Published in Recent Advances in Parallel Virtual Machine and Message Passing Interface, Jack Dongarra and Dieter Kranzlmller (Eds.), Proceedings of the 9th European PVM/MPI Users' Group Meeting, EuroPVM/MPI 2002, Sep. 29 – Oct. 2, Linz, Austria, LNCS, Springer-Verlag, 2002.

Communication and Optimization Aspects on Hybrid Architectures

Rolf Rabenseifner

High-Performance Computing-Center (HLRS), University of Stuttgart
Allmandring 30, D-70550 Stuttgart, Germany
rabenseifner@hlrs.de,
www.hlrs.de/people/rabenseifner/

Abstract. Most HPC systems are clusters of shared memory nodes. Parallel programming must combine the distributed memory parallelization on the node inter-connect with the shared memory parallelization inside of each node. The hybrid MPI+OpenMP programming model is compared with pure MPI and compiler based parallelization. The paper focuses on bandwidth and latency aspects, but also whether programming paradigms can separate the optimization of communication and computation. Benchmark results are presented for hybrid and pure MPI communication.

Keywords. OpenMP, MPI, Hybrid Parallel Programming, Threads and MPI, HPC.

1 Motivation

The hybrid MPI+OpenMP programming model on clusters of SMP nodes is already used in many applications, but often there is only a small benefit as, e.g., reported with the climate model calculations of one of the Gordon Bell Prize finalists at SC 2001 [6], or sometimes losses are reported compared to the pure MPI model, e.g., as shown with an discrete element modeling algorithm in [4]. In the hybrid model, each SMP node is executing one multi-threaded MPI process. With pure MPI programming, each processor executes a single-threaded MPI process, i.e., the cluster of SMP nodes is treated as a large MPP (massively parallel processing) system.

One of the major drawbacks of the hybrid MPI-OpenMP programming model is based on a very simple usage of this hybrid approach: If the MPI routines are invoked only outside of parallel regions, all threads except the master thread are sleeping while the MPI routines are executed.

This paper will discuss this phenomenon and other hybrid MPI-OpenMP programming strategies. Sect. 2 shows different methods to combine MPI and OpenMP. Further rules on hybrid programming are discussed in Sect. 3, and pure MPI on hybrid architectures in Sect. 4. Sect. 5 presents benchmark results of the communication in both models. Sect. 6 to 8 compare the MPI based programming models with compiler based parallelization.

2 MPI and Thread-Based Parallelization

The combination of MPI and thread-based parallelization was already addressed by the MPI-2 Forum in Sect. 8.7 *MPI and Threads* in [8]. For hybrid programming, the MPI-1 routine MPI_Init() should be substituted by a call to MPI_Init_threads() which has the input argument named *required* to define which thread-support the application requests from the MPI library, and the output argument *provided* which is used by the MPI library to tell the application which thread-support is available. MPI libraries may support the following thread-categories (higher categories are supersets of all lower ones):

MPI_THREAD_SINGLE – No thread-support.

MPI_THREAD_FUNNELED – Only the master thread is allowed to call MPI routines. The other threads may run other application code while the master thread calls an MPI routine.

MPI_THREAD_SERIALIZED – Multiple threads may make MPI-calls, but only one thread may execute an MPI routine at a time.

MPI_THREAD_MULTIPLE – Multiple threads may call MPI without any restrictions.

The constants MPI_THREADS_... are monotonically increasing.

Between MPI_THREAD_SINGLE and FUNNELED, there are intermediate levels of thread support, not yet addressed by the standard:

T1a – The MPI process may be multi-threaded but only the master thread may call MPI routines **AND** only while the other threads do not exist, i.e., parallel threads created by a parallel region must be destroyed before an MPI routine is called. An MPI library supporting this class (and not more) must also return provided=MPI_THREAD_SINGLE (i.e., no thread-support) because of the lack of this definition in the MPI-2 standard[1].

T1b – The definition T1a is relaxed in the sense that more than one thread may exist during the call of MPI routines, but all threads except the master thread must sleep, i.e., must be blocked in some OpenMP synchronization. As in T1a, an MPI library supporting T1b but not more must also return provided=MPI_THREAD_SINGLE.

