Achieving *Performance Portability* with *SKaMPI* for High-Performance MPI Programs

Ralf Reussner and Gunnar Hunzelmann

Chair Computer Science for Engineering and Science
Universität Karlsruhe (T.H.)
Am Fasanengarten 5, D-76128 Karlsruhe, Germany.
{reussner|gunnar}@ira.uka.de

Abstract. Current development processes for parallel software often fail to deliver portable software. This is because these processes usually require a tedious tuning phase to deliver software of good performance. This tuning phase often is costly and results in machine specific tuned (i.e., less portable) software. Designing software for performance *and* portability in early stages of software design requires performance data for all targeted parallel hardware platforms. In this paper we present a publicly available database, which contains data necessary for software developers to design and implement portable and high performing MPI software.

1 Introduction

The cost of todays parallel software mostly exceeds the cost of sequential software of comparable size. This is mainly caused by parallel programming models, which are more complex than the imperative sequential programming model. Although the simplification of programming models for parallel and distributed systems is a promising research area, in the next years no change of the dominance of the currently message-passing or shared-memory imperative models for parallel and distributed systems is to be expected. One common problem of the mentioned programming models for parallel hardware is that parallelisation must be stated explicitely (despite a slightly better support for purely data-parallel programs through e.g., Fortran 90 or HPF [KLS+94]). (The need for manual parallelisation arises because compilers do not support parallelisation as good as vectorisation.)

Current software processes for the development of parallel software reflect this need of manual parallelisation in two phases [Fos94].

Design phase: Various ways of problem specific parallelisation are known, such as multi-grid methods, event-driven simulations, specialised numerical algorithms, etc. All these approaches are used during the design phase to create a design which either reflects the parallelism inherent in the problem, or applies parallel algorithms to speed up computations.

Tuning phase: Software development processes for parallel software contain a special tuning phase, which comes after the integration test of the software. In this tuning phase software is optimised for specific hardware platforms. That is, code is evaluated

V.N. Alexandrov et al. (Eds.): ICCS 2001, LNCS 2074, pp. 841–850, 2001.

with measurements, and then several actions are applied: e.g., rearrangement of code, hiding communication latency, replacement of standardised communication operations by more specific ones, etc. These steps are often required to make the software as fast as necessary on the targeted platform. Unfortunately, portability to other platforms is usually lost.

Hence, machine specific performance considerations are not explicitely made during design stage, but only in the tuning phase. Some of the performance measurements done in the tuning phase are specific for the program's design, but other measurements concern the underlying MPI and hardware performance. In this paper we present a public database containing MPI performance data.

MPI as the most widely used standard for developing message passing software manifested *compiling portability*. This means, that the user is able to develop software which just has to be recompiled when changing the hardware platform, opposed to limited prortability when using vendor specific libraries. (Ommiting here all the remaining tiny peculiarities when trying to develop really portable software with the C programming language.)

Due to the above mentioned platform specific program tuning, this compiling portability usually does not help much when writing high performance software. What is really needed is *performance portability*. This term refers to the requirement, that portability for a parallel program does not only mean that it is compilable on serveral platforms, but also that it shows at least good performance on all these platforms without modifications. (A more quantitative notion of portability would take into account (a) the effort of making a program compilable, (b) the effort of showing good (or best) performance, and (c) the really achieved performance before and after modifications.)

In section 2 we present work related to MPI benchmarking. How performance data can be used during the design of parallel software is discussed in section 3. Section 4 contains a description of our performance database. An example is presented in section 5. Finally, section 6 concludes.

2 Related Work

Benchmarking in general and benchmarking communication operations on parallel platforms in particular require a certain caution to ensure the validity of benchmarked results for "normal application programs". Gropp et al. present guidelines for MPI benchmarking [GL99]. Hempel gives some additional advice and formulates as a general benchmarking principle, that benchmarks never should show better results when lowering the performance of the MPI implementation (the benchmarked entity in general) [Hem99]. Most interestingly, he describes a case, where making MPI point to point communication slower resulted in better results of the COMMS1 – COMMS3 suites of PARKBENCH [PAR94]. Currently no benchmark exactly fulfils all these requirements. The mechanisms applied by *SKaMPI* to tackle MPI benchmarking problems are described in [RSPM98,Reu99]. Some new algorithms to benchmark collective operations reliable are given in [dSK00]. The benchmarks which come probably closest to *SKaMPI*'s goals are the following two: A widely used MPI benchmark is the one

shipped with the `mpich`[1] implementation of MPI; it measures nearly all MPI operations. Its primary goal is to validate `mpich` on the given machine; hence it is less flexible than *SKaMPI*, has less refined measurement mechanisms and is not designed for portability beyond `mpich`.

