh

pssh — run a command on a remote node.

### Synopsis

**pssh** [ -e `envlist`] [ -l *name*] [ -r ] [ -v ] { -n `node-id`} { -h `hostname`} [`command` [`args`]]

**pssh** [ -V ]

**pssh** [ -? ]

### Description

**pssh** runs a command on a remote node. It is similar to the **ssh** command, except that the *ParaStation MPI* daemon is used to start the remote command.

### Global options

`-e, --exports=` *`envlist`*
: Export the environment variables *`envlist`* to the remote command. *`Envlist`* may be a single variable name or a colon separated list of names.

`-h, --host=` *`node`*
: Run command on node *`node`*.

`-l, --login=` *`name`*
: Login as user *`name`*. Only root may login as a different user on the node.

`-n, --node=` *`node-id`*
: Run command on node with ID *`node-id`*.

`-r, --rusage`
: Report resource usage upon exit.

`-v`
: Be more verbose.

`-V`
: Print version and exit.

`-?, --help`
: Show a help message.

`--usage`
: Show a usage message.

### Extended description

The **pssh** command runs a shell or a command *`command`* providing arguments *`args`* as an admin-task on a remote node. The local user must be within the `adminuser` list of the *ParaStation MPI* daemon or must be a member of the `admingroup` list. Refer to psiadmin(8) and *ParaStation MPI User's Guide* for details how to configure `adminusers` or `admingroups`.

Only root may use the `-l` option to login as a different user.

The destination host may be either specified using the hosts *ParaStation MPI* node ID (option `-n`) or using the hosts name (option `-h`).

The admin-task run by **pssh** will not be counted for the *ParaStation MPI* process placement and will not be started obeying the placement rules enforced by *ParaStation MPI*.

To run a serial task using the *ParaStation MPI* process placement rules, use psmstart(8). To run a command in parallel on multiple nodes, use psh(8). To run a parallel job, use mpiexec(8).

## Limitations

Currently, there is no way to change the user id except for root. There is no way to provide a password. The data is not encrypted while transfered across the network.

## See also

psmstart(8), psh(8), psiadmin(8) and parastation.conf(8).

# pscp

pscp — copy a file to multiple nodes in parallel.

## Synopsis

**pscp** [ -p? ] [ -h *hostlist* | -n *nodelist* ] [ -i *filename* ] [ -o *filename* ] [ -I *command* ] [ -O *command* ] [ -m *num* ] [ -s *server* ] [ -l *port* ] [ --maxsize *size* ] [ --tokens *num* ] [ --lowtokens *num* ] [ -v *level* ]

**pscp** [ -V ]

## Description

The **pscp** command is designed to copy files to many nodes in real parallelism. The implemented strategy ensures scalability up to hundreds of nodes for arbitrary file sizes.

Beside copying a single file to all destination hosts, the **pscp** command may generate and unpack on-the-fly archives to copy a set of files at once. Data compression could also be enabled by defining proper compress/uncompress commands.

## Options

`-h, --hosts` *hostlist*
: List of hosts to copy the file(s) to.

`-n, --nodes` *nodelist*
: List of node ids to copy the file(s) to.

`-i, --input` *filename*
: Filename to be sent to remote nodes.

`-o, --output` *filename*
: Filename stored on the remote nodes. In case of the option `-i` is given, this option must be given, too, even if input and output file are the same.

`-I, --icmd` *command*
: Command to pack a file or list of files to be transmitted. The file or list of files to be transmitted must be part of *command*, see examples below. The command must send its output to stdout.

`-O, --ocmd` *command*
: Command to unpack a file or list of files received.

`-C, --cp` *files*
: Create and unpack a tar-file on-the-fly including sub-directories.

  This is a short hand for `-I "/bin/tar cvPf -` *files*`" -O "/bin/tar xPf -"` .

`-m, --manual` *num*
: Manually start up *num* clients. Only for debugging purposes.

`-s, --server` *server*
: Server to connect to for clients started using `-m` option.

`-l, --lport` *port*
: Port number to connect to for clients started using `-m` option.

`-p, --progress`
: Show progress information.

`--maxsize` *size*
: Size of chunks to send or receive at once.

`--tokens` *num*
: Number of tokens to use for flow control stop.

`--lowtokens` *`num`*
: Number of low tokens to use for flow control start.

`-v, --verbose` *`level`*
: Be more verbose. *`Level`* could be in the range from 0 to 3.

`-V, --version`
: Print **pscp** version and exit.

`-?, --help`
: Print usage and exit.

*`Hostlist`* and *`nodelist`* may be comma separated lists of host names or host id, respectively.

