Intel ® MPI Benchmarks

User Guide and Methodology Description

Intel Corporation Document Number: 320714-001

Disclaimer and Legal Information

1 Introduction

This document 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 release (Version 3.2) is the successor of the quite well known package PMB (Version 2.2) from Pallas GmbH, Intel MPI Benchmarks (IMB) 2.3, 3.0, and 3.1.

This document accompanies version 3.2 of IMB. The code is written in *ANSI C plus standard MPI* (about 10,000 lines of code, 108 functions in 37 source files).

The IMB 3.2 package consists of 3 parts:

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

For each part, a separate executable can be built. If you do not have the MPI-2 extensions available, you can install and use just IMB-MPI1. Only standard MPI-1 functions are used, no dummy library is needed.

Section 2 is a brief installation guide.

Section 3 is dedicated to IMB-MPI1. Section 3.3 defines the single benchmarks in detail. IMB introduces a classification of its benchmarks. *Single Transfer, Parallel Transfer, and Collective* are the classes. Roughly speaking, single transfers run *dedicated*, without obstructions from other transfers, undisturbed results are to be expected (`PingPong` being the most well known example). Parallel transfers test the system under global load, with concurrent actions going on. Finally, collective is a proper MPI classification, where these benchmarks test the quality of the implementation for the higher level collective functions.

Chapter 4 is dedicated to the MPI-2 functionality of IMB.

Section 5 defines the methodology and rules of IMB, section 6 shows templates of output tables. In section 7, further important details are explained, in particular a results checking mode for IMB.

1.1 Changes IMB_3.2 against IMB_3.1

IMB_3.2 has different default settings with respect to IMB_3.1, and there are now Microsoft* Visual Studio* project folders that can be used for Microsoft* Windows* platforms. In turn, Makefiles for Windows `nmake` that had been contained in IMB_3.1 have been removed.

1.1.1 Run time control by default

The improved run time control that is associated with the flag `–time`, and that was introduced in IMB_3.1 (see 1.2.2 and 5.1.2.6), has become a default for the 3 executables (with a maximum run time per sample set to 10 s by parameter SECS_PER_SAMPLE in the include file IMB_settings.h).

1.1.2 Makefiles

Windows* nmake files have been removed (and replaced by Microsoft* Visual Studio* solutions, see 1.1.3).

The Linux version Makefiles have 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.

1.1.3 Microsoft* Visual Studio* Project Folders

IMB 3.2 contains Microsoft* Visual Studio* solutions based on an installation of Intel® MPI Library. A dedicated folder for the Microsoft* Windows* versions has been created, however 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.

Since such solutions are highly version dependent, we refrain from elaborate documentation here and refer to the corresponding ReadMe.txt files that unpack with the folder and will be updated continuously. We recommend familiarity with Microsoft* Visual Studio philosophy and the run time environment of your Windows cluster at hand.

1.2 Changes IMB_3.1 against IMB_3.0

The changes against the previous version, 3.0, are new benchmarks, new flags and a Windows* version of IMB 3.1.
As to the new control flags, most important are

- a 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 (see 4.2.2.2.1)

1.2.1 New benchmarks

The 4 benchmarks

- Gather
- Gatherv
- Scatter
- Scatterv

were added and are to be used in the usual IMB style.

1.2.2 New command line flags for better control

The 4 flags added are

-off_cache, -iter, -time, -mem (see 5.1.2 for the details).

-off_cache:
when measuring performance on high speed interconnects or, in particular, across the shared memory within a node, traditional IMB results eventually included a very beneficial cache re-usage of message buffers which led to idealistic results. The flag –off_cache allows for (largely) avoiding cache effects and lets IMB 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 traditional IMB settings.

(!) In IMB_3.2, the -time flag has been implemented as default

-mem
is used to determine an a priori maximum (per process) memory usage of IMB for the overall message buffers.

1.2.3 Miscellaneous changes

- in the "`Exchange`" benchmark, the 2 buffers sent by `MPI_Isend` are separate now
- the command line is repeated in the output
- memory management is now completely encapsulated in functions "`IMB_v_alloc / IMB_v_free`"

1.3 Changes IMB_3.0 against IMB_2.3

The changes of IMB_3.0 against version 2.3 had been:

- added a call to the function "`MPI_Init_thread`" to determine the MPI threading environment. The MPI threading environment is reported each time an Intel MPI Benchmark application is executed.
- added 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.
- added the "`Alltoallv`" benchmark.
- added a command-line flag "-h[elp]" to display the calling sequence for each benchmark application.
- removed outdated Makefile templates. Now there are three complete makefiles called `Makefile`, `make_ict`, and `make_mpich`.
- Better command line argument checking, clean message and break on most invalid arguments.

2 Installation and Quick Start of IMB

In order to run IMB-MPI1, you need:

- `cpp`, ANSI C compiler, `gmake on Linux* or Unix*`.
- For Microsoft Windows, it is recommend that you use the enclosed Microsoft Visual* C++ solutions as a basis.
- MPI installation, including a startup mechanism for parallel MPI programs.

See 7.1 for the memory requirements of IMB.

2.1 Installing and running

After unpacking, the directory contains:

a file `ReadMe_first`

and 5 subdirectories

`./doc` (ReadMe_IMB.txt; IMB_ug-3.2.pdf, this file)

`./src` (program source- and Make-files)

`./WINDOWS` (Visual Studio Solutions)

`./license` (license agreements text)

`./versions_news` (version history and news)

Please read the license agreements first:

- *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 "Intel® MPI Benchmarks"*

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

On Linux, you can remove legacy binary object files and executables by typing the command:

```
make clean
```

You can then build selected executables with the command:

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

or all three executables with the command:

```
make all
```

The above command assumes that the environment variables `CC` has been set appropriately prior to the makefile command invocation.

On Microsoft Windows, you can use the enclosed solution files as basis.

After installation, use your style of starting MPI programs, e.g.

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

to get the full suite of all benchmarks. For more selective running, see 5.1.2

3 IMB-MPI1

This section is dedicated to the part of IMB measuring the 'classical' message passing functionality of MPI-1.

3.1 General

The idea of IMB is to provide a concise set of elementary MPI benchmark kernels. With one executable, all of the supported benchmarks, or a subset specified by the command line, can be run. The rules, such as time measurement (including a repetitive call of the kernels for better clock synchronization), message lengths, selection of communicators to run a particular benchmark (inside the group of all started processes) are program parameters.

IMB has a *standard* and an *optional* configuration (see 5.2.1). In the standard case, all parameters mentioned above are fixed and must not be changed.

In standard mode, message lengths are varied from 0,1,2,4,8,16 … to 4194304 bytes. Through a command line flag, an arbitrary set of message lengths can be input by a file (flag `–msglen`, see 5.1.2.9).

The minimum `P_min` and maximum number `P` of processes can be selected via command line, the benchmarks run on P_min, 2P_min, 4P_min, ... 2^xP_min<P and `P` processes. See chapter 5.1.2.2 for the details.

You have some choice for the mapping of processes. For instance, when running on a clustered system, a benchmark such as `PingPong`, can be run intra node and inter node, without changing a mapping file (`-map` flag, see 5.1.2.10)

3.2 The benchmarks

The current version of IMB-MPI1 contains the benchmarks:

- `PingPong`
- `PingPing`
- `Sendrecv`
- `Exchange`
- `Bcast`
- `Allgather`
- `Allgatherv`
- `Scatter`
- `Scatterv`
- `Gather`
- `Gatherv`
- `Alltoall`
- `Alltoallv`
- `Reduce`
- `Reduce_scatter`
- `Allreduce`
- `Barrier`

The exact definitions will be given in section 3.3. Section 5 describes the benchmark methodology.

IMB-MPI1 allows for running all benchmarks in more than one process group. For example, when running `PingPong` on N≥4 processes, you may request (see 5.1.2.3) that N/2 disjoint groups of 2 processes each be formed, all and simultaneously running `PingPong`.

Note that these multiple versions have to be carefully distinguished from their standard equivalents. They will be called:

- `Multi-PingPong`
- `Multi-PingPing`
- `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`

For a distinction, sometimes we will refer to the standard (non `Multi`) benchmarks as *primary* benchmarks.

The way of interpreting the timings of the `Multi`-benchmarks is quite easy, given a definition for the primary cases: per group, this is as in the standard case. Finally, the max timing (min throughput) over all groups is displayed. On request, all per group information can be reported, see 5.1.2.3.

3.3 IMB-MPI1 benchmark definitions

In this chapter, the single benchmarks are described. Here we focus on the elementary *patterns* of the benchmarks. The methodology of measuring these patterns (message lengths, sample repetition counts, timer, synchronization, number of processes and communicator management, display of results) are defined in chapters 5 and 6.

3.3.1 Benchmark classification

For a clear structuring of the set of benchmarks, IMB introduces classes of benchmarks: *Single Transfer, Parallel Transfer, and Collective*. This classification refers to different ways of interpreting results, and to a structuring of the code itself. It does not actually influence the way of using IMB. Also holds this classification hold for IMB-MPI2 (see 4.2.1).

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

3.3.1.1 Single Transfer benchmarks

The benchmarks in this class are to focus on a *single* message transferred between two processes. As for the `PingPong` benchmark, this is the usual way of looking at it. From an IMB interpretation, `PingPing` measures the same as `PingPong`, under the particular circumstance that a message is obstructed by an oncoming one (sent simultaneously by the same process that receives the own one).

Single transfer benchmarks only run with 2 active processes (see 5.2.2 for the definition of *active*).

For `PingPing`, pure timings will be reported, and the throughput is related to a *single* message. Expected numbers, very likely, are between half and full `PingPong` throughput. With this, `PingPing` determines the throughput of messages under non optimal conditions (namely, oncoming traffic).

