

Intel® MPI Benchmarks

User Guide and Methodology Description

Contents

Legal Information	6
1. Introduction	7
1.1. Introducing Intel® MPI Benchmarks	7
1.2. What's New.....	7
1.2.1. Changes in Intel® MPI Benchmarks 2017 Update 2	7
1.2.2. Changes in Intel® MPI Benchmarks 2017 Update 1	7
1.2.3. Changes in Intel® MPI Benchmarks 2017	8
1.2.4. Changes in Intel® MPI Benchmarks 4.1 Update 1.....	8
1.2.5. Changes in Intel® MPI Benchmarks 4.1	8
1.2.6. Changes in Intel® MPI Benchmarks 4.0	8
1.3. About this Document.....	8
1.4. Getting Help and Support.....	9
1.5. Related Information	9
2. Installation and Quick Start.....	10
2.1. Memory and Disk Space Requirements.....	10
2.2. Software Requirements	11
2.3. Installing Intel® MPI Benchmarks.....	11
2.4. Building Intel® MPI Benchmarks	11
2.4.1. Linux* OS.....	11
2.4.2. Windows* OS.....	12
2.5. Running Intel® MPI Benchmarks	14
2.6. Running Benchmarks in Multiple Mode.....	14
3. MPI-1 Benchmarks.....	16
3.1. Classification of MPI-1 Benchmarks.....	17
3.1.1. Single Transfer Benchmarks.....	18
3.1.2. Parallel Transfer Benchmarks.....	18
3.1.3. Collective Benchmarks.....	18
3.2. Single Transfer Benchmarks.....	18
3.2.1. PingPong, PingPongSpecificSource, PingPongAnySource	19
3.2.2. PingPing, PingPingSpecificSource, PingPingAnySource.....	20
3.3. Parallel Transfer Benchmarks.....	21
3.3.1. Sendrecv.....	21
3.3.2. Exchange	22
3.3.3. Uniband	23
3.3.4. Biband	23
3.4. Collective Benchmarks.....	24
3.4.1. Reduce.....	24
3.4.2. Reduce_scatter	25
3.4.3. Allreduce	25
3.4.4. Allgather	26
3.4.5. Allgatherv.....	26
3.4.6. Scatter	26
3.4.7. Scatterv	27
3.4.8. Gather	27

3.4.9. Gatherv	27
3.4.10. Alltoall	28
3.4.11. Bcast	28
3.4.12. Barrier	28
4. MPI-2 Benchmarks	30
4.1. Naming Conventions	32
4.2. IMB-MPI-2 Benchmark Classification	32
4.2.1. Single Transfer Benchmarks	33
4.2.2. Parallel Transfer Benchmarks	33
4.2.3. Collective Benchmarks	33
4.2.4. MPI-2 Benchmarks Classification	33
4.3. MPI-2 Benchmark Modes	34
4.3.1. Assured Completion of Transfers	35
4.4. IMB-EXT Benchmarks	35
4.4.1. Unidir_Put	35
4.4.2. Unidir_Get	36
4.4.3. Bidir_Put	37
4.4.4. Bidir_Get	37
4.4.5. Accumulate	38
4.4.6. Window	39
4.5. IMB-IO Blocking Benchmarks	40
4.5.1. S_[ACTION]_indv	41
4.5.2. S_[ACTION]_expl	42
4.5.3. P_[ACTION]_indv	43
4.5.4. P_ACTION_expl	44
4.5.5. P_[ACTION]_shared	45
4.5.6. P_[ACTION]_priv	47
4.5.7. C_[ACTION]_indv	48
4.5.8. C_[ACTION]_expl	48
4.5.9. C_[ACTION]_shared	49
4.5.10. Open_Close	49
4.6. IMB-IO Non-blocking Benchmarks	50
4.6.1. Exploiting CPU	50
4.6.2. Displaying Results	51
5. MPI-3 Benchmarks	52
5.1. IMB-NBC Benchmarks	52
5.1.1. Measuring Communication and Computation Overlap	53
5.1.2. Measuring Pure Communication Time	53
5.1.3. lallgather	54
5.1.4. lallgather_pure	54
5.1.5. lallgatherv	55
5.1.6. lallgatherv_pure	55
5.1.7. lallreduce	55
5.1.8. lallreduce_pure	56
5.1.9. lalltoall	56
5.1.10. lalltoall_pure	57
5.1.11. lalltoallv	57
5.1.12. lalltoallv_pure	57
5.1.13. lbarrier	58

5.1.14. lbarrier_pure	58
5.1.15. lbcast	58
5.1.16. lbcast_pure	59
5.1.17. lgather	59
5.1.18. lgather_pure	59
5.1.19. lgatherv	60
5.1.20. lgatherv_pure	60
5.1.21. lreduce	61
5.1.22. lreduce_pure	61
5.1.23. lreduce_scatter	61
5.1.24. lreduce_scatter_pure	62
5.1.25. lscatter	63
5.1.26. lscatter_pure	63
5.1.27. lscatterv	63
5.1.28. lscatterv_pure	64
5.2. IMB-RMA Benchmarks	64
5.2.1. IMB-RMA Benchmark Modes	64
5.2.2. Classification of IMB-RMA Benchmarks	65
5.2.3. Accumulate	66
5.2.4. All_get_all	66
5.2.5. All_put_all	67
5.2.6. Bidir_get	67
5.2.7. Bidir_put	67
5.2.8. Compare_and_swap	68
5.2.9. Exchange_Get	68
5.2.10. Exchange_Put	68
5.2.11. Fetch_and_op	69
5.2.12. Get_accumulate	69
5.2.13. Get_all_local	69
5.2.14. Get_local	70
5.2.15. One_put_all	70
5.2.16. One_get_all	71
5.2.17. Put_all_local	71
5.2.18. Put_local	71
5.2.19. Truly_passive_put	72
5.2.20. Unidir_get	72
5.2.21. Unidir_put	72
6. Benchmark Methodology	74
6.1. Control Flow	74
6.2. Command-line Control	74
6.2.1. Benchmark Selection Arguments	75
6.2.2. -npmin Option	76
6.2.3. -multi Option	76
6.2.4. -off_cache cache_size[,cache_line_size] Option	77
6.2.5. -iter Option	78
6.2.6. -iter_policy Option	78
6.2.7. -time Option	79
6.2.8. -mem Option	79
6.2.9. -input <File> Option	79
6.2.10. -msglen <File> Option	80

6.2.11. -map PxQ Option	80
6.2.12. -include [[benchmark1] benchmark2 ...]	83
6.2.13. -exclude [[benchmark1] benchmark2 ...]	83
6.2.14. -msglog [<minlog>:]<maxlog>	83
6.2.15. -thread_level Option	84
6.2.16. -sync Option	84
6.2.17. -root_shift Option	84
6.2.18. -imb_barrier Option	84
6.3. Parameters Controlling Intel® MPI Benchmarks	85
6.4. Hard-Coded Settings	87
6.4.1. Communicators, Active Processes	87
6.4.2. Message/I-O Buffer Lengths	87
6.4.3. Buffer Initialization	87
6.4.4. Other Preparations for Benchmarking	88
6.4.5. Warm-Up Phase (IMB-MPI1, IMB-EXT, IMB-NBC, and IMB-RMA)	89
6.4.6. Synchronization	89
6.4.7. Actual Benchmarking	89
6.5. Checking Results	90
6.6. Output	91
6.6.1. Sample 1 - IMB-MPI1 PingPong Allreduce	91
6.6.2. Sample 2 - IMB-MPI1 PingPing Allreduce	93
6.6.3. Sample 3 - IMB-IO p_write_indv	95
6.6.4. Sample 4 - IMB-EXT.exe	97

Legal Information

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

* Other names and brands may be claimed as the property of others.

Intel, Xeon Phi and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

© Intel Corporation.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

1. Introduction

1.1. Introducing Intel® MPI Benchmarks

Intel® MPI Benchmarks performs a set of MPI performance measurements for point-to-point and global communication operations for a range of message sizes. Intel® MPI Benchmarks is developed using ANSI C plus standard MPI. It is distributed as an open source project to enable use of benchmarks across various cluster architectures and MPI implementations.

The generated benchmark data fully characterizes:

- Performance of a cluster system, including node performance, network latency, and throughput
- Efficiency of the MPI implementation used

The Intel® MPI Benchmarks package consists of the following components:

- **IMB-MPI1** – benchmarks for MPI-1 functions.
- Two components for **MPI-2** functionality:
 - IMB-EXT – one-sided communications benchmarks.
 - IMB-IO – input/output (I/O) benchmarks.
- Two components for **MPI-3** functionality:
 - IMB-NBC – benchmarks for non-blocking collective (NBC) operations.
 - IMB-RMA – one-sided communications benchmarks. These benchmarks measure the Remote Memory Access (RMA) functionality introduced in the MPI-3 standard.

Each component constitutes a separate executable file. You can run all of the supported benchmarks, or specify a single executable file in the command line to get results for a specific subset of benchmarks.

If you do not have the MPI-2 or MPI-3 extensions available, you can install and use IMB-MPI1 that uses only standard MPI-1 functions.

1.2. What's New

This section provides changes for the Intel® MPI Benchmarks as compared to the previous versions of this product.

1.2.1. Changes in Intel® MPI Benchmarks 2017 Update 2

- Extended description for the `-multi` and `-map` options. See [Command-Line Control](#).

1.2.2. Changes in Intel® MPI Benchmarks 2017 Update 1

This release includes the following updates as compared to the Intel® MPI Benchmarks 2017:

- Added a new option `-imb_barrier`.
- The **PingPong** and **PingPing** benchmarks are now equivalent to `PingPongSpecificSource` and `PingPingSpecificSource`, respectively. Their old behavior (with `MPI_ANY_SOURCE`) is available in `PingPongAnySource` and `PingPingAnySource`.

1.2.3. Changes in Intel® MPI Benchmarks 2017

This release includes the following updates as compared to the Intel® MPI Benchmarks 4.1 Update 1:

- Changed default values for the `-sync` and `-root_shift` options. See [Command-Line Control](#).
- Support for the Microsoft* Visual Studio* 2015. Microsoft* Visual Studio* 2010 support is removed.
- Minor improvements and bug fixes.

1.2.4. Changes in Intel® MPI Benchmarks 4.1 Update 1

This release includes the following updates as compared to the Intel® MPI Benchmarks 4.1:

- Minor improvements and bug fixes.

1.2.5. Changes in Intel® MPI Benchmarks 4.1

This release includes the following updates as compared to the Intel® MPI Benchmarks 4.0:

- Introduced new benchmarks: `Uniband` and `Biband`. See [Parallel Transfer Benchmarks](#).
- Introduced new options: `-sync` and `-root_shift`. See [Command-Line Control](#).

1.2.6. Changes in Intel® MPI Benchmarks 4.0

This release includes the following updates as compared to the Intel® MPI Benchmarks 3.2.4:

- Introduced new components `IMB-NBC` and `IMB-RMA` that conform to the MPI-3.0 standard.
- Introduced a new feature to set the appropriate policy for automatic calculation of iterations. You can set the policy using the `-iter` and `-iter_policy` options.
- Added new targets to the Linux* OS Makefiles:
 - `NBC` for building `IMB-NBC`
 - `RMA` for building `IMB-RMA`
- Updated Microsoft* Visual Studio* solutions to include the `IMB-NBC` and `IMB-RMA` targets.
- Support for the Microsoft* Visual Studio* 2013. Microsoft* Visual Studio* 2008 support is removed.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

1.3. About this Document

This User's Guide provides a complete description of the Intel® MPI Benchmarks, including usage information and detailed description of each benchmark.

The following conventions are used in this document:

Style	Description
<code>This type style</code>	Commands, arguments, options, file names
<code>THIS_TYPE_STYLE</code>	Environment variables
<code><this type style></code>	Placeholders for actual values
<code>[items]</code>	Optional items
<code>{ item item }</code>	Selectable items separated by vertical bar(s)

1.4. Getting Help and Support

Your feedback is very important to us. To receive technical support for the tools provided in this product and technical information including FAQ's and product updates, you need to register for an Intel® Premier Support account at the Registration Center.

This package is supported by Intel® Premier Support. Direct customer support requests at:
<https://premier.intel.com>

General information on Intel® product-support offerings may be obtained at:
<http://www.intel.com/software/products/support>

Intel® MPI Benchmarks home page can be found at: <http://www.intel.com/go/imb>

When submitting a support issue to Intel® Premier Support, please provide specific details of your problem, including:

- The Intel® MPI Benchmarks package name and version information.
- Host architecture (for example, Intel® 64 architecture).
- Compiler(s) and versions.
- Operating system(s) and versions.
- Specifics on how to reproduce the problem. Include makefiles, command lines, small test cases, and build instructions.

1.5. Related Information

For more information, you can see the following related resources:

- [Intel® MPI Benchmarks Download Page](#)
- [Intel® MPI Library Product](#)

2. Installation and Quick Start

This section explains how to install and start using the Intel® MPI Benchmarks.

2.1. Memory and Disk Space Requirements

The table below lists memory requirements for benchmarks run with the default settings (standard mode) and with the user-defined settings (optional mode). In this table:

- Q is the number of active processes.
- X is the maximal size of the passing message.

Benchmarks	Standard Mode	Optional Mode
Alltoall	Q*8 MB	Q*2X bytes
Allgather, Allgatherv	(Q+1)*4 MB	(Q+1)*X bytes
Exchange	12 MB	3X bytes
All other MPI-1 benchmarks	8 MB	2X bytes
IMB-EXT	80 MB	2 max (X, OVERALL_VOL) bytes
IMB-IO	32 MB	3X bytes
Ialltoall, Ialltoall_pure	Q*8 MB	Q*2X bytes
Iallgather, Iallgatherv, Iallgather_pure, Iallgatherv_pure	(Q+1)*4 MB	(Q+1)*X bytes
All other IMB-NBC benchmarks	8 MB	2X bytes
Compare_and_swap	12 B	12 B
Exchange_put, Exchange_get	16 MB	4X bytes
All other IMB-RMA benchmarks	8 MB	2X bytes

NOTE

If you do not select the `-cache` flag, add 2X cache size to all of the above.

For IMB-IO benchmarks, make sure you have enough disk space available:

- 16MB in the standard mode
- max (X, OVERALL_VOL) bytes in the optional mode

For instructions on enabling the optional mode, see [Parameters Controlling Intel® MPI Benchmarks](#).

2.2. Software Requirements

To run the Intel® MPI Benchmarks, you need:

- `cpp`, ANSI C compiler, `gmake` on Linux* OS or Unix* OS.
- Enclosed Microsoft Visual* C++ solutions as the basis for Microsoft Windows* OS.
- MPI installation, including a startup mechanism for parallel MPI programs.

2.3. Installing Intel® MPI Benchmarks

To install the Intel® MPI Benchmarks, unpack the installation file.

If you install the product within Intel® Parallel Studio XE Cluster Edition, the default installation directory will be: `<psxe_install_dir>/imb`

The installation directory structure is as follows:

- `ReadMe_IMB.txt`
- `./doc` – documentation directory that contains the User Guide, in PDF and HTML formats:
 - `IMB_Users_Guide.pdf`
 - `IMB_Users_Guide.htm`
- `./license` – license agreement directory that contains the following files:
 - `license.txt` – specifies the source code license granted to you.
 - `use-of-trademark-license.txt` – specifies the license for using the name and/or trademark of the Intel® MPI Benchmarks.
- `./src` – program source- and Makefiles.
- `./WINDOWS` – Microsoft* Visual Studio* solution files.

For basic instructions on how to use the Intel® MPI Benchmarks, see `ReadMe_IMB.txt`.

See Also

[Building Intel® MPI Benchmarks](#)

2.4. Building Intel® MPI Benchmarks

2.4.1. Linux* OS

To build the benchmarks for Linux* OS, do the following:

1. Set the `CC` variable to point to the appropriate compiler wrapper, `mpiicc` or `mpicc`.
2. Run one or more makefile commands listed below.

Command	Description
<code>make clean</code>	Remove legacy binary object files and executable files
<code>make MPI1</code>	Build the executable file for the IMB-MPI1 component.
<code>make EXT</code>	Build the executable file for one-sided communications benchmarks.

make IO	Build the executable file for I/O benchmarks.
make NBC	Build the executable file for IMB-NBC benchmarks.
make RMA	Build the executable file for IMB-RMA benchmarks.
make all	Build all executable files available.

To build the benchmarks for Intel® Many Integrated Core Architecture (Intel® MIC Architecture), follow these steps:

1. Build the Intel MPI Benchmarks for the host system:

```
host$ source <psxe_install_dir>/bin/psxevars.sh
host$ cd <path to IMB directory>/src
host$ make -f make_ict
```

2. Build the Intel MPI Benchmarks for the target system based on the Intel® MIC Architecture.

```
host$ cd <path to IMB directory>/src
host$ make -f make_ict_mic
```

For details on running the resulting executable files on the Intel® MIC Architecture, see the *Intel® MPI Library* documentation.

2.4.2. Windows* OS

To build the benchmarks for IMB-MPI1, IMB-IO, IMB-EXT, IMB-NBC, or IMB-RMA, follow these steps:

1. Check the environment variable settings for Include, Lib, and Path. Make sure they are set in accordance with this table:

Intel® 64 Architecture Settings	IA-32 Architecture Settings
%I_MPI_ROOT%\intel64\include	%I_MPI_ROOT%\ia32\include
%I_MPI_ROOT%\intel64\lib	%I_MPI_ROOT%\ia32\lib
%I_MPI_ROOT%\intel64\bin	%I_MPI_ROOT%\ia32\bin

NOTE

2. Intel® MPI Library 5.0 does not support the IA-32 architecture. Use an earlier version of Intel MPI Library to build IA-32 architecture benchmarks.

3. Go to the subfolder that corresponds to the Intel® MPI Benchmarks component you would like to build and the Microsoft* Visual Studio* version installed on your system. For example, to build IMB-EXT.exe with the Visual* Studio* 2010, go to IMB-EXT_VS_2010.
4. Open the .vcproj or .vcxproj file in Visual Studio*. The executable file for one of the Intel MPI Benchmarks components is created:
 - IMB-EXT.exe
 - IMB-IO.exe
 - IMB-MPI1.exe
 - IMB-NBC.exe
 - IMB-RMA.exe

5. From the **Solution Platforms** drop-down list, choose the required architecture (**x64** or **Win32**).
6. From the **Solution Configurations** drop-down list, choose **Release**.
7. Highlight the project folder in the **Solution Explorer**.
8. Go to **Project > Properties** to open **Configuration Properties** dialog box. Make sure you have something like the following settings:

6.

Setting	Value	Notes
General > Project Defaults		
Character Set	Use Multi-Byte Character Set	
Debugging		
Debugger to launch	Local Windows Debugger	Depending on your system configuration, you may select other debuggers.
Command	<ul style="list-style-type: none"> x64: %I_MPI_ROOT%\intel64\bin\mpiexec.exe IA-32: %I_MPI_ROOT%\ia32\bin\mpiexec.exe 	
Command Arguments	-n 2 %TargetPath%	%TargetPath% should be quoted as in: "-n 2 %TargetPath%"
C/C++ > General		
Additional Include Directories	<ul style="list-style-type: none"> x64: %I_MPI_ROOT%\intel64\include IA-32: %I_MPI_ROOT%\ia32\include 	
Warning Level	to Level 1 (/W1)	
C/C++ > Preprocessor		
Preprocessor Definition	<ul style="list-style-type: none"> IMB-EXT: WIN_UMB, _CRT_SECURE_NO_DEPRECATED, EXT IMB-IO: WIN_UMB, _CRT_SECURE_NO_DEPRECATED, MPIIO IMB-MPI1: WIN_UMB, _CRT_SECURE_NO_DEPRECATED, MPI1 IMB-NBC: WIN_UMB, _CRT_SECURE_NO_DEPRECATED, NBC IMB-RMA: WIN_UMB, _CRT_SECURE_NO_DEPRECATED, RMA 	
Linker > Input		

Additional Dependencies	<ul style="list-style-type: none"> • x64: %I_MPI_ROOT%\intel64\lib\impi.lib • IA-32: %I_MPI_ROOT%\ia32\lib\impi.lib 	
--------------------------------	---	--

- 7.
8. Go to **Build > Build Solution** to create an executable file.
9. Run the executable file using **Debug > Start Without Debugging** command.

See Also

[Running Intel® MPI Benchmarks](#)

2.5. Running Intel® MPI Benchmarks

To run the Intel® MPI Benchmarks, use the following command-line syntax:

```
$ mpirun -n <P> IMB-<component> [arguments]
```

where

- <P> is the number of processes. P=1 is recommended for all I/O and message passing benchmarks except the single transfer ones.
- <component> is the component-specific suffix that can take MPI1, EXT, IO, NBC, and RMA values.

By default, all benchmarks run on *Q* active processes defined as follows:

$Q = [1,] 2, 4, 8, \dots, \text{largest } 2^x$

For example, if P=11, the benchmarks run on $Q = [1,] 2, 4, 8, 11$ active processes. Single transfer IMB-IO benchmarks run with $Q=1$. Single transfer IMB-EXT and IMB-RMA benchmarks run with $Q=2$.

To pass control arguments other than P, you can use (argc, argv). Process 0 in MPI_COMM_WORLD reads all command-line arguments and broadcasts them to all other processes. Control arguments can define various features, such as time measurement, message length, and selection of communicators. For details, see [Command-Line Control](#).

See Also

[Command-Line Control](#)

[Parameters Controlling Intel® MPI Benchmarks](#)

2.6. Running Benchmarks in Multiple Mode

Intel® MPI Benchmarks provides a set of elementary MPI benchmarks.

You can run all benchmarks in the following modes:

- Standard (or non-multiple, default) – the benchmarks run in a single process group.
- Multiple – the benchmarks run in several process groups.

In the multiple mode, the number of groups may differ depending on the benchmark. For example, if PingPong is running on $N \geq 4$ processes, $N/2$ separate groups of two processes are formed. These process groups are running PingPong simultaneously. Thus, the benchmarks of the single transfer class behave as parallel transfer benchmarks when run in the multiple mode.

