SKaMPI: The Special Karlsruher
MPI-Benchmark
User Manual ¹

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

January 13, 1999

¹This document appeared as Interner Bericht (Technical Report) 99/02 at the Department of Informatics, University of Karlsruhe, Germany
SKaMPI is the Special Karlsruher MPI-Benchmark. SKaMPI measures the performance of MPI [3][1] implementations, and of course of the underlying hardware. It performs various measurements of several MPI functions. SKaMPI’s primary goal is giving support to software developers. The knowledge of MPI function’s performance has several benefits: The software developer knows the right way of implementing a program for a given machine, without (or with shortening) the tedious time costly tuning, which usually has to take place. The developer has not to wait until the code is written, performance issues can also be considered during the design stage. Developing for performance even can take place, also if the considered target machine is not accessible.

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

1 Running SKaMPI 2
  1.1 Introduction ................................................. 2
  1.2 Installation ..................................................... 3
    1.2.1 Getting SKaMPI ........................................ 3
    1.2.2 Compiling SKaMPI ........................................ 4
  1.3 Running SKaMPI ..................................................... 4
  1.4 Post-processing .................................................. 5
  1.5 Generating a report ............................................. 6
  1.6 The measurements: A short overview .......................... 7
    1.6.1 Ping-pong tests ........................................... 7
    1.6.2 Measurements with the master worker scheme .......... 8
    1.6.3 Collective Operations ..................................... 10
    1.6.4 Local Operations .......................................... 16

2 Customizing and trouble-shooting 18
  2.1 Configuring SKaMPI- The parameter file ..................... 19
    2.1.1 The sections .............................................. 19
    2.1.2 Example and default values ............................... 21
    2.1.3 Grammar for sections ..................................... 22
    2.1.4 The MEASUREMENTS-section ............................... 23
    2.1.5 Example of an entry ..................................... 26
    2.1.6 A Note to the preference of the parameters Max_Steps, Time_Suite and Standard_error, Time_Measurement .. 26
    2.1.7 Grammar of the MEASUREMENTS-Section ................. 27
  2.2 Configuring the report generator ............................. 29
    2.2.1 Comparisons .............................................. 29
    2.2.2 Additional tex-modules .................................. 30
    2.2.3 More detailed graphs .................................... 30
    2.2.4 Given module files ...................................... 31
    2.2.5 Extra text for suites .................................... 31
  2.3 When SKaMPI crashes .......................................... 31
3 Measurements in detail

3.1 But what is measured? ........................................... 34
  3.1.1 Example ......................................................... 35
  3.1.2 Point-to-Point pattern .......................... 37
  3.1.3 Master-Worker pattern ....................... 37
  3.1.4 Collective pattern .......................... 39
  3.1.5 Simple pattern ............................................. 39

3.2 The call-back functions ........................................... 40
  3.2.1 Call-backs of the Point-to-Point pattern .... 40
  3.2.2 Call-backs of the Master-Worker pattern .... 50
  3.2.3 Call-backs of the Collective pattern ........ 60
  3.2.4 Call-backs of the Simple pattern .......... 70

3.3 The output file .................................................... 72
Acknowledgements

This technical report mainly offsprings from my diploma thesis [2]. I would like to express my gratitude to my advisers P. Sanders and I. Prechelt. Especially the algorithm for automatic parameter refinement is based on ideas of P. Sanders. I would like to thank for many fruitful discussions.
Chapter 1

Running SKaMPI

1.1 Introduction

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

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

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

Second, if the programmer knows the MPI function's performance on several machines, the programs can be developed for performance for all considered target platforms. So we can speak of a performance portability, instead of compile portability. Compile portability means that a parallel program, developed and tuned on platform A, is recompiled on platform B, and has to be tuned for platform B. So this is not what we really understand under portability. Unlike compile portability, performance portability means that a program is developed with MPI function's performance on all targeted platforms in mind, so that you really just have to recompile.

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

1.2 Installation

1.2.1 Getting SKaMPI

The easiest way to obtain the SKaMPI-Packet is to load it down from the SKaMPI-homepage: http://www.ipd.ira.uka.de/skampi/ The SKaMPI-file you find there is a gnu-zipped tar-file. Thus you can unpack it with tar -xvzf skampi.tgz

However, this will create the whole directory-tree of SKaMPI:

/skampi
/skampi/report_generator

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

\footnote{If your version of tar has no option z, you can call gnu-unzip first (gunzip skampi.tgz) and then tar (tar -xvf skampi.tar)}
1.2.2 Compiling SKaMPI

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

Please compile the program `pposf.c` in the same manner. This program is only used for post processing the results. This will be explained in Section 1.4.

1.3 Running SKaMPI

Unfortunately starting an MPI program is as dependent on your system as compiling. Usually you can start MPI programs with the `mpirun`-command, but there is no standard for its parameters. Using mpich you start the benchmark with `mpirun -np 16 skosfile` with 16 processors. Note: Some systems like the IBM SP have a different command for starting parallel programs (`poe`) than `mpirun`. In case of trouble, you may ask your local administrator.

SKaMPI wants to be started with two or more processors. How many you use, depends on what you want to measure.\(^4\) Some operating environments request further information on the program to start, such as memory or time requirements. The memory that SKaMPI needs depends on what is given in the @MEMORY-section in the parameter file (.skampi). (Please see section 2.1 for further information about the parameter file.) As rule of the thumb you should give a megabyte extra, for internal buffers, etc. The time that SKaMPI needs to measure depends on the accuracy you request, and the number of measurements you asked SKaMPI to perform.\(^5\) To say a typical value: SKaMPI runs with

\(^{2}\)skampi-in-onemodule
\(^{3}\)During development we use several modules, which are merged together to skosfile.c. This eases distribution, versioning, and compiling and on the target platforms. If you are interested in reusing the code, please send an email to obtain the modules, which probably eases understanding of the code.
\(^{4}\)Well, you may ask, what is measured. For a quick overview please have a look in the example-report `skarep.example.p` in the Section 1.6. A more detailed technical description you will find in the Section 3.1.
\(^{5}\)You can change them in the @STANDARDERR- and @MEASUREMENTS-section respectively. You also can give a time limit for measurements through the sections @TIMESUITE_DEFAULT and @TIMEDEFAULT. (For further information please see Section 2.1.4.)
all measurements and an accuracy of 3 percent less than half an hour on an IBM SP using 16 nodes using an 8 MB message buffer. SKaMPI stores its results in a text file. The name of this text file is `skampi.out` by default. To change that edit the `@OUTFILE`-section in the parameter file (see 2.1.1). While other processes running during measuring, their load may disturb SKaMPI. So you might find it useful running SKaMPI more than once. For every run SKaMPI creates a new output file `skampi.out.1`, `skampi.out.2` and so on. Note that the results of the actual run are always stored in `skampi.out`. The other file SKaMPI creates is a log file (`skampi.log`). It is used by the recovery-mechanism. But you may also have a look into. Several warnings and comments are stored in it.

Before starting the Benchmark we urgently recommend to fill out the `@MACHINE`, `@NODE` and `@NETWORK` sections of the parameter file `.skampi` in a detailed manner.

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

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

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

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

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

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

### 1.4 Post-processing

Since we may have more than one output file, we would like to merge all these files together, so that all measurements performed are used. The post-processing

---

6Its name can be changed in the `@LOGFILE` section of the parameter file.
does exactly this. It reads all output files and creates a new one (concrete: a new `skampi.out`). This new file is used for storing the medians of all other corresponding measurements.

If you do not want SKaMPI to perform the post-processing, you just have to write `@POSTPROCESSING no` (instead of `yes`) in the parameter file. Then you can call the post-processing manually: `post`.

### 1.5 Generating a report

Since we run SKaMPI, we would like to know its results. Let’s assume that the results are stored in `skampi.out`, which is the default. Then we just call `dorep.pl` to create a postscript report named `skampi.out.ps`.

Just call `dorep.pl othername` if your output file is not named `skampi.out` but “`othername`”. In this case, the result will be stored in `othername.ps`.

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

`dorep.pl` needs several programs to work.

<table>
<thead>
<tr>
<th>Program</th>
<th>my Version</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>perl</td>
<td>version 5.003</td>
<td>interpreting and execution</td>
</tr>
<tr>
<td>gnuplot</td>
<td>version 3.5, patchlevel 3.50.1.17, 27 Aug 93</td>
<td>Generating eps-graphics</td>
</tr>
<tr>
<td>latex</td>
<td>Version 3.14159 (C version 6.1)</td>
<td>Text formatting</td>
</tr>
<tr>
<td>dvips</td>
<td>dvipsk 5.58f</td>
<td>Converting <code>.dvi</code>-files into <code>.ps</code>-files.</td>
</tr>
</tbody>
</table>

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

---

[7] Further on let’s say, that if we had several runs of SKaMPI, we would have called the post-processing.

[8] The real Perl-freak knows: the is a solution for this problem, a magic line, which forces the shell to search for Perl. But it does not works, when using the C-shell. (So we forget it.)
1.6 The measurements: A short overview

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

1.6.1 Ping-pong tests

In a ping-pong test one node sends a message to another, which replies it. We can use for these point-to-point communication different MPI operations.

All ping-pong measurements are varied over the message length.

MPI_Send-MPI_Recv

This is the “standard”-ping-pong test. A message is send with MPI_Send from a node to another receiving with MPI_Recv. The receiving nodes replies also with MPI_Send. As result the bandwidth of a node is given. That is incoming bandwidth plus outgoing bandwidth.

This measurement serves as reference for all other ping-pong measurements.

MPI_Send-MPI_Iprobe_Recv

This ping-pong test waits busily via calling MPI_Iprobe before calling MPI_Recv at the sending and receiving node. It differs in no way else from the standard ping-pong.

MPI_Send-MPI_Irecv

Here we replace the MPI_Recv of the standard ping-pong test with a combined MPI_Irecv and MPI_Wait. The idea is to see possible advantages of the non-blocking version.

MPI_Send-MPI_Recv_with.Any_Tag

This measurement is just the standard ping-pong test. It only differs in receiving without a specified tag. Here we use the tag MPI_ANY_TAG to determine whether this is more expensive or not.

MPI_Ssend-MPI_Recv

In this measurement we use MPI_Ssend for sending and MPI_Recv for receiving. Here we can fix the overhead of synchronous sends.
**MPI_Send-MPI_Recv**

Now we use `MPI_Send` for sending and `MPI_Recv` for receiving. After the non-blocking send we use an `MPI_Wait`. So we can determine the advantage of non-blocking sends combined with Waits.

**MPI_Isend-MPI_Recv**

Now we use `MPI_Isend` for sending and `MPI_Recv` for receiving. After the non-blocking send we use an `MPI_Wait`. So we can determine the advantage or cost of non-blocking synchronizing sends combined with Waits. Also comparisons to `MPI_Isend` are interesting.

**MPI_Bsend-MPI_Recv**

In this measurement we use `MPI_Bsend` for sending and `MPI_Recv` for receiving. Here we can fix the overhead of managing user-defined buffers.

**MPI_Sendrecv**

In this measurement we use `MPI_Sendrecv` for sending and receiving at the sender and the receiver. This can be compared with the standard-ping-pong test and with the following test of `MPI_Sendrecv_replace`.

**MPI_Sendrecv_replace**

In this measurement we use `MPI_Sendrecv_replace` for sending and receiving at the sender and the receiver. This can be compared with the standard-ping-pong test and with the previous test of `MPI_Sendrecv`.

### 1.6.2 Measurements with the master worker scheme

The following measurements correspond to the master worker scheme. The master dispatches suborders to several workers. These workers send a reply for every received order. With this way we try to measure the network throughput and how it can handle simultaneous communication.