See 3.3.2.1 for exact definition.

3.3.1.2 Parallel Transfer benchmarks

These benchmarks focus on *global mode*, say, patterns. The activity at a certain process is in concurrency with other processes, and thus the benchmark measures message passing efficiency under global load.

For the interpretation of `Sendrecv` and `Exchange`, more than 1 message (per sample) counts. As to the throughput numbers, the *total turnover* (the number of *sent plus the number of received bytes*) at a certain process is taken into account. For instance, for the case of 2 processes, `Sendrecv` becomes the *bi-directional* test: perfectly bi-directional systems are rewarded by a double `PingPong` throughput here.

Thus, the throughputs are scaled by certain factors. See 3.3.3.1 and 3.3.3.2 for exact definitions. As to the timings, raw results without scaling will be reported.

The `Multi` mode secondarily introduces into this class

- `Multi-PingPong`
- `Multi-PingPing`
- `Multi-Sendrecv`
- `Multi-Exchange`

3.3.1.3 Collective benchmarks

This class contains all benchmarks that are collective in proper MPI convention. Not only is the message passing power of the system relevant here, but also the quality of the implementation.

For simplicity, we also include the `Multi` versions of these benchmarks into this class.

Raw timings and no throughput are reported.

Note that certain collective benchmarks (namely the reductions) play a particular role as they are not pure message passing tests, but also depend on an efficient implementation of certain numerical operations.

3.3.2 Definition of Single Transfer benchmarks

This section describes the single transfer benchmarks in detail. Each benchmark is run with varying message lengths of X bytes, and timings are averaged over multiple samples. See 5.2.4 for the description of the methodology. Here we describe the view of one single sample, with a fixed message length X bytes. 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, when time is in µsec).

3.3.2.1 PingPong

`PingPong` is the classical pattern used for measuring startup and throughput of a single message sent between two processes.

Measured pattern	As symbolized between in Figure 1; two active processes only (Q=2, see 5.2.2)
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

3.3.2.2 PingPing

`PingPong`, and `PingPing` 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.

Measured pattern	As symbolized between in Figure 2; two active processes only (Q=2, 5.2.2)
based on	`MPI_Isend/MPI_Wait, MPI_Recv`
`MPI_Datatype`	`MPI_BYTE`
reported timings	`time = Δt` (in µsec) as indicated in Figure 2
reported throughput	`X/1.048576/time`

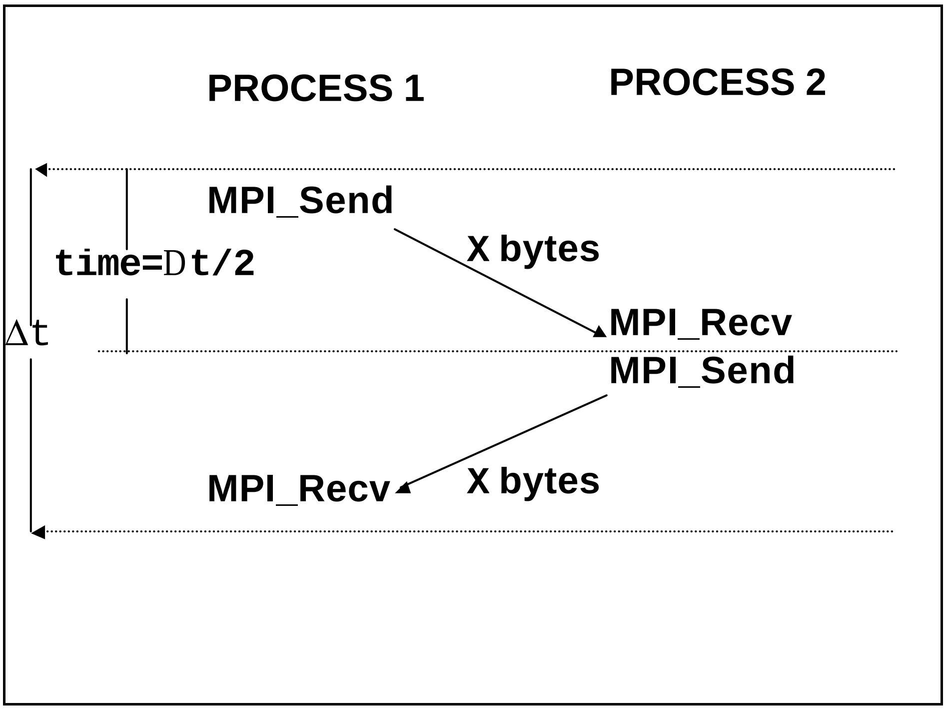

Figure 1: `PingPong` pattern

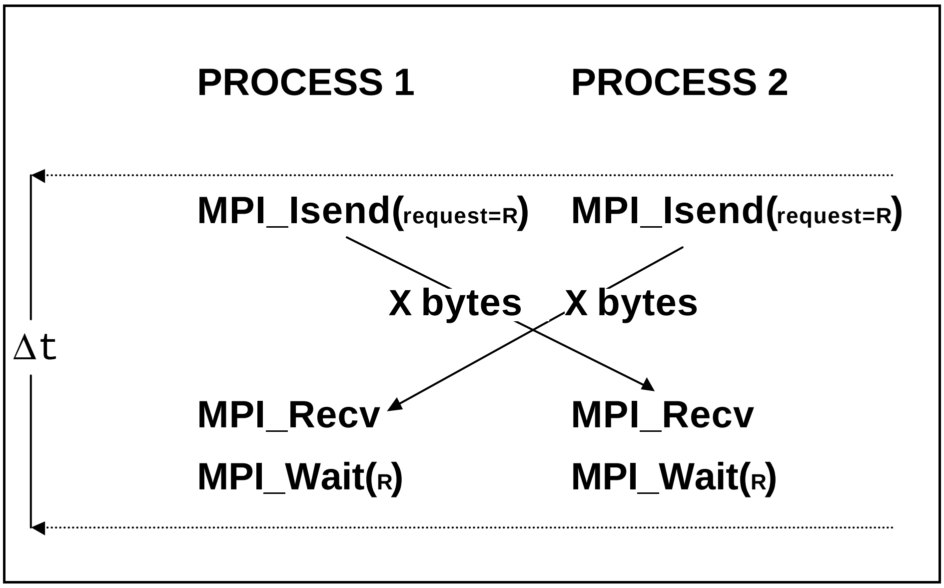

Figure 2: `PingPing` pattern

3.3.3 Definition of Parallel Transfer benchmarks

This section describes the parallel transfer benchmarks in detail. Each benchmark is run with varying message lengths of `X` bytes, and timings are averaged over multiple samples. See section 5 for the description of the methodology. Here we describe the view of one single sample, with a fixed message length of `X` bytes. The basic MPI data-type for all messages is `MPI_BYTE`.

The throughput calculations of the benchmarks described here take into account the (per sample) 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).

3.3.3.1 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 2 messages per sample (1 in, 1 out) for each process.

Sendrecv is equivalent with the Cshift benchmark and, in case of 2 processes, the PingPing benchmark of IMB1.x. For 2 processes, it will report the bi-directional bandwidth of the system, as obtained by the (optimized) MPI_Sendrecv function.

Measured pattern	As symbolized between ⇇ in Figure 3
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

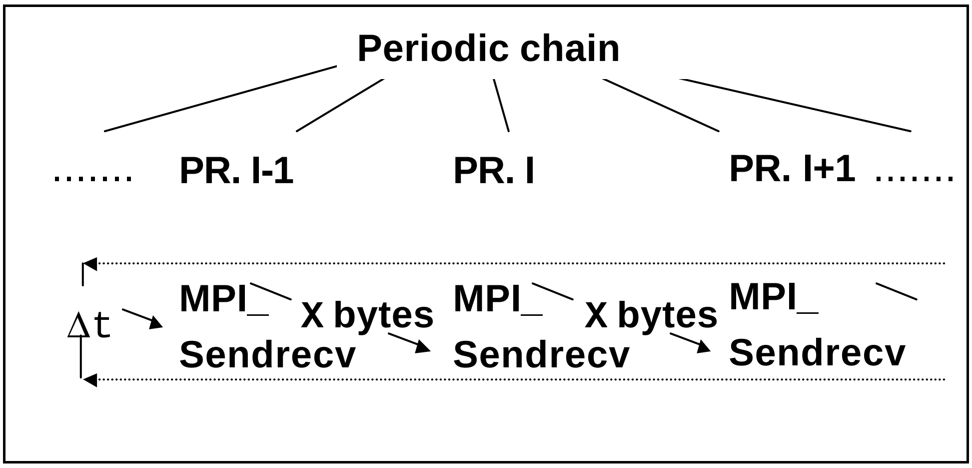

Figure 3: Sendrecv pattern

3.3.3.2 Exchange

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

The turnover count is 4 messages per sample (2 in, 2 out) for each process.

For the 2 Isend messages, separate buffers are used (new in IMB 3.1).

