SKaMPI: The Special Karlsruher MPI-Benchmark User Manual[1]

– updated and revised version for *SKaMPI* 4 by Werner Augustin

R. H. Reussner
Universität Karlsruhe
Department of Informatics
Germany
reussner@ira.uka.de

December 3, 2002

[1]This document originally appeared as Interner Bericht (Technical Report) 99/02 at the Department of Informatics, Universität Karlsruhe, Germany

Abstract

SKaMPI is the Special Karlsruher MPI-Benchmark. *SKaMPI* measures the performance of MPI [6][2] implementations, and of course of the underlying hardware. It performs various measurements of several MPI functions. *SKaMPI*s primary goal is giving support to software developers. The knowledge of MPI function's performance has several benefits: The software developer knows the right way of implementing a program for a given machine, without (or with shortening) the tedious time costly tuning, which usually has to take place. The developer has not to wait until the code is written, performance issues can also be considered during the design stage. Developing for performance even can take place, also if the considered target machine is not accessible.

MPI performance knowledge is especially important, when developing portable parallel programs. So the code can be developed for all considered target platforms in an optimal manner. So we achieve *performance* portability, which means that code runs without time consuming tuning after recompilation on a new platform.

Contents

Acknowledgements

This technical report mainly offsprings from my diploma thesis [4]. I would like to express my gratitude to my advisers P. Sanders and L. Prechelt. Especially the algorithm for automatic parameter refinement is based on ideas of P. Sanders. I would like to thank for many fruitful discussions.

Chapter 1

Running *SKaMPI*

1.1 Introduction

SKaMPI is the Special Karlsruher MPI-Benchmark. *SKaMPI* measures the performance of MPI implementations, and of course of the underlying hardware. It performs various measurements of several MPI (Ver. 1.1) functions. The results are stored in a text file, from which a report can be generated automatically.

*SKaMPI*s primary goal is giving support to software developers. Software developers are faced with severals problems when designing and implementing code for parallel environments. First of all the code has to show the best performance. This implies that a program's performance has to be measured and tuned during numerous sessions. Further on, cost intensive software development is more profitable, when the product can be used on several platforms, i.e., is portable without a new tuning for each machine. The message passing interface (MPI) [6][2] is a standard for a library to program message passing machines. MPI has been created by the MPI-forum, a group of researchers from academia and industry. MPI is a big step forward towards portable software for parallel platforms, since programmers no can rely on one interface standard, instead of several vendor-dependent interfaces. Instead of principal excluding efficient ways of implementing the MPI standard on certain machines, the MPI standard comprises several similar functions. So MPI offers many alternatives when designing and implementing a parallel algorithm. These alternatives offer a great potential for optimization.

This potential is twofold: First, the knowledge of several MPI function's performance allows the software developer the right way of implementing a program for a given machine, without (or with shortening) the tedious tuning. Even better, the developer has not to wait until the code is written, performance issues can also be considered during the design stage. In fact, developing for performance even can take place, also if the considered target machine is not accessible, or a workstation is used for development, which also can lower cost of development.

Second, if the programmer knows the MPI function's performance on several machines, the programs can be developed for performance for all considered target platforms. So we can speak of a *performance* portability, instead of *compile* portability. Compile portability means that a parallel program, developed

and tuned on platform A, is recompiled on platform B, and has to be tuned for platform B. So this in not what we really understand under portability. Unlike compile portability, performance portability means that a program is developed with MPI function's performance on all targeted platforms in mind, so that you really just have to recompile.

The *SKaMPI* project [5] tries to support these goal of performance and performance portability through two issues: First we offer a user configurable benchmark suite and a report generator, downloadable from the web. So each user can measure the performance of accessible machines in terms of MPI, generate a report, and can draw its own conclusions from this. Second, we provide a public result database, where we store *SKaMPI*s results from many machines, if permitted. So, please, email a copy of your result file to us (that is: `reussner@ira.uka.de`). So you can support performance portability and design for performance, because for these concepts we need the data of many machines.

1.2 Installation

1.2.1 Getting *SKaMPI*

The easiest way to obtain the *SKaMPI* packet is to download it from the *SKaMPI* homepage: `http://liinwww.ira.uka.de/~skampi/` The *SKaMPI*-file you find there is a gnu-zipped tar-file. Thus you can unpack it with `tar -xvzf skampi.tgz`[1].
However, this will create the whole directory-tree of *SKaMPI*:

```
/skampi
/skampi/report_generator
```

In the *SKaMPI* directory are the source files you need for compiling *SKaMPI*. In the directory `skampi/report_generator` you will find the report generator and its driver files.

1.2.2 Compiling *SKaMPI*

The benchmark program itself consists of one source-file (`skosfile.c`[2]), so that you can compile it with just one compiler call.[3] This compiler call depends on your machine. When using `mpich`, you usually have a makefile, so just call `make skosfile`. On an IBM SP under AIX call `mpcc -lm -o skosfile skosfile`. However, note that the math-library (`-lm`) is necessary for linking. You should not request any optimizations by the compiler. Some of *SKaMPI*s function calls do not have many parameters. The compiler would load the parameter into registers. This would give an unrealistic touch to our data, since this would not happen in realistic "real" applications. Also *SKaMPI* contains empty dummy

[1] If your version of tar has no option z, you can call gnu-unzip first (`gunzip skampi.tgz` and then tar (`tar -xvf skampi.tar`)

[2] `sk`ampi-in-one`source`file

[3] During development we use several modules, which are merged together to `skosfile.c`. This eases distribution, versioning, and compiling and on the target platforms. If you are interested in reusing the code, please send an email to obtain the modules, which probably eases understanding of the code.

functions, just created to measure the overhead of a function call. These function should also no be optimized away.

1.3 Running *SKaMPI*

Unfortunately starting an MPI program is as dependent on your system as compiling. Usually you can start MPI programs with the `mpirun`-command, but there is no standard for its parameters. Using mpich you start the benchmark with `mpirun -np 16 skosfile` with 16 processes. Note: Some systems like the IBM SP have a different command for starting parallel programs (`poe`) than mpirun. In case of trouble, please ask your local system administrator.
SKaMPI needs to be started with two or more processes. How many you use, depends on what you want to measure.[4] Some operating environments request further information on the program to start, such as memory or time requirements. The memory needed by *SKaMPI* is specified with the @MEMORY keyword in the parameter file (`.skampi`). (Please see section 2.1 for further information about the parameter file.) As rule of the thumb you should give a megabyte extra, for internal buffers, etc. The time that *SKaMPI* needs to measure depends on the accuracy you request, and the number of measurements you asked *SKaMPI* to perform.[5] To mention a typical value: *SKaMPI* runs with all measurements and an accuracy of 3 percent less than half an hour on an IBM SP using 16 nodes using an 8 MB message buffer.
SKaMPI stores its results in a text file. The name of this text file is `skampi.out` by default. To change that edit the `@OUTFILE` line in the parameter file (see 2.1.1). If other processes run during measuring, they may disturb *SKaMPI*. So you might find it useful running *SKaMPI* more than once. For every run *SKaMPI* creates a new output file `skampi.out.1`, `skampi.out.2` and so on. Note that the results of the actual run are always stored in `skampi.out`.
The other file *SKaMPI* creates is a log file (`skampi.log`)[6]. It is used by the recovery-mechanism. But you may also have a look into it. Several warnings and comments are stored in it.

> Before starting the Benchmark we urgently recommend to fill out the `@MACHINE`, `@NODE` and `@NETWORK` parts of the parameter file `.skampi` in a detailed manner.
>
> `@COMMENT` Section for comments. You may enter any text you want. (Well, text without other section names, of course!)
>
> `@MACHINE` The text in this section describes the machine, you run *SKaMPI* on. You can add any other relevant details of a measurement here. Note that there are also special sections for the network (`@NETWORK`) and the nodes (`@NODE`). *SKaMPI* assumes that the first line of the @MACHINE-section contains just the name of the machine.

[4] Well, you may ask, what is measured. For a quick overview please have a look in the example report `skarep.example.ps` or in the section 1.5. A more detailed technical description you will find in section 3.1.

[5] You can change them in the `@STANDARDERROR`- and `@MEASUREMENTS`-section respectively. You also can give a time limit for measurements through the sections `@TIMESUITEDEFAULT` and `@TIMEMEASDEFAULT`. (For further information please see Section 2.1.4.)

[6] Its name can be changed in the `@LOGFILE` line of the parameter file.

`@NODE` In this section you may describe the type of nodes you use. If there are several types, please describe them all, e.g., in terms of processor type, clock rate, caches and memory (types, organization, size), operating system, etc.

`@NETWORK` Here you may type in, which network you use. Often there are several versions of a communication network for one machine (for example the IBM SP).

`@USER` Here is your place. The first line of this section is used by the report-generator (`dorep.pl`) and should contain only your name.

The report generator requires these data to create a report of the results.

1.3.1 Changes to previous *SKaMPI* version

The most important change from version 3 to version 4 is the addition of a second measurement method for collective operations — called synchronous measured collective measurement — which is much more accurate (details in 3.1.5). To distinguish between the two measurement methods a new naming scheme was introduced (see section 1.5.3). To prevent users from unintentionally using parameter files with the old naming convention *SKaMPI* looks for a new `@VERSION` parameter. Though everything works fine if a line `@VERSION 4.0` is introduced, users are strongly encouraged to also adjust the measurement names (you only have to add the ending `-BM` to the measurements of collective operations) or use one of the provided parameter files.

1.3.2 The different parameter files

During start *SKaMPI* reads a parameter file `.skampi`. Currently *SKaMPI* ships with four different configuration files. They use the new measurement method for collective operations described in the previous section, so their results can't be compared without further thought with results obtained with previous versions of *SKaMPI*.

.skampi The standard configuration file. It containes the same suites of measurements like the well-known `.skampi` of *SKaMPI* 2. This file does not contain any user defined datatypes. `MPI_Int` will be used for all measurements. When starting *SKaMPI* without any renaming any file, this file will be used.

