



Intel ® MPI Benchmarks

Users Guide and Methodolgy Description

Intel GmbH
Hermülheimer Str. 8a
D-50321 Brühl, Germany

1	INTRODUCTION	4
2	INSTALLATION AND QUICK START OF IMB.....	4
2.1	Installing and running.....	4
3	IMB-MPI1	5
3.1	General	5
3.2	The benchmarks.....	6
3.3	IMB-MPI1 benchmark definitions.....	7
3.3.1	Benchmark classification	7
3.3.1.1	Single Transfer benchmarks.....	7
3.3.1.2	Parallel Transfer benchmarks.....	7
3.3.1.3	Collective benchmarks.....	8
3.3.2	Definition of Single Transfer benchmarks.....	8
3.3.2.1	PingPong	8
3.3.2.2	PingPing.....	9
3.3.3	Definition of Parallel Transfer benchmarks.....	10
3.3.3.1	Sendrecv.....	10
3.3.3.2	Exchange.....	11
3.3.4	Definition of Collective benchmarks.....	11
3.3.4.1	Reduce.....	12
3.3.4.2	Reduce_scatter.....	12
3.3.4.3	Allreduce	12
3.3.4.4	Allgather.....	13
3.3.4.5	Allgatherv	13
3.3.4.6	Alltoall.....	13
3.3.4.7	Bcast.....	13
3.3.4.8	Barrier	14
4	MPI-2 PART OF IMB.....	15
4.1	The benchmarks.....	15
4.2	IMB-MPI2 benchmark definitions.....	16
4.2.1	Benchmark classification	16
4.2.1.1	Single Transfer benchmarks.....	17
4.2.1.2	Parallel Transfer benchmarks.....	17
4.2.1.3	Collective benchmarks.....	17
4.2.2	Benchmark modes	18
4.2.2.1	Blocking / nonblocking mode (only IMB-IO)	18
4.2.2.2	Aggregate / Non Aggregate mode.....	18
4.2.3	Definition of the IMB-EXT benchmarks.....	19
4.2.3.1	Unidir_Put	19
4.2.3.2	Unidir_Get.....	20
4.2.3.3	Bidir_Put.....	21
4.2.3.4	Bidir_Get.....	22
4.2.3.5	Accumulate	23
4.2.3.6	Window	23
4.2.4	Definition of the IMB-IO benchmarks (blocking case).....	25
4.2.4.1	S_[ACTION]_indv	26
4.2.4.2	S_[ACTION]_expl	27
4.2.4.3	P_[ACTION]_indv	28
4.2.4.4	P_[ACTION]_expl	29
4.2.4.5	P_[ACTION]_shared.....	30
4.2.4.6	P_[ACTION]_priv	31
4.2.4.7	C_[ACTION]_indv	32
4.2.4.8	C_[ACTION]_expl	32

4.2.4.9	C_[ACTION]_shared.....	32
4.2.4.10	Open_Close	33
4.2.5	Nonblocking I/O Benchmarks	33
4.2.5.1	Exploiting CPU	34
4.2.5.2	Displaying results	34
4.2.6	Multi - versions	35
5	BENCHMARK METHODOLOGY	35
5.1	Running IMB, command line control.....	36
5.1.1	Default case	36
5.1.2	Command line control	36
5.1.2.1	Benchmark selection arguments	36
5.1.2.2	-npmin selection.....	37
5.1.2.3	-multi <outflag> selection.....	37
5.1.2.4	-input <File> selection.....	37
5.1.2.5	-msglen <File> selection	37
5.1.2.6	-map PxQ selection.....	38
5.2	IMB parameters and hard-coded settings	38
5.2.1	Parameters controlling IMB.....	38
5.2.2	Communicators, active processes	40
5.2.3	Other preparations	40
5.2.3.1	Window (IMB_EXT)	40
5.2.3.2	File (IMB-IO)	40
5.2.3.3	Info	40
5.2.3.4	View (IMB-IO)	41
5.2.4	Message / I-O buffer lengths.....	41
5.2.4.1	IMB-MPI1, IMB-EXT	41
5.2.4.2	IMB-IO.....	41
5.2.5	Buffer initialization	41
5.2.6	Warm-up phase (MPI1, EXT).....	42
5.2.7	Synchronization.....	42
5.2.8	The actual benchmark.....	42
5.2.8.1	MPI1 case	42
5.2.8.2	EXT and blocking I/O case	43
5.2.8.3	Nonblocking I/O case.....	43
6	OUTPUT.....	43
6.1	Sample 1.....	45
6.2	Sample 2.....	46
6.3	Sample 3.....	47
7	FURTHER DETAILS	49
7.1	Memory requirements.....	49
7.2	Results checking.....	49
8	REVISION HISTORY.....	49

Disclaimer and Legal Information

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications.

This document as well as the software described in it, is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The software described in this document may contain software defects which may cause the product to deviate from published specifications. Current characterized software defects are available on request.

Copyright © Intel Corporation 1996 - 2004.

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.
- don't impose much of an interpretation on the measured results: report bare timings instead. Show throughput values, if and only if these are well defined.

The package is the successor of the quite well known package PMB (Version 2.2) from Pallas GmbH.

This document accompanies the version 2.3 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 2.3 package consists of 3 parts:

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

For each part, a separate executable can be built. Users who don't have the MPI-2 extensions available, 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*, *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, 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.

2 Installation and Quick Start of IMB

In order to run IMB-MPI1, one needs:

- `cpp`, ANSI C compiler, `make`.
- MPI installation, including 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

File ReadMe_first

and 5 subdirectories

./doc (ReadMe_IMB.txt; IMB.pdf, this file)

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

./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.

3 IMB-MPI1

This sections 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 5.1.2.5).

The minimum P_{min} and maximum number P of processes can be selected by the user via command line, the benchmarks run on P_{min} , $2P_{min}$, $4P_{min}$, ... $2 * P_{min} < P$ and P processes. See chapter 5.1.2.2 for the details.

The user has some choice for the mapping of processes. E.g., when running on a clustered system, a benchmark, lets take `PingPong`, can be run intra node and inter node, without changing a mapping file (`-map` flag, see 5.1.2.6)

3.2 The benchmarks

The current version of IMB-MPI1 contains the benchmarks

- PingPong
- PingPing
- Sendrecv
- Exchange
- Bcast
- Allgather
- Allgatherv
- Alltoall
- 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. E.g., when running PingPong on $N \geq 4$ processes, on user request (see 5.1.2.3) $N/2$ disjoint groups of 2 processes each will 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-Alltoall
- 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	Reduce
	Multi-Sendrecv	Reduce_scatter
	Multi-Exchange	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 to `PingPong`, this is the usual way of looking at. In 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 and 0 for exact definitions.

3.3.1.2 Parallel Transfer benchmarks

Benchmarks focusing on *global mode*, say, patterns. The activity at a certain process is in concurrency with other processes, 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. E.g., 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 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. Basic MPI datatype for all messages is `MPI_BYTE`.