Measured pattern	As symbolized between ⇇ in Figure 4
based on	MPI_Isend/MPI_Waitall, MPI_Recv
MPI_Datatype	MPI_BYTE
reported timings	time = Δt (in μsec) as indicated in Figure 4
reported throughput	4X/1.048576/time

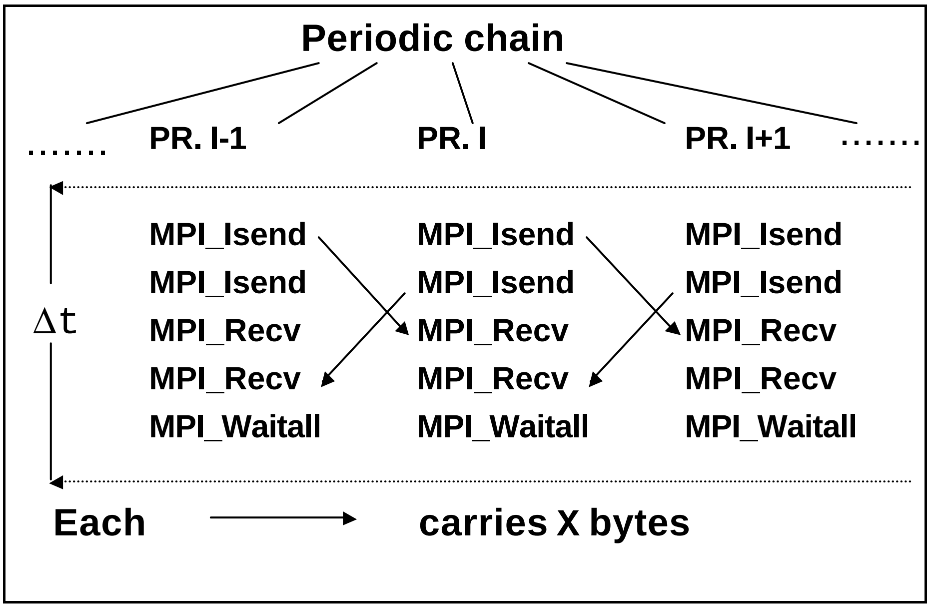

Figure 4: Exchange pattern

3.3.4 Definition of Collective benchmarks

This section describes the Collective benchmarks in detail. Each benchmark is run with varying message lengths of X bytes, and timings are averaged over multiple samples. See section 5 for the description of the methodology. Here we describe the view of one single sample, with a fixed message length of X bytes. 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 is displayed.

3.3.4.1 Reduce

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, and the MPI operation is MPI_SUM.

The root of the operation is changed round robin.

See also the remark in the end of 3.3.1.3.

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

3.3.4.2 Reduce_scatter

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.

See also the remark in the end of 3.3.1.3.

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

3.3.4.3 Allreduce

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.

See also the remark in the end of 3.3.1.3.

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

3.3.4.4 Allgather

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

3.3.4.5 Allgatherv

Functionally 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

3.3.4.6 Scatter

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

3.3.4.7 Scatterv

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

3.3.4.8 Gather

Benchmark for the MPI_Gather function. All processes input X bytes, and 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

3.3.4.9 Gatherv

Benchmark for the MPI_Gatherv function. All processes input X bytes, and 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

3.3.4.10 Alltoall

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

3.3.4.11 Alltoallv

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_Alltoallv
MPI_Datatype	MPI_BYTE
reported timings	bare time
reported throughput	none

3.3.4.12 Bcast

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

3.3.4.13 Barrier

measured pattern	MPI_Barrier
reported timings	bare time
reported throughput	none

4 MPI-2 part of IMB

This section describes how the MPI-2 semantics of IMB, IMB-EXT and IMB-IO, are handled.

4.1 The benchmarks

Table 1 below contains a list of all IMB-MPI2 benchmarks. The exact definitions are given in section 4.2, in particular refer to 4.2.2.2 for an explanation of the *Aggregate Mode*, 4.2.5 for the *Non-blocking Mode* column. Section 5 describes the benchmark methodology.

The non-blocking modes of IMB-IO `read / write` benchmarks are defined as different benchmarks, with `Read / Write` replaced by `IRead / IWrite` in the benchmark names.

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

Table 1: IMB-MPI-2 benchmarks

The naming conventions for the benchmarks are as follows:

- Unidir/Bidir stand for unidirectional/bidirectional one-sided communications. These are the *one-sided equivalents of* PingPong *and* PingPing.
- the Multi- prefix is defined as in 3.2. It is to be interpreted as multi-group version of the benchmark.
- prefixes S_/P_/C_ mean Single/Parallel/Collective. The classification is the same as in the MPI1 case. In the I/O case, a *Single* transfer is defined as a data transfer between *one* MPI process and *one* individual window or file. *Parallel* means that eventually more than 1 process participates in the overall pattern, whereas *Collective* is meant in proper MPI sense. See 3.3.1.
- 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 on each process).

4.2 IMB-MPI2 benchmark definitions

In this section, all IMB-MPI2 benchmarks are described. The definitions focus on the elementary *patterns* of the benchmarks. The methodology of measuring these patterns (transfer sizes, sample repetition counts, timer, synchronization, number of processes and communicator management, display of results) is defined in sections 5 and 6.

4.2.1 Benchmark classification

To clearly structure the set of benchmarks, IMB introduces three classes of benchmarks: *Single Transfer*, *Parallel Transfer*, and *Collective*. This classification refers to different ways of interpreting results, and to a structuring of the benchmark codes. It does not actually influence the way of using IMB. Note that this is the classification already introduced for IMB-MPI1 (3.3.1). Two special benchmarks, measuring accompanying overheads of one sided communications (MPI_Win_create / MPI_Win_free) and of I/O (MPI_File_open / MPI_File_close), have not been assigned a class.

Single Transfer	Parallel Transfer	Collective	Other
Unidir_Get Unidir_Put Bidir_Get Bidir_Put	Multi-Unidir_Get Multi-Unidir_Put Multi-Bidir_Get Multi-Bidir_Put	Accumulate Multi-Accumulate	Window (also Multi)
S_[I]Write_indv S_[I]Read_indv S_[I]Write_expl S_[I]Read_expl	P_[I]Write_indv P_[I]Read_indv P_[I]Write_expl P_[I]Read_expl P_[I]Write_shared P_[I]Read_shared P_[I]Write_priv P_[I]Read_priv	C_[I]Write_indv C_[I]Read_indv C_[I]Write_expl C_[I]Read_expl C_[I]Write_shared C_[I]Read_shared Multi- versions	Open_close (also Multi)

Table 2: IMB-MPI2 benchmark classification

4.2.1.1 Single Transfer benchmarks

The benchmarks in this class 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. Analogously, the target can be an MPI process or an MPI file. Note that with this definition,

- single transfer IMB-EXT benchmarks only run with 2 active processes
- single transfer IMB-IO benchmarks only run with 1 active process (see 5.2.2 for the definition of "active").

Single transfer benchmarks, roughly speaking, are *local mode*. The particular pattern is purely local to the participating processes. There is no concurrency with other activities. Best case results are to be expected.

Raw timings will be reported, and the well-defined throughput.

4.2.1.2 Parallel Transfer benchmarks

These benchmarks focus on *global mode*, say, patterns. The activity at a certain process is in concurrency with other processes, the benchmark timings are produced under global load. The number of participating processes is arbitrary.

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

This definition is applied *per group* in the `Multi` - cases, see 5.1.2.3, and the results of the worst group are displayed.

4.2.1.3 Collective benchmarks

This class contains benchmarks of functions that are collective in the proper MPI sense. Not only is the power of the system relevant here, but also the quality of the implementation for the corresponding higher level functions.

Time is measured as maximum over all single processes' timings, no throughput is calculated.

4.2.2 Benchmark modes

Certain benchmarks have different *modes* to run.

4.2.2.1 Blocking / non-blocking mode (only IMB-IO)

This distinction is in the proper MPI-IO sense. Blocking and non-blocking mode of a benchmark are separated in two single benchmarks, see Table 1. See 4.2.5 for the methodology.

4.2.2.2 Aggregate / Non Aggregate mode

For certain benchmarks, IMB defines a distinction between aggregate and non aggregate mode:

- all one sided communications benchmarks
- all blocking (!) IMB-IO `Write` benchmarks, using some flavor of MPI-IO file writing.

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). It is important to define what "assure completion" means.

4.2.2.2.1 Assure completion of transfers

Assure completion means:

- `MPI_Win_fence` (IMB-EXT)
- A triplet
  `MPI_File_sync / MPI_Barrier (file_communicator) / MPI_File_sync` (IMB-IO `Write`). Following the MPI standard, this is the minimum sequence of operations after which all processes of the file's communicator have a consistent view after a write. This fixes the non sufficient definition in IMB_3.0.

4.2.2.2.2 Mode definition

The basic pattern of these benchmarks is shown in Figure 5. Here,

- `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 (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.

IMB runs the corresponding benchmarks with two settings:

- `M = 1` (non aggregate mode)
- `M = n_sample` (aggregate mode), with `n_sample` as defined later, refer to 5.2.8.

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

Figure 5: 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 5) are accumulated (for example by a caching mechanism), without an immediate completion. At certain stages (system and runtime dependent), at best only in the assure completion part, the accumulated transfers are completed as a whole. This approach may save 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. Expensive synchronizations are taken into account eventually. The expectation is that this strategy would produce (about) equal results for aggregate and non aggregate case.

4.2.3 Definition of the IMB-EXT benchmarks

This section describes the benchmarks in detail. They will run with varying transfer sizes `X` (in bytes), and timings will be averaged over multiple samples. See 5 for the description of the methodology. Here we describe the view of one single sample, with a fixed transfer size `X`.

Note that the `Unidir (Bidir)` benchmarks are exact equivalents of the message passing `PingPong` (`PingPing`, respectively). Their interpretation and output is analogous to their message passing equivalents.

