SKaLib: SKaMPI as a library

R. H. Reussner
University of Karlsruhe
Department of Informatics
Germany
reußner@ira.uka.de

June 10, 1999

\footnote{This document appeared as Interner Bericht (Technical Report) 99/07 at the Department of Informatics, University of Karlsruhe, Germany}
Abstract

SKaLib is a library to support the development of benchmarks. It offsprings from the SKaMPI-project [2]. SKaMPI is a benchmark to measure the performance of MPI-operations [6]. Many mechanisms and function of the SKaMPI-benchmark program are also useful when benchmarking other functions than MPI's. The goal of SKaLib is to offer the benchmarking mechanisms of SKaMPI to a broader range of applications. The mechanisms are: precision adjustable measurement of time, controlled standard error, automatic parameter refinement, and merging results of several benchmarking runs.

This document fulfills two purposes: on the one hand it should be a manual to use the library SKaLib and explains how to benchmark an operation. On the other hand this report complements the SKaMPI-user manual [4]. The latter report explains the configurations and the output of SKaMPI, whereas this reports gives a detailed description of the internal data structures and operations used in the SKaMPI-benchmark.

There is also a scientific section which motivates and describes the algorithms and underlying formulas used by SKaMPI.
Contents

1 Introduction 2

2 Using SKaLib 4

3 Mechanisms of SKaLib 5
  3.1 Portable measurement of time .......................... 5
  3.2 Automatic control of the standard error .............. 8
    3.2.1 Repetition of measurements ...................... 8
    3.2.2 Forming a measurement .......................... 10
    3.2.3 Interface of the ASEC module .................. 10
  3.3 Automatic parameter refinement ..................... 11
    3.3.1 Algorithm .................................. 11
    3.3.2 Estimation of the maximal error ................ 13
    3.3.3 Implementation and Interface .................. 14
  3.4 Automatic merging of results ....................... 14
    3.4.1 Merging results .............................. 14
    3.4.2 Finding results .............................. 15
    3.4.3 Putting it all together ....................... 15

4 Example: How SKaLib is used in SKaMPI 17
  4.1 The Patterns-layer .................................. 17
  4.2 Autodist-layer .................................... 19
  4.3 Post processing-layer .............................. 20

5 SKaMPI's Main data structures 21

6 Enhancements of SKaMPI 27
  6.1 New sections of the parameter file ................... 27
  6.2 New measurements .................................. 31
  6.3 New patterns ...................................... 34
# Index of all functions

## 7 Module skampi

7.1 Function main ................................................. 38

## 7.2 Module autodist

7.2.1 Function measure_suite ..................................... 39
7.2.2 Function calculate_key ....................................... 39

## 7.3 Module automeasure

7.3.1 Function am_init .............................................. 40

## 7.4 Module "standard_error..." given at the beginning of function

7.4.1 Function am_free ............................................. 40
7.4.2 am_control_end ............................................... 40
7.4.3 Function am_fill_data ....................................... 41
7.4.4 Function double_cmp ......................................... 41

## 7.5 Module col

7.5.1 Function col_pattern ......................................... 42

## 7.6 Module col_test1

7.6.1 Functions col_init, ........................................... 42
7.6.2 Functions measure, ......................................... 42

## 7.7 Module mw

7.7.1 Function mw_pattern ......................................... 43

## 7.8 Module mw_test1

7.8.1 Functions mw_init, .......................................... 43
7.8.2 Functions master_receive_ready_test ....................... 43

## 7.9 Module p2p

7.9.1 Function p2p_find_max_min .................................. 44
7.9.2 Function p2p_pattern ......................................... 44

## 7.10 Module p2p_test1

7.10.1 Functions p2p_init, ....................................... 45
7.10.2 Functions server, ....................................... 45
7.10.3 Functions client, ....................................... 45

## 7.11 Module simple

7.11.1 Function simple_pattern .................................... 46

## 7.12 Module simple_test1

7.12.1 Functions simple_init, .................................... 46
7.12.2 Functions measure, ....................................... 46

## 7.13 Module dataist

7.13.1 Function init_list .......................................... 47
7.13.2 Function add ............................................... 47
7.13.3 Function read_item ......................................... 47
7.13.4 Function read_item_1 ... ................................. 48
7.13.5 Function item_addr ......................................... 48
7.13.6 Function item_address_item ................................. 48
7.13.7 Function is_end ................................................. 49
7.13.8 Function is_start .............................................. 49
7.13.9 Function number_of_elements ................................. 49
7.13.10 Function remove_file ........................................... 49
7.13.11 Function free_data_list ....................................... 50
7.13.12 Function minimum ............................................. 50
7.13.13 Function maximum ............................................. 50
7.13.14 Function variance ............................................. 51
7.13.15 Function average_of_lists ................................. 51
7.13.16 Function average ............................................. 51
7.13.17 Function write_to_file ....................................... 51
7.13.18 Function read_from_file .................................... 52
7.14 Module slampi_error .............................................. 52
7.14.1 Function output_error .......................................... 52
7.15 Module slampi_mem ............................................... 52
7.15.1 Function allocate_mem ......................................... 53
7.15.2 Function free_mem .............................................. 53
7.15.3 Function mem_init_one_buffer .............................. 53
7.15.4 Function mem_init_two_buffers ............................ 54
7.15.5 Function mem_init_two_buffers_gather ..................... 54
7.15.6 Function mem_init_two_buffers_call .............................................. 54
7.15.7 Function mem_init_two_buffers_attach .................. 55
7.15.8 Function find_mml ............................................. 55
7.15.9 Function mem_init_two_buffers_attach_p2p .................. 56
7.15.10 Function mem_release_detach ............................ 56
7.15.11 Function mem_release_detach_nw ....................... 56
7.15.12 Function mem_init_nw_Waitsome ......................... 57
7.15.13 Function mem_init_nw_Waitany ......................... 57
7.15.14 Function mem_init_col_Waitall ......................... 58
7.15.15 Function mem_release ......................................... 58
7.16 Module slampi_params ........................................... 58
7.16.1 Function read_parameters ..................................... 58
7.16.2 Function init_params ......................................... 59
7.16.3 Function parse_parameter_file .......................... 59
7.16.4 Function line_mode ............................................ 59
7.16.5 Function send_text ............................................ 60
7.16.6 Function recv_text ............................................ 60
7.16.7 Function read_next_char ................................... 60
7.16.8 Function unread_next_char ................................ 61
7.16.9 Function init_symboltable ................................ 61
7.16.10 Function lookup
7.16.11 Function insert
7.16.12 Function lexan
7.16.13 Function match
7.16.14 Function parse
7.16.15 Function measurement
7.16.16 Function variation_style
7.16.17 Function scale_style
7.16.18 Function int_or_max
7.16.19 Function int_or_default
7.16.20 Function int_or_float
7.16.21 Function yes_or_no
7.16.22 Function float_or_default
7.16.23 Function float_or_default_or_invalid
7.16.24 Function initialize_type
7.16.25 Function token_to_str
7.17 Module slampi_post
7.17.1 Function post_processing
7.17.2 Function init_post_proc
7.17.3 Function free_post_proc
7.17.4 Function free_all_lists
7.17.5 Function skip_to_next_meas
7.17.6 Function find_meas
7.17.7 Function read_one_list_of meas
7.17.8 Function read_all_lists_of_next_meas
7.17.9 Function combine_lists
7.17.10 Function all_finished
7.17.11 Function interpolate_data
7.17.12 Function post_process
7.17.13 Function data_cmp
7.18 Module slampi_tools
7.18.1 Function write_to_log_file
7.18.2 Function measurement_data_to_string
7.18.3 Function read_header
7.18.4 Function write_header
7.18.5 Function write_text_to_file
7.18.6 Function insert_in_text
7.18.7 Function read_from_text
7.18.8 Function free_text
7.18.9 Function read_old_log_file
7.18.10 Function new_name
A Derivation of the standard error formula
Acknowledgments

This technical report mainly offsprings from my diploma thesis [3]. 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

Introduction

During the SKaMPI-project [2] we developed many methods for accurate, reliable and detailed MPI-benchmarking. Today the code of SKaMPI is reused in several other projects. This reuse demands for a more detailed description of the SKaMPI-internals than given in [3]. Furtheron, we think that SKaMPI code reuse can be supported by packaging SKaMPI’s functions in a library, with a clearly described interface. Hence, this report is both: a technical reference for SKaMPI and a description of "SKaMPI as a library (SKaLib)". This library consists of four parts: (a) precision adjustable measurement of time (section 3.1), (b) routines for automatic standard error control (section 3.2), (c) automatic parameter refinement (sec. 3.3), and (d) the mechanism to summarize the results of several runs to a single result file (sec. 3.4).

How these mechanisms are applied in SKaMPI is show as an example use of SKaLib in chapter 4. Main data structures used in the library you can find in chapter 5. Some "typical" extensions to SKaMPI such as new sections in the parameter file, new measurements, new patterns, etc. are described in chapter 6. Chapter 7 is an index of all functions used in SKaMPI. Here you can lookup a short description of each function.

A short glossary

Before starting, lets clear some expressions.2

Single measurement: Calls of a (MPI) routine to be measured in a pattern.
(E.g., MPI_Send-MPI_Recv at 1 MB message length.) The number of calls depends from the precision requested by the user (see section 3.1 for the calculation of precision).

---

1For goals of SKaMPI look at [4], for example.
2In difference to the definition given in [4] we here allow several calls to form a single measurement. In this way the precision can be adjusted.
Measurement: A measurement is the determination of a value at an exactly defined (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 requested, the time allowed, and an upper and lower bound.

Pattern: A frame, where similar measurements can be plugged in. In the parallel case patterns are useful for the coordination of a measurement’s processes.

Suite of measurements: Measurements varied over their common 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. (The selection is done in the parameter file.) Usually for each run a report is generated.
Chapter 2

Using SKaLib

Getting SKaLib

SKaLib can be downloaded from the SKaLib homepage http: wwwpd.ira.uka.de/skampi/skalib.html. It arrives as skalib.tar.gz.

Installing SKaLib

This file can be unpacked with the UNIX command gunzip -c skalib.tar.gz
(in case no gunzip is available on your system, use: gunzip -c skalib.tar.gz | tar -xf -) The directory skalib/ will be created relative to your current directory. It contains all source files and two examples: skalib-ex (the sequential example) and the SKaMPI benchmark, as an example of a benchmark using MPI, and skalib-ex.c

Compiling SKaLib

All steps required to build the SKaLib library file are performed by the makefiles skalib-mpi.mak and skalib-seq.mak. Probably some macros in the makefile have to be adapted to your system. Start the building process for the sequential library with the command make -f skalib-seq.mak in the skampi directory. Use make -f skalib-mpi.mak for the parallel version of the library. In this case you need MPI Ver. 1.0 or higher installed.

To use SKaLib, adapt one of this makefiles. (Depending on whether to use MPI or not.)

We resigned from creating a library with the ar command in the makefiles, since it is highly system dependent. In principle, it is possible first to build up a library with ar, and then to link it with your object file.
Chapter 3

Mechanisms of SKaLib

This chapter explains mechanisms of SKaLib, which are used in SKaMPI. One common goal of all mechanisms is to improve the reliability of data and to decrease the influence of disturbances of any kind. As the most basic mechanism, we will talk first about portable measurement of time (sec. 3.1). According to the terminology given at the beginning of this report, the automatic control of the standard error (sec. 3.2) combines single measurements to measurements. The automatic parameter refinement (sec. 3.3) is used to form suites of measurements out of measurements. To combine the results of several runs to one result file the mechanism automatic merging of results (sec. 3.4) is used.

3.1 Portable measurement of time

For a benchmarking program the measurement of time is crucial, of course. Portability of time measurement is hindered by the fact, that the resolution of the clock is system dependent. In the following we present a method measuring time with a user defined resolution on all systems, i.e., the resolution of the result is not system dependent.

MPI offers two valuable functions: MPI_Wtime for time measurement and MPI_Wtick which returns the resolution of the clock. When not using MPI (that is in the sequential case) a portable way to measure time is using the ANSI C function clock. This function returns a value of type clock_t.\(^1\) The following example illustrates the usage of clock and motivates the way SKaLib uses it.

```c
clock_t
    start_time,
    end_time;
```

\(^1\)Experience shows, that MPI_Wtime has a much better resolution than clock on most systems. So we prefer using MPI_Wtime when available.
double
    time;

start_time = clock();
/* measure something */
end_time = clock();
time = ((double)(end_time - start_time)/(double)CLOCKS_PER_SEC);

Of course we need to know the unit of time. Since the result of clock
is system dependent, we need to divide through the constant CLOCKS_PER_SEC
which is defined in time.h.2 Because we cannot assume that CLOCKS_PER_SEC is
defined as an integer value on all systems, we cast to double before the division.3
However, the main problem with the above “algorithm” is that we do not know
the resolution of clock, which not necessarily equals to CLOCKS_PER_SEC. This
may be a serious issue, because nobody guarantees, that the resolution is fine
enough the for our measurement (i.e., end_time - start_time may be zero).
Unlike MPI, ANSI C does not offer a portable function to determine the clock’s
resolution.

When not using MPI, the following algorithm is used to determine the res-
olution of clock (routine clock_resolution in module skampi_tools).

clock_t
    start_time,
    end_time;

    long int
        i;

    for (i = 0, start_time = clock(); ; ++i)
    {
        end_time = clock();
        if ((end_time - start_time) >= 1)
            break;
    }

    return (((double)(end_time - start_time)) /
        ((double)CLOCKS_PER_SEC));

(Interestingly enough, omitting the counting of the index variable i let some
compilers produce bad code (even without any optimization), always returning
zero.) However, so we find the smallest possible difference between two calls
of clock in “CLOCKS”. This difference divided through CLOCKS_PER_SEC is the
resolution $R_{stem}$ in seconds.

---

2In time.h also the constant CLock is defined, but its value seems useless on some systems.
3This is only done for safety because it is reasonable, that when CLOCKS_PER_SEC has no
   integral type, then clock should also be no integer. But this is not guaranteed.
3.1. PORTABLE MEASUREMENT OF TIME

In both cases, when using MPI or ANSI C, we would like to have an equal precision on all systems, that is a user definable precision instead of a precision defined by the system. To achieve a user definable resolution we have to repeat measurements.

clock_t
    start_time,
    end_time;
double
time;

long int
i;

for (i = 0, start_time = clock(); i < N; ++i)
{
    /* something to measure */
}
end_time = clock();
time = ((double)(end_time - start_time)) /
      ((double)CLOCKS_PER_SEC * N));

Since we know that the result of $N$ repeated measurements in the variable time has the resolution $R_{system}$, we know that the resolution of one measurement is $R_{system}/N$. This relies on the assumption that each of the $N$ measurements consumes the same time. This assumption is not always valid on multitasking systems. $N$ can be determined, when the user gives a wished resolution $R_{user}$.

$$N := \frac{R_{system}}{R_{user}}$$

(3.1)

$1/R_{user}$ is given in the constant `WISHED_RESOLUTION` in the file `skalib_const.h`

$N$ is stored in the global variable `repetitions` which is set in `init_skalib`.

One problem remains: The time consumed even by $N$ measurements needs not to be higher than $R_{system}$. So again we repeat our $N$ measurements until `end_time - start_time` is larger than zero.

clock_t
    start_time,
    end_time;
double
time;

long int
i,
a;
for (a = 0, start_time = clock(); end_time - start_time > 0; ++a)
3.2 Automatic control of the standard error

For each measurement, the number \( n \) of repetitions is determined individually to achieve the minimum effort required for the accuracy requested. This is achieved through the automatic control of the standard error (ASEC).

