

# The SKaMPI 5 manual

Werner Augustin and Thomas Worsch

April 16, 2008

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Overview . . . . .	3
1.2	Warnings . . . . .	3
<b>2</b>	<b>Usage of SKaMPI 5</b>	<b>4</b>
2.1	The basic structure . . . . .	4
2.2	Scope of SKaMPI 5 . . . . .	5
2.3	Command line arguments for SKaMPI . . . . .	5
2.4	Input and output file names . . . . .	5
2.5	Basics of run-time configuration . . . . .	5
2.6	Building SKaMPI 5 . . . . .	7
<b>3</b>	<b>All SKaMPI 5 measurement functions</b>	<b>9</b>
3.1	Measurements of point to point communication . . . . .	9
3.2	Measurements of collective communication . . . . .	10
3.3	Measurements of one-sided communication . . . . .	11
3.4	Miscellaneous measurements . . . . .	14
<b>4</b>	<b>SKaMPI 5 helper functions and iterators</b>	<b>14</b>
4.1	Helper functions . . . . .	14
4.2	Iterators . . . . .	16
<b>5</b>	<b>Syntax and semantics of SKaMPI 5 input files</b>	<b>16</b>
5.1	Data types . . . . .	16
5.2	Predefined Constants . . . . .	16
5.3	Control structures . . . . .	17
5.3.1	Loops . . . . .	17
5.3.2	Conditional statements . . . . .	18
5.3.3	Measurement statements . . . . .	18
<b>6</b>	<b>Extending SKaMPI 5</b>	<b>18</b>
6.1	Buffer handling . . . . .	19
6.2	Writing a new measurement function . . . . .	20
6.2.1	Doing synchronized measurements . . . . .	20
6.2.2	Functions available inside <code>measure_</code> functions . . . . .	21
6.3	Writing a new iterator . . . . .	22
6.4	Writing a new helper function . . . . .	23
6.5	Building an extended version of SKaMPI . . . . .	23
<b>7</b>	<b>Miscellaneous</b>	<b>24</b>
7.1	Format of output file . . . . .	24
7.2	Preparing diagrams from SKaMPI output files . . . . .	25
7.3	Our TODO list . . . . .	26
<b>8</b>	<b>Pitfalls of benchmarking</b>	<b>26</b>



# 1 Introduction

SKaMPI 5 is a benchmark for MPI implementations. If you don't know what MPI is, this software is not for you. But you may want to go to <http://www.mpi-forum.org> to find out more.

If you only want a *very short* introduction at the moment, have a look at the accompanying documentation entitled “SKaMPI 5 for the impatient” which you may find in the file surprisingly named `impatient.ps`.

If you only want a *short* introduction at the moment, read Section 2 (and maybe have a look at the C files in the subdirectory `measurements`).

Please read the warnings in Section 1.2.

## 1.1 Overview

The remainder of this manual is subdivided into the following sections:

**Section 2:** We start with a description of the basic structure of SKaMPI 5 and its usage.

**Section 3:** There you can find the complete list of all measurements that can be performed by the current version of SKaMPI 5.

**Section 4:** There you can find the complete list of all builtin helper functions and iterators in SKaMPI.

**Section 5:** There you can find the detailed syntax for SKaMPI 5 input files and what the syntactic constructs mean to the interpreter inside SKaMPI.

**Section 6:** This is for you in case you are missing some functionality in Section 3 or Section 4: SKaMPI 5 is a complete rewrite. Its major advantage over previous releases is its easy extendibility. In Section 6 you will find everything you need to know for writing your own pieces of benchmarking code to be used by SKaMPI 5.

**Section 7:** Here we have collected pieces of information which did not fit well into the previous sections.

**Section 8:** Some remarks on general problems with benchmarking MPI libraries.

## 1.2 Warnings

*Please* keep the following in mind when using SKaMPI:

- We have no experience with SKaMPI running on machines with *many* processors. We are aware of the following potential scalability problem:
  - If your MPI library does not provide synchronous clocks, i.e. `MPI_WTIME_IS_GLOBAL` is false, the “time management” of SKaMPI may use too much time of its own on large machines.
- If you are running SKaMPI with really many processes you should be very careful calling `set_max_nr_node_times` with the size of `MPI_COMM_WORLD` as argument.

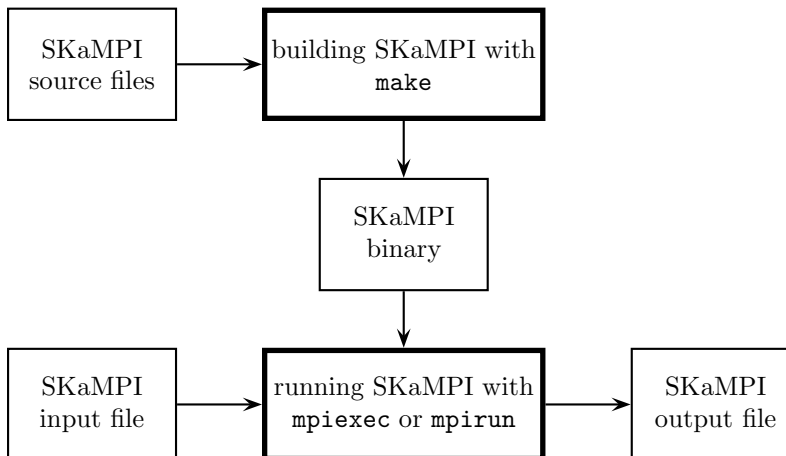


Figure 1: How SKaMPI is used. (See Section 6 for a more detailed view.)

- Be careful that you might have measurements which take so little time (e.g. pings on really fast interconnects) that the overhead of the calls of `MPI_Wtime` is significant. In that case increase the `iterations` parameter for the pingpong measurements.
- This documentation is usable, but not perfect.
- The contents of the default `.ski` files coming with SKaMPI may change in forthcoming releases.
- The names of measurement and helper functions may change in the future. We will try to minimize these changes.

Please let us know any positive or negative experiences, so that we can improve SKaMPI.

## 2 Usage of SKaMPI 5

### 2.1 The basic structure

Figure 1 shows you the relations between several (sets of) files. You will usually compile the SKaMPI source files only once (unless you are extending SKaMPI with your own measurement functions; see Section 6). That will give you the SKaMPI binary, called `skampi` by default. (See Section 2.6 for more details about building SKaMPI 5.)

The resulting binary is an MPI application which you will then probably run repeatedly using different input files, different numbers of processors and/or on different hardware platforms. Depending on your parallel environment this may be achieved using a command line like

```
mpiexec -n 7 skampi -i coll.ski -o foo.sko
```

The SKaMPI 5 binary offers a lot of functions for benchmarking different aspects of an MPI library. The input file contains the set of measurements and their corresponding parameters which should be used for a specific execution of SKaMPI.

In Subsection 2.2 we give a quick overview of what parts of an MPI implementation can be benchmarked with the present version of SKaMPI 5.

In Subsection 2.3 all command line arguments for SKaMPI 5 will be explained. In Subsection 2.4 you will find informations about how the default input and output file names for SKaMPI 5 are constructed.

Subsection 2.5 will show you the basics of how to specify a sequence of measurements in a SKaMPI 5 input file.

## 2.2 Scope of SKaMPI 5

The present version of SKaMPI 5 allows you to do measurements of (two-sided) point to point communication, collective communication operations and a few other functions. If your MPI implementation supports it, one-sided communication can be benchmarked, too. In Section 3 we provide the complete lists of all measurements currently possible.

