CORE
Metadata, citation and similar papers at core.ac.uk
Provided by MUCC (Crossref)

Scientific Programming 10 (2002) 55–65
IOS Press

SKaMPI: a comprehensive benchmark for public benchmarking of MPI

Ralf Reussner[a,*], Peter Sanders[b] and Jesper Larsson Träff[c]
[a]*Distributed Systems Technology Center (DSTC), Monash University Caulfield Campus, 900 Dandenong Road, Caulfield East, VIC 3145, Australia*
E-mail: reussner@acm.org
[b]*Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, D-66123 Saarbrücken, Germany*
E-mail: sanders@mpi-sb.mpg.de
[c]*C&C Research Laboratories, NEC Europe, Rathausallee 10, D-53757 Sankt Augustin, Germany*
E-mail: traff@ccrl-nece.de

Abstract: The main objective of the MPI communication library is to enable *portable parallel programming* with high performance within the message-passing paradigm. Since the MPI standard has no associated performance model, and makes no performance guarantees, comprehensive, detailed and accurate performance figures for different hardware platforms and MPI implementations are important for the application programmer, both for understanding and possibly improving the behavior of a given program on a given platform, as well as for assuring a degree of predictable behavior when switching to another hardware platform and/or MPI implementation. We term this latter goal *performance portability*, and address the problem of attaining performance portability by benchmarking. We describe the SKaMPI benchmark which covers a large fraction of MPI, and incorporates well-accepted mechanisms for ensuring accuracy and reliability. SKaMPI is distinguished among other MPI benchmarks by an effort to maintain a public performance database with performance data from different hardware platforms and MPI implementations.

1. Introduction

The *Message-Passing Interface* (MPI) [6,15,29] is arguably the most widespread communication interface for writing dedicated parallel applications on (primarily) distributed memory machines.[1] The programming model of MPI is a distributed memory model with explicit message-passing communication among processes, coupled with powerful collective operations over sets of processes. MPI ensures portability of application programs to the same extent that the supported application programming languages C, C++, and Fortran are portable, and is carefully designed to be efficiently implementable on a wide variety of hardware platforms. Indeed, many high-quality vendor implementations achieve MPI communication performance close to that of their native communication subsystem.

Apart from basic semantic properties (liveness etc.) there are no performance model or performance guarantees associated with MPI, and the MPI standard stipulates no performance requirements for a valid MPI implementation. Without empirical performance data it is therefore not possible to predict/analyze the performance of a parallel application using MPI, much less to predict and obtain good performance when moving to another platform and/or MPI implementation. Reliable figures for performance characteristics of MPI implementations for as many different platforms as possible are indispensable to guide the design of efficient and *performance portable* parallel applications with MPI. Performance characteristics include the "raw" performance of MPI communication primitives, both for message-passing and collective communication for

*The work described in this paper was done while this author was at Lehrstuhl Informatik für Ingenieure und Naturwissenschaftler, Universität Karlsruhe, Germany

[1] Throughout this paper MPI denotes the message-passing core of the interface, MPI-1 [29]. We expressly say so when addressing MPI-2 extensions.

ISSN 1058-9244/02/$8.00 © 2002 – IOS Press. All rights reserved

varying parameters (message lengths, number of processes), performance under "load" (e.g. bisection bandwidth) or with typical communication patterns (e.g. master-slave, ring), as well as comparative measurements of different realizations of collective operations. Such information allows the application programmer to tune his application for a specific platform by choosing the appropriate communication primitives, and to tune for good performance across different platforms.

There are several benchmarks for MPI which partly address these issues. In this paper we describe the *Special Karlsruher MPI* benchmark, SKaMPI, which in particular addresses the issue of cross platform performance portability by maintaining a *public performance database* of performance measurements for different platforms. Some main features of SKaMPI are:

- Coverage of (almost) all of the MPI standard, including collective operations and user-defined datatypes.
- Assessment of performance under different communication patterns, e.g. ping-pong and master-slave.
- Automatic parameter refinement for accuracy, reliability and speed of benchmarking.
- Operation controlled by configuration files, which allow for detailed and flexible planning of experiments; the benchmark comes with a default set of measurement suites.
- A report generator, which allows for automatic preparation of measurements into a readable form.
- Last but not least a *public performance database* available on the WWW, which allows for interactive comparison of MPI performance characteristics across different implementations and platforms.

The SKaMPI project was initiated by Ralf Reussner, Peter Sanders, Lutz Prechelt and Matthias Müller at the University of Karlsruhe in 1996-97 [23,25,28], and has since then developed with new features and broader MPI coverage [26]. The interactive WWW-database was implemented by Gunnar Hunzelmann [11,24].

URL of the SKaMPI-project:
http://liinwww.ira.uka.de/~skampi/

1.1. Related work

Benchmarking has always played an important role in high-performance computing. For MPI, several benchmarks exist which differ in philosophy, goals, and level of ambition. In this section we briefly review some other well-known MPI benchmarks in relation to SKaMPI; the discussion is not meant to be exhaustive. A general discussion of problems and pitfalls in (MPI) benchmarking can be found in [7] and [10]; SKaMPI adheres to the sound advice of these papers.

Benchmarking of application kernels [1,2,18] is traditionally used to get an idea of the overall performance of a given machine, but such benchmarks measure communication in a specific, complex context and can only indirectly be used to guide the development of efficient programs.