.skampi-dt-short This is a short configuration file to measure the performance of user defined datatypes. The basetype is `MPI_Int` and the various constructors are used and compared.

.skampi-dt-long This is a long configuration file to measure the performance of user defined datatypes. Additionally to the measurement suites of the `.skampi-dt-short` this file also contains measurements with a more complex basetype. Use this configuration file to compare flat and deep MPI user defined types.

.skampi-all-collectives This parameter file includes all collective operations measured with both measurement methods. It can be used to analyze the differences between them.

1.4 Generating a report

Since we run *SKaMPI*, we would like to know its results. Lets assume that the results are stored in `skampi.out`, which is the default. Then we just call `dorep.pl` to create a postscript report named `skampi.out.ps`.
Just call `dorep.pl other_name` if your output file is not named `skampi.out` but "`other_name`". In this case, the result will be stored in `other_name.ps`.

> A note to `dorep.pl`: As you may have seen by the file extension, the report generator is a perl-5-script. There are several reasons for using perl, perhaps the most important is, that we do not have to worry about compiling (since perl is interpreted). But there is still a little point to look at: `dorep.pl` has to find the perl-binary. Therefore its first line contains *my* path to the perl-interpreter (`#!/usr/bin/perl -w`). At some systems the path differs from this one.[7] So adaption may be required.

`dorep.pl` needs several programs to work.

Program	my Version	Purpose
perl	version 5.003	interpreting and execution
gnuplot	version 3.5, patchlevel 3.50.1.17, 27 Aug 93	Generating eps-graphics
latex	Version 3.14159 (C version 6.1)	Text formatting
dvips	dvips(k) 5.86	Converting `.dvi`-files into `.ps`-files.

Information on configuring the report generator is given in Section 2.2. Note: The report generator relies on filled entries `@MACHINE` and `@USER` as described in section 1.3.

1.5 The measurements: a short overview

This section is a short guide through all measurements, that are performed by the *standard-suite*. This suite is given in the default *SKaMPI* parameter file. Changing the parameters is shown in Section 2.1.

1.5.1 Measurement of point-to-point operations

In a ping-pong test one node sends a message to another, which returns it. For measuring point-to-point communications we use different MPI operations. All point-to-point measurements are varied over the message length. The names from the different measurements are derived from the names of the used MPI

[7] The real perl-freak knows: there is a solution for this problem, a magic line, which forces the shell to search for perl. But it does not work when using the C-shell. (So we forget it.)

operations, so for example MPI_Send-MPI_Irecv uses MPI_Send for sending and MPI_Irecv for receiving. Non-blocking operations are followed by a MPI_Wait to make sure, that the whole time for transfering the data is measured. The bandwith stated in the report is the overall bandwith, i.e. the sum of the incoming and outgoing bandwith.

MPI_Send-MPI_Recv

MPI_Send-MPI_Iprobe_Recv

This measurement waits busily calling `MPI_Iprobe` before using `MPI_Recv` to receive the message.

MPI_Send-MPI_Irecv

MPI_Send-MPI_Recv_with_Any_Tag

Similar to MPI_Send-MPI_Recv, the only difference of this measurement is the use of `MPI_ANY_TAG` in the receive function.

MPI_Ssend-MPI_Recv

MPI_Isend-MPI_Recv

MPI_Issend-MPI_Recv

MPI_Bsend-MPI_Recv

MPI_Sendrecv

MPI_Sendrecv_replace

p2p_dummy

Dummy operation used to determine the overhead of the point-to-point measurement pattern.

1.5.2 Measurements with the master worker scheme

The following measurements belong to the master worker scheme. The master dispatches concurrent subtasks to several workers. These workers send a reply for every received subtask. The measurements tests the network throughput and its handling of simultaneous communications. They can be varied over the number of subtasks (*chunks*), the length of the sent messages or the number of workers (expressed in the endings `chunks`, `length` and `nodes`). The bandwidth stated in the report is the total bandwidth of the master process.

MPI_Waitsome-{chunks|length|nodes}

These measurements use `MPI_Waitsome` to coordinate the incoming worker messages. This function guarantees a fair coordination of the workers.

MPI_Waitany-length

This measurement uses `MPI_Waitany` to coordinate the incoming worker messages. A fair coordination of the workers is not guaranteed, race conditions are possible (i.e. some workers might wait longer for terminating their communication because others were preferred).

MPI_Recv_Any_Source-length

In this measurement the master process receives the messages of the workers using `MPI_Recv` with `MPI_ANY_SOURCE` as source. No special coordination functions are used.

MPI_{Send|Ssend|Isend|Bsend}-length

In these measurements the master process uses `MPI_Send` (respectively `MPI_Ssend`, `MPI_Isend` or `MPI_Bsend`) for sending and `MPI_Recv` with an explicit source for receiving messages.

mw_dummy

Dummy operation used to determine the overhead of the master-worker measurement pattern.

1.5.3 Measurements of collective operations

The following measurements concern collective MPI operations. These operations synchronize processes i.e., `MPI_Barrier` or transmit data between them (other operations than `MPI_Barrier`). The time until completion on *all* nodes is measured. In all cases the result is the bandwidth at one node.

Version 4 introduced a new and more exact measurement method. To distinguish between these two methods the endings `-BM` and `-SM` (barrier measurement and synchronous measurement) were introduced (details of the actual measurement methods can be found in section 3.1.4 and 3.1.5). Measurements of collective operations with previous versions of *SKaMPI* correspond to the `-BM` measurements. Furthermore the following measurements can exist in four different flavours: `nodes`, `nodes-short`, `nodes-long` and `length`. The `nodes-short` and `nodes-long` measurements vary over the number of nodes and use short (256 bytes) and long (64 kbytes = 65536 bytes) message lengths, though these values are arbitrary and can be changed in the *SKaMPI* configuration file `.skampi`. The measurements with only `nodes` in their name also use 256 bytes message length (where applicable). Fnally, the `length` measurement varies over the message length, using all nodes. Measurements without further explanation use the corresponding MPI operation as it is defined in the MPI standard [3].

MPI_Bcast-{nodes-short|nodes-long|length}-{BM|SM}

MPI_Bcast_Send_Recv-{nodes-short|nodes-long|length}-{BM|SM}

This is an alternative implementation of `MPI_Bcast` using `MPI_Send` and `MPI_Recv` to send messages. A comparison between this operation and the regular `MPI_Bcast` can show the level or lack of optimization of the latter.

MPI_Barrier-{BM|SM}

MPI_Reduce-{nodes|length}-{BM|SM}

MPI_Scan-{nodes|length}-{BM|SM}

MPI_Alltoall-{nodes-short|nodes-long|length}-{BM|SM}

MPI_Alltoall_Isend_Irecv-{nodes-short|nodes-long|length}-{BM|SM}

This is an alternative implementation of `MPI_Alltoall` using `MPI_Isend` and `MPI_Irecv` to send and receive the messages.

MPI_Gather-{nodes-short|nodes-long|length}-{BM|SM}

MPI_Gather_SR-{nodes-short|nodes-long|length}-{BM|SM}

This is an alternative implementation of `MPI_Gather` using `MPI_Send` and `MPI_Recv`.

MPI_Gather_ISWA-{nodes-short|nodes-long|length}-{BM|SM}

Yet another implementation of `MPI_Gather`, this time using `MPI_Isend` and `MPI_Waitall`. It can be very interesting to compare these last three different implementation of the same operation.

MPI_Scatter-{nodes-short|nodes-long|length}-{BM|SM}

MPI_Allreduce-{nodes|length}-{BM|SM}

MPI_Reduce_Bcast-{nodes|length}-{BM|SM}

This is an alternative implementation of `MPI_Allreduce` using `MPI_Reduce and MPI_Bcast`.

MPI_Reduce_scatter-{nodes|length}-{BM|SM}

MPI_Allgather-{nodes-short|nodes-long|length}-{BM|SM}

MPI_Scatterv-{nodes-short|nodes-long|length}-{BM|SM}

MPI_Gatherv-{nodes-short|nodes-long|length}-{BM|SM}

MPI_Allgatherv-{nodes-short|nodes-long|length}-{BM|SM}

MPI_Alltoallv-{nodes-short|nodes-long|length}-{BM|SM}

MPI_Alltoallv_Isend_Irecv-{nodes-short|nodes-long|length}-{BM|SM}

Similar to the case with `MPI_Alltoall` this is an alternative implementation of `MPI_Alltoallv` using `MPI_Isend` and `MPI_Irecv` to send and receive the messages.

MPI_Reduce_Scatterv-{nodes|length}-{BM|SM}

This operation performs a data reduction operation on all participating processes with `MPI_Reduce` and then distributes the result partially to all participating nodes with `MPI_Scatterv`. Every node receives a different part of the result-array. This kind of result distribution to all participating nodes is similar

to the one of `MPI_Reduce_scatter`, so it is interesting to compare this operation to `MPI_Reduce_scatter`, which distributes the result to all nodes in one call.

MPI_Comm_split-nodes-{BM|SM}

MPI_Comm_dup-nodes-{BM|SM}

{col|syncol}_dummy

Dummy operation used to determine the overhead of the barrier (`col` respectively synchronous (`syncol` measured collective pattern.

1.5.4 Measurement of local operations

The following measurements are *local*, i.e. that they involve only one process, without any need for communication. They do not have any parameters and their names correspond to the MPI operations used. Usually they should be very fast, fast enough to be irrelevant for a user programm. But because they are used very often, it makes sense to check this claim.

MPI_Wtime

The most often used function in the *SKaMPI* benchmark programm. The duration of MPI_Wtime (together with the result of MPI_Wtick) obviously give a lower bound of the accuracy of all the time measurements performed in the whole program.

MPI_Comm_rank

MPI_Comm_size

MPI_Iprobe

Only the case with no waiting message i.e. the *unsuccessfull* `MPI_Probe` is measured because it is the common case.