4.2.3.1 Unidir_Put

Benchmark for the `MPI_Put` function. Table 3 below shows the basic definitions. Figure 6 is a schematic view of the pattern.

measured pattern	as symbolized between in Figure 6; 2 active processes only
based on	`MPI_Put`
`MPI_Datatype`	`MPI_BYTE` (origin and target)
reported timings	t=t(M) (in μsec) as indicated in Figure 6, non aggregate (M=1) and aggregate (cf. 0; M=n_sample, see 5.2.8)
reported throughput	X/t, aggregate and non aggregate

Table 3 : Unidir_Put definition

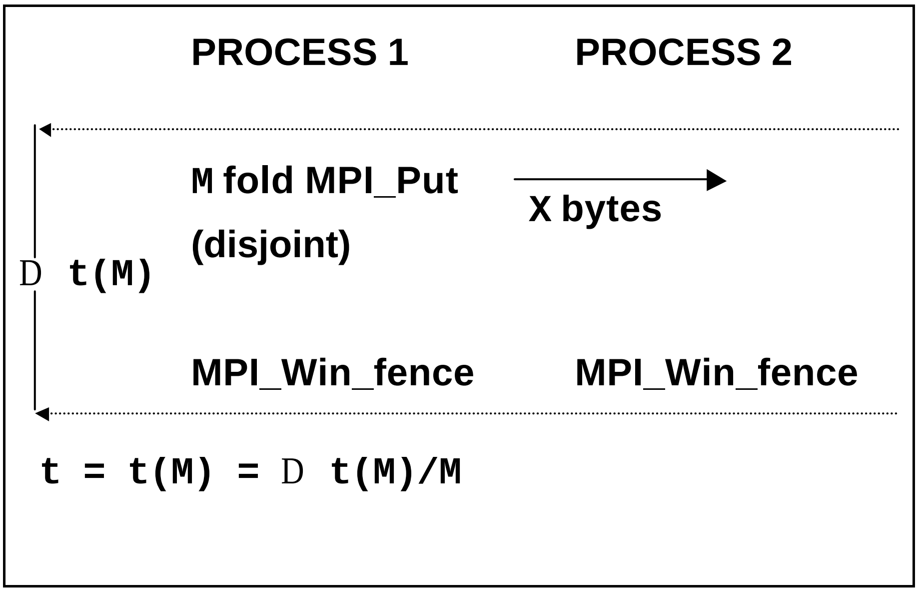

Figure 6: Unidir_Put pattern

4.2.3.2 Unidir_Get

Benchmark for the `MPI_Get` function.

Table 4 below shows the basic definitions. Figure 7 is a schematic view of the pattern.

measured pattern	as symbolized between in Figure 7; 2 active processes only
based on	`MPI_Get`
`MPI_Datatype`	`MPI_BYTE` (origin and target)
reported timings	t=t(M) (in μsec) as indicated in Figure 7, non aggregate (M=1) and aggregate (cf. 0; M=n_sample, see 5.2.8)
reported throughput	X/t, aggregate and non aggregate

Table 4: Unidir_Get definition

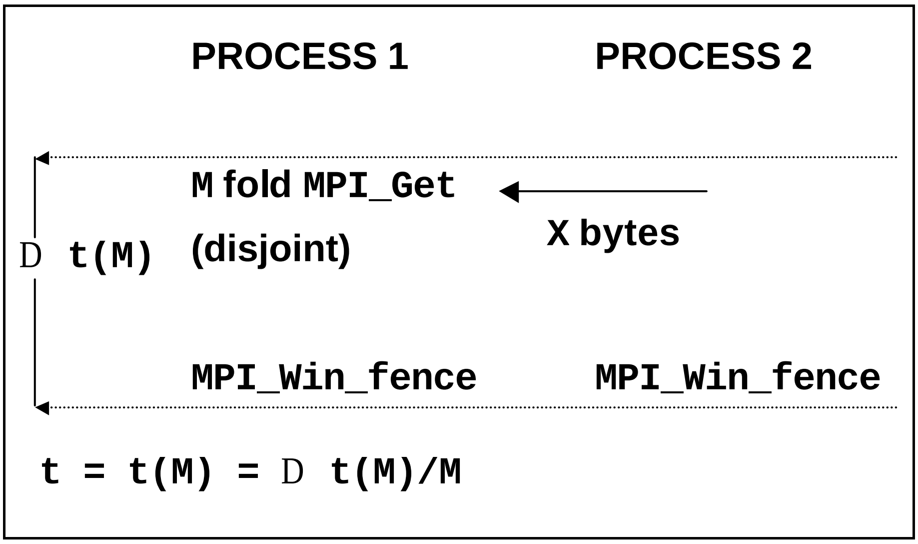

Figure 7: Unidir_Get pattern

4.2.3.3 Bidir_Put

Benchmark for `MPI_Put`, with bi-directional transfers.

Table 5 below shows the basic definitions. Figure 8 is a schematic view of the pattern.

measured pattern	as symbolized between in Figure 8; 2 active processes only
based on	`MPI_Put`
`MPI_Datatype`	`MPI_BYTE` (origin and target)
reported timings	t=t(M) (in μsec) as indicated in Figure 8, non aggregate (M=1) and aggregate (cf. 0; M=n_sample, see 5.2.8)
reported throughput	X/t, aggregate and non aggregate

Table 5: Bidir_Put definition

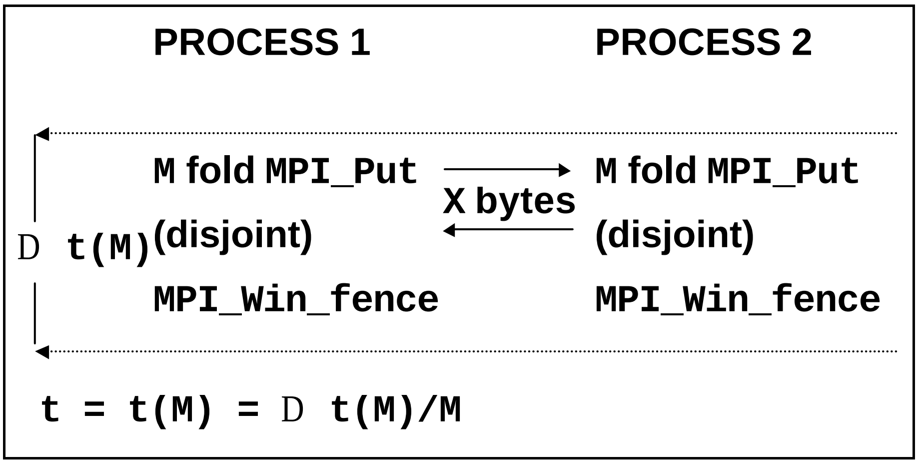

Figure 8: Bidir_Put pattern

4.2.3.4 Bidir_Get

Benchmark for the `MPI_Get` function, with bi-directional transfers.

Table 6 below shows the basic definitions. Figure 9 is a schematic view of the pattern.

measured pattern	as symbolized between ◄····· ◄····· in Figure 9; 2 active processes only
based on	MPI_Get
`MPI_Datatype`	MPI_BYTE (origin and target)
reported timings	t=t(M) (in µsec) as indicated in Figure 9, non aggregate (M=1) and aggregate (cf. 0; M=n_sample, see 5.2.8)
reported throughput	X/t, aggregate and non aggregate

Table 6: Bidir_Get definition

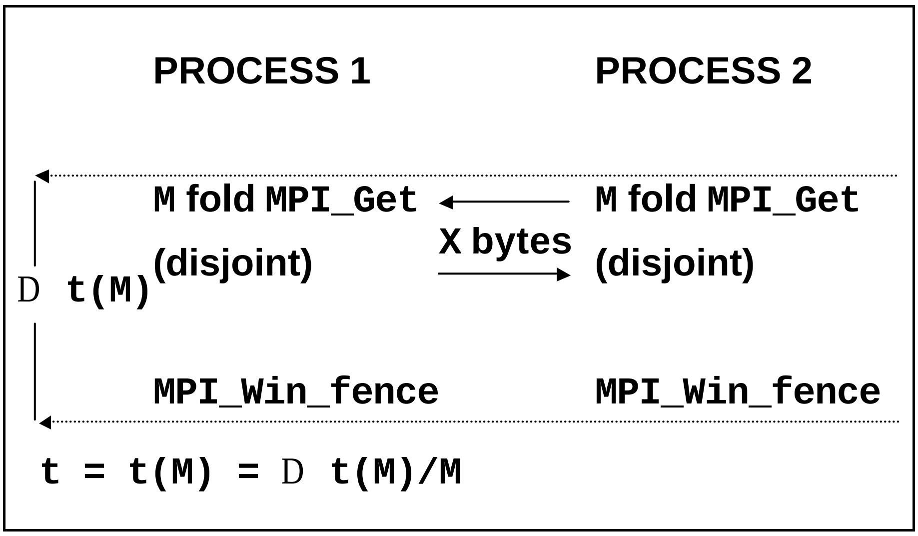

Figure 9: Bidir_Get pattern

4.2.3.5 Accumulate

Benchmark for the `MPI_Accumulate` function. It reduces a vector of length `L` = X/sizeof(float) float items. The MPI data-type is `MPI_FLOAT`, and the MPI operation is `MPI_SUM`.