Throughput values are defined in $\text{MBytes} / \text{sec} = 2^{20} \text{ bytes} / \text{sec}$ scale (i.e. $\text{throughput} = X / 2^{20} * 10^6 / \text{time} = X / 1.048576 / \text{time}$, 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	<code>MPI_Send</code> , <code>MPI_Recv</code>
<code>MPI_Datatype</code>	<code>MPI_BYTE</code>
reported timings	$\text{time} = \Delta t / 2$ (in μsec) as indicated in Figure 1
reported throughput	$X / 1.048576 / \text{time}$

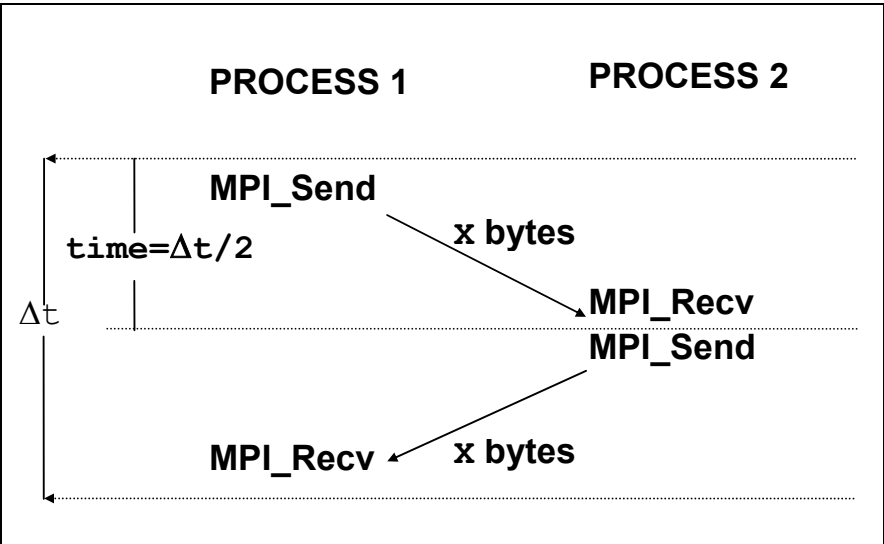



Figure 1: PingPong pattern

3.3.2.2 PingPing

As PingPong, PingPing measures 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	$X / 1.048576 / \text{time}$

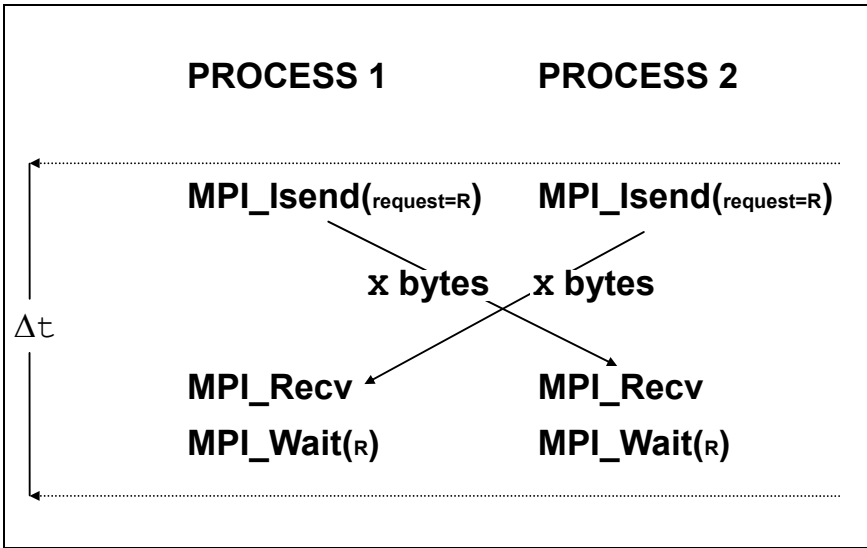


Figure 2: PingPing pattern

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 x bytes, and timings are averaged over multiple samples. See 5 for the description of the methodology. Here we describe the view of one single sample, with a fixed message length x bytes. Basic MPI datatype for all messages is `MPI_BYTE`.

The throughput calculations of the benchmarks described here take into account the (per sample) multiplicity n_{msg} 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, $n_{msg}=2$. In the `Exchange` case, we have $4x$ bytes turnover, $n_{msg}=4$.

Throughput values are defined in $MBytes/sec = 2^{20} \text{ bytes} / \text{sec scale}$ (i.e.


$\text{throughput} = n_{msg} * x / 2^{20} * 10^6 / \text{time} = n_{msg} * x / 1.048576 / \text{time}$, 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 based on	As symbolized between  in Figure 3
MPI_Datatype	<code>MPI_BYTE</code>
reported timings	$\text{time} = \Delta t$ (in μsec) as indicated in Figure 3
reported throughput	$2x / 1.048576 / \text{time}$

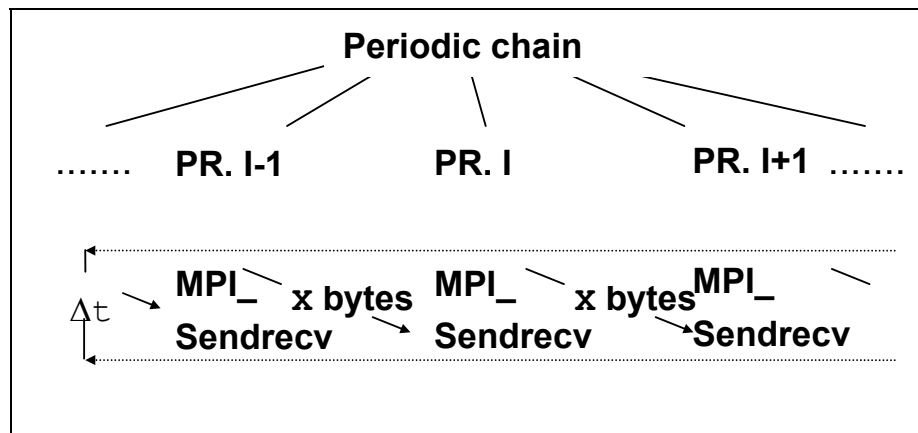



Figure 3: `Sendrecv` pattern

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.

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

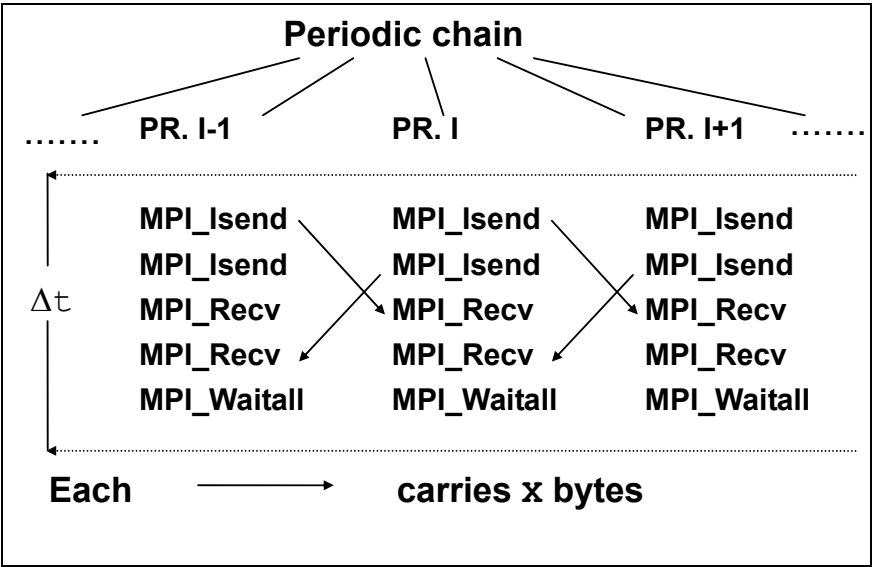


Figure 4: Exchange pattern

3.3.4 Definition of Collective benchmarks

This section describes the Collective benchmarks in detail. Each benchmark is run with varying message lengths x bytes, and timings are averaged over multiple samples. See 5 for the description of the methodology. Here we describe the view of one single sample, with a fixed message length x bytes. Basic MPI datatype 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 of the `MPI_Reduce` function. Reduces a vector of length $L = X/\text{sizeof}(\text{float})$ float items. The MPI datatype is `MPI_FLOAT`, the MPI operation is `MPI_SUM`.

The root of the operation is changed cyclically.

See also the remark in the end of 3.3.1.3.

measured pattern	<code>MPI_Reduce</code>
<code>MPI_Datatype</code>	<code>MPI_FLOAT</code>
<code>MPI_Op</code>	<code>MPI_SUM</code>
root	changing
reported timings	bare time
reported throughput	none

3.3.4.2 Reduce_scatter

Benchmark of the `MPI_Reduce_scatter` function. Reduces a vector of length $L = X/\text{sizeof}(\text{float})$ float items. The MPI datatype is `MPI_FLOAT`, the MPI operation is `MPI_SUM`. In the scatter phase, the L items are split as evenly as possible. Exactly, when

$np = \text{\#processes}$, $L = r*np + s$ ($s = L \bmod np$),

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

See also the remark in the end of 3.3.1.3.

measured pattern	<code>MPI_Reduce_scatter</code>
<code>MPI_Datatype</code>	<code>MPI_FLOAT</code>
<code>MPI_Op</code>	<code>MPI_SUM</code>
reported timings	bare time
reported throughput	none

3.3.4.3 Allreduce

Benchmark of the `MPI_Allreduce` function. Reduces a vector of length $L = X/\text{sizeof}(\text{float})$ float items. The MPI datatype is `MPI_FLOAT`, the MPI operation is `MPI_SUM`.

See also the remark in the end of 3.3.1.3.

measured pattern	<code>MPI_Allreduce</code>
<code>MPI_Datatype</code>	<code>MPI_FLOAT</code>
<code>MPI_Op</code>	<code>MPI_SUM</code>
reported timings	bare time
reported throughput	none

3.3.4.4 Allgather

Benchmark of the `MPI_Allgather` function. Every process inputs x bytes and receives the gathered $x * (\#processes)$ bytes.

Measured pattern	<code>MPI_Allgather</code>
<code>MPI_Datatype</code>	<code>MPI_BYTE</code>
reported timings	bare time
reported throughput	none

3.3.4.5 Allgatherv

Functionally the same as `Allgather`, however with the `MPI_Allgatherv` function. Shows whether MPI produces overhead due to the more complicated situation as compared to `MPI_Allgather`.

Measured pattern	<code>MPI_Allgatherv</code>
<code>MPI_Datatype</code>	<code>MPI_BYTE</code>
reported timings	bare time
reported throughput	none

3.3.4.6 Alltoall

Benchmark of 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	<code>MPI_Alltoall</code>
<code>MPI_Datatype</code>	<code>MPI_BYTE</code>
reported timings	bare time
reported throughput	none

3.3.4.7 Bcast

Benchmark of `MPI_Bcast`. A root process broadcasts x bytes to all.

The root of the operation is changed cyclically.

measured pattern	<code>MPI_Bcast</code>
<code>MPI_Datatype</code>	<code>MPI_BYTE</code>
root	Changing
reported timings	bare time
reported throughput	None

3.3.4.8 Barrier

measured pattern	MPI_Barrier
reported timings	bare time
reported throughput	none

4 MPI-2 part of IMB

This section the MPI-2 sections 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 *Nonblocking Mode* column. Section 5 describes the benchmark methodology.

The nonblocking 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	Nonblocking Mode
IMB-EXT		
Window		
Unidir_Put	x	
Unidir_Get	x	
Bidir_Get	x	
Bidir_Put	x	
Accumulate	x	
Multi- versions of the above	x	
Benchmark	Aggregate Mode	Nonblocking Mode
IMB-IO		
Open_Close		
S_Write_indv	x	S_IWrite_indv
S_Read_indv		S_IRead_indv
S_Write_expl	x	S_IWrite_expl
S_Read_expl		S_IRead_expl
P_Write_indv	x	P_IWrite_indv
P_Read_indv		P_IRead_indv
P_Write_expl	x	P_IWrite_expl
P_Read_expl		P_IRead_expl
P_Write_shared	x	P_IWrite_shared
P_Read_shared		P_IRead_shared
P_Write_priv	x	P_IWrite_priv
P_Read_priv		P_IRead_priv
C_Write_indv	x	C_IWrite_indv
C_Read_indv		C_IRead_indv
C_Write_expl	x	C_IWrite_expl
C_Read_expl		C_IRead_expl
C_Write_shared	x	C_IWrite_shared
C_Read_shared		C_IRead_shared
Multi-versions of the above	(x)	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	Multi-Unidir_Get	Accumulate	Window (also Multi)
Unidir_Put	Multi-Unidir_Put	Multi-Accumulate	
Bidir_Get	Multi-Bidir_Get		
Bidir_Put	Multi-Bidir_Put		
S_[I]Write_indv	P_[I]Write_indv	C_[I]Write_indv	Open_close (also Multi)
S_[I]Read_indv	P_[I]Read_indv	C_[I]Read_indv	
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		

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 / nonblocking mode (only IMB-IO)

This distinction is in the proper MPI-IO sense. Blocking and nonblocking 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 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.
- *assure completion* means
`MPI_Win_fence` (IMB-EXT),
`MPI_File_sync` (IMB-IO `Write`).

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. E.g., 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 (e.g., 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`, resp.). Their interpretation and output is analogous to their message passing equivalents.

4.2.3.1 Unidir_Put

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


measured pattern	as symbolized between  in Figure 6; 2 active processes only
based on	<code>MPI_Put</code>
<code>MPI_Datatype</code>	<code>MPI_BYTE</code> (origin and target)
reported timings	$t = t(M)$ (in μsec) as indicated in Figure 6, non aggregate ($M=1$) and aggregate (cf. 4.2.2.2; $M=n_{\text{sample}}$, see 5.2.8)
reported throughput	X/t , aggregate and non aggregate

Table 3 : `Unidir_Put` definition

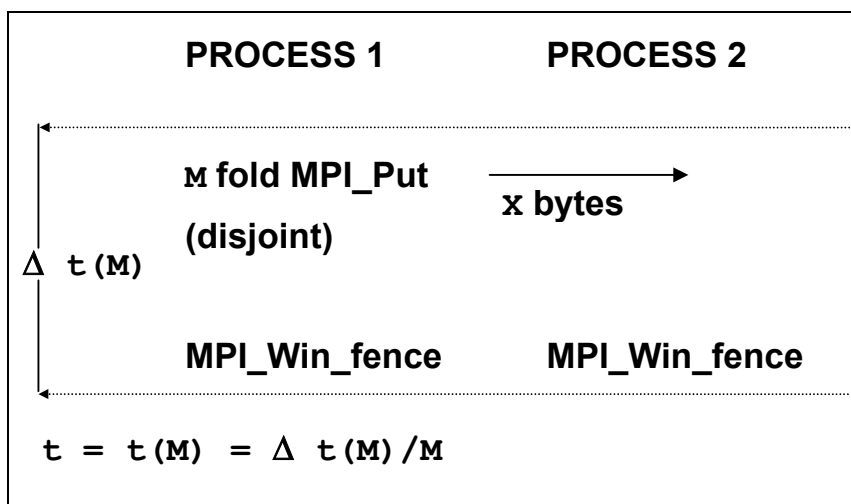


Figure 6: Unidir_Put pattern

4.2.3.2 Unidir_Get

Benchmark of the MPI_Get function.

Table 4 below shows the basic definitions, Figure 7 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. 4.2.2.2; $M=n_{\text{sample}}$, see 5.2.8)
reported throughput	X/t , aggregate and non aggregate

Table 4: Unidir_Get definition

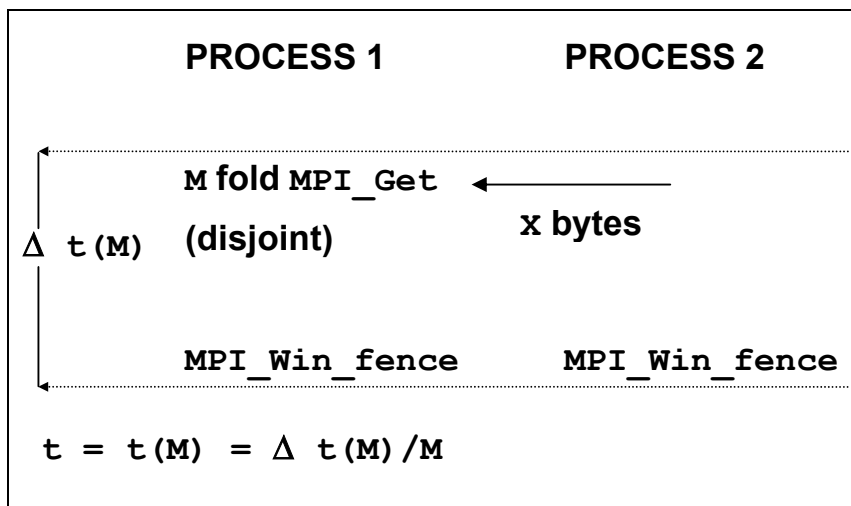


Figure 7: Unidir_Get pattern

4.2.3.3 Bidir_Put

Benchmark of MPI_Put, with bi-directional transfers.

Table 5 below shows the basic definitions, Figure 8 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) as indicated in Figure 8, non aggregate ($M=1$) and aggregate (cf. 4.2.2.2; $M=n_{\text{sample}}$, see 5.2.8)
reported throughput	X/t , aggregate and non aggregate

Table 5: Bidir_Put definition

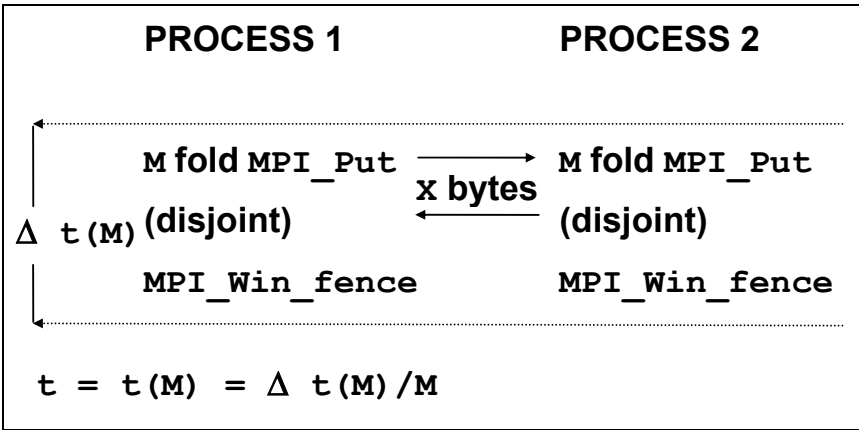


Figure 8: Bidir_Put pattern

4.2.3.4 Bidir_Get

Benchmark of the MPI_Get function, with bi-directional transfers.

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


measured pattern	as symbolized between 2 active processes only	 in Figure 9;
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. 4.2.2.2; M=n_sample, see 5.2.8)	
reported throughput	X/t, aggregate and non aggregate	

Table 6: Bidir_Get definition

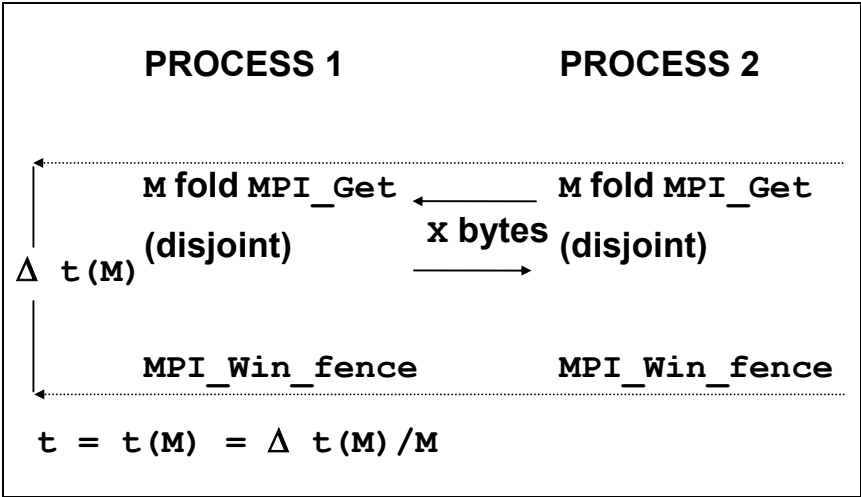


Figure 9: Bidir_Get pattern

4.2.3.5 Accumulate

Benchmark of the `MPI_Accumulate` function. Reduces a vector of length $L = X/\text{sizeof}(\text{float})$ float items. The MPI datatype is `MPI_FLOAT`, the MPI operation is `MPI_SUM`.

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


measured pattern	as symbolized between  in Figure 10
based on	<code>MPI_Accumulate</code>
<code>MPI_Datatype</code>	<code>MPI_FLOAT</code>
<code>MPI_Op</code>	<code>MPI_SUM</code>
Root	0
reported timings	$t = t(M)$ (in μsec) as indicated in Figure 10, non aggregate ($M=1$) and aggregate (cf. 4.2.2.2; $M=n_{\text{sample}}$, see 5.2.8)
reported throughput	none

Table 7: Accumulate definition

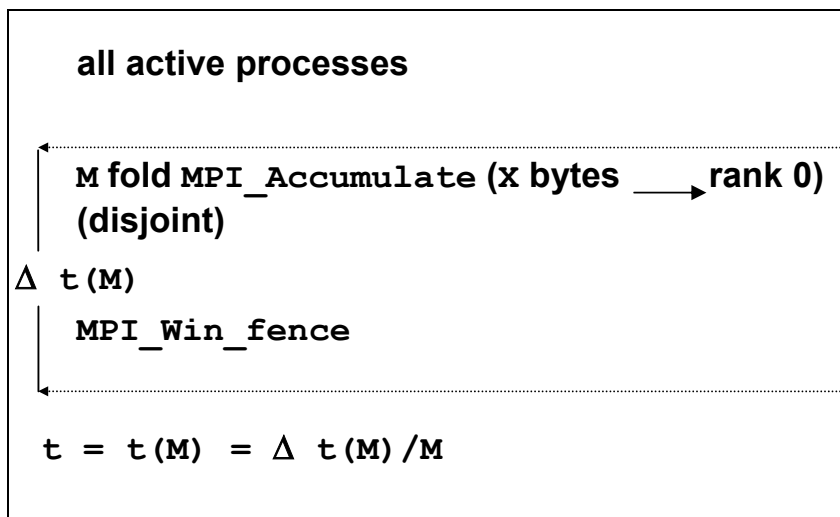


Figure 10: Accumulate pattern

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` is to properly initialize an access epoch (this is a correction in version 2.2 as compared to earlier releases).

Table 8 below shows the basic definitions,

Figure 11 a schematic view of the pattern.

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

Table 8: Window definition

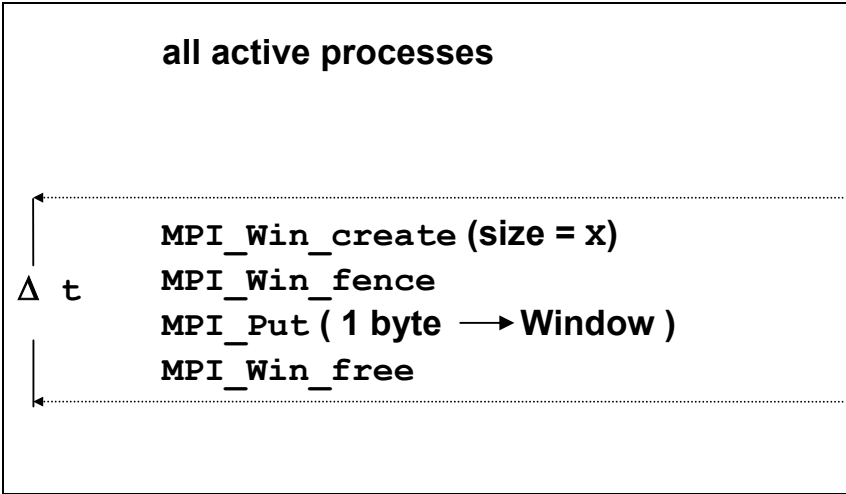


Figure 11: Window pattern

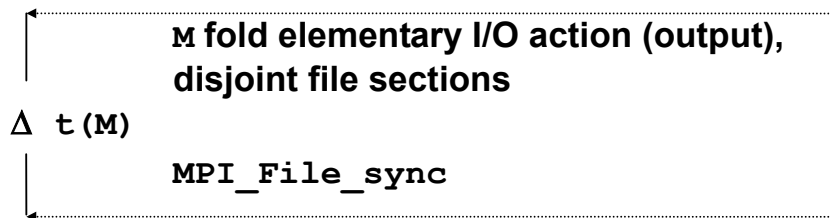
4.2.4 Definition of the IMB-IO benchmarks (blocking case)

This section describes the blocking I/O benchmarks in detail (see 4.2.5 for the nonblocking 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 x . Basic MPI datatype for all data buffers is `MPI_BYTE`.

All benchmark flavors, of course, have a `Write` and a `Read` component. In the sequel, a symbol `[ACTION]` will be used to denote `Read` or `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 mode.

Output: M fold aggregation



non aggregate mode:

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

aggregate mode:

$$t = \Delta t(M = n_sample) / M$$

(choice of $M = n_sample$: see 5.2.8)

Input: No aggregation

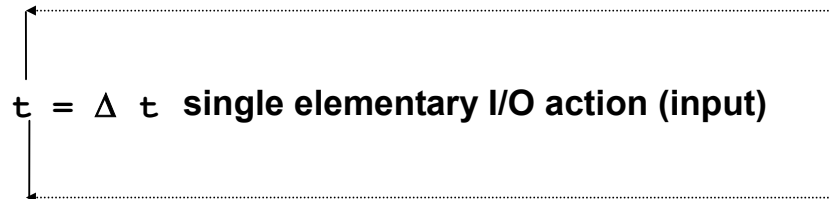


Figure 12: I/O benchmarks, aggregation for output

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 1 a schematic view of the pattern.

measured pattern	as symbolized in Figure 12
elementary I/O action	as symbolized in Figure 1
based on resp. for nonblocking mode	MPI_File_write / MPI_File_read MPI_File_irewrite / MPI_File_iread
etype	MPI_BYTE
filetype	MPI_BYTE
MPI_Datatype	MPI_BYTE
reported timings	t (in μ sec) 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

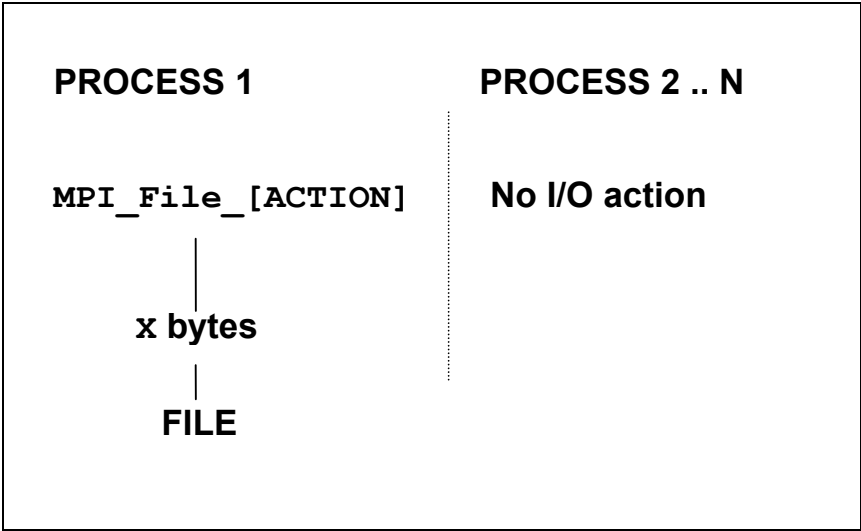


Figure 13: S_[ACTION]_indv pattern

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 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 μ sec) 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