There are also measurements for MPIIO, but they are not yet documented.

Virtual topologies and derived data types are two aspects of MPI which were covered by SKaMPI 4. Please note, that they are not yet supported by SKaMPI 5, but we plan to add those back, of course.

## 2.3 Command line arguments for SKaMPI

SKaMPI 5 understands a number of command line arguments.

-h	display a short help message similar to this table and exit immediately
-i <i>&lt;infile&gt;</i>	specify the name of the input file
-o <i>&lt;outfile&gt;</i>	specify the name of the output file
-n	perform a syntax check of the input file but don't run any measurements
-s	include the contents of the ski file in the sko file
-d <i>&lt;flags&gt;</i>	for debugging purposes of the SKaMPI 5 developers

## 2.4 Input and output file names

If an input file name is explicitly specified on the command line, the configuration for the SKaMPI 5 run is taken from that file. Otherwise the default file name `skampi.ski` is used.

If an output file name is explicitly specified on the command line, SKaMPI 5 will write all of its results to that file. Otherwise the results are written to standard out.

## 2.5 Basics of run-time configuration

A SKaMPI 5 input file roughly looks like a program written in a special purpose (namely for benchmarking) imperative programming language. When SKaMPI 5 is started, for each process an interpreter reads the input file, parses it and takes the appropriate actions, i.e. it uses the SPMD (single program multiple data) execution model already well-known from MPI applications.

You have

- variables
- function calls. It is important to distinguish two types of functions:
  - measurement functions:** Only these can be used in so-called measure statements and actually perform a measurement.
  - helper functions:** These are all the other functions. They must not be used in so-called measure statements. They may have side effects (e.g. setting the size of send and receive buffers) and/or a return value (e.g. a communicator of a requested size)
- expressions
- data types: integer, double, string, MPI\_Comm, MPI\_Datatype, MPI\_Op and MPI\_Info
- control structures:
  - if *<cond>* then *<then-part>* [else *<else-part>*] fi
  - for *<loop spec>* do *<loop body>* od
- measure statements, which have to be put into **measurement** blocks
- comments, which can be started anywhere on any line by the # character and extend to the end of the line.

Here is a simple example input file for SKaMPI 5, broken up into a few segments, each followed by some comments. The first thing you will notice: Do not put a semicolon at the end of a statement; and do not put several statements in one line.

```

1  set_min_repetitions(8)
2  set_max_repetitions(20)
3  set_max_relative_standard_error(0.03)

```

In lines 1–3 three functions are called which set some global parameters for the following measurements. In Subsection 5.3.3 we will describe these features in more detail.

```

4  set_skampi_buffer(128kb)

```

SKaMPI 5 provides two buffers for each process, which are called the send buffer and the receive buffer. In line 4 SKaMPI 5 is told to allocate a total of 128 kbytes for these buffers. The exact description of how this memory is used for the buffers is too long for this short introduction. Please see Subsection 6.1 for all details.

```

5  comm_pt2pt = comm2_max_latency_with_root()

```

In line 5 a communicator is stored in the variable `comm_pt2pt`. It is determined by the builtin function `comm2_max_latency_with_root()` which tries to find a process whose latency for point-to-point communications with process 0 of `MPI_COMM_WORLD` is maximum among all processes.

Finally, here is a typical specification of a measurement block:

```

6  begin measurement "MPI_Send-MPI_Recv"
7    for count = 1 to ... step *sqrt(2) do
8      measure comm_pt2pt Pingpong_Send_Recv(count, MPI_INT, 0, 42)
9    od
10 end measurement

```

A measurement block has to be started with a line of the form

```
begin measurement "<name>"
```

and it has to be finished by a line

```
end measurement
```

where the *<name>* may be any string not containing a double quote character ". The *<name>* should be informative (for your own happiness). Between these lines some actual measurement(s) should be requested. The basic syntax is

```
measure <communicator> : <measurement function with args>
```

The *<measurement function>* will carry out its measurements in the specified *<communicator>*. In line 8 above you can see, that obviously `Pingpong_Send_Recv()` is a measurement function (which does a typical pingpong measurement using `MPI_Send` and `MPI_Recv`). For the complete list of all such functions with their parameters see Section 3. `Pingpong_Send_Recv` expects 4 parameters: a number of data elements to be sent forth and back, the MPI datatype of a single element, a message tag and the number of iterations of one pingpong. In the example, `MPI_INT` is chosen as the fixed datatype, 0 as the fixed tag and 42 pingpongs are requested. The number of elements is in `count`, and it is varied inside a loop.

In line 7 you see that its initial value is 1. The upper bound for the last value is not specified explicitly (which it could be). Instead there are three dots ... which are interpreted as follows: The loops ends when for the first time the value of `count` is such that at least one of the processes which are involved in the measurement would need more memory than allocated via the `set_skampi_buffer` call. Thus, if you change the value in line 4 to say 1MB, more measurements (for larger message sizes) will be done without any change to the measurement loop.

The "increment" for the loop variable is specified as `*sqrt(2)`. This means that after each measurement, it is updated by the assignment

```
count = count * sqrt(2)
```

followed by some rounding to an integer. But you don't have think about the rounding yourself, SKaMPI does that for you. More precisely, the first numbers without rounding are 1.0, 1.4..., 2.0..., etc. After rounding one gets 1, 1 once again, 2 and so on. SKaMPI 5 is smart enough to drop the second 1.

For all the details of the correct syntax for SKaMPI input files please see Section 5.

For the list of all builtin helper functions (like `sqrt()` above) in SKaMPI see Section 4.

## 2.6 Building SKaMPI 5

The sources for SKaMPI 5 are provided in a single gzipped tar file, the name of which has one of the following forms:

- `skampi-5.<x>.<y>.tar.gz`
- `skampi-5.<x>.<y>-r<rev>.tar.gz`

If you only want to *use* SKaMPI 5, you just have to compile it once. If you got a file named `skampi-5.<x>.<y>.tar.gz` the following simple steps should do the job:

1. `tar xzf skampi-5.<x>.<y>.tar.gz`

2. `cd skampi-5.<x>.<y>`

3. `make`

In the case of a file named `skampi-5.<x>.<y>-r<rev>.tar.gz` you can proceed analogously.

If `make` succeeds you should have the executable called `skampi`. Start it as any other application using MPI (see Sections 2.1 and 2.3 above).

1. **Unpacking:** You should have received SKaMPI 5 in a gzipped tar file whose name looks like `<skampidir>.tar.gz` where `<skampidir>` is either `skampi-5.<x>.<y>` or `skampi-5.<x>.<y>-r<rev>`. With GNU `tar` you can unpack the archive using

```
tar xzf <skampidir>.tar.gz
```

If your version of `tar` cannot handle `.gz` file, use

```
gzip -cd <dirname>.tar.gz | tar xf -
```

or

```
zcat skampi-5.<x>.<y>.tar.gz | tar xf -
```

After unpacking you should have a directory `skampi-5.<x>.<y>` or `skampi-5.<x>.<y>-r<rev>` containing the C source files, example input files and the documentation for SKaMPI 5.

2. **Compilation** should in many cases be as simple as

```
cd <skampidir>; make
```

This works if the value `mpicc` of the variable `MPICC` in `Makefile` is correct for your installation. If you have to use a different program for compiling MPI sources, please change it.