A widely used MPI benchmark is the `mpptest` shipped with the MPICH implementation of MPI [8, 16]; it measures nearly all MPI operations, but is less flexible than SKaMPI and has limited coverage of user-defined datatypes. The low-level part of the *PARKBENCH* benchmarks [18] measures communication performance and provides a result database, but does not give much information about the performance of individual MPI operations. The MPI part of P.J. Mucci's *Low-Level Characterization Benchmarks* (*LLCbench*) [13], `mpbench`, pursues similar goals to SKaMPI, but it covers only a part of MPI and makes rather rough measurements assuming a "dead" machine. The *Pallas MPI Benchmark* (*PMB*) [17] is easy to use and has a simple, well-defined measurement procedure, but covers relatively few functions, and offers no graphical evaluation. PMB is one of the few MPI benchmarks that covers MPI-2 functionality (one-sided communications, some MPI-I/O). Rolf Rabenseifner's *effective bandwidth benchmark* [4] attempts to give a realistic picture of the achievable communication bandwidth. Bandwidth is measured by a ring pattern over all processes, which is implemented using both simple send and receive operations, as well as by a collective `MPI_Alltoallv` operation. Results from a number of high-performance platforms are publicly available. The effective bandwidth benchmark has recently been complemented by a similar I/O benchmark [20,21].

A comparison of SKaMPI, PMB, `mpptest` and `mpbench` benchmarks is given in [14] for benchmarking MPI on an SGI Origin2000. The benchmarks give roughly similar results, but differ in finer details due to different assumptions on use of cache, and placement and size of communication buffers. The `mpbench` is confirmed to be sensitive to other activities on the machine.

Many studies measure selected functions in more detail [5,19,22] but the codes are often not publicly available, not user configurable, and not designed for ease of use, portability, and robust measurements.

2. Performance considerations

MPI is an extensive interface and communication can be expressed in many different ways. MPI offers two basic types of communication: point-to-point message-passing where information is passed explicitly between a sending and a receiving process, and collective communication where a set of processes jointly performs a communication operation, possibly involving computation as in MPI_Reduce. For the applications programmer this raises a number of questions to be answered in order to get the best possible performance on a given platform/MPI implementation, as well as for being able to obtain and/or predict performance when moving to a different platform/MPI implementation.

Selection of communication mode: MPI differentiates between blocking and non-blocking point-to-point communication, which can be further adapted by different *communication modes*: standard, synchronous, ready and buffered. There also exist specialized compound operations like MPI_Sendrecv for simultaneous sending and receiving of data. It is possible to receive non-deterministically by using wildcards like MPI_ANY_TAG and/or MPI_ANY_SOURCE. The performance of point-to-point communication thus depends on the application context, the MPI implementation, and hardware capabilities that may allow especially efficient implementations of some of these primitives in special contexts.

Use of collective operations: A number of collective operations are available, but are not always used, either because they are not sufficiently known, or because their implementation is distrusted by some users. The question is whether an available MPI library provides good implementations. Do the implementations compare favorably to simple(r), *ad hoc* point-to-point based implementations?

Use of compound collectives: The MPI standard offers certain compound collective operations (MPI_Allreduce, MPI_Allgather, and others) that can easily be expressed in terms of more primitive collectives (e.g. MPI_Reduce, MPI_Gather, and MPI_Bcast). These compound operations are included in MPI since better algorithms than simple concatenation of more primitive collectives exist. Is this exploited in a given MPI implementation?

MPI user-defined datatypes: MPI has a powerful mechanism for working with user-structured, possibly non-consecutive data, but not all MPI implementations support user-defined datatypes equally well [9,30,26]. Is the best performance achieved by maintaining non-consecutive data "manually" or by relying on the MPI mechanism?

3. The SKaMPI benchmark

The SKaMPI benchmark package consists of three parts: the skampi.c benchmarking program itself, an optional post-processing program (also a C program), and a report generation tool (a Perl script). It is complemented by an interactive public database of benchmark results, accessible through the WWW. A run of SKaMPI is controlled by a configuration file, .skampi, which can be modified for more selective or detailed benchmarking. A default configuration file defines a standard run of the benchmark, and the results from such a run can be reported to the SKaMPI database.

The configuration file starts with a preamble identifying the benchmarker, the MPI implementation, and the machine and network used. It defines output and logfiles, and sets various default values. For the standard configuration file only the @MACHINE, @NODE, @NETWORK, @MPIVERSION and @USER fields have to be modified. The @MEMORY field controls the total size of communication buffers per processor (in KBytes). Figure 1 shows a sample configuration file. We refer to this example in the following.

The benchmark program produces an ASCII text file skampi.out (selected by @OUTFILE) in a documented format [27]; it can be further processed for various purposes. The post-processing program is only needed when the benchmark is run several times (see Section 4.3). Post-processing can also be done by SKaMPI itself by setting @POSTPROC to yes. 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 can also be customized via a parameter file. Reports (actually: output files) are collected in the SKaMPI result database in Karlsruhe, which can be queried for both textual and graphical presentation of results (including downloadable encapsulated postscript figures).

The @MEASUREMENT keyword starts the description of the actual experiments to be performed. This is

```
@MACHINE IBM SP
@NODE thin
@NETWORK hpf-switch3
@MPIVERSION aix-mpi library
@USER R. Reussner
@MEMORY 8192
@OUTFILE skampi.out
@LOGFILE skampi.log
...
@POSTPROC no
@CACHEWARMUP 5
@BASETYPE1 MPI_INT
@MEASUREMENTS
MPI_Send-MPI_Recv-dynamicVector1
{
  Type = 1;
  Basetype_Number = 1;
  Send_Datatype_Number = 50;
  Receive_Datatype_Number = 50;
  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 = 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;
}
```

Fig. 1. A `.skampi` configuration file with a one-measurement suite.