Usually, the application cannot distinguish whether an OpenMP based parallelization or an automatic parallelization needs T1a or T1b to allow calls to MPI routines outside of OpenMP parallel regions, because it is not defined, whether at the end of a parallel region the team of threads is sleeping or is destroyed. And usually, this category is chosen, when the MPI routines are called outside of parallel regions. Therefore, one should summarize the cases T1a and T1b to only one case:

T1 – The MPI process may be multi-threaded but only the master thread may call MPI routines **AND** only outside of parallel regions (in case of OpenMP) or outside of parallelized code (if automatic parallelization is used). We define here an additional constant **THREAD_MASTERONLY** with a value between MPI_THREAD_SINGLE and MPI_THREAD_FUNNELED.

[1] This may be solved in the revision 2.1 of the MPI standard.

3 Rules with hybrid programming

THREAD_MASTERONLY defines the most simple hybrid programming model with MPI and OpenMP, because MPI routines may be called only outside of parallel regions. The new cache coherence rules in OpenMP 2.0 guarantee that the outcome of an MPI routine is visible to all threads in a subsequent parallel region, and that the outcome of all threads of a parallel region is visible to a subsequent MPI routine.

The programming model behind MPI_THREAD_FUNNELED can be achieved by surrounding the call to the MPI routine with the OMP MASTER and OMP END MASTER directives inside of a parallel region. One must be very careful, because OMP MASTER does not imply an automatic barrier synchronization or an automatic cache flush neither at the entry to nor at the exit from the master section. If the application wants to send data computed in the previous parallel region or wants to receive data into a buffer that was also used in the previous parallel region (e.g., to use the data received in the previous iteration), then a barrier with implied cache flush is necessary prior to calling the MPI routine, i.e., prior to the master section. If the data or buffer is also used in the parallel region after the exit of the MPI routine and its master section, then also a barrier is necessary after the exit of the master section. If both barriers must be done, then while the master thread is executing the MPI routine, all other threads are sleeping, i.e., we are going back to the case T1b.

The rules of MPI_THREAD_SERIALIZED can be achieved by using the OMP SINGLE directive, which has an implied barrier only at the exit (unless NOWAIT is specified). Here again, the same problems as with FUNNELED must be taken into account.

These problems with FUNNELED and SERIALIZED arise, because the communication must be funneled from all threads to one thread (an arbitrary thread with OMP SINGLE, and the master thread with OMP MASTER). Only MPI_THREAD_MULTIPLE allows a direct message passing from each thread in one node to each thread in another node.

Based on these reasons and because THREAD_MASTERONLY is available on nearly all clusters, often, hybrid and portable parallelization is using only this parallelization scheme. This paper will evaluate this hybrid model by comparing it with the non-hybrid pure MPI model described in the next section.

4 Pure MPI on hybrid architectures

Using a pure MPI model, the cluster must be viewed as a hybrid communication network with typically fast communication paths inside of each SMP node and slower paths between the nodes. It is important to implement a good mapping of the communication paths used by application to the hybrid communication network of the cluster. The MPI standard defines virtual topologies for this purpose, but the optimization algorithm isn't yet implemented in most MPI implementations. Therefore, in most cases, it is important to choose a good ranking in MPI_COMM_WORLD. E.g., on a Hitachi SR8000, the MPI library allows two different ranking schemes, round robin (ranks 0, N, 2*N, ... on node 0; ranks 1, N+1, 2*N+1, ... on node 1, ...; with N=number of nodes) and sequential (rank 0–7 on node 0, ranks 8–15 on node 1, ...), and the user has to decide which scheme may fit better to the communication needs of his application.