If neither *`hostlist`* nor *`nodelist`* is given, the file will be copied to all but the local host. All files will be copied relatively to the current working directory.

The options `-i`, `-o` and `-I`, `-O`, `-C` are mutually exclusive.

## Extended description

**Pscp** copies an input file or an input data stream to an output file or an output data stream on many nodes in parallel. Writing data to disk and forwarding data to the next node takes place in parallel.

**Pscp** uses the *ParaStation MPI* `psport` library for data transfers, that automatically will use the most effective communication channel available. If required, the communication layer may be controlled using environment variables, refer to ps_environment(7) for details. The client process on each node is spawned using the *ParaStation MPI* process management.

As **pscp** uses administrative *ParaStation MPI* tasks to spawn the client processes, the user must be a member of the `adminuser` list or the user's group must be a member of the `admingroup` list. Refer to *ParaStation MPI User's Guide* and psiadmin(8) for details.

## Examples

Copy the file `/boot/vmlinuz` to all other cluster nodes:

```
  pscp -i /boot/vmlinuz -o /boot/vmlinuz
```

To copy all files and directories within `/boot` to hosts `node5` up to `node9` and show the progress, enter

```
  pscp -C /boot -h node5,node6,node7,node8,node9 -p
```

To send the file `testfile2` to nodes `node01` to `node19` and compress/decompress it on-the-fly, enter

```
  pscp -h `seq -s , -f "node%02g" 10 19` -I \
    "bzip2 -c testfile2" -O "bunzip2 > testfile2"
```

The same will be accomplished using

```
  pscp -n `seq -s , -f "node%02g" 10 19` -I \
    "/bin/tar cvPfj - testfile2" -O "/bin/tar xPfj -"
```

The options `-i` and `-O` may be combined to do things like

```
  pscp -h node03,node17 -i myfiles.tar.gz -O "/bin/tar xPfz -"
```

This command will transfer the already existing compressed tar file `myfiles.tar.gz` to nodes `node03` and `node17` and will decompress and unpack the archive on each node.

## Limitations

The size of the file or archive transfered to the remote nodes is only limited by the available disk space on the particular nodes and the underlying filesystem.

## See also

psh(8), psiadmin(8). ps_environment(7).

# process_placement

process_placement, spawning — Process placement strategies within *ParaStation MPI*

## Description

The placement of processes for a parallel task within *ParaStation MPI* can be biased by various environment variables. Setting this variables is optional. If not set, the default behavior will make sure, that the usability of the cluster will not be jeopardized.

As long as overbooking of CPUs is not explicitly required, the algorithm will ensure, that only one process [1] per CPU is spawned. This behavior ensures that parallel applications will influence each other as minimal as possible.

If no environment variables are set, *ParaStation MPI* tries to select nodes, where fewest compute processes are running and which have the lowest system load. Typically, this nodes are unused. On this nodes, processes are spawned in a manner, that consecutive ranks are placed on the same node, if possible. If there are not enough CPUs available, the spawning facility will **not** wait for free CPUs and will also **not** overbook CPUs.

### Pre-defined node selection

While starting up a parallel task, the following environment variables control the creation of the temporary node list used internally for spawning processes. Defining one of these variables enable the user to control the placement of processes. Likewise, batch systems might use this variables to place parallel jobs on dedicated nodes.

`PSI_NODES`
: Contains a comma separated list of node ID ranges. Each node ID range consists of a single node ID (numerical value) or a range of node IDs, including both the first and last ID noted and separated by a "-".

  E.g. defining the environment variable

```
PSI_NODES="0,1,3,17-20"    # must be exported!
```

  will enable the nodes with IDs 0, 1, 3 and 17 up to (and including) 20 to form the partition used by all subsequent parallel task.

`PSI_HOSTS`
: contains a list of host names, separated by white spaces.

  Defining the environment variable

```
PSI_HOSTS="node0 node1 node3 node17 node18 node19 node20"
```

  will enable the nodes 0, 1, 3 and 17 up to 20 to form the partition used by all subsequent parallel task.

`PSI_HOSTFILE`
: contains a filename listing all desired nodes by name, one per line.

A particular node can be listed several times. Depending on further environment variables, especially `PSI_NODES_SORT`, this node will be used more than once. This behavior is important for nodes housing more than one CPU.

The environment variables will be evaluated in the listed order. The first defined variable will be used, all following ones will be ignored. E.g., if `PSI_NODES` is set, `PSI_HOSTS` and `PSI_HOSTNAME` will not be recognized.

