Platform MPI
Version 9 Release 1.2

Release Notes for Linux

GI13-1896-01

Platform MPI
Version 9 Release 1.2

Release Notes for Linux

GI13-1896-01

Note
Before using this information and the product it supports, read the information in "Notices" on page 67.

First edition

This edition applies to version 9, release 1 of Platform MPI (product number 5725G83) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Information about this release

Announcement

IBM Platform MPI (Platform MPI) 9.1.2 is the fully functional IBM implementation of the Message Passing Interface standard for Linux. Platform MPI 9.1.2 for Linux is supported on Intel/AMD x86 32-bit, AMD Opteron and EM64T servers running CentOS 5, Red Hat Enterprise Linux AS 4, 5, and 6, and SuSE Linux Enterprise Server 9, 10, and 11 operating systems.

Platform MPI product information

Platform MPI is a high-performance and production-quality implementation of the Message Passing Interface standard. Platform MPI fully complies with the MPI-2.2 standard. Platform MPI provides an application programming interface and software libraries that support parallel, message-passing applications that are efficient, portable, and flexible.

Platform MPI enhancements provide low latency and high bandwidth point-to-point and collective communication routines. On clusters of shared-memory servers, Platform MPI supports the use of shared memory for intranode communication. Internode communication uses a high-speed interconnect when available.

Platform MPI supports a variety of high-speed interconnects and enables you to build a single executable that transparently uses the supported high-performance interconnects. This greatly reduces efforts to make applications available on the newest interconnect technologies.

Platform MPI is available as shared libraries. To use shared libraries, Platform MPI must be installed on all machines in the same directory or accessible through the same shared network path.

Platform MPI Community Edition

Platform MPI Community Edition is a no-charge edition of Platform MPI that supports core MPI features.

Platform MPI Community Edition embodies all of the core features of Platform MPI Standard Edition and is available for download and deployment at no charge. IBM Business Partners and Independent Solution Vendors (ISVs) who want to embed or include a free MPI as part of their solutions can distribute Platform MPI Community Edition at no charge or royalty upon registering with IBM.

An optional yearly subscription (Entry Support, or Elite Support provided by ISVs) is available for users who require technical support or additional functionality, such as greater scalability. Upgrading from Platform MPI Community Edition to either the yearly subscription or to the full Platform MPI Standard Edition only requires you to add an entitlement file (`pmpi.entitlement`) in the `$MPI_ROOT/etc` directory.

The following are the differences between Platform MPI Community Edition, Platform MPI Community Edition with Entry Support or Elite Support, and Platform MPI Standard Edition:

- Platform MPI Community Edition:
  - Scalability is limited to 4096 ranks. Larger rank counts will fail.
  - Cluster test tools work for up to four nodes in a single run. The benchmarking (BM), hello world (HW), and ping pong ring (PPR) tools are limited to 4096 ranks.
  - Support for high availability (HA) applications is disabled.
- Platform MPI Community Edition with Entry Support or Elite Support:
  - Scalability is limited to 8192 ranks. Larger rank counts will fail.
  - Cluster test tools work for up to four nodes in a single run. The benchmarking (BM), hello world (HW), and ping pong ring (PPR) tools are limited to 8192 ranks.
  - Support for high availability (HA) applications is disabled.
- Platform MPI Standard Edition:
  - There are no rank restrictions.
  - Cluster test tools work with no restrictions.
  - Support for high availability (HA) applications is enabled.

Platforms supported

Table 1. Systems Supported

Platform	Interconnects
Intel/AMD x86 32-bit	• Myrinet • InfiniBand • Ethernet Ethernet includes 10baseT, 100baseT, GigE, 10GbE, and 10GbE with RoCE.
AMD Opteron (32-bit and 64-bit)	• Myrinet • InfiniBand • Quadrics • Ethernet
EM64T (32-bit and 64-bit)	• Myrinet • InfiniBand • Quadrics • Ethernet

Platform MPI runs on the following Intel/AMD platforms: x86_64 (32- and 64-bit) and i386, and supports the following interconnects:

Table 2. Interconnect Support for Platform MPI 9.1.2

Protocol	Option	Supported Architectures	NIC Version	Driver Version
Shared memory on SMP	N/A	i386, x86_64	N/A	N/A

Table 2. Interconnect Support for Platform MPI 9.1.2 (continued)

Protocol	Option	Supported Architectures	NIC Version	Driver Version
OpenFabrics	`-IBV`	i386, x86_64	Any IB card	OFED 1.0, 1.1, 1.2, 1.3, 1.4, 1.5
uDAPL Standard	`-UDAPL`	i386, x86_64	• IB vendor specific • 10GbE vendor specific	uDAPL 1.1, 1.2, 2.0
QLogic PSM	`-PSM`	x86_64	QHT7140, QLE7140	PSM 1.0, 2.2.1, 2.2
Myrinet MX	`-MX`	i386, x86_64	Rev D, E, F, 10G	• MX 2g, 10g • V1.2.x
Myrinet GM	`-GM`	i386, x86_64	Rev D, E, F	GM 2.0 and later
TCP/IP	`-TCP`	i386, x86_64	All cards that support IP	Ethernet Driver, IP

Note:

Support for the Mellanox VAPI and Quadrics Elan protocols have been discontinued in Platform MPI.

Compilers

Platform MPI strives to be compiler neutral. Platform MPI 9.1.2 for Linux is supported with the following compilers:

- GNU 3.x, 4.x
- GNU glibc 2.x
- Intel 10.x, 11.x, 12.x
- PathScale 2.x, 3.x
- Portland Group 10.x, 11.x, 12.x

Directory structure

All Platform MPI (for Linux) files are stored under the `/opt/ibm/platform_mpi` directory.

If you choose to move the Platform MPI installation directory from its default location in `/opt/ibm/platform_mpi`, set the **`MPI_ROOT`** environment variable to point to the new location.

Table 3. Directory Structure

Subdirectory	Contents
bin	Command files for the Platform MPI utilities, gather_info script
help	Source files for the example programs
include	Header files
lib/linux_ia32	Platform MPI Linux 32-bit libraries

Table 3. Directory Structure (continued)

Subdirectory	Contents
lib/linux_amd64	Platform MPI Linux 64-bit libraries for Opteron and EM64T
newconfig	Configuration files and Copyright text
share/man/man1	manpages for the Platform MPI utilities
share/man/man3	manpages for the Platform MPI library
doc	Release Notes
EULA	License files

Bug fixes in Platform MPI 9.1.2

Platform MPI 9.1.2 has the following bug fixes from Platform MPI 9.1:

Bug fixes:

- Fix a bug on PSM with `-intra=mix` (use Platform MPI shared memory for on-host messages smaller than 2KB, and PSM off-host or >=2KB messages).
- Fixed a bug in `-ha` mode that might have resulted in an incorrect function name in an error message if there was an error.
- Fixed a occasional segv when using instrumentation `-i` *instrfile*
- Fixed the **MPI_TYPE_EXTENT** signature in the Fortran `module.F` file.
- Fixed a possible hang condition in **mpirun**/**mpid** when a failure occurs early in the application launching process.
- The **mpirun** options `-intra=nic` and `-commd` are mutually exclusive and **mpirun** now checks if the user attempted to use both.
- The use of the XRC (**mpirun** option `-xrc` or `MPI_IBV_XRC=1`) and on-demand connections (`PCMP_ONDEMAND_CONN=1`) are mutually exclusive and **mpirun** now checks if the user attempted to use both.
- Fixed a possible hang when on-demand connections (`-e PCMP_ONDEMAND_CONN=1`) are used with Infiniband VERBS. This issue could be seen when a rank rapidly sends many messages to ranks with which it has previously communicated, followed by a message to a rank with which it has not previously communicated.
- Fixed the affinitiy (**mpirun** option `-aff`) tunables **MPI_AFF_SKIP_GRANK** and **MPI_AFF_SKIP_LRANK** when multiple mpids within an application are located on the same node.
- Made various improvements to the non-blocking collective API implementation from the MPI-3 standard. Most notably, the messages used in the implementation of non-blocking collectives can no longer incorrectly match with users' point-to-point messages. A number of other quality improvements have been made as well such as argument checking (`MPI_FLAGS=Eon`) and better progression of non-blocking collectives.
- Fixed the case where multiple mpids running on a single node within an application can cause the following messages during startup: "MPI_Init: ring barrier byte_len error".
- Fixed the following runtime error that may occur with the on-demand connection features on IBV (`-e PCMP_ONDEMAND_CONN`): "Could not pin pre-pinned rdma region".

- Fixed a rare wrong answer introduced in 9.1.0.0 in -1sided **MPI_Accumulate**.
- Fixed a rare wrong answer introduced in 9.1.0.0 in the 101 collective algorithms, including **MPI_Allgather**, **MPI_Allgatherv**, **MPI_Alltoall**, **MPI_Alltoallv**, **MPI_Bcast**, **MPI_Reduce**, and **MPI_Scatter**.
- Fixed a frequent 32-bit Windows crash when using collective algorithms.
- Fixed a potential hang caused by conflicting user and internal messages introduced in 9.1.0.0.
- Fixed the following Windows error: "Error in cpu affinity, during shared memory startup".
- Fixed a launching error when using LSF **bsub -n min,max** on Windows.
- Fixed **MPI_Reduce** when using MPI_IN_PLACE and Mellanox FCA Collective Offloading.
- Fixed a potential hang when running a mix of ranks linked against the multi-threaded library and ranks linked with the single-threaded library.
- Fixed GPU wrong answers when using `-e PMPI_GPU_AWARE=1 -e PMPI_CUDAIPC_ENABLE=1`.
- Dynamically increased the number of SRQ buffers when getting close to RNR timeouts.
- Removed extraneous internal deadlock detection on shared memory pouches.
- Enabled ondemand connections for windows.
- Changed to use two digit version number sub fields: 09.01.00.01.
- Changed mpirun -version to align output correctly.
- Fixed a minor warning when MPI_ ROOT != mpirun path.
- Improved error messages in some collective algorithms.

HA changes

- Fixed `-ha` to support MPIHA_Failure_ack()
- Fixed `-ha` to prevent MPI_ANY_SOURCE requests from checking broken TCP links.
- Fixed an uninitialized value problem in MPI_COMM_SHRINK
- Fixed `-ha` so that multithreaded blocking recv will never return MPI_ERR_PENDING.
- Fixed allocation during Communicator creation in `-ha` not allocating enough.
- Fixed **MPI_Cancel** to work properly with MPI_ANY_SOURCE requests in `-ha`.
- Fixed a possible hang during **Finalize** in presence of revoked ranks.
- Fixed benchmarking to not run unneeded warmup loops.

Platform MPI 9.1.2 has the following bug fixes from Platform MPI 8.3:

- Enabled the **Allgather[v]_160** FCA algorithms by default.
- Fixed a ~1000 usec slowdown in FCA progression.
- Fixed a Windows issue in which ranks running on a local host will not use SHMEM to communicate if they were launched by different mpids.
- Improved **Alltoall** flooding on TCP by adding **Alltoall[v]_120** algorithms.
- Forced the **Alltoall[v]_120** algorithms. This fix encodes the new selection logic in this algorithm, rather than regenerating new data files.
- Improved error reporting for out of memory errors.
- Fixed the MPI-Log file in `/var/log/messages` during **MPI_Finalize** to report "start" and "end" job messages correctly.

- Added `parselog.pl` to be packaged for Linux to help you determine your MPI usage.
- Resolved an issue that occured during the initialization of InfiniBand with MPMD (multi-program multi-data) jobs.

Bug fixes in Platform MPI 9.1

Platform MPI 9.1 has the following bug fixes and improvements from Platform MPI 8.3:

- Stopped reading the `hpmpi.conf` file. Platform MPI now only reads `pmpi.conf` files.
- Stopped displaying debug information when using **`wlm`** features.
- Fixed the lazy deregistration issue.
- Fixed mpif77/mpif90 wrappers to work with gfortran syntax for auto-double.
- Fixed **`MPI_Finalize`** to include a progression thread delay.
- Fixed RDMA progression bug when running with Coalescing.
- Fixed a typo when using the **`MPE`** and **`jumpshot`** binaries.
- Updated FCA headers to be able to use FCA 2.2 libraries.
- Added a dynamic growth of internal SRQ buffer pools to provide better performance by reducing network flooding when using SRQ.
- Tuned some scheduled startup delays to improve startup times.
- Updated the hwloc module for CPU affinity features to hwloc 1.4.2.
- Improved the performance of the shared memory copy routines use for intra-node messages.

New or changed features in Platform MPI 9.1.2

Platform MPI 9.1.2 for Linux includes the following new or changed features.

New installer and installation instructions

ThePlatform MPI installer is now packaged using InstallAnywhere to provide a common installer for both Linux and Windows platforms. Therefore, the installation process is now changed to the same process for Linux and Windows. For more details, refer to "Installing Platform MPI" on page 54.

Event-based progression (Linux only)

Platform MPI for Linux includes a new execution mode designed to reduce CPU utilization and perform well in oversubscribed situations. Running with **`mpirun`** option `-nospin` will use the network to communicate between all process pairs (`-intra=nic`) and will block within the operating system when waiting on communication. This mode is particularly efficient for oversubscribed execution, threaded applications, and for applications that require **`MPI_THREAD_MULTIPLE`**. With the `-nospin` option, multiple threads will not compete for access to the MPI library, rather, multiple threads can be functioning within the MPI library simultaneously. Single-threaded applications can also run efficiently oversubscribed, and single-threaded applications will use less CPU and therefore less power when waiting on messages.

Note: When specifying `-nospin`, you must also specify `-e MPI_TCP_POLL=1`. This requirement will be removed in a future version.

Dynamic shared memory

The internal use of shared memory is now redesigned to allow Platform MPI to extend the amount of shared memory it uses. Previously, the amount of shared memory the Platform MPI could use during execution was fixed at MPI_Init time. The amount of shared memory available is now dynamic and can change during the application run. The direct benefit to users is that some collective algorithms optimized to make use of shared memory can now be used more often than before. This is particularly important for applications that create their own communicators as these user-created communicators will be able to use the best performing collective algorithms in the same way as the **MPI_COMM_WORLD** communicator. This change only impacts shared-memory used internally by Platform MPI. The prior tunables for controlling shared-memory visible to the user (for example, through MPI_Alloc_mem) remain unchanged.

Scale launching with DNS

Platform MPI includes improved scale launching in the presence of DNS. When an appfile or command line specifies the same host multiple times (or when cyclic rank placement is used), the Platform MPI startup process interacts with DNS to allow faster startup at scale. Setting the environment variable `PCMPI_CACHE_DNS=0` will turn off DNS caching and require a separate request each time a rank is launched on a host.

ISV licensing removed

ISV licensing is now removed from Platform MPI. Messages related to ISV licensing are no longer displayed.

CPU affinity and srun

CPU affinity (**mpirun** option `-aff`) now works with srun launching (**mpirun** option `-srun`)

File cache flushing integrated with CPU binding

Platform MPI has the capability to flush the file cache, and this is now integrated into the CPU binding. Use the MPI_FLUSH_FCACHE environment variable to modify the behavior of file cache flushing:

```
MPI_FLUSH_FCACHE = integer[,additional_options]
```

where the additional options are as follows:

full Clear the full calculated memory range. This overrides the default of stopping early if the swap space is low (the equivalent of **slimit:0**)

loc:*number*

Specifies when the file cache is flushed:

- 1: In mpid before ranks are created
- 2: As ranks are created. In Linux, this is after forking and before execution. In Windows, this is before **CreateProcess**.
- 3: During _init constructor invocation. In Windows, this option is convered to 4 (in MPI_Init).
- 4: In MPI_Init.

slimit:*size*
The value, in MB, where it will stop early if the swap space drops below this value. The default value is 1.25**chunk_size*, and the chunk size is usually 256MB.

split Each rank writes on a portion of memory. This overrides the default of writing by the first rank per host.

to:*size* The value, in MB, where it will stop early if the file cache size reaches the target. The default is 32.

v Enable verbose mode.

Alternate lazy deregistration

To improve performance of large messages over IBV, Platform MPI uses a feature on Linux called lazy deregistration. This features defers the deregistering (also known as unpinning or unlocking) of memory pages so that the pages do not need to be registered during a subsequent communication call using those same pages. However, if the pages of memory are released by the process, the lazy deregistration software must be made aware of this so that new pages located at the same virtual address are not incorrectly assumed to be pinned when they are not. To accomplish this, Platform MPI intercepts calls to **munmap** and disables negative **sbrk()** calls via **mallopt()**, which are the two primary methods that pages of memory are released by a processes.

In cases where an application makes its own **mallopt()** calls that would interfere with **mallopt()** settings for Platform MPI, or the application does not wish to disable negative **sbrk()** calls in the malloc library, an alternative mechanism is available. By using `-e MPI_DEREG_FREE=1`, Platform MPI will work with negative **sbrk()** by making the lazy deregistration system less aggressive. Turning on this setting automatically turns off **sbrk()** protection with Platform MPI that would otherwise be on.

Applications that allocate and release memory using mechanisms other than **munmap** or use of the **malloc** library must either turn off the lazy deregistration features (using `-ndd` on the **mpirun** command line) or they must invoke the following Platform MPI callback function whenever memory is released to the system in a way that Platform MPI does not track:

```
int  hpmp_dereg_freeunused_withregion(void* buf, size_t size, int flag);
```

The first argument is the start of the memory region being released and the size is the number of bytes in the region being released. The value of flag should be 0.

The most common example of when this is needed is the use of shared memory. When memory is released from a process using **shmdt()**, if any portion of this memory has been passed to a communicating MPI call, either `-ndd` must be used or the callback **hpmp_dereg_freeunused_withregion** must be called immediately before **shmdt()** is called.

Support for Torus-configured Infiniband networks

Platform MPI has support for Torus-configured Infiniband networks.

To enable Torus code, edit the `$MPI_ROOT/etc/pmpi.conf` file and uncomment the following line:

```
PCMPI_IB_DYNAMIC_SL = 1
```

This enables proper connections in a Torus configuration.

To confirm that correct SL/QoS levels are being set, obtain additional debug information by setting the following environment variable:

```
PCMPI_IB_SL_DUMP = 1
```

Intel MIC usage

Platform MPI supports either the MPI+offload or MPI+OpenMP+offload programming models. Intel MIC is the Intel "Many Integrated Core" Architecture. The Intel MIC offers an automatic offloading capability with source code pragma statements and the Intel Compilers.

Platform MPI does not support running MPI code on Intel MIC processors.

The application source code must be modified to include the MIC programming pragma statements. Without these pragma statements, the application will only be executed on the Intel Xeon processors.

