Software News and Update

Speeding Up Parallel GROMACS on High-Latency Networks

CARSTEN KUTZNER,[1] DAVID VAN DER SPOEL,[2] MARTIN FECHNER,[1] ERIK LINDAHL,[3] UDO W. SCHMITT,[1]
BERT L. DE GROOT,[1] HELMUT GRUBMÜLLER[1]

[1] *Department of Theoretical and Computational Biophysics, Max-Planck-Institute for Biophysical Chemistry, Am Fassberg 11, 37077 Göttingen, Germany*

[2] *Department of Cell and Molecular Biology, Uppsala University, Husargatan 3, S-75124 Uppsala, Sweden*

[3] *Stockholm Bioinformatics Center, SCFAB, Stockholm University, SE-10691, Stockholm, Sweden*

Received 15 August 2006; Revised 2 November 2006; Accepted 21 November 2006
DOI 10.1002/jcc.20703
Published online 2 April 2007 in Wiley InterScience (www.interscience.wiley.com).

Abstract: We investigate the parallel scaling of the GROMACS molecular dynamics code on Ethernet Beowulf clusters and what prerequisites are necessary for decent scaling even on such clusters with only limited bandwidth and high latency. GROMACS 3.3 scales well on supercomputers like the IBM p690 (Regatta) and on Linux clusters with a special interconnect like Myrinet or Infiniband. Because of the high single-node performance of GROMACS, however, on the widely used Ethernet switched clusters, the scaling typically breaks down when more than two computer nodes are involved, limiting the absolute speedup that can be gained to about 3 relative to a single-CPU run. With the LAM MPI implementation, the main scaling bottleneck is here identified to be the all-to-all communication which is required every time step. During such an all-to-all communication step, a huge amount of messages floods the network, and as a result many TCP packets are lost. We show that Ethernet flow control prevents network congestion and leads to substantial scaling improvements. For 16 CPUs, e.g., a speedup of 11 has been achieved. However, for more nodes this mechanism also fails. Having optimized an all-to-all routine, which sends the data in an ordered fashion, we show that it is possible to completely prevent packet loss for any number of multi-CPU nodes. Thus, the GROMACS scaling dramatically improves, even for switches that lack flow control. In addition, for the common HP ProCurve 2848 switch we find that for optimum all-to-all performance it is essential how the nodes are connected to the switch's ports. This is also demonstrated for the example of the Car-Parinello MD code.

© 2007 Wiley Periodicals, Inc. J Comput Chem 28: 2075–2084, 2007

Key words: GROMACS parallel molecular dynamics; Car-Parrinello MD; Ethernet flow control; MPI_Alltoall; network congestion

Introduction

Numerical molecular dynamics (MD) is computationally very demanding. A protein immersed in a water box is a typical example of an MD system which often consists of several hundred thousands of atoms. A numerical simulation that covers a significant time-span—to reveal protein unfolding for example—can take up to years even on a modern parallel computer. Scientists needing to perform parallel MD can today choose from several packages, some of which are freely available, e.g. GROMACS[1] and NAMD,[2] and others that come at a licensing cost, e.g. CHARMM[3] and AMBER.[4]

The GROMACS[5,6] MD engine makes extremely efficient use of a computer's processor with highly optimized assembly code, rendering it one of the fastest MD programs available. Of course, the faster a time step is executed on a single processor, the more the parallel speedup is hindered by communication latencies. But since a low-latency network is expensive, most clusters rest on the standard Ethernet interconnect only. However, as will be shown here, tremendous scaling improvements and speedups can be achieved by digging a bit into the technical details. As will be shown, nearly the same speedups can be achieved by our improvements on simple Ethernet compared to switching to e.g. Myrinet. Accordingly, one will be rewarded by saving a lot of computer time, real time, and also money. We here point out steps that enable decent scaling on off-the-shelf technology clusters. This is not only of interest for the

Correspondence to: C. Kutzner; e-mail: ckutzne@gwdg.de

continuously increasing community of GROMACS users,[1] since much of this will also be applicable to other parallel programs that suffer from similar scaling problems.

GROMACS uses the Message Passing Interface (MPI) standard[7] for communication between the processors. At the start of a simulation the atoms are distributed to the processors (particle distribution). Coordinates and short-range nonbonded forces (pairwise Coulomb and Van-der-Waals interactions) are at each time step transferred in a ring structure,[8] similar to a systolic loop method.[9] The long-range electrostatic forces are evaluated with the Particle-Mesh-Ewald (PME) method[10,11] which is significantly more accurate than a simple cutoff treatment.

The 3.3 version of the code scales well on shared memory supercomputers like the IBM Regatta as well as on Linux clusters with a high bandwidth/low latency network like Myrinet or Infiniband. However, on cost-efficient Gigabit Ethernet clusters, parallel runs on three computer nodes are often even slower than on two nodes (each node containing one or two CPUs in our study). We find that the breakdown in scaling is due to network congestion in phases of all-to-all communication.

In this paper we investigate the all-to-all routine, as it is often the main scaling bottleneck for GROMACS on Ethernet. The MPI_Alltoall routine is called twice each time step. It performs a parallel transpose within the Fast Fourier Transformation (FFT), the latter being essential for parallel PME calculation. During an all-to-all call, each process sends a unique message to every other process. Therefore, this communication type is also referred to as all-to-all personalized communication (AAPC). In our case, the individual messages within an all-to-all have the same size.

To systematically investigate the origin of the scaling problems we benchmark both the entire application (with two typical MD systems) as well as the isolated all-to-all routine. This we do for LAM/MPI,[12–14] which is mostly used for GROMACS on Ethernet, and for MPICH,[15,16] which few GROMACS users adopt for Ethernet.

We find that the MPI_Alltoall performs significantly better when IEEE 802.3x link-layer flow control[17] is activated. For network switches not supporting flow control, we have implemented an ordered all-to-all routine that performs reliably well for typical GROMACS all-to-all message sizes. Both flow control and the ordered all-to-all routine substantially enhance the scaling properties on Ethernet.

Setup

The reported benchmarks were carried out on a Beowulf cluster consisting of 32 identical nodes running the Linux 2.6.5 kernel. The nodes had 2 GHz dual-CPU AMD Opteron processors, 1024 kB cache, 1 GB RAM, and Broadcom NetXtreme BCM5704 onboard Ethernet cards. They were connected with Gigabit Ethernet to a 48-port HP ProCurve switch (HP 2848). The switch was configured to operate in *qos-passthrough-mode*, which reduces the number of queues for traffic of different priority from 4 to 2. As a result, more of the switch's memory is available for MPI messages. In this mode, which is recommended for lossless data transfer at line speed,[18] we got better timings especially for large CPU numbers. Note that earlier benchmarks on a 32-node dual-CPU Intel Xeon cluster (2.4 kernel) with Intel 82541 onboard Ethernet cards and the same switch have shown comparable results, both for the GROMACS speedups as well as for the systematic all-to-all tests. For the latter cluster version 1.2.6 of MPICH was used.

We use either LAM 7.1.1 or MPICH-2 1.0.3 (with the ch3:ssm device) and the GNU C 3.3 compiler. Both LAM and MPICH exploit fast shared memory communication when messages are sent between CPUs on the same node. The timing behavior of the program is logged with MPICH's MPE utilities[15] which are compatible to any MPI implementation. Time measurements have been done with the MPI_Wtime timer, which for our purposes has a sufficiently high resolution of 10^{-6} s. The reported GROMACS timings are averages over 100 time steps.

For the parallel speedup measurements we chose as an MD test system an Aquaporin-1 (AQP1) protein tetramer embedded in a lipid bilayer membrane surrounded by water,[19] which comprises ca. 80,000 atoms. We also report benchmarks for the DPPC system, which is part of the benchmark suite that can be downloaded from the GROMACS homepage. It consists of 1024 dipalmitoylphosphatidylcholine (DPPC) lipids in a bilayer configuration plus water, in total ca. 120,000 atoms. For both systems, Coulomb interactions were evaluated with PME, meaning that all forces within the cutoff radius r_c are directly calculated while the long-range forces are each time step interpolated from a mesh. We chose $r_c = 1.0$ nm for AQP1 and $r_c = 1.2$ nm for DPPC.

The parallel speedup Sp_N of an application running on N processors is the execution time t_1 on 1 CPU divided by the execution time t_N on N CPUs, $Sp = t_1/t_N$. The scaling Sc_N is the speedup divided by the number of processors, $Sc_N = t_1/(N \cdot t_N)$. Some exemplary speedups for the AQP1 system on high performance networks are $Sp_8 = 6.2, Sp_{16} = 10, Sp_{32} = 12$ for Myrinet (on the Xeon Cluster), $Sp_{16} = 11$, $Sp_{32} = 14$ for Infiniband (on Sun dual-CPU 2.2 GHz Opterons), and $Sp_{32} = 21$ on one Regatta node (shared memory).

For comparison, we also performed the benchmarks with Myrinet-2000 interconnect hardware. The Myrinet cluster consisted of 2 GHz Opterons with 1024 kB cache and 1 GB RAM, and was running the 2.6.14 kernel. Here, MPICH-MX 1.2.6 was used.

GROMACS Parallel Speedups on Ethernet

With LAM, on Gigabit Ethernet the maximum speedup that can be reached with the switch's factory settings is $Sp_4 = 3.1$ for AQP1 (Fig. 1, +) and $Sp_4 = 3.4$ for DPPC (Table 1). The speedup maximum occurs for the case of two dual-CPU nodes. Employing more nodes leads to slower execution.

This behaviour does not depend upon the switch in use. We also tested a 3Com 3870, a 3Com 5500, a HP 3400CL/24, and a D-Link DGS-1016D switch for up to 10 nodes and got the same results as in the case of the HP 2848. While the DGS-1016D does not support flow control, this mechanism was disabled by default in all of the other switches.

With the help of MPE's Jumpshot program[20] a detailed picture is obtained of what each processor does during program execution. Figure 2 shows two time steps while running the AQP1 system on six CPUs (3 nodes) using LAM. The second time step is significantly longer than the first one. Most of the time is spent in a call to MPI_Alltoall, from which CPUs 4 and 6 return normally, while the others are delayed for 200–250 ms.

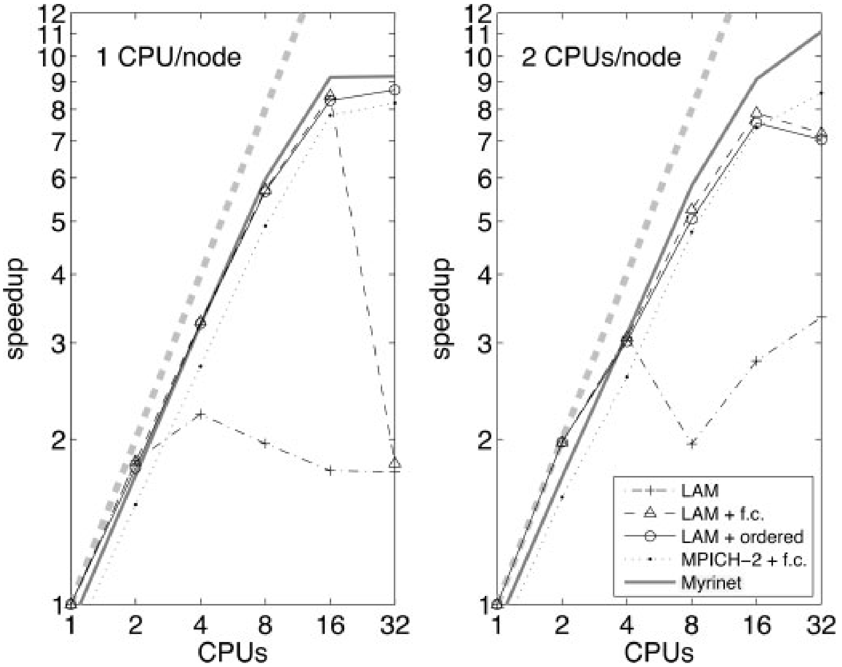

Figure 1. GROMACS 3.3.1 speedups for AQP1 employing different all-to-alls with and without link-layer flow control (f.c.). +: LAM MPI_Alltoall without f.c., △: LAM MPI_Alltoall with f.c., ∘: LAM with ordered all-to-all (same results with and without f.c.), ·: MPICH-2 MPI_Alltoall with f.c. Dashed grey line indicates ideal speedup. All speedups relative to the LAM single CPU case. For comparison, the solid grey line shows the speedups reached with a Myrinet interconnect.

These delays exclusively happen in the MPI_Alltoall and are more frequent when more nodes take part in the communication. As a result, the average time step length cannot be reduced below 200 ms for >2 nodes since then more and more MPI_Alltoall calls get delayed.

The reason for these delays are lost TCP packets due to network congestion. Lost packets show up e.g. in the *ifconfig* utility for the corresponding network interfaces or in the switch's port counters.

While e.g. OpenMPI[21] and MPICH-2 use distinct algorithms depending on message size and CPU number,[22] the all-to-all implementation in LAM and MPICH-1*, prior to version 1.2.6, is rather simple: First, all the sends and receives are initiated, then MPI_Waitall is called to wait for the communication to finish. Thus, it is not controlled in which order the messages reach the receivers. Because in an all-to-all communication on N processors altogether $N \cdot (N - 1)$ messages are sent, it becomes more and more likely with growing N that two or more senders transmit simultaneously to the same receiver. The switch can only forward part of that data to the receiver while the other data fills the switch's buffers. If these buffers are full before the senders stop sending, the switch drops excess packets. This is inefficient since the operating system has to detect the drops and then has to invoke a retransmit, resulting in the observed time delay of approximately 200 ms.

