Intel® MPI Benchmarks

User Guide and Methodology Description

Document Number: 320714-007EN

Revision: 3.2.3

World Wide Web: http://www.intel.com

Disclaimer and Legal Information

Document Number: 320714-007EN

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

Contents

Revision History

Document Number	Revision Number	Description	Revision Date
320714-001	2.3	Initial version	/10/2004
320714-002	3.0	The following topics were added: • Descriptions of environment amendments • The Alltoallv benchmark	/06/2006
320714-003	3.1	The following updates were added: • Description of Windows version • Four new benchmarks (Scatter(v), Gather(v)) • IMB-IO functional fix	/07/2007
320714-004	3.2	The following topics were added: • Run time control as default • Microsoft* Visual Studio* solution templates	/08/2008
320714-005	3.2.1	The following updates were added: • Fix of the memory corruption • Fix in accumulate benchmark related to using the CHECK conditional compilation macro • Fix for 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	/04/ 2010
320714-006	3.2.2	The following updates were added: • Support for large buffers greater than 2 GB for some MPI benchmark • New benchmarks PingPongSpecificSource and PingPingSpecificSource • New options -include/-exclude	/09/2010
320714-007	3.2.3	The following topics were updated and added: • Changes in the Intel® MPI Benchmarks 3.2.3 • Command-line Control • Parameters Controlling IMB • Microsoft* Visual Studio* 2010 project folder support	/08/2011

1 About this Document

This Guide presents the Intel® MPI Benchmarks (IMB) suite. Its objectives are:

- Provide a concise set of benchmarks targeted at measuring the most important MPI functions.
- Set forth a precise benchmark methodology.
- Do not impose much of an interpretation on the measured results: report bare timings instead. Show throughput values, if and only if these are well defined.

This document accompanies Intel® MPI Benchmarks 3.2.2. The code is written in ANSI C plus standard MPI (about 10,000 lines of code, 110 functions in 37 source files).

The Intel MPI Benchmarks package consists of three parts:

- IMB-MPI1
- Two MPI-2 functionality parts
- IMB-EXT (One-sided Communications benchmarks) and IMB-IO (I/O benchmarks)

You can build a separate executable file for each part. If you do not have the MPI-2 extensions available, you can install and use IMB-MPI1. Only standard MPI-1 functions are used. No dummy library is needed.

1.1 Intended Audience

This *Guide* provides advanced users with all details about the Intel® MPI benchmark and how to use it.

1.2 Using Doc Type Field

This *Guide* contains the following sections:

Table 1-1 Document Organization

Section	Description
Section 1 About this Document	Section 1 introduces this document
Section 2 What's new	Section 2 changes for the Intel® MPI Benchmarks compared to the previous versions of this product
Section 3 Installation and Quick Start	Section 3 explains how to install and start the product
Section 4 IMB MPI1	Section 4 gives information about benchmarks for testing MPI-1 functions
Section 5 MPI2	Section 5 provides detailed information about performance testing of MPI-2 functions and MPI Input/Output functionality

Section 6 Benchmark Methodology	Section 6 describes available options, different settings and testing methodology
Section 7 Output	Section 7 explains output examples
Section 8 Further Details	Section 8 contains memory requirements and details about results checking

1.3 Conventions and Symbols

The following conventions are used in this document.

Table 1-2 Conventions and Symbols used in this Document

This type style	Document or product names
This type style	Hyperlinks
`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 Related Information

The following related documents that might be useful to the user:

Intel® IMB Benchmarks Download Page

Product Web Site

Intel® MPI Library Support

Intel® Cluster Tools Products

Intel® Software Development Products

2 What's New

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

2.1 Changes in the Intel® MPI Benchmarks 3.2.3

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

- A new 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.
- The new option `-thread_level` to specify the desired thread level support for `MPI_Init_thread`.
- New support for Microsoft* Visual Studio* 2010 project folder.

2.2 Changes in the Intel® MPI Benchmarks 3.2.2

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

- New 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 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.

2.3 Changes in the Intel® MPI Benchmarks 3.2.1

This release includes the following updates 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 Benchmark 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.

2.4 Changes in the Intel® MPI Benchmarks 3.2

The Intel® MPI Benchmarks 3.2 has different default settings compared to the previous version, and has the addition of Microsoft* Visual Studio* project folders that can be used on the Microsoft* Windows* platforms. In turn, Makefiles for the Windows `nmake` utility provided with the Intel® MPI Benchmarks 3.1 are removed.

2.4.1 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`).

2.4.2 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

2.4.3 Microsoft* Visual Studio* Project Folders

The 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. We recommend familiarity with Microsoft* Visual Studio* philosophy and the run time environment of your Windows cluster at hand.

2.5 Changes in the Intel® MPI Benchmarks 3.1

This release includes the following updates 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:

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:

are there 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:

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

2.5.1 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.

2.6 Changes in the Intel® MPI Benchmarks 3.0

This release includes the following updates 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 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

perform additional optimizations for Intel microprocessors than it performs for non-Intel microprocessors.

- Better command-line argument checking, clean message and break on most invalid arguments.

3 Installation and Quick Start of Intel® MPI Benchmarks

To run IMB-MPI1, you need:

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

3.1 Installing and Running

After you unpack the installation file, the directory contains a file `ReadMe_first` and five subdirectories:

- `./doc` (ReadMe_IMB.txt; Users_Guide.pdf, this file)
- `./src` (program source- and Make-files)
- `./WINDOWS` (Visual Studio Solutions)
- `./license` (license agreements text)

  In the license agreement directory, you can see the following files:
  - The license.txt file specifies the source code license granted to you.
  - The use-of-trademark-license.txt specifies the license for using the name and/or trademark Intel® MPI Benchmarks.
- `./versions_news` (version history and news)

To get a quick start, see `./doc/ReadMe_IMB.txt`.

Use the following instructions if you are installing the Intel® MPI Benchmarks on Linux* OS:

To remove legacy binary object files and executable files, use the command:

```
make clean
```

To build the selected executable files, use the following command:

```
make MPI1 (or EXT or IO)
```

To build all three executables, use the following command below:

```
make all
```

NOTE: The above command assumes that the environment variable `CC` has been set appropriately before the `makefile` command invocation.

Use the enclosed solution files as basis on Microsoft Windows OS.

After installation, use your style of starting MPI programs, for example,

```
mpirun -np <P> IMB-MPI1 (IMB-EXT,IMB-IO)
```

to get the full suite of all benchmarks. For more selective running, see 6.1.2

4 IMB-MPI1

The Intel® MPI Benchmarks provides a set of elementary MPI1 benchmarks. You can run all of the supported benchmarks, or a subset specified in the command line using one executable file. The rules, such as time measurement, message lengths, and selection of communicators are command-line parameters.

You can run the benchmarks in standard and multiple modes. The default mode is the standard mode.

4.1 The Benchmarks

The current version of IMB-MPI1 contains the following benchmarks:

- `PingPong`
- `PingPongSpecificSource` (excluded by default)
- `PingPing`
- `PingPingSpecificSource` (excluded by default)
- `Sendrecv`
- `Exchange`
- `Bcast`
- `Allgather`
- `Allgatherv`
- `Scatter`
- `Scatterv`
- `Gather`
- `Gatherv`
- `Alltoall`
- `Alltoallv`
- `Reduce`
- `Reduce_scatter`
- `Allreduce`
- `Barrier`

In the multiple mode, the benchmarks behavior is changed. In this case the IMB-MPI1 benchmarks are run in more than one process group. For example, in the multiple mode, if `PingPong` is run on N≥4 processes, N/2 disjoint groups of two processes are formed and N/2 instances of `PingPong` are executed.

The following list shows the multiple versions of the benchmarks:

- Multi-PingPong
- Multi-PingPongSpecificSource (excluded by default)
- Multi-PingPing
- Multi-PingPingSpecificSource (excluded by default)
- Multi-Sendrecv
- Multi-Exchange
- Multi-Bcast
- Multi-Allgather
- Multi-Allgatherv
- Multi-Scatter
- Multi-Scatterv
- Multi-Gather
- Multi-Gatherv
- Multi-Alltoall
- Multi-Alltoallv
- Multi-Reduce
- Multi-Reduce_scatter
- Multi-Allreduce
- Multi-Barrier

4.2 IMB-MPI1 Benchmark Classification

The Intel® MPI Benchmarks introduces classes of benchmarks:

- Single Transfer
- Parallel Transfer
- Collective benchmarks

This classification refers to different ways of interpreting results.

The 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 defined in MBytes / sec = 2^{20} bytes / sec

scale (throughput = X / 2^{20} * 10^{6} / time = X / 1.048576 / time,

where time is in µsec and x is the length of a message in bytes).

The Parallel Transfer benchmarks involve more than two active processes into communication. 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.

The throughput calculations of the benchmarks take into account the multiplicity nmsg of messages outgoing from or incoming at a particular process. In the Sendrecv benchmark, a particular process sends and receives X bytes, the turnover is 2X bytes, nmsg=2. In the Exchange case, we have 4X bytes turnover, nmsg=4.

Throughput values are defined in MBytes/sec = 2^{20} bytes / sec

scale (throughput = nmsg*X/2^{20} * 10^6/time = nmsg*X / 1.048576 / time,

when time is in µsec and x is the length of a message in bytes).

The Collective benchmarks perform 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 the pure data movement functions and MPI_FLOAT for the reductions.

For all Collective benchmarks, only bare timings and no throughput data are displayed.

IMB-MPI1		
Single Transfer	**Parallel Transfer**	**Collective**
PingPong PingPongSpecificSource PingPing PingPingSpecificSource	Sendrecv Exchange Multi-PingPong Multi-PingPing Multi-Sendrecv Multi-Exchange	Bcast Allgather Allgatherv Alltoall Alltoallv Scatter Scatterv Gather Gatherv Reduce Reduce_scatter Allreduce Barrier Multi-versions of these

Figure 4-1 Benchmarks classification

4.2.1 Single Transfer

4.2.1.1 PingPong and PingPongSpecificSource

PingPong and PingPongSpecificSource are the classical pattern used for measuring startup and throughput of a single message sent between two processes. The difference is that PingPong uses the value MPI_ANY_SOURCE for destination rank and PingPongSpecificSource uses the explicit value.

measured pattern	As symbolized between ◄····· ◄····· ; two active processes only (Q=2)
based on	MPI_Send, MPI_Recv
MPI_Datatype	MPI_BYTE
reported timings	time = Δt/2 (in µsec) as indicated in Figure 1
reported throughput	X/(1.048576*time)

