SKaMPI: A Detailed, Accurate MPI Benchmark

Ralf Reussner[1], Peter Sanders[2], Lutz Prechelt[1], and Matthias Müller[1]

[1] University of Karlsruhe, D-76128 Karlsruhe
[2] Max-Planck Institute for Computer Science, D-66123 Saarbrücken
e-mail: `skampi@ira.uka.de`

Abstract. *SKaMPI* is a benchmark for MPI implementations. Its purpose is the detailed analysis of the runtime of individual MPI operations and comparison of these for different implementations of MPI. *SKaMPI* can be configured and tuned in many ways: operations, measurement precision, communication modes, packet sizes, number of processors used etc. The technically most interesting feature of *SKaMPI* are measurement mechanisms which combine accuracy, efficiency and robustness. Postprocessors support graphical presentation and comparisons of different sets of results which are collected in a public web-site. We describe the *SKaMPI* design and implementation and illustrate its main aspects with actual measurements.

1 The Role of Benchmarking in Parallel Programming

The primary purpose of parallel (as opposed to sequential) programming is achieving high processing speed. There are two basic approaches to parallel programming. The high-level approach tries to maintain ease of programming as much as possible and is exemplified in languages such as HPF [3]. The low-level approach attempts to achieve maximum speed: the programmer devises all distribution of data and processing explicitly and manually optimizes the program for the idiosyncratic behavior of a particular target machine. This approach is exemplified by message passing programming.

Historically, message passing programs were very machine-specific and hardly portable; libraries such as PVM and MPI overcome this deficiency. However, one must not assume that functional portability also means *performance portability*: Often what works on one machine will still work on another, but be (relatively) much slower. Worse yet, the same may also happen when switching from one MPI implementation to another on the same machine.

Significant investments into performance evaluation tools have been made which can help to identify optimization possibilities. However, these tools have a number of limitations. They require all target configurations to be available for development which is often not the case if a program has to perform well on several platforms or if production runs involve more processors than available for development. Also, these tools can only be applied in late development when the performance-critical parts of the program are already working. Finally, the measurement process itself can distort the results, in particular for the most challenging applications which use many processors and fine-grained communication.

Therefore, we consider a benchmark suite covering all important communication functions to be essential for developing efficient MPI-programs. It must be easy to use and produce detailed, accurate, and reliable results in a reasonable amount of time. If standardized measurements for a wide spectrum of platforms are collected in a publicly available data-base, such a benchmark makes it possible to plan for portable efficiency already in the design stage, thus leading to higher performance at lower development cost. At the same time, the benchmark must be flexible enough to be adapted to special needs and expandable, e.g., to incorporate the functionality of MPI-2.

The *SKaMPI* benchmark was designed with these goals in mind. As a side effect, it can be used to compare different machines and MPI-implementations. Section 2 provides an overview of the benchmark architecture. The specific capabilities of *SKaMPI* and mechanisms for measurements are disussed in Section 3. Section 4 illustrates some of the techniques with actual measurement results.

Related work

Benchmarking has always played an important role in high performance computing, but most benchmarks have different goals or are less sophisticated than *SKaMPI*. Benchmarking of applications or application kernels is a good way of evaluating machines (e.g. [1,4]) but can only indirectly guide the development of efficient programs. 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 low level part of the *PARKBENCH* benchmarks [4] measure communication performance and have a managed result database[2] but do not give much information about the performance of individual MPI operations.

P. J. Mucci's[3] *mpbench* pursues similar goals as *SKaMPI* but it covers less functions and makes only rather rough measurements assuming a "dead quite" machine.

The *Pallas MPI Benchmark (PMB)*[4] is easy to use and has a simple well defined measurement procedure but has no graphical evaluation yet and only covers relatively few functions.