The IEEE 802.3x standard defines a flow control mechanism at the link level to prevent network congestion.[17,23,24] It gives the receiving device R, be it a switch or a computer's network interface, the possibility to send a PAUSE frame to tell the source to stop sending. The source then interrupts sending for the time period requested in the PAUSE frame or until it receives another PAUSE frame from R with a wait time of zero. Flow control has to be activated on both the computer's network interfaces and on the corresponding ports on the switch. Note that enabling flow control on the HP 2848 at

Table 1. GROMACS 3.3.1 Speedups for the DPPC System, Relative to the LAM Single-CPU Case.

	CPUs	1	4	24	8	16	32
1 $\frac{\text{CPU}}{\text{node}}$	LAM	1.00	1.96	2.7	1.9	2.0	1.9
	LAM + flow control	1.00	1.95	3.70	6.75	11.2	3.8
	LAM + ordered all-to-all	1.00	1.89	3.68	6.43	10.7	14.5
	MPICH-2 + flow control	0.97	1.93	3.37	6.58	11.8	15.8
	MPICH-1.2.6 with Myrinet	0.95	2.02	3.87	7.72	13.4	15.0
2 $\frac{\text{CPUs}}{\text{node}}$	LAM	1.00	2.08	3.44	1.8	2.7	2.8
	LAM + flow control	1.00	2.08	3.44	6.13	10.3	12.7
	LAM + ordered all-to-all	1.00	2.07	3.30	6.02	10.2	11.5
	MPICH-2 + flow control	0.97	2.02	3.28	6.15	11.5	15.1
	MPICH-1.2.6 with Myrinet	0.95	1.95	3.69	7.23	13.2	17.3

*The all-to-all source codes can be found in the following locations, relative to the parent directory of the respective distribution: LAM: ./share/ssi/coll/lam_basic/src/ssi_coll_lam_basic_alltoall.c, MPICH-pre 1.2.6: ./src/coll/intra_fns.c, MPICH-1.2.6 or later: ./src/coll/intra_fnc_new.c.

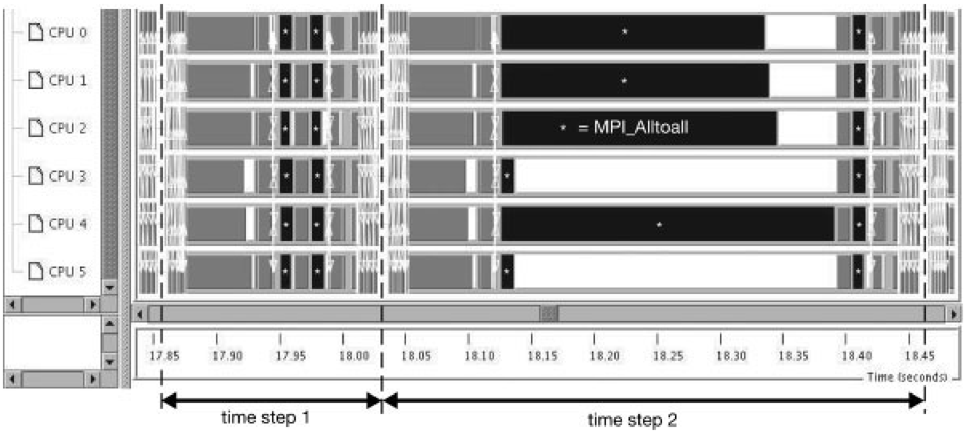

Figure 2. Time versus CPU number for two GROMACS time steps as logged with the MPE tools. First time step spans 0.18 s (t = 17.85 s–18.03 s), second time step 0.42 s (t = 18.03 s–18.45 s). Black bars (*) indicate MPI_Alltoall calls.

first showed no effect. After updating from firmware version I.07.31 to I.08.61 it worked as expected.

With active flow control, the all-to-all delays disappear for up to 16 nodes, and the overall scaling is significantly better, see Figure 1 (△) and Table 1. For AQP1, the average time-step length reduces to 68 ms on 16 single-CPU nodes, the corresponding speedups are $Sp_8 = 5.7$ and $Sp_{16} = 8.4$. The speedups on dual-CPU nodes are slightly smaller, $Sp_8 = 5.3$, $Sp_{16} = 7.8$, because the available bandwidth is shared among two CPUs each. In the case of 32 nodes there is severe packet loss even with flow control.

The dotted line in Figure 1 shows the AQP1 scaling with MPICH-2 and flow control. On up to 16 CPUs the scaling is comparable to the LAM case with flow control, on 32 CPUs the scaling is even better. The absolute execution time for this benchmark on 1–8 CPUs is about 15% less for LAM compared to MPICH. When comparing the execution times of a serial GROMACS (i.e. without MPI support) to an MPICH and a LAM/MPI capable version, it turns out that actually the LAM version is faster than the other two. This results from LAM's special memory management. When configuring LAM *--without-memory-manager* the execution times of the serial, LAM, and MPICH version on a single CPU are comparable.

It has to be noted that we experienced such a big performance difference between LAM and MPICH only for the AQP1 benchmark. For other MD systems, e.g. for the DPPC benchmark, the single-CPU execution times only differ by a few percent (see Table 1).

Systematic Tests with MPI All-to-All Communication

For a more systematic analysis we created a synthetic benchmark that performs all-to-all communication with various data volumes for different CPU numbers (download available[1]). In an all-to-all on N processors, a given process sends $N - 1$ distinct messages of size S to the $N - 1$ other processes. Hence, $N(N - 1)$ messages of size S are transferred. Let Δt be the time needed to transfer all those messages, then the throughput T per CPU is $T = (N - 1)S/\Delta t$.

Figures 3 (for LAM) and 4 (for MPICH) show T as a function of S, averaged over 25 calls. The lines and symbols indicate the average T while the individual throughputs for a given S lie within the grey vertical bars. Note that variation of the network parameters (e.g. in */proc/sys/net/ipv4/*) and NIC driver parameters (e.g. with *ethtool*) has an effect on individual throughput measurements but does not change the overall picture.

