Recent Advances of SKaMPI

R. H. Reussner

Chair Informatics for Engineering and Science
University of Karlsruhe
D-76128 Karlsruhe

Abstract. The goal of SKaMPI is the creation of a database containing performance measurements of parallel computers in terms of MPI. This data supports software developers in creating portable and fast programs. To meet this goal portability is a crucial property of our this benchmark. A large number of platforms to test on and to examine is important for the realization of our vision. Since access to the Stuttgart Cray T3E in October 1998 we were able to improve the code’s portability, to solve memory alignment problems, to compare the quality Cray’s implemented MPI gather algorithm to the one used in IBM’s SP MPI library, and to investigate the blocking behavior of collective MPI operations. Recently we added the parameterization of measurements suites with (even user defined) datatypes.

1 Introduction

SKaMPI is the Special Karlsruher MPI-Benchmark [5]. 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) [8] 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 now 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 to choose 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 when the considered target machine is not accessible, or a workstation is used for development. This also can lower cost of development.

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

The SKaMPI project 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.

It is clear, that the achievement of this goal relies on a portable benchmark suite. Recently problems and constraints of reliable MPI benchmarking have been discussed in [3, 2]. SKaMPI tries to accomplish these constraints.

Access to the Stuttgart Cray T3E has several benefits for our project. Related to the access to the T3E from October 1998 to February 1999 we worked on the following topics, presented in this paper: solving problems with memory alignment (section 2.1), evaluation of implementations of 

2.1 Solved problems with memory alignment

Measurements on the Cray T3E revealed two problems in the internal memory management mechanism of SKaMPI. First of all, generated 32-bit addresses
which are not aligned to 64-bit were the reason for unstable behavior of SKaMPI on
the T3E. These problems never occurred on our SunOS/Solaris or Linux worksta-
tions, or on the IBM SP2 in Karlsruhe. Through improving the internal memory
management we improved SKaMPI’s portability. Second, we found that message
transfers (e.g., with MPI_Send / MPI_Recv) with a message length not a multiple
of 4 byte need significantly more time than transfers of messages with a length
divisible by four (see figure 1). Since this effect has not occurred on DEC alpha
workstations (which also use a memory alignment of four byte) we assume that
it is caused by Cray’s communication hardware.

![Graph showing the relationship between message length and time (microsec.)](image)

**Fig. 1.** Memory alignment effects at MPI_Send, MPI_Recv

We resolved this problem through the introduction of a new parameter, en-
tered in the configuration file. All possible message length can be restricted to a
multiple of this given value. Without access to the Cray T3E we would not have
been able to detect this effect.
2.2 The quality of algorithms for MPI\_Gather

Collective operations play a crucial role in programming message passing systems [1]. Besides measuring the performance of collective operations, also an evaluation of their quality gives useful information for the software developer. The quality of a collective operation is given through the selection of an algorithm appropriate to the given hardware and through a good implementation of this algorithm. If the software developer is aware of the bad quality of an certain collective operation, this operation can be replaced by an implementation of his own.

One way to determine the quality of a collective operation’s implementation is to compare it to other implementations with known quality. In the SKaMPI project we focused on the quality of MPI\_Gather. The gather operation collects data at a so called root node from other nodes. It is quite useful when implementing matrix algorithms, for example.

We created two alternative implementations. One is the naive way (low quality): All nodes send with MPI\_Send their message to the root node, which receives with MPI\_Recv. The pseudo code at the root node looks like:

```c
for (node=1; node \leq\ MAX\_NODE; node++)
    receive (buffer[node], node);
```

Since the communication is blocking, messages are accepted at the root node in a certain order (i.e., first message from sending node number one, than from number two, ...). Messages sent from a node with a high number have to wait until the messages from all nodes with a lower number are received. We will call this implementation the Gather\_SR implementation. The other way is somehow more sophisticated: The clients use MPI\_Isend and the receiving root node uses MPI\_Waitall. Through this non blocking communication no order on the incoming messages is induced at the root node. So there are no artificial delays, because every arriving message is accepted immediately.

One of the interesting results of the comparison of this three gather operations is, that on the IBM SP the naive Gather\_SR is fastest at small message lengths (below 2 KBytes), see figure 2.