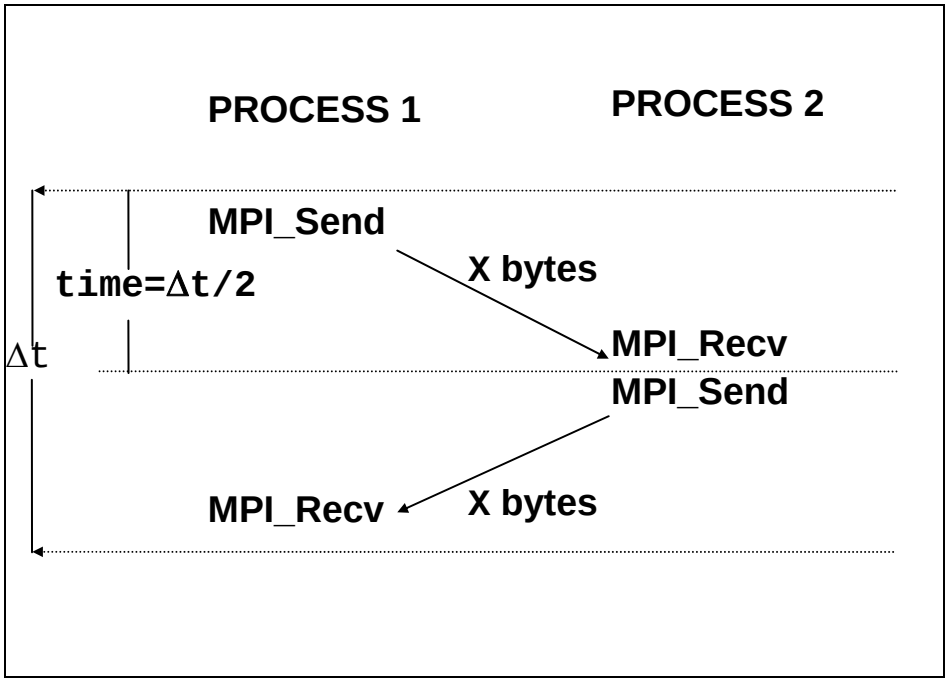

Figure 4-2: PingPong pattern

4.2.1.2 PingPing and PingPingSpecificSource

PingPong, and PingPingSpecificSource measure startup and throughput of single messages, with the crucial difference that messages are obstructed by oncoming messages. For this, two processes communicate (MPI_Isend/MPI_Recv/MPI_Wait) with each other, with the MPI_Isend's issued simultaneously. The difference is that PingPing uses the value MPI_ANY_SOURCE for destination rank and PingPingSpecificSource uses the explicit value.

measured pattern	As symbolized between ◄····· ◄····· ,two active processes only (Q=2),
based on	MPI_Isend/MPI_Wait, MPI_Recv
MPI_Datatype	MPI_BYTE
reported timings	time = Δt (in µsec)
reported throughput	X/(1.048576*time)

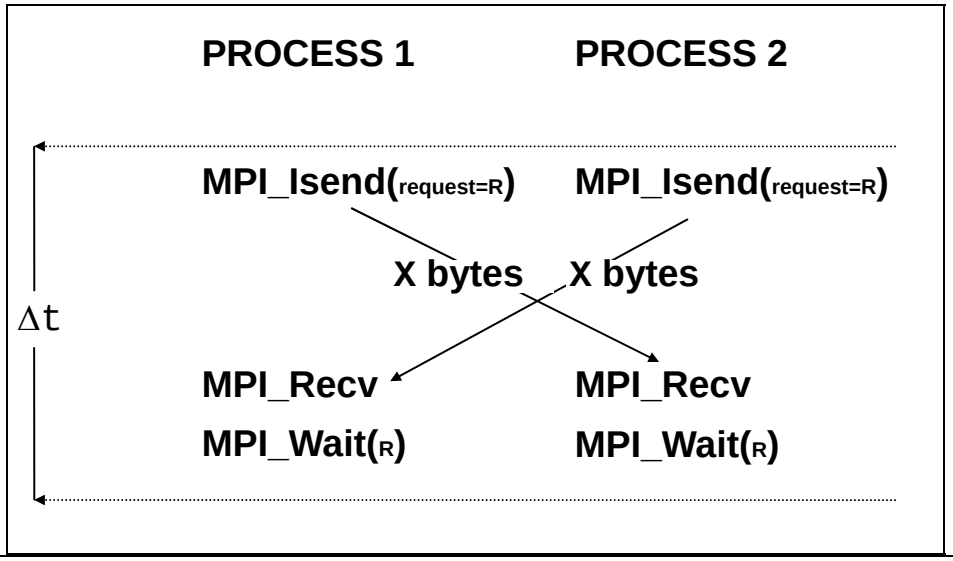

Figure 4-3: PingPing pattern

4.2.2 Parallel Transfer

4.2.2.1 Sendrecv

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

`Sendrecv` is equivalent with the `Cshift` benchmark. In case of two processes, `Sendrecv` is equivalent with the `PingPing` benchmark of IMB1.x. For two processes, it will report the bi-directional bandwidth of the system, as obtained by the (optimized) `MPI_Sendrecv` function.

measured pattern	As symbolized between ◄····· ◄·····
based on	`MPI_Sendrecv`
`MPI_Datatype`	`MPI_BYTE`
reported timings	`time = Δt (in µsec)` as indicated in Figure 3
reported throughput	`2X/(1.048576*time)`

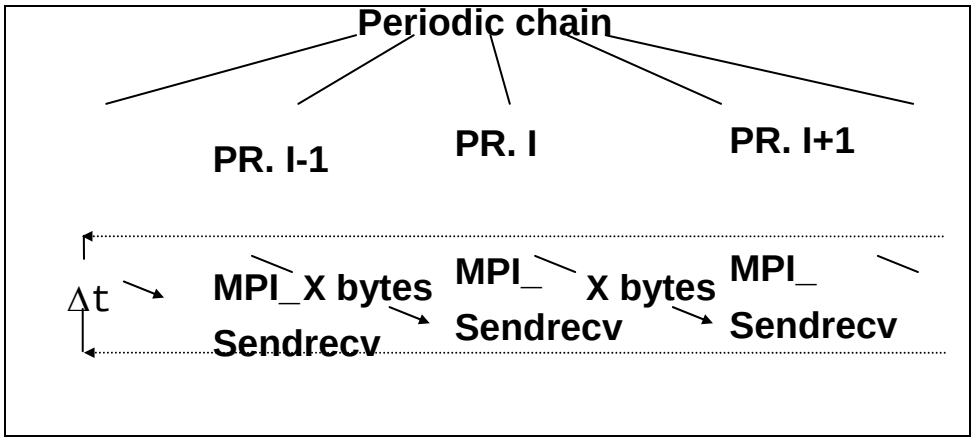

Figure 4-4: Sendrecv pattern

4.2.2.2 Exchange

`Exchange` is a communications pattern that often occurs in grid splitting algorithms (boundary exchanges). The group of processes looks as 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.

measured pattern	As symbolized between ⇐ ⇐
based on	`MPI_Isend/MPI_Waitall, MPI_Recv`
`MPI_Datatype`	`MPI_BYTE`
reported timings	`time = Δt (in µsec)`
reported throughput	`4X/(1.048576*time)`

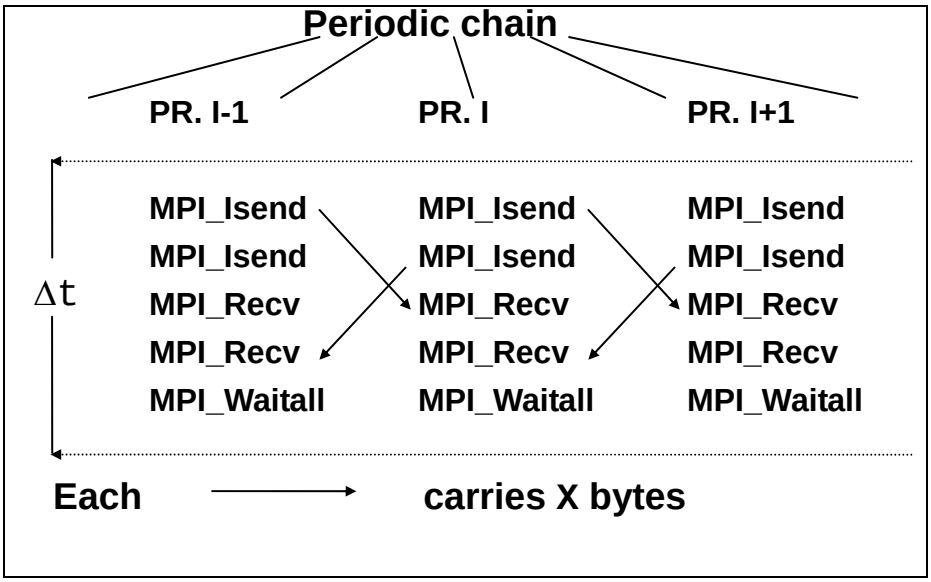

Figure 4-5: Exchange pattern

4.2.3 Collective Benchmarks

4.2.3.1 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.

measured pattern	MPI_Reduce
MPI_Datatype	MPI_FLOAT
MPI_Op	MPI_SUM
root	i%num_procs in iteration i
reported timings	bare time
reported throughput	none

4.2.3.2 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. Exactly, when

np = #processes, L = r*np+s (s = L mod np),

then process with rank i gets r+1 items when i<s, and r items when i≥s.

measured pattern	MPI_Reduce_scatter
MPI_Datatype	MPI_FLOAT
MPI_Op	MPI_SUM
reported timings	bare time
reported throughput	none

4.2.3.3 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.

measured pattern	MPI_Allreduce
MPI_Datatype	MPI_FLOAT
MPI_Op	MPI_SUM
reported timings	bare time
reported throughput	none

4.2.3.4 Allgather

The benchmark for the MPI_Allgather function. Every process inputs X bytes and receives the gathered X*(#processes) bytes.

measured pattern	MPI_Allgather
MPI_Datatype	MPI_BYTE
reported timings	bare time
reported throughput	none

4.2.3.5 Allgatherv

Functionally, this benchmark is the same as Allgather. However, with the MPI_Allgatherv function it shows whether MPI produces overhead due to the more complicated situation as compared to MPI_Allgather.

measured pattern	MPI_Allgatherv
MPI_Datatype	MPI_BYTE
reported timings	bare time
reported throughput	none

4.2.3.6 Scatter

The benchmark for the MPI_Scatter function. The root process inputs X*(#processes) bytes (X for each process); all processes receive X bytes. The root of the operation is changed round robin.

measured pattern	MPI_Scatter
MPI_Datatype	MPI_BYTE
Root	i%num_procs in iteration i
reported timings	bare time
reported throughput	none

4.2.3.7 Scatterv

The benchmark for the MPI_Scatterv function. The root process inputs X*(#processes) bytes (X for each process); all processes receive X bytes. The root of the operation is changed round robin.

measured pattern	MPI_Scatterv
MPI_Datatype	MPI_BYTE
root	i%num_procs in iteration i
reported timings	bare time
reported throughput	none

4.2.3.8 Gather

The benchmark for the MPI_Gather function. All processes input X bytes. The root process receives X*(#processes) bytes (X from each process). The root of the operation is changed round robin.

measured pattern	MPI_Gather
MPI_Datatype	MPI_BYTE
root	i%num_procs in iteration i
reported timings	bare time

reported throughput	none

4.2.3.9 Gatherv

The benchmark for the `MPI_Gatherv` function. All processes input `X` bytes. The root process receives `X*`(#processes) bytes (`X` from each process). The root of the operation is changed round robin.

measured pattern	`MPI_Gather`
`MPI_Datatype`	`MPI_BYTE`
`root`	`i%num_procs in iteration i`
reported timings	bare time
reported throughput	none

4.2.3.10 Alltoall

The benchmark for the `MPI_Alltoall` function. Every process inputs `X*`(#processes) bytes (`X` for each process) and receives `X*`(#processes) bytes (X from each process).

measured pattern	`MPI_Alltoall`
`MPI_Datatype`	`MPI_BYTE`
reported timings	bare time
reported throughput	none

4.2.3.11 Bcast

The benchmark for `MPI_Bcast`. A root process broadcasts `X` bytes to all. The root of the operation is changed round robin.

measured pattern	`MPI_Bcast`
`MPI_Datatype`	`MPI_BYTE`
`root`	`i%num_procs in iteration i`
reported timings	bare time
reported throughput	none

4.2.3.12 Barrier

measured pattern	`MPI_Barrier`
reported timings	bare time
reported throughput	none

5 MPI-2 Part of Intel® MPI Benchmarks

This section introduces a list of all IMB-MPI2 benchmarks.