The top two panels of the figures show T without flow control. Generally one expects the throughput to rise with the message size until it converges against a maximum. With a standard MTU (maximum transmission unit), an Ethernet frame is of size 1500 byte + 26 byte TCP overhead. Thus, the maximum throughput evaluates to $T_{\max} = 1\ \text{Gbit}/s \cdot (1500/1526) \approx 117\ \text{MB}/s$ or $1/P$ of that value for nodes with P processors. In the case of multi-CPU nodes, this maximum can be exceeded because of the exploitation of on-node shared memory communication. This is seen in the case of two dual-CPU nodes, right panels of Figures 3 and 4. Moreover, for two identical nodes the chance for congestion is negligible, since exactly one full-duplex connection (node A $\rightleftharpoons$ switch $\rightleftharpoons$ node B) is involved (Fig. 3, upper right panel, △).

If more than one connection is involved, we find that at a critical S the throughput drops down an order of magnitude or more before it slowly recovers again for larger message sizes. The more CPUs take part in the communication, the more pronounced the drop is in depth and width. The critical S is lower for larger CPU numbers. This behaviour is most dramatic for the LAM case. With MPICH, packet loss degenerates performance only when using more than eight CPUs (Fig. 4, upper panels).

The GROMACS standard PME parameters (fourierspacing = 0.12 nm, PME order = 4) yield a fourier grid of $90 \times 88 \times 80 = 633{,}600$ points for the AQP1 system. The message size S that is transferred within the FFT all-to-all from each to each CPU is then 173,184 bytes (on four CPUs), 43,296 bytes (on eight CPUs), 11,808 bytes (on 16 CPUs) or 2952 bytes on 32 CPUs, respectively. A somewhat smaller system ($\approx$ 4000 atoms, a single Guanylin protein in water) yields a PME grid of $30 \times 30 \times 21 = 18{,}900$ points. This results in sizes S of 5632 bytes (on four CPUs) or 1 408 bytes (on eight CPUs). No speedups can be gained by running such a small system on more CPUs. Thus, MD systems, when run in parallel on $\geq$4 processors, typically lie in the range $S = [1500 \ldots 175{,}000]$ bytes. In Figures 3 and 4 this range is indicated by the shaded area.

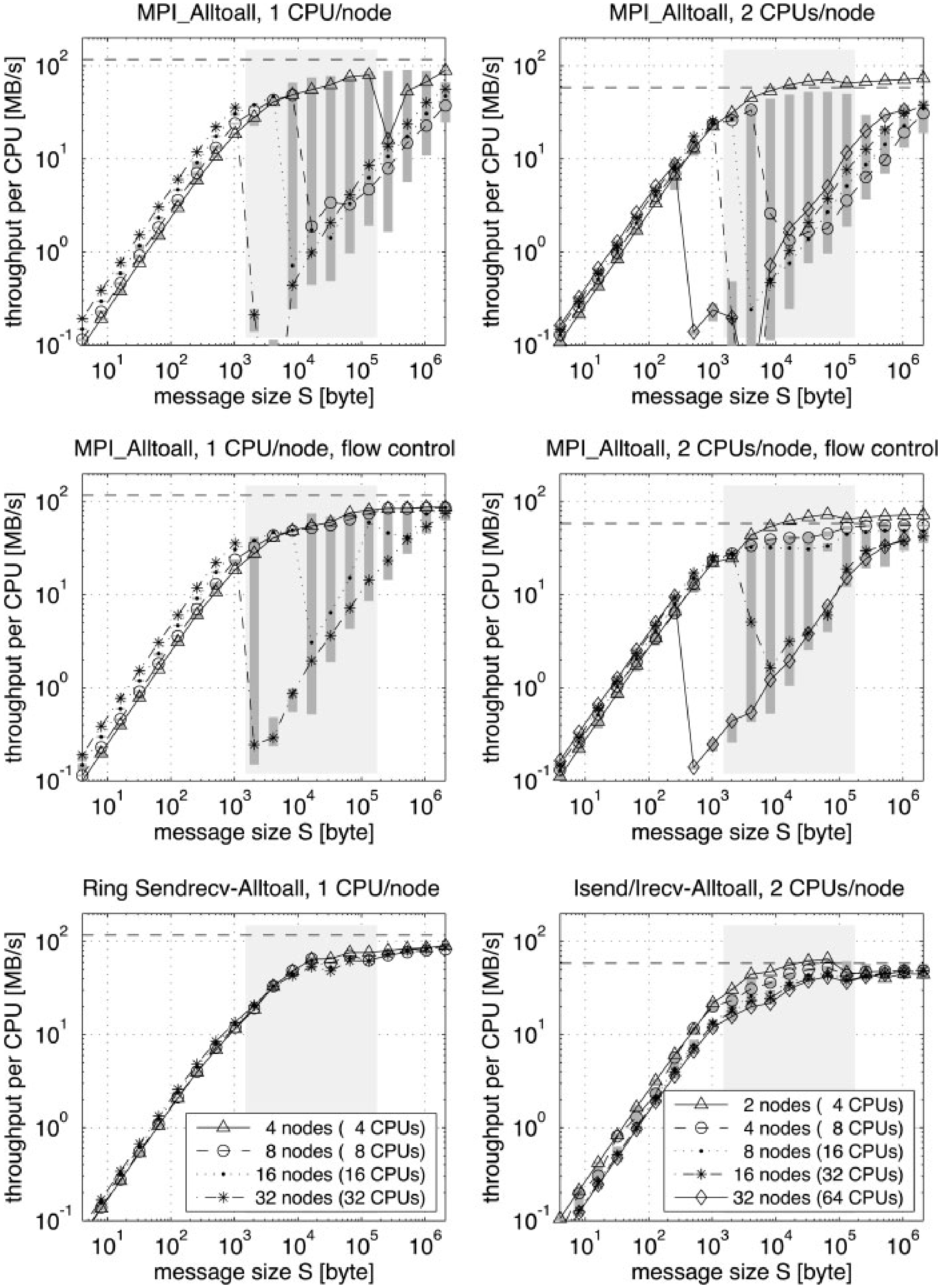

Figure 3. Throughput T for LAM all-to-all communication. Left 1 CPU/node, right 2 CPUs/node. Top: MPI_Alltoall without flow control (f.c.), middle: MPI_Alltoall with f.c., bottom: ordered all-to-all (same results with and without f.c.) Legends are given in lowermost plots. Dashed line indicates maximum throughput. Typical MD systems lie in the shaded area.