See Also

[Classification of MPI-1 Benchmarks](#)

[Classification of MPI-2 Benchmarks](#)

[MPI-3 Benchmarks](#)

3. MPI-1 Benchmarks

IMB-MPI1 component of the Intel® MPI Benchmarks provides benchmarks for MPI-1 functions. IMB-MPI1 contains the following benchmarks:

Standard Mode	Multiple Mode
PingPong	Multi-PingPong
PingPongSpecificSource (excluded by default)	Multi-PingPongSpecificSource (excluded by default)
PingPongAnySource (excluded by default)	Multi-PingPongAnySource (excluded by default)
PingPing	Multi-PingPing
PingPingSpecificSource (excluded by default)	Multi-PingPingSpecificSource (excluded by default)
PingPingAnySource (excluded by default)	Multi-PingPingAnySource (excluded by default)
Sendrecv	Multi-Sendrecv
Exchange	Multi-Exchange
Uniband	Multi-Uniband
Biband	Multi-Biband
Bcast	Multi-Bcast
Allgather	Multi-Allgather
Allgatherv	Multi-Allgatherv
Scatter	Multi-Scatter
Scatterv	Multi-Scatterv
Gather	Multi-Gather
Gatherv	Multi-Gatherv
Alltoall	Multi-Alltoall
Alltoallv	Multi-Alltoallv
Reduce	Multi-Reduce

Reduce_scatter	Multi-Reduce_scatter
Allreduce	Multi-Allreduce
Barrier	Multi-Barrier

3.1. Classification of MPI-1 Benchmarks

Intel® MPI Benchmarks introduces the following classes of benchmarks:

- Single Transfer
- Parallel Transfer
- Collective benchmarks

The following table lists the MPI-1 benchmarks in each class:

Single Transfer	Parallel Transfer	Collective
PingPong	Sendrecv	Bcast Multi-Bcast
PingPongSpecificSource	Exchange	Allgather Multi-Allgather
PingPongAnySource	Multi-PingPong	Allgatherv Multi-Allgatherv
PingPing	Multi-PingPing	Alltoall Multi-Alltoall
PingPingSpecificSource	Multi-Sendrecv	Alltoallv Multi-Alltoallv
PingPingAnySource	Multi-Exchange	Scatter Multi-Scatter
	Uniband	Scatterv Multi-Scatterv
	Biband	Gather Multi-Gather
	Multi-Uniband	Gatherv Multi-Gatherv
	Multi-Biband	Reduce Multi-Reduce
		Reduce_scatter

		Multi-Reduce_scatter
		Allreduce Multi-Allreduce
		Barrier Multi-Barrier

Each class interprets results in a different way.

3.1.1. Single Transfer Benchmarks

Single transfer benchmarks involve two active processes into communication. Other processes wait for the communication completion. Each benchmark is run with varying message lengths. The timing is averaged between two processes. The basic MPI data type for all messages is `MPI_BYTE`.

Throughput values are measured in MBps and can be calculated as follows:

$\text{throughput} = X/\text{time}$

where

- `time` is measured in μ sec.
- `x` is the length of a message, in bytes.

3.1.2. Parallel Transfer Benchmarks

Parallel transfer benchmarks involve more than two active processes into communication. Each benchmark runs with varying message lengths. The timing is averaged over multiple samples. The basic MPI data type for all messages is `MPI_BYTE`. The throughput calculations of the benchmarks take into account the multiplicity `nmsg` of messages outgoing from or incoming to a particular process. For the `Sendrecv` benchmark, a particular process sends and receives `x` bytes, the turnover is `2x` bytes, `nmsg=2`. For the `Exchange` benchmark, the turnover is `4x` bytes, `nmsg=4`.

Throughput values are measured in MBps and can be calculated as follows:

$\text{throughput} = \text{nmsg} * X / \text{time},$

where

- `time` is measured in μ sec.
- `x` is the length of a message, in bytes.

3.1.3. Collective Benchmarks

Collective benchmarks measure MPI collective operations. Each benchmark is run with varying message lengths. The timing is averaged over multiple samples. The basic MPI data type for all messages is `MPI_BYTE` for pure data movement functions and `MPI_FLOAT` for reductions.

Collective benchmarks show bare timings.

3.2. Single Transfer Benchmarks

The following benchmarks belong to the single transfer class:

- `PingPong`
- `PingPongSpecificSource`


- PingPongAnySource
- PingPing
- PingPingSpecificSource
- PingPingAnySource

See sections below for definitions of these benchmarks.

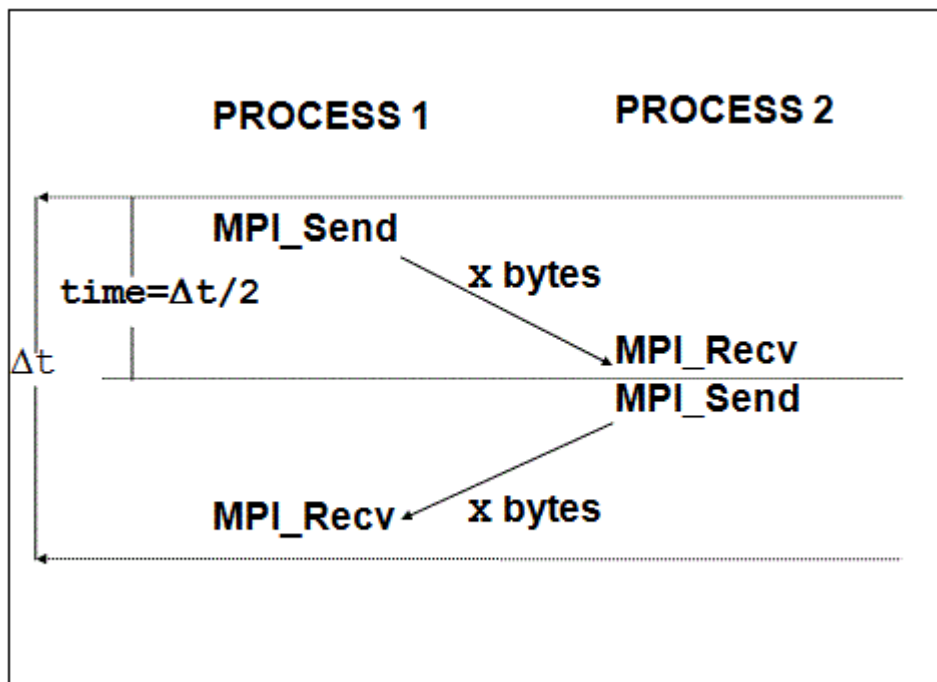
3.2.1. PingPong, PingPongSpecificSource, PingPongAnySource

Use PingPong, PingPongSpecificSource, and PingPongAnySource for measuring startup and throughput of a single message sent between two processes. PingPongAnySource uses the MPI_ANY_SOURCE value for destination rank, while PingPong and PingPongSpecificSource use an explicit value.

PingPong Definition

Property	Description
Measured pattern	As symbolized between  in the figure below. This benchmark runs on two active processes ($Q=2$) .
MPI routines	MPI_Send, MPI_Recv
MPI data type	MPI_BYTE
Reported timings	time= $\Delta t/2$ (in μsec) as indicated in the figure below.
Reported throughput	X/time


PingPong Pattern



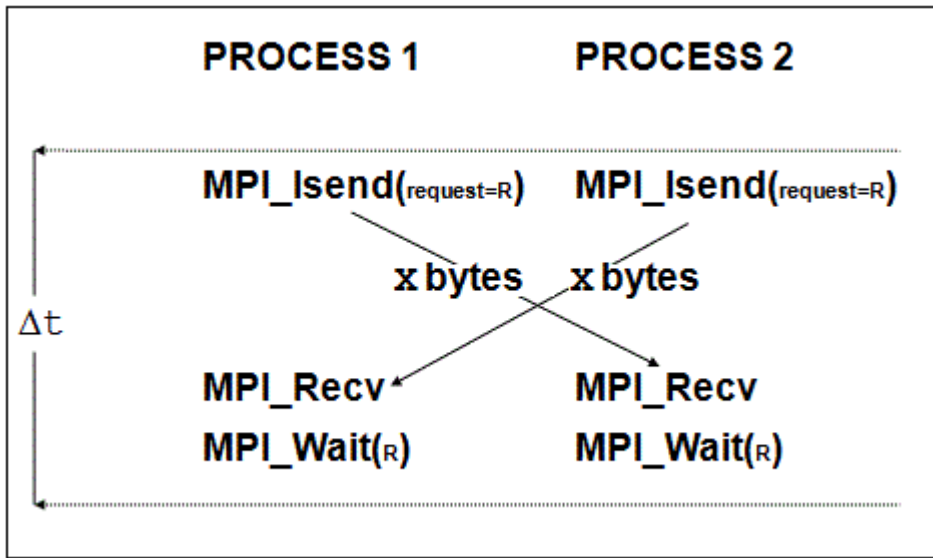
3.2.2. PingPing, PingPingSpecificSource, PingPingAnySource

PingPing, PingPingSpecificSource, and PingPingAnySource measure startup and throughput of single messages that are obstructed by oncoming messages. To achieve this, two processes communicate with each other using `MPI_Isend/MPI_Recv/MPI_Wait` calls. The `MPI_Isend` calls are issued simultaneously by both processes. For destination rank, PingPingAnySource uses the `MPI_ANY_SOURCE` value, while PingPing and PingPingSpecificSource use an explicit value.

PingPing Definition

Property	Description
Measured pattern	As symbolized between  in the figure below. This benchmark runs on two active processes ($Q=2$).
MPI routines	<code>MPI_Isend/MPI_Wait</code> , <code>MPI_Recv</code>
MPI data type	<code>MPI_BYTE</code>
Reported timings	$\text{time} = \Delta t$ (in μsec)
Reported throughput	X / time

PingPing Pattern



3.3. Parallel Transfer Benchmarks

The following benchmarks belong to the parallel transfer class:

- Sendrecv
- Exchange
- Uniband
- Biband
- Multi-PingPong
- Multi-PingPing
- Multi-Sendrecv
- Multi-Exchange
- Multi-Uniband
- Multi-Biband

See sections below for definitions of these benchmarks.

NOTE


The definitions of the multiple mode benchmarks are analogous to their standard mode counterparts in the single transfer class.

3.3.1. Sendrecv

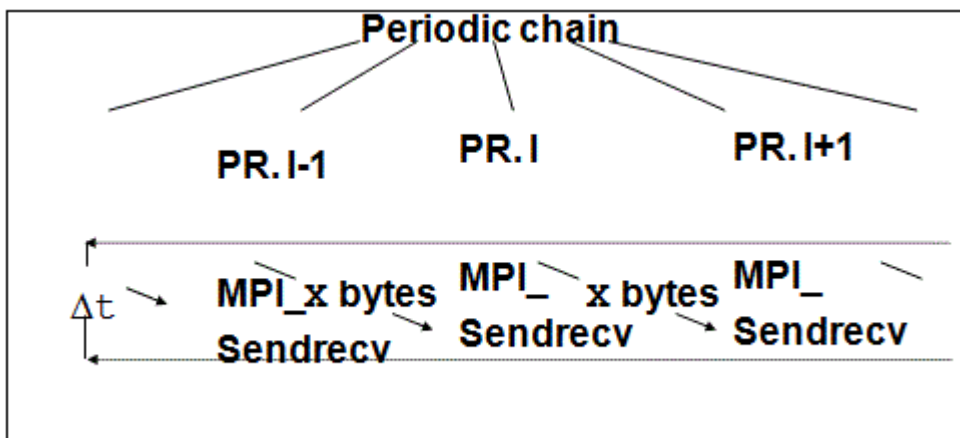
The `Sendrecv` benchmark is based on `MPI_Sendrecv`. In this benchmark, the processes form a periodic communication chain. Each process sends a message to the right neighbor and receives a message from the left neighbor in the chain. The turnover count is two messages per sample (one in, one out) for each process.

In the case of two processes, `Sendrecv` is equivalent to the `PingPing` benchmark of `IMB1.x`. For two processes, it reports the bidirectional bandwidth of the system, as obtained by the optimized `MPI_Sendrecv` function.

Sendrecv Definition

Property	Description
Measured pattern	As symbolized between  in the figure below.
MPI routines	MPI_Sendrecv
MPI data type	MPI_BYTE
Reported timings	time= Δt (in μsec) as indicated in the figure below.
Reported throughput	$2X/\text{time}$

Sendrecv Pattern




3.3.2. Exchange

Exchange is a communication pattern that often occurs in grid splitting algorithms (boundary exchanges). The group of processes is similar to a periodic chain, and each process exchanges data with both left and right neighbor in the chain.

The turnover count is four messages per sample (two in, two out) for each process.

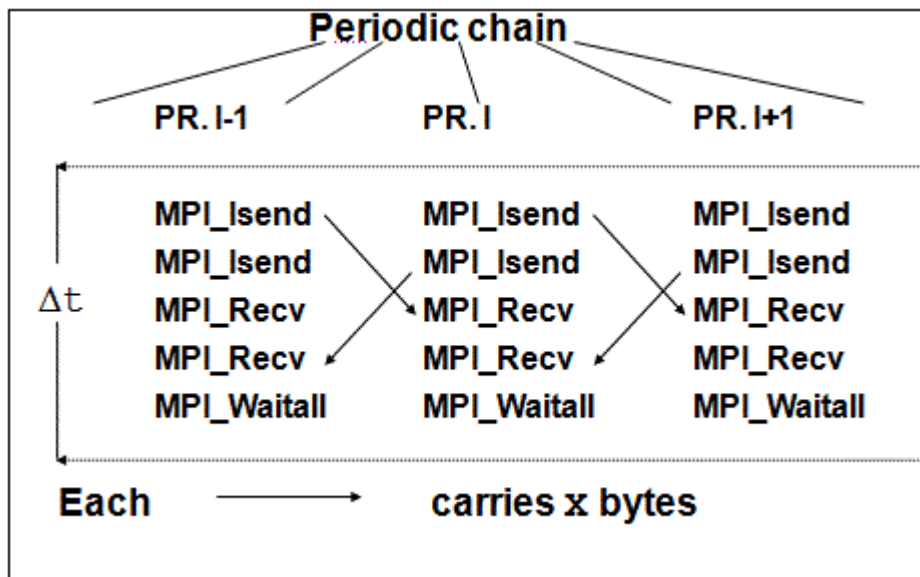
For two `Isend` messages, separate buffers are used.

Exchange Definition

Property	Description
Measured pattern	As symbolized between  in the figure below.
MPI routines	MPI_Isend/MPI_Waitall, MPI_Recv
MPI data type	MPI_BYTE

Reported timings	$\text{time} = \Delta t$ (in μsec)
Reported throughput	$4X/\text{time}$

Exchange Pattern



3.3.3. Uniband

The **Uniband** benchmark measures the cumulative bandwidth and message rate values. To achieve this, the first half of ranks communicates with the second half using **MPI_Isend**/**MPI_Recv**/**MPI_Wait** calls. In case of the odd number of processes, one of them does not participate in the message exchange. The bunch of **MPI_Isend** calls are issued by each rank in the first half of ranks to its counterpart from the second half of ranks. The number of messages issued at each iteration step is defined with the **MAX_WIN_SIZE** constant. The same buffer is used for every send event in the iteration.

Uniband Definition

Property	Description
Measured pattern	$(\text{MAX_WIN_SIZE} * \text{MPI_Isend}) / (\text{MAX_WIN_SIZE} * \text{MPI_Irecv}) / \text{Waitall}$
MPI data type	MPI_BYTE (origin and target)
Reported message rate	$\text{MAX_WIN_SIZE} * N / \text{MAX}(\text{process_timings})$, where N is the number of communicating pairs
Reported throughput	MBps ($\text{msg_rate} * \text{size_in_bytes} / 1000000.0$)

3.3.4. Biband

The **Biband** measures the cumulative bandwidth and message rate values. To achieve this, the first half of ranks communicates with the second half using **MPI_Isend**/**MPI_Recv**/**MPI_Wait** calls. In case of the odd

number of processes, one of them does not participate in the message exchange. The bunch of `MPI_Isend` calls are issued by each rank in the first half of ranks to its counterpart from the second half of ranks, and vice versa. The number of messages issued at each iteration step is defined with the `MAX_WIN_SIZE` constant. The same buffer is used for every send event in the iteration.

Uniband Definition

Property	Description
Measured pattern	$(\text{MAX_WIN_SIZE} * \text{MPI_Isend}) / (\text{MAX_WIN_SIZE} * \text{MPI_Irecv}) / \text{Waitall}$
MPI data type	<code>MPI_BYTE</code> (origin and target)
Reported message rate	$2 * \text{MAX_WIN_SIZE} * N / \text{MAX}(\text{process_timings})$, where N is the number of communicating pairs
Reported throughput	$\text{MBps} (\text{msg_rate} * \text{size_in_bytes} / 1000000.0)$

3.4. Collective Benchmarks

The following benchmarks belong to the collective class:

- Bcast/multi-Bcast
- Allgather/multi-Allgather
- Allgatherv/multi-Allgatherv
- Alltoall/multi-Alltoall
- Alltoallv/multi-Alltoallv
- Scatter/multi-Scatter
- Scatterv/multi-Scatterv
- Gather/multi-Gather
- Gatherv/multi-Gatherv
- Reduce/multi-Reduce
- Reduce_scatter/multi-Reduce_scatter
- Allreduce/multi-Allreduce
- Barrier/multi-Barrier

See sections below for definitions of these benchmarks.

3.4.1. Reduce

The benchmark for the `MPI_Reduce` function. It reduces a vector of length $L = X / \text{sizeof}(\text{float})$ float items. The MPI data type is `MPI_FLOAT`. The MPI operation is `MPI_SUM`. The root of the operation is changed round-robin.

Property	Description
Measured pattern	<code>MPI_Reduce</code>

MPI data type	MPI_FLOAT
MPI operation	MPI_SUM
Root	$i \% \text{num_procs}$ in iteration i
Reported timings	Bare time
Reported throughput	None

3.4.2. Reduce_scatter

The benchmark for the `MPI_Reduce_scatter` function. It reduces a vector of length $L = X/\text{sizeof}(\text{float})$ float items. The MPI data type is `MPI_FLOAT`. The MPI operation is `MPI_SUM`. In the scatter phase, the L items are split as evenly as possible. To be exact, for np number of processes:

$$L = r * np + s$$

where

- $r = \lfloor L/np \rfloor$
- $s = L \bmod np$

In this case, the process with rank i gets:

- $r+1$ items when $i < s$
- r items when $i \geq s$

Property	Description
Measured pattern	<code>MPI_Reduce_scatter</code>
MPI data type	<code>MPI_FLOAT</code>
MPI operation	<code>MPI_SUM</code>
Reported timings	Bare time
Reported throughput	None

3.4.3. Allreduce

The benchmark for the `MPI_Allreduce` function. It reduces a vector of length $L = X/\text{sizeof}(\text{float})$ float items. The MPI data type is `MPI_FLOAT`. The MPI operation is `MPI_SUM`.

Property	Description
Measured pattern	<code>MPI_Allreduce</code>
MPI data type	<code>MPI_FLOAT</code>

MPI operation	MPI_SUM
Reported timings	Bare time
Reported throughput	None

3.4.4. Allgather

The benchmark for the `MPI_Allgather` function. Every process inputs x bytes and receives the gathered $x \cdot n_p$ bytes, where n_p is the number of processes.

Property	Description
Measured pattern	MPI_Allgather
MPI data type	MPI_BYTE
Reported timings	Bare time
Reported throughput	None

3.4.5. Allgatherv

The benchmark for the `MPI_Allgatherv` function. Every process inputs x bytes and receives the gathered $x \cdot n_p$ bytes, where n_p is the number of processes. Unlike `Allgather`, this benchmark shows whether MPI produces overhead.

Property	Description
Measured pattern	MPI_Allgatherv
MPI data type	MPI_BYTE
Reported timings	Bare time
Reported throughput	None

3.4.6. Scatter

The benchmark for the `MPI_Scatter` function. The root process inputs $x \cdot n_p$ bytes (x for each process). All processes receive x bytes. The root of the operation is changed round-robin.

Property	Description
Measured pattern	MPI_Scatter
MPI data type	MPI_BYTE
Root	$i \% \text{num_procs}$ in iteration i

Reported timings	Bare time
Reported throughput	None

3.4.7. Scatterv

The benchmark for the `MPI_Scatterv` function. The root process inputs $X \times np$ bytes (X for each process). All processes receive X bytes. The root of the operation is changed round-robin.

Property	Description
Measured pattern	<code>MPI_Scatterv</code>
MPI data type	<code>MPI_BYTE</code>
Root	<code>i%num_procs</code> in iteration <code>i</code>
Reported timings	Bare time
Reported throughput	None

3.4.8. Gather

The benchmark for the `MPI_Gather` function. The root process inputs $X \times np$ bytes (X from each process). All processes receive X bytes. The root of the operation is changed round-robin.

Property	Description
Measured pattern	<code>MPI_Gather</code>
MPI data type	<code>MPI_BYTE</code>
Root	<code>i%num_procs</code> in iteration <code>i</code>
Reported timings	Bare time
Reported throughput	None

3.4.9. Gatherv

The benchmark for the `MPI_Gatherv` function. All processes input X bytes. The root process receives $X \times np$ bytes, where `np` is the number of processes. The root of the operation is changed round-robin.

Property	Description
Measured pattern	<code>MPI_Gatherv</code>
MPI data type	<code>MPI_BYTE</code>

Root	<code>i%num_procs</code> in iteration <code>i</code>
Reported timings	Bare time
Reported throughput	None

3.4.10. Alltoall

The benchmark for the `MPI_Alltoall` function. In the case of `np` number of processes, every process inputs `X*np` bytes (`X` for each process) and receives `X*np` bytes (`X` from each process).

Property	Description
Measured pattern	<code>MPI_Alltoall</code>
MPI data type	<code>MPI_BYTE</code>
Reported timings	Bare time
Reported throughput	None

3.4.11. Bcast

The benchmark for `MPI_Bcast`. The root process broadcasts `x` bytes to all other processes. The root of the operation is changed round-robin.