MPI_attach

Sequence of `MPI_Buffer_attach` and `MPI_Buffer_detach`. The size of the attached buffer is `MPI_BSEND_OVERHEAD` (the official MPI buffered send overhead) plus a couple of bytes needed for some MPI implementations.

simple_dummy

Dummy operation used to determine the overhead of the simple measurement pattern.

Chapter 2

Customizing *SKaMPI* and trouble-shooting

This is a more detailed chapter containing information about customizing the measurements to your personal needs. Further on we introduce the recovery-mechanism of *SKaMPI*, and what's to do, when it fails.

But before that, let's make clear some terms.

Single measurement: A single call of a (MPI) routine to be measured in a pattern (see section 3.1 for *patterns*). (E.g., `MPI_Send-MPI_Recv` at 1 MB message length.)

Measurement: A measurement is the determination of a value on a (set of) parameter(s). The result of a measurement is built of several single measurements. In this benchmark the number of single measurements necessary for one measurement is determined by the accuracy wanted (and an upper and lower bound).

Suite of measurements: Measurements varied over their parameter. In the report generated by the report generator every subsection represents a suite of measurements. (E.g., `MPI_Send-MPI_Recv` from 0..16 MB message length.)

Run: A run of the benchmark is the execution of all selected suites. (Selection is done in the parameter file.) Usually for each run a report is generated.

2.1 Configuring the *SKaMPI* parameter file

2.1.1 The sections

The parameter file is a ASCII-text file describing the settings to control *SKaMPI*. The parameter file should be accessible in the directory, where *SKaMPI* is started. Its name is always `.skampi`. Thus, do not rename it. Here you can see how to adapt the parameter file to your personal needs.
The parameter file is divided into sections. Each section sets one parameter

(which may be a list). If one section is omitted, the default value for this parameter will be assumed. A name of a section always starts with an "@". A section reaches to the start of another section (or end of file). The order of the sections is irrelevant, but it may be considered practical, to use the "@MEASUREMENTS" -section as the last one. So you can see all the other (usually shorter) sections at the beginning of the parameter file. In all sections ending with "...DEFAULT" you can fill in a default value for this parameter, e.g., the value given in `STANDARDERRORDEFAULT` is used for the standard error defined in every suite, when the standard error of the suite is set do `Default_Value`.

We urgently recommend to fill out the `@MACHINE`, `NODE` and `@NETWORK` sections in a detailed manner.

`@COMMENT` Section for comments. You may enter any text you want. (Well, text without other section names, of course!)

`@MACHINE` The text in this section describes the machine, you run *SKaMPI* on. You can add any other relevant details of a measurement here. Note that there are also special sections for the network (`@NETWORK`) and the nodes (`@NODE`). *SKaMPI* assumes that the first line of the @MACHINE-section contains just the name of the machine.

`@NODE` In this section you may describe the type of nodes you use. If there are several types, please describe them all.

`@NETWORK` Here you may type in, which interconnection network you use. Often there are several versions of a communication network for one machine (for example the IBM SP).

`@USER` Here is your place. The first line of this section is used by the report-generator (`dorep.pl`) and should only contain your name.

`@MEMORY` This section is just an integer. It describes the amount of memory in kilobyte, which should be reserved for message buffers on each node, e.g. `@MEMORY 8192` means 8 megabyte message buffers.

`@VERSION` This section is just a float. It denotes the version, this configuration file was written for. Because the names of some measurements were changed (details in 1.3.1), *SKaMPI* version 4.0 stops with a warning when used with an old configuration file.

`@OUTFILE` The name of the output file. This name should also be entered in the first line (e.g. `@OUTFILE skampi.out`). Note that there is a blank between @OUTFILE and the filename!

`@LOGFILE` The name of the log file. This name should also be entered in the first line (e.g. `@LOGFILE skampi.log`). Note that there is a blank between @LOGFILE and the filename!

`@MAXSTEPSDEFAULT` This section is also just an integer. It describes the number of measurements to be performed in the parameter-range. This value is the default value for `Max_Steps`.

`@MAXREPDEFAULT` This integer describes the maximal number of measurements repetitions can be performed. It's is the default value for `Max_Repetition`.

`@MINREPDEFAULT` This integer describes the minimal number of repetitions a measurement can be performed. It's the default value for `Min_Repetition`.

`@MULTIPLEOF` Any argument a measurement is called with has to be a multiple of this integer value. For example "8" might be quite useful to avoid memory alignment effect on 64-bit machines. This integer is the default value for `Multiple_of`.

`@CACHEWARMUP` Starting with version 3, *SKaMPI* usually does not warm up the cache before measuring. (Note that older versions of *SKaMPI* did.) The reason for not warming the cache is, that user programs usually also do not warm up the cache before sending data. However, if you want to warm up the cache (both, data- and instruction cache), you can use `@CACHEWARMUP number`, where `number` is an positive integer number, giving the number of single measurements performed, before the time is measured.

`NAMEOFRUN` The name of the run describes the measurements performed in a run. The common `.skampi`-file, packaged with *SKaMPI* contains measurments of the so called `StandardSuite`.

`@TIMESUITEDEFAULT` This float sets the default value of the parameter `Time_Suite`.

`@TIMEMEASDEFAULT` Float default value of the parameter `Time_Measurement`.

`@CUTQUANTILEDEFAULT` Float default value of the parameter `Cut_Quantile`.

`@STANDARDERRORDEFAULT` Here you can enter a float, noting the max allowed standard-error for a measurement. The measurements are repeated until this accuracy is reached (unless the max. number of repetitions is reached.) `@STANDARDERRORDEFAULT 0.05` means that a standard-error of five percent is allowed.

`@ABSOLUTE` Please enter just a `yes` or a `no` in this section. If "yes", *SKaMPI* will try to correct the measured data, that is subtracting the overhead. This option should only be activated, if it is clear that there is low (or better no) other load on the machine. (Otherwise you can get negative performing-times, because the measurement of the overhead can be disturbed by the other load.) E.g. `@ABSOLUTE yes`.

`@POSTPROC` Please enter just a `yes` or a `no` in this section. You can do several runs of *SKaMPI*. Each successful run will build a new output file (e.q. `skampi.out`, `skampi.out.1`, `skampi.out.2`, ...) If "yes", *SKaMPI* will perform the post-processing. That is merging all output files together. Note if *SKaMPI* is restarted after an abort, no new output file will be created. In this case *SKaMPI* appends the results to the output file of the previous run. If you do not want *SKaMPI* to perform this kind of post-processing (`@POSTPROC no`).

`@BLOCKS, @BLOCKSIZE, @VECTORSTRIDE, @BASETYPE`*n* are used when creating user defined datatypes. They are explained in section 2.1.8.

`@MEASUREMENTS` This section describes all measurements to be performed by *SKaMPI*. Since it has its own grammar, there is an extra section devoted for it (2.1.4) in the documentation.

2.1.2 Example and default values

First we show the filled text sections. Please use them to describe your machine in detail. Note that the report generator needs this data, to correctly produce a report.

```
@COMMENT My machines at home
@MACHINE Pentium - 386 Linux Power Workstation Cluster
@NODE Pentium S 133 Mhz, i386-33Mhz
@NETWORK (slow) Ethernet, Western Digital Network adapter
@USER Ralf Reussner
```

The following examples initializes all sections with their *default values*. So here you can see, which values will be assumed by *SKaMPI*, if a section is omitted.

```
@MEMORY 4096
@VERSION 4.0
@OUTFILE skampi.out
@LOGFILE skampi.log
@MAXSTEPSDEFAULT 16
@MAXREPDEFAULT 20
@MINREPDEFAULT 4
@MULTIPLEOFDEFAULT 4
@STANDARDERRORDEFAULT 0.05
@TIMEMEASDEFAULT 0.0
@TIMESUITEDEFAULT 0.0
@BLOCKS 10
@BLOCKSIZE 7
@VECTORSTRIDE 11
@COMMENT
To use TIMEMEASDEFAULT and TIMESUITEDEFAULT please
replace the 0.0 with your required values and change
the "Invalid_Value" in each measurement to "Default_Value".
@CUTQUANTILEDEFAULT 0.25
@CACHEWARMUP 0
@NAMEOFRUN StandardSuite
@ABSOLUTE no
@POSTPROC yes
@MEASUREMENTS
```

The empty sections (like @COMMENT, or @MACHINE, etc.) are initialized empty. You may enter free text in them (text without section names). An exception is the MEASUREMENTS-Section (see section 2.1.4).

2.1.3 The grammar of the different sections

The grammar used for the above sections is shown below. Only nonterminals appear.

```
SECTION ::=       TEXT_SECTION SECTION
                | INT_SECTION SECTION
                | FLOAT_SECTION SECTION
                | YESNO_SECTION SECTION
```

```
                | MEASUREMENTS_SECTION SECTION
                | <epsilon>

TEXT_SECTION ::=  @COMMENT text
                | @MACHINE text
                | @NETWORK text
                | @NODE text
                | @USER text
                | @OUTFILE text
                | @LOGFILE text
                | @NAMEOFRUN text
                | @BASETYPE1 text
                   ...
                | @BASETYPE10 text

INT_SECTION ::=   @MEMORY int
                | @MAXSTEPSDEFAULT int
                | @MAXREPDEFAULT int
                | @MINREPDEFAULT int
                | @MULTIPLEOFDEFAULT int
                | @CACHEWARMUP int
                | @BLOCKS int
                | @BLOCKSIZE int
                | @VECTORSTRIDE int

FLOAT_SECTION ::= @STANDARDERRORDEFAULT float
                | @TIMEMEASDEFAULT float
                | @TIMESUITEDEFAULT float
                | @CUTQUANTILE float
                | @VERSION float

YESNO_SECTION ::= @ABSOLUTE
                | @POSTPROC
```

Production rules for the nonterminal `MEASUREMENTS_SECTION` are found in section 2.1.7. The nonterminals `int` and `float` are the usual C data types and `text` is plain ASCII text without quotes. More than one line of text is possible.

