To be published in the International Journal of High Performance Computing Applications, 2002. ©Sage Publications.

Communication and Optimization Aspects of Parallel Programming Models on Hybrid Architectures

Rolf Rabenseifner* and Gerhard Wellein†

Summary

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, compiler based parallelization, and other parallel programming models on hybrid architectures. The paper focuses on bandwidth and latency aspects, and also whether programming paradigms can separate the optimization of communication and computation. Benchmark results are presented for hybrid and pure MPI communication. This paper analyzes the strength and waekness of several parallel programming models on clusters of SMP nodes.

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. Such hybrid systems range from small clusters of dual-CPU PCs up to the largest systems, e.g., the ASCI systems in USA [1], or the Earth Simulator in Japan with 640 nodes each equipped with 8 vector CPUs and with an aggregated peak performance of 40 TFLOP/s [8, 27, 30], ranked number 1 in the TOP500 lists in 2002 [20]. To achieve a minimal parallelization overhead, often a hybrid programming model is proposed, e.g., OpenMP [24] or automatic compiler based thread parallelization inside of each SMP node, and message passing (e.g., with MPI [18]) on the node interconnect.

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, e.g., as reported with the climate model calculations of one of the Gordon Bell Prize finalists at SC 2001 [16], or sometimes losses are reported compared to the pure MPI model, e.g., as shown with an discrete element modeling algorithm in [13]. 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 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, and pure MPI on hybrid architectures in Sect. 5. Sect. 6 presents benchmark results of the communication in both models. Sect. 7 compares the MPI based programming models with compiler based parallelization. In Sect. 8, we analyze the optimization of communication and computation, its separation (as used in the MPI model), and its combination as used in OpenMP on clusters.

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

†Regionales Rechenzentrum Erlangen, Martensstraße 1, D-91058 Erlangen, Germany
gerhard.wellein@rrze.uni-erlangen.de

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 [12] or on data distribution extensions [17]. These cluster extensions may also benefit from the availability of software-based shared virtual memory (SVM, or distributed shared memory [DSM]) [4, 28, 29]. 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 arrays 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 [7]. 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 [23] or Unified Parallel C (UPC) [6, 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) compiling 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 [19]. These library-based methods allow to store (fetch) data to (from) the memory of another process in a SPMD environment with a distributed memory model. 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) [18, 19]. 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 between 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 comparison 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 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 [5] for the NAS parallel benchmarks. Different SMP parallelization strategies in the hybrid model are also studied in [31]. High Performance Fortran (HPF) is also available on clusters of SMPs. In [3], HPF based on hybrid MPI+OpenMP is compared with pure MPI.

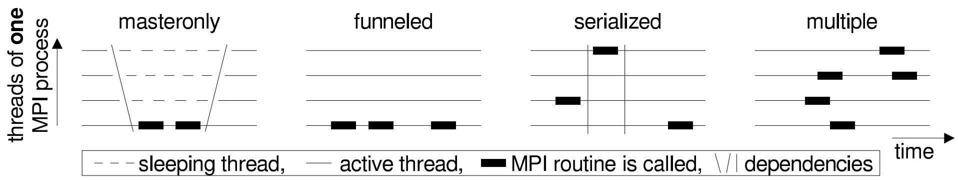

Figure 1: Timeline diagrams showing the different MPI thread-support categories of MPI-2.

3 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 [19]. 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 MPI_THREAD_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].

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

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, if an MPI library supports T1b but not more, this library 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.

The diagrams in Figure 1 illustrate the different MPI thread-support categories.

4 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 the master section nor at the exit. 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 arise with FUNNELED and SERIALIZED because the communication must be funnelled 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 evaluates this hybrid model by comparing it with the non-hybrid pure MPI model described in the next section.

5 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 is not 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 ($\uparrow\downarrow \rightleftarrows \swarrow\nearrow$). 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 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 $2\times2\times2$ 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) is twice the number of bytes in the hybrid model. The same ratio is shown in the topology in Fig. 2. In the symmetric case, the intra-node and inter-node communication has the same transfer volume.

6 Benchmark Results

The following benchmark results 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

		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.

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 [15, 25]. 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);

The benchmark measures a set of different message sizes for each pattern. For each pattern, the following subcategories are presented in the columns of 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.

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,
- s_{MPP}/s_{hybrid}: Based on the considerations in Sect. 5, we assume that the ratio of the transferred data (s_{MPP}/s_{hybrid}) is 2,
- T_{hybrid}/T_{MPP}: To compare the comunication time of both models, one must take into consideration the measured bandwidth values and the assumption that the total amount of transferred bytes is larger in the MPP model. The ratio is given by $T_{hybrid}/T_{MPP} = (s_{hybrid}/b_{hybrid})/s_{MPP}/b_{MPP})$ $= (b_{MPP}/b_{hybrid})/(s_{MPP}/s_{hybrid})$

Note that this comparison was done without any especially 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 available inter-node bandwidth due to software implementation problems or hardware design.

