Pallas MPI Benchmarks - PMB, Part MPI-1
1 INTRODUCTION  

2 INSTALLATION AND QUICK START OF PMB-MPI1  
2.1 Download  
2.2 Installation  
2.3 Running PMB-MPI1  

3 OVERVIEW OF PMB-MPI1  
3.1 General  
3.2 The Benchmarks  
3.3 Version changes  
3.3.1 Version 2.1 vs. 2.0  
3.3.2 Version 2.2 vs. 2.1  
3.4 PMB-MPI1 vs. PMB1.x Definitions  
3.4.1 Changed Definitions  
3.4.2 Throughput Calculations  
3.4.3 Corrected Methodology  

4 PMB-MPI1 BENCHMARK DEFINITIONS  
4.1 Benchmark Classification  
4.1.1 Single Transfer Benchmarks  
4.1.2 Parallel Transfer Benchmarks  
4.1.3 Collective Benchmarks  
4.2 Definition of Single Transfer Benchmarks  
4.2.1 PingPong  
4.2.2 PingPing  
4.3 Definition of Parallel Transfer Benchmarks  
4.3.1 Sendrecv  
4.3.2 Exchange  
4.4 Definition of Collective Benchmarks  
4.4.1 Reduce  
4.4.2 Reduce_scatter  
4.4.3 Allreduce  
4.4.4 Allgather  
4.4.5 Allgatherv  
4.4.6 Alltoall  
4.4.7 Bcast  
4.4.8 Barrier  

5 BENCHMARK METHODOLOGY  
5.1 Running PMB, Command Line Control
5.1.1 Default Case 20
5.1.2 Command Line Control 20

5.2 PMB Parameters and Hard Coded Settings 22
5.2.1 Parameters Controlling PMB 22
5.2.2 Communicators, Active Processes 24
5.2.3 Message Lengths 24
5.2.4 Buffer Initialization 24
5.2.5 Warm-up Phase 25
5.2.6 Synchronization 25
5.2.7 The Actual Benchmark 25

6 OUTPUT 26
6.1 Sample 1 27
6.2 Sample 2 29
6.3 Sample 3 31

7 FURTHER DETAILS 33
7.1 Memory Requirements 33
7.2 SRC Directory 33
7.3 Results Checking 33
7.4 Use of MPI 34

8 REVISION HISTORY 34

REFERENCES 34
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 (the focus of this document)
- PMB-MPI2 (see [3]), subdivided into
  - 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.

This document is dedicated to PMB-MPI1.

Section 2 is a brief installation guide, in section 3 an overview of the suite is given.

Section 4 defines the single benchmarks in detail. PMB 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.

Section 5 defines the methodology and rules of PMB, section 6 shows the output tables format. In section 7, further important details are explained, in particular a results checking mode for PMB.
2 Installation and Quick Start of PMB-MPI1

In order to run PMB-MPI1, one needs:

- `cpp`, ANSI C compiler, `make`.
- Full MPI-1 installation, including startup mechanism for parallel MPI programs.

See 7.1 for the memory requirements of PMB-MPI1.

2.1 Download

Get `PMB.tar.gz` at `http://www.pallas.de/pages/pmbd.htm`

2.2 Installation

After unpacking, on the current directory is created:

- `PMB2` (directory)
- `PMB2/SRC` (subdirectory containing sources and `Makefile`, see 7.2)
- `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` will perform the installation.

For defining the machine dependent settings, the section

```plaintext
##### 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`:
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.
2.3 Running PMB-MPI1

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

```bash
mpirun -np P PMB-MPI1
```

(\(P\) being the number of processes to load; \(P=1\) is allowed!). This will run all of PMB on a varying number of processes \((2, 4, 8, \ldots, 2^\times P, P)\) and output results on stdout. Also is possible

