Published in the proceedings of WOMPEI 2002, International Workshop on OpenMP: Experiences and Implementations, part of ISHPC-IV, International Symposium on High Performance Computing, May, 15–17., 2002, Kansai Science City, Japan. LNCS, Springer-Verlag, 2002.

Communication Bandwidth of Parallel Programming Models 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. This paper introduces several programming models for hybrid systems. It focuses on programming methods that can achieve optimal inter-node communication bandwidth and on the hybrid MPI+OpenMP approach and its programming rules. The communication behavior is compared with the pure MPI programming paradigm and with RDMA and NUMA based programming models.

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

1 Motivation

Today, most systems in high performance computing (HPC) are clusters of SMP (symmetric multi-processor) nodes, i.e., they are hybrid architectures, shared memory systems are inside of each node, and a distributed memory parallel (DMP) system is across the node boundaries. To achieve a minimal parallelization overhead, often a hybrid programming model is proposed, e.g., OpenMP [21] or automatic compiler based thread parallelization inside of each SMP node, and message passing (e.g., with MPI [16]) on the node interconnect. Another often used programming model is the flat and pure massively parallel processing (MPP) MPI model, where separate single-threaded MPI processes are running on each CPU. Using the hybrid programming model instead of the MPP-MPI model has the advantage that there is no message passing overhead inside of each SMP node, because the threads can access the data provided by other threads directly by accessing the shared memory instead of passing the data through a message.

The hybrid MPI+OpenMP programming model 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 [14], or sometimes losses are reported compared to the MPP-MPI model, e.g., as shown with an discrete element modelling algorithm in [12].

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. In Sect. 2, an overview on hybrid programming models is given. Sect. 3 shows different methods to combine MPI and OpenMP. Further rules on hybrid programming are discussed in Sect. 4. Pure MPI can also be used on hybrid architectures, as shown Sect. 5. Sect. 6 presents the results of a parallel communication benchmark. Sect. 7 compares the MPI based programming models with major shared and virtual shared memory models.

2 Programming Models on Hybrid Architectures

The available programming models depend on the type of cluster hardware. If the node interconnect allows cache-coherent or non-cache-coherent non-uniform memory access (ccNUMA and nccNUMA), i.e., if the memory access inside of each SMP node and across the cluster interconnect is implemented by the same instructions, then one can use programming models which need a shared memory access across the whole cluster. This includes OpenMP on the whole cluster, usage of nested parallelism inside of OpenMP, but also OpenMP with cluster extensions, that are primarily based on a first touch mechanism [11] or on data distribution extensions [15]. These cluster extensions may also benefit from the availability of software based shared virtual memory (SVM) [5, 25, 26]. At NASA/Ames, a hybrid approach was developed. The parallelization is organized in two levels: The upper level is process based, and in the lower level each process is multi-threaded with OpenMP. The processes are using a Fortran wrapper around the System V shared memory module *shm*, that allows to fork the processes, to initialize a shared memory segment, to associate portions of this segment with Cray pointer based array in each process, and to make a barrier synchronization over all processes. This system is named as Multi Level Parallelism (MLP) and it allows very flexible, dynamic and simple way of load balancing: At each start of a parallel region inside of each MLP process, the number of threads, i.e., the number of used CPUs, may be adapted [8]. Although MLP is a proprietary method of NASA/Ames, the programming style based on *shm* is non-proprietary.

If the node interconnect requires different methods for accessing local and cluster-wide memory, but if there are remote direct memory access (RDMA) methods available, i.e., if one node can access the memory of another node without interaction of a CPU on that node, then further programming methods are available: Such systems can be programmed with Co-Array Fortran [20] or Unified Parallel C (UPC) [7, 9]. In Co-Array Fortran, the access to an array of another process or thread is done by using an additional trailing array subscript in square brackets addressing that process or thread. Both language extensions can also be used to program clusters of SMP nodes, because they neither add a message passing overhead nor the overhead of additional copies. A key issue for a more widespread usage of UPC and Co-Array Fortran is the availability of (portable) commpiling systems for a wide range of platforms with a clear development path to achieve an optimal performance, as it was presented for MPI by the early MPICH implementation [10]. Another approach to use the RDMA hardware is based on one-sided communication, e.g., in Cray's *shmem* library or in MPI-2 [17]. These library-based methods allow to store (fetch) data to (from) the memory of another process in a SPMD environment. The *shmem* library was ported by many vendors to their systems. All programming models available for RDMA-class node-interconnect are also usable on NUMA-class interconnects.