Figure 2 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

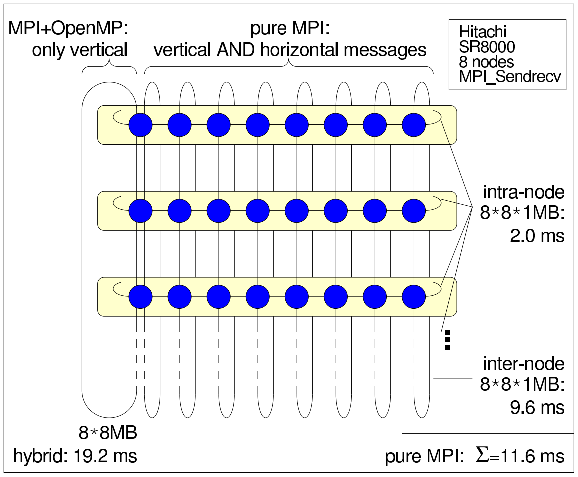

Figure 2: Parallel communication in a cartesian topology.

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. Figure 3 shows the measured transfer time for several message sizes (left diagram) and the ratio of the transfer time in the hybrid model to the transfer time of inter-node plus intra-node communication (right diagram). Note that for the hybrid 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 diagrams show 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.

Fig. 4 shows the same behavior on an IBM SP system with 16 Power3 CPUs on each SMP node. The intra-node communication is done as a ring with all 16 CPUs. For the inter-node communication, the aggregated messages in the hybrid test-case are therefore 16 times longer than in the pure MPI experiment. The left diagram shows one experiment with 16 SMP nodes, i.e., with 256 CPUs; the left (logarithmic) scale is used for the execution time, the right scale for the ratio of the execution time of the hybrid experiment to the time of the pure MPI experiment. The right diagram shows this ratio for several experiments with different numbers of nodes. Each node has 2 network adapters. Ratio values larger than 2 show that the master thread is not able to saturate (or use) both adapters.

A similar communication behavior can be expected on other platforms if the inter-node network cannot be saturated by a single processor in each SMP node. Reasons are that each node can have more than one network adapter and one CPU cannot saturate all adapters of a node, or that internal local MPI copying (e.g., from user space to a system buffer) cannot be overlapped with the real inter-node communication. E.g., on the Earth Simulator [8], the inter-node network can be saturated by one thread only, if the application buffers are located in the global memory by the application: 11.76 GB/s inter-node ping-pong MPI bandwidth are reported in [33]; the maximum rate of the link from each SMP node to the crossbar switch is 12.3 GB/s. If the application buffers are not allocated in the global memory, then additional copying between local and global memory must be executed and the single-thread inter-node bandwidth is reduced to about 60 % of the global memory inter-node ping-pong bandwidth. In this case, only the parallel usage of multiple threads (with hybrid MPI+OpenMP) or processes (pure MPI) can saturate the inter-node network.

The shown ratio of hybrid to pure MPI transfer time may be a major reason for an application running faster in the pure MPI model than in the hybrid model.

7 Comparison

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.

7.1 Hybrid MPI+OpenMP versus pure MPI

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 in the inter-node communication. The number of MPI processes is reduced. This may cause a bet-

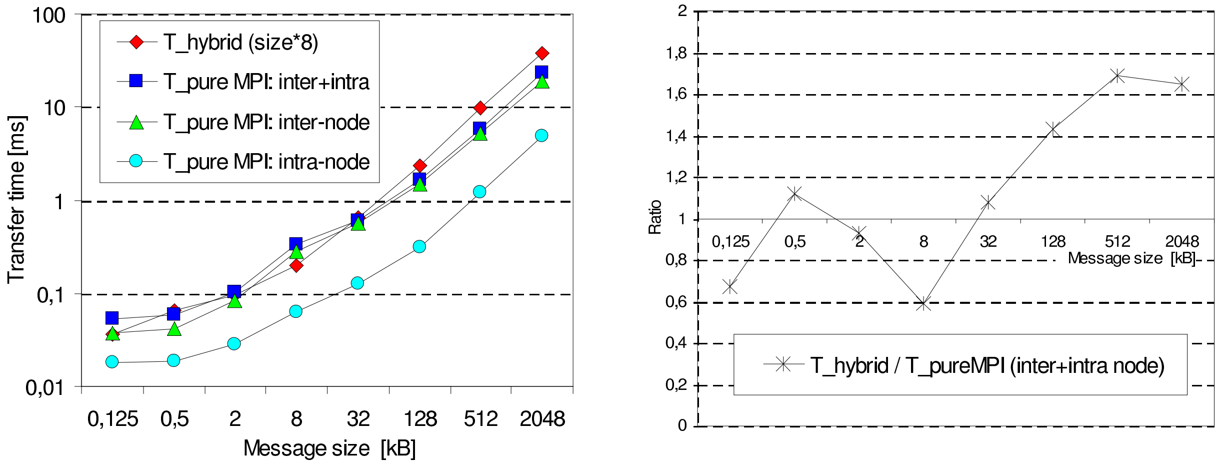

Figure 3: Benchmark results comparing hybrid MPI+OpenMP with pure MPI on a Hitachi SR8000 with 8 CPUs per SMP node.

