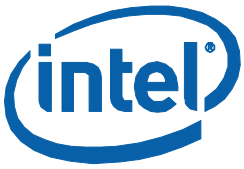


Intel® MPI Benchmarks

User Guide and Methodology Description

Contents

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.

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

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.

1.2.2. 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.

1.2.3. 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.

1.2.4. 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.

1.2.5. Changes in Intel® MPI Benchmarks 3.2.4

This release includes the following updates as compared to the Intel® MPI Benchmarks 3.2.3:

- Changes of document layout.

1.2.6. Changes in Intel® MPI Benchmarks 3.2.3

This release includes the following updates as compared to the Intel® MPI Benchmarks 3.2.2:

- Option `-msglog` to control the message length. Use this option to control the maximum and the second largest minimum of the message transfer sizes. The minimum message transfer size is always 0.
- Thread safety support in the MPI initialization phase. Use `MPI_Init()` by default because it is supported for all MPI implementations. You can choose `MPI_Init_thread()` by defining the appropriate macro.
- Option `-thread_level` to specify the desired thread level support for `MPI_Init_thread`.
- Support for the Microsoft* Visual Studio* 2010 project folder.

1.2.7. Changes in Intel® MPI Benchmarks 3.2.2

This release includes the following updates as compared to the Intel® MPI Benchmarks 3.2.1:

- Support for large buffers greater than 2 GB for some MPI collective benchmarks (`Allgather`, `Alltoall`, `Scatter`, `Gather`) to support large core counts.
- New benchmarks: `PingPongSpecificSource` and `PingPingSpecificSource`. The exact destination rank is used for these tests instead of `MPI_ANY_SOURCE` as in the `PingPong` and `PingPing` benchmarks. These are not executed by default. Use the `-include` option to enable the new benchmarks. For example,

```
$ mpirun  n 2 IMB_MPI -include PingPongSpecificSource \
PingPingSpecificSource
```

- New options `-include/-exclude` for better control over the benchmarks list. Use these options to include or exclude benchmarks from the default execution list.

1.2.8. Changes in Intel® MPI Benchmarks 3.2.1

This release includes the following updates as compared to the Intel® MPI Benchmarks 3.2:

- Fix of the memory corruption issue when the command-line option `-msglen` is used with the Intel® MPI Benchmarks executable files.
- Fix in the accumulated benchmark related to using the `CHECK` conditional compilation macro.

- Fix for the integer overflow in dynamic calculations on the number of iterations.
- Recipes for building IA-32 executable files within Microsoft* Visual Studio* 2005 and Microsoft* Visual Studio* 2008 project folders associated with the Intel® MPI Benchmarks.

1.2.9. Changes in Intel® MPI Benchmarks 3.2

Intel® MPI Benchmarks 3.2 has the following changes as compared to the previous version:

- The default settings are different.
- Microsoft* Visual Studio* project folders are added and can be used on the Microsoft* Windows* platforms.
- Makefiles for the Microsoft* Windows* `nmake` utility provided with the Intel® MPI Benchmarks 3.1 are removed.

Run Time Control by Default

The improved run time control that is associated with the `-time` flag. This is the default value for the Intel® MPI Benchmarks executable files (with a maximum run time per sample set to `10` seconds by the `SECS_PER_SAMPLE` parameter in the include file `IMB_settings.h`).

Makefiles

The `nmake` files for Windows* OS were removed and replaced by Microsoft* Visual Studio* solutions.

The Linux* OS Makefiles received new targets:

- Target `MPI1` (default) for building `IMB-MPI1`
- Target `EXT` for building `IMB-EXT`
- Target `IO` for building `IMB-IO`
- Target `all` for building all three of the above

Microsoft* Visual Studio* Project Folders

Intel® MPI Benchmarks 3.2 contains Microsoft* Visual Studio* solutions based on an installation of the Intel® MPI Library. A dedicated folder is created for the Microsoft* Windows* OS without duplicating source files. The solutions refer to the source files that are located at their standard location within the Intel® MPI Benchmarks directory structure.

As such solutions are highly version-dependent, see the information in the corresponding `ReadMe.txt` files that unpack with the folder. You are recommended to learn about the Microsoft* Visual Studio* philosophy and the run time environment of your Windows cluster.

1.2.10. Changes in Intel® MPI Benchmarks 3.1

This release includes the following updates as compared to the Intel® MPI Benchmarks 3.0:

- New control flags
- Better control of the overall repetition counts, run time, and memory exploitation
- A facility to avoid cache re-usage of message buffers as far as possible
- A fix of IMB-IO semantics
- New benchmarks
    - `Gather`
    - `Gatherv`
    - `Scatter`
    - `Scatterv`
- New command-line flags for better control
    - `-off_cache`

Use this flag when measuring performance on high speed interconnects or, in particular, across the shared memory within a node. Traditional Intel® MPI Benchmarks results included a very beneficial cache re-usage of message buffers which led to idealistic results. The flag -off_cache allows avoiding cache effects and lets the Intel® MPI Benchmarks use message buffers which are very likely not resident in cache.

- `-iter, -time`

  Use these flags for enhanced control of the overall run time, which is crucial for large clusters, where collectives tend to run extremely long in the traditional Intel® MPI Benchmarks settings.

CAUTION

In the Intel® MPI Benchmarks, the -time flag has been implemented as default.

- `-mem`

  Use this flag to determine an a priori maximum (per process) memory usage of the Intel® MPI Benchmarks for the overall message buffers.

Miscellaneous Changes

In the `Exchange` benchmark, the two buffers sent by `MPI_Isend` are separate. The command line is repeated in the output. Memory management is completely encapsulated in the functions `IMB_v_alloc/IMB_v_free`.

1.2.11. Changes in Intel® MPI Benchmarks 3.0

This release includes the following updates as compared to the Intel® MPI Benchmarks 2.3:

- A call to the `MPI_Init_thread` function to determine the MPI threading environment. The MPI threading environment is reported each time an Intel® MPI Benchmark application is executed.
- A call to the function `MPI_Get_version` to report the version of the Intel MPI library implementation that the three benchmark applications are linking to.
- New `Alltoallv` benchmark.
- New command-line flag `-h[elp]` to display the calling sequence for each benchmark application.
- Removal of the outdated `Makefile` templates. There are three complete `makefiles` called `Makefile`, `make_ict`, and `make_mpich`. The `make_ict` option uses the Intel® Composer XE compilers. This option is available for both Intel and non-Intel microprocessors but it may result in additional optimizations for Intel microprocessors.
- Better command-line argument checking, clean message and break on most invalid arguments.

1.3. About this Document

The objectives of the Intel® MPI Benchmarks User Guide are:

- Provide a concise set of benchmarks targeted at measuring the most important MPI functions.
- Set forth a precise benchmark methodology.
- Report bare timings rather than provide interpretation of the measured results. Show throughput values if and only if these values are well-defined.

1.3.1. Intended Audience

This guide is intended for users who want to measure performance of MPI implementations.

1.3.2. Notational Conventions

The following conventions are used in this document:

Style	Description
`This type style`	Commands, arguments, options, file names
`THIS_TYPE_STYLE`	Environment variables
<this type style>	Placeholders for actual values
`[ items ]`	Optional items
`{ item | item }`	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
`make clean`	Remove legacy binary object files and executable files
`make MPI1`	Build the executable file for the IMB-MPI1 component.
`make EXT`	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%\intel64\lib`
`%I_MPI_ROOT%\intel64\bin`
	`%I_MPI_ROOT%\ia32\include`
	`%I_MPI_ROOT%\ia32\lib`
	`%I_MPI_ROOT%\ia32\bin`

NOTE

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.