This kind of measurements can be varied over the number of suborders (`chunks`), the length of the messages sent or the number of workers.

We display the bandwidth reached at the master node.

**MPI_Waitsome-nodes**

In this measurement we use the `MPI_Waitsome` routine to coordinate the incoming worker messages. This function guarantees a *fair* coordination of the workers, because messages of every sending worker will be received. Here the measurements are varied over the number of workers.
1.6. THE MEASUREMENTS: A SHORT OVERVIEW

MPI\_Waitsome-chunks
This is the same measurement as above, but now we vary it over the number of chunks.

MPI\_Waitsome-length
This is the same measurement as above, but now it is varied over the message length.

MPI\_Waitany-length
In this measurement we use the MPI\_Waitany-routine to coordinate the incoming worker messages. This function does not guarantee a fair coordination of the workers, because possibly a worker’s messages are always overtaken by the messages of its colleagues. But because of its simplicity it may be faster than the MPI\_Waitsome.

We vary over the message length.

MPI\_Recv\_Any\_Source-length
In this measurement the master receives the messages of the workers via MPI\_Recv using the MPI\_ANY\_SOURCE as source. Thus this is a master-worker scheme only realized with point-to-point communication operations. For sending MPI\_Send is used.

Here we vary over the message length.

MPI\_Send-length
Here the master uses MPI\_Send for sending and MPI\_Recv for receiving. But contrary to the measurement above, the source is specified in the call of MPI\_Recv. This measurement serves as reference for the following three measurements. But you also can compare it with the measurement above.

It is varied over the message length.

MPI\_Ssend-length
This measurement only differs in using MPI\_Ssend instead of MPI\_Send. It shows the extra costs of the synchronous sending.

MPI\_Isend-length
This measurement only differs in using MPI\_Isend instead of MPI\_Ssend. The non-blocking sending will be faster than the blocking variants, if the network allows.
MPI\_Bsend-length
This measurement only differs in using MPI\_Bsend instead of MPI\_Send. We can see the costs of extra buffer handling to MPI\_Send.

1.6.3 Collective Operations

The following measurements concern collective MPI operations. These operations synchronize processes MPI\_Barrier or transmit data between them. The time until completion on all nodes is measured. In all cases the result is the bandwidth at one node.

MPI\_Bcast-nodes-short
Here we test the MPI\_Bcast operation with short messages (256 Bytes). We vary over the number of processes. The results are compared with the results of the following measurement.

MPI\_Bcast-nodes-long
Now we test the MPI\_Bcast operation with long messages (64 KBytes). We vary over the number of processes.

MPI\_Bcast-length
This measurement also tests the Broadcast operation. But here we vary over the message length. The number of the participating nodes is fixed.

MPI\_Barrier-nodes
This test synchronizes several processes via MPI\_Barrier. This measurement is interesting because this operation usually is called very often. We vary over the number of nodes. (Since there are no messages sent, we cannot vary over message length.)

MPI\_Reduce-nodes
Here me measure the time MPI\_Reduce consumes. This operation performs a tree-wise data reduction operation (here: bit-wise or) on all participating processes. The result is stored at a root node. We vary over the number of nodes.

MPI\_Reduce-length
This measurement is the same like the one above. But now we vary over the message length.
MPI_Scan-nodes

The MPI_Scan operation performs a prefix reduction on data distributed across the participating processes. First we vary over the nodes. This measurement can be compared with MPI_Reduce.

MPI_Scan-length

This is the measurement described above. Now it is varied over the message length.

MPI_Alltoall-nodes-short

The MPI_Alltoall operation sends a message from every node to every node. We vary over the number of nodes. The messages have the length of 256 Bytes (for each node).

MPI_Alltoall-nodes-long

This measurement is similar to the above. But now the messages have the length of 64 KBytes (for each node).

MPI_Alltoall-length

This is the same measurement as above, only that we vary over the message length.

MPI_Gather-nodes-short

Using the MPI_Gather operation a root process collects data distributed on several nodes and writes the received data in one contiguous buffer. We vary over the number of nodes buffer. The messages have the length of 256 Bytes (for each node).

MPI_Gather-nodes-long

Here we also measure the MPI_Gather operation varied over the number of nodes. But in this case the messages have the length of 64 KBytes (for each node).

MPI_Gather-length

Here we measure MPI_Gather varied over the message length.
MPI_Gather_SR-nodes-short
Using a Gather operation a root process collects data distributed on several nodes and writes the received data in one contiguous buffer. Here we implemented this operation with MPI_Send and MPI_Recv. It is interesting to compare this implementation with the MPI implemented MPI_Gather or our other implementation of gather (MPI_Gather_ISWA). We vary over the number of nodes. The messages have the length of 256 Bytes (for each node).

MPI_Gather_SR-nodes-long
Here we also measure the Gather operation implemented with MPI_Send and MPI_Recv varied over the number of nodes. But in this case the messages have the length of 64 KBytes (for each node).

MPI_Gather_SR-length
Here we measure our MPI_Send - MPI_Recv implementation of Gather varied over the message length.

MPI_Gather_ISWA-nodes-short
Using a Gather operation a root process collects data distributed on several nodes and writes the received data in one contiguous buffer. Here we implemented this operation with MPI_Isend and MPI_Waitall. It is interesting to compare this implementation with the MPI implemented MPI_Gather or our other implementation of gather (Send-Receive). We vary over the number of nodes. The messages have the length of 256 Bytes (for each node).

MPI_Gather_ISWA-nodes-long
Here we also measure the Gather operation implemented with MPI_Isend and MPI_Waitall varied over the number of nodes. But in this case the messages have the length of 64 KBytes (for each node).

MPI_Gather_ISWA-length
Here we measure our MPI_Isend - MPI_Waitall implementation of Gather varied over the message length.

MPI_Scatter-nodes-short
In the MPI_Scatter operation a root process distributes data to every node. The messages have the length of 256 Bytes (for each node).
MPI\_Scatter\_nodes-long
Here we also measure MPI\_Scatter varied over the number of nodes, but the messages have the length of 64 KBytes (for each node).

MPI\_Scatter\_length
We measure MPI\_Scatter varied over the message length.

MPI\_Allreduce\_nodes
This operation performs a tree-wise data reduction operation (here: bit-wise or) on all participating processes and distributes the result to all participating nodes. This result distribution to all participating nodes is the difference to the normal MPI\_Reduce operation, where the result is stored in a single root processor. So it is interesting to compare this operation to the normal MPI\_Reduce and to a MPI\_Reduce followed by an MPI\_Bcast operation (our measurement MPI\_Reduce\_Bcast), which also distributes the result to all nodes. We vary over the number of nodes with a message length of 256 Bytes for each node.

MPI\_Allreduce\_length
Here we also measure the performance of MPI\_Allreduce. This time we vary over the message length.

MPI\_Reduce\_Bcast\_nodes
This operation performs a tree-wise data reduction operation (here: bit-wise or) on all participating processes with MPI\_Reduce and then distributes the result to all participating nodes with MPI\_Bcast. This result distribution to all participating nodes is the difference to the normal MPI\_Reduce operation, where the result is stored in a single root processor. So it is interesting to compare this operation to MPI\_Allreduce, which also distributes the result to all nodes in one call. We vary over the number of nodes with a message length of 256 Bytes for each node.

MPI\_Reduce\_Bcast\_length
Here we also measure the performance of MPI\_Reduce followed by MPI\_Bcast. This time we vary over the message length.

MPI\_Reduce\_scatter\_nodes
This operation performs a tree-wise data reduction operation (here: bit-wise or) on all participating processes with MPI\_Reduce\_scatter and then distributes the
CHAPTER 1. RUNNING SKAMPI

result partially to all participating nodes. Every node receives a different part of the result-array. This kind of result distribution to all participating nodes is the difference to the normal MPI_Reduce or MPI_Allreduce operation, where the result is stored in a single root processor or is transferred completely to all nodes. So it is interesting to compare this operation to MPI_Allreduce, which distributes the result to all nodes in one call. MPI_Reduce_scatter can also be compared with MPI_Reduce followed by MPI_Scatterv, which we measure as MPI_Reduce_Scatter. We vary over the number of nodes with a message length of 256 Bytes for each node.

**MPI_Reduce_scatter-length**

Here we also measure the performance of MPI_Reduce_scatter. This time we vary over the message length.

**MPI_Allgather-nodes-short**

The MPI_Allgather operation collects data from every node and concats the received data in one contiguous buffer. In difference to the MPI_Gather operation, all nodes collect the data, not only a root process. We vary over the number of nodes. The messages have the length of 256 Bytes (for each node).

**MPI_Allgather-nodes-long**

Here we also measure the MPI_Allgather operation varied over the number of nodes. But in this case the messages have the length of 64 KBytes (for each node).

**MPI_Allgather-length**

Here we measure MPI_Allgather varied over the message length.

**MPI_Scatterv-nodes-short**

In the MPI_Scatterv operation a root process distributes data to every node. In addition to MPI_Scatter a displacement and length can be given, which determine which data from the root process’ buffer is send to the other nodes. It is interesting to see the extra costs compared to MPI_Scatter. We vary over the number of nodes. The messages have the length of 256 Bytes (for each node).

**MPI_Scatterv-nodes-long**

Here we also measure MPI_Scatterv varied over the number of nodes, but the messages have the length of 64 KBytes (for each node).
MPI\textsubscript{Scatterv-length}

We measure \texttt{MPI\_Scatterv} varied over the message length.

MPI\textsubscript{Gatherv-nodes-short}

In the \texttt{MPI\_Gatherv} operation a root process collects data from every node and concats the received data in one buffer. In addition to the \texttt{MPI\_Gather} operation, we can use per processor receiving from a specific \textit{displacement} and \textit{length}, which determine where to write received data in the root’s buffer and how many bytes to receive from any processor. Of course, it is interesting to see, what are the extra costs of this features. We vary over the number of nodes. The messages have the length of 256 Bytes (for each node).

MPI\textsubscript{Gatherv-nodes-long}

Here we also measure the \texttt{MPI\_Gatherv} operation varied over the number of nodes. But in this case the messages have the length of 64 KBytes (for each node).

MPI\textsubscript{Gatherv-length}

Here we measure \texttt{MPI\_Gatherv} varied over the message length.

MPI\textsubscript{Allgatherv-nodes-short}

The \texttt{MPI\_Allgatherv} operation each process collects data from any other process and concats the received data in one buffer. In addition to the \texttt{MPI\_Allgather} operation, we can use per processor receiving from another processes a specific \textit{displacement} and \textit{length}, which determine where to write received data in the root’s buffer and how many bytes to receive from any processor. Of course, it is interesting to see, what are the extra costs of this features. We vary over the number of nodes. The messages have the length of 256 Bytes (for each node).

MPI\textsubscript{Allgatherv-nodes-long}

Here we also measure the \texttt{MPI\_Allgatherv} operation varied over the number of nodes. But in this case the messages have the length of 64 KBytes (for each node).

MPI\textsubscript{Allgatherv-length}

Here we measure \texttt{MPI\_Allgatherv} varied over the message length.
\textbf{MPI\_Alltoally\_nodes\_short}

The MPI\_Alltoally operation sends a message from \textit{every} node to \textit{every} node. In addition to the “normal” MPI\_Alltoall operation here we able to specify which data from a process’ sending buffer should be send to any other process (send displacement and send lengths) and we can specify where a process’ data received from any other process should be stored (receive displacement and receives lengths). We vary over the number of nodes. The messages have the length of 256 Bytes (for each node).

\textbf{MPI\_Alltoally\_nodes\_long}

This measurement is similar to the above. But now the messages have the length of 64 KBytes (for each node).