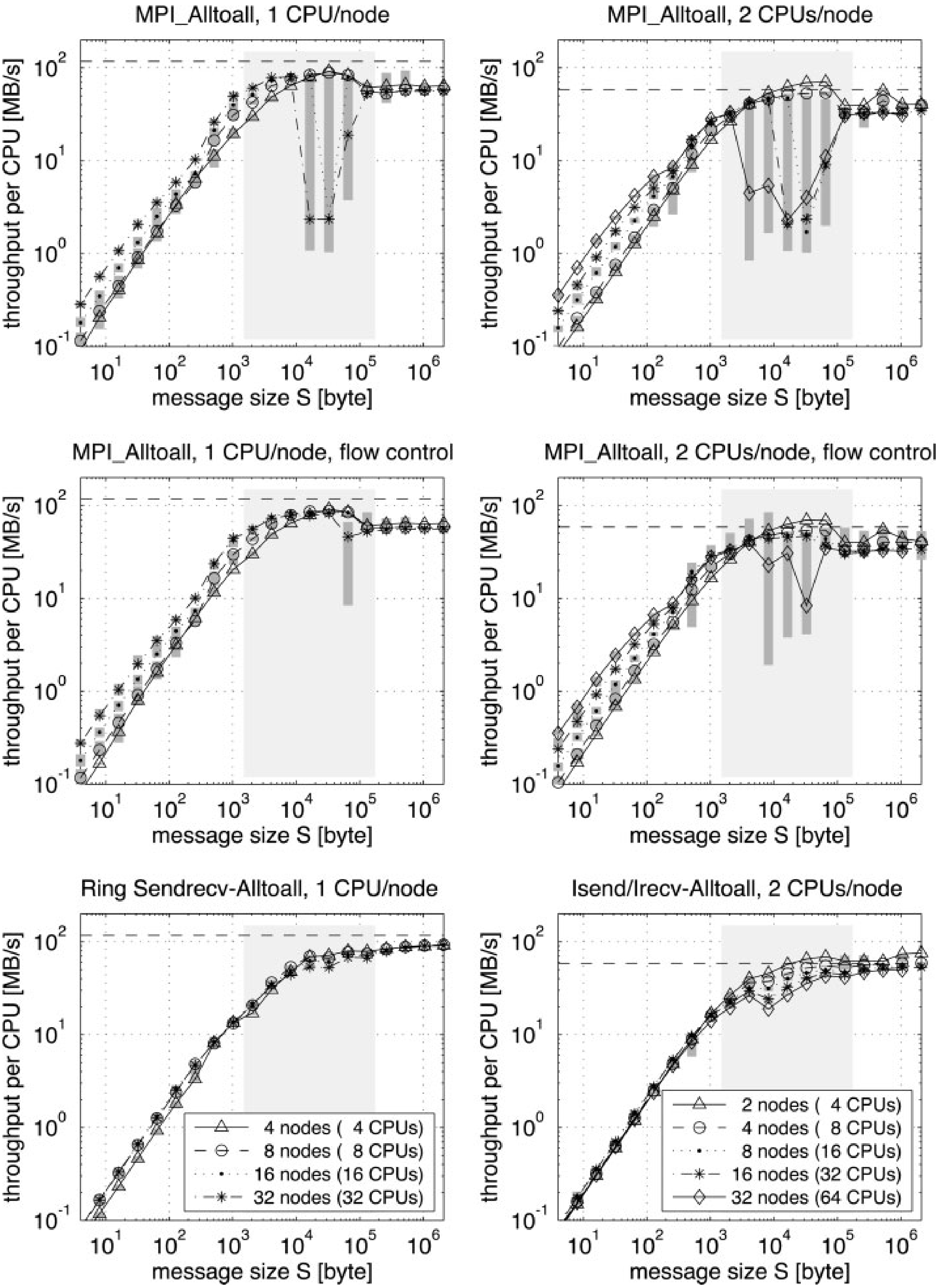

Figure 4. Throughput T for all-to-all communication. Same as Figure 3, but for MPICH-2.

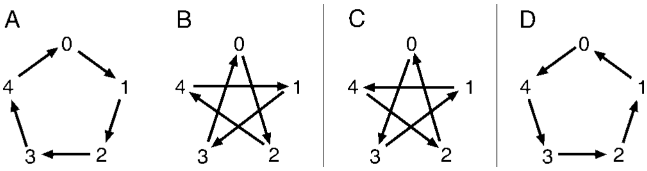

Figure 5. Example of an ordered communication scheme for five processes.

Clearly the MPI_Alltoall performance is far from optimal in that region.

The two middle panels of Figures 3 and 4 show the test results with activated flow control. Now, the all-to-all works without packet loss for up to 8 nodes in the case of LAM, and for up to 16 nodes in the case of MPICH. However, for more nodes the throughput remains suboptimal. For large node numbers, when a receiver sends out a PAUSE frame, packets might have to be dropped nonetheless, because too many of them are already on their way at that time.

Ordered All-to-All Communication

Since on the one hand not every switch supports flow control and on the other hand flow control does not guarantee good all-to-all performance for higher numbers of nodes, we tested two alternative all-to-all routines that are designed to avoid network congestion. This is achieved by ordering the communication in such a way that each (full-duplex) Ethernet link is used for exactly one message (in each direction) at a time. The idea to organize the communication in barrier-separated phases that are each congestion-free is described by Karwande et al.[25] who follow a more general approach.

Most known all-to-all algorithms are compiled in Faraj and Yuan[26] (see also the references therein), who give detailed performance results for LAM and MPICH on Fast Ethernet clusters (100 Mbit/s) of different topologies. Kale et al.[27] optimized the all-to-all routine for NAMD,[28] which like GROMACS uses this communication pattern for the FFT transpose within PME. Optimized collective communication routines, including scheduled all-to-alls, are also implemented in MPICH-1.2.6 as well as in MPICH-2[22] and are provided by OpenMPI from version 1.1 on.

The ordered communication scheme works as follows: For P processes communication gets scheduled in $P-1$ barrier-separated phases, as illustrated in Figure 5. In phase $i = A \ldots D$ each CPU sends clockwise to (and receives counterclockwise from) its ith neighbouring CPU, the others respectively. This is easily implemented with MPI_Sendrecv calls, see Figure 6 for a C code example.

Separating the phases with a barrier ensures that for large messages the available bandwidth can be exploited without risking congestion. For small messages $S \leq S_{\text{barr}}$ the barrier is not needed if both with and without flow control packet loss does not occur. For our setup we empirically determined $S_{\text{barr}} = 16$ kB for single-CPU nodes and $S_{\text{barr}} = 4$ kB for dual-CPU nodes. Omitting the barrier for small messages results in an all-to-all execution time of $\Delta t \approx 1.4$ ms compared to $\Delta t \approx 9.2$ ms with barrier on 32 single-CPU nodes. This boosts the throughput of the ordered routine by a factor of ≈ 6 for $S < 512$ bytes compared to the case with barrier.