Concerning measurements for one-sided communication the situation is as follows: If your MPI library claims that it implements the full MPI-2 standard, one-sided measurements are compiled into SKaMPI. Otherwise, if you know that your MPI implementation does provide one-sided communication, you may define `USE_ONESIDED` to have one-sided measurements compiled into SKaMPI, i.e. you should add `-DUSE_ONESIDED` to the `make` variable `CFLAGS`.

Concerning measurements for MPI-IO the situation is similar: If your MPI library claims that it implements the full MPI-2 standard, measurements for MPI-IO are compiled into SKaMPI. Otherwise, if you know that your MPI implementation does provide MPI-IO, you may define `USE_MPI_IO` to have the corresponding measurements compiled into SKaMPI, i.e. you should add `-DUSE_MPI_IO` to the `make` variable `CFLAGS`.

3. **Installation:** There is no `make install`. Just use the SKaMPI 5 binary, which by default has the name `skampi`.

If you want to know more, out of curiosity or because you want to *extend* SKaMPI 5, please have a look at section 6. In particular subsection 6.5 explains the building process in more detail.

### 3 All SKaMPI 5 measurement functions

You can find all SKaMPI 5 measurement functions by looking for functions having a name of the form `measure_⟨name⟩` in all the files residing in subdirectory `measurements` of the SKaMPI source distribution.

#### 3.1 Measurements of point to point communication

Here is a complete list of all measurement functions in the present version of SKaMPI 5 for point to point communication. At the moment all are simple pingpong measurements. For example the complete measurement function `Pingpong_Send_Recv` looks like this:

```
double measure_Pingpong_Send_Recv(int count, MPI_Datatype datatype,
                                  int tag, int iterations)
{
    double start_time, end_time;
    MPI_Status status;
    int i;

    if (iterations<0) { return -1.0; }
    if (iterations==0) { return 0.0; }

    if( get_measurement_rank() == 0 ) {
        start_time = MPI_Wtime();
        for (i=0; i<iterations; i++) {
            MPI_Send(get_send_buffer(), count, datatype, 1, tag,
                     get_measurement_comm());
            MPI_Recv(get_recv_buffer(), count, datatype, 1, tag,
                    get_measurement_comm(), &status);
        }
        end_time = MPI_Wtime();
        MPI_Send(get_send_buffer(), count, datatype, 1, tag, get_measurement_comm());
    } else {
        MPI_Recv(get_recv_buffer(), count, datatype, 0, tag,
                get_measurement_comm(), &status);
        start_time = MPI_Wtime();
        for (i=0; i<iterations; i++) {
            MPI_Send(get_send_buffer(), count, datatype, 0, tag,
                     get_measurement_comm());
            MPI_Recv(get_recv_buffer(), count, datatype, 0, tag,
                    get_measurement_comm(), &status);
        }
        end_time = MPI_Wtime();
    }
    return (end_time - start_time)/iterations;
}
```

That is, process 0 does **iterations** many pingpong roundtrips and process 1 does **iterations** many pongping roundtrips. Both processes return the average time needed for *one full message roundtrip*. Note that this is the time for one “ping” plus that for one “pong”. *This is different from some other benchmarks which return half of that time.*

The other functions listed below are (mostly obvious) variations.

function	parameters	remarks
Pingpong_Send_Recv	$\langle cnt \rangle, \langle dt \rangle, \langle tag \rangle, \langle iters \rangle$	
Pingpong_Send_Iprobe_Recv	$\langle cnt \rangle, \langle dt \rangle, \langle tag \rangle, \langle iters \rangle$	<sup>a</sup>
Pingpong_Send_Irecv	$\langle cnt \rangle, \langle dt \rangle, \langle tag \rangle, \langle iters \rangle$	
Pingpong_Send_Recv_AT	$\langle cnt \rangle, \langle dt \rangle, \langle tag \rangle, \langle iters \rangle$	<sup>b</sup>
Pingpong_Ssend_Recv	$\langle cnt \rangle, \langle dt \rangle, \langle tag \rangle, \langle iters \rangle$	
Pingpong_Isend_Recv	$\langle cnt \rangle, \langle dt \rangle, \langle tag \rangle, \langle iters \rangle$	
Pingpong_Issend_Recv	$\langle cnt \rangle, \langle dt \rangle, \langle tag \rangle, \langle iters \rangle$	
Pingpong_Bsend_Recv	$\langle cnt \rangle, \langle dt \rangle, \langle tag \rangle, \langle iters \rangle$	
Pingpong_Sendrecv	$\langle s\_cnt \rangle, \langle s\_dt \rangle, \langle s\_tag \rangle,$ $\langle r\_cnt \rangle, \langle r\_dt \rangle, \langle r\_tag \rangle, \langle iters \rangle$	
Pingpong_Sendrecv_replace	$\langle cnt \rangle, \langle dt \rangle,$ $\langle s\_tag \rangle, \langle r\_tag \rangle, \langle iters \rangle$	

<sup>a</sup>Process 0 actively waits using MPI\_Iprobe before receiving.

<sup>b</sup>The calls to MPI\_Recv specify MPI\_ANY\_TAG.

### 3.2 Measurements of collective communication

Here is a complete list of all measurement functions in the present version of SKaMPI 5 for collective communication.

function	parameters	remarks
Bcast	$\langle cnt \rangle, \langle dt \rangle, \langle root \rangle$	
Bcast_Send_Recv	$\langle cnt \rangle, \langle dt \rangle, \langle root \rangle$	<sup>a</sup>
Barrier	—	
Reduce	$\langle cnt \rangle, \langle dt \rangle, \langle op \rangle, \langle root \rangle$	
Allreduce	$\langle cnt \rangle, \langle dt \rangle, \langle op \rangle$	
Reduce_Bcast	$\langle cnt \rangle, \langle dt \rangle, \langle op \rangle, \langle root \rangle$	
Reduce_scatter	$\langle cnt \rangle, \langle dt \rangle, \langle op \rangle, \langle root \rangle$	
Alltoall	$\langle s\_cnt \rangle, \langle s\_dt \rangle,$ $\langle r\_cnt \rangle, \langle r\_dt \rangle$	
Alltoall_Isend_Irecv	$\langle s\_cnt \rangle, \langle s\_dt \rangle,$ $\langle r\_cnt \rangle, \langle r\_dt \rangle$	
Gather	$\langle s\_cnt \rangle, \langle s\_dt \rangle,$ $\langle r\_cnt \rangle, \langle r\_dt \rangle, \langle root \rangle$	
Gather_SR	$\langle s\_cnt \rangle, \langle s\_dt \rangle,$ $\langle r\_cnt \rangle, \langle r\_dt \rangle, \langle root \rangle$	<sup>b</sup>