2.1.4 The MEASUREMENTS section

The MEASUREMENTS section is a list in which every entry describes a suite of measurements (i.e., measurements varied over their parameter range). An entry starts with the name of the measurement. This name should be usable as filename. It is followed by a fixed record, describing specific properties of this suite. An example is given in section 2.1.5. This record is explained below.

`Type` Each measurement must have a type assigned. This type (an simple integer) describes the MPI-function and the pattern which should be measured. A couple of tables in section 3.1 (page 32) shows which number is assigned to which MPI-function.

`Basetype_Number` The number of the basetype. As described in the user defined datatypes section (sec. 2.1.8), several basetypes can be defined. (`@BASETYPE`<*number*>). With the `Basetype_Number =` <*number*> you

can select the basetype for this measurement among the previously defined basetypes by its *number*.

`Send_Datatype_Number` The number of the type constructor the create the datatype, which will be used for at the sender of communication operations. The possible constructors are described in section 2.1.8.

`Receive_Datatype_Number` The number of the type constructor the create the datatype, which will be used for at the receiver of communication operations. The possible constructors are described in section 2.1.8.

`Variation` Here you can enter the variable to be used to vary over (e.g., the message length, or the number of nodes). The variables contained by a pattern you can see in Table 2.1.

`Scale` This parameter describes the scale of the x- and y-axis (linear or logarithmic) and it determines how to find the arguments for this suite (fixed or dynamic). Possible values are:

`Fixed_linear` The arguments begin at `Start_Argument` and end at `End_Argument`. The distance is `Stepwidth`. Both scales are linear. The variables `Max_Steps`, `Min_Distance` and `Max_Distance` have no meaning.

`Fixed_log` The arguments are powers of the parameter `stepwidth`. (stepwidth^1, stepwidth^2, stepwidth^3 ... until `End_Argument` has been reached.) Both axis are logarithmic. The variables `Max_Steps`, `Min_Distance` and `Max_Distance` have no meaning.

`Dynamic_linear` The arguments begin at `Start_Argument` and end at `End_Argument`. The distance is `Stepwidth`. After doing the measurements this way, the number `Max_Steps` of measurements is filled up with automatically placed measurements. These measurements are never nearer than `Min_Distance`.
Both axes are linear.

`Dynamic_log` The arguments are powers of the parameter `stepwidth`. (stepwidth^1, stepwidth^2, stepwidth^3 ... until `End_Argument` has been reached.) After having done measurements this way, the number `Max_Steps` of measurements is filled up with automatically placed measurements. These measurements are never nearer than `Min_Distance`. Both axis are logarithmic.

`Max_Repetition` Here you can enter the maximal number of measurement repetitions. If you do not want to change this value in every entry, you just write `Default_Value` instead of the number, and the value given in the `@MAXREPDEFAULT` section is used.

`Min_Repetition` Here you can enter the minimal number of repetitions performed for a measurement. If you do not want to change this value in every entry, you just write `Default_Value` instead of the number, and the value given in the `@MINREPDEFAULT` section is used.

`Multiple_of` Any argument a measurement is called with has to be a multiple of this integer value. For example "8" might be quite useful to avoid

memory alignment effects on 64-bit machines, or 4 for 32-bit systems. This integer's default value is set in the section `@MULTIPLEOF`.

`Time_Suite` The value given here sets the time limit for one suite of measurements in minutes. A suite of measurements is a set of measurements, containing measurements varied over some parameters (compare to definition at the beginning of this chapter). This means that no new measurements are started, when the time consumed by the already executed measurements of this suite exceeds this limit time[1]. This limit has no influence on other suites. So exceeding this limit time means that only this suite stops measuring. It does not mean, that the whole benchmark is aborted. Information regarding the preference of this parameter and `Max_Steps` is given in subsection 2.1.6. If you do not want to change this value in every entry, you just write `Default_Value` instead the number, and the value given in the `@TIMESUITEDEFAULT` section is used. If you do not want to give any time limit at all, please enter `Invalid_Time` instead of a value.

`Time_Measurement` This value gives the time limit for one measurement in minutes. (A measurement is the repetition of several single measurements. Compare to definition at the beginning of this chapter). This means that no new single measurements is started, when the time consumed by the already executed single measurements of this measurement exceeds this limit time[2]. Information regarding the preference of this parameter and `Standard_error` is given in subsection 2.1.6. If you do not want to change this value in every entry, you just write `Default_Value` instead of the number, and the value given in the `@TIMESUITEDEFAULT` section is used. If you do not want to give any time limit at all, please enter `Invalid_Time` instead of a value.

`Node_Times` This boolean value can be set to `yes` or `no`. In case of `yes` *SKaMPI* measures besides the result also the execution times of the measured routine on *all* nodes[3]. This may be useful to see whether overlapping communication and computation can take place, or to measure effects of contention. In the patterns Simple and Master-Worker this feature will be ignored, since in the simple pattern only one process is involved, and in the Master-Worker pattern the workers work until they receive the stop signal. So it is not interesting to measure, when the workers stop.

The times are given in microseconds in the output file. Note that the node times are only given for the last single measurement of a measurement. This means that node times do not represent a mean value of the execution times of several results as the measurement's result does. So is is possible that the result differs from the node time of process 0.

`Cut_Quantile` This value defines the upper and lower quantile of single measurements' results, which are disregarded, when computing the result of a

[1] This means that the time of all measurements can be larger than the limit, because the last measurement will not be aborted when exceeding the limit time.

[2] This means that the time of all single measurements can be larger than the limit, because the last single measurement will not be aborted when exceeding the limit time.

[3] The *result* is the time the routine to measure needs on the measuring root node. The benchmark assures that the routine to measure has finished on all other nodes, when finished on the root node. So the execution times on the individual nodes is usually lower.

Pattern	Variables to vary over
Point-to-Point	Length, Nodes
Master-Worker	Length, Nodes, Chunks
Barrier Measured Collective	Length, Nodes
Synchronous Measured Collective	Length, Nodes
Simple	none

Table 2.1: Which pattern can be varied with which variables?

measurement. If you do not want to throw any results away, use `0.0`. If you assume that the upper an lower quartile of your results are outliers, use `0.25`. If you do not want to change this value in every entry, you just write `Default_Value` instead the number, and the value given in the `@CUTQUANTILEDEFAULT` section is used.

`Start_Argument` If the `Variation` is linear, this number will be used as starting argument. (In case of logarithmic scale it has no meaning, since measurements always are started by 1.)

`End_Argument` This is the maximal argument, which is never exceeded in this measurement. If you vary over the message length it will depend on the amount of memory you entered in the `@MEMORY` section. If you vary over the number of nodes, it will depend on the number of nodes, *SKaMPI* started with. To make it easier to determine these values, you can just enter `Max_Value` here, and *SKaMPI* computes the actual values during run-time.

`Max_Steps` explained under `Variation`.

`Min_Distance` explained under `Variation`.

`Max_Distance` explained under `Variation`.

`Standard_error` Measurements are repeated until its standard error has dropped below of this value here. (But the number of repetitions is never less than `Min_Repetition` and never larger than `Max_Repetition`. The standard error is a metric for the reliability of a the data, whereas the standard *deviation* is a metric for dispersion.

2.1.5 Example of an entry

```
MPI_Send-MPI_Recv
{
  Type = 1;
  Basetype_Number = 1;
  Send_Datatype_Number = 4;
  Receive_Datatype_Number = 4;
  Variation = Length;
  Scale = Dynamic_log;
  Max_Repetition = Default_Value;
```

```
  Min_Repetition = Default_Value;
  Multiple_of = Default_Value;
  Time_Measurement = Invalid_Value;
  Time_Suite = Invalid_Value;
  Node_Times = No;
  Cut_Quantile = Default_Value;
  Default_Chunks = 0;
  Default_Message_length = 256;
  Start_Argument = 1;
  End_Argument = Max_Value;
  Stepwidth = 128;
  Max_Steps = 30;
  Min_Distance = 128;
  Max_Distance = 512;
  Standard_error = Default_Value;
}
```

2.1.6 A note to the preference of the parameters Max_Steps, Time_Suite and Standard_error, Time_Measurement

The termination of a measurement is controlled by four parameters: `Standard_error`, `Max_Repetition`, `Min_Repetition`, and `Time_Measurement`. The termination of a suite of measurements is controlled by the two parameters `Max_Steps` and `Time_Suite`. Conflicts between these parameters are resolved in the following way.

Termination of a measurement

If `Time_Measurement` is set to `Invalid_Value` than (a) the number of single measurements is always between `Min_Repetition` and `Max_Repetition`, (b) if the the standard error of the single measurement's results fall below `Standard_error` the measurement is finished. (If the single measurements are repeated `Max_Repetition` time, than the measurement is also finished, independent of the value for the standard error.)

If `Time_Measurement` is set to any other value as `Invalid_Value` (that is a float or `Default_Value`), then no further single measurement will be started, when the sum of the execution times of the already executed single measurements exceeds the value of `Time_Measurement`. The values of `Standard_error`, and `Min_Repetition` will not be considered in this case. But in any case, there will not be more measurements started than `Max_Repetitions`[4]. If you want to use only `Time_Measurement` to control the termination, so choose a high value for `Max_Steps`.

Termination of a suite of measurements

If `Time_Suite` is set to `Invalid_Value` then the number of measurements in this suite equals always to `Max_Steps`.

If `Time_Suite` is set to any other value as `Invalid_Value` (that is a float or `Default_Value`), then no further measurement will be started, when the sum

[4] This is because *SKaMPI* uses these values for internal buffer allocation.

of the execution times of the already executed measurements exceeds the value of `Time_Suite`.