2. 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* 2013, go to `IMB-EXT_VS_2013`.
3. 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`
4. From the **Solution Platforms** drop-down list, choose the required architecture (**x64** or **Win32**).
5. From the **Solution Configurations** drop-down list, choose **Release**.
6. Highlight the project folder in the **Solution Explorer**.
7. Go to **Project > Properties** to open **Configuration Properties** dialog box. Make sure you have something like the following settings:

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	• 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	• x64: `%I_MPI_ROOT%\intel64\include` • IA-32: `%I_MPI_ROOT%\ia32\include`	
Warning Level	`to Level 1 (/W1)`	
C/C++ > Preprocessor		
Preprocessor Definition	• IMB-EXT: `WIN_IMB, _CRT_SECURE_NO_DEPRECATE, EXT` • IMB-IO: `WIN_IMB, _CRT_SECURE_NO_DEPRECATE, MPIIO` • IMB-MPI1: `WIN_IMB, _CRT_SECURE_NO_DEPRECATE, MPI1` • IMB-NBC: `WIN_IMB, _CRT_SECURE_NO_DEPRECATE, NBC` • IMB-RMA: `WIN_IMB, _CRT_SECURE_NO_DEPRECATE, RMA`	
Linker > Input		
Additional Dependencies	• x64: `%I_MPI_ROOT%\intel64\lib\impi.lib` • IA-32: `%I_MPI_ROOT%\ia32\lib\impi.lib`	

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 -np <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, ..., 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 (default) – the benchmarks run in a single process group.
- Multiple – the benchmarks run in several process groups.

To run the benchmarks in the multiple mode, add the multi- prefix to the benchmark name.

In the multiple mode, the number of groups may differ depending on the benchmark. For example, if PingPong is running on N≥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)
PingPing	Multi-PingPing
PingPingSpecificSource (excluded by default)	Multi-PingPingSpecificSource (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`
`PingPing`	`Multi-PingPong`	`Allgatherv` `Multi-Allgatherv`
`PingPingSpecificSource`	`Multi-PingPing`	`Alltoall` `Multi-Alltoall`
	`Multi-Sendrecv`	`Alltoallv` `Multi-Alltoallv`
	`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:

`throughput = X/2^20 * 10^6/time = X/1.048576/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:

`throughput = nmsg*X/2^20 * 10^6/time = nmsg*X/1.048576/time`,

where

- `time` is measured in `μsec`.
- `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`
- `PingPing`
- `PingPingSpecificSources`

See sections below for definitions of these benchmarks.

3.2.1. PingPong, PingPongSpecificSource

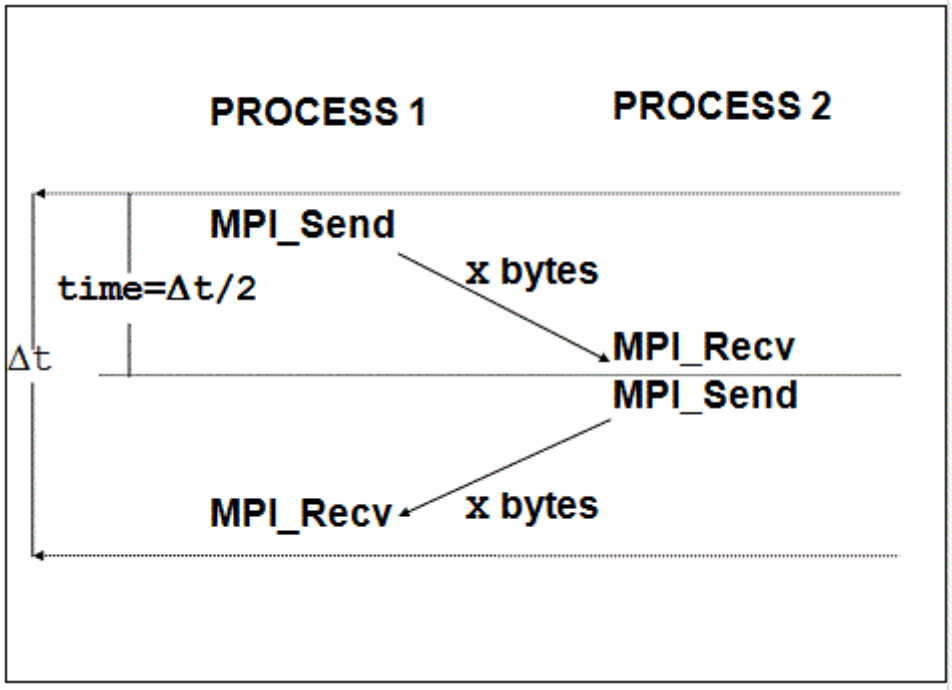
Use `PingPong` and `PingPongSpecificSource` for measuring startup and throughput of a single message sent between two processes. `PingPong` uses the `MPI_ANY_SOURCE` value for destination rank, while `PingPongSpecificSource` uses an explicit value.

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=Δt/2` (in `µsec`) as indicated in the figure below.
Reported throughput	`X/(1.048576*time)`

PingPong Pattern



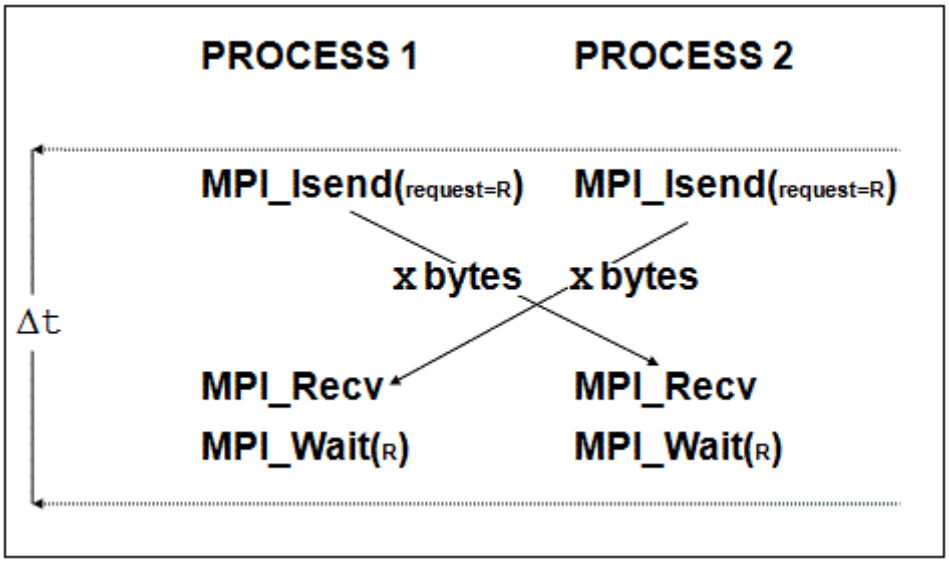
3.2.2. PingPing, PingPingSpecificSource

`PingPing` and `PingPingSpecificSource` 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, `PingPing` uses the `MPI_ANY_SOURCE` value, while `PingPingSpecificSource` uses 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	`MPI_Isend/MPI_Wait, MPI_Recv`
MPI data type	`MPI_BYTE`
Reported timings	`time=Δt` (in `µsec`)
Reported throughput	`X/(1.048576*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.

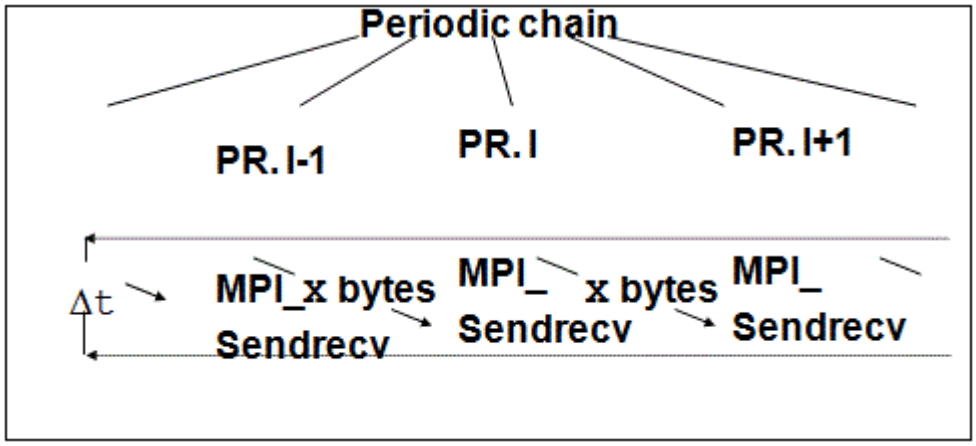
In the case of two processes, `Sendrecv` is equivalent to the `PingPing` benchmark of `IMB1.x`. For two processes, it reports the bidirectional bandwidth of the system, as obtained by the optimized `MPI_Sendrecv` function.