\textbf{MPI\_Alltoally\_length}

This is the same measurement as above, only that we vary over the message length.

\textbf{MPI\_Reduce\_Scatterv\_nodes}

This operation performs a tree-wise data reduction operation (here: bit-wise or) on all participating processes with MPI\_Reduce and then distributes the result partially to all participating nodes with MPI\_Scatterv. Every node receives a different part of the result-array. This result kind of distribution to all participating nodes is similar to the one of MPI\_Reduce\_scatter, so it is interesting to compare this operation to MPI\_Reduce\_scatter, which distributes the result to all nodes in one call. We vary over the number of nodes with a message length of 256 Bytes for each node.

\textbf{MPI\_Reduce\_Scatterv\_length}

Here we also measure the performance of MPI\_Reduce\_Scatterv. This time we vary over the message length.

\textbf{MPI\_Commsplit\_nodes}

The MPI\_Commsplit operation splits a given communicator into several others. In this measurement the communicator is divided it into two new ones. This measurement can only be varied over the number of nodes.

1.6.4 Local Operations

The following measurements are \textit{local}. This means that they are executed on only one processor. Also they do not have any parameters.
1.6. THE MEASUREMENTS: A SHORT OVERVIEW

**MPI_Wtime**

This **measurement** should fix the time used for one call of **MPI_Wtime**. This MPI routine is used in the whole benchmark for measuring. The result is a lower bound of our accuracy.

**MPI_Commrank**

This routine is used to get the process-id of the calling process. (This ID corresponds to the used MPI communicator.) The costs of this operation are relevant, because many subroutines have to find out their process-id. Usually this information is not given as a parameter to the subroutine, but the communicator is.

**MPI_Commsize**

This MPI operation gives the number of processes grouped in a communicator. We are interested in its costs because of the same reasons for the operation above.

**MPI_Iprobe**

Many receiving routines test whether a message came in or not using **MPI_Iprobe**. Most calls are not successful in the mean that **MPI_Iprobe** is called, when no message arrived.

Here we fix the costs of an unsuccessful **MPI_Iprobe**.

**simple_dummy**

This measurement determines the overhead of measuring these local operations.
Chapter 2

Customizing SKaMPI and trouble-shooting

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

But before that, lets clear some expressions.

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

Measurement: A measurement is the determination of a value 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 wanted (and an upper and lower bound).

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. (Selection is done in the parameter file.) Usually for each run a report is generated.
2.1 Configuring SKaMPI - The parameter file

2.1.1 The sections

The parameter file is an ASCII-text file describing the settings to control SKaMPI.

The parameter file should be accessible in the directory where SKaMPI is started. Its name is always .skampi. Thus, do not rename it. Here you can see how to adapt the parameter file to your personal needs.

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

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

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

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

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

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

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

@MEMORY This section is just an integer. It describes the amount of memory in KBytes, which should be reserved for message buffers on each node, e.g. @MEMORY 8192 == 8 Megabytes message buffers.
OUTFILE The name of the output file. This name should also be entered in the first line (e.g. OUTFILE skampi.out). Note that there is a blank between OUTFILE and the filename!

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

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

MAXREPDDEFAULT This integer describes the maximal number of measurements repetitions can be performed. This value is the default value for Max_Repetition.

MINREPDDEFAULT This integer describes the minimal number of repetitions a measurement can be performed. This value is the default value for Min_Repetition.

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

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

TIMEMEASDEFAULT This float sets the default value of the parameter Time_Measurement.

CUTQUANTILEDEFAULT This float sets the default value of the parameter Cut_Quantile.

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

ABSOLUTE Please enter just a yes or a no in this section. If “yes”, SKaMPI will try to correct the measured data, that is subtracting the overhead. This option should only be activated, if it is clear that there is low (or better no) other load on the machine. (Otherwise you can get negative performing-times, because the measurement of the overhead can be disturbed by the other load.) E.g. ABSOLUTE yes.
@POSTPROC Please enter just a yes or a no in this section. You can do several runs of SKaMPI. Each successful run will build a new output file (e.g. skampi.out, skampi.out.1, skampi.out.2, ...). If “yes”, SKaMPI will perform the post-processing. That is merging all output files together. Note if SKaMPI is restarted after an abort, no new output file will be created. In this case SKaMPI appends the results to the output file of the previous run. If you do not want SKaMPI to perform the post-processing (@POSTPROC no), because it is not a truly parallel application, and you do not want to waste the time of your supercomputer doing text file manipulations, then you may also call the post-processing separately with post.

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

2.1.2 Example and default values

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

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

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

@MEMORY 4096
@OUTFILE skampi.out
@LOGFILE skampi.log
@MAXSTEPSDEFAULT 16
@MAXREPDDEFAULT 20
@MINREPDDEFAULT 4
@MULTIPLEOFDEFAULT 4
@STANDARDERRORDEFAULT 0.05
@TIMEMEASDEFAULT 0.0
@TIMESUITEDEFAULT 0.0
@COMMENT
To use TIMEMEASDEFAULT and TIMESUITEDEFAULT please replace the 0.0 with your required values and change the "Invalid_Value" in each measurement to "Default_Value".
@CUTQUANTILEDEFAULT 0.25
@ABSOLUTE no
@POSTPROC yes
@MEASUREMENTS

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

2.1.3 Grammar for sections

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

SECTION ::= TEXT_SECTION SECTION
           | INT_SECTION SECTION
           | FLOAT_SECTION SECTION
           | YESNO_SECTION SECTION
           | MEASUREMENTS_SECTION SECTION
           | <epsilon>

TEXT_SECTION ::= @COMMENT text
               | @MACHINE text
               | @NETWORK text
               | @NODE text
               | @USER text
               | @OUTFILE text
               | @LOGFILE text

INT_SECTION ::= @MEMORY int
               | @MAXSTEPDEFALUT int
               | @MAXREPEDEFALUT int
               | @MINREPEDEFALUT int
               | @MULTIPLEOFEDEFALUT int

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

YESNO_SECTION ::= @ABSOLUTE
                | @POSTPROC

Production rules for the nonterminal MEASUREMENTS_SECTION are found in section 2.1.7. The nonterminals int and float are that what you would expect as C-Programmer. text means some\footnote{some is here 1000hex == 4096, defined through the constant TEXT_LINES in skompi_tools.h.} strings.
2.1.4 The MEASUREMENTS-section

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

**Type** Each measurement must have a type assigned. This type (an simple integer) describes the MPI-function and the pattern which should be measured. Tables 3.1 (page 35) shows which number is assigned to which MPI-function.

**Variation** Here you can enter the variable varied. The variables contained by a pattern you can see in Table 2.1.

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

- **Fixed Linear** The arguments begin at Start Argument and end at End Argument. The distance is Stepwidth. Both scales are linear. The variables Max Steps, Min Distance and Max Distance have no meaning.

- **Fixed Log** The arguments are powers of the parameter stepwidth. (stepwidth\(^1\), stepwidth\(^2\), stepwidth\(^3\) ... until End Argument has been reached.) Both axis are logarithmic. The variables Max Steps, Min Distance and Max Distance have no meaning.

- **Dynamic Linear** The arguments begin at Start Argument and end at End Argument. The distance is Stepwidth. After doing the measurements this way, the number Max Steps of measurements is filled up with automatically placed measurements. These measurements are never nearer than Min Distance. Both axes are linear.

- **Dynamic Log** The arguments are powers of the parameter stepwidth. (stepwidth\(^1\), stepwidth\(^2\), stepwidth\(^3\) ... until End Argument has been reached.) After having done measurements this way, the number Max Steps of measurements is filled up with automatically placed measurements. These measurements are never nearer than Min Distance. Both axis are logarithmic.

**Max Repetition** Here you can enter the maximal number of measurement repetitions. If you do not want to change this value in every entry, you just write Default Value instead the number, and the value given in the MAXREPDDEFAULT-Section is used.
Min\textunderscore Repetition Here you can enter the minimal number of repetitions performed for a measurement. If you do not want to change this value in every entry, you just write Default\textunderscore Value instead the number, and the value given in the \texttt{@MINREPDDEFAULT}-section is used.

Multiple\textunderscore of Any argument a measurement is called with has to be a multiple of this integer value. For example "8" might be quite useful to avoid memory alignment effects on 64-bit machines, or 4 for 32-bit systems. This integer’s default value is set in the section \texttt{@MULTIPLEOF}.

Time\textunderscore Suite The value given here sets the time limit for one suite of measurements in minutes. A suite of measurements is a set of measurements, containing measurements varied over some parameters (compare to definition at the beginning of this chapter). This means that no new measurements are started, when the time consumed by the already executed measurements of this suite exceeds this limit time.\footnote{This means that the time of all measurements can be larger than the limit, because the last measurement will not be aborted when exceeding the limit time.} This limit has no influence on other suites. So exceeding this limit time means that only this suite stops measuring. It does not mean, that the whole benchmark is aborted. Information regarding the preference of this parameter and Max\textunderscore Steps is given in subsection 2.1.6. If you do not want to change this value in every entry, you just write Default\textunderscore Value instead the number, and the value given in the \texttt{@TIMESUITEDEFAULT}-section is used. If you do not want to give any time limit at all, please enter Invalid\textunderscore Time instead of a value.

Time\textunderscore Measurement This value gives the time limit for one measurement in minutes. (A measurement is the repetition of several single measurements. Compare to definition at the beginning of this chapter). This means that no new single measurements is started, when the time consumed by the already executed single measurements of this measurement exceeds this limit time.\footnote{This means that the time of all single measurements can be larger than the limit, because the last single measurement will not be aborted when exceeding the limit time.} Information regarding the preference of this parameter and Standard\textunderscore error is given in subsection 2.1.6. If you do not want to change this value in every entry, you just write Default\textunderscore Value instead the number, and the value given in the \texttt{@TIMESUITEDEFAULT}-section is used. If you do not want to give any time limit at all, please enter Invalid\textunderscore Time instead of a value.

Node\textunderscore Times This boolean value can be set to yes or no. In case of yes SKaMPI measures besides the result also the execution times of the mea-
sured routine on all nodes. This may be useful to see, whether overlapping communication and computation can take place, or to measure effects of contention. In the patterns Simple and Master-Worker this feature will be ignored, since in the simple pattern the to be measured routine runs on exactly one processor, and in Master-Worker pattern the workers work until they receive the stop signal. So it is not interesting to measure, when the workers stop.

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

**Cut Quantile** This value defines the upper and lower quantile of single measurements' results, which are disregarded, when computing the result of a measurement. If you do not want to throw any results away, use 0.0. If you assume that the upper an lower quartile of your results are outliers, use 0.25. If you do not want to change this value in every entry, you just write Default Value instead the number, and the value given in the \texttt{CUTQUANTILEDEFAULT}-section is used.

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

**End Argument** This is the maximal argument, which is never exceeded. If you vary over the message length it will depend on the amount of memory you entered in the \texttt{MEMORY}-section. If you vary over the number of nodes, it will depend on the number of nodes, SKaMPI started with. To make it easier to determine these values, you can just enter Max Value here, and SKaMPI computes the actual values during run-time.

**Max Steps** explained under Variation.

**Min Distance** explained under Variation.

**Max Distance** explained under Variation.

**Standard error** Measurements are repeated until its standard error has fallen short of this value here. (But the number of repetitions is never less than Min Repetition and never larger than Max Repetition. The standard

---

4The result is the time the routine to measure needs on the measuring root node. The benchmark assures that the routine to measure has finished on all other nodes, when finished on the root node. So the execution times on the single nodes is usually lower.
CHAPTER 2. CUSTOMIZING AND TROUBLE-SHOOTING

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Variables to vary over</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point-to-Point</td>
<td>Length, Nodes</td>
</tr>
<tr>
<td>Master-Worker</td>
<td>Length, Nodes, Chunks</td>
</tr>
<tr>
<td>Collective</td>
<td>Length, Nodes</td>
</tr>
<tr>
<td>Simple</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 2.1: Which pattern can varied with which variables?

error is a metric for the reliability of a the data, whereas the standard deviation is a metric for dispersion.