<sup>a</sup>a very naïve self-made Bcast

<sup>b</sup>a very naïve self-made Alltoall using Send and Recv

function	parameters	remarks
Gather_ISWA	$\langle s\_cnt \rangle, \langle s\_dt \rangle,$ $\langle r\_cnt \rangle, \langle r\_dt \rangle, \langle root \rangle$	<sup>a</sup>
Allgather	$\langle s\_cnt \rangle, \langle s\_dt \rangle,$ $\langle r\_cnt \rangle, \langle r\_dt \rangle$	
Scatter	$\langle s\_cnt \rangle, \langle s\_dt \rangle,$ $\langle r\_cnt \rangle, \langle r\_dt \rangle, \langle root \rangle$	
Reduce_Scatterv	$\langle cnt \rangle, \langle dt \rangle, \langle op \rangle, \langle root \rangle$	
Alltoallv	$\langle s\_cnt \rangle, \langle s\_dt \rangle,$ $\langle r\_cnt \rangle, \langle r\_dt \rangle$	
Alltoallv_Isend_Irecv	$\langle s\_cnt \rangle, \langle s\_dt \rangle,$ $\langle r\_cnt \rangle, \langle r\_dt \rangle$	<sup>b</sup>
Gatherv	$\langle s\_cnt \rangle, \langle s\_dt \rangle,$ $\langle r\_cnt \rangle, \langle r\_dt \rangle, \langle root \rangle$	
Allgatherv	$\langle s\_cnt \rangle, \langle s\_dt \rangle,$ $\langle r\_cnt \rangle, \langle r\_dt \rangle$	
Scatterv	$\langle s\_cnt \rangle, \langle s\_dt \rangle,$ $\langle r\_cnt \rangle, \langle r\_dt \rangle, \langle root \rangle$	
Scan	$\langle cnt \rangle, \langle dt \rangle, \langle op \rangle$	
Comm_split	—	
Comm_dup	—	
MPI_Wtime	—	
myalltoallv	$\langle s\_cnt \rangle, \langle s\_dt \rangle,$ $\langle r\_cnt \rangle, \langle r\_dt \rangle,$ $\langle charged\_rank \rangle, \langle ratio \rangle$	<sup>c</sup>

<sup>a</sup>a very naïve self-made Alltoall using Isend and Waitall

<sup>b</sup>a very naïve self-made Alltoall

<sup>c</sup>Read the source for details.

### 3.3 Measurements of one-sided communication

Here is a complete list of all measurement functions in the present version of SKaMPI 5 for one-sided communication.

function	parameters	remarks
MPI_Win_fence_open	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle assert \rangle$	
MPI_Win_fence_close	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle assert \rangle,$ $\langle dput \rangle$	
MPI_Win_fence_close_collective	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle assert \rangle,$ $\langle dput \rangle$	
MPI_Win_fence_close_delayed	$\langle int \rangle, \langle dt \rangle, \langle info \rangle,$ $\langle msg\_cnt \rangle, \langle delay\_us \rangle$	
MPI_Win_fence_openclose	$\langle int \rangle, \langle dt \rangle, \langle info \rangle,$ $\langle dput \rangle$	
MPI_Win_start	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle assert \rangle$	

function	parameters	remarks
MPI_Win_start_delayed_post	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle assert \rangle, \langle delay\_us \rangle, \langle delay\_rank \rangle$	
MPI_Win_post	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle assert \rangle$	
MPI_Win_complete	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle assert \rangle, \langle dput \rangle$	
MPI_Win_startcomplete	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle assert \rangle, \langle dput \rangle$	
MPI_Win_complete_n	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle assert \rangle, \langle msg\_cnt \rangle, \langle delay\_us \rangle$	
MPI_Win_complete_delayed_wait	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle assert \rangle, \langle dput \rangle, \langle delay\_us \rangle$	
MPI_Win_wait_early_complete	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle assert \rangle, \langle dput \rangle, \langle delay\_us \rangle$	
MPI_Win_wait_delayed_complete	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle assert \rangle, \langle dput \rangle, \langle delay\_us \rangle$	
MPI_Win_test	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle assert \rangle, \langle dput \rangle$	
MPI_Win_test_delayed	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle assert \rangle, \langle dput \rangle, \langle delay\_us \rangle$	
MPI_Win_create	$\langle int \rangle, \langle dt \rangle, \langle info \rangle$	
MPI_Win_free	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle dput \rangle$	
MPI_Win_lock	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle assert \rangle, \langle lock\_type \rangle, \langle dest \rangle$	
MPI_Win_lock_desync	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle assert \rangle, \langle lock\_type \rangle, \langle dest \rangle, \langle delay\_us \rangle$	
MPI_Win_unlock	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle assert \rangle, \langle lock\_type \rangle, \langle dest \rangle, \langle dput \rangle, \langle delay\_us \rangle$	
MPI_Put_Pingpong	$\langle int \rangle, \langle dt \rangle, \langle info \rangle$	
MPI_Put_fence_bidirectional	$\langle int \rangle, \langle dt \rangle, \langle info \rangle$	
MPI_Put_callduration	$\langle int \rangle, \langle dt \rangle, \langle info \rangle$	
MPI_Isend_callduration	$\langle int \rangle, \langle dt \rangle$	
MPI_Put_dedicated	$\langle int \rangle, \langle dt \rangle, \langle info \rangle$	
MPI_Put_passive	$\langle int \rangle, \langle dt \rangle, \langle info \rangle$	
MPI_Put_passive_concurrent	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle lock\_type \rangle$	
MPI_Put_fence	$\langle int \rangle, \langle dt \rangle, \langle info \rangle$	
MPI_Put_activewait_get	$\langle int \rangle, \langle info \rangle$	
MPI_Put_activewait_twosided	$\langle int \rangle, \langle info \rangle$	
MPI_Put_activewait_twosided_sleep	$\langle int \rangle, \langle info \rangle, \langle delay\_us \rangle$	
MPI_Get_Pingpong	$\langle int \rangle, \langle dt \rangle, \langle info \rangle$	

function	parameters	remarks
MPI_Get_callduration	$\langle int \rangle, \langle dt \rangle, \langle info \rangle$	
MPI_Get_dedicated	$\langle int \rangle, \langle dt \rangle, \langle info \rangle$	
MPI_Get_passive	$\langle int \rangle, \langle dt \rangle, \langle info \rangle$	
MPI_Get_fence	$\langle int \rangle, \langle dt \rangle, \langle info \rangle$	
MPI_Get_activewait	$\langle int \rangle, \langle info \rangle$	
MPI_Get_activewait_sleep	$\langle int \rangle, \langle info \rangle, \langle delay\_us \rangle$	
MPI_Accumulate	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle op \rangle$	
MPI_Accumulate_activewait_twosided	$\langle int \rangle, \langle info \rangle$	
MPI_Accumulate_concurrent	$\langle int \rangle, \langle dt \rangle, \langle info \rangle,$ $\langle op \rangle, \langle disjoint \rangle$	
Caching	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle delay\_us \rangle$	
Combining_fence	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle msg\_cnt \rangle$	
Combining_dedicated	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle msg\_cnt \rangle$	
Datatype_complex_everytime	$\langle int \rangle, \langle info \rangle$	
Datatype_complex_once	$\langle int \rangle, \langle info \rangle$	
Datatype_complex_mixed_everytime	$\langle int \rangle, \langle info \rangle$	
Datatype_complex_mixed_once	$\langle int \rangle, \langle info \rangle$	
Datatype_complex_Get	$\langle int \rangle, \langle info \rangle$	
Datatype_int	$\langle int \rangle, \langle info \rangle$	
Datatype_simple	$\langle int \rangle, \langle info \rangle$	
Datatype_simple_Get	$\langle int \rangle, \langle info \rangle$	
MPI_Put_Shift	$\langle int \rangle, \langle dt \rangle, \langle info \rangle,$ $\langle distance \rangle$	
Exchange	$\langle int \rangle, \langle dt \rangle, \langle msg\_cnt \rangle$	
MPI_Put_Exchange	$\langle int \rangle, \langle dt \rangle, \langle info \rangle,$ $\langle msg\_cnt \rangle$	
MPI_Put_Exchange_passive	$\langle int \rangle, \langle dt \rangle, \langle info \rangle,$ $\langle msg\_cnt \rangle$	
MPI_Accumulate_concurrent_multi	$\langle int \rangle, \langle dt \rangle, \langle info \rangle,$ $\langle msg\_cnt \rangle, \langle op \rangle, \langle disjoint \rangle$	
onesided_bcast	$\langle int \rangle, \langle dt \rangle, \langle info \rangle,$ $\langle bcast\_count \rangle$	
onesided_alltoall	$\langle int \rangle, \langle dt \rangle, \langle info \rangle,$ $\langle alltoall\_count \rangle$	
multi_Alltoall	$\langle snd\_cnt \rangle, \langle snd\_dt \rangle,$ $\langle rcv\_cnt \rangle, \langle rcv\_dt \rangle, \langle alltoall\_count \rangle$	
onesided_reduce	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle op \rangle$	
onesided_allreduce	$\langle int \rangle, \langle dt \rangle, \langle info \rangle, \langle op \rangle$	
onesided_borderexchange	$\langle int \rangle, \langle info \rangle, \langle iteration\_count \rangle,$ $\langle compute \rangle$	
twosided_borderexchange	$\langle int \rangle, \langle iteration\_count \rangle,$	