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/(1.048576*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.

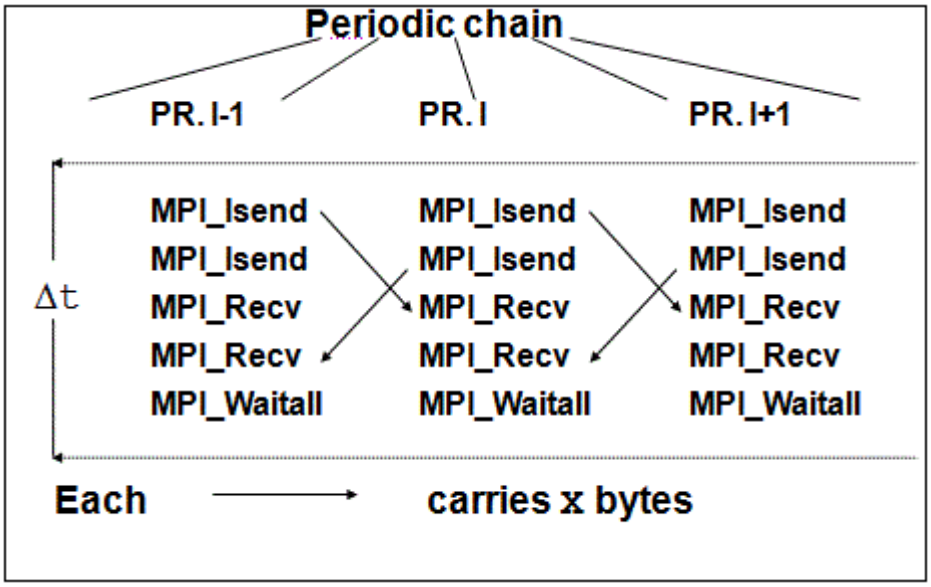
The turnover count is four messages per sample (two in, two out) for each process.

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	`time=Δt` (in `µsec`)
Reported throughput	`4X/(1.048576*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	(MAX_WIN_SIZE *MPI_Isend)/(MAX_WIN_SIZE *MPI_Irecv)/Waitall
MPI data type	`MPI_BYTE` (origin and target)
Reported message rate	`MAX_WIN_SIZE*N/MAX(process_timings)`, where `N` is the number of communicating pairs
Reported throughput	`MBps (msg_rate *size_in_bytes/ 1048576.0)`

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	(MAX_WIN_SIZE *MPI_Isend)/(MAX_WIN_SIZE *MPI_Irecv)/Waitall

MPI data type	`MPI_BYTE` (origin and target)
Reported message rate	`2*MAX_WIN_SIZE*N/MAX(process_timings)`, where `N` is the number of communicating pairs
Reported throughput	`MBps (msg_rate *size_in_bytes/ 1048576.0)`

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/sizeof(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	`MPI_Reduce`
MPI data type	`MPI_FLOAT`
MPI operation	`MPI_SUM`
Root	`i%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/sizeof(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 = ⌊L/np⌋`
- `s = L mod np`

In this case, the process with rank `i` gets:

- `r+1` items when `i<s`
- `r` items when `i≥s`

Property	Description
Measured pattern	`MPI_Reduce_scatter`
MPI data type	`MPI_FLOAT`
MPI operation	`MPI_SUM`
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/sizeof(float)` float items. The MPI data type is `MPI_FLOAT`. The MPI operation is `MPI_SUM`.

Property	Description
Measured pattern	`MPI_Allreduce`
MPI data type	`MPI_FLOAT`
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*np` bytes, where `np` 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*np` bytes, where `np` 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*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_Scatter`
MPI data type	`MPI_BYTE`
Root	`i%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*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_Scatterv`
MPI data type	`MPI_BYTE`
Root	`i%num_procs` in iteration `i`
Reported timings	Bare time
Reported throughput	None

3.4.8. Gather

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

Property	**Description**
Measured pattern	`MPI_Gather`
MPI data type	`MPI_BYTE`
Root	`i%num_procs` in iteration `i`
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*np` bytes, where `np` is the number of processes. The root of the operation is changed round-robin.

Property	**Description**
Measured pattern	`MPI_Gatherv`
MPI data type	`MPI_BYTE`
Root	`i%num_procs` in iteration `i`
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	`MPI_Alltoall`
MPI data type	`MPI_BYTE`
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	`MPI_Bcast`
MPI data type	`MPI_BYTE`
Reported timings	Bare time
Reported throughput	None

3.4.12. Barrier

The benchmark for the `MPI_Barrier` function.

Property	Description
Measured pattern	`MPI_Barrier`
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` `Multi-P_Write_indv`	Supported	`P_IWrite_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 `PingPong` and `PingPing`.
`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 `MPI_COMM_SELF`.
`[ACTION]`	A placeholder for `Read` or `Write` component of the benchmark name.
`I`	Non-blocking flavor. For example, `S_IWrite_indv` is the nonblocking flavor of the `S_IWrite_indv` 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.

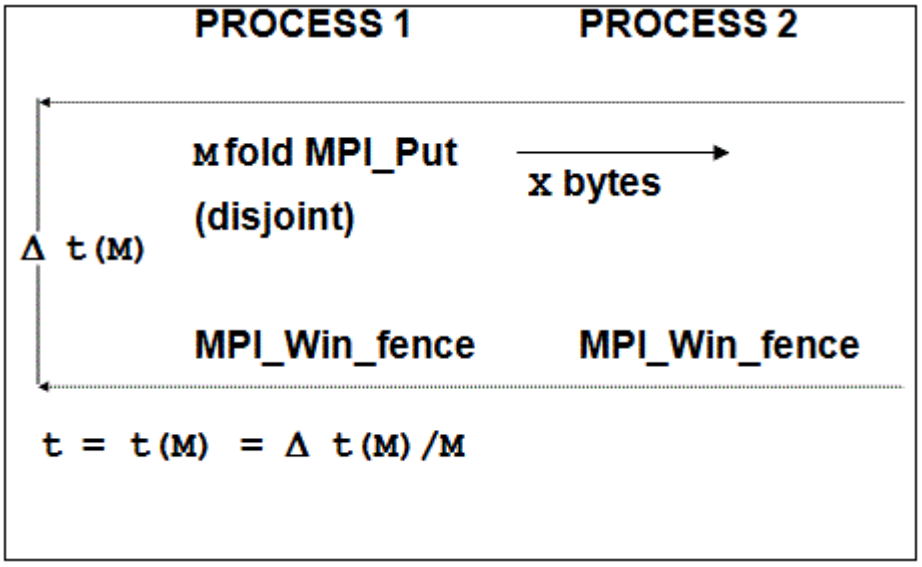
4.4.1. Unidir_Put

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	`MPI_Put`
MPI data type	`MPI_BYTE` (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



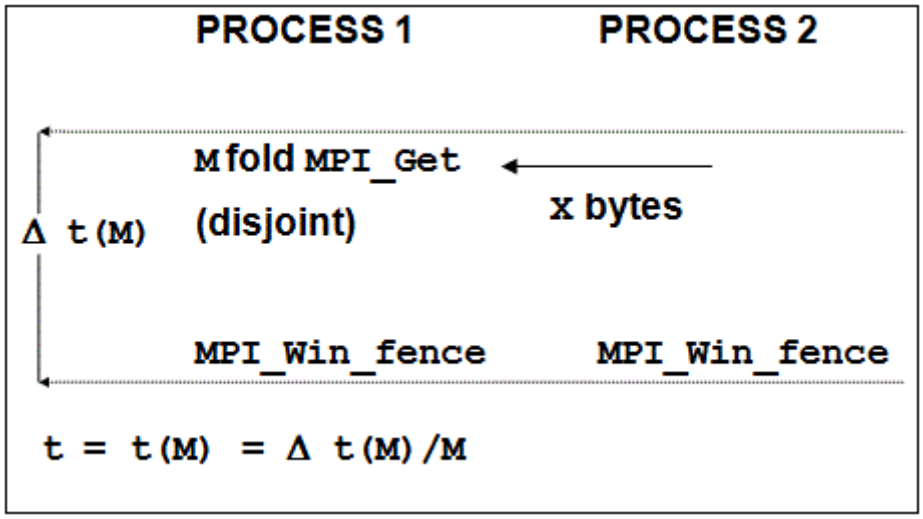
4.4.2. Unidir_Get

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	`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_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	`MPI_Put`
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_sample`).