```bash
mpirun -np P PMB-MPI1 [Benchmark names]
```

where the names are one or more of \(\text{PingPong, PingPing, Sendrecv, Exchange, Reduce, Reduce\_scatter, Allreduce, Bcast, Allgather, Allgatherv, Alltoall, Barrier}\). The selected benchmarks will run, their meaning should be clear to MPI experts.

For the details, see 5.1.

3 Overview of PMB-MPI1

3.1 General

The idea of PMB 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.

PMB has a standard and an optional configuration. In the standard case, all parameters mentioned above are fixed and must not be changed.

For certain systems, it may be interesting to extend the results tables (in particular, run larger message sizes than provided in the standard case). For this, the user can set certain parameters at own choice. See 5.2.1.

The minimum \(P\_\text{min}\) and maximum number \(P\) of processes can be selected by the user via command line, the benchmarks run on \(P\_\text{min}, 2P\_\text{min}, 4P\_\text{min}, \ldots, 2^\times P\_\text{min}<P\) and \(P\) processes. See chapter 5 for the details.

3.2 The Benchmarks

The current version of PMB-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 4. Section 5 describes the benchmark methodology.

PMB-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, 6.3.

3.3 Version changes

3.3.1 Version 2.1 vs. 2.0
• Alltoall added (see 4.4.6)  
• Optional settings mode included (see 5.2.1.)

3.3.2 Version 2.2 vs. 2.1
Default variable initializations (function set_default) were added (March 2000).

3.4 PMB-MPI1 vs. PMB1.x Definitions

Compared to older PMB1.x releases, all primary benchmark names except Sendrecv and Alltoall are the same. Cshift and Xover of PMB1.x have been removed. PMB1.x did not support the Multi versions.
Most important is that certain definitions have changed. Please check carefully.

Table 2 shows an overview of the changes.

### 3.4.1 Changed Definitions

The main changes are in **PingPing** and **Exchange**. **PingPing** even uses a different pattern (elementary messages rather than **Sendrecv**, see 4.3.1). Moreover, the scaling of timings and throughput data has changed.

<table>
<thead>
<tr>
<th></th>
<th>PMB-MPI1</th>
<th>PMB1.x</th>
<th>PMB-MPI1</th>
<th>PMB1.x</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PingPing</strong></td>
<td>1</td>
<td>0.5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td><strong>Exchange</strong></td>
<td>1</td>
<td>0.25</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 1: Scaling factors PMB-MPI1 vs. PMB1.x

Thus, the corresponding tables show different values when comparing PMB1.x and PMB-MPI1 on a particular system.

The PMB1.x scaling factors for the timings gives confusing answers for the startup components (small message sizes). The scaling of **PingPing** throughputs by 2 is reasonable (bi-directional throughput), however PMB imposes a different interpretation, leaving the bi-directional throughput measurement to the **Sendrecv** benchmark.

**Sendrecv** and **Alltoall** are new benchmarks. Functionally, **Sendrecv** exactly corresponds to the **Cshift** benchmark of PMB1.x, however, displays timings with a different scaling. The **Xover** benchmark of PMB1.x has been removed, as it has shown no significant information on any tested system.

### 3.4.2 Throughput Calculations

Throughput results are based on real MBytes (1048576 bytes) in PMB-MPI1, in contrast to PMB1.x, which used 1 MByte = 1000000 bytes.

In contrast to PMB1.x, PMB-MPI1 does not display throughput values for the global operations **Bcast**, **Allgather** and **Allgatherv**.

### 3.4.3 Corrected Methodology

PMB1.x was not cleanly defined in the case that a certain benchmark was run in a process group strictly smaller than the group of all started MPI processes.

In PMB-MPI1, all non active processes will wait for the active ones in an **MPI_Barrier(MPI_COMM_WORLD)**.

In PMB1.x, non active processes immediately went through the output-collecting phase (**MPI_Gather**) and then, eventually, switched to the following benchmark(s). This may induce unpredictable obstructions of the active processes. The **MPI_Barrier** may also, but now the way is well defined and reasonable.

See 5 for precise definitions of the methodology.
Table 2: PMB-MPI1 vs. PMB1.x benchmarks

<table>
<thead>
<tr>
<th>PMB-MPI1 Benchmark name</th>
<th>Contained in releases 1.x</th>
<th>Compared to PMB-1.x, in PMB-MPI1 there is</th>
</tr>
</thead>
<tbody>
<tr>
<td>PingPong</td>
<td>×</td>
<td>slight change in throughput data due to re-definition of 1MByte = 1048576 bytes</td>
</tr>
<tr>
<td>PingPing</td>
<td>×</td>
<td>other pattern, no scaling expectation: timings doubled, throughputs halved</td>
</tr>
<tr>
<td>Sendrecv</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Exchange</td>
<td>×</td>
<td>4 fold timings, equal throughputs</td>
</tr>
<tr>
<td>Bcast</td>
<td>×</td>
<td>no output of throughput data</td>
</tr>
<tr>
<td>Allgather</td>
<td>×</td>
<td>no output of throughput data</td>
</tr>
<tr>
<td>Allgatherv</td>
<td>×</td>
<td>no output of throughput data</td>
</tr>
<tr>
<td>Alltoall</td>
<td>×</td>
<td>no change</td>
</tr>
<tr>
<td>Reduce</td>
<td>×</td>
<td>no change</td>
</tr>
<tr>
<td>Reduce_scatter</td>
<td>×</td>
<td>no change</td>
</tr>
<tr>
<td>Allreduce</td>
<td>×</td>
<td>no change</td>
</tr>
<tr>
<td>Allreduce</td>
<td>×</td>
<td>no change</td>
</tr>
<tr>
<td>Barrier</td>
<td>×</td>
<td>no change</td>
</tr>
</tbody>
</table>

PMB1.x benchmarks that are no longer in PMB-MPI1

| Cshift                  | Sendrecv benchmark is a full substitute |
| Xover                   | Has shown no significant information   |

4 PMB-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.

4.1 Benchmark Classification

For a clear structuring of the set of benchmarks, PMB now 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 PMB. Also holds this classification hold for the PMB-MPI2 part [3].
4.1.1 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 it. In PMB 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, roughly speaking, are local mode. The particular pattern is purely local to the participating processes, there is no concurrency with other message passing activity. Best case message passing results are to be expected. Important for this is that single transfer benchmarks only run with 2 active processes (see 3.4.3, 5.2.2 for the definition of active).

For PingPing, and this is in contrast to PMB1.x and other code systems containing this benchmark, 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 4.2.1 and 4.2.2 for exact definitions.

4.1.2 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 4.3.1 and 4.3.2 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

<table>
<thead>
<tr>
<th>PMB-MPI1</th>
<th>Single Transfer</th>
<th>Parallel Transfer</th>
<th>Collective</th>
</tr>
</thead>
<tbody>
<tr>
<td>PingPong</td>
<td>Sendrecv</td>
<td></td>
<td>Bcast</td>
</tr>
<tr>
<td>PingPing</td>
<td>Exchange</td>
<td></td>
<td>Allgather</td>
</tr>
<tr>
<td></td>
<td>Multi-PingPong</td>
<td></td>
<td>Alltoall</td>
</tr>
<tr>
<td></td>
<td>Multi-PingPing</td>
<td></td>
<td>Reduce</td>
</tr>
<tr>
<td></td>
<td>Multi-Sendrecv</td>
<td></td>
<td>Reduce_scatter</td>
</tr>
<tr>
<td></td>
<td>Multi-Exchange</td>
<td></td>
<td>Allreduce</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Barrier</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Multi-versions of these</td>