function	parameters	remarks
	$\langle compute \rangle$	
onesided_borderexchange_fence	$\langle int \rangle$ , $\langle info \rangle$ , $\langle iteration\_count \rangle$ , $\langle compute \rangle$	
Send	$\langle int \rangle$ , $\langle dt \rangle$ , $\langle tag \rangle$	

### 3.4 Miscellaneous measurements

function	parameters	remarks
Wtime	$\langle iters \rangle$	tries to determine the average time needed for one of $\langle iters \rangle$ many calls to MPI_Wtime

## 4 SKaMPI 5 helper functions and iterators

### 4.1 Helper functions

returns	function	parameters	remarks
—	set_min_repetitions	$\langle min \rangle$	set the minimum number of single measurements used for one result
—	set_max_repetitions	$\langle max \rangle$	set the maximum number of single measurements used for one result
—	set_max_relative_standard_error	$\langle f \rangle$	set the maximum standard error for single measurements
—	set_max_nr_node_times	$\langle count \rangle$	set the maximum nr of procs for which times are logged in the output file (see also Subsection 7.1)

For the management of send and receive buffers the following functions are available:

returns	function	parameters
—	set_skampi_buffer	$\langle size \rangle$
—	switch_buffer_cycling_on	—
—	switch_buffer_cycling_off	—
—	set_send_buffer_alignment	$\langle a \rangle$
—	set_recv_buffer_alignment	$\langle a \rangle$
—	set_cache_size	$\langle size \rangle$
—	set_skampi_buffer_mpi_alloc_mem	$\langle size \rangle$ , $\langle info \rangle$

There are some useful functions (mainly) for constructing new communicators.

returns	function	parameters
$\langle int \rangle$	get_comm_size	$\langle comm \rangle$
$\langle comm \rangle$	comm2_max_latency_with_root	—
$\langle comm \rangle$	comm	$\langle size \rangle$
$\langle comm \rangle$	comm2	$\langle rank \rangle$ , $\langle rank \rangle$
$\langle comm \rangle$	copy_comm	$\langle comm \rangle$
$\langle comm \rangle$	comm_first_half	$\langle comm \rangle$

returns	function	parameters
$\langle comm \rangle$	<code>comm_second_half</code>	$\langle comm \rangle$

Some useful mathematical functions:

returns	function	parameters
$\langle double \rangle$	<code>sqrt</code>	$\langle x \rangle$
$\langle double \rangle$	<code>cbrt</code>	$\langle x \rangle$
$\langle double \rangle$	<code>sqr</code>	$\langle x \rangle$
$\langle int \rangle$	<code>floor</code>	$\langle x \rangle$
$\langle int \rangle$	<code>ceil</code>	$\langle x \rangle$
$\langle int \rangle$	<code>round_to_fourbytes</code>	$\langle size \rangle$
$\langle int \rangle$	<code>atoi</code>	$\langle str \rangle$
$\langle int \rangle$	<code>modulo</code>	$\langle val \rangle, \langle mod \rangle$
$\langle double \rangle$	<code>power</code>	$\langle base \rangle, \langle exp \rangle$

Now some functions for simple constructions of derived data types:

returns	function	parameters
$\langle dt \rangle$	<code>mpi_type_contiguous</code>	$\langle cnt \rangle, \langle old\_dt \rangle$
$\langle dt \rangle$	<code>mpi_type_vector</code>	$\langle cnt \rangle, \langle length \rangle, \langle stride \rangle, \langle old\_dt \rangle$
$\langle dt \rangle$	<code>mpi_type_hvector</code>	$\langle cnt \rangle, \langle length \rangle, \langle stride \rangle, \langle old\_dt \rangle$

Three functions which allow the use of `MPI_Info` variables:

returns	function	parameters
$\langle info \rangle$	<code>info_create</code>	—
—	<code>info_free</code>	$\langle info \rangle$
—	<code>info_set</code>	$\langle info \rangle, \langle key \rangle, \langle value \rangle$

If you know, what you are doing, you can switch the methods for benchmarking collective operations.

There is one case, though, where you must change something: *If you are running measurements in a communicator of size one, please call `choose_no_synchronization()` in your `.ski` file before doing the measurements.*

Furthermore, if you know that the running times of your measurements are large enough so that you can ignore the time consumed by a call to `MPI_Barrier()` (or if the default `choose_real_synchronization()` gives you problems<sup>1</sup>) you can call `choose_barrier_synchronization()` in your `.ski` file before doing the measurements.

returns	function	parameters
—	<code>choose_no_synchronization</code>	—
—	<code>choose_barrier_synchronization</code>	—
—	<code>choose_real_synchronization</code>	—
—	<code>init_time_accounting</code>	—
—	<code>print_time_accounting_info</code>	—

<sup>1</sup>We do have one user reporting that SKaMPI 5 freezes with this for large collective benchmarks.

## 4.2 Iterators

Currently there are only two (silly) builtin iterators for demonstration purposes. See the functions `iterator_squares()` and `iterator_range()` in `measurements/demo.c`.

See Section 6.3 for how to write your own iterators.

## 5 Syntax and semantics of SKaMPI 5 input files

- the content of the input file is a program which is interpreted in a way similar to SIMD
- free format
- comments start with '#' and extend to the end of the line
- the program consists of measurement blocks with local variables and statements (with local scope) and global variables and statements outside of these blocks
- variables use dynamic types; global variables can't be changed inside of a measurement block, a new local variables of the same name is initialized

### 5.1 Data types

- integer: usual arithmetic operations, suffixes of kb, KiB, mb, MiB, gb or GiB denote a multiplication with  $2^{10}$ ,  $2^{20}$  and  $2^{30}$  respectively.
- double: usual arithmetic operations
- strings (no operations implemented, can be used for parameters to self-implemented functions)
- `MPI_Datatype`, `MPI_Comm`, `MPI_Op` and `MPI_Info`

### 5.2 Predefined Constants

- all data-types: `MPI_CHAR`, `MPI_BYTE`, `MPI_SHORT`, `MPI_INT`, `MPI_LONG`, `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_UNSIGNED`, `MPI_UNSIGNED_CHAR`, `MPI_UNSIGNED_LONG`, `MPI_UNSIGNED_LONG_LONG`, `MPI_LONG_DOUBLE`, `MPI_FLOAT_INT`, `MPI_LONG_INT`, `MPI_DOUBLE_INT`, `MPI_SHORT_INT`, `MPI_2INT`, `MPI_LONG_DOUBLE_INT`, `MPI_LONG_LONG_INT`
- all reduction operators: `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, `MPI_PROD`, `MPI_LAND`, `MPI_BAND`, `MPI_LOR`, `MPI BOR`, `MPI_LXOR`, `MPI_BXOR`, `MPI_MINLOC`, `MPI_MAXLOC`  
For one-sided communication there is also `MPI_REPLACE`.
- `MPI_COMM_WORLD`
- `MPI_UNDEFINED`
- assertions: `MPI_MODE_NOPRECEDE`, `MPI_MODE_NOSUCCEED`, `MPI_MODE_NOCHECK`, `MPI_MODE_NOSTORE`, `MPI_MODE_NOPUT`

## 5.3 Control structures

### 5.3.1 Loops

Loops can be nested. There are several types of loops. First there are what one could call (lacking a better word) *computed loops*.