The b_{eff} benchmark of Rabenseifner measures network performance data from a perspective interesting for the application programmer and complements the measures included in *SKaMPI*. The results are publicly available on the web[2]. As intended for users of the Cray T3E and other machines installed at the HLRS in Stuttgart this database of the benchmark does not cover such a wide range of machines as *SKaMPI* does.

The low level part of the *PARKBENCH* benchmarks [PAR94] measure communication performance and have a managed result database[3] but do not give much information about the performance of individual MPI operations.

P. J. Mucci's[4] *mpbench* pursues similar goals as *SKaMPI* but it covers less functions and makes only rather rough measurements assuming a "quite dead" machine.

The *Pallas MPI Benchmark (PMB)*[5] is easy to use and has a simple well defined measurement procedure but has no graphical evaluation yet and only covers relatively few functions.

Many studies measure a few functions in more detail [GHH97,PFG97,RBB97,O.W96] but these codes are usually not publicly available, not user configurable, and are not designed for ease of use, portability, and robust measurements.

As one can imagine, the performance of a parallel computers (especially with a complex communication hardware) cannot be described by one number (even a single processor cannot be specified by one number). So many of the currently used benchmarks (e.g. [BBB+94,PAR94]) may be useful to rank hardware (like in the "top500" list[6]), but do not give much advice for program optimisation or even performance design.

3 Design of Parallel Software with MPI Performance Data Aware

When designing an MPI program for performance *and* portability the following questions arise:

Selection of point to point communication mode? MPI offers four modes for point to point communication: standard, buffered, ready, synchronous. Their performance depends on the implementation (esp. `MPI_Send`, where the communication protocol is intentionally unspecified) and the hardware support of these operations. (The ready mode is said to only be supported by the Intel Paragon... .) Additionally, MPI differentiates between blocking and non-blocking communication. Furthermore, there exist specialised operations like `MPI_SendRecv` for the common case of data exchange

[1] http://www.mcs.anl.gov/Projects/mpi/mpich/
[2] http://www.hlrs.de/mpi/b_eff
[3] http://netlib2.cs.utk.edu/performance/html/PDStop.html
[4] http://www.cs.utk.edu/~mucci/DOD/mpbench.ps
[5] http://www.pallas.de/pages/pmbd.htm
[6] http://www.top500.org

between two MPI processes. Also one is able to use wildcards like `MPI_ANY_TAG` or `MPI_ANY_SOURCE` which modify the behaviour of these operations. The MPI reference [GHLL+98] does a good job explaining the semantics of all these similar operations, but their performance depends highly on the MPI implementation and on hardware support.

Should compound collective operations be used? The MPI standard offers some compound collective operations (like `MPI_Allreduce`, `MPI_Allgather`, `MPI_Reduce-scatter`, `MPI_Alltoall`) which can be replaced by other, more primitive collective MPI operations (e.g., `MPI_Bcast` and `MPI_Reduce`). The compound collective operations are provided by the MPI standard, since it is possible to provide better algorithms for the compound operation than just putting some primitive collectives together. The question for the application software designer is: Is a compound operation worth the effort (e.g., designing particular data structures / partitionings) for using it?

Use of collective operations or hand made operations? Similar to compound collective operations made by more simple collective operations, also simple collective operations can be made solely out of point to point communicating operations. Again, the question arises, whether the MPI library provides a good implementation for the targeted platform. It might be worth the effort to reimplement collective operations for specific platforms with point to point communication in the application program.

Optimality of vendor provided MPI operations? The above question of the optimality of collective operations can be asked for all MPI operations provided by the vendor. Often a vendor provides some optimised MPI operations, while other operations perform suboptimal. Unfortunately these subsets of well-implemented MPI operations vary from vendor to vendor (i.e., from platform to platform).

Knowing the detailed performance of specific MPI operations helps to decide which MPI operation to choose when many similar MPI operations are possible. Of course, the best choice is the MPI operation performing well on all targeted platforms. If such a everywhere-nice operation does not exist, one can decide which operations on which platform must be replaced by hand-made code. Introducing static `#ifdef PLAT-FORM_1 ...` alternatives in the code simplifies the task to create portable software. Code selection during run-time is also feasible (by dynamically querying the platform), but introduces new costs during run-time.