2.1.7 Grammar of the MEASUREMENTS section

The grammar used for the measurement section is shown below. Terminals are set in "", nonterminals not.

```
MEASUREMENTS_SECTION ::=
          file_name_str
          ``{''
          ``Type =''TYPE_RANGE``;''
          ``Variation =''VARIATION_STYLE``;''
          ``Scale =''SCALE_STYLE``;''
          ``Max_Repetition =''INT_OR_DEFAULT``;''
          ``Min_Repetition =''INT_OR_DEFAULT``;''
          ``Multiple_of ='' INT_OR_DEFAULT``;''
          ``Time_Measurement ='' FLOAT_OR_DEFAULT_OR_INVALID``;''
          ``Time_Suite ='' FLOAT_OR_DEFAULT_OR_INVALID``;''
          ``Cut_Quantile =''FLOAT_OR_DEFAULT``;''
          ``Default_Chunks =''INT_OR_FLOAT``;''
          ``Default_Message_length =''INT_OR_FLOAT``;''
          ``Start_Argument =''int``;''
          ``End_Argument =''INT_OR_MAX``;''
          ``Stepwidth =''int``;''
          ``Max_Steps =''int``;''
          ``Min_Distance =''int``;''
          ``Max_Distance =''int``;''
          ``Standard_error =''FLOAT_OR_DEFAULT``;''
          ``}''

VARIATION_STYLE ::= ``Length''
                  |  ``Nodes''
                  |  ``Chunks''

SCALE_STYLE ::= ``Fixed_linear''
              |  ``Fixed_log''
              |  ``Dynamic_linear''
              |  ``Dynamic_log''

INT_OR_DEFAULT ::= int
                 |  ``Default_Value''

INT_OR_FLOAT ::= int
               |  float

MAX_OR_DEFAULT ::= int
                 |  ``Max_Value''

FLOAT_OR_DEFAULT ::= float
```

```
                    |  ``Default_Value''

FLOAT_OR_DEFAULT_OR_INVALID ::= float
                              |  ``Default_Value''
                              |  ``Invalid_Value''
```

`file_name_str` is what your operating system allows as a file name. In the grammar above `file_name_str` stands for the name of the measurement. In the report generator `dorep.pl` there will be some files created temporarily, which contain this string in their names.
As above, the nonterminals `int` and `float` are what you would expect as C-Programmer.

Hint for editing the `@MEASUREMENTS` section: if you want to skip some measurements, just write `@COMMENT` before the measurements you intend to skip, and `@MEASUREMENTS` behind them.

2.1.8 User-defined types in *SKaMPI*

Basetypes

A basetype is a type which is used to define other types (such as structs or vectors). A basetype can be given in two ways:

- as a string, denoting a predefined MPI datatype. Currently `MPI_CHAR`, `MPI_BYTE`, `MPI_INT`, `MPI_LONG`, `MPI_DOUBLE`, and `MPI_LONG_DOUBLE` are allowed.
- or as a comma-separated list of triples (see below.)

Each triple describes an element of the basetype. The first entry of a triple gives the length of the element in bytes. The second element gives the displacement of this element in bytes, i.e. the address where the element starts relative to the beginning of the basetype. The unit of this offset is extent of the given predefined type (the last element). The last element of the triple is the name of a MPI predefined datatype like `MPI_DOUBLE`. For example, `(1,0,MPI_BYTE),(2,1,MPI_INT)` denotes a basetype constructed out of one `MPI_BTYE` at the beginning (offset 0) and two `MPI_INT` at offset one (so behind the one `MPI_BYTE`).

Currently up to 10 basetypes can be specified in the *SKaMPI* parameter file with the commands `@BASETYPE1` ... `@BASETYPE10`.

Using basetypes to construct other types

SKaMPI defines several 'type constructors' to define nested types out of a user-defined basetype. These constructors are selected by a number. Since working with user-defined types requires a certain acquaintance with the appropriate MPI functions, we describe the different constructors in documenting their code.

The definitions use the constants `params.blocks`, `params.blocksize`, and `params.vectorstride`. They can be set as sections in the parameter file, i.e., `@BLOCKS`, `@BLOCKSIZE`, and `@VECTORSTRIDE`. If not explicitely set there, the default values are used (`@BLOCKS` 10, `@BLOCKSIZE` 7, `@VECTORSTRIDE` 11). Note

that the vectorstride must be larger than the blocksize and blocksize must be larger that three when using the special type constructor (number 70).

Number 1

Simply `MPI_BYTE`.

Number 2

Simply `MPI_CHAR`.

Number 3

Simply `MPI_FLOAT`.

Number 4

Simply `MPI_DOUBLE`.

Number 5

Simply `MPI_INT`.

Number 20

A user-defined basetype, selected by the `Send_Basetype_Number` respectively `Receive_Basetype_Number` in the parameter file.

Number 30

```
MPI_Type_contiguous(params.blocks * params.blocksize,_basetype[bt_number],d);
```

Number 31

```
MPI_Type_vector(params.blocks,params.blocksize,params.vectorstride,
                _basetype[bt_number],d);
```

Number 32

```
MPI_Type_create_hvector(params.blocks,params.blocksize,params.vectorstride*
                        _basetype_size[bt_number],
                        _basetype[bt_number],d);
```

Number 33

```
blen[0] = params.blocksize;
disp[0] = 0;
k = 1;
for (i=1; i<params.blocks; i++)
{
  blen[i] = params.blocksize-k;
  disp[i] = disp[i-1]+blen[i-1]+1;
  k = 1-k;
```

```
}
blen[params.blocks-1] += params.blocks/2;
MPI_Type_indexed(params.blocks,blen,disp,_basetype[bt_number],d);
```

Number 34

```
/* requires MPI 2 */
for (i=0; i<params.blocks; i++)
{
  disp[i] = params.blocksize*i+i;
}
MPI_Type_create_indexed_block(params.blocks,params.blocksize,disp,
                              _basetype[bt_number],d);
```

Number 35

```
blen[0] = params.blocksize;
offset[0] = 0;
k = 1;
for (i=1; i<params.blocks; i++)
{
  blen[i] = params.blocksize-k;
  offset[i] = offset[i-1]+blen[i-1]+1;
  k = 1-k;
}
for (i=0; i<params.blocks; i++)
{
  offset[i] *= _basetype_size[bt_number];
}
blen[params.blocks-1] += params.blocks/2;

MPI_Type_create_hindexed(params.blocks,blen,offset,_basetype[bt_number],d);
```

Number 36

```
blen[0] = params.blocksize;
offset[0] = 0;
type[0] = _basetype[bt_number];
k = 1;
for (i=1; i<params.blocks; i++)
{
  blen[i] = params.blocksize-k;
  offset[i] = offset[i-1]+blen[i-1]+1;
  type[i] = _basetype[bt_number];
  k = 1-k;
}
for (i=0; i<params.blocks; i++)
{
  offset[i] *= _basetype_size[bt_number];
}
```

```
blen[params.blocks-1] += params.blocks/2;

MPI_Type_create_struct(params.blocks,blen,offset,type,d);
```

Number 37

```
/* requires MPI 2, especially subarrays */
for (i=0; i<params.blocks; i++)
{
  blen[i] = SUBARRAYOFFSET+params.blocksize+SUBARRAYOFFSET;
  sublen[i] = params.blocksize;
  disp[i] = SUBARRAYOFFSET;
}
MPI_Type_create_subarray(params.blocks,blen,sublen,disp,MPI_ORDER_C,
                         _basetype[bt_number],d);
```

Number 50

```
MPI_Type_vector(params.blocks,len*params.blocksize,len*params.vectorstride,
                _basetype[bt_number], d);
```

Number 51

```
MPI_Type_vector(len,params.blocks*params.blocksize,
                params.blocks*params.vectorstride,
                _basetype[bt_number],d);
```

Number 52

```
MPI_Type_vector(len*params.blocks,params.blocksize,params.vectorstride,
                _basetype[bt_number],d);
#ifdef MPI_2
MPI_Type_get_extent(*d, &lb, &myextent);
#else /* and that is still the default */
MPI_Type_extent(*d, &myextent);
#endif
```

Number 53

```
blen[0] = params.blocksize;
offset[0] = 0;
type[0] = _basetype[bt_number];
k = 1;
for (i=1; i<len*params.blocks; i++)
{
  blen[i] = params.blocksize-k;
  offset[i] = offset[i-1]+blen[i-1]+1;
  type[i] = _basetype[bt_number];
  k = 1-k;
}
for (i=0; i<len*params.blocks; i++)
```

```
{
  offset[i] *= _basetype_size[bt_number];
}
blen[i-1] += i/2;
MPI_Type_create_struct(len*params.blocks,blen,offset,type,d);
```

Number 54

```
blen[0] = params.blocksize;
offset[0] = 0;
type[0] = _basetype[bt_number];
k = 1;
for (i=1; i<len; i++)
{
  blen[i] = params.blocks*params.blocksize-k;
  offset[i] = offset[i-1]+blen[i-1]+1;
  type[i] = _basetype[bt_number];
  k = 1-k;
}
for (i=0; i<len; i++)
{
  offset[i] *= _basetype_size[bt_number];
}
blen[i-1] += i/2;
MPI_Type_create_struct(len,blen,offset,type,d);
```

Number 55

```
blen[0] = params.blocksize;
offset[0] = 0;
type[0] = _basetype[bt_number];
k = 1;
for (i=1; i<params.blocks; i++)
{
  blen[i] = len*params.blocksize-k;
  offset[i] = offset[i-1]+blen[i-1]+1;
  type[i] = _basetype[bt_number];
  k = 1-k;
}
for (i=0; i<params.blocks; i++)
{
  offset[i] *= _basetype_size[bt_number];
}
blen[i-1] += i/2;
MPI_Type_create_struct(params.blocks,blen,offset,type,d);
```

Number 56

```
MPI_Type_vector(params.blocksize,len,len+params.vectorstride,
              _basetype[bt_number],&bvector1);
```

```
D16(typecounter++;)
MPI_Type_commit(&bvector1); /* superfluous(?), but why not */
MPI_Type_vector(params.blocks,1,2,bvector1,d);
```