The described algorithm works as long as each processor has an own network adapter. If two or more CPUs share a node's network connection, congestion still occurs. In the example case of two CPUs per node, processor $i = 0, 2, 4, \ldots$ shares a node with processor $i+1$. In phase A the sending is done in a ring resulting in one arriving, one intra-node and one leaving message on each node which works fine on full-duplex Ethernet. In phase B however, CPU 0 on node zero sends to CPU 2 (which is on node one) while at the same time CPU 1 on node zero sends to CPU 3, which is also on node one. That yields two arriving and two leaving messages at each node in a phase, and congestion is likely to occur. It turns out, however, that congestion is less severe with this scheme than without any order.

For multi-CPU nodes we have implemented the following method: In an outer loop, the ordered communication pattern is established on the basis of the nodes. In this context Figure 5 holds for 5 nodes, each containing e.g. two CPUs. The communication pattern defines for a node to which receiver node it has to send and from which sender node it has to receive. Each CPU of a given node sends to every CPU on the receiver node, see Figure 7. When this is done the next communication phase is entered.

Within each phase the switch simply forwards the data from each sending node to each receiving node. It does not matter in which order the individual messages of the CPUs get transferred. Therefore, all communication within a phase can be initiated with MPI_Isend/MPI_Irecv and then finalized with MPI_Waitall. Figure 8 shows a C code segment.

The two lower panels of Figures 3 and 4 show the performance of the ordered all-to-all routines. The results do not depend on whether

```c
for (i=0; i<ncpu; i++) /* loop over all CPUs */
{
   /* send to destination CPU while receiving from source CPU: */
   dest = (cpuid+i) % ncpu;
   source = (ncpu+cpuid-i) % ncpu;
   MPI_Sendrecv(sbuf+dest*schunk, scount, stype, dest, 0,
                rbuf+source*rchunk, rcount, rtype, source, 0,
                comm, &status);
   /* separate the communication phases for large chunks : */
   if ((schunk > sbarr) && (i < ncpu-1))
      MPI_Barrier(comm);
}
```

Figure 6. C code fragment of the ordered all-to-all for single CPU nodes using MPI_Sendrecv. ncpu: total number of CPUs, cpuid: CPU number.

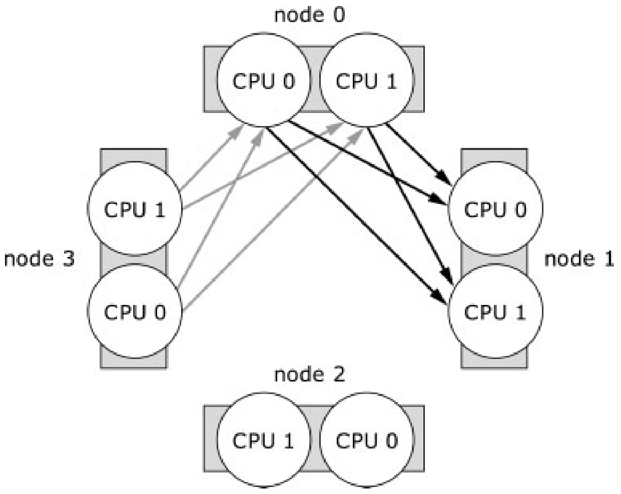

Figure 7. Ordered communication scheme for the case of four dual-CPU nodes. Shown is just the traffic to and from node 0 in a single phase (corresponding to phase A of Fig. 5).

flow control is active or not. For the single-CPU nodes the Sendrecv scheme of Figures 5 and 6 is shown, for the dual-CPU nodes the multi-CPU scheme of Figures 7–8. The performance of the Sendrecv scheme and of the Isend/Irecv scheme for single-CPU nodes is about the same.

The most important observation is that packet loss does not occur for any message size on any number of CPUs. Compared to the LAM case without flow control, the throughput of the orderd routines is significantly higher in the message size range of interest (grey area) when employing more than four CPUs. When using 16+ nodes with LAM, the performance is greatly improved by the ordered routines within the grey area.

Though the MPICH all-to-all generally performs fine with flow control (except on 32 nodes) there is a lot to be gained by switching to the ordered routine for the case without flow control (grey areas).

The MPICH all-to-all[22] chooses from four different algorithms depending on message size and number of CPUs (for the source code see *./src/mpi/coll/alltoall.c* in the basic directory of the MPICH-2 distribution). For short messages ($S \leq 256$ byte) and a minimum of eight CPUs, the Bruck index algorithm is used.[29] The higher throughput for messages $S \leq 256$ byte that is easily seen for the dual-CPU nodes can be attributed to this algorithm. The classic MPICH algorithm is used for 256 byte $< S \leq 32$ kB (as well as for less than eight CPUs when $S \leq 256$ byte). In the classic algorithm – very similar to LAM – all messages first are initiated with Isend/Irecv and then waited upon with MPI_Waitall. For large messages and a power-of-two number of CPUs, a pairwise exchange algorithm is used, for non-power-of-two numbers a Sendrecv algorithm similar to the one described in Figures 5 and 6 is used.

We also tested the pairwise exchange algorithm (with barrier) but found the performance to be similar to the described multi-CPU algorithm. The main difference between our all-to-all and the MPICH algorithm for large messages is the existence of a barrier between the communication phases, which acts as a synchronization between the CPUs in our case which helps when flow control is not available.

For small all-to-alls ($S < 10^3$ bytes) for both LAM and MPICH the throughput of the MPI_Alltoall is higher by a factor of about two, the exact value depending on the number of CPUs. For MPICH, the ordered all-to-all has a higher asymptotic bandwidth, e.g. for

```
for (i=0; i<nnodes; i++) /* loop over all nodes */
{
    /* send to destination node while receiving from source node: */
    destnode = (nodeid+i) % nnodes;
    sourcenode = (nnodes+nodeid-i) % nnodes;
    /* loop over CPUs on a node: */
    for (j=0; j < cpus_per_node; j++)
    {
       /* source and destination CPU: */
       destcpu = destnode*cpus_per_node + j;
       sourcecpu = sourcenode*cpus_per_node + j;
       MPI_Irecv(rbuf+sourcecpu*rchunk, rcount, rtype,
                 sourcecpu, 0, comm,
                 &requests[j+cpus_per_node]);
       MPI_Isend(sbuf+destcpu*schunk, scount, stype,
                 destcpu, 0, comm,
                 &requests[j]);
    }
    /* wait for communication to finish: */
    MPI_Waitall(2*procs_pn, requests, statuses);
    /* separate the communication phases for large chunks: */
    if ((sendcount > sbarr) && (i<nnodes-1))
       MPI_Barrier(comm);
}
```

Figure 8. C code fragment of the ordered all-to-all routine for multi-CPU nodes using MPI_Isend and MPI_Irecv. nnodes: total number of nodes, nodeid: node number.

single-CPU nodes $T_{\text{asym}} \approx 60$ MB/(s·CPU) for the MPI_Alltoall and $T_{\text{asym}} \approx 90$ MB/(s·CPU) for the ordered one. The main benefit of the ordered routines is, however, that they reliably avoid packet loss.