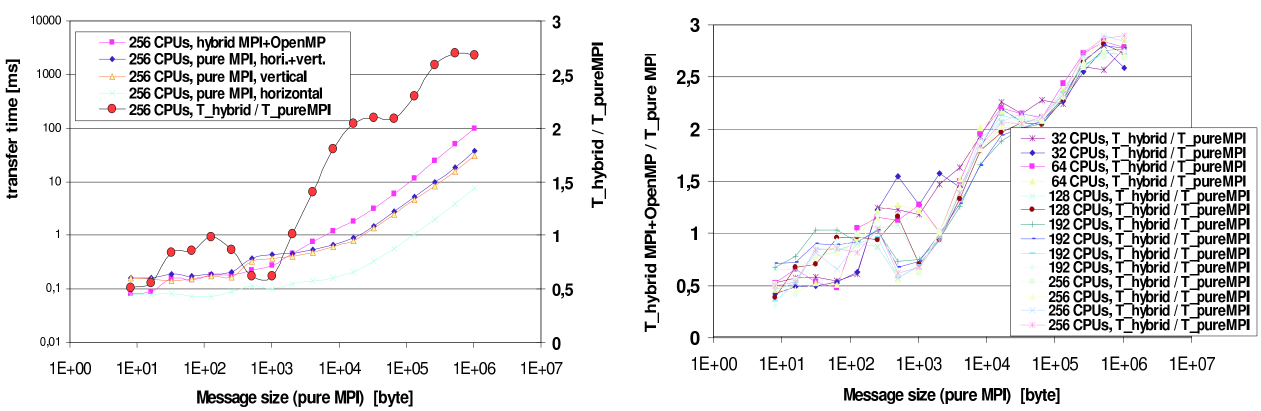

Figure 4: Benchmark results comparing hybrid MPI+OpenMP with pure MPI on an IBM SP with 16 Power3 CPUs per SMP node.

ter 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:

- 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.
- 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.

7.2 Comparing hybrid MPI+OpenMP programming schemes

Now we want to compare three different hybrid programming schemes: In the ***masteronly*** scheme, only the master thread communicates 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.

7.2.1 Reserved threads for communicating

Wellein et al. compared in [32] the two hybrid programming schemes *masteronly* (named vector-mode by Wellein et al.) 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 $1 - \frac{1}{n} \leq \epsilon \leq 2 - \frac{1}{n}$ if n is the number of threads of each SMP node, and one thread is reserved for communication. In general, m threads are reserved for communication. $T_{masteronly}$ is the wall-clock execution time with the *masteronly* programming scheme. It can be divided into three fractions: $f_{comm}T_{masteronly}$ is the communication time consumed by the master thread; $f_{comp}T_{masteronly}$ is the wall-clock computation time, consumed by all threads in parallel. Only parts of this fraction can be overlapped with communication in the *funneled* or *multiple* scheme. For this, the computation fraction must be divided into $f_{comp,non}$ and $f_{comp,overlap}$, and the sum is:

$$f_{comm} + f_{comp,non} + f_{comp,overlap} = 1 \qquad (1)$$

In the *funneled* scheme, $m = 1$ thread is reserved for communication and $n - m$ threads are used for computation. In the *multiple scheme*, m may be any value, but based on the results in the last section, m should not be chosen larger than the number of CPUs needed to saturate the communication network. The natural lower bound for m is given by $m_{min} = \frac{f_{comm}}{f_{comp,overlap}+f_{comm}}$. If we expect no further overhead by using the *multiple* scheme and if we expect that the $f_{comp,non}$ fraction is parallelized on all n threads, while $f_{comp,overlap}$ is parallelized only on the remaining $n - m$ threads, then the execution time is

$$\begin{aligned} T_{multiple} = \Big[\quad & f_{comp,non} \\ & + \max(f_{comm}\frac{1}{m}, f_{comp,overlap}\frac{n}{n-m}) \\ \Big] \quad & T_{masteronly} \end{aligned} \qquad (2)$$

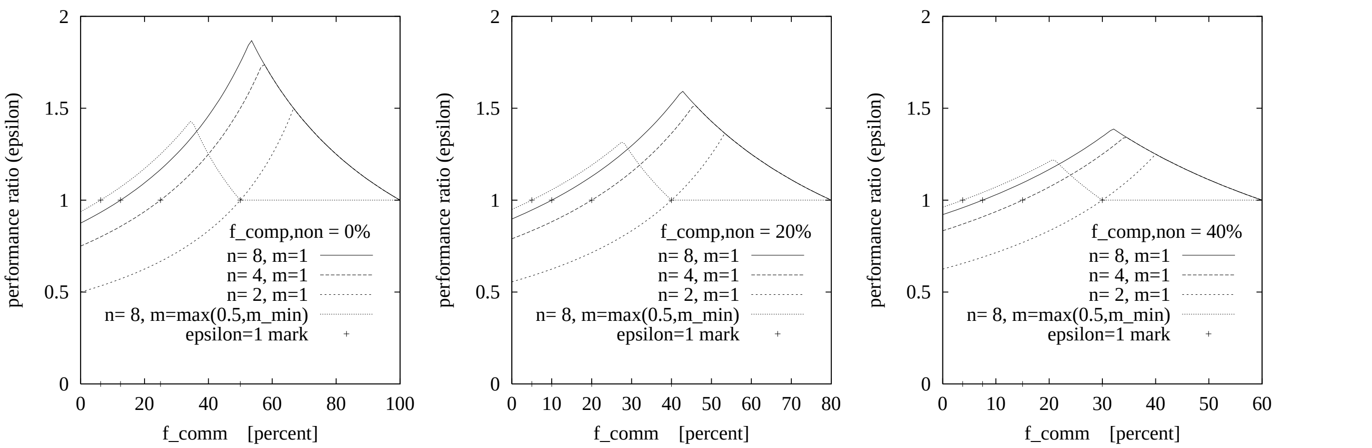

Figure 5: The performance ratio ϵ plotted according Equation (3).