Property	Description
Measured pattern	<code>MPI_Bcast</code>
MPI data type	<code>MPI_BYTE</code>
Reported timings	Bare time
Reported throughput	None

3.4.12. Barrier

The benchmark for the `MPI_Barrier` function.

Property	Description
Measured pattern	<code>MPI_Barrier</code>
Reported timings	Bare time
Reported throughput	None

4. MPI-2 Benchmarks

Intel® MPI Benchmarks provides benchmarks for MPI-2 functions in two components: IMB-EXT and IMB-IO. The table below lists all MPI-2 benchmarks available and specifies whether they support the aggregate mode. For I/O benchmarks, the table also lists nonblocking flavors.

Benchmark	Aggregate Mode	Non-blocking Mode
IMB-EXT		
Window Multi-Window		
Unidir_Put Multi-Unidir_Put	Supported	
Unidir_Get Multi-Unidir_Get	Supported	
Bidir_Get Multi-Bidir_Get	Supported	
Bidir_Put Multi-Bidir_Put	Supported	
Accumulate Multi-Accumulate	Supported	
Benchmark	Aggregate Mode	Non-blocking Mode
IMB-IO		
Open_Close Multi-Open_Close		
S_Write_indv Multi-S_Write_indv	Supported	S_IWrite_indv Multi-S_IWrite_indv
S_Read_indv Multi-S_Read_indv		S_IRead_indv Multi-S_IRead_indv
S_Write_expl Multi-S_Write_expl	Supported	S_IWrite_expl Multi-IS_Write_expl
S_Read_expl Multi-S_Read_expl		S_IRead_expl Multi-IS_Read_expl
P_Write_indv	Supported	P_IWrite_indv

Multi-P_Write_indv		Multi-P_IWrite_indv
P_Read_indv Multi-P_Read_indv		P_IRead_indv Multi-P_IRead_indv
P_Write_expl Multi-P_Write_expl	Supported	P_IWrite_expl Multi-P_IWrite_expl
P_Read_expl Multi-P_Read_expl		P_IRead_expl Multi-P_IRead_expl
P_Write_shared Multi-P_Write_shared	Supported	P_IWrite_shared Multi-P_IWrite_shared
P_Read_shared Multi-P_Read_shared		P_IRead_shared Multi-P_IRead_shared
P_Write_priv Multi-P_Write_priv	Supported	P_IWrite_priv Multi-P_IWrite_priv
P_Read_priv Multi-P_Read_priv		P_IRead_priv Multi-P_IRead_priv
C_Write_indv Multi-C_Write_indv	Supported	C_IWrite_indv Multi-C_IWrite_indv
C_Read_indv Multi-C_Read_indv		C_IRead_indv Multi-C_IRead_indv
C_Write_expl Multi-C_Write_expl	Supported	C_IWrite_expl Multi-C_IWrite_expl
C_Read_expl Multi-C_Read_expl		C_IRead_expl Multi-C_IRead_expl
C_Write_shared Multi-C_Write_shared	Supported	C_IWrite_shared Multi-C_IWrite_shared
C_Read_shared Multi-C_Read_shared		C_IRead_shared Multi-C_IRead_shared

See Also

[Benchmark Modes](#)

[IMB-IO Nonblocking Benchmarks](#)

4.1. Naming Conventions

MPI-2 benchmarks have the following naming conventions:

Convention	Description
Unidir/Bidir	Unidirectional/bidirectional one-sided communications. These are the one-sided equivalents of <code>PingPong</code> and <code>PingPing</code> .
S_	Single transfer benchmark.
C_	Collective benchmark.
P_	Parallel transfer benchmark.
expl	I/O with explicit offset.
indv	I/O with an individual file pointer.
shared	I/O with a shared file pointer.
priv	I/O with an individual file pointer to one private file for each process opened for <code>MPI_COMM_SELF</code> .
[ACTION]	A placeholder for <code>Read</code> or <code>Write</code> component of the benchmark name.
I	Non-blocking flavor. For example, <code>S_IWrite_indv</code> is the nonblocking flavor of the <code>S_IWrite_indv</code> benchmark.
Multi-	The benchmark runs in the multiple mode.

4.2. IMB-MPI-2 Benchmark Classification

Intel® MPI Benchmarks introduces three classes of benchmarks:

- Single Transfer
- Parallel Transfer
- Collective

Each class interprets results in a different way.

NOTE

The following benchmarks do not belong to any class:

- `Window` - measures overhead of one-sided communications for the `MPI_Win_create` / `MPI_Win_free` functions
-

- `Open_close` - measures overhead of input/output operations for the `MPI_File_open` / `MPI_File_close` functions

4.2.1. Single Transfer Benchmarks

This class contains benchmarks of functions that operate on a single data element transferred between one source and one target. For MPI-2 benchmarks, the source of the data transfer can be an MPI process or, in the case of `Read` benchmarks, an MPI file. The target can be an MPI process or an MPI file.

For I/O benchmarks, the single transfer is defined as an operation between an MPI process and an individual window or a file.

- Single transfer `IMB-EXT` benchmarks only run with two active processes.
- Single transfer `IMB-IO` benchmarks only run with one active process.

4.2.2. Parallel Transfer Benchmarks

This class contains benchmarks of functions that operate on several processes in parallel. The benchmark timings are produced under a global load. The number of participating processes is arbitrary.

In the Parallel Transfer, more than one process participates in the overall pattern.

The final time is measured as the maximum of timings for all single processes. The throughput is related to that time and the overall amount of transferred data (sum over all processes).

4.2.3. Collective Benchmarks

This class contains benchmarks of functions that are collective as provided by the MPI standard. The final time is measured as the maximum of timings for all single processes. The throughput is not calculated.

4.2.4. MPI-2 Benchmarks Classification

Single Transfer	Parallel Transfer	Collective	Other
Unidir_Get	Multi_Unidir_Get	Accumulate	Window
Unidir_Put	Multi_Unidir_Put	Multi_Accumulate	Multi_Window
Bidir_Get	Multi_Bidir_Get		
Bidir_Put	Multi_Bidir_Put		
S_[I]Write_indv	P_[I]Write_indv	C_[I]Write_indv	Multi-C_[I]Write_indv
S_[I]Write_indv	P_[I]Write_indv	C_[I]Write_indv Multi-C_[I]Write_indv	Open_close Multi-Open_close
S_[I]Read_indv	P_[I]Read_indv	C_[I]Read_indv Multi-C_[I]Read_indv	

S_[I]Write_expl	P_[I]Write_expl	C_[I]Write_expl Multi- C_[I]Write_expl	
S_[I]Read_expl	P_[I]Read_expl	C_[I]Read_expl Multi- C_[I]Read_expl	
	P_[I]Write_shared	C_[I]Write_shared Multi- C_[I]Write_shared	
	P_[I]Read_shared	C_[I]Read_shared Multi- C_[I]Write_shared	
	P_[I]Write_priv		
	P_[I]Read_priv		

4.3. MPI-2 Benchmark Modes

MPI-2 benchmarks can run in the following modes:

- **Blocking/nonblocking mode.** These modes apply to the `IMB-IO` benchmarks only. For details, see sections [IMB-IO Blocking Benchmarks](#) and [IMB-IO Nonblocking Benchmarks](#).
- **Aggregate/non-aggregate mode.** Non-aggregate mode is not available for nonblocking flavors of `IMB-IO` benchmarks.

The following example illustrates aggregation of `M` transfers for `IMB-EXT` and blocking Write benchmarks:

```
Select a repetition count M
time = MPI_Wtime();
issue M disjoint transfers
assure completion of all transfers
time = (MPI_Wtime() - time) / M
```

In this example:

- `M` is a repetition count:
 - `M = 1` in the non-aggregate mode
 - `M = n_sample` in the aggregate mode. For the exact definition of `n_sample` see the [Actual Benchmarking](#) section.
- A transfer is issued by the corresponding one-sided communication call (for `IMB-EXT`) and by an `MPI-IO` write call (for `IMB-IO`).
- *Disjoint* means that multiple transfers (if `M > 1`) are to/from disjoint sections of the window or file. This permits to avoid misleading optimizations when using the same locations for multiple transfers.

The variation of `M` provides important information about the system and the MPI implementation, crucial for application code optimizations. For example, the following possible internal strategies of an implementation could influence the timing outcome of the above pattern.

- Accumulative strategy. Several successive transfers (up to M in the example above) are accumulated without an immediate completion. At certain stages, the accumulated transfers are completed as a whole. This approach may save time of expensive synchronizations. This strategy is expected to produce better results in the aggregate case as compared to the non-aggregate one.
- Non-accumulative strategy. Every Transfer is completed before the return from the corresponding function. The time of expensive synchronizations is taken into account. This strategy is expected to produce equal results for aggregate and non-aggregate cases.

4.3.1. Assured Completion of Transfers

Following the MPI standard, *assured completion of transfers* is the minimum sequence of operations after which all processes of the file communicator have a consistent view after a write.

The aggregate and non-aggregate modes differ in when the assured completion of data transfers takes place:

- after each transfer (non-aggregate mode)
- after a bunch of multiple transfers (aggregate mode)

For Intel® MPI Benchmarks, assured completion means the following:

- For IMB-EXT benchmarks, `MPI_Win_fence`
- For IMB-IO Write benchmarks, a triplet `MPI_File_sync/MPI_Barrier(file_communicator)/MPI_File_sync`. This fixes the non-sufficient definition in the Intel® MPI Benchmarks 3.0.

4.4. IMB-EXT Benchmarks


This section provides definitions of IMB-EXT benchmarks. The benchmarks can run with varying transfer sizes X , in bytes. The timings are averaged over multiple samples. See the [Benchmark Methodology](#) section for details. In the definitions below, a single sample with a fixed transfer size X is used.

The `Unidir` and `Bidir` benchmarks are exact equivalents of the message passing `PingPong` and `PingPing`, respectively. Their interpretation and output are analogous to their message passing equivalents.

4.4.1. Unidir_Put

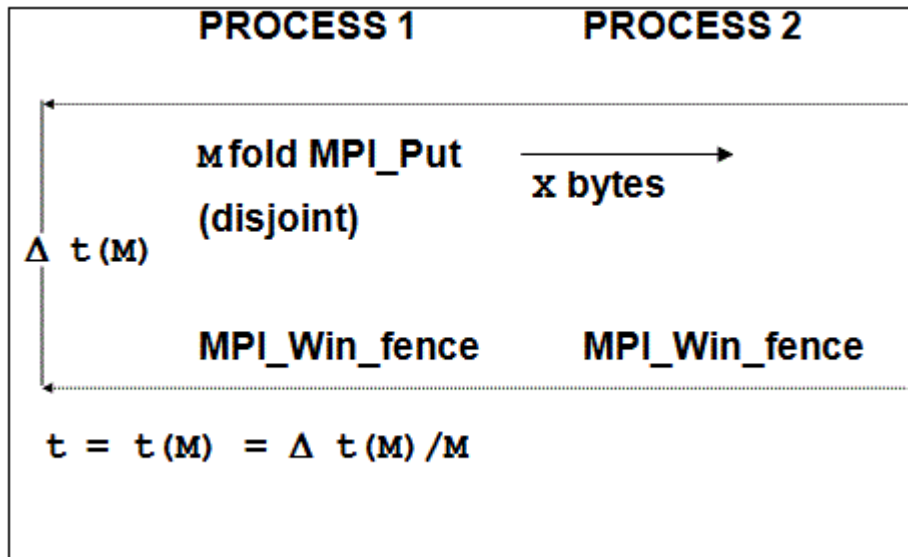
This is the benchmark for the `MPI_Put` function. The following table and figure provide the basic definitions and a schematic view of the pattern.

Unidir_Put Definition

Property	Description
Measured pattern	As symbolized between  in the figure below. This benchmark runs on two active processes ($Q=2$).
MPI routine	<code>MPI_Put</code>
MPI data type	<code>MPI_BYTE</code> (origin and target)
Reported timings	$t=t(M)$ (in μsec) as indicated in the figure below, non-aggregate ($M=1$) and aggregate ($M=n_sample$). For details, see Actual Benchmarking .

Reported throughput	x/t , aggregate and non-aggregate
---------------------	-------------------------------------


Unidir_Put Pattern



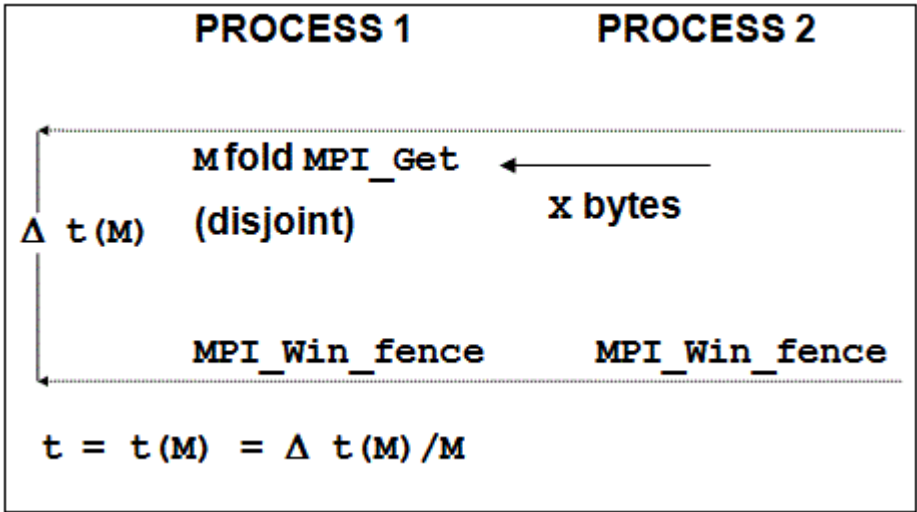
4.4.2. Unidir_Get

This is the benchmark for the `MPI_Get`

Unidir_Get Definition

Property	Description
Measured pattern	As symbolized between  in the figure below. This benchmark runs on two active processes ($Q=2$).
MPI routine	<code>MPI_Get</code>
MPI data type	<code>MPI_BYTE</code> , for both origin and target
Reported timings	$t=t(M)$ (in μsec) as indicated in the figure below, non-aggregate ($M=1$) and aggregate ($M=n_{\text{sample}}$). For details, see Actual Benchmarking .
Reported throughput	x/t , aggregate and non-aggregate


Unidir_Get Pattern



4.4.3. Bidir_Put

This is the benchmark for the `MPI_Put` function with bidirectional transfers. See the basic definitions below.


Bidir_Put Definition

Property	Description
Measured pattern	As symbolized between  in the figure below. This benchmark runs on two active processes ($Q=2$) .
MPI routine	<code>MPI_Put</code>
MPI data type	<code>MPI_BYTE</code> , for both origin and target
Reported timings	$t=t(M)$ (in μsec) as indicated in the figure below, non-aggregate ($M=1$) and aggregate ($M=n_{\text{sample}}$). For details, see Actual Benchmarking .
Reported throughput	X/t , aggregate and non-aggregate

4.4.4. Bidir_Get

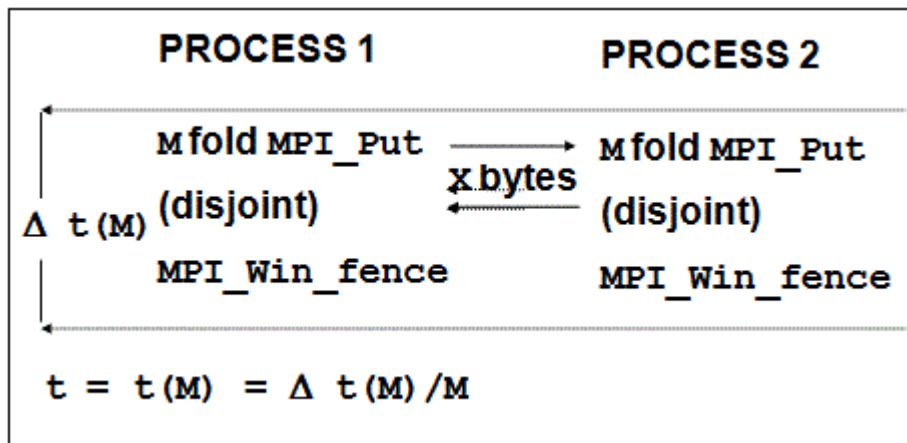
This is the benchmark for the `MPI_Get` function, with bidirectional transfers. Below see the basic definitions and a schematic view of the pattern.

Bidir_Get Definition

Property	Description
Measured pattern	As symbolized between  in the figure below. This benchmark runs on two active processes ($Q=2$).

MPI routine	MPI_Get
MPI data type	MPI_BYTE, for both origin and target
Reported timings	$t = t(M)$ (in μsec) as indicated in the figure below, non-aggregate ($M=1$) and aggregate ($M=n_{\text{sample}}$). For details, see Actual Benchmarking .
Reported throughput	X/t , aggregate and non-aggregate


Bidir_Get Pattern



4.4.5. Accumulate

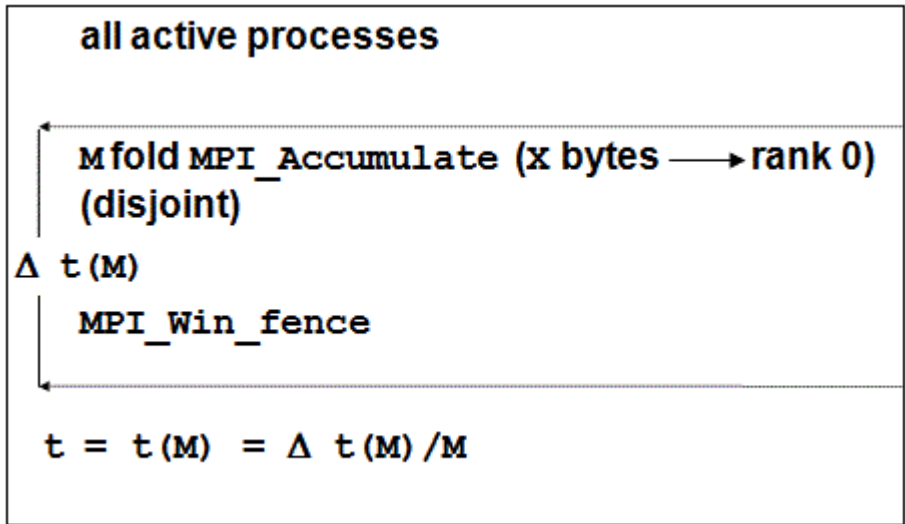
This is the benchmark for the MPI_Accumulate function. It reduces a vector of length $L = x/\text{sizeof}(\text{float})$ of float items. The MPI data type is MPI_FLOAT. The MPI operation is MPI_SUM. See the basic definitions and a schematic view of the pattern below.

Accumulate Definition

Property	Description
Measured pattern	As symbolized between  in the figure below. This benchmark runs on two active processes ($Q=2$).
MPI data type	MPI_FLOAT
MPI operation	MPI_SUM
Root	0
Reported timings	$t = t(M)$ (in μsec) as indicated in the figure below, non-aggregate ($M=1$) and aggregate ($M=n_{\text{sample}}$). For details, see Actual Benchmarking .

Reported throughput	None
---------------------	------

Accumulate Pattern



4.4.6. Window

This is the benchmark for measuring the overhead of an MPI_Win_create/MPI_Win_fence/MPI_Win_free combination. In the case of an unused window, a negligible non-trivial action is performed inside the window. It minimizes optimization effects of the MPI implementation.

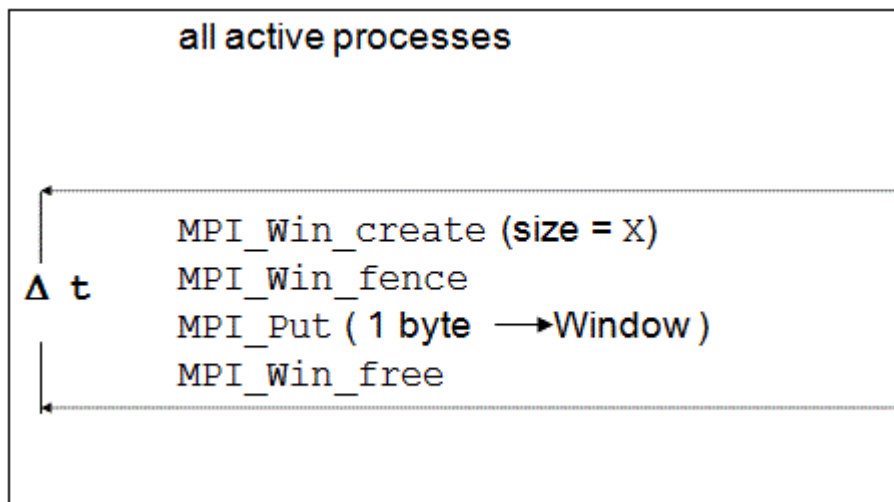
The MPI_Win_fence function is called to properly initialize an access epoch. This is a correction as compared to earlier releases of the Intel® MPI Benchmarks.

See the basic definitions and a schematic view of the pattern below.

Window Definition

Property	Description
Measured pattern	MPI_Win_create/MPI_Win_fence/MPI_Win_free
Reported timings	$t = \Delta t(M)$ (in μsec) as indicated in the figure below.
Reported throughput	None

Window Pattern



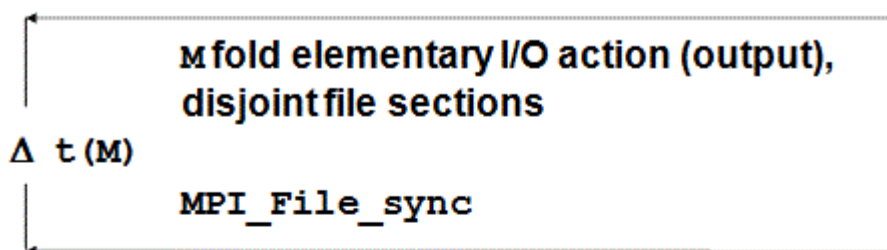
4.5. IMB-IO Blocking Benchmarks

This section describes blocking I/O benchmarks. The benchmarks can run with varying transfer sizes x , in bytes. The timings are averaged over multiple samples. The basic MPI data type for all data buffers is `MPI_BYTE`. In the definitions below, a single sample with a fixed I/O size x is used.