1. Set the environment using the Intel compiler and Intel Openmp library and setting the MPI_ROOT environment variable.

   ```
   $ . /shared/intel/bin/compilervars.sh intel64
   $ export MPI_ROOT=/opt/ibm/platform_mpi
   ```

2. Compile the application. Because the source includes the Intel MIC pragmas, an `a.outMIC` executable will be automatically created by the compiler:

   ```
   $ ls
   hellomp.c
   $ $MPI_ROOT/bin/mpicc -openmp -o hellomp hellomp.c
   $ ls
   hellomp  hellomp.c  hellompMIC
   $ file hellomp hellompMIC
   hellomp:    ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses
   shared libs), for GNU/Linux 2.6.18, not stripped
   hellompMIC: ELF 64-bit LSB shared object, version 1 (SYSV), dynamically linked, not stripped
   ```

The application will be run on the Intel MIC cores, one thread on each MIC execution unit. Launching a single rank with Platform MPI, without setting **OMP_NUM_THREADS**, the execution will include one thread per Intel MIC execution unit. In this case, there are 52 MIC processors, with 4 execution units each, with one MIC reserved for job management will produce 204 threads: (52-1)*4=204.

```
$ mpirun -np 1 ./hellomp
Hello from thread 192 out of 204 from rank 0 out of 1 on intel1.
Hello from thread 193 out of 204 from rank 0 out of 1 on intel1.
...
...
...
Hello from thread 106 out of 204 from rank 0 out of 1 on intel1.
```

Command line aliasing

Platform MPI allows aliases to be created for common **mpirun** command line arguments, options, and environment variables. The aliases must be predefined in a file, and can be used as a shorthand for complex command line options or for application-specific tuning options.

The general format of an alias definition is as follows:

```
ALIAS alias_name {
    -option1 -option2
    # Comments are permitted
```

```
    -e MYVAR=val
    -e MYPATH=${PATH}
    # Linux path
    -e MYDIR="/tmp"
    # Windows path
    -e MYWINDIR="C:\Application Data\Temp"
}
```

The ALIAS keyword must be all caps, and is followed by the alias name. The alias definition is contained in matching curly braces { and }. Any valid **mpirun** command line option can be included inside an alias definition. Quoted strings are permitted, but must be contained on a single line. All tokens are whitespace-delimited.

To use a pre-defined alias on the **mpirun** command line, the `-cmd` syntax is:
`-cmd=alias1[,alias2,...]`

More than one alias can be included in a comma-separated list. The `-cmd` option may be included more the one time on the **mpirun** command line. The aliases are expanded, in place, before any other command line parsing is done. An alias may only be listed, or expanded, onto a command line one time. A second use of an alias on a single command line will result in an error.

Environment variable values can be expanded from the shell environment where the **mpirun** command is run. Note that the runtime environment may be the same local node where the **mpirun** command is issued, or on a remote node if a job scheduler is used.

Alias files are read from three locations, in the following order depending on the operating system:

- Linux:
  1. `$MPI_ROOT/etc/pmpi.alias`
  2. `/etc/pmpi.alias`
  3. `$HOME/.pmpi.alias`
- Windows:
  1. `"%MPI_ROOT%\etc\pmpi.alias"`
  2. `"%ALLUSERSPROFILE%\Application Data\IBM\Platform Computing\Platform-MPI\pmpi.alias"`
  3. `"%USERPROFILE%\Application Data\IBM\Platform Computing\Platform-MPI\pmpi.alias"`

All three alias files are read, if they exist. The alias names are matched in reverse order, from last to first. This allows alias names to be redefined in later files.

Setting **PCMPI_ALIAS_VERBOSE=1** will print the fully-expanded command line as tokens. This option must be set in the environment when **mpirun** is invoked to work.

```
#------------------------------------------------------------------
#  Example pmpi.alias
#------------------------------------------------------------------
#
# Set some common debugging options and environment variables
#
ALIAS debug {
 -v -prot
 -e MPI_AFF_VERBOSE=1
```

```
 -e PCMPI_CPU_DEBUG=1
 -e MPI_NUMA_VERBOSE=1
 -e MPI_COLL_FCA_VERBOSE=1
}

#
# Setup some increasing levels of debugging output
#

# Debug level 1
ALIAS debug1 {
 -cmd=debug
}

# Debug level 2
ALIAS debug2 {
 -cmd=debug1
 -e MPI_FLAGS=v,D,l,Eon
 -e MPI_COLL_FORCE_ALL_FAILSAFE=1
}
#--------------------------------------------------------------------
```

GPU message copy improvements

Platform MPI now supports using CUDA IPC to improve shared memory communication performance. Enabling this feature improves MPI performance when using GPU memory for MPI messages by enabling a DMA copy from GPU memory to GPU memory when possible.

To enable, use both `-e PMPI_CUDAIPC_ENABLE=1` and the old **gpudirect** option `-e PMPI_GPU_AWARE=1` to turn on the feature. The `-e PMPI_GPU_AWARE=1` option will only enable the GPU Direct support and not enable the CUDA IPC copy features.

TCP Alltoall flooding algorithm

When called using TCP networking protocol, Alltoall has the potential to flood the TCP network, which can seriously degrade performance of all TCP traffic. To alleviate TCP flooding, Platform MPI 9.1 introduced a new Alltoall algorithm that is called specifically when TCP is used. This algorithm will help prevent flooding but does have the effect of slightly degraded performance for Alltoall. This should only occur when using TCP. If you wish to use the older algorithms to obtain better performance and are sure your application will not flood the TCP network, include the following environment variable for the MPI run: `-e MPI_COLL_IGNORE_ALLTOALL=120`.

RDMA to TCP failover

This release allows support for network failover from the IBV protocol to TCP. This option is intended to allow failover from IBV on an Infiniband network to TCP on a separate Ethernet network. Failover from IBV to TCP-over-IB on the same physical network is possible; however, a network failure will likely cause both protocols to be unusable, making the failover feature to be of no benefit. To enable this feature, run with `-ha:detect` and set the **PCMPI_IBV2TCP_FAILOVER** environment variable. Using this feature will sometimes result in higher communication latency as additional overhead is required to detect failures and record information necessary to retransmit messages on the TCP network.

In this release, there is no ability to transition back to IBV if the Infiniband network is restored. Once a connection has transitioned to TCP, it will continue to

use TCP for the remainder of the execution. A future release may allow an administrator to signal to the application that the IBV network failures have been addressed.

MPI 3.1 High Availability features

In an effort to support current practices and standardization efforts for high availability support, the fault tolerance model for `-ha:recover` has changed in this release. The previous functionality as described in the current User's Guide Appendix B under the sections *Failure recovery (-ha:recover)* and *Clarification of the functionality of completion routines in high availability mode* has changed. This release fully implements the MPI Forum Process Fault Tolerance Proposal. This proposal is not part of the MPI Standard at this time, but is being considered for inclusion in the next version of the MPI Standard. Users of previous versions of Platform MPI will find that migration to the new approach to HA is straightforward. The previous approach to recovery redefined **`MPI_Comm_dup`**. The new approach provides the same functionality using new routine names **`MPI_Comm_revoke`** and **`MPI_Comm_shrink`**. There are also some differences in when errors are reported, how **`MPI_ANY_SOURCE`** is handled, and the return code used to designate that a process failure occurred. In addition, the proposal also adds a utility function for consensus (**`MPI_Comm_agree`**).

The Process Fault Tolerance Proposal includes the following new routines:

- **`MPI_Comm_revoke`**
- **`MPI_Comm_shrink`**
- **`MPI_Comm_agree`**
- **`MPI_Comm_iagree`**
- **`MPI_Comm_failure_ack`**
- **`MPI_Comm_failure_get_acked`**

Because these are not officially part of the MPI Standard at this time, they are named as follows in this release:

- **`MPIHA_Comm_revoke`**
- **`MPIHA_Comm_shrink`**
- **`MPIHA_Comm_agree`**
- **`MPIHA_Comm_iagree`**
- **`MPIHA_Comm_failure_ack`**
- **`MPIHA_Comm_failure_get_acked`**

Because recovery is now handled by new routine names rather than through the pre-existing **`MPI_Comm_dup`**, **`-ha:detect`** and **`-ha:recover`** are now functionally identical.

MPI 3.0 non-blocking collective support/preview

Platform MPI has preliminary support for non-blocking collectives on Linux. Support is for the "c" interface, single-threaded, non-HA and contiguous data types only. In order to use them, add the `-DNON_BLOCKING_COLLECTIVES` compiler option.

Prototypes are provided in `mpi.h`

SR-IOV

Platform MPI includes the ability to support Single-Root I/O Virtualization (SR-IOV) for IBV connection. Without any modification, users can run MPI applications across SR-IOV supported virtual machines. The application performance will depend on different hardware and virtual machine configuration.

When using SR-IOV on virtual machines as your IBV connection, performance will degrade compared to using IBV directly on native hardware. PingPong latency performance is 6-8x slower, while bandwidth is 2-3x slower for middle message sizes. As messages increase in size, performance starts approaching actual hardware performance. For collectives, performance difference range from a slight difference to 2x slower. Each collective and message size has different characteristics. To determine your deltas, perform comparisons for your specific hardware and VMs.

Multi-threaded collective performance improvements

Platform MPI features enhancements to the performance of collective algorithms in the multi-threaded library. Prior to this release, the multi-threaded library supported multi-threaded application code which executed MPI collective calls simultaneously by different threads on the same communicator. This is explicitly not allowed by the MPI Standard, but is handled correctly by Platform MPI. If an application relies on Platform MPI's ability to support simultaneous collective calls on the same communicator, the application must now specifically request this support by setting **PMPI_STRICT_LOCKDOWN**. The new default behavior allows for faster and more scalable collective performance by allowing some collective calls to avoid an expensive distributed lockdown protocol formally required by each collective call.

It is important for users of the multi-threaded library to understand that the libcoll infrastructure selection is based on the relative performance of algorithms in the single-threaded library. Therefore, it is especially important that multi-threaded library users create their own benchmarking tables which reflect the performance of collectives called by the multi-threaded library. The libcoll library selection process will demote some algorithms when running with the multi-threaded library which are known to require a distributed lockdown step and therefore tend to perform poorly when called from the multi-threaded library. However, best results will be achieved when a full benchmark table is produced directly from the multi-threaded library.

As part of this redesign to allow significantly better overall multi-threaded collective performance, some collective operations, particularly communicator creation routines, now perform multiple distributed lockdown operations. As a result, the performance of some of these routines, which most users do not consider performance sensitive, may be slightly poorer than in previous releases. This known regression will be addressed in a future release.

On-demand connections for SRQ/TCP

Platform MPI includes the ability to enable on-demand connections for IBV, IBV/SRQ and TCP. On-demand support for IBV was added in Platform MPI 8.1. For Platform MPI 9.1, on-demand support for TCP and IBV/SRQ is included.

To enable on-demand connections, set the environment variable **PCMP_ONDEMAND_CONN=1**. This will enable IBV, IBV/SRQ or TCP connections between two ranks in an MPI run only if the ranks communicate with each other. If two

ranks do not send messages to each other, the connection is never established, saving the resources necessary to connect these ranks. If an application does not use collectives, and not all the ranks send messages to other ranks, this could enable performance gains in startup, teardown and resource usage.

Scale improvements to 128K ranks support

Previous versions of Platform MPI had some possible barriers at 32K ranks that have been eliminated over time. Platform MPI is now designed to fully support 128K rank runs.

Currently there are no known limits to scale Platform MPI beyond 128K ranks, but some additional work may be needed to better use system resources process management and network connections. If you want to scale beyond 128K and run into issues, contact IBM for assistance on run-time tunables that can be used to scale to larger rank counts.

PSM -intra=mix mode

The `-intra=mix` mode is only supported for some interconnects: Infiniband IBV and PSM. In this mode, messages less than or equal to a certain threshold are sent using MPI shared memory and messages above a certain size are sent using the interconnect. The default threshold varies but is 2k for PSM, and can be controlled using the **`MPI_RDMA_INTRALEN`** setting in bytes.

XRC multi-card

Support added to XRC protocol striping data over multiple cards. No new command line options are added to support this feature. Existing multicard options (such as the `MPI_IB_STRINGS` option) should be used to define multicard options when using XRC. With the addition of this support XRC Multicard support will be enabled by default if more than one card is detected on the system and XRC protocol is used. Use existing multi-card options to restrict XRC traffic to one card if necessary.

PE-POE startup support

Platform MPI now supports the IBM PE-POE launching utility. To launch Platform MPI applications using the PE/POE launching utility, include the `-poe` option on the **`mpirun`** command line:

```
mpirun mpirun-args -poe poe-args
```

For example,

```
mpirun -prot -poe program.x -hfile hosts -procs 16
```

Running under the pdb debugger is supported using `-pdb` on the command line.

For example,

```
mpirun mpirun-args -pdb poe-args
```

CPU affinity features for Platform MPI 9.1

CPU affinity involves setting what CPU or mask of CPUs each rank in which an MPI job will run.

To aid in explaining each affinity concept, the following two example machines will be used for most of the examples: Both example machines have two sockets, each containing two NUMA nodes, where each NUMA node contains four cores. One machine (example-host-1) has hyperthreading turned off, so each core contains one hyperthread, while the other (example-host-2) has hyperthreading turned on, so each core contains two hyperthreads.

Table 4. Representation of machine with hyperthreading off

example-host-1															
socket								socket							
numa				numa				numa				numa			
c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c
h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h

Table 5. Representation of machine with hyperthreading on

example-host-2																											
socket												socket															
numa								numa								numa								numa			
c		c		c		c		c		c		c		c		c		c		c		c		c		c	
h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h

Each hyperthread ("h") in these machines has an associated number assigned by the operating system, and bitmasks are used to identify what set of hyperthreads a process can run on. The pattern by which numbers are assigned to the hyperthreads can vary greatly from one machine to another.

The MPI notation used in "verbose" mode will be described in more detail later, but these example machines might be displayed as

```
[0 2 4 6,8 10 12 14],[1 3 5 7,9 11 13 15]
```

and

```
[0+16 2+18 4+20 6+22,8+24 10+26 12+28 14+30],[1+17 3+19 5+21 7+23,9+25 11+27 13+29 15+31]
```

which shows the hardware in logical order visually identifying the sockets, NUMA nodes, cores, and hyperthreads and shows the number associated with each hyperthread.

If an MPI rank were to be assigned to the first core of example-host-2 that would correspond to hyperthreads 0 and 16, which, when expressed as a bitmask, would be `1<<0 + 1<<16 or 0x10001`. The MPI notation used in verbose mode to display this bindings would be

```
[11 00 00 00,00 00 00 00],[00 00 00 00,00 00 00 00]  : 0x10001
```

Alternatively, to assign a rank to the whole first NUMA of example-host-1 that would correspond to hyperthreads 0, 2, 4, and 6, would be bitmask `0x55`. The MPI verbose display for this binding would be

```
[1 1 1 1,0 0 0 0],[0 0 0 0,0 0 0 0]  : 0x55
```

The main binding options (that is, the categories of affinity selection) can be organised into the following three groups:

automatic pattern-based
Automatic pattern-based bindings are based on the concept of placing blocks of ranks and then cycling between topology elements for the next block of rank assignments.

manual
Manual masks involve specifying hex masks for each rank. This offers great flexibility if the hardware being run on is known.

the `hwloc_distribute()` function
The **`hwloc_distribute()`** function resembles the pattern-based options but is less rigid. It divides the available processing units more or less evenly among the ranks.

Main binding options for automatic pattern-based binding

The following pieces of information are used to define the pattern:

- What topology elements to cycle when cycling occurs (`-affcycle`)
- What size mask to assign for an individual rank (`-affwidth`)
- How many contiguous topology elements to assign in a block before explicitly triggering cycling (`-affblock`)
- How many consecutive ranks to assign the exact same binding before stepping to the next contiguous topology element (`-affstep`)

The following examples can clarify each individual concept:

Table 6. -affcycle examples (all with width=core, block=1, step=1)

example-host-1																	
socket								socket									
numa				numa				numa				numa					
c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c		
h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h		
0	4			1	5			2	6			3	7			**`-affcycle=numa`**	**`-np 8`**
0	2	4	6	8				1	3	5	7	9				**`-affcycle=socket`**	**`-np 10`**
0	8	1	9	2		3		4		5		6		7		**`-affcycle=2core`**	**`-np 10`**
0	2	4	6	8				1	3	5	7	9				**`-affcycle=2numa`**	**`-np 10`**
0	4			2				1				3				**`-affcycle=numa,socket`**	**`-np 5`**

Table 7. -affwidth examples (all with cycle=numa, block=1, step=1)

example-host-1																	
socket								socket									
numa				numa				numa				numa					
c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c		
h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h		
0	4			1	5			2	6			3	7			**`-affwidth=core`**	**`-np 8`**
0	0	4	4	1	1	5	5	2	2	6	6	3	3	7	7	**`-affwidth=2core`**	**`-np 8`**
0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3	**`-affwidth=numa`**	**`-np 4`**
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	**`-affcycle=socket`**	**`-np 2`**

For the above example, having cycle set to something smaller or equal to the width effectively disables the explicit cyclng.

Table 8. -affblock examples (all with cycle=numa, width=core, step=1)

example-host-1																	
socket								socket									
numa				numa				numa				numa					
c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c		
h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h		
0	4			1	5			2	6			3	7			**-affblock=1**	**-np 8**
0	1			2	3			4	5			6	7			**-affblock=2**	**-np 8**

Table 9. -affstep examples (all with cycle=numa, width=core, block=1)

example-host-1																	
socket								socket									
numa				numa				numa				numa					
c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c		
h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h		
0	4			1	5			2	6			3	7			**-affstep=1**	**-np 8**
0	8			2				4				6				**-affstep=2**	**-np 9**
1				3				5				7				(this example spans two lines)	

The command line options controlling the above patterns are shown below. In the following descriptions, # is a positive non-zero integer, and *unit* can be `socket`, `numa`, `L2`, `core`, or `execunit`:

-affcycle=#*unit*,#*unit*,...
: What topology elements to use when cycling.

-affcycle=all
: `all` is a keyword to cycle through all topology elements.

-affcycle=none
: `none` is a keyword that results in packed contiguous bindings. This is the default for `-affcycle`.

-affwidth=#*unit*
: Size of mask for an individual rank. The default value is `-affwidth=core`.

-affblock=#
: Contiguous assignments before cycling. The default value is `-affblock=1`

-affstep=#
: Consecutive ranks to receive the same binding. The default value is `-affstep=1`.

Main binding options for manual masks

The following are main binding options for manual masks:

-affmanual=0x*hex***:0x***hex***:0x***hex***
: The ranks for each host are assigned masks from the list. By default, the index of the mask is the host-local rank ID (cyclically if the number of host-local ranks is larger than the number of masks specified in the list).

-affmanual=seq

`seq` is a keyword that expands into all the bits on the machine in sequence. For example, 0x1:0x2:0x4:0x8:0x10:...