[1] `http://www.mcs.anl.gov/Projects/mpi/mpich/`
[2] `http://netlib2.cs.utk.edu/performance/html/PDStop.html`
[3] `http://www.cs.utk.edu/~mucci/mpbench/`
[4] `http://www.pallas.de/pages/pmbd.htm`

Many studies measure a few functions in more detail [2, 5, 6, 8] but these codes are usually not publicly available, not user configurable, and are not designed for ease of use, portability, and robust measurements.

2 Overview of *SKaMPI*

The *SKaMPI* benchmark package consists of three parts: (a) the benchmarking program itself, (b) a postprocessing program and (c) a report generation tool. For ease of portability the benchmarking and postprocessing program are both ANSI-C programs, installed by just a single call of the compiler. The report generator is a Perl script which calls gnuplot and LaTeX.

The `.skampi` run-time parameter file describes all measurements with specific parameters. For a default run, only `NODE`, `NETWORK` and `USER` need to be specified by the user in order to identify the measurement configuration, but very many other customizations are also possible without changing the source code.

The benchmark program produces an ASCII text file `skampi.out` in a documented format [7]; it can be further processed for various purposes.

The postprocessing program is only needed when the benchmark is run several times, refer to Section 3.3 for details.

The report generator reads the output file and generates a postscript file containing a graphical representation of the results. This includes comparisons of selected measurements. The report generator is also adjustable through a parameter file.

Reports (actually: output files) are collected in the *SKaMPI* result database in Karlsruhe[5] where they are fed through the report generator and `latex2html`.

2.1 Structuring measurements

Since we investigate parallel operations, we have to coordinate several processes. Measurements with similar coordination structure are grouped into a *pattern*. Thus, when extending the benchmark to a new MPI function one only has to add a small core function; the measurement infrastructure is automatically reused. We now give an overview of the four patterns used by *SKaMPI*: ping-pong, collective, master-worker, and simple.

The *ping-pong pattern* benchmarks point-to-point communication between a pair of processors. For good reasons, MPI provides many variants, so we currently measure nine cases including `MPI_Send`, `MPI_Ssend`, `MPI_Isend`, `MPI_Bsend`, `MPI_Sendrecv` and `MPI_Sendrecv_replace`. The ping-pong pattern uses two processors with maximum ping-pong latency in order to avoid misleading results on clusters of SMP machines.

The *collective pattern* measures functions such as `MPI_Bcast, MPI_Barrier, MPI_Reduce, MPI_Alltoall, MPI_Scan,` or `MPI_Comm_split`, in which a subset of the processes works together. We synchronize the processors using

[5] `http://wwwipd.ira.uka.de/~skampi/`

`MPI_Barrier`, measure the time on process 0, and subtract the running time of the barrier synchronization.

Ping-pong cannot model performance-relevant aspects such as the contention arising when one processor communicates with several others at once. Some MPI functions like `MPI_Waitsome` are specifically designed for such situations. We measure this using the *master-worker-pattern* which models the following frequent situation: A master process partitions a problem into smaller pieces and dispatches them to several worker processes. These workers send their results back to the master which assembles them into a complete solution. We currently measure nine different implementations focusing either on contention for receiving results or the capability to send out work using different MPI functions.

Finally, we have a *simple pattern*, which measures MPI-operations called on just a single node such as `MPI_Wtime`, `MPI_Comm_rank`, and unsuccessful `MPI_Iprobe`.

3 Measurement Mechanisms

We now describe *SKaMPI*'s approach to efficiently measuring execution times to a given relative accuracy ϵ. The same methods are also useful for other benchmarking purposes.

3.1 A Single Parameter Setting

Each *SKaMPI* result is eventually derived from multiple measurements of single calls to a particular communication pattern, e.g., a ping-pong exchange of two messages of a given length. For each measurement, the number n of repetitions is determined individually to achieve the minimum effort required for the accuracy requested. We need to control both the *systematic* and the *statistical error*.