A single measurement consists of the data measured through one call of the routine to be measured with fixed parameters (e.g., `MPI_Send` with one MB message length). Data gained in this way contains both the systematic and the statistical error. Systematic error occurs due to the measurement overhead including the call of `MPI_Wtime`. It is usually small and can be corrected by subtracting the time for an empty measurement. Additionally, we warm-up the cache by a dummy call of the measurement routine before actually starting to measure.

Individual measurements are repeated in order to control three sources of statistical error: finite clock resolution, execution time fluctuations from various sources, and outliers.

Two questions arise: (1) how many repetitions are necessary? Since we do not want to waste expensive supercomputer time, we do not want to perform too many repetitions. And (b) how to combine data of these single measurements to a measurement's result?

The first question is handled by the routine `am_control_end`, the second by `am_fill_data`. All routines of this section can be found in module `automeasure.c`.

3.2.1 Repetition of measurements

In principle, not all suites of measurements are equally important for the user. Some suites are only used for a rough overview of a function's performance, whereas other suites are very important for tuning an MPI implementation, or to ponder which MPI operation to use. Therefore, the user can give a limiting standard error per suite (variable `standard_error` in the struct `measurement_struct`, see chapter 5) (In the skampi-parameter file this variable can be set through the parameter file.) Single measurements are repeated until the standard error of
the performed single measurements falls below the given limit. The standard error is a metric for the reliability of the data, whereas the standard deviation is a metric for the dispersion of the data. The standard error \( \sigma_x \) is defined as:

\[
\sigma_x = \frac{\sigma}{\sqrt{n}}
\]

(3.2)

where \( n \) is the number of single measurements, the \( x_i \) \((i = 1 \ldots n)\) are the single measurement’s results, \( \bar{x} \) is the mean of the \( x_i \), and \( \sigma \) is the standard deviation:

\[
\sigma = \sqrt{\frac{\sum_{i=1}^{n} (\bar{x} - x_i)^2}{n - 1}}
\]

(3.3)

The above definition for the standard error (3.2) is not used in am.control.end. In am.control.end the standard error is calculated on the fly (i.e., after each single measurement with updates of the variables counter \((= n)\), result_sum_all \(= \sum_{i=1}^{n} x_i\), square_result_sum_all \(= \sum_{i=1}^{n} x_i^2\), and mean_value_all \(= \bar{x}\) using formula (3.4).

\[
\sigma_x = \sqrt{\frac{\sum_{i=1}^{n} x_i^2 - \frac{(\sum_{i=1}^{n} x_i)^2}{n}}{n \cdot (n - 1)}}
\]

(3.4)

\( \sigma_x \) is used as an estimator for the standard error of the mean.\(^4\) Here we assume that the error in the \( x_i \) has a Gaussian distribution [5].

To see, why formula (3.4) is equal to the definition (3.2), appendix A presents a short derivation.

Additionally to the standard error limit, the user can enter a time limit \time\ meas in measurement struct. This time limit guarantees that no new measurement is started, when the time limit is exceeded (even when the standard error is higher than the standard error limit). Note that no running single measurement is aborted, so possibly a measurement may take a little bit more time than the given time limit.

As a third factor to control the number of single measurements a range can be given through \max\ rep and \min\ rep. \max\ rep is used to allocate buffer for the single measurement’s results (in am.init). So never more than \max\ rep single measurements are performed. So \max\ rep overrides all other variables. Opposed to that \min\ rep does not. There can be less than \min\ rep single measurements, in case the time limit is exceeded. This is done, because, when the user gives a time limit, probably the time limit for SKaMPI when started on a parallel machine relies on the \time\ meas. (In the case no time limit given, there are \min\ rep single measurements, even when the standard error is below

\(^4\)As explained in the next section, we use the mean \( \bar{x} \) to form a measurement’s result out of the \( x_i \).
the standard error limit at less than min,rep single measurements.}

3.2.2 Forming a measurement

Assume we have an array\(^5\) with at most max,rep results of single measurements. cut,quartile in the measurement\(\_\)struct struct defines which results are used to form the result of the measurement. cut,quartile gives the size of the upper resp. lower quartile of single measurements which are disregarded. E.g., cut,quartile=0.25 than the upper quarter and the lower quarter of the results are ignored. (So we only have the middle 50 % of values left. In the example illustrated in figure 3.1 the shaded values are disregarded.)

3.2.3 Interface of the ASEC module

ASEC is implemented in the module automeasure. The interface of the ASEC mechanism consists of the functions:

am_init initializes all data structures of the ASEC module and allocates memory to store results of size many single measurements. It returns TRUE in case of success, FALSE in case of no memory.

am_control_end controls whether the measurement ms at argument arg should be repeated (returning TRUE) or not (returning FALSE). The parameters are:

the current measurement ms, the actual argument arg, the measured time (tbit, time), the node time\(^6\) node_time, the partner (-process) also involved into this measurement. If there is no other process involved, partner is set to NO,COMMUNICATION. If it is set to USE,COMMUNICATOR, the argument local,communicator will be used for communication to more than one other process. The root is process 0. It assumes that am_init has been called before.

In the sequential case the parameters partner, node_time, and local,communicator are omitted.

\(^5\)tbit,buffer in the source code.

\(^6\)The node time is the time measured on a node ad is measured on every node, whereas the tbit, time (to be measured time) is the time measured on the root process (process 0).
3.3. **AUTOMATIC PARAMETER REFINEMENT**

**am_fill_data** fills the **dummy time**, the standard error of the dummy time (\texttt{double}) into the data record **data**, therefore it uses some information about the actual suite of measurements **ms**. It assumes that **am_init** has been called before.

**am_free** frees all allocated resources, assumes that **am_init** has been called before.

All functions require **skalib_init** called before.

### 3.3 Automatic parameter refinement

The automatic parameter refinement (APR) feature is motivated by the observation that graphs of suites of measurements (time versus varied arguments; its performance graph) are often non-continuous; when the underlying implementation of the routine to be measured switches the algorithm, the performance graph has a saltus. Of course we are interested to determine these points exactly. On the other hand, the performance graph is not smooth at many arguments due to several reasons: limited accuracy, disturbed results, etc. Since we cannot avoid these facts, we are not interested in investing a lot of time to measure this noise more exactly than necessary.

So to build a suite of measurements we have to know at which arguments we should call a measurement. The arguments should be chosen to determine salti with an high accuracy, but since computing time is expensive we do not want to invest a lot of time in not “interesting” areas of the performance graph.

#### 3.3.1 Algorithm

Here we present an algorithm fulfilling the above requirement. The description mainly is cited from [2]. Let’s assume that we measure function \( t: P \to R \), i.e., taking a parameter \( m \in P \) and mapping it to a result in the real numbers \( R \). Furtheron, we assume that \( P \) are the integers from \( m_{\min} \cdots m_{\max} \). Furtheron, \( \sigma > 1 \) is the stepwidth of the **measurements** struct.

When using a logarithmic scale, we measure at \( m_{\max} \) and at \( m_{\min} \sigma^k \) for all \( k \) such that \( m_{\min} \sigma^k < m_{\max} \). On a logarithmic scale these values are equidistant. (What also is the case on a linear scale.)

Now the idea is to adaptively subdivide those segments where a linear interpolation would be most inaccurate. Since nonlinear behavior of \( t(m) \) between two measurements can be overlooked, the initial stepwidth \( \sigma \) should not be too large (\( \sigma = \sqrt{2} \) or \( \sigma = 2 \) are typical values). Fig. 3.2 shows a line segment between measured points \( (m_b, t_b) \) and \( (m_c, t_c) \) and its two surrounding segments. Either of the surrounding segments can be extrapolated to “predict” the opposite point of the middle segment.
Let $\Delta_1$ and $\Delta_2$ denote the prediction errors. We use

$$K_{m_\alpha, m_\beta} := \min(\|\Delta_1\| / t_b, \|\Delta_2\| / t_c, (m_c - m_b) / m_b)$$

(3.5)

as an estimate for the error incurred by not subdividing the middle segment.\footnote{We also considered using the maximum of $|\Delta_1|/t_b$ and $|\Delta_2|/t_c$, but this leads to many superficial measurements near jumps or sharp bends which occur due to changes of communication mechanisms for different message lengths.}

We keep all segments in a priority queue. If $m_b$ and $m_c$ are the abscissae of the segment with largest error, we subdivide it at $\sqrt{m_b m_c}$. We stop when the maximum error drops below $\epsilon$ or a bound on the number of measurements is exceeded. In the latter case, the priority queue will ensure that the maximum error is minimized given the available computational resources.

To see, why this scheme works, let’s assume two cases: The algorithm decides to start a measurement between the points $(m_b, t_b)$ and $(m_c, t_c)$. Now, in the first case, assume that the result lies on the line between $(m_b, t_b)$ and $(m_c, t_c)$ as shown in figure 3.3. That is, the point lies exactly where we would have assumed it without refinement. When calculating the $\Delta_1$ and $\Delta_2$ for the new segments, the $\min(\Delta_1, \Delta_2) = 0$ (figure 3.3). Hence, no further refinement would be done at this area.

In the other case, the algorithm also decides to start a measurement between the points $(m_b, t_b)$ and $(m_c, t_c)$. But now, opposed to the first case, assume that the result lies somewhere on the line $\alpha$ or somewhere on line $\beta$. Then the situation is, in principle, again the same as shown in figure 3.2. So, further refinement takes place, the point of the salus is determined with higher accuracy. This happens until the precision, given in $x_{\text{min.dist}}$, is reached.

The APR can be switched of ($x_{\text{scale}} = \text{FIXED.\!LIN}$ or $\text{FIXED.\!LOG}$).

Note that the APR works with fixed ($\text{DY\!N.\!LIN}$) and logarithmic ($\text{DY\!N.\!LOG}$)
3.3. Automatic Parameter Refinement

The analysis of this algorithm shows, that the cost of determining one of the salts $x_j$ is $\log Q \sigma_{\text{meas}} (\text{in the worst case}).$ Altogether with the $x_{\text{meas}}$ measurements of phase one we need $m(\sigma) = \frac{x_{\text{meas}}}{\sigma} + \log \frac{\sigma_{\text{meas}}}{\sigma} \cdot s$ measurements. In case we know the number of salts $s$ (through theoretical analysis of the function $t$ to measure or through a prior run of measurements), we can adjust $\sigma$ to yield a minimum of $m.$ Therefore we set $\frac{dm}{d\sigma} = 0,$ and get $\sigma = \frac{(x_{\text{meas}} - \sigma_0)\ln 2}{s}.$

3.3.2 Estimation of the maximal error

The error (i.e., the difference between $t$ and the reconstruction through measurements) can be bounded through $x_{\text{min}} \cdot m_i,$ where $m_i$ is the slope of the considered linear “piece” of $t.$ This is because we chose the stepwidth $\sigma$ larger than $M,$ the minimum length of a linear “piece” of $t.$

A different approach to parameter refinement calculates as a key $K_{(i,i+1)}$ the second derivative of $t$ at the points $x_i.$ Here the normalized discrete numerical approximation

$$K_{m_a,m_i} := \frac{1}{t_b} \frac{\frac{1}{m_a} - \frac{1}{m_i}}{m_a - m_i} \frac{t_b}{m_a + m_i}$$

is used. Note that

$$K_{m_a,m_i} = \frac{2}{t_b} \frac{(m_c - m_b)(t_b - t_a) - (m_b - m_a)(t_c - t_b)}{(m_b - m_a)(m_b - m_i)(m_c - m_a)}$$

Let $\ell_i := \max((m_c - m_b),(m_b - m_a),(m_c - m_a))$ during refinement step $i.$ During refinement $\ell_i$ approaches $x_{\text{min}}$ and $K_{m_a,m_i}$ increases like a function from $O(1/\ell^2).$ That means that the key increases despite further refinements.
(and so lower errors). So the key does not correspond to the errors of a segment. As a consequence, a particular area is refined albeit other areas of $t$ may contain higher errors. This does not happen with the key function defined in equation (3.5), because this key is independent from the segment's length. So, as a advantage, our algorithms lowers the errors at all parts of $t$ equally.

3.3.3 Implementation and Interface

The automatic parameter refinement is implemented in function `measuresuite` in module `autodist.c`. It computes for the suite of measurements $m$ all arguments where to measure, and performs the measurements. The measurements are called through the routine `tba` (to be measured), which is passed as a function pointer. Since the APR mechanism is independent of the pattern used, `tba` decides which pattern to use, not `measure_suite`. For each possible parameter to vary over (message length, nodes and chunks in the parallel case) we have one extra routine, which can be used for the `tba` parameter: `call_length`, `call_nodes`, `call_chunks` in the parallel case, and `call_length` also for the sequential case. All these "call routines" are in module `skampiface`.

3.4 Automatic merging of results

The exist situations, where a single run’s results are not reliable (mainly because of disturbance of the network or processors by other processes). Of course, it is best to repeat the run, when the source of disturbance disappeared. But that is not always possible. In this case the results of several disturbed runs can form a more reliable result file. The questions is: How to merge several result files into one?

Since merging result files heavily depends on the format of the output file, here the automatic merging of results (AMR) mechanism used by SKaMPI is described. For other output file formats the mechanism is still applicable, but the format specific parsing of output files has to be adapted to the other format.

3.4.1 Merging results

SKaMPI merges a measurement’s result depending on the kind of result: For the result of the `times` it is the weighted median. For example we merge three output files: the measurement of a suite of measurements at a fixed argument has in the first result file the result 899 measured with 10 single measurements, the second file has 901 (4 single measurements), and the last file has 910 with 4 single measurements. Then the result is 899. The `standard error` of this measurement is used for the `standard error` of the merged result. The same holds for the `node times`. This operation is performed by the routine `post_process`
3.4. AUTOMATIC MERGING OF RESULTS

in the module skampi.post.c, which takes an array of result data (given in parameter data_t *result_data) and gives back the filled structure new_data. Note that the global variable nif (set in init.post.proc) contains the number of input files.

3.4.2 Finding results

As described in section 3.3 the APR mechanism determines the arguments of some measurements of a suite of measurements. So when merging suites of measurements from several files, the problem can arise, that not every measurement of one file has its pendant in other files with the same argument. (E.g., the measurement at argument 1020 bytes of suite MPI_Send-MPI_Recv of the first file cannot find a measurement at this argument 1020 bytes of suite MPI_Send-MPI_Recv in the second file.

This problem is solved through interpolating the missing measurement through its neighboring measurements. In the above example, the measurement at argument 1020 bytes will be “created” through linear interpolation between the neighboring measurements at argument 1008 bytes and 1024 bytes.\(^8\) This interpolation is done in function interpolate_data called by function combine_lists.

3.4.3 Putting it all together

The post processing is performed in function post_processing, which gets the name of the first input file as argument.\(^9\) (It assumes, that the following input files are named as input_file.1, input_file.2, ...).

In pseudocode the way post_processing works looks like

while not END-OF-FILE first_file
    read a suite of measurements
    for all other file
        search this suite of measurements; read it
        for all measurements in the first suite
            if measurement at this argument is not available in another file
                interpolate this value;
                find median;
                store it in skampi.out

Functions depending on the input files' format are: find_meas (finding a certain measurement), skip_to_next_meas (skipping to the next measurement), and read_one_meas assumes text files as input. Generally the interface

\(^8\)For this example we will assume these are the neighboring values. Of course, in general there may be others.