Table 7 below shows the basic definitions. Figure 10 is a schematic view of the pattern.

measured pattern	as symbolized between ⟵ ⟵ in Figure 10
based on	`MPI_Accumulate`
`MPI_Datatype`	`MPI_FLOAT`
`MPI_Op`	`MPI_SUM`
Root	0
reported timings	t=t(M) (in µsec) as indicated in Figure 10, non aggregate (M=1) and aggregate (cf. 0; M=n_sample, see 5.2.8)
reported throughput	none

Table 7: Accumulate definition

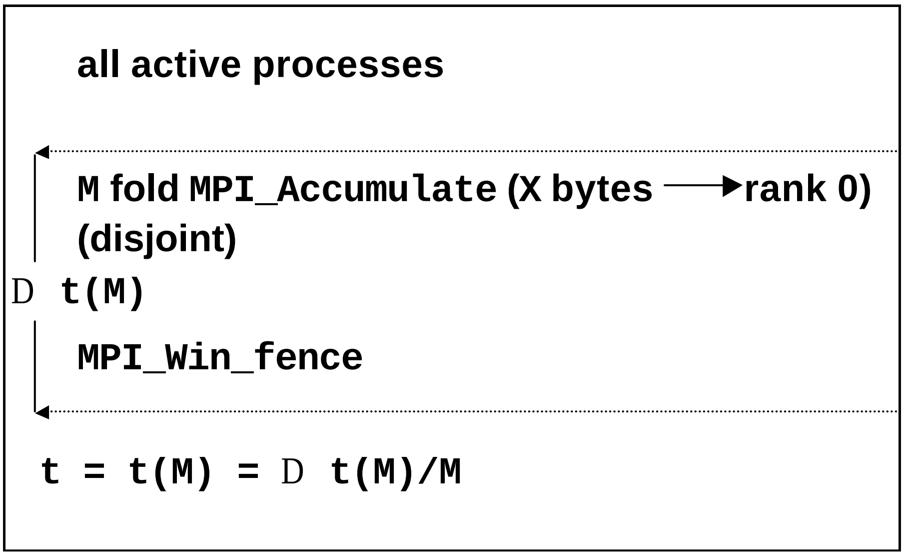

Figure 10: Accumulate pattern

4.2.3.6 Window

Benchmark measuring the overhead of an `MPI_Win_create / MPI_Win_fence / MPI_Win_free` combination. In order to prevent the implementation from optimizations in case of an unused window, a negligible non trivial action is performed inside the window. 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).

Table 8 below shows the basic definitions. Figure 11 is a schematic view of the pattern.

measured pattern	`MPI_Win_create / MPI_Win_fence / MPI_Win_free`
reported timings	t=Δt (in μsec) as indicated in Figure 11
reported throughput	none

Table 8: Window definition

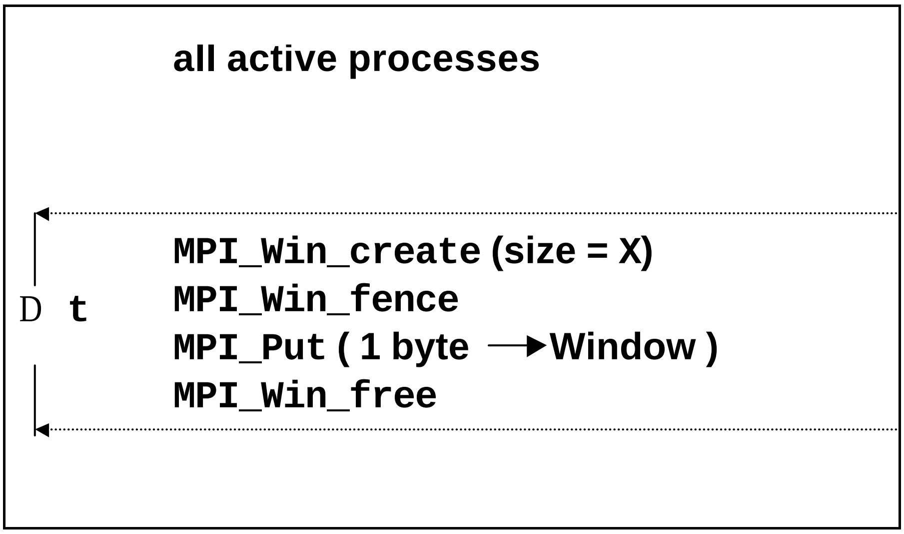

Figure 11: Window pattern

4.2.4 Definition of the IMB-IO benchmarks (blocking case)

This section describes the blocking I/O benchmarks in detail (see 4.2.5 for the non-blocking case). The benchmarks will run with varying transfer sizes `X` (in bytes), and timings are averaged over multiple samples. See section 5 for the description of the methodology. Here we describe 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. In the sequel, 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. Moreover, in the `Write` cases, a file synchronization is included, with different placements for aggregate and non aggregate modes.

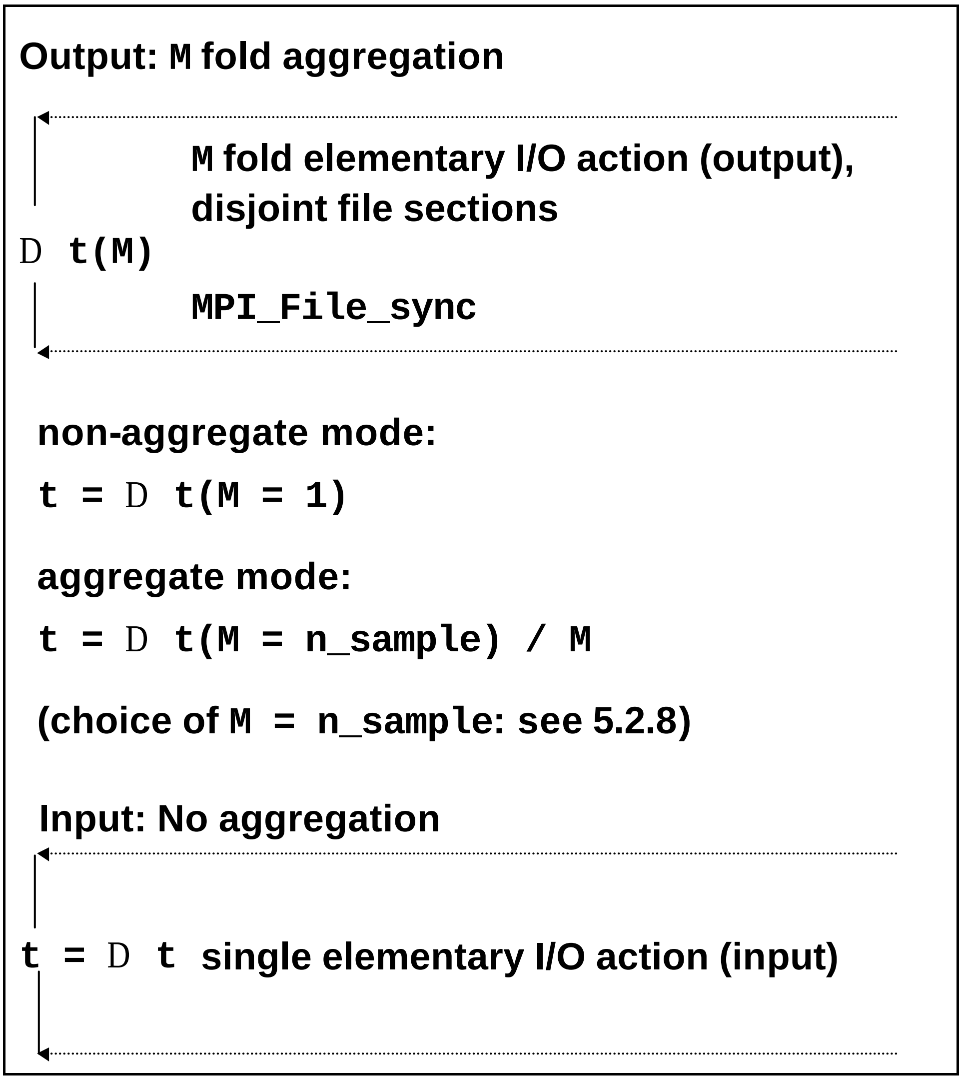

Figure 12: I/O benchmarks, aggregation for output

4.2.4.1 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.

Table 9 below shows the basic definitions. Figure 13: S_[ACTION]_indv pattern is a schematic view of the pattern.

measured pattern	as symbolized in Figure 12
elementary I/O action	as symbolized Figure 1
based on resp. for nonblocking mode	`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) as indicated in Figure 12, aggregate and non aggregate for `Write` case
reported throughput	X/t, aggregate and non aggregate for `Write` case

Table 9: S_[ACTION]_indv definition

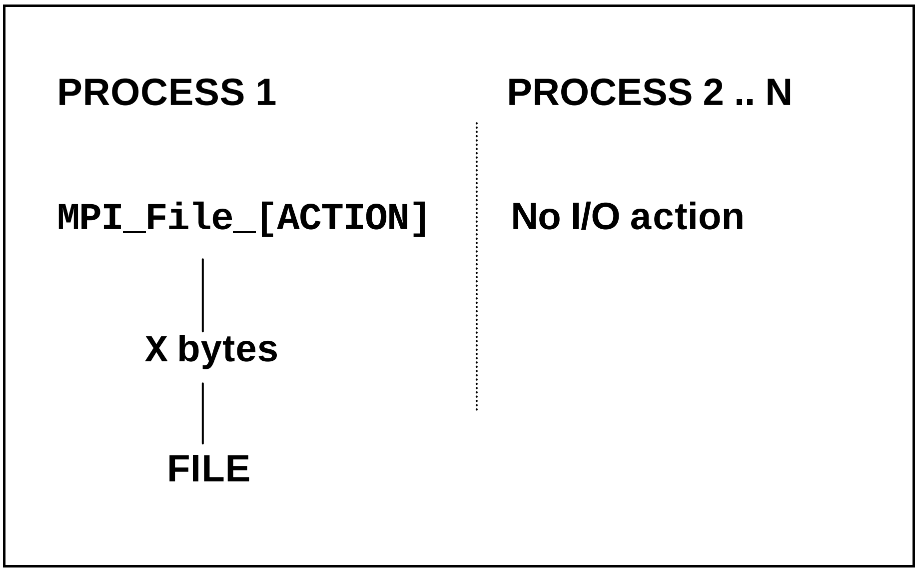

Figure 13: S_[ACTION]_indv pattern

4.2.4.2 S_[ACTION]_expl

Mimics the same situation as `S_[ACTION]_indv`, with a different strategy to access files, however.