Every benchmark contains an elementary I/O action, denoting a pure read or write. Thus, all benchmark flavors have a `Write` and a `Read` component. The `[ACTION]` placeholder denotes a `Read` or a `Write` alternatively.

The `Write` flavors of benchmarks include a file synchronization with different placements for aggregate and non-aggregate modes.

I/O Benchmarks, Aggregation for Output

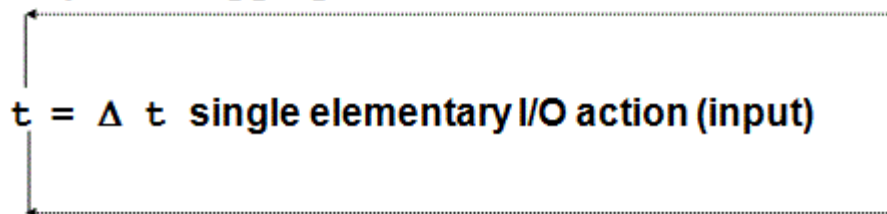
Output: m fold aggregation**non-aggregate mode:**

$$t = \Delta t (M = 1)$$

aggregate mode:

$$t = \Delta t (M = n_{\text{sample}}) / M$$

(choice of $M = n_{\text{sample}}$)

Input: No aggregation

4.5.1. S_[ACTION]_indv

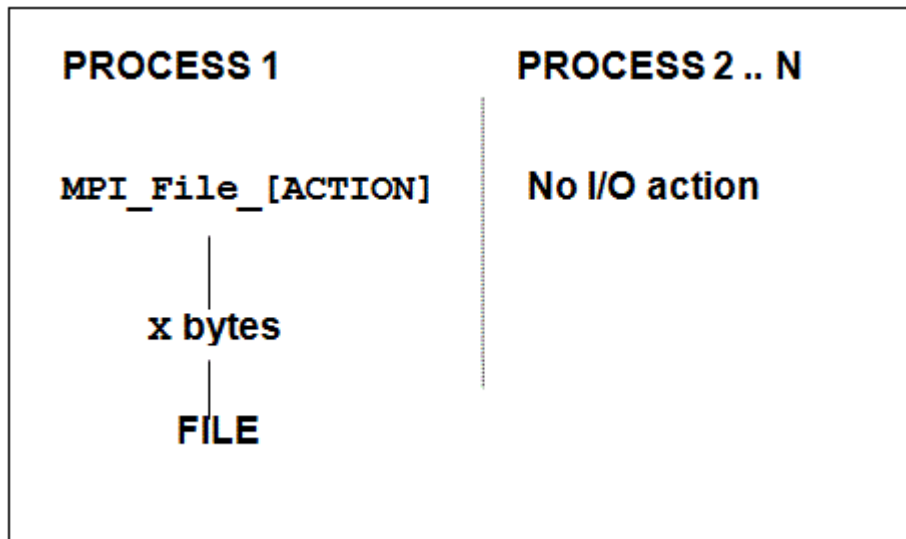
File I/O performed by a single process. This pattern mimics the typical case when a particular master process performs all of the I/O. See the basic definitions and a schematic view of the pattern below.

S_[ACTION]_indv Definition

Property	Description
Measured pattern	As symbolized in figure I/O benchmarks, aggregation for output
Elementary I/O action	As symbolized in the figure below.
MPI routines for the blocking mode	MPI_File_write/MPI_File_read
MPI routines for the nonblocking mode	MPI_File_iread/MPI_File_iwrite
etype	MPI_BYTE

File type	MPI_BYTE
MPI data type	MPI_BYTE
Reported timings	t (in μsec) as indicated in the figure I/O benchmarks, aggregation for output , aggregate and non-aggregate for the Write flavor.
Reported throughput	x/t , aggregate and non-aggregate for the Write flavor

S_[ACTION]_indv Pattern



4.5.2. S_[ACTION]_expl

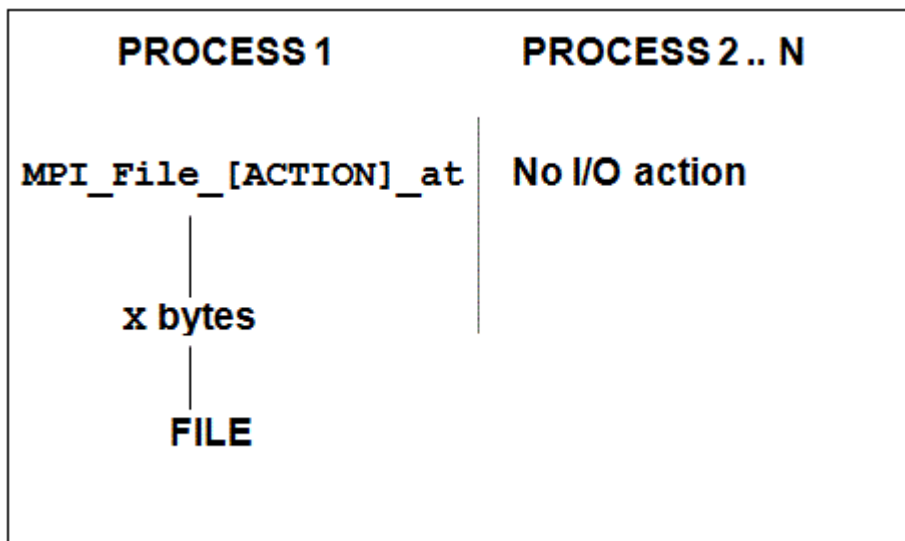
This benchmark mimics the same situation as S_[ACTION]_indv, with a different strategy to access files. See the basic definitions and a schematic view of the pattern below.

S_[ACTION]_expl Definition

Property	Description
Measured pattern	As symbolized in figure I/O benchmarks, aggregation for output
Elementary I/O action	As symbolized in the figure below.
MPI routines for the blocking mode	MPI_File_write_at/MPI_File_read_at
MPI routines for the nonblocking mode	MPI_File_irewrite_at/MPI_File_iread_at
etype	MPI_BYTE

File type	MPI_BYTE
MPI data type	MPI_BYTE
Reported timings	t (in μsec) as indicated in the figure I/O benchmarks, aggregation for output , aggregate and non-aggregate for the Write flavor.
Reported throughput	x/t , aggregate and non-aggregate for the Write flavor

S_[ACTION]_expl pattern



4.5.3. P_[ACTION]_indv

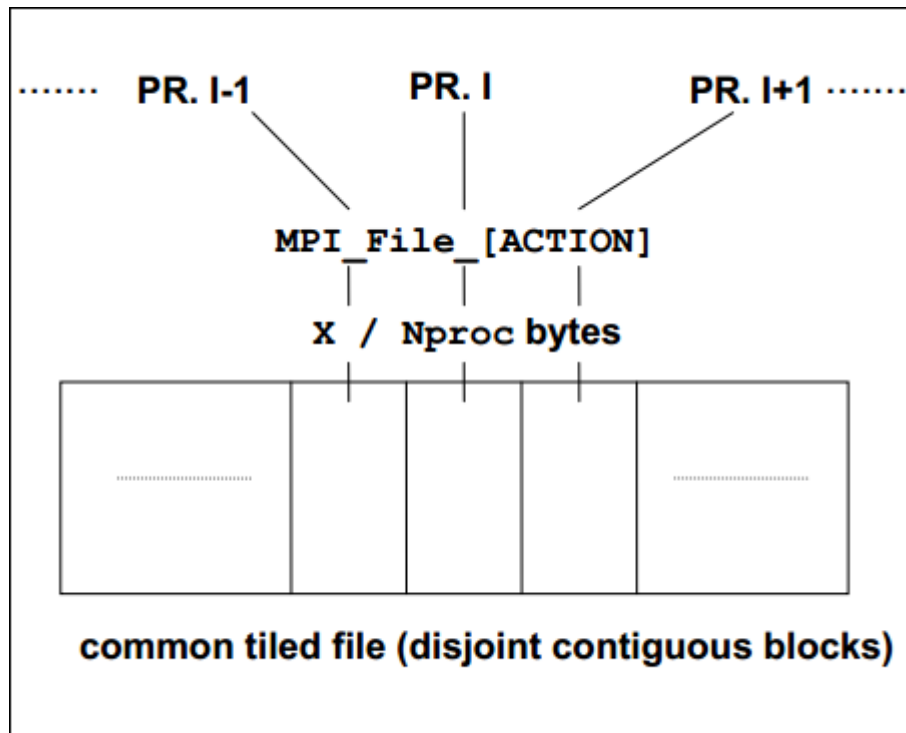
This pattern accesses the file in a concurrent manner. All participating processes access a common file. See the basic definitions and a schematic view of the pattern below.

P_[ACTION]_indv Definition

Property	Description
Measured pattern	As symbolized in figure I/O benchmarks, aggregation for output
Elementary I/O action	As symbolized in the figure below. In this figure, N_{proc} is the number of processes.
MPI routines for the blocking mode	MPI_File_write/MPI_File_read
MPI routines for the nonblocking mode	MPI_File_iread/MPI_File_iwrite
etype	MPI_BYTE

File type	Tiled view, disjoint contiguous blocks
MPI data type	MPI_BYTE
Reported timings	t (in μsec) as indicated in the figure I/O benchmarks, aggregation for output , aggregate and non-aggregate for the Write flavor.
Reported throughput	x/t , aggregate and non-aggregate for the Write flavor

P_[ACTION]_indv Pattern



4.5.4. P_ACTION_expl

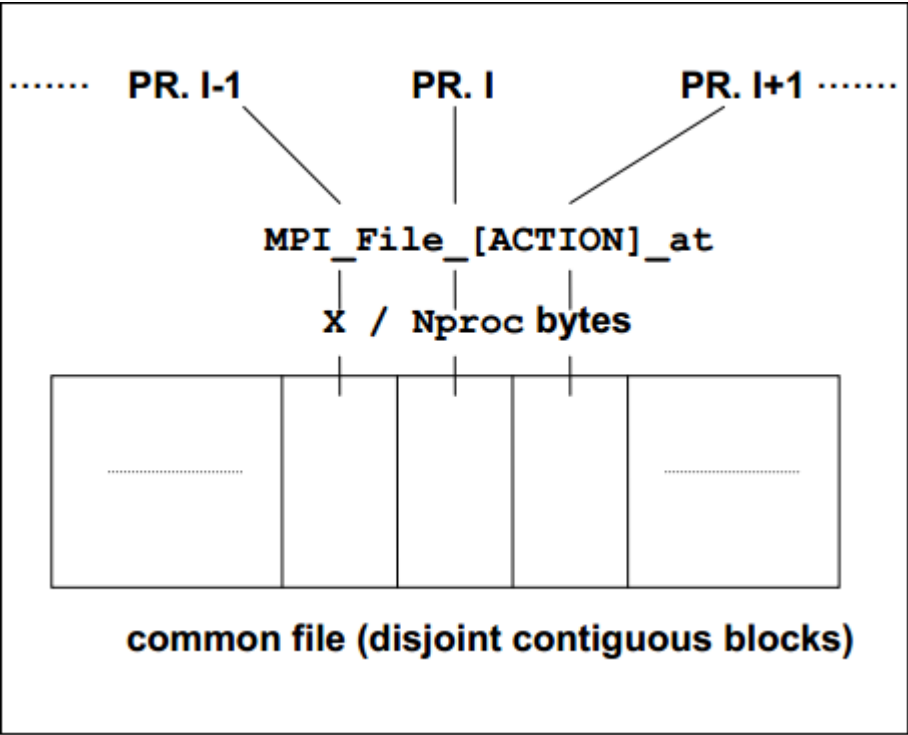
P_[ACTION]_expl follows the same access pattern as **P_[ACTION]_indv** with an explicit file pointer type. See the basic definitions and a schematic view of the pattern below.

P_[ACTION]_expl Definition

Property	Description
Measured pattern	As symbolized in figure I/O benchmarks, aggregation for output
Elementary I/O action	As symbolized in the figure below. In this figure, N_{proc} is the number of processes.

MPI routines for the blocking mode	MPI_File_write_at/MPI_File_read_at
MPI routines for the nonblocking mode	MPI_File_irewrite_at/MPI_File_iread_at
etype	MPI_BYTE
File type	MPI_BYTE
MPI data type	MPI_BYTE
Reported timings	t (in μsec) as indicated in the figure I/O benchmarks, aggregation for output , aggregate and non-aggregate for the Write flavor.
Reported throughput	x/t, aggregate and non-aggregate for the Write flavor

P_[ACTION]_expl Pattern



4.5.5. P_[ACTION]_shared

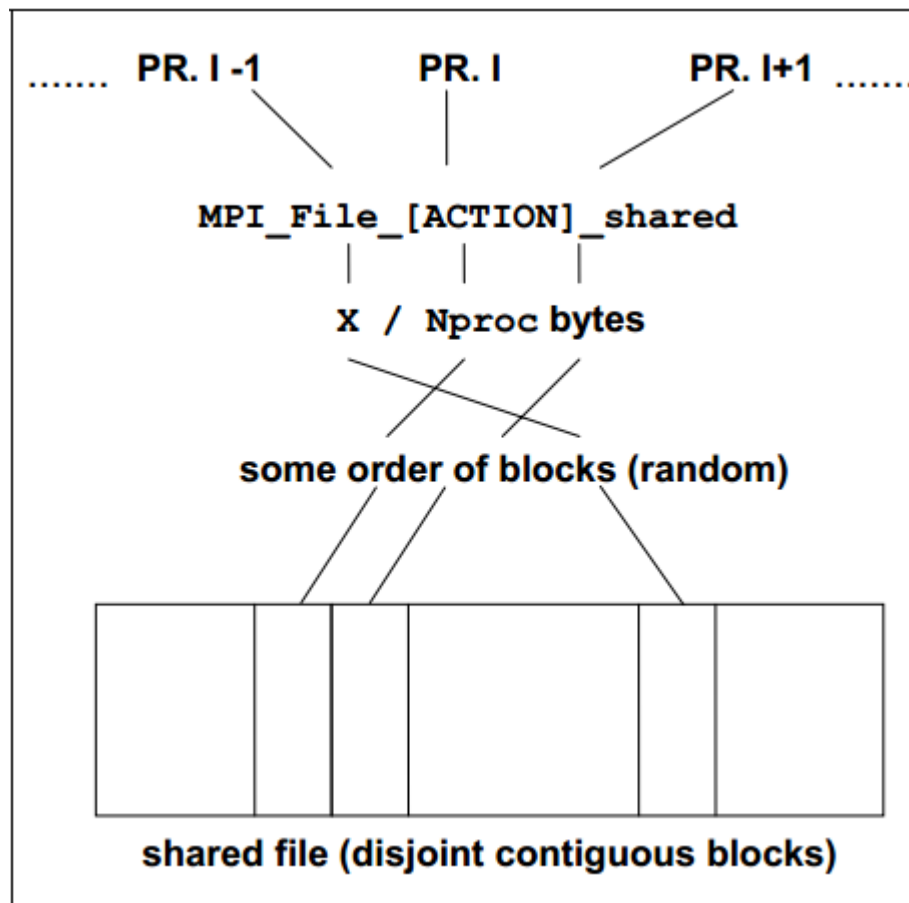
Concurrent access to a common file by all participating processes, with a shared file pointer. See the basic definitions and a schematic view of the pattern below.

P_[ACTION]_shared Definition

Property	Description
----------	-------------

Measured pattern	As symbolized in figure I/O benchmarks, aggregation for output
Elementary I/O action	As symbolized in the figure below. In this figure, N_{proc} is the number of processes.
MPI routines for the blocking mode	<code>MPI_File_write_at/MPI_File_read_at</code>
MPI routines for the nonblocking mode	<code>MPI_File_iread_at/MPI_File_iwrite_at</code>
etype	<code>MPI_BYTE</code>
File type	<code>MPI_BYTE</code>
MPI data type	<code>MPI_BYTE</code>
Reported timings	t (in μsec) as indicated in the figure I/O benchmarks, aggregation for output , aggregate and non-aggregate for the <code>Write</code> flavor.
Reported throughput	x/t , aggregate and non-aggregate for the <code>Write</code> flavor

P_[ACTION]_shared Pattern



4.5.6. P_[ACTION]_priv

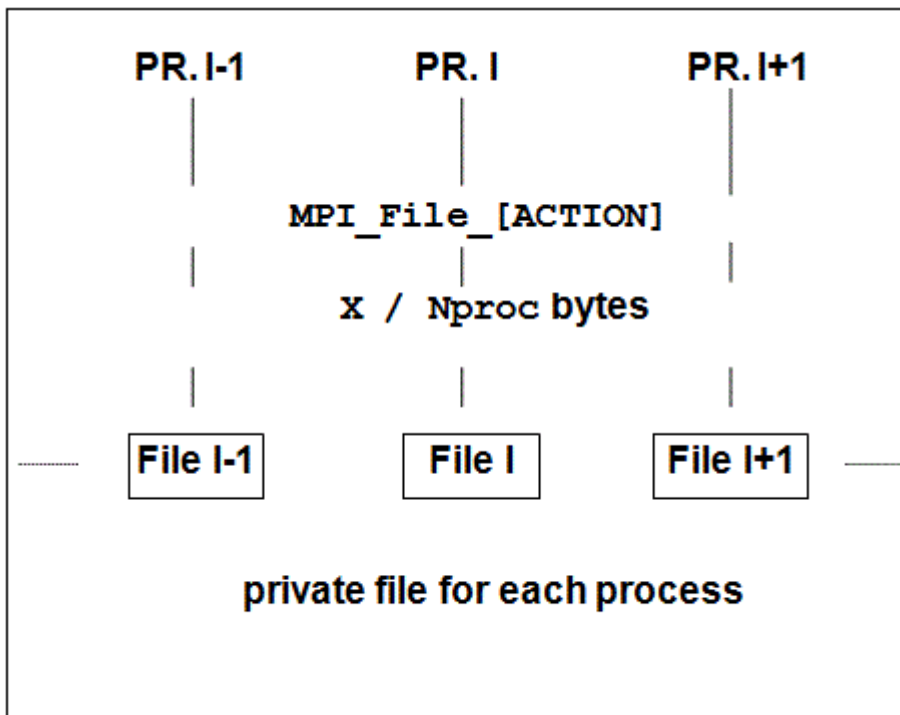
This pattern tests the case when all participating processes perform concurrent I/O to different private files. This benchmark is particularly useful for the systems that allow completely independent I/O operations from different processes. The benchmark pattern is expected to show parallel scaling and obtain optimum results. See the basic definitions and a schematic view of the pattern below.

P_[ACTION]_priv Definition

Property	Description
Measured pattern	As symbolized in figure I/O benchmarks, aggregation for output
Elementary I/O action	As symbolized in the figure below. In this figure, Nproc is the number of processes.
MPI routines for the blocking mode	MPI_File_write/MPI_File_read
MPI routines for the nonblocking mode	MPI_File_iread/MPI_File_iwrite
etype	MPI_BYTE

File type	MPI_BYTE
MPI data type	MPI_BYTE
Reported timings	Δt (in μsec), aggregate and non-aggregate for the Write flavor.
Reported throughput	$x/\Delta t$, aggregate and non-aggregate for the Write flavor

P_[ACTION]_priv Pattern



4.5.7. C_[ACTION]_indv

C_[ACTION]_indv tests collective access from all processes to a common file, with an individual file pointer. Below see the basic definitions and a schematic view of the pattern.

This benchmark is based on the following MPI routines:

- MPI_File_read_all/MPI_File_write_all for the blocking mode
- MPI_File_.._all_begin/MPI_File_.._all_end for the nonblocking mode

All other parameters and the measuring method are the same as for the P_[ACTION]_indv benchmark.

See Also

[P_\[ACTION\]_indv](#)

4.5.8. C_[ACTION]_expl

This pattern performs collective access from all processes to a common file, with an explicit file pointer.

This benchmark is based on the following MPI routines:

- `MPI_File_read_at_all/MPI_File_write_at_all` for the blocking mode
- `MPI_File_..._at_all_begin/MPI_File_..._at_all_end` for the nonblocking mode

All other parameters and the measuring method are the same as for the `P_[ACTION]_expl` benchmark.

See Also

[P_\[ACTION\]_expl](#)

4.5.9. C_[ACTION]_shared

The benchmark of a collective access from all processes to a common file, with a shared file pointer.

This benchmark is based on the following MPI routines:

- `MPI_File_read_ordered/MPI_File_write_ordered` for the blocking mode
- `MPI_File_..._ordered_begin/MPI_File_..._ordered_end` for the nonblocking mode

All other parameters and the measuring method are the same as for the `P_[ACTION]_shared` benchmark.

See Also

[P_\[ACTION\]_shared](#)

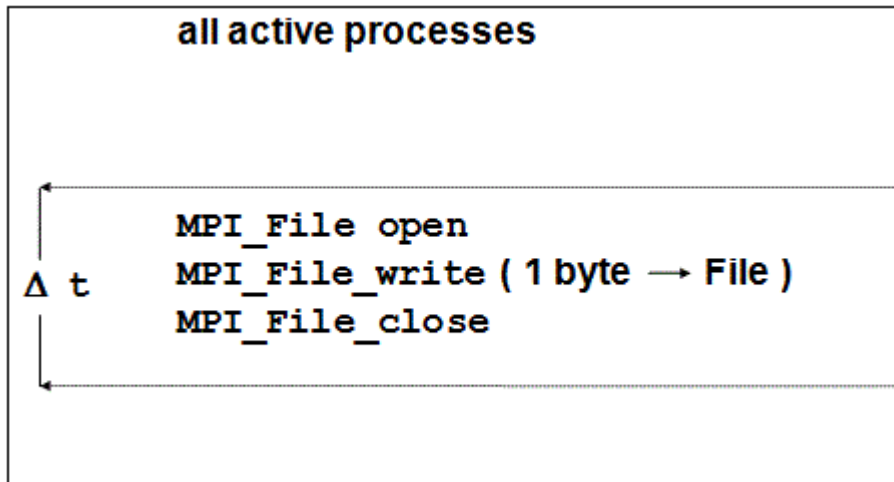
4.5.10. Open_Close

The benchmark for the `MPI_File_open/MPI_File_close` functions. All processes open the same file. To avoid MPI implementation optimizations for an unused file, a negligible non-trivial action is performed with the file. See the basic definitions of the benchmark below.