2.1.5 Example of an entry

MPI_Send-MPI_Recv
{
    Type = 1;
    Variation = Length;
    Scale = Dynamic_log;
    Max_Repetition = Default_Value;
    Min_Repetition = Default_Value;
    Multiple_of = Default_Value;
    Time_Measurement = Invalid_Value;
    Time_Suite = Invalid_Value;
    Node_Times = No;
    Cut_Quantile = Default_Value;
    Default_Chunks = 0;
    Default_Message_length = 256;
    Start_Argument = 1;
    End_Argument = Max_Value;
    Stepwidth = 128;
    Max_Steps = 30;
    Min_Distance = 128;
    Max_Distance = 512;
    Standard_error = Default_Value;
}

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

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

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

If `Time_Measurement` is set to any other value as `Invalid_Value` (that is a float or `Default_Value`), than no further single measurement will be started, when the sum of the execution times of the already executed single measurements exceeds the value of `Time_Measurement`. The values of `Standard_error`, and `Min_Repetition` will not be regarded in this case. But in any case, there will not be more measurements started than `Max_Repetitions`.\(^5\) If you want to use only `Time_Measurement` to control the termination, so choose a high value for `Max_Steps`.

Termination of a Suite of Measurements

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

If `Time_Suite` is set to any other value as `Invalid_Value` (that is a float or `Default_Value`), than no further measurement will be started, when the sum of the execution times of the already executed measurements exceeds the value of `Time_Suite`.

### 2.1.7 Grammar of the `MEASUREMENTS-Section`

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

```plaintext
MEASUREMENTS_SECTION ::= file_name_str
  `{`
    `Type = "TYPE_RANGE";
    `Variation = "VARIATION_STYLE";
    `Scale = "SCALE_STYLE";
    `Max_Repetition = "INT_OR_DEFAULT";
    `Min_Repetition = "INT_OR_DEFAULT";
    `Multiple_of = "INT_OR_DEFAULT";
    `Time_Measurement = "FLOAT_OR_DEFAULT_OR_INVALID";
    `Time_Suite = "FLOAT_OR_DEFAULT_OR_INVALID";
    `Cut_Quantile = "FLOAT_OR_DEFAULT";
    `Default_Chunks = "INT_OR_FLOAT";
    `Default_Message_length = "INT_OR_FLOAT";
  `}
```

\(^5\) This is because `SKaMPI` uses this values for internal buffer allocation.
file_name_str is what your operating system allows as a file name. In the
grammar above file_name_str stands for the name of the measurement. In the
report generator dorep.pl there will be some files created temporarily, which
contain this string in their names.
As above, the nonterminals int and float are what you would expect as C-
Programmer.

Tip for editing the @MEASUREMENTS-Section: if you want to skip some
measurements, just write @COMMENT before the measurements you intend to skip,
and @MEASUREMENTS behind them.
Table 2.2: The mapping of patterns to prefixes

<table>
<thead>
<tr>
<th>Type numbers</th>
<th>Pattern</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - 9</td>
<td>Point-to-Point</td>
<td>p2p_</td>
</tr>
<tr>
<td>10 - 16</td>
<td>Master-Worker</td>
<td>mw_</td>
</tr>
<tr>
<td>17 - 23</td>
<td>Collective</td>
<td>col_</td>
</tr>
<tr>
<td>24 - 29</td>
<td>Simple</td>
<td>simple_</td>
</tr>
<tr>
<td>29 - 32</td>
<td>internal measurements</td>
<td>-</td>
</tr>
<tr>
<td>33</td>
<td>Collective</td>
<td>col_</td>
</tr>
<tr>
<td>34</td>
<td>Point-to-Point</td>
<td>p2p_</td>
</tr>
<tr>
<td>35 - 46</td>
<td>Collective</td>
<td>col_</td>
</tr>
</tbody>
</table>

2.2 Configuring the report generator

Usually you do not have to adjust dorep.pl. It inspects which measurements are performed and processes them. So if you add or omit measurements, they will appear in (respectively disappear from) the report.

2.2.1 Comparisons

What the generator does not know is, which measurements you want to compare.\(^6\) To manipulate the “Comparisons”-Section in skarep.ps you can edit the .dorep file. This file has a simple structure. Every line describes one comparison. The first part of the line is the name of the comparison. This name may be a normal string, but it must not contain any ”:”, because that is its delimiter. After the “:” follows a list with names of suites of measurements.

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

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

As explained in the section 2.1.1 each suite of measurements has its own name (usually the name of the MPI function measured). It may happen, that one MPI function is used in two (or more) patterns, so you have to add a prefix, describing the pattern.\(^7\)

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

---
\(^6\)Here a comparison is a plot of two or more function graphs. The report generator also creates a table with some results to compare.

\(^7\)The problem of identifying the suite with a name, which may occur twice, does not exist in .skampi. Here the corresponding pattern is stored with the name, so that it is always clear, what suite is called.
1. We want to name our comparison: Comp. `MPI_Send-MPI_Recv` and `MPI_Iprobe` (followed by `MPI_Recv`).

2. In `.skampi` you find the name `MPI_Send-MPI_Recv`. This is the name of one suite we want to see in our comparison. The other suite is called `MPI_Send-MPI_Iprobe_Recv`.

3. Since both suites belong to the point-to-point pattern, table 2.2 tells us we have to add the prefix `p2p`.

4. The resulting line in `.dorep` is:
   
   ```
   Comp. `MPI_Send-MPI_Recv` and `MPI_Iprobe` (followed by `MPI_Recv`):
   p2p `MPI_Send-MPI_Recv`, p2p `MPI_Send-MPI_Iprobe_Recv`.
   Note: this has to be written as one line.
   ```

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

### 2.2.2 Additional tex-modules

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

**Example**

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

### 2.2.3 More detailed graphs

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

```
/*@inp2p_MPI_Bsend-MPI_Recv.ski*/
#Description of the MPI_Bsend-MPI_Recv measurement:
#Pattern: Point-to-Point varied over the message length.
#The x scale is linear, automatical x wide adaption,
#range: 0 - 256, stepwidth: 16.000000.
#default values: 2 nodes.
```
#max. allowed standard error is 10.00 %

Format: message length (%d) <space> time (microsec.)

(\%f) (standard error) (\%f) count (%d)

arg result standard_error count
0 7004.000000 1.000000 2
16 7316.000000 3.000000 2
32 11538.000000 2716.566473 6
40 7498.500000 6.500000 2

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

2.2.4 Given module files

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

2.2.5 Extra text for suites

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

2.3 When SKaMPI crashes.

Since MPI-implementation are no trivial pieces of software, we have to assume that SKaMPI may crash while measuring. In this case all measured suites are stored, only the actual one is lost.

In this case you can use the automatic recovery mechanism. Simply start SKaMPI again. Please do not change the output or log file. SKaMPI tries to find out which measurement caused the trouble. Then SKaMPI skips the measurement and starts with the measurement behind. The erroneous measurement will be called after all others. So if it crashes again, you will have completed all other measurements. This mechanism will also work, if several measurements crash.

If this does not work, you can recover manually.

---

*To see which files are created temporarily by dorep.pl just comment out its line "unlink @files2delete;". Then you may have a look into its files. But be careful: Before the next run of the generator delete these files manually, because the generator does not overwrite them as explained above. (Delete the files: *.tmd *.gpl *.eps.)
*dri means "dorep-information".
*And (err) SKaMPI neither...
1. Find out which measurement caused the crash. In order to do this, look into `skampi.out`, go to the end of file and backward-search the string `"@in"` You will find the name of the last completed measurement after that string.

```
... 
#----------------------------------------
#/0inp2p_MPI_Send-MPI_Recv.ski*/
#Description of the MPI_Send-MPI_Recv measurement:
#Pattern: Point-to-Point varied over the message length.
... 
```

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

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

3. Then find the entry of the crashed measurement. The crashed measurement is the measurement behind the last completed measurement, you know from above. Write `"@MEASUREMENTS"` after the crashed measurement entry. In our case if `MPI_Send-MPI_Recv` is the last completed measurement, then `MPI_Send-MPI_Recv_with:Any_Tag` failed. Therefore we place `"@MEASUREMENTS"` before the next entry (i.e., `MPI_Send-MPI_Recv`).

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

...

4. Delete the current logfile `skampi.log`.
5. Rename `skampi.out` to another file.
6. Start SKaMPI again with the same command.
7. When SKaMPI finished, you can append the new `skampi.out` file to the old renamed one.
Chapter 3

Measurements in detail

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

3.1 But what is measured?

So far we know how to measure, but what is actually measured?
Since we investigate parallel operations, we have to coordinate several processes. Measurements, which have a similar coordination of its processes, are grouped to a so called pattern.
To know, which measurements are performed, when measuring with a certain type, you first should know which pattern and initializer is used in this type.
To do so, have a look in tables 3.3 and 3.1 (page 35).
In the following we will have a look to all four patterns skampi uses. Each pattern calls one or more call-back functions. You can find these functions in the next section. To know, which call-backs you are measuring with a type, simply look at the initializer. They are listed with the call-backs, sorted by patterns.
3.1. BUT WHAT IS MEASURED?

<table>
<thead>
<tr>
<th>Number</th>
<th>MPI-function(s)</th>
<th>Initializer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MPI_Send, MPI_Recv</td>
<td>p2p_init_Send_Recv</td>
</tr>
<tr>
<td>2</td>
<td>MPI_Send, MPI_Recv, MPI_ANY_TAG</td>
<td>p2p_init_Send_Recv_AT</td>
</tr>
<tr>
<td>3</td>
<td>MPI_Send, MPI_RECV</td>
<td>p2p_init_Send_Recv</td>
</tr>
<tr>
<td>4</td>
<td>MPI_Ssend, MPI_RECV</td>
<td>p2p_init_Send_Recv</td>
</tr>
<tr>
<td>5</td>
<td>MPI_Ssend, MPI_RECV</td>
<td>p2p_init_Send_Recv</td>
</tr>
<tr>
<td>6</td>
<td>MPI_Ssend, MPI_RECV</td>
<td>p2p_init_Send_Recv</td>
</tr>
<tr>
<td>7</td>
<td>MPI_Ssend, MPI_RECV</td>
<td>p2p_init_Send_Recv</td>
</tr>
<tr>
<td>8</td>
<td>MPI_Ssend, MPI_RECV_replace</td>
<td>p2p_init_Send_Recv_replace</td>
</tr>
<tr>
<td>9</td>
<td>MPI_Waitsome</td>
<td>mw_init_Waitsome</td>
</tr>
<tr>
<td>10</td>
<td>MPI_Waitany</td>
<td>mw_init_Waitany</td>
</tr>
<tr>
<td>11</td>
<td>MPI_Waitany</td>
<td>mw_init_Waitany</td>
</tr>
<tr>
<td>12</td>
<td>MPI_Recv, MPI_ANY_SOURCE</td>
<td>mw_init_Recev_AS</td>
</tr>
<tr>
<td>13</td>
<td>MPI_Send</td>
<td>mw_init_Send</td>
</tr>
<tr>
<td>14</td>
<td>MPI_Ssend, MPI_Ssend</td>
<td>mw_init_Ssend</td>
</tr>
<tr>
<td>15</td>
<td>MPI_Ssend, MPI_Ssend</td>
<td>mw_init_Ssend</td>
</tr>
<tr>
<td>16</td>
<td>MPI_Bsend, MPI_Bcast</td>
<td>col_init_Bcast</td>
</tr>
<tr>
<td>17</td>
<td>MPI_Barrier</td>
<td>col_init_Barrier</td>
</tr>
<tr>
<td>18</td>
<td>MPI_Reduce</td>
<td>col_init_Reduce</td>
</tr>
<tr>
<td>19</td>
<td>MPI_Alltoall</td>
<td>col_init_Alltoall</td>
</tr>
<tr>
<td>20</td>
<td>MPI_Scan</td>
<td>col_init_Scan</td>
</tr>
<tr>
<td>21</td>
<td>MPI_Comm_split</td>
<td>col_init_Comm_split</td>
</tr>
<tr>
<td>22</td>
<td>memcpy (ANSI-C)</td>
<td>col_init_memcpy</td>
</tr>
<tr>
<td>23</td>
<td>memcpy (ANSI-C)</td>
<td>col_init_memcpy</td>
</tr>
<tr>
<td>24</td>
<td>MPI_Wtime</td>
<td>simple_init_Wtime</td>
</tr>
<tr>
<td>25</td>
<td>MPI_Comm_rank</td>
<td>simple_init_Comm_rank</td>
</tr>
<tr>
<td>26</td>
<td>MPI_Comm_size</td>
<td>simple_init_Comm_size</td>
</tr>
<tr>
<td>27</td>
<td>MPI_Probe (not successful)</td>
<td>simple_init_Probe</td>
</tr>
<tr>
<td>28</td>
<td>MPI_Buffer_attach</td>
<td>simple_init_Buffer</td>
</tr>
<tr>
<td>29</td>
<td>Dummy Point-to-point measurement</td>
<td>p2p_init_dummy</td>
</tr>
<tr>
<td>30</td>
<td>Dummy Master-Worker measurement</td>
<td>mw_init_dummy</td>
</tr>
<tr>
<td>31</td>
<td>Dummy collective measurement</td>
<td>col_init_dummy</td>
</tr>
<tr>
<td>32</td>
<td>Dummy simple measurement</td>
<td>simple_init_dummy</td>
</tr>
</tbody>
</table>