[1] The term *process* in this chapter refers to a compute process, initiated by *ParaStation MPI*. Other processes running on node, e.g. system processes or daemons are not considered.

If none of these variables is set, all nodes within the cluster will be taken into account for the temporary node list.

## Node availability and partitioning

In a next step, the previously defined temporary node list will be further checked for various constraints prohibiting the startup of processes on this nodes. In particular, the following verifications are made:

- Is the node currently available? Which means, is there currently a connection to the *ParaStation MPI* daemon on this node?

  If a node is shut down or crashed, the connections to the psid(8) on this node will time out, and this node will be declared as "dead".
- Is the node supposed to run processes?

  Nodes can be excluded from running compute processes by setting the `runJobs` attribute to no for a node within the configuration file `/etc/parastation.conf`.
- Is the node preallocated for other users or group of users?

  Nodes can be preallocated for a user or a group of users by using the `set user` or `set group` command while running **psiadmin**.
- Is the node currently used exclusively by another task?

  See below.
- Is the number of "regular" processes[1] currently running on this node less than the maximum number of processes allowed on this node? See `set maxproc` of **psiadmin**.
- Is the number of "regular" processes currently running on this node less than the number of CPUs available on this node?

  *ParaStation MPI* will only count physical CPUs, even if Hyperthreading is enabled. For more information on physical and logical CPUs, refer to *ParaStation MPI Administrator's Guide*.
- Is the communication hardware and the underlying protocol available?

As already mentioned, there are more environment variables influencing the selection for the temporary node list:

- `PSI_EXCLUSIVE`

  Only those nodes will be selected, where currently no other process is running. In addition, this nodes will be blocked for further tasks, until the current task terminates.
- `PSI_OVERBOOK`

  Normally, only as many processes can be run on a node as CPUs are available. If this environment variable is set, this limitation is no longer considered and any number of processes can be run on this node.

  Currently, defining this variable will also enforce `PSI_EXCLUSIVE`.

For both variables, it is sufficient to be defined. The actual value will not be recognized.

## Sorting nodes

The temporary node list will be sorted accordingly to the value of the environment variable `PSI_NODES_SORT`, if defined, or the configuration parameter **PSINodesSort** in parastation.conf(5). See also psiadmin(8).

- `PROC`

Sorting is done according to the number of currently running processes on each node.

- `LOAD` or `LOAD1`

  The node list will be sorted according to system load of the last 1 minute.

- `LOAD_5`

  The node list will be sorted according to system load of the last 5 minutes.

- `LOAD_15`

  The node list will be sorted according to system load of the last 15 minutes.

- `PROC+LOAD`

  The node list will be sorted according to the number of currently running processes per node (PROC) and the load average for the last minute (LOAD) of these particular node.

- `NONE`

  No sorting at all is done.

If neither the environment variable `PSI_NODES_SORT` is defined nor the parameter **PSINodesSort** in parastation.conf(5) is configured, the partition table will be sorted according to the number of processes per node (PROC). If the variable is defined, but the value is not known, an error will be reported. The value of this variable is not case sensitive.

Beside the listed sorting criteria(s), there are additional ones applied afterwards:

- Different number of CPUs

  If there are nodes with different number of CPUs, nodes with higher CPU count will be sorted before nodes with less CPU count.

- Identical number of CPUs

  Nodes with equal CPU count will be sorted according to there *ParaStation MPI* ID.

These criteria will enforce an explicit order of the temporary node list for all possible states of the particular nodes.

## Process placement

The real distribution of the processes on the nodes defined by the temporary node list is controlled by two more environment variables: `PSI_OVERBOOK` and `PSI_LOOP_NODES_FIRST`. Depending whether this variables are defined, the processes of a parallel task will be spread on the temporary node list.

- none defined:

  Beginning with the first node of the temporary node list, processes will be placed on the node as long as the current process count is less than the number of CPUs. This will happen as long as all processes are placed or the list is exhausted. If all nodes in the list are done and there are still processes to place, the startup of the parallel task will be canceled.

- `PSI_LOOP_NODES_FIRST` is defined:

  Beginning with the first node of the temporary node list, one process will be placed on each node of the list, if the number of processes on this node is less than the number of CPUs. The end of the list will wrap to the beginning. Searching the list will be done as long as there are still processes left. If there are no more nodes available where the number of processes is less than the number of CPUs, the startup of the parallel task will be canceled.

- `PSI_OVERBOOK` is defined:

  If there are at least as many "unused" CPUs on all the nodes of the temporary node list as processes to start, the behavior is identical to the action if this variable is not defined.