Open_Close Definition

Property	Description
Measured pattern	<code>MPI_File_open/MPI_File_close</code>
etype	<code>MPI_BYTE</code>
File type	<code>MPI_BYTE</code>
Reported timings	$t = \Delta t$ (in μsec), as indicated in the figure below.
Reported throughput	None

Open_Close Pattern



4.6. IMB-IO Non-blocking Benchmarks

Intel® MPI Benchmarks implements blocking and nonblocking modes of the IMB-IO benchmarks as different benchmark flavors. The `Read` and `Write` components of the blocking benchmark name are replaced for nonblocking flavors by `IRead` and `IWrite`, respectively.

The definitions of blocking and nonblocking flavors are identical, except for their behavior in regard to:

- Aggregation. The nonblocking versions only run in the non-aggregate mode.
- Synchronism. Only the meaning of an elementary transfer differs from the equivalent blocking benchmark.

Basically, an elementary transfer looks as follows:

```
time = MPI_Wtime()
for ( i=0; i<n_sample; i++ )
{
    Initiate transfer
    Exploit CPU
    Wait for the end of transfer
}
time = (MPI_Wtime()-time)/n_sample
```

The `Exploit CPU` section in the above example is arbitrary. Intel® MPI Benchmarks exploits CPU as described below.

4.6.1. Exploiting CPU

Intel® MPI Benchmarks uses the following method to exploit the CPU. A kernel loop is executed repeatedly. The kernel is a fully vectorizable multiplication of a 100x100 matrix with a vector. The function is scalable in the following way:

```
IMB_cpu_exploit(float desired_time, int initialize);
```

The input value of `desired_time` determines the time for the function to execute the kernel loop, with a slight variance. At the very beginning, the function is called with `initialize=1` and an input value for `desired_time`. This determines an Mflop/s rate and a timing `t_CPU`, as close as possible to `desired_time`, obtained by running without any obstruction. During the actual benchmarking, `IMB_cpu_exploit` is called

with `initialize=0`, concurrently with the particular I/O action, and always performs the same type and number of operations as in the initialization step.

4.6.2. Displaying Results

Three timings are crucial to interpret the behavior of nonblocking I/O, overlapped with CPU exploitation:

- `t_pure` is the time for the corresponding pure blocking I/O action, non-overlapping with CPU activity
- `t_CPU` is the time the `IMB_cpu_exploit` periods (running concurrently with nonblocking I/O) would use when running dedicated
- `t_ovrl` is the time for the analogous nonblocking I/O action, concurrent with CPU activity (exploiting `t_CPU` when running dedicated)

A perfect overlap means: $t_{ovrl} = \max(t_{pure}, t_{CPU})$

No overlap means: $t_{ovrl} = t_{pure} + t_{CPU}$.

The actual amount of overlap is:

$overlap = (t_{pure} + t_{CPU} - t_{ovrl}) / \min(t_{pure}, t_{CPU})$ (*)

The Intel® MPI Benchmarks result tables report the timings `t_ovrl`, `t_pure`, `t_CPU` and the estimated overlap obtained by the (*) formula above. At the beginning of a run, the Mflop/s rate is corresponding to the `t_CPU` displayed.

5. MPI-3 Benchmarks

Intel® MPI Benchmarks provides two sets of benchmarks conforming to the MPI-3 standard:

- **IMB-NBC** - benchmarks for nonblocking collective (NBC) operations
- **IMB-RMA** - one-sided communications benchmarks that measure the Remote Memory Access (RMA) functionality introduced in the MPI-3 standard.

5.1. IMB-NBC Benchmarks

Intel® MPI Benchmarks provides two types of benchmarks for nonblocking collective (NBC) routines that conform to the MPI-3 standard:

- Benchmarks for measuring the overlap of communication and computation
- Benchmarks for measuring pure communication time

TIP

When you run the **IMB-NBC** component, only the overlap benchmarks are enabled by default. To measure pure communication time, specify the particular benchmark name or use the `-include` command-line parameter to run the `_pure` flavor of the benchmarks.

The following table lists all **IMB-NBC** benchmarks:

Benchmarks Measuring Communication and Computation Overlap (Enabled by Default)	Benchmarks Measuring Pure Communication Time (Disabled by Default)
Ibcast	Ibcast_pure
Iallgather	Iallgather_pure
Iallgatherv	Iallgatherv_pure
Igather	Igather_pure
Igatherv	Igatherv_pure
Iscatter	Iscatter_pure
Iscatterv	Iscatterv_pure
Ialltoall	Ialltoall_pure
Ialltoallv	Ialltoallv_pure
Ireduce	Ireduce_pure
Ireduce_scatter	Ireduce_scatter_pure
Iallreduce	Iallreduce_pure
Ibarrier	Ibarrier_pure

See Also

[Measuring Communication and Computation Overlap](#)
[Measuring Pure Communication Time](#)

5.1.1. Measuring Communication and Computation Overlap

Semantics of nonblocking collective operations enables you to run inter-process communication in the background while performing computations. However, the actual overlap depends on the particular MPI library implementation. You can measure a potential overlap of communication and computation using IMB-NBC benchmarks. The general benchmark flow is as follows:

1. Measure the time needed for a pure communication call.
2. Start a nonblocking collective operation.
3. Start computation using the `IMB_cpu_exploit` function, as described in the [IMB-IO Nonblocking Benchmarks](#) chapter. To ensure correct measurement conditions, the computation time used by the benchmark is close to the pure communication time measured at step 1.
4. Wait for communication to finish using the `MPI_Wait` function.

Displaying Results

The timing values to interpret the overlap potential are as follows:

- `t_pure` is the time of a pure communication operation, non-overlapping with CPU activity.
- `t_CPU` is the time the `IMB_cpu_exploit` function takes to complete when run concurrently with the nonblocking communication operation.
- `t_ovrl` is the time of the nonblocking communication operation takes to complete when run concurrently with a CPU activity.
 - If `t_ovrl = max(t_pure, t_CPU)`, the processes are running with a perfect overlap.
 - If `t_ovrl = t_pure + t_CPU`, the processes are running with no overlap.

Since different processes in a collective operation may have different execution times, the timing values are taken for the process with the biggest `t_ovrl` execution time. The IMB-NBC result tables report the timings `t_ovrl`, `t_pure`, `t_CPU` and the estimated overlap in percent calculated by the following formula:

$$\text{overlap} = 100 \cdot \max(0, \min(1, (t_{\text{pure}} + t_{\text{CPU}} - t_{\text{ovrl}}) / \min(t_{\text{pure}}, t_{\text{CPU}})))$$

See Also

[IMB-NBC Benchmarks](#)
[Measuring Pure Communication Time](#)

5.1.2. Measuring Pure Communication Time

To measure pure execution time of nonblocking collective operations, use the `_pure` flavor of the IMB-NBC benchmarks. The benchmark methodology is consistent with the one used for regular [collective operations](#):

- Each rank performs the predefined amount of iterations and calculates the mean value.
- The basic MPI data type for all messages is `MPI_BYTE` for pure data movement functions and `MPI_FLOAT` for reductions.
- If the operation requires the root process to be specified, the root process is selected round-robin through iterations.

These benchmarks are not included into the default list of `IMB-NBC` benchmarks. To run a benchmark, specify the particular benchmark name or use the `-include` command-line parameter. For example:

```
$ mpirun -np 2 IMB-NBC Ialltoall_pure
$ mpirun -np 2 IMB-NBC -include Iallgather_pure Ialltoall_pure
```

Displaying Results

Pure nonblocking collective benchmarks show bare timing values. Since execution time may vary for different ranks, three timing values are shown: maximum, minimum, and the average time among all the ranks participating in the benchmark measurements.

See Also

[IMB-NBC Benchmarks](#)

[Measuring Communication and Computation Overlap](#)

[Command-Line Control](#)

5.1.3. Iallgather

The benchmark for `MPI_Iallgather` that measures communication and computation overlap.

Property	Description
Measured pattern	<code>MPI_Iallgather/IMB_cpu_exploit/MPI_Wait</code>
MPI data type	<code>MPI_BYTE</code>
Reported timings	<ul style="list-style-type: none"> <code>t_ovrl</code> <code>t_pure</code> <code>t_CPU</code> <code>overlap=100.*max(0,min(1, (t_pure+t_CPU-t_ovrl) / min(t_pure, t_CPU))</code> <p>For details, see Measuring Communication and Computation Overlap.</p>
Reported throughput	None

5.1.4. Iallgather_pure

The benchmark for the `MPI_Iallgather` function that measures pure communication time. Every process inputs `X` bytes and receives the gathered `X*np` bytes, where `np` is the number of processes.

Property	Description
Measured pattern	<code>MPI_Iallgather/MPI_Wait</code>
MPI data type	<code>MPI_BYTE</code>
Reported timings	Bare time
Reported throughput	None

5.1.5. Iallgather

The benchmark for `MPI_Iallgather` that measures communication and computation overlap.

Property	Description
Measured pattern	<code>MPI_Iallgather/IMB_cpu_exploit/MPI_Wait</code>
MPI data type	<code>MPI_BYTE</code>
Reported timings	<ul style="list-style-type: none"> <code>t_ovrl</code> <code>t_pure</code> <code>t_CPU</code> <code>overlap=100.*max(0,min(1, (t_pure+t_CPU-t_ovrl) / min(t_pure, t_CPU))</code> <p>For details, see Measuring Communication and Computation Overlap.</p>
Reported throughput	None

5.1.6. Iallgather_pure

The benchmark for the `MPI_Iallgather` function that measures pure communication time. Every process inputs `X` bytes and receives the gathered `X*np` bytes, where `np` is the number of processes. Unlike `Iallgather_pure`, this benchmark shows whether MPI produces overhead.

Property	Description
Measured pattern	<code>MPI_Iallgather/MPI_Wait</code>
MPI data type	<code>MPI_BYTE</code>
Reported timings	Bare time
Reported throughput	None

5.1.7. Iallreduce

The benchmark for `MPI_Iallreduce` that measures communication and computation overlap.

Property	Description
Measured pattern	<code>MPI_Iallreduce/IMB_cpu_exploit/MPI_Wait</code>
MPI data type	<code>MPI_FLOAT</code>
MPI operation	<code>MPI_SUM</code>
Reported timings	<ul style="list-style-type: none"> <code>t_ovrl</code> <code>t_pure</code>

	<ul style="list-style-type: none"> • <code>t_CPU</code> • <code>overlap=100.*max(0,min(1, (t_pure+t_CPU-t_ovrl) / min(t_pure, t_CPU))</code> <p>For details, see Measuring Communication and Computation Overlap.</p>
Reported throughput	None

5.1.8. `lallreduce_pure`

The benchmark for the `MPI_Iallreduce` function that measures pure communication time. It reduces a vector of length $L = X/\text{sizeof}(\text{float})$ float items. The MPI data type is `MPI_FLOAT`. The MPI operation is `MPI_SUM`.

Property	Description
Measured pattern	<code>MPI_Iallreduce/MPI_Wait</code>
MPI data type	<code>MPI_FLOAT</code>
MPI operation	<code>MPI_SUM</code>
Reported timings	Bare time
Reported throughput	None

5.1.9. `lalltoall`

The benchmark for `MPI_Ialltoall` that measures communication and computation overlap.

Property	Description
Measured pattern	<code>MPI_Ialltoall/IMB_cpu_exploit/MPI_Wait</code>
MPI data type	<code>MPI_BYTE</code>
Reported timings	<ul style="list-style-type: none"> • <code>t_ovrl</code> • <code>t_pure</code> • <code>t_CPU</code> • <code>overlap=100.*max(0,min(1, (t_pure+t_CPU-t_ovrl) / min(t_pure, t_CPU))</code> <p>For details, see Measuring Communication and Computation Overlap.</p>
Reported throughput	None

5.1.10. ialltoall_pure

The benchmark for the `MPI_Ialltoall` function that measures pure communication time. In the case of `np` number of processes, every process inputs `X*np` bytes (`X` for each process) and receives `X*np` bytes (`X` from each process).

Property	Description
Measured pattern	<code>MPI_Ialltoall/MPI_Wait</code>
MPI data type	<code>MPI_BYTE</code>
Reported timings	Bare time
Reported throughput	None

5.1.11. ialltoallv

The benchmark for `MPI_Ialltoallv` that measures communication and computation overlap.

Property	Description
Measured pattern	<code>MPI_Ialltoallv/IMB_cpu_exploit/MPI_Wait</code>
MPI data type	<code>MPI_BYTE</code>
Reported timings	<ul style="list-style-type: none"> <code>t_ovrl</code> <code>t_pure</code> <code>t_CPU</code> <code>overlap=100.*max(0,min(1, (t_pure+t_CPU-t_ovrl) / min(t_pure, t_CPU))</code> <p>For details, see Measuring Communication and Computation Overlap.</p>
Reported throughput	None

5.1.12. ialltoallv_pure

The benchmark for the `MPI_Ialltoallv` function that measures pure communication time. In the case of `np` number of processes, every process inputs `X*np` bytes (`X` for each process) and receives `X*np` bytes (`X` from each process).

Property	Description
Measured pattern	<code>MPI_Ialltoallv/MPI_Wait</code>
MPI data type	<code>MPI_BYTE</code>
Reported timings	Bare time

Reported throughput	None
---------------------	------

5.1.13. Ibarrier

The benchmark for `MPI_Ibarrier` that measures communication and computation overlap.

Property	Description
Measured pattern	<code>MPI_Ibarrier/IMB_cpu_exploit/MPI_Wait</code>
Reported timings	<ul style="list-style-type: none"> <code>t_ovrl</code> <code>t_pure</code> <code>t_CPU</code> $\text{overlap} = 100 \cdot \max(0, \min(1, (t_{\text{pure}} + t_{\text{CPU}} - t_{\text{ovrl}}) / \min(t_{\text{pure}}, t_{\text{CPU}})))$ <p>For details, see Measuring Communication and Computation Overlap.</p>
Reported throughput	None

5.1.14. Ibarrier_pure

The benchmark for the `MPI_Ibarrier` function that measures pure communication time.

Property	Description
Measured pattern	<code>MPI_Ibarrier/MPI_Wait</code>
Reported timings	Bare time
Reported throughput	None

5.1.15. Ibcast

The benchmark for `MPI_Ibcast` that measures communication and computation overlap.

Property	Description
Measured pattern	<code>MPI_Ibcast/IMB_cpu_exploit/MPI_Wait</code>
MPI data type	<code>MPI_BYTE</code>
Reported timings	<ul style="list-style-type: none"> <code>t_ovrl</code> <code>t_pure</code> <code>t_CPU</code> $\text{overlap} = 100 \cdot \max(0, \min(1, (t_{\text{pure}} + t_{\text{CPU}} - t_{\text{ovrl}}) / \min(t_{\text{pure}}, t_{\text{CPU}})))$ <p>For details, see Measuring Communication and Computation Overlap.</p>

Reported throughput	None
---------------------	------

5.1.16. Ibcast_pure

The benchmark for `MPI_Ibcast` that measures pure communication time. The root process broadcasts x bytes to all other processes. The root of the operation is changed round-robin.

Property	Description
Measured pattern	<code>MPI_Ibcast/MPI_Wait</code>
MPI data type	<code>MPI_BYTE</code>
Reported timings	Bare time
Reported throughput	None

5.1.17. Igather

The benchmark for `MPI_Igather` that measures communication and computation overlap.

Property	Description
Measured pattern	<code>MPI_Igather/IMB_cpu_exploit/MPI_Wait</code>
MPI data type	<code>MPI_BYTE</code>
Root	<code>i%num_procs</code> in iteration <code>i</code>
Reported timings	<ul style="list-style-type: none"> <code>t_ovrl</code> <code>t_pure</code> <code>t_CPU</code> $\text{overlap} = 100 \cdot \max(0, \min(1, (t_{\text{pure}} + t_{\text{CPU}} - t_{\text{ovrl}}) / \min(t_{\text{pure}}, t_{\text{CPU}})))$ <p>For details, see Measuring Communication and Computation Overlap.</p>
Reported throughput	None

5.1.18. Igather_pure

The benchmark for the `MPI_Igather` function that measures pure communication time. The root process inputs $x \cdot np$ bytes (x from each process). All processes receive x bytes. The root of the operation is changed round-robin.

Property	Description
Measured pattern	<code>MPI_Igather/MPI_Wait</code>

MPI data type	MPI_BYTE
Root	<code>i%num_procs</code> in iteration <code>i</code>
Reported timings	Bare time
Reported throughput	None

5.1.19. Igatherv

The benchmark for `MPI_Igatherv` that measures communication and computation overlap.

Property	Description
Measured pattern	<code>MPI_Igatherv/IMB_cpu_exploit/MPI_Wait</code>
MPI data type	MPI_BYTE
Root	<code>i%num_procs</code> in iteration <code>i</code>
Reported timings	<ul style="list-style-type: none"> <code>t_ovrl</code> <code>t_pure</code> <code>t_CPU</code> <code>overlap=100.*max(0,min(1, (t_pure+t_CPU-t_ovrl) / min(t_pure, t_CPU))</code> <p>For details, see Measuring Communication and Computation Overlap.</p>
Reported throughput	None

5.1.20. Igatherv_pure

The benchmark for the `MPI_Igatherv` function that measures pure communication time. All processes input `X` bytes. The root process receives `X*np` bytes, where `np` is the number of processes. The root of the operation is changed round-robin.

Property	Description
Measured pattern	<code>MPI_Igatherv/MPI_Wait</code>
MPI data type	MPI_BYTE
Root	<code>i%num_procs</code> in iteration <code>i</code>
Reported timings	Bare time
Reported throughput	None

5.1.21. Ireduce

The benchmark for `MPI_Ireduce` that measures communication and computation overlap.

Property	Description
Measured pattern	<code>MPI_Ireduce/IMB_cpu_exploit/MPI_Wait</code>
MPI data type	<code>MPI_FLOAT</code>
MPI operation	<code>MPI_SUM</code>
Root	<code>i%num_procs</code> in iteration <code>i</code>
Reported timings	<ul style="list-style-type: none"> <code>t_ovrl</code> <code>t_pure</code> <code>t_CPU</code> <code>overlap=100.*max(0,min(1, (t_pure+t_CPU-t_ovrl) / min(t_pure, t_CPU))</code> For details, see Measuring Communication and Computation Overlap .
Reported throughput	None

5.1.22. Ireduce_pure

The benchmark for the `MPI_Ireduce` function that measures pure communication time. It reduces a vector of length $L = X/\text{sizeof}(\text{float})$ float items. The MPI data type is `MPI_FLOAT`. The MPI operation is `MPI_SUM`. The root of the operation is changed round-robin.

Property	Description
Measured pattern	<code>MPI_Ireduce/MPI_Wait</code>
MPI data type	<code>MPI_FLOAT</code>
MPI operation	<code>MPI_SUM</code>
Root	<code>i%num_procs</code> in iteration <code>i</code>
Reported timings	Bare time
Reported throughput	None

5.1.23. Ireduce_scatter

The benchmark for `MPI_Ireduce_scatter` that measures communication and computation overlap. It reduces a vector of length $L = X/\text{sizeof}(\text{float})$ float items. The MPI data type is `MPI_FLOAT`. The MPI operation is `MPI_SUM`. In the scatter phase, the L items are split as evenly as possible. To be exact, for `np` number of processes:

$$L = r \cdot np + s$$

where

- $r = \lfloor L/np \rfloor$
- $s = L \bmod np$

In this case, the process with rank i gets:

- $r+1$ items when $i < s$
- r items when $i \geq s$

Property	Description
Measured pattern	MPI_Ireduce_scatter/IMB_cpu_exploit/MPI_Wait
MPI data type	MPI_FLOAT
MPI operation	MPI_SUM
Reported timings	<ul style="list-style-type: none"> • t_{ovrl} • t_{pure} • t_{CPU} • $\text{overlap} = 100 \cdot \max(0, \min(1, (t_{\text{pure}} + t_{\text{CPU}} - t_{\text{ovrl}}) / \min(t_{\text{pure}}, t_{\text{CPU}})))$ <p>For details, see Measuring Communication and Computation Overlap.</p>
Reported throughput	None

5.1.24. Ireduce_scatter_pure

The benchmark for the `MPI_Ireduce_scatter` function that measures pure communication time. It reduces a vector of length $L = X/\text{sizeof}(\text{float})$ float items. The MPI data type is `MPI_FLOAT`. The MPI operation is `MPI_SUM`. In the scatter phase, the L items are split as evenly as possible. To be exact, for np number of processes:

$$L = r \cdot np + s$$

where

- $r = \lfloor L/np \rfloor$
- $s = L \bmod np$

In this case, the process with rank i gets:

- $r+1$ items when $i < s$
- r items when $i \geq s$

Property	Description
Measured pattern	MPI_Ireduce_scatter/MPI_Wait
MPI data type	MPI_FLOAT

MPI operation	MPI_SUM
Reported timings	Bare time
Reported throughput	None

5.1.25. Iscatter

The benchmark for `MPI_Iscatter` that measures communication and computation overlap.

Property	Description
Measured pattern	MPI_Iscatter/IMB_cpu_exploit/MPI_Wait
MPI data type	MPI_BYTE
Root	<code>i%num_procs</code> in iteration <code>i</code>
Reported timings	<ul style="list-style-type: none"> <code>t_ovrl</code> <code>t_pure</code> <code>t_CPU</code> <code>overlap=100.*max(0,min(1, (t_pure+t_CPU-t_ovrl) / min(t_pure, t_CPU))</code> <p>For details, see Measuring Communication and Computation Overlap.</p>
Reported throughput	None

5.1.26. Iscatter_pure

The benchmark for the `MPI_Iscatter` function that measures pure communication time. The root process inputs `X*np` bytes (`X` for each process). All processes receive `X` bytes. The root of the operation is changed round-robin.