Pattern	Type	MPI operations
Ping-pong	1	MPI_Send – MPI_Recv
	2	MPI_Send – MPI_Recv(MPI_ANY_TAG)
	3	MPI_Send – MPI_Irecv
	4	MPI_Send – MPI_Iprobe – MPI_Recv
	5	MPI_Ssend – MPI_Recv
	6	MPI_Isend – MPI_Recv
	7	MPI_Bsend – MPI_Recv
	8	MPI_Sendrecv
	9	MPI_Sendrecv_replace
	34	MPI_Issend
Collective	17	MPI_Bcast
	18	MPI_Barrier
	19	MPI_Reduce
	20	MPI_Alltoall
	21	MPI_Scan
	22	MPI_Comm_split
	23	memcpy
	33	MPI_Gather
	35	MPI_Scatter
	36	MPI_Allreduce
	37	MPI_Reduce – MPI_Bcast
	38	MPI_Reduce_scatter
	39	MPI_Allgather
	40	MPI_Scatterv
	41	MPI_Gatherv
	42	MPI_Allgatherv
	43	MPI_Alltoallv
	44	MPI_Reduce – MPI_Scatterv
	45	Gatherv by MPI_Send – MPI_Recv
	46	Gatherv by MPI_Isend MPI_Irecv – MPI_Waitall
Master-Worker	10	MPI_Waitsome
	11	MPI_Waitany
	12	MPI_Recv(MPI_ANY_SOURCE)
	13	MPI_Send
	14	MPI_Ssend
	15	MPI_Isend
	16	MPI_Bsend
Simple	24	MPI_Wtime
	25	MPI_Comm_rank
	26	MPI_Comm_size
	27	MPI_Iprobe (unsuccessful)
	28	MPI_Buffer_attach

Fig. 2. The instances (column `Type`) of the SKaMPI patterns.

a list of named *measurement suites*, each of which controls a set of measurements to be performed. The configuration file in Fig. 1 lists only the single suite named `MPI_Send-MPI_Recv-dynamicVector1`. Each *suite* has a `Type` which identifies the *pattern* used to control and time the execution of the MPI operations to be benchmarked. More precisely a type is an *instance* of one of the four SKaMPI patterns to a specific combination of MPI functions. Individual measurements with given parameters are repeated a number of times determined by default value settings (`Max_Repetition`, `Min_Repetition`) and by SKaMPI's adaptive parameter refinement mechanism. The set of measurements to be performed by a suite is furthermore determined by the selected dimension to be varied along, which can be either message length, number of processes, or number of chunks (for the master-worker pattern). For variation along message length, interval and stepwidth must be given (`Start_Argument`, `End_Argument`, and `Stepwidth`). The overall time which should be spend measuring this suite can be set in `Time_Suite`. Table 2 lists the currently existing types of suites and the patterns to which they belong. A SKaMPI *run* is the results for the whole list of suites.

3.1. *The patterns*

The way execution times are measured and reported, and the way a set of measurements is coordinated is determined by a so-called *pattern*, of which SKaMPI currently has four. These predefined *measurement strate-*

gies make it easy to extend SKaMPI with new *pattern instances* to cover new MPI functions and/or MPI functions in new contexts. Only a small core function with the proper MPI calls has to be written; the measurement infrastructure of the pattern is automatically reused. All current pattern instances are listed in Table 2.

The ping-pong pattern coordinates point-to-point communication between a pair of processors. The ping-pong exchange is between two processors with maximum ping-pong latency; this in order to avoid misleading results on clusters of SMP machines. SKaMPI automatically selects such a pair based on measurements. Time is measured for one of the processes. Parameter variation is on message `Length`. There are currently 9 instances of the ping-pong pattern, corresponding to (some of) the possible combinations of blocking and non-blocking communication calls under different modalities.

The collective pattern measures operations that are collective in the sense that all processors play a symmetric role. Processors are synchronized with `MPI_Barrier`. Execution time is measured on process 0 (the root), and the running time of the barrier synchronization is subtracted. Parameter variation can be either on message `Length` or number of `Nodes`. There are instances of the collective pattern for all collective MPI communication operations, for collective bookkeeping operations like `MPI_Comm_split`, and for some collectives implemented with point-to-point communication (e.g. gather-functions). There is also an instance of the collective pattern for measuring the performance of the `memcpy` function. This can be used to compare memory bandwidth with communication performance.

Master-worker pattern: Certain performance-relevant aspects such as the contention arising when one processor simultaneously communicates with several other processors cannot be captured by the ping-pong pattern. To compensate for this a *master-worker-pattern* is introduced. A master process partitions a problem into smaller chunks and dispatches them to several worker processes. These workers send their results back to the master which assembles them into a complete solution. Time is measured at the master process. Variation on message `Length`, number of `Chunks`, and number of `Nodes` is possible. Currently 7 instances of this pattern are implemented.

The simple pattern measures MPI-operations with *local completion semantics* such as `MPI_Wtime`, `MPI_Comm_rank`, and unsuccessful `MPI_Iprobe`. No parameter variation is possible.

3.2. User-defined datatypes

The MPI user-defined datatypes is a mechanism for describing the layout of user data in memory to the MPI implementation. All MPI communication operations can operate with complex data described by a user-defined datatype. Communication performance is usually dependent on the datatype, and the extent to which different MPI implementations work well with user-defined datatypes is known to vary. To be able to assess the quality of the datatype handling, SKaMPI incorporates a set of datatype patterns that is orthogonal to the communication pattern instances.

In SKaMPI the data used in a measurement suite are structured according to either a *base type* or a *datatype pattern* over a base type. Base types are defined in the preamble (`@BASETYPE1, ...`), and can be either a built-in MPI type (e.g. `MPI_INT`, `MPI_CHAR`, `MPI_DOUBLE`), or a *simple structure* given by a sequence of triples (c_i, o_i, t_i) each consisting of repetition count, offset, and an built-in MPI type. As the unit of communication either a base type or a type pattern over a base type is selected (`Basetype_Number`, `Send_Datatype_Number`, and `Recv_Datatype_Number`). SKaMPI contains a number of fixed type patterns, including instances of all MPI type constructors, as well as various nested types. Type patterns can be further customized in the preamble. All type patterns are constructed to have the same *size*, i.e. encompass the same amount of data. Therefore send and receive type can be chosen independently, and can be different type patterns. This gives rich possibilities to gauge the handling of user-defined datatypes of a given MPI implementation. The datatype patterns are described in more detail in [26].