Table 3.1: The mapping of type-numbers to measured MPI-functions

3.1.1 Example

Let's ask, what is measured in type 16? First we have a look in table 3.3, on page 36. We see: The measurement type 16 belongs to the master-worker-pattern. Table 3.1 (page 35) shows that it is initialized with function mw_init_Bsend. The measured call-back of this pattern is the dispatch-call-back. (What we know from the description of the pattern on page 37.) So we have to find out which dispatch-call-back is used in type 16. We have a look into the ini-
CHAPTER 3. MEASUREMENTS IN DETAIL

33  MPI_Gather  colinit_Gather
34  MPI_Ssend  p2pinit_Ssend
35  MPI_Scatter  colinit_Scatter
36  MPI_Allreduce  colinit_Allreduce
37  MPI_Reduce followed by colinit_Reduce_Bcast
   MPI_Bcast
38  MPI_Reduce_scatter  colinit_Reduce_scatter
39  MPI_Allgather  colinit_Allgather
40  MPI_Scatterv  colinit_Scatterv
41  MPI_Gatherv  colinit_Gatherv
42  MPI_Allgatherv  colinit_Allgatherv
43  MPI_Alltoallv  colinit_Alltoallv
44  MPI_Reduce followed by colinit_Reduce_Scatterv
   MPI_Scatter
45  Implementation of Gather with
   MPI_Send and MPI_Recv
46  Implementation of Gather with
   MPI_Ssend, MPI_Recv, and
   MPI_Waitall

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

<table>
<thead>
<tr>
<th>Range of type numbers</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - 9</td>
<td>Point-to-point</td>
</tr>
<tr>
<td>10 - 16</td>
<td>Master-Worker</td>
</tr>
<tr>
<td>17 - 23</td>
<td>Collective</td>
</tr>
<tr>
<td>24 - 28</td>
<td>Simple</td>
</tr>
<tr>
<td>29 - 32</td>
<td>internal measurements</td>
</tr>
<tr>
<td>33</td>
<td>new Collective</td>
</tr>
<tr>
<td>34</td>
<td>new Point-to-Point</td>
</tr>
<tr>
<td>35 - 46</td>
<td>new Collective</td>
</tr>
</tbody>
</table>

Table 3.3: The mapping of type-numbers to patterns
The internal measurements are used to determine the overhead of measurements. The order of new measurements is somehow grown historically. To avoid incompatibilities I resigned from reordering the measurements.

...tializer (page 50). There we see that the name of our dispatch-call-back is master_dispatch_Bsend. This call-back is described on page 50.
3.1.2 Point-to-Point pattern

The ping-pong-pattern calls the routine_to_be_measured to communicate with the farthest node or the nearest node. These calls are varied over message length. Every parameter set is called repetitions times and the average value is stored. We have distinct code for the server (measurement) and the client (just answering).

/* Server-node */
max_node := node with maximum latency;

do
  start_time := MPI_Wtime;
  routine_to_be_measured (max_node, message_length);
  end_time :=MPI_Wtime;
while to_measure (end_time - start_time);

/* Client code */
actions to answer the max/min_node determination;

if (I am the max_node) 
do
  client answer for the routine_to_be_measured (message_length);
while not stop

Measured routine: This is the routine, which is used by the server to initiate communication to the client. The time consumed by it will be measured.

Client routine: This routine answers the communication initiated by the above routine. If the measured routine depends on an answer of this routine, it will be measured indirectly.

3.1.3 Master-Worker pattern

The Master-worker-pattern corresponds to the typical master-worker-scheme: a master process divides a problem in several sub-problems (here called chunks) and dispatches them several worker processes. When finished a worker sends his result to the master and requests for a new piece of work (and so on). When all work is done, the master sends an stop-signal to the workers.

This scheme is important in practice, since it automatically balances load. In pseudo-code the Master-worker-scheme looks like:

\[ \text{This means node with the maximum or minimum latency. We use the node with the maximum latency by default.} \]
/* master-code */

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

chunk := 0;
start_time := MPI_Wtime;

while chunk < all_chunks
    dispatch (chunk, msglen);
    chunks := chunks + 1;

end_time := MPI_Wtime;

for each worker
    send stop signal;

/* worker-code */

forever
    send ready signal to master;

    receive signal (msglen);

    if signal == stop signal
        exit;

    do work;                      /* corresponding to the received signal */
    send result;

endforever

Every abstract communication “code” in the scheme above can be filled with concrete MPI Code. We measure the time consumed by dispatch work. This code sequence does for example this:

/* dispatch work: */
wait for a worker;
receive work from worker;
send actual piece of work to worker;
set ready to receive next piece of work from worker;
actual piece of work := next piece of work;

Here we have to define the following call-back functions:

**Master receive ready:** This function can be used for posting the a receive for each worker.

**Master dispatch:** This is the routine, which dispatches work (sending to workers) and collects the results (it receives from the workers). Since it is
something like the “kernel” of this pattern, it is the routine measured.

**Master send stop signal:** This routine sends the stop signal to a worker.

**Worker receive:** This routine is used by a worker to receive its signals from the master process.

**Worker send:** The worker sends its result via this routine.

### 3.1.4 Collective pattern

We want to use the following pattern to measure collective operations:

```c
/* server-code */

MPI_Barrier;
do
  start_time := MPI_Wtime;
routine_to_be_measured;
MPI_Barrier;
end_time := MPI_Wtime;
while to_measure

/* client code */

MPI_Barrier;
do
  client_routine;  /* as answer for routine_to_be_measured */
MPI_Barrier;
while not stop;
```

Usually all the collective operations use the same function whether you are process zero (which measures and initiates communication) or not. But for the sake of flexibility we can use different routines. One for process zero (server) and one for the others (clients).

### 3.1.5 Simple pattern

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

```c
if I am node zero
do
  start_time := MPI_Wtime;
routine_to_be_measured;
end_time := MPI_Wtime();
while to_measure;
```

The only call-back function is the `routine_to_be_measured`. 
3.2 The call-back functions

This section serves as a reference, when you want to know exactly, what is measured. All call-back functions are listed below. Their role in the different patterns is explained in the last section.

3.2.1 Call-backs of the Point-to-Point pattern


to be measured.

(p2p_init...) and routines containing the MPI-Functions to be measured.

{ 
  
}

p2p_init_dummy

- Measured routine: p2p_dummy.
- Client-routine: p2p_dummy.

p2p_init_Send_Recv

- Measured routine: server_Send_Recv.
- Client-routine: client_Recv_Send.

p2p_init_Send_Iprobe_Recv

- Measured routine: server_Send_Iprobe_Recv.
- Client-routine: client_Iprobe_Recv_Send.

p2p_init_Send_Irecv

- Measured routine: server_Send_Irecv.
- Client-routine: client_Irecv_Send.

p2p_init_Send_Recv_AT

- Measured routine: server_Send_Recv_AT.
- Client-routine: client_Recv_AT_Send.
3.2. THE CALL-BACK FUNCTIONS

\[\text{p2p\_init\_Ssend\_Recv}\]
- Measured routine: `server\_Ssend\_Recv`.
- Client-routine: `client\_Recv\_Ssend`.

\[\text{p2p\_init\_Isend\_Recv}\]
- Measured routine: `server\_Isend\_Recv`.
- Client-routine: `client\_Recv\_Isend`.

\[\text{p2p\_init\_Issend\_Recv}\]
- Measured routine: `server\_Issend\_Recv`.
- Client-routine: `client\_Recv\_Issend`.

\[\text{p2p\_init\_Bsend\_Recv}\]
- Measured routine: `server\_Bsend\_Recv`.
- Client-routine: `client\_Recv\_Bsend`.

\[\text{p2p\_init\_Sendrecv}\]
- Measured routine: `server\_Sendrecv`.
- Client-routine: `client\_Sendrecv`.

\[\text{p2p\_init\_Sendrecv\_replace}\]
- Measured routine: `server\_Sendrecv\_replace`.
- Client-routine: `client\_Sendrecv\_replace`.

\[\text{init\_empty}\]

\[\text{init\_attach}\]

\[\text{free\_empty}\]

\[\text{void free\_empty (int msglen)}\]
\{   return;
\}
free_attach

void free_attach (int msglen)
{
    int buflen = msglen * sizeof(char) +
    MPI_BSEND_OVERHEAD + MY_OVERHEAD;
    MPI_Buffer_detach (_skampi_buffer, &buflen);
    return;
}

p2p_dummy

MPI_Status p2p_dummy (int msglen, int max_node,
    MPI_Comm communicator)
{
    MPI_Status status;
    /* be dummy */
    return (status);
}

server_Send_Recv

MPI_Status server_Send_Recv(int msglen, int max_node,
    MPI_Comm communicator)
{
    MPI_Status status;

    MPI_Send (_skampi_buffer, msglen, MPI_CHAR,
        max_node, 0, communicator);
    MPI_Recv (_skampi_buffer, msglen, MPI_CHAR,
        max_node, 1, communicator,
        &status);

    return (status);
}

server_Send_Iprobe_Recv

MPI_Status server_Send_Iprobe_Recv(int msglen, int max_node,
    MPI_Comm communicator)
{
    MPI_Status status;
    int flag;

    MPI_Send (_skampi_buffer, msglen, MPI_CHAR,
max_node, 0, communicator);

do {
   MPI_Iprobe (max_node, 1, communicator,
      &flag, &status);
} while (!flag);

MPI_Recv (_skampi_buffer, msglen, MPI_CHAR,
   max_node, 1, communicator, &status);

return (status);
}

server_Send_Irecv
MPI_Status server_Send_Irecv(int msglen, int max_node,
      MPI_Comm communicator)
{
   MPI_Status status;
   MPI_Request req;