Property	Description
Measured pattern	MPI_Iscatter/MPI_Wait
MPI data type	MPI_BYTE
Root	<code>i%num_procs</code> in iteration <code>i</code>
Reported timings	Bare time
Reported throughput	None

5.1.27. Iscatterv

The benchmark for `MPI_Iscatterv` that measures communication and computation overlap.

Property	Description
Measured pattern	MPI_Isscatterv/IMB_cpu_exploit/MPI_Wait
MPI data type	MPI_BYTE
Root	<code>i%num_procs</code> in iteration <code>i</code>
Reported timings	<ul style="list-style-type: none"> <code>t_ovrl</code> <code>t_pure</code> <code>t_CPU</code> <code>overlap=100.*max(0,min(1, (t_pure+t_CPU-t_ovrl) / min(t_pure, t_CPU))</code> For details, see Measuring Communication and Computation Overlap .
Reported throughput	None

5.1.28. Isscatterv_pure

The benchmark for the `MPI_Isscatterv` function that measures pure communication time. The root process inputs `X*np` bytes (`X` for each process). All processes receive `X` bytes. The root of the operation is changed round-robin.

Property	Description
Measured pattern	MPI_Isscatterv/MPI_Wait
MPI data type	MPI_BYTE
Root	<code>i%num_procs</code> in iteration <code>i</code>
Reported timings	Bare time
Reported throughput	None

5.2. IMB-RMA Benchmarks

Intel® MPI Benchmarks provides a set of remote memory access (RMA) benchmarks that use the passive target communication mode to measure one-sided operations compliant with the MPI-3 standard.

5.2.1. IMB-RMA Benchmark Modes

When running the IMB-RMA benchmarks, you can choose between the following modes:

- Standard (default) or multiple mode.** You can enable the multiple mode for all IMB-RMA benchmarks using the `-multi <outflag>` command-line parameter. For details, see [Running Benchmarks in Multiple Mode](#).

- **Aggregate or non-aggregate mode.** For details on these modes, see the [MPI-2 Benchmark Modes](#) chapter. Some `IMB-RMA` benchmarks support the non-aggregate mode only. To determine which benchmarks can run in the aggregate mode, see [Classification of IMB-RMA Benchmarks](#).

5.2.2. Classification of IMB-RMA Benchmarks

All the `IMB-RMA` benchmarks fall into the following categories:

Single Transfer

In these benchmarks, one process accesses the memory of another process, in unidirectional or bidirectional manner. Single Transfer `IMB-RMA` benchmarks only run on two active processes. Throughput values are measured in MBps and can be calculated as follows:

$$\text{throughput} = X/\text{time},$$

where

- `time` is measured in `µsec`.
- `X` is the length of a message, in bytes.

Multiple Transfer

In these benchmarks, one process accesses the memory of several other processes.

Throughput values are measured in MBps and can be calculated as follows:

$$\text{throughput} = X/\text{time} * N, \text{ where}$$

- `time` is measured in `µsec`.
- `X` is the length of a message, in bytes.
- `N` is the number of target processes.

NOTE

The final throughput value is multiplied by the amount of target processes since the transfer is performed to every process except the origin process itself.

Parallel Transfer

This class contains benchmarks that operate on several processes in parallel. These benchmarks show bare timing values: maximum, minimum, and the average time among all the ranks participating in the benchmark measurements.

The table below lists all `IMB-RMA` benchmarks and specifies their properties:

Benchmark	Type	Aggregated Mode
Unidir_put	Single Transfer	Supported
Unidir_get	Single Transfer	Supported
Bidir_put	Single Transfer	Supported
Bidir_get	Single Transfer	Supported
One_put_all	Multiple Transfer	N/A
One_get_all	Multiple Transfer	N/A

All_put_all	Parallel Transfer	N/A
All_get_all	Parallel Transfer	N/A
Put_local	Single Transfer	Supported
Put_all_local	Multiple Transfer	N/A
Exchange_put	Parallel Transfer	N/A
Exchange_get	Parallel Transfer	N/A
Accumulate	Single Transfer	Supported
Get_accumulate	Single Transfer	Supported
Fetch_and_op	Single Transfer	Supported
Compare_and_swap	Single Transfer	Supported
Truly_passive_put	Single Transfer*	N/A
Get_local	Single Transfer	Supported
Get_all_local	Multiple Transfer	N/A

* The output format differs from the regular Single Transfer output. For details, see [Truly_passive_put](#).

5.2.3. Accumulate

This benchmark measures the `MPI_Accumulate` operation in passive target communication mode. The benchmark runs on two active processes. The target process is waiting in the `MPI_Barrier` call.

Property	Description
Measured pattern	<code>MPI_Accumulate/MPI_Win_flush</code>
MPI data type	<code>MPI_FLOAT</code> (origin and target)
MPI operation	<code>MPI_SUM</code>
Reported timings	Bare time
Reported throughput	MBps

5.2.4. All_get_all

The benchmark tests the scenario when all processes communicate with each other using the `MPI_Get` operation. To avoid congestion due to simultaneous access to the memory of a process by all other processes,

different ranks choose different targets at each particular step. For example, while looping through all the possible target ranks, the next target is chosen as follows: $(\text{target_rank} + \text{current_rank}) \% \text{num_ranks}$.

Property	Description
Measured pattern	$(N * \text{MPI_Get}) / \text{MPI_Win_flush_all}$, where N is the number of target processes
MPI data type	<code>MPI_BYTE</code> (origin and target)
Reported timings	Bare time
Reported throughput	None

5.2.5. All_put_all

The benchmark tests the scenario when all processes communicate with each other using `MPI_Put` operation. To avoid congestion due to simultaneous access to the memory of a process by all other processes, different ranks choose different targets at each particular step. For example, while looping through all the possible target ranks, the next target is chosen as follows: $(\text{target_rank} + \text{current_rank}) \% \text{num_ranks}$.

Property	Description
Measured pattern	$(N * \text{MPI_Put}) / \text{MPI_Win_flush_all}$, where N is the number of target processes
MPI data type	<code>MPI_BYTE</code> (origin and target)
Reported timings	Bare time
Reported throughput	None

5.2.6. Bidir_get

This benchmark measures the bidirectional `MPI_Get` operation in passive target communication mode. The benchmark runs on two active processes. These processes initiate an access epoch to each other using the `MPI_Lock` function, get data from the target, close the access epoch, and call the `MPI_Barrier` function.

Property	Description
Measured pattern	<code>MPI_Get/MPI_Win_flush</code>
MPI data type	<code>MPI_BYTE</code> (origin and target)
Reported timings	Bare time
Reported throughput	MBps

5.2.7. Bidir_put

This benchmark measures the bidirectional `MPI_Put` operation in passive target communication mode. The benchmark runs on two active processes. These processes initiate an access epoch to each other using the `MPI_Lock` function, transfer data, close the access epoch, and call the `MPI_Barrier` function.

Property	Description
Measured pattern	MPI_Put/MPI_Win_flush
MPI data type	MPI_BYTE (origin and target)
Reported timings	Bare time
Reported throughput	MBps

5.2.8. Compare_and_swap

This benchmark measures the `MPI_Compare_and_swap` operation in passive target communication mode. The target process is waiting in the `MPI_Barrier` call.

Property	Description
Measured pattern	MPI_Compare_and_swap/MPI_Win_flush
MPI data type	MPI_INT (origin and target)
Reported timings	Bare time
Reported throughput	MBps

5.2.9. Exchange_Get

This benchmark tests the scenario when each process exchanges data with its left and right neighbor processes using the `MPI_Get` operation.

Property	Description
Measured pattern	$(2 * \text{MPI_Get}) / (2 * \text{MPI_Win_flush})$
MPI data type	MPI_BYTE (origin and target)
Reported timings	Bare time
Reported throughput	None

5.2.10. Exchange_Put

This benchmark tests the scenario when each process exchanges data with its left and right neighbor processes using the `MPI_Put` operation.

Property	Description
Measured pattern	$(2 * \text{MPI_Put}) / (2 * \text{MPI_Win_flush})$

MPI data type	MPI_BYTE (origin and target)
Reported timings	Bare time
Reported throughput	None

5.2.11. Fetch_and_op

This benchmark measures the `MPI_Fetch_and_op` operation in passive target communication mode. The benchmark runs on two active processes. The target process is waiting in the `MPI_Barrier` call.

Property	Description
Measured pattern	MPI_Fetch_and_op/MPI_Win_flush
MPI data type	MPI_FLOAT (origin and target)
MPI operation	MPI_SUM
Reported timings	Bare time
Reported throughput	MBps

5.2.12. Get_accumulate

This benchmark measures the `MPI_Get_Accumulate` operation in passive target communication mode. The benchmark runs on two active processes. The target process is waiting in the `MPI_Barrier` call.

Property	Description
Measured pattern	MPI_Get_Accumulate/MPI_Win_flush
MPI data type	MPI_FLOAT (origin and target)
MPI operation	MPI_SUM
Reported timings	Bare time
Reported throughput	MBps

5.2.13. Get_all_local

This benchmark tests the `MPI_Get` operation where one active process obtains data from all other processes. All target processes are waiting in the `MPI_Barrier` call, while the active process performs the transfers. The completion of the origin process is ensured by the `MPI_Win_flush_local_all` operation. Since local completion of the `MPI_Get` operation is semantically equivalent to a regular completion, the benchmark flow is very similar to the `One_get_all` benchmark.

NOTE

This benchmark is not enabled in `IMB-RMA` by default. Specify the benchmark name in the command line or use the `-include` command-line parameter to run this benchmark.

Property	Description
Measured pattern	$(N * \text{MPI_Get}) / \text{MPI_Win_flush_local_all}$, where N is the number of target processes
MPI data type	<code>MPI_BYTE</code> (origin and target)
Reported timings	Bare time
Reported throughput	MBps

5.2.14. Get_local

This benchmark measures the combination of `MPI_Get` and `MPI_Win_flush_all` operations in passive target communication mode. The benchmark runs on two active processes. The target process is waiting in the `MPI_Barrier` call. Since local completion of the `MPI_Get` operation at the origin side is semantically equivalent to a regular completion, performance results are expected to be very close to the `Unidir_Get` benchmark results.

NOTE

This benchmark is not enabled in `IMB-RMA` by default. Specify the benchmark name in the command line or use the `-include` command-line parameter to run this benchmark.

Property	Description
Measured pattern	$\text{MPI_Get} / \text{MPI_Win_flush_local}$
MPI data type	<code>MPI_BYTE</code> (origin and target)
Reported timings	Bare time
Reported throughput	MBps

5.2.15. One_put_all

This benchmark tests the `MPI_Put` operation using one active process that transfers data to all other processes. All target processes are waiting in the `MPI_Barrier` call while the origin process performs the transfers.

Property	Description
Measured pattern	$(N * \text{MPI_Put}) / \text{MPI_Win_flush_all}$, where N is the number of target processes

MPI data type	MPI_BYTE (origin and target)
Reported timings	Bare time
Reported throughput	MBps

5.2.16. One_get_all

This benchmark tests the `MPI_Get` operation using one active process that gets data from all other processes. All target processes are waiting in the `MPI_Barrier` call while the origin process accesses their memory.

Property	Description
Measured pattern	$(N * \text{MPI_Get}) / \text{MPI_Win_flush_all}$, where N is the number of target processes.
MPI data type	MPI_BYTE (origin and target)
Reported timings	Bare time
Reported throughput	MBps

5.2.17. Put_all_local

This benchmark tests the `MPI_Put` operation where one active process transfers data to all other processes. All target processes are waiting in the `MPI_Barrier` call, while the origin process performs the transfers. The completion of the origin process is ensured by the `MPI_Win_flush_local_all` operation.

Property	Description
Measured pattern	$(N * \text{MPI_Put}) / \text{MPI_Win_flush_local_all}$, where N is the number of target processes
MPI data type	MPI_BYTE (origin and target)
Reported timings	Bare time
Reported throughput	MBps

5.2.18. Put_local

This benchmark measures the combination of `MPI_Put` and `MPI_Win_flush_all` operations in passive target communication mode. The benchmark runs on two active processes. The target process is waiting in the `MPI_Barrier` call.

Property	Description
Measured pattern	$\text{MPI_Put} / \text{MPI_Win_flush_local}$
MPI data type	MPI_BYTE (origin and target)

Reported timings	Bare time
Reported throughput	MBps

5.2.19. Truly_passive_put

This benchmark verifies whether the MPI implementation supports the truly one-sided communication mode. In this mode, the origin process can complete its access epoch even if the target process is outside the MPI stack.

The `Truly_passive_put` benchmark returns two timing values:

- The time needed for the origin process to complete the `MPI_Put` operation while the target process is waiting in the MPI stack in the `MPI_Barrier` call.
- The time needed for the origin process to complete the `MPI_Put` operation while the target process performs computations outside the MPI stack before the `MPI_Barrier` call.

To ensure measurement correctness, the time spent by the target process in the computation function should be comparable to the time needed for successful completion of the `MPI_Put` operation by the origin process.

Property	Description
Measured pattern	<code>MPI_Put/MPI_Win_flush</code> , while the target process performs computations before the <code>MPI_Barrier</code> call
MPI data type	<code>MPI_BYTE</code> (origin and target)
Reported timings	Bare time
Reported throughput	None

5.2.20. Unidir_get

This benchmark measures the `MPI_Get` operation in passive target communication mode. The benchmark runs on two active processes. The target process is waiting in the `MPI_Barrier` call.

Property	Description
Measured pattern	<code>MPI_Get/MPI_Win_flush</code>
MPI data type	<code>MPI_BYTE</code> (origin and target)
Reported timings	Bare time
Reported throughput	MBps

5.2.21. Unidir_put

This benchmark measures the `MPI_Put` operation in passive target communication mode. The benchmark runs on two active processes. The target process is waiting in the `MPI_Barrier` call.

Property	Description
Measured pattern	MPI_Put/MPI_Win_flush
MPI data type	MPI_BYTE (origin and target)
Reported timings	Bare time
Reported throughput	MBps

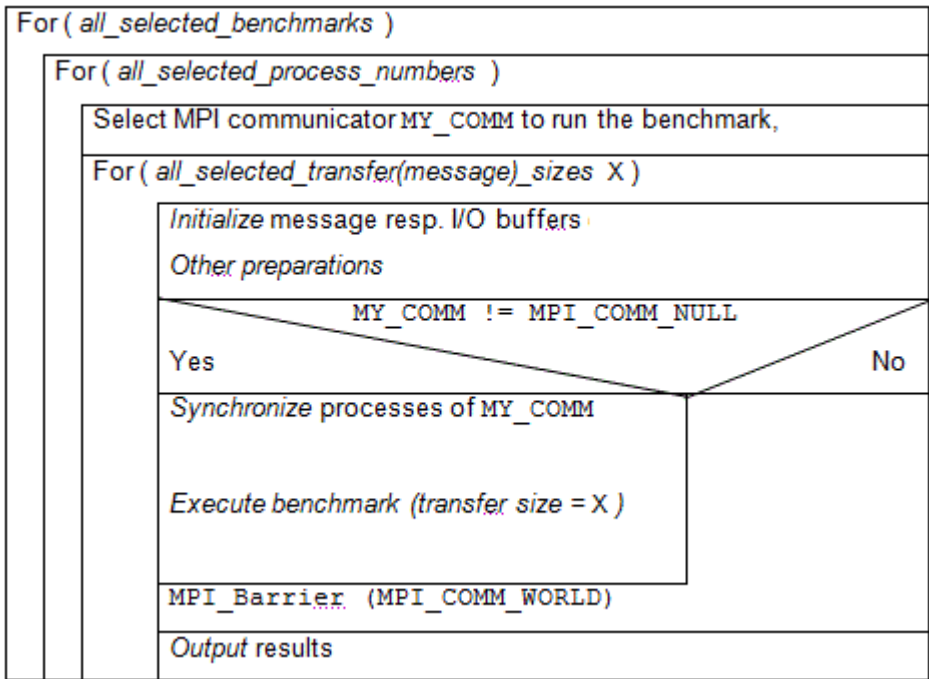
6. Benchmark Methodology

This section describes:

- Different ways to manage Intel® MPI Benchmarks control flow
- Command-line syntax for running the benchmarks
- Sample output data from Intel MPI Benchmarks

6.1. Control Flow

The following graph shows the control flow inside the Intel® MPI Benchmarks.



Intel® MPI Benchmarks provides different ways to manage its control flow:

- Hard-coded control mechanisms. For example, setting process numbers for running the central benchmarks. See the [Hard-coded Settings](#) section for details.
- Preprocessor parameters. Set the control parameters through the command line, or in the `settings.h / setting_io.h` include files. See [Parameters Controlling Intel® MPI Benchmarks](#) for details.

Intel® MPI Benchmarks also offers different modes of control:

- *Standard mode*. In this mode, all parameters are predefined and should not be changed. This ensures comparability for result tables.
- *Optional mode*. In this mode, you can set these parameters at your choice. You can use this mode to extend the result tables to larger transfer sizes.

6.2. Command-line Control

You can control all the aspects of the Intel® MPI Benchmarks through the command-line. The general command-line syntax is the following:

```

IMB-MPI1    [-h{elp}]
            [-npmin      <P_min>]
            [-multi      <outflag>]
            [-off_cache  <cache_size[,cache_line_size]>]
            [-iter       <msgspersample[,overall_vol[,msgs_nonaggr[,iter_policy]]]>]
            [-iter_policy <iter_policy>]
            [-time       <max_runtime per sample>]
            [-mem        <max. mem usage per process>]
            [-msglen     <Lengths_file>]
            [-map        <PxQ>]
            [-input      <filename>]
            [-include]   [benchmark1 [,benchmark2 [,...]]]
            [-exclude]   [benchmark1 [,benchmark2 [,...]]]
            [-msglog     [<minlog>:.<maxlog>]
            [-thread_level <level>]
            [-sync <mode>]
            [-root_shift <mode>]
            [-imb_barrier]
            [benchmark1 [,benchmark2 [,...]]]

```

The command line is repeated in the output. The options may appear in any order.

Examples:

Get out-of-cache data for PingPong:

```
mpirun -np 2 IMB-MPI1 PingPong -off_cache -1
```

Run a very large configuration, with the following parameters:

- Maximum iterations: 20
- Maximum run time per message: 1.5 seconds
- Maximum message buffer size: 2 GBytes

```
mpirun -np 512 IMB-MPI1 -npmin 512 alltoallv -iter 20 -time 1.5 -mem 2
```

Run the P_Read_shared benchmark with the minimum number of processes set to seven:

```
mpirun -np 14 IMB-IO P_Read_shared -npmin 7
```

Run the IMB-MPI1 benchmarks including PingPongAnySource and PingPingAnySource, but excluding the Alltoall and Alltoallv benchmarks. Set the transfer message sizes as 0, 4, 8, 16, 32, 64, 128:

```
mpirun -np 16 IMB-MPI1 -msglog 2:7 -include PingPongAnySource PingPingAnySource -
exclude Alltoall Alltoallv
```

Run the PingPong, PingPing, PingPongAnySource and PingPingAnySource benchmarks with the transfer message sizes 0, 2⁰, 2¹, 2², ..., 2¹⁶:

```
mpirun -np 4 IMB-MPI1 -msglog 16 PingPong PingPing PingPongAnySource
PingPingAnySource
```

6.2.1. Benchmark Selection Arguments

Benchmark selection arguments are a sequence of blank-separated strings. Each string is the name of a benchmark in exact spelling, case insensitive.

For example, the string IMB-MPI1 PingPong Allreduce specifies that you want to run PingPong and Allreduce benchmarks only:

```
mpirun -np 10 IMB-MPI1 PingPong Allreduce
```

By default, all benchmarks of the selected component are run.

6.2.2. -npmin Option

Specifies the minimum number of processes P_{\min} to run all selected benchmarks on. The P_{\min} value after `-npmin` must be an integer.

Given P_{\min} , the benchmarks run on the processes with the numbers selected as follows:

P_{\min} , $2P_{\min}$, $4P_{\min}$, ..., largest $2 \times P_{\min} < P$, P

NOTE

You may set P_{\min} to 1. If you set $P_{\min} > P$, Intel MPI Benchmarks interprets this value as $P_{\min} = P$.

For example, to run the `IMB-EXT` benchmarks with minimum number of processes set to five, call:

```
mpirun -np 11 IMB-EXT -npmin 5
```

By default, all active processes are selected as described in the [Running Intel® MPI Benchmarks](#) section.

6.2.3. -multi Option

Defines whether the benchmark runs in multiple mode. In this mode `MPI_COMM_WORLD` is split into several groups, which run simultaneously. The argument after `-multi` is a meta-symbol `<outflag>` that can take an integer value of 0 or 1:

- `outflag = 0` display only maximum timings (minimum throughputs) over all active groups
- `outflag = 1` report on all groups separately. The report may be long in this case.

This flag controls only benchmark results output style, the running procedure is the same for both `-multi 0` and `-multi 1` options.

When the number of processes running the benchmark is more than half of the overall number of ranks in `MPI_COMM_WORLD`, the multiple mode benchmark execution coincides with the non-multiple one, as not more than one process group can be created.

For example, if you run this command:

```
mpirun -np 16 IMB-MPI1 -multi 0 bcast -npmin 12
```

The benchmark will run in non-multiple mode, as the benchmarking starts from 12 processes, which is more than half of `MPI_COMM_WORLD`.

When a benchmark is set to be run on a set of different numbers of processes, its launch mode is determined based on the number of processes for each run. It is easy to tell if the benchmark is running in multiple mode by looking at the benchmark results header. When the name of the benchmark is printed out with the `Multi-` prefix, it is a multiple mode run.