Therefore, the performance ratio is

$$\epsilon = [f_{comp,non} + \max(f_{comm}\frac{1}{m}, f_{comp,overlap}\frac{n}{n-m})]^{-1} \tag{3}$$

The best performance ratio can be achieved if all CPUs are busy with communication or computation. In this case both terms in max(...,...) in (3) must be equal, i.e.,

$$f_{comm}\frac{1}{m} = f_{comp,overlap}\frac{n}{n-m} \tag{4}$$

Then, $\epsilon = [f_{comp,non} + f_{comm}\frac{1}{m}]^{-1}$ and with (4) and (1) the performance ratio of the best case is

$$\epsilon_{max} := \frac{1 + m(1-\frac{1}{n})}{1 + f_{comp,non}m(1-\frac{1}{n})} \tag{5}$$

This best ratio can be achieved if f_{comm} satisfies (4) or with (1), if

$$f_{comm} = f_{comm,best} := \frac{1}{1+\frac{1}{m}-\frac{1}{n}}(1 - f_{comp,non}) \tag{6}$$

The value ϵ_{max} is an upper bound for ϵ, i.e., for any fractions f_{comm}, $f_{comp,non}$, and $f_{comp,overlap}$,

$$\epsilon \le \frac{1 + m(1-\frac{1}{n})}{1 + f_{comp,non}m(1-\frac{1}{n})} \le 1 + m(1-\frac{1}{n}) \tag{7}$$

If $f_{comm} > f_{comm,best}$ and $m \ge m_{min}$ then always the *multiple* scheme is better than the *masteronly* scheme, i.e., $\epsilon > 1$.

The upper bound expresses the chance of a performance gain, if the load balancing is done in a way that the first thread(s) is (are) communicating and computing and the other threads are only computing, and there is no idle time due to a bad balancing.

On the other hand, what are the risks with the *funneled* and *multiple* scheme? A performance loss can emerge, if more threads are reserved for communication than needed, i.e., if these threads idle therefore. Then, the term $\frac{n}{n-m}$ in (3) reduces the performance:

$$\begin{aligned}
\epsilon &= [f_{comp,non} + f_{comp,overlap}\tfrac{n}{n-m}]^{-1} && (8)\\
&= [f_{comp,non} + (1 - f_{comp,non} - f_{comm})\tfrac{n}{n-m}]^{-1} \\
&\ge [f_{comp,non} + (1 - f_{comp,non})\tfrac{n}{n-m}]^{-1} \\
&= \left(1 - \tfrac{m}{n}\right) / \left(1 - f_{comp,non}\tfrac{m}{n}\right) =: \epsilon_{min} && (9)\\
&\ge 1 - \tfrac{m}{n}
\end{aligned}$$

Both schemes have the same performance, i.e. $\epsilon = 1$, if $f_{comm} = f_{comm,equiv} := \frac{m}{n}(1 - f_{comp,non})$. The proof is directly based on (8) and (1).

In reality, the performance win may be worse, because normally the separation of the computational parts that can be overlapped with the communication from those computational parts that need some information from neighbor processes causes some overhead. In the case of vector processing, additional effort may be necessary to achieve a long vector size in the *funneled* and *multiple* programming scheme.

For example, if $m = 1$, $n = 8$, and $f_{comp,non} = 20\%$, then $\epsilon_{max} = 1.60$ (5) is achived for $f_{comm,best} = 43\%$ (6). For $f_{comm,equiv} = 10\%$, the performance of both models are equal, i.e., $\epsilon = 1$. And for smaller f_{comm}, the ratio can decrease to $\epsilon_{min} = 0.90$ for $f_{comm} = 0\%$ (9). Fig. 5 shows the performance ratio for different parammeters.

7.2.2 Experimental results with reserved communication threads

The basic principles as discussed in Fig. 5 also hold for real world applications, e.g. sparse matrix-vector-multiplication (MVM) on SMP clusters. We selected the hybrid parallel implementation of sparse MVM as described in Ref. [32]. Of course, scalability of MVM strongly depends on the sparsity pattern, thus we consider the matrix representation of the seven point discretisation of the differential operator on a three dimensional Cartesian grid (with periodic boundary conditions). The conversion of Cartesian coordinates ($\{i,j,k\}$) to a linear index l is defined as follows

$$\{i,j,k\} \longrightarrow l = i + (j-1)\cdot n_i + (k-1)\cdot n_i \cdot n_j \quad (10)$$
$$(i = 1,\ldots,n_i\ ;\ j = 1,\ldots,n_j\ ;\ k = 1,\ldots,n_k)$$

and block-wise parallelisation using $n_{\rm proc}$ MPI processes is done along the k-direction ($n_k^{\rm loc} = n_k/n_{\rm proc}$), the communication scheme is independent of problem sizes and involves nearest neighbor communication only. Therefore, we can easily control the communication and computation costs as a function of problem size and MPI processes for the *masteronly* scheme:

$$f_{comm}T_{masteronly} = x_{comm} \times n_i \times n_j \quad (11)$$
$$f_{comp,non}T_{masteronly} = x_{MVM,non} \times n_i \times n_j \times n_k^{loc} \quad (12)$$
$$f_{comp,overlap}T_{masteronly} = x_{MVM,overlap} \times n_i \times n_j \times n_k^{loc} \quad (13)$$