   MPI_Send (_skampi_buffer, msglen, MPI_CHAR,
      max_node, 0, communicator);
   MPI_Irecv (_skampi_buffer, msglen, MPI_CHAR,
      max_node, 1, communicator, &req);
   MPI_Wait (&req, &status);

   return (status);
}

server_Send_Recv_AT
MPI_Status server_Send_Recv_AT(int msglen, int max_node,
      MPI_Comm communicator)
{
   MPI_Status status;

   MPI_Send (_skampi_buffer, msglen, MPI_CHAR,
      max_node, 0, communicator);
   MPI_Recv (_skampi_buffer, msglen, MPI_CHAR,
      max_node, MPI_ANY_TAG,
      communicator, &status);

   return (status);
}
server_Bsend_Recv
MPI_Status server_Bsend_Recv(int msglen, int max_node,
MPI_Comm communicator)
{
    MPI_Status status;

    MPI_Bsend (_skampi_buffer, msglen, MPI_CHAR,
               max_node, 0, communicator);

    MPI_Recv (_skampi_buffer, msglen, MPI_CHAR,
              max_node, 1, communicator, &status);

    return (status);
}

server_Isend_Recv
MPI_Status server_Isend_Recv (int msglen, int max_node,
MPI_Comm communicator)
{
    MPI_Status status;
    MPI_Request req;

    MPI_Isend (_skampi_buffer, msglen, MPI_CHAR,
               max_node, 0, communicator, &req);
    MPI_Wait (&req, &status);
    MPI_Recv (_skampi_buffer, msglen, MPI_CHAR,
              max_node, 1, communicator, &status);

    return (status);
}

server_Issend_Recv
MPI_Status server_Issend_Recv (int msglen, int max_node,
MPI_Comm communicator)
{
    MPI_Status status;
    MPI_Request req;

    MPI_Issend (_skampi_buffer, msglen, MPI_CHAR,
                max_node, 0, communicator, &req);
    MPI_Wait (&req, &status);
    MPI_Recv (_skampi_buffer, msglen, MPI_CHAR,
              max_node, 1, communicator, &status);
return (status);
}

clientRecv_Send

MPI_Status clientRecv_Send (int msglen, int node, 
MPI_Comm communicator)
{
    MPI_Status status;

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

    MPI_Send (_skampi_buffer, msglen, MPI_CHAR, 
              0, 1, communicator);

    return (status);
}

clientIprobe_RECV_send

MPI_Status clientIprobe_RECV_send (int msglen, int node, 
MPI_Comm communicator)
{
    MPI_Status status;
    int flag;

    MPI_Iprobe (0, 0, communicator, &flag, &status);
    MPI_Recv (_skampi_buffer, msglen, MPI_CHAR, 
              0, 0, communicator, &status);
    MPI_Send (_skampi_buffer, msglen, MPI_CHAR, 
              0, 1, communicator);

    return (status);
}

clientIrecv_Send

MPI_Status clientIrecv_Send (int msglen, int node, 
MPI_Comm communicator)
{
    MPI_Status status;
    MPI_Request req;

    MPI_Irecv (_skampi_buffer, msglen, MPI_CHAR,
0, 0, communicator, 
&req);
MPI_Wait (&req, &status);

MPI_Send (_skampi_buffer, msglen, MPI_CHAR, 
0, 1, communicator);

return (status);
}

client_Recv_AT_Send

MPI_Status client_Recv_AT_Send (int msglen, int node, 
MPI_Comm communicator)
{
    MPI_Status status;

    MPI_Recv (_skampi_buffer, msglen, MPI_CHAR, 
0, MPI_ANY_TAG, communicator, &status);
    MPI_Send (_skampi_buffer, msglen, MPI_CHAR, 
0, 1, communicator);

    return (status);
}

client_Recv_Bsend

MPI_Status client_Recv_Bsend (int msglen, int node, 
MPI_Comm communicator)
{
    MPI_Status status;

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

    MPI_Bsend (_skampi_buffer, msglen, MPI_CHAR, 
0, 1, communicator);

    return (status);
}

client_Recv_Isend

MPI_Status client_Recv_Isend (int msglen, int node, 
MPI_Comm communicator)
3.2. THE CALL-BACK FUNCTIONS

{
    MPI_Status status;
    MPI_Request req;

    MPI_Recv (_skampi_buffer, msglen, MPI_CHAR, 
    0, 0, communicator, &status);
    MPI_Isend (_skampi_buffer, msglen, MPI_CHAR, 
    0, 1, communicator, &req);
    MPI_Wait (&req, &status);

    return (status);
}

client_Recv_Isend
MPI_Status client_Recv_Isend (int msglen, int node, 
    MPI_Comm communicator)
{
    MPI_Status status;
    MPI_Request req;

    MPI_Recv (_skampi_buffer, msglen, MPI_CHAR, 
    0, 0, communicator, &status);
    MPI_Isend (_skampi_buffer, msglen, MPI_CHAR, 
    0, 1, communicator, &req);
    MPI_Wait (&req, &status);

    return (status);
}

server_Ssend_Recv
MPI_Status server_Ssend_Recv(int msglen, int max_node, 
    MPI_Comm communicator)
{
    MPI_Status status;

    MPI_Ssend (_skampi_buffer, msglen, MPI_CHAR, 
    max_node, 0, communicator);
    MPI_Recv (_skampi_buffer, msglen, MPI_CHAR, 
    max_node, i, communicator, 
    &status);

    return (status);
}
client_{Recv, Ssend}

MPI_Status client_{Recv, Ssend} (int msglen, int node,
    MPI_Comm communicator)
{
    MPI_Status status;

    MPI_Recv (_skampi_buffer, msglen, MPI_CHAR, 0, 0, communicator, &status);
    MPI_Ssend (_skampi_buffer, msglen, MPI_CHAR, 0, 1, communicator);

    return (status);
}

server_Send

MPI_Status server_Send(int msglen, int max_node,
    MPI_Comm communicator)
{
    MPI_Status status;

    MPI_Send (_skampi_buffer, msglen, MPI_CHAR, max_node, 0, communicator);

    return (status);
}

server_Isend

MPI_Status server_Isend(int msglen, int max_node,
    MPI_Comm communicator)
{
    MPI_Status status;
    MPI_Request req;

    MPI_Isend (_skampi_buffer, msglen, MPI_CHAR, max_node, 0, communicator, &req);
    MPI_Wait (&req, &status);

    return (status);
}

server_Ssend

MPI_Status server_Ssend (int msglen, int max_node,
MPI_Comm communicator
{
MPI_Status status;

MPI_Ssend (_skampi_buffer, msglen, MPI_CHAR, 
max_node, 0, communicator);

return (status);
}

client_Recv
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);
}

server_Sendrecv_replace
MPI_Status server_Sendrecv_replace (int msglen, int node, 
MPI_Comm communicator)
{
MPI_Status status;

MPI_Sendrecv_replace (_skampi_buffer, msglen, MPI_CHAR, 
node, 0, node, 1, communicator, &status);
return (status);
}

client_Sendrecv_replace
MPI_Status client_Sendrecv_replace (int msglen, int node, 
MPI_Comm communicator)
{
MPI_Status status;

MPI_Sendrecv_replace (_skampi_buffer, msglen, MPI_CHAR, 
0, 1, 0, 0, communicator, &status);
return (status);
server_Sendrecv

MPI_Status server_Sendrecv (int msglen, int node,
       MPI_Comm communicator)
{
       MPI_Status status;

       MPI_Sendrecv (_skampi_buffer, msglen, MPI_CHAR, node, 0,
                      _skampi_buffer_2, msglen, MPI_CHAR, node, 1,
                      communicator, &status);
       return (status);
}

client_Sendrecv

MPI_Status client_Sendrecv (int msglen, int node,
       MPI_Comm communicator)
{
       MPI_Status status;

       MPI_Sendrecv (_skampi_buffer, msglen, MPI_CHAR, 0, 1,
                      _skampi_buffer_2, msglen, MPI_CHAR, 0, 0,
                      communicator, &status);
       return (status);
}

3.2.2 Call-backs of the Master-Worker pattern


to be measured.

(mw_init_...) and routines containing the MPI-Functions to be measured.
3.2. THE CALL-BACK FUNCTIONS

**mw_init_dummy**
- Master receive ready routine: `master_receive_ready_empty`.
- Master dispatch routine: `master_dispatch_dummy`.
- Routine to send stop signals: `master_worker_stop_recv`.
- Worker receive routine: `worker_receive_test`.
- Worker send routine: `worker_send_test`.

**mw_init_Waitsome**
- Master receive ready routine: `master_receive_ready_test`.
- Master dispatch routine: `master_dispatch_Waitsome`.
- Routine to send stop signals: `master_worker_stop_wait`.
- Worker receive routine: `worker_receive_test`.
- Worker send routine: `worker_send_test`.

**mw_init_Waitany**
- Master receive ready routine: `master_receive_ready_test`.
- Master dispatch routine: `master_dispatch_Waitany`.
- Routine to send stop signals: `master_worker_stop_test`.
- Worker receive routine: `worker_receive_test`.
- Worker send routine: `worker_send_test`.

**mw_init_Recv_AS**
- Master receive ready routine: `master_receive_ready_empty`.
- Master dispatch routine: `master_dispatch_Recv_AS`.
- Routine to send stop signals: `master_worker_stop_recv`.
- Worker receive routine: `worker_receive_test`.
- Worker send routine: `worker_send_test`. 
**mw_init_Send**

- Master receive ready routine: `master_receive_ready_empty`.
- Master dispatch routine: `master_dispatch_Send`.
- Routine to send stop signals: `master_worker_stop_recv`.
- Worker receive routine: `worker_receive_test`.
- Worker send routine: `worker_send_test`.

**mw_init_Ssend**

- Master receive ready routine: `master_receive_ready_empty`.
- Master dispatch routine: `master_dispatch_Ssend`.
- Routine to send stop signals: `master_worker_stop_recv`.
- Worker receive routine: `worker_receive_test`.
- Worker send routine: `worker_send_test`.

**mw_init_Isend**

- Master receive ready routine: `master_receive_ready_empty`.
- Master dispatch routine: `master_dispatch_Isend`.
- Routine to send stop signals: `master_worker_stop_recv`.
- Worker receive routine: `worker_receive_test`.
- Worker send routine: `worker_send_test`.