The slat at 2 KBytes is caused by a the two different protocols used for message transfers with MPI\_Send. Messages with a length below 2 KBytes are sent immediately to the receiver, because it can be assumed that the receiver has sufficient memory buffer to store the message. Messages larger than 2 KBytes are sent in two phases: in phase one the the sending node requests sufficient buffer at the receiving node. Phase two starts, when the receiver commits the allocation of the buffer. Then the message is transfered. Obviously the latter protocol for long messages needs more time than the first, which results in the slat. Nevertheless, when using a combination of our two implementations an
Fig. 2. Different Gather implementations on the IBM SP2
implementation of MPI\texttt{Gather} is yielded that is faster in many cases than the standard MPI implementation of IBM. For conclusions like this one it is interesting to know the performance of the gather versions on other machines. Although the described result only concerns the IBM's implementation, the results on the Cray T3E (figure 3) are important. Through our measurements on the Cray T3E we know, that a good implementation of MPI\texttt{Gather} also can be much faster than our alternative implementations. A fact which even amplifies our findings on the IBM.

2.3 Contention in collective operations

Due to the important role of collective operations, as mentioned in sec. 2.2, we started further effort in the measurement of collective operations' performance. Our investigations are twofold. First, we extended our number of measurements: we included measurements of all collective MPI operations. This allows comparisons between defined MPI operations and 'hand-made' counterparts (e.g.,

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{figure3.png}
\caption{Different Gather implementations on the Cray T3E}
\end{figure}
MPI\_Reduce\_Scatter compared with MPI\_Reduce followed by MPI\_Scatterv, or
MPI\_Allreduce compared with MPI\_Reduce followed by MPI\_Bcast. Due to the
oretical results the combined MPI operations often can perform better than the
handwritten counterparts. So comparisons between supplied MPI operations and
their 'handmade' counterparts also assess the quality of the MPI implementation.
In figure 4 the comparison between MPI\_Allreduce and MPI\_Reduce followed by
MPI\_Bcast is shown for the Cray T3E.

![Graph](image)

**Fig. 4.** Faster MPI\_Reduce followed MPI\_Bcast than MPI\_Allreduce at message
length 256 Bytes on the Cray T3E

Second, besides our extension of the number of measurements, we also ex-
tended the data collected during one measurement. The performance of a col-
teptive operations is not only given by the consumed time on the root node. An
other interesting property of a collective operation is its blocking behavior on
all nodes. Although a collective operation only can considered as finished, when
the result is available (in many cases at the root node), it is also interesting
how long each participating processing element needs until the collective op-
eration returns. That is the question, how long each processor is blocked through a
collective operations and when it can proceed. This time is called \textit{node time} in \textsc{SKaMPI} terminology. It is clear that the node times of a collective operations are another interesting topic for comparisons. First results of our investigations show that it is absolutely possible, that a routine which is faster in completion at the root node may use more time on other nodes than a routine which is slower at the root node. In our example in figure 5 \texttt{MPI\_Allreduce} at message length 512 KBytes and 8 nodes is slower than the handmade counterpart regarding the completion at the root node. But this result does no longer hold, when looking at all nodes. Regarding all node times, \texttt{MPI\_Allreduce} needs less time than \texttt{MPI\_Reduce} plus \texttt{MPI\_Bcast}.

![Graph](image)

\textbf{Fig. 5.} Comparison of nodes times between \texttt{MPI\_Allreduce} and \texttt{MPI\_Reduce} followed \texttt{MPI\_Bcast} at message length 512 KBytes

\section{Measurements with different datatypes}

Related to the memory alignment problem (sec. 2.1) is the question which MPI datatypes are used for measurements. Until version 3 of \textsc{SKaMPI} we used always \texttt{MPI\_BYTE}, because this datatype has the same extent and size on all machines, i.e., the results are comparable also between different machines. Unfortunately, this datatype is not as relevant as e.g., \texttt{MPI\_DOUBLE} or \texttt{MPI\_INT} for application programmers. In the current release of \textsc{SKaMPI} (release 3.0) we added datatypes to \textsc{SKaMPI} as a orthogonal concept. That means that we did not added new measurements with different datatypes to \textsc{SKaMPI}. Datatypes as a orthogonal concept means that every suite of measurements in \textsc{SKaMPI} can be performed with a datatype specified as a parameter for this suite. (That corresponds to the MPI standard, where also operations are parameterized with datatypes.) This introduced also a rework of several \textsc{SKaMPI} internals, e.g., when varying over
the message length the automatic parameter refinement [5] works with the unit
of elements of the datatype, whereas the output is given in the unit bytes, to
remain comparable of different machines. Access to the Cray T3E was very
helpful to develop a stable and portable new release of SKaMPI fast.