-affopt=global

This option changes the decision of which mask in the list goes to which rank. With this option, the list is indexed by global rank ID (cyclically).

The following section on *Printing CPU masks or interpreting manual masks* shows options for varying the interpretation of the hexadecimal masks. The default is that the bits represent hyperthreads using the operating system ordering.

Main binding options for hwloc_distribute()

The following are main binding options for the **`hwloc_distribute()`** function:

-affdistrib=socket, numa, core, L2, execunit, or explicit depth #

Uses the built-in **`hwloc_distribute()`** function from the HWLOC project to place the ranks. Generally speaking this function spreads the ranks out over the whole machine so things like cache and memory bandwidth will be fully used while putting consecutive ranks on nearby resources.

The input argument specifies the smallest size unit an individual rank is to be given.

-affopt=single

The masks returned from hwloc_distribute() are often large, allowing ranks to drift more than might be ideal. With this option each rank's mask is shrunk to a single core.

Reordering automatically-generated masks

For getting the best overall bandwidth it is often good to span all the resources in the hardware tree, which can be done using the -affcycle=all option, but the result does not put consecutive ranks near each other in the topology.

For example,

Table 10. -affblock examples (all with cycle=numa, width=core, step=1)

example-host-1																	
socket								socket									
numa				numa				numa				numa					
c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c		
h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h		
0	4			2				1	5			3				**`-affcycle=numa,socket`**	**`-np 6`**
0	1			2				3	4			5				**`-affopt=reorder`**	**`-np 6`**

The reorder option takes the existing set of bindings and sorts them logically so consecutive rank IDs are as near each other as possible:

```
-affopt=reorder
```

What to do when the produced pattern results in oversubscription

Sometimes when a binding cycles through the whole machine, it can result in oversubscription.

For example, when using **-affwidth=2core -np 5**:

```
- R0: [11 11 00 00],[00 00 00 00]  : 0x00000505
- R1: [00 00 11 11],[00 00 00 00]  : 0x00005050
- R2: [00 00 00 00],[11 11 00 00]  : 0x00000a0a
- R3: [00 00 00 00],[00 00 11 11]  : 0x0000a0a0
- R4: [11 11 00 00],[00 00 00 00]  : 0x00000505
```

Oversubscription has a substantial enough penalty that the default in this case is to partially unbind every rank onto the whole of any NUMA node it occupies:

```
- R0: [11 11 11 11],[00 00 00 00]  : 0x00005555
- R1: [11 11 11 11],[00 00 00 00]  : 0x00005555
- R2: [00 00 00 00],[11 11 11 11]  : 0x0000aaaa
- R3: [00 00 00 00],[11 11 11 11]  : 0x0000aaaa
- R4: [11 11 11 11],[00 00 00 00]  : 0x00005555
```

The options for the behavior in the presence of such oversubscription are as follows:

-affoversub=ok
: Accept the binding as-is.

-affoversub=unbind
: Fully unbind, expanding mask to whole machine.

-affoversub=partial
: Partial unbind, expanding mask to NUMA node.

Printing CPU masks or interpreting manual masks

-affopt=osindex
: The bits in the mask represent PUs (hyperthreads) using the operating system ordering. This is the default value.

-affopt=logicalindex
: The bits in the mask still represent PUs but are ordered by logical_index, which is the intuitive order if the topology were drawn in a tree so neighbors in the tree have consecutive bits.

-affopt=coreindex
: This is similar to logicalindex but each bit represents a core instead of a PU. This option can be better than logicalindex if working with machines where some have hyperthreading on and some off.

For these options, a smaller example machine will be used, with two sockets, four cores per socket, and two hyperthreads per core.

Using the operating system index (default) to label each PU (hyperthread):

socket				**socket**				**socket**				**socket**			
core		core		core		core		core		core		core		core	
0	8	2	10	4	12	6	14	1	9	3	11	5	13	7	15

Under this labeling system, a mask containing the second core on the first socket would have bits 2 and 10 set giving binary 0100,0000,0100 or hexadecimal 0x404.

If the affinity option `-affopt=logicalindex` were used with this same machine, the numbering of the hyperthreads would instead be the following:

socket				**socket**				**socket**				**socket**			
core		core		core		core		core		core		core		core	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

This would make the hexadecimal labeling for the second core on the first socket contain bit 2 and bit 3, giving binary 1100 or hexadecimal 0xc.

When using `-affopt=coreindex`, the indexes represent the cores and the hyperthreads under them are ignored, so this machine would then be labeled as follows:

socket				**socket**			
core	core	core	core	core	core	core	core
0	1	2	3	4	5	6	7

Inheriting existing binding or attempting to break out

-affopt=inherit
Bind within the inherited mask. This is the default value.

-affopt=inherit:full
Special mode, run in inherited mask unmodified.

-affopt=inherit:seq
Special mode, portion out the bits in the inherited mask consecutively to the ranks.

-affopt=noinherit
Attempt breaking out of the inherited binding to use the whole machine

Skipping affinity for select ranks

For the pattern-based bindings it may be convenient to have some ranks not included in the binding. The options for this are as follows:

MPI_AFF_SKIP_GRANK
Accepts a comma-separated list of global ranks.

MPI_AFF_SKIP_LRANK
Accepts a comma-separated list of host-local ranks.

For example, if you want to use regular bandwidth-binding on example-host-1, but you are creating an extra rank on each host which you wish to be unbound. To keep that rank from messing up the binding for the other ranks, run a command such as the following:

```
mpirun -affcycle=all -e MPI_AFF_SKIP_LRANK=0 ...
```

This would result in binding the host:

example-host-1															
socket								socket							
numa				numa				numa				numa			
c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c
h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	5	9	13	3	7	11	15	2	6	10	14	4	8	12	16

The ranks 1-16 were bound without being disrupted by rank 0, which is unbound.

Verbose options

```
-affopt=v | vv
```

If the verbose option "vv" is specified, information about the host is displayed in a visual format, where

- Brackets [] surround each socket
- A comma , separates NUMA nodes
- Hyperthreads on a core are adjacent

For example, a host with two sockets, each of which is a NUMA node and four cores per socket, with hyperthreading enabled might look like the following:

```
> Host 0 -- ip 127.0.0.1 -- [0+8 2+10 4+12 6+14],[1+9 3+11 5+13 7+15]
> - R0: [11 00 00 00],[00 00 00 00]  : 0x00000101
> - R1: [00 11 00 00],[00 00 00 00]  : 0x00000404
> - R2: [00 00 11 00],[00 00 00 00]  : 0x00001010
> - R3: [00 00 00 00],[11 00 00 00]  : 0x00000202
> - R4: [00 00 00 00],[00 11 00 00]  : 0x00000808
> - R5: [00 00 00 00],[00 00 11 00]  : 0x00002020
```

In this example, six ranks have been bound with **-affcycle=all -affopt=reorder** to get the best overall bandwidth.

If the verbose option "v" is specified, the display is abbreviated as follows:

- An empty socket becomes [--] or [----] if it contains multiple NUMA nodes.
- An empty NUMA node becomes -- or ---- if it contains multiple sockets
- Full sockets/NUMAs become the corresponding elements [##]

Under this scheme, the previous example would become the following:

```
> Host 0 -- ip 127.0.0.1 -- [0+8 2+10 4+12 6+14],[1+9 3+11 5+13 7+15]
> - R0: [11 00 00 00],--  : 0x00000101
> - R1: [00 11 00 00],--  : 0x00000404
> - R2: [00 00 11 00],--  : 0x00001010
> - R3: --,[11 00 00 00]  : 0x00000202
> - R4: --,[00 11 00 00]  : 0x00000808
> - R5: --,[00 00 11 00]  : 0x00002020
```

Other examples of abbreviated displays are as follows:

```
[11 11 11,00 00 00],[00 00 00,00 00 00],[00 00 00,00 00 00]
```

is abbreviated to

```
[##,--],[----],[----]
```

```
[00 11 11][00 00 00],[00 00 00][00 00 00],[11 11 11][11 11 11]
```

is abbreviated to

```
[00 11 11][--],----,####
```

Abbreviations

The `-aff` option is unnecessary in the new model, but it can still be used for backward compatibility and to supply convenient abbreviations:

-aff is short for `-affcycle=all -affopt=reorder`

-aff=bandwidth
is short for `-affcycle=all -affopt=reorder`

-aff=latency
is short for `-affcycle=none`

-aff=manual:...
is the same as `-affmanual=...`

The abbreviations are overridden if more specific options are used instead. Other options such as `-aff=automatic:bandwidth` or `-aff=automatic:latency` also still work and are equivalent to the previous options.

Cluster test tools

Platform MPI provides a pre-built program **mpitool** which runs as part of the system check feature and includes a variety of programs it can run based on command line arguments.

For example,

```
$MPI_ROOT/bin/mpirun -np 4 $MPI_ROOT/bin/mpitool -hw
```

This runs four ranks of the **hello-world** program

```
$MPI_ROOT/bin/mpirun -hostfile hosts $MPI_ROOT/bin/mpitool -ppr 1024
```

This runs the **ping-pong-ring** program with message size 1024 over the hosts in the `hosts` file.

Options

The full list of options provided in the **mpitool** utility is as follows:

- Basic:

  -hw hello world

  -ppr ping-pong ring

  -alltoallseq
  slow sequential alltoall test

- Tuning:

  -bm collective benchmarking

- Utilities:

  -replicate
  copies files and directories with MPI

-run runs a command on every host

- Cluster testing:

  -pphr host-level ping-pong ring (produces a two-dimensional graph)

  -flood host-level flooding test (produces a two-dimensional graph)

  -allpairs
  ping-pong on all host pairs (produces a three-dimensional graph)

  -ringstress
  network stress test

Examples

Examples to use these options are as follows:

- hello world (**-hw**)

  Runs the hello-world program at each rank.

  For example,

```
$MPI_ROOT/bin/mpirun -hostlist hostA:2,hostB:2 $MPI_ROOT/bin/mpitool -hw

> Hello world! I'm 1 of 4 on hostA
> Hello world! I'm 3 of 4 on hostB
> Hello world! I'm 0 of 4 on hostA
> Hello world! I'm 2 of 4 on hostB
```

- ping-pong ring (**-ppr**)

  Runs the ping-pong ring program over all ranks, which involves a ping-pong between each pair of adjacent ranks using the natural ring ordering, one pair at a time. The program accepts one argument which specifies the number of bytes to use in each ping-pong.

  For example,

```
$MPI_ROOT/bin/mpirun -hostlist hostA:2,hostB:2 $MPI_ROOT/bin/mpitool -ppr 10000

> [0:hostA] ping-pong 10000 bytes ...
> 10000 bytes: 2.45 usec/msg
> 10000 bytes: 4089.35 MB/sec
> [1:hostA] ping-pong 10000 bytes ...
> 10000 bytes: 10.42 usec/msg
> 10000 bytes: 960.06 MB/sec
> [2:hostB] ping-pong 10000 bytes ...
> 10000 bytes: 2.66 usec/msg
> 10000 bytes: 3759.05 MB/sec
> [3:hostB] ping-pong 10000 bytes ...
> 10000 bytes: 10.31 usec/msg
> 10000 bytes: 969.72 MB/sec
```

- sequential alltoall (**-alltoallseq**)

  This test provides two very slow alltoall routines where a single message traverses all rank pairs one by one. The selection of which of the two all-to-all tests to run and what message size to use are controlled by the command line:

  – **-nbytes #**

  – **-sequential**: each rank in turn does ping-pong with all

  – **-bounce**: a single message bounces until complete

  In the `-sequential` mode each rank acts as leader for one iteration during which time it sends and recvs a message with every other rank, ending with its right-neighbor who becomes the next leader. An example of the message order is as follows:

```
(0 is initially the leader)
0 sends to 2, and 2 sends back to 0
0 sends to 3, and 3 sends back to 0
0 sends to 1, and 1 sends back to 0
(1 is now the leader)
1 sends to 3, and 3 sends back to 1
1 sends to 0, and 0 sends back to 1
1 sends to 2, and 2 sends back to 1
(2 is now the leader)
2 sends to 0, and 0 sends back to 2
2 sends to 1, and 1 sends back to 2
2 sends to 3, and 3 sends back to 2
(3 is now the leader)
3 sends to 1, and 1 sends back to 3
3 sends to 2, and 2 sends back to 3
3 sends to 0, and 0 sends back to 3
```

The `-sequential` version actually sends twice the messages of a plain alltoall because each leader does ping-pong with all peers. The `-sequential` version also prints progress reports after each leader finishes its batch of messages.

In the `-bounce` mode rank 0 sends a single message to its right neighbor and then each rank sends to whichever peer it has not yet sent to (starting with its right neighbor and iterating). The one message bounces around the system until all pairs have been traversed.

For example, `0 -> 1 -> 2 -> 3 -> 0 -> 2 -> 0 -> 3 -> 1 -> 3 -> 2 -> 1 -> 0`

This can be shown to always complete at all ranks by induction, noting that if some rank has completed then due to the order the messages are sent in at each rank its right neighbor has also just completed.

The `-bounce` version only prints a single time at the end since it is harder to have meaningful progress reports in the middle.

For example (using `-sequential`, which is the default):

```
$MPI_ROOT/bin/mpirun -hostlist hostA:2,hostB:2 $MPI_ROOT/bin/mpitool -alltoallseq -nbytes 4000

> Rank 0 completed iteration as leader: 0.10 ms
> Rank 1 completed iteration as leader: 0.12 ms
> Rank 2 completed iteration as leader: 0.08 ms
> Rank 3 completed iteration as leader: 0.04 ms
> total time 0.00 sec
```

- collective benchmarking (`-bm`)

  This test is largely orthogonal to the other tests in **mpitool** and is described in more detail in the section on collective benchmarking. System Check can run an optional benchmark of selected internal collective algorithms. This benchmarking allows the selection of internal collective algorithms during the actual application runtime to be tailored to the specific runtime cluster environment.

- copy files with (`-replicate`)

  This is a utility that allows Platform MPI to be used to copy files across a cluster. It is most useful on systems lacking a cluster file system. This utility allows much faster copying than **scp** for example because it can use whatever high speed networks and protocols Platform MPI has, and it uses **MPI_Bcast** to send the data to the whole cluster at once.

  This program uses rank-0 as the source and copies to every other host that ranks are launched onto.

  For example, if there is a file called `hosts` with the following contents:

  ```
  hostA
  hostB
  hostC
  hostD
  ```

```
hostE
hostF
hostG
hostH
```

Run the following command:

```
$MPI_ROOT/bin/mpirun -hostfile hosts $MPI_ROOT/bin/mpitool -replicate /tmp/some_application

> - so far: 540.00 Mb at 170.23 Mb/sec
> - so far: 1071.00 Mb at 173.50 Mb/sec
> Total transfer: 1206.07 Mb at 179.36 Mb/sec (to each destination)
> (time 6.72 sec)
```

The main argument is the directory to be copied. The other available arguments are as follows:

-rmfirst
Erase `/path/to/directory_or_file` on each remote host before copying. By default the untar command on the remote hosts won't erase anything that's already there so this is needed if the remote directories need to be cleaned up.

-show Display the exact command that would be executed on each host but do not run anything.

-z Use the z option (compression) to tar. This is generally not recommended since it tends to be much slower, but the option is provided since results might vary from cluster to cluster.

- run a command on every host (`-run`)

  This is a utility that allows Platform MPI to be used to launch arbitrary commands across a cluster. The output is either collected to files or printed to stdout.

  Use `-run` in the following ways:

  ```
  -run [-name name] command and args

  -run -cmd name "command and args" [-cmd name2 ...]
  ```

  In the first syntax if "-" is provided as the name, stdout is used instead of sending to a file. If any other string is provided as the name, the output goes to files `out.<name>.d.<hostname>`written by rank 0.

  For example, if there is a file called `hosts` with the following contents:

  ```
  hostA
  hostB
  hostC
  hostD
  hostE
  hostF
  hostG
  hostH
  ```

  Run the following command:

```
$MPI_ROOT/bin/mpirun -hostfile hosts $MPI_ROOT/bin/mpitool -run -name - uptime

> -------- [0] hostA --------------------------------
>  18:38:02 up 29 days,  2:27,  0 users,  load average: 0.01, 0.03, 0.13
> -------- [1] hostB --------------------------------
>  18:38:02 up 42 days, 14:51,  0 users,  load average: 0.00, 0.03, 0.10
> -------- [2] hostC --------------------------------
>  18:38:02 up 42 days, 14:53,  0 users,  load average: 0.00, 0.03, 0.11
> -------- [3] hostD --------------------------------
>  18:38:02 up 42 days, 14:52,  2 users,  load average: 0.08, 0.03, 0.07
> -------- [4] hostE --------------------------------
>  18:38:02 up 27 days,  2:56,  0 users,  load average: 0.01, 0.02, 0.07
> -------- [5] hostF --------------------------------
```

```
>  18:38:02 up 27 days,  2:55,  0 users,  load average: 0.00, 0.02, 0.07
> -------- [6] hostG -------------------------------
>  18:38:02 up 42 days, 14:58,  0 users,  load average: 0.00, 0.05, 0.12
> -------- [7] hostH -------------------------------
>  18:38:02 up 42 days, 14:15,  0 users,  load average: 0.08, 0.04, 0.08
$MPI_ROOT/bin/mpirun -hostfile hosts $MPI_ROOT/bin/mpitool -run -name misc cat /proc/cpuinfo \| grep \''cpu MHz'\'
```

this produces output that resembles the following

```
> cpu MHz         : 2266.830
> cpu MHz         : 2266.830
> cpu MHz         : 2266.830
> cpu MHz         : 2266.830
> cpu MHz         : 2266.830
> cpu MHz         : 2266.830
> cpu MHz         : 2266.830
> cpu MHz         : 2266.830
```

in each of several files similar to `out.misc.00000003.hostD`.

In the above example,

1. The **mpirun** line has as part of its command line arguments: `cat /proc/cpuinfo \| grep \''cpu MHz'\'`
2. The local shell parses some of this leaving **mpirun** and the subsequent mpitool with argv[] entries as

   ```
   argv[i+0] = cat
   argv[i+1] = /proc/cpuinfo
   argv[i+2] = |
   argv[i+3] = grep
   argv[i+4] = 'cpu MHz'
   ```

3. From this, it constructs the following string:

   ```
   cat /proc/cpuinfo | grep 'cpu MHz'
   ```

   which is given (via **system()**) to the shell on the remote host.

In general simpler commands are better, but knowing the levels of parsing allows more complex commands to be used.

In addition, only one command is run per host. If more than one rank had been run on the same host that would have been detected and only one of the ranks would have run a command while the others would have been idle.