The third class of hardware supports neither NUMA access nor RDMA. Only pure message passing is available on the node-interconnect. Programming models designed for this class of hardware have the major advantage that they are applicable to all other already mentioned classes. This paper focuses on this type of hardware. The commonly accepted standard for message passing between the nodes is the Message Passing Interface (MPI) [16, 17]. The major programming styles are pure MPI, i.e., the MPP model that uses each CPU for one MPI process, and hybrid models, e.g., MPI on the node-interconnect and OpenMP or automatic or semi-automatic compiler based thread-parallelization inside of each SMP node. Inside of each node mainly two different SMP parallelization strategies are used: (a) A coarse-grain SPMD-style parallelization similar to the work distribution betwen the processes in a message passing program is applied; this method allows a similar computational efficiency as with the pure MPI parallelization; the efficiency of the communication is a major factor in the omparance of this hybrid approach with the pure MPI solution. The present paper is focused on the communication aspects. (b) A fine-grained SMP parallelization is done in an incremental effort of parallelizing loops inside of the MPI processes. The efficiency of such hybrid solution depends on both, the efficiency of the computation (Amdahl's law must be considered on both levels of parallelization) and of the communication, as shown in [6] for the NAS parallel benchmarks. Different SMP parallelization strategies in the hybrid model are also studied in [27]. High-Performance Fortran (HPF) is also available on clusters of SMPs. In [3], HPF based on hybrid MPI+OpenMP is compared with pure MPI.

3 MPI and Thread-Based Parallelization

This model was already addressed by the MPI-2 Forum in Sect. 8.7 *MPI and Threads* in [17]. 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):

T0 – No thread-support, represented *provided=MPI_THREAD_SINGLE*.

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. This class is not mentioned in the MPI standard and an MPI library supporting this class (and not more) must also return provided=MPI_THREAD_SINGLE 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.

T2 – Only the master thread will make calls to MPI routines. The other threads may run other application code while the master thread calls an MPI routine. This is allowed if the MPI library returns a value greater or equal to MPI_THREAD_FUNNELED in *provided*.

T3 – Multiple threads may make MPI-calls, but only one thread may execute an MPI routine at a time. This requires *provided* ≥ MPI_THREAD_SERIALIZED.

T4 – Multiple threads may call MPI without any restrictions. This hybrid programming style is available when *provided* = MPI_THREAD_MULTIPLE was returned.

The constants are monotonic, i.e.,
MPI_THREAD_SINGLE ≤ MPI_THREAD_FUNNELED ≤ ... .

Usually, the application cannot distinguish whether an OpenMP parallelization needs T1 or T2 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.

4 Rules with hybrid programming

T1 is 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[2], and that the outcome of all threads of a parallel region is visible to a subsequent MPI routine.

[1] This may be solved in the revision 2.1 of the MPI standard.
[2] There is still a lack in the draft from Nov. 2001 for the C language binding

T2 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 either at the entry to or 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.

T3 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 T2 must be taken into account.

These problems with T2 and T3 arise, because the communication needs must be funneled from all threads to one thread (an arbitrary thread in T3, and the master thread in T2). Only T4 allows a direct message passing from each thread in one node to each thread in another node.

Based on these reasons and because T1 is available on nearly all clusters, most hybrid and portable parallelization is using only the programming scheme described in T1. This paper will evaluate this hybrid model by comparing it with the non-hybrid model described in the next section.

5 MPP-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 MPP-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 ($\uparrow\downarrow \rightleftarrows \swarrow\!\!\nearrow$). Bringing this model on a cluster with 8-way SMP nodes, on each node, we should execute the domains belonging to a $2\times2\times2$ cube.

Domain-to-domain communcation 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 MPP-MPI model with a hybrid model, assuming that the domains (in the MPP-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 MPP-MPI model, the total amount of transferred bytes (inter-node plus intra-node) will be twice the number of bytes in the hybrid model. This result is independent from the way, the inner-node parallelization is implemented in the hybrid model, i.e., whether it is done in a coarse grained domain decomposition style or as fine grained loop parallelism.