4. Measurement mechanisms

We now describe SKaMPI's approach to efficiently measure execution times to a given relative accuracy ϵ. The standard error is set in each suite by the `Standard_error` parameter.

4.1. A single parameter setting

Each SKaMPI result is eventually derived from multiple measurements of single calls to particular MPI functions as determined by the pattern instances, e.g. in the corresponding ping-pong pattern instance measurements of an `MPI_Send` followed by an `MPI_Recv` call for given message lengths. For each measurement, the number n of repetitions needed to achieve the required accuracy with the minimum effort is determined individually. We need to control both the *systematic* and the *statistical error*.

A *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, the user can choose to "warm up" the cache by setting the number of dummy `@CACHEWARMUP` calls to the MPI functions before actual measuring is started.

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, upper and lower bounds on the number of repetitions are imposed, `Min_Repetition` and `Max_Repetition`. Under some operating conditions one will observe huge *outliers* due to external delays such as operating system interrupts or other jobs. These can make $\bar{x}$ highly inaccurate. Therefore, we ignore the slowest and fastest run times before computing the average, as determined by `Cut_Quantile`. Note that we cannot just use the median of the measured values, because its accuracy is limited by the resolution of the clock.

4.2. Adaptive parameter refinement

In general we would like to know the behavior of a 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 must be written as an explicit sequence of one-dimensional measurements.

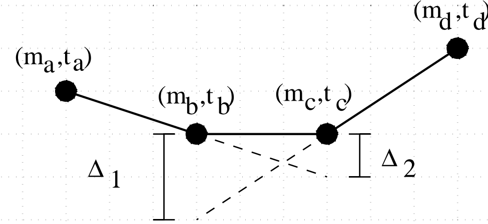


Fig. 3. Deciding on the refinement of a segment $(m_b, t_b) - (m_c, t_c)$.

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). Figure 3 shows a line segment between measured points (m_b, t_b) and (m_c, t_c) and its two surrounding segments. Either of the surrounding segments can be extrapolated to "predict" the opposite point of the middle segment.

Let 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. The reason for the last term in the minimum is to avoid superfluous measurements near sharp jumps in running times which often occur where an MPI implementation switches to a different communication protocol. We keep all segments in a priority queue. If m_b and m_c are the abscissae of the segment with largest error, we subdivide it at $\sqrt{m_b m_c}$. We stop when the maximum error drops below ϵ or the upper 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.

4.3. 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 suites not measured yet, and will finally retry the suite which led to the crash. This process can be repeated.

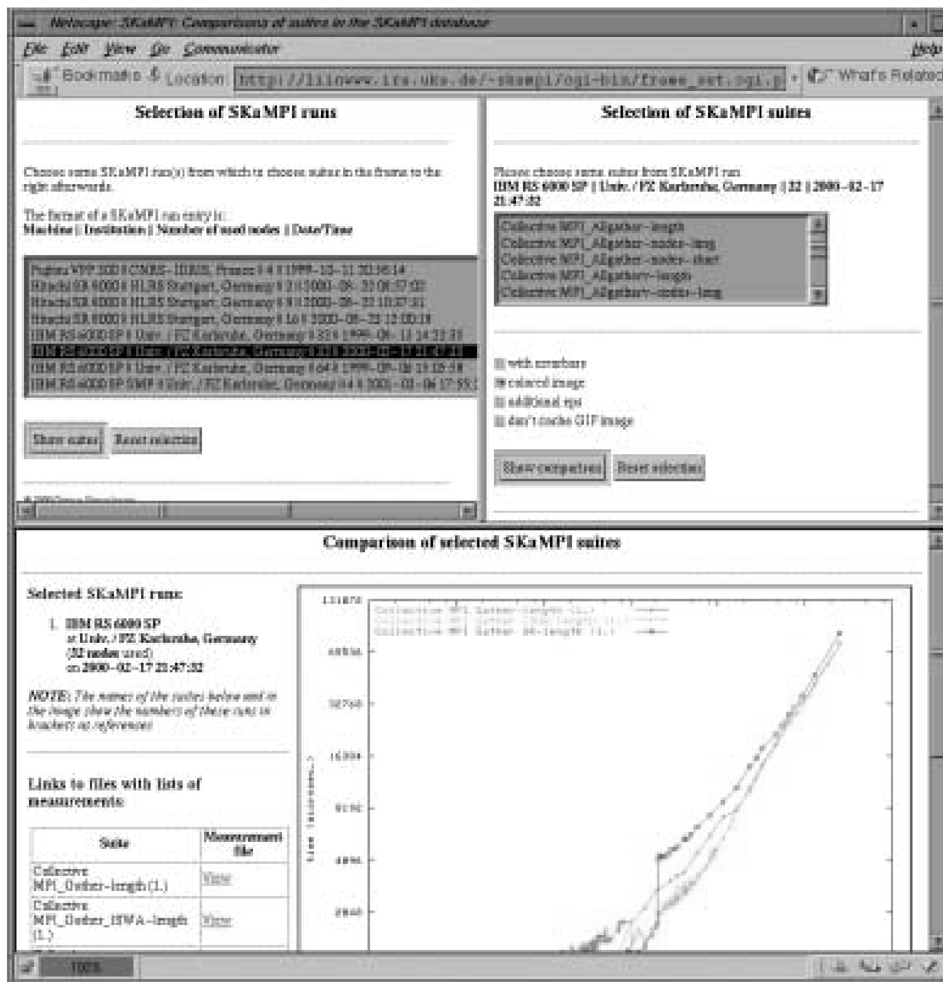


Fig. 4. The WWW interface for querying the result database.