\(^9\)Note that the output files of a benchmark are the input files for the function post_processing.
to input files is provided through routines as `read_from_file`, `write_to_file`, `read_header`, and `write_head_of_outfile` (which are all found in `skampi_tools`).
Chapter 4

Example: How SKaLib is used in SKaMPI

This chapter shows how to apply SKaLib it two example applications. One example application is the SKaMPI benchmarks, so this chapter can also be used as design document for SKaMPI. The other example is skalib-ex which shows the application of SKaLib for a sequential benchmark.

SKaMPI and skalib-ex use the AMR, APR, and ASEC mechanisms on different levels (or layers). The most internal layer is a pattern where ASEC is used; the APR is used at the autodist-layer. Finally, AMR is used at the post processing level. These layers are described below.

In principle SKaMPI and skalib-ex differ most in their patterns and their measurement of time. Their functions main are similar. Look at skalib-ex.c to see a typical function main and the declared variables and the included files. In the file skalib_const.h are some constants, which may be adapted by the user. (Note that in general recompilation of the library is required that these constants have affect.) Some constants are described in this document, refer to the index entry constants.

4.1 The Patterns-layer

In the mean of SKaMPI a pattern is a procedures which has two responsibilities: First, it has to execute the routine to be measured. This may be a simple function call or, in the parallel case, a SPMD\(^1\) fragment, which coordinates several processes to perform the routine to be measured. Second, a pattern has

\(^1\)Single Program Multiple Data: A program, run in several instances simultaneously on a parallel computer, which can perform different branches of its control flow in dependence of its process number.
to measure to routine to be measured. The advantage to separate the routine
to be measured from the measuring routine is an eased enhance-ability. A new
routine to be measured is just a new simple callback; we do not have to worry
about the measurement mechanism again. Furtheron, also the separation of
the coordination of the routine to be measured from the routine to be measured
itself makes sense. Since many routine are coordinated in the same way (e.g., all
master worker routines, or collective operations), we can reuse the coordination
mechanism several times. This also guarantees the comparability of the routines
measured with a specific pattern.

In terms of SKaLib, the result of a pattern's call is a measurement. To
achieve this, all patterns in SKaMPI use the ASEC mechanism. For examples
of pattern, we have a look to the four patterns used in SKaMPI.

**The Point to point pattern** coordinates and measures all MPI routines per-
forming point to point communication (e.g., `MPI_Send-MPI_Recv`).

**The Master worker pattern** consist of a master process (dispatching “work”
to several worker processes. The process of dispatching is measured. Here
MPI functions like `MPI_Waitsome` are measured.

**The Collective pattern** handles collective MPI operations like `MPI_Bcast`.
Here we apply a certain scheme to measure collective operations without
assuming synchronous clocks on each process. See [4] for details.

**The Simple pattern** measures the sequential operations without varying pa-
rameters (e.g., `MPI_Comm_split`).

In principle, each pattern has the following structure:

```c
if (CACHE_WARMUP > 0)
{
    /* cache warmups, not measured */
    am_init(CACHE_WARMUP);
    do
    {
        org_time = (start_time = MPI_Wtime()) - end_time;
        /* measure */
        ms->data.p2p_data.server_op (ms->data.p2p_data.len,
            ms->data.p2p_data.max_node,
            ms->data.p2p_data.communicator);
        end_time = MPI_Wtime();
        tmb_time = end_time - start_time;
    }while (am_control_end(ms, (-1) * CACHE_WARMUP, tmb_time, org_time,
            ms->data.p2p_data.max_node, tmb_time,
            MPI_COMM_NULL));
    am_free();
}
```
4.2 AUTODIST-LAYER

The above part warms up the cache. But also here you can see the usual pattern: First a call of am_init initializes all internal data structures of ASEC. (am_init gets the argument CACHE_WARMUP, i.e., the number of repetitions.) Then in a while-loop the routine to be measured is called as a callback. (In this example we use the p2p-callback ms->data.p2p.data_server_op.) Note that in this loop the time is measured. The loop is controlled by am_control_end (with the argument -1 * CACHE_WARMUP noting, that no results are stored.) am_free frees all internal data structures of ASEC.

Now we measure for real: like above, we perform several single measurements (the number is controlled by am_control_end and not fixed here) and form a measurement with am_fill_data. Everything else is as described above.

am_init(ms->max_rep);
do{
  org_time = (start_time = MPI_Wtime()) - end_time;
  /* measure */
  ms->data.p2p.data_server_op (ms->data.p2p.data.len, 
                              ms->data.p2p.data.max_node, 
                              ms->data.p2p.data.communicator);
  end_time = MPI_Wtime();
  tcm.time = end_time - start_time;
}while (am_control_end(ms, ms->data.p2p.data.len, tcm.time, org_time, 
                   ms->data.p2p.data.max_node, tcm.time, MPI_COMM_NULL));

am_fill_data (ms, ms->data.p2p.data.len, ms->data.p2p.data.dummy_time, 
              ms->data.p2p.data.dummy_time_se, ms->data.p2p.data.result);
am_free();

In this example we used the point-to-point pattern, but all other patterns have the same scheme of am_init, am_control_end, am_fill_data, and am_free.

4.2 Autodist-layer

This layer uses the APR mechanism. Remember from section 3.3 that the routine measure_suite calls a callback tcm with the calculated argument. In principle, tcm could be a pattern. In fact, tcm is a routine, which depends on the parameter varied over. In SKaMPI this routine can be one of the following: call_length, call_nodes, or call_chunks. These functions form an “intermediate” layer and call the appropriate pattern. This intermediate layer does some work for initialization depending on the pattern and parameter varied over. For sake of flexibility this work has been factored out of measure_suite ad

\footnote{Note that this callback is not the callback, called by a pattern.}
the patterns. This intermediate layer also initializes the dummy values. In this variables the dummy time is stored, i.e., the time of a measurement, induced by the overhead.

4.3 Post processing-layer

The idea of the post processing is to minimize the influences of the operating environment to one run of SKaMPI. So the post processing deals with the results of several runs of SKaMPI. This is the main reason why the post processing is separated into the extra program post.c. This seems reasonable also for other benchmarks basing on SkLib.

The other solution is to put the post processing in the benchmarks itself. We also realized this for sake of user’s convenience.

The post processing is called via the routine post-processing, its parameter input file name stores the filename of the base output name. (This is the name of the first output file; in SKaMPI its skmpi.out. post processing expects output files of older runs to be renamed to <basename>.1, <basename>.2, ... .
Chapter 5

SKaMPI’s Main data structures

The main data structures are declared in skmpi.h. Here you can find the measurement struct, which is the central of the whole benchmark. The values, which have to be initialized when calling a measurement (via its pattern), are marked with “IN”, values reached out with “OUT”. The data stored in measurement is necessary for every suite of measurements, except measurements with the simple-pattern. Since this pattern has no variation, the variables $x_{\text{start}}$, $x_{\text{end}}$, and $x_{\text{stepwidth}}$ have no sense.

First lets have a look at the variables of the measurement structure, which describe this suite of measurement.

typedef struct
{
    char *name;    /* name of this measurement IN */
    int pattern;   /* which pattern should be applied IN */
    #ifndef SEQ
    MPI_Comm communicator;
    #endif /*SEQ*/
}

Each suite of measurements has a unique name (in SKaMPI this name is defined in the parameter file [4]). This name is stored in name. In our context a pattern is a unique form, how several processes work together. (SKaMPI is developed to benchmark parallel programs, which means several processes may have to cooperate to perform a measured operation.) Technically spoken, pattern determines which function is called to measure an operation. If you measure MPI operations, you need a communicator, which defines the participating processes of a measurement.

The following variables describe parameters of the suite of measurements,
ie, which parameter to vary over (variation), parameter range (x_start and x_end), and some more, described more detailed below.

    int variation; /* NODES, LENGTH, CHUNK */
    int x_scale;  /* FIXED_LIN, FIXED_LOG, DYN_LIN, DYN_LOG */
    int x_start; /* lowest argument, start of the variation */
    int x_end;  /* max. argument, never succeeded by variation */
    double x_stepwidth; /* semantic:
     FIXED_LIN: x stepwidth between to measurements
       all other x_scales: first stepwidth */
    int x_max_steps;
    int x_min_dist; /* semantic:
       FIXED__: no meaning
       DYN_LIN: smallest stepwidth
       DYN_LOG: smallest stepwidth of the first two steps */
    int x_max_dist; /* semantic:
       FIXED__: no meaning
       DYN__: highest stepwidth */

    double
      time_suite, /* max. allowed time for a suite of measurements in minutes IN */
      act_time_suite; /* actual used time for one suite in minutes OUT */

    int multiple_of; /* every argument is a mutliple of this value (or 0) IN */

The x_scale determines, whether the arguments are chosen with constant distance (x_stepwidth) in the parameter range (...LIN), or logarithmic, which means, that measurements are performed a arguments (stepwidth¹, stepwidth², stepwidth³ ... until x_end has been reached (...LOG). The parameter x_scale is also used to switch on the automatic parameter refinement (refer to sec. 3.3). DYN_... as a value's prefix turn automatic parameter refinement of; FIXED_ off. When automatic parameter refinement is used, x_min_dist is the smallest distance between two arguments.¹ x_max_steps gives the maximum number of measurements in this suite of measurements. Note that when not using the automatic parameter refinement, the number of measurements is determined through the range of the argument and the stepwidth. (So the variable is only in use, when automatic parameter refinement is switched on, and the time limit time_suite is set appropriate.) act_time_suite gives the time actually used by this suite in seconds. multiple_of defines the integer every argument has to be a multiple of.

The following variables describe the measurements of this suite of measurements.

¹x_max_dist is not used until now.
int max_rep; /* max. number of calls of measurements in a pattern IN */
int min_rep; /* min. number of calls of measurements in a pattern IN */

int node_times; /* true iff execution times per node should be stored */

double standard_error, /* the max. allowed standard error: used to determine */
/* the end of measurements at one arg */
/* time_meas can overrides standard_error, in case that time_meas */
exceeded but the standard error of the measurement has not been
fallen below "standard_error"

time_suite can override x_max_steps and x_end, in case of time
suite exceeded, and not all measurements have been done.
*/

time_meas, /* max. allowed time for a measurement in minutes IN */
cut_quantile; /* quantile to cut of the results of a */
/* single measurements IN */

max_rep and min_rep define the range, how often the single measurements
of a measurement are repeated. (Note that the actual number of repetition
is defined through the given standard error and time limit time_meas, refer to
section 3.2) node_times is a boolean, in case of FALSE the time is only measured
by a master process. But possibly a parallel routine may have finished on other
processes while still running on the master process. To measure this effect, this
variable can be set on true. cut_quantile specifies the single measurement's
results, which a used to compute the measurement's result (refer to section 3.2).

The variable result_list

data_list_t *result_list; /* list of results */

/* default values */
int nodes;

/* routines for allocating and freeing resources:*/
memsize = the size in bytes (!) of the memory declared in params.memory
nor = number of repetitions (usually max_rep)
nom = number of measurements (usually x_max_steps)
nop = number of processes ind this communicator */

long (*server_init) (int nor, int nom, int nop);
void (*server_free) (void);
ifndef SEQ
long (*client_init) (int nor, int nom, int nop);}
void (*client_free) (void);
#endif /*SEQ*/

The pattern specific data structures (data) store the information, need for one measurement by an specific pattern.

All four pattern-specific data structures contain callback functions (here implemented with function pointers). These callbacks hold the functions to be measured. The meaning of the different callbacks is explained in the user manual, section “But what is measured?”.

union /* patternspecific data_structures IN */
{
    #ifndef SEQ
    p2p_pattern_data_t p2p_data;
    mw_pattern_data_t mw_data;
    col_pattern_data_t col_data;
    simple_pattern_data_t simple_data;
    #else
    simple_pattern_data_t simple_data;
    seqmeas_pattern_data_t seqmeas_data;
    #endif /*SEQ*/
}data;
measurement_t;

The specific data for the p2p-pattern is stored in the p2p_pattern_data_t struct.

/* bundle of data reached in the p2p_pattern */
typedef struct
{
    /* Pointer to function measured by server */
    MPI_Status (*server_op) (int, int, MPI_Comm);

    /* Pointer to client function */
    MPI_Status (*client_op) (int, int, MPI_Comm);
    /* second int is just dummy, so that client_op has the same type as server_op */

    int which_to_measure; /* which node should be used for measurement? the one with the max. latency or the one with the min. */

    /* both _node variables are filled in the routine p2p_find_max_min of module p2p.c */
    int max_node; /* number of the node with max. latency */
    int min_node; /* ... with min. latency */

    int len; /* the actual message length IN */
int def_nodes; /* the number of nodes used for this measurement IN */
MPI_Comm communicator; /* Communicator used for measurement IN */
data_t *result; /* Measured results OUT */
double dummy_time; /* dummy time for that communicator and pattern */
double dummy_time_se; /* and its standard error */
}p2p_pattern_data_t;

The last three items are common to each pattern-data-struct. They are not stored in the measurements_t-struct, because they are specific for one measurement and not for the suite of measurements.²

The master-worker data looks like:

/∗ bundle of data reached in the mw_pattern ∗/
typedef struct
{
  void (* master_receive_ready) (int, int len, MPI_Comm);
  int (* master_dispatch) (int now, int work, int chunks,
                           int len, MPI_Comm);
  void (* master_worker_stop) (int worker, int len, MPI_Comm);
  int (* worker_receive) (int len, MPI_Comm);
  void (* worker_send) (int len, MPI_Comm);
}MPI_Comm communicator; /* Communicator used for measurement IN */

int len; /* message length IN */
int def_nodes; /* the number of nodes used for this measurement IN */
int chunks; /* Number of work pieces IN */
data_t *result; /* Measured results OUT */
double dummy_time; /* dummy time for that communicator and pattern */
double dummy_time_se; /* and its standard error */
}mw_pattern_data_t;

The meaning of the callbacks is explained in the user manual, section “But what is measured?”.
The data for the collective-pattern:

typedef struct
{
  /* preparations for the routine_to_measure, only at the server site.
   * this function is not measured */
  void (* init_routine_to_measure) (int len,MPI_Comm);

  /* this function is measured */
  void (* routine_to_measure) (int len,MPI_Comm);

²According to the definitions given at the beginning of this report, a suite of measurements is a number of similar measurements varied over a parameter.
/* preparations for the routine_to_measure on the client site. 
   this function is not measured */
void (* init_client_routine) (int len,MPI_Comm);

void (* client_routine) (int len,MPI_Comm);
int len;        /* message length */
int def_nodes;  /* the number of nodes used for this measurement IN */
MPI_Comm communicator;   /* Communicator used for measurement */
data_t *result;       /* Measured results */
double dummy_time;    /* dummy time for that communicator an pattern */
double dummy_time_se; /* and its standard error */
}col_pattern_data_t;

In praxis the routine_to_measure and the client_routine point to the
same function. But to increase flexibility, we left two different function-pointers.
The “simple” data:

typedef struct
{
    void (* routine_to_measure) (void);
data_t *result;       /* Measured results OUT */
double dummy_time;    /* dummy time for that communicator an pattern IN */
double dummy_time_se; /* and its standard error IN */
}simple_pattern_data_t;
Chapter 6

Enhancements of SKaMPI

The following sections give hints for some enhancements. All these extensions require a new compilation of SKaMPI. This can be done in two ways. First you can use the makefile given with the mpich implementation [1] of MPI for application programming. (This is the way I used.) So you can use the different modules, which may ease understandability.

The more portable (but also more time consuming) way is to create one source file from all modules and compile this one. This is just one compiler call, and you do not have to worry about some dependencies, because in every call the whole code of SKaMPI is compiled. This is the “SKaMPI in one sourcefile” mechanism (skosfile).