Table 10 below shows the basic definitions. Figure 14 is a schematic view of the pattern.

measured pattern	as symbolized in Figure 12
elementary I/O action	as symbolized in Figure 14
based on resp. for nonblocking mode	`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 10: S_[ACTION]_expl definition

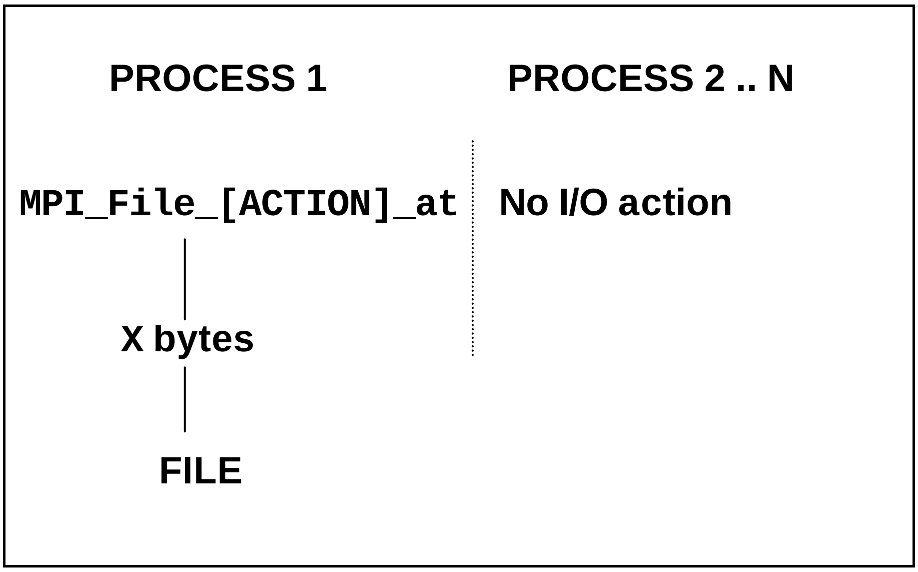

Figure 14: S_[ACTION]_expl pattern

4.2.4.3 P_[ACTION]_indv

This pattern accesses the file in a concurrent manner. All participating processes access a common file.

Table 11 below shows the basic definitions. Figure 15 is a schematic view of the pattern.

measured pattern	as symbolized in Figure 12
elementary I/O action	as symbolized in Figure 15 (Nproc = number of processes)
based on resp. for nonblocking mode	`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) as indicated in Figure 12, aggregate and non aggregate for `Write` case
reported throughput	X/t, aggregate and non aggregate for `Write` case

Table 11: P_[ACTION]_indv definition

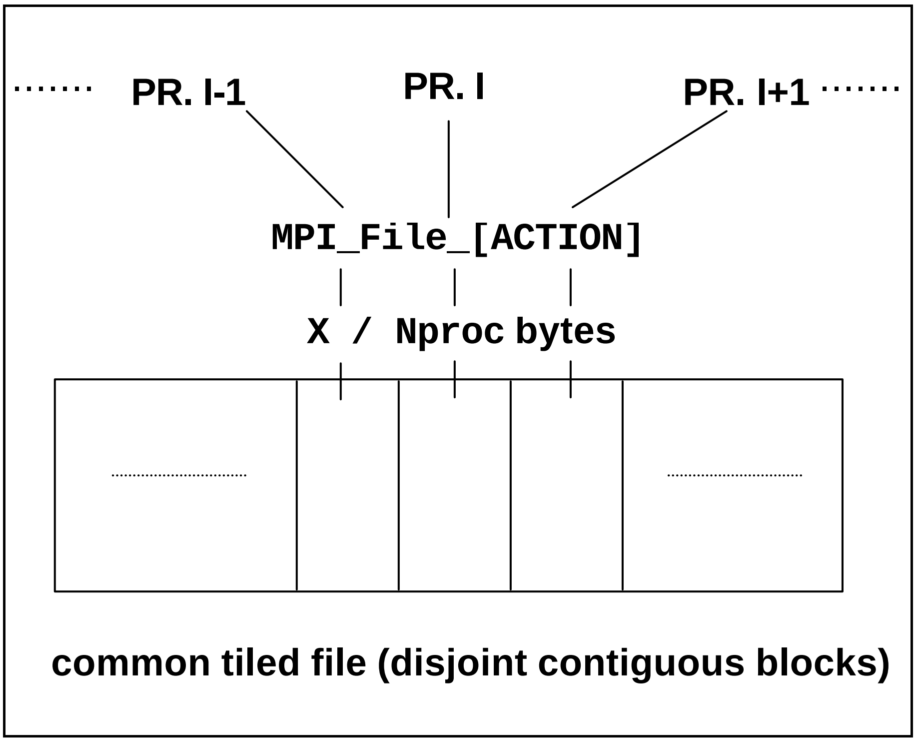

Figure 15: P_[ACTION]_indv pattern

4.2.4.4 P_[ACTION]_expl

`P_[ACTION]_expl` follows the same access pattern as `P_[ACTION]_indv`, with an explicit file pointer type, however.

Table 12 below shows the basic definitions. Figure 16 is 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)
based on resp. for nonblocking mode	`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 12: P_[ACTION]_expl definition

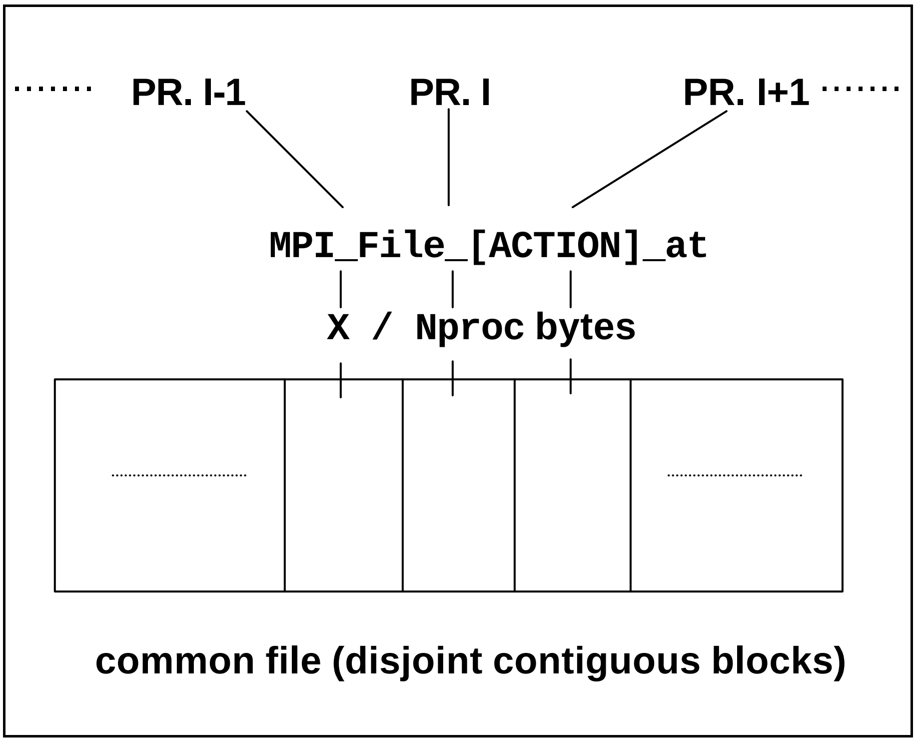

Figure 16: P_[ACTION]_expl pattern

4.2.4.5 P_[ACTION]_shared

Concurrent access to a common file by all participating processes, with a shared file pointer.

Table 13 below shows the basic definitions. Figure 17 is a schematic view of the pattern.

measured pattern	as symbolized in Figure 12
elementary I/O action	as symbolized in Figure 17 (`Nproc` = number of processes)

based on
resp. for nonblocking mode
	`MPI_File_write_shared` /
	`MPI_File_read_shared`
	`MPI_File_iwrite_shared` /
	`MPI_File_iread_shared`

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 13: P_[ACTION]_shared definition

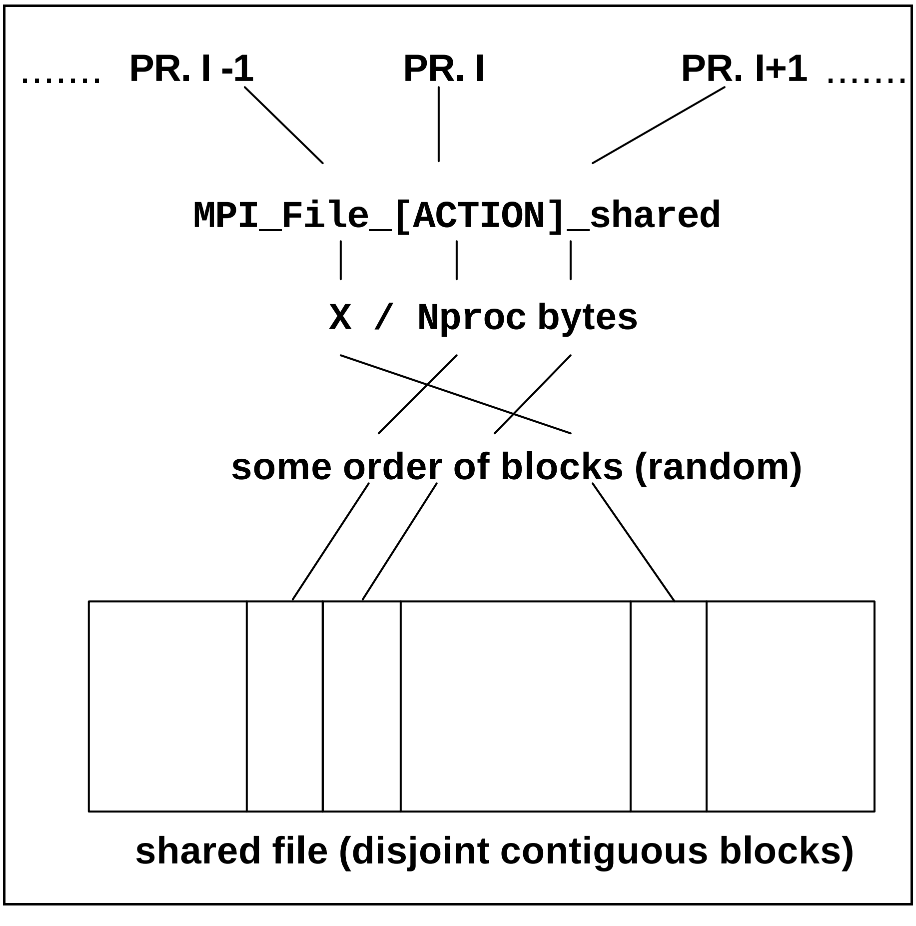

Figure 17: P_[ACTION]_shared pattern

4.2.4.6 P_[ACTION]_priv

This pattern tests the (very important) case that all participating processes perform concurrent I/O, however to different (private) files. It is of particular interest for systems allowing completely independent I/O from different processes. In this case, this pattern should show parallel scaling and optimum results.