If no crash occurred, all measurements are repeated yielding another output file. Multiple output files can be fed to a post-processor 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.

4.4. Cache behavior

Whether data is sent from cache or not can make a large difference in communication performance. A benchmark that is used for program design should therefore state its assumptions regarding the cache content during communication. In SKaMPI the assumption is that code, MPI internal data, and user data will as far as possible reside in cache. SKaMPI therefore provides the simple means of "warming up" the cache outlined above. The main reason for "warming up" is that this avoids the effect that the first single measurement takes much longer than later repetitions so that the variance of the measured times is increased. Communication of data outside the cache can be measured by choosing message lengths that are larger than the size of the cache. A different caching assumption is made by `mpptest`, which makes an effort to ensure that the user data sent is not in the cache [7].

5. The result database

The SKaMPI result database has two WWW user-interfaces. One is for downloading detailed reports of the various runs on all machines. The other interface

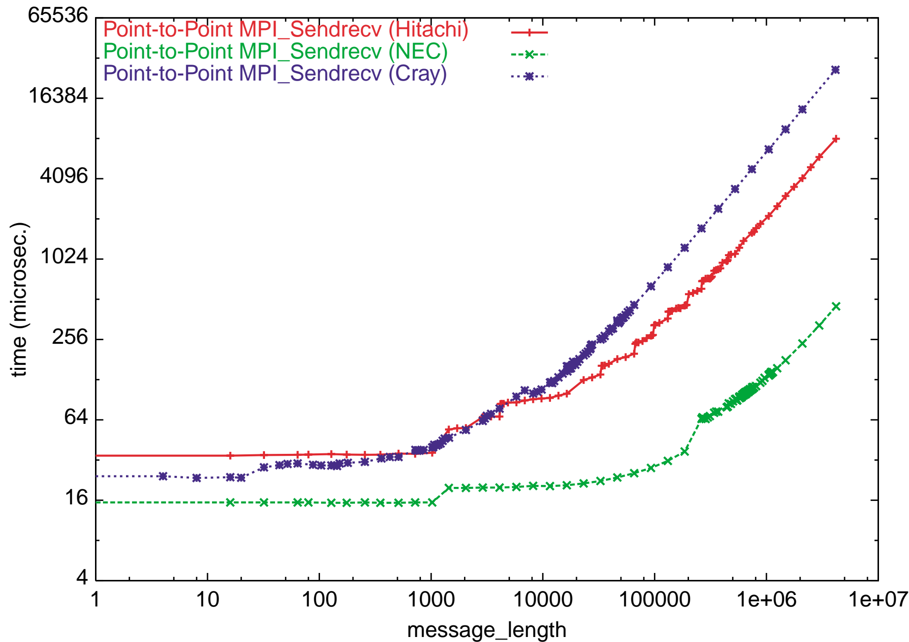


Fig. 5. Performance of compound send-receive for the Hitachi SR 8000, Cray T3E and NEC SX-5.

enables interactive comparison of measurements for different pattern instances/machines. A snapshot of this interface is shown in Fig. 4. Querying the database takes three steps:

1. Choosing a run. The user selects one or more machine(s) of interest. On some machines several runs may have been performed, for example with different number of MPI processes, and it is possible to choose among these. Selection of runs is done in the upper left part of the browser window.
2. Choosing the suites. After selecting the runs the database is queried for the suites of these runs. The available suites for each selected run are presented at the upper right part of the browser-window. For each run the suites of interest are selected; results from different runs can be combined.
3. Finally the database is queried for the selected measurements, and a single plot for all selected suites is created. The plot is shown in the lower half of the browser window, and can also be downloaded as an encapsulated postscript file. It is also possible to zoom into the plot.

The detailed design of the database is described in [11].

6. Examples

The public database currently has results for the default set of suites for Fujitsu VPP 300, Hitachi SR 8000, IBM RS 6000, NEC SX-5, SGI Origin 2000, Cray T3E. All results are supplied by users, who have run SKaMPI on their machine. We give three examples of the use of the benchmark database.

In Fig. 5 the performance of the `MPI_Sendrecv` primitive as measured with ping-pong pattern instance 8 on the Hitachi SR8000, the Cray T3E, and the NEC SX-5 is given. The suite varies on message length, and shows many cases of adaptive parameter refinement (high concentration of measurements). The shapes of the curves are sufficiently similar that `Sendrecv` will probably not pose problems when porting applications among these machines.

In Fig. 6 we compared four different ping-pong pattern instances on the Hitachi SR 8000, namely blocking send-recv, non-blocking either send or receive, and compound send-receive. It is worth noticing that the compound send-receive is clearly better than the other alternatives, about a factor 2 for short messages up to 1 KBytes. For this machine it thus seems advisable to use `MPI_Sendrecv` wherever possible. Similar, or even more complicated pictures appear for the other machines; in particular, the advice to always use `MPI_Sendrecv` is not universally true. The user may