For example, in the case of the same `Bcast` benchmark execution without `-npmin` parameter:

```
mpirun -np 16 IMB-MPI1 -multi 0 bcast
```

the benchmark will be executed 4 times: for 2, 4 and 8 processes in multiple mode, and for 16 processes in standard (non-multiple) mode. The benchmark results headers will look as follows:

```
#-----
# Benchmarking Multi-Bcast
# ( 8 groups of 2 processes each running simultaneous )
# Group  0:      0      1
#
# Group  1:      2      3
#
# Group  2:      4      5
#
# Group  3:      6      7
```

```

#
# Group 4:      8      9
#
# Group 5:     10     11
#
# Group 6:     12     13
#
# Group 7:     14     15
#
...
#-----
# Benchmarking Multi-Bcast
# ( 4 groups of 4 processes each running simultaneous )
# Group 0:      0      1      2      3
#
# Group 1:      4      5      6      7
#
# Group 2:      8      9     10     11
#
# Group 3:     12     13     14     15
#
...
#-----
# Benchmarking Multi-Bcast
# ( 2 groups of 8 processes each running simultaneous )
# Group 0:      0      1      2      3      4      5      6      7
#
# Group 1:      8      9     10     11     12     13     14     15
#
...
#-----
# Benchmarking Bcast
# #processes = 16

```

For each but the last execution the header contains:

- Multi- prefix before the benchmark name
- The list of `MPI_COMM_WORLD` ranks, aggregated in each group

By default, Intel® MPI Benchmarks run non-multiple benchmark flavors.

6.2.4. `-off_cache cache_size[,cache_line_size]` Option

Use the `-off_cache` flag to avoid cache re-use. If you do not use this flag (default), the same communications buffer is used for all repetitions of one message size sample. In this case, Intel® MPI Benchmarks reuses the cache, so throughput results might be non-realistic.

The argument after `off_cache` can be a single number (`cache_size`), two comma-separated numbers (`cache_size,cache_line_size`), or `-1`:

- `cache_size` is a float for an upper bound of the size of the last level cache, in MB.
- `cache_line_size` is assumed to be the size of a last level cache line (can be an upper estimate).
- `-1` uses values defined in `IMB_mem_info.h`. In this case, make sure to define values for `cache_size` and `cache_line_size` in `IMB_mem_info.h`.

The sent/received data is stored in buffers of size `~2x MAX(cache_size, message_size)`. When repetitively using messages of a particular size, their addresses are advanced within those buffers so that a single message is at least 2 cache lines after the end of the previous message. When these buffers are filled up, they are reused from the beginning.

`-off_cache` is effective for IMB-MPI1 and IMB-EXT. Avoid using this option for IMB-IO.

Examples:

Use the default values defined in `IMB_mem_info.h`:

```
-off_cache -1
```

2.5 MB last level cache, default line size:

```
-off_cache 2.5
```

16 MB last level cache, line size 128:

```
-off_cache 16,128
```

The `off_cache` mode might also be influenced by eventual internal caching with the Intel® MPI Library. This could make results interpretation complicated.

Default: no cache control.

6.2.5. -iter Option

Use this option to control the number of iterations executed by every benchmark.

By default, the number of iterations is controlled through parameters `MSGSPERSAMPLE`, `OVERALL_VOL`, `MSGS_NONAGGR`, and `ITER_POLICY` defined in `IMB_settings.h`.

You can optionally add one or more arguments after the `-iter` flag, to override the default values defined in `IMB_settings.h`. Use the following guidelines for the optional arguments:

- To override the `MSGSPERSAMPLE` value, use a single integer.
- To override the `OVERALL_VOL` value, use two comma-separated integers. The first integer defines the `MSGSPERSAMPLE` value. The second integer overrides the `OVERALL_VOL` value.
- To override the `MSGS_NONAGGR` value, use three comma-separated integer numbers. The first integer defines the `MSGSPERSAMPLE` value. The second integer overrides the `OVERALL_VOL` value. The third overrides the `MSGS_NONAGGR` value.
- To override the `-iter_policy` argument, enter it after the integer arguments, or right after the `-iter` flag if you do not use any other arguments.

Examples:

To define `MSGSPERSAMPLE` as 2000, and `OVERALL_VOL` as 100, use the following string:

```
-iter 2000,100
```

To define `MSGS_NONAGGR` as 150, you need to define values for `MSGSPERSAMPLE` and `OVERALL_VOL` as shown in the following string:

```
-iter 1000,40,150
```

To define `MSGSPERSAMPLE` as 2000 and set the `multiple_np` policy, use the following string (see `-iter_policy`):

```
-iter 2000,multiple_np
```

6.2.6. -iter_policy Option

Use this option to set a policy for automatic calculation of the number of iterations. Use one of the following arguments to override the default `ITER_POLICY` value defined in `IMB_settings.h`:

Policy	Description
dynamic	Reduces the number of iterations when the maximum run time per sample (see <code>-time</code>) is

	expected to be reached. Using this policy ensures faster execution, but may lead to inaccuracy of the results.
<code>multiple_np</code>	Reduces the number of iterations when the message size is getting bigger. Using this policy ensures the accuracy of the results, but may lead to longer execution time. You can control the execution time through the <code>-time</code> option.
<code>auto</code>	Automatically chooses which policy to use: <ul style="list-style-type: none"> • applies <code>multiple_np</code> to collective operations where one of the processes acts as the root of the operation (for example, <code>MPI_Bcast</code>) • applies <code>dynamic</code> to all other types of operations
<code>off</code>	The number of iterations does not change during the execution.

You can also set the policy through the `-iter` option. See `-iter`.

By default, the `ITER_POLICY` defined in `IMB_settings.h` is used.

6.2.7. `-time` Option

Specifies the number of seconds for the benchmark to run per message size. The argument after `-time` is a floating-point number.

The combination of this flag with the `-iter` flag or its default alternative ensures that the Intel® MPI Benchmarks always chooses the maximum number of repetitions that conform to all restrictions.

A rough number of repetitions per sample to fulfill the `-time` request is estimated in preparatory runs that use ~1 second overhead.

Default: `-time` is activated. The floating-point value specifying the run-time seconds per sample is set in the `SECS_PER_SAMPLE` variable defined in `IMB_settings.h`, or `IMB_settings_io.h`.

6.2.8. `-mem` Option

Specifies the number of GB to be allocated per process for the message buffers. If the size is exceeded, a warning is returned, stating how much memory is required for the overall run.

The argument after `-mem` is a floating-point number.

Default: the memory is restricted by `MAX_MEM_USAGE` defined in `IMB_mem_info.h`.

6.2.9. `-input <File>` Option

Use the ASCII input file to select the benchmarks. For example, the `IMB_SELECT_EXT` file looks as follows:

```
#
# IMB benchmark selection file
#
# Every line must be a comment (beginning with #), or it
# must contain exactly one IMB benchmark name
#
#Window
Unidir_Get
#Unidir_Put
#Bidir_Get
#Bidir_Put
Accumulate
```

With the help of this file, the following command runs only `Unidir_Get` and `Accumulate` benchmarks of the `IMB-EXT` component:

```
mpirun .... IMB-EXT -input IMB_SELECT_EXT
```

6.2.10. -msglen <File> Option

Enter any set of non-negative message lengths to an ASCII file, line by line, and call the Intel® MPI Benchmarks with arguments:

```
-msglen Lengths
```

The `Lengths` value overrides the default message lengths. For `IMB-IO`, the file defines the I/O portion lengths.

6.2.11. -map PxQ Option

Use this option to re-number the ranks for parallel processes in `MPI_COMM_WORLD` along rows of the matrix:

0	P	...	(Q-2)P	(Q-1)P
1				
...				
P-1	2P-1		(Q-1)P-1	QP-1

For example, to run `Multi-PingPong` between two nodes, `P` processes on each (`ppn=P`), with each process on one node communicating with its counterpart on the other, call:

```
mpirun -np <2P> IMB-MPI1 -map <P>x2 -multi 0 PingPong
```

or:

```
mpirun -np <2P> IMB-MPI1 -map  
x2 -multi 1 PingPong
```

The `P*Q` product must not be less than the total number of ranks, otherwise a command line parsing error is issued. The `P=1` and `Q=1` cases are treated as meaningless and are just ignored.

See the examples below for a more detailed explanation of the `-map` option.

Example 1. `PingPong` benchmark with a `4x2` map, 8 ranks in total on 2 nodes.

a) `-map 4x2` combined with `-multi <outflag>`, multiple mode:

```
mpirun -np 8 IMB-MPI1 -map 4x2 -multi 0 PingPong
```

The `MPI_COMM_WORLD` communicator originally consists of 8 ranks:

```
{ 0, 1, 2, 3, 4, 5, 6, 7 }
```

The given option `-map 4x2` reorders this set of ranks into the following set (in terms of `MPI_COMM_WORLD` ranks):

```
{ 0, 4, 1, 5, 2, 6, 3, 7 }
```

The `-multi <outflag>` makes Intel® MPI Benchmarks split the communicator into 4 subgroups, 2 ranks in each, with a `MPI_Comm_split` call. As a result, the communicator looks like this:

```
{ { 0, 1 }, { 0, 1 }, { 0, 1 }, { 0, 1 } }
```

In terms of the original `MPI_COMM_WORLD` rank numbers, this means that there are 4 groups of ranks, and the benchmark is executed simultaneously for each:

```
Group 1: { 0, 4 }; Group 2: { 1, 5 }; Group 3: { 2, 6 }; Group 4: { 3, 7 }
```


This grouping is shown in the benchmark output header and can be easily verified:

```
#-----
# Benchmarking Multi-PingPong
# ( 4 groups of 2 processes each running simultaneous )
# Group 0:      0      4
#
# Group 1:      1      5
#
# Group 2:      2      6
#
# Group 3:      3      7
```

As can be seen in the output, ranks in the pairs belong to different nodes, so this benchmark execution will measure inter-node communication parameters.

b) -map 4x2 without -multi <outflag>, non-multiple mode:

```
mpirun -np 8 IMB-MPI1 -map 4x2 PingPong
```

The same rules or rank numbers transformation are applied in this case, but since the multiple mode is not set, communicator splitting is not performed. Only two ranks will participate in actual communication, as the PingPong benchmark covers a pair of ranks only. The benchmark will cover only the first group:

Group: { 0, 4 }

and the other ranks from MPI_COMM_WORLD will be idle. This is reflected in the benchmark results output:

```
#-----
# Benchmarking PingPong
# #processes = 2; rank order (rowwise):
#   0      4
#
# ( 6 additional processes waiting in MPI_Barrier)
```

Example 2. Biband benchmark with the 2x4 map, 8 ranks in total on 2 nodes

a) -map 2x4 combined with -multi <outflag>, multiple mode:

```
mpirun -np 8 IMB-MPI1 -map 2x4 -multi 0 Biband
```

The MPI_COMM_WORLD communicator originally consists of 8 ranks:

```
{ 0, 1, 2, 3, 4, 5, 6, 7 }
```

The given option -map 2x4 reorders this set of ranks into the following set (in terms of MPI_COMM_WORLD ranks):

```
{ 0, 2, 4, 6, 1, 3, 5, 7 }
```

The communicator splitting, which is required by the -multi <outflag> option, then depends on the number of processes to be used for execution. In this case, 2-process, 4-process and 8-process run cycles will be executed:

1) NP=2: Reordered communicator is split into 4 groups of 2 processes because of the multiple mode:

```
{ { 0, 1 }, { 0, 1 }, { 0, 1 }, { 0, 1 } }
```

In terms of MPI_COMM_WORLD ranks, the groups are:

Group 1: { 0, 2 }; Group 2: { 1, 3 }; Group 3: { 4, 6 }; Group 3: { 5, 7 }

```
#-----
# Benchmarking Multi-Biband
# ( 4 groups of 2 processes each running simultaneous )
# Group 0:      0      2
#
# Group 1:      1      3
```

```
#
# Group 2:      4      6
#
# Group 3:      5      7
#
```

All the pairs belong to a single node here, so no cross-node benchmarking is performed in this case.

2) NP=4: Reordered communicator is split into 2 groups of 4 processes because of the multiple mode:

```
{ { 1, 2, 3, 4 }, { 1, 2, 3, 4 } }
```

In terms of `MPI_COMM_WORLD` ranks, the groups are:

Group 1: { 0, 2, 4, 6 }; Group 2: { 1, 3, 5, 7 };

```
#-----
# Benchmarking Multi-Biband
# ( 2 groups of 4 processes each running simultaneous )
# Group 0:      0      2      4      6
#
# Group 1:      1      3      5      7
#
```

Execution groups mix ranks from different nodes in this case, and due to the `Biband` benchmark pairs ordering rules (see [Biband](#)), only inter-node pairs will be tested.

3) NP=8: No communicator splitting can be performed, since the ranks can fit only a single group:

Group: { 0, 2, 4, 6, 1, 3, 5, 7 }

```
#-----
# Benchmarking Biband
# #processes = 8; rank order (rowwise):
#   0      2      4      6
#
#   1      3      5      7
#
```

The group is half-by-half spread within 2 execution nodes, but as a result of reordering all the pairs in the `Biband` test (see [Biband](#)) appear to be intra-node ones, which is totally opposite to the default case (no `-map` option) and the `NP=4` case.

b) `-map 2x4` without `-multi <outflag>` option, non-multiple mode:

```
mpirun -np 8 IMB-MPI1 -map 2x4 Biband
```

The same rules or rank numbers transformation are applied in this case, but since the multiple mode is not set, no communicator splitting is performed. The set of ranks that are covered by the benchmark depends on the number of processes to be used for execution. In this case, 2-process, 4-process and 8-process run cycles will be executed, and they just use the first 2, 4 and 8 ranks of the reordered communicator for actual benchmark execution:

1) NP=2: first 2 ranks of the reordered communicator form the group (in terms of `MPI_COMM_WORLD` ranks):

Group: { 0, 2 };

```
#-----
# Benchmarking Biband
# #processes = 2; rank order (rowwise):
#   0      2
#
# ( 6 additional processes waiting in MPI_Barrier)
```

2) NP=4: first 4 ranks of the reordered communicator form the group (in terms of `MPI_COMM_WORLD` ranks):

```
Group: { 0, 2, 4, 6 };
```

```
#-----
# Benchmarking Biband
# #processes = 4; rank order (rowwise):
#   0     2     4     6
#
# ( 4 additional processes waiting in MPI_Barrier)
```

3) NP=8: all the ranks of the reordered communicator form the group (in terms of MPI_COMM_WORLD ranks):

```
Group: { 0, 2, 4, 6, 1, 3, 5, 7 }
```

```
#-----
# Benchmarking Biband
# #processes = 8; rank order (rowwise):
#   0     2     4     6
#
#   1     3     5     7
#
```

As can be seen in the output, the NP=2 and NP=4 executions of the Biband test launched with and without the `-multi <outflag>` option are almost the same. The only difference is that in the non-multiple mode only one group is active, and all other processes are idle. For the NP=8 case, the Biband benchmark executions performed with and without the `-multi <outflag>` option are completely identical.

6.2.12. `-include [[benchmark1] benchmark2 ...]`

Specifies the list of additional benchmarks to run. For example, to add PingPongAnySource and PingPingAnySource benchmarks, call:

```
mpirun -np 2 IMB-MPI1 -include PingPongAnySource PingPingAnySource
```

6.2.13. `-exclude [[benchmark1] benchmark2 ...]`

Specifies the list of benchmarks to be excluded from the run. For example, to exclude Alltoall and Allgather, call:

```
mpirun -np 2 IMB-MPI1 -exclude Alltoall Allgather
```

6.2.14. `-msglog [<minlog>:]<maxlog>`

This option allows you to control the lengths of the transfer messages. This setting overrides the MINMSGLOG and MAXMSGLOG values. The new message sizes are 0, 2^{minlog} , ..., 2^{maxlog} .

For example, if you run the following command line:

```
mpirun -np 2 IMB-MPI1 -msglog 3:7 PingPong
```

Intel® MPI Benchmarks selects the lengths 0, 8, 16, 32, 64, 128, as shown below:

```
#-----
# Benchmarking PingPong
# #processes = 2
#-----
#bytes #repetitions      t[μsec]  Mbytes/sec
      0           1000        0.70         0.00
       8           1000        0.73        10.46
      16           1000        0.74        20.65
      32           1000        0.94        32.61
      64           1000        0.94        65.14
     128           1000        1.06       115.16
```

Alternatively, you can specify only the `maxlog` value, enter:

```
mpirun -np 2 IMB-MPI1 -msglog 3 PingPong
```

In this case Intel® MPI Benchmarks selects the lengths 0, 1, 2, 4, 8:

```
#-----
# Benchmarking PingPong
# #processes = 2
#-----
#bytes  #repetitions      t[μsec]    Mbytes/sec
      0           1000         0.69         0.00
      1           1000         0.72         1.33
      2           1000         0.71         2.69
      4           1000         0.72         5.28
      8           1000         0.73        10.47
```

6.2.15. -thread_level Option

This option specifies the desired thread level for `MPI_Init_thread()`. See description of `MPI_Init_thread()` for details. The option is available only if the Intel® MPI Benchmarks is built with the `USE_MPI_INIT_THREAD` macro defined. Possible values for `<level>` are `single`, `funneled`, `serialized`, and `multiple`.

6.2.16. -sync Option

This option is relevant only for benchmarks measuring collective operations. It controls whether all ranks are synchronized after every iteration step by means of the `MPI_Barrier` operation. The `-sync` option can take the following arguments:

Argument	Description
0 off disable no	Disables processes synchronization at each iteration step.
1 on enable yes	Enables processes synchronization at each iteration step. This is the default value.

6.2.17. -root_shift Option

This options is relevant only for benchmarks measuring collective operations that utilize the root concept (for example `MPI_Bcast`, `MPI_Reduce`, `MPI_Gather`, etc). It defines whether the root is changed at every iteration step or not. The `-root_shift` option can take the following arguments:

Argument	Description
0 off disable no	Disables root change at each iteration step. Rank 0 acts as a root at each iteration step. This is the default value.
1 on enable yes	Enables root change at each iteration step. Root rank is changed in a round-robin fashion.

6.2.18. -imb_barrier Option

Implementation of the `MPI_Barrier` operation may vary depending on the MPI implementation. Each MPI implementation might use a different algorithm for the barrier, with possibly different synchronization characteristics, so the Intel MPI Benchmarks results may vary significantly as a result of `MPI_Barrier`

implementation differences. The internal, MPI-independent barrier function `IMB_barrier` is provided to make the synchronization effect more reproducible.

Use this option to use the `IMB_barrier` function to get consistent results of collective operation benchmarks.

Argument	Description
0 off disable no	Use the standard <code>MPI_Barrier</code> operation. This is the default value.
1 on enable yes	Use the internal barrier implementation for synchronization.

6.3. Parameters Controlling Intel® MPI Benchmarks

Parameters controlling the default settings of the Intel® MPI Benchmarks are set by preprocessor definition in files `IMB_settings.h` (for IMB-MPI1 and IMB-EXT benchmarks) and `IMB_settings_io.h` (for IMB-IO benchmarks). Both include files have identical structure, but differ in the predefined parameter values.

To enable the optional mode, define the `IMB_OPTIONAL` parameter in the `IMB_settings.h/IMB_settings_io.h`. After you change the settings in the optional section, you need to recompile the Intel® MPI Benchmarks.

The following table describes the Intel MPI Benchmarks parameters and lists their values for the standard mode.

Parameter	Values in <code>IMB_settings.h</code>	Values in <code>IMB_settings_io.h</code>	Description
<code>USE_MPI_INIT_THREAD</code>	Not set	Not set	Set to initialize Intel® MPI Benchmarks by <code>MPI_Init_thread()</code> instead of <code>MPI_Init()</code>
<code>IMB_OPTIONAL</code>	Not set	Not set	Set to activate optional settings
<code>MINMSGLOG</code>	0	0	The second smallest data transfer size is $\max(\text{unit}, 2^{\text{MINMSGLOG}}$ (the smallest size is always 0), where <code>unit=sizeof(float)</code> for reductions, <code>unit=1</code> for all other cases. You can override this parameter value using the <code>-msglog</code> flag.
<code>MAXMSGLOG</code>	22	24	The largest message size used is $2^{\text{MAXMSGLOG}}$ You can override this parameter value using the <code>-msglog</code> flag.
<code>ITER_POLICY</code>	<code>imode_dynamic</code>		The policy used for calculating the number of iterations. You can override this parameter value using the <code>-iter_policy</code> or <code>-iter</code> flag.
<code>MSGSPERSAMPLE</code>	1000	50	The maximum repetition count for all IMB-MPI1 benchmarks. You can override this parameter value

			using the <code>-iter</code> flag.
MSGS_NONAGGR	100	10	The maximum repetition count for non-aggregate benchmarks (relevant only for <code>IMB-EXT</code>). You can override this parameter value using the <code>-time</code> flag.
OVERALL_VOL	40 Mbytes	16*1048576	For all sizes smaller than <code>OVERALL_VOL</code> , the repetition count is reduced so that not more than <code>OVERALL_VOL</code> bytes are processed all in all. This permits you to avoid unnecessary repetitions for large message sizes. Finally, the real repetition count for message size <code>X</code> is $\text{MSGSPERSAMPLE } (X=0),$ $\max(1, \min(\text{MSGSPERSAMPLE}, \text{OVERALL_VOL}/X))$ $(X>0)$ Note that <code>OVERALL_VOL</code> does <i>not</i> restrict the size of the maximum data transfer. $2^{\text{MAXMSGLOG}}$ <code>OVERALL_VOL</code> . You can override this parameter value using the <code>-mem</code> flag.
SECS_PER_SAMPLE	10		Number of iterations is dynamically set so that this number of run time seconds is not exceeded per message length.
N_BARR	2	2	Number of <code>MPI_Barrier</code> for synchronization.
TARGET_CPU_SECS	0.01 seconds	0.1 seconds	CPU seconds (as float) to run concurrently with nonblocking benchmarks (irrelevant for <code>IMB-MPI1</code>)

In the example below, the `IMB_settings_io.h` file has the `IMB_OPTIONAL` parameter enabled, so that user-defined parameters are used. I/O sizes of 32 and 64 MB, and a smaller repetition count are selected, extending the standard mode tables. You can modify the optional values as required.

```
#define FILENAME IMB_out
#define IMB_OPTIONAL
#ifdef IMB_OPTIONAL
#define MINMSGLOG 25
#define MAXMSGLOG 26
#define MSGSPERSAMPLE 10
#define MSGS_NONAGGR 10
#define OVERALL_VOL 16*1048576
#define SECS_PER_SAMPLE 10
#define TARGET_CPU_SECS 0.1 /* unit seconds */
#define N_BARR 2
#else
/*Do not change anything below this line*/
#define MINMSGLOG 0
#define MAXMSGLOG 24
#define MSGSPERSAMPLE 50
#define MSGS_NONAGGR 10
#define OVERALL_VOL 16*1048576
#define TARGET_CPU_SECS 0.1 /* unit seconds */
```

```
#define N_BARR    2
#endif
```

If `IMB_OPTIONAL` is deactivated, Intel MPI Benchmarks uses the default standard mode values.