Number 57

```
MPI_Type_vector(len,params.blocksize,params.vectorstride,
              _basetype[bt_number],&bvector2);
D16(typecounter++;)
MPI_Type_commit(&bvector2); /* superfluous(?), but why not */
MPI_Type_vector(params.blocks,1,2,bvector2,d);
```

Number 58

```
MPI_Type_vector(params.blocks,params.blocksize,params.vectorstride,
              _basetype[bt_number],&bvector3);
D16(typecounter++;)
MPI_Type_commit(&bvector3); /* superfluous(?), but why not */
MPI_Type_vector(len,1,2,bvector3,d);
```

Number 61

```
MPI_Type_vector(params.blocksize,1,2,_basetype[bt_number],&bvector);
MPI_Type_commit(&bvector);
MPI_Type_vector(params.blocks,1,2,bvector,d);
```

Number 62

```
for (i=0; i<params.blocksize; i++)
  {
    blen[i] = 1;
    disp[i] = 2 * i;
  }
MPI_Type_indexed(params.blocksize,blen,disp,
                 _basetype[bt_number],&bindex);
MPI_Type_commit(&bindex);
MPI_Type_vector(params.blocks,1,2,bindex,d);
D16(typecounter++;) MPI_Type_commit(d);
```

Number 63

```
MPI_Type_vector(params.blocksize,1,2,_basetype[bt_number],&bvector);
MPI_Type_commit(&bvector);
for (i=0; i<params.blocks; i++)
  {
    blen[i] = 1;
    disp[i] = 2*i;
  }
MPI_Type_indexed(params.blocks,blen,disp,bvector,d);
D16(typecounter++;) MPI_Type_commit(d);
isconstr = TRUE;
```

Number 64

```
for (i=0; i<params.blocksize; i++)
  {
    blen[i] = 1;
    disp[i] = 2*i;
  }
MPI_Type_indexed(params.blocksize,blen,disp,_basetype[bt_number],&bindex);
MPI_Type_commit(&bindex);

for (i=0; i<params.blocks; i++)
  {
    blen[i] = 1;
    disp[i] = 2*i;
  }
MPI_Type_indexed(params.blocks,blen,disp,bindex,d);
D16(typecounter++;) MPI_Type_commit(d);
```

Number 70

```
/* Special types (assume here that BLOCKSIZE>=3) */

MPI_Type_vector(params.blocks,1,params.blocksize,_basetype[bt_number],&comp1);
D16(typecounter++;)MPI_Type_commit(&comp1);
for (i=0; i<params.blocks; i++) {
  blen[i] = 1;
  disp[i] = i*params.blocksize;
}
MPI_Type_indexed(params.blocks,blen,disp,_basetype[bt_number],&comp2);
D16(typecounter++;)MPI_Type_commit(&comp2);
for (i=0; i<params.blocks; i++) {
  blen[i] = params.blocksize-2;
  offset[i] = i*params.blocksize*_basetype_size[bt_number];
  type[i] = _basetype[bt_number];
}
#if MPI_2
MPI_Type_create_struct(params.blocks,blen,offset,type,&comp3);
#else
MPI_Type_struct(params.blocks,blen,offset,type,&comp3);
#endif
D16(typecounter++;)MPI_Type_commit(&comp3);
blen[0] = 1; offset[0] = 0; type[0] = comp1;
blen[1] = 1; offset[1] = _basetype_size[bt_number]; type[1] = comp2;
blen[2] = 1; offset[2] = 2*_basetype_size[bt_number]; type[2] = comp3;
#ifdef MPI_2
MPI_Type_create_struct(3,blen,offset,type,d);
#else
MPI_Type_struct(3,blen,offset,type,d);
#endif
```

Type numbers	Pattern	Prefix
1-9,29,34	Point-to-Point	
10-16,30	Master-Worker	
17-23,31,33,35-46,72,74,76,78	Barrier Measured Collective	
24-28,32	Simple	
50-53	internal measurements	
54-71,73,75,77,79	Synchronous Measured Collective	

Table 2.2: The mapping of patterns to prefixes

2.2 Configuring the report generator

The report generater `dorep.pl` usually comes along with a standard configuration file (`.dorep`). So it needs no further configuration. But if you want to create specialized reports and to fully exploit the generator you can make use of its various parameters and features. This section only provides a brief overview. More detailed information can be found on the following URL in the internet:

`http://liinwww.ira.uka.de/ skampi/dorep3.html`

`dorep` inspects which measurements are performed and processes their results. So if you add or omit measurements, they will automatically appear in (or respectively disappear from) the report.

2.2.1 Comparisons

What the generator does not know is, which measurements you want to compare[5]. To manipulate the "Comparisons" section in `skarep.ps` you can edit the `.dorep` file. This file has a simple structure. Every line describes one comparison. The first part of the line is the name of the comparison. This name may be a normal string, but it must not contain any ":", because that is its delimiter. After the ":" follows a list with names of suites of measurements.

```
Name of the comparison: suite1, suite2, suite3
```

Note that the lists are separated by ",". But where to get the names of the suites from? For that you may have a look in the parameter file `.skampi`.

As explained in the section 2.1.1 each suite of measurements has its own name (usually the name of the MPI function measured). It may happen, that one MPI function is used in two (or more) patterns, so you have to add a prefix, describing the pattern[6].

Table 2.2 shows the patterns prefixes. For example you want to compare the first two suites in `.skampi`:

1. We want to name our comparison: `Comp. MPI_Send-MPI_Recv and MPI_Iprobe (followed by MPI_Recv)`.

[5] Here a comparison is a plot of two or more function graphs. The report generator also creates a table with some results to compare.

[6] The problem of identifying the suite with a name, which may occur twice, does not exist in `.skampi`. Here the corresponding pattern is stored with the name, so that it is always clear, what suite is called.

2. In `.skampi` you find the name `MPI_Send-MPI_Recv`. This is the name of one suite we want to see in our comparison.
   The other suite is called `MPI_Send-MPI_Iprobe_Recv`.
3. Since both suites belong to the point-to-point pattern, table 2.2 tells us we have to add the prefix `p2p_`.
4. The resulting line in `.dorep` is:
   `Comp. MPI_Send-MPI_Recv and MPI_Iprobe (followed by MPI_Recv): p2p_MPI_Send-MPI_Recv, p2p_MPI_Send-MPI_Iprobe_Recv`.
   Note: this has to be written as *one* line.

For every comparison you have to ensure that the first suite's parameter range includes the parameter ranges of the other suites. `dorep` does not check the meaning of a comparison.

2.2.2 Additional tex-modules

Besides the comparisons, there is another simple way to create more individual reports. If you create a tex-module with the extension `.tma` (tex module additional), this file will be included automatically in front of the "Comparison" section. Here a "tex-module" is a file which contains tex-commands which can occur between `\begin{document}` and `\end{document}`.

Example

```
\section{Comments}
My opinion of SKaMPI: delete it!
Oops!
```

2.2.3 More detailed graphs

If you want a more detailed graph of a special parameter range, you may edit the skampi.out in the following way.

```
/*@inp2p_MPI_Bsend-MPI_Recv.ski*/
#Description of the MPI_Bsend-MPI_Recv measurement:
#Pattern: Point-to-Point varied over the message length.
#The x scale is linear, automatical x wide adaption,
#range: 0 - 256, stepwidth: 16.000000.
#default values: 2 nodes.
#max. allowed standard error is 10.00 %
#Format: message length (%d) <space> time (microsec.)
   (%f) (standard error) (%f) count (%d)
#arg result standard_error count
0 7004.000000 1.000000 2
16 7316.000000 3.000000 2
32 11538.000000 2716.566473 6
40 7498.500000 6.500000 2
```

Edit the `range` line. For example you may write `range: 16 - 128` if you are only interested in this part of the graph.

2.2.4 Given module files

Another possibility manipulate the reports is to use your own *module* files. For every suite `suite-name` the report generator creates a gnuplot-command file named `suite-name.gpl` and a tex module file `suite-name.tmd`. If the `dorep.pl` finds such a file, it uses the your given file[7].

2.2.5 Extra text for suites

For every suite of the standard parameter file an extra text is printed as header. This text is stored in a an ASCII-text file `suite-name.dri`[8].

2.3 When *SKaMPI* crashes

Since the MPI-implementations are no trivial pieces of software[9], we have to assume that *SKaMPI* may crash while measuring. In this case all measured suites are stored, only the actual one is lost.
In this case you can use the automatic recovery mechanism. Simply start *SKaMPI* again. Please do not change the output or log file. *SKaMPI* tries to find out which measurement caused the trouble. Then *SKaMPI* skips the measurement and starts with the measurement behind. The erroneous measurement will be called `after` all others. So if it crashes again, you will have completed all other measurements. This mechanism will also work, if several measurements crash.
If this does not work, you can recover manually.

1. Find out which measurement caused the crash. In order to do this, look into `skampi.out`, go to the end of file and backward-search the string "/*@in" You will find the name of the last completed measurement after that string.

   ```
   ...
   #-------------------------------
   #/*@inp2p_MPI_Send-MPI_Irecv.ski*/
   #Description of the MPI_Send-MPI_Irecv measurement:
   #Pattern: Point-to-Point varied over the message length.
   ...
   ```

   So the name we look for is `p2p_MPI_Send-MPI_Irecv`.

2. Edit `.skampi`. Here you replace `@MEASUREMENT` with `@COMMENT` (You switch of all measurements).

3. Then find the entry of the crashed measurement. The crashed measurement is the measurement behind the last completed measurement, you

[7] To see which files are created temporarily by `dorep.pl` just comment out its line "`unlink @files_to_delete;`". Then you may have a look into its files. But be careful: Before the next run of the generator delete these files manually, because the generator does not overwrite them as explained above. (Delete the files:`*.tmd *.gpl *.eps`.)

[8] `dri` means "dorep-information".