  If there are more processes requested than unused CPUs available, the algorithm evenly distributes the processes on all CPUs. The actual placement is done in the order of the temporary node list. Each node will be filled up with the calculated number of processes. Limits defined by the administrator, e.g. `set maxproc` will be enforced. If not all processes can be placed on a node, the startup of the parallel task will be canceled.
- `PSI_LOOP_NODES_FIRST` and `PSI_OVERBOOK` are defined:

  If there are enough "unused" CPUs available, the behavior for this combination is identical to the behavior describe previously for "`PSI_LOOP_NODES_FIRST` is defined". Otherwise the placement is done in a manner that the processes are evenly distributed on all nodes in the temporary node list. For this purpose the node list is cyclic traversed and each time a process is placed on the node. All defined limits will be obeyed. If it's not possible to place all processes, the startup of the parallel task will be canceled.

The environment variable `PSI_WAIT` controls the behavior, if the startup of the task was previously canceled due to node constrains. If not defined, an error will be reported and the process terminates.

If this variable is defined, the startup request will be queued. Each time the resource allocation within the cluster changes, e.g. if a task terminates or a new node is detected, the startup requests queued up to now will be reevaluated as long as the next request cannot be fulfilled. Requests are queued and dequeued in a "first come first server" order. There is only one queue for the entire cluster. It is not possible for a request to bypass other requests, algorithms like "backfilling" are not implemented.

### Process pinning

Each process and all its child processes started up on a node may be pinned to a particular CPU-slot (= virtual core). Therefore these processes will not interfere to each other with respect to CPU cycles. On process startup, the system will use the default mapping as defined in the configuration file to map CPU-slots to physical cores. In case this mapping is not appropriate, the environment variable `__PSI_CPUMAP` may be used to override the default mapping, if user based mapping is enabled.

In case a process is creating child processes or is using threads, the variable `PSI_TPP` may be used to define how many CPU-slots per process will be allocated and therefore are available for a particular process. This is in particular interesting for applications using OpenMP. Hence, the environment variable `OMP_NUM_THREADS` is also honored. If both variables are defined, `PSI_TPP` takes precedence. if none of them is defined, it defaults to 1.

Process pinning may be disabled for a particular job by defining the environment variable `__PSI_NO_PINPROC`. The value itself is thereby irrelevant. By doing so, it's almost always a good idea to also disable memory binding by defining `__PSI_NO_BINDMEM`.

## See also

ps_environment(7), parastation.conf(5) and psiadmin(8)

# ps_environment

ps_environment — *ParaStation* environment variables

## Description

The behavior of the *ParaStation* system when starting up parallel tasks using mpiexec(1) or submitting serial jobs using psmstart(1) might be affected using different environment variables.

Further variables may be used in order to modify the behavior of the logging facilities implementing a reliable forwarding of input and output.

The last section describes some less frequently used environment variables which affect the behavior of the MPIch system implementing the MPI interface on top of *ParaStation*.

## Variables managing the program startup

The following environment variables are used during startup of parallel tasks or while distributing serial jobs throughout a cluster. Depending on their value a splitting of the cluster into virtual partitions is done and the load balancing strategy is controlled.

`OMP_NUM_THREADS`
: Defines the number of cores allocated for each process. May be overwritten by `psenvironment_PSI_TPP`.

`PSI_EXCLUSIVE`
: Only unused nodes are considered for spawning new processes. In addition, the nodes chosen for the current job will be locked for further jobs, consequently no additional processes will be started on this nodes until the current job terminates.

  This variable does not define, how many processes of a job will be placed per node. See also `PSI_OVERBOOK` and **set maxproc** of psiadmin(8).

`PSI_EXPORTS` *VAR* [, *VAR*]...
: A list of environment variables which should be exported to remote processes during spawning. Some environment variables are exported by default: `HOME`, `USER`, `SHELL`, `TERM`, `LD_LIBRARY_PATH`, `LD_PRELOAD`, `MPID_PSP_MAXSMALLMSG`, In addition, all variables named `PSP_*`, `__PSI_*` or `OMP_*` are exported. Therefore, the variable `OMP_NUM_THREADS` is exported automatically.

  Furthermore `PWD` is set correctly for remote processes. In addition, the environment used for partitioning the cluster (i.e. `PSI_NODES`, `PSI_HOSTFILE` or `PSI_HOSTS` and `PSI_NODES_SORT`) is propagated to remote processes.

`PSI_NODES` *`number`* [, *`number`*]...
: Defines the nodes building the partition used to spawn new processes to. Depending on the variable `PSI_NODES_SORT` the ordering may be relevant. If the number of processes to spawn exceed the number of nodes in the partition, some nodes may get more than one process.