To test the quality of provided MPI operations we reimplemented some operations with naive algorithms. Is the vendor provided implementation worse than these naive approaches, the application programmer can easily replace the vendor implementation. (Additionally, it says a lot about the quality of the vendor's implementation.)

How to use the database to answer the above question questions is shown in the next section, after a brief introduction of the database's terms.

4 The Public Result Database

The detailed design of the database is described in [Hun99]. Summarising the key concepts we present the following terms in a bottom-up order:

Single Measurement: A single measurement contains the MPI operation to perform. It has a right for its own, since it unifies calling different parameterised MPI operations in a unique way.
Measurement: A *measurement* is the accumulated value of repeated single measurement's results: Several single measurements are performed at the same argument (e.g., `MPI_SendRecv` at 1024 Kbytes). Their results are stored in an array. After reducing the influence of outliers by cutting (user defined) quartiles of that array, the average value is taken as the result of the measurement. The number of single measurements to perform before computing the result of the measurements depends on the standard error allowed for this measurement. Attributes of a measurement are: the single measurement to perform, the allowed standard error, the maximum and minimum number of repetitions.
Pattern: *Patterns* organise the way measurements are performed. For example collective operations must be measured completely different from point-to-point communicating MPI operations. *SKaMPI* currently contains four patterns: (a) for point-to-point measurements, (b) for measurements of collective operations, (c) for measuring communication structures arising in the master-worker scheme [KGGK94], and (d) for simple, i.e., one-sided operations like `MPI_Commsplit`. Besides grouping measurements the benefit of patterns lies in the comparability of measurements performed by the same pattern. Attributes of a pattern include the kinds and units of different arguments.
Suite of Measurements: A *suite of measurements* contains all measurements which measure the same MPI operation with the same pattern. (Note that the same MPI operation may be measured with two patterns, such as `MPI_Send` in the point-to-point pattern and the master worker pattern.) In a suite of measurements the measurements are varied over one parameter (such as the message length or the number of MPI processes). Informally spoken, the graph describing the performance of an MPI operation is given by the suite of measurements. The points of the graph represent the measurements. Important attributes of the suite are: the MPI operation, the pattern used, range of parameters, step-width between parameters (if fixed), scale of the axes (linear or logarithmic), use of automatic parameter refinement.
Run of Benchmark: A *run* is the top most relation, it includes several suites of measurements and their common data, such as: a description of the hardware (processing elements, network, connection topology), the date, the user running the benchmark, the operating system (and its version), and the settings of global switches of *SKaMPI*.

The structure of the relations used in the database are shown in figure 1. The *SKaMPI* result database has two web-user-interfaces . One is for downloading detailed reports of the various runs on all machines (http://liinwww.ira.uka.de/∼skampi/cgi-bin/run_list.cgi.pl). The other interface (http://liinwww.ira.uka.de/∼skampi/cgi-bin/frame_set.cgi.pl) is able to compare the operations between different machines according to the user's selection. In the following we describe this user-interface (see figure 2).

After loading the database's web site with a common netbrowser, querying is performed in three steps:

1. Choosing a run. Here you select one or more machine(s) you are interested in. Since on some machines several runs have been performed (e.g., with a different number

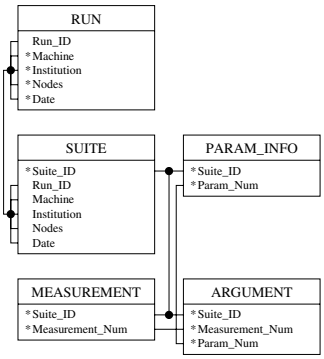

Fig. 1. Design of the *SKaMPI* result database

of MPI processes) you often can choose between several runs of one machine. This selection is performed in the upper left part of the user-interface (see figure 2).

2. Choosing the suites. After you selected some runs, the database is queried for the suites belonging to a run. The available runs are presented in a list for each selected run at the upper right part of the user-interface. There you now can select for each run the suites you are interested in (e.g., `MPI_Reduce`). Of course you may also select different suites on different machines (such as `MPI_GatherSR` in run A and `MPI_Gather` in run B [7].
3. After choosing the suites of interest the database is queried for all relevant measurements. The user-interface creates a single plot for all selected suites. The user is able to download an additional plot in encapsulated postscript (if selected in the previous step). There also exists the possibility to zoom into the plot.