</tr>
</tbody>
</table>
4.1.3 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.

4.2 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 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 MB/s = \(2^{20}\) bytes/sec scale (i.e. \(\text{throughput} = \frac{X}{2^{20} \times 10^6} / \text{time} = \frac{X}{1.048576} / \text{time}, \text{when time is in \(\mu\)sec}.\)
4.2.1 PingPong

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

<table>
<thead>
<tr>
<th>Measured pattern</th>
<th>As symbolized between [\text{\texttt{MPI_Send}}, \text{\texttt{MPI_Recv}}] in Figure 1; two active processes only ((Q=2), see 5.2.2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>based on</td>
<td>\text{MPI_Send}, \text{MPI_Recv}</td>
</tr>
<tr>
<td>MPI_Datatype</td>
<td>\text{MPI_BYTE}</td>
</tr>
<tr>
<td>reported timings</td>
<td>\text{MPI_Send, MPI_Recv}</td>
</tr>
<tr>
<td>reported throughput</td>
<td>\text{MPI_Send, MPI_Recv}</td>
</tr>
</tbody>
</table>

![Diagram of PingPong pattern](image)

Figure 1: PingPong pattern
4.2.2 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.

<table>
<thead>
<tr>
<th>Measured pattern</th>
<th>As symbolized between ▼——▼ in Figure 2; two active processes only (Q=2, 5.2.2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>based on</td>
<td>MPI_Isend/MPI_Wait, MPI_Recv</td>
</tr>
<tr>
<td>MPI_Datatype</td>
<td>MPI_BYTE</td>
</tr>
<tr>
<td>reported timings</td>
<td>time = Δt (in μsec) as indicated in Figure 2</td>
</tr>
<tr>
<td>reported throughput</td>
<td>X/1.048576/time</td>
</tr>
</tbody>
</table>

![Diagram of PingPing pattern]

**Figure 2:** PingPing pattern
4.3 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 Section 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 $nmsg$ of messages outgoing from or incoming at a particular process. In the Sendrecv benchmark, a particular process sends and receives $x$ bytes, the turnover is $2x$ bytes, $nmsg=2$. In the Exchange case, we have $4x$ bytes turnover, $nmsg=4$.

Throughput values are defined in $\text{MBytes/sec} = 2^{20} \text{ bytes / sec scale}$ (i.e. $\text{throughput} = nmsg \times X / 2^{20} \times 10^6 / \text{time} = nmsg \times X / 1.048576 / \text{time}$, when time is in $\mu\text{sec}$).
4.3.1 **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 PMB1.x. For 2 processes, it will report the bi-directional bandwidth of the system, as obtained by the (optimized) `MPI_Sendrecv` function.

<table>
<thead>
<tr>
<th>Measured pattern</th>
<th>As symbolized between</th>
</tr>
</thead>
<tbody>
<tr>
<td>based on</td>
<td><code>MPI_Sendrecv</code></td>
</tr>
<tr>
<td><code>MPI_Datatype</code></td>
<td><code>MPI_BYTE</code></td>
</tr>
<tr>
<td>reported timings</td>
<td><code>time = Δt</code> (in µsec) as indicated</td>
</tr>
<tr>
<td>reported throughput</td>
<td><code>2X/1.048576/time</code></td>
</tr>
</tbody>
</table>

![Figure 3: Sendrecv pattern](attachment:image.png)
4.3.2 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.

<table>
<thead>
<tr>
<th>Measured pattern</th>
<th>As symbolized between</th>
</tr>
</thead>
<tbody>
<tr>
<td>based on MPI_Isend/MPI_Waitall, MPI_Recv</td>
<td>in Figure 4</td>
</tr>
<tr>
<td>MPI_Datatype</td>
<td>MPI_BYTE</td>
</tr>
<tr>
<td>reported timings</td>
<td>time = $\Delta t$ (in $\mu$s) as indicated in Figure 4</td>
</tr>
<tr>
<td>reported throughput</td>
<td>4X/1.048576/time</td>
</tr>
</tbody>
</table>

Figure 4: Exchange pattern

4.4 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.
4.4.1 Reduce

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

The root of the operation is changed cyclically, see 5.2.7

See also the remark in the end of 4.1.3.

<table>
<thead>
<tr>
<th>measured pattern</th>
<th>MPI_Reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Datatype</td>
<td>MPI_FLOAT</td>
</tr>
<tr>
<td>MPI_Op</td>
<td>MPI_SUM</td>
</tr>
<tr>
<td>root</td>
<td>changing</td>
</tr>
<tr>
<td>reported timings</td>
<td>bare time</td>
</tr>
<tr>
<td>reported throughput</td>
<td>none</td>
</tr>
</tbody>
</table>

4.4.2 Reduce_scatter

Benchmark of the MPI_Reduce_scatter function. Reduces a vector of length $L = X/\text{sizeof(float)}$ 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 $np = \#\text{processes}$, $L = r*np+s (s = L \mod np)$,

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 4.1.3.

<table>
<thead>
<tr>
<th>measured pattern</th>
<th>MPI_Reduce_scatter</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Datatype</td>
<td>MPI_FLOAT</td>
</tr>
<tr>
<td>MPI_Op</td>
<td>MPI_SUM</td>
</tr>
<tr>
<td>reported timings</td>
<td>bare time</td>
</tr>
<tr>
<td>reported throughput</td>
<td>none</td>
</tr>
</tbody>
</table>

4.4.3 Allreduce

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

See also the remark in the end of 4.1.3.

<table>
<thead>
<tr>
<th>measured pattern</th>
<th>MPI_Allreduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Datatype</td>
<td>MPI_FLOAT</td>
</tr>
<tr>
<td>MPI_Op</td>
<td>MPI_SUM</td>
</tr>
<tr>
<td>reported timings</td>
<td>bare time</td>
</tr>
<tr>
<td>reported throughput</td>
<td>none</td>
</tr>
</tbody>
</table>
4.4.4 Allgather

Benchmark of the MPI_Allgather function. Every process inputs $x$ bytes and receives the gathered $x \times (#\text{processes})$ bytes.

<table>
<thead>
<tr>
<th>Measured pattern</th>
<th>MPI_Allgather</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Datatype</td>
<td>MPI_BYTE</td>
</tr>
<tr>
<td>reported timings</td>
<td>bare time</td>
</tr>
<tr>
<td>reported throughput</td>
<td>none</td>
</tr>
</tbody>
</table>

4.4.5 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.

<table>
<thead>
<tr>
<th>Measured pattern</th>
<th>MPI_Allgatherv</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Datatype</td>
<td>MPI_BYTE</td>
</tr>
<tr>
<td>reported timings</td>
<td>bare time</td>
</tr>
<tr>
<td>reported throughput</td>
<td>none</td>
</tr>
</tbody>
</table>

4.4.6 Alltoall

Benchmark of the MPI_Alltoall function. Every process inputs $X \times (#\text{processes})$ bytes ($X$ for each process) and receives $X \times (#\text{processes})$ bytes ($X$ from each process).

<table>
<thead>
<tr>
<th>Measured pattern</th>
<th>MPI_Alltoall</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Datatype</td>
<td>MPI_BYTE</td>
</tr>
<tr>
<td>reported timings</td>
<td>bare time</td>
</tr>
<tr>
<td>reported throughput</td>
<td>none</td>
</tr>
</tbody>
</table>

4.4.7 Bcast

Benchmark of MPI_Bcast. A root process broadcasts $x$ bytes to all. The root of the operation is changed cyclically, see 5.2.7.

<table>
<thead>
<tr>
<th>measured pattern</th>
<th>MPI_Bcast</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Datatype</td>
<td>MPI_BYTE</td>
</tr>
<tr>
<td>root</td>
<td>changing</td>
</tr>
<tr>
<td>reported timings</td>
<td>bare time</td>
</tr>
<tr>
<td>reported throughput</td>
<td>none</td>
</tr>
</tbody>
</table>
4.4.8 Barrier

<table>
<thead>
<tr>
<th>measured pattern</th>
<th>MPI_Barrier</th>
</tr>
</thead>
<tbody>
<tr>
<td>reported timings</td>
<td>bare time</td>
</tr>
<tr>
<td>reported throughput</td>
<td>none</td>
</tr>
</tbody>
</table>

5 Benchmark Methodology

Recall that in chapter 4 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 message size.

The following graph shows an overview of the flow of control inside PMB. All emphasized items will be explained in more detail.