6 Benchmark Results

The following benchmark results will compare the communication behavior of the hybrid MPI+OpenMP model with the pure 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 MPP-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 [13, 22, 23]. 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 metrics:

- b_eff – the accumulated bandwidth average for several ring and random patterns;
- 3-d-cyclic-Lmax – a 3-dimensional cyclic communication pattern with 6 neighbors for each MPI process; the bandwidth is measured with 8 MB messages.
- 3-d-cyclic-avg – same, but an average of 21 different message sizes.

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 MPP-MPI model (96 MPI processes with sequential ranking in MPI_COMM_WORLD),
- and in parentheses the same bandwidth per process,

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

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

- 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 MPP model is twice of the size in the hybrid model, i.e., $s_{MPP}/s_{hybrid} = 2$, as shown in Sect.5.

Note, that this comparison was done with no special optimized topology mapping in the MPP model. The result shows, that the MPP 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.

This communication behavior may be a major reason when an application is running faster in the MPP model than in the hybrid model.

7 Comparison

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

7.1 Hybrid MPI+OpenMP versus pure MPI

Although the benchmark results in the last section show a clear advantage of the MPP 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 in the inter-node communication. 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 numerics 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.

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. But in this case, the application must implement a load balancing algorithm to guarantee that the load on the communicating master thread is equal to the load on the other threads. This means that minimal transfer time can be achieved with the hybrid model, but at the costs of implementing an optimal load balancing between the thread(s) that computes and communicates and those threads that only compute.

7.2 MPI versus Remote Memory Access

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 3-d-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., [8] 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 [18], 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 OpenMP compiler research already studies the bandwidth optimization on SMP clusters [24], but it can be easily implemented as an directive-based

Access method	copies	remarks	bandwidth $b(message\ size)$
2-sided MPI	2	internal MPI buffer + application receive buffer + application receive buffer	$b_\infty/(1+\frac{b_\infty T_{lat}}{size})$, e.g., 300 MB/s / $(1+\frac{300\,\mathrm{MB/s}\times 10\,\mu\mathrm{s}}{10\,\mathrm{kB}})$ = 232 MB/s
1-sided MPI	1	application receive buffer	same formula, but probably better b_∞ and T_{lat}
UPC, Co-Array Fortran, HPF, OpenMP with cluster extensions	1	page based transfer	extremely poor
	0	**word based access**	8 byte / T_{lat}, e.g., 8 byte / 0.33 μs = **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.

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.

7.3 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, as already mentioned, these problems can be solved by tiny additional directives, like the proposed migration and memory-pinning directives in [11], 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 [11, 15]). 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., similar to the parallel region, one can define an *autoparallel* region. Usual rules for nested parallelism can apply, i.e., a compiler can define that it cannot handle nested parallelism.

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 [2, 19]. 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.

8 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 funnelled 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 trafic, e.g., with an user-directed optimization of page migration. and by features for latency-hiding, e.g., by allowing a user-directed transfer of the total boundary all at once.

9 Future Work

In the future, we also want to examine hybrid programming models based on aspects of latency and latency-hiding, especially in combination with vectorizing codes. Optimization of hybrid MPI+OpenMP programming will be done with thread-parallel MPI techniques and compared with topology-optimized non-hybrid MPI parallelization. To evaluate cluster programming models, we want to compare proposed distributed OpenMP extensions [11, 15] with fully thread-parallel MPI and hybrid MPI+OpenMP solutions varying the chunk sizes and cluster parameters to examine the influence of latency and bandwidth of local and remote memory accesses. The work should be a basis to study and to optimize MPI-based parallelization libraries on hybrid systems.

If OpenMP with the described cluster extensions should be used as a basic programming concept, it is important, that an automatic analysis of the remote and local memory access (e.g., integrated in existing analysis tools, e.g., in VAMPIR [4]), and improved tools for detecting race-conditions are available [1].

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 MPP model, and Thomas Bönisch, Matthias Müller, Uwe Küster, and John M. Levesque for discussions on OpenMP cluster extensions and vectorization.