GROMACS Parallel Speedups with Ordered All-to-All

We now substituted the MPI_Alltoall calls in the FFT by the ordered routine from Figure 8 and benchmarked the modified MD code. The measured AQP1 speedups are shown by the solid lines in Figure 1 (see Table 1 for DPPC).

The speedup Sp_{ordered} of the modified code (without flow control) nearly reaches the speedup of Sp_{fc} the original code when flow control is active. For up to 16 CPUs $Sp_{\text{ordered}}/Sp_{\text{fc}} > 0.95$.

Thus, by incorporating the ordered all-to-all into the program, a fallback mechanism can be provided for network setups where link-layer flow control is not available or for larger systems where more than 16 CPUs are to be used in parallel. A patch that makes the necessary changes to GROMACS 3.3.1 can be found on the GROMACS web page.

Influence of the Switch

During the tests with 32 nodes we experienced the switch itself to pose a bottleneck. The HP 2848 is constructed out of four 12-port BroadCom BCM5690 subswitches that are connected to a BCM5670 switch fabric. The links between the fabric and subswitches have a capacity of 10 Gbit/s (full duplex). That implies that each subgroup of 12 ports that is connected to the fabric can at most transfer 10 Gbit/s to the remaining ports.

With the ordered all-to-all we experimentally determined that a maximum of nine ports per subswitch can be used without losing packets in the switch. For high-performance applications this reduces the 48 available to effectively 36 ports that can be used simultaneously. Figure 9 illustrates the performance difference of the MPI_Alltoall when using 32 subsequent ports (1–32 in our case) compared to a scattered scheme where a maximum of 9 out of each subgroup of 12 ports were used. With subsequent ports the all-to-all heavily suffers from packet loss when individual messages S are larger than 1 kB. On scattered ports there is only packet loss for S around 32 kB, while the ordered scheme performs optimal.

The results do not depend on whether the port assignment is accomplished in software (list of nodes in host- or machinefile) or hardware (wiring at the switch). All benchmarks described above were obtained with the scattered scheme ensuring optimum switch performance.

To demonstrate the influence of the port assignment scheme for a different scientific software application than GROMACS we performed an additional benchmark using the Car-Parinello (CP) molecular dynamics approach.[30] Car-Parrinello molecular dynamics is an efficient tool for classical MD where the interaction potential is given by the Kohn-Sham density functional method. In the CP scheme the forces on the classical nuclei and on the electronic expansion coefficients of the Kohn-Sham orbitals are computed from first principle at each time step. The well-established CPMD simulation package[31,32] employs a plane-wave basis, which results in a significant complexity reduction of the underlying CP equations. Furthermore, the plane-wave basis allows to use the numerically efficient FFT method. Because of the need to transform several times between real and reciprocal space during one CPMD integration step many parallel transposes are called, which leads to an extensive use of the all-to-all communication. For details on the overall parallel CPMD scheme the reader is kindly referred to Marx and Hutter[33] and Hutter and Curioni.[34]

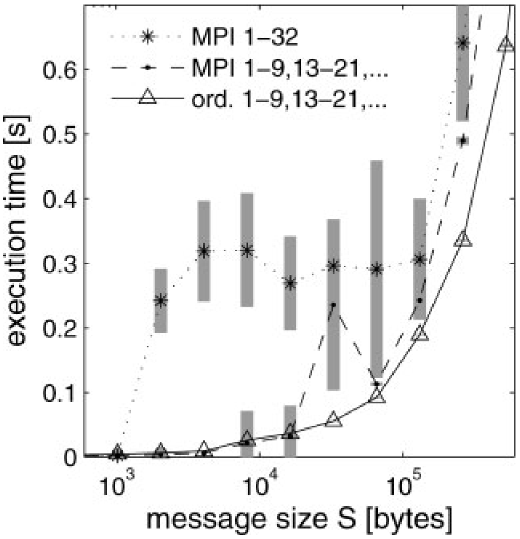

Figure 9. MPICH-2 all-to-all performance for different assignment schemes of 32 dual-CPU nodes onto 32 switch ports. ∗/Dots: MPI_Alltoll on ports 1–32; ·/dashes: same on scattered ports (1–9, 13–21, 25–33, 37–41). △/Solid: ordered all-to-all on scattered ports.

To elucidate the scaling properties of the CPMD code on the Linux cluster already outlined in detail above, we run a simulation of 64 water molecules inside a cubic box with length 12.42 Å subject to periodic boundary conditions. The BLYP density functional[35] in connection with norm-conserving Troullier-Martins pseudopotentials[36] have been used. The plane-wave kinetic energy cutoff was set to 120 Ry. The CP equations of motion were solved numerically with a time step of 4 a.u. and a fictitious electron mass of 350 a.u. was employed. The benchmark was performed using the CPMD 3.9.1 version compiled with the Intel 8.1 Fortran compiler in 32-bit, and a compatible LAM 7.1.1 MPI implementation was used throughout.

Using the subsequent ports 1–32 on the switch, the speedups gained relative to the eight CPU case ($Sp_8 = 1.0$) are $Sp_{16} = 1.9$, $Sp_{32} = 3.2$, and $Sp_{64} = 4.5$. These values have been gained with flow control using two CPUs per node and the standard MPI_Alltoall. Here also, without flow control, no decent scaling is possible at all. When changing to the scattered port allocation scheme, a speedup of $Sp_{64,\text{scat}} = 6.1$ is reached for 32 nodes, increasing the corresponding scaling from $Sc_{64} = 0.56$ to $Sc_{64,\text{scat}} = 0.76$. With scattered ports the CPMD benchmark does not suffer from TCP packet loss within the switch any more, which was the case when using subsequent ports. Here, with the scattered port scheme, there is no need to replace the standard MPI_Alltoall by the ordered version, because the standard all-to-all does not suffer from packet loss anyhow. This is because the CPMD benchmark resulted in small enough all-to-all message sizes where congestion does not occur.

Conclusions

On Ethernet-connected Beowulf clusters, TCP packet loss in the MPI_Alltoall routine can severely degrade the parallel scaling of GROMACS, especially when used on top of LAM/MPI. A typical indication for this problem is that the scaling abruptly breaks down when employing more than two nodes, which means involving more than one full-duplex Ethernet connection.

The most straightforward step to overcome this bottleneck is to activate IEEE 802.3x flow control on the network interfaces of the nodes and on the corresponding ports on the switch. By default, flow control was disabled in all of the switches that we tested. With flow control, on 16 CPUs a speedup of $Sp_{16} \approx 8$ is reached for the 80 k atom MD sytem, and $Sp_{16} \approx 11$ for the 120 k atom system. Larger systems have the potential to scale similarly well on even more processors.