Table 5-1: IMB-MPI-2 benchmarks

Benchmark	Aggregate Mode	Non-blocking Mode
IMB-EXT		
Window		
Unidir_Put		
Unidir_Get		
Bidir_Get		
Bidir_Put		
Accumulate		
Multi- versions of the above		
Benchmark	**Aggregate Mode**	**Nonblocking Mode**
IMB-IO		
Open_Close		
S_Write_indv		S_IWrite_indv
S_Read_indv		S_IRead_indv
S_Write_expl		S_IWrite_expl
S_Read_expl		S_IRead_expl
P_Write_indv		P_IWrite_indv
P_Read_indv		P_IRead_indv
P_Write_expl		P_IWrite_expl
P_Read_expl		P_IRead_expl
P_Write_shared		P_IWrite_shared
P_Read_shared		P_IRead_shared
P_Write_priv		P_IWrite_priv
P_Read_priv		P_IRead_priv

C_Write_indv		C_IWrite_indv
C_Read_indv		C_IRead_indv
C_Write_expl		C_IWrite_expl
C_Read_expl		C_IRead_expl
C_Write_shared		C_IWrite_shared
C_Read_shared		C_IRead_shared
Multi-versions of the above	(×)	Multi-versions of the above

The following list is the naming conventions for the benchmarks:

- Unidir/Bidir stand for unidirectional/bidirectional one-sided communications. These are the one-sided equivalents of PingPong and PingPing.
- If the Multi- prefix is defined, the benchmark runs in the multi mode.
- Prefixes S_/P_/C_ mean Single/Parallel/Collective. The classification is the same as in the MPI1 case. In the I/O case, the Single Transfer is defined as an operation between one MPI process and one individual window or a file. In the Parallel Transfer, more than one process participates in the overall pattern, whereas Collective means the same term as in MPI standard.
- The postfixes mean:
  - 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

5.1 IMB-MPI2 Benchmark Classification

The Intel® MPI Benchmarks has three classes of benchmarks:

- Single Transfer
- Parallel Transfer
- Collective.

Two special benchmarks that measure accompanying overheads of one-sided communications (MPI_Win_create / MPI_Win_free) and overhead of I/O operations, (MPI_File_open / MPI_File_close) are not assigned to any class.

Table 5-2 IMB-MPI2 benchmarks classification

Single Transfer	Parallel Transfer	Collective	Other
Unidir_Get	Multi-Unidir_Get	Accumulate	Window
Unidir_Put	Multi-Unidir_Put	Multi-Accumulate	(also Multi)
Bidir_Get	Multi-Bidir_Get		
Bidir_Put	Multi-Bidir_Put		
S_[I]Write_indv	P_[I]Write_indv	C_[I]Write_indv	Open_close

S_[I]Read_indv	P_[I]Read_indv	C_[I]Read_indv	(also Multi)
S_[I]Write_expl	P_[I]Write_expl	C_[I]Write_expl	
S_[I]Read_expl	P_[I]Read_expl	C_[I]Read_expl	
	P_[I]Write_shared	C_[I]Write_shared	
	P_[I]Read_shared	C_[I]Read_shared	
	P_[I]Write_priv	Multi- versions	
	P_[I]Read_priv		

5.1.1 Single Transfer Benchmarks

The Single Transfer benchmarks focus on a single data transferred between *one* source and *one* target. In IMB-MPI2, the source of the data transfer can be an MPI process or, in case of Read benchmarks, an MPI file. The target can be an MPI process or an MPI file.

- The single transfer IMB-EXT benchmarks only run with two active processes
- The single transfer IMB-IO benchmarks only run with one active process.

5.1.2 Parallel Transfer Benchmarks

In the Parallel Transfer benchmarks, the activity at a certain process is in concurrency with other processes. The benchmark timings are produced under a global load. The number of participating processes is arbitrary.

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

5.1.3 Collective Benchmarks

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

5.1.3.1 Benchmark Modes

Certain benchmarks have different modes to run:

- Blocking / non-blocking mode (only IMB-IO)
- Aggregate / Non Aggregate mode
- Aggregate / Non Aggregate mode is not available for non-blocking benchmarks in IMB-IO

5.1.3.2 Assured Completion of Transfers

The key point is where to assure completion of a data transfers either after each single one (non aggregate) or after a bunch of multiple transfers (aggregate). Assured completion means the following:

- `MPI_Win_fence` (`IMB-EXT`)
- A triplet
- `MPI_File_sync` / `MPI_Barrier` (`file_communicator`) / `MPI_File_sync` (`IMB-IO` Write). Following the MPI standard, the minimum sequence of operations after which all processes of the file communicator have a consistent view after a write. This fixes the non sufficient definition in the Intel® MPI Benchmarks 3.0.

5.1.3.2.1 Mode Definition

The basic patterns of these benchmarks are:

- `M` is some repetition count
- 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 the multiple transfers (if `M`>1) are to/from disjoint sections of the window or file. This is to circumvent misleading optimizations when using the same locations for multiple transfers.

The Intel® MPI Benchmarks runs the corresponding benchmarks with two settings:

- `M` = 1 (non aggregate mode)
- `M` = n_sample (aggregate mode), with n_sample

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

Figure 5-1: Aggregation of M transfers (IMB-EXT and blocking Write benchmarks)

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

- Accumulative strategy. Several successive transfers (up to M in Figure 6) are accumulated (for example by a caching mechanism), without an immediate completion. At certain stages (system and runtime dependent), at best only in the assured completion part, the accumulated transfers are completed as a whole. This approach may save time of expensive synchronizations. The expectation is that this strategy would provide for (much) better results in the aggregate case as compared to the non aggregate one.
- Non-accumulative strategy. Every Single Transfer is automatically completed before the return from the corresponding function. The time of expensive synchronizations is taken into account. The expectation is that this strategy would produce equal results for aggregate and non aggregate case.

5.1.4 Definition of the IMB-EXT Benchmarks

This section describes the benchmarks in detail. The benchmarks run with varying transfer sizes X (in bytes). The timings are averaged over multiple samples. Below you can see the description of one single sample with a fixed transfer size X.

NOTE: The Unidir (Bidir) benchmarks are exact equivalents of the message passing PingPong. Their interpretation and output is analogous to their message passing equivalents.

5.1.4.1 Unidir_Put

This is the benchmark for the MPI_Put function. Below see the basic definitions and a schematic view of the pattern.

Table 5-3 : Unidir_Put definition

measured pattern	as symbolized between ;2 active processes only
based on	MPI_Put
MPI_Datatype	MPI_BYTE (origin and target)
reported timings	t=t(M) (in µsec) non aggregate (M=1) and aggregate (cf. 0; M=n_sample)
reported throughput	X/t, aggregate and non aggregate

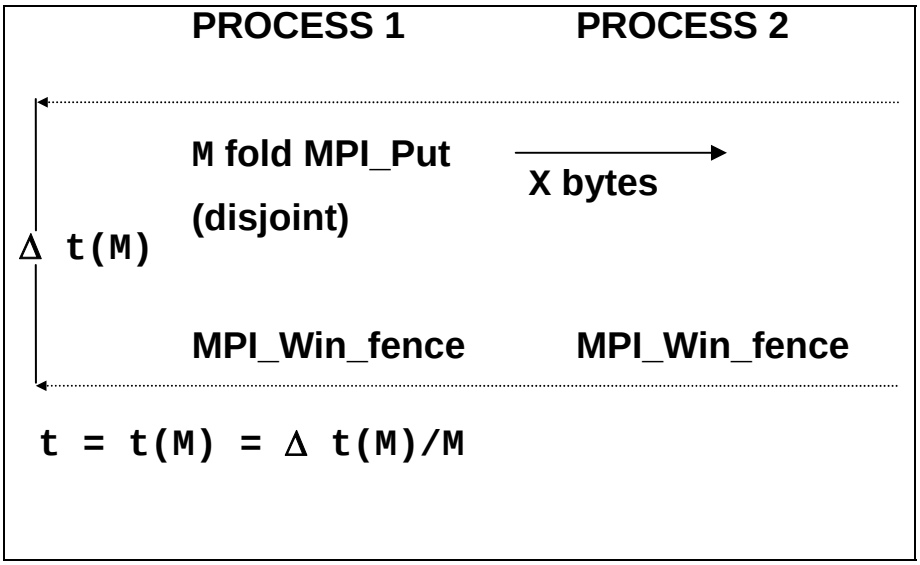

Figure 5-1: Unidir_Put pattern

5.1.4.2 Unidir_Get

This is the benchmark for the MPI_Get function. Below see the basic definitions and a schematic view of the pattern.

Table 5-4: Unidir_Get definition

measured pattern	as symbolized between ;2 active

	processes only
based on	MPI_Get
MPI_Datatype	MPI_BYTE (origin and target)
reported timings	t=t(M) (in µsec), non aggregate (M=1) and aggregate (cf. 0; M=n_sample)
reported throughput	X/t, aggregate and non aggregate

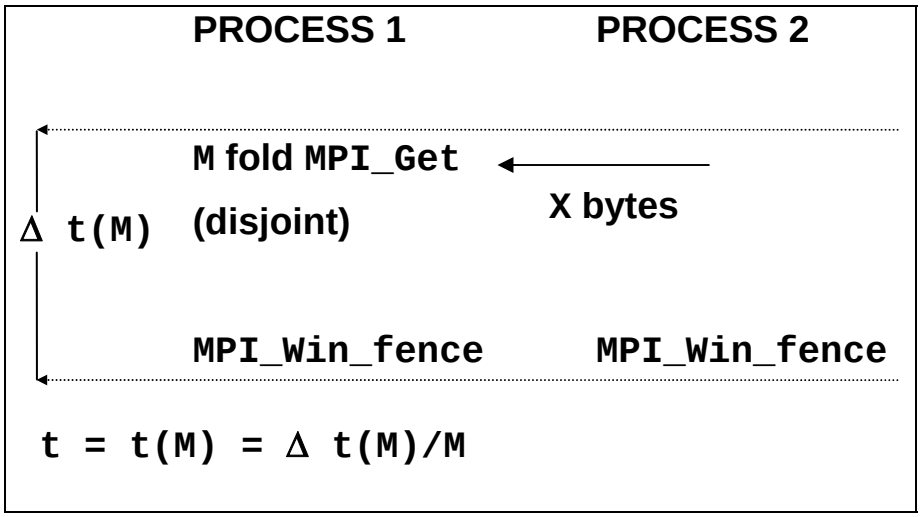

Figure 5-2: Unidir_Get pattern

5.1.4.3 Bidir_Put

This is the benchmark for the MPI_Put function with bi-directional transfers. Below see the basic definitions and a schematic view of the pattern.

measured pattern	as symbolized between ;2 active processes only
based on	MPI_Put
MPI_Datatype	MPI_BYTE (origin and target)
reported timings	t=t(M) (in µsec), non aggregate (M=1), and aggregate (cf. 0; M=n_sample, see 6.2.8)
reported throughput	X/t, aggregate and non aggregate

Table 5-5: Bidir_Put definition

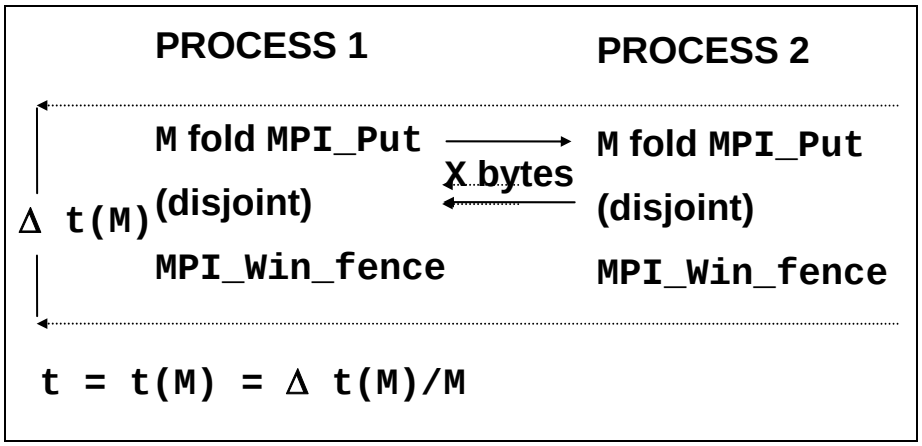

Figure 5-3: Bidir_Put pattern

5.1.4.4 Bidir_Get

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

measured pattern	as symbolized between ;2 active processes only
based on	`MPI_Get`
`MPI_Datatype`	`MPI_BYTE` (origin and target)
reported timings	`t=t(M)` (in µsec), non aggregate (`M=1`) and aggregate (`cf. 0; M=n_sample`)
reported throughput	X/t, aggregate and non aggregate

Table 5-6: Bidir_Get definition

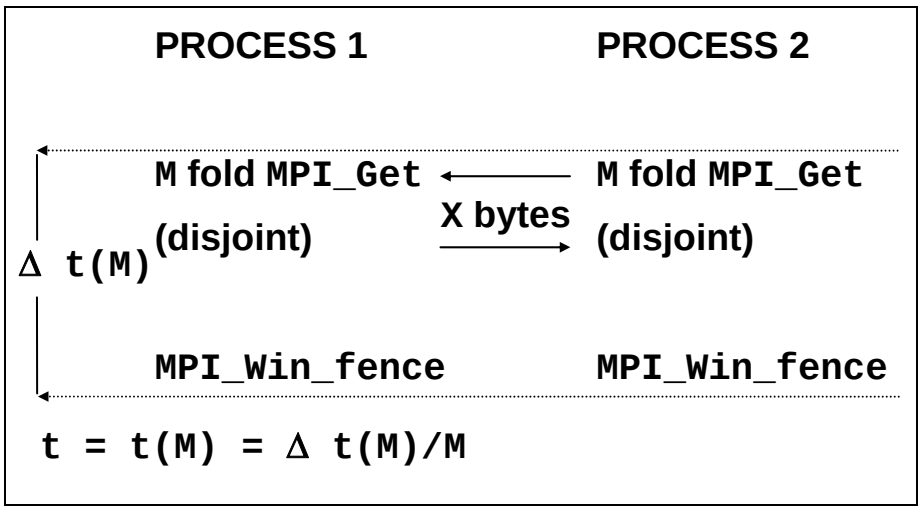

Figure 5-4: Bidir_Get pattern

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, and the MPI operation is MPI_SUM. Below see the basic definitions and a schematic view of the pattern.

measured pattern	as symbolized between ◄····· ◄·····
based on	MPI_Accumulate
MPI_Datatype	MPI_FLOAT
MPI_Op	MPI_SUM
Root	0
reported timings	t=t(M) (in µsec), non aggregate (M=1) and aggregate (cf. 0; M=n_sample)
reported throughput	none

Table 5-7: Accumulate definition

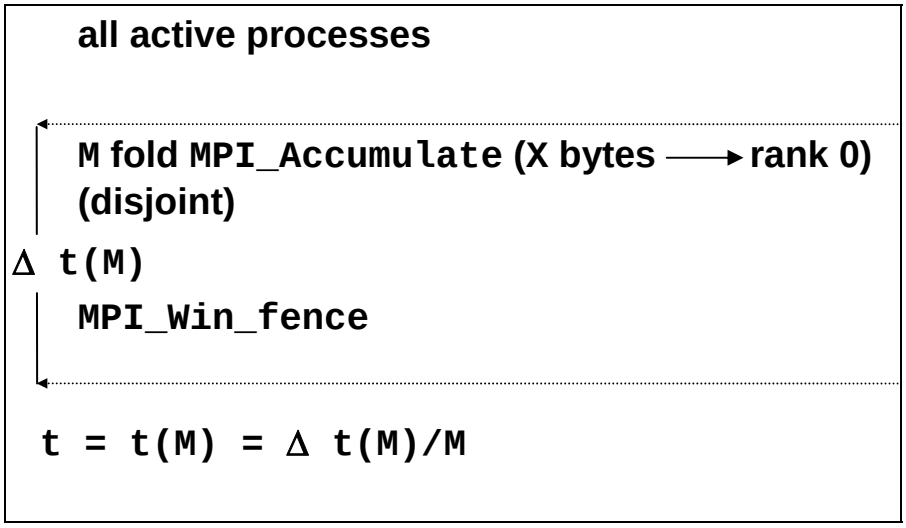

Figure 5-5: Accumulate pattern

5.1.4.5 Window

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

Below see the basic definitions and a schematic view of the pattern.

measured pattern	MPI_Win_create / MPI_Win_fence /

	MPI_Win_free
reported timings	t=Δt (in μsec)
reported throughput	none

Table 5-8: Window definition

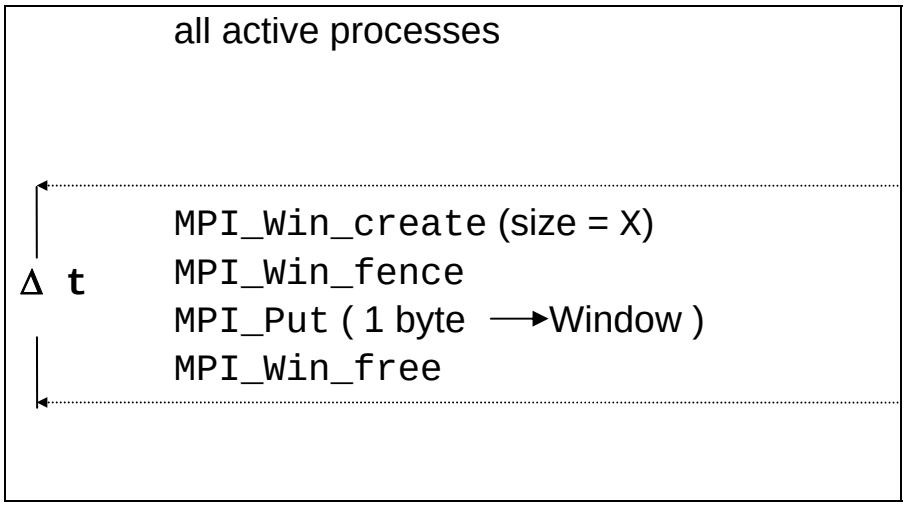

Figure 5-6: Window pattern

5.1.5 Definition of the IMB-IO Benchmarks (Blocking Case)

This section describes the blocking I/O benchmarks in detail. The benchmarks are run with varying transfer sizes X (in bytes).The timings are averaged over multiple samples. Below see the view of one single sample with a fixed I/O size of X. Basic MPI data-type for all data buffers is MPI_BYTE.

All benchmark flavors have a Write and a Read component. A symbol [ACTION] will be used to denote a Read or a Write alternatively.

Every benchmark contains an elementary I/O action, denoting the pure read/write. In the Write cases, a file synchronization is included with different placements for aggregate and non aggregate modes.

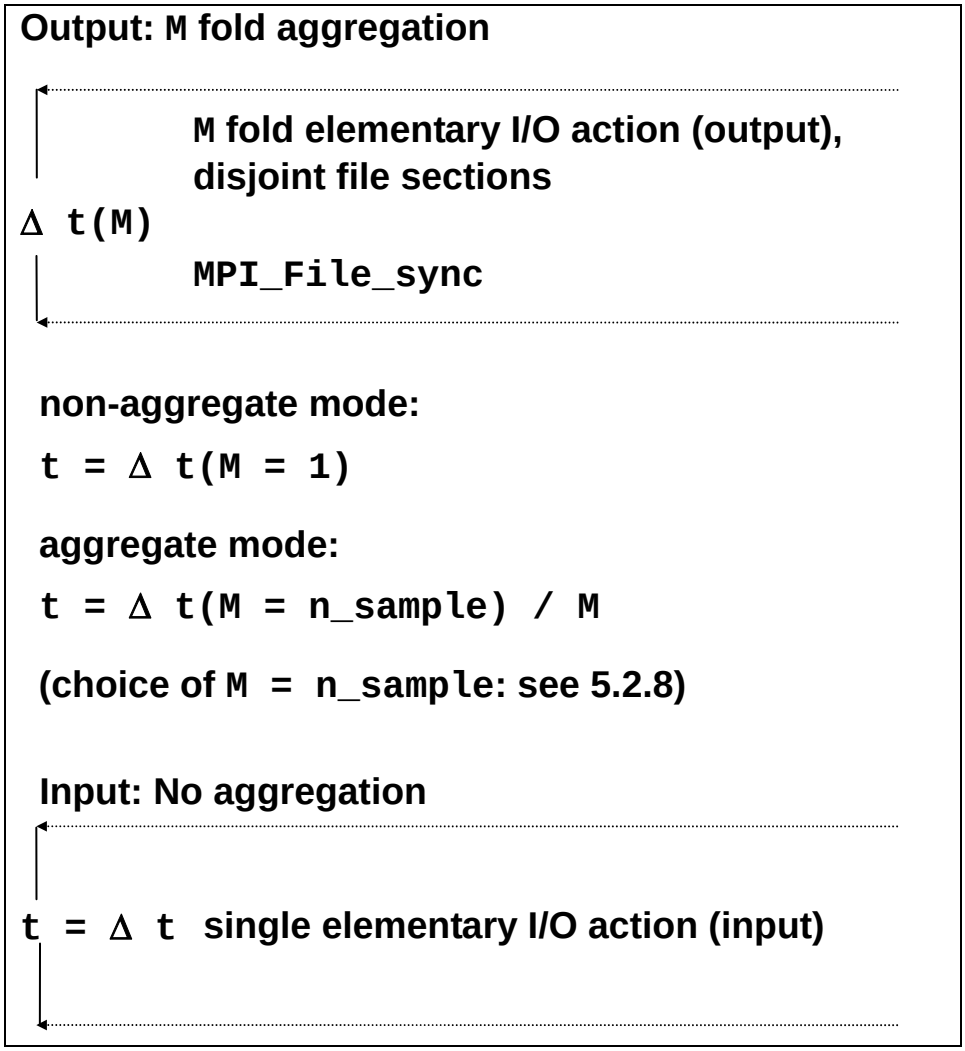

Figure 5-7: I/O benchmarks, aggregation for output

5.1.5.1 S_[ACTION]_indv

File I/O performed by a single process. This pattern mimics the typical case that one particular (master) process performs all of the I/O. Below see the basic definitions and a schematic view of the pattern.