  See also `PSI_HOSTS` and `PSI_HOSTFILE`.

`PSI_HOSTS` *`hostname`* [ *`hostname`*]...
: Space separated list of hostnames on which new processes should be spawned on. Similar to `PSI_NODES`, but with hostnames instead of logical ParaStation node numbers. If `PSI_NODES` is set too, it is dominant over `PSI_HOSTS`.

  See also `PSI_NODES` and `PSI_HOSTFILE`.

`PSI_HOSTFILE` *`filename`*
: The name of a file containing a list of nodes' hostnames which should be used for spawning. Similar to `PSI_HOSTS` but the actual information is within the file instead of the environment variable. If `PSI_NODES` or `PSI_HOSTS` are set too, they are dominant over `PSI_HOSTFILE`.

See also `PSI_NODES` and `PSI_HOSTFILE`.

`PSI_NODES`, `PSI_HOSTS` and `PSI_HOSTFILE` are evaluated in the given order. If more than one of the discussed variables is set, only the first one will be used in order to create the partition. The latter ones will be silently ignored.

`PSI_LOOP_NODES_FIRST`
: This variable controls the behavior of *ParaStation* when placing processes on nodes. If `PSI_LOOP_NODES_FIRST` is not defined, *ParaStation* first of all will try to use all available CPUs on a node for the current job. If necessary, more processes will be placed on the next nodes. If `PSI_LOOP_NODES_FIRST` is defined, *ParaStation* will place one process per node, and if more processes as available nodes are requested, it will start putting an additional process on each node, as long as all processes are placed; or the placement couldn't be fullfilled, e.g. due to the fact that not enough CPUs are available.

`PSI_NODES_SORT` *`mode`*
: This variable defines the sorting criterion used to reorder the nodes building a virtual partition. This order will be used to spawn remote processes. The following values of *`mode`* are recognized:

  `ROUNDROBIN`
  : No sorting of nodes before a spawn request. The nodes are used in round robin fashion as they are set in `PSI_NODES`, `PSI_HOSTS` or `PSI_HOSTFILE`.

  `NONE`
  : Same as `ROUNDROBIN`

  `LOAD`
  : The nodes are sorted by load before new processes are spawned. Therefore nodes with the least load are used first.

    To be more specific, the load average over the last minute is used as the sorting criterion, i.e. this option is equivalent to `LOAD_1`.

  `LOAD_1`
  : The nodes are sorted corresponding to the 1 minute load average.

    This option is equivalent to `LOAD`.

  `LOAD_5`
  : The nodes are sorted corresponding to the 5 minute load average.

  `LOAD_15`
  : The nodes are sorted corresponding to the 15 minute load average.

  `PROC+LOAD`
  : The nodes are sorted corresponding to the sum of the 1 minute load and the number of running *ParaStation* processes. This will lead to fair load-balancing even if processes are started without notification to the *ParaStation* management facility.

  `PROC`
  : The nodes are sorted by the number of running *ParaStation* processes before new processes are spawned. This is the default behavior.

`PSI_OVERBOOK`
: If defined, more processes per node will be placed than CPUs available, if necessary. If undefined, only as many processes will be placed on a node as unused CPUs (= number(CPU) - number(currently running processes)) are available.

  See also **set maxproc** of psiadmin(8), which takes precedence over `PSI_OVERBOOK`.

`PSI_TPP`
: Defines the number of cores allocated per process. If undefined, defaults to 1.

See also `psenvironment_OMP_NUM_THREADS`.

`PSI_WAIT`
: If defined, new job start request will be queued, if not enough resources are currently available. See Chapter 3, *Starting up programs* and psmstart(1) for more details.

`PSI_RARG_PRE_{n}`
: Preceding arguments for remote processes. For example: use `PSI_RARG_PRE_0=/usr/bin/time` to execute the process chain `/usr/bin/time <yourApplication> <yourArgs>` on the remote nodes.

`PMI_BARRIER_ROUNDS`
: This parameter defines after how many `PMI_BARRIER_TMOUT` cycles a job will be terminated, if not all processes have joined the PMI barrier. Defaults to 1.

The parameter should remain at the default value in production environments. This parameter's primary use is for diagnostic purposes as it allows the user to observe slower clients join an PMI barrier over multiple timeout periods. As such, the parameter helps administrators identify possible filesystem or network issues that occur on specific client nodes.

PMI barriers are totally unrelated to MPI barriers! These type of barriers are typically called during `MPI_INIT()`.