[9] And (err) *SKaMPI* neither...

know from above. Write `@MEASUREMENTS` *after* the crashed measurement entry. In our case if `MPI_Send-MPI_Irecv` is the last completed measurement, then `MPI_Send-MPI_Recv_with_Any_Tag` failed. Therefore we place `@MEASUREMENTS` before the next entry (i.e., `MPI_Ssend-MPI_Recv`).

```
...
MPI_Send-MPI_Recv_with_Any_Tag
{
  Type = 4;
  Variation = Length;
  Scale = Dynamic_log;
  Max_Repetition = Default_Value;
  Min_Repetition = Default_Value;
  Multiple_of = Default_Value;
  Time_Measurement = Invalid_Value;
  Time_Suite = Default_Value;
  Node_Times = Yes;
  Cut_Quantile = Default_Value;
  Default_Chunks = 0;
  Default_Message_length = 256;
  Start_Argument = 0;
  End_Argument = Max_Value;
  Stepwidth = 1.414213562;
  Max_Steps = Default_Value;
  Min_Distance = 2;
  Max_Distance = 512;
  Standard_error = Default_Value;
}
@MEASUREMENTS
MPI_Ssend-MPI_Recv
{
  Type = 5;
  Variation = Length;
:

        ...
```

4. Delete the current logfile `skampi.log`.
5. Rename `skampi.out` to another file.
6. Start *SKaMPI* again with the same command.
7. When *SKaMPI* finished, you can append the new `skampi.out` file to the old renamed one.

Chapter 3

Measurements in detail

In the last chapter of this manual the different measurements are treated in detail. First we explain how to get the measured code for each measurement. In the last section we will see the format of the oubtput file.

3.1 But what is measured?

So far we know how to measure, but what is actually measured? Since we investigate parallel operations, we have to coordinate several processes. Measurements, which have a similar coordination of its processes, are grouped to a so called *pattern*. To get an impression of the working of these pattern and the functions called, there is an example in the next section. The following sections then introduce every single pattern.

3.1.1 Example

Lets ask, what is measured in type 16? First we have a look at table 3.1 (page 32) and see that the measurement type 16 belongs to the master-worker-pattern. Table 3.3 (page 33) shows that it is initialized with the function `mw_init_Bsend`.

Type numbers	Pattern
1-9,29,34	Point-to-Point
10-16,30	Master-Worker
17-23,31,33,35-46,72,74,76,78	Barrier Measured Collective
24-28,32	Simple
50-53	internal measurements
54-71,73,75,77,79	Synchronous Measured Collective

Table 3.1: The mapping of type numbers to patterns

The internal measurements are used to determine the overhead and the accuracy of measurements. The order of new measurements has somehow grown historically, but any reordering would break a lot of things (results of old measurements, old configuration files etc.)

Number	MPI-function(s)	Initializer
1	MPI_Send-MPI_Recv	p2p_init_Send_Recv
2	MPI_Send-MPI_Recv_any_tag	p2p_init_Send_Recv_AT
3	MPI_Send-MPI_IRecv	p2p_init_Send_Irecv
4	MPI_Send-MPI_Iprobe_MPI_Recv	p2p_init_Send_Iprobe_Recv
5	MPI_Ssend-MPI_Recv	p2p_init_Ssend_Recv
6	MPI_Isend-MPI_Recv	p2p_init_Isend_Recv
7	MPI_Bsend-MPI_Recv	p2p_init_Bsend_Recv
8	MPI_Sendrecv	p2p_init_Sendrecv
9	MPI_Sendrecv_replace	p2p_init_Sendrecv_replace
29	dummy Point-to-point measurement	p2p_init_dummy
34	MPI_Issend	p2p_init_Issend

Table 3.2: The mapping of type numbers to measured MPI-functions (point-to-point pattern)

Number	MPI-function(s)	Initializer
10	MPI_Waitsome	mw_init_Waitsome
11	MPI_Waitany	mw_init_Waitany
12	MPI_Recv_Any_Source	mw_init_Recv_AS
13	MPI_Send	mw_init_Send
14	MPI_Ssend	mw_init_Ssend
15	MPI_Isend	mw_init_Isend
16	MPI_Bsend	mw_init_Bsend
30	dummy Master-Worker measurement	mw_init_dummy

Table 3.3: The mapping of type numbers to measured MPI-functions (master-worker pattern)

Number	MPI-function(s)	Initializer
24	MPI_Wtime	simple_init_Wtime
25	MPI_Comm_rank	simple_init_Comm_rank
26	MPI_Comm_size	simple_init_Comm_size
27	MPI_Iprobe (not successful)	simple_init_Iprobe
28	MPI_attach	simple_init_attach
32	dummy simple measurement	simple_init_dummy

Table 3.4: The mapping of type numbers to measured MPI-functions (simple pattern)

Number	MPI-function(s)	Initializer
17,47	MPI_Bcast	col_init_Bcast
18,49	MPI_Barrier	col_init_Barrier
19,54	MPI_Reduce	col_init_Reduce
20,55	MPI_Alltoall	col_init_Alltoall
21,56	MPI_Scan	col_init_Scan
22,57	MPI_Comm_split	col_init_Comm_split
23,58	memcpy (ANSI-C)	col_init_memcpy
31,59	dummy collective measurement	col_init_dummy
33,48	MPI_Gather	col_init_Gather
35,60	MPI_Scatter	col_init_Scatter
36,61	MPI_Allreduce	col_init_Allreduce
37,62	MPI_Reduce followed by MPI_Bcast	col_init_Reduce_Bcast
38,63	MPI_Reduce_scatter	col_init_Reduce_scatter
39,64	MPI_Allgather	col_init_Allgather
40,65	MPI_Scatterv	col_init_Scatterv
41,66	MPI_Gatherv	col_init_Gatherv
42,67	MPI_Allgatherv	col_init_Allgatherv
43,68	MPI_Alltoallv	col_init_Alltoallv
44,69	MPI_Reduce followed by MPI_Scatterv	col_init_Reduce_Scatterv
45,70	Implementation of Gather with MPI_Send and MPI_Recv	col_init_Gather_Send_Recv
46,71	Implementation of Gather with MPI_Isend, MPI_Irecv, and MPI_Waitall	col_init_Gather_Isend_Waitall
72,73	Implementation of Broadcast with MPI_Send and MPI_Recv	col_init_Bcast_Send_Recv
74,75	Implementation of Alltoall with MPI_Isend and MPI_Irecv	col_init_Alltoall_Isend_Irecv
76,77	MPI_Comm_dup	col_init_Comm_dup
78,79	Implementation of Alltoallv with MPI_Isend and MPI_Irecv	col_init_Alltoallv_Isend_Irecv

Table 3.5: The mapping of type numbers to measured MPI-functions (collective patterns)

Now we take our C source code protection suit (don't forget the gloves) and dive into the source code (i.e. this huge file called skosfile.c). We have a look at the function `mw_init_Bsend`:

```
mw_init_Bsend (measurement_t *ms, data_t *data)
{
  ms->pattern = MASTER_WORKER;
  ms->data.mw_data.master_receive_ready = master_receive_ready_empty;
  ms->data.mw_data.master_dispatch = master_dispatch_Bsend;

  ms->server_init = mem_init_two_buffers_attach_mw;
  ms->server_free = mem_release_detach;
  ms->client_init = mem_init_two_buffers_attach_mw;
  ms->client_free = mem_release_detach;

  ms->data.mw_data.master_worker_stop = master_worker_stop_recv;
  ms->data.mw_data.worker_receive = worker_receive_test;
  ms->data.mw_data.worker_send = worker_send_test;
  ms->data.mw_data.communicator = MPI_COMM_WORLD;
  ms->data.mw_data.result = data;
  ms->data.mw_data.chunks = DEF_CHUNKS_VALUE;
  ms->data.mw_data.len = DEF_LEN_VALUE;
  ms->data.mw_data.def_nodes = DEF_NODES_VALUE;

}
```

Because we have already read the description of the master worker pattern on page 37 we know that the most important functions are `master_dispatch`, `worker_send` and `worker_receive` which are set to `master_dispatch_Bsend`, `worker_send_test` and `worker_receive_test` and which look like this:

```
int
master_dispatch_Bsend (int number_of_workers,
                       int work, int chunks, int len,
                       MPI_Comm communicator, MPI_Datatype data_type)
{
  MPI_Status
    status;

  MPI_Recv (_skampi_buffer, len, data_type, (work % number_of_workers) + 1,
            1, communicator, &status);
  /* sending next chunk of work to this worker */
  MPI_Bsend (_skampi_buffer, len, data_type,
             (work % number_of_workers) + 1,
            1, communicator);

  D(fprintf (stderr, "master: sending job_no %d to worker %d\n",
           work,(work % number_of_workers) + 1);)

  return (1);
}
```

```
void
worker_send_test (int len, MPI_Comm communicator, MPI_Datatype data_type)
{
  MPI_Ssend (_skampi_buffer, 0, data_type,
             0, 1, communicator);
}

int
worker_receive_test (int len, MPI_Comm communicator, MPI_Datatype data_type)
{
  MPI_Status status;

  MPI_Recv (_skampi_buffer, len, data_type, 0,
            MPI_ANY_TAG, communicator, &status);

  if (status.MPI_TAG == 0)  /* STOP working */
    return (FALSE);

  return (TRUE);
}
```

And hopefully we now know exactly what's going on.