	For details, see Actual Benchmarking.
Reported throughput	`X/t`, aggregate and non-aggregate

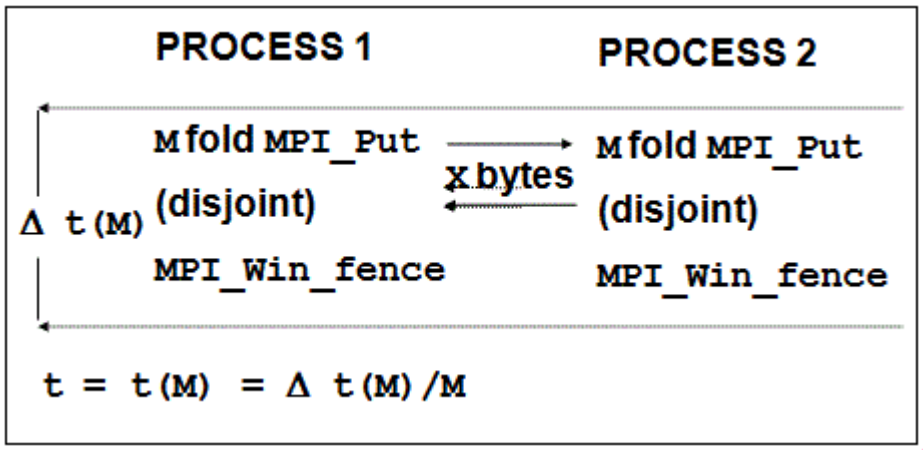
4.4.4. Bidir_Get

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_sample`). For details, see Actual Benchmarking.
Reported throughput	`X/t`, aggregate and non-aggregate

Bidir_Get Pattern



4.4.5. Accumulate

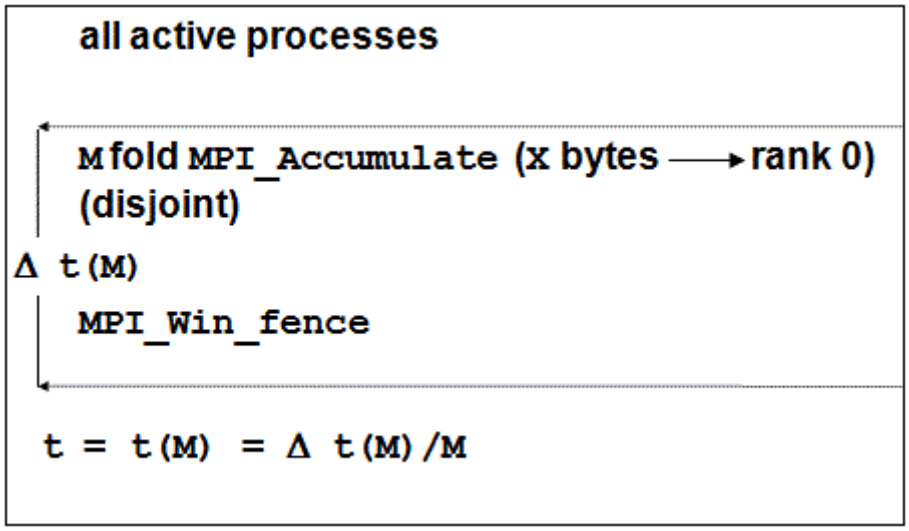
This is the benchmark for the `MPI_Accumulate` function. It reduces a vector of length `L = x/sizeof(float)` of float items. The MPI data type is `MPI_FLOAT`. The MPI operation is `MPI_SUM`. See the basic definitions and a schematic view of the pattern below.

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_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.

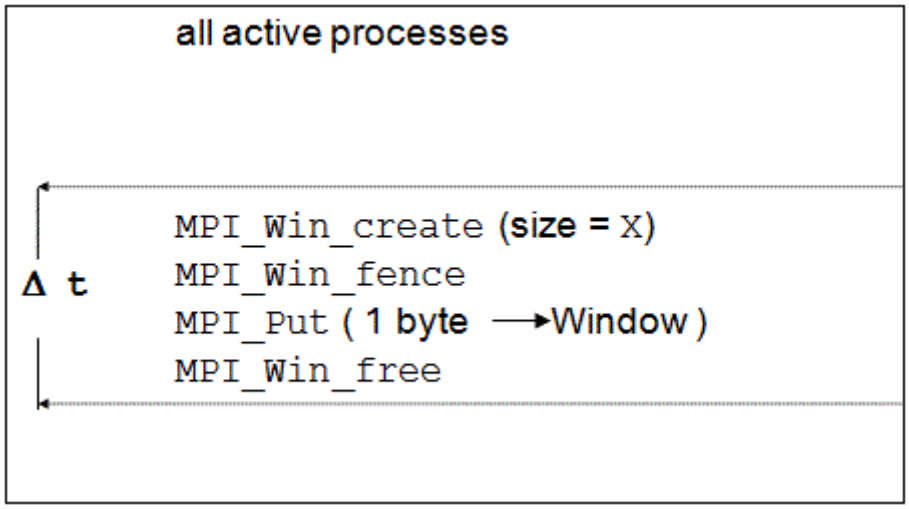
The `MPI_Win_fence` function is called to properly initialize an access epoch. This is a correction as compared to earlier releases of the Intel® MPI Benchmarks.

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=Δt(M)` (in `µsec`) as indicated in the figure below.
Reported throughput	None