2.5 User defined datatypes

The MPI standard offers the possibility to the application programmer to create
application specific user defined datatypes. Their usage can improve a programs
readability and maintainability. One drawback of user defined datatypes is their
unknown performance which strongly depends on the deployed MPI implementa-
tion. We reflected this problem of user defined datatypes in the SKaMPI bench-
mark through the possibility to create user defined datatypes and use them for
measurements. Therefore we enhanced the syntax of the SKaMPI configuration
file. E.g. the user write the following statements

@BASTETYPE 1 MPI_INT

MPI_Send-MPI_Recv
{
...    
   Basetype_number 1;
   Send_Datatype_Number = 50;
   Send_Datatype_Number = 50;
...    
}

The user can choose form a list a predefined "constructors". This constructor
uses a basetype to create a user defined datatype from it. E.g. a vector (con-
structor) of MPI_INT (basetype). Basetypes can also be structs, which are given
in a list of triples, like

@BASETYPE1 (2,0,MPI_BYTE),(1,2,MPI_DOUBLE)

This means that at offset 0 we have two MPI_BYTES and at offset 2 we have
one MPI_DOUBLE. This allows the construction of nested datatypes, like vectors
of structs. A detailed discussion of datatypes in SKaMPI and new results can
be found in [6].

3 Typical usage of the Cray T3E

Due to the better network connection, we mainly use the Karlsruhe IBM RS6000/SP
as the general development platform. The Stuttgart Cray T3E is primarily used
for two purposes. On the one hand we perform compatibility and portability
checks with SKaMPI's code after each change. As described in section 2.1 this
lead to significant changes of SKaMPI, which were not motivated by our tests
of the IBM SP or our workstations. These runs typically use 4 to 32 processing
elements and take up to 1 hour.

On the other hand we use the Cray T3E for performance measurements
(e.g., like described in section 2.3) and for selected comparisons against other
MPI implementations (as, for example, shown in section 2.2). These runs can
use up to 256 processing elements. Their run time varies heavily (between 30
min. and more than 2 hours).

4 Future work in the SKaMPI project

In the SKaLib project SKaMPI is packaged in a general framework for bench-
mark development [7]. So SKaMPI's code can be reused easily in other applica-
tions. One application is the measurement of monitoring overhead in instru-
mented MPI operations, as described in [4], which is done in cooperation with
the University of Linz, Austria.

Furtheron, the SKaMPI's kernel functionality will be extended, e.g., two
dimensional measurements.

According to our described experience in the past we consider access to the
Stuttgart Cray T3E as very helpful and important for the further progress in
our project.

References

1. Vasanth Bala, Jehoshua Bruck, Robert Cypher, Pablo Elustondo, Alex Ho, Ching-
Tien Ho, Shlomo Kipnis, and Marc Snir. CCL: A portable and tunable collective
communication library for scalable parallel computers. IEEE Transactions on Par-

2. W. Gropp and E. Lusk. Reproducible measurements of MPI performance character-
istics. In J. J. Dongarra, E. Luede, and Tomas Margalef, editors, Recent advances in parallel virtual machine and message passing interface: 6th European PVM/MPI
Users' Group Meeting, Barcelona, Spain, September 26–29, 1999: proceedings,


4. D. Kranzmueller, R. Reussner, and C. Schrauschläger. Monitor overhead mea-
surement with SKaMPI. In J. J. Dongarra, E. Luede, and Tomas Margalef, edi-
tors, Recent advances in parallel virtual machine and message passing interface: 6th
European PVM/MPI Users' Group Meeting, Barcelona, Spain, September 26–29,