In this approach only the dominant contributions to the total computing time are considered, with x_{comm} representing the MPI communication costs and $x_{MVM,overlap}$ ($x_{MVM,non}$) measuring the (non–) overlapping part of the total MVM computation time. According to Eqs. (1)–(3), the performance ratio ϵ can easily be calculated as a function of the problem size:

$$\epsilon = \Big(x_{comm} + x_{MVM,non} \times n_k^{loc} + x_{MVM,overlap} \times n_k^{loc} \Big) \Big/ \Big(x_{MVM,non} \times n_k^{loc} + \max(\frac{1}{m} \times x_{comm},$$

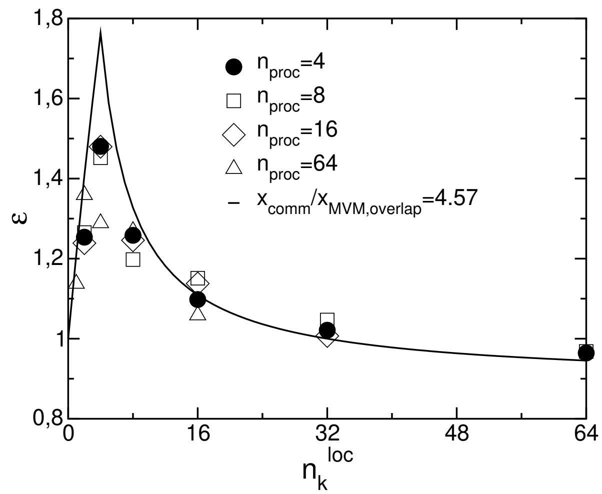

Figure 6: Performance ratio ϵ for sparse MVM algorithm: Measurements were performed with a maximum of $n_{proc} = 64$ Hitachi SR8000-F1 nodes ($n = 8; m = 1$) using $n_i = n_j = 512$. The dotted line is plotted according to Eq. (14) with $x_{comm}/x_{MVM,overlap} = 4.57$. The matrix dimension of the sparse matrix used in the MVM step is given as follows: $D_{mat} = 512 \times 512 \times n_k^{loc} \times n_{proc}$.

$$\frac{n}{n-m} \times x_{MVM,overlap} \times n_k^{loc})) \quad (14)$$

Furthermore, for the sparsity pattern described above and the parallel MVM implementation as introduced in [32] $x_{MVM,non}/x_{MVM,overlap} = 1/6$ holds and there is only one adjustable parameter ($x_{comm}/x_{MVM,overlap}$) left in Eq. (14).

Results on a Hitachi SR8000 system

As a testcase we fixed $n_i = n_j = 512$ and varied n_k^{loc} at different number of MPI processes, i.e. the total communication cost remained constant while the local workload per process was changed. Performance measurements of ***masteronly*** scheme and ***multiple*** scheme were done with up to 64 SMP nodes on the Hitachi SR8000-F1 ($n = 8; m = 1$) at LRZ Munich and with up to 16 SMP nodes on the IBM SP-Power3 at NERSC ($N = 16; m = 1$). The corresponding performance ratios are plotted in Fig. 6 and Fig. 7 as a function of local workload per node. For the Hitachi SR8000-F1, we find consistent with Eq. (14) that at fixed n_k^{loc} only a weak dependence with the number of nodes can be found. Moreover, the complex interplay of communication and computation costs as described in the theory above can be found in the experimental results: The *multiple* scheme is favored ($\epsilon > 1$) at low and intermedi-

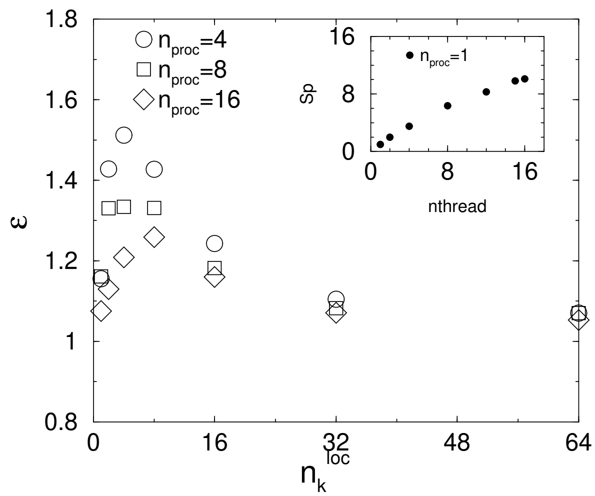

Figure 7: Performance ratio ϵ for sparse MVM algorithm: Measurements were performed with a maximum of $n_{proc} = 16$ IBM SP-Power3 nodes ($n = 16; m = 1$) using $n_i = n_j = 512$. The matrix dimension of the sparse matrix used in the MVM step is given as follows: $D_{mat} = 512 \times 512 \times n_k^{loc} \times n_{proc}$. The inset shows the Speed-Up (Sp) on one node using different number of threads (nthreads).