5 Hamlet's Question: Using `MPI_Gather` or Not ?

As an example we discuss a special case of the question "Using collective operations or hand made operations?", as posed in section 3.[8] Here we look at the `MPI_Gather` operation, because one can replace it relatively simple by a hand-made substitute. The `MPI_Gather` operation is a collective operation which collects data from MPI processes at a designated root process. Two extremely naive implementations are provided by the *SKaMPI* benchmark. Both implementations can be programmed easily and fast by any application programmer.

The first implementation (`MPI_GatherSR`) simply uses point to point communication (implemented with `MPI_Send` - `MPI_Recv`) from all processes to the root process.

[7] An enumeration with detailed description of all operations measured by *SKaMPI* and all possible alternative naive implementations of MPI operations can be found in the *SKaMPI* user manual [Reu99]. This report is available online (http://liinwww.ira.uka.de/∼reussner/ib-99-02.ps)

[8] and not by W. Shakespeare, as the heading might suggest.

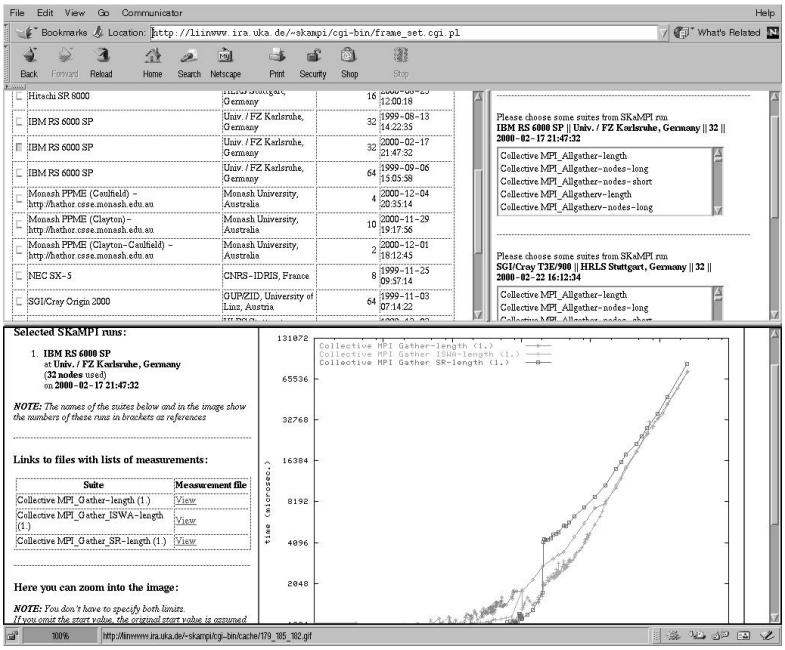

Fig. 2. Interactively querying the result database

Here the root process sequentially posts the receives. A receive is finished when the data arrived; each (but the first) receive has to wait until the previous receive finished (even if its data is ready to receive before).

The second implementation (`MPI_GatherISWA`) is only slightly more clever: the processes use the non-blocking `MPI_ISend` and the root process posts non-blocking receives (`MPI_IRecv`). This allows the root process to receive data from the other processes in the order the data arrives at the root process (assuming that the posting of the non-blocking receives is finished before any data arrived). The root process controls receiving with `MPI_Waitall`.

More sophisticated algorithms of gather operations usually perform much better. Their complexity depends on the topology of the underlying communication network; details can be found in e.g., [KGGK94,KdSF+00].

Consider you are developing an application for two of the most common parallel architectures: an IBM RS 6000 SP and a Cray T3E. Your implementation will make use of `MPI_Gather`. One of many questions is, whether `MPI_Gather` should be replaced by hand-made operations on all machines, or on one machine (which?), or never?

Looking at the *SKaMPI* result database for the Cray T3E shows, that the vendor provided `MPI_Gather` performs much better than the naive implementations. Concerning the results for an IBM RS 6000 SP one has to say that the IBM RS 6000 SP is a family of parallel computers rather than one single machine. Many different processors, connection networks and even topologies exist. Hence, generalising from these results

to the whole family is clearly invalid. But this shows once more the importance detailed performance information. The results presented here are measured consistently on the Karlsruhe IBM RS 6000 SP from 1999 – 2000. [9] These measurements show that the hand-made `MPI_GatherSR` is faster than the provided `MPI_Gather` (by approximately the factor of 1.5) for a short message length of 256 Bytes on two to 32 processes (figure 3 (left)). Regarding the time consumed by the root process of the