**Computed Loops:** Its simplest form is

- `for  $\langle var \rangle = \langle first.val \rangle$  to  $\langle last.val \rangle$  do  $\langle loop\ body \rangle$  od`

By default  $\langle var \rangle$  is incremented by 1. Optionally you can specify a different number using the following syntax:

- `for  $\langle var \rangle = \langle first.val \rangle$  to  $\langle last.val \rangle$  step  $\langle step.val \rangle$  do  $\langle loop\ body \rangle$  od`

The following variation allows to request that the loop variable is not incremented by the specified amount but that it is multiplied by that amount:

- `for  $\langle var \rangle = \langle first.val \rangle$  to  $\langle last.val \rangle$  step *  $\langle step.val \rangle$  do  $\langle loop\ body \rangle$  od`

For loops with a multiplicative `step` one may additionally request that the resulting values for the loop variable are rounded to a multiple of a given integer  $\langle j \rangle$  using the syntax

- `for  $\langle var \rangle = \langle first.val \rangle$  to  $\langle last.val \rangle$  step *  $\langle step.val \rangle$  multipleof  $\langle j \rangle$  do  $\langle loop\ body \rangle$  od`

Consider this example:

- `for i = 1 to 23 step *sqrt(2) multipleof 4 do etc.`

This would assign the following values to variable `i`:

In all forms of computed loops the upper bound for the loop variable may be specified as ... — that is: three consecutive dots without spaces in between. This means that the iterations repeats until the maximum buffer size (or half the buffer size when using buffer cycling) is reached.

**Autorefined loops:** For all of the above computed loops there is a variation called *autorefine loop*. Its syntax is the same as above, except that the keyword is `autorefine` instead of `for`. For example

- `autorefine  $\langle var \rangle = \langle first.val \rangle$  to  $\langle last.val \rangle$  step *  $\langle step.val \rangle$  do  $\langle loop\ body \rangle$  od`

An autorefine loop works as follows: First it is executed as if it were a for loop. Then SKaMPI uses a heuristic to check whether there are unexpected looking “jumps” or “discontinuities” in the sequence of timing results. If there are, SKaMPI does additional measurements at additional values for the loop variable which are “close” to the discontinuity.

**Enumerated loops:** You can loop over an explicitly given list of values like this:

- `for  $\langle var \rangle$  in [  $\langle list\ of\ values \rangle$  ] do  $\langle loop\ body \rangle$  od`

For example:

```
for dt in [MPI_INT, MPI_DOUBLE, MPI_BYTE] do
  would assign the three values (of type MPI_Datatype) to variable dt.
```

**Iterators:** `for <var> in <iterator>(<args>) do <loop body> od` loop over the values produced by an iterator

### 5.3.2 Conditional statements

- `if <cond> then <then-part> fi`
- `if <cond> then <then-part> else <else-part> fi`

In the conditions the usual comparison operators for numerical values are available.

### 5.3.3 Measurement statements

- `measure <communicator> : <measurement function>(<args>)`

Execute a measurement sequence. This means that the measurement function is called with the given parameters several times. Only processes of the mentioned communicator are participating in this measurement sequence. The number of repetitions is determined as follows:

- First the measurement function is called  $k$  times, where  $k$  is the argument of the most recent call to `set_min_repetitions`; if the function has not been called, a default value of 8 is used.
- Then the relative standard error of the running times measured so far is computed. If it is smaller than (or equal to) a prespecified relative standard error  $e$ , the measurement sequence is finished. Here,  $e$  is the argument to the most recent call to `set_max_relative_standard_error`; if the function has not been called, a default value of 0.1 is used.
- Otherwise, while the current relative standard error is larger than  $e$  and a maximum number  $m$  of repetitions has not been carried out, the measurement function is called again with the same arguments. Here,  $m$  is the argument to the most recent call to `set_max_repetitions`; if the function has not been called, a default value of 33 is used.

Do not use more than one measurement statement inside a measurement block.

## 6 Extending SKaMPI 5

Definitions of self-made functions can be placed in any .c-file in the subdirectory 'measurements'. A Perl script is used to extract the interfaces of these functions and to generate the source file `m.c` used by skampi; see Figure 2. This means that a working perl interpreter is needed as soon as SKaMPI is extended with new functions. Of course, it is possible to transfer the generated C source file to another target host without perl interpreter.

Functions which should be usable by SKaMPI have to have the following prefixes (additionally at the moment function definitions are restricted to one source code line):

- `func_` normal helper function

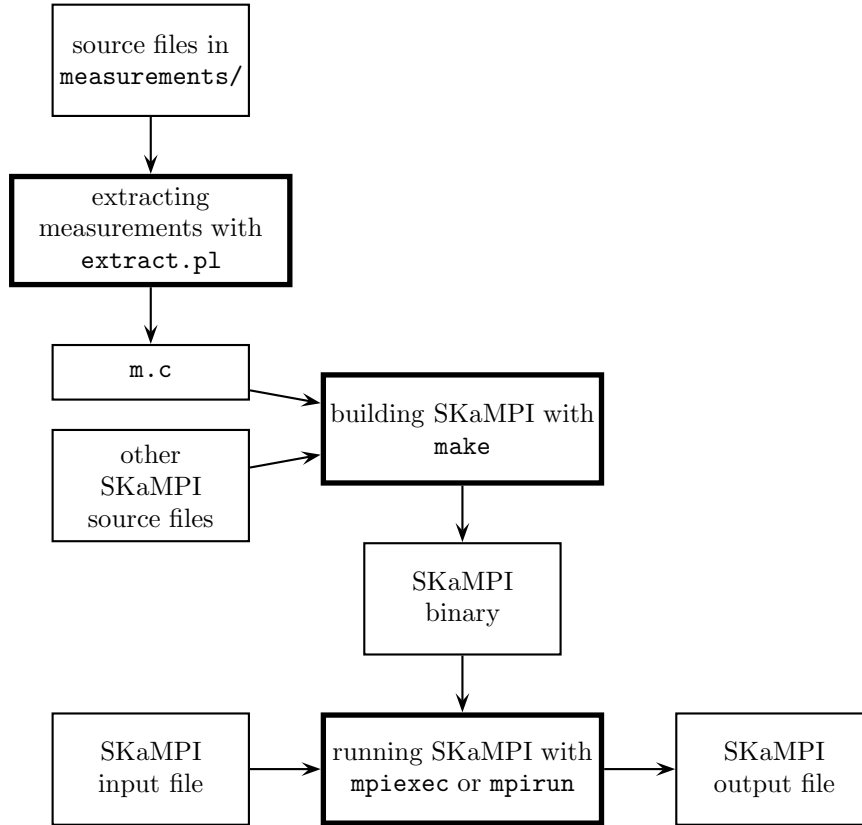


Figure 2: A more detailed view of how SKaMPI is built.