6.4. Hard-Coded Settings

The sections below describe Intel® MPI Benchmarks hard-coded settings. These are the settings that you can change through the command line, or in the source code directly:

- [Communicators, Active Processes](#)
- [Message /I-O Buffer Lengths](#)
- [Buffer Initialization](#)
- [Other Preparations for Benchmarking](#)
- [Warm-Up Phase \(MPI-1, EXT\)](#)
- [Synchronization](#)
- [Actual Benchmarking](#)

6.4.1. Communicators, Active Processes

Communicator management is repeated in every "select MY_COMM" step. If it exists, the previous communicator is freed. When you run $Q \leq P$ processes, the first Q ranks of `MPI_COMM_WORLD` are put into one group, and the remaining $P-Q$ get `MPI_COMM_NULL`.

The group of `MY_COMM` calls the active processes group.

6.4.2. Message/I-O Buffer Lengths

IMB-MPI1, IMB-EXT

Set in `IMB_settings.h` and used unless the `-msglen` flag is selected.

IMB-IO

Set in `IMB_settings_io.h` and used unless the `-msglen` flag is selected.

6.4.3. Buffer Initialization

Communication and I/O buffers are dynamically allocated as `void*` and used as `MPI_BYTE` buffers for all benchmarks except `Accumulate`, see [Memory Requirements](#). To assign the buffer contents, a cast to an assignment type is performed. This facilitates result checking which may become necessary. Besides, a sensible data type is mandatory for `Accumulate`.

Intel® MPI Benchmarks sets the buffer assignment type `assign_type` in

`IMB_settings.h/IMB_settings_io.h`. Currently, `int` is used for `IMB-IO`, `float` for `IMB-EXT`. The values are set by a macro definition as follows.

For `IMB-EXT` benchmarks:

```
#define BUF_VALUE(rank,i) (0.1*((rank)+1)+(float)(i))
```

For `IMB-IO` benchmarks:

```
#define BUF_VALUE(rank,i) 10000000*(1+rank)+i%10000000
```

In every initialization, communication buffers are seen as typed arrays and initialized as follows:

```
((assign_type*)buffer)[i] = BUF_VALUE(rank,i;
```

where rank is the MPI rank of the calling process.

6.4.4. Other Preparations for Benchmarking

Window (IMB-EXT and IMB-RMA)

1. An Info is set and MPI_Win_create is called, creating a window of size X for MY_COMM.
2. For IMB-EXT, MPI_Win_fence is called to start an access epoch.

NOTE

IMB-RMA benchmarks do not require MPI_Win_fence since they use passive target communication mode.

File (IMB-IO)

To initialize the IMB-IO file, follow these steps:

1. Select a file name. This parameter is located in the IMB_settings_io.h include file. In the case of a multi-<MPI command>, a suffix _g<groupid> is appended to the name. If the file name is per process, a second event suffix _<rank> is appended.
2. Delete the file if it exists: open the file with MPI_MODE_DELETE_ON_CLOSE and close it. The file is deleted.
3. Select a communicator to open the file: MPI_COMM_SELF for S_benchmarks and P_[ACTION]_priv.
4. Select a mode: MPI_MODE_CREATE | MPI_MODE_RDWR
5. Select an info routine as explained below.

Info

Intel® MPI Benchmarks uses an external function User_Set_Info which you implement for your local system. The default version is:

```
#include mpi.h
void User_Set_Info ( MPI_Info* opt_info)
#ifdef MPIIO
{/* Set info for all MPI_File_open calls */
*opt_info = MPI_INFO_NULL;
}
#endif
#ifdef EXT
{/* Set info for all MPI_Win_create calls */
*opt_info = MPI_INFO_NULL;
}
#endif
```

The Intel® MPI Benchmarks has no assumptions or restrictions on the implementation of this routine.

View (IMB-IO)

The file view is determined by the following settings:

- disp = 0,
- datarep = native
- etype, filetypeas defined in the [IMB-IO Blocking Benchmarks](#) section

- `info` as defined in the "Info" section above

6.4.5. Warm-Up Phase (IMB-MPI1, IMB-EXT, IMB-NBC, and IMB-RMA)

Before starting the actual benchmark measurement for IMB-MPI1, IMB-EXT, IMB-NBC, and IMB-RMA, the selected benchmark is executed `N_WARMUP` times with a `sizeof(assign_type)` message length. The `N_WARMUP` value is defined in `IMB_settings.h`, see [Parameters Controlling Intel® MPI Benchmarks](#) for details. The warm-up phase eliminates the initialization overhead from the benchmark measurement.

6.4.6. Synchronization

Before the actual benchmark measurement is performed, several consecutive barrier calls are made to ensure perfect processes synchronization. The `N_BARR` constant defines the number of consecutive calls. The constant is defined in `IMB_settings.h` and `IMB_settings_io.h`, with the current value of 2.

The barrier calls are either ordinary `MPI_Barrier(comm)` calls, or `IMB_barrier(comm)` calls (in case the `-imb_barrier` option is specified).

See figure [Control flow of IMB](#) to understand the way all the processes are synchronized.

6.4.7. Actual Benchmarking

To reduce measurement errors caused by insufficient clock resolution, every benchmark is run repeatedly. The repetition count is as follows:

For IMB-MPI1, IMB-NBC, and aggregate flavors of IMB-EXT, IMB-IO, and IMB-RMA benchmarks, the repetition count is `MSGSPERSAMPLE`. This constant is defined in `IMB_settings.h` and `IMB_settings_io.h`, with 1000 and 50 values, respectively.

To avoid excessive run times for large transfer sizes `X`, an upper bound is set to `OVERALL_VOL/X`. The `OVERALL_VOL` value is defined in `IMB_settings.h` and `IMB_settings_io.h`, with 4MB and 16MB values, respectively.

Given transfer size `X`, the repetition count for all aggregate benchmarks is defined as follows:

```
n_sample = MSGSPERSAMPLE (X=0)
```

```
n_sample = max(1,min(MSGSPERSAMPLE,OVERALL_VOL/X)) (X>0)
```

The repetition count for non-aggregate benchmarks is defined completely analogously, with `MSGSPERSAMPLE` replaced by `MSGNONAGGR`. It is recommended to reduce the repetition count as non-aggregate run times are usually much longer.

In the following examples, *elementary transfer* means a pure function (`MPI_[Send, ...], MPI_Put, MPI_Get, MPI_Accumulate, MPI_File_write_XX, MPI_File_read_XX`), without any further function call. Assured completion transfer completion is:

- IMB-EXT benchmarks: `MPI_Win_fence`
- IMB-IO Write benchmarks: a triplet
`MPI_File_sync/MPI_Barrier(file_communicator)/MPI_File_sync`
- IMB-RMA benchmarks: `MPI_Win_flush, MPI_Win_flush_all, MPI_Win_flush_local, or MPI_Win_flush_local_all`
- Other benchmarks: empty

MPI-1 Benchmarks

```
for ( i=0; i<N_BARR; i++ ) MPI_Barrier(MY_COMM)
```

```
time = MPI_Wtime()
for ( i=0; i<n_sample; i++ )
    execute MPI pattern
time = (MPI_Wtime()-time)/n_sample
```

IMB-EXT and Blocking I/O Benchmarks

For aggregate benchmarks, the kernel loop looks as follows:

```
for ( i=0; i<N_BARR; i++ )MPI_Barrier(MY_COMM)
/* Negligible integer (offset) calculations ... */
time = MPI_Wtime()
for ( i=0; i<n_sample; i++ )
    execute elementary transfer
    assure completion of all transfers
time = (MPI_Wtime()-time)/n_sample
```

For non-aggregate benchmarks, every transfer is completed before going on to the next transfer:

```
for ( i=0; i<N_BARR; i++ )MPI_Barrier(MY_COMM)
/* Negligible integer (offset) calculations ... */
time = MPI_Wtime()
for ( i=0; i<n_sample; i++ )
{
    execute elementary transfer
    assure completion of transfer
}
time = (MPI_Wtime()-time)/n_sample
```

Non-blocking I/O Benchmarks

A nonblocking benchmark has to provide three timings:

- `t_pure` - blocking pure I/O time
- `t_ovrl` - nonblocking I/O time concurrent with CPU activity
- `t_CPU` - pure CPU activity time

The actual benchmark consists of the following stages:

- Calling the equivalent blocking benchmark, as defined in [Actual Benchmarking](#) and taking benchmark time as `t_pure`.
- Closing and re-opening the related file(s).
- Re-synchronizing the processes.
- Running the nonblocking case, concurrent with CPU activity (exploiting `t_CPU` when running undisturbed), taking the effective time as `t_ovrl`.

You can set the desired CPU time `t_CPU` in `IMB_settings_io.h`:

```
#define TARGET_CPU_SECS 0.1 /* unit seconds */
```

6.5. Checking Results

To check whether your MPI implementation is working correctly, you can use the `CPP` flag `-DCHECK`.

Activate the `CPP` flag `-DCHECK` through the `CPPFLAGS` variable and recompile the Intel® MPI Benchmarks executable files. Every message passing result from the Intel® MPI Benchmarks are checked against the expected outcome. Output tables contain an additional column called Defects that displays the difference as floating-point numbers.

NOTE

The `-DCHECK` results are not valid as real benchmark data. Deactivate `-DCHECK` and recompile to get the proper results.

6.6. Output

The benchmark output includes the following information:

- General information:
machine, system, release, and version are obtained by `IMB_g_info.c`.
- The calling sequence (command-line flags) are repeated in the output chart
- Results for the non-multiple mode

After a benchmark completes, three time values are available, extended over the group of active processes:

- `Tmax` - the maximum time
- `Tmin` - the minimum time
- `Tavg` - the average time

The time unit is μ .

Single Transfer Benchmarks:

Display `X = message size [bytes]`, `T=Tmax[usec]`, `bandwidth = X / T`

Parallel Transfer Benchmarks:

Display `X = message; size`, `Tmax`, `Tmin` and `Tavg`, `bandwidth based on time = Tmax`

NOTE

IMB-RMA benchmarks show only bare timings for Parallel Transfer benchmarks.

Collective Benchmarks:

Display `X = message size; (except for Barrier)`, `Tmax`, `Tmin`; and `Tavg`

Results for the multiple mode

- `-multi 0`: the same as above, with min, avg over all groups.
- `-multi 1`: the same for all groups, max, min, avg over single groups.

6.6.1. Sample 1 - IMB-MPI1 PingPong Allreduce

The following example shows the results of the PingPong and Allreduce benchmark:

```
<..> np 2 IMB-MPI1 PingPong Allreduce
#-----
# Intel ® MPI Benchmark Suite V3.2, MPI1 part
#-----
# Date           : Thu Sep  4 13:20:07 2008
# Machine        : x86_64
# System         : Linux
# Release        : 2.6.9-42.ELsmp
# Version        : #1 SMP Wed Jul 12 23:32:02 EDT 2006
# MPI Version    : 2.0
# MPI Thread Environment: MPI_THREAD_SINGLE
```

```

# New default behavior from Version 3.2 on:
# the number of iterations per message size is cut down
# dynamically when a certain run time (per message size sample)
# is expected to be exceeded. Time limit is defined by variable
# SECS_PER_SAMPLE (=> IMB_settings.h)
# or through the flag => -time

# Calling sequence was:

# ./IMB-MPI1 PingPong Allreduce

# Minimum message length in bytes: 0
# Maximum message length in bytes: 4194304
#
# MPI_Datatype           : MPI_BYTE
# MPI_Datatype for reductions : MPI_FLOAT
# MPI_Op                 : MPI_SUM
#
#
# List of Benchmarks to run:

# PingPong
# Allreduce

#-----
# Benchmarking PingPong
# #processes = 2
#-----


| #bytes  | #repetitions | t[μsec] | Mbytes/sec |
|---------|--------------|---------|------------|
| 0       | 1000         | ..      | ..         |
| 1       | 1000         |         |            |
| 2       | 1000         |         |            |
| 4       | 1000         |         |            |
| 8       | 1000         |         |            |
| 16      | 1000         |         |            |
| 32      | 1000         |         |            |
| 64      | 1000         |         |            |
| 128     | 1000         |         |            |
| 256     | 1000         |         |            |
| 512     | 1000         |         |            |
| 1024    | 1000         |         |            |
| 2048    | 1000         |         |            |
| 4096    | 1000         |         |            |
| 8192    | 1000         |         |            |
| 16384   | 1000         |         |            |
| 32768   | 1000         |         |            |
| 65536   | 640          |         |            |
| 131072  | 320          |         |            |
| 262144  | 160          |         |            |
| 524288  | 80           |         |            |
| 1048576 | 40           |         |            |
| 2097152 | 20           |         |            |
| 4194304 | 10           |         |            |


#-----
# Benchmarking Allreduce
# ( #processes = 2 )
#-----

```

```

#bytes  #repetitions  t_min[μsec]  t_max[μsec]  t_avg[μsec]
    0           1000      ..           ..           ..
    4           1000
    8           1000
   16           1000
   32           1000
   64           1000
  128           1000
  256           1000
  512           1000
 1024           1000
 2048           1000
 4096           1000
 8192           1000
16384           1000
32768           1000
65536            640
131072           320
262144           160
524288            80
1048576           40
2097152           20
4194304           10

# All processes entering MPI_Finalize

```

6.6.2. Sample 2 - IMB-MPI1 PingPing Allreduce

The following example shows the results of the PingPing

```

<...>
-np 6 IMB-MPI1
  pingping allreduce -map 2x3 -msglen Lengths -multi 0
Lengths file:
0
100
1000
10000
100000
1000000
#-----
# Intel ® MPI Benchmark Suite V3.2.2, MPI1 part
#-----
# Date                : Thu Sep 4 13:26:03 2008
# Machine              : x86_64
# System               : Linux
# Release              : 2.6.9-42.ELsmp
# Version              : #1 SMP Wed Jul 12 23:32:02 EDT 2006
# MPI Version          : 2.0
# MPI Thread Environment: MPI_THREAD_SINGLE

# New default behavior from Version 3.2 on:
# the number of iterations per message size is cut down
# dynamically when a certain run time (per message size sample)
# is expected to be exceeded. Time limit is defined by variable
# SECS_PER_SAMPLE (=> IMB_settings.h)
# or through the flag => -time

# Calling sequence was:

```

```

# IMB-MPI1 pingping allreduce -map 3x2 -msglen Lengths
#           -multi 0

# Message lengths were user-defined
#
# MPI_Datatype           : MPI_BYTE
# MPI_Datatype for reductions : MPI_FLOAT
# MPI_Op                  : MPI_SUM
#
#
# List of Benchmarks to run:
# (Multi-)PingPing
# (Multi-)Allreduce
#-----
# Benchmarking Multi-PingPing
# ( 3 groups of 2 processes each running simultaneously )
# Group  0:      0      3
#
# Group  1:      1      4
#
# Group  2:      2      5
#
#-----
# bytes #rep.s t_min[μsec] t_max[μsec] t_avg[μsec] Mbytes/sec
#      0    1000      ..      ..      ..      ..
#     100    1000
#    1000    1000
#   10000    1000
#  100000    419
# 1000000     41
#-----
# Benchmarking Multi-Allreduce
# ( 3 groups of 2 processes each running simultaneously )
# Group  0:      0      3
#
# Group  1:      1      4
#
# Group  2:      2      5
#
#-----
#bytes #repetitions t_min[μsec] t_max[μsec] t_avg[μsec]
#      0          1000      ..      ..      ..
#     100          1000
#    1000          1000
#   10000          1000
#  100000          419
# 1000000           41
#-----
# Benchmarking Allreduce
#
#processes = 4; rank order (rowwise):
#      0      3
#
#      1      4
#
# ( 2 additional processes waiting in MPI_Barrier)
#-----
# bytes #repetitions t_min[μsec] t_max[μsec] t_avg[μsec]

```

```

    0          1000      ..      ..      ..
   100         1000
  1000         1000
 10000         1000
100000         419
1000000         41
#-----
# Benchmarking Allreduce
#
# processes = 6; rank order (rowwise):
#   0      3
#
#   1      4
#
#   2      5
#
#-----
# bytes #repetitions  t_min[μsec]  t_max[μsec]  t_avg[μsec]
#   0          1000      ..      ..      ..
#  100         1000
# 1000         1000
#10000         1000
#100000         419
#1000000         41
# All processes entering MPI_Finalize

```

6.6.3. Sample 3 - IMB-IO p_write_indv

The following example shows the results of the p_write_indv benchmark:

```

<..> IMB-IO -np 2 p_write_indv -npmin 2
#-----
# Date           : Thu Sep  4 13:43:34 2008
# Machine        : x86_64
# System         : Linux
# Release        : 2.6.9-42.ELsmp
# Version        : #1 SMP Wed Jul 12 23:32:02 EDT 2006
# MPI Version    : 2.0
# MPI Thread Environment: MPI_THREAD_SINGLE

# New default behavior from Version 3.2 on:
# the number of iterations per message size is cut down
# dynamically when a certain run time (per message size sample)
# is expected to be exceeded. Time limit is defined by variable
# SECS_PER_SAMPLE (=> IMB_settings.h)
# or through the flag => -time

# Calling sequence was:
# ./IMB-IO p_write_indv -npmin 2
# Minimum io portion in bytes:    0
# Maximum io portion in bytes:   16777216
#
#
#
# List of Benchmarks to run:
# P_Write_Indv
#-----
# Benchmarking P_Write_Indv

```

```

# #processes = 2
# -----
#
#      MODE: AGGREGATE
#
# #bytes #rep.s t_min[μsec]      t_max      t_avg Mb/sec
#
#   0      50      ..      ..      ..      ..
#   1      50
#   2      50
#   4      50
#   8      50
#  16      50
#  32      50
#  64      50
# 128      50
# 256      50
# 512      50
#1024      50
#2048      50
#4096      50
#8192      50
#16384     50
#32768     50
#65536     50
#131072     50
#262144     50
#524288     32
#1048576     16
#2097152      8
#4194304      4
#8388608      2
#16777216      1
#
# -----
# Benchmarking P_Write_Indv
# #processes = 2
# -----
#
#      MODE: NON-AGGREGATE
#
# #bytes #rep.s t_min[μsec]      t_max      t_avg Mb/sec
#
#   0      10      ..      ..      ..      ..
#   1      10
#   2      10
#   4      10
#   8      10
#  16      10
#  32      10
#  64      10
# 128      10
# 256      10
# 512      10
#1024      10
#2048      10
#4096      10
#8192      10
#16384      10
#32768      10
#65536      10

```



```

131072      10
262144      10
524288      10
1048576     10
2097152      8
4194304      4
8388608      2
16777216     1

```

```
# All processes entering MPI_Finalize
```

6.6.4. Sample 4 - IMB-EXT.exe

The example below shows the results for the Window benchmark received after running IMB-EXT.exe on a Microsoft Windows* cluster using two processes. The performance diagnostics for Unidir_Get, Unidir_Put, Bidir_Get, Bidir_Put, and Accumulate are omitted.

```

<..> -n 2 IMB-EXT.exe

#-----
#   Intel ® MPI Benchmark Suite V3.2.2, MPI-2 part
#-----
# Date           : Fri Sep 05 12:26:52 2008
# Machine        : Intel64 Family 6 Model 15 Stepping 6, GenuineIntel
# System         : Windows Server 2008
# Release        : .0.6001
# Version        : Service Pack 1
# MPI Version    : 2.0
# MPI Thread Environment: MPI_THREAD_SINGLE

# New default behavior from Version 3.2 on:
# the number of iterations per message size is cut down
# dynamically when a certain run time (per message size sample)
# is expected to be exceeded. Time limit is defined by variable
# SECS_PER_SAMPLE (=> IMB_settings.h)
# or through the flag => -time

# Calling sequence was:
# \\master-node\MPI_Share_Area\IMB_3.1\src\IMB-EXT.exe
# Minimum message length in bytes: 0
# Maximum message length in bytes: 4194304
#
# MPI_Datatype           : MPI_BYTE
# MPI_Datatype for reductions : MPI_FLOAT
# MPI_Op                 : MPI_SUM
#
#
# List of Benchmarks to run:
# Window
# Unidir_Get
# Unidir_Put
# Bidir_Get
# Bidir_Put
# Accumulate

#-----
# Benchmarking Window
# #processes = 2

```

```
#-----
#bytes #repetitions t_min[μsec] t_max[μsec] t_avg[μsec]
      0           100      ..      ..      ..
      4           100
      8           100
     16           100
     32           100
     64           100
    128           100
    256           100
    512           100
   1024           100
   2048           100
   4096           100
   8192           100
  16384           100
  32768           100
  65536           100
 131072           100
 262144           100
 524288           80
1048576           40
2097152           20
4194304           10
...
# All processes entering MPI_Finalize
```

The above example listing shows the results of running `IMB-EXT.exe` on a Microsoft Windows* cluster using two processes.

The listing only shows the result for the `Window` benchmark. The performance diagnostics for `Unidir_Get`, `Unidir_Put`, `Bidir_Get`, `Bidir_Put`, and `Accumulate` are omitted.