measured pattern	as symbolized in Figure 5-8
elementary I/O action	as symbolized Figure 4-2
for nonblocking mode based on	MPI_File_write / MPI_File_read MPI_File_iwrite / MPI_File_iread
etype	MPI_BYTE
filetype	MPI_BYTE
MPI_Datatype	MPI_BYTE

reported timings	t (in µsec), aggregate and non-aggregate for Write case
reported throughput	X/t, aggregate and non-aggregate for Write case

Table 5-9: S_[ACTION]_indv definition

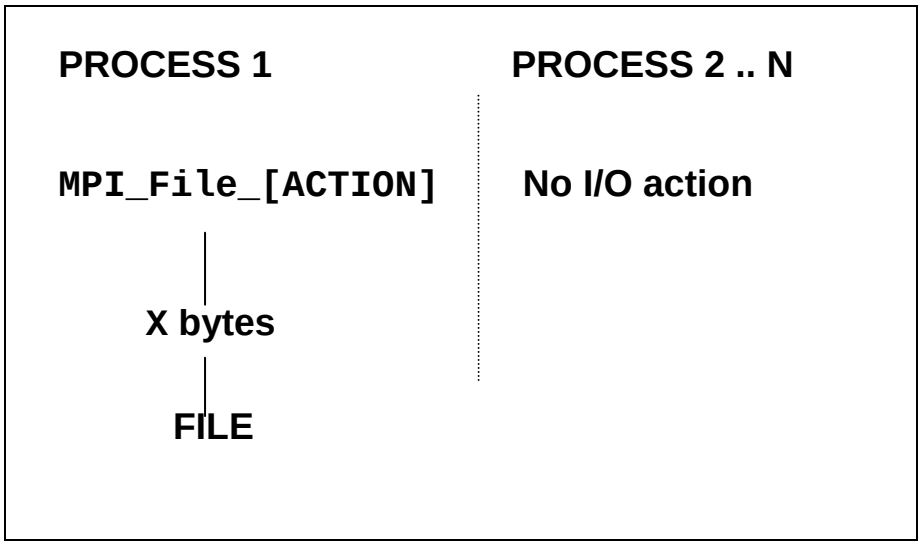

Figure 5-8: S_[ACTION]_indv pattern

5.1.5.2 S_[ACTION]_expl

Mimics the same situation as S_[ACTION]_indv, with a different strategy to access files. Below see the basic definitions and a schematic view of the pattern.

measured pattern	as symbolized in Figure 5-9
elementary I/O action	as symbolized in Figure 5-9
for nonblocking mode based on	MPI_File_write_at / MPI_File_read_at MPI_File_iwrite_at / MPI_File_iread_at
etype	MPI_BYTE
filetype	MPI_BYTE
MPI_Datatype	MPI_BYTE
reported timings	t (in µsec), aggregate and non-aggregate for Write case
reported throughput	X/t, aggregate and non-aggregate for Write case

Table 5-10: S_[ACTION]_expl definition

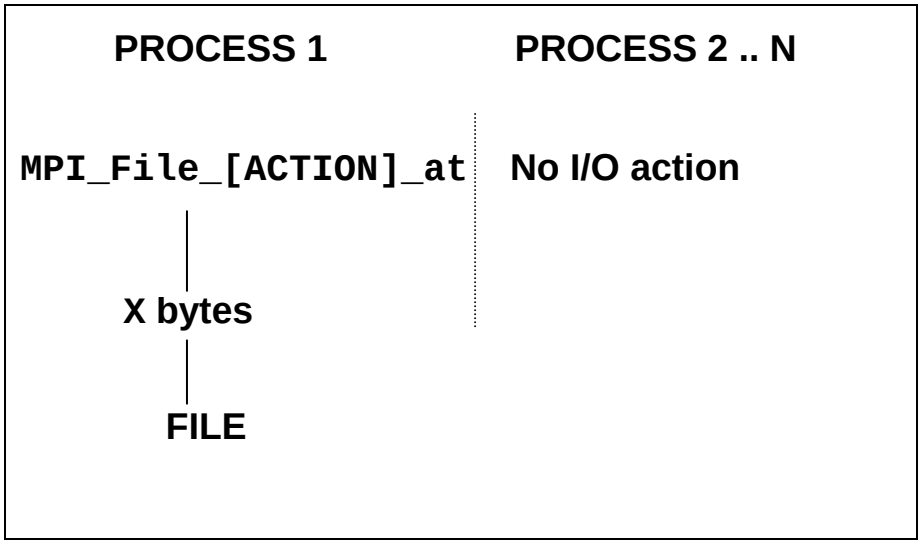

Figure 5-9: S_[ACTION]_expl pattern

5.1.5.3 P_[ACTION]_indv

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

measured pattern	as symbolized in Figure 5-8
elementary I/O action	as symbolized in Figure 4-2 (Nproc = number of processes)
for non-blocking mode based on	`MPI_File_write` / `MPI_File_read` `MPI_File_iwrite` / `MPI_File_iread`
`etype`	`MPI_BYTE`
`filetype`	tiled view, disjoint contiguous blocks
`MPI_Datatype`	`MPI_BYTE`
reported timings	t (in μsec), aggregate and non-aggregate for Write case
reported throughput	X/t, aggregate and non- aggregate for Write case

Table 5-11: P_[ACTION]_indv definition

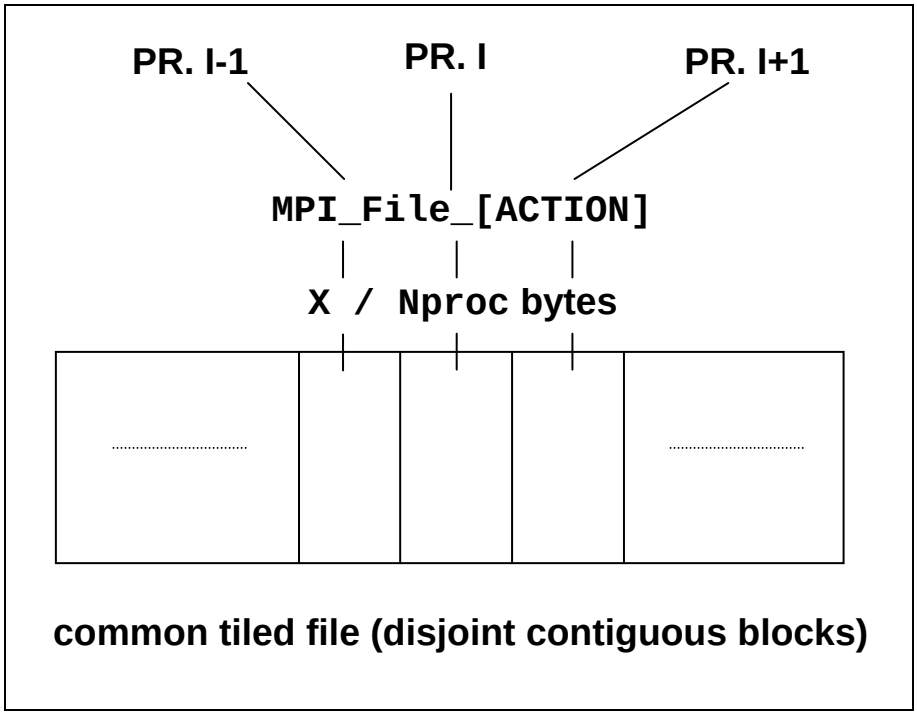

Figure 5-10: P_[ACTION]_indv pattern

5.1.5.4 P_[ACTION]_expl

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

measured pattern	as symbolized in Figure 12
elementary I/O action	as symbolized in Figure 16 (Nproc = number of processes)
for non-blocking mode based on	`MPI_File_write_at` / `MPI_File_read_at` `MPI_File_iwrite_at` / `MPI_File_iread_at`
`etype`	`MPI_BYTE`
`filetype`	`MPI_BYTE`
`MPI_Datatype`	`MPI_BYTE`
reported timings	t (in μsec) as indicated in Figure 12, aggregate and non- aggregate for Write case
reported throughput	X/t, aggregate and non- aggregate for Write case