Table 14 below shows the basic definitions. Figure 18 is 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)
based on resp. for nonblocking mode	`MPI_File_write / MPI_File_read` `MPI_File_iwrite / MPI_File_iread`
etype	`MPI_BYTE`
filetype	`MPI_BYTE`
`MPI_Datatype`	`MPI_BYTE`
reported timings	Δt (in μsec) as indicated in Figure 12, aggregate and non aggregate for `Write` case
reported throughput	X/Δt, aggregate and non aggregate for `Write` case

Table 14: P_[ACTION]_priv definition

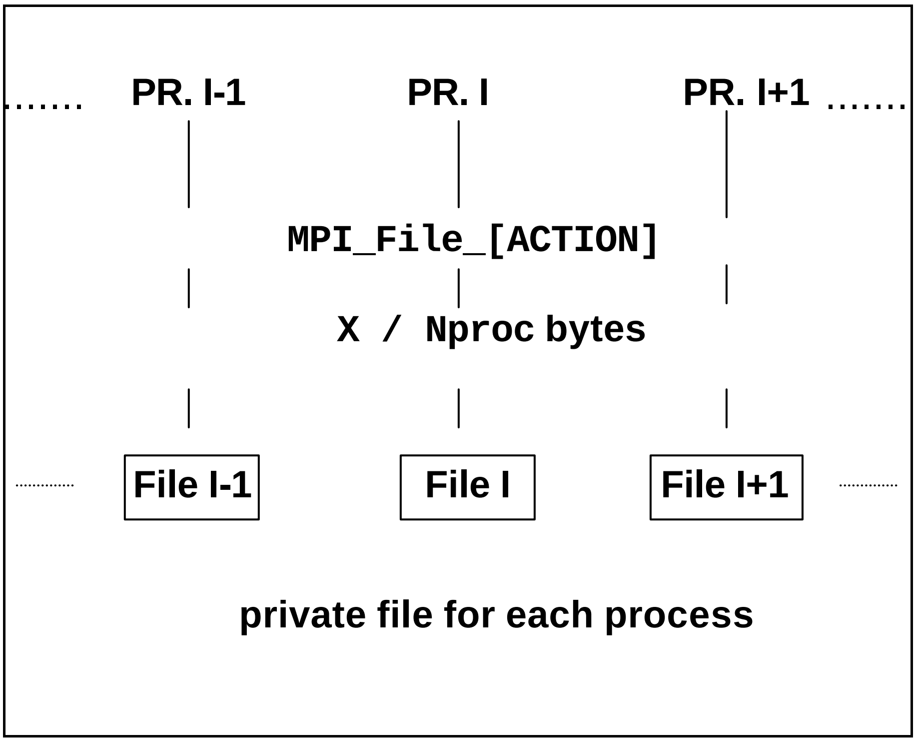

Figure 18: P_[ACTION]_priv pattern

4.2.4.7 C_[ACTION]_indv

C_[ACTION]_indv tests collective access from all processes to a common file, with an individual file pointer.

Table 15 below shows the basic definitions, and a schematic view of the pattern is shown in Figure 15.

based on resp. for nonblocking mode	MPI_File_read_all / MPI_File_write_all MPI_File_..._all_begin - MPI_File_..._all_end
all other parameters, measuring method	see 4.2.4.3

Table 15: C_[ACTION]_indv definition

4.2.4.8 C_[ACTION]_expl

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

Table 16 below shows the basic definitions, and a schematic view of the pattern is shown in Figure 16.

based on resp. for nonblocking mode	MPI_File_read_at_all / MPI_File_write_at_all MPI_File_..._at_all_begin - MPI_File_..._at_all_end
all other parameters, measuring method	see 4.2.4.4

Table 16: C_[ACTION]_expl definition

4.2.4.9 C_[ACTION]_shared

Finally, here a collective access from all processes to a common file, with a shared file pointer is benchmarked.

Table 17 below shows the basic definitions, and a schematic view of the pattern is shown in Figure 17, with the crucial difference that here the order of blocks is preserved.

based on resp. for nonblocking mode	MPI_File_read_ordered / MPI_File_write_ordered MPI_File_..._ordered_begin- MPI_File_..._ordered_end
all other parameters, measuring method	see 4.2.4.5

Table 17: C_[ACTION]_shared definition

4.2.4.10 Open_Close

Benchmark of an `MPI_File_open / MPI_File_close` pair. All processes open the same file. In order to prevent the implementation from optimizations in case of an unused file, a negligible non trivial action is performed with the file, see Figure 19. Table 18 below shows the basic definitions.

measured pattern	`MPI_File_open / MPI_File_close`
etype	`MPI_BYTE`
filetype	`MPI_BYTE`
reported timings	t=Δt (in μsec) as indicated in Figure 19
reported throughput	none

Table 18: Open_Close definition

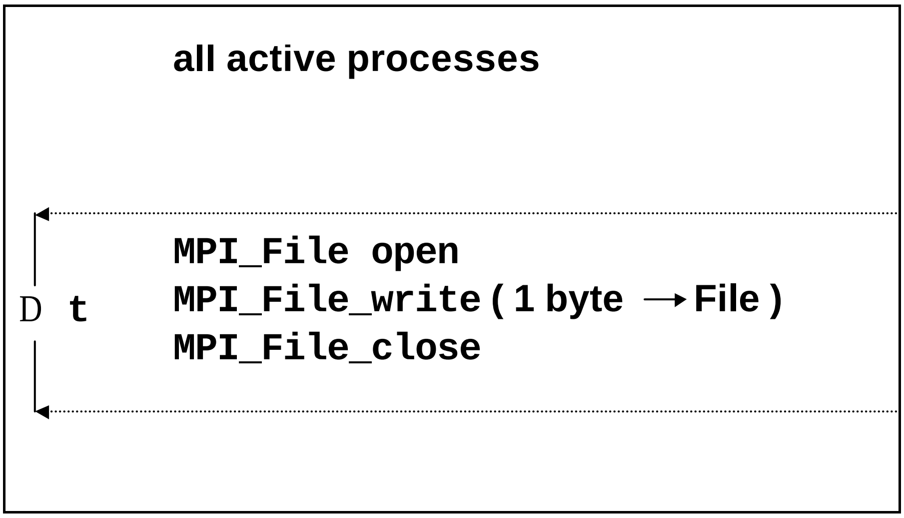

Figure 19: Open_Close pattern

4.2.5 Non-blocking I/O Benchmarks

Each of the non-blocking benchmarks (see Table 1) has a blocking equivalent explained in section 4.2.4. 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 IMB 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.

4.2.5.1 Exploiting CPU

IMB uses the following method to exploit the CPU. A kernel loop is executed repeatedly. The kernel is a fully vectorizable multiply of a 100 $\times$ 100 matrix with a vector. The function is scaleable 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.

4.2.5.2 Displaying results

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

- t_pure = time for the corresponding pure blocking I/O action, non overlapping with CPU activity
- t_CPU = time the CPU_Exploit periods (running concurrently with non-blocking I/O) would use when running dedicated
- t_ovrl = 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) (*)

IMB 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.

4.2.6 Multi - versions

The definition and interpretation of the `Multi-` prefix is analogous to the definition in the MPI1 section (see 3.2).

5 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 IMB. 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. For instance, this mode can be used to extend the results tables as to larger transfer sizes.

The following graph shows the flow of control inside IMB. All *emphasized* items will be explained in more detail.