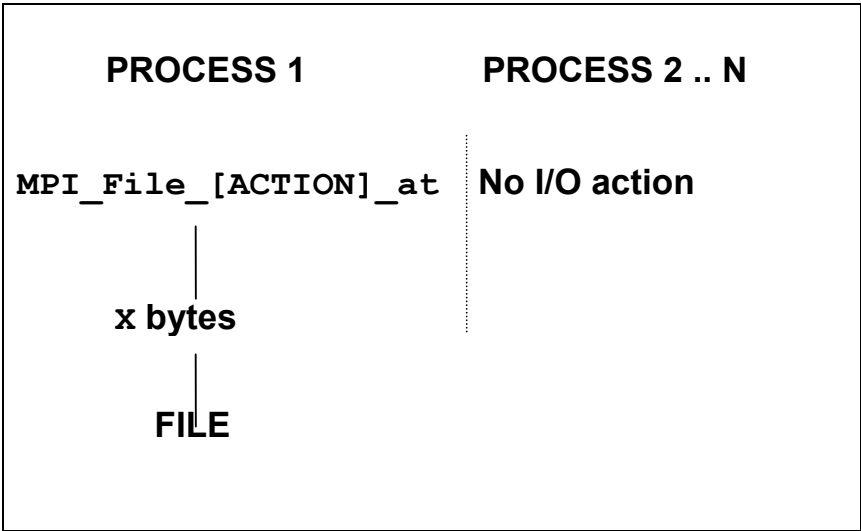


Figure 14: S_[ACTION]_expl pattern

4.2.4.3 P_[ACTION]_indv

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

Table 11 below shows the basic definitions, Figure 15 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_iread / MPI_File_iwrite
etype	MPI_BYTE
filetype	tilde view, disjoint contiguous blocks
MPI_Datatype	MPI_BYTE
reported timings	t (in μ sec) 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

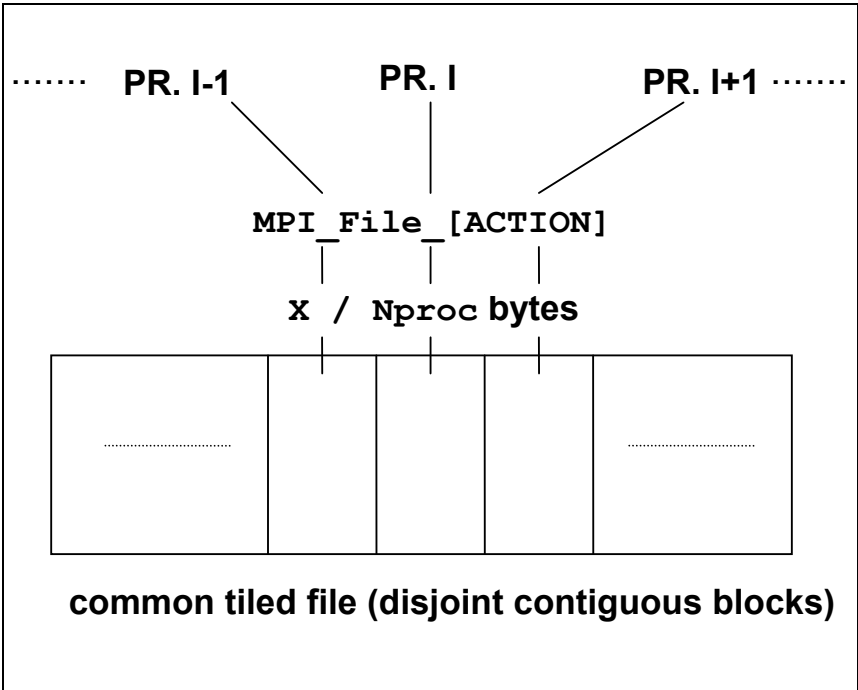


Figure 15: P_[ACTION]_indv pattern

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 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

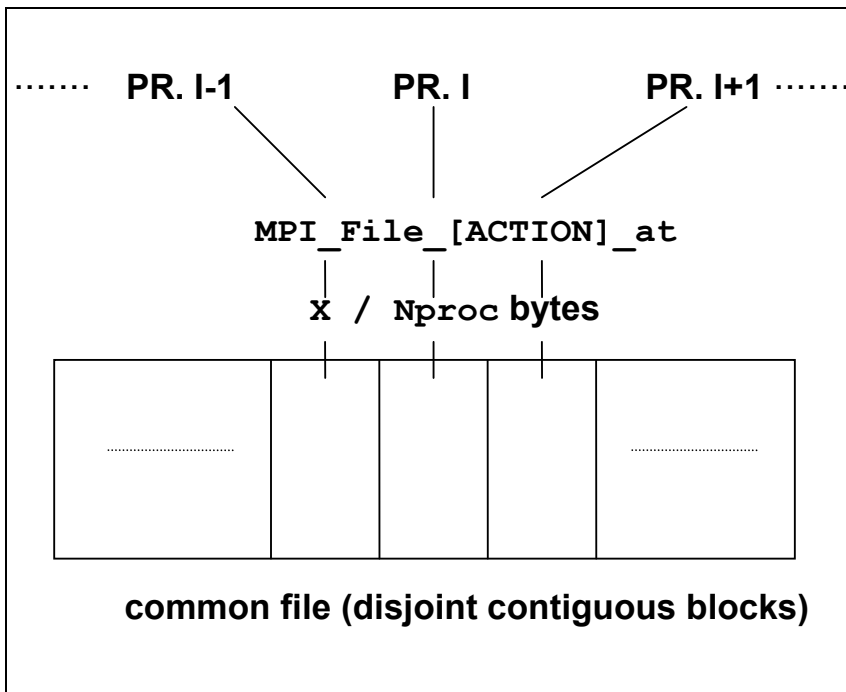


Figure 16: P_[ACTION]_expl pattern

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 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

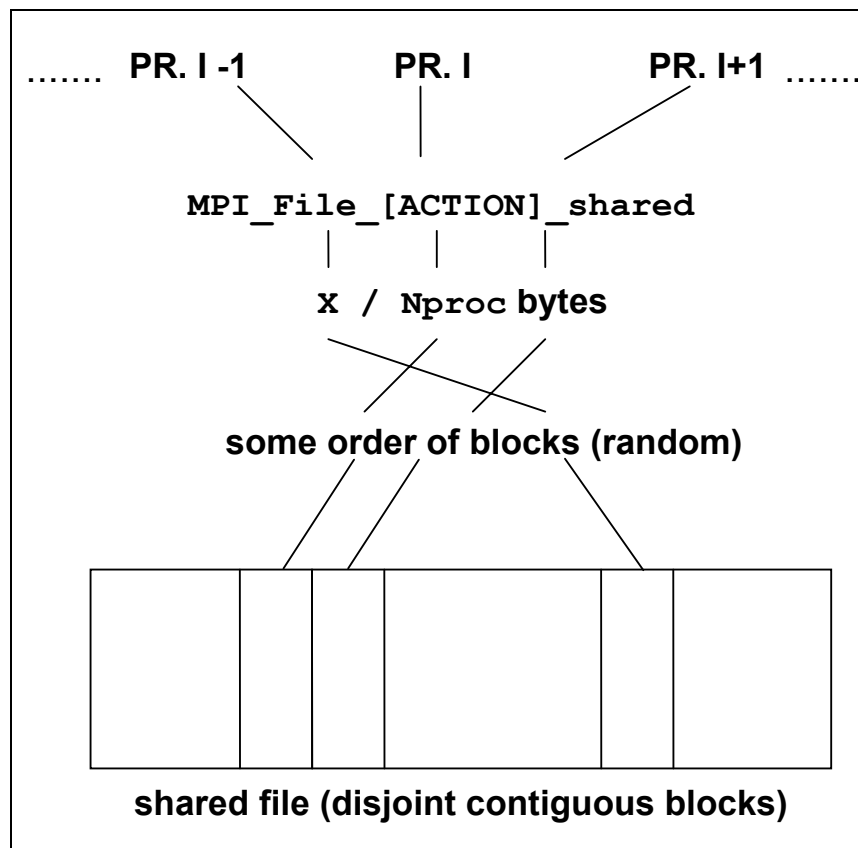


Figure 17: P_[ACTION]_shared pattern

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 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	$X/\Delta t$, aggregate and non aggregate for Write case