Window Pattern



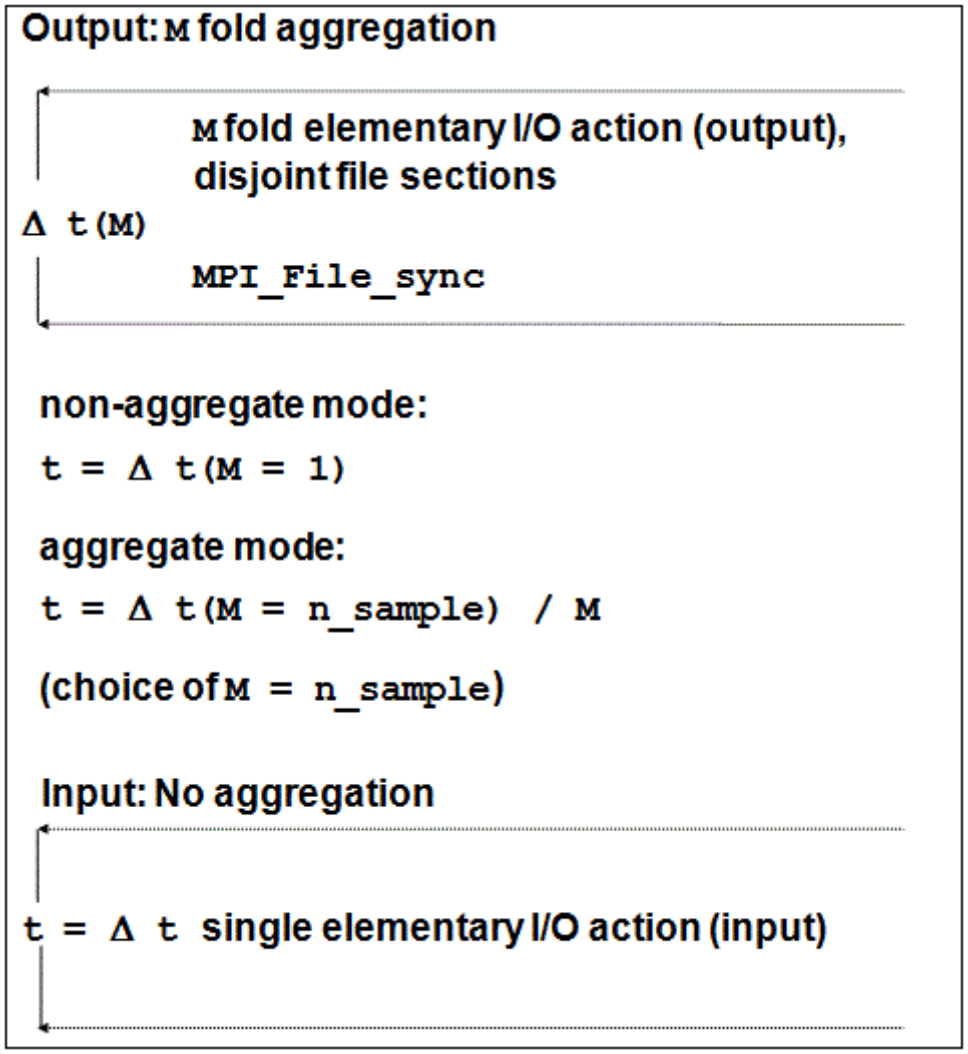
4.5. IMB-IO Blocking Benchmarks

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



4.5.1. S_[ACTION]_indv

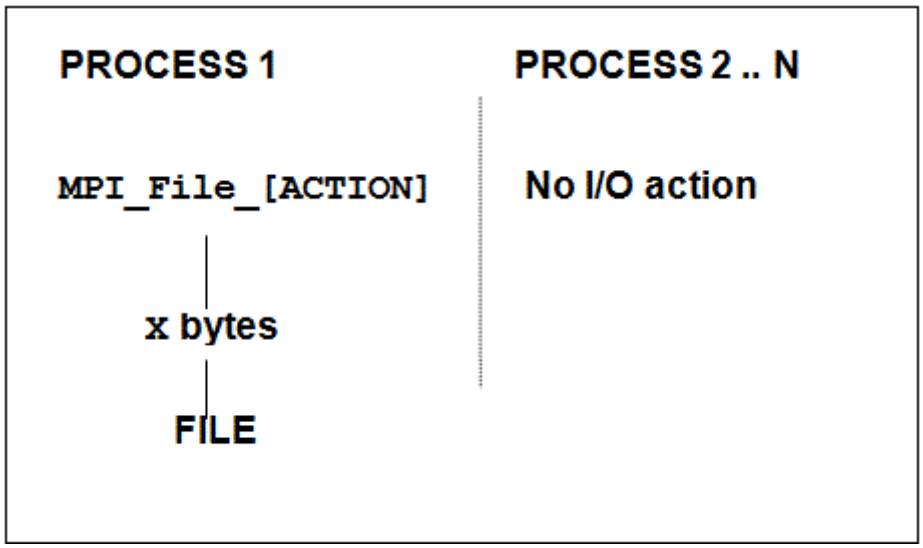
File I/O performed by a single process. This pattern mimics the typical case when a particular master process performs all of the I/O. See the basic definitions and a schematic view of the pattern below.

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_iwrite/MPI_File_iread`
`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



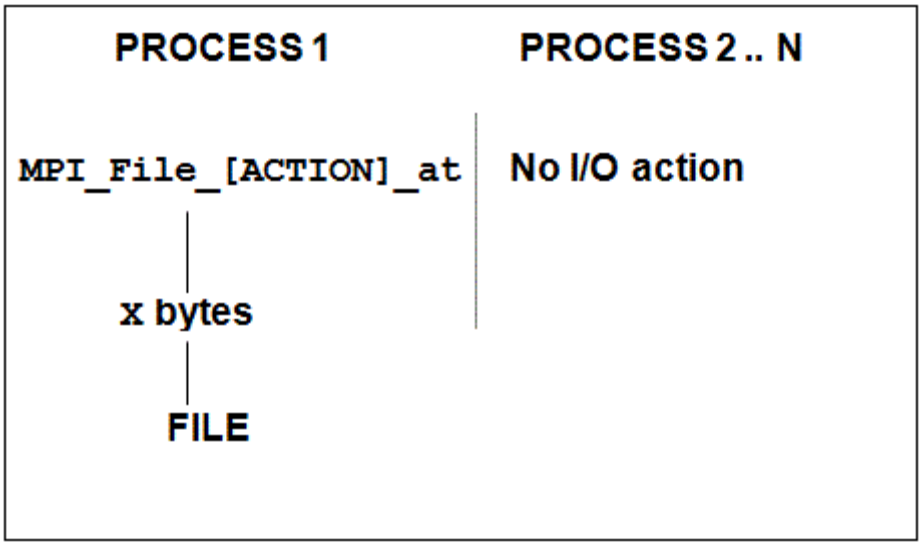
4.5.2. S_[ACTION]_expl

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_iwrite_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



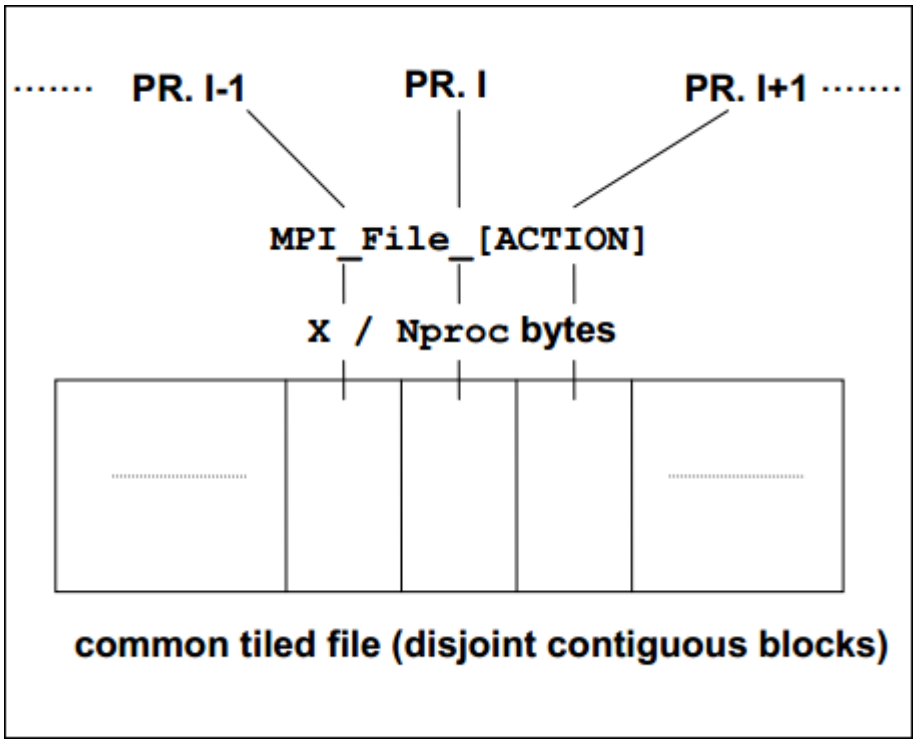
4.5.3. P_[ACTION]_indv

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, `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_iwrite/MPI_File_iread`
`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