The pure MPI programming model implies additional message transfers due to the higher number of MPI processes and higher number of boundaries. Let us consider, for example, a 3-dimensional cartesian domain decomposition. Each domain may have to transfer boundary information to its neighbors in all six cartesian directions (↑↓ ⇄ ↙↗). Bringing this model on a cluster with 8-way SMP nodes, on each node, we should execute the domains belonging to a 2×2×2 cube. Domain-to-domain communication occurs as node-to-node (int**er**-node) communication and as int**ra**-node communication between the domains inside of each cube. Hereby, each domain has 3 neighbors inside the cube and 3 neighbors outside, i.e., in the inter-node and the intra-node communication the amount of transferred bytes should be equivalent. If we compare this pure MPI model with a hybrid model, assuming that the domains (in the pure MPI model) in each 2×2×2 cube are combined to a super-domain in the hybrid model, then the amount of data transferred on the node-interconnect should be the same in both models. This implies that in the pure MPI model, the total amount of transferred bytes (inter-node plus intra-node) will be twice the number of bytes in the hybrid model. The same ratio is shown in the topology in the left diagram of Fig. 1. In the symmetric case, the intra-node and inter-node communication has the same transfer volume.

5 Benchmark Results

The following benchmark results will compare the communication behavior of the hybrid MPI+OpenMP model with the pure MPI model that can be named also as MPP-MPI model. Based on the domain decomposition scenario discussed in the last section, we compare the bandwidth of both models and the ratio of the total communication time presuming that in the pure MPI model, the total amount of transferred data is twice the amount in the hybrid model. The benchmark was done on a Hitachi SR8000 with 16 nodes from which 12 nodes are available for MPI parallel applications. Each node has 8 CPUs. The effective communication benchmark b_eff is used [5, 12]. It accumulates the communication bandwidth values of the communication done by each MPI process. To determine the bandwidth of each process, the maximum time needed by all processes is used, i.e., this benchmark models an application behavior, where the node with the slowest communication controls the real execution time. To compare both models, we use the following benchmark patterns:

- b_eff – the accumulated bandwidth average for several ring and random patterns (this is the major benchmark pattern of the b_eff benchmark);
- 3D-cyclic – a 3-dimensional cyclic communication pattern with 6 neighbors for each MPI process (this is an additional pattern measured by the b_eff benchmark);

		b_eff (avg.)	b_eff at Lmax	3D-cyclic (average)	3D-cyclic at Lmax
b_{hybrid}	[MB/s]	1535	5565	1604	5638
(per node)	[MB/s]	(128)	(464)	(134)	(470)
b_{MPP}	[MB/s]	5299	16624	5000	18458
(per process)	[MB/s]	(55)	(173)	(52)	(192)
b_{MPP}/b_{hybrid}	(measured)	3.45	2.99	3.12	3.27
s_{MPP}/s_{hybrid}	(assumed)	2	2	2	2
T_{hybrid}/T_{MPP}	(concluding)	1.73	1.49	1.56	1.64

Table 1. Comparing the hybrid and the MPP communication needs.

With the following sub-options, we get 4 metrics (columns) in Table 1:

- average – the average bandwidth of 21 different message sizes (8 byte – 8 MB);
- at Lmax – the bandwidth is measured with 8 MB messages.

For each metrics, the following rows are presented in Tab. 1:

- b_{hybrid}, the accumulated bandwidth b for the hybrid model measured with a 1-threaded MPI process on each node (12 MPI processes),
- and in parentheses the same bandwidth per node,
- b_{MPP}, the accumulated bandwidth for the pure MPI model (96 MPI processes with sequential ranking in MPI_COMM_WORLD),
- and in parentheses the same bandwidth per process,
- b_{MPP}/b_{hybrid}, the ratio of accumulated MPP bandwidth and accumulated hybrid bandwidth,
- T_{hybrid}/T_{MPP}, the ratio of execution times T, assuming that total size s of the transferred data in the pure MPI model is twice of the size in the hybrid model, i.e., $s_{MPP}/s_{hybrid} = 2$, as shown in Sect.4. For this calculation, it is assumed that the measured bandwidth values are approximately valid also for doubled message sizes.