Systematic error occurs due to the measurement overhead including the call of `MPI_Wtime`. It is usually small and can be corrected by subtracting the time for an empty measurement. Additionally, we warm-up the cache by a dummy call of the measurement routine before actually starting to measure.

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

The total time for all repetitions must be at least `MPI_WTick`$/\epsilon$ in order to adapt to the *finite resolution* of the clock.

Execution time fluctuations are controlled by monitoring the standard error $\sigma_{\bar{x}} := \sigma/\sqrt{n}$ where n is the number of measurements, $\sigma = \sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2/n}$ is the measured standard deviation, and $\bar{x} = \sum_{i=1}^{n} x_i/n$ is the average execution time. The repetition is stopped as soon as $\sigma_{\bar{x}}/\bar{x} < \epsilon$. Additionally, we impose an upper and a lower bound on the number of repetitions.

Under some operating conditions one will observe huge *outliers* due to external delays such as operating system interrupts or other jobs. These can render $\bar{x}$ highly inaccurate. Therefore, we ignore the 25% slowest and the 25% fastest run times for computing the average. Note that we cannot just use the median because its accuracy is limited by the resolution of the clock.

3.2 Adaptive Parameter Refinement

In general, we would like to know the behavior of some communication routine over a range of possible values for the message length m and the number P of processors involved. *SKaMPI* varies only one of these parameters at a time; two-dimensional measurements are written as an explicit sequence of one-dimensional measurements.

Let us focus on the case were we want to find the execution time $t_P(m)$ for a fixed P and message lengths in $[m_{\min}, m_{\max}]$.

First, we measure at $m_{\max}$ and at $m_{\min}\gamma^k$ for all k such that $m_{\min}\gamma^k < m_{\max}$, with $\gamma > 1$. On a logarithmic scale these values are equidistant.

Now the idea is to adaptively subdivide those segments where a linear interpolation would be most inaccurate. Since nonlinear behavior of $t_P(m)$ between two measurements can be overlooked, the initial stepwidth γ should not be too large ($\gamma = \sqrt{2}$ or $\gamma = 2$ are typical values). Fig. 1 shows a line segment between measured points (m_b, t_b) and (m_c, t_c) and its two surrounding segments. Either of the surrounding segments can be extrapolated to "predict" the opposite point of the middle segment.

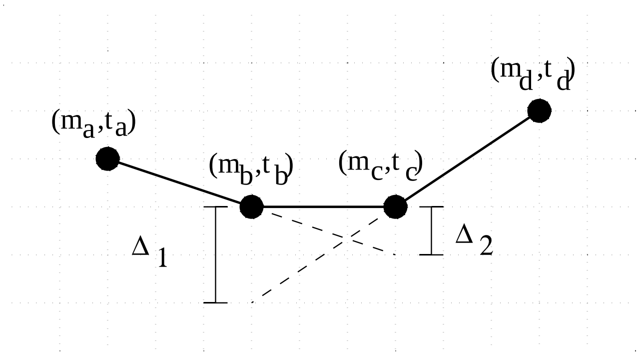

Fig. 1. Deciding about refining a segment $(m_b, t_b) - (m_c, t_c)$.

Let Δ_1 and Δ_2 denote the prediction errors. We use $\min(|\Delta_1|/t_b, |\Delta_2|/t_c, (m_c - m_b)/m_b)$ as an estimate for the error incurred by not subdividing the middle segment.[6] We keep all segments in a priority queue. If m_b and m_c are the abscissae of the segment with largest error, we

[6] We also considered using the maximum of $|\Delta_1|/t_b$ and $|\Delta_2|/t_c$ but this leads to many superfluous measurements near jumps or sharp bends which occur due to changes of communication mechanisms for different message lengths.

subdivide it at $\sqrt{m_b m_c}$. We stop when the maximum error drops below ϵ or a bound on the number of measurements is exceeded. In the latter case, the priority queue will ensure that the maximum error is minimized given the available computational resources.