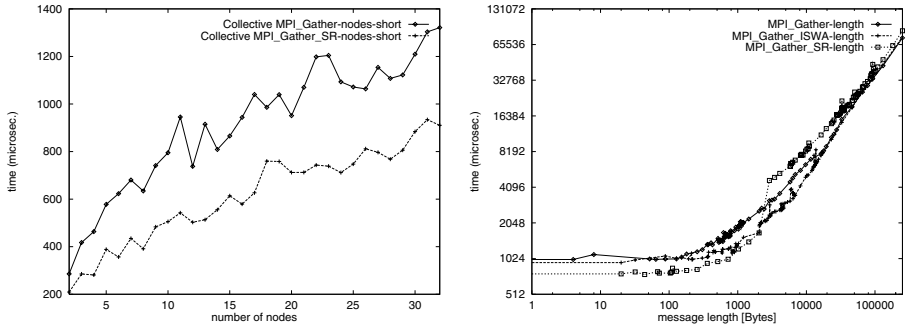

Fig. 3. (Left) Comparison of `MPI_Gather` and `MPI_GatherSR` on an IBM RS 6000 SP varied over the number of nodes at a fixed message length of 256 bytes. (Right) Comparison of `MPI_Gather`, `MPI_GatherSR`, and `MPI_GatherISWA` on an IBM RS 6000 SP with 32 processes varied over the message length

three implementations (`MPI_Gather`, `MPI_GatherSR`, `MPI_GatherISWA`) varied over the message length at the fixed number of 32 MPI processes (figure 3 (right)), we see that `MPI_GatherSR` is fastest if the message length is below 2048. For longer messages the used `MPI_Send` changes its internal protocol from a one-way direct-sending to a two-way request-sending protocol. For message lengths up to 16 KBytes the `MPI_GatherISWA` algorithm is fastest. For longer message lengths the difference between `MPI_Gather` and `MPI_GatherISWA` is not relevant; `MPI_GatherSR` is clearly the slowest. Till now, we looked at the time consumed by the root process. A detailed dicussion on timing collective operation lies beyond the scope of this paper (refer for example to [dSK00]). However, since MPI 1 collective operations are all blocking, we we might consider the non-blocking `MPI_GatherISWA` as useful, if our application allows the processes to perform computations without using (the still occupied) send buffer. The timing for all processes is shown in figure 4. From these results we can draw the conclusion, that we should provide a hand-made replacement for `MPI_Gather` for an IBM RS 6000 SP if we deal with short message lengths (below 10 KBytes). Whether we use the simple send-receive algorithm or the little more sophisticated non-blocking algorithm depends from the message length (smaller or greater 2 KBytes). If the processes can perform calculations without using the message buffers, we can hide communication latency by using the non-blocking `MPI_GatherISWA`. However, it is necessary to switch to the vendor provided `MPI_Gather` implementa-

[9] E.g., http://liinwww.ira.uka.de/∼skampi/skampi/run2/l2h/

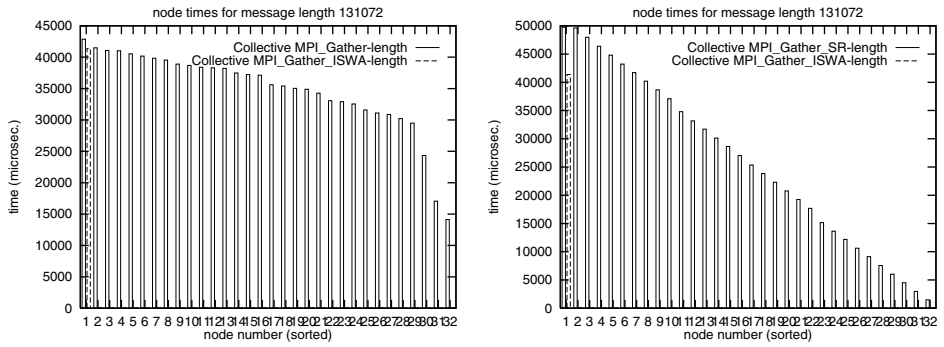

Fig. 4. Times consumed for each single MPI process for the different `MPI_Gather` implementations on an IBM RS 6000 SP.

tion on the Cray T3E (even when communication latency hiding would be possible, the vendor supplied `MPI_Gather` is to be preferred to the `MPI_GatherISWA`).

6 Conclusion

We presented a new process for the development of parallel software which moves the non-functional design considerations of performance and portability to the early stages of design and implementations. This lowers costs, even if only one platform is targeted. This process is supported by a publicly available MPI performance database. How performance data influences the design of MPI programs was discussed and an example for one particular case was presented.

To achieve an amount of standardised software in the parallel computing area more similar to that in the area of sequential programs, parallel software has to increase its portability. Considering this background, the creation of software development processes and tools supporting the engineering of more portable software is really crucial.

URL of the *SKaMPI* project
http://liinwww.ira.uka.de/∼skampi

References

[BBB+94] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. Report RNR-94-007, Department of Mathematics and Computer Science, Emory University, March 1994.

[dSK00] B. R. de Supinski and N. T. Karonis. Accurately measuring mpi broadcasts in a computational grid. In *Proc. 8th IEEE Symp. on High Performance Distributed Computing (HPDC-8)*, Redondo Beach, CA, August 2000. IEEE.

[Fos94] Ian T. Foster. *Designung and Building Parallel Programs – Concepts and Tools for Parallel Software Engineering*. Addison Wesley, Reading, MA, 1994.

[GHH97] V. Getov, E. Hernandez, and T. Hey. Message-passing performance of parallel computers. In *Proceedings of EuroPar '97 (LNCS 1300)*, pages 1009–1016, 1997.

[GHLL+98] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI: The Complete Reference. Volume 1 & 2*. MIT Press, Cambridge, MA, USA, second edition, 1998.

[GL99] W. Gropp and E. Lusk. Reproducible measurements of MPI performance characteristics. In J. J. Dongarra, E. Luque, 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*, volume 1697 of *Lecture Notes in Computer Science*, pages 11–18. Springer-Verlag, 1999.

[Hem99] Rolf Hempel. Basic message passing benchmarks, methodology and pitfalls, September 1999. Presented at the SPEC Workshop (www.hlrs.de/mpi/b_eff/hempel_wuppertal.ppt).

[Hun99] Gunnar Hunzelmann. Entwurf und Realisierung einer Datenbank zur Speicherung von Leistungsdaten paralleler Rechner. Studienarbeit, Department of Informatics, University of Karlsruhe, Am Fasanengarten 5, D-76128 Karlsruhe, Germany, October 1999.

[KdSF+00] N. T. Karonis, B. R. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *Proceedings of the 14th International Conference on Parallel and Distributed Processing Symposium (IPDPS-00)*, pages 377–386, Los Alamitos, May 1–5 2000. IEEE.

[KGGK94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings, Redwood City, CA, 1994.

[KLS+94] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran handbook*. Scientific and engineering computation. MIT Press, Cambridge, MA, USA, January 1994.

[O.W96] C. O.Wahl. Evaluierung von Implementationen des Message Passing Interface (MPI)-Standards auf heterogenen Workstation-clustern. Master's thesis, RWTH Aachen, Germany, 1996.

[PAR94] PARKBENCH Committee, Roger Hockney, chair. Public international benchmarks for parallel computers. *Scientific Programming*, 3(2):iii–126, Summer 1994.

[PFG97] J. Piernas, A. Flores, and J. M. Garcia. Analyzing the performance of MPI in a cluster of workstations based on Fast Ethernet. In *Recent advances in parallel virtual machine and message passing interface: Proceedings of the 4th European PVM/MPI Users' Group Meeting*, number 1332 in Lecture Notes in Computer Science, pages 17–24, 1997.

[RBB97] M. Resch, H. Berger, and T. Bönisch. A comparison of MPI performance on different MPPs. In *Recent advances in parallel virtual machine and message passing interface: Proceedings of the 4th European PVM/MPI Users' Group Meeting*, number 1332 in Lecture Notes in Computer Science, pages 25–32, 1997.

[Reu99] Ralf H. Reussner. SKaMPI: The Special Karlsruher MPI-Benchmark–User Manual. Technical Report 02/99, Department of Informatics, University of Karlsruhe, Am Fasanengarten 5, D-76128 Karlsruhe, Germany, 1999.

[RSPM98] R. Reussner, P. Sanders, L. Prechelt, and M. Müller. SKaMPI: A detailed, accurate MPI benchmark. In V. Alexandrov and J. J. Dongarra, editors, *Recent advances in parallel virtual machine and message passing interface: 5th European PVM/MPI Users' Group Meeting, Liverpool, UK, September 7–9, 1998: proceedings*, volume 1497 of *Lecture Notes in Computer Science*, pages 52–59, 1998.