ate local workloads, while a crossover-point $\epsilon \approx 1$ occurs around $n_k^{loc} \approx 32$; for higher local workload the (fixed) communication cost is too small to be paid off by a separate thread spent for communication. To compare the measurements in Fig. 6 with the performance ratio as predicted by Eq. (14) $x_{comm}/x_{MVM,overlap} = 4.57$ has been chosen. This choice fixes the crossover-point $\epsilon = 1$ at $n_k^{loc} = 32$ in Eq. (14) and represents a realistic number, e.g., $x_{comm}/x_{MVM,overlap} \approx 4.17$ was estimated from a profiling run with $n_{proc} = 4$ and $n_k^{loc} = 4$. Although the total problem sizes cover more than two orders of magnitude ($D_{mat} \approx 1\times10^6, \ldots, 2.5\times10^8$) as well as a large range of number of nodes ($n_{proc} = 4, \ldots, 64$) is considered, we find a very good qualitative agreement between the theoretical approach and the measurements for the whole range of workloads. Regarding the maximum performance gain achieved by the *multiple* mode, the theoretical approach gives a good approximation for the position but overestimates the absolute value.

Results on IBM SP

These results also hold qualitatively for the IBM SP system (cf. Fig. 7), where two features have to be discussed in more detail: First, although the position of the maximum of ϵ is again independent of the number of nodes used, the absolute values of ϵ now decrease with increasing node counts at small values of n_k^{loc}. This effect may depend on the position of the nodes in the network (which comprises more than 200 nodes) and is intended to be examined in future work. Second, in contrast to the Hitachi we cannot find ϵ decreasing below 1. This fact, however, can easily be understood, if we take into account that the memory bandwidth within the SMP node does not scale with the number of processors. As can be seen from the inset of Fig. 7 a speed-up of only 9.8 (10.1) is achieved on one node when using 15 (16) threads instead of one thread. In other words, spending one thread for communication does not increase almost the time for the overlapping computations in the multiple model as assumed in Eq. (2). Thus the effective value of m in Eq. (9) is approximately zero, giving a lower bound $\epsilon_{min} \approx 1$. Please note, in case of the Hitachi the memory bandwidth within one node can saturate the single processor bandwidth of all eight processors. A speed-up of more than 7.2 has been demonstrated on eight processors of one Hitachi node [11]. Of course a detailed description of communication and computation – going beyond the simple approach in Eqs. (11)–(13) – is required to improve the quantitative agreement between the theoretical model and the measurements.

7.3 MPI versus Compiler-based Parallelization

After comparing the pure MPI and different MPI based hybrid parallel programming models, we now 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., [7] reports on a ccNUMA system a latency of 0.33–1 μs, which implies

Access method	copies	remarks	bandwidth $b(message\ size)$
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.

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 [21], but this method is still not used in all compilers. Hess et al. [14] have shown that one can reach with OpenMP on top of a DSM system speed-ups ranging between 70 % and 100 % of the MPI speed-up, except if the communication fraction is high.

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 [26], 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 still one has a single source for serial and parallel compilation. On a cluster of SMP nodes, OpenMP has also two major disadvantages: OpenMP is based on a flat memory model (it knows only *private* and *shared* variables) and there is not any strategy to achieve a contiguous data transfer to access shared array variables in a remote memory. Especially this second problem makes it difficult 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 [12], 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 [12, 17]). 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 *auto-parallel* 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, 22]. 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 interconnect. On such architectures, the standard programming models of SMP or MPP systems do not longer fit well. This paper identifies three major problems.

First, using only one thread per node to perform the inter-node communication and calling the MPI routines only outside of parallel regions, one cannot achieve the full inter-node bandwidth. In the benchmark with the Cartesian topology, this hybrid *masteronly* MPI+OpenMP model needs about 60–100 % more communication time than communicating in a pure MPI model, although additional intra-node message transfers were necessary with pure MPI. This problem may have several reasons: One thread cannot use different hardware links at the same time, or the MPI library is not able to saturate the network due to additional internal memory copying and a limited memory bandwidth on each thread. If the overall communication percentage is only in the area of less than 15 %, then this effect of the hybrid *masteronly* model only causes an additional overhead of about 9–15 %. Together with the OpenMP based synchronization overhead, a 20 % larger execution time of hybrid programs is not a surprise.

Second, we compare the masteronly model with a hybrid model where we **overlap** communication and computation. Instead of implementing a full load balancing, we decided to **funnel** the communication through the master thread and to **reserve** one thread for communication while the other threads are used for computation. On a cluster with 8-way SMP nodes, we can exhibit, that theoretically the total application can run 1.75 times faster than in the masteronly model. This factor depends on the fraction used for communication and on the fraction of computation that can be overlapped with communication. In the experiment we can demonstrate a factor of 1.5. Unfortunately, this fast hybrid *overlapping&funnel&reserve* model does not allow to use the sophisticated work-sharing constructs of OpenMP. OpenMP is reduced to an interface for basic thread usage. Work-sharing must be done by the application program based on the ranks of the threads.

And third, we look at compiler based parallelization on hybrid platforms. Here, two problems arise: a) How can we achieve a communication speed that is in the area of the hardware inter-node bandwidth, and b), does the cluster-capable compiler deliver the same computational optimizations as the SMP-local compiler? Both of these problems are still unsolved. Therefore, at least for programs with a relevant communication fraction, MPI based parallelization (pure or combined with OpenMP) is still the method of first choice.