Note that this comparison was done with no special optimized topology mapping in the pure MPI model. The result shows that the pure MPI communication model is faster than the communication in the hybrid model. There are at least two reasons: (1) In the hybrid model, all communication was done by the master thread while the other threads were inactive; (2) One thread is not able to saturate the total inter-node bandwidth that is available for each node.

Fig. 1 shows a similar experiment. In the hybrid MPI+OpenMP communication scheme, only the left thread sends inter-node messages. Therefore, the message size is 8 times the size used in the pure MPI scheme. Here, each CPU communicates in the vertical (inter-node) and horizontal (intra-node) direction. The total communication time with the hybrid model (19.2ms) is 66% greater than with the pure MPI communication (11.6ms), although with pure MPI, the total amount of transferred data is doubled due to the additional intra-node communication. The left diagram shows the measured transfer time for several message sizes and the ratio of the transfer time in the hybrid model to the transfer time of inter-node plus intra-node communication. Note that for the hybrid

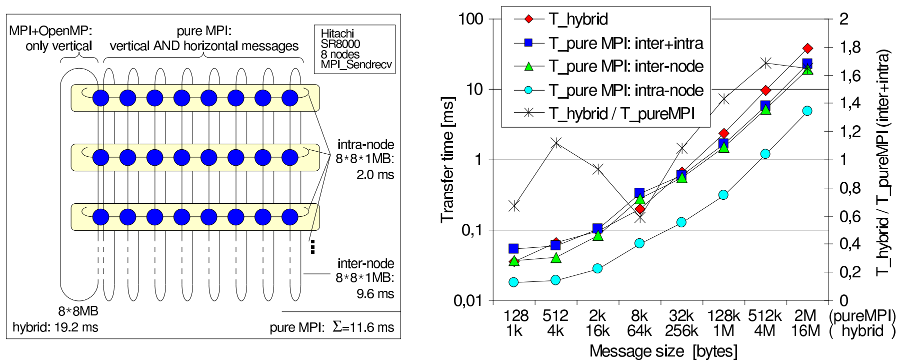

Fig. 1. Parallel communication in a cartesian topology.

measurements, the message size must reflect that the inter-node data exchange of all threads is communicated by the master thread, and therefore, the message size is chosen 8 times larger, i.e., it ranges from 1 kB to 16 GB. The diagram shows that for message sizes greater than 32 kB, the pure MPI model is faster than the hybrid model in this experiment. With smaller message sizes, the ratio $T_{hybrid}/T_{pureMPI}$ depends mainly on the latencies of the underlying protocols that may differ due to the larger message sizes in the hybrid model.

A similar communication behavior can be expected on other platforms if the inter-mode network cannot be saturated by a single processor in each SMP node. This paper cannot analyze the reasons based on decisions in the hardware or software (MPI library) design of a system. For example, additional local copying for user space to system space and bad pipelining or parallelization of process-local activities and inter-node data transfer may cause that one CPU cannot reach the inter-node peak bandwidth. The shown ratio of hybrid to pure MPI transfer time may be a major reason when an application is running faster in the pure MPI model than in the hybrid model.

6 Comparison of hybrid MPI+OpenMP versus pure MPI

The comparison in this paper focuses on bandwidth and latency aspects, i.e., how to achieve a major percentage of the physical inter-node network bandwidth with various parallel programming models.

Although the benchmark results in the last section show advantages of the pure MPI model, there are also advantages of the hybrid model. In the hybrid model there is no communication overhead inside of a node. The message size of the boundary information of one process may be larger (although the total amount of communication data is reduced). This reduces latency based overheads. The number of MPI processes is reduced. This may cause a better speedup based on Amdahl's law and may cause a faster convergence if, e.g., the parallel implementation of a multigrid numeric is only computed on a partial grid. To reduce the MPI overhead by communicating only through one thread, the MPI communication routines should be relieved by unnecessary local work, e.g., concatenation of data should be better done by copying the data to a scratch buffer with a thread-parallelized loop, instead of using derived MPI datatypes. MPI reduction operations can be split into the inter-node communication part and the local reduction part by using user-defined operations, but a local thread-based parallelization of these operations may cause problems because these threads are running while an MPI routine may communicate.