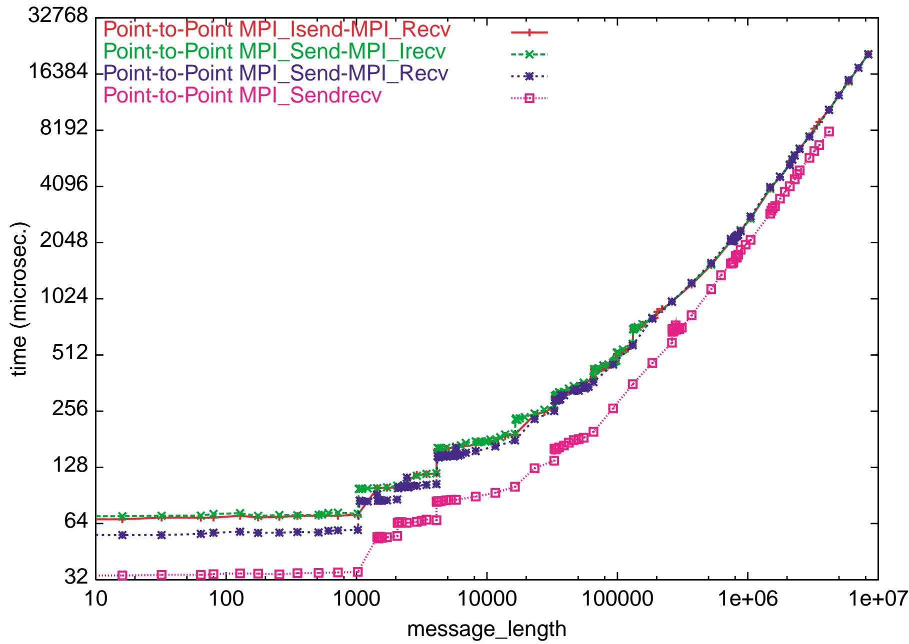


Fig. 6. Performance of different combinations of blocking and non-blocking send and receive operations on the Hitachi SR 8000.

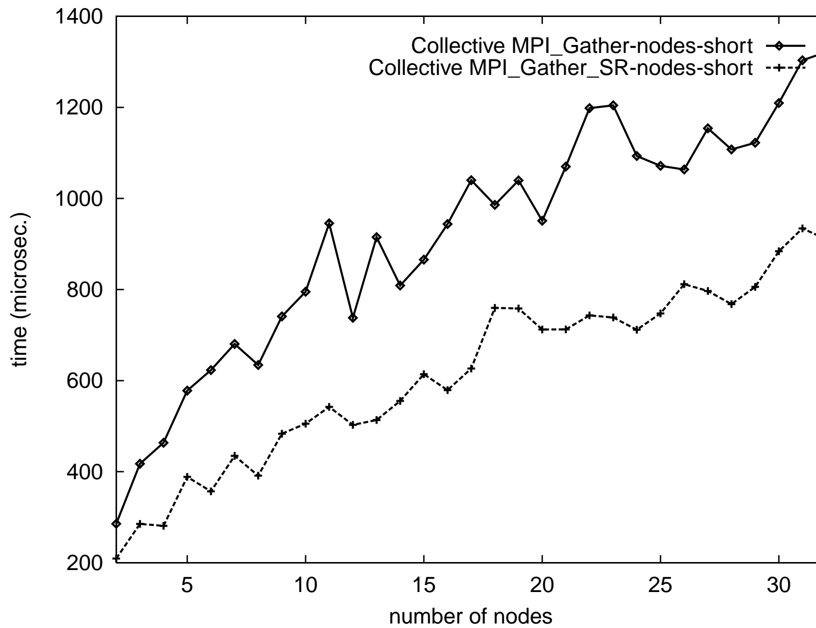


Fig. 7. Two implementations of a gather operation on the IBM RS 6000 SP. Variation over number of processors, message length 256 bytes. Measurements performed on the machine installed in Karlsruhe, February 2000.

gain performance on a particular machine by looking more closely in this direction.

Our last example investigates the implementation of the `MPI_Gather` collective on the IBM RS 6000 SP [28,24]. SKaMPI has a corresponding collective pattern instance, as well as two pattern instances which measure "hand-written" implementations of the gather functionality using point-to-point communication (see Table 2). Figure 7 compares the vendor implemented `MPI_Gather`-operation to an implementation with

blocking send and receive operations for fixed, short messages of 256 bytes, varied over number of processors. For short messages the hand-written, naive implementation performs significantly better than the MPI_Gather implementation of the vendor MPI library. Findings like this may discourage users from relying on the MPI collectives, but should rather incite vendors to improve their library.

7. Future extensions

Although many aspects of MPI are covered by SKaMPI, there is still room for improvements in various directions. Of immediate concern are more collective pattern instances, e.g. for ring-communication (as in the effective bandwidth benchmark [4]), for "bisection bandwidth" where half the processes simultaneously communicate with the other half, and for more alternative implementations of collective operations, either in terms of point-to-point communication or in terms of other MPI collectives.

The benchmarking of the "irregular" (or vector) variants of MPI collectives like MPI_Alltoallv is rather rudimentary, in particular there is no room for individually varying the amount of data communicated between each pair of processes. In future versions of SKaMPI it should be made possible to select among different distributions and vary more flexibly over message lengths. Also more accurate measurement mechanisms for collective operations should be considered, see [3].

Another natural extension is towards MPI-2 functionality [6]. Particularly relevant, but also easy to incorporate in the existing patterns is the one-sided communications, but perhaps also I/O should be covered by the benchmark; for some thoughts on MPI-IO benchmarking, see [12,17,20,21]. More flexible control over cache utilization may be required for realistic I/O benchmarking.

8. Summary

In the absence of an analytical performance model for MPI, accurate, reliable, and realistic benchmark data are necessary to guide the development and tuning of application programs. We described the SKaMPI benchmark which performs such detailed benchmarking of a large fraction of MPI, both in isolation and in more complex (master-worker) patterns. Not covered are the construction of datatypes and the construction and use of user-defined topologies. Complemented by application kernels benchmark, SKaMPI can give a realistic picture of the performance of a given MPI implementation on a given machine. The SKaMPI project is distinguished from other MPI benchmarks by a directed effort to collect results from a standard run into a public performance database. This public database can be a powerful aid to users who want to port their application to a machine to which they do not yet have access.

The most obvious problem with maintaining a public result database is that results get obsolete; this is especially so since benchmark results are supplied voluntarily by users. To keep up with the technical progress in MPI implementations and changes of machine details, support by vendors would be welcome.