Future work will include the examination of mixing MPI and OpenMP inside of SMP nodes to benefit together from the saturation of the inter-node network and from the hybrid programming.

Acknowledgments

The authors would like to acknowledge their colleagues and all the people that supported these projects with suggestions and helpful discussions. We 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, and Thomas Bönisch, Matthias Müller, Uwe Küster, and John M. Levesque for discussions on OpenMP cluster extensions and vectorization. This research used resources of the HLRS Stuttgart and LRZ Munich, and resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy. Part of this work was supported by KONWIHR project *cxHPC*.

Biographies

Rolf Rabenseifner studied mathematics and physics at the University of Stuttgart. Since 1984, he is working at the High Performance Computing Center Stuttgart (HLRS). He led the projects DFN-RPC, a remote procedure call tool, and MPI-GLUE, the first metacomputing MPI combining different vendor's MPIs without loosing the full MPI interface. In his dissertation, he developed a controlled logical clock as global time for trace-based profiling of parallel and distributed applications. Since 1996, he has been a member of the MPI-2 Forum. From January to April 1999, he was an invited reseacher at the Center for High-Performance Computing at Dresden University of Technology. Currently, he is head of the Department for Parallel Computing of the HLRS. His research interests include hybrid parallel programming models, MPI profiling, and benchmarking of parallel communication and I/O. He is also involved in teaching projects.

Gerhard Wellein studied physics at the University of Bayreuth (Germany) from 1989-1994. Working in the field of exact diagonalization of strongly correlated electron-lattice systems he got his doctorate degree in theoretical physics in 1998 (University of Bayreuth). He then started as an HPC consultant at the Scientific Computing Center Erlangen (RRZE). Since 2000 he is a member of the Competence Network for Technical and Scientific High Performance Computing in Bavaria (KONWIHR). Currently, he is head of the HPC group at RRZE. His research interests include optimization strategies for new processor architectures, exact diagonalization and density matrix renormalization group methods for strongly correlated quantum systems on parallel computers and hybrid parallel programming techniques for large scale application programs. He is also involved in research projects in condensed matter physics.

References

[1] Advanced Simulation and Computing (ASC, formerly Accelerated Strategic Computing Initiative [ASCI]). http://www.nnsa.doe.gov/asc/home.htm

[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. www.it.lth.se/ewomp99/papers/ayguade.pdf

[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] R. Berrendorf, M. Gerndt, W. E. Nagel and J. Prumerr, *SVM Fortran*, Technical Report IB-9322, KFA Jlich, Germany, 1993. www.fz-juelich.de/zam/docs/printable/ib/ib-93/ib-9322.ps

[5] 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 www.sc2000.org/techpapr/papers/pap.pap214.pdf

[6] 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.

[7] 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.

[8] The Earth Simulator. www.es.jamstec.go.jp

[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] G. Hager, F. Deserno and G. Wellein, *Pseudo-Vectorization and RISC Optimization Techniques for the Hitachi SR8000 Architecture*, accepted for publication in High Performance Computing in Science and Engineering in Munich '02, Springer-Verlag Berlin Heidelberg, 2003.

[12] Jonathan Harris, *Extending OpenMP for NUMA Architectures*, in proceedings of the Second European Workshop on OpenMP, EWOMP 2000. www.epcc.ed.ac.uk/ewomp2000/proceedings.html

[13] 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`
`www.sc2000.org/techpapr/papers/pap.pap154.pdf`

[14] Matthias Hess, Gabriele Jost, Matthias Müller, and Roland Rühle, *Experiences using OpenMP based on Compiler Directed Software DSM on a PC Cluster*, in WOMPAT2002: Workshop on OpenMP Applications and Tools, Arctic Region Supercomputing Center, University of Alaska, Fairbanks, Aug. 5–7, 2002.

[15] 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, p. 196.

[16] 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.
`www.sc2001.org/papers/pap.pap189.pdf`

[17] 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/proceedings.html`

[18] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Rel. 1.1, June 1995, `www.mpi-forum.org`.

[19] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997, `www.mpi-forum.org`.

[20] Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst D. Simon, Universities of Mannheim and Tennessee, *TOP500 Supercomputer Sites*, `www.top500.org`.

[21] Matthias M. Müller, *Compiler-Generated Vector-based Prefetching on Architectures with Distributed Memory*, in High Performance Computing in Science and Engineering '01, Transactions of the High Performance Computing Center Stuttgart (HLRS), W. Jäger and E. Krause (eds), Springer, 2001, pp 527–539.

[22] The NANOS Project, Jesus Labarta, et al., `http://research.ac.upc.es/hpc/nanos/`.

[23] 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`.

[24] OpenMP Group, `www.openmp.org`.

[25] 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/`.

[26] 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`

[27] Tetsuya Sato, *The Earth Simulator*, SC2002, Baltimore, Nov. 16-22, 2002. `www.sc2002.org`

[28] 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.

[29] 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`.