**mw_init_Bsend**

- Master receive ready routine: `master_receive_ready_empty`.
- Master dispatch routine: `master_dispatch_Bsend`.
- Routine to send stop signals: `master_worker_stop_recv`.
- Worker receive routine: `worker_receive_test`.
- Worker send routine: `worker_send_test`. 
master_receive_ready_test

void master_receive_ready_test (int worker, int len, 
    MPI_Comm communicator) 
{
    MPI_Irecv (_mw_buffer[worker - 1], 0, MPI_CHAR, 
        worker, MPI_ANY_TAG, 
        communicator, _mw_req + worker - 1);
}

master_worker_stop_wait

void master_worker_stop_wait (int worker, int len, 
    MPI_Comm communicator) 
{
    MPI_Wait (_mw_req + (worker - 1), 
        master_stati + (worker - 1));
    MPI_Ssend (_skampi_buffer, 0, MPI_CHAR, 
        worker, 0, communicator);
}

master_worker_stop_test

void master_worker_stop_test (int worker, int len, MPI_Comm communicator) 
{
    MPI_Ssend (_skampi_buffer, 0, MPI_CHAR, 
        worker, 0, communicator);
}

master_worker_stop_recv

void master_worker_stop_recv (int worker, int len, MPI_Comm communicator) 
{
    MPI_Status 
        status;
    MPI_Recv (_skampi_buffer, 0, MPI_CHAR, 
        worker, 1, communicator, &status);
    MPI_Ssend (_skampi_buffer, 0, MPI_CHAR, 
        worker, 0, communicator);
}
worker receive test
int worker_receive_test (int len, MPI_Comm communicator)
{
    MPI_Status status;

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

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

    return (TRUE);
}

worker send test
void worker_send_test (int len, MPI_Comm communicator)
{
    MPI_Ssend (_skampi_buffer, 0, MPI_CHAR,
                0, 1, communicator);
}

master init empty
master free empty
void master_free_empty (int mw_numprocs)
{
    return;
}

master receive ready empty
void master_receive_ready_empty (int worker, int len,
{
    return;
}

master worker stop empty
void master_worker_stop_empty (int worker, int len,
{
    return;
}
worker_send_empty

```c
void worker_send_empty (int len, MPI_Comm communicator)
{
    return;
}
```

master_dispatch_dummy

```c
int master_dispatch_dummy (int number_of_workers, int work,
                           int chunks, int len,
                           MPI_Comm communicator)
{
    return (1);
}
```

master_dispatch_Waitsome

```c
int master_dispatch_Waitsome (int number_of_workers, int work,
                               int chunks,
                               int len, MPI_Comm communicator)
{
    int i,
        worker,
        eingaenge;

    MPI_Waitsome (number_of_workers, _mw_req, &eingaenge,
                  _mw_index, master_stati);

    D1 (fprintf (stderr, "master: eingaenge: %d at len %d\n",
                eingaenge, len));

    for (i = 0; i < eingaenge; i++)
    {
        worker = _mw_index[i] + 1;

        /* posting new recv for this worker, because the old one has been used */
        MPI_Irecv (_mw_buffer[worker - 1], 0, MPI_CHAR,
                    worker, MPI_ANY_TAG, communicator,
                    _mw_req + worker - 1);

        /* sending next chunk of work to this worker */
        MPI_Send (_skampi_buffer, len, MPI_CHAR,
CHAPTER 3. MEASUREMENTS IN DETAIL

worker, 1, communicator);
D1(fprintf (stderr, "master: sending job_no %d to worker %d\n", \
work, worker);
#endif
}
return (eingaenge);
}

master_init_Waitsome
master_free_Waitsome
void master_free_Waitsome (int mw_numprocs)
{
  int worker;

  free (_mw_index);
  free (_mw_req);
  free (master_stati);

  for (worker = 0; worker < mw_numprocs - 1; worker++)
    free (_mw_buffer[worker]);

  free (_mw_buffer);
}

master_dispatch_Waitany
int master_dispatch_Waitany (int number_of_workers, \nint work, int chunks, int len, \nMPI_Comm communicator)
{
  int
    worker;

  MPI_Status
    status;

  MPI_Waitany (number_of_workers, _mw_req, \n    &worker, &status);
worker++;

/* posting new recv for this worker, because the old one has been used */
MPI_Irecv (_mw_buffer[worker], 0, MPI_CHAR, worker,
    MPI_ANY_TAG, communicator, _mw_req + worker);

/* sending next chunk of work to this worker */
MPI_Send (_skampi_buffer, len, MPI_CHAR, worker, 1, communicator);

D(fprintf (stderr, "master: sending job_no %d to worker %d
", work, worker);

return (1);
}

master_init_Waitany

master_free_Waitany
void master_free_Waitany (int mw_numprocs)
{
    int worker;

    free (_mw_req);

    for (worker = 0; worker < mw_numprocs - 1; worker++)
        free (_mw_buffer[worker]);

    free (_mw_buffer);
}

master_dispatch_Recv_AS
int master_dispatch_Recv_AS (int number_of_workers,
    int work, int chunks, int len,
    MPI_Comm communicator)
{
    int
    worker;

    MPI_Status
    status;

    MPI_Recv (_skampi_buffer, 0, MPI_CHAR, MPI_ANY_SOURCE,
MPI_ANY_TAG, communicator, &status);

worker = status.MPI_SOURCE;

/* sending next chunk of work to this worker */
MPI_Send (_skampi_buffer, len, MPI_CHAR, 
           worker, 1, communicator);

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

if (++work == chunks)
{
    return (chunks);
}

return (1);
}

master_dispatch_Send
int master_dispatch_Send (int number_of_workers,
                          int work, int chunks, int len,
                          MPI_Comm communicator)
{
    MPI_Status
    status;

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

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

    return (1);
}

master_dispatch_Ssend
int master_dispatch_Ssend (int number_of_workers,
                           int work, int chunks, int len,
                           MPI_Comm communicator)
{
3.2. THE CALL-BACK FUNCTIONS

```c
MPI_Status
status;

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

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

return (1);
}

master_dispatch_Isend
int master_dispatch_Isend (int number_of_workers,
    int work, int chunks, int len,
    MPI_Comm communicator)
{
    MPI_Request
    req;

    MPI_Status
    status;

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

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

return (1);
}

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

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

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

    return (1);
}

master_init_attach

master_free_attach

void master_free_attach (int mw_numprocs)
{
    int buflen = max_msg_len * sizeof(char)
        + MPI_BSEND_OVERHEAD + MY_OVERHEAD;
    MPIBuffer_detach (_skampi_buffer, &buflen);
}

3.2.3 Call-backs of the Collective pattern


col_init_dummy

- measured routine: measure_col_dummy.
- client-routine: measure_col_dummy

col_init_Bcast

- measured routine: measure_broadcast.
- client-routine: measure_broadcast
3.2. THE CALL-BACK FUNCTIONS

\textbf{colinit\_Barrier}

- measured routine: \texttt{measure\_barrier}.
- client-routine: \texttt{measure\_barrier}

\textbf{colinit\_Reduce}

- measured routine: \texttt{measure\_Reduce}.
- client-routine: \texttt{measure\_Reduce}

\textbf{colinit\_Allreduce}

- measured routine: \texttt{measure\_Allreduce}.
- client-routine: \texttt{measure\_Allreduce}

\textbf{colinit\_Reduce\_Bcast}

- measured routine: \texttt{measure\_Reduce\_Bcast}.
- client-routine: \texttt{measure\_Reduce\_Bcast}

\textbf{colinit\_Reduce\_scatter}

- measured routine: \texttt{init\_measure\_Reduce\_scatter}.
- client-routine: \texttt{init\_measure\_Reduce\_scatter}
- measured routine: \texttt{measure\_Reduce\_scatter}.
- client-routine: \texttt{measure\_Reduce\_scatter}

\textbf{colinit\_Reduce\_Scatterv}

- measured routine: \texttt{init\_measure\_Reduce\_Scatterv}.
- client-routine: \texttt{init\_measure\_Reduce\_Scatterv}
- measured routine: \texttt{measure\_Reduce\_Scatterv}.
- client-routine: \texttt{measure\_Reduce\_Scatterv}

\textbf{colinit\_Scan}

- measured routine: \texttt{measure\_Scan}.
- client-routine: \texttt{measure\_Scan}
col\_init\_Alltoall
  
  - measured routine: measure\_Alltoall.
  
  - client-routine: measure\_Alltoall

col\_init\_Alltoallv
  
  - measured routine: init\_measure\_recvlens\_displs.
  
  - client-routine: init\_measure\_recvlens\_displs
  
  - measured routine: measure\_Alltoallv.
  
  - client-routine: measure\_Alltoallv

col\_init\_Gather
  
  - measured routine: measure\_Gather.
  
  - client-routine: measure\_Gather

col\_init\_Gather\_Send\_Recv
  
  - measured routine: measure\_Gather\_Recv\_server.
  
  - client-routine: measure\_Gather\_Send\_client

col\_init\_Gather\_Isend\_Waitall
  
  - measured routine: measure\_Gather\_Waitall\_server.
  
  - client-routine: measure\_Gather\_Isend\_client

col\_init\_Gatherv
  
  - measured routine: init\_measure\_recvlens\_displs.
  
  - client-routine: init\_measure\_recvlens\_displs
  
  - measured routine: measure\_Gatherv.
  
  - client-routine: measure\_Gatherv

col\_init\_Allgather
  
  - measured routine: measure\_Allgather.
  
  - client-routine: measure\_Allgather
3.2. THE CALL-BACK FUNCTIONS

\texttt{col\_init\_Allgatherv}
- measured routine: \texttt{init\_measure\_recvlens\_displs}.
- client-routine: \texttt{init\_measure\_recvlens\_displs}
- measured routine: \texttt{measure\_Allgatherv}.
- client-routine: \texttt{measure\_Allgatherv}

\texttt{col\_init\_Scatter}
- measured routine: \texttt{measure\_Scatter}.
- client-routine: \texttt{measure\_Scatter}

\texttt{col\_init\_Scatterv}
- measured routine: \texttt{init\_measure\_recvlens\_displs}.
- client-routine: \texttt{init\_measure\_recvlens\_displs}
- measured routine: \texttt{measure\_Scatterv}.
- client-routine: \texttt{measure\_Scatterv}

\texttt{col\_init\_Comm\_dup}
- measured routine: \texttt{measure\_Comm\_dup}.
- client-routine: \texttt{measure\_Comm\_dup}

\texttt{col\_init\_Comm\_split}
- measured routine: \texttt{measure\_Comm\_split}.
- client-routine: \texttt{measure\_Comm\_split}

\texttt{col\_init\_memcpy}
- measured routine: \texttt{measure\_memcpy}.
- client-routine: \texttt{measure\_col\_dummy}

\texttt{measure\_col\_dummy}
\begin{verbatim}
void measure\_col\_dummy (int len, MPI\_Comm communicator)
{
    // just for dummy measurement */
    return;
}
\end{verbatim}
measure_broadcast

void measure_broadcast (int len, MPI_Comm communicator)
{
    MPI_Bcast(_skampi_buffer, len, MPI_CHAR, 0, communicator);
}

measure_barrier

void measure_barrier (int len, MPI_Comm communicator)
{
    MPI_Barrier(communicator);
}

measure_Reduce

void measure_Reduce (int len, MPI_Comm communicator)
{
    MPI_Reduce(_skampi_buffer, _skampi_buffer_2, len, MPI_BYTE, 
               MPI_BOR, 0, communicator);
}

measure_Allreduce

void measure_Allreduce (int len, MPI_Comm communicator)
{
    MPI_Allreduce(_skampi_buffer, _skampi_buffer_2, len, MPI_BYTE, 
                  MPI_BOR, communicator);
}

measure_Reduce_Bcast

void measure_Reduce_Bcast (int len, MPI_Comm communicator)
{
    MPI_Reduce(_skampi_buffer, _skampi_buffer_2, len, MPI_BYTE, 
               MPI_BOR, 0, communicator);
    MPI_Bcast(_skampi_buffer, len, MPI_CHAR, 0, communicator);
}
measure_Reduce_scatter

void measure_Reduce_scatter (int len, MPI_Comm communicator)
{
    MPI_Reduce_scatter(_skampi_buffer, _skampi_buffer_2, recvlens, MPI_BYTE,
                       MPI_BOR, communicator);
}

measure_Reduce_Scatterv

void measure_Reduce_Scatterv (int len, MPI_Comm communicator)
{
    MPI_Reduce(_skampi_buffer, _skampi_buffer_2, len, MPI_BYTE,
               MPI_BOR, 0, communicator);

    MPI_Scatterv (_skampi_buffer_2, recvlens, displs, MPI_CHAR,
                  _skampi_buffer, len, MPI_CHAR, 0, communicator);