- `measure_` actual measurement function, possibly accompanied by
- `init_` optional initialization function for a measurement function
- `finalize_` optional de-initialization function for a measurement function
- `iterator_` iterator

## 6.1 Buffer handling

One needs some basic understanding of SKaMPI's buffer handling in order to be able to write meaningful measurements functions. In each process there are two buffers available, which are called the *send buffer* and the *receive buffer*. In a `measure_` function use calls of `get_send_buffer()` and `get_recv_buffer()` to get pointers to these buffers. Repeated calls of the same function may return different addresses. This is an attempt to reduce cache effects. You should call `set_send_buffer_usage()` and `set_recv_buffer_usage()` in the `init_` functions to tell SKaMPI how many bytes will be needed in the send and receive buffers.

## 6.2 Writing a new measurement function

For each new measurement the corresponding `init_`, `measure_` and `finalize_` functions have to have the same parameters. `init_` and `finalize_` functions are `void`-functions, `measure_` functions return a `double` value meaning the measured time in seconds. Processes which can't return a meaningful time, should return a negative value. The data-types of the parameters are restricted to the data-types used in the input file.

Definition of functions relevant to skampi have to be inside a block surrounded by the following lines;

```
#pragma weak begin_skampi_extensions
#pragma weak end_skampi_extensions
```

(In fact, the end marker can be omitted.)

A measurement with a fixed set of parameters consists of one call of the `init_` function (if defined), several calls of the `measure_` function and finally one call of the `finalize_` function (if defined).

The `init_` function has to set a couple of things:

- the use of send/receive buffer is set with `set_send_buffer_usage()` resp. `set_recv_buffer_usage()`. The default is 0, for operations like `MPI_Gather` the sizes can be different for the different processes.
- `set_reported_message_size()` sets the message size reported in the output file. The default is 0. It should be the number of bytes transferred by one send. In the SKaMPI 5 output file this number is printed in each measurement line (see also Sections 6.2.2 and 7.1).

You can also allocate memory, communicators etc.

In the `finalize_` function these objects can be released.

You do *not* have to take care of the measurement communicators; SKaMPI 5 does that for you.

### 6.2.1 Doing synchronized measurements

SKaMPI 5 supports what we call *synchronized measurements*, i.e. all processes of the measurement communicator start at (approximately) the same time, and a time interval is reserved for the whole duration of the measurement. This is *not* achieved (in fact *cannot* be achieved) by calls to `MPI_Barrier`. Instead SKaMPI 5 keeps track of the differences between the local times (as reported by `MPI_Wtime`) on different processors in order to know a(n approximation of) global time. This gives much better results; see our paper in the proceedings of EuroPVM/MPI 2002.

In order to use this method (which we highly recommend, in particular for benchmarking collective operations) you should do the following:

- Use `#include "../synchronize.h"` at the top of the file.
- In the `init_` function for your measurement call `init_synchronization()`.
- In the `measure_` function for your measurement insert calls of `start_synchronization()` and `stop_synchronization()` as follows:

- `start_synchronization()` should be called immediately before the code block you want to benchmark. For convenience this call will return the current local time (it calls `MPI_Wtime()` as its last action and returns that value).
- `stop_synchronization()` should be called immediately after the code block you want to benchmark. For convenience this call will return the current local time (it calls `MPI_Wtime()` as its first action and returns that value).

You can think of `start_synchronization()` and `stop_synchronization()` as two glorified calls to `MPI_Wtime()` which in addition to taking the time carry out some administrative tasks.

### 6.2.2 Functions available inside `measure_` functions

There are several additional functions which are very useful for writing measurement function. They are available after the line

```
#include "../misc.h"
```

at the top of your file.

Of course, each measurement is carried out in a communicator, the one specified in the `SKaMPI` input file (see Section 2.5). The following functions give you programmatic access to it:

ret type	function	return value
<code>MPI_Comm</code>	<code>get_measurement_comm()</code>	the current measurement communicator
<code>int</code>	<code>get_measurement_rank()</code>	rank of calling process in the current measurement communicator
<code>int</code>	<code>get_measurement_size()</code>	size of the current measurement communicator

For the handling of communication buffers the following functions are useful:

ret type	function	return value
<code>void *</code>	<code>get_send_buffer()</code>	ptr to current send buffer
<code>void *</code>	<code>get_recv_buffer()</code>	ptr to current recv buffer
<code>void</code>	<code>set_send_buffer_usage(MPI_Aint)</code>	set size of current send buffer (bytes)
<code>MPI_Aint</code>	<code>get_send_buffer_usage()</code>	get size of current send buffer (bytes)
<code>void</code>	<code>set_recv_buffer_usage(MPI_Aint)</code>	set size of current recv buffer (bytes)
<code>MPI_Aint</code>	<code>get_recv_buffer_usage()</code>	get size of current recv buffer (bytes)

The following functions “guessess” in a relatively simple-minded way how much memory is needed for storing a certain number of data elements of a specified `MPI_Datatype`. Be warned that the return value may be wrong for datatypes with holes and funny offsets. Help for improvements is welcome.

ret type	function
<code>MPI_Aint</code>	<code>get_extent(int, MPI_Datatype)</code>

Because it is not so simple to find out, how many bytes are sent, e.g. by an `MPI_Send`, you can and always should tell `SKaMPI` explicitly, which number should be reported in the output file:

ret type	function	
void	set_reported_message_size(MPI_Aint)	set current reported message size (bytes)
MPI_Aint	get_reported_message_size()	get current reported message size (bytes)

If you need to allocate memory, the following functions may be useful. They allocate the specified number of elements of the specified data-type with `malloc` and check whether the memory is really available. The check is doing using `assert`, i.e. the program is aborted if not enough memory is available.

ret type	function	return value
double *	malloc_doubles(int)	pointer to memory segment
int *	malloc_ints(int)	pointer to memory segment
char *	malloc_chars(int)	pointer to memory segment
MPI_Request *	malloc_reqs(int)	pointer to memory segment

Some simple mathematical functions:

ret type	function	return value
int	imax2(int, int)	maximum of two integers
int	imin2(int, int)	minimum of two integers
int	imax3(int, int, int)	maximum of three integers
double	fmin2(double, double)	minimum of two doubles
double	fmax2(double, double)	maximum of two doubles
double	fsqr(double)	square of a double

### 6.3 Writing a new iterator

The prototype of an iterator function has to look like this:

```
<ret.type> iterator_<itname>(void ** self, <form.args>)
```

The iterator would then be used in the SKaMPI 5 input file in the form

```
for <var> in <itname>(<act.args>) do ...od
```

i.e. the `self` is not visible at the point of usage. It is provided by the runtime environment of the interpreter; all the other arguments are taken from the *<act.args>*.

The `self` argument is used internally to realize the following protocol:

- For getting the real value from the iterator, it is called with `*self==NULL`. This is the indication for the iterator, that it should initialize itself in whatever way is reasonable.

The iterator can change `*self` to a pointer to real memory, where it can store some internal state between subsequent calls.

The return value from the function is the initial value assigned to the loop variable.

- For each iteration the function is called again and again.

- The first time the iterator is called after it has produced its last reasonable value, it should set `*self` to `NULL` again, indicating to the runtime environment that it is exhausted.

The environment will then know that the whole loop has been completed, and the return value will be ignored.