3.3 Multiple Runs

If a measurement run crashed, the user can simply start the benchmark again. *SKaMPI* will identify the measurement which caused the crash, try all functions not measured yet, and will only finally retry the function which led to the crash. This process can be repeated.

If no crash occurred, all measurements are repeated yielding another output file. Multiple output files can be fed to a postprocessor which generates an output file containing the medians of the individual measurements. In this way the remaining outliers can be filtered out which may have been caused by jobs competing for resources or system interrupts taking exceptionally long.

4 Example Measurements

Naturally, it is difficult to demonstrate the entire range of functions measured by *SKaMPI*. We therefore refer to `http://wwwipd.ira.uka.de/~skampi` for details and concentrate on two condensed examples of particularly interesting measurements.

Fig. 2 shows the execution time of `MPI_Reduce` on 32 processors of an IBM SP (120 MHz POWER-2 processors) and a Cray T3E with 450 MHz processors. While the Cray achieves an order of magnitude lower latency for small messages, the IBM has higher bandwidth for long messages. Apparently, at message lengths of about 24KByte, it switches to a pipelined implementation. The measurement shows that for this machine configuration, the switch should happen earlier. Similar effects can be observed for `MPI_Bcast` and `MPI_Alltoall`. In an application where vector-valued collective operations dominate communication costs (which are not uncommon) fine-tuning for the IBM-SP might therefore require to artificially inflate certain vectors.

Fig. 3 compares `MPI_Send` and `MPI_Ssend` on an IBM SP and on a Cray T3E. `MPI_Ssend` has twice as much peak bandwidth than `MPI_Send` on the Cray while the latter incurs a lower startup overhead. Hence, for optimal performance one needs to have an idea about the message size going to be used in order to select the right function. The IBM apparently switches to `MPI_Ssend` automatically.

5 Conclusions and Future Work

The *SKaMPI* benchmark infrastructure provides fast and accurate individual measurements, adaptive argument refinement, and a publicly accessible database of detailed MPI measurements on a number of platforms. This makes *SKaMPI*

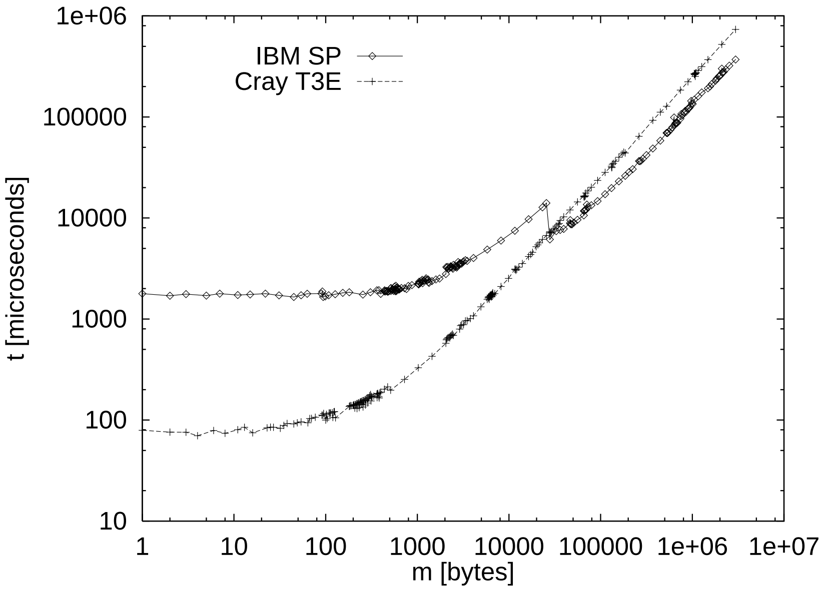

Fig. 2. `MPI_Reduce` execution time by message length on the IBM SP and Cray T3E.