    /* in the above call the "0" is featuring as root Note: the pointers
       _skampi_buffer and _skampi_buffer_2 are interchanged in this
       call. This is done, because so we can use the memory initializing
       for MPI_Gather.
       recvlens are used here as send lengths */
}

measure_Scan

void measure_Scan (int len, MPI_Comm communicator)
{
    MPI_Scan (_skampi_buffer, _skampi_buffer_2, len, MPI_BYTE,
              MPI_BOR, communicator);
}

measure_Alltoall

void measure_Alltoall (int len, MPI_Comm communicator)
{
    MPI_Alltoall (_skampi_buffer, len, MPI_CHAR,
                  _skampi_buffer_2, len, MPI_CHAR, communicator);
}
measure_Alltoallv

void measure_Alltoallv (int len, MPI_Comm communicator)
{
    MPI_Alltoallv (_skampi_buffer, recvlens, displs, MPI_CHAR,
        _skampi_buffer_2, recvlens, displs, MPI_CHAR, communicator);
    /* the first occurrence of recvlens and displs should be read as
     send lens and send displacements */
}

measure_Gather

void measure_Gather (int len, MPI_Comm communicator)
{
    MPI_Gather (_skampi_buffer, len, MPI_CHAR,
        _skampi_buffer_2, len, MPI_CHAR, 0, communicator);
    /* in the above call the "0" is featuring as root */
}

measure_Gather_Recv_server

void measure_Gather_Recv_server (int len, MPI_Comm communicator)
{
    int
        i,
        numprocs;
    MPI_Status
        status;
    D7(int myrank);
    D7(MPI_Comm_rank(communicator, &myrank));
    MPI_Comm_size(communicator,&numprocs);
    for (i = 1; i < numprocs; i++)
    {
        D7(fprintf(stderr,"proc %d: receiving from %d\n", myrank, i));
        MPI_Recv (_skampi_buffer_2 + (i-1)*len, len, MPI_CHAR,
            i, 0, communicator, &status);
        D7(fprintf(stderr,"proc %d: received from %d\n", myrank, i));
    }
3.2. THE CALL-BACK FUNCTIONS

measure_Gather_Send_client

```c
void measure_Gather_Send_client (int len, MPI_Comm communicator)
{
    D7(int myrank;)
    D7(MPI_Comm_rank(communicator, &myrank;));
    D7(fprintf(stderr,"proc %d: sending to root\n", myrank;));
    MPI_Send (_skampi_buffer, len, MPI_CHAR,
             0, 0, communicator);
}
```

measure_Gather_Waitall_server

```c
void measure_Gather_Waitall_server (int len, MPI_Comm communicator)
{
    int
    i,
    numprocs;
    D7(int myrank;)
    D7(MPI_Comm_rank(communicator, &myrank;));
    MPI_Comm_size(communicator,&numprocs);
    for (i = 1; i < numprocs; i++)
    {
        D7(fprintf(stderr,"proc %d: receiving from %d\n", myrank, i;))
        MPI_Irecv (_skampi_buffer_2 + (i-1)*len, len, MPI_CHAR,
                    i, 0, communicator, _col_req + (i - 1));
        D7(fprintf(stderr,"proc %d: received from %d\n", myrank, i;))
    }
    D7(fprintf(stderr,"proc %d: left loop, numprocs %d\n", myrank, numprocs;))
    MPI_Waitall(numprocs - 1, _col_req, _col stati);
}
```

measure_Gather_Isend_client

```c
void measure_Gather_Isend_client (int len, MPI_Comm communicator)
{
    MPI_Request
    req;
    D7(int myrank;)
    D7(MPI_Comm_rank(communicator, &myrank;));
    D7(fprintf(stderr,"proc %d: sending to root\n", myrank;))
    MPI_Isend (_skampi_buffer, len, MPI_CHAR,
                  0, 0, communicator);
}
```
0, 0, communicator, &req);

  /* We do not use a completion operation here, since the barrier sync
   * after every col operation assures, that the wait all of the server
   * is finished, when proceeded. */

measure_Gatherv

void measure_Gatherv (int len, MPI_Comm communicator)
{
    MPI_Gatherv (_skampi_buffer, len, MPI_CHAR,
                 _skampi_buffer_2, recvlens, displs, MPI_CHAR, 0, communicator);

    /* in the above call the "0" is featuring as root */
}

measure_Allgather

void measure_Allgather (int len, MPI_Comm communicator)
{
    MPI_Allgather (_skampi_buffer, len, MPI_CHAR,
                   _skampi_buffer_2, len, MPI_CHAR, communicator);
}

measure_Allgatherv

void measure_Allgatherv (int len, MPI_Comm communicator)
{
    MPI_Allgatherv (_skampi_buffer, len, MPI_CHAR,
                    _skampi_buffer_2, recvlens, displs, MPI_CHAR, communicator);
}

measure_Scatter

void measure_Scatter (int len, MPI_Comm communicator)
{
    MPI_Scatter (_skampi_buffer_2, len, MPI_CHAR,
                 _skampi_buffer, len, MPI_CHAR, 0, communicator);

    /* in the above call the "0" is featuring as root Note: the pointers
     * _skampi_buffer and _skampi_buffer_2 are interchanged in this
     * call. This is done, because so we can use the memory initializing

for MPI_Gather. */
}

measure_Scatterv

void measure_Scatterv (int len, MPI_Comm communicator)
{
    MPI_Scatterv (_skampi_buffer_2, recvlens, displs, MPI_CHAR,
    _skampi_buffer, len, MPI_CHAR, 0, communicator);

    /* in the above call the "0" is featuring as root Note: the pointers
    _skampi_buffer and _skampi_buffer_2 are interchanged in this
    call. This is done, because so we can use the memory initializing
    for MPI_Gather.
    recvlens are used here as send lengths */
}

measure_Comm_dup

void measure_Comm_dup (int len, MPI_Comm communicator)
{
    MPI_Comm new_comm;

    MPI_Comm_dup (communicator, &new_comm);
}

measure_Comm_split

void measure_Comm_split (int len, MPI_Comm communicator)
{
    MPI_Comm new_comm;

    MPI_Comm_split (communicator, _skampi_myid % 2, 0, &new_comm);
}

measure_memcpy

void measure_memcpy (int len, MPI_Comm communicator)
{
    memcpy (_skampi_buffer, _skampi_buffer_2, len);
}
init_measure_Reduce_scatter
init_measure_reclens_displs
init_measure_Reduce_Scatterv

3.2.4 Call-backs of the Simple pattern


to be measured.

(simple_init_...) and routines containing the MPI-Functions to be measured.
{
}

simple_init_dummy
  • measured routine: measure_dummy.

simple_init_Wtime
  • measured routine: measure_Wtime.

simple_init_2Wtime
  • measured routine: measure_2Wtime.

simple_init_Comm_size
  • measured routine: measure_Comm_size.

simple_init_Comm_rank
  • measured routine: measure_Comm_rank.

simple_init_Iprobe
  • measured routine: measure_Iprobe.

simple_init_attach
  • measured routine: measure_attach.
3.2. THE CALL-BACK FUNCTIONS

measure_dummy
void measure_dummy ()
{
    return;
}

measure_Wtime
void measure_Wtime ()
{
    double _dummy;
    _dummy = MPI_Wtime();
}

measure_2Wtime
void measure_2Wtime ()
{
    double _dummy;
    _dummy = MPI_Wtime();
    _dummy = MPI_Wtime();
}

measure_Comm_size
void measure_Comm_size ()
{
    int _dummy;
    MPI_Comm_size (MPI_COMM_WORLD, &_dummy);
}

measure_Comm_rank
void measure_Comm_rank ()
{
    int _dummy;
    MPI_Comm_rank (MPI_COMM_WORLD, &_dummy);
}
measure_Iprobe
void measure_Iprobe ()
{
    MPI_Status
    status;
    int
    _dummy;

    MPI_Iprobe (1, 0, MPI_COMM_WORLD, &_dummy, &status);
}

measure_attach
void measure_attach ()
{
    int buflen = MPI_BSEND_OVERHEAD + MY_OVERHEAD;

    MPI_Buffer_attach (_skampi_buffer, buflen);
    MPI_Buffer_detach (&_skampi_buffer, &buflen);
}

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

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

# @MACHINE IBM RS/6000 SP
# @NODE thin node P2SC 120 MHz
# @NETWORK High Performance Switch TB3
# @USER Ralf Reussner
# @SKAMPIVERSION 1.20
# @OSNAME AIX
# @OSRELEASE 2
# @OSVERSION 4
# @HOSTNAME p071
Data

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

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

A typical header of the simple-pattern looks like:

```
#Description of the MPI_Wtime measurement:
#Pattern: Simple.
#max. allowed standard error is 3.00 %
#Format: <args> <results> time_cleaned [microsec.] (%f) standard_error_cleaned
         [%] (%f) count_cleaned [number] (%d) time_all [microsec.] (%f) standard_error_all
         [%] (%f) count_all [number] (%d)
```

Note that the @in-command is used by the report generator, to identify the measurements. All other lines start with a  , so that gnuplot treats these lines as comments.

The small header for suites of the simple-pattern looks different, because this pattern does not have information on scale, range and default values. (But both list-headers have the same length of eight lines.)

---

2and to create temporary files.
3For implementors: This string is created in the function measurement.data_to_string in module skmpi_tools.
Note the following line giving the typing information of the result list (the result list is described in the next subsection).

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

These lines should be read as one continuous line. The basic idea is, that the formats of the result-lists may differ. So it is important to describe each list’s format.

The format-line starts with "#Format:”, followed by a tag (<args>), which means, that a description of arguments follows. (In case of multi dimensional measurements more than one argument belongs to one measurement.) Each argument is described with its name (in our example number_of_nodes) than its unit ([number]) and its format in C-Syntax given in round brackets (e.g., (%d)). Each so described argument corresponds to one column of the result-list. The arguments describing list is followed by another list, the results describing list. Each entry describes a column of the result list. An entry is formed by the following data (similar to an entry of the argument list): name, unit, and format.

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

```
2 176.059111 3.034745 8 176.059111 3.034745 8
3 386.971049 14.221803 8 386.971049 14.221803 8
4 370.513008 14.726381 8 370.513008 14.726381 8
5 553.763306 26.948681 11 553.763306 26.948681 11
6 521.403970 10.311949 8 521.403970 10.311949 8
7 577.031024 9.031125 8 577.031024 9.031125 8
8 484.304333 24.567614 11 484.304333 24.567614 11
9 706.000973 35.550781 68 706.000973 35.550781 68
10 171.232959 25.582080 8 171.232959 25.582080 8
11 802.918863 33.229863 8 802.918863 33.229863 8
12 806.794216 37.361757 11 806.794216 37.361757 11
13 766.557961 21.876852 8 766.557961 21.876852 8
14 818.220084 37.641216 9 818.220084 37.641216 9
15 827.972894 36.904118 9 827.972894 36.904118 9
16 758.197092 36.257975 14 758.197092 36.257975 14
#eol
```

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

**Trailer**

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


Index

Parameter files

dorep, 29
skampi, 5, 19

C
compile portability, 3
contention, 25

D
default values, 21
dynamic linear, 23
dynamic log, 23

F
fixed linear, 23
fixed log, 23

H
homepage, 3

M
measurement, 18
scale of, 23
single, 18

suite of
eight, 26
time limit of a, 24
type of, 23
measurements
performed by SKaMPI, 7
suite of, 18
memory alignment problems, 24

N
node times, 24

P
parameter file, 5, 19
pattern, 34
performance portability, 3
portability, 3

R
report generator, 6
run, 18
S
scale of measurement, 23
single measurement, 18
skampi, 1, 2
goal, 2
homepage, 3
skosfile, 4
standard error, 26
suite of measurements, 18

T
time limit
  of a measurement, 24
  of a suite, 24
type of measurement, 23