Figure 6.1 shows the steps to yield SKaMPI in one source file, which can be compiled to SKaMPI. After calling rsplit.pl *.ch, several source files will be created in the subdirectory onesourcefile. Then change in this directory and call the shell-script sk2if¹, which creates skosfile.c. This file can be compiled with your local MPI-C-Compiler. (Note that you have to link with the math library (-lm)).

6.1 New sections of the parameter file

For demonstration how to add a new section, will look to all steps including the @NEWSECTION-section. It should be a section containing text.

1. Add a new mode name in file skampi_params.h. We will name it NEWSECTION_MODE.

   The new steps enum may look like this:

   enum NO_MODE, USER_MODE, MEM_MODE, STEPS_MODE, NETWORK_MODE, NODE_MODE,
   MACHINE_MODE, COMMENT_MODE, OUTFILE_MODE, LOGFILE_MODE, MAX_REP_MODE,

¹sk2if means “SKaMPI to one file”.

27
MIN_REP_MODE, STANDARD_ERROR_MODE, POST_MODE, MEASUREMENTS_MODE, ABSOLUTE_MODE, TIME_MEAS_MODE, TIME_SUITE_MODE, CUT_QUANTILE_MODE, MULTIPLE_OF_MODE, MY_NEW_SECTION);

2. Add the new parameter in the params_t-struct in skampi.params.h. Here we use that it will be a text-section.

typedef struct
{
    text_t user;
    text_t out_file;
    text_t log_file;
    text_t machine;
    text_t network;
    text_t node;
    unsigned memory;
    unsigned max_steps_default;
    unsigned max_rep_default;
    unsigned min_rep_default;
    unsigned multiple_of_default;
}
6.1. NEW SECTIONS OF THE PARAMETER FILE

```c
double standard_error_default;
double time_meas_default;
double time_suite_default;
double cut_quantile_default;
int absolute;
int post_proc;
text_t comment;
text_t measurements;
text_t my_new_section;
}params_t;
```

3. Add in function init_params in module skampi_params.c a line assigning a default value the new parameter.

```c
params_t *
init_params (params_t *params)
{
    params->user[0] = NULL;
    params->out_file[0] = OUTFILE;
    params->out_file[1] = NULL;
    params->log_file[0] = LOGFILE;
    params->log_file[1] = NULL;
    params->machine[0] = NULL;
    params->network[0] = NULL;
    params->node[0] = NULL;
    params->memory = MEM_DEFAULT;
    params->max_steps_default = MAX_STEPS_DEFAULT;
    params->max_rep_default = MAX_REP_DEFAULT;
    params->min_rep_default = MIN_REP_DEFAULT;
    params->standard_error_default = STANDARD_ERROR_DEFAULT;
    params->time_meas_default = TIME_MEAS_DEFAULT;
    params->time_suite_default = TIME_SUITE_DEFAULT;
    params->cut_quantile_default = CUT_QUANTILE_DEFAULT;
    params->multiple_of_default = MULTIPLE_OF_DEFAULT;
    params->absolute = FALSE; /* as default */
    params->post_proc = TRUE; /* as default */
    params->comment[0] = NULL;
    params->measurements[0] = NULL; /* or all? */
    params->new_section[0] = "Hello World";
    params->new_section[1] = NULL;

    return (params);
}
```

Look at the last two assignments: Since we have defined new_section as a text_t (definition in file skampi_tools.h) it is an array of strings. Note that this array is NULL-terminated. The constant TEXT_LINES describing its size is defined in skampi_tools.h.
4. In function `parse_parameter_file` you have to add code converting a line of the parameter file (which is provided in `corrected_line`) into the format of the parameter. We will use the function `insert_in_text`, to add this line at its correct position in the text `new_section`.

```c
    case NEWSECTION_MODE:
        insert_in_text (corrected_line, &(params->new_section),
                        line_counter);
        break;
```

At this position you may also do some syntax checking. SKaMPI usually aborts, if an syntax error occurs.

5. The function `line_mode` is responsible for recognising the sections in the parameter file. (A line of this file is provided in `line`. Here you have to add code, which shouts, when hitting our new section. Then you have to set the mode and to correct the line. This means cutting of the keyword. Our keyword in the parameter file will be `@NEWSECTION`.

```c
    if ((new_line = strchr(line, "@NEWSECTION")) != NULL) {
        *mode = NEWSECTION_MODE;
        return (new_line + strlen ("@NEWSECTION"));
    } else ... 
```

6. Function `read_parameters` is the chief-parameter-parser. It coordinates all other functions. Here we must ensure, that the successfully parsed section is send to all other processes. We do this with MPI-Functions (Ok, not really surprisingly). For sending a text, we have special functions: `send_text` and `receive_text`. Note that the order sending all parameters is important. It has to be the same as receiving the parameters.

```c
    ...
    send_text (&(params->node));
    ...
    MPI_Bcast (&(params->absolute), 1, MPI_INT, 0,
               default_communicator);
    send_text (&(params->comment));
    send_text (&(params->measurements));
    ...
    send_text (&(params->new_section));
} else /* so not proc. 0 */
    { /* receive params-struct */
    ...
```
6.2 New measurements

When creating a new measurement the first decision is which pattern is to use. (There is no measurement possible, without using a pattern.) The patterns are introduced in the user manual [4]. If you cannot find a suitable pattern, you have to build a new one. Please see section 6.3 for further information. In this example we want to add new ping-measurement, which uses MPI_Send for sending a message to another node. No reply is expected.

1. For this measurement the point-to-point pattern can be used, since only two nodes are involved (sender and receiver). First we have to find out which callback functions we have to provide.\(^2\) In this case we have to code the two callbacks server\_op and client\_op. Since the first one is measured, it will contain the call of MPI_Send.

\[\begin{align*}
\text{MPI\_Status server\_Send (int msglen, int max\_node,} \\
\text{MPI\_Comm communicator)}
\end{align*}\]

\[\begin{align*}
\{ \\
\text{MPI\_Status status;} \\
\text{MPI\_Send (_skampi\_buffer, msglen, MPI\_CHAR,} \\
\text{max\_node, 0, communicator);} \\
\text{return (status);} \\
\}
\]

You may claim, that status is never used. That is right, but the p2p-pattern expects this prototype: MPI\_Status server\_Send (int msglen,

\(^2\text{Which callbacks you need depends on the pattern you use. All patterns are described in the user manual, section "But what is measured?".}\)
int max_node, MPI_Comm communicator). (See section “Data structures” for getting the right prototypes the patterns use.) The parameters supplied: the message length, the number of the communication partner and the communicator.

Usually the callbacks are grouped together in files ...test1.c. So we will add them to the file p2p_test1.c. The other callback client_send contains the corresponding receive.

```c
MPI_Status client_recv (int msglen, int node,
                        MPI_Comm communicator)
{
    MPI_Status status;

    MPI_Recv (_skampi_buffer, msglen, MPI_CHAR, 0, 0, communicator,
              &status);
    return (status);
}
```

Perhaps you asked, what is up with the address _skampi_buffer. This buffer is provided, after the call of with mem_init_one_buffer. SKaMPI (more exactly: the function measure_suite) takes care, that msglen will never exceed the size of this buffer.

2. Now we write the initialization function. Here we determine p2p data of the measurement struct.

```c
void p2p_init_Send (measurement_t *ms, data_t *data)
{
    ms->pattern = P2P;
    ms->server_init = mem_init_one_buffer;
    ms->client_init = mem_init_one_buffer;
    ms->server_free = mem_release;
    ms->client_free = mem_release;
    ms->data.p2p_data.server_op = server_Send;
    ms->data.p2p_data.client_op = client_Send;
    ms->data.p2p_data.which_to_measure = MEASURE_MAX;
    ms->data.p2p_data.len = DEF_MESSAGE_LEN;
    ms->data.p2p_data.result = data;
    ms->data.p2p_data.communicator = MPI_COMM_WORLD;
}
```

Note that ms->data.p2p_data.len = DEF_MESSAGE_LEN only will concern the message length, if you do not vary over message length. If you want to communicate with the node of minimum latency, set which_to_measure

---

3 This happens through the callback server.init, see below and section 6.2.
4 Constant found in sklib_const.h.
6.2. NEW MEASUREMENTS

= MEASURE_MIN;
To work with two buffers _skampi_buffer and _skampi_buffer_2 use
mem_init_two_buffers. The memory is released in all cases with mem_release.
All memory management functions belong to the module skampi_mem. The
next section comes up with further information.

To avoid compiler warnings use function prototypes. (The one of the
init-function are placed in p2p_test1.h.)

3. Now we provide a facility for controlling our new measurement throughout
the parameter file. So we have to change the function initialize_type
in module skampi_params.c.

    case 50: /* or another unused number */
            p2p_init_Send (ms, NULL);
            break;

The number you use in the first line here is the type in the parameter file,
to identify your measurement. (Note that for sequential measurements
(i.e., SEQ defined) we reuse the numbers of parallel case, since parallel ad
sequential measurements never can occur during the same run.) For exam-
ple this control block in the parameter file can initialize our measurement.

MPI_Send
{
    Type = 50;
    Variation = Length;
    Scale = Dynamic_log;
    Max_Repetition = Default_Value;
    Min_Repetition = Default_Value;
    Multiple_of = Default_Value;
    Time_Measurement = Default_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;
}
Message buffer handling

As we saw in the last section, when writing a callback function, we assume that _skampi_buffer (or also _skampi_buffer_2) is set to an valid memory address. To do this we just have to initialize the client/server initialization function pointer of the measurement struct to the routines mem_init_one_buffer (or mem_init_one_buffer respectively). But what is to do, if we need other buffers. (Like perhaps for callback of the master-worker pattern?) In this case we define in this pointers (for example in the suitable ...test1.c-file). Then we declare the in skampi_mem.c as extern. Then we can write our own memory initialization routine. Note the following facts:

- Our function must have this type: int mem_init_name (int nor, int nom, int nop), where nor is the number of repetitions\(^5\), nom the number of measurements\(^6\), is the number of processes involved in this measurements. For the existing memory-initializers it proved useful knowing this numbers.

- We can assume, that _skib (SKaMPI internal buffer) is already set to allocated memory (done by routine allocate_memory). At this location we have _skib_size bytes memory.

- We have to return the memory size in bytes, which we dispatched for _skampi_buffer. If you do not want to use _skampi_buffer at all, you should return the size, which should be the maximum message length.

6.3 New patterns

If you want to add a new pattern, you should ask yourself some questions.

- Over which variables should be varied ?

- Does the new pattern has any callback functions and what is the type of them?

- Do I want to use the automatic repetition mechanism of SKaMPI?

- Can I use the existing memory-initializers ? (See section 6.2 for further information.)

The following list shows the steps for adding a new pattern.

---

\(^5\)That is the size which is declared by Max_Repetition in the parameter file.

\(^6\)That is the size which is declared by Max_Steps in the parameter file.

\(^7\)New variables to vary over will called “variation” in the following.
1. First you have to declare a new constant. This constant is used in variable
\texttt{measurement\_t->pattern} to indicate, that your pattern should be used.
The declaration should made in \texttt{skampi.h}.

\begin{verbatim}
enum{P2P, MASTER\_WORKER, COLLECTIVE, SIMPLE, MY\_NEW\_PATTERN};
\end{verbatim}

If your new pattern has an extra variable to vary over, you can enter this also in this file.

\begin{verbatim}
enum{NODES, LENGTH, CHUNK, NO\_VARIATION, MY\_NEW\_VARIATION};
\end{verbatim}

2. Before implementing the pattern, we should group together the data, which is specific for this pattern (so not included in the \texttt{measurement\_t-}
structure). We can code this new struct also in \texttt{skampi.h}.

\begin{verbatim}
typedef struct
{
    int specific_data; /* whatever is useful */
    void (* a_callback) (int);
    data_t *result; /* Measured results OUT */
    double dummy\_time; /* dummy time for measurements in that communicator 
                        and pattern IN */
    double dummy\_time\_se; /* and its standard error IN */
}my\_new\_pattern\_data\_t;
\end{verbatim}

This is just an example with one callback (implemented as a pointer to a function) returning void and getting an int. The integer \texttt{specific\_data}
stands for any data declaration you can do here.

The last three declarations must be included in every pattern data struct. 
(They are used in the calling mechanism of SKaMPI.)

3. Patch the \texttt{measurement\_t-struct} (also found in \texttt{skampi.h}): add your new data struct in the data union.

\begin{verbatim}
union /* patternspecific data structures IN */
{
    p2p\_pattern\_data\_t p2p\_data;
    mw\_pattern\_data\_t mw\_data;
    col\_pattern\_data\_t col\_data;
    simple\_pattern\_data\_t simple\_data;
    my\_new\_pattern\_data\_t my\_new\_data;
}data;
\end{verbatim}
4. Now you can implement your new pattern. Usually every pattern is coded in an extra file (say `my_new_pattern.c`), which has to be linked or you have to adapt the script `sk2if`. (See next point.) However your pattern must have the type like `int my_new_pattern (measurement_t *ms)`. A prototype of your pattern should be placed in a header-file (e.g. `my_new_pattern.h`).

5. If you use the skosfile-mechanism, you have to adapt the `sk2if` script.

```bash
... 
cat ../any.h >>skosfile.c 
cat ../pqtypes.h >>skosfile.c 
cat pq Globe.h >>skosfile.c 
cat ../col.h >>skosfile.c 
cat ../mw.h >>skosfile.c 
cat ../p2p.h >>skosfile.c 
cat ../simple.h >>skosfile.c 
# new header of pattern 
cat ../my_new_pattern.h >>skosfile.c 

cat ../mw_test1.h >>skosfile.c 
cat ../col_test1.h >>skosfile.c 
cat ../p2p_test1.h >>skosfile.c 
cat ../simple_test1.h >>skosfile.c 
# new header of callbacks 
cat ../my_new_pattern_test1.h >>skosfile.c 

cat mw_test1_source.c >>skosfile.c 
cat col_test1_source.c >>skosfile.c 
cat p2p_test1_source.c >>skosfile.c 
cat simple_test1_source.c >>skosfile.c 
# new source of callbacks 
cat my_new_pattern_test1_source.c >>skosfile.c 

cat skampi_source.c >>skosfile.c 
cat datalist_source.c >>skosfile.c 
cat skampi_error_source.c >>skosfile.c 
cat skampi_params_source.c >>skosfile.c 
... 
```

6. If you have an extra variation for your new pattern, you will have to do some extra work. It is explained in the next section.

7. Now look for all functions, which depend on the patterns. Mainly these are `measure` in `skampi.c`. In `skampi`, `call` you have to look at: `call_length`, `call_nodes`, `fill dummy values` and in `skampi_tools.c` `measurement_data to string`
Implementing a new variation

This section gives some additional tips when implementing a new variation.