`PMI_BARRIER_TMOUT`
: The `PMI_BARRIER_TMOUT` variable defines the delay (in seconds) allowed for each process to successfully join an PMI barrier. If not all processes joined, a corresponding warning is printed to stdout.

If `PMI_BARRIER_TMOUT` is not set, the timeout will be 60sec + (# of processes * 0.5µsec). If `PMI_BARRIER_TMOUT` equals `-1`, no barrier timeout is used and the job will not terminate because of failure to join the barrier from any one process. If `PMI_BARRIER_TMOUT` is set to *`num`*, then the timeout is set to *`num`* seconds.

See also *ParaStation MPI Administrator's Guide*.

`__PSI_NO_PINPROC`
: If set, suppress pinning of processes, even if enabled globally (value irrelevant).

`__PSI_NO_BINDMEM`
: If set, suppress binding to memory-node, even if enabled globally (value irrelevant).

## Variables controlling the communication layer

This variables control the individual communication paths used by the `pscom` library. Communication paths may be different interconnects and / or protocols. In addition, tuning variables for the particular communication paths are listed.

The following table lists all currently available communication paths in descending order. Using this variables, transports may be prioritized or completely disabled. Assigning a value of `0` to a variable completely disables this communication path. Assigning a value of 2 or more prioritizes the path over all others.

| Variable name | Communication path | Description |
|---|---|---|
| PSP_SHM | Shared memory | Used only for communication within a node. Disabled otherwise. Identical to the deprecated variable PSP_SHAREDMEM. |
| PSP_OPENIB | InfiniBand (libopenib) | |
| PSP_OFED | InfiniBand (libopenib) | Using UD |
| PSP_MVAPI | InfiniBand (libmvapi) | |
| PSP_ELAN | QsNet | Disabled by default. |
| PSP_DAPL | InfiniBand (libdapl) | |
| PSP_GM | Myrinet (libgm) | |
| PSP_P4S | *ParaStation* p4sock protocol | Identical to the deprecated variable PSP_P4SOCK. |
| PSP_TCP | TCP | |

Table 3. Variables controlling the pscom communication paths

Not all transports may be available at run time due to missing hardware or low level libraries. Furthermore, not all transports are enabled within the precompiled packages.

`PSP_LIB`

Using this environment variable, it is possible to define the communication library to use, independent of the variables mentioned above. This library must match the currently available interconnect and protocol, otherwise an error will occur.

The library name must be specified using the full path and filename, e.g. `PSP_LIB=/opt/parastation/lib64/libpsport4openib.so`.

`PSP_NETWORK` *`network`* [, *`network`*]

A comma or space separated list of networks enabled to do optimized *ParaStation* communication using the p4sock protocol or TCP. Each *`network`* is a resolvable hostname in the chosen network, the IP address of a host in this network or the IP address of this network. The corresponding network has to be bound to a NIC of the current node.

If `PSP_NETWORK` is set, each *`network`* should be bound to a distinct NIC. This card then is used in order to do communication operations. If more than one *`network`* is given, the first one found to be bound to a local NIC is used.

If `PSP_NETWORK` is not set, *ParaStation* uses the NIC bound to the IP address, the local hostname resolves to.

`PSP_RETRY` *`count`*

Retry counter for all `connect()` calls within the `pscom` library. Default is 3.

`PSP_TCP_BACKLOG` *`count`*

TCP `listen()` backlog length. Only required for `pscom` library version below version 5.0.34.

The actual backlog is the minimum of `PSP_TCP_BACKLOG` and `net.core.somaxconn`, defined by the operating system.

Tuning Parameters

`PSP_ONDEMAND`

If set to 1, use "on demand" connections with `PSP_OPENIB`. This means, establish connections between ranks and allocate there associated communication buffers with the first byte send. This could cause application aborts at any time, if the application runs out of resources (e.g. a final all to one communication pattern could fail)! Default is to establish all connections at startup time (inside MPI_Init()) which assures, that there are enough resources available for all connections. If not, MPI_Init() will fail.

`PSP_SO_SNDBUF`, `PSP_SO_RCVBUF`
: These variables define the TCP buffer size used for TCP sockets. Defaults to 32k.

`PSP_TCP_NODELAY`
: If set to 1 (default), the socket option `NODELAY` will be used for TCP sockets.

`PSP_TCP_BACKLOG`
: control the size of the TCP backlog when listening for new connections.