[30] Satoru Shingu, Hiroshi Takahara, Hiromitsu Fuchigami, Masayuki Yamada, Yoshinori Tsuda, Wataru Ohfuchi, Yuji Sasaki, Kazuo Kobayashi, Takashi Hagiwara, Shin-ichi Habata, Mitsuo Yokokawa, Hiroyuki Itoh and Kiyoshi Otsuka, *A 26.58 Tflops Global Atmospheric Simulation*

with the Spectral Transform Method on the Earth Simulator, SC2002, Baltimore, Nov. 16-22, 2002. www.sc2002.org/paperpdfs/pap.pap331.pdf

[31] 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. www.cs.uh.edu/wompat2000/

[32] G. Wellein, G. Hager, A. Basermann, and H. Fehske, *Fast sparse matrix-vector multiplication for TeraFlop/s computers*, in proceedings of VECPAR'2002, 5th Int'l Conference on High Performance Computing and Computational Science, Porto, Portugal, June 26–28, 2002, part I, pp 57–70. http://vecpar.fe.up.pt/

[33] Hitoshi Uehara, Masanori Tamura, and Mitsuo Yokokawa, *An MPI Benchmark Program Library and Its Application to the Earth Simulator*, in proceedings of the 4th International Symposium on High Performance Computing, ISHPC 2002, H. Zima et al. (Eds.), Kansai Science City, Japan, May 15-17, LNCS 2327, Springer, 2002, pp 219–230.

The following addenda are not part of the published paper.

9.0.1 Addendum 1

Proof of (5):

$$\begin{aligned}
(4) \text{ and } (1) \quad &\Rightarrow \quad f_{comm}(\frac{1}{m} + \frac{n}{n-m}) = (1 - f_{comp,non})\frac{n}{n-m} \\
\epsilon \quad &= \quad [f_{comp,non} + f_{comm}\frac{1}{m}]^{-1} \\
&= \quad [f_{comp,non} + (1 - f_{comp,non})\frac{n}{n-m}\frac{1}{\frac{1}{m} + \frac{n}{n-m}}\frac{1}{m}]^{-1} \\
&= \quad [f_{comp,non} + (1 - f_{comp,non})\frac{n}{n-m+nm}]^{-1} \\
&= \quad [f_{comp,non}\frac{1-m/n+m}{1-m/n+m} + (1 - f_{comp,non})\frac{1}{1-m/n+m}]^{-1} \\
&= \quad (1-m/n+m)[1 + f_{comp,non}((1-m/n+m)-1)]^{-1} \\
&= \quad \frac{1+m(1-\frac{1}{n})}{1+f_{comp,non}m(1-\frac{1}{n})}
\end{aligned}$$

9.0.2 Addendum 2

Proof of (6):

$$\begin{aligned}
(4) \text{ and } (1) \Rightarrow f_{comm}(\frac{1}{m} + \frac{n}{n-m}) \quad &= \quad (1 - f_{comp,non})\frac{n}{n-m} \\
f_{comm} \quad &= \quad \frac{\frac{n}{n-m}}{\frac{1}{m} + \frac{n}{n-m}}(1 - f_{comp,non}) \\
&= \quad \frac{1}{\frac{n-m}{nm} + 1}(1 - f_{comp,non}) \\
&= \quad \frac{1}{\frac{1}{m} - \frac{1}{n} + 1}(1 - f_{comp,non})
\end{aligned}$$

9.0.3 Addendum 3

Calculation of ϵ_{min} in detail:

$$\begin{aligned}
\epsilon \quad &= \quad [f_{comp,non} + f_{comp,overlap}\tfrac{n}{n-m}]^{-1} \qquad (8) \\
&= \quad [f_{comp,non} + (1 - f_{comp,non} - f_{comm})\tfrac{n}{n-m}]^{-1}
\end{aligned}$$

$$
\begin{aligned}
&\geq \quad [f_{comp,non} + (1 - f_{comp,non})\tfrac{n}{n-m}]^{-1} \\
&= \quad \tfrac{n-m}{n}[f_{comp,non}\tfrac{n-m}{n} + (1 - f_{comp,non})]^{-1} \\
&= \quad (1 - \tfrac{m}{n})[1 - f_{comp,non}(1 - \tfrac{n-m}{n})]^{-1} \\
&= \quad \left(1 - \tfrac{m}{n}\right) / \left(1 - f_{comp,non}\tfrac{m}{n}\right) =: \epsilon_{min} \qquad (9) \\
&\geq \quad 1 - \tfrac{m}{n}
\end{aligned}
$$

9.0.4 Addendum 4

Proof of the Theorem: Both schemes have the same performance ($\epsilon = 1$) if $f_{comm} = \frac{m}{n}(1 - f_{comp,non})$.

$$
\begin{aligned}
(8) \text{ and} \epsilon = 1 \to f_{comp,non} + f_{comp,overlap}\frac{n}{n-m} &= 1 \\
\text{with } (1) \to f_{comp,non} + (1 - f_{comp,non} - f_{comm})\frac{n}{n-m} &= 1 \\
f_{comp,non}(1 - \frac{n}{n-m}) + \frac{n}{n-m} - 1 &= f_{comm}\frac{n}{n-m} \\
f_{comp,non}(\frac{n-m}{n} - 1) + 1 - \frac{n-m}{n} &= f_{comm} \\
f_{comp,non}(-\frac{m}{n}) + \frac{m}{n} &= f_{comm} \\
f_{comm} &= \frac{m}{n}(1 - f_{comp,non})
\end{aligned}
$$