```
For ( all_selected_benchmarks )
   For ( all_selected_process_numbers )
      Select MPI communicator MY_COMM to run the benchmark, (see 5.2.2)
      For ( all_selected_message_lengths X ) (see 5.2.3)
        Initialize communication buffers (see 5.2.4)
        X == first_selected_message_length
          Yes
          No
          Warm_up (see 5.2.5)
            Yes
            No
            Synchronize processes of MY_COMM (see 5.2.6)
            Execute benchmark (message size = X ) (see 5.2.7)
            MPI_Barrier (MPI_COMM_WORLD)
            Output results (see 6)

Figure 5: Control flow in PMB
```

The control parameters obviously necessary are either command line arguments (see 5.1) or parameter selections inside the PMB include file settings.h (see 5.2).
5.1 Running PMB, Command Line Control

After installation, see 2.2, an executable PMB-MPI1 should exist.

Given \( P \), the (normally user selected) number of MPI processes to run PMB-MPI1, a startup procedure has to load parallel PMB-MPI1. Lets assume, for sake of simplicity, that this done by

\[
\text{mpirun} \ -\text{np} \ P \ \text{PMB-MPI1} \ \text{[arguments]}
\]

\( P=1 \) is allowed, will be ignored only by Single Transfer benchmarks. Control arguments (in addition to \( P \)) can be passed to PMB-MPI1 via \((\text{argc,argv})\) which will be read by PMB-MPI1 process 0 (in \text{MPI_COMM_WORLD} ranking) and then distributed to all processes.

5.1.1 Default Case

Just invoke

\[
\text{mpirun} \ -\text{np} \ P \ \text{PMB-MPI1}
\]

All primary (non Multi) benchmarks will run on \( Q=2, 4, 8, \ldots \), largest \( 2^P, P \) processes

(E.g \( P=11 \), then 2, 4, 8, 11 processes will be selected). The \( Q= P \) processes running the benchmark are called active processes. A communicator is formed out of a group of \( Q \) processes, see 5.2.2., which is used as communicator argument to the MPI functions crucial for the benchmark.

5.1.2 Command Line Control

The general syntax is

\[
\text{mpirun} \ -\text{np} \ P \ \text{PMB-MPI1} \\
\quad \text{[Benchmark1 [Benchmark2 [ ... ] ]]} \\
\quad [-\text{npmin} \ P_{\text{min}}] \\
\quad [-\text{multi} \ \text{Outflag}] \\
\quad [-\text{input} <\text{File}>]
\]

(where the 4 major \([\text{ ]}\) may appear in any order).

Examples:

\[
\begin{align*}
\text{mpirun} \ &-\text{np} \ 8 \ \text{PMB-MPI1} \\
\text{mpirun} \ &-\text{np} \ 10 \ \text{PMB-MPI1} \ \text{PingPing} \ \text{Reduce} \\
\text{mpirun} \ &-\text{np} \ 11 \ \text{PMB-MPI1} \ -\text{npmin} \ 5 \\
\text{mpirun} \ &-\text{np} \ 4 \ \text{PMB-MPI1} \ -\text{npmin} \ 4 \ -\text{input} \ \text{PMB\_SELECT\_MPI1} \\
\text{mpirun} \ &-\text{np} \ 14 \ \text{PMB-MPI1} \ -\text{multi} \ 0 \ \text{PingPong} \ \text{Barrier} \\
&\quad \ -\text{npmin} \ 7
\end{align*}
\]

5.1.2.1 Benchmark Selection Arguments

A set of blank-separated strings, each being the name of one primary (non Multi) PMB benchmark (in exact spelling, case insensitive).

Default (no benchmark selection): select all primary benchmark names.

Given a name selection, either
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): as P_min = 2

Given P_min, the selected process numbers are
Q=P_min, 2P_min, 4P_min, ..., largest 2^n P_min <P, P.

Exception: Single Transfer benchmarks will only run on Q=2 and ignore P_min when P_min ≠ 2.

Now, running on a subset of Q<=P processes means that a communicator with a group of Q active processes is formed (or eventually several such communicators in the Multi cases), see 5.2.2. This communicator is used as argument to the MPI functions crucial for the benchmark.

5.1.2.3 –multi Outflag Selection

–multi activates the Multi versions of the benchmarks. 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.

See also 6.2, 6.3.
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_MPI1 looking as follows:

```
# # PMB benchmark selection file #
# every line must be a comment (beginning with #), or it # must contain exactly 1 PMB benchmark name # # PingPong PingPing Allreduce # Alltoall
```

mpirun .... PMB-MPI1 -input PMB_SELECT_MPI1
would run benchmarks PingPing and Allreduce.

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

A complete list and explanation of the parameters is in Figure 6 below.

Only settings.h is relevant here. It is important that (in contrast to PMB 2.0) PMB 2.2 allows for two sets of parameters: standard and optional.
Figure 6: PMB parameters

<table>
<thead>
<tr>
<th>Parameter (standard mode value)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMB_OPTIONAL (not set)</td>
<td>has to be set when user optional settings are to be activated</td>
</tr>
<tr>
<td>MINMSGLOG (0)</td>
<td>second smallest data transfer size is max(unit, $2^{\text{MINMSGLOG}}$) (the smallest always being 0), where unit = sizeof(float) for reductions, unit = 1 else</td>
</tr>
<tr>
<td>MAXMSGLOG (22)</td>
<td>largest message size is $2^{\text{MAXMSGLOG}}$ Sizes 0, $2^i$ ($i=\text{MINMSGLOG},...,\text{MAXMSGLOG}$) are used</td>
</tr>
<tr>
<td>MSGSPERSAMPLE (1000)</td>
<td>max. repetition count for all PMB-MPI1 benchmarks</td>
</tr>
<tr>
<td>MSGS_NONAGGR (100)</td>
<td>max. repetition count for non aggregate benchmarks (cf. [3], irrelevant for PMB-MPI1)</td>
</tr>
<tr>
<td>OVERALL_VOL (40 MBytes)</td>
<td>for all sizes &lt; OVERALL_VOL, the repetition count is eventually reduced so that not more than OVERALL_VOL bytes overall are processed. This avoids unnecessary repetitions for large message sizes. Finally, the real repetition count for message size X is MSGSPERSAMPLE (X=0), min(MSGSPERSAMPLE, max(1, OVERALL_VOL/X)) (X&gt;0)</td>
</tr>
<tr>
<td>NOTE: OVERALL_VOL does not restrict the size of the max. data transfer. $2^{\text{MAXMSGLOG}}$ is the largest size, independent of OVERALL_VOL</td>
<td></td>
</tr>
<tr>
<td>N_WARMUP (2)</td>
<td>Number of Warmup sweeps (see 5.2.5)</td>
</tr>
<tr>
<td>N_BARR (2)</td>
<td>Number of MPI_Barrier for synchronization (see 5.2.6)</td>
</tr>
<tr>
<td>TARGET_CPU_SECS (0.01)</td>
<td>CPU seconds (as float) to run concurrent with nonblocking benchmarks (currently irrelevant for PMB-MPI1)</td>
</tr>
</tbody>
</table>

Figure 7 below shows a sample of file settings.h. Here, PMB_OPTIONAL is set, so that user defined parameters are used. Message sizes 8 and 16 MBytes 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.*
Communicator management is repeated in every select `MY_COMM` step in Figure 5. If exists, the previous communicator is freed. Given \( Q \leq P \) as in 5.1.2.2, subcommunicators are formed out of the groups consisting of the MPI_COMM_WORLD ranks

\[ \{0, \ldots, Q-1\} \quad \text{(non Multi case)}, \]
\[ \{0, \ldots, Q-1\}, \{Q, \ldots, 2Q-1\} \ldots \quad \text{(Multi case)}. \]

All processes belonging to such a group are called *active* processes, the corresponding communicator is called `MY_COMM` in Figure 5. It is used as communicator argument to the MPI functions defining the pattern. All non active processes get `MY_COMM = MPI_COMM_NULL`.

### 5.2.3 Message Lengths
Set in `settings.h`, see 5.2.1

### 5.2.4 Buffer Initialization
Communication buffers are dynamically allocated as `void*` and used as MPI_BYTE buffers for all (non reduction) benchmarks. See 7.1 for an estimate of the memory requirement. To assign the buffer contents, a cast to an assignment type is performed. On the one hand, a sensible datatype is mandatory for reduction benchmarks. 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. Currently, `float` is selected for PMB-MPI1 (as this is sensible for reductions). The values are set by a CPP macro, currently

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

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

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

where rank is the MPI rank of the calling process.

### 5.2.5 Warm-up Phase

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.

### 5.2.6 Synchronization

Before the actual benchmark, `N_BARR` (defined in settings.h, see 5.2.1) many `MPI_Barrier(MY_COMM)` (ref. Figure 5) are used for process synchronization.

### 5.2.7 The Actual Benchmark

In order to reduce inaccuracies due to insufficient clock resolutions, every benchmark is run repeatedly. The repetition count is `MSGSPERSAMPLE` (constant defined in settings.h, see 5.2.1). In order to avoid an excessive run time in case of large message lengths `X`, an upper bound is set to `OVERALL_VOL / X` (OVERALL_VOL defined in settings.h). Finally,