- host-level ping-pong ring (`-pphr`)

  This test is conceptually similar to the simpler ping-pong-ring, but is performed on a per-host basis and involves ping-pong between multiple ranks on one host with multiple peer ranks on the neighbor host. The hosts are ordered in a natural ring and the results are shown in a two-dimensional graph where host indexes are on the x-axis and bandwidths are on the y-axis. When multiple ranks are present on each host, multiple lines are graphed for tests with varying numbers of ranks participating per host. The program takes three optional integers on the command line to specify:

  1. The number of bytes to use in the ping-pong messages
  2. How many iterations per timed inner loop (default 1000)
  3. How many times to collect in the outer loop (default 5)

  The only reason for the inner or outer loop as opposed to just timing 5000 iterations is to get some feel for how volatile the data is. On each line of stdout, the minimum and maximum of the five datapoints is reported and the relative standard error (expected relative standard deviation of the average).

  Besides the stdout, the test produces a `.datinfo` and corresponding `.dat` file that can be given to the `mkreport.pl` command to produce a graph of the data.

  For example, if there is a file called `hosts` with the following contents:

```
                        hostA:8
                        hostB:8
                        hostC:8
                        hostD:8
                        hostE:8
                        hostF:8
                        hostG:8
                        hostH:8
$MPI_ROOT/bin/mpirun -hostfile hosts $MPI_ROOT/bin/mpitool -pphr 100000

> - ping-pong 100000 bytes, using 1 ranks per host
>   - [  0] (hostA): avg   2537.52 (2515-2549) Mb/sec (rse 0.21%)
>   - [  1] (hostB): avg   2537.88 (2536-2540) Mb/sec (rse 0.02%)
>   - [  2] (hostC): avg   2531.42 (2529-2533) Mb/sec (rse 0.03%)
>   - [  3] (hostD): avg   2527.44 (2527-2529) Mb/sec (rse 0.01%)
>   - [  4] (hostE): avg    426.77 (425-429) Mb/sec (rse 0.16%)
>   - [  5] (hostF): avg    430.26 (423-437) Mb/sec (rse 0.59%)
>   - [  6] (hostG): avg   2530.84 (2529-2533) Mb/sec (rse 0.02%)
>   - [  7] (hostH): avg   2537.35 (2535-2539) Mb/sec (rse 0.02%)
>   *** most suspicious host indices: 4 5
> - ping-pong 100000 bytes, using 2 ranks per host
>   - [  0] (hostA): avg   3280.02 (3274-3287) Mb/sec (rse 0.07%)
>   - [  1] (hostB): avg   3603.41 (3553-3633) Mb/sec (rse 0.36%)
>   - [  2] (hostC): avg   3232.06 (3230-3234) Mb/sec (rse 0.02%)
>   - [  3] (hostD): avg   3183.20 (3180-3188) Mb/sec (rse 0.04%)
>   - [  4] (hostE): avg    329.86 (281-421) Mb/sec (rse 6.73%)
>   - [  5] (hostF): avg    310.24 (298-342) Mb/sec (rse 2.15%)
>   - [  6] (hostG): avg   3576.20 (3569-3583) Mb/sec (rse 0.07%)
>   - [  7] (hostH): avg   3564.91 (3558-3581) Mb/sec (rse 0.11%)
>   *** most suspicious host indices: 4 5
> - ping-pong 100000 bytes, using 3 ranks per host
>   - [  0] (hostA): avg   4092.09 (4061-4138) Mb/sec (rse 0.29%)
>   - [  1] (hostB): avg   4018.45 (3965-4084) Mb/sec (rse 0.50%)
>   - [  2] (hostC): avg   4030.09 (4010-4053) Mb/sec (rse 0.17%)
>   - [  3] (hostD): avg   4022.87 (4000-4040) Mb/sec (rse 0.17%)
>   - [  4] (hostE): avg    312.27 (311-314) Mb/sec (rse 0.17%)
>   - [  5] (hostF): avg    308.84 (303-314) Mb/sec (rse 0.57%)
>   - [  6] (hostG): avg   4082.11 (4032-4165) Mb/sec (rse 0.52%)
>   - [  7] (hostH): avg   4112.89 (4077-4162) Mb/sec (rse 0.33%)
>   *** most suspicious host indices: 4 5
> - ping-pong 100000 bytes, using 4 ranks per host
>   - [  0] (hostA): avg   4750.42 (4725-4780) Mb/sec (rse 0.19%)
>   - [  1] (hostB): avg   4691.40 (4626-4734) Mb/sec (rse 0.39%)
>   - [  2] (hostC): avg   4643.43 (4613-4673) Mb/sec (rse 0.19%)
>   - [  3] (hostD): avg   4668.42 (4654-4684) Mb/sec (rse 0.11%)
>   - [  4] (hostE): avg    295.31 (294-297) Mb/sec (rse 0.16%)
>   - [  5] (hostF): avg    293.90 (292-295) Mb/sec (rse 0.19%)
>   - [  6] (hostG): avg   4675.50 (4634-4704) Mb/sec (rse 0.25%)
>   - [  7] (hostH): avg   4666.32 (4634-4692) Mb/sec (rse 0.19%)
>   *** most suspicious host indices: 4 5
> - ping-pong 100000 bytes, using 5 ranks per host
>   - [  0] (hostA): avg   5722.31 (5688-5752) Mb/sec (rse 0.21%)
>   - [  1] (hostB): avg   5476.99 (5455-5502) Mb/sec (rse 0.14%)
>   - [  2] (hostC): avg   5492.86 (5473-5516) Mb/sec (rse 0.13%)
>   - [  3] (hostD): avg   5618.14 (5575-5665) Mb/sec (rse 0.25%)
>   - [  4] (hostE): avg    275.74 (274-277) Mb/sec (rse 0.14%)
>   - [  5] (hostF): avg    277.35 (276-278) Mb/sec (rse 0.14%)
>   - [  6] (hostG): avg   5725.02 (5693-5754) Mb/sec (rse 0.19%)
>   - [  7] (hostH): avg   5705.59 (5655-5771) Mb/sec (rse 0.33%)
>   *** most suspicious host indices: 4 5
> - ping-pong 100000 bytes, using 6 ranks per host
>   - [  0] (hostA): avg   5621.85 (5600-5648) Mb/sec (rse 0.15%)
>   - [  1] (hostB): avg   5558.90 (5545-5568) Mb/sec (rse 0.07%)
>   - [  2] (hostC): avg   5561.29 (5542-5600) Mb/sec (rse 0.17%)
>   - [  3] (hostD): avg   5578.57 (5520-5610) Mb/sec (rse 0.25%)
>   - [  4] (hostE): avg    279.22 (277-282) Mb/sec (rse 0.30%)
>   - [  5] (hostF): avg    279.26 (278-281) Mb/sec (rse 0.15%)
```

```
>   - [  6] (hostG): avg   5587.60 (5565-5612) Mb/sec (rse 0.13%)
>   - [  7] (hostH): avg   5612.19 (5598-5630) Mb/sec (rse 0.08%)
>   *** most suspicious host indices: 4 5
> - ping-pong 100000 bytes, using 7 ranks per host
>   - [  0] (hostA): avg   5765.28 (5714-5797) Mb/sec (rse 0.22%)
>   - [  1] (hostB): avg   5718.80 (5690-5736) Mb/sec (rse 0.13%)
>   - [  2] (hostC): avg   5719.09 (5707-5728) Mb/sec (rse 0.06%)
>   - [  3] (hostD): avg   5698.18 (5667-5715) Mb/sec (rse 0.14%)
>   - [  4] (hostE): avg    275.50 (274-277) Mb/sec (rse 0.16%)
>   - [  5] (hostF): avg    276.61 (275-278) Mb/sec (rse 0.13%)
>   - [  6] (hostG): avg   5714.00 (5684-5735) Mb/sec (rse 0.13%)
>   - [  7] (hostH): avg   5765.01 (5742-5778) Mb/sec (rse 0.12%)
>   *** most suspicious host indices: 4 5
> - ping-pong 100000 bytes, using 8 ranks per host
>   - [  0] (hostA): avg   5869.22 (5843-5883) Mb/sec (rse 0.10%)
>   - [  1] (hostB): avg   5817.97 (5805-5826) Mb/sec (rse 0.06%)
>   - [  2] (hostC): avg   5816.97 (5797-5827) Mb/sec (rse 0.09%)
>   - [  3] (hostD): avg   5808.06 (5742-5838) Mb/sec (rse 0.26%)
>   - [  4] (hostE): avg    274.06 (273-276) Mb/sec (rse 0.15%)
>   - [  5] (hostF): avg    275.07 (274-276) Mb/sec (rse 0.12%)
>   - [  6] (hostG): avg   5796.26 (5760-5814) Mb/sec (rse 0.15%)
>   - [  7] (hostH): avg   5850.21 (5839-5867) Mb/sec (rse 0.08%)
>   *** most suspicious host indices: 4 5
> Data written to out.pingpong_hosts.100000.dat and .datinfo.
> Viewable graphically via: mkreport.pl out.pingpong_hosts.100000.datinfo
```

In the output, the host name listed in parenthesis is the left-neighbor of the ping-pong. So, for example, in the above data when the lines for (hostE) and (hostF) both look bad, that means the ping-pongs between hostE-hostF and between hostF-hostG were bad, suggesting host-F has a problem. The automated statistics that identify suspicious host indices doesn't consider that aspect though and just reports hostE and hostF as being suspicious.

The data can also be viewed graphically which is helpful on larger clusters. This is accomplished with the **$MPI_ROOT/bin/mkreport.pl** command which can be run on the `out.pingpong_hosts.100000.datinfo` output file:

```
$MPI_ROOT/bin/mkreport.pl out.pingpong_hosts.100000.datinfo

> Parsing data from out.pingpong_hosts.100000.datinfo
>   - output is at
>     1. graph.pingpong_hosts.100000.png (all in 1 graph)
>     2. table.pingpong_hosts.100000.html (see the numbers if you want)
>   - Suspicious hosts from dataset pingpong_hosts.100000:
>     hostE
>     hostF
```

It also produces an html report of the same data at `Report/report.html`

- host-level flooding (`-flood`)

  In this test each host receives a flood of messages from a gradually increasing sequence of its neighbors and the two data points of interest for each host are what total bandwidth was achieved while flooding, and how many neighbors were able to flood before bandwidth became low. It is a synchronized flood, meaning that rather than having each peer sending to the root as fast as it individually can, the root sends one ping message and all the peers send one message back at roughly the same time so the messages end up throttled by the slowest peer. A two dimensional graph is made for each set of data points. The flooding is performed both with 1 rank per host being active, and with n ranks per host.

  The program takes two optional integers on the command line to specify.

  1. The number of bytes to use in the message traffic
  2. How much time (in ms) to spend on each flooding step, this is the amount of time to spend at each peer count (default 10 ms).

3. Percent below best bandwidth to stop flooding (default 75%)

Besides the stdout, the test produces a `.datinfo` and corresponding `.dat` file that can be given to the `mkreport.pl` command to produce a graph of the data.

For example, if there is a file called `hosts` with the following contents:

```
hostA:8
hostB:8
hostC:8
hostD:8
hostE:8
hostF:8
hostG:8
hostH:8
```

When using a 32000 byte messages:

```
$MPI_ROOT/bin/mpirun -hostfile hosts $MPI_ROOT/bin/mpitool -flood 32000

(abbreviating the output somewhat)

> Running with root-host 0 (hostA) (1/host)
>   at k=1 got 1465.5558 Mb/sec (best so far 1465.5558, this is 100.00%)
>   at k=2 got 2024.7761 Mb/sec (best so far 2024.7761, this is 100.00%)
>   at k=3 got 2332.7060 Mb/sec (best so far 2332.7060, this is 100.00%)
>   at k=4 got 2519.0502 Mb/sec (best so far 2519.0502, this is 100.00%)
>   at k=5 got 2636.3160 Mb/sec (best so far 2636.3160, this is 100.00%)
>   at k=6 got 2754.1264 Mb/sec (best so far 2754.1264, this is 100.00%)
>   at k=7 got 2821.1453 Mb/sec (best so far 2821.1453, this is 100.00%)
> Running with root-host 1 (hostB) (1/host)
>   at k=1 got 1496.9156 Mb/sec (best so far 1496.9156, this is 100.00%)
>   at k=2 got 2051.7504 Mb/sec (best so far 2051.7504, this is 100.00%)
>   at k=3 got 2350.0500 Mb/sec (best so far 2350.0500, this is 100.00%)
>   at k=4 got 2543.1471 Mb/sec (best so far 2543.1471, this is 100.00%)
>   at k=5 got 2693.2953 Mb/sec (best so far 2693.2953, this is 100.00%)
>   at k=6 got 2779.0316 Mb/sec (best so far 2779.0316, this is 100.00%)
>   at k=7 got 2815.8659 Mb/sec (best so far 2815.8659, this is 100.00%)
> ...
> Running with root-host 0 (hostA) (8/host)
>   at k=1 got 3090.4636 Mb/sec (best so far 3090.4636, this is 100.00%)
>   at k=2 got 3315.8279 Mb/sec (best so far 3315.8279, this is 100.00%)
>   at k=3 got 3316.0933 Mb/sec (best so far 3316.0933, this is 100.00%)
>   at k=4 got 3335.5825 Mb/sec (best so far 3335.5825, this is 100.00%)
>   at k=5 got 3333.6434 Mb/sec (best so far 3335.5825, this is 99.94%)
>   at k=6 got 3337.5867 Mb/sec (best so far 3337.5867, this is 100.00%)
>   at k=7 got 3335.9388 Mb/sec (best so far 3337.5867, this is 99.95%)
> Running with root-host 1 (hostB) (8/host)
>   at k=1 got 3079.9224 Mb/sec (best so far 3079.9224, this is 100.00%)
>   at k=2 got 3323.9715 Mb/sec (best so far 3323.9715, this is 100.00%)
>   at k=3 got 3332.9590 Mb/sec (best so far 3332.9590, this is 100.00%)
>   at k=4 got 3334.5739 Mb/sec (best so far 3334.5739, this is 100.00%)
>   at k=5 got 3334.5684 Mb/sec (best so far 3334.5739, this is 100.00%)
>   at k=6 got 3334.9656 Mb/sec (best so far 3334.9656, this is 100.00%)
>   at k=7 got 3333.7031 Mb/sec (best so far 3334.9656, this is 99.96%)
> ...
> Data written to out.oneall_saturate_count.32000.dat and .datinfo.
> Viewable graphically via: mkreport.pl out.oneall_saturate_count.32000.datinfo
> Data written to out.oneall_saturate_bw.32000.dat and .datinfo.
> Viewable graphically via: mkreport.pl out.oneall_saturate_bw.32000.datinfo
```

The data can also be viewed graphically which is helpful on larger clusters. This is accomplished with the **$MPI_ROOT/bin/mkreport.pl** command which can be run on the `out.oneall_saturate_bw.32000.datinfo` output file (and also on `out.oneall_saturate_count.32000.datinfo`). The first graph shows bandwidth numbers for each host, the second shows how many peers were able to flood the host.

```
% $MPI_ROOT/bin/mkreport.pl out.oneall_saturate_bw.32000.datinfo

> Parsing data from out.oneall_saturate_bw.32000.datinfo
>   - output is at
>     1. graph.oneall_saturate_bw.32000.png (all in 1 graph)
>     2. table.oneall_saturate_bw.32000.html (see the numbers if you want)
>   - No suspicious hosts from dataset oneall_saturate_bw.32000.
```

It also produces an html report of the same data at `Report/report.html`

- ping-pong on all host pairs (`-allpairs`)

  This test runs ping-pong between all host pairs, with several ping-pongs run simultaneously to make it faster. Between each host pair, multiple ranks try their pairings and the slowest is reported in the three-dimensional output graph. stdout also includes notes when one rank-pair was a lot slower than other rank-pairs within the same host-pair. Each host is paired with self+1, then self+2, then self+3, and so on.

  The program takes the following optional command options:

  – **-nbytes #**: for each ping pong
  – **-blocksize #**: distance between hosts initiating ping pongs
  – **-usec #**: target usec per ping pong, default 100000
  – **-nperhost #**: ranks active per host
  – **-factor <float>**: number greater than 1.0, default 1,5,

  On large clusters the option `-nperhost 1` might be necessary for the test to finish in a reasonable time. That option effectively disables the notion of testing multiple paths between host-pairs since there is then only a single path between any two hosts. On each line of stdout the minimum or maximum value of the 5 data points is reported and the relative standard error (expected relative standard deviation of the average).

  Besides the stdout, the test produces a `.datinfo` and corresponding `.dat` file that can be given to the `mkreport.pl` command to produce a three-dimensional graph of the data.

  For example, using 1000000 byte ping-pongs between each host:

  For example, if there is a file called `hosts` with the following contents:

  ```
  hostA
  hostB
  hostC
  hostD
  ```

```
$MPI_ROOT/bin/mpirun -hostfile hosts $MPI_ROOT/bin/mpitool -allpairs -nbytes 1000000

> - notes on expected runtime (ping-pong nbytes 1000000):
>   - 2 offsets
>   - 2 stages per offset
>   - 1 ping-pong paths tested per stage
>   - 100000 target usec per pingpong
>   - very rough estimate 0 seconds total
> If the above projection is super-long, consider reducing
> the time per ping-pong with -usec # or reducing the number
> of ping-pong paths tested per stage with -nperhost #
> --------------------------------------------------
> running ping pongs with offset 1
> running ping pongs with offset 2
> Data written to out.allpairs.1000000.dat and .datinfo.
> Viewable graphically via: mkreport.pl out.allpairs.1000000.datinfo
```

The following example uses an artificially small value for **-factor** so lines will be displayed reporting differences between best and worst paths for each host pair even though.