References

1. Assure and AssureView, `http://www.kai.com/parallel/kappro/assure/`.
2. Eduard Ayguade, Marc Gonzalez, Jesus Labarta, Xavier Martorell, Nacho Navarro, and Jose Oliver, *NanosCompiler: A Research Platform for OpenMP Extensions*, in proceedings of the 1st European Workshop on OpenMP (EWOMP'99), Lund, Sweden, Sep. 1999.
3. Siegfried Benkner, Thomas Brandes, *High-Level Data Mapping for Clusters of SMPs*, in proceedings of the 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments, HIPS 2001, San Francisco, USA, April 2001, Springer LNCS 2026, pp 1–15.
4. Holger Brunst, Wolfgang E. Nagel, and Hans-Christian Hoppe, *Group Based Performance Analysis for Multithreaded SMP Cluster Applications*, in proceedings of Euro-Par2001, R. Sakellariou, J. Keane, J. Gurd, L. Freeman (Eds.), Manchester, UK, August 28.–31., 2001, LNCS 2150, Springer, 2001, pp 148–153.
5. R. Berrendorf, M. Gerndt, W. E. Nagel and J. Prumerr, *SVM Fortran*, Technical Report IB-9322, KFA Julich, Germany, 1993, `www.fz-juelich.de/zam/docs/printable/ib/ib-93/ib-9322.ps`.
6. Frank Cappello and Daniel Etiemble, *MPI versus MPI+OpenMP on the IBM SP for the NAS benchmarks*, in Proc. Supercomputing'00, Dallas, TX, 2000. `http://citeseer.nj.nec.com/cappello00mpi.html`
7. William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren, *Introduction to UPC and Language Specification*, CCS-TR-99-157, May 13, 1999, `http://www.super.org/upc/`, `www.gwu.edu` and `http://projects.seas.gwu.edu/~hpcl/upcdev/upctr.pdf`.
8. 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`.
9. Tarek El-Ghazawi, and Sébastien Chauvin, *UPC Benchmarking Issues*, proceedings of the International Conference on Parallel Processing, 2001, pp 365–372, `http://projects.seas.gwu.edu/~hpcl/upcdev/UPC_bench.pdf`.
10. W. Gropp and E. Lusk and N. Doss and A. Skjellum, *A high-performance, portable implementation of the MPI message passing interface standard*, in Parallel Computing 22–6, Sep. 1996, pp 789–828.

11. Jonathan Harris, *Extending OpenMP for NUMA Architectures*, in proceedings of the Second European Workshop on OpenMP, EWOMP 2000, www.epcc.ed.ac.uk/ewomp2000/.
12. 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
13. 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.
14. 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.
15. John Merlin, *Distributed OpenMP: Extensions to OpenMP for SMP Clusters*, in proceedings of the Second European Workshop on OpenMP, EWOMP 2000, www.epcc.ed.ac.uk/ewomp2000/.
16. Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Rel. 1.1, June 1995, www.mpi-forum.org.
17. Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997, www.mpi-forum.org.
18. 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.
19. The NANOS Project, Jesus Labarta, et al, http://research.ac.upc.es/hpc/nanos/.
20. R. W. Numrich, and J. K. Reid, *Co-Array Fortran for Parallel Programming*, ACM Fortran Forum, volume 17, no 2, 1998, pp 1–31, www.co-array.org and ftp://matisa.cc.rl.ac.uk/pub/reports/nrRAL98060.ps.gz.
21. OpenMP Group, www.openmp.org.
22. 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/Thera, Greece, LNCS 2131, Y. Cotronis, J. Dongarra (Eds.), Springer, 2001, pp 24–35.
23. Rolf Rabenseifner, *Effective Bandwidth (b_eff) and I/O Bandwidth (b_eff_io) Benchmark*, www.hlrs.de/mpi/b_eff/ and www.hlrs.de/mpi/b_eff_io/.
24. 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
25. Alex Scherer, Honghui Lu, Thomas Gross, Willy Zwaenepoel, *Transparent Adaptive Parallelism on NOWs using OpenMP*, in proceedings of the Seventh Conference on Principles and Practice of Parallel Programming (PPoPP '99), May 1999, pp 96–106.
26. Weisong Shi, Weiwu Hu, and Zhimin Tang, *Shared Virtual Memory: A Survey*, Technical report No. 980005, Center for High Performance Computing, Institute of Computing Technology, Chinese Academy of Sciences, 1998, www.ict.ac.cn/chpc/dsm/tr980005.ps.
27. Lorna Smith and Mark Bull, *Development of Mixed Mode MPI / OpenMP Applications*, in proceedings of Workshop on OpenMP Applications and Tools (WOMPAT 2000), San Diego, July 2000.