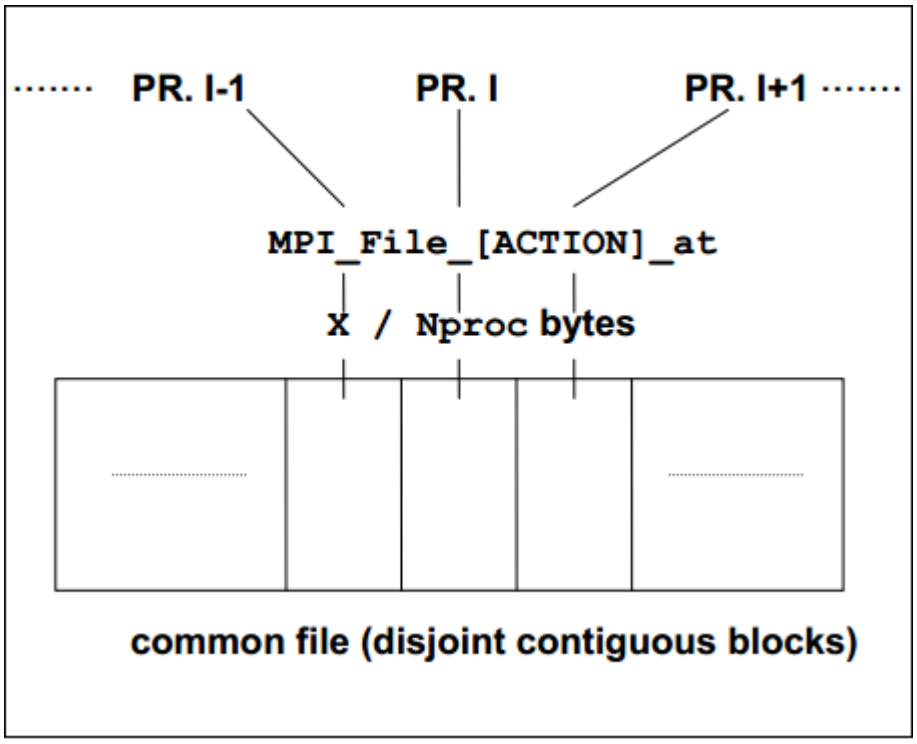
4.5.4. P_ACTION_expl

`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, `Nproc` 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_iwrite_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



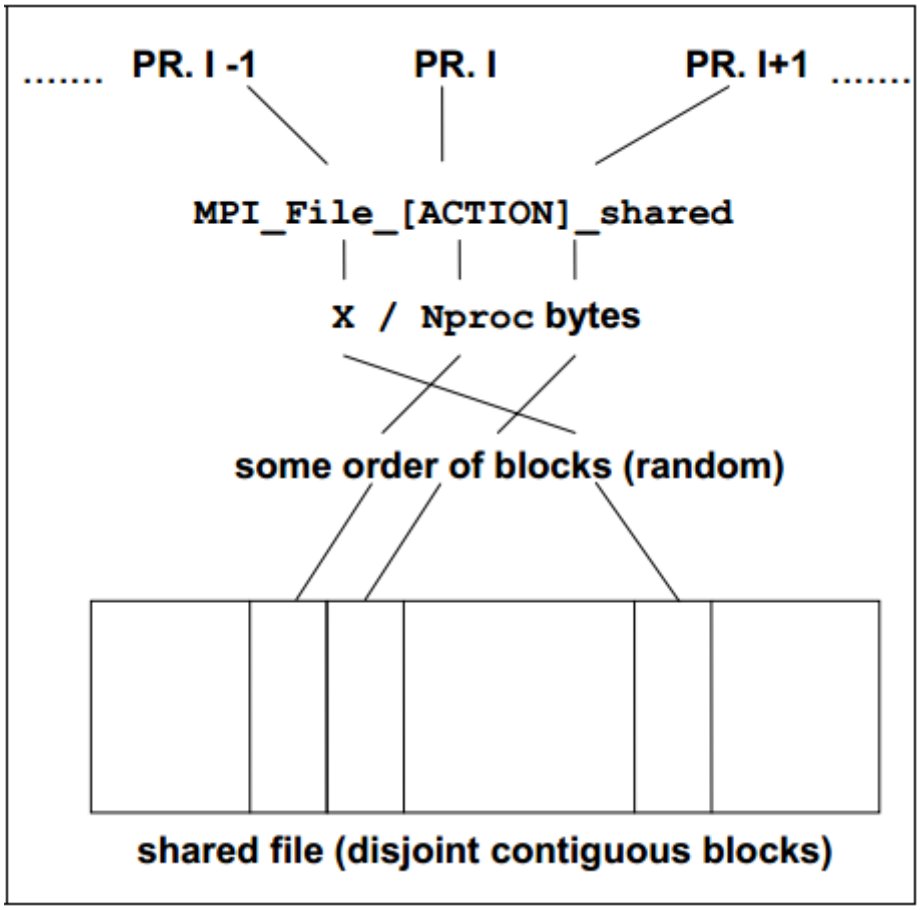
4.5.5. P_[ACTION]_shared

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, `Nproc` 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_iwrite_at/MPI_File_iread_at`
`etype`	`MPI_BYTE`
File type	`MPI_BYTE`
MPI data type	`MPI_BYTE`
Reported timings	`t` (in `usec`) 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]_shared Pattern



4.5.6. P_[ACTION]_priv

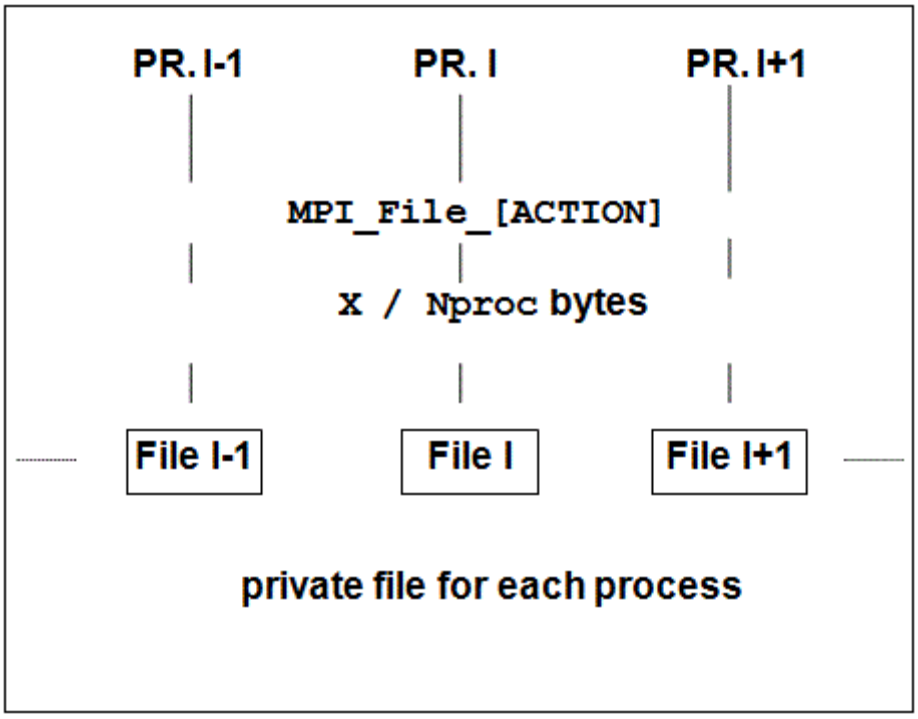
This pattern tests the case when all participating processes perform concurrent I/O to different private files. This benchmark is particularly useful for the systems that allow completely independent I/O operations from different processes. The benchmark pattern is expected to show parallel scaling and 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_iwrite/MPI_File_iread`
`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/Δt`, 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`