For example, if there is a file called `hosts` with the following contents:

```
                        hostA:8
                        hostB:8
                        hostC:8
                        hostD:8
$MPI_ROOT/bin/mpirun -hostfile hosts $MPI_ROOT/bin/mpitool -allpairs -nbytes 1000000 -factor 1.001

> - notes on expected runtime (ping-pong nbytes 1000000):
>   - 2 offsets
>   - 2 stages per offset
>   - 16 ping-pong paths tested per stage
>   - 100000 target usec per pingpong
>   - very rough estimate 6 seconds total
> If the above projection is super-long, consider reducing
> the time per ping-pong with -usec # or reducing the number
> of ping-pong paths tested per stage with -nperhost #
> ------------------------------------------------
> running ping pongs with offset 1
> - host 0:1 hostA:hostB pair 5:1 - min 3154.29 avg 3156.92 max 3158.93 MB/sec
> - host 1:2 hostB:hostC pair 7:7 - min 3154.02 avg 3157.02 max 3158.96 MB/sec
> - host 2:3 hostC:hostD pair 3:3 - min 3154.02 avg 3157.50 max 3158.96 MB/sec
> - host 3:0 hostD:hostA pair 6:6 - min 3153.52 avg 3156.90 max 3158.89 MB/sec
> running ping pongs with offset 2
> - host 0:2 hostA:hostC pair 6:2 - min 3148.03 avg 3155.95 max 3158.24 MB/sec
> - host 1:3 hostB:hostD pair 2:6 - min 3148.03 avg 3156.90 max 3158.56 MB/sec
> - host 2:0 hostC:hostA pair 4:0 - min 3154.98 avg 3157.66 max 3159.71 MB/sec
> - host 3:1 hostD:hostB pair 4:4 - min 3154.42 avg 3156.96 max 3159.71 MB/sec
> Data written to out.allpairs.1000000.g1_worst.dat and .datinfo.
> Viewable graphically via: mkreport.pl out.allpairs.1000000.g1_worst.datinfo
> Data written to out.allpairs.1000000.g2_best.dat and .datinfo.
> Viewable graphically via: mkreport.pl out.allpairs.1000000.g2_best.datinfo
```

Changed default installation path

The default installation path is changed to `/opt/ibm/platform_mpi`. You may change this installation directory by using the **-installdir=**<*dir*> option when running the installer.

MPI 3.0 non-blocking collective support/preview

Platform MPI has preliminary support for non-blocking collectives on Linux. Support is for the c interface, single threaded, non-HA and contiguous data types only. In order to use them, add the `-DNON_BLOCKING_COLLECTIVES` compiler option.

Prototypes are provided in `mpi.h`.

Removed FLEXlm license file requirement

There is no longer a requirement to have a FLEXlm license file in the `$MPI_ROOT/licenses` directory.

Setting memory policies with libnuma for internal buffers

There is a soft requirement on any version of **libnuma** installed on the system. To allow Platform MPI to set memory policies for various internal buffers, ensure that the user's LD_LIBRARY_PATH includes any version of `libnuma.so`.

The job will run without memory policies if there is no **libnuma** available.

Performance enhancements to collectives

The following general performance enhancements have been made to the collectives:

- Performance enhancements to **MPI_Gatherv**, **MPI_Allgatherv**, and **MPI_Scatterv** for zero-byte message sizes.
- Performance enhancements and optimized algorithms for **MPI_Scatter** and **MPI_Gather** for messages smaller than 1KB.

Enhanced error messaging for FCA failures

Currently, when FCA errors occur from the lower level FCA/IB hardware, Platform MPI issues a generic error message. To help better debug lower level FCA errors, users can now enable the error message to be more verbose by setting the environment variable `MPI_COLL_FCA_VERBOSE=`*number* where *number* is a numeric value between 1-9. The higher the number, the more verbose the message. The verbosity level is defined by the Mellanox FCA functionality. For more verbosity in FCA errors, it is recommended that users use between 1 and 3, and only use higher verbosity if requested by Mellanox or Platform when debugging the issue.

GPU-Direct 2.0 enhancements

Platform MPI supports Nvidia GPU-Direct and Unified Virtual Addressing (UVA), which enables the MPI library to differentiate between device memory and host memory without any hints from the user and transfer primitives copy data directly to and from GPU memory. This brings less application code changes for data transfers between the GPU and CPU.

To enable GPU-Direct functionality, set the environment variable `PMPI_GPU_AWARE=1`.

For example, assuming that `libcuda.so` is in the `LD_LIBRARY_PATH` environment variable on each node,

```
mpirun -hostlist "hpgpu[01-04]":12 -e PMPI_GPU_AWARE=1 -IBV ./testgather
```

Infiniband QoS service level

Platform MPI now features the ability to define the IB QoS Service level. These service levels are set up or defined by the system administrator in the subnet manager (refer to your subnet manager documentation for information on how to set up additional service levels). If additional service levels have been set up, users may specify the MPI job's IB connection to use one of these non-default service levels. To define the service level for the IB connections, set the `PCMPI_IB_SL` environment variable to the desired service level, which is between 1 and 14. The default service level is 0.

-env_inherit flag

The **-env_inherit** environment variable is an option for **mpirun**. With this option, the **mpirun** current environment is propagated into each rank's environment.

There is a set of fixed environment variables that will not automatically propagate from the **mpirun** environment to the ranks. This list is different for Windows and Linux. The exclusion lists are intended to prevent conflicts between the runtime environment of the **mpirun** process and the ranks. For example, on Linux the `HOSTNAME` environment variable is not propagated.

Users can also include environment variables they would like to prevent from propagating to the rank environments by using `MPI_ENV_FILTER`. This environment variable is a comma-separated list of environment variables to prevent from being propagated from the current environment. Filtered environment variables can include a wild card "`*`" but only as a post-fix to the environment variable.

For example,

```
setenv MPI_ENV_FILTER "HOSTNAME,MYID,MYCODE_*"
```

Note:

In some shells, the wild card character may need to be escaped even when embedded in a quoted string.

In this example, the environment variables `HOSTNAME`, `MYID`, and any environment variable that starts with `MYCODE_` will not be propagated to the ranks' environments. In addition, the `MPI_ENV_FILTER` environment variable is also never propagated to the ranks.

The `MPI_ENV_CASESENSITIVE` environment variable can change the behavior of case sensitivity when matching the filtered references. By default, case sensitivity is the same as the OS environment (that is, case sensitive for Linux and case insensitive for Windows). To change the default behavior, set `MPI_ENV_CASESENSITIVE` to yes or no (`1`, `y`, or `yes` to set; `0`, `n`, or `no` to unset).

Platform MPI also offers the **mpirun** command line option `-e` *var_name=var_value* that will explicitly set that environment variable in the process prior to exec'ing the rank. The `-envlist var_name[,`*var_name*`,...]` option can also be used to explicitly propagate variables from the **mpirun** environment to the ranks' environments.

System Check benchmarking improvements

System Check benchmarking has been improved to write out the binary data file after each step in the benchmarking process. If the benchmarking run completes normally, all the intermediate files are removed, and only the final binary data file will remain.

If the benchmarking run terminates before completing, the last complete binary data file, and sometimes an incomplete binary data file will be in `MPI_WORKDIR` on the node with rank 0. The incremental binary datafiles are named *filename.number*.

The lowest numbered file that is on the system will be the last complete binary data file.

This enhancement allows the benchmarking run to be run in a job scheduler for a fixed amount of time, and the "best effort" made to benchmark the cluster during that time.

Single mpid per host

Platform MPI now consolidates its internal mpid process so that in normal runs, only one mpid will be created per host. This can help to conserve system resources. Previously, when ranks were launched cyclically across a set of hosts (for example, **$MPI_ROOT/bin/mpirun -hostlist hostA,hostB,hostC,hostD -np 16 ...**), Platform MPI would create a separate mpid process for each contiguous block of ranks on a host, resulting, in this example, in four mpids on each of the four hosts. With this feature, Platform MPI creates only one mpid per host in this example.

Note that there are two expected exceptions to the mpid consolidation. In the following cases, it is expected that Platform MPI launches multiple mpids:

1. When two IP addresses in the host list or appfile resolve to the same host (for example, a multi-homed host).
2. When using an appfile and providing different environment variables to different ranks.

Progression thread

Platform MPI contains a progression thread that can be enabled with the **-progtd[=***options***]** argument, which accepts a comma-separated list of the following options:

unbind

Unbind progression thread. By default, the progression thread inherits the same binding as the rank that it is running under.

u<*number*>

Specific amount of time to usleep per advance. The default is 500.

ym0 | ym1 | ym2

Levels of **sched_yield** to occur inside the loop:

- ym0: Busy spin, no **sched_yield** in the loop.
- ym1: Medium spin, **sched_yield** each loop where no active requests are seen.
- ym2: Lazy spin, **sched_yield** each loop

adv<*number*>

Only allow <*number*> advances per iteration. The default is unlimited.

If **-progtd** is used without options, the default is equivalent to **-progtd=u500,ym1**.

This default option will use a small amount of extra CPU cycles, but in some applications, the guaranteed progression of messages is worth that cost.

New Infiniband/RoCE card selection

When a machine has multiple Infiniband cards or ports, Platform MPI can stripe messages across the multiple connections. To allow easier selection of which cards or ports to use, the `MPI_IB_STRINGS` environment variable can be set to a comma-separated list of

string[:*port*]

where the *string* is the card name as identified by `ibv_devinfo`.

For example, *string* can be a card name such as `mthca0` or `mlx4_0`, and the port is a 1-based number such as `1` or `2`.

The `MPI_IB_STRINGS` environment variable can also be set to one of several keywords:

nonroce

Only use regular non-RoCE IB ports.

default

Use non-RoCE if available, but switch to roce if no regular IB ports exist.

all

Use all available IB ports, both RoCE and non-RoCE.

roce

Only use RoCE ports.

v

Verbose, shows what cards or ports each rank decided to use.

Using the KNEM module

If the **knem** kernel module is installed on a machine, the **-e MPI_1BCOPY=***number-of-bytes* option can be used to specify a threshold above which **knem** is to be used to transfer messages within a host. The lowest meaningful threshold is 1024. Below that amount, shared memory is always used. The MPI_1BCOPY=1 value is a special case meaning "2048", which is a suggested starting default.

By default, **knem** is not used to transfer any messages within a host.

Topology querying with -cpu_bind

Note:

-cpu_bind is not the preferred affinity solution for use with Platform MPI and is provided for compatibility. Consider using the -aff and -affopt command line options instead.

The PCMPI_TOPOLOGY_METHOD environment variable was introduced in HP-MPI 2.2.5.1. This option was originally intended for large SMP machines (such as HP Superdome). With new higher core count processors, you may need to be aware of this environment variable again.

When the core count of a system is larger than 32, direct inspection of the /sys/devices/system/node directory is used to gather topology information. Newer versions of **libnuma** and **numactl** should make this unnecessary. However, for users with an older or unknown version of **libnuma**, direct inspection of the system may yield better results.

On jobs that run on heterogeneous clusters that include nodes with 32 cores or less and nodes with more than 32 nodes, the PCMPI_TOPOLOGY_METHOD environment variable should be set. This will force all nodes to use the same topology discovery method, and will likely lead to more predictable CPU Affinity results for jobs across nodes with differing core counts.

In general, the best advice is to force both the "new" and "old" method, and determine which set of results is preferable for the specific application being used. If that experiment cannot be done, the next best guidance is that the "old" method with a new version of **libnuma** is more likely to be correct.

```
PCMPI_TOPOLOGY_METHOD=old | new
```

The **cpu_bind** option uses a number of different methods to determine the topology layout and relative placement of the cores on a machine. The specific topology discovery method can be selected using the PCMPI_TOPOLOGY_METHOD environment variable.

The "old" method is used by default when the number of cores on the machine is 32 or fewer. The "new" method is used by default when the number of cores on the machine is greater than 32.

Note:

The test for "old" versus "new" is based on the total number of cores on the machine, and NOT the number of ranks on the machine or the number of cores that are in use by the job.

The "old" method relies on the **numa_node_to_cpus** call found in the **libnuma** library. If the **libnuma** library is not up to date on the hardware, it is possible that incorrect information will be returned by that call.

The "new" method walks the `/sys/devices/system/node` directory and examines the entries to determine the CPU to **numa** node topology of the machine. In general, this method is slower and less portable than the "old" method, but seems to produce more consistent results on newer hardware architectures. The "new" method has been tested using RHEL 5.x and 6.x, on both AMD and Intel based systems containing up to 64 cores.

-machinefile flag

This flag launches the same executable across multiple machines. This flag is synonymous with **-hostfile**, and has the following usage:

`mpirun ... -machinefile` *`file_name`* `...`

where *`file_name`* is a text file with machine names and optional rank counts separated by spaces or new lines with the following format: *`host_name`*`:`*`number`*.

For example:

```
hostA:8
hostB:8
hostC:12
hostD:12
```

Shared memory usage optimization

Platform MPI features management of the shared memory that is used for both the collectives and the communicator locks. This reduces the communicator memory footprint.

RDMA buffer message alignment for better performance

Some newer types of servers are sensitive to memory alignment for RDMA transfers. Platform MPI features memory optimization that aligns the RDMA buffers for better performance.

RDMA buffer alignment

The alignment of the data being transferred can affect RDMA performance. Platform MPI uses protocols that realign the data when possible for the best performance.

General memory alignment

To improve performance in general and to decrease the odds of user buffers being unaligned, Platform MPI causes memory allocations to be aligned on 64-byte boundaries by default.

Control the memory alignment by using the following option to define the `MPI_ALIGN_MEM` environment variable:

```
-e MPI_ALIGN_MEM=n_bytes | 0
```

Aligns memory on the *n_bytes* boundaries. The *nbytes* value is always rounded up to a power of two.

To disable memory alignment, specify `MPI_ALIGN_MEM=0`. This was the previous behavior.

The default value is 64 bytes (the cache line size).

When using this option (which is on by default) to align general memory, the **`realloc()`** call functions more slowly. If an application is negatively impacted by this feature, disable this option for `realloc()` calls by using the following option to define the `MPI_REALLOC_MODE` environment variable:

```
-e MPI_REALLOC_MODE=1 | 0
```

Mode 1 (`MPI_REALLOC=1`) makes **`realloc()`** calls aligned but a bit slower than they would be otherwise. This is the default mode.

Mode 2 (`MPI_REALLOC=0`) makes the **`realloc()`** calls fast but potentially unaligned.

GPU-enabled systems and applications

Platform MPI features support and optimizations for GPU-enabled applications and systems by using the following environment variables:

- `MPI_1SIDED_MODE=2`

  If set, uses the slower TCP/commd path for one-sided calls while still using InfiniBand for point-to-point messaging.

  This is required for GPU-enabled systems and applications.

- `MPI_RDMA_REPIN=1`

  Controls the MPI buffer optimization for RDMA messaging. If set, Platform MPI checks if the buffer can be pinned for RDMA messaging. If the buffer cannot be pinned (for example, for GPU usage), PMPI will use an RDMA protocol that will not attempt to "repin" the MPI buffer.

GPU-Direct 1.0

For systems that have GPU-Direct installed, Platform MPI supports use of GPU and RDMA pinned memory regions to share the same memory region. This has a significant memory improvement for GPU applications.

Platform MPI only supports virtual addresses referring to memory allocations on the host. It is up to the application to DMA messages from the GPU to the host memory before passing to the MPI and after receiving from the MPI. This limitation is resolved with the GPU-Direct 2.0 enhancements.

RDMA transfers that do not guarantee bit order

A large number of InfiniBand hardware guarantees bit order when using RDMA message transfers. As newer hardware comes on the market using the same IBV RDMA protocols, not all the new hardware guarantees bit order. This will cause MPI messaging errors if the order is not guaranteed. The following environment variable controls support for RDMA transfers that do not guarantee bit order:

```
MPI_RDMA_ORDERMODE=1 | 2
```

Specify `MPI_RDMA_ORDERMODE=2` so MPI messages do not depend on guaranteed bit order. This causes a 2-3% performance loss for message transfers.

Specify `MPI_RDMA_ORDERMODE=1` to assume guaranteed bit order.

The default value is 1 (assumes guaranteed bit order).

KNEM kernel-memcopy modules

Platform MPI now supports KNEM kernel-memcopy modules for DMA memory copies of shard memory messages. This can increase performance for some applications for shared memory messages (ranks on the same node), but can decrease performance for most other cases.

To enable support for KNEM, download and install the open source kernel from http://runtime.bordeaux.inria.fr/knem, then use the following option to specify the `MPI_1BCOPY` environment variable:

```
-e MPI_1BCOPY=1.
```

FCA 2.0 features

Platform MPI adds the following FCA 2.0 features to existing FCA 1.x infrastructure:

- Ability for the FCA layer to handle up to MAX_INT-sized messages.
- Platform MPI supports the following FCA collectives on clusters greater than 16 nodes: Barrier, Bcast, Reduce, and Allreduce. Allgather and Allgatherv can be enabled if desired (see the *Known issues* section for more details). For running on clusters smaller than 16 nodes, force the FCA collectives by using the **`-noa=fca:force`** option.
- New progression improvements added in FCA 2.0 to allow non-FCA messages to progress, which may eliminate some deadlock cases in older FCA-enabled jobs.

For more information on the existing FCA 1.x infrastructure, refer to the *FCA integration into libcoll and collective libraries* section in these release notes.

uDAPL protocol selection

Platform MPI supports uDAPL protocol selection by using the following environment variable:

```
MPI_HASIC_UDAPL=string
```

where *`string`* must match the name at the beginning of a line in the `/etc/dat.conf` file. If the string matches a name in the dat-conf file, that protocol is used for

uDAPL communication. This option will override default selections, and and the MPI_UDAPL_IFACE_INDEX and MPI_UDAPL_VERSION environment variables.

-cpu_bind and the NUMA library

-cpu_bind will use the NUMA library specified by the MPI_LIBNUMA environment variable to determine the appropriate CPU affinity settings to use. In addition, if libnuma.so is not found in LD_LIBRARY_PATH, libnuma.so.1 will also be checked. This change is recommended for AMD Magna Cours processors.

Use single quotes for submissions to HPC scheduler

In previous versions, it was necessary to "double-quote" strings with spaces because the HPC command line parser strips the first set of quotes. The latest version of HPC no longer does this, which causes the double-quoted strings to parse incorrectly. To correct this, Platform MPI now allows the use of single quotes for strings with spaces.

To enable the automatic parser to use single quotes for strings with spaces, enable the PCMPI_HPC_SINGLEQUOTE_ENV environment variable.

RDMA options for NUMA control

Added the **rdma** option for NUMA control.

This option controls where pre-pinned regions for RDMA protocol and small messages is allocated. For more details, see the following section.

NUMA control

Platform MPI includes NUMA options using libnuma to control memory placement policies for the ranks' main memory and the shared memory used in communication. This feature requires the libnuma library to be installed on the system. Any libnuma library already installed on the system can be used by explicitly setting the MPI_LIBNUMA environment variable to the desired library.

The most detailed control of the NUMA options is available using the **-numa** option on **mpirun**:

-numa=[shmem:*setting*[:*parameters*]][,mainmem:*setting*[:*parameters*]][,rdma:*setting*[:*parameters*]][,v]

or use the following to disable all libnuma usage:

-numa=none

The settings can also be controlled by setting MPI_NUMA_OPTIONS to any string that -numa would accept.

Platform MPI supports specifying the NUMA policy indepedently for the following:

Shared memory

The MPI shared memory use for communication between the ranks.

Main memory

The coarse granularity policy applied to the entire process.

RDMA messaging buffer

The pre-pinned regions for RDMA protocol and small messages.

- Use the `shmem:` option to specify the `libnuma` policy for the shared memory communication buffers.
- Use `mainmem:` to specify the policy for the ranks' main memory.
- Use `rdma:` to control where pre-pinned regions for RDMA protocol and small messages is allocated.

The **-numa** option is turned on by default (if **libnuma** version 2 is available) with the default policy set to interleaved allocation for the shared memory, and local allocation for the ranks' RDMA memory and main memory. The default setting is equivalent to **-numa=shmem:interleaved,rdma:local,mainmem:local**.

On some machines such as the SL390 with GPU, it is possible that the **-numa=rdma:nodes:0x1** option or possibly the more aggressive **-numa=rdma:nodes:0x1,mainmem:nodes:0x1** option will improve performance. The mroe aggressive latter option also puts the entire main program on node 0, which would only be advisable if the application is small enough to fit.

For each of the `shmem:`, `mainmem:`, and `rdma:` options, *setting* can be any of the following:

local

Sets the memory to the `libnuma` local policy.

interleaved

Sets the memory to the `libnuma` interleaved policy, where the node mask defaults to the whole machine, but can be controlled further with additional options (as indicated by *parameters*).

nodes

Sets the memory to the `libnuma` node subset policy, where the node mask representing the subset of nodes defaults to the whole machine, but can be controlled further with additional options (as indicated by *parameters*).

For each of the `shmem:` and `mainmem:` options using the `interleaved` or `nodes` settings, *parameters* represents the additional options that are used to specify the node masks for these settings, and can be set to any of the following:

all

This represents all the nodes on the host.

job

Platform MPI queries the CPU affinity of all the ranks and determines the associated nodes, and the resulting node mask represents the union of all those nodes.

rank

Each rank is given a different node mask representing the nodes associated with its CPU affinity.

hex_number **(for example, 0x3)**

Specify an explicit node mask.

hex_number1:*hex_number2:* **... (for example 0x1:0x2:0x1:0x2)**

Specify a list of node masks. The ranks each use a different mask from the list, cycling through the masks in the list based on the relative order of the ranks on the host.

Restriction:

NUMA control is not available on Windows and is only available on Linux if `libnuma` version 2 is available.

The general purpose environment variable `MPI_PRELAUNCH` can also be used to insert any utility in front of the rank at runtime.

For example,

```
$MPI_ROOT/bin/mpirun -e MPI_PRELAUNCH="numactl -i all" -np 2 ./pp.x
```

would be the equilavent to the following:

```
$MPI_ROOT/bin/mpirun -np 2 numactl -i all ./pp.x
```

However, the **-numa** option should cover most settings that numactl could provide.

Note:

Installing the `Platform_MPI_Extras` package is required to enable NUMA control features. This installs the open source `libnuma205.so` library into the Platform MPI `$MPI_ROOT` directory.

Collective algorithms

Platform MPI 9.1.2 includes additional collective algorithms added to the collective library. The additional collective algorithms include the new binomial tree Scatter and Gather algorithms.

TCP performance improvements

Platform MPI 9.1.2 has various performance improvements for point-to-point TCP interconnects.

Tunable TCP large message protocols

Platform MPI 9.1.2 has a new environment variable (`MPI_TCP_LSIZE`) that allows the alteration of long-message protocols for TCP messages:

The TCP protocol in Platform MPIsends short messages without waiting for the receiver to arrive while longer messages involve more coordination between the sender and receiver to transfer a message. By default, the transition from short- to long-message protocol occurs at 16384 bytes, but this is configurable using the `MPI_TCP_LSIZE` setting:

`MPI_TCP_LSIZE=`*bytes*

The default value is 16384. Many applications will see a higher performance with larger values such as 262144 because of the lower overhead that comes from not requiring the sender and receiver to coordinate with each other. This can involve slightly more buffering overhead when an application receives messages in a

different order than they were sent but this overhead is usually negligible compared to the extra header/acknowledgement synchronization overhead involved in the long message protocol.

The main disadvantage to larger settings it the increased potential for TCP flooding if many ranks send to the same rank at about the same time. The larger the quantity of data sent in this manner the worse it is for the network. The long-message protocol usually reduces the problem by only sending headers and staggering the message bodies. However, there are cases where the opposite is true: if a rank sends to all its peers using a long-message protocol it can be flooded with acknowledgements where the same sequence of messages using short-message protocol would have caused no flooding.

In general, applications whose message patterns are not prone to TCP flooding will be faster with larger `MPI_TCP_LSIZE` settings, while applications that are prone to flooding may need to be examined and experimented with to determine the best overall setting.

Support for the LSF_BIND_JOB environment variable in Platform LSF

Platform MPI 9.1.2 has increased support for Platform LSF jobs with integrating support for the `LSF_BIND_JOB` environment variable.

Since Platform LSF and Platform MPI both use CPU affinity, these features are integrated. Platform MPI 9.1.2 reads the `LSF_BIND_JOB` environment variable and translates it to the equivalent **-aff=***protocol* flag.

`LSF_BIND_JOB` is translated as follows:

- `BALANCE` = **-aff=automatic:bandwidth**
- `PACK` = **-aff=automatic:latency**
- `ANY` = **-aff=manual:0x1:0x2:0x4:0x8:...** with `MPI_AFF_PERHOST=1`, which makes it cycle through that manual list on a per-host basis (`host local rank_ID`) rather than by global rank ID.
- `USER` uses **LSB_USER_BIND_JOB** settings, which can be `Y`, `N`, `NONE`, `BALANCE`, `PACK`, or `ANY`. Note that `Y` is mapped to `NONE` and `N` is mapped to `ANY`.
- `USER_CPU_LIST` binds all ranks to the mask represented by `LSB_USER_BIND_CPU_LIST` formatted as `#,#,#-#,...`, that is, a comma-separated list of numbers and number-ranges, each of which represents a core ID.

Support for the bkill command in Platform LSF

Platform MPI 9.1.2 has increased support for Platform LSF jobs with the use of **bkill** for signal propagation when using **blaunch** to start ranks in a Platform LSF job.

Platform MPI automatically enables **bkill** usage if `LSF_JOBID` exists and any of the following conditions are true:

- The WLM selection is WLM_LSF (that is, the same circumstance where `MPI_REMSH` is currently set to `blaunch`).
- `MPI_REMSH` is set to either of the following:
  - `blaunch`
  - `blaunch` *arg arg arg*
  - */path/to/*`blaunch`

– */path/to/*blaunch *arg arg arg*

- MPI_USE_BKILL is set to 1.

Platform MPI will force the **bkill** mode to not be used if either of the following conditions are true:

- LSF_JOBID is not set.
- MPI_USE_BKILL is set to 1.

FCA integration into libcoll and collective libraries

Fabric Collective Accelerator (FCA) algorithms for Platform MPI collectives are integrated into the Platform MPI Collective library for 9.1.2.

The FCA collectives supported in Platform MPI 9.1.2 are **MPI_Bcast**, **MPI_Barrier**, **MPI_Reduce**, and **MPI_Allreduce**.

When the FCA algorithms are enabled, Platform MPI uses the Mellanox FCA algorithm accelerators provided by Mellanox InfiniBand switches. Contact Mellanox for information on required software and hardware to support FCA.By default, Platform MPI will try to detect FCA hardware and use FCA algorithms if available, when profitable. No additional flags are required at runtime to enable FCA; however, given the variety of clusters and environments, additional tuning may be required to optimize FCA performance.

Platform MPI provides the following environment variables and **mpirun** flags to alter FCA at runtime.

mpirun flags for Network Optimzied Algorithms (**-noa**):

```
-noa=[none | fca | FCA][:force][:verbose[=x]]
```

By default, FCA is used if available and profitable.

The **-noa** options are as follows:

none

Disables any **noa** optimizations available and does not use them if they exist.

fca

Enables the FCA-specific collective algorithms. These algorithms are used if profitiable. An error will occur if FCA is not available on a node in the MPI run.

FCA

Enables the FCA-specific collective algorithms, and forces the use of FCA for every use of a collective algorithm if a FCA options are available. An error will occur if FCA is not available on a node in the MPI run. This run will not use the most profitable collective algorithm, but does guarentee that FCA algorithms are used in all cases. This is the same as `-noa=fca:force`.

:force

Forces the use of the **noa** algorithms even if not profitable.

:verbose

Enables verbose output levels. Verbose level 1 is the least verbose while level 9 is the most verbose. Any level above 1 will enable Mellanox FCA logging at the given level.

Environment variables:

PCMPI_FCA_THRESHOLD

Indicates the minimum number of nodes required in the communicator to enable FCA algorithm to run. To enable FCA algorithms sooner, use a lower `PCMPI_FCA_THRESHOLD`.

`-noa=FCA` is equivalent to **`-noa=fca`** and setting `PCMPI_FCA_THRESHOLD=2`.

PCMPI_FCA_LEN_MAX_TIMES

allows chunking of messages by 132 bytes to enable larger messages to use FCA algorithms. The default value is 1.

To allow chunking messages up to 264, set `PCMPI_FCA_LEN_MAX_TIMES=2`.

-rank0 flag

This flag will take the first host of a job allocation and will schedule the first rank (rank 0) on this host. No other ranks will be allocated to that host. Job allocation can come from a scheduler allocation, **`-hostlist`** or **`-hostfile`**. The syntax for this flag is as follows:

```
mpirun ... -lsf -np 16 -rank0 app.exe
```

The actual number of ranks for the job may not match the `-np #` indicated in the **`mpirun`** command. The first host may allocate additional cores/slots on the first host, but because this feature will only start one rank per core/solt on the first host, the total ranks for the job will be short the "unallocated first host cores/slot" ranks.

For example, on a cluster with eight cores per host and assuming hosts are fully allocated, the following run will have 57 ranks. The first host will count eight towards the allocated 64 cores, but only one rank will be started for that "group of eight" ranks:

```
mpirun -lsf -np 64 -rank0 app.exe
```

The following example will start a job with 25 ranks, one rank on node1 and eight ranks on node2, node3, and node4:

```
mpirun -hostlist node1:8,node2:8,node3:8,node4:8 -rank0 app.exe
```

This flag is ignored if used with an appfile (**`-f appfile`**).

RDMA Coalescing improvements

Platform MPI 9.1.2 includes improvements to the RDMA coalescing feature. When using the `MPI_RDMA_COALESCING=0` flag, the MPI library would wait for the lower level Infiniband to send an IB message before returning. For applications that perform a large amount of computations before making any MPI calls, performance can be affected as some ranks may be waiting on a coalesced message. This will guarentee messages are sent before returning to the application.

Platform MPI 9.1.2 also added a progression thread option. Set `MPI_USE_PROGTD=1` to enable a progression thread, which will also allow coalesced messages to be sent without delay if the application has large computation sections before calling MPI code.

Both these environment variables will allow lower level IB messages to progress if the application has a large computation section. Enabling these by default will affect performance, so enabling by default is not recommended if your appplication does not have long spans where MPI calls are not made.

On-demand connections

Platform MPI 9.1.2 includes the ability to enable on-demand connections for IBV. To do this, set the environment variable `PCMP_ONDEMAND_CONN=1`. This will enable IBV connections between two ranks in an MPI run only if the ranks communicate with each other. If two ranks do not send messages to each other, the IBV connection is never established, saving the resources necessary to connect these ranks.

If an application does not use collectives, and not all the ranks send messages to other ranks, this could enable performance gains in startup, teardown and resource usage.

On-demand connections are supported for -rdma and -srq modes.

WLM scheduler functionality

Platform MPI 9.1.2 supports automatic scheduler submission for LSF and Windows HPCS. Current `-hpc`*`optionname`* options (such as **-hpcout**) are deprecated and will be removed in future releases. These options are now supported as `-wlm`*`optionname`* options and can be used for any supported schedulers. For example, **-hpcout** is now **-wlmout**. Currently, Platform MPI supports two schedulers in this fashion: LSF and Windows HPC.

Platform MPI continues to support legacy methods of scheduling such as LSF, **srun**, or PAM.

For LSF, support is included on both Windows and Linux platforms, and options should be consistent between the two.

To schedule and execute a job on a scheduler, include one of the scheduler options:

- **-hpc**: Include the **-hpc** option to use the Windows HPC Job Scheduler.

  This is used to automatically submit the job to HPCS scheduler and for HPCS Platform MPI jobs on the **mpirun** command line. This imples the use of reading the available hosts in the HPC job, and indicates how to start remote tasks using the scheduler.

  This is only supported on Windows HPC Server 2008.
- **-lsf**: Include the **-lsf** option to use the LSF scheduler.

  This is used to automatically submit the job to LSF, and on the LSF job **mpirun** commmand line. This flag implies the use of the **-lsb_mcpu_hosts** option and the use of **blaunch** to start remote processes.

These scheduler options are used for the MPI job command to set scheduler-specific functionality and for automatic job submission.

By including the scheduler options on the **mpirun** command line, this will enable certain scheduler functionality within **mpirun** to help construct the correct MPI job, and to help launch remote processes.

When using the **-lsf** option, this implies the use of the **-lsb_mcpi_hosts** option and also implies the use of `-e MPI_REMSH=blaunch`.

When using -hpc, this implies the use of reading the available hosts in the HPC job, and indicates how to start remote tasks via the scheduler.

By using the scheduler options, Platform MPI allows the use of the same **mpirun** command for all launch methods, with the only difference being the scheduler option used to indicate how to launch and create the MPI job. For more information on submitting WLM scheduler jobs, refer to "Submitting WLM scheduler jobs" on page 50.

System Check

The Platform MPI 9.1.2 library includes a lightweight System Check API that does not require a separate license to use. This feature was previously available only on Linux, and has been added to Windows for the Platform MPI 9.1.2 release. The System Check functionality allows you to test the basic installation and setup of Platform MPI without the prerequisite of a license. An example of how this API can be used can be found at `$MPI_ROOT/help/system_check.c`.

With System Check, you can list any valid option on the **mpirun** command line. The **PCMPI_SYSTEM_CHECK** API cannot be used if **MPI_Init** has already been called, and the API will call **MPI_Finalize** before returning. During the system check, the following tests are run:

1. **hello_world**
2. **ping_pong_ring**

These tests are similar to the code found in `$MPI_ROOT/help/hello_world.c` and `$MPI_ROOT/help/ping_pong_ring.c`. The ping_pong_ring test in system_check.c defaults to a message size of 4096 bytes. To specify an alternate message size, use an optional argument to the system check application. The `PCMPI_SYSTEM_CHECK` environment variable can be set to run a single test. Valid values of `PCMPI_SYSTEM_CHECK` are as follows:

- `all`: Runs both tests. This is the default value.
- `hw`: Runs the **hello_world** test.
- `ppr`: Runs the **ping_pong_ring** test.

As an alternate invocation mechanism, when the `$PCMPI_SYSTEM_CHECK` variable is set during an application run, that application runs normally until **MPI_Init** is called. Before returning from **MPI_Init**, the application runs the system check tests. When the System Check tests are complete, the application exits. This allows the normal application launch procedure to be used during the test, including any job schedulers, wrapper scripts, and local environment settings.

System Check benchmarking option

System Check can now run an optional benchmark of selected internal collective algorithms. This benchmarking allows the selection of internal collective algorithms during the actual application runtime to be tailored to the specific runtime cluster environment.

The benchmarking environment should be as close as practical to the application runtime environment, including the total number of ranks, rank-to-node mapping, CPU binding, RDMA memory and buffer options, interconnect, and other **mpirun** options. If two applications use different runtime environments, you need to run separate benchmarking tests for each application.

The time required to complete a benchmark varies significantly with the runtime options, total number of ranks, and interconnect. By default, the benchmark runs over 20 tests, and each test prints a progress message to **stdout** when it is complete. The benchmarking test should be run in a way that mimics the typical Platform MPI job, including rank count, **mpirun** options, and environment variables.

For clusters that include FCA hardware, it is recommended that the benchmarking run be done with up to 32 nodes. Beyond 32 nodes, there is limited benefit to additional testing. For jobs with larger rank counts, it is recommended that the rank count during benchmarking be limited to 512 with IBV/IBAL, 256 with TCP over IPoIB or 10G, and 128 with TCP over GigE. Above those rank counts, there is no benefit for better algorithm selection and the time for the benchmarking tests is significantly increased. The benchmarking tests can be run at larger rank counts; however, the benchmarking tests will automatically stop at 4092 ranks.

To run the System Check benchmark, compile the System Check example:

```
# $MPI_ROOT/bin/mpicc -o syscheck.x $MPI_ROOT/help/system_check.c
```

To create a benchmarking data file, set the `$PCMPI_SYSTEM_CHECK` environment variable to "BM" (benchmark). The default output file name is `pmpi820_coll_selection.dat`, and will be written into the `$MPI_WORKDIR` directory. The default output file name can be specified with the `$MPI_COLL_OUTPUT_FILE` environment variable by setting it to the desired output file name (relative or absolute path). Alternatively, the output file name can be specified as an argument to the `system_check.c` program:

```
# $MPI_ROOT/bin/mpirun -e PCMPI_SYSTEM_CHECK=BM \
[other_options] ./syscheck.x [-o output_file]
```

To use a benchmarking file in an application run, set the `$PCMPI_COLL_BIN_FILE` environment variable to the filename (relative or absolute path) of the benchmarking file. The file will need to be accessible to all the ranks in the job, and can be on a shared file system or local to each node. The file must be the same for all ranks.

```
# $MPI_ROOT/bin/mpirun -e PCMPI_COLL_BIN_FILE=file_path \
[other_options] ./a.out
```

Tuning the message checking on MPI_ANY_SOURCE

If an application spends a significant amount of time in **MPI_Test** or **MPI_Iprobe** checking for messages from `MPI_ANY_SOURCE`, the performance can be affected by how aggressively MPI looks for messages at each call. If the number of calls is much larger than the number of messages being received, less aggressive checking will often improve performance. This can be tuned using the following runtime option:

```
-e MPI_TEST_COUNT=integer
```

The value is the number of possible sources that will be checked for a waiting message on each call to **MPI_Test** or **MPI_Iprobe**. This option can have a value from 1 up to the number of ranks (larger values are truncated). The default value is 1 for Infiniband and 8 for TCP.

Dynamic library interface

Platform MPI 9.1.2 allows runtime selection of which MPI library interface to use (regular, multi-threaded, or diagnostic) as well as runtime access to multiple layers of PMPI interface wrapper libraries as long as they are shared libraries.

The main MPI libraries for Linux are as follows:

- regular: `libmpi.so.1`
- multi-threaded: `libmtmpi.so.1`
- diagnostic: `libdmpi.so.1`

In previous versions of Platform MPI, an application had to link against the desired library and any debugging or diagnostic work involving switching libraries would require re-linking. A new option (**-entry**) allows dynamic selection between the above libraries and also includes a copy of the open source MPE logging library from Argonne National Labs, version `mpe2-1.1.1`, which uses the PMPI interface to provide graphical profiles of MPI traffic for performance analysis.

