Chapter 10

Efficient MPI programming

Substantial optimization potential is hidden in many MPI codes. After making sure that single-process performance is close to optimal by applying the methods described in Chapters 2 and 3, an MPI program should always be benchmarked for performance and scalability to unveil any problems connected to parallelization. Some of those are not related to message passing or MPI itself but emerge from well-known general issues such as serial execution (Amdahl's Law), load imbalance, unnecessary synchronization, and other effects that impact all parallel programming models. However, there are also very specific problems connected to MPI, and many of them are caused by implicit but unjustified assumptions about distributed-memory parallelization, or from over-optimistic notions regarding the cost and side effects of communication. One should always keep in mind that, while MPI was designed to provide portable and efficient message passing functionality, the performance of a given code is *not* portable across platforms.

This chapter tries to sketch the most relevant guidelines for efