

Pallas MPI Benchmarks - PMB, Part MPI-2



Pallas GmbH
Hermülheimer Str. 10
D-50321 Brühl
Phone: +49-(0)2232-1896-0
Fax: +49-(0)2232-1896-29
<http://www.pallas.com>

1	INTRODUCTION	3
2	INSTALLATION AND QUICK START	4
2.1	Installation	4
2.2	Running PMB	6
3	OVERVIEW OF PMB-MPI2	7
3.1	The List of Benchmarks	7
3.2	Release 2.2 vs. Earlier	9
4	PMB-MPI2 BENCHMARK DEFINITIONS	9
4.1	Benchmark Classification	9
4.1.1	Single Transfer Benchmarks	10
4.1.2	Parallel Transfer Benchmarks	10
4.1.3	Collective Benchmarks	10
4.2	Benchmark Modes	11
4.2.1	Blocking / Nonblocking Mode (only PMB-IO)	11
4.2.2	Aggregate / Non Aggregate Mode	11
4.3	Definition of the PMB-EXT Benchmarks	12
4.3.1	Unidir_Put	13
4.3.2	Unidir_Get	14
4.3.3	Bidir_Put	15
4.3.4	Bidir_Get	16
4.3.5	Accumulate	17
4.3.6	Window	18
4.4	Definition of the PMB-IO Benchmarks (Blocking Case)	19
4.4.1	S_[ACTION]_indv	20
4.4.2	S_[ACTION]_expl	21
4.4.3	P_[ACTION]_indv	22
4.4.4	P_[ACTION]_expl	23
4.4.5	P_[ACTION]_shared	24
4.4.6	P_[ACTION]_priv	25
4.4.7	C_[ACTION]_indv	26
4.4.8	C_[ACTION]_expl	26
4.4.9	C_[ACTION]_shared	26
4.4.10	Open_Close	27
4.5	Nonblocking I/O Benchmarks	28
4.5.1	Exploiting CPU	28
4.5.2	Displaying Results	29
4.6	Multi - Versions	29
5	BENCHMARK METHODOLOGY	30

5.1	Running PMB, Command Line Control	31
5.1.1	Default Case	31
5.1.2	Command Line Control	31
5.2	PMB Parameters and Hard-coded Settings	32
5.2.1	Parameters Controlling PMB	32
5.2.2	Communicators, Active Processes	34
5.2.3	Other Preparations	34
5.2.4	Buffer Lengths	35
5.2.5	Buffer Initialization	35
5.2.6	Synchronization	36
5.2.7	The Actual Benchmark, Blocking Case	36
5.2.8	The Actual Benchmark, Nonblocking I/O Case	37
6	OUTPUT	37
6.1	Table of a PMB-IO Write Benchmark	37
6.2	Table of a PMB-IO Nonblocking Benchmark	40
7	FURTHER DETAILS	42
7.1	Memory and Disk Space Requirements	42
7.2	SRC Directory	42
7.3	Results Checking	42
7.4	Use of MPI	43
8	REVISION HISTORY	43
9	REFERENCES	44

1 Introduction

In an effort to define a standard API for message-passing programming, a forum of HPC vendors, researchers and users has developed the Message Passing Interface. MPI-1 [1] and MPI-2 [2] are now firmly established as the premier message-passing API, with implementations available for a wide range of platforms in the high-performance and general computing area, and a growing number of applications and libraries using MPI. To help compare the performance of various computing platforms and/or MPI implementations, the need for a set of well-defined MPI benchmarks arises.

This document presents the Pallas MPI Benchmarks (PMB) 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.

This document accompanies the version 2.2 of PMB. The code is written in ANSI C plus standard MPI (about 10 000 lines of code, 100 functions in 48 source files).

The PMB 2.2 package consists of 3 separate parts:

- PMB-MPI1 (see [3]),
- PMB-MPI2, subdivided into the two parts
PMB-EXT (Onesided Communications benchmarks),
PMB-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 PMB-MPI1. Only standard MPI-1 functions [1] are used, no dummy library is needed. Similarly, PMB-EXT (PMB-IO) can be built with a library containing full MPI-1 plus only Onesided Communications (only I/O, resp.) extensions.

The MPI-1 section has been defined in [3]. It is recommended to read [3] first, as much of the methodology and the basic ideas are very similar.

This document defines PMB-MPI2. The subdivision into I/O and non-I/O parts is sensible, as the methodology differs in both cases (see section 5).

In section 3 an overview of the suite is given.

Section 3.2 defines the single benchmarks in detail. A classification, corresponding to the one in [3], is included (section 4.1). *Single Transfer*, *Parallel Transfer*, *Collective* are the classes. Roughly speaking, Single transfers are *dedicated*, without obstructions from other transfers, best case result are to be expected. 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.

PMB-MPI2 introduces a new aspect, the *mode* of a benchmark (see 4.2). Branches of certain benchmarks are distinguished as to

- aggregate / non aggregate mode
- blocking / nonblocking mode

To sketch the meaning of *aggregation*, e.g., when writing to a file (PMB-IO

output benchmarks), it is important to know whether a system/implementation is capable of accumulating several output requests (before physically flushing them to a disk), and thus saving expensive file synchronizations. A very similar question arises in the one sided communications context. PMB-MPI2 will run certain benchmarks in two ways, aggregate and non aggregate, in order to provide for a (partial) answer to these questions. See 4.2 for more details. The corresponding benchmark tables will automatically contain aggregate and non aggregate results, without PMB demanding for a user selection.

Another mode distinction for all `Read` and `Write` MPI-IO benchmarks is *blocking* / *nonblocking* (in proper MPI definition). For all those benchmarks, PMB includes both modes. However, the blocking and the nonblocking mode of a certain benchmark are kept in two different single benchmarks (when `<.>_Read<...>` / `<.>_Write<...>` are certain blocking benchmarks, see Table 1, then `<.>_IRead<...>` / `<.>_IWrite<...>` are the nonblocking equivalents). See 4.5 for the definitions.

Section 5 defines the methodology and rules of PMB. Section 6 shows sample output tables, section 7 explains further details.

2 Installation and Quick Start

In order to run PMB2, one needs:

- `cpp`, ANSI C compiler, `make`.
- For PMB-MPI1: Full MPI-1 installation, including startup mechanism for parallel MPI programs.
- For PMB-EXT: Full MPI-1 installation, plus a library implementing Onesided Communications extensions of MPI-2
- For PMB-IO: Full MPI-1 installation, plus a library implementing I/O extensions of MPI-2

See 7.1 for the resource requirements of PMB.

2.1 Installation

After unpacking, on the current directory is created:

PMB2 (directory)

PMB2/SRC (subdirectory containing sources and Makefile, see 7.1)

PMB2/RESULTS (subdirectory with sample results from various machines)

PMB2/DOC (subdirectory containing this document in postscript format)

The installation is performed in the SRC subdirectory to keep the structure easy. Here, a generic `Makefile` can be found. All rules and dependencies are defined there. Only a few (7, precisely) machine dependent variables have to be set, whereafter an easy

```
make PMB-MPI1
```

or

```
make PMB-EXT
```

or

```
make PMB-IO
```

will perform the installation. When the default rule is defined in `Makefile`, like

default: PMB-IO

the corresponding executable can be built by just

make

For defining the machine dependent settings, the section

```
##### User configurable options #####
```

```
#include make_i86
```

```
#include make_solaris
```

```
#include make_dec
```

```
#include make_sp2
```

```
#include make_sr2201
```

```
#include make_vpp
```

```
#include make_t3e
```

```
#include make_sgi
```

```
#include make_sx4
```

```
### End User configurable options ###
```

is provided in the Makefile header. The listed `make_*` files are on the directory and have been used successfully on certain systems. *First check whether one of these can be used for your purpose (with no or marginal changes).*

However, usually the user will have to edit an own `make_mydefs` file. This has to contain 7 variable assignments used by Makefile:

```
CC = <name of the ANSI C compiler>
```

```
CLINKER = <name of the ANSI C linker>
```

```
OPTFLAGS = <flags for $(CC) compilation step>
```

```
CPPFLAGS = <cpp flags>
```

```
/*
```

```
Allowed values: -DnoCHECK, -DCHECK (see 7.3;
```

```
check that proper benchmarks are always with the cpp  
flag -DnoCHECK!)
```

```
*/
```

```
LIB_PATH = <path of libraries (libmpi.a,...)> ,
```

```
/* in the form "-L<path1> -L<path2> .." */
```

```
LIBS = <lib flags for link step>, e.g. -lmpi -l....
```

```
MPI_INCLUDE = <directory containing MPI include file  
mpi.h>
```

Activate these flags by editing the Makefile header:

```
##### User configurable options #####
include make_mydefs
#include make_i86
#include make_solaris
#include make_dec
#include make_sp2
#include make_sr2201
#include make_vpp
#include make_t3e
#include make_sgi
#include make_sx4
### End User configurable options ###
```

User flags will be used in the following way:

```
$(CC) -I$(MPI_INCLUDE) $(CPPFLAGS) $(OPTFLAGS) -c
```

(for compilation)

```
$(CLINKER) -o <exe-name> [.o's] $(LIB_PATH) $(LIBS)
```

(for linking).

Of course, the user may define own auxiliary variables in `make_mydefs`.

Now, type

```
make [PMB-xxx]
```

Compilation should be quite short, and executable `PMB-xxx` will be generated.

2.2 Running PMB

Check the right way of running parallel MPI programs on your system. Usually, a startup procedure has to be invoked, like

```
mpirun -np P PMB-xxx
```

(P being the number of processes to load; $P=1$ is allowed!). This will run all of PMB on $1, 2, 4, \dots, P$ processes and will output results on stdout. Also is possible

```
mpirun -np P PMB-xxx [Benchmark names]
```

where the names are valid PMB benchmark names (see Table 1).

For the details, see 5.1.

3 Overview of PMB-MPI2

In this section, a complete list of the benchmarks is given. Then, the changes in release 2.2 as compared to earlier releases re denoted.

3.1 The List of Benchmarks

Table 1 below contains a list of all PMB-MPI2 benchmarks. The exact definitions are given in section 3.2, in particular refer to 4.2.2 for an explanation of the *Aggregate Mode*, 4.5 for the *Nonblocking Mode* column. Section 5 describes the benchmark methodology.

The nonblocking modes of PMB-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
PMB-EXT		
Window		
Unidir_Put	x	
Unidir_Get	x	
Bidir_Get	x	
Bidir_Put	x	
Accumulate	x	
Multi- versions of the above	x	

... continued

Benchmark	Aggregate Mode	Nonblocking Mode
PMB-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: PMB-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* [3].
- the `Multi-` prefix is defined as in [3]. 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 [3]. 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 4.1.
- 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).

3.2 Release 2.2 vs. Earlier

There was a semantic bug in the PMB-EXT part of earlier releases: the used MPI window access epochs were not properly started. This bug has been fixed, see 5.2.3.1 and 4.3.6. Another change is related to both MPI1 and MPI2 parts of PMB, see [3]. A function `set_default` initializes the basic control structure in the very beginning of PMB-xxx, with default values.

4 PMB-MPI2 Benchmark Definitions

In this section, all PMB-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.

4.1 Benchmark Classification

To clearly structure the set of benchmarks, PMB 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 PMB. Note that this is the classification already introduced for PMB-MPI1 [3]. 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
Unidir_Put	Multi-Unidir_Put	Multi-Accumulate	(also Multi)
Bidir_Get	Multi-Bidir_Get		
Bidir_Put	Multi-Bidir_Put		
S_[I]Write_indv	P_[I]Write_indv	C_[I]Write_indv	Open_close
S_[I]Read_indv	P_[I]Read_indv	C_[I]Read_indv	(also Multi)
S_[I]Write_expl	P_[I]Write_expl	C_[I]Write_expl	
S_[I]Read_expl	P_[I]Read_expl	C_[I]Read_expl	
	P_[I]Write_shared	C_[I]Write_shared	
	P_[I]Read_shared	C_[I]Read_shared	
	P_[I]Write_priv	Multi- versions	
	P_[I]Read_priv		

Table 2: PMB-MPI2 benchmark classification

4.1.1 Single Transfer Benchmarks

The benchmarks in this class focus on a *single* data transferred between *one* source and *one* target. In PMB-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 PMB-EXT benchmarks only run with 2 active processes
- single transfer PMB-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.

4.1.2 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 [3], and the results of the worst group are displayed.

4.1.3 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 Benchmark Modes

Certain benchmarks have different *modes* to run.

4.2.1 Blocking / Nonblocking Mode (only PMB-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 4.5 for the methodology.

4.2.2 Aggregate / Non Aggregate Mode

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

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

The basic pattern of these benchmarks is shown in Figure 1. Here,

- `M` is some repetition count
- a transfer is issued by the corresponding one sided communication call (for PMB-EXT) and by an MPI-IO write call (PMB-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` (PMB-EXT),
`MPI_File_sync` (PMB-IO `Write`).

PMB 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 5.2.7.

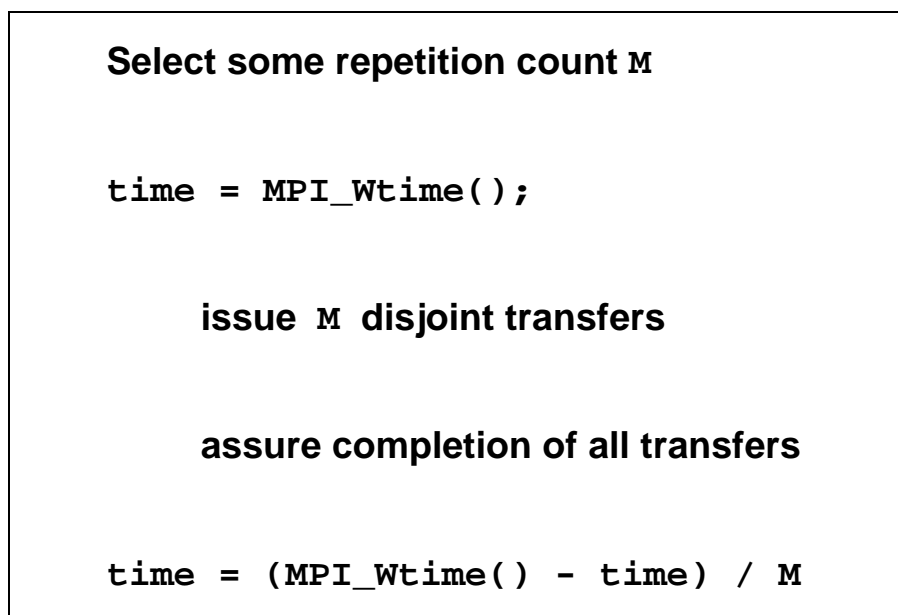


Figure 1: Aggregation of M transfers (PMB-EXT and blocking Write benchmarks)

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 1) 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.

4.3 Definition of the PMB-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.), described in [3]. Their interpretation and output is analogous to their message passing equivalents.

4.3.1 Unidir_Put

Benchmark of the `MPI_Put` function. Table 3 below shows the basic definitions, Figure 2 a schematic view of the pattern.


measured pattern	as symbolized between  in Figure 2; 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 2, non aggregate ($M=1$) and aggregate (cf. 4.2.2; $M=n_{\text{sample}}$, see 5.2.7)
reported throughput	X/t , aggregate and non aggregate

Table 3 : Unidir_Put definition

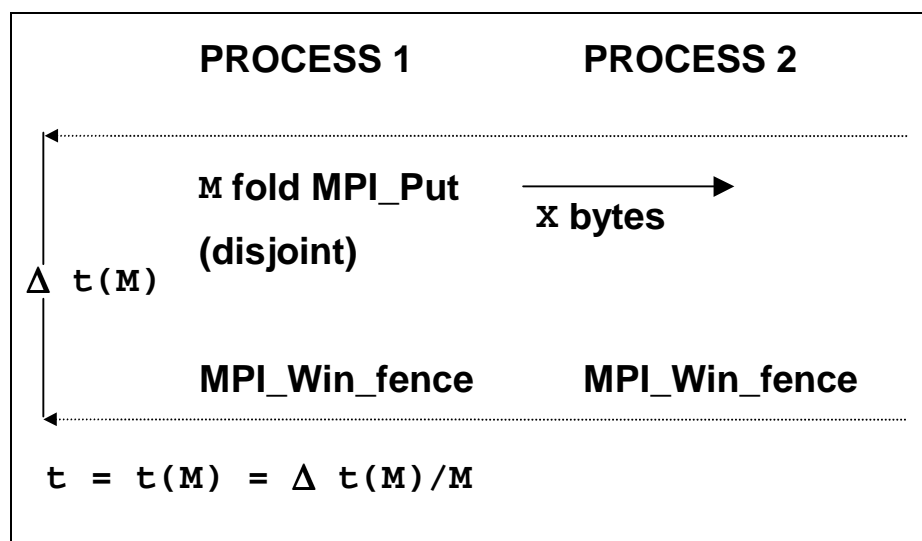


Figure 2: Unidir_Put pattern

4.3.2 Unidir_Get

Benchmark of the `MPI_Get` function.

Table 4 below shows the basic definitions, Figure 3 a schematic view of the pattern.


measured pattern	as symbolized between  in Figure 3; 2 active processes only
based on	<code>MPI_Get</code>
<code>MPI_Datatype</code>	<code>MPI_BYTE</code> (origin and target)
reported timings	$t = t(M)$ (in μsec) as indicated in Figure 3, non aggregate ($M=1$) and aggregate (cf. 4.2.2; $M=n_{\text{sample}}$, see 5.2.7)
reported throughput	X/t , aggregate and non aggregate

Table 4: Unidir_Get definition

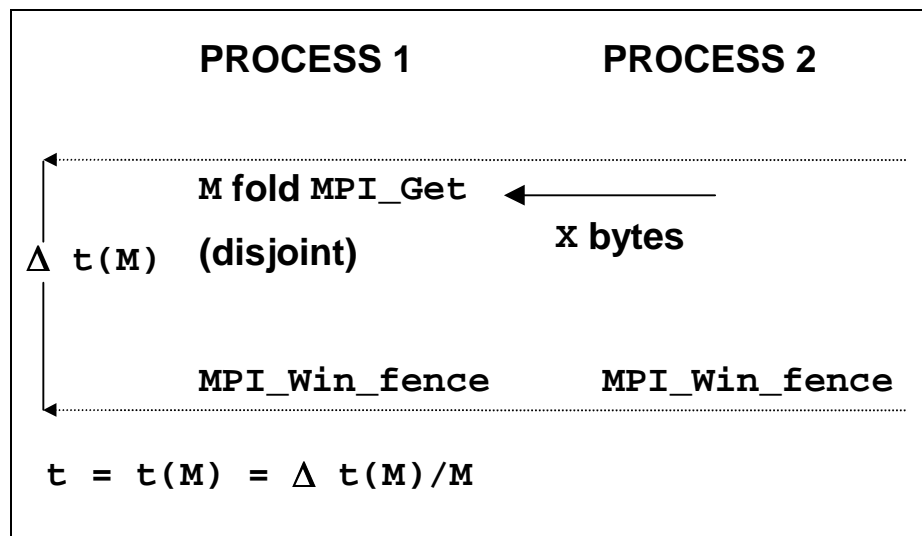


Figure 3: Unidir_Get pattern

4.3.3 Bidir_Put

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

Table 5 below shows the basic definitions, Figure 4 a schematic view of the pattern.


measured pattern	as symbolized between  in Figure 4; 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 4, non aggregate ($M=1$) and aggregate (cf. 4.2.2; $M=n_{\text{sample}}$, see 5.2.7)
reported throughput	X/t , aggregate and non aggregate

Table 5: Bidir_Put definition

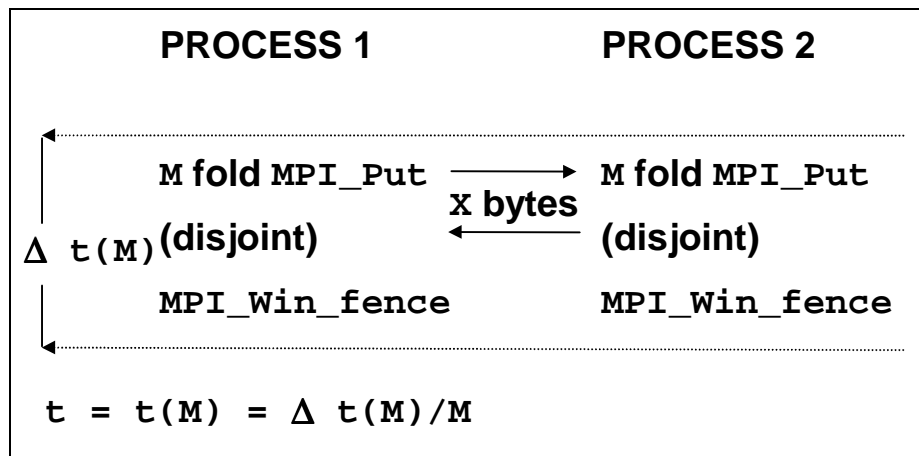


Figure 4: Bidir_Put pattern

4.3.4 Bidir_Get

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

Table 6 below shows the basic definitions, Figure 5 a schematic view of the pattern.


measured pattern	as symbolized between  in Figure 5; 2 active processes only
based on	<code>MPI_Get</code>
<code>MPI_Datatype</code>	<code>MPI_BYTE</code> (origin and target)
reported timings	$t = t(M)$ (in μsec) as indicated in Figure 5, non aggregate ($M=1$) and aggregate (cf. 4.2.2; $M=n_{\text{sample}}$, see 5.2.7)
reported throughput	X/t , aggregate and non aggregate

Table 6: Bidir_Get definition

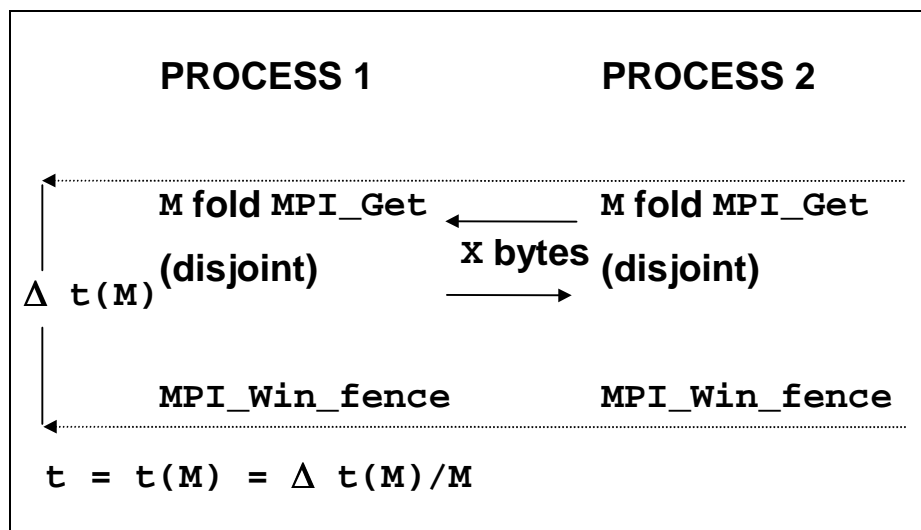


Figure 5: Bidir_Get pattern

4.3.5 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 6 a schematic view of the pattern.


measured pattern	as symbolized between  in Figure 6
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 6, non aggregate ($M=1$) and aggregate (cf. 4.2.2; $M=n_{\text{sample}}$, see 5.2.7)
reported throughput	none

Table 7: Accumulate definition

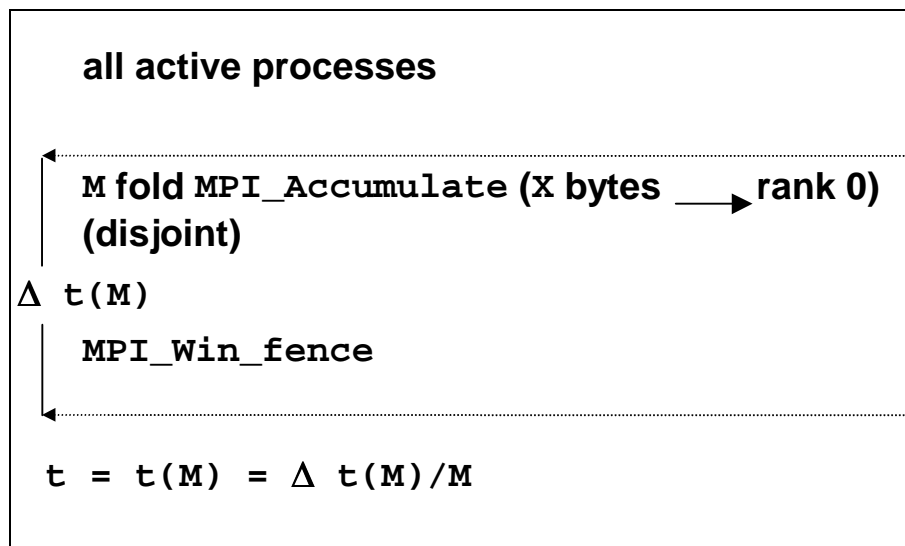


Figure 6: Accumulate pattern

4.3.6 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 7 a schematic view of the pattern.

measured pattern	<code>MPI_Win_create</code> / <code>MPI_Win_fence</code> / <code>MPI_Win_free</code>
reported timings	$t = \Delta t$ (in μsec) as indicated in Figure 7
reported throughput	none

Table 8: Window definition

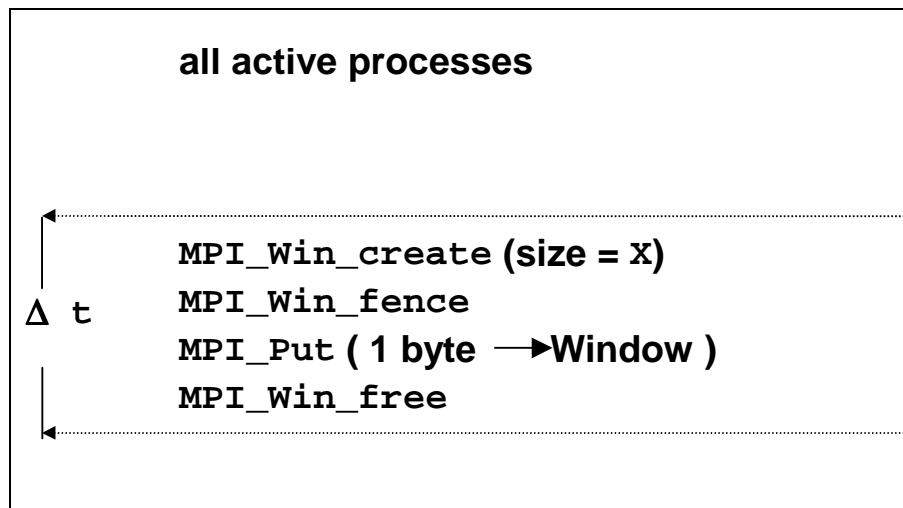


Figure 7: Window pattern

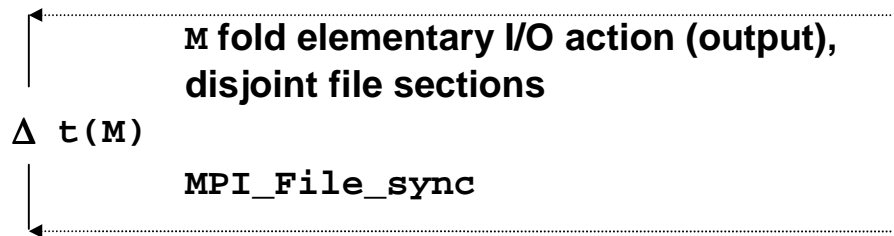
4.4 Definition of the PMB-IO Benchmarks (Blocking Case)

This section describes the blocking I/O benchmarks in detail (see 4.5 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 5.2.7)

Input: No aggregation

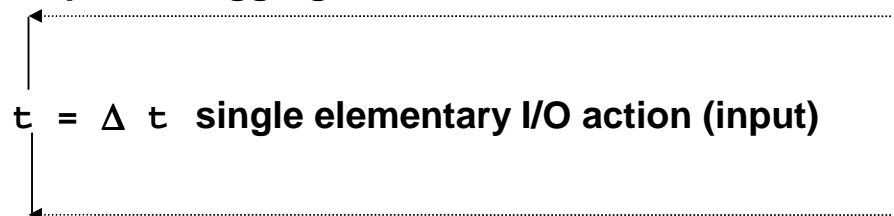


Figure 8: I/O benchmarks, aggregation for output

4.4.1 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 9 a schematic view of the pattern.

measured pattern	as symbolized in Figure 8
elementary I/O action	as symbolized in Figure 9
based on resp. for nonblocking mode	MPI_File_write / MPI_File_read MPI_File_iread / MPI_File_iwrite
etype	MPI_BYTE
filetype	MPI_BYTE
MPI_Datatype	MPI_BYTE
reported timings	t (in μ sec) as indicated in Figure 8, aggregate and non aggregate for Write case
reported throughput	X/t, aggregate and non aggregate for Write case

Table 9: S_[ACTION]_indv definition

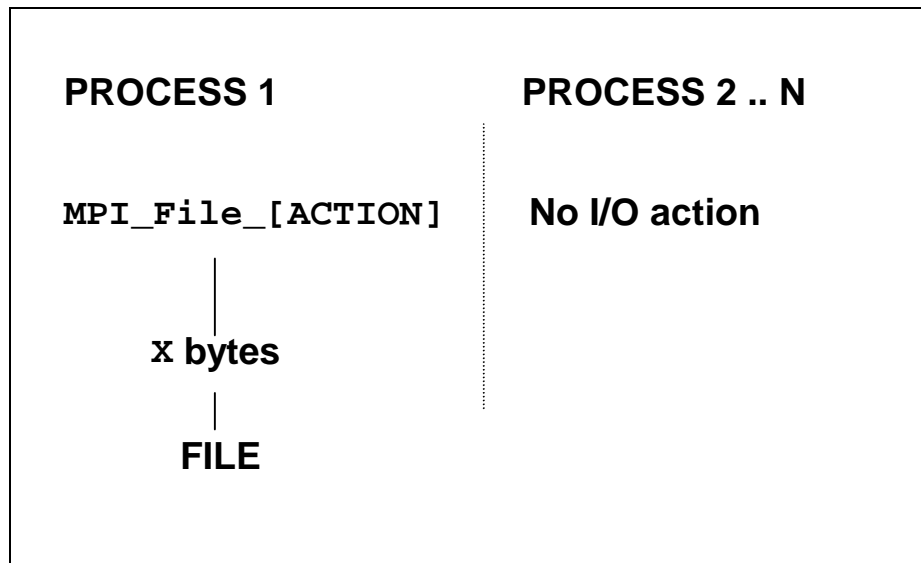


Figure 9: S_[ACTION]_indv pattern

4.4.2 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 10 a schematic view of the pattern.

measured pattern	as symbolized in Figure 8
elementary I/O action	as symbolized in Figure 10
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 8, aggregate and non aggregate for Write case
reported throughput	X/t, aggregate and non aggregate for Write case

Table 10: S_[ACTION]_expl definition

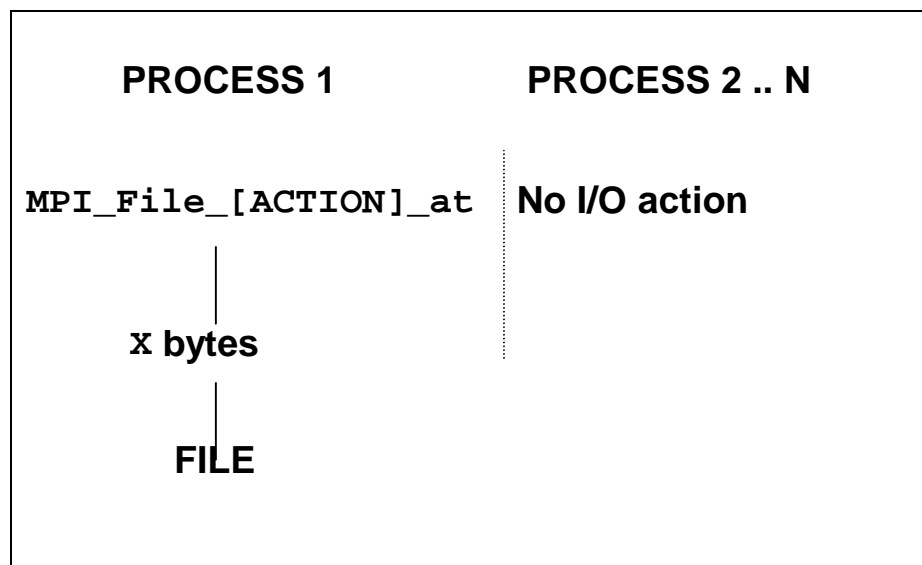


Figure 10: S_[ACTION]_expl pattern

4.4.3 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 11 a schematic view of the pattern.

measured pattern	as symbolized in Figure 8
elementary I/O action	as symbolized in Figure 11 (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	tilde view, disjoint contiguous blocks
MPI_Datatype	MPI_BYTE
reported timings	t (in μsec) as indicated in Figure 8, aggregate and non aggregate for Write case
reported throughput	X/t, aggregate and non aggregate for Write case

Table 11: P_[ACTION]_indv definition

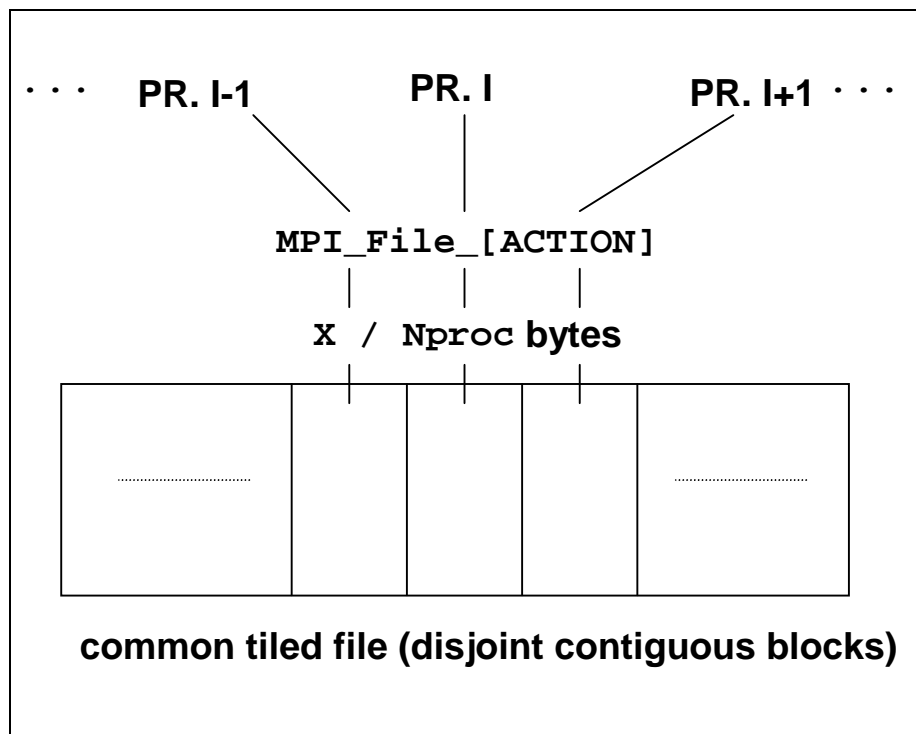


Figure 11: P_[ACTION]_indv pattern

4.4.4 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 12 a schematic view of the pattern.

measured pattern	as symbolized in Figure 8
elementary I/O action	as symbolized in Figure 12 (N_{proc} = 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 8, aggregate and non aggregate for Write case
reported throughput	x/t , aggregate and non aggregate for Write case

Table 12: P_[ACTION]_expl definition

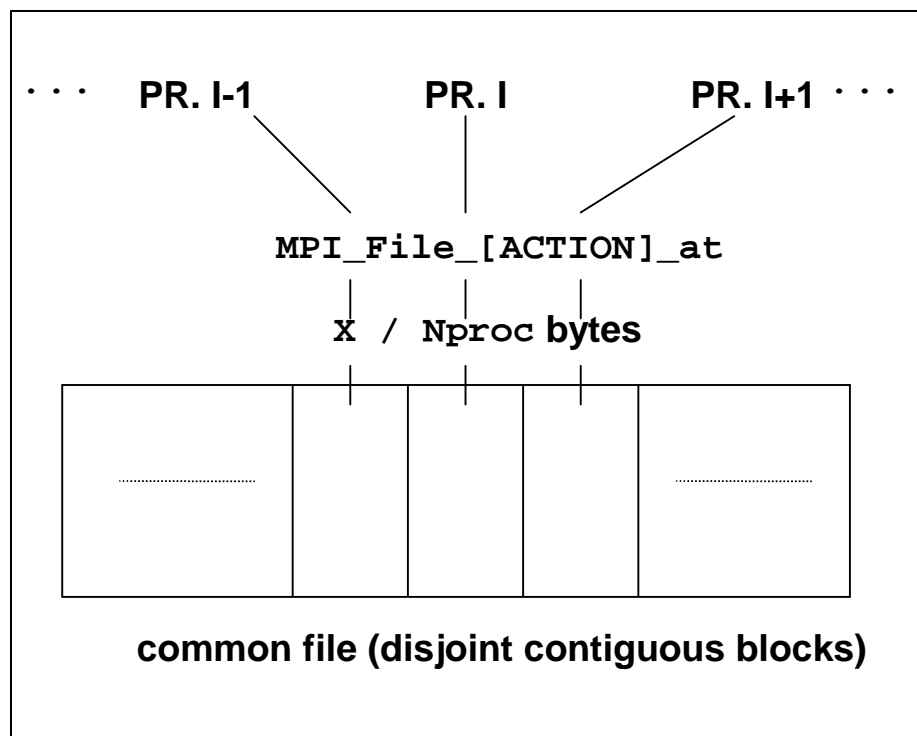


Figure 12: P_[ACTION]_expl pattern

4.4.5 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 13 a schematic view of the pattern.

measured pattern	as symbolized in Figure 8
elementary I/O action	as symbolized in Figure 13 (N _{proc} = 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 μ sec) as indicated in Figure 8, aggregate and non aggregate for Write case
reported throughput	X/t, aggregate and non aggregate for Write case

Table 13: P_[ACTION]_shared definition

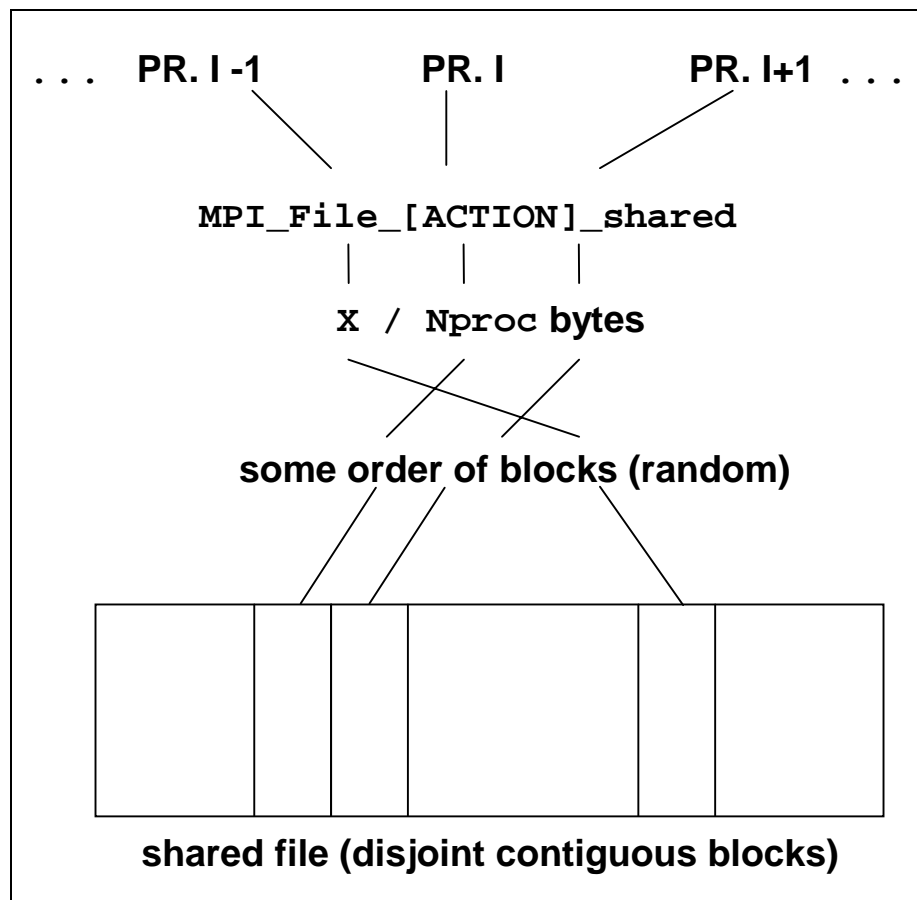


Figure 13: P_[ACTION]_shared pattern

4.4.6 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 14 a schematic view of the pattern.

measured pattern	as symbolized in Figure 8
elementary I/O action	as symbolized in Figure 14 (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	MPI_BYTE
MPI_Datatype	MPI_BYTE
reported timings	Δt (in μsec) as indicated in Figure 8, 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

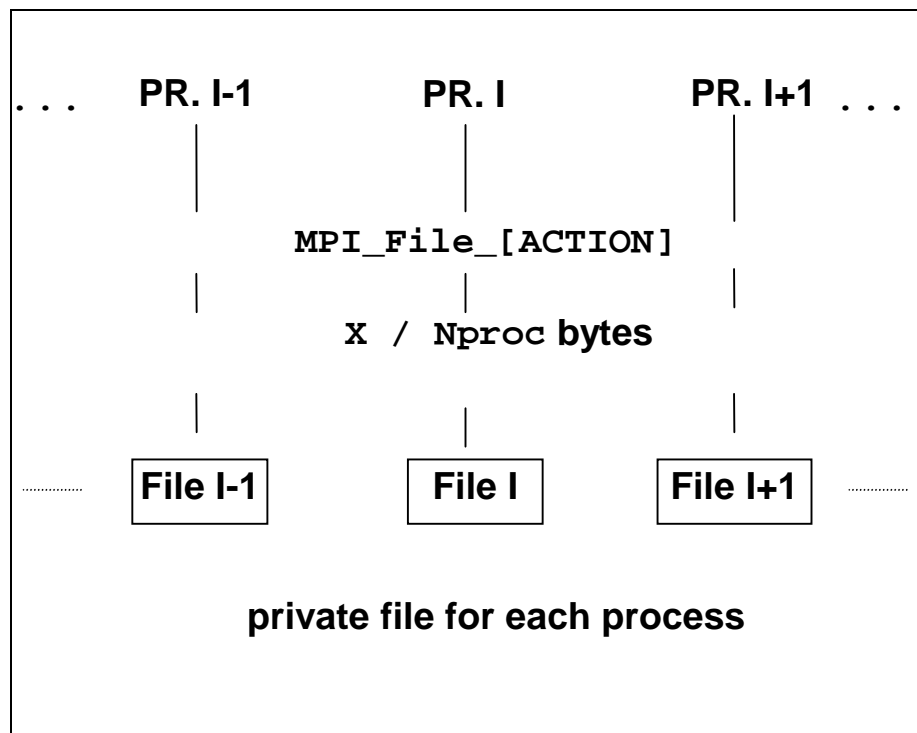


Figure 14: P_[ACTION]_priv pattern

4.4.7 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 11.

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 4.4.3

Table 15: C_[ACTION]_indv definition

4.4.8 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 12.

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 4.4.4

Table 16: C_[ACTION]_expl definition

4.4.9 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 13, 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 4.4.5

Table 17: C_[ACTION]_shared definition

4.4.10 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 15. 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 15
reported throughput	none

Table 18: Open_Close definition

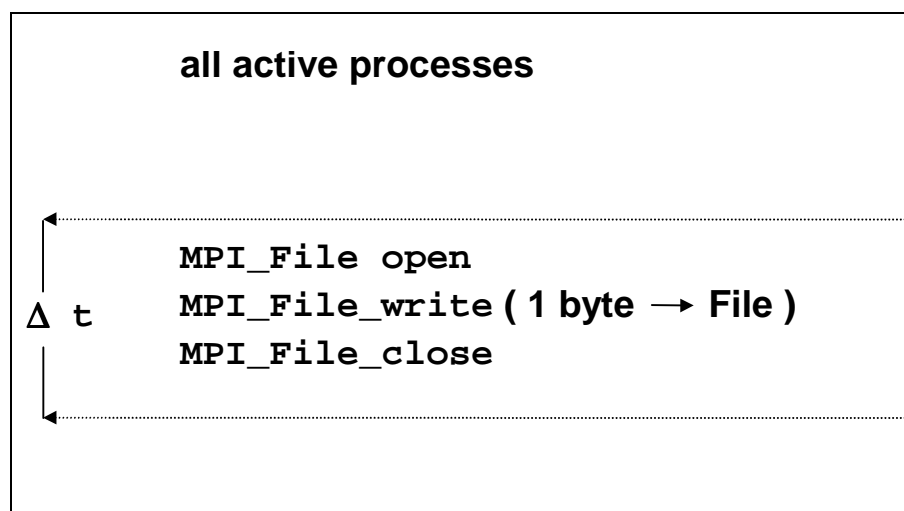


Figure 15: Open_Close pattern

4.5 Nonblocking I/O Benchmarks

Each of the nonblocking benchmarks, see Table 1, has a blocking equivalent explained in 4.4. 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 PMB 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.

4.5.1 Exploiting CPU

PMB 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.

4.5.2 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: $t_{\text{ovrl}} = \max(t_{\text{pure}}, t_{\text{CPU}})$.

No overlap would mean: $t_{\text{ovrl}} = t_{\text{pure}} + t_{\text{CPU}}$.

The actual amount of overlap is

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

PMB results tables will report the timings t_{ovrl} , t_{pure} , t_{CPU} and the estimated overlap obtained by (*) above (see 6.2). In the beginning of a run the Mflop/s rate corresponding to t_{CPU} is displayed.

4.6 Multi - Versions

The definition and interpretation of the `Multi-` prefix is analogous to the definition in the MPI1 part [3].

5 Benchmark Methodology

Recall that in chapter 3.2 only the underlying patterns of each benchmark have been defined. In this section, the measuring method for those patterns is explained.

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. Important is that (in contrast to the previous release 2.0) there is a *standard* and an *optional* mode to control PMB. 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 PMB. All *emphasized* items will be explained in more detail.

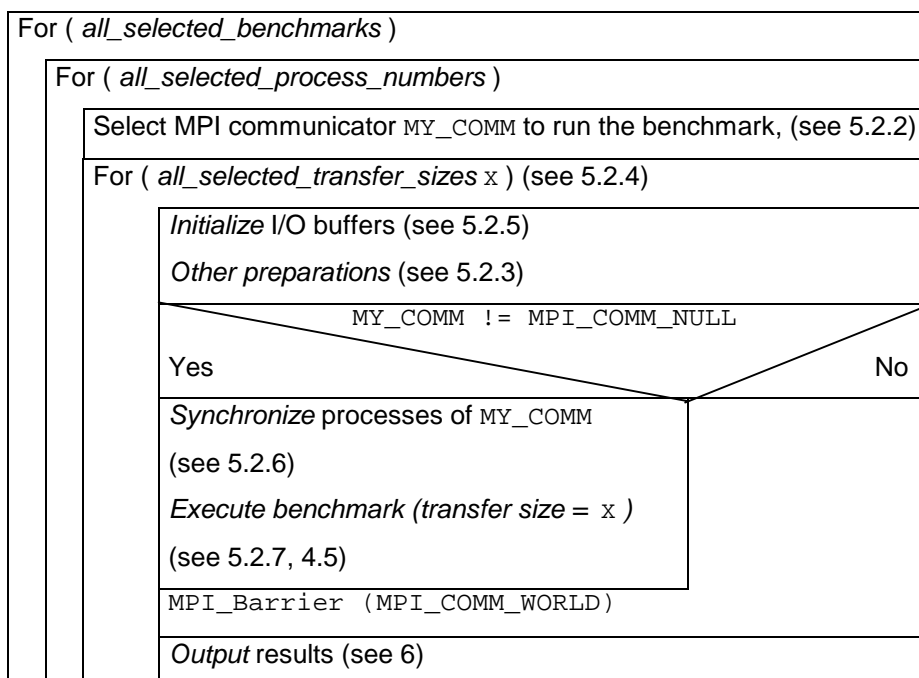


Figure 16: Control flow of PMB

The control parameters obviously necessary are either *command line arguments* (see 5.1) or parameter selections inside the PMB include files *settings.h* / *setting_io.h* (see 5.2).

5.1 Running PMB, Command Line Control

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

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

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

$P=1$ is allowed. Control arguments (in addition to P) can be passed to PMB 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 PMB-<...>
```

All benchmarks will run on $Q=[1, 2, 4, 8, \dots, \text{largest } 2^x < P, P]$ processes ($Q=1$ only for PMB-IO). E.g. $P=11$, then $Q=[1, 2, 4, 8, 11]$ processes will be selected. Single Transfer PMB-IO benchmarks will run only with $Q=1$, Single Transfer PMB-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 PMB-<...>
    [Benchmark1 [Benchmark2 [ ... ] ] ]
    [-npmin P_min]
    [-multi Outflag]
    [-input <Input_file>]
```

(where the 4 major [] may appear in any order).

– Examples:

```
mpirun -np 8 PMB-IO
mpirun -np 10 PMB-IO S_Write_indv P_IWrite_indv
mpirun -np 11 PMB-EXT -npmin 5
mpirun -np 14 PMB-IO P_Read_shared -npmin 7
mpirun -np 3 PMB-EXT -input PMB_SELECT_EXT
```

5.1.2.1 Benchmark Selection Arguments

A sequence of blank-separated strings, each being the name of one PMB-<...> 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 `2xP_min < P`, `P`.

5.1.2.3 -multi Outflag Selection

For selecting Multi/non Multi mode. Refer to the exact definition in [3].

5.1.2.4 -input <File> Selection

An ASCII input file is used to select the benchmarks to run, e.g. a file `PMB_SELECT_EXT` looking as follows:

```
#
# PMB benchmark selection file
#
# every line must be a comment (beginning with #), or it
# must contain exactly 1 PMB benchmark name
#
#Window
Unidir_Get
#Unidir_Put
#Bidir_Get
#Bidir_Put
Accumulate
```

By aid of this file,

```
mpirun .... PMB-EXT -input PMB_SELECT_EXT
```

would run PMB-EXT benchmarks `Unidir_Get` and `Accumulate`.

5.2 PMB Parameters and Hard-coded Settings

5.2.1 Parameters Controlling PMB

There are 9 parameters (set by preprocessor definition) controlling PMB. The definition is in the files

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

A complete list and explanation of the parameters is in Table 19 below.

It is important that (in contrast to PMB 2.0) PMB 2.2 allows for two sets of parameters: *standard* and *optional*. PMB-EXT uses the same include file as PMB-MPI1 [3]. Both include files are almost identical in structure, but differ in the standard settings. Here, we only look at `settings_io.h`. Note that some names contain `MSG` (for “message”), in consistency with `settings.h`.

Parameter (standard mode value)	Meaning
FILENAME (pmb_out)	base name of I/O files (see 5.2.3.2)
PMB_OPTIONAL (not set)	has to be set when user optional settings are to be activated
MINMSGLOG (0)	second smallest data transfer size is $2^{\text{MINMSGLOG}}$ (the smallest always being 0)
MAXMSGLOG (24)	largest transfer size is $2^{\text{MAXMSGLOG}}$ Sizes $0, 2^i$ ($i=\text{MINMSGLOG}, \dots, \text{MAXMSGLOG}$) are used
MSGSPERSAMPLE (50)	max. repetition count for all benchmarks, unless they run in non-aggregate mode
MSGSG_NONAGGR (10)	max. repetition count for non aggregate benchmarks
OVERALL_VOL (16 Mbytes)	for all sizes $< \text{OVERALL_VOL}$, the repetition count is eventually reduced so that not more than OVERALL_VOL bytes overall are processed. This avoids unnecessary repetitions for large transfer sizes. Finally, the real repetition count for transfer size X is $\text{MSGSPERSAMPLE} \ (X=0),$ $\min(\text{MSGSPERSAMPLE}, \max(1, \text{OVERALL_VOL}/X))$ $(X>0)$ Note: OVERALL_VOL does not restrict the size of the max. data transfer. $2^{\text{MAXMSGLOG}}$ is the largest size, independent of OVERALL_VOL
N_BARR (2)	Number of MPI_Barrier for synchronization (see 5.2.6)
TARGET_CPU_SECS (0.01)	CPU seconds (as float) to run concurrent with nonblocking benchmarks

Table 19: PMB parameters

Below a sample of file `settings_io.h` is shown. Here, `PMB_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 `PMB_OPTIONAL` is deactivated, the obvious standard mode values are taken.

Note:

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

```

#define FILENAME "pmb_out"
#define PMB_OPTIONAL
#ifndef PMB_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 16. 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 16.

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

5.2.3 Other Preparations

5.2.3.1 Window (PMB_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 (correction in 2.2 as compared to previous releases, March 2000).

5.2.3.2 File (PMB-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

PMB 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
```

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

5.2.3.4 View (PMB-IO)

The file view is the determined by the settings

- `disp = 0`
- `datarep = native`
- `etype`, filetype as defined in the single definitions in section 3.2
- `info` as defined in 5.2.3.3

5.2.4 Buffer Lengths

5.2.4.1 PMB-IO

Set in `settings_io.h` (see 5.2.1)

5.2.4.2 PMB-EXT

Set in `settings.h` (see 5.2.1)

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 7.3).

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

Currently, `int` is used for PMB-IO, `float` for PMB-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)
(PMB-EXT), and
#define BUF_VALUE(rank,i) 10000000*(1+rank)+i%10000000
(PMB-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 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 16) assure that all processes are synchronized. As with PMB-MPI1 [3], in PMB-EXT a so called “Warm up” phase is included.

5.2.7 The Actual Benchmark, Blocking Case

In order to reduce measurement errors caused by to insufficient clock resolution, every benchmark is run repeatedly. The repetition count for aggregate 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 `MSG_NONAGGR` (a reduced count is sensible as non aggregate runtimes are normally much longer).

In the following, *elementary transfer* means the pure function (`MPI_Put`, `MPI_Get`, `MPI_Accumulate`, `MPI_File_write_XX`, `MPI_File_read_XX`) as specified in section 3.2, 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.

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 The Actual Benchmark, Nonblocking I/O Case

As explained in 4.5, 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 5.2.7 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 style of output tables is basically as explained in [3]. PMB-EXT tables for `Unidir_put/Unidir_get` look like PingPong, `Bidir_put/Bidir_get` like PingPing tables, with the only difference that aggregate and non aggregate numbers are displayed. This in turn is done as in the case of PMB-IO Write tables, see below.