Hybrid programming is often done in two different ways: (a) the domain decomposition is used for the inter-node parallelization with MPI and also for the intra-node parallelization with OpenMP, i.e., in both cases, a coarse grained parallelization is used. (b) The intra-node parallelization is implemented as a fine grained parallelization, e.g., mainly as loop parallelization. The second case also allows automatic intra-node parallelization by the compiler, but Amdahl's law must be considered independently for both parallelizations.

Now we want to compare three different hybrid programming schemes: In the ***masteronly*** scheme, only the master thread communicates and only outside of parallel regions. The computation is parallelized on all CPUs of an SMP node and inside of parallel regions. In the ***funneled*** scheme, the communication on the master thread is done in parallel with the computation on the other threads. For this, the application has to be restructured to allow the overlap of communication and computation. In the ***multiple*** scheme, all threads may communicate and compute in parallel. If the other application threads do not sleep while the master thread is communicating with MPI then communication time T_{hybrid} in Tab. 1 counts only the eighth (a node has 8 CPUs on the SR8000) because only one instead of 1 (active) plus 7 (idling) CPUs is communicating. In this hybrid programming style, the factor T_{hybrid}/T_{MPP} must be reduced to the eighth, i.e. from about 1.6 to about 0.2. This can be implemented by dedicating one thread for communication and the other threads of a node for computing, but also with full load balancing with different mixes of computation and communication on all threads.

Wellein et al. compared in [14] the two hybrid programming schemes *masteronly* (named vector-mode in [14]) and *funneled* (task-mode). They show that the performance ratio $\epsilon = (\frac{T_{funneled\ or\ multiple}}{T_{masteronly}})^{-1}$ of *funneled* (or *multiple*) to *masteronly* execution has the bounds $1-\frac{1}{n} \le \epsilon \le 2-\frac{1}{n}$ if n is the number of threads of each SMP node. In general, if m threads are reserved for communication, T_{COMM} and T_{COMP} being the accumulated communication and computation time, and $f := \frac{T_{COMM}}{T_{COMM}+T_{COMP}}$ being the real communication percentage, then ϵ is bounded[2] by $\boldsymbol{1-\frac{m}{n} \le \epsilon \le 1+m(1-\frac{1}{n})}$ and $\boldsymbol{\epsilon \le 1+fn(1-\frac{1}{n})}$. If $m \ge 1$ or $\frac{m}{n} \ge f$, then the *funneled* scheme is faster if $f \ge \frac{2}{n(n/m-1)+1}$. The maximum of ϵ is given for $f=\frac{m}{n}$. E.g., if n=8 and m=1, the first upper bound of ϵ indicates that the *funneled* scheme may be up to 1.875 times faster than *masteronly*, but if the communication ratio f does not fit to m/n, only small profits may be shown, as indicated with the second upper bound $1+fn(1-\frac{1}{n})$ and benchmarks in [14].

[2] If m is non-integer, $m<1$, and $\frac{m}{n}<f$, then the lower bound is $m-\frac{m}{n} \le \epsilon$.

Access method	copies	remarks	bandwidth $b(\mathit{message\ size})$
2-sided MPI	2	internal MPI buffer + application receive buffer	$b_\infty/(1+\frac{b_\infty T_{lat}}{size})$, e.g., $300\,\mathrm{MB/s}\ /\ (1+\frac{300\,\mathrm{MB/s}\times 10\,\mu\mathrm{s}}{10\,\mathrm{kB}})$ $= 232\,\mathrm{MB/s}$
1-sided MPI	1	application receive buffer	same formula, but probably better b_∞ and T_{lat}
Co-Array Fortran, UPC, HPC, OpenMP with cluster extensions	1	page based transfer	extremely poor, if only parts of the page are needed
	0	**word based access**	8 byte / T_{lat}, e.g., 8 byte / $0.33\,\mu\mathrm{s}$ = **24 MB/s**
	0	latency hiding with pre-fetch	b_∞
	1	latency hiding with buffering	see 1-sided communication

Table 2. Memory copies from remote memory to local CPU register.

7 MPI versus Compiler-based Parallelization

Now, we compare the MPI based models with the NUMA or RDMA based models. To access data on another node with MPI, the data must be copied to a local memory location (so called halo or shadow) by message passing, before it can be loaded into the CPU. Usually all necessary data should be transferred in one large message instead of using several short messages. Then, the transfer speed is dominated by the asymptotic bandwidth of the network, e.g., as reported for 3D-cyclic-Lmax in Tab. 1 per node (470 MB/s) or per process (192 MB/s). With NUMA or RDMA, the data can be loaded directly from the remote memory location into the CPU. This may imply short accesses, i.e., the access is latency bound. Although the NUMA or RDMA latency is usually 10 times shorter than the message passing latency, the total transfer speed may be worse. E.g., [2] reports on a ccNUMA system a latency of 0.33–1 μs, which implies a bandwidth of only 8–24 MB/s for a 8 byte data. This effect can be eliminated if the compiler has implemented a remote pre-fetching strategy as described in [9], but this method is still not used in all compilers.

The remote memory access can also be optimized by buffering or pipelining the data that must be transferred. This approach may be hard to automate, and current research in OpenMP compiler technology already studies the bandwidth optimization on SMP clusters [13], but it can be easily implemented as an directive-based optimization technique: The application thread can define the (remote) data it will use in the next simulation step and the compiled OpenMP code can pre-fetch the whole remote part of the data with a bandwidth-optimized transfer method. Table 2 summarizes this comparison.

8 Parallelization and Compilation

Major advantages of OpenMP based programming are that the application can be *incrementally parallelized* and that one still has a single source for serial and parallel compilation. On a cluster of SMPs, the major disadvantages are that

OpenMP has a flat memory model and that it does not know buffered transfers to reach the asymptotic network bandwidth. But, these problems can be solved by tiny additional directives, like the proposed migration and memory-pinning directives in [3], and additional directives that allow a contiguous transfer of the whole boundary information between each simulation step. Those directives are optimization features that do not modify the basic OpenMP model, as this would be done with directives to define a full HPF-like user-directed data distribution (as in [3, 7]). Another lack in the current OpenMP standard is the absence of a strategy of combining automatic parallelization with OpenMP parallelization, although this is implemented in a non-standardized way in nearly all OpenMP compilers. This problem can be solved, e.g., by adding directives to define scopes where the compiler is allowed to automatically parallelize the code, e.g., with *auto-parallel* regions. An OpenMP-based parallel programming model for SMP-clusters should be usable for both, fine grained loop parallelization, and coarse grained domain decomposition. There should be a clear path from MPI to such an OpenMP cluster programming model with a performance that should not be worse than with pure MPI or hybrid MPI+OpenMP.

It is also important to have a good compilation strategy that allows the development of well optimizing compilers on any combination of processor, memory access, and network hardware. The MPI based approaches, especially the hybrid MPI+OpenMP approach, clearly separate remote from local memory access optimization. The remote access is optimized by the MPI library, and the local memory access must be improved by the compiler. Such separation is realized, e.g., in the NANOS project OpenMP compiler [1, 10]. The separation of local and remote access optimization may be more essential than the chance of achieving a zero-latency by remote pre-fetching (Tab. 2) with direct compiler generated instructions for remote data access. Pre-fetching can also be done via macros or library calls in the input for the local (OpenMP) compiler.

9 Conclusion

For many parallel applications on hybrid systems, it is important to achieve a high communication bandwidth between the processes on the node-to-node inter-connect. On such architectures, the standard programming models of SMP or MPP systems do not longer fit well. The rules for hybrid MPI+OpenMP programming and the benchmark results in this paper show that a hybrid approach is not automatically the best solution if the communication is funneled by the master thread and long message sizes can be used. The MPI based parallel programming models are still the major paradigm on HPC platforms. OpenMP with further optimization features for clusters of SMPs and bandwidth based data transfer on the node interconnect have a chance to achieve a similar performance together with an incremental parallelization approach, but only if the current SMP model is enhanced by features that allow an optimization of the total inter-node traffic. Same important is a strategy that allows independently the optimization of the computation (e.g., choosing the best available compiler for the processor and programming language) and the communication.