The syntax for the new mpirun option is as follows:

`-entry=[manual:][verbose:]` *list*

where *list* is a comma-separated list of the following items:

- `reg` (refers to `libmpi.so.1`)
- `mtlib` (refers to `libmtmpi.so.1`)
- `dlib` (refers to libdmpi.so.1)
- `mtdlib` (refers to `dlib:mtlib`)
- `mpio` (refers to `libmpio.so.1`)
- `mpe (means libmpe.so)`

If you precede the list with the `verbose:` mode, a few informational messages are printed so you can see what libraries are being dlopened.

If you precede the list with the `manual:` mode, the given library list is used exactly as specified.

This option is best explained by first discussing the traditional non-dynamic interface. An MPI application contains calls to functions like **MPI_Send** and **MPI_File_open**, and is linked against the MPI libraries which define these symbols, in this case, `libmpio.so.1` and `libmpi.so.1`. These libraries define both the MPI entrypoints (like **MPI_Send**) and a PMPI interface (like **PMPI_Send**) which is a secondary interface into the same function. In this model a user can write a set of MPI function wrappers where a new library `libmpiwrappers.so` defines **MPI_Send** and calls **PMPI_Send**, and if the application is relinked against `libmpiwrappers.so` along with `libmpio.so.1` and `libmpi.so.1`, the application's calls into **MPI_Send** will go into `libmpiwrappers.so` and then into `libmpi.so.1` for the underlying **PMPI_Send**.

The traditional model requires the application to be relinked to access the wrappers, and also does not allow layering of multiple interface wrappers intercepting the same calls. The new **-entry** option allows both runtime control over the MPI/PMPI call sequence without relinking and the ability to layer numerous wrapper libraries if desired.

The **-entry** option specifies a list of shared libraries, always ending with `libmpio.so.1` and `libmpi.so.1`. A call from the application into a function like **MPI_Send** will be directed into the first library in the list which defines that function. When a library in the list makes a call into another **MPI_*** function that call is searched for in that library and down, and when a library in the list makes a call into **PMPI_*** that call is searched for strictly below the current library in the list. That way the libraries can be layered, each defining a set of **MPI_*** entrypoints and calling into a combination of **MPI_*** and **PMPI_*** routines.

When using **-entry** without the `manual:` mode, `libmpio.so.1` and `libmpi.so.1` will be added to the library list automatically. In manual mode, the complete library list must be provided. It is recommended that any higher level libraries like MPE or wrappers written by users occur at the start of the list, and the lower-level Platform-MPI libraries occur at the end of the list (`libdmpi`, then `libmpio`, then `libmpi`).

Example 1:

The traditional method to use the Platform-MPI diagnostic library is to relink the application against `libdmpi.so.1` so that a call into **MPI_Send** would resolve to **MPI_Send** library `libdmpi.so.1` which would call **PMPI_Send** which would resolve to **PMPI_Send** in `libmpi.so.1`. The new method requires no relink, simply the runtime option **-entry=dlib** (which is equivalent to **-entry=dlib,mpio,reg** because those base libraries are added automatically when manual mode is not used). The resulting call sequence when the app calls **MPI_Send** is the same: the app calls **MPI_Send** which goes into **MPI_Send** in `libdmpi.so.1` first then when that library calls **PMPI_Send**, that call is directed into the **MPI_Send** call in `libmpi.so.1` (`libmpio.so.1` was skipped over because that library doesn't define an **MPI_Send**).

Example 2:

The traditional method to use the MPE logging wrappers from Argonne National Labs is to relink against `liblmpe.so` and a few other MPE components. With the new method the runtime option **-entry=mpe** has the same effect (our build actually combined those MPE components into a single libmpe.so but functionally the behavior is the same).

For example,

```
-entry=verbose:mpe

-entry=manual:mpe,mpio,reg

-entry=dlib
```

Performance notes: If the **-entry** option is used, some overhead is involved in providing the above flexibility. Although the extra function call overhead involved is modest it could be visible in applications which call tight loops of **MPI_Test** or **MPI_Iprobe** for example. If **-entry** is not specified on the **mpirun** command line the dynamic interface described above is not active and has no effect on performance.

Limitations: This option is currently only available on Linux. It is also not compatible with the **mpich** compatibility modes.

Aggressive RDMA progression

Platform MPI on Infiniband has a feature called "message coalescing" which improves the message rate of streaming applications (applications which send many small messages quickly from rank-A to rank-B with little, if any traffic in the opposite direction). This feature is turned on by default (`MPI_RDMA_COALESCING=1`).

A side-effect of message coalescing is that sometimes in applications like the following, the message from rank-A to rank-B might not be available until rank-A re-enters MPI after the computation:

rank-A: **MPI_Send** to rank-B ; long computation; more MPI calls

rank-B: **MPI_Recv** from rank-A

This is generally undesirable especially since at the higher level, rank-A believes it has finished its message. So the following option is available to disable message coalescing and turn on more aggressive message progression:

```
-e MPI_RDMA_COALESCING=0
```

Submitting WLM scheduler jobs

To schedule and execute a job on a WLM scheduler, include one of the following scheduler options:

- **-hpc**: Include the **-hpc** option to use the Windows HPC Job Scheduler.

  This is used to automatically submit the job to HPCS scheduler and for HPCS Platform MPI jobs on the **mpirun** command line. This imples the use of reading the available hosts in the HPC job, and indicates how to start remote tasks using the scheduler.

  This is only supported on Windows HPC Server 2008.
- **-lsf**: Include the **-lsf** option to use the LSF scheduler.

  This is used to automatically submit the job to LSF, and on the LSF job **mpirun** commmand line. This flag implies the use of the **-lsb_mcpu_hosts** option and the use of **blaunch** to start remote processes.

These scheduler options are used for the MPI job command to set scheduler-specific functionality and for automatic job submission. By including these options on the **mpirun** command line, this will enable certain scheduler functionality within **mpirun** to help construct the correct MPI job, and to help launch remote processes. The scheduler options also allow you to use the same **mpirun** command for all launch methods, with the scheduler option being the only differentiator to indicate how to launch and create the MPI job.

To allow you to use a single **mpirun** command for different schedulers, Platform MPI supports automatic job submission. For LSF and HPC, **mpirun** can create and submit the scheduler job for you. You can include additional scheduler parameters by using the **-wlm** parameters.

To submit the job to the scheduler, include the scheduler flag, and if the mpirun command is not running in a scheduled job, it will create the proper scheduler command and submit itself as a scheduled job.

For example, "`mpirun -prot -np 16 -lsf rank`" will submit a job requesting 16 slots to the LSF scheduler. No additional work is necessary.

To change this command to a different scheduler (such as HPC), all you need to do is change the scheduler option.

For example, change `-lsf` to `-hpc` as follows: "`mpirun -prot -np 16 -hpc rank`"

To include additional scheduler options, use the appropriate **`-wlm`** option. Note that there are more WLM options than each scheduler supports. If you specify a WLM option that the scheduler does not support, the command silently ignores the option and will still create the job. This allows you to include a wide variety of options for all WLM-supported schedulers and not have to alter your command line command except for the scheduler option.

WLM support includes the following options:

- `-np` *number_of_ranks*

  Specifies the number of ranks to execute and the number of "units" to request for the job from the scheduler. The specific "units" will vary depending on the scheduler (such as slots for LSF or nodes/cores/sockets for HPC).
- `-wlmblock`

  Automatically schedules block ranks for HPC job size.
- `-wlmcyclic`

  Automatically schedules cyclic ranks for HPC job size.
- `-wlmwait`

  Waits until the job is finished before returning to the command prompt. For LSF, this implies the **`bsub -I`** command.
- `-wlmcluster` *cluster_name*

  Schedules jobs on the specified HPC cluster.
- `-wlmout` *file_name*

  Uses the specified file for the job **`stdout`** file location.
- `-wlmerr` *file_name*

  Uses the specified file for the job **`stderr`** file location.
- `-wlmin` *file_name*

  Uses the specified file for the job **`stdin`** file location.
- `-wlmproject` *project_name*

  Assigns the specified project name to the scheduled job.
- `-wlmname` *job_name*

  Uses the specified job name to the scheduled job.
- `-wlmsave`

  Configures the scheduled job to the scheduler without submitting the job.
- `-wlmjobtemplate` *job_template*

  Assigns the specified job template to the scheduled job.
- `-wlmnodegroups` *node_group* [`,`*nodegroup2 ...*]

  Assigns one or more specified node groups to the scheduled job.
- `-wlmpriority lowest | belowNormal | normal | aboveNormal | Highest`

  Assigns one or more specified node groups to the scheduled job.
- `-wlmunit core | socket | node`

Schedules ranks to the specified job resource unit type.

- `-wlmmaxcores` *units*

  Sets the maximum number of units that can be scheduled for the scheduled job.

- `-wlmmincores` *units*

  Sets the minimum number of units that can be scheduled for the scheduled job.

- `-wlmmaxmemory` *memsize*

  Sets the maximum memory size for the compute nodes for the job. The specific memory unit is defined by each scheduler. For example, HPC defines the memory size in MB.

- `-wlmminmemory` *memsize*

  Sets the minimum memory size for the compute nodes for the job. The specific memory unit is defined by each scheduler. For example, HPC defines the memory size in MB.

- `-wlmtimelimit` *time*

  Sets a time limit for the scheduled job. The specific unit of time is defined by each scheduler. For example, if the normal time limit for the specified scheduler is minutes, this specified time limit will also be in minutes.

WLM parameters are used for automatic job submission only. If used on an **mpirun** command within a job, the WLM parameters are ignored.

For example,

- To start an MPI job using 16 cores on an HPC sheduler:

  ```
  # mpirun -hpc -prot -np 16 rank
  ```

  Use the same command to start an MPI job using 16 slots on an LSF scheduler, but using the **-lsf** option:

  ```
  # mpirun -lsf -prot -np 16 rank
  ```

- To include an output file path and have the ranks cyclicly scheduled on HPC or LSF:

  ```
  # mpirun -hpc -prot -np 16 -wlmout out.txt -wlmcyclic rank
  # mpirun -lsf -prot -np 16 -wlmout out.txt -wlmcyclic rank
  ```

Platform MPI will construct the proper scheduler commands and submit the job to the scheduler. This also extends to other forms of creating node lists. Automatic submission to schedules supports the use of **-hostlist**, **-hostfile**, and **-f** *appfile*.

For example, if you have the following command without using a scheduler:

```
# mpirun -hostlist node1:2,node2:2,node3:3 -prot rank
```

Platform MPI will launch ranks 0/1 on node1, ranks 2/3 on node2, and ranks 3/4/5 on node3. The command starts remote processes using **ssh** for Linux and the Platform MPI Remote Launch Service for Windows.

If you wish to use the same command with a scheduler, all you need to do is add a scheduler option to the command and you can expect the same results:

```
# mpirun -lsf -hostlist node1:2,node2:2,node3:3 -prot rank
```

This command will schedule an LSF job and request nodes node1, node2, and node3. When the job executes, it will launch ranks 0/1 on node1, ranks 2/3 on

node2, and ranks 3/4/5 on node3. If the scheduler does not have access to compute nodes node1, node2, or node3, the submission will fail.

The same is done for **-hostlist** and **-f** *appfile*. For **-hostlist**, Platform MPI reads the hosts from a file and Platform MPI will request the specific resources from the host file. For **-f** *appfile*, Platform MPI reads the app file, builds a host list from the app file, and requests these resources for the job.

Although you cannot use the **-np** *number* option with the **-f** *appfile* option, you can use the **-np** *number* option with **-hostlist** and **-hostfile**. When used in combination, the resources are defined by **-hostlist** and **-hostfile**. However, the ranks started are defined by **-np** *number*. If there are more hosts than *number*, the job will be undersubscribed.

For example,

```
# mpirun -lsf -hostlist node1:4,node2:4 rank
```

Without **-np** *number*, six ranks are started: ranks 0 to 3 on node1, and ranks 4 to 7 on node2.

```
# mpirun -lsf -hostlist node1:4,node2:4 -np 5 rank
```

With **-np** 5 present, five ranks are started in a block fashion: ranks 0 to 3 on node1, and rank 4 on node2.

If the ranks are started by **-np** *number* and there are fewer hosts than *number*, the job will be oversubscribed.

For example,

```
# mpirun -lsf -hostlist node1:4,node2:4 -np 12 rank
```

With **-np** 12 present, 12 ranks are started: ranks 0 to 3 on node1, and ranks 4 to 7 on node2. After this, it will wrap around and start from the beginning again, therefore, it will start ranks 8 to 11 on node1. This wraparound functionality is similar to how **-hostlist** currently operates.

If you want to run the ranks cyclicly, you can accomplish this in the following two ways:

- `# mpirun -lsf -hostlist node1:4,node2:4 -wlmcyclic rank`

  This command will schedule ranks 0, 2, 4, and 6 on node1 and ranks 2, 3, 5, and 7 on node2.
- `# mpirun -lsf -hostlist node1,node2 -np 8 rank`

  This command will accomplish the same goal, but by wrapping around the resource list when block allocating.

There are many options when scheduling jobs; however, automatic job submission should schedule jobs in the same fashion as non-scheduler jobs when using **-hostlist**, **-hostfile**, and **-f** *appfile*. This method of scheduling may not be the best way to utilize scheduler resources, but it is an efficient way to schedule specific resources when needed.

The recommended method is still to let the scheduler select resources and to keep it simple by using a scheduler option and **-np** *number*, for example:

```
# mpirun -np 48 -lsf rank
```

Output files

When submitting jobs using automatic submission, if you do not specify an output file using **-wlmout**, the command assigns one using the *rank* base file name with the job ID appended and an `.out` extension. The command uses the same file name convention for error files, but with an `.err` extension. For example, if you use "`mpirun -np 48 -lsf rank`" , the results are sent to `rank-jobid.out` and stderr output is sent to `rank-jobid.err`

Listing environment variables

Use the **-envlist** option to list environment variables that are propagated to all MPI ranks from the existing environment.

```
-envlist env1[,env2,...]
```

For example,

```
# export EXAMPLE_ENV1=value1
# export EXAMPLE_ENV2=value2
# mpirun ... -envlist EXAMPLE_ENV1,EXAMPLE_ENV2 ... rank
```

The three previous commands are equivalent to the following command:

```
# mpirun ... -e EXAMPLE_ENV1=value1 -e EXAMPLE_ENV2=value2 ... rank
```

This allows the use of "short hand" to propagate existing variables in the current shell environment to the ranks.

Installing Platform MPI

Platform MPI is packaged using InstallAnywhere to provide a common installer for both Linux and Windows platforms. The installers are 32-bit executables bundled with IBM's 32-bit JRE, and are run as follows:

- Linux: `./platform_mpi-09.1.2.0r.x64.bin` (run as root)
- Windows: `platform_mpi-09.1.2.0-rc8.x64.exe` (run as a user with Administrator privileges)

For more information on the command line options supported by the installer, run the installer with the single argument `--help`.

Installer modes

The installer provides the following installation modes to suit different requirements:

Graphical user interface (GUI)

The GUI-based installation is used by default (or by explicitly specifying the `-i swing` option) when running the installer.

Before running the installer in Linux, you must ensure that your `DISPLAY` environment is set up correctly.

Console

The console or text-based installation behaves the same as the GUI-based installer, but is run in text-only mode. Use the console installer by specifying the `-i console` option when running the installer.

Silent Install in silent mode if you wish to use all of the defaults and to accept the license agreement ahead of time at command invoation time. Use the installer in silent mode by specifying the `-i silent` option when running the installer.

Installation sets

The Linux installer uses a single installation set and installs all of the files at every installation.

The Windows installer has different installation sets based on how the Platform MPI service is run:

Service mode (default)
In service mode, Platform MPI installs its service to run at boot time. This service is used at launch time to launch MPI ranks. Selecting this mode will also prompt for port information.

Service only
Use this installation set if you already installed Platform MPI onto a shared location but need to install the service on each node of a cluster to launch MPI ranks.

HPC This installs Platform MPI without installing the service. This is useful for Windows HPC, which uses Windows HPC to launch MPI ranks.

Using a response file for unattended installations with non-default options

If you would like to install Platform MPI with non-default options (such as a non-default location) on many nodes, run the installer on one node and gather the responses of the installer to use as input to the installer for all of the other nodes. To do this, the installer supports generating a response file.

To generate a response file on the first node, specify `-r` *`response_file`* with either `-i "console"` or `-i "swing"` options as arguments to the installer. The installer recognizes that there are no response files in the specified location and will create a new file.

After completing the installation and generating a response file, use the same `-r` *`response_file`* option with `-i "silent"` as arguments to the installer. The installer recognizes that a response file already exists and will use that as input for the installation. This provides a mechanism for you to specify non-default arguments to the installer across many installations.

Uninstalling Platform MPI

To uninstall Platform MPI, run the installer in the following location:

- Linux: `$MPI_ROOT/_IBM_PlatformMPI_installation/Change\ IBM_PlatformMPI\ Installation`

  where `$MPI_ROOT` is the top-level installation directory (`/opt/ibm/platform_mpi/` by default).
- Windows: `"%MPI_ROOT%\_IBM_PlatformMPI_installation\Change IBM_PlatformMPI Installation"`

  where `%MPI_ROOT%` is the top-level installation directory (`C:\Program Files(x86)\IBM\Platform-MPI\` by default).

The installer remembers which installation mode was used (GUI, Console, or Silent) and uses the same mode to uninstall Platform MPI. To explicitly specify a mode, use the `-i` option (`-i "swing" | "console" | "silent"`).

Known issues

For more details on known issues with the installer, refer to "Installer GUI mode on Redhat 6.x or Suse 11.x" and "Installer might not detect previous versions when installing to the same location."

Known issues

Event-based progression (-nospin) requires -e MPI_TCP_POLL=1 to be set

When specifying `-nospin` to use event-based progression, you must also specify `-e MPI_TCP_POLL=1`. This requirement will be removed in a future version.

Installer GUI mode on Redhat 6.x or Suse 11.x

The GUI mode may not work on Redhat 6.x or Suse 11.x. These distributions are missing the following required 32-bit compatibility packages:

- `xulrunner.i686`
- `libXp*.i686`
- `libXt*.i686`

The installer will detect the missing packages and fall back to the console installer.

If you prefer to use the GUI installer, install the above packages using the package manager for your distribution. For example:

```
sudo yum install xulrunner.i686
sudo yum install libXp*.i686
sudo yum install libXt*.i686
```

Installer might not detect previous versions when installing to the same location

When upgrading from a previous version of Platform MPI in the same location, the installer may not detect the old version. When installing Platform MPI to the same location as the old version, you must first uninstall the old version before installing the new version.