In the file skampi_call.c we have to create a new call_{...}-function. The goal of this function is calling the pattern with the correct value of the variable parameter. (So call_length calls the p2p-, mw- or col-pattern with a specific message length.) Another point not forget: Usually we have to control our new pattern through the parameter file. So we have to implement a new keyword for the variation-entry. We consider something like this:

```c
... 
Type = 30; 
Variation = My_new_variation; 
Scale = Dynamic_log; 
... 
```

First we have an new keyword, which can easily added to the keywords-struct in skampi_params.c.

```c
... 
{"Dynamic_linear", DYN_LIN_SCALE},
{"Dynamic_log", DYN_LOG_SCALE},
{"Max_Value", MAX_VALUE},
{"Default_Value", DEFAULT_VALUE},
{"My_new_variation", MY_NEW_VARIATION_TOKEN},
{NULL, 0}
... 
```

Here **MY_NEW_VARIATION_TOKEN** is a new token, which we declare at the begining of this file. As a last step, we have to change *variation_style*.

```c
... 
    case CHUNKS_VAR: 
        ms->variation = CHUNK; 
        break; 
    case NO_VAR: 
        ms->variation = NO_VARIATION; 
        break; 
    case MY_NEW_VARIATION_TOKEN: 
        ms->variation = MY_NEW_VARIATION; 
        break; 
    default: 
        sprintf (_skampi_msg,"syntax error in line: %d: unknown variation.\n%\n", 
            lineno, (*text)[lineno]); 
        ERROR(USER,_skampi_msg);
        output_error (TRUE);
        ... 
```
Chapter 7

Index of all functions

7.1 Module skampi

This is the main module. It contains main() and the most global variables.
(Other specific globals can be found in mw.c.) The debug-switches here are
valid for all other modules if you use skosfile.

7.1.1 Function main

Prototype: int main(int argc, char **argv);

Purpose: reads parameters, creates log_file, output_file, calls all selected mea-
surements, logs measurements, calls postprocessing (if wanted).

Parameters: standard command line (argc, argv)

Returns: 0 if success

Position: lines 106 - 134.

Sideeffects: sets all global variables

7.2 Module autodist

This module is responsible for the automatic determination of the arguments
where to measure. Its interface can be found in autodist.h.
measure_suite calls the function tbm with the arguments computed and collects
the results in a list (stored in ms.result_list).
7.3. **MODULE AUTOMEASURE**

7.2.1 **Function measure-suite**

**Prototype:** void measure-suite (measurement J *ms, tbn J tbn);

**Purpose:** computes all arguments, where to measure the measurement ms and calls it via the routine tbn (to be measured).

**Parameters:** above.

**Returns:** nothing

**Position:** lines 55 - 475.

**Sideffects:** sets ms ->x_end to effective value, if it is initialized to MAX_ARGUMENT, exits in case of error.

**Assumes:** _skmpi.myid set.

7.2.2 **Function calculate_key**

**Prototype:** double calculate_key (measurement J *ms, PqData pqdata, int logflag);

**Purpose:** computes the key for the x_axis - segment for inserting it into the Priority-queue. We use the result cleaned (which is a design decision).

**Parameters:** the actual measurement, the x_axis segment pqdata.

**Returns:** the key

**Position:** lines 491 - 562.

**Assumes:** MACRO FUN defined.

7.3 **Module automeasure**

This modules offers the routines controling the repetitions of measurements.
Its interface is found in automeasure.h. Before using any other routine am_init
should be called (and am_free as last). Called after a measurement am_control_end
determines if a repetition is necessary. If finished, call am_fill_data to store the
accumulated data.
7.3.1 Function am_init

Prototype: int am_init (int size);

Purpose: initializes all private (static) variables allocates memory.

Parameters: the size (in bytes) of memory to be allocated.

Returns: TRUE iff allocation ok, FALSE otherwise.

Position: lines 83 - 106.

Sideeffects: changing the mentioned variables.

7.4 Module "standard_error..." given at the beginning of function


7.4.1 Function am_free

Prototype: void am_free (void);

Purpose: frees the allocated buffers

Parameters: none.

Returns: nothing.

Position: lines 118 - 127.

Sideeffects: memory freed, variables set back to zero.

Assumes: ttm_buffer has been allocated before.

7.4.2 am_control_end

Prototype: depending whether sequential or MPI version of SKaLib used.
  Please refer to the code.

Purpose: controls whether the ms at arg arg should be repeated (returns
  TRUE) or not (returns FALSE).

Parameters: the current measurement ms, the actual argument arg, the mea-
  sured time (ttm_time), the node_time, the partner (-process) also involved
  into this measurement. If there is no other process involved, partner is
  set to NO_COMMUNICATION. If it is set to USE_COMMUNICATOR,
the argument local_communicator will be used. (For communication
to more than one other processes. The root is process 0).

Returns: see above.

Position: lines 160 - 343.

Sideeffects: none the static variables.

Assumes: am_init has been called before.

7.4.3 Function am_fill_data

Prototype: void am_fill_data (measurement *, *ms, int arg, double dummy_time,
double dummy, data *, data);

Purpose: fills the data (dummy_time, standard_error of the dummy time into
the actual measured data.

Parameters: above

Returns: nothing.

Position: lines 358 - 476.

Assumes: am_init has been called before. critical_min_time set.

7.4.4 Function double_cmp

Prototype: int double_cmp (const void *d1, const void *d2);

Purpose: compares to doubles, used for qsort-calls

Parameters: two pointers to doubles d1, d2

Returns: 0 iff equal, -1 iff d1 < d2, 1 else

Position: lines 488 - 493.

7.5 Module col

This module contains the collective-pattern. This pattern is used to measure
collective MPI operations. The interface is described in col.h.
7.5.1 Function col_pattern

Prototype: void col_pattern (measurement *ms);

Purpose: the collective pattern.

Parameters: the actual measurement (which should be one of the collective pattern).

Returns: nothing.

Position: lines 46 - 162.

7.6 Module col_test1

This module contains all routines to be measured with the col-pattern. These are routines to initialize (col_init...) and routines containing the MPI-Functions to be measured.

7.6.1 Functions col_init...

Purpose: the following col_init... functions initialize the ms with the correct data to measure the specific collective MPI function.

Parameters: measurement ms and the place to hold the measured results (data).

Returns: nothing.


7.6.2 Functions measure...

Purpose: call the MPI-Function to be measured. The reason not to call this MPI-Function directly is to achieve a function-header common to all measured functions.

Parameters: message length len, Communicator communicator.

Returns: nothing.

Position of first: lines 516 - 519.

Assumes: skmpi_buffer (skmpi_buffer_2 set correctly, done with mem_init_one_buffers or mem_init_two_buffers.
7.7 Module mw

This module is simply the master-worker-pattern. This pattern is used to mea-
sure all the measurements of the master-worker-pattern. The interface is de-
scribed in mw.h.

7.7.1 Function mw_pattern

Prototype: int mw_pattern (measurement *ms);

Purpose: executes the master-worker pattern

Parameters: the actual measurement

Returns: TRUE in case if success

Position: lines 43 - 141.

7.8 Module mw_test1

This module contains all routines to be measured with the master-worker-
pattern. These are routines to initialize (mw_init...) and routines containing
the MPI-Functions to be measured.

7.8.1 Functions mw_init...

Purpose: initialize the measurement *ms (address for measured results is data.)

Parameters: above

Returns: nothing.


7.8.2 Functions master_receive_ready_test

Purpose: call the MPI-Function to be measured. The reason not to call this
MPI-Function directly is to achieve a function-header comman to all mea-
sured functions.

Parameters: message length len, Communicator communicator.

Returns: nothing.

Position of first: lines 311 - 315.
Assumes: the buffers set correctly, done with the routines called through ms->server_init /ms->client_init

7.9 Module p2p

Document created automatically by docmeas.pl at Wed Mar 17 13:17:44 1999. This module is simply the p2p-pattern. This pattern is used to measure all the measurements of the p2p-pattern. The interface is described in p2p.h.

7.9.1 Function p2p_find_max_min

Prototype: int p2p_find_max_min (measurement *ms);

Purpose: finds nodes with minimum/maximum latency

Parameters: a measurement with the p2p pattern.

Returns: TRUE in case of success

Position: lines 63 - 218.

Sideeffects: modifies ms->data.p2p_data.max_node and ms->data.p2p_data.min_node

Assumes: syscall_myid set

7.9.2 Function p2p_pattern

Prototype: int p2p_pattern (measurement *ms);

Purpose: the p2p pattern.

Parameters: the actual measurement (which could be one of the p2p pattern).

Returns: TRUE in case of success.

Position: lines 230 - 411.

7.10 Module p2p_test1

Document created automatically by docmeas.pl at Wed Mar 17 13:17:44 1999. This module contains all routines to be measured with the p2p-pattern. These are routines to initialize (p2p_init...) and routines containing the MPI-Functions to be measured.
7.10.1 Functions p2p\_init\_...

**Purpose:** the following p2p\_init\_... functions initialize the ms with the correct data, to measure the specific point-to-point MPI function.

**Parameters:** measurement ms and the place to hold the measured results (data).

**Returns:** nothing.

**Position of first:** lines 82 - 94.

7.10.2 Functions server\_...

**Purpose:** call the MPI-Function to be measured by the process 0 (sometimes named server). The reason not to call this MPI-Function directly is to achieve a function-header comman to all measured functions.

**Parameters:** message length len, number of the node to communicate with, Communicator comminicator.

**Returns:** nothing.

**Position of first:** lines 316 - 326.

**Assumes:** skampi\_buffer ( skampi\_buffer\_2 set correctly, done with mem\_init\_one\_buffers or mem\_init\_two\_buffers.

7.10.3 Functions client\_...

**Purpose:** call the MPI-Function to be measured by the processes not 0 (sometimes named clients). The reason not to call this MPI-Function directly is to achieve a function-header comman to all measured functions.

**Parameters:** message length len, number of the node to communicate with, Communicator comminicator.

**Returns:** nothing.

**Position of first:** lines 446 - 456.

**Assumes:** skampi\_buffer ( skampi\_buffer\_2 set correctly, done with mem\_init\_one\_buffers or mem\_init\_two\_buffers.

7.11 Module simple

Document created automatically by documeas.pl at Wed Mar 17 13:17:45 1999. This module is simply the simple-pattern. This pattern is used to measure all the measurements of the simple-pattern. The interface is described in simple.h.
7.11.1 Function simple_pattern

Prototype: void simple_pattern (measurement_t *ms);

Purpose: the simple pattern.

Parameters: th actual measurement (which should be one of the simple pattern).

Returns: nothing.

Position: lines 39 - 104.

Assumes: none.

7.12 Module simple_test1

Document created automatically by document.pl at Wed Mar 17 13:17:45 1999. This module contains all routines to be measured with the simple-pattern. These are routines to initialize (simple_init...) and routines containing the MPI-Functions to be measured.

7.12.1 Functions simple_init...

Purpose: the following simple_init... functions initialize the ms with the correct data, to measure the specific simple MPI function.

Parameters: measurement ms and the place to hold the measured results (data).

Returns: nothing.

Position of first: lines 52 - 61.

7.12.2 Functions measure...

Purpose: call the MPI-Function to be measured. The reason not to call this MPI-Function directly is to achieve a function-header common to all measured functions.

Parameters: message length len, Communicator communicator.

Returns: nothing.

Position of first: lines 154 - 156.

Assumes: the routines pointed by ms->server_init ms->client_init are called.
7.13 Module datalist

Document created automatically by documeas.pl at Wed Mar 17 13:17:43 1999. This module provides all the basic routines for maintaining double-linked-lists. It is not only appliable for skampi, but some minor changes have been made, to improve usability. So the functions item_addr and item_addr_at_item have been added for interaction with the priority-queue in module autodist. The complete interface can be found (as usual) in the header datalist.h. Note that for use in skampi, the routine init_data needs the number of PEs skampi is running on, which is provided in the variable numprocs, which is set in skampi.c

7.13.1 Function init_list

Prototype: data_list *init_list(data_list *l);

Purpose: initializes the data list l to the empty list.

Parameters: pointer to the list l.

Returns: the adress of the list, or NULL iff no memory available.

Position: lines 54 - 68.

7.13.2 Function add

Prototype: data_list *add (data_list *l, int mode, signed int pos, data *data, int *error);

Purpose: adds data data element (*data) to the list *l at the position pos relative to start or list (mode == START), or end (mode == END), or to last accesed element (mode == LAST).

Parameters: (add. to above) *error, in which the error code is retuned.

Returns: pointer to list, or NULL in case of error.

Position: lines 87 - 197.

7.13.3 Function read_ele

Prototype: data *read_ele (data_list *l, int mode, signed int pos, data *data, int *error);

Purpose: reads data data element (*data) of the list *l at the position pos relative to start or list (mode == START), or end (mode == END), or to last accesed element (mode == LAST).

Parameters: (add. to above) *error, in which the error code is retuned.
Returns: pointer to list, or NULL in case of error.


7.13.4 Function read_item_ele

Prototype: data_t *read_item_ele(data_list_t *1, list_item_t *local_ptr, signed int pos, data_t *data, int *error);

Purpose: reads data data element (*data) of the list *l at the position pos relative to the element pointed to with local_ptr.

Parameters: (add. to above) *error, in which the error code is returned.

Returns: pointer to list, or NULL in case of error.


7.13.5 Function item_addr

Prototype: list_item_t *item_addr(data_list_t *1, int mode, signed int pos, int *error);

Purpose: returns the address of the data element of the list *l at the position pos relative to start or end (mode == START), or end (mode == END), or to last accessed element (mode == LAST).

Parameters: (add. to above) *error, in which the error code is returned.

Returns: pointer to list, or NULL in case of error.

Position: lines 338 - 389.

7.13.6 Function item_addr_at_item

Prototype: list_item_t *item_addr_at_item(data_list_t *1, list_item_t *local_ptr, signed int pos, int *error);

Purpose: returns the address of the data element of the list *l at the position pos relative to the element pointed to with local_ptr.

Parameters: (add. to above) *error, in which the error code is returned.

Returns: pointer to list, or NULL in case of error.

Position: lines 406 - 441.
7.13.7 Function is_end
Prototype: int is_end (list_item_t *item);
Purpose: tests if *item is the last elemnt of its list
Parameters: item of a list *item
Returns: TRUE iff last element
Position: lines 453 - 455.
Assumes: item != NULL.

7.13.8 Function is_start
Prototype: int is_start (list_item_t *item);
Purpose: tests if *item is the first elemnt of its list
Parameters: item of a list *item
Returns: TRUE iff first element
Position: lines 467 - 469.
Assumes: item != NULL.

7.13.9 Function number_of_elements
Prototype: int number_of_elements (data_list_t *l);
Purpose: returns number of elements of the list *l.
Parameters: above
Returns: above
Position: lines 481 - 486.
Sideeffects: none

7.13.10 Function remove_ele
Prototype: data_list_t *remove_ele (data_list_t *l, int mode, signed int pos);
Purpose: removes data data element of the list *l at the position pos relative to start or list (mode == START), or end (mode == END), or to last accessed element (mode == LAST).
Parameters: above.
Returns: pointer to list, or NULL in case of error.

Position: lines 502 - 568.

### 7.13.11 Function free\_data\_list

**Prototype:** void free\_data\_list (data\_list\* l, int mode);

**Purpose:** free-es all elements of the data\_list *l and (only if mode == DYNAMIC) also the memory pointed by l. If this is not wanted (e.g. because l is address of statically allocated variable) call with mode == STATIC.

**Parameters:** above

**Returns:** nothing.

**Position:** lines 582 - 603.

### 7.13.12 Function minimum

**Prototype:** double minimum (data\_list\* l, int *arg);

**Purpose:** find the minimum of the list *l, returns the argument, of the minimal element (refers to cleaned). (needs the structure of the data stored in a list element)

**Parameters:** above.

**Returns:** above.

**Position:** lines 622 - 642.

### 7.13.13 Function maximum

**Prototype:** double maximum (data\_list\* l, int *arg);

**Purpose:** find the maximum of the list *l, returns the argument, of the maximal element (refers to cleaned). (needs the structure of the data stored in a list element)

**Parameters:** above.

**Returns:** above.

**Position:** lines 656 - 676.
7.13. MODULE DATALIST

7.13.14 Function variance
Prototype: double variance (data_list_t *l);
Purpose: returns the variance of the list *l (needs the structure of the data stored in a list element)
Parameters: above.
Returns: above.
Position: lines 689 - 707.