For switches that do not support flow control, the incorporation of an ordered all-to-all routine into GROMACS serves as a fallback mechanism, preventing packet loss in the most critical part of the code and thus allowing similar speedups as with flow control. Note that if flow control is available, it should in any case be enabled, since with a growing number of CPUs network congestion can become an issue in parts of the code that perform nicely on a small number of CPUs.

Because MPICH-2 also uses an optimized all-to-all routine, packet loss is much more unlikely compared to the LAM pendant. This results in better parallel speedups, especially when using more than eight CPUs.

Our benchmarks show that flow control cannot prevent packet loss in the MPI_Alltoall when the number of nodes exceeds 16, making an ordered communication scheme inevitable. Ordered communication schemes as e.g. offered by MPICH-2 and OpenMPI rely upon the switch not to drop packets when forwarding all datastreams simultaneously. Packet loss within the common ProCurve 2848 switch can be avoided by using a maximum of 9 out of 12 consecutive ports. We demonstrated this by synthetic tests and also on the example of the CPMD application.

These findings are very likely applicable to other parallel programs that suffer from similar performance problems with MPI communication via the Ethernet.

Acknowledgments

The authors thank Jürgen Haas and Gerrit Groenhof for useful discussions and assistance. Renate Dohmen performed the benchmarks on the Regatta and on the Infiniband-cluster. Thank you for your kind and helpful e-mail support: Graham E. Fagg, Jeffrey Squires, George Bosilca, Peter Kjellström, Bogdan Costescu, Brian Barrett, Anthony Chan, Manfred Arndt, Xin Yuan, Ahmed Faraj, and Giuseppe Ciaccio. The present work was supported by the Division of Chemical Sciences, Office of Basic Energy Sciences, U.S. Department of Energy under Contract No. W-7405-Eng-82 with Iowa State University through the Ames Laboratory.

References

1. http://www.gromacs.org/.
2. http://www.ks.uiuc.edu/Research/namd/.
3. http://www.charmm.org/.
4. http://amber.scripps.edu/.
5. Lindahl, E.; Hess, B.; van der Spoel, D. J Mol Model 2001, 7, 306.
6. van der Spoel, D.; Lindahl, E.; Hess, B.; Groenhof, G.; Mark, A. E.; Berendsen, H. J. C. J Comput Chem 2005, 26, 1701.
7. The MPI Forum, http://www.mpi-forum.org/, 1997.
8. Bekker, H.; Dijkstra, E. J.; Berendsen, H. J. C. In Parallel Computing: From Theory to Sound Practice; Wu, P. T. and Milgrom, E., Eds.; IOS Press: Amsterdam, 1992; pp. 268–279.
9. Raine, A. R. C.; Fincham, D.; Smith, W. Comp Phys Comm 1989, 55, 13.
10. Darden, T.; York, D.; Pedersen, L. J Chem Phys 1993, 98, 10089.
11. Essmann, U.; Perera, L.; Berkowitz, M. L.; Darden, T.; Lee, H.; Pedersen, L. G. J Chem Phys 1995, 103, 8577.
12. Burns, G.; Daoud, R.; Vaigl, J. LAM: An Open Cluster Environment for MPI. In Proceedings of Supercomputing Symposium, University of Toronto, Toronto, Canada, 1994; pp. 379–386.
13. Squyres, J. M.; Lumsdaine, A. A Component Architecture for LAM/MPI. In Proceedings, 10th European PVM/MPI Users' Group Meeting; Springer-Verlag: Venice, Italy, 2003. Lecture Notes in Computer Science 2003, 2840, 379.
14. http://www.lam-mpi.org/.
15. Gropp, W.; Lusk, E.; Doss, N.; Skjellum, A. Parallel Comput 1996, 22, 789.
16. http://www-unix.mcs.anl.gov/mpi/mpich/.
17. IEEE. (IEEE 802.3, New York). pp. 1472–1481. 2000. Annex 31BL: MAC control PAUSE operation.
18. The Hewlett-Packard Development Company. Management and Configuration Guide for the ProCurve Switch 2600 Series, Switch 2600-PWR Series, Switch 2800 Series, Switch 4100gl Series, and Switch 6108, 2005. Chapter 10.
19. de Groot, B. L.; Grubmüller, H. Science 2001, 294, 2353.
20. Zaki, O.; Lusk, E.; Gropp, W.; Swider, D. Int J High Perform Comput Appl 1999, 13, 277.
21. http://www.open-mpi.org/.
22. Thakur, R.; Rabenseifner, R.; Gropp, W. Int J High Perform Comput Appl 2005, 19, 49.
23. Seifert, R. Gigabit Ethernet: Technology and Applications for High-Speed LANs, Addison-Wesley, 1998.
24. Kadambi, J.; Crayford, I.; Kalkunte, M. Gigabit Ethernet: Migrating to High-Bandwidth LANs, Prentice-Hall: New Jersey, 1998.
25. Karwande, A.; Yuan, X.; Lowenthal, D. K. J Parallel Distrib Comput 2005, 65, 1123.
26. Faraj, A.; Yuan, X. An empirical approach for efficient all-to-all personalized communication on Ethernet switched clusters. In ICPP, 2005; pp. 321–328.
27. Kale, L. V.; Kumar, S.; Varadarajan, K. A Framework for collective personalized communication. In International Parallel and Distributed Processing Symposium (IPDPS'03), 2003.
28. Phillips, J. C.; Braun, R.; Wang, W.; Gumbart, J.; Tajkhorshid, E.; Villa, E.; Chipot, C.; Skeel, R. D.; Kalé, L.; Schulten, K. J Comput Chem 2005, 26, 1781.
29. Bruck, J.; Ho, C.-T.; Kipnis, S.; Upfahl, E.; Weathersby, D. IEEE Trans Parallel Distrib Syst 1997, 8, 1143.
30. Car, R.; Parrinello, M. Phys Rev Lett 1985, 55, 2471.
31. Hutter, J.; Ballone, P.; Bernasconi, M.; Focher, P.; Fois, E.; Goedecker, S.; Marx, D.; Parrinello, M.; M. Tuckerman; MPI für Festkörperforschung, S.; Laboratory, I. Z. R., 1995.
32. http://www.cpmd.org/.
33. Marx, D. and Hutter, J. In Modern Methods and Algorithms of Quantum Chemistry; Grotendorst, J., Ed.; NIC: FZ Jülich, 2000; pp. 301–449.
34. Hutter, J.; Curioni, A. Chem Phys Chem 2005, 6, 1788.
35. Becke, A. D. Phys Rev A 1988, 38, 3098.
36. Troullier, N.; Martins, J. L. Phys Rev B 1991, 43, 1993.