```
n_sample = MSGSPERSAMPLE (X=0)
```

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

is the repetition count for all benchmarks, given message size `X`. Now, the key measurement is performed according to

```c
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
```

Important to stress is that `execute MPI pattern` really means the pure pattern as specified in 4, without any further function call. (Bcast and Reduce need a root process for their operation. In both cases, the root process is changed so that in iteration `i` the root rank is `i%(#processes in group)`. This is an additional integer operation inside the loop, looked upon as negligible.)

The communicator argument to the `MPI_XX` functions constituting the pattern is as defined in 5.2.2.
6 Output

Most easily, output is explained by sample outputs, see the tables below (generated with the previous version 2.1, looking identical with 2.2). What one sees is the following.

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

```c
#include <sys/utsname.h>
void 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], \( T = \text{Tmax}[\mu\text{sec}] \), bandwidth = \( X / 1.048576 / T \)  
  **Parallel Transfer Benchmarks:**  
  Display \( X = \) message size, \( T_{\text{max}}, T_{\text{min}} \) and \( T_{\text{avg}} \), bandwidth based on \( T_{\text{max}} \)  
  **Collective Benchmarks:**  
  Display \( X = \) message size (except for \text{Barrier}), \( T_{\text{max}}, T_{\text{min}} \) and \( T_{\text{avg}} \)

- **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

mpirun -np 2 PMB-MPI1 PingPong Allreduce
#---------------------------------------------------
#    PALLAS MPI Benchmark Suite V2.1, MPI-1 part
#---------------------------------------------------
# Date       : Thu Sep 10 10:22:58 1998
# Machine    : alpha# System     : OSF1
# Release    : V4.0
# Version    : 564
#
#
# 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 )
#---------------------------------------------------

<table>
<thead>
<tr>
<th>bytes</th>
<th>repetitions</th>
<th>t[usec]</th>
<th>Mbytes/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>4.51</td>
<td>0.00</td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td>5.41</td>
<td>0.18</td>
</tr>
<tr>
<td>2</td>
<td>1000</td>
<td>5.41</td>
<td>0.35</td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>5.41</td>
<td>0.70</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>5.41</td>
<td>1.41</td>
</tr>
<tr>
<td>16</td>
<td>1000</td>
<td>5.00</td>
<td>3.05</td>
</tr>
<tr>
<td>32</td>
<td>1000</td>
<td>5.84</td>
<td>5.23</td>
</tr>
<tr>
<td>64</td>
<td>1000</td>
<td>8.34</td>
<td>7.32</td>
</tr>
<tr>
<td>128</td>
<td>1000</td>
<td>8.76</td>
<td>13.94</td>
</tr>
<tr>
<td>256</td>
<td>1000</td>
<td>9.59</td>
<td>25.46</td>
</tr>
<tr>
<td>512</td>
<td>1000</td>
<td>12.51</td>
<td>39.03</td>
</tr>
<tr>
<td>1024</td>
<td>1000</td>
<td>18.35</td>
<td>53.22</td>
</tr>
<tr>
<td>2048</td>
<td>1000</td>
<td>30.44</td>
<td>64.16</td>
</tr>
<tr>
<td>4096</td>
<td>1000</td>
<td>57.55</td>
<td>67.88</td>
</tr>
<tr>
<td>bytes</td>
<td>repetitions</td>
<td>t_min[usec]</td>
<td>t_max[usec]</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
<td>1.43</td>
<td>1.46</td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>25.85</td>
<td>25.85</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>26.69</td>
<td>26.69</td>
</tr>
<tr>
<td>16</td>
<td>1000</td>
<td>26.69</td>
<td>26.69</td>
</tr>
<tr>
<td>32</td>
<td>1000</td>
<td>27.52</td>
<td>27.52</td>
</tr>
<tr>
<td>64</td>
<td>1000</td>
<td>31.69</td>
<td>31.69</td>
</tr>
<tr>
<td>128</td>
<td>1000</td>
<td>32.53</td>
<td>32.53</td>
</tr>
<tr>
<td>256</td>
<td>1000</td>
<td>35.86</td>
<td>35.86</td>
</tr>
<tr>
<td>512</td>
<td>1000</td>
<td>42.53</td>
<td>42.53</td>
</tr>
<tr>
<td>1024</td>
<td>1000</td>
<td>62.55</td>
<td>62.55</td>
</tr>
<tr>
<td>2048</td>
<td>1000</td>
<td>98.25</td>
<td>98.25</td>
</tr>
<tr>
<td>4096</td>
<td>1000</td>
<td>180.54</td>
<td>180.54</td>
</tr>
<tr>
<td>8192</td>
<td>1000</td>
<td>309.07</td>
<td>309.07</td>
</tr>
<tr>
<td>16384</td>
<td>1000</td>
<td>630.30</td>
<td>630.30</td>
</tr>
<tr>
<td>32768</td>
<td>1000</td>
<td>1180.77</td>
<td>1180.77</td>
</tr>
<tr>
<td>65536</td>
<td>640</td>
<td>2571.14</td>
<td>2571.14</td>
</tr>
<tr>
<td>131072</td>
<td>320</td>
<td>4929.39</td>
<td>4929.39</td>
</tr>
<tr>
<td>262144</td>
<td>160</td>
<td>9909.67</td>
<td>9909.67</td>
</tr>
<tr>
<td>524288</td>
<td>80</td>
<td>20586.26</td>
<td>20596.66</td>
</tr>
<tr>
<td>1048576</td>
<td>40</td>
<td>46326.08</td>
<td>46326.08</td>
</tr>
<tr>
<td>2097152</td>
<td>20</td>
<td>105728.35</td>
<td>105728.35</td>
</tr>
<tr>
<td>4194304</td>
<td>10</td>
<td>235253.11</td>
<td>235253.11</td>
</tr>
</tbody>
</table>
6.2 Sample 2

```bash
mpirun -np 7 PMB-MPI1 reduce -npmin 3 -multi 0
(PMB_OPTIONAL mode)
```