7.13.15 Function average_of_lists
Prototype: data_list_t *average_of_lists (data_list_t **l);
Purpose: creates a new list, where the i-th data element is the average of all i-th elements of the data lists given in the (NULL-terminated!) array of lists l. (needs the structure of the data stored in a list element)
Parameters: above.
Returns: new list, NULL in case of error.
Position: lines 723 - 782.

7.13.16 Function average
Prototype: double average (data_list_t *l);
Purpose: returns the average of the list *l. (refers to cleaned results) (needs the structure of the data stored in a list element)
Parameters: above.
Returns: above.
Position: lines 802 - 818.

7.13.17 Function write_to_file
Prototype: int write_to_file (data_list_t *l, FILE *file);
Purpose: writes data list *l to the file *file.
Parameters: above
Returns: TRUE iff successful, FALSE otherwise
Position: lines 832 - 871.
7.13.18 Function read_from_file

Prototype: data_list_t * read_from_file (data_list_t *l, FILE **file, int *error);

Purpose: reads data list *l from the file **file. note: file is ** so that reading
in file changes the filepointer, useful for reading consecutive lists in one
file.

Parameters: above

Returns: address of the list read, NULL in case of error.

Position: lines 885 - 1029.

Sideeffects: none

Assumptions: **file is valid handle of an open file.

7.14 Module skampi_error

This module provides the error handling, including the standard error classes
and messages.

7.14.1 Function output_error

Prototype: void output_error (int really_end);

Purpose: prints error message [in skampi_error] to stderr and (only if re-
ally_end == TRUE) aborts the running program.

Parameters:

Returns:


7.15 Module skampi_mem

Here you can find the management of the message-buffers (the memory for
storing the results is allocated in automeasure.c and datalist.c resp.).
7.15. **MODULE SKAMPI_MEM**

7.15.1 **Function allocate_mem**

*Prototype:* int allocate_mem (int memsize);

*Purpose:* allocates the memory for the internal buffer

*Parameters:* the size of memory to allocate (in bytes).

*Returns:* TRUE iff successful, FALSE otherwise.

*Position:* lines 97 - 128.

*Sideeffects:* manipulation of the static variables.

*Assumes:* first call or free_mem called before.

7.15.2 **Function free_mem**

*Prototype:* void free_mem (void);

*Purpose:* frees all allocated memory.

*Parameters:* none.

*Returns:* nothing.

*Position:* lines 141 - 148.

*Sideeffects:* manipulation of the static variables.

*Assumes:* allocate_mem called before and no call of free_mem after that call of allocate_mem.

7.15.3 **Function mem_init_one_buffer**

*Prototype:* long int mem_init_one_buffer (int nor, int nom, int nop);

*Purpose:* sets _skampi_buffer to _skib (i.e. a location of allocated memory.)

*Parameters:* number of repetitions (nor), number of measurements (nom),
number of processes involved in the measurement (nop).

*Returns:* the size of memory available at _skampi_buffer.

*Position:* lines 163 - 182.

*Sideeffects:* manipulation of _skampi_buffer.

*Assumes:* assumes allocate_mem called once before with no other mem_init_one_buffer called between.
7.15.4 Function mem_init_two_buffers

Prototype: long int mem_init_two_buffers (int nor, int nom, int nop);

Purpose: sets _skampi_buffer and _skampi_buffer_2 a location of allocated memory.

Parameters: number of repetitions (nor), number of measurements (nom), number of processes involved in the measurement (nop).

Returns: the size of memory available at _skampi_buffer and _skampi_buffer_2.

Position: lines 199 - 224.

Side effects: manipulation of _skampi_buffer.

Assumes: assumes allocate_mem called once before with no other mem_init_. between.

7.15.5 Function mem_init_two_buffers_gather

Prototype: long int mem_init_two_buffers_gather (int nor, int nom, int nop);

Purpose: sets _skampi_buffer and _skampi_buffer_2 a location of allocated memory suitable for the MPI_Gather operation. (_skampi_buffer for sending, _skampi_buffer_2 for receiving.)

Parameters: number of repetitions (nor), number of measurements (nom), number of processes involved in the measurement (nop).

Returns: the size of memory available at _skampi_buffer and _skampi_buffer_2.

Position: lines 243 - 270.

Side effects: manipulation of _skampi_buffer.

Assumes: assumes allocate_mem called once before with no other mem_init_. between.

7.15.6 Function mem_init_two_buffers_alltoall

Prototype: long int mem_init_two_buffers_alltoall (int nor, int nom, int nop);

Purpose: sets _skampi_buffer and _skampi_buffer_2 a location of allocated memory. The difference to mem_init_two_buffers is, that its result is divided by nop, because MPI_Alltoall need a buffer of the size message length * nop.

Parameters: number of repetitions (nor), number of measurements (nom), number of processes involved in the measurement (nop).
7.15. MODULE SKAMPI_MEM

Returns: the size of memory available at skampi_buffer and skampi_buffer_2.

Position: lines 290 - 294.

Sideeffects: manipulation of skampi_buffer.

Assumes: assumes allocate_mem called once before with no other mem_init...

between.

7.15.7 Function mem_init_two_buffers_attach

Prototype: long int mem_init_two_buffers_attach (int nor, int nom, int nop);

Purpose: sets skampi_buffer and skampi_buffer_2 a location of allocated memory. The difference to mem_init_two_buffers is, that its result is divided by
nop, because Bsend of the mw-pattern need a buffer of the size message
length * nop.

Parameters: number of repetitions (nor), number of measurements (nom),
number of processes involved in the measurement (nop).

Returns: the size of memory available at skampi_buffer and skampi_buffer_2.


Sideeffects: manipulation of skampi_buffer.

Assumes: assumes allocate_mem called once before with no other mem_init...

between, default_communicator set

7.15.8 Function find_mml

Prototype: long int find_mml (int nor, int nom, int nop);

Purpose: computes the max message length when using MPI_Bsend

Parameters: number of repetitions (nor), number of measurements (nom),
number of processes involved in the measurement (nop).

Returns: above

Position: lines 369 - 397.

Assumes: skib_size, default_communicator set
7.15.9 Function mem_init_two_buffers_attach_p2p

Prototype: long int mem_init_two_buffers_attach_p2p (int nor, int nom, int nop);

Purpose: sets skampi_buffer and skampi_buffer_2 a location of allocated memory. The difference to mem_init_two_buffers is, that its result is NOT divided by anything (special for p2p pattern)

Parameters: number of repetitions (nor), number of measurements (nom), number of processes involved in the measurement (nop).

Returns: the size of memory available at skampi_buffer and skampi_buffer_2.

Position: lines 416 - 418.

Sideeffects: manipulation of skampi_buffer.

Assumes: assumes allocate_mem called once before with no other mem_init...
          between, default_communicator set

7.15.10 Function mem_init_two_buffers_attach_mw

Prototype: long int mem_init_two_buffers_attach_mw (int nor, int nom, int nop);

Purpose: sets skampi_buffer and skampi_buffer_2 a location of allocated memory. The difference to mem_init_two_buffers is, that its result is NOT divided by mop and nom (special for mw pattern)

Parameters: number of repetitions (nor), number of measurements (nom), number of processes involved in the measurement (nop).

Returns: the size of memory available at skampi_buffer and skampi_buffer_2.


Sideeffects: manipulation of skampi_buffer.

Assumes: assumes allocate_mem called once before with no other mem_init...
          between, default_communicator set

7.15.11 Function mem_release_detach

Prototype: void mem_release_detach (void);

Purpose: releases the skampi_buffers. It must be called after allocate_mem. It does NOT free the allocated memory of skib. It is the counterpart of the mem_init_two_buffers_attach functions.
Parameters: none.

Returns: nothing.

Position: lines 453 - 467.

7.15.12 Function mem_init_mw_Waitsome

Prototype: long int mem_init_mw_Waitsome (int nor, int nom, int nop);

Purpose: sets _skampi_buffer and the __mw__... variable to locations of allocated memory. (Special for master_dispatch_Waitsome in mw_test1.c.)

Parameters: number of repetitions (nor), number of measurements (nom), number of processes involved in the measurement (nop).

Returns: the size of memory available at _skampi_buffer.

Position: lines 484 - 527.

Sideeffects: manipulation of the mentioned variables.

Assumes: assumes allocate_mem called once before with no other mem_init__... between.

7.15.13 Function mem_init_mw_Waitany

Prototype: long int mem_init_mw_Waitany (int nor, int nom, int nop);

Purpose: sets _skampi_buffer and the __mw__... variable to locations of allocated memory. (Special for master_dispatch_Weather in mw_test1.c.)

Parameters: number of repetitions (nor), number of measurements (nom), number of processes involved in the measurement (nop).

Returns: the size of memory available at _skampi_buffer.

Position: lines 544 - 582.

Sideeffects: manipulation of the mentioned variables.

Assumes: assumes allocate_mem called once before with no other mem_init__... between.
7.15.14 Function mem_init_col_Waitall

Prototype: long int mem_init_col_Waitall (int nor, int nom, int nvp);

Purpose: sets _skampi_buffer and the _col_ variable to locations of allocated
memory. (Special for col_init_Gather_Waitall_server in col_test1.c.)

Parameters: number of repetitions (nor), number of measurements (nom),
number of processes involved in the measurement (nvp).

Returns: the size of memory available at _skampi_buffer.

Position: lines 601 - 646.

Sideeffects: manipulation of the mentioned variables.

Assumptions: assumes allocate_mem called once before with no other mem_init_col_ between.

7.15.15 Function mem_release

Prototype: void mem_release (void);

Purpose: 'releases' the _skampi_buffers. It must be called after allocate_mem.
It does NOT free the allocated memory of _skib. It is the counterpart of
the mem_init_one_buffer and the mem_init_two_buffers functions.

Parameters: none.

Returns: nothing.

Position: lines 663 - 672.

7.16 Module skampi_params

This module provides the complete parameter file parser. This means the
routines for dividing this file into its several sections (each begining with an
@). To parse the MEASUREMENTS:Section, the 'real' parser will be used.
The structure of the used "compiler" is from Aho, Sethi, Ullman, Compilerbauer
I, Kap. 2 (i.e. german edition of the dragon-book) Add.Wes., 1988

7.16.1 Function read_parameters

Prototype: measurement *read_parameters (char *parameter_file_name, params *params, int *no_params);
7.16. MODULE SKAMPI_PARAMS

**Purpose:** reads the parameter file, fills params and returns a filled measurements array, in *no_meas* the number of measurements is given back returns NULL in case of error

**Parameters:** name of parameter file, the params-struct which will be filled the number of measurements (*no_meas*) (also filled here)

**Returns:** returns to an array of *no_meas* filled measurements in case of success.

**Position:** lines 237 - 389.

**Sideeffects:** in case of error it aborts the program.

**Assumes:** skampi_myid set.

### 7.16.2 Function init_params

**Prototype:** params_t *init_params (params_t *params);

**Purpose:** initializes the parameter struct with its default values, the definition of the constants can be found in skampi_error.h called by read_parameters.

**Parameters:** the parameter array to be filled.

**Returns:**

**Position:** lines 404 - 428.

### 7.16.3 Function parse_parameter_file

**Prototype:** params_t *parse_parameter_file (FILE *parameter_file, params_t *params);

**Purpose:** parses the parameter file into the struct *params.

**Parameters:** above

**Returns:** the filled params-struct or in case of error NULL.

**Position:** lines 440 - 683.

### 7.16.4 Function line_mode

**Prototype:** char *line_mode (char *line, int *mode);

**Purpose:** analyzes *line and ectl. sets *mode to a new found mode.

**Parameters:** above.

**Returns:** a pointer to the line (without the keyword).

**Position:** lines 695 - 778.
7.16.5 Function send_text

Prototype: void send_text (text_t *text);

Purpose: sends a text to all other processes in default communicator, process zero is root.

Parameters: the text to send.

Returns: nothing.

Position: lines 819 - 834.

Assumes: brackets default communicator set.

7.16.6 Function recv_text

Prototype: text_t *recv_text (text_t *text);

Purpose: receives the text which has been send via send_text. (process zero in default communicator is root.) Note: the text_t-struct has to be allocated, not the memory for all the strings, this is done here.

Parameters: above.

Returns: a pointer to the filled text structure.

Position: lines 849 - 877.

Assumes: default communicator set.

7.16.7 Function read_next_char

Prototype: int read_next_char (text_t *text);

Purpose: reads next character (not whitespace) from the text *text and returns it. Note: a char is treated as an (signed!) int (which is necessary, because EOT an other constants are negative.)

Parameters: above.

Returns: returns character, or EOT (end of text, if there is no further character)

Position: lines 895 - 912.

Sideeffects: manipulates pos and lineno.

Assumes: pos and lineno are initialized.
7.16. **MODULE SKAMPIPARAMS**

### 7.16.8 Function unread_next_char

**Prototype:** char *unread_next_char (int t, text_t *text);

**Purpose:** unread the last character, like ungetc of the standard library.

**Parameters:** the character to unread (t, not used actually, only for similarity to ungetc). Note: a char is treated as an (signed!) int (which is necessary, because EOT an other constants are negative.)

**Returns:** a pointer to the actual character to read in the text *text.

**Position:** lines 927 - 934.

**Sideeffects:** manipulates pos.

**Assumes:** pos and lineno are initialized.

### 7.16.9 Function init_symboltable

**Prototype:** void init_symboltable (void);

**Purpose:** initializes the symboltable with the reserved words. So to add a new reserved word, just add it to the array keywords.

**Parameters:** none

**Returns:** nothing.

**Position:** lines 948 - 953.

**Sideeffects:** manipulates symboltable.

**Assumes:** keywords initialized

### 7.16.10 Function lookup

**Prototype:** int lookup (char *s);

**Purpose:** looks up the string *s in the symboltable.

**Parameters:** above.

**Returns:** the index of *s if found, 0 otherwise.

**Position:** lines 967 - 975.

**Assumes:** symboltable and lastentry initialized.
7.16.11  Function insert

Prototype: int insert (char *s, int tok);

Purpose: inserts the string *s (known as token tok) into the symboltable.

Parameters: above.

Returns: the index of *s in the symboltable.

Position: lines 988 - 1011.

Skeleffects: increases lastentry.

Assumes: symboltable and lastentry initialized.

7.16.12  Function lexan

Prototype: int lexan(int *text);

Purpose: scans next token in the text.

Parameters:

Returns: returns next token if found (the (int)tokenval) the index of its actual value in the symboltable unless: token is INT (then (int)tokenval) has its value. token is FLOAT (then (tokenval) has its value. token is DONE if EOT reached.

Position: lines 1028 - 1101.

Skeleffects: through calling read_next_char.

Assumes: symboltable initialized.

7.16.13  Function match

Prototype: void match(int t, int *text);

Purpose: compares the lookahead character with the expected (t) and calls the scanner.

Parameters: additional: *text for calling the scanner.

Returns: nothing

Position: lines 1115 - 1127.

Skeleffects: aborts with error message if comparison fails.
7.16. MODULE SKAMPIPARAMS

7.16.14 Function parse

Prototype: void parse (measurement *ms, text *text);

Purpose: analyses entries of the @MEASUREMENTS-Section of the parameter-file.

Parameters: a pointer to an array of measurements. This array has to be big enough. (The size of the array can be obtained with count_measurements.) The text is a pointer to params.measurements usually.

Returns: nothing.

Position: lines 1144 - 1151.

Sideeffects: measurement called aborts in case of error.