`PSP_SCHED_YIELD`
: If set to 1, call sched_yield() in polling loops instead of busy polling. This might improve shared memory performance a lot, when there is more than one process per CPU core running, but slowdown communication performance in the common case of one process per core. (see also overbooking)

`PSP_OPENIB_PATH_MTU`
: Control the path MTU of InfiniBand connections. Default is 3 which correspond to 1024 bytes. (1 = 256 bytes, 2 = 512 bytes, 3 = 1024 bytes)

`PSP_OPENIB_SENDQ_SIZE`, `PSP_OPENIB_RECVQ_SIZE`
: These variables define the InfiniBand buffer counts used for InfiniBand connections. (Default = 16)

## Variables controlling the logger/forwarder

In order to modify the behavior of the logger and the forwarders controlling the remotely spawned processes, the following environment variable can be used:

`PSI_INPUTDEST` *`rank`*
: If set, psilogger will forward all input to the process with the corresponding rank within the process group. The default is to give all available input to process 0.

`PSI_RUSAGE`
: If set, psilogger will print a message about the user and system time consumed by each process of the parallel task upon exit of this process.

`PSI_SOURCEPRINTF`
: If set, psilogger gives information about the source of the received output, i.e. it will prepend every output by “[id]:”, where id is the rank of the printing process within the process group. Usually the id coincides with the MPI-rank. If `PSI_LOGGERDEBUG` is also set, every output is prepended by “[id, len]”, where id is the rank again and len is the length of the printed message in bytes.

`PSI_NOMSGLOGGERDONE`
: If set, psilogger will **not** print out the message “PSIlogger: done” at the end of a parallel run.

`PSI_LOGGERDEBUG`
: If set, psilogger gives debug output about connecting and detaching clients as well as received output from the clients.

`PSI_FORWARDERDEBUG`
: If set, debug output of the psiforwarder about connected programs, received input and received output is printed.

## Variables controlling MPIch

The environment variables within this section might be used less frequently. They are mainly listed within this document for completeness.

`MPID_PSP_MAXSMALLMSG`
: Length (in bytes) of the largest message sent without rendezvous.

`MPID_PSP_START`
Define the method used in order to spawn remote processes. The possible values are:

`PSID`
Start remote processes with the *ParaStation* start mechanism.

This is the default. If `MPID_PSP_START` is not set at all, *ParaStation* is used in order to spawn remote processes.

`SSH`
Start remote processes with ssh(1). `MPID_PSP_HOSTS` must be set.

`NONE`
Do not start any remote process. The remote processes must be started manually. A commandline template is printed to stdout.

This start mode is for debugging purposes only and should not be used by the end-user.

`MPID_PSP_HOSTS` *`hostname`* [, *`hostname`*]...
Comma separated list of hostnames. Used for `MPID_PSP_START=`*SSH* only.

## Variables controlling the TCP bypass

The environment variables within this section control the TCP bypass.

`LD_PRELOAD`
defines (beside others) the path to the required preload library to enable the TCP bypass. It must be set to `/opt/parastation/lib64/libp4tcp.so`

## Variables controlling debugging

The environment variables within this section control the debug information output by *ParaStation*.

`PSI_DEBUGMASK`
defines the debug mask controlling the process management information. The following bits are defined:

| Bit pattern | Name | Description |
|---|---|---|
| `0x0001` | PSC_LOG_PART | partitioning functions (i.e. PSpart_()) |
| `0x0002` | PSC_LOG_TASK | task structure handling (i.e. PStask_()) |
| `0x0004` | PSC_LOG_VERB | Various, less interesting messages |
| `0x0010` | PSI_LOG_PART | partition handling |
| `0x0020` | PSI_LOG_SPAWN | spawning |
| `0x0040` | PSI_LOG_INFO | info requests |
| `0x0080` | PSI_LOG_COMM | daemon communication |
| `0x0100` | PSI_LOG_VERB | more verbose stuff, e.g. function calls |

Table 4. PSI_DEBUGMASK flags

These debug flags may be set as hex numbers, e.g. `PSI_DEBUGMASK=0x07`.

`PSP_DEBUG`
defines the debugging level for the *ParaStation* `psport4` library. Higher values generally give more output.

## See also

process_placement(7)

# Glossary

| | |
|---|---|
| Address Resolution Protocol | A sending host decides, through a protocols routing mechanism, that it wants to transmit to a target host located some place on a connected piece of a physical network. To actually transmit the hardware packet usually a hardware address must be generated. In the case of Ethernet this is 48 bit Ethernet address. The addresses of hosts within a protocol are not always compatible with the corresponding hardware address (being different lengths or values).<br><br>The Address Resolution Protocol (ARP) is used by the sending host in order to resolve the Ethernet address of the target host from its IP address. It is described in the RFC 826. The ARP is part of the TCP/IP protocol family. |
| Administration Network | The administration network is used for exchanging (meta) data used for administrative tasks between cluster nodes.<br><br>This network typically carries only a moderate data rate and can be entirely separated from the data network. Almost always, Ethernet (Fast or more and more Gigabit) is used for this purpose. |
| Administrative Task | A single process running on one of the compute nodes within the cluster. This process does not communicate with other processes using MPI.<br><br>This task will not be accounted within the *ParaStation MPI* process management, ie. it will not allocate a dedicated CPU. Thus, administration tasks may be started in addition to parallel tasks.<br><br>See also Serial Task for tasks accounted with *ParaStation MPI*. |
| admin-task | See Administrative Task. |
| ARP | See Address Resolution Protocol. |
| Data Network | The data network is used for exchanging data between the compute processes on the cluster nodes. Typically, high bandwidth and low latency is required for this kind of network.<br><br>Interconnect types used for this network are Myrinet or InfiniBand, and (Gigabit) Ethernet for moderate bandwidth and latency requirements.<br><br>Especially for Ethernet based clusters, the administration and data network are often collapsed into a single interconnect. |
| CPU | Modern multi-core CPUs provide multiple CPU cores within a physical CPU package. Within this document, the term `CPU` will be used to refer to a independing computing core, independent of the physical packaging. |
| DMA | See Direct Memory Access. |
| Direct Memory Access | In the old days devices within a computer were not able to put data into memory on their own but the CPU had to fetch it from them and to store it to the final destination manually.<br><br>Nowadays devices as Ethernet cards, harddisk controllers, Myrinet cards etc. are capable to store chunks of data into memory on their own. E.g. a disk controller is told to fetch an amount of memory from a hard disk and |

| | |
|---|---|
| | to store it to a given address. The rest of the jobs is done by this controller without producing further load to the CPU.<br><br>Obviously this concept helps to disburden the CPU from work which is not its first task and thus gives more power to solve the actual application. |
| Forwarder | See *ParaStation MPI* Forwarder. |
| Logger | See *ParaStation MPI* Logger. |
| Master Node | The evaluation of temporary node lists while spawning new tasks is done only by one particular psid(8) within the cluster. The node running this daemon is called *master node*.<br><br>The master node is dynamically selected within the cluster and may change, if the current master node is no longer available. Election is based on the node IDs, refer to parastation.conf(5). |
| Network Interface Card | The physical device which connects a computer to a network. Examples are Ethernet cards (which are nowadays often found to be on board) or Myrinet cards. |
| NIC | See Network Interface Card. |
| Non-Uniform memory access (NUMA) | Non-Uniform memory access describes the fact that for some multiprocessor design the access time to the memory depends on the location of this memory. Within this designs, the memory is typically closely attached to a CPU. CPUs have access to memory attached to other CPUs using additional logic inducing additional latency. Therefore the access time for different memory addresses may vary. |
| Parallel Task | A bunch of processes distributed within the cluster forming an instance of a parallel application. E.g. a MPI program running on several nodes of a cluster can only act as a whole but consists of individual processes on each node. *ParaStation MPI* knows about their relationship and can handle them as a distributed parallel task running on the cluster.<br><br>Sometimes also referred as *job*. |
| *ParaStation MPI* Logger | The counterpart to the *ParaStation MPI* Forwarder. This process receives all output collected by the forwarder processes and sends it to the final destination, stdout or stderr. Furthermore input to the *ParaStation MPI* task is forwarded to a specific process.<br><br>The first process of the task started usually converts to the logger processes after spawning all the other processes of the parallel task. |
| *ParaStation MPI* Forwarder | Collects output written by *ParaStation MPI* controlled processes to `stdout` or `stderr` and sends it to the *ParaStation MPI* Logger.<br><br>Furthermore the forwarder controls the process and sends information about its exit status to the local daemon. |
| PMI | Process Manager Interface: protocol to standardize startup of tasks of a parallel job. Implemented in **mpd** and *ParaStation MPI* **psid**. |
| Process | The atomic part of a Parallel Task. A process is at first a standard Unix process. Since *ParaStation MPI* knows about its membership in a parallel task, it can be handled in a peculiar way if an event takes place on some |

| | |
|---|---|
| | other node (e.g. another process of the task dies unexpectedly, a signal is send to the task, etc.). |
| Serial Task | A single process running on one of the compute nodes within the cluster. This process does not communicate with other processes using MPI. *ParaStation MPI* knows about this process and where it is started from.<br><br>A serial task may use multiple threads to execute, but all this threads have to share a common address space within a node. |