As an example (taken from `measurements/demo.c`) have a look at the following simple iterator, which is to produce all integers starting from a `start` value up to, but excluding an `end` value:

```
int int_state;

int iterator_range(void **self, int start, int end)
{
    if( *self == NULL ) {      // we are called the first time; let's initialize!
        *self = &int_state;    // use this for storing some internal state
        int_state = start+1;    // remember what to return next time
        return start;
    };

    if( int_state >= end ) {    // our range is exhausted
        *self = NULL;          // tell the environment, that we're done
        return 42;              // the return value will be ignored
    }

    // in all other cases
    return int_state++;         // return next value and update internal state
}
```

The disadvantage of this iterator is that it uses a global variable for storing its internal state. As a consequence you cannot have two (or more) nested loops which both use the range iterator. For this to be possible, the storage for the internal state of the iterator should be `malloced` during initialization and `freed` at the end. See the function `iterator_squares()` in `measurements/demo.c` for an example.

## 6.4 Writing a new helper function

(to be done)

## 6.5 Building an extended version of SKaMPI

(to be done)

## 7 Miscellaneous

### 7.1 Format of output file

As long as there is no better description, here is an example. Assume that we have SKaMPI running on four processors with a simple measurement block like this:

```
begin measurement "MPI_Bcast-length"
  for nodes = 2 to 4 do
    for count = 1 to ... step *sqrt(2) do
      measure comm(nodes) Bcast(count, MPI_INT, 0)
    od
  od
end measurement
```

produces the following lines in the output file:

```
begin result "MPI_Bcast-length"
nodes= 2 count= 1   4  67.3   1.4    9  11.3  67.3
nodes= 2 count= 1   4  63.8   0.4    9   9.0  63.8
nodes= 2 count= 2   8  65.3   0.9   13   9.1  65.3
nodes= 2 count= 3  12  64.6   0.9    8   9.2  64.6
...
nodes= 3 count= 64  256  133.9   0.1   11  12.5  133.9  108.7
nodes= 3 count= 91  364  167.2   0.8   11  13.7  167.2  128.5
nodes= 3 count= 128 512  207.2   1.2    9  13.0  207.2  159.8
...
nodes= 4 count= 91  364  275.8   3.1   10  22.0  175.6  148.1  275.8
nodes= 4 count= 128 512  326.7   1.6    8  22.5  216.7  174.0  326.7
nodes= 4 count= 181 724  394.7   2.1   13  20.6  267.3  207.6  394.7
nodes= 4 count= 256 1024 508.0   3.0    8  20.7  345.1  265.9  508.0
...
end result "MPI_Bcast-length"
```

Let us call a *field* something which is separated from other fields in a line by whitespace.

- If there are  $k$  nested loops, the first  $2k$  fields are names and values of all loop variables, with the outermost loop variable always listed first.
- The next field is always the value set with `set_reported_message_size()`.
- The next three fields are
  - the result time in microseconds,
  - the standard error in microseconds and
  - the number of single measurements that have actually been done for the given set of parameters.

- All following fields are the result times of different processes.

At least for benchmarks collective communication for large communicators one does *not* want to know the times of all processes. Therefore SKaMPI uses the following approach. You can call a function `set_max_nr_node_times( $\langle max \rangle$ )` with a non-negative integer  $\langle max \rangle$  in the skampi input file to limit to  $\langle max \rangle$  the maximum number of times measured on single processes which are reported in the skampi output file. By default, if you do not call `set_max_nr_node_times()` yourself, skampi behaves as if you had called `set_max_nr_node_times(16)`.

It remains to explain *which* times are reported if there are more than M. First, the running times of all processes are sorted. Then

- if  $\langle max \rangle$  is 0: no running time is reported.
- if  $\langle max \rangle$  is 1: the "middle" running time is reported.
- if  $\langle max \rangle$  is  $\geq 2$ : the smallest and the largest time are reported as well as  $\langle max \rangle - 2$  "equidistantly distributed" in between.

## 7.2 Preparing diagrams from SKaMPI output files

SKaMPI 4 had a so-called report generator. At the moment the SKaMPI 5 distribution is still lacking such a tool. You cannot use the old one, because we had to change the format of the output files.

Since the format of SKaMPI's output files is sufficiently straightforward, for simple use cases you can use gnuplot directly (using its 'using' feature ...). Here is a simple example. Assume that your .sko file includes a section with some measurement results looking like this:

```
# begin result "Pingpong_Send_Recv"
iterations= 1      4      6.0      0.0      160      5.7      5.7
iterations= 2      4      4.5      0.0      80      4.5      4.4
iterations= 3      4      4.0      0.0      80      3.9      3.9
iterations= 4      4      3.8      0.0      80      3.7      3.7
iterations= 6      4      3.5      0.0      80      3.5      3.5
iterations= 8      4      3.4      0.0      80      3.3      3.3
iterations= 11     4      3.2      0.0      80      3.2      3.2
iterations= 16     4      3.2      0.0      80      3.1      3.1
iterations= 23     4      3.1      0.0      80      3.1      3.1
iterations= 32     4      3.1      0.0      80      3.1      3.1
iterations= 45     4      3.0      0.0      80      3.0      3.0
iterations= 64     4      3.0      0.0      80      3.0      3.0
iterations= 91     4      3.0      0.0      80      3.0      3.0
iterations= 128    4      3.0      0.0      80      3.0      3.0
iterations= 181    4      3.0      0.0      80      3.0      3.0
iterations= 256    4      3.0      0.0      80      3.0      3.0
iterations= 362    4      3.0      0.0      80      3.0      3.0
iterations= 512    4      3.0      0.0      80      3.0      3.0
# end result "Pingpong_Send_Recv"
```

We have included this as file `demo.sko` in the SKaMPI 5 distribution.

What you (probably) want to plot is how the measured roundtrip time of the pings, which can be found in field 4 of each line, changes as the number of roundtrips, which can be found in field 2 of each line, increases. Here we number the fields starting from 1, because `gnuplot` does it that way, too. Therefore basically the `plot` command to be given to `gnuplot` should look like this:

```
plot "demo.sko" using ($2):($4) with linespoints title "Pingpong\\_Send\\_Recv"
```

The important point here is the use of `($2)` and `($4)` for selecting the second and fourth field of each data line to be used for plotting. A demo `gnuplot` file `demo.gpl` is included in the SKaMPI 5 distribution which actually does a little bit more. If you run the command

```
gnuplot demo.gpl
```

an eps file named `demo.eps` will be generated.

### 7.3 Our TODO list

There are some nice features in SKaMPI 4 which have not yet been integrated into SKaMPI 5, as well as some other nice ideas. The following topics will be addressed by forthcoming releases of SKaMPI:

- Auto-refinement (more precise measurement at interesting spots like steps in the output graph) for 2 and more dimensions. This case can get complicated.
- Self-defined data-types: support exists, only predefined constructor functions are still missing, but can be implemented very easily (example is `mpi_type_contiguous` in `measurements/datatypes.c`)
- virtual topologies
- an include mechanism for `ski` files
- some test cases

## 8 Pitfalls of benchmarking

TODO:

- show how SKaMPI can help to avoid at least of the pitfalls of benchmarking listed in the paper “Reproducible Measurements of MPI Performance Characteristics” by Gropp and Lusk (EuroPVM/MPI 1999);
- discuss the particular difficulties of benchmarking MPIIO;

## 9 Acknowledgements

We are grateful to all the users who contributed in one way or another to improvements of SKaMPI 5. We’d like to thank (in alphabetical order) Kevin Ball and Phil Livermore for taking the time to give us some feedback.