#---------------------------------------------------
# PALLAS MPI Benchmark Suite V2.1, MPI-1 part
#---------------------------------------------------
# Date       : Thu Sep 10 10:30:49 1998
# Machine    : alpha# System     : OSF1
# Release    : V4.0
# Version    : 564
#
# # Minimum message length in bytes:   0
# Maximum message length in bytes:   1024
# #
# # MPI_Datatype                   :   MPI_BYTE
# MPI_Datatype for reductions      :   MPI_FLOAT
# MPI_Op                         :   MPI_SUM
#
# #
# # !! Attention: results have been achieved in
# # !! PMB_OPTIONAL mode.
# # !! Results may differ from standard case.
# #
# # List of Benchmarks to run:
#
# (Multi-)Reduce

#-------------------------------------------------------------
# Benchmarking Multi-Reduce
# ( 2 groups of 3 processes each running simultaneous )
# Group    0                     : 0 1 2
# Group    1                     : 3 4 5
# ( 1 additional process waiting in MPI_Barrier)
#-------------------------------------------------------------
#bytes #repetitions  t_min[usec]  t_max[usec]  t_avg[usec]
0   1000       0.64         0.83         0.77
4   1000       142.84       145.34       143.54
8   1000       141.61       142.44       141.75
16  1000       141.61       142.44       142.03
32  1000       142.44       143.28       142.72
64  1000       161.66       161.66       161.66
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>1000</td>
<td>164.30</td>
<td>164.30</td>
</tr>
<tr>
<td>256</td>
<td>1000</td>
<td>173.47</td>
<td>174.31</td>
</tr>
<tr>
<td>512</td>
<td>1000</td>
<td>197.22</td>
<td>198.89</td>
</tr>
<tr>
<td>1024</td>
<td>1000</td>
<td>246.03</td>
<td>246.03</td>
</tr>
</tbody>
</table>

# Benchmarking Reduce
# ( #processes = 6 )
# ( 1 additional process waiting in MPI_Barrier)

<table>
<thead>
<tr>
<th>#bytes</th>
<th>#repetitions</th>
<th>t_min[usec]</th>
<th>t_max[usec]</th>
<th>t_avg[usec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>0.64</td>
<td>0.83</td>
<td>0.78</td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>246.64</td>
<td>247.47</td>
<td>246.92</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>232.96</td>
<td>233.79</td>
<td>233.65</td>
</tr>
<tr>
<td>16</td>
<td>1000</td>
<td>188.43</td>
<td>189.26</td>
<td>188.57</td>
</tr>
<tr>
<td>32</td>
<td>1000</td>
<td>169.73</td>
<td>169.73</td>
<td>169.73</td>
</tr>
<tr>
<td>64</td>
<td>1000</td>
<td>195.52</td>
<td>196.35</td>
<td>195.66</td>
</tr>
<tr>
<td>128</td>
<td>1000</td>
<td>197.30</td>
<td>198.14</td>
<td>197.72</td>
</tr>
<tr>
<td>256</td>
<td>1000</td>
<td>204.92</td>
<td>205.75</td>
<td>205.61</td>
</tr>
<tr>
<td>512</td>
<td>1000</td>
<td>218.65</td>
<td>220.31</td>
<td>219.62</td>
</tr>
<tr>
<td>1024</td>
<td>1000</td>
<td>266.56</td>
<td>266.56</td>
<td>266.56</td>
</tr>
</tbody>
</table>