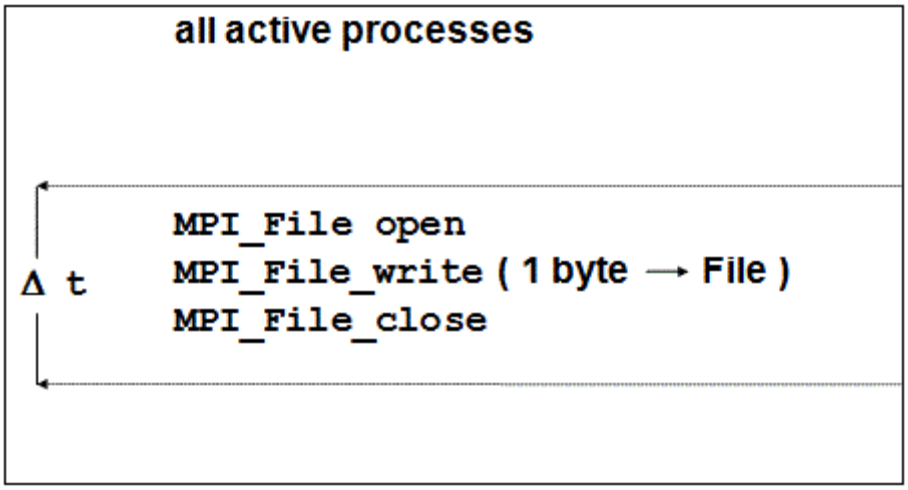
4.5.10. Open_Close

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	`MPI_File_open/MPI_File_close`
`etype`	`MPI_BYTE`
File type	`MPI_BYTE`
Reported timings	`t=Δt` (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:

```
overlap = 100.*max(0,min(1, (t_pure+t_CPU-t_ovrl) / min(t_pure, t_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	`MPI_Iallgather/IMB_cpu_exploit/MPI_Wait`
MPI data type	`MPI_BYTE`
Reported timings	• `t_ovrl` • `t_pure` • `t_CPU` • `overlap=100.*max(0,min(1, (t_pure+t_CPU-t_ovrl) / min(t_pure, t_CPU))` For details, see Measuring Communication and Computation Overlap.
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	`MPI_Iallgather/MPI_Wait`
MPI data type	`MPI_BYTE`
Reported timings	Bare time
Reported throughput	None

5.1.5. Iallgatherv

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

Property	Description
Measured pattern	`MPI_Iallgatherv/IMB_cpu_exploit/MPI_Wait`
MPI data type	`MPI_BYTE`
Reported timings	• `t_ovrl` • `t_pure` • `t_CPU` • `overlap=100.*max(0,min(1, (t_pure+t_CPU-t_ovrl) /`

	`min(t_pure, t_CPU))` For details, see Measuring Communication and Computation Overlap.
Reported throughput	None

5.1.6. Iallgatherv_pure

The benchmark for the `MPI_Iallgatherv` 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	`MPI_Iallgatherv/MPI_Wait`
MPI data type	`MPI_BYTE`
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	`MPI_Iallreduce/IMB_cpu_exploit/MPI_Wait`
MPI data type	`MPI_FLOAT`
MPI operation	`MPI_SUM`
Reported timings	• `t_ovrl` • `t_pure` • `t_CPU` • `overlap=100.*max(0,min(1, (t_pure+t_CPU-t_ovrl) / min(t_pure, t_CPU))` For details, see Measuring Communication and Computation Overlap.
Reported throughput	None

5.1.8. Iallreduce_pure

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

Property	Description
Measured pattern	`MPI_Iallreduce/MPI_Wait`
MPI data type	`MPI_FLOAT`

MPI operation	`MPI_SUM`
Reported timings	Bare time
Reported throughput	None

5.1.9. Ialltoall

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

Property	**Description**
Measured pattern	`MPI_Ialltoall/IMB_cpu_exploit/MPI_Wait`
MPI data type	`MPI_BYTE`
Reported timings	• `t_ovrl` • `t_pure` • `t_CPU` • `overlap=100.*max(0,min(1, (t_pure+t_CPU-t_ovrl) / min(t_pure, t_CPU))` For details, see Measuring Communication and Computation Overlap.
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	`MPI_Ialltoall/MPI_Wait`
MPI data type	`MPI_BYTE`
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	`MPI_Ialltoallv/IMB_cpu_exploit/MPI_Wait`
MPI data type	`MPI_BYTE`
Reported timings	• `t_ovrl` • `t_pure`

	• `t_CPU` • `overlap=100.*max(0,min(1, (t_pure+t_CPU-t_ovrl) / min(t_pure, t_CPU))` For details, see Measuring Communication and Computation Overlap.
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	`MPI_Ialltoallv/MPI_Wait`
MPI data type	`MPI_BYTE`
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	`MPI_Ibarrier/IMB_cpu_exploit/MPI_Wait`
Reported timings	• `t_ovrl` • `t_pure` • `t_CPU` • `overlap=100.*max(0,min(1, (t_pure+t_CPU-t_ovrl) / min(t_pure, t_CPU))` For details, see Measuring Communication and Computation Overlap.
Reported throughput	None

5.1.14. Ibarrier_pure

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

Property	**Description**
Measured pattern	`MPI_Ibarrier/MPI_Wait`
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	`MPI_Ibcast/IMB_cpu_exploit/MPI_Wait`
MPI data type	`MPI_BYTE`
Reported timings	• `t_ovrl` • `t_pure` • `t_CPU` • `overlap=100.*max(0,min(1, (t_pure+t_CPU-t_ovrl) / min(t_pure, t_CPU))` For details, see Measuring Communication and Computation Overlap.
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	`MPI_Ibcast/MPI_Wait`
MPI data type	`MPI_BYTE`
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	`MPI_Igather/IMB_cpu_exploit/MPI_Wait`
MPI data type	`MPI_BYTE`
Root	`i%num_procs` in iteration `i`
Reported timings	• `t_ovrl` • `t_pure` • `t_CPU` • `overlap=100.*max(0,min(1, (t_pure+t_CPU-t_ovrl) / min(t_pure, t_CPU))` For details, see Measuring Communication and Computation Overlap.

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*np` bytes (`X` from each process). All processes receive `X` bytes. The root of the operation is changed round-robin.

Property	**Description**
Measured pattern	`MPI_Igather/MPI_Wait`
MPI data type	`MPI_BYTE`
Root	`i%num_procs` in iteration `i`
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	`MPI_Igatherv/IMB_cpu_exploit/MPI_Wait`
MPI data type	`MPI_BYTE`
Root	`i%num_procs` in iteration `i`
Reported timings	• `t_ovrl` • `t_pure` • `t_CPU` • `overlap=100.*max(0,min(1, (t_pure+t_CPU-t_ovrl) / min(t_pure, t_CPU))` For details, see Measuring Communication and Computation Overlap.
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	`MPI_Igatherv/MPI_Wait`
MPI data type	`MPI_BYTE`

Root	`i%num_procs` in iteration `i`
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	`MPI_Ireduce/IMB_cpu_exploit/MPI_Wait`
MPI data type	`MPI_FLOAT`
MPI operation	`MPI_SUM`
Root	`i%num_procs` in iteration `i`
Reported timings	• `t_ovrl` • `t_pure` • `t_CPU` • `overlap=100.*max(0,min(1, (t_pure+t_CPU-t_ovrl) / min(t_pure, t_CPU))` 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/sizeof(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	`MPI_Ireduce/MPI_Wait`
MPI data type	`MPI_FLOAT`
MPI operation	`MPI_SUM`
Root	`i%num_procs` in iteration `i`
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/sizeof(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 = ⌊L/np⌋`
- `s = L mod np`

In this case, the process with rank `i` gets:

- `r+1` items when `i<s`
- `r` items when `i≥s`

Property	Description
Measured pattern	`MPI_Ireduce_scatter/IMB_cpu_exploit/MPI_Wait`
MPI data type	`MPI_FLOAT`
MPI operation	`MPI_SUM`
Reported timings	• `t_ovrl` • `t_pure` • `t_CPU` • `overlap=100.*max(0,min(1, (t_pure+t_CPU-t_ovrl) / min(t_pure, t_CPU))` For details, see Measuring Communication and Computation Overlap.
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/sizeof(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 = ⌊L/np⌋`
- `s = L mod np`

In this case, the process with rank `i` gets:

- `r+1` items when `i<s`
- `r` items when `i≥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	`i%num_procs` in iteration `i`
Reported timings	• `t_ovrl` • `t_pure` • `t_CPU` • `overlap=100.*max(0,min(1, (t_pure+t_CPU-t_ovrl) / min(t_pure, t_CPU))` For details, see Measuring Communication and Computation Overlap.
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	`i%num_procs` in iteration `i`
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_Iscatterv/IMB_cpu_exploit/MPI_Wait`

MPI data type	`MPI_BYTE`
Root	`i%num_procs` in iteration `i`
Reported timings	• `t_ovrl` • `t_pure` • `t_CPU` • `overlap=100.*max(0,min(1, (t_pure+t_CPU-t_ovrl) / min(t_pure, t_CPU))` For details, see Measuring Communication and Computation Overlap.
Reported throughput	None

5.1.28. Iscatterv_pure

The benchmark for the `MPI_Iscatterv` 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_Iscatterv/MPI_Wait`
MPI data type	`MPI_BYTE`
Root	`i%num_procs` in iteration `i`
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` 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:

`throughput = X/2^20 * 10^6/time = X/1.048576/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:

`throughput = (X/2^20 * 10^6/time) * N = X/1.048576/time * N`, 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	`MPI_Accumulate/MPI_Win_flush`
MPI data type	`MPI_FLOAT` (origin and target)
MPI operation	`MPI_SUM`
Reported timings	Bare time
Reported throughput	`MBps`

5.2.4. 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: `(target_rank + current_rank)%num_ranks`.

Property	**Description**
Measured pattern	`(N*MPI_Get)/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	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: `(target_rank + current_rank)%num_ranks`.

Property	**Description**
Measured pattern	`(N*MPI_Put)/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	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	`MPI_Get/MPI_Win_flush`
MPI data type	`MPI_BYTE` (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*MPI_Get)/(2*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*MPI_Put)/(2*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*MPI_Get)/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`

5.2.14. 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	`MPI_Get/MPI_Win_flush_local`
MPI data type	`MPI_BYTE` (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*MPI_Put)/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*MPI_Get)/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*MPI_Put)/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	`MPI_Put/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	`MPI_Put/MPI_Win_flush`, while the target process performs computations before the `MPI_Barrier` call
MPI data type	`MPI_BYTE` (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	`MPI_Get/MPI_Win_flush`
MPI data type	`MPI_BYTE` (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`

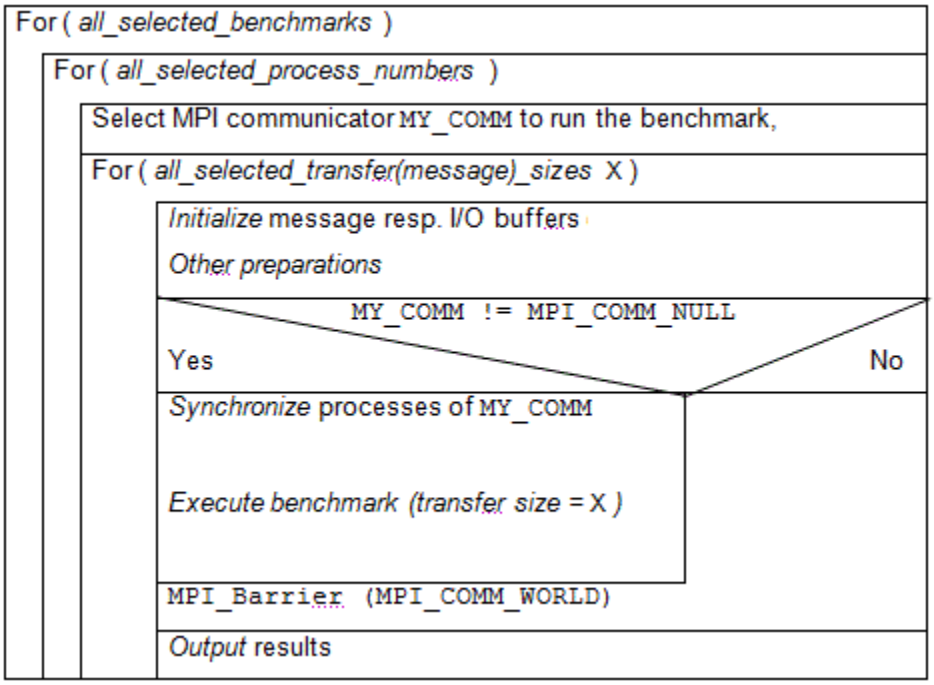
6. Benchmark Methodology

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>]
            [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 restrictions:

- 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 `PingPongSpecificSource` and `PingPingSpecificSource`, 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 PingPongSpecificsource
PingPingSpecificsource -exclude Alltoall Alltoallv
```