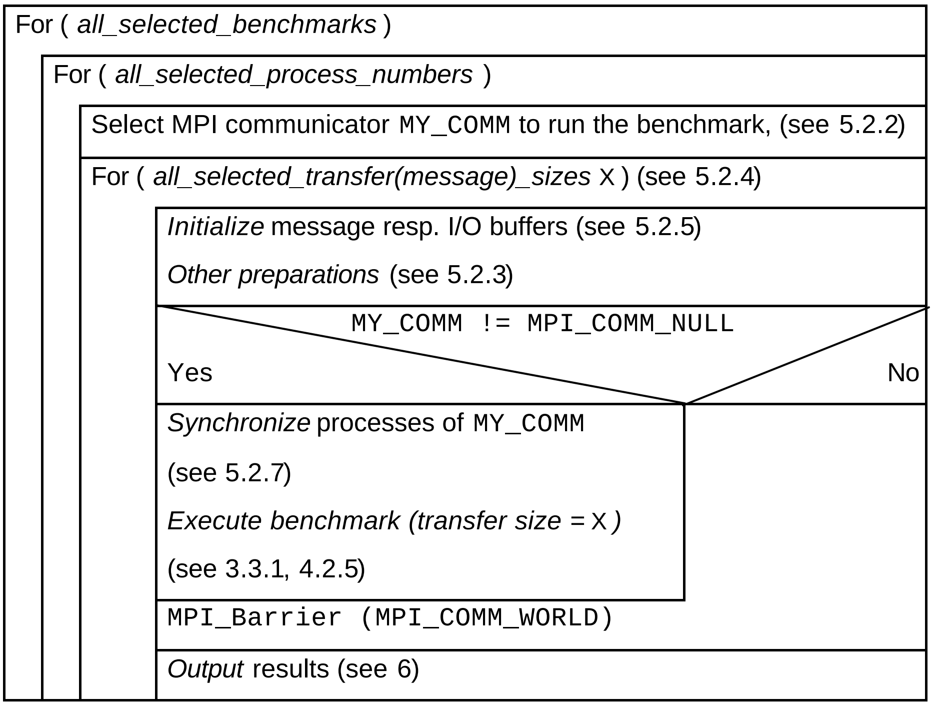

Figure 20: Control flow of IMB

The control parameters that are obviously necessary are either *command line arguments* (see 5.1.2) or parameter selections inside the IMB include files `settings.h / settting_io.h` (see 5.2).

5.1 Running IMB, command line control

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

Given P**,** the (normally user selected) number of MPI processes to run IMB, a startup procedure has to load parallel IMB. Lets assume, for sake of simplicity, that this done by

```
mpirun -np P IMB-<..> [arguments]
```

P=1 is allowed and sensible for all IO and (if you like) also for all message passing benchmarks except the Single Transfer ones. Control arguments (in addition to P) can be passed to IMB via (`argc,argv`). Command line arguments are only read by process 0 in `MPI_COMM_WORLD`. However, the command line options are broadcast to all other processes.

5.1.1 Default case

Just invoke

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

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

The Q processes driving the benchmark are called the *active processes*.

5.1.2 Command line control

The command line will be repeated in the Output (new in IMB 3.1). 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>]
            [benchmark1 [,benchmark2 [,...]]]
```

(where the 11 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
```

5.1.2.1 Benchmark selection arguments

A sequence of blank-separated strings, each being the name of one IMB-<..> benchmark (in exact spelling, case insensitive). The benchmark names are listed in Table 1.

Default (no benchmark selection): select all benchmarks.

5.1.2.2 `-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): see 5.1.1.

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

5.1.2.3 `-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)

Note:
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 1 group can be created.

Default:

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

5.1.2.4 `-off_cache` cache_size[,cache_line_size] selection

The argument after off_cache can be either 1 single number (cache_size), or 2 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; most likely, cache reusage 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 (Bytes) 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 (eventually), will they then will be re-used from the beginning.

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

Remark: -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);

NOTE: 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

5.1.2.5 `–iter`

The argument after -iter can be 1 single, 2 comma separated, or 3 comma separated integer numbers, which override the defaults

MSGSPERSAMPLE, OVERALL_VOL, MSGS_NONAGGR of =>IMB_settings.h (Table 19)

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 (Table 19).

NOTE: !! New in versions from IMB 3.2 on !!

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

5.1.2.6 `-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 (variable SECS_PER_SAMPLE, current value 10)

5.1.2.7 -mem

The argument after `-mem` is a float, specifying that at most that many GBytes 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

5.1.2.8 -input <File> selection

An ASCII input file is used to select the benchmarks to run, for example a file `IMB_SELECT_EXT` looking as follows:

```
#
# 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 IMB-EXT benchmarks `Unidir_Get` and `Accumulate`.

5.1.2.9 –msglen <File> selection

Enter any set of nonnegative message lengths to an ASCII file, line by line. Call it, for example, “Lengths” and call IMB with arguments:

```
-msglen Lengths
```

This lengths value then overrides the default message lengths (see 5.2.4). For `IMB-IO`, the file defines the I/O portion lengths.

5.1.2.10 `–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, in order 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
```

5.2 IMB parameters and hard-coded settings

5.2.1 Parameters controlling IMB

There are 9 parameters (set by preprocessor definition) controlling default IMB (note, however, that `MSGSPERSAMPLE`, `MSGS_NONAGGR`, `OVERALL_VOL` can be overridden by the `–iter`, `-time`, `-mem` 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` is in Table 19 below.

Both include files are almost identical in structure, but differ in the standard settings. Note that some names in IMB_`settings_io.h` contain `MSG` (for “message“), in consistency with IMB_`settings.h`.

Parameter (standard mode value)	Meaning
IMB_OPTIONAL (not set)	has to be set when optional settings are to be activated
MINMSGLOG (0)	second smallest data transfer size is max(unit,$2^{MINMSGLOG}$) (the smallest always being 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 eventually 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 (5.2.7)
TARGET_CPU_SECS (0.01)	CPU seconds (as float) to run concurrent with non-blocking benchmarks (currently irrelevant for IMB-MPI1)

Table 19: IMB (MPI1/EXT) parameters (settings.h)

IMB allows for two sets of parameters: *standard* and *optional*.

Below a sample of file settings_io.h is shown. Here, IMB_OPTIONAL is set, so that user defined parameters are used. I/O sizes 32 and 64 Mbytes (and a smaller repetition count) are selected, extending the standard mode tables.

If IMB_OPTIONAL is deactivated, the obvious standard mode values are taken.

Remark:
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
```

5.2.2 Communicators, active processes

Communicator management is repeated in every “select `MY_COMM`“ step in Figure 20. 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` in Figure 20.

The group of MY_COMM is called the *active processes* group.

5.2.3 Other preparations

5.2.3.1 Window (IMB_EXT)

An Info is set (see section 5.2.3.3) 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.

5.2.3.2 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`,
  `MY_COMM` as selected in 5.2.2 above else.
- selecting a mode = MPI_MODE_CREATE | MPI_MODE_RDWR
- selecting an info, see 5.2.3.3

5.2.3.3 Info

IMB uses an external function `User_Set_Info` which *you are allowed to implement at best 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
```

IMB uses no assumptions and imposes no restrictions on how this routine will be implemented.

5.2.3.4 View (IMB-IO)

The file view is determined by the settings:

- disp = 0
- datarep = native
- etype, filetype as defined in the single definitions in section 0
- info as defined in 5.2.3.3

5.2.4 Message / I-O buffer lengths

5.2.4.1 IMB-MPI1, IMB-EXT

Set in `settings.h` (see 5.2.1), used unless `–msglen` flag is selected (ref. 5.1.2.9).

5.2.4.2 IMB-IO

Set in `settings_io.h` (see 5.2.1), and is used unless `–msglen` flag is selected (ref. 5.1.2.9).

5.2.5 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 eventually (see 7.2).

IMB sets the buffer assignment type by `typedef assign_type` in `settings.h/settings_io.h`

Currently, `int` is used for IMB-IO, `float` for IMB-EXT (as this is sensible for `Accumulate`). 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.

5.2.6 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.

5.2.7 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)
```

(ref. Figure 20) so as to assure that all processes are synchronized.

5.2.8 The actual benchmark

In order 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). In order 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 Mbytes). 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.

5.2.8.1 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
```

5.2.8.2 EXT and blocking I/O case

For the aggregate case, the kernel loop looks like:

```
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
```

5.2.8.3 Non-blocking I/O case

As explained in section 4.2.5, 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). Thus, the actual benchmark consists of

- Calling the equivalent blocking benchmark as defined in 5.2.8 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 */
```

6 Output

The output results are most easily explained by sample outputs, and therefore you should examine the tables below. What you would see is the following:

- *General information*
  Machine, System, Release, and, Version are obtained by the code `IMB_g_info.c`.

- (New in IMB 3.1)
  The calling sequence (command line flags) are repeated in the output chart.

- *Non multi case numbers*
  After a benchmark completes, 3 time values are available: `Tmax`, `Tmin`, `Tavg`, the maximum, minimum and average time, respectively, extended over the group of active processes. The time unit is `μsec`.
  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
  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.

6.1 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
```

6.2 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, 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
```

6.3 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
```

6.4 Sample 4 – IMB-EXT.exe

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

#---------------------------------------------------
#    Intel (R) MPI Benchmark Suite V3.2, 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 2 processes. Note that the listing shows only the result for the "Window" benchmark. The performance diagnostics for "Unidir_Get", "Unidir_Put", "Bidir_Get" "Bidir_Put", and "Accumulate" have been omitted.

7 Further details

7.1 Memory requirements

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

Table 20: Memory requirements with standard settings

7.2 Results checking

By activating the `cpp` flag `–DCHECK` through the `CPPFLAGS` variable (see section 2.1), and recompiling, every message passing result from the IMB executables will be checked against the expected outcome (note that the contents of each buffer is well defined, see section 5.2.5). Output tables will contain an additional column displaying the diffs as floats (named *defects*).

Attention: `-DCHECK` results are not valid as real benchmark data! Do not forget to deactivate DCHECK and recompile in order to get proper results.

8 Revision History

Release No.	Date	
2.3	Nov. 2004	Describes the initial version IMB, derived from PMB (Pallas MPI Benchmarks)
3.0	June 2006	Descriptions added of environment amendments, new `Alltoallv` benchmark
3.1	July 2007	Description added of: Windows version; 4 new benchmarks (`Scatter(v), Gather(v)`); IMB-IO functional fix
3.2	August 2008	Run time control as default Microsoft* Visual Studio* solution templates included