# Benchmarking Reduce
# ( #processes = 7 )

<table>
<thead>
<tr>
<th>#bytes</th>
<th>#repetitions</th>
<th>t_min[usec]</th>
<th>t_max[usec]</th>
<th>t_avg[usec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>0.64</td>
<td>0.83</td>
<td>0.78</td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>223.25</td>
<td>224.08</td>
<td>223.61</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>209.92</td>
<td>210.75</td>
<td>210.39</td>
</tr>
<tr>
<td>16</td>
<td>1000</td>
<td>208.65</td>
<td>209.48</td>
<td>209.13</td>
</tr>
<tr>
<td>32</td>
<td>1000</td>
<td>211.58</td>
<td>212.41</td>
<td>211.82</td>
</tr>
<tr>
<td>64</td>
<td>1000</td>
<td>235.74</td>
<td>235.74</td>
<td>235.74</td>
</tr>
<tr>
<td>128</td>
<td>1000</td>
<td>238.39</td>
<td>239.23</td>
<td>238.51</td>
</tr>
<tr>
<td>256</td>
<td>1000</td>
<td>250.20</td>
<td>251.03</td>
<td>250.68</td>
</tr>
<tr>
<td>512</td>
<td>1000</td>
<td>279.79</td>
<td>279.79</td>
<td>279.79</td>
</tr>
<tr>
<td>1024</td>
<td>1000</td>
<td>331.10</td>
<td>332.77</td>
<td>331.69</td>
</tr>
</tbody>
</table>
6.3 Sample 3

mpirun -np 5 PMB-MPI1 pingping -multi 1
(PMB_OPTIONAL mode)

#---------------------------------------------------
#    PALLAS MPI Benchmark Suite V2.1, MPI-1 part
#---------------------------------------------------

# Date       : Thu Sep 10 10:37:46 1998
# Machine    : alpha# System     : OSF1
# Release    : V4.0
# Version    : 564
#
#
# Minimum message length in bytes:   0
# Maximum message length in bytes:   256
#
# MPI_Datatype                   :   MPI_BYTE
# MPI_Datatype for reductions    :   MPI_FLOAT
# MPI_Op                         :   MPI_SUM
#
#
# !!!! Attention: results have been achieved in
# !!!! PMB_OPTIONAL mode.
# !!!! Results may differ from standard case.
#
#
# List of Benchmarks to run:

# (Multi-)PingPing
# Benchmarking Multi-PingPing
# ( 2 groups of 2 processes each running simultaneous )
# Group 0 : 0 1
# Group 1 : 2 3
# ( 1 additional process waiting in MPI_Barrier)

<table>
<thead>
<tr>
<th>Group</th>
<th>#bytes</th>
<th>#repetitions</th>
<th>t[usec]</th>
<th>Mbytes/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1000</td>
<td>9.98</td>
<td>0.00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1000</td>
<td>78.21</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1000</td>
<td>9.98</td>
<td>0.10</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1000</td>
<td>70.59</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1000</td>
<td>9.15</td>
<td>0.21</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1000</td>
<td>71.40</td>
<td>0.03</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1000</td>
<td>9.15</td>
<td>0.42</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1000</td>
<td>70.74</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1000</td>
<td>9.17</td>
<td>0.83</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1000</td>
<td>70.69</td>
<td>0.11</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1000</td>
<td>9.16</td>
<td>1.67</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1000</td>
<td>70.80</td>
<td>0.22</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1000</td>
<td>10.00</td>
<td>3.05</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1000</td>
<td>71.53</td>
<td>0.43</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1000</td>
<td>17.49</td>
<td>3.49</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1000</td>
<td>86.63</td>
<td>0.70</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>1000</td>
<td>19.99</td>
<td>6.11</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>1000</td>
<td>89.02</td>
<td>1.37</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>1000</td>
<td>22.06</td>
<td>11.07</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>1000</td>
<td>92.74</td>
<td>2.63</td>
</tr>
</tbody>
</table>
7 Further details

7.1 Memory Requirements

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Standard mode memory demand per process ((Q)) active processes)</th>
<th>Optional mode memory demand per process ((X = 2^{\text{MAXMSGLOG}}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alltoall, Allgather, Allgatherv</td>
<td>(Q \times 8) MBytes ((Q+1) \times 4) MBytes ((Q+1) \times X) bytes</td>
<td>(Q \times 2X) bytes ((Q+1) \times X) bytes (2X) bytes</td>
</tr>
</tbody>
</table>

Table 3: Memory 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 (irrelevant here):
- 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, q_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 \texttt{cpp} flag \texttt{-DCHECK} through the \texttt{CPPFLAGS} variable (see 2.2), and recompiling, at PMB runtime every message passing result will be checked against the expected outcome (note that the contents of each buffer is well defined, see 5.2.4). Output tables will contain an additional column displaying the diffs as floats (named \texttt{defects}).

\textbf{Attention:} \texttt{-DCHECK} results are not valid as real benchmark data! Don’t forget to deactivate \texttt{DCHECK} and recompile in order to get proper results.
7.4 Use of MPI

Except for documented use in the benchmark kernels, MPI is used to the following extent in PMB-MPI1:

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_Error_string, MPI_Errhandler_create, MPI_Errhandler_set,
MPI_Errhandler_free, MPI_Abort
MPI_Finalize

8 Revision History

<table>
<thead>
<tr>
<th>Release No.</th>
<th>Date</th>
<th>Content</th>
<th>Related Software Releases</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1997/06</td>
<td>draft documentation</td>
<td>PMB 1.0, 1.1, 1.2, 1.3</td>
</tr>
<tr>
<td>2.0</td>
<td>1998/06</td>
<td>complete definition</td>
<td>PMB 2.0</td>
</tr>
<tr>
<td>2.1</td>
<td>1998/09</td>
<td>4.4.6 added, 5.2.1 added, minor textual changes</td>
<td>PMB 2.1</td>
</tr>
<tr>
<td>2.2</td>
<td>2000/03</td>
<td>3.3 updated</td>
<td>PMB 2.2</td>
</tr>
</tbody>
</table>

9 References

3 Pallas MPI Benchmarks - PMB, part MPI-2