Run the `PingPong`, `PingPing`, `PingPongSpecificSource` and `PingPingSpecificSource` benchmarks with the transfer message sizes `0, 2^0, 2^1, 2^2, ..., 2^16`:

```
mpirun -np 4 IMB-MPI1 -msglog 16 PingPong PingPing
PingPongSpecificSource PingPingSpecificSource
```

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^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.

When the number of processes running the benchmark is more than half of the overall number `MPI_COMM_WORLD`, the multiple benchmark 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`.

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 `-time`) is expected to be reached. Using this policy ensures faster execution, but may lead to inaccuracy of the results.
`multiple_np`	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 `-time` option.
`auto`	Automatically chooses which policy to use: • applies `multiple_np` to collective operations where one of the processes acts as the root of the operation (for example, `MPI_Bcast`) • applies `dynamic` to all other types of operations
`off`	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 number the processes 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 of size P, with each process on one node communicating with its counterpart on the other, call:

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

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

Specifies the list of additional benchmarks to run. For example, to add `PingPongSpecificSource` and `PingPingSpecificSource` benchmarks, call:

```
mpirun -np 2 IMB-MPI1 -include PingPongSpecificSource PingPingSpecificSource
```

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.

6.3. 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 IMB_settings.h	Values in IMB_settings_io.h	Description
`USE_MPI_INIT_THREAD`	Not set	Not set	Set to initialize Intel® MPI Benchmarks by `MPI_Init_thread()` instead of `MPI_Init()`
`IMB_OPTIONAL`	Not set	Not set	Set to activate optional settings
`MINMSGLOG`	0	0	The second smallest data transfer size is `max(unit,` $2^{\texttt{MINMSGLOG}}$ (the smallest size is always 0), where `unit=sizeof(float)` for reductions, `unit=1` for all other cases. You can override this parameter value using the `-msglog` flag.
`MAXMSGLOG`	22	24	The largest message size used is $2^{\texttt{MAXMSGLOG}}$

			You can override this parameter value using the `-msglog` flag.
`ITER_POLICY`	imode_dynamic		The policy used for calculating the number of iterations. You can override this parameter value using the `-iter_policy` or `-iter` flag.
`MSGSPERSAMPLE`	1000	50	The maximum repetition count for all IMB-MPI1 benchmarks. You can override this parameter value using the `-iter` flag.
`MSGS_NONAGGR`	100	10	The maximum repetition count for non-aggregate benchmarks (relevant only for `IMB-EXT`). You can override this parameter value using the `-time` flag.
`OVERALL_VOL`	40 Mbytes	16*1048576	For all sizes smaller than `OVERALL_VOL`, the repetition count is reduced so that not more than `OVERALL_VOL` 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 X is `MSGSPERSAMPLE (X=0)`, `max(1,min(MSGSPERSAMPLE,OVERALL_VOL/X)) (X>0)` Note that `OVERALL_VOL` does *not* restrict the size of the maximum data transfer. $2^{MAXMSGLOG}$ `OVERALL_VOL`. You can override this parameter value using the `-mem` 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 `MPI_Barrier` for synchronization.
`TARGET_CPU_SECS`	0.01 seconds	0.1 seconds	CPU seconds (as float) to run concurrently with nonblocking benchmarks (irrelevant for IMB-MPI1)

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<=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, the constant `N_BARR` is used to regulate calls to:

```
MPI_Barrier(MPI_COMM_WORLD)
```

The `N_BARR` constant is defined in `IMB_settings.h` and `IMB_settings_io.h`, with the current value of 2.

See figure Control flow of IMB to ensure that all 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 `MSGS_NONAGGR`. 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[µsec], bandwidth = X / 1.048576 / 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.