Acknowledgments

We would like to thank Lutz Prechelt for contributions to the basic design and to the original SKaMPI-conference paper [25], Matthias Müller for the design of the web pages, Gunnar Hunzelmann for implementing the result database and enhancing the report-generator significantly, Thomas Worsch and Werner Augustin for their current redesign of collective measurements, and Roland Vollmar for his ongoing administrative and financial support; all of the University of Karlsruhe, Germany. Furthermore, we would like to thank the following people for their extensive comments, detailed suggestions and submitted measurements: Ed Benson (DEC, USA), Julien Bourgeois (Univ. de Besançon, France), Bronis R. de Supinski (Lawrence Livermore National Laboratory, USA), Dieter Kranzlmüller (Univ. of Linz, Austria), John May (Lawrence Livermore National Laboratory, USA), Hermann Mierendorff (FhG-GMD, Germany), Jean-Philippe Proux (CNRS/IDRIS, France), Rolf Rabenseifner (HLRS Stuttgart, Germany), Umesh Kumar V. Rajasekaran (Univ. of Cincinnaiti, USA), Christian Schaubschläger (Univ. of Linz, Austria), and Scott Taylor (Lawrence Livermore National Laboratory, USA).

References

[1] 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, 1994.

[2] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo and M. Yarrow, The NAS parallel benchmarks 2.0, Report NAS-95-020, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, 1995.

[3] B.R. de Supinski and N.T. Karonis, Accurately measuring MPI broadcasts in a computational grid, *Proc. 8th IEEE Symp. on High Performance Distributed Computing* (*HPDC-8*), 2000.

[4] Effective Bandwidth Benchmark, *http://www.hlrs.de/organization/par/services/models/mpi/b_eff/*.

[5] V. Getov, E. Hernandez and T. Hey, Message-passing performance of parallel computers, *Euro-Par'97 Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, 1997, pp. 1009–1016.

[6] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir and M. Snir, *MPI – The Complete Reference*, volume 2, The MPI Extensions, MIT Press, 1998.

[7] W. Gropp and E. Lusk, Reproducible measurements of MPI performance characteristics, *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 6th European PVM/MPI Users' Group Meeting*, volume 1697 of *Lecture Notes in Computer Science*, 1999, pp. 11–18.

[8] W. Gropp, E. Lusk, N. Doss and A. Skjellum, A high-performance, portable imlementation of the MPI message passing interface standard, *Parallel Computing* **22**(6) (1996), 789–828.

[9] W.D. Gropp, E. Lusk and D. Swider, Improving the performance of MPI derived datatypes, *Third MPI Developer's and User's Conference* (*MPIDC'99*), 1999, pp. 25–30.

[10] R. Hempel, Basic message passing benchmarks, methodology and pitfalls, SPEC Workshop on Benchmarking Parallel and High-Performance Computing Systems, Wuppertal, Germany, http://www.hlrs.de/mpi/b_eff/hempel_wuppertal.ppt, 1999.

[11] G. 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, 1999.

[12] D. Lancaster, C. Addison and T. Oliver, A parallel I/O test suite, *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 5th European PVM/MPI Users' Group Meeting*, volume 1497 of *Lecture Notes in Computer Science*, 1998, pp. 36–43.

[13] LLCBench Home Page, http://icl.cs.utk.edu/projects/llcbench/

[14] H. Mierendorff, K. Cassirer and H. Schwamborn, Working with MPI benchmarking suites on ccNUMA architectures, *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 7th European PVM/MPI Users' Group Meeting*, volume 1908 of *Lecture Notes in Computer Science*, 2000, pp. 18–26.

[15] The MPI Forum, http://www.mpi-forum.org/

[16] MPICH – A Portable Implementation of MPI, http://www.mcs.anl.gov/Projects/mpi/mpich/

[17] The Pallas MPI Benchmark, http://www.pallas.de/pages/pmbd.htm.

[18] PARKBENCH Committee and R. Hockney (chair), Public international benchmarks for parallel computers, *Scientific Programming* **3**(2) (1994), iii–126.

[19] J. Piernas, A. Flores and J.M. Garcia, Analyzing the performance of MPI in a cluster of workstations based on Fast Ethernet, *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 4th European PVM/MPI Users' Group Meeting*, volume 1332 of *Lecture Notes in Computer Science*, 1997, pp. 17–24.

[20] R. Rabenseifner and A.E. Koniges, Effective file-I/O bandwidth benchmark, *Euro-Par 2000 Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, 2000, pp. 1273–1283.

[21] R. Rabenseifner and A. E. Koniges, Effective communication and file-I/O bandwidth benchmarks, *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 8th European PVM/MPI Users' Group Meeting*, volume 2131 of *Lecture Notes in Computer Science*, 2001, pp. 24–35.

[22] M. Resch, H. Berger and T. Boenisch, A comparison of MPI performance on different MPPs, *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 4th European PVM/MPI Users' Group Meeting*, volume 1332 of *Lecture Notes in Computer Science*, 1997, pp. 25–32.

[23] R. Reussner, Portable Leistungsmessung des Message Passing Interfaces, Diplomarbeit, Department of Informatics, University of Karlsruhe, Am Fasanengarten 5, D-76128 Karlsruhe, Germany, 1997.

[24] R. Reussner and G. Hunzelmann, Achieving performance portability with SKaMPI for high-performance MPI programs, *Computational Science – ICCS 2001. Proc. of ICCS 2001, International Conference on Computational Science, Part II, Special Session on Tools and Environments for Parallel and Distributed Programming.*, volume 2074 of *Lecture Notes in Computer Science*, 2001, pp. 841–850.

[25] R. Reussner, P. Sanders, L. Prechelt and M. Müller, SKaMPI: A detailed, accurate MPI benchmark, *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 5th European PVM/MPI Users' Group Meeting*, volume 1497 of *Lecture Notes in Computer Science*, 1998, pp. 52–59.