Table 5-12: P_[ACTION]_expl definition

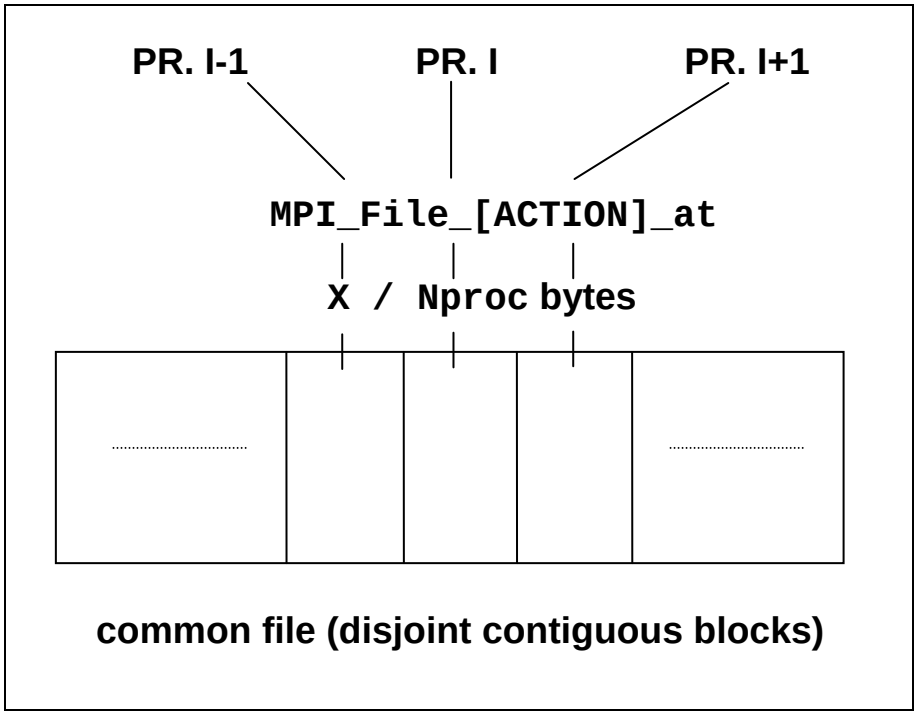

Figure 5-11: P_[ACTION]_expl pattern

5.1.5.5 P_[ACTION]_shared

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

Table 5-13: P_[ACTION]_shared definition

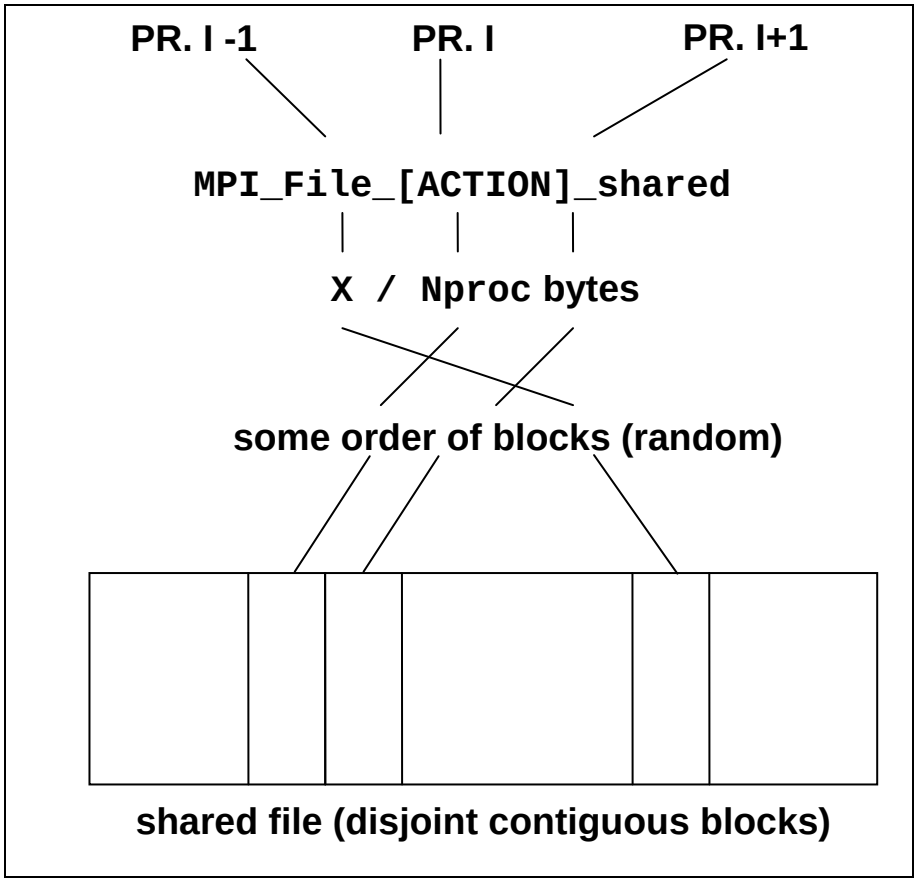

Figure 5-12: P_[ACTION]_shared pattern

5.1.5.6 P_[ACTION]_priv

This pattern tests the case that all participating processes perform concurrent I/O to different, private files. This benchmark is particularly useful for the systems allowing completely independent I/O operations from different processes. It is expected that the benchmark pattern should show parallel scaling and obtain optimum results. Below see the basic definitions and a schematic view of the pattern.

measured pattern	as symbolized in Figure 12
elementary I/O action	as symbolized in Figure 18 (Nproc = number of processes)
for non-blocking mode based on	`MPI_File_write` / `MPI_File_read` `MPI_File_iwrite` / `MPI_File_iread`
`etype`	`MPI_BYTE`
`filetype`	`MPI_BYTE`
`MPI_Datatype`	`MPI_BYTE`
reported timings	Δ t (in μsec), aggregate and non-aggregate for Write case
reported throughput	X/Δt, aggregate and non-aggregate for Write case

Table 5-14: P_[ACTION]_priv definition

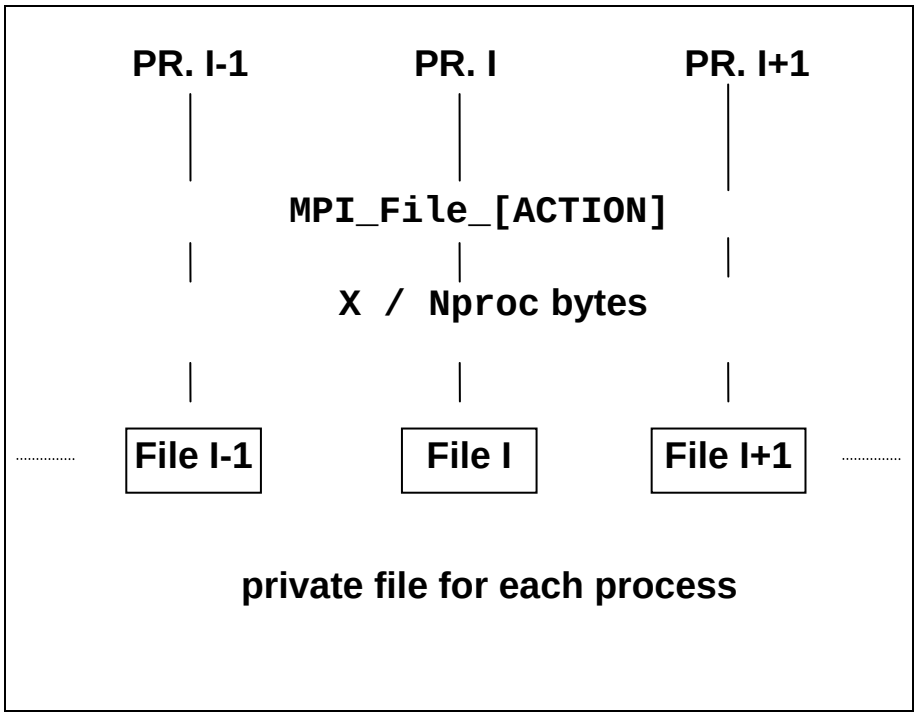

Figure 5-13: P_[ACTION]_priv pattern

5.1.5.7 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.

for non-blocking mode based on	MPI_File_read_all / MPI_File_write_all MPI_File_.._all_begin - MPI_File_.._all_end
all other parameters, measuring method	see 5.1.5.3

Table 5-15: C_[ACTION]_indv definition

5.1.5.8 C_[ACTION]_expl

This pattern performs Collective access from all processes to a common file, with an explicit file pointer. Below see the basic definitions and a schematic view of the pattern.

for non-blocking mode based on	MPI_File_read_at_all / MPI_File_write_at_all MPI_File_.._at_all_begin - MPI_File_.._at_all_end
all other parameters,	see 5.1.5.4

measuring method	

Table 5-16: C_[ACTION]_expl definition

5.1.5.9 C_[ACTION]_shared

A collective access from all processes to a common file, with a shared file pointer is benchmarked. Below see the basic definitions and a schematic view of the pattern.

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

all other parameters, measuring method	see 5.1.5.5

Table 5-17: C_[ACTION]_shared definition

5.1.5.10 Open_Close

The benchmark of the `MPI_File_open / MPI_File_close` pair. All processes open the same file. To prevent the implementation from optimizations in case of an unused file, a negligible non-trivial action is performed with the file. Below see the basic definitions.

measured pattern	`MPI_File_open / MPI_File_close`
Etype	`MPI_BYTE`
Filetype	`MPI_BYTE`
reported timings	$t=\Delta t$ (in μsec)
reported throughput	none

Table 5-18: Open_Close definition

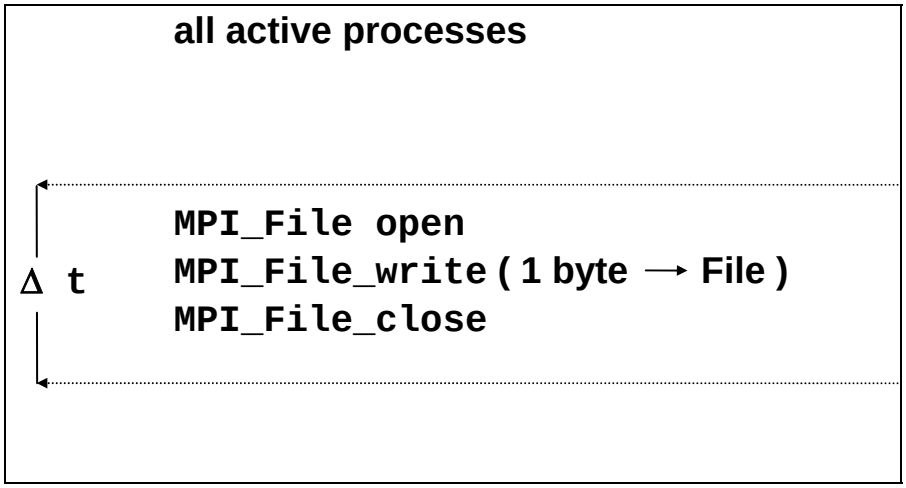

Figure 5-14: Open_Close pattern

5.1.6 Non-blocking I/O Benchmarks

Each of the non-blocking benchmarks has a blocking equivalent. All the definitions can be transferred identically, except their behavior with respect to:

- aggregation (the non-blocking versions only run in aggregate mode)
- synchronism

As to 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 end of transfer

     }

time = (MPI_Wtime()-time)/n_sample
```

The Exploit CPU section is arbitrary. A benchmark such as The Intel® MPI Benchmarks can only decide for one particular way of exploiting the CPU, and will answer certain questions in that special case. There is *no way to cover generality*, only hints can be expected.

5.1.6.1 Exploiting CPU

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

```
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, of course). In the very beginning, the function has to be called with initialize=1 and an input value for `desired_time`. It will determine an Mflop/s rate and a timing `t_CPU` (as close as possible to `desired_time`), obtained by running without any obstruction. Then, during the proper benchmark, it will be called (concurrent with the particular I/O action), with initialize=0 and always performing the same type and number of operations as in the initialization step.

5.1.6.2 Displaying Results

Three timings are crucial to interpret the behavior of non-blocking 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 is the time the `CPU_Exploit` periods (running concurrently with non-blocking I/O) would use when running dedicated
- `t_ovrl` is the time for the analogous non-blocking I/O action, concurrent with CPU activity (exploiting `t_CPU` when running dedicated)
- A perfect overlap would mean: `t_ovrl = max(t_pure,t_CPU)`. No overlap would mean: `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 results tables will report the timings `t_ovrl`, t_pure,t_CPU and the estimated overlap obtained by (*) above. In the beginning of a run the `Mflop/s` rate corresponding to `t_CPU` is displayed.

5.1.7 Multi - versions

The definition and interpretation of the `Multi-` prefix is analogous to the definition in the MPI1 section.

6 Benchmark Methodology

Some control mechanisms are hard coded (like the selection of process numbers to run the benchmarks on); some are set by preprocessor parameters in a central include file. There is a *standard* and an *optional* mode to control the Intel® MPI Benchmarks. In standard mode, all configurable sizes are predefined and should not be changed. This assures comparability for a result tables in standard mode. In optional mode, you can set those parameters at own choice. You can use this mode to extend the results tables as to larger transfer sizes.