3.1.2 Point-to-point pattern

The point-to-point pattern is an alternating sequence of `server_op` and `client_op` called on the server respectively client. By default the node with the largest latency is chosen as client, but it is also possible to select the one with the smallest latency with a switch in the *SKaMPI* parameter file.

```
/* server node */

max_node := determine node with maximum latency
min_node := determine node with minimum latency

repeat
  start_time := MPI_Wtime()
  server_op()
  end_time := MPI_Wtime()
until result exact enough

send stop signal


/* client node */

actions to answer the max/min_node determination

if client is max/min_node
```

```
  repeat
    client_op()
  until stop signal received
```

3.1.3 Master-worker pattern

The Master-worker pattern corresponds to the typical master-worker scheme: a master process divides a problem into several sub-problems (here called chunks) and dispatches them to several worker processes (`master_dispatch`). The workers initialize with `master_receive_ready` (perhaps a `MPI_Irecv` if necessary) and subsequently send results and receive new chunks of work with `worker_send` and `worker_receive`. When all work is done, the master sends a stop-signal to the workers (`master_worker_stop`). This scheme is important in practice, because it is the simplest method to do load balancing.

```
/* master code */

for each worker
  master_receive_ready /* set ready to receive e.g. MPI_Irecv */

chunk := 0
start_time := MPI_Wtime()

while chunk < all_chunks
  master_dispatch()
  chunk := chunk + 1

end_time := MPI_Wtime()

for each worker
  master_worker_stop() /* send stop signal */


/* worker code */

repeat
  worker_send() /* send ready signal to master */
  worker_receive() /* get new work or stop signal */
until stop signal received
```

3.1.4 Barrier measured collective pattern

This is the original collective pattern used in previous versions of *SKaMPI*. Though usually all collective MPI operations use the same function calls and parameters to perform a collective communication a distinction between the root or server and the nodes or clients is made (`routine_to_measure` and `client_routine`). This simplifies the use of self-made alternative implementations of these collective operations.
`finalize_server_routine` and `finalize_client_routine` are used for administrative things like the release of resources or waiting for the end of an asyn-

chronous send. The latter is mainly done because of some race conditions between these send operations and the following busy waiting operations performed in the new synchronous collective pattern described in the next section.

```
/* server code */

MPI_Barrier()
repeat
  start_time := MPI_Wtime()
  routine_to_measure()
  finalize_server_routine()
  MPI_Barrier()
  end_time := MPI_Wtime()
until result exact enough
send stop signal


/* client code */

MPI_Barrier()
repeat
  client_routine() /* counterpart of routine_to_measure */
  finalize_client_routine()
  MPI_Barrier()
until stop signal received
```

3.1.5 Synchronous measured collective pattern

The measurement method described in the last section lacks accuracy especially for fast operations with short message lengths. Therefore a much better measurement method was developed. It uses synchronized clocks and avoids any interference with the measured collective operation (details in ([1] or for a shorter description in English [7]). Simplified it looks like this (the functions implementing the collective operations are the same as in the previous pattern):

```
/* server code */

clock synchronization
repeat
  start synchronous with other nodes
  start_time := MPI_Wtime()
  routine_to_measure()
  end_time := MPI_Wtime()
  finalize_server_routine()
  wait till end of time slot
  collect results, maximum is the result of single measurement
until result exact enough
send stop signal

/* client code */
```

```
clock synchronization
repeat
  start synchronous with other nodes
  start_time := MPI_Wtime()
  client_routine() /* counterpart of routine_to_measure */
  end_time := MPI_Wtime()
  finalize_client_routine()
  wait till end of time slot
  send result to server
until stop signal received
```

Unfortunately this new measurement method is slower an takes about twice as much time as the old one. Therefore and for keeping old measurements comparable the new method has not completely replaced the old one, though it should be used whenever possible.

3.1.6 Simple pattern

Some routines of MPI seem to be so simple, that they are measured in a very simple "pattern". In this pattern we measure all operations with local effects.

```
/* root node */

repeat
  start_time := MPI_Wtime()
  routine_to_measure()
  end_time := MPI_Wtime()
until result exact enough
```

3.2 The output file

The output file is a pure ASCII-text file. Its name is usually `skampi.out` by default but can be changed in the `@OUTFILE`-section of the parameter file (see section 2.1.1 for further information). Roughly speaking it has three sections: the header, the data, and the trailer.

Header

The header stores all information characterizing the context of the measurements stored in this file. These are the sections `@MACHINE`, `@NODE`, `@NETWORK`, `@USER`, and `@ABSOLUTE` which are filled with data from from the parameter file. Additional sections are filled by the benchmark. A typical header can look like:

```
#@MACHINE  IBM RS/6000 SP
#@NODE  thin node P2SC 120 MHz
#@NETWORK  High Performance Switch TB3
#@USER  Ralf Reussner
#@SKAMPIVERSION 1.20
#@OSNAME AIX
```

```
#@OSRELEASE 2
#@OSVERSION 4
#@HOSTNAME p071
#@ARCHITECTURE 000089978100
#@ABSOLUTE yes
#@DATE Thu Oct 29 11:25:34 1998
```

Data

This section is a list of suites of measurements. Each suite starts with a "small" list-header, describing this suite, followed by a result-list For all patterns except the simple-pattern the header looks like:

```
#-------------------------------
#/*@incol_MPI_Bcast-nodes-short.ski*/
#Description of the MPI_Bcast-nodes-short measurement:
#Pattern: Collective varied over the number of nodes [number] (%%d).
#The x scale is linear, no automatic x wide adaption
#range: 2 - 64, stepwidth: 1.000000.
#default values: 64 nodes, message length 256 bytes, max. / act. time for suit
e disabled/0.31 min.
#max. allowed standard error is 3.00 %, cut quantile is 0.00 %
#Format: <args> number of nodes [number] (%%d) <results> time_cleaned [microse
c.] (%f) standard_error_cleaned [%] (%f) count_cleaned [number] (%d) time_all
[microsec.] (%f) standard_error_all [%] (%f) count_all [number] (%d)
```

A typical header of the simple-pattern looks like:

```
#/*@insimple_MPI_Wtime.ski*/
#Description of the MPI_Wtime measurement:
#Pattern: Simple.
#
#
#
#max. allowed standard error is 3.00 %
#Format: <args>  <results> time_cleaned [microsec.] (%f) standard_error_cleane
d [%] (%f) count_cleaned [number] (%d) time_all [microsec.] (%f) standard_erro
r_all [%] (%f) count_all [number] (%d)
```

Note that the `@in`-command is used by the report generator, to identify the measurements[1]. All other lines start with a ♯, so that `gnuplot` treats these lines as comments.
The small header for suites of the simple-pattern look different, because this pattern does not has information on scale, range and default values. (But both list-headers have the same length of eight lines.[2])

Note the following line giving the typing information of the result list (the result list is described in the next subsection).

[1] and to create temporary files.

[2] For implementors: This string is created in the function `measurement_data_to_string` in module `skampi_tools`.

```
#Format: <args> number_of_nodes [number] (%%d) <results> time_cleaned
[microsec.] (%f) standard_error_cleaned [%] (%f) count_cleaned
[number] (%d) time_all [microsec.] (%f)
standard_error_all [%] (%f) count_all [number] (%d)
```

These lines should be read as one continuous line. The basic idea is, that the formats of the result-lists may differ. So it is important to describe each list's format.
The format-line starts with "`#Format:`", followed by a tag (`<args>`), which means, that a description of arguments follows. (In case of multi dimensional measurements more than one argument belongs to one measurement.) Each argument is described with its *name* (in our example `number_of_nodes`) than its *unit* ([`number`]) and its `format` in C-Syntax given in round brackets (e.g., `(%d)`). Each so described argument corresponds to one column of the result-list. The arguments describing list is followed by another list, the results describing list. Each entry describes a column of the result list. An entry is formed by the following data (similar to an entry of the argument list): *name*, *unit*, and *format*.

After each list-header follows a *result-list* of measurements for each suite. (This list may contain only one element.)

```
2 176.059111 3.034745 8 176.059111 3.034745 8
3 386.971049 14.221803 8 386.971049 14.221803 8
4 370.513008 14.726381 8 370.513008 14.726381 8
5 573.763306 26.948681 11 573.763306 26.948681 11
6 521.403970 10.311949 8 521.403970 10.311949 8
7 577.031024 9.031125 8 577.031024 9.031125 8
8 484.304333 24.567614 11 484.304333 24.567614 11
9 706.000973 35.550781 68 706.000973 35.550781 68
10 701.232959 25.582020 8 701.232959 25.582020 8
11 802.918861 33.229652 8 802.918861 33.229652 8
12 806.794216 37.361757 11 806.794216 37.361757 11
13 766.557961 21.876852 8 766.557961 21.876852 8
14 818.220084 37.641216 9 818.220084 37.641216 9
15 827.972894 36.904118 9 827.972894 36.904118 9
16 758.197092 36.257975 14 758.197092 36.257975 14
#eol
```

To mark the end of this list, skampi prints an `#eol`.

Trailer

The trailer is just the last line of the output file. If skampi finishes correctly, the last line will contain the string "`skampi finished.`". If this file was created by post processing, there will be additionally the stamp: `-postprocessed`.

Bibliography

[1] W. Augustin. *Neue Messverfahren für kollektive Operationen*, Diplomarbeit, Universität Karlsruhe, Fakultät für Informatik, Dezember 2001.

[2] W. Gropp, E. Lusk. *User's Guide to mpich, a portable Implementation of MPI*, Technical Report ANL/MCS-TM-ANL-96/6, Argonne National Laboratories, 1996

[3] Message-Passing Interface Forum. *MPI-2.0: Extensions to the Message-Passing Interface*, MPI Forum, June 1997. `http://www.mpi-forum.org/docs/docs.html`.

[4] R. Reussner. *Portable Leistungsmessung des Message Passing Interfaces*, Diplomarbeit, Universität Karlsruhe, Fakultät für Informatik, 1997

[5] SKaMPI-Projekt. SKaMPI-Homepage. `http://liinwww.ira.uka.de/~skampi/`.

[6] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and L. Dongarra. *MPI – The Complete Reference.* 2nd. Ed., MIT Press, Cambridge, Massachusetts, 1998

[7] Th. Worsch, R. Reussner and W. Augustin *On Benchmarking Collective Operations*,

Recent advances in parallel virtual machine and message passing interface: 9th European PVM/MPI User's Group Meeting Linz, Austria, September/October 2002

Index