Assumes: see above.

7.16.15 Function measurement

Prototype: void measurement (measurement *ms, text *text);

Purpose: fills one measurement-struct with the data parsed.

Parameters: the measurement *ms to be filled, and the *text which to parse.

Returns: nothing

Position: lines 1164 - 1307.

Sideeffects: aborts in case of error.

7.16.16 Function variation_style

Prototype: void variation_style (measurement *ms, text *text);

Purpose: decides which variation style lookahead is.

Parameters: the measurement *ms to be filled, and the *text which to parse.

Returns: nothing

Position: lines 1320 - 1344.

Sideeffects: aborts in case of error.
7.16.17 Function scale_style

Prototype: void scale_style (measurement_t *ms, text_t *text);

Purpose: decides which scale style lookahead is.

Parameters: the measurement *ms to be filled, and the *text which to parse.

Returns: nothing

Position: lines 1357 - 1379.

Side effects: aborts in case of error.

7.16.18 Function int_or_max

Prototype: void int_or_max (measurement_t *ms, text_t *text);

Purpose: decides whether lookahead is an int or the keyword MAX_VALUE.

Parameters: the measurement *ms to be filled, and the *text which to parse.

Returns: nothing

Position: lines 1392 - 1409.

Side effects: aborts in case of error.

7.16.19 Function int_or_default

Prototype: void int_or_default (int *val, text_t *text);

Purpose: decides whether lookahead is an int or the keyword DEFAULT_VALUE.

Parameters: the value val to be filled, and the *text which to parse.

Returns: nothing

Position: lines 1422 - 1439.

Side effects: aborts in case of error.

7.16.20 Function int_or_float

Prototype: void int_or_float (double *val, text_t *text);

Purpose: decides whether lookahead is an int or a float.

Parameters: the value val to be filled, and the *text which to parse.

Returns: nothing

Position: lines 1451 - 1466.

Side effects: aborts in case of error.
7.16. MODULE SKAMPI_PARAMS

7.16.21  Function yes_or_no

Prototype: void yes_or_no (int *val, text *text);

Purpose: decides whether lookahead is a "yes" or a "no".

Parameters: the value val be filled (1 = yes, 0 = no), and the *text which
to parse.

Returns: nothing

Position: lines 1479 - 1496.

Sideeffects: aborts in case of error.

7.16.22  Function float_or_default

Prototype: void float_or_default (double *val, text *text);

Purpose: decides whether lookahead is a float or the keyword DEFAULT_VALUE.

Parameters: the value val to be filled, and the *text which to parse.

Returns: nothing

Position: lines 1509 - 1526.

Sideeffects: aborts in case of error.

7.16.23  Function float_or_default_or_invalid

Prototype: void float_or_default_or_invalid (double *val, text *text);

Purpose: decides whether lookahead is a float or the keyword DEFAULT_VALUE
or the keyword INVALID_VALUE.

Parameters: the value val to be filled, and the *text which to parse.

Returns: nothing

Position: lines 1539 - 1559.

Sideeffects: aborts in case of error.
7.16.24 Function initialize_type
Prototype: void initialize_type (measurement *ms, int index, text *text);
Purpose: initializes the pattern specific data of *ms. There for the type of an measurement (index) is used.
Parameters: additional: *text, for scanner.
Returns: nothing.
Position: lines 1575 - 1858.
Side effects: manipulates *ms, aborts in case of error.

7.16.25 Function token_to_str
Prototype: char * token_to_str (int token);
Purpose: converts a token into a string, which is returnend. Used for debugging only.
Parameters: above.
Returns: above.

7.17 Module skampi_post
This module contains all routines need for the postprocessing (i.e. merging the output-files of several skampi-runs together to one file. This will mainly used by skampi.c and post.c.

7.17.1 Function post_processing
Prototype: int post_processing (char *input_file_name);
Purpose: complete postprocessing.
Parameters: name of input_file (which is the outputfile in skampi usually) a pointer to the array of measurements (not needed now, because measure- ment_data_to_ext command file and measurement_data_to_ext_module are now implemented in the perl-script dorep.pl. nom: number of measurements. (some as with ms].
Returns: TRUE. (return-type is int for further error management)
Sideeffects: aborts in case of error.
Assumes: params-struct is filled when called.

7.17.2 Function init_post_proc

Prototype: int init_post_proc (char *input_file_name);

Purpose: initializes all static variables of this module and opens all input files
(== output_files of skimpi).

Parameters: name of input file.
Returns: TRUE. (return-type is int for further error management)
Position: lines 170 - 250.
Sideeffects: aborts in case of error.

7.17.3 Function free_post_proc

Prototype: void free_post_proc (void);

Purpose: frees all allocated memory for internal use and closes all here
opened files
Parameters: none.
Returns: nothing.
Position: lines 264 - 279.
Sideeffects: on the internal variables.
Assumes: init_post_proc run before.

7.17.4 Function free_all_lists

Prototype: void free_all_lists (void);

Purpose: frees all data elements of every list. It does NOT free the array of
lists (this is done in free_post_proc.)
Parameters: none.
Returns: nothing.
Position: lines 292 - 298.
Sideeffects: on the data stored in the lists.
7.17.5 Function skip_to_next_meas

Prototype: char *skip_to_next_meas (int index);

Purpose: skips to next measurement of file input_files[index] returns name of that measurement

Parameters: above.

Returns: name of the measurement found or NULL in case of EOF.

Position: lines 311 - 324.

Sideeffects: on file pointer input_files[index]

Assumes: input_file[index] open (i.e. init_post_proc called before).

7.17.6 Function find_meas

Prototype: int find_meas (int index, char *search);

Purpose: finds measurement with the name *search in input_file[index]

Parameters: additional: name of measurement to look for.

Returns: returns TRUE iff found FALSE otherwise

Position: lines 336 - 349.

Sideeffects: on file pointer input_files[index]

Assumes: input_file[index] open (i.e. init_post_proc called before).

7.17.7 Function read_one_list_of_meas

Prototype: int read_one_list_of_meas (int index, char *meas);

Purpose: reads measurement with the name *meas in input_file[index] into the list addressed by lists[index].

Parameters: additional: name of measurement to look for.

Returns: returns TRUE iff found FALSE otherwise

Position: lines 363 - 406.

Sideeffects: on file pointer input_files[index] and on lists[index]

Assumes: input_file[index] open (i.e. init_post_proc called before).
7.17. **Module skmpi**

7.17.8 **Function read_all_lists_of_next_meas**

**Prototype:** char *read_all_lists_of_next_meas (void);

**Purpose:** reads all lists of next measurement (the next means the next in input_files[0]) and stores the read data in the lists **lists**.

**Parameters:** additional: name of measurement to look for.

**Returns:** returns name of that meas iff success

**Position:** lines 421 - 451.

**Sideeffects:** on file pointers input_files and on lists.

**Assumes:** init_post_proc called before.

7.17.9 **Function combine_lists**

**Prototype:** int combine_lists (data_list_t **result_list);

**Purpose:** combines all lists of the **lists-array to one new result_list.

**Parameters:** above.

**Returns:** TRUE if result_list contains really data elements.

**Position:** lines 464 - 574.

**Sideeffects:** if result_list == NULL it is allocated and initialized. Aborts in case of error.

**Assumes:** result list is initialized unless it is NULL.

7.17.10 **Function all_finished**

**Prototype:** int all_finished (int *vector);

**Purpose:** tests if all entries in vector are TRUE.

**Parameters:** above.

**Returns:** TRUE iff all elements of vector are TRUE, FALSE otherwise.

**Position:** lines 587 - 596.
7.17.11 Function interpolate_data

Prototype: data_t *interpolate_data (int value, data_t *left, data_t *right, data_t *data);

Purpose: interpolates a complete data_t-struct at value (relative to entry arg) between left and right.

Parameters: above.

Returns: pointer to the interpolated data-struct.

Position: lines 609 - 638.

Sideeffects: allocates new data element if data == NULL.

7.17.12 Function post_process

Prototype: data_t *post_process (data_t *new_data, data_t *result_data);

Purpose: this is the function which really decides how to merge several data-struct (stored in the array new_data) to one (result_data). It refers to the ...all values (which is certainly a design decision).

Parameters: above.

Returns: a pointer to result_data.

Position: lines 652 - 673.

Assumes: nif set.

7.17.13 Function data_cmp

Prototype: int data_cmp (const void *d1, const void *d2);

Purpose: compares the results of two data_t-structs, used for qsort-calls. Since used in function post_process, it refers to the result_all.

Parameters: two pointers to data_t-structs d1, d2

Returns: 0 iff equal, -1 iff d1 < d2, 1 else

Position: lines 686 - 691.

7.18 Module skampi_tools

This module contains several small and handy tools. Its routines are used by nearly every other skampi-module. The interface is declared in skampi_tools.h.
7.18. Function write_to_log_file

Prototype: int write_to_log_file (char *msg);

Purpose: writes the message msg to the log file (which also can be stdout or stderr. If it is really a file it will be opened and closed.

Parameters: above.

Returns: TRUE iff successful, FALSE in case of error.

Position: lines 81 - 115.

Assumes: skampi_myid and log_filename set.

7.18.2 Function measurement_data_to_string

Prototype: char *measurement_data_to_string (measurement_t *ms, char *string);

Purpose: builds a printable string string containing most of the data stored in *ms.

Parameters: above.

Returns: pointer to this string.

Position: lines 129 - 255.

Sideeffects: uses skampi_msg (as little buffer...)

7.18.3 Function read_header

Prototype: FILE * read_header (FILE *file, text_t *text);

Purpose: reads header (containing HEADER_LINES lines) of the measurement at the current position of the file-pointer into text.

Parameters: above.

Returns: the manipulated file_handle.

Position: lines 482 - 501.

Assumes: *file is a valid file handle of an open file, and its file-pointer really points to a header.
7.18.4 Function write_header

Prototype: FILE * write_header (FILE *file, text *text, char *name);

Purpose: writes the header (containing HEADER_LINES lines) stored in *text
in the file, this header will be named name there.

Parameters: above.

Returns: the manipulated file-handel.

Position: lines 515 - 533.

7.18.5 Function write_text_to_file

Prototype: void write_text_to_file (text *text, char *s, FILE *file);

Purpose: writes text to file *file in a section s. note: file is ** so that writing
in file changes the filepointer!

Parameters: above.

Returns: nothing.

Position: lines 546 - 556.

7.18.6 Function insert_in_text

Prototype: text * insert_in_text (char *line, text *text, int pos);

Purpose: inserts line into array text at position pos. (Note: text is (should)
always (be) an NULL-Pointer terminated array.

Parameters: above.

Returns: pointer to text, or NULL in case of error.

Position: lines 572 - 600.

Assumess: text points to array of TEXT_LINES char * or is NULL (than allo-
cates).

7.18.7 Function read_from_text

Prototype: char * read_from_text (char *line, text *text, int pos);

Purpose: reads the textline at position pos from text and stores it at line.

Parameters: above.

Returns: pointer to line.

Position: lines 613 - 628.
7.18. Function \texttt{free_text}

**Prototype:** void \texttt{free_text} (text* text, int mode);

**Purpose:** free-es all memory occupied by the textlines. If mode == DYNAMIC it free-es also the pointer-array text. (Call with mode == STATIC if not wanted.)

**Parameters:** above.

**Returns:** nothing.

**Position:** lines 642 - 651.

7.18.9 Function \texttt{read_old_log_file}

**Prototype:** int * \texttt{read_old_log_file} (char *log_file_str, int *work_array, measurements* ms, int number_of_measurements);

**Purpose:** reads the old log file (which is the log file of the previous run), to analyze which measurements run before, and which failed. The results are send from process 0 to all others in default_communicator.

**Parameters:** name of the old log file (log_file_str), an integer array (work_array), which will be filled with control-info, the initialized array of all measurements.

**Returns:** pointer to modified work_array.

**Position:** lines 670 - 757.

**Assumes:** skampi.myid and default_communicator set.

7.18.10 Function \texttt{new_name}

**Prototype:** char * \texttt{new_name} (char *name);

**Purpose:** returns a new output file name which is name.<number> with number high enough, the the returned name is new.

**Parameters:** the name which to append with the number.

**Returns:** a pointer to the new name.

**Position:** lines 771 - 778.
7.18.11 Function number_of_output_files

Prototype: int number_of_output_files (char *name);

Purpose: returns max value for files existing in working directory with <name>.<return_value>
assumed that if <name>.n exists than also <name>.n - 1 exists, unless n = 1

Parameters: above.

Returns: above. (or NULL in case of error.)

Position: lines 793 - 816.

7.18.12 Function output_file_complete

Prototype: int output_file_complete (FILE *file);

Purpose: tests if file is a complete skampi output file.

Parameters: above.

Returns: TRUE iff complete, FALSE otherwise.

Position: lines 828 - 845.

7.18.13 Function output_file_postprocessed

Prototype: int output_file_postprocessed (FILE *file);

Purpose: tests if file is a skampi output file, which was created by postprocessing.

Parameters: above.

Returns: TRUE iff postprocessed, FALSE otherwise.

Position: lines 858 - 873.

7.18.14 Function create_log_file

Prototype: void create_log_file (void);

Purpose: creates log file and renames ectl existing log file of previous run.

Parameters: none.

Returns: nothing.

Position: lines 886 - 929.

Sideeffects: prints error messages in case of trouble.

Assumes: _skampi_myid set.
7.18.15  **Function create_output_file**

**Prototype:** void create_output_file (int *new_run);

**Purpose:** creates new output file and determines if a complete previous run has been performed. (then *new_run is set to TRUE.) The value of new_run is sent to all other processes in default_communicator.

**Parameters:** above.

**Returns:** nothing.

**Position:** lines 943 - 1000.

**Sideeffects:** prints error messages in case of trouble.

**Assumes:** skampi_myid and default_communicator set.

7.18.16  **Function ExtractVersionNumber**

**Prototype:** char *ExtractVersionNumber (char *PtrTarget, char *PtrSource);

**Purpose:** extracts from the RCS id string the version number

**Parameters:** Pointer to the target string and pointer to rcstring

**Returns:** Pointer to target string

**Position:** lines 1013 - 1023.

**Sideeffects:** writes to target pointer the version number

7.18.17  **Function write_head_of_outfile**

**Prototype:** void write_head_of_outfile (FILE **file, char *no_runs);

**Purpose:** writes the head of the outfile (i.e. the machine, node, network, user and absolute -section of the params file.

**Parameters:** a pointer to a filehandle, and a string (no_runs), which can contain the number of skampi-runs used for postprocessing. (This option is only used by postprocessing.)

**Returns:** nothing.

**Position:** lines 1039 - 1110.

**Sideeffects:** manipulates file (sets the filepointer ahead).

**Assumes:** numprocs set
7.18.18 Function linear_interpolate
Prototype: double linear_interpolate (double arg_inter, double arg1, double arg2, double res1, double res2);
Purpose: interpolates a double at value between (arg1,res1) and (arg2,res2).
Parameters: above.
Returns: pointer to the interpolated data-struct.
Position: lines 1123 - 1129.
Sideeffects: allocates new data element if data == NULL.