Pinning shared memory and lazy deregistration

Applications that allocate and release memory using mechanisms other than **`munmap`** or use of the **`malloc`** library must either turn off the lazy deregistration features (using `-ndd` on the **`mpirun`** command line) or invoke a Platform MPI callback function whenever memory is released. For more details, refer to "Alternate lazy deregistration" on page 8.

MPI_Status field shows 0 bytes received when using IBV-to-TCP failover

When using the IBV-to-TCP failover feature (`-e PCMPI_IBV2TCP_FAILOVER=1`), there is a known issue in which the MPI_Status field for message length of a restarted

MPI_Recv call may show 0 bytes received instead of the actual amount of data received.. If an application does not use the MPI_Status field on MPI_Recv calls, or does not use long messages (as defined by **MPI_RDMA_MSGSIZE**), this will not impact the application.

IBV-to-TCP failover is not supported with `-1sided`. IBV-to-TCP failover only supports the default setting of **MPI_RDMA_MSGSIZE**, therefore, do not modify **MPI_RDMA_MSGSIZE** when using **PCMPI_IBV2TCP_FAILOVER**.

High availability mode does not support certain collective operations

The use of the high availability mode (`-ha[:`*options*`]`) forces the use of particular collective operations that are adapted to comply with the requirements of running in high availability mode. Therefore, selecting specific collective operations has no affect when running in this mode. For example, selecting a reduce operation that ensures a repeatable order of operations (`-e MPI_COLL_FORCE_ALLREDUCE=10`) has no affect and will be silently ignored.

System benchmarking tools require the single-threaded library

The System Check example application (`$MPI_ROOT/help/system_check.c`) can only be compiled and used with the single-threaded Platform MPI library. Using the System Check application with the multi-threaded library will produce the following error message and the job will exit early:

```
syschk/tools requested but not available in this mode.
```

This restriction applies to any use of the multi-threaded library. That is, both the compile time option `-lmtmpi` and the run time option `-entry=mtlib` will trigger the error message.

Similarly, the **mpitool** utility (`$MPI_ROOT/bin/mpitool`) can only be used with the single-threaded Platform MPI library.

To work around this restriction, use the single-threaded library with the System Check example application or **mpitool** utility.

New MPI 3.0 non-blocking collectives no longer supported

New MPI 3.0 non-blocking collectives are no longer supported due to a hang or mismatched traffic. Support for these collectives will be restored in a future release.

MPI_ANY_SOURCE requests using -ha

Using `-ha`, MPI_ANY_SOURCE requests that return MPI_ERR_PENDING will not match messages until the user acknowledges the failure with an **MPIHA_Comm_failure_ack ()**.

Connect/accept using multi-threaded library

If two multi-threaded MPI processes simultaneously attempt to call **MPI_Connect** to each other at the same time, this can potentially cause a hang. This is a known issue and will be fixed in a future release.

Applications cannot create more than 3200 COMMS

Platform MPI 8.3 applications are able to create more than 12000 COMMs before running out of special memory used for COMM creation. For Platform MPI 9.1 applications, this is temporarily reduced to approximately 3200 COMMs. This should not affect any users. The ability to create a larger number of COMMs will be restored in a future release or Fix Pack.

On-demand connections cannot be used with one-sided communication

On-demand connections (**PCMP_ONDEMAND_CONN=1**) cannot be used with one-sided communication (`-1sided`). If this combination is used, on-demand connections will be turned off and a warning is issued.

wlm-lsf.so open error or liblsf.so not found

When using Platform LSF with **mpirun**, the MPI job fails to start and outputs one of the following errors:

- `wlm-lsf.so open error`
- `liblsf.so not found`

When using Platform LSF, Platform MPI uses `liblsf.so` in its environment. Most installations of LSF include the `LSF_LIBDIR` path in the user's `LD_LIBRARY_PATH`. However, some legacy LSF environments (such as LSF Uniform-Path) do not include `LSF_LIBDIR` in `LD_LIBRARY_PATH`, nor is the `LSF_LIBDIR` environment variable defined outside an LSF job. Because Platform MPI depends on `liblsf.so` when using Platform MPI LSF options (for example, **-lsf**, **-lsb_mcpu_hosts**), having `LSF_LIBDIR` in the `LD_LIBRARY_PATH` is necessary.

If users are having problems with Platform MPI and errors loading `wlm-lsf.so` or `liblsf.so`, check that `LSF_LIBDIR` is defined their environment and included in `LD_LIBRARY_PATH`. Because each LSF installation varies, users need to contact their system administrators to determine the correct `LSF_LIBDIR` path if this is not defined in their environment. Refer to the *Platform LSF Configuration Reference* guide, and the sections regarding `cshrc.lsf` and `profile.lsf` for more information on the LSF environment setup and `LSF_LIBDIR`.

As an alternative, users can issue a **bsub** command with the appropriate **mpirun** commands as part of the **bsub** command. Users may need to construct their hostlist/appfile without referencing Platform MPI LSF flags on the **mpirun** command (such as **-lsf**).

RDMA memory pin or registration error with OFED 1.5.3

OFED 1.5.3 changed the default number of memory regions that can be registered with the driver at any given time. On large clusters, this can lead to two different error messages from Platform MPI:

- `MPI_Init: hpmp_rdmaregion_alloc() failed`
- `ibv_reg_mr() failed`

To resolve this issue, it may be necessary to increase the amount of allowable registered memory in the driver parameters. To do this, add the following line to `/etc/modprobe.conf`:

```
options mlx4_core log_num_mtt=24
```

Allgather/Allgatherv

Because of some issues with Allgather/Allgatherv that could not be resolved at this time, FCA 2.0 Allgather/Allgatherv support for FCA has been disabled by default. The problem occurs when using Allgather with `MPI_IN_PLACE` and non-contigious data. This issue is rare, but the error results in wrong answers.

If you wish to use FCA Allgather/Allgatherv, set the following environment variable: `PCMPI_FCA_ENABLE_ALLGATHER=1`. This will allow the FCA Allgather/Algatherv algorithms to be selected and run under the same conditions as other FCA algorithms (always if forced, only if profitable for "normal" enabled FCA).

Diagnostic library

The diagnostic library does not call all the optimized collective algorithms available, but instead uses the "failsafe" algorithms.

Running on iWarp hardware

- When running on iWARP hardware, you might see messages similar the following when applications exit:

  ```
  disconnect: ID 0x2b65962b2b10 ret 22
  ```

  This is a debugging message that prints erroneously from the uDAPL library and can be ignored. The message can be completely suppressed by passing the **`-e DAPL_DBG_TYPE=0`**option to **`mpirun`**. Alternatively, you can set **`DAPL_DBG_TYPE=0`** in the `$MPI_ROOT/etc/hpmmpi.conf` file to avoid having to pass the option on the **`mpirun`** command line.
- Users might see the following error during launches of Platform MPI applications on Chelsio iWARP hardware:

  ```
  Rank 0:0: MPI_Init: dat_evd_wait()1 unexpected event number 16392
  Rank 0:0: MPI_Init: MPI BUG: Processes cannot connect to rdma device
  MPI Application rank 0 exited before MPI_Finalize() with status 1
  ```

  To prevent these errors, Chelsio recommends passing the `peer2peer=1` parameter to the `iw_cxgb3` kernel module. This is accomplished by running the following commands as root on all nodes:

  ```
  # echo "1" > /sys/module/iw_cxgb3/parameters/peer2peer
  # echo "options iw_cxgb3 peer2peer=1" >> /etc/modprobe.conf
  ```

  The second command is optional and makes the setting persist across a system reboot.
- Users of iWARP hardware might see errors similar to the following:

  ```
  dapl async_event QP (0x2b27fdc10d30) ERR 1
  dapl_evd_qp_async_error_callback() IB async QP err - ctx=0x2b27fdc10d30
  ```

  Previous versions of HP-MPI required passing`-e MPI_UDAPL_MSG1=1` on some iWARP hardware. As of Platform MPI 9.1.2, no iWARP implementations are known to require this setting, and you must remove it from all scripts unless otherwise instructed.

OFED firmware

The OpenFabrics Alliance (OFA) documents minimum supported firmware revisions for a variety of InfiniBand adapters. If unsupported firmware is used, Platform MPI might experience issues with abnormal application teardown or other problems.

Spawn on remote nodes

Codes which call either **MPI_Comm_spawn()** or **MPI_Comm_spawn_multiple()** might not be able to locate the commands to spawn on remote nodes. To work around this issue, you can either specify an absolute path to the commands to spawn, or users can set the **MPI_WORKDIR** environment variable to the path of the command to spawn.

Default interconnect for -ha option

The–ha option in previous HP-MPI releases forced the use of TCP for communication. Both IBV and TCP are possible network selections when using–ha. If no forced selection criteria (for example,–TCP,–IBV, or equivalent **MPI_IC_ORDER** setting) is specified by the user, then IBV is selected where it is available. Otherwise, TCP is used.

Linking without compiler wrappers

To support the-dd (deferred deregistration) option, Platform MPI must intercept calls to glibc routines that allocate and free memory. The compiler wrapper scripts included with Platform MPI attempt to link MPI applications to make this possible. If you choose not to link your application with the provided compiler wrappers, you must either ensure that libmpi.so precedes libc.so on the linker command line or specify "-e LD_PRELOAD=%LD_PRELOAD:libmpi.so" on the **mpirun** command line.

Locating the instrumentation output file

Whether **mpirun** is invoked on a host where at least one MPI process is running or on a host remote from all MPI processes, Platform MPI writes the instrumentation output file prefix.instr to the working directory on the host that is running rank 0 (when instrumentation for multihost runs is enabled). When using-ha, the output file is located on the host that is running the lowest existing rank number at the time the instrumentation data is gathered during **MPI_Finalize()**.

Using the ScaLAPACK library

Prebuilt applications that are linked against ScaLAPACK must use the Platform MPI MPICH compatibility mode. When the application is built, mpicc.mpich or mpif77.mpich or mpif90.mpich must be used. At runtime, **mpirun.mpich** must be used to launch the application. Mpich compatibility mode is not a requirement for users compiling ScaLAPACK and applications that use ScaLAPACK.

Increasing shared memory segment size

Platform MPI uses shared memory for communications between processes on the same node and might attempt to allocate a shared-memory segment that is larger than the operating system allows. The most common issue you might experience is an error message like:

```
Cannot create shared memory segment of <size> bytes.
```

To increase the maximum allowed shared memory segment size, enter the following command as root:

```
# /sbin/sysctl -w kernel.shmmax=<size in bytes>
```

To make changes to the kernel.shmmax setting persist across a reboot, add the following line to the `/etc/sysctl.conf` file:

```
kernel.shmmax=<size in bytes>
```

Using MPI_FLUSH_FCACHE

The **MPI_FLUSH_FCACHE** environment variable is silently ignored if the `-cpu_bind` option is not specified. This limitation will be removed in a future release. See the mpienv(1) manpage for more information.

Using MPI_REMSH

Platform MPI uses the **$MPI_REMSH** command to launch on remote machines. The commands constructed are of the general form "`$MPI_REMSH <host> -n <command>`". If a remote shell command is desired for which the`-n` in the above syntax is not appropriate, you can use a shell script such as the following for the **MPI_REMSH** command:

```
#!/bin/sh
    host="$1"
    shift
    shift
    ssh $host "$@"
```

Increasing pinned memory

InfiniBand requires pages to be pinned (locked in memory) for message passing. This can become a problem when a child process is forked and a pinned page exists in both the parent's and child's address spaces. Normally a copy-on-write would occur when one of the processes touches memory on a shared page, and the virtual to physical mapping would change for that process. In the context of InfiniBand, such a change in the mapping results in data corruption when an RDMA sends data to the original physical address.

OFED 1.2 and later (with a fork safety mode enabled) avoids this problem by not using copy-on-write behavior during a fork for pinned pages. Instead, any access to these pages by the child process results in a segmentation violation of the child, and the parent's mapping remains unchanged so that the parent can continue running normally with no data corruption.

Platform MPI turns on this option by default when the IBV or uDAPL protocols are being used. If the fork safety mode is not desired, you can turn it off with the MPI environment variable **MPI_IBV_NO_FORK_SAFE=1**.

By setting the environment variable **MPI_PAGE_ALIGN_MEM=1**, Platform MPI page-aligns and page-pads libc memory allocation requests that are large enough to be pinned during MPI message transfer. This results in slightly more memory being allocated, but reduces the likelihood that a forked process writes to a page of memory that was also being used for message transfer when a fork call occurred.

Disabling fork safety

Applications running on Linux systems with kernels earlier than 2.6.12 might display the following warning message:

```
libibverbs: Warning: fork()-safety requested but init failed
```

This warning message appears because the Platform MPI library enabling the OFED 1.2 fork safety feature is not supported by Linux kernels earlier than 2.6.12. It does not affect the application run. To disable Platform MPI fork safety, set the environment variable **MPI_IBV_NO_FORK_SAFE**, as in the following example:

```
# opt/platform_mpi/bin/mpirun -np 4 -prot -e MPI_IBV_NO_FORK_SAFE=1 \
-hostlist nodea,nodeb,nodec,noded /my/dir/hello_world
```

Using fork with OFED

Applications using **fork()** might crash on configurations with InfiniBand using OFED on kernels earlier than v2.6.18. You can avoid known problems with **fork()** and OFED in any of the following ways:

- Run on a system with kernel 2.6.18 or later.
- Run InfiniBand with non-OFED drivers. (This option is not available on configurations with ConnectX InfiniBand Host Channel Adapters where OFED is required.)

Memory pinning with OFED 1.2

The initial release of OFED 1.2 contains a bug that causes the memory pinning function to fail after certain patterns of **malloc** and **free**. The symptom, which is visible from Platform MPI, might be any of several error messages such as:

```
> prog.x: Rank 0:1: MPI_Get: Unable to pin memory for put/get
```

This bug has already been fixed in OFED 1.3, but if you are running with the initial release of OFED 1.2, the only workaround is to set **MPI_IBV_NO_FORK_SAFE=1**.

Upgrading to OFED 1.2

When upgrading to OFED 1.2 from earlier versions, the installation script might not stop the previous OFED version before uninstalling it. Therefore, stop the old OFED stack before upgrading to OFED 1.2. For example:

```
# /etc/init.d/openibd stop
```

Increasing the nofile limit

The nofile limit on large Linux clusters needs to be increased in `/etc/security/limits.conf`

```
* soft nofile 1024
```

For larger clusters, Platform recommends a setting of at least:

- 2048 for clusters of 1900 cores or fewer
- 4096 for clusters of 3800 cores or fewer
- 8192 for clusters of 7600 cores or fewer

MPI_Issend call limitation on Myrinet MX

Some earlier versions of Myrinet MX have a known resource limitation involving outstanding **MPI_Issend()** calls. If more than 128 **MPI_Issend()** calls are issued and not yet matched, further MX communication can hang. The only known workaround is to have your application issue less than 128 unmatched **MPI_Issend()** calls at a time. This limitation is fixed in versions 1.1.8 and later.

Terminating shells

When a foreground Platform MPI job is run from a shell window, if the shell is terminated, the shell sends signal SIGHUP to the **mpirun** process and its underlying ssh processes, thus killing the entire job.

When a background Platform MPI job is run and the shell is terminated, the job might continue depending on the actual shell used. For `/bin/bash`, the job is killed. For `/bin/sh` and `/bin/ksh`, the job continues. If `nohup` is used when launching the job, only background ksh jobs can continue. This behavior might vary depending on your system.

libpthread dependency

The Platform MPI 9.1.2 library for Linux contains a dependency which requires **libpthread**. The `mpicc`, `mpif90`, etc. compiler wrapper scripts automatically add the necessary`-lpthread`, but manually linked applications must explicitly add `-lpthread`.

Fortran calls wrappers

Profiling routines built for C calls do not cause the corresponding Fortran calls to be wrapped automatically. To profile Fortran routines, you must write separate wrappers for the Fortran calls.

Bindings for C++ and Fortran 90

Platform MPI complies with the MPI-1.2 standard, which defines bindings for Fortran 77 and C, but not Fortran 90 or C++. Platform MPI also complies with the C++ binding definitions detailed in the MPI-2 standard. The MPI-2.2 standard has deprecated the C++ bindings and they will be removed in a future release. The C++ bindings are not thread safe and should not be used with the Platform MPI threaded libraries (for example, libmtmpi). Platform MPI does not provide bindings for Fortran 90. Some features of Fortran 90 might interact with MPI non-blocking semantics to produce unexpected results. For details, see the *Platform MPI User's Guide*.

Using TotalView

To use the`-tv` option to **mpirun**, the TotalView binary must be in the user's `PATH`, or the **TOTALVIEW** environment variable must be set to the full path of the TotalView binary.

```
% export TOTALVIEW=/usr/toolworks/totalview/bin/totalview
```

Platform MPI 9.1.2 includes the `$MPI_ROOT/bin/tv_launch` script to be used to instruct TotalView to attach to the Platform MPI job, rather than launching through TotalView. To enable this alternate debugging method with TotalView, ensure that

the `totalview` executable is in your path on all compute nodes, and set the TOTALVIEW environment variable to point to this script rather than to the `totalview` executable. Launch your job through `mpirun -tv` as normal. Each rank will be paused in `MPI_Init`.

Extended collectives with lightweight instrumentation

Extended collectives with intercommunicators are not profiled by the Platform MPI lightweight instrumentation mode.

Using high availability with diagnostic library

You cannot use high availability (`-ha`) mode and the diagnostic library simultaneously.

Using MPICH with diagnostic library

You cannot use MPICH mode and the diagnostic library simultaneously.

Using high availability with MPICH

High availability (`-ha`) mode has not been tested with MPICH mode.

Using MPI-2 with diagnostic library

The diagnostic library strict mode is not compatible with some MPI-2 features.

Additional product information

Product documentation

Additional product documentation:

- Platform MPI manpages are installed in`$MPI_ROOT/share/man`

Table 11. Manpage Categories

Category	Manpages	Description
General	MPI.1	The general features of Platform MPI
Compilation	• mpicc.1 • mpiCC.1 • mpif77.1 • mpif90.1	The available compilation utilities
Runtime	• mpiclean.1 • mpidebug.1 • mpienv.1 • mpiexec.1 • mpijob.1 • mpimtsafe.1 • mpirun.1 • mpistdio.1 • autodbl.1 • system_check.1	runtime utilities, environment variables, debugging, thread-safe, and diagnostic libraries

Product packaging

Platform MPI is packaged as an optional software product installed in `/opt/ibm/platform_mpi` by default.

Software availability in native languages

Platform MPI only supports the English language, however, users may translate and replace `nls/msg/C/hpmpi.cat` as required.

Notices

This information was developed for products and services offered in the U.S.A.

IBM® may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web

sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Intellectual Property Law
Mail Station P300
2455 South Road,
Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application

programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java™ and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

LSF®, Platform, and Platform Computing are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Printed in USA

GI13-1896-01