Table 14: P_[ACTION]_priv definition

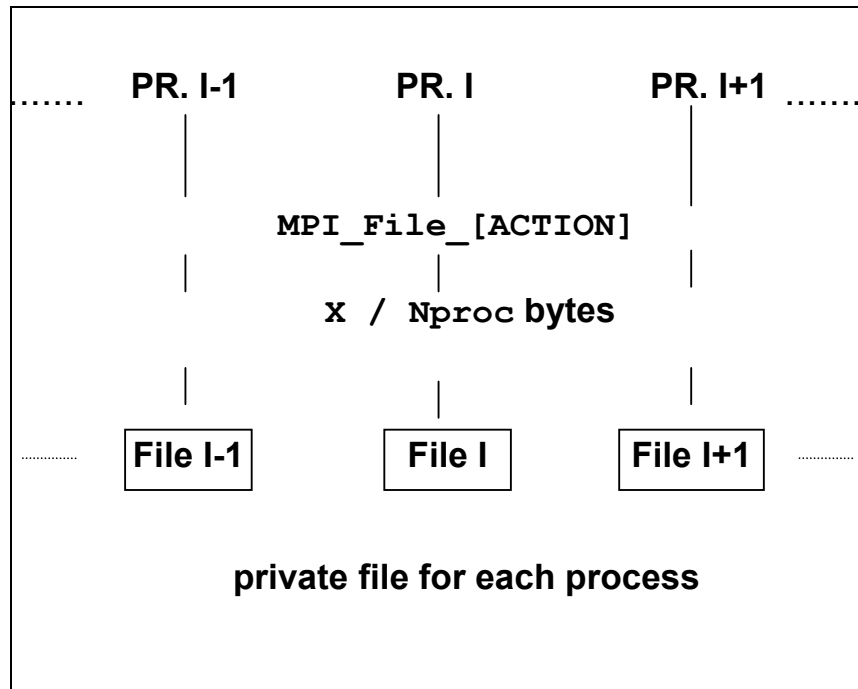


Figure 18: P_[ACTION]_priv pattern

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, a schematic view of the pattern is as 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, a schematic view of the pattern is as 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, a schematic view of the pattern is as 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	<code>MPI_File_open</code> / <code>MPI_File_close</code>
etype	<code>MPI_BYTE</code>
filetype	<code>MPI_BYTE</code>
reported timings	$t = \Delta t$ (in μsec) as indicated in Figure 19
reported throughput	none

Table 18: Open_Close definition

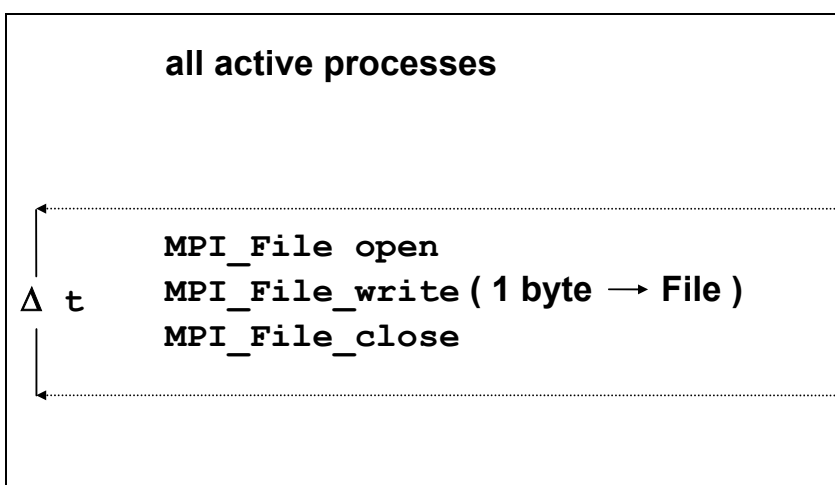


Figure 19: Open_Close pattern

4.2.5 Nonblocking I/O Benchmarks

Each of the nonblocking benchmarks, see Table 1, has a blocking equivalent explained in 4.2.4. All the definitions can be transferred identical, except their behavior with respect to

- aggregation (the nonblocking 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, of course, absolutely arbitrary. A benchmark as IMB can only decide for one particular way of exploiting 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 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 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 a 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 nonblocking 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 nonblocking I/O) would use when running dedicated
- `t_ovrl` = time for the analogous nonblocking 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, the user 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.

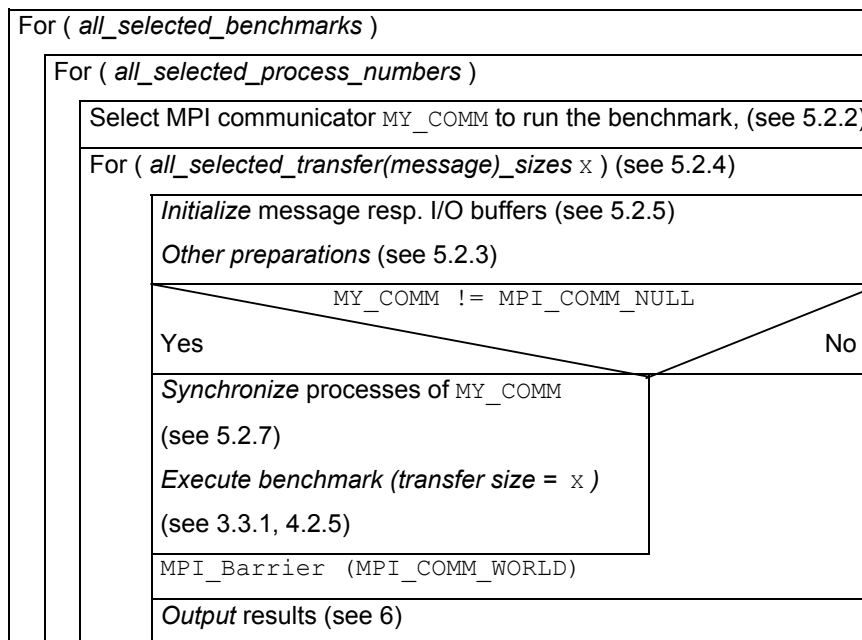


Figure 20: Control flow of IMB

The control parameters obviously necessary are either *command line arguments* (see 5.1.2) or parameter selections inside the IMB include files `settings.h` / `setting_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 one likes) 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`, whereafter they are broadcast to all other processes.