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

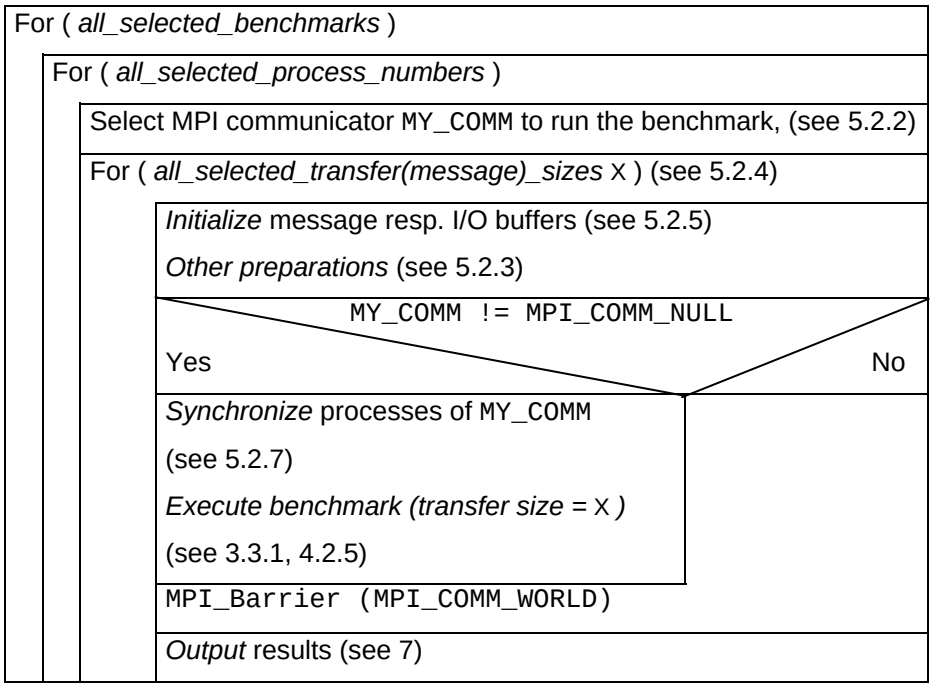

Figure 6-1: Control flow of IMB

The control parameters that are obviously necessary are either command- line arguments or parameter selections inside the Intel MPI Benchmarks include files `settings.h` / `settting_io.h`.

6.1 Running IMB, Command-line Control

After installation, the executables files IMB-MPI1, IMB-EXT and/or IMB-IO should exist.

Given `P`, the (normally user selected) number of MPI processes to run the Intel® MPI Benchmarks, a startup procedure has to load parallel Intel MPI Benchmarks. This is done by `mpirun -np P IMB-<..> [arguments]`

P=1 is allowed and sensible for all I/O and also for all message passing benchmarks except the Single Transfer ones. Control arguments (in addition to `P`) can be passed to the Intel® MPI

Benchmarks through (argc,argv). The command-line arguments are only read by process 0 in MPI_COMM_WORLD. However, the command-line options are broadcast to all other processes.

6.1.1 Default Case

Invoke the following command:

```
mpirun -np P IMB-<..>
```

All benchmarks will run on Q=[1,] 2, 4, 8, ..., largest 2x<P, P processes (Q=1 as discussed above IMB-IO). For example: P=11, then Q=[1,]2,4,8,11 processes will be selected. The Single Transfer IMB-IO benchmarks will run only with Q=1, the Single Transfer IMB-EXT benchmarks only with Q=2.

The Q processes are the *active processes*.

6.1.2 Command-line Control

The command line is repeated in the output. The general command-line syntax is:

```
IMB-MPI1     [-h{elp}]

             [-npmin      <NPmin>]

             [-multi      <MultiMode>]

             [-off_cache <cache_size[,cache_line_size]>

             [-iter
            <msgspersample[,overall_vol[,msgs_nonaggr]]>]

             [-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>]

             [benchmark1 [,benchmark2 [,...]]]
```

```
(where the 13 major [ ] may appear in any order).

Examples:

mpirun -np 8  IMB-IO

mpirun –np 10 IMB-MPI1 PingPing Reduce

mpirun -np 11 IMB-EXT  -npmin 5

mpirun -np 14 IMB-IO   P_Read_shared -npmin 7

mpirun –np 2 IMB-MPI1 pingpong –off_cache -1

(get out-of-cache data for PingPong)

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

(very large configuration – restrict iterations to 20, max. 1.5 seconds run
time per message size, max. 2 GBytes for message buffers)

mpirun -np 3  IMB-EXT  -input IMB_SELECT_EXT

mpirun –np 14 IMB-MPI1 –multi 0 PingPong Barrier
                       -map 2x7

mpirun –np 16 IMB-MPI1 –msglog 2:7 -include PingPongSpecificsource
PingPingSpecificsource –exclude Alltoall Alltoallv

mpirun –np 4 IMB-MPI1 –msglog 16 PingPong PingPing PingPongSpecificsource
PingPingSpecificsource
```

6.1.2.1 Benchmark Selection Arguments

Benchmark selection arguments are a sequence of blank-separated strings. Each argument is the name of one IMB-<..> benchmark (in exact spelling, case insensitive).

Default (no benchmark selection): select all benchmarks.

6.1.2.2 `-npmin` Selection

The argument after `-npmin` has to be an integer `P_min`, specifying the minimum number of processes to run all selected benchmarks.

- `P_min` may be 1
- `P_min` > P is handled as `P_min = P`

Default:

(no `-npmin` selection):

Given P_min, the selected process numbers are P_min, 2P_min, 4P_min, ..., largest 2xP_min <P, P.

6.1.2.3 -multi <outflag> Selection

For selecting Multi/non-Multi mode. The argument after –multi is the meta-symbol <outflag> and this meta-symbol represents an integer value of either 0 or 1. This flag just controls the way of displaying results.

- Outflag = 0: only display max timings (min throughputs) over all active groups
- Outflag = 1: report on all groups separately (may become longish)

When the number of processes running the benchmark is more than half of the overall (MPI_COMM_WORLD) number, the multi benchmark coincides with the non-multi one, as no more than one group can be created.

NOTE: Default:

NOTE: (no –multi selection): run primary (non-Multi) versions.

6.1.2.4 -off_cache cache_size[,cache_line_size] Selection

The argument after off_cache can be either one single number (cache_size), or two comma separated numbers (cache_size,cache_line_size), or just -1,

By default, without this flag, the communications buffer is the same within all repetitions of one message size sample; cache re-usage is yielded and thus throughput results that might be non-realistic.

With -off_cache, it is attempted to avoid cache re-usage.

cache_size is a float for an upper bound of the size of the last level cache in Mbytes, cache_line_size is assumed to be the size of a last level cache line (can be an upper estimate).

The sent/recv'd data are stored in buffers of size ~ 2 x 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. Only when those buffers have been marched through, will they then will be re-used from the beginning.

cache_size and cache_line_size are assumed as statically defined in => IMB_mem_info.h; these are used when -off_cache -1 is entered.

-off_cache is effective for IMB-MPI1, IMB-EXT, but not IMB-IO. Examples:

-off_cache -1 (use defaults of IMB_mem_info.h);

-off_cache 2.5 (2.5 MB last level cache, default line size);

-off_cache 16,128 (16 MB last level cache, line size 128);

The off_cache mode might also be influenced by eventual internal caching with the MPI library. This could make the interpretation intricate.

Default:

no cache control, data likely to come out of cache most of the time

6.1.2.5 -iter

The argument after `-iter` can be one single, two comma separated, or three comma separated integer numbers, which override the defaults

`MSGSPERSAMPLE, OVERALL_VOL, MSGS_NONAGGR of =>IMB_settings.h`

NOTE: Examples

```
-iter 2000         (override MSGSPERSAMPLE by value 2000)
-iter 1000,100     (override OVERALL_VOL by 100)
-iter 1000,40,150 (override MSGS_NONAGGR by 150)
```

Default:

iteration control through parameters `MSGSPERSAMPLE`,`OVERALL_VOL`,`MSGS_NONAGGR => IMB_settings.h`

The `iter` selection is overridden by a dynamic selection that is a new default in the Intel® MPI Benchmarks 3.2: when a maximum run time (per sample) is expected to be exceeded, the iteration number will be cut down; see -time flag.

6.1.2.6 -time

The argument after `-time` is a float, specifying that a benchmark will run at most that many seconds per message size the combination with the `-iter` flag or its defaults is so that always the maximum number of repetitions is chosen that fulfills all restrictions.

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

Default:

`-time` is activated; the float value specifying the run time seconds per sample is set in `IMB_settings.h / IMB_settings_io.h` (the variable `SECS_PER_SAMPLE`, current value `10`)

6.1.2.7 -mem

The argument after `-mem` is a float, specifying that at most that many GBs are allocated per process for the message buffers benchmarks / message. If the size is exceeded, a warning will be output, stating how much memory would have been necessary, if the overall run is to not be interrupted.

Default:

the memory is restricted by `MAX_MEM_USAGE => IMB_mem_info.h`

6.1.2.8 `-input` <File> Selection

Use the ASCII input file to select the benchmarks. The `IMB_SELECT_EXT` file is the following:

```
#
# IMB benchmark selection file
#
# every line must be a comment (beginning with #), or it
# must contain exactly 1 IMB benchmark name
```

```
#
#Window
Unidir_Get
#Unidir_Put
#Bidir_Get
#Bidir_Put
Accumulate
```

By aid of this file,

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

would run the `IMB-EXT` benchmarks `Unidir_Get` and `Accumulate`.

6.1.2.9 `-msglen` <File> Selection

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

This lengths value then overrides the default message lengths. For IMB-IO, the file defines the I/O portion lengths.