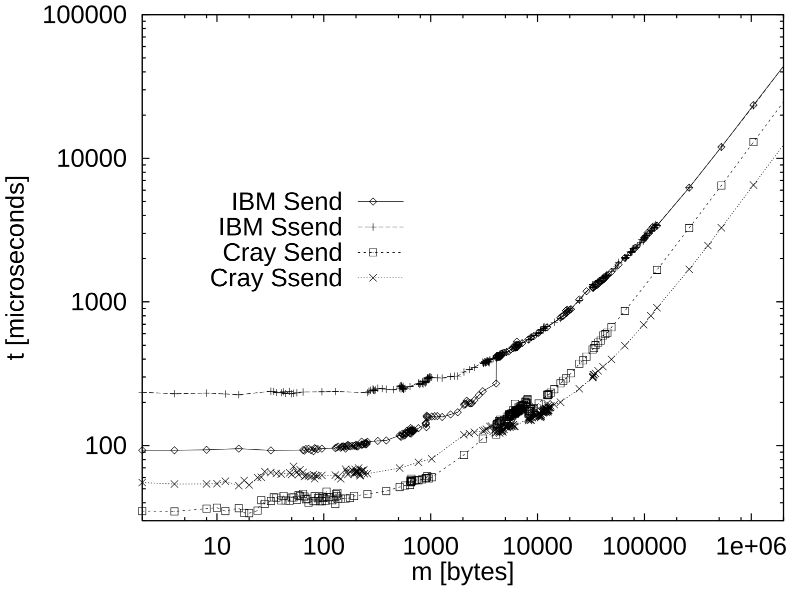

Fig. 3. Point-to-point communication on the IBM SP and Cray T3E.

a unique tool for the design of performance-portable MPI programs. The data can also be used to evaluate machines and MPI implementations or to build quantitative performance models of parallel programs.

Nevertheless, many things remain to be done. We are continuing to refine the measurement mechanisms. These are also useful for other applications so that it might make sense to factor them out as a reusable library. *SKaMPI* should eventually encompass an even more complete set of measurements including MPI-2 and more complicated communication patterns including the communication aspects of common application kernels. The current report generator is only a first step in evaluating the results. More compact evaluations including comparisons of different machines could be generated automatically and we even might derive piecewise closed form expressions for the cost of the functions which could then be used in sophisticated adaptive programs which automatically choose the most efficient algorithm depending on problem size and machine configuration.

References

1. D. Bailey, E. Barszcz, J. Barton, D. Browning, and R. Carter. The NAS parallel benchmarks. Technical Report RNR-94-007, RNR, 1994.
2. Vladimir Getov, Emilio Hernandez, and Tony Hey. Message–Passing Performance on Parallel Computers. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro–Par '97*, pages 1009–1016, New York, 1997. Springer. LNCS 1300.
3. Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele, and Mary E. Zosel. *The High Performance Fortran Handbook*. MIT Press, Cambridge, MA, 1994.
4. Parkbench Committee. Public International Benchmarks for Parallel Computers. *Scientific Programming*, 3(2):101–146, 1994. Report 1.
5. J. Piernas, A. Flores, and J. M. Garcia. Analyzing the performance of MPI in a cluster of workstations based on fast ethernet. In *Fourth European PVM/MPI User's Group Meeting*, pages 17–24, New York, November 1997. Springer. LNCS 1332.
6. M. Resch, H. Berger, and T. Boenisch. A comparison of MPI performance in different MPPs. In *Fourth European PVM/MPI User's Group Meeting*, pages 17–24, New York, November 1997. Springer. LNCS 1332.
7. R. Reussner. Portable Leistungsmessung des Message Passing Interfaces. Diplomarbeit, Universität Karlsruhe, Germany, 1997.
8. C.O. Wahl. Evaluierung von Implementationen des Message Passing Interface (MPI)-Standards auf heterogenen Workstation-clustern. Diplomarbeit, RWTH Aachen, Germany, 1996.