5.1.1 Default case

Just invoke

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

All benchmarks will run on $Q=[1,] 2, 4, 8, \dots$, largest $2^* < P$, P processes ($Q=1$ as discussed above IMB-IO). E.g. $P=11$, 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 general syntax is

```
mpirun -np P IMB-<..>
    [Benchmark1 [Benchmark2 [ ... ] ] ]
    [-npmin P_min]
    [-multi Outflag]
    [-input <Input_file>]
    [-msglen <File>]
    [-map <P>x<Q>]
```

(where the 6 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 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*P_min < P`, `P`.

5.1.2.3 -multi <outflag> selection

For selecting Multi/non Multi mode. The argument after `-multi` has to be an integer `Outflag`, 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)
- **Default** (no `-multi` selection): run primary (non Multi) versions.

5.1.2.4 -input <File> selection

An ASCII input file is used to select the benchmarks to run, e.g. 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`.

5.1.2.5 -msglen <File> selection

Enter any set of nonnegative message lengths to an ASCII file, line by line. Call it, e.g., "Lengths" and call IMB with arguments

```
-msglen Lengths
```

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

5.1.2.6 `-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

E.g., 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 IMB. 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 `settings_io.h` contain MSG (for “message”), in consistency with `settings.h`.

Parameter (standard mode value)	Meaning
IMB_OPTIONAL (not set)	has to be set when user optional settings are to be activated
MINMSGLOG (0)	second smallest data transfer size is $\max(\text{unit}, 2^{\text{MINMSGLOG}})$ (the smallest always being 0), where unit = sizeof(float) for reductions, unit = 1 else
MAXMSGLOG (22)	largest message size is $2^{\text{MAXMSGLOG}}$ Sizes $0, 2^i$ ($i=\text{MINMSGLOG}, \dots, \text{MAXMSGLOG}$) are used
MSGSPERSAMPLE (1000)	max. repetition count for all IMB-MPI1 benchmarks
MSGSPERSAMPLE (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 $\text{MSGSPERSAMPLE} \quad (X=0),$ $\min(\text{MSGSPERSAMPLE}, \max(1, \text{OVERALL_VOL}/X)) \quad (X>0)$ NOTE: OVERALL_VOL does <i>not</i> restrict the size of the max. data transfer. $2^{\text{MAXMSGLOG}}$ is the largest size, independent of OVERALL_VOL
N_WARMUP (2)	Number of <i>Warmup</i> sweeps (see 5.2.6)
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 nonblocking 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.

Note:

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 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 exists, the previous communicator is freed. When running $Q \leq P$ processes, the first Q ranks of `MPI_COMM_WORLD` are put into one group, the remaining $P-Q$ get `MPI_COMM_NULL` in Figure 20.

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

5.2.3 Other preparations

5.2.3.1 Window (IMB_EXT)

An Info is set (see 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:
Parameter 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 evt.) 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 `amode = 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 the *user is 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. 5.1.2.5).

5.2.4.2 IMB-IO

Set in `settings_io.h` (see 5.2.1), used unless `-msglen` flag is selected (ref. 5.1.2.5).

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 datatype 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 set by a CPP macro, currently

```
#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)

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

5.2.7 Synchronization

Before the actual benchmark, `N_BARR` (constant defined in `settings.h` and `settings_io.h`, current value 2) many

```
MPI_Barrier(MY_COMM)
```

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

5.2.8 The actual benchmark

In order to reduce measurement errors caused by to 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 analogous, with `MSGSPERSAMPLE` replaced by `MSGSPERNONAGGR` (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 Nonblocking I/O case

As explained in 4.2.5, a nonblocking benchmark has to provide three timings (blocking pure I/O time t_{pure} , nonblocking 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

Most easily, output is explained by sample outputs, see the tables below. What one sees is the following.

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

```

#include <sys/utsname.h>

void    IMB_make_sys_info()
{
    struct utsname info;
    int err;
    err = uname( &info );

    fprintf(unit, "# Machine      : %s" ,info.machine);
    fprintf(unit, "# System      : %s\n",info.sysname);
    fprintf(unit, "# Release     : %s\n",info.release);
    fprintf(unit, "# Version     : %s\n",info.version);
}

```

- **Non multi case numbers**

After a benchmark, 3 time values are available: Tmax, Tmin, Tavg, the maximum, minimum and average time, resp., extended over the group of active processes. 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

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

```
#-----
#   Intel (R) MPI Benchmark Suite V2.3, MPI-1 part
#-----
# Date       : Wed Sep 29 22:01:21 2004
# Machine    : ia64# System      : Linux
# Release    : 2.4.21-9.EL
# Version    : #1 SMP Thu Jan 8 16:54:40 EST 2004

#
# 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

6.2 Sample 2

```
<..> -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 V2.3, MPI-1 part
#-----
# Date       : Wed Sep 29 22:16:05 2004
# Machine    : ia64# System      : Linux
# Release    : 2.4.21-9.EL
# Version    : #1 SMP Thu Jan 8 16:54:40 EST 2004
#
# 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:

```
# (Multi-)PingPing
# (Multi-)Allreduce
```

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

```

# Group 0:      0      2
#
# Group 1:      1      3
#
#-----
#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      2
#
#   1      3
#
#-----
#bytes #repetitions  t_min[usec]  t_max[usec]  t_avg[usec]
#   0           1000          ..          ..          ..
#  100           1000
# 1000           1000
#10000           1000
#100000           419
#1000000           41

```

6.3 Sample 3

<.> IMB-IO -np 2 p_write_indv -npmin 2

```

#-----
# Intel (R) MPI Benchmark Suite V2.3, MPI-IO part
#-----
# Date       : Thu Sep 30 13:41:30 2004
# Machine    : ia64# System      : Linux
# Release    : 2.4.21-9.EL
# Version    : #1 SMP Thu Jan 8 16:54:40 EST 2004
#
# 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
```

7 Further details

7.1 Memory requirements

Benchmarks	Standard mode memory demand per process (Q active processes)	Optional mode memory demand per process ($X = 2^{\text{MAXMSGLOG}}$)
Alltoall	$Q \times 8$ MBytes	$Q \times 2X$ bytes
Allgather, Allgatherv	$(Q+1) \times 4$ MBytes	$(Q+1) \times X$ bytes
All other MPI1 benchmarks	8 MBytes	2X bytes
IMB-EXT	80 Mbytes	$2 \max(X, \text{OVERALL_VOL})$ bytes
IMB-IO	32 Mbytes	2X bytes
	disk space overall	disk space overall
IMB-IO	16 Mbytes	$\max(X, \text{OVERALL_VOL})$ bytes

Table 20 : Memory requirements with standard settings

7.2 Results checking

By activating the `cpp` flag `-DCHECK` through the `CPPFLAGS` variable (see 2.1), and recompiling, at IMB runtime every message passing result will be checked against the expected outcome (note that the contents of each buffer is well defined, see 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! Don't forget to deactivate `DCHECK` and recompile in order to get proper results.

8 Revision History

Release No.	Date
2.3	Nov. 2004