7.18.19 Function double clock_resolution
Prototype: double clock_resolution(void);
Purpose: determines the system's clock accessible resolution
Parameters: none
Returns: resolution
Position: lines 1140 - 1160.
Sideeffects: none
Assumes: nothing

7.18.20 Function init_skalib
Prototype: */void init_skalib (measurement* measurements_array, int *work_array,
int *number_of_measurements);
Purpose: in the parallel case: initializes global variables numprocs, _skampi_nid,
default_communicator, processor_name in all cases. Fills global variables repetitions, out_file_name, log_file_name, old_log_file_name. Reads PARAMETER_FILE, the old log file, creates the output file and the new log file. Sets the pointers (!) measurements_array and work_array (There given as **). measurements_array is an array of all suites of measurements described in the parameter file. Suite i is described in a measurements_i struct (*measurements_array)[i]. If suite i has to be performed (*work_array)[i] is set to TODO, else to SKIP. *number_of_measurements is set to the size of these arrays. Also *new_run is set to TRUE iff this run is not a continuation of an aborted run. Allocates memory (also calls allocate_mem).
Parameters: above.

Returns: nothing.

Position: lines 1190 - 1322.

Sideeffects: aborts in case of errors (IO or memory) prints messages to stdout
   (if define O(A) A).

Assumes: mentioned global variables defined, MPI_Init called

7.18.21 Function perform_measurements

Prototype:

Purpose: performs measurement stored in measurements_array (with its tag in
   work_array set to TODO. Writes results in outfile and remarks in logfile.
   number_of_measurements is the size of these arrays.

Parameters: above.

Returns: nothing.

Position: lines 1338 - 1490.

Sideeffects: aborts in case of errors (IO or memory) prints messages to stdout
   (if define O(A) A).

Assumes: init_skalib called or equivalent operations performed
Appendix A

Derivation of the formula used to calculate the standard error

To show:

\[ \sigma_x = \sqrt{\frac{\sum_{i=1}^{n} x_i^2 - \frac{1}{n} (\sum_{i=1}^{n} x_i)^2}{n \cdot (n - 1)}} \quad (A.1) \]

From the definition we know:

\[ \sigma_x := \frac{\sigma}{\sqrt{n}} \quad (A.2) \]

where \( \sigma := \sqrt{\frac{\sum_{i=1}^{n} (x - \bar{x})^2}{n-1}} \). Put in (A.2), we yield

\[ \sigma_x := \sqrt{\frac{\sum_{i=1}^{n} (\bar{x} - x_i)^2}{n \cdot (n - 1)}} \quad (A.3) \]

Comparing the right hand sides of (A.1) and (A.3) we see, that we have to show the following equation (for sake of readability we omit the indices of the sums, since they are not manipulated in the following).

\[ \sum x_i^2 - \frac{1}{n} \cdot (\sum x_i)^2 = \sum (\bar{x} - x_i)^2 \quad (A.4) \]

\[ \iff \sum x_i^2 - \frac{1}{n} \cdot (\sum x_i)^2 = \sum \bar{x}^2 - 2\bar{x} \sum x_i + \sum x_i^2 \quad (A.5) \]

\[ \iff -\frac{1}{n} \cdot (\sum x_i)^2 = \sum \bar{x}^2 - 2\bar{x} \sum x_i \quad (A.6) \]
To see this, we transform:

\[-\frac{1}{n} \cdot (\sum x_i)^2 = \frac{1}{n} \cdot (\sum x_i)^2 - \frac{2}{n} \cdot (\sum x_i)^2\]

\[= n \cdot \frac{1}{n^2} \cdot (\sum x_i)^2 - 2 \cdot \frac{\sum x_i}{n} \cdot \sum x_i\]

\[= n \cdot (\sum \frac{x_i}{n})^2 - 2 \cdot \bar{x} \cdot \sum x_i\]

\[= n \cdot \bar{x}^2 - 2 \cdot \bar{x} \cdot \sum x_i\]

\[= \sum_{i=1}^{n} \bar{x}^2 - 2 \cdot \bar{x} \cdot \sum x_i\]

q.e.d.
Bibliography


Index

Internal variables
  _skampi_buffer, 32–34
  _skampi_buffer_2, 33, 34
  _skib, 34
  _skib_size, 34

A
  act_time_suite, 22
  add, 47
  all_finished, 69
  allocate_mem, 53
  allocate_memory, 34
  am_control_end, 8, 9, 40
  am_fill_data, 8, 41
  am_free, 40
  am_init, 9, 40
  AMR, 14
  APR, 11
  ASEC, 8
  autodist, 38
  autodist.c, 14
  automeasure, 39
  automeasure.c, 8
  average, 51
  average_of_lists, 51

C
  calculate_key, 39
  call_length, 36, 37
  call_nodes, 36
  client_, 45
  col, 41
  col_init_, 42
  col_pattern, 42
  col_test1, 42

  combine_lists, 15, 69
  constant
    DEF_MESSAGE_LEN, 32
    DYN_LEN, 22
    DYN_LOG, 22
    FIXED_LEN, 22
    FIXED_LOG, 22
    MEASURE_MIN, 33
    TEXT_LINES, 29
  corrected_line, 30
  counter, 9
  create_log_file, 74
  create_output_file, 75
  cut_quartile, 23
  cut_quartile, 10

D
  data, 24
  data_structures, 21
  data_cmp, 70
data_list, 47
datastructure
  measurement_t, 9
  text_t, 29
datastructure
  keywords, 37
  measurement_struct, 8
  measurement_t, 10, 21, 32
  mem_init_one_buffer, 34
  params_t, 28
  DEF_MESSAGE_LEN, 32
  double_clock_resolution, 76
double_cmp, 41
  DYN_LEN, 22
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocate_memory</td>
<td>34</td>
</tr>
<tr>
<td>am_control_end</td>
<td>8, 9</td>
</tr>
<tr>
<td>am_fill_data</td>
<td>8</td>
</tr>
<tr>
<td>am_init</td>
<td>9</td>
</tr>
<tr>
<td>call_length</td>
<td>36, 37</td>
</tr>
<tr>
<td>call_nodes</td>
<td>36</td>
</tr>
<tr>
<td>combine_lists</td>
<td>15</td>
</tr>
<tr>
<td>find_meas</td>
<td>15, 68</td>
</tr>
<tr>
<td>find_mxml</td>
<td>55</td>
</tr>
<tr>
<td>fill_dummy_values</td>
<td>36</td>
</tr>
<tr>
<td>float_or_default</td>
<td>65</td>
</tr>
<tr>
<td>float_or_default_or_invalid</td>
<td>65</td>
</tr>
<tr>
<td>free_all_lists</td>
<td>67</td>
</tr>
<tr>
<td>free_data_list</td>
<td>50</td>
</tr>
<tr>
<td>free_mem</td>
<td>53</td>
</tr>
<tr>
<td>free_post_proc</td>
<td>67</td>
</tr>
<tr>
<td>free_text</td>
<td>73</td>
</tr>
<tr>
<td>function</td>
<td></td>
</tr>
<tr>
<td>initialize_type</td>
<td>33, 66</td>
</tr>
<tr>
<td>insert</td>
<td>62</td>
</tr>
<tr>
<td>insert_in_text</td>
<td>30, 72</td>
</tr>
<tr>
<td>insert_postProc</td>
<td>15, 67</td>
</tr>
<tr>
<td>init</td>
<td>47</td>
</tr>
<tr>
<td>init_postProc</td>
<td>29, 59</td>
</tr>
<tr>
<td>init_postProc</td>
<td>15, 67</td>
</tr>
<tr>
<td>init_skalib</td>
<td>76</td>
</tr>
<tr>
<td>init_symboltable</td>
<td>61</td>
</tr>
<tr>
<td>is_end</td>
<td>49</td>
</tr>
<tr>
<td>is_start</td>
<td>49</td>
</tr>
<tr>
<td>measure</td>
<td>36</td>
</tr>
<tr>
<td>measurement_data_to_string</td>
<td>36</td>
</tr>
<tr>
<td>mem_init_buffer</td>
<td>32, 34</td>
</tr>
<tr>
<td>mem_release</td>
<td>33</td>
</tr>
<tr>
<td>parse_parameter_file</td>
<td>30</td>
</tr>
<tr>
<td>post_process</td>
<td>14, 15</td>
</tr>
<tr>
<td>post_processing</td>
<td></td>
</tr>
<tr>
<td>read_from_file</td>
<td>16</td>
</tr>
<tr>
<td>read_header</td>
<td>16</td>
</tr>
<tr>
<td>read_one_meas</td>
<td>15</td>
</tr>
<tr>
<td>read_parameters</td>
<td>30</td>
</tr>
<tr>
<td>receive_text</td>
<td>30</td>
</tr>
<tr>
<td>send_text</td>
<td>30</td>
</tr>
<tr>
<td>skip_to_next_meas</td>
<td>15</td>
</tr>
<tr>
<td>write_head_of_outfile</td>
<td>16, 31</td>
</tr>
<tr>
<td>write_to_file</td>
<td>16</td>
</tr>
</tbody>
</table>

**INDEX**

<table>
<thead>
<tr>
<th>E</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Enhancements</td>
<td>27</td>
</tr>
<tr>
<td>ExtractVersionNumber</td>
<td>75</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>automeasure.c</td>
<td>8</td>
</tr>
<tr>
<td>p2p_test1.c</td>
<td>32</td>
</tr>
<tr>
<td>p2p_test1.h</td>
<td>33</td>
</tr>
<tr>
<td>sk21f</td>
<td>36</td>
</tr>
<tr>
<td>skalib_const.h</td>
<td>32</td>
</tr>
<tr>
<td>skalmpi.c</td>
<td>36</td>
</tr>
<tr>
<td>skalmpi.h</td>
<td>21, 35</td>
</tr>
<tr>
<td>skalmpi_call.c</td>
<td>37</td>
</tr>
<tr>
<td>skalmpi_mem</td>
<td>33</td>
</tr>
<tr>
<td>skalmpi_params.c</td>
<td>29, 33, 37</td>
</tr>
<tr>
<td>skalmpi_params.h</td>
<td>28</td>
</tr>
<tr>
<td>skalmpi_tools.c</td>
<td>31, 36</td>
</tr>
<tr>
<td>skalmpi_tools.h</td>
<td>29</td>
</tr>
<tr>
<td>skosfile.c</td>
<td>27</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Index of functions</td>
<td>38</td>
</tr>
<tr>
<td>init_list</td>
<td>47</td>
</tr>
<tr>
<td>init_params</td>
<td>29, 59</td>
</tr>
<tr>
<td>init_postProc</td>
<td>15, 67</td>
</tr>
<tr>
<td>init_skalib</td>
<td>76</td>
</tr>
<tr>
<td>init_symboltable</td>
<td>61</td>
</tr>
<tr>
<td>initialize_type</td>
<td>33, 66</td>
</tr>
<tr>
<td>insert</td>
<td>62</td>
</tr>
<tr>
<td>insert_in_text</td>
<td>30, 72</td>
</tr>
<tr>
<td>insert_postProc</td>
<td>64</td>
</tr>
<tr>
<td>insert_float</td>
<td>64</td>
</tr>
<tr>
<td>insert_max</td>
<td>64</td>
</tr>
<tr>
<td>interpolate_data</td>
<td>15, 70</td>
</tr>
<tr>
<td>is_end</td>
<td>49</td>
</tr>
<tr>
<td>is_start</td>
<td>49</td>
</tr>
</tbody>
</table>
INDEX

item_addr, 48
item_addr_at_item, 48

K
keywords, 37

L
lexan, 62
line, 30
line_mode, 30, 59
linear_interpolate, 76
lookup, 61

M
main, 38
master_receive_ready_test, 43
match, 62
max_rep, 9, 23
maximum, 50
mean_value_all, 9
measure, 36
measure..., 42, 46
MEASURE_MIN, 33
measure_suite, 14, 32, 39
measurement, 3, 63

single, 2
measurement_data_to_string, 37, 71
measurement_struct, 8
measurement..., 9, 10, 21, 32
measurement->pattern, 35
measurements

suite of, 3
measurements..., 11
mem_init_col_Waitall, 58
mem_init_nw_Waitany, 57
mem_init_nw_Waitsome, 57
mem_init_one_buffer, 32, 34, 53
mem_init_two_buffers, 54
mem_init_two_buffers_alltoall, 54
mem_init_two_buffers_attach, 55
mem_init_two_buffers_attach_nw, 56
mem_init_two_buffers_attach_p2p, 56
mem_init_two_buffers_gather, 54
mem_release, 33, 58
mem_release_detach, 56
merging_results, 14
min_rep, 9, 23
minimum, 50
module

skampi_post.c, 15
skampi_tools, 16
ms->data.p2p_plata_len, 32
msglen, 32
multiple_of, 22
mw, 43
mw_init..., 43
mw_pattern, 43
mw_test1, 43

N
name, 21
new_name, 73
nif, 15
node_times, 23
nom, 34
nor, 34
number_of_elements, 49
number_of_output_files, 74

O
output_error, 52
output_file_complete, 74
output_file_postprocessed, 74

P
p2p, 44
p2p_find_max_min, 44
p2p_init..., 45
p2p_pattern, 44
p2p_test1, 44
p2p_test1.c, 32
p2p_test1.h, 33
parameter

refinement of (APR), 11
params..., 28
parse, 63
parse parameter file, 30, 59
pattern, 3, 21
definition, 17
perform measurements, 77
post process, 14, 70
post processing, 15, 66

R
read_all_lists of next meas, 69
read file, 47
read from file, 16, 52
read from text, 72
read header, 16, 71
read item file, 48
read next char, 60
read old log file, 73
read one list of meas, 68
read one meas, 15
read parameters, 30, 58
receive text, 30
recv_text, 60
remove file, 49
result list, 23
result sum all, 9
results
merging, 14
routine to be measured, 17
run, 3

S
scale style, 64
send text, 30, 60
server__, 45
simple, 45
simple init__, 46
simple pattern, 46
simple test1, 46
single measurement, 2
sk21f, 36
skalib, 1
skalib const.h, 32
skampi, 38
skampi.c, 36
skampi.h, 35
skampi_all.c, 37
skampi error, 52
skampi mem, 33, 52
skampi params, 58
skampi params.c, 29, 33, 37
skampi params.h, 28
skampi post, 66
skampi post.c, 15
skampi tools, 16, 70
skampi tools.c, 31, 36
skampi tools.h, 29
skip to next meas, 15, 68
skos file, 27
skos file.c, 27
square result sum all, 9
standard error, 8
standard error, 8
standard error... given at the beginning of function, 40
step width, 11
suite of measurements, 3

T
tbm buffer, 10
TEXT LINES, 29
text lines, 29
time meas, 9, 23
time suite, 22
token to str, 66

U
unread next char, 61

V
variable
  _skampi buffer, 32-34
  _skampi buffer2, 33, 34
  _skib, 34
  _skib size, 34
  act time suite, 22
  autolist.c, 14
INDEX

corrected_line, 30
counter, 9
cut_quartile, 23
cut_quartile, 10
data, 24
init_params, 29
line, 30
max_rep, 9, 23
mean_value_all, 9
measurement->pattern, 35
measurements, 11
min_rep, 9, 23
ms->data.p2p.data.len, 32
msglen, 32
multiple_of, 22
name, 21
nif, 15
node_times, 23
nom, 34
nor, 34
pattern, 21
result_list, 23
result_sum_all, 9
square_result_sum_all, 9
standard_error, 8
stepwidth, 11
tbm_buffer, 10
time_meas, 9, 23
time_suite, 22
variation, 22
which_to_measure, 33
x_end, 21
x_min_list, 12, 22
x_scale, 12, 22
x_start, 21
x_stepwidth, 21

write_header, 72
write_header_outfile, 16, 31, 75
write_header, 72
write_text_profile, 72
write_to_file, 16, 51
write_to_logfile, 71

X
x_end, 21
x_min_list, 12, 22
x_scale, 12, 22
x_start, 21
x_stepwidth, 21

Y
yes_or_no, 65

W
which_to_measure, 33