6.1 Table of a PMB-IO Write Benchmark

The next table shows the PMB output of `P_Write_indv` (obtained with an experimental workstation implementation). Note that automatically both aggregate and non aggregate mode results are displayed.

```
mpirun -np 2 PMB-IO P_write_indv -npmin 2
#-----
#   PALLAS MPI Benchmark Suite V2.1, MPI-IO part
#-----
# Date       : Thu Sep 10 13:16:59 1998
# Machine    : alpha# System      : OSF1
# Release    : V4.0
```

```
# Version      : 564
# Minimum io portion in bytes:  0
# Maximum io portion in bytes:  4194304

# !! Attention: results have been achieved in
# !! PMB_OPTIONAL mode.
# !! Results may differ from standard case.

# List of Benchmarks to run:

# P_Write_Indv
```

```
#-----
# Benchmarking P_Write_indv
# ( #processes = 2 )
#-----
#
#   MODE: AGGREGATE
#
#bytes #rep t_min[usec]  t_max[usec]   t_avg[usec] Mbytes/sec
      0  50   50082.99    61439.18    55761.09    0.00
      1  50   95786.38    99579.19    97682.79    0.00
      2  50   66040.80    77630.81    71835.80    0.00
      4  50  109813.40   111591.39   110702.40    0.00
      8  50  144057.20   167726.21   155891.70    0.00
     16  50  185798.60   209419.20   197608.90    0.00
     32  50  159754.40   169569.78   164662.09    0.00
... etc ...
4194304    4 1130361.74  1140056.73  1135209.23    3.51
```

```
#-----
#Benchmarking P_Write_indv
#( #processes = 2 )
#-----
#
#   MODE: NON-AGGREGATE
#
#bytes #reps  t_min[usec] t_max[usec]   t_avg[usec] Mbytes/sec
      0  10  186069.20   195866.20   190967.70    0.00
      1  10  250006.20   268192.01   259099.10    0.00
      2  10  232049.39   251983.62   242016.51    0.00
      4  10  278203.42   303294.18   290748.80    0.00
      8  10  410216.62   426004.39   418110.50    0.00
     16  10  352347.59   373439.22   362893.40    0.00
     32  10  366177.82   384650.61   375414.22    0.00
... etc ...
4194304    4 3970449.00  3978125.24  3974287.12    1.01
```


6.2 Table of a PMB-IO Nonblocking Benchmark

The following table shows the PMB output of a nonblocking benchmark (cf. 4.5.2). Note that in the beginning, the Mflop/s rate of the `CPU_Exploit` function is reported.

```
mpirun -np 2 PMB-IO S_iread_indv -npmin 2
#-----
#   PALLAS MPI Benchmark Suite V2.1, MPI-IO part
#-----
# Date       : Thu Sep 10 13:19:18 1998
# Machine    : alpha# System      : OSF1
# Release    : V4.0
# Version    : 564
#
# Minimum io portion in bytes:    0
# Maximum io portion in bytes:  4194304
#
# !! Attention: results have been achieved in
# !! PMB_OPTIONAL mode.
# !! Results may differ from standard case.
#

# List of Benchmarks to run:

# S_IRead_Indv

# For nonblocking benchmarks:

# Function CPU_Exploit obtains an undisturbed
# performance of    44.22 MFlops
```

```
#-----
# Benchmarking S_IRead_Indv
# ( #processes = 1 )
#-----
#
# MODE: AGGREGATE
#
#bytes #rep t_ovrl[usec] t_pure[usec] t_CPU[usec] overlap[%]
0 50 102299.59 7.80 96513.99 0.00
1 50 145626.00 21277.19 96513.99 0.00
2 50 132357.60 25307.20 96513.99 0.00
4 50 156007.60 38118.20 96513.99 0.00
8 50 147662.40 55407.81 96513.99 7.69
16 50 147389.41 46820.19 96513.99 0.00
32 50 161181.81 46291.80 96513.99 0.00
64 50 150327.61 43981.00 96513.99 0.00
128 50 139864.59 47158.60 96513.99 8.07
256 50 150371.38 38163.21 96513.99 0.00
512 50 145441.20 45654.20 96513.99 0.00
1024 50 158331.61 38216.40 96513.99 0.00
2048 50 139538.41 44165.80 96513.99 2.58
4096 50 157613.40 44234.61 96513.99 0.00
8192 50 150432.61 44018.82 96513.99 0.00
16384 50 150432.80 44603.80 96513.99 0.00
32768 50 155883.00 43982.60 96513.99 0.00
65536 50 156317.21 59924.20 96513.99 0.02
131072 50 161281.80 55277.59 96513.99 0.00
262144 50 166273.62 71014.60 96513.99 1.77
524288 32 220125.01 145095.80 96513.99 22.26
1048576 16 255813.98 209089.99 96513.99 51.59
2097152 8 386824.80 379491.19 96513.99 92.40
4194304 4 610285.49 580649.73 96513.99 69.29
```

7 Further Details

7.1 Memory and Disk Space Requirements

Benchmarks	Standard mode	Optional mode
	memory demand per process	memory demand per process ($X = 2^{\text{MAXMSGLOG}}$)
PMB-EXT	80 Mbytes	$2 \max(X, \text{OVERALL_VOL})$ bytes
PMB-IO	32 Mbytes	$2X$ bytes
	disk space overall	disk space overall
PMB-IO	16 Mbytes	$\max(X, \text{OVERALL_VOL})$ bytes

Table 20: Resource Requirements

7.2 SRC Directory

The following source files are on the directory:

PMB-MPI1 benchmark kernels:

PingPing.c, PingPong.c, Exchange.c, Sendrecv.c, Allgather.c, Allgatherv.c, Allreduce.c, Alltoall.c, Bcast.c, Reduce.c, Reduce_scatter.c, Barrier.c.

PMB-MPI2 benchmark kernels:

Window.c, OneS_accu, OneS_bidir.c, OneS_unidir.c, Write.c, Read.c, Open_Close.c.

Driver routines:

pmb.c, pmb_init.c, Output.c, BenchList.c, Warm_Up.c, declare.c, g_info.c, Err_Handler.c, strgs.c, Mem_Manager.c, chk_diff.c, Parse_Name_EXT.c, Parse_Name_IO.c, Parse_Name_MPI1.c.
Init_File.c, Init_Transfer.c, User_Set_Info.c, CPU_Exploit.c

Include files:

Benchmark.h, Comments.h, appl_errors.h, comm_info.h, declare.h, err_check.h, settings.h, settings_io.h, Bnames_EXT.h, Bnames_IO.h, Bnames_MPI1.h.

7.3 Results Checking

By activating the CPP flag

-DCHECK

a results check will be performed, similar to [3].

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

7.4 Use of MPI

Except its documented use in the benchmark kernels, MPI is used to the following extent:

```

MPI_Init
MPI_Bcast, MPI_Recv, MPI_Get_count, MPI_Send, MPI_Gather
MPI_Comm_size, MPI_Comm_rank, MPI_Comm_group,
MPI_Group_translate_ranks, MPI_Comm_split, MPI_Comm_free
MPI_Type_size, MPI_Type_struct, MPI_Type_commit,
MPI_Type_free
MPI_Info_get_nkeys, MPI_Info_get_nthkey,
MPI_Info_get_valuelen,
MPI_Info_get, MPI_Info_free
MPI_File_set_size
MPI_Error_string, MPI_Errhandler_create,
MPI_Win_create_errhandler,
MPI_File_create_errhandler, MPI_Errhandler_set,
MPI_Win_set_errhandler, MPI_File_set_errhandler,
MPI_Errhandler_free
MPI_Abort
MPI_Finalize

```

8 Revision History

Release No.	Date	Content/Changes	Related Software Releases
1.0	1997/12	draft definition	PMB 2.1
1.1	1998/05	complete definition	
1.2	1998/05	Minor textual changes	
2.0	omitted		
2.1	1998/09	Release No. adapted to PMB-MPI1 document Chapters 2 and 7 completed; "Optional Mode" chapter added (5.2.1)	PMB 2.1
2.1.1	1998/09	Table of contents, minor textual corrections	
2.2	2000/03	3.2 added, 4.3.6, 5.2.3.1 corrected	PMB 2.2

9 References

- 1 MPI: A Message-Passing Interface Standard. Message Passing Interface Forum, 1995
- 2 MPI-2: Extensions to the Message-Passing Interface. Message Passing Interface Forum, 1997
- 3 Pallas MPI Benchmarks - PMB, part MPI1