6.1.2.10 `-map` PxQ Selection

Numbers 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
```

6.1.2.11 -include [[benchmark1] benchmark2 …]

The option 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
```

6.1.2.12 -exclude [[benchmark1] benchmark2 …]

The option specifies the list of benchmarks to be exclude from run. For example, to exclude `Allotall` and `Allgather` call:

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

6.1.2.13 -msglog [<minlog>:]<maxlog>

The option allows you to control the lengths of the transfer messages. This setting overrides the MINMSGLOG and MAXMSGLOG hard-coded values. The new message sizes will be 0, 2^minlog, …, 2^maxlog

For example, the command

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

selects the lengths – 0,8,16,32,64,128

```
#---------------------------------------------------
# Benchmarking PingPong
# #processes = 2
#---------------------------------------------------
       #bytes #repetitions      t[usec]   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. The following command

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

selects the lengths – 0,1,2,4,8

```
#---------------------------------------------------
# Benchmarking PingPong
# #processes = 2
#---------------------------------------------------
       #bytes #repetitions      t[usec]   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
```

6.1.2.14 -thead_level <level>

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

6.2 Parameters and Hard-coded Settings

6.2.1 Parameters Controlling IMB

There are ten parameters (set by preprocessor definition) controlling the default the Intel® MPI Benchmarks. MSGSPERSAMPLE, MSGS_NONAGGR, OVERALL_VOL, MINMSGLOG, and MAXMSGLOG can be overridden by the -iter, -time, -mem, and -msglog flags. The definition is in the files

settings.h (IMB-MPI1, IMB-EXT) and settings_io.h (IMB-IO)

A complete list and explanation of settings.h see below.

Both include files are almost identical in structure, but differ in the standard settings. Some names in IMB_settings_io.h contain MSG (for "message"), in consistency with IMB_settings.h.

Table 6-1: IMB (MPI1/EXT) parameters

Parameter (standard mode value)	Meaning
USE_MPI_INIT_THEAD (not set)	set to init MPI by MPI_Init_thread() instead of MPI_Init()
IMB_OPTIONAL(not set)	set when optional settings are to be activated
MINMSGLOG (0)	second smallest data transfer size is max(unit,$2^{MINMSGLOG}$) (the smallest always 0), where unit = sizeof(float) for reductions, unit = 1 else
MAXMSGLOG (22)	largest message size is $2^{MAXMSGLOG}$ Sizes 0, 2^i (i=MINMSGLOG,...,MAXMSGLOG) are used
MSGSPERSAMPLE (1000)	max. repetition count for all IMB-MPI1 benchmarks
MSGS_NONAGGR (100)	max. repetition count for non aggregate benchmarks (relevant only for IMB-EXT)
OVERALL_VOL (40 MBytes)	for all sizes < OVERALL_VOL, the repetition count is reduced so that not more than OVERALL_VOL bytes overall are processed. This avoids unnecessary repetitions for large message sizes. Finally, the real repetition count for message size X is MSGSPERSAMPLE (X=0), min(MSGSPERSAMPLE,

	max(1,OVERALL_VOL/X)) (X>0) Note that OVERALL_VOL does *not* restrict the size of the max. data transfer. $2^{MAXMSGLOG}$ is the largest size, independent of OVERALL_VOL
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)	Number of MPI_Barrier for synchronization
TARGET_CPU_SECS(0.01)	CPU seconds (as float) to run concurrent with non-blocking benchmarks (currently irrelevant for IMB-MPI1)

The Intel MPI Benchmarks allows for two sets of parameters: standard and optional. Below see a sample of file settings_io.h. Here, IMB_OPTIONAL is set, so that user defined parameters are used. I/O sizes 32 and 64 MB (and a smaller repetition count) are selected, extending the standard mode tables. If IMB_OPTIONAL is deactivated, the obvious standard mode values are taken.

IMB has to be re-compiled after a change of settings.h/settings_io.h.

```
#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

/*DON'T change anything below here !!*/

#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
```

6.2.2 Communicators, Active Processes

Communicator management is repeated in every "select MY_COMM" step. If it exists, the previous communicator is freed. When running 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.

6.2.3 Other Preparations

6.2.3.1 Window (IMB_EXT)

An Info is set and MPI_Win_create is called, creating a window of size X for MY_COMM. Then, MPI_Win_fence is called to start an access epoch.

6.2.3.2 File (IMB-IO)

The file initialization consists of:

selecting a file name: This parameter is located in include file settings_io.h. In a Multi case, a suffix _g<groupid> is appended to the name. If the file name is per process, a (second event) suffix _<rank> will be appended.

deleting the file if exists:
open it with MPI_MODE_DELETE_ON_CLOSE
close it

selecting a communicator to open the file, which will be:MPI_COMM_SELF for S_benchmarks and P_[ACTION]_priv.

selecting a mode = MPI_MODE_CREATE | MPI_MODE_RDWR

selecting an info

6.2.3.3 Info

The Intel® MPI Benchmarks uses an external function User_Set_Info which you implement for the current machine. 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 use no assumptions and imposes no restrictions on how this routine is implemented.

6.2.3.4 View (IMB-IO)

The file view is determined by the settings:

- `disp = 0`
- `datarep = native`
- `etype, filetype` as defined in the single definitions
- info as defined in 6.2.3.3

6.2.4 Message / I-O Buffer Lengths

6.2.4.1 IMB-MPI1, IMB-EXT

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

6.2.4.2 IMB-IO

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

6.2.5 Buffer Initialization

Communication and I/O buffers are dynamically allocated as void* and used as `MPI_BYTE` buffers for all benchmarks except Accumulate. See 7.1 for the memory requirements. To assign the buffer contents, a cast to an assignment type is performed. On the one hand, a sensible data-type is mandatory for Accumulate. On the other hand, this facilitates results checking which may become necessary.

The Intel MPI Benchmarks set the buffer assignment type by type `assign_type` in `settings.h/settings_io.h`

Currently, it is used for `IMB-IO`, float for `IMB-EXT`. The values are current set by a CPP macro:

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

(IMB-EXT), and

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

(IMB-IO)

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

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

where rank is the MPI rank of the calling process.

6.2.6 Warm-up Phase (MPI1, EXT)

Before starting the actual benchmark measurement for IMB-MPI1 and IMB-EXT, the selected benchmark is executed N_WARMUP (defined in settings.h, see 5.2.1) times with a sizeof(assign_type) message length. This is to hide eventual initialization overheads of the message passing system.

6.2.7 Synchronization

Before the actual benchmark is run, the constant N_BARR (constant defined in IMB_settings.h and IMB_settings_io.h, with a current value of 2) is used to regulate calls to: MPI_Barrier(MPI_COMM_WORLD) so as to assure that all processes are synchronized.

6.2.8 The Actual Benchmark

To reduce measurement errors caused by insufficient clock resolution, every benchmark is run repeatedly. The repetition count for MPI1- or aggregate EXT / IO benchmarks is MSGSPERSAMPLE (constant defined in settings.h/settings_io.h, current values 1000 / 50). To avoid excessive runtimes for large transfer sizes X, an upper bound is set to OVERALL_VOL/X (OVERALL_VOL constant defined in settings.h / settings_io.h, current values 4 / 16 MB). Finally,

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

is the repetition count for all aggregate benchmarks, given transfer size X. The repetition count for non-aggregate benchmarks is defined completely analogously, with MSGSPERSAMPLE replaced by MSGS_NONAGGR (a reduced count is sensible as non-aggregate runtimes are normally much longer).

In the following, *elementary transfer* means the pure function (MPI_[Send, …], MPI_Put, MPI_Get, MPI_Accumulate, MPI_File_write_XX, MPI_File_read_XX), without any further function call. Recall that assure transfer completion means MPI_Win_fence (one sided communications), MPI_File_sync (I/O Write benchmarks), and is empty for all other benchmarks.

6.2.8.1 MPI1 Case

```
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
```

6.2.8.2 EXT and Blocking I/O Case

For the aggregate case, the kernel loop looks the following:

```
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
```

In the non-aggregate case, every Single Transfer is safely completed:

```
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
```

6.2.8.3 Non-blocking I/O Case

A non-blocking benchmark has to provide three timings (blocking pure I/O time `t_pure`, non-blocking I/O time `t_ovrl` (concurrent with CPU activity), pure CPU activity time `t_CPU`). The actual benchmark consists of

- Calling the equivalent blocking benchmark as defined in 6.2.8 and taking benchmark time as `t_pure`
- Closing and re-opening the particular file(s)
- Once again synchronizing the processes
- Running the non-blocking case, concurrent with CPU activity (exploiting t_CPU when running undisturbed), taking the effective time as t_ovrl.

The desired CPU time to be matched (approximately) by `t_CPU` is set in `settings_io.h:`

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

7 Output

The output results are most easily explained by sample outputs. See the tables below.

- General information:
  machine, system, release, and, version are obtained by the code `IMB_g_info.c`.
- The calling sequence (command-line flags) are repeated in the output chart
- Non-multi case numbers

After a benchmark completes, three time values are available: `Tmax`, `Tmin`, and `Tavg`, which represent the maximum, minimum, and average time, respectively, extended over the group of active processes. The time unit is µsec.

The Single Transfer Benchmarks:

Display X = message size [bytes], T=Tmax[µsec],

bandwidth = X / 1.048576 / T

The Parallel Transfer Benchmarks:

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

The Collective Benchmarks:

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

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

7.1 Sample 1 – IMB-MPI1 PingPong Allreduce

```
<..> np 2 IMB-MPI1 PingPong Allreduce

#---------------------------------------------------
#    Intel (R) MPI Benchmark Suite V3.2, MPI-1 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[usec]   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[usec]  t_max[usec]   t_avg[usec]
      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
```

7.2 Sample 2 – IMB-MPI1 PingPing Allreduce

```
<..> -np 6 IMB-MPI1
  pingping allreduce -map 2x3 -msglen Lengths -multi 0

Lengths file:
0
100
1000
10000
100000
1000000

#---------------------------------------------------
#    Intel (R) MPI Benchmark Suite V3.2.2, MPI-1 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 simultaneous )
# Group  0:     0    3
#
# Group  1:     1    4
#
# Group  2:     2    5
#
#-----------------------------------------------------------------
  # bytes #rep.s t_min[usec] t_max[usec] t_avg[usec] Mbytes/sec
      0   1000       ..          ..          ..          ..
    100   1000
   1000   1000
  10000   1000
 100000    419
1000000     41

#-----------------------------------------------------------------
# Benchmarking Multi-Allreduce
# ( 3 groups of 2 processes each running simultaneous )
# Group  0:     0    3
#
# Group  1:     1    4
#
# Group  2:     2    5
#
#-----------------------------------------------------------------
 #bytes #repetitions  t_min[usec]  t_max[usec]  t_avg[usec]
      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[usec]  t_max[usec]  t_avg[usec]
      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[usec]   t_max[usec]   t_avg[usec]
      0         1000         ..           ..            ..
    100         1000
   1000         1000
  10000         1000
 100000          419
1000000           41


# All processes entering MPI_Finalize
```

7.3 Sample 3 – IMB-IO p_write_indv

```
<..> 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[usec]      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[usec]      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
```

7.4 Sample 4 – IMB-EXT.exe

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

#---------------------------------------------------
#    Intel (R) 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               : 6.0.6001
# 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[usec]   t_max[usec] t_avg[usec]
            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.

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

8 Further Details

8.1 Memory Requirements

Table 8-1: Memory requirements with standard settings

Benchmarks	Standard mode memory demand per process (Q active processes)	Optional mode memory demand per process (X = max. occurring message size)
Alltoall	Q × 8 MB	Q × 2X bytes
Allgather, Allgatherv	(Q+1) × 4 MB	(Q+1) × X bytes
Exchange	12 MB	3X bytes
All other MPI1 benchmarks	8 MB	2X bytes
IMB-EXT	80 Mbytes	2 max(X,OVERALL_VOL) bytes
IMB-IO	32 Mbytes	2X bytes
(to all of the above, add 2x cache size in case –cache is not selected)		
	disk space overall	disk space overall
IMB-IO	16 MB	max(X,OVERALL_VOL) bytes

8.2 Results Checking

By activating the cpp flag –DCHECK through the CPPFLAGS variable, and recompiling, every message passing result from the Intel MPI Benchmarks executable files are checked against the expected outcome. Output tables contain an additional column displaying the diffs as floats (named *defects*).

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