Acknowledgments

The author would like to acknowledge his colleagues and all the people that supported these projects with suggestions and helpful discussions. He would especially like to thank Alice Koniges, David Eder and Matthias Brehm for productive discussions of the limits of hybrid programming, Bob Ciotti and Gabrielle Jost for the discussions on MLP, Gerrit Schulz for his work on the benchmarks, Gerhard Wellein for discussions on network congestion in the pure MPI model, and Thomas Bönisch, Matthias Müller, Uwe Küster, and John M. Levesque for discussions on OpenMP cluster extensions and vectorization.

References

1. E. Ayguade, M. Gonzalez, J. Labarta, X. Martorell, N. Navarro, and J. Oliver, *NanosCompiler: A Research Platform for OpenMP Extensions*, in proceedings of the 1st European Workshop on OpenMP (EWOMP'99), Lund, Sweden, Sep. 1999.
2. Robert B. Ciotti, James R. Taft, and Jens Petersohn, *Early Experiences with the 512 Processor Single System Image Origin2000*, proceedings of the 42nd International Cray User Group Conference, SUMMIT 2000, Noordwijk, The Netherlands, May 22–26, 2000, `www.cug.org`.
3. Jonathan Harris, *Extending OpenMP for NUMA Architectures*, in proceedings of the Second European Workshop on OpenMP, EWOMP 2000.
4. D. S. Henty, *Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling*, in Proc. Supercomputing'00, Dallas, TX, 2000. `http://citeseer.nj.nec.com/henty00performance.html`
5. Alice E. Koniges, Rolf Rabenseifner, Karl Solchenbach, *Benchmark Design for Characterization of Balanced High-Performance Architectures*, in proceedings, 15th International Parallel and Distributed Processing Symposium (IPDPS'01), Workshop on Massively Parallel Processing, April 23-27, 2001, San Francisco, USA.
6. Richard D. Loft, Stephen J. Thomas, and John M. Dennis, *Terascale spectral element dynamical core for atmospheric general circulation models*, in proceedings, SC 2001, Nov. 2001, Denver, USA.
7. John Merlin, *Distributed OpenMP: Extensions to OpenMP for SMP Clusters*, in proceedings of the Second European Workshop on OpenMP, EWOMP 2000.
8. Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997, `www.mpi-forum.org`.
9. Matthias M. Müller, *Compiler-Generated Vector-based Prefetching on Architectures with Distributed Memory*, in High Performance Computing in Science and Engineering '01, W. Jger and E. Krause (eds), Springer, 2001.
10. The NANOS Project, Jesus Labarta, et al., `//research.ac.upc.es/hpc/nanos/`.
11. OpenMP Group, `www.openmp.org`.
12. Rolf Rabenseifner and Alice E. Koniges, *Effective Communication and File-I/O Bandwidth Benchmarks*, in Recent Advances in Parallel Virtual Machine and Message Passing Interface, proceedings of the 8th European PVM/MPI Users' Group Meeting, Santorini, Greece, LNCS 2131, Y. Cotronis, J. Dongarra (Eds.), Springer, 2001, pp 24-35, `www.hlrs.de/mpi/b_eff/`, `www.hlrs.de/mpi/b_eff_io/`.
13. Mitsuhisa Sato, Shigehisa Satoh, Kazuhiro Kusano and Yoshio Tanaka, *Design of OpenMP Compiler for an SMP Cluster*, in proceedings of the 1st European Workshop on OpenMP (EWOMP'99), Lund, Sweden, Sep. 1999, pp 32–39. `http://citeseer.nj.nec.com/sato99design.html`
14. G. Wellein, G. Hager, A. Basermann, and H. Fehske, *Fast sparse matrix-vector multiplication for TeraFlop/s computers*, in proceedings of Vector and Parallel Processing - VECPAR'2002, Porto, Portugal, June 26–28, 2002, Springer LNCS.