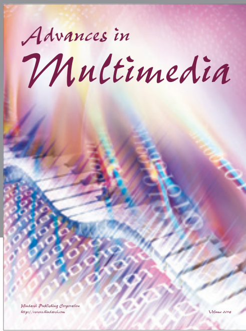
[26] R. Reussner, J.L. Träff and G. Hunzelmann, A benchmark for MPI derived datatypes, *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 7th European PVM/MPI Users' Group Meeting*, volume 1908 of *Lecture Notes in Computer Science*, 2000, pp.10–17.

[27] R.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.

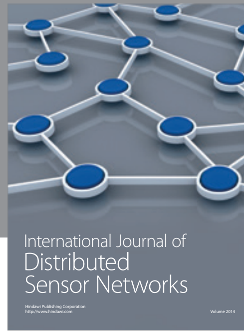
[28] R.H. Reussner, Recent Advances in SKaMPI, in: *High Performance Computing in Science and Engineering 2000*, E. Krause and W. Jäger, eds, Transactions of the High Performance Computing Center Stuttgart (HLRS), Springer-Verlag, 2001, pp. 520–530.

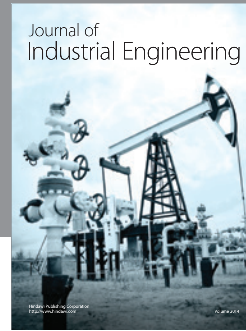
[29] M. Snir, S. Otto, S. Huss-Lederman, D. Walker and J. Dongarra, *MPI – The Complete Reference*, volume 1, The MPI Core, MIT Press, second edition, 1998.

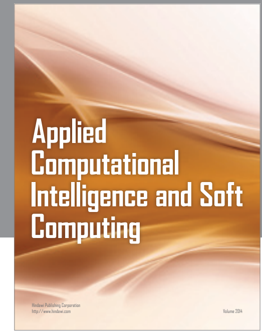
[30] J.L. Träff, R. Hempel, H. Ritzdorf and F. Zimmermann, Flattening on the fly: efficient handling of MPI derived datatypes, *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 6th European PVM/MPI Users' Group Meeting*, volume 1697 of *Lecture Notes in Computer Science*, pages 109–116, 1999.

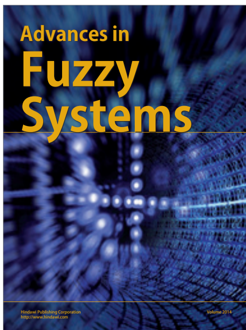

Advances in
Multimedia

The Scientific
World Journal

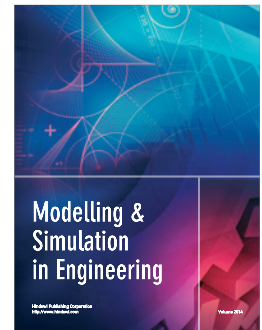

International Journal of
Distributed
Sensor Networks

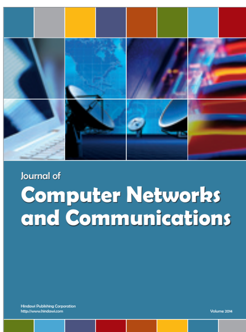

Journal of
Industrial Engineering

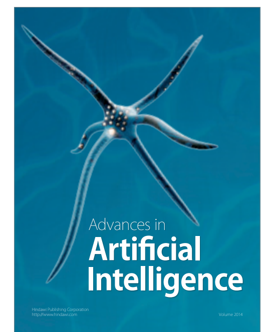

Applied
Computational
Intelligence and Soft
Computing

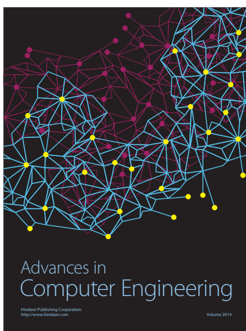

Advances in
Fuzzy
Systems

Hindawi
Submit your manuscripts at
http://www.hindawi.com

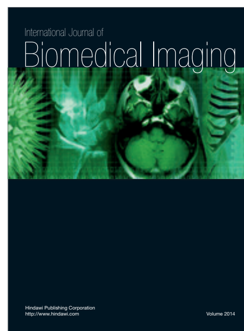

Modelling &
Simulation
in Engineering

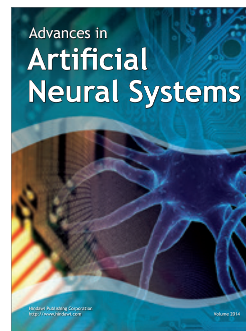

Journal of
Computer Networks
and Communications


Advances in
Artificial
Intelligence


Advances in
Computer Engineering

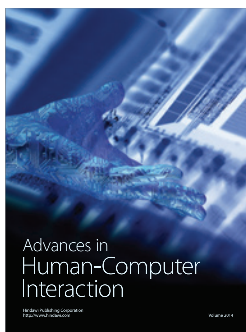
International Journal of
Computer Games
Technology

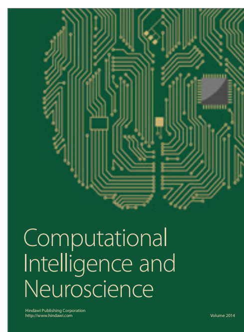

International Journal of
Biomedical Imaging


Advances in
Artificial
Neural Systems

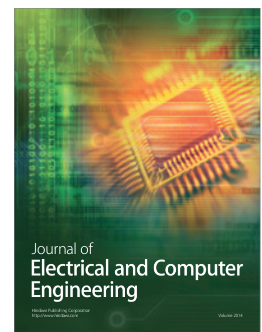
Advances in
Software Engineering

Journal of
Robotics


Advances in
Human-Computer
Interaction


Computational
Intelligence and
Neuroscience

International Journal of
Reconfigurable
Computing


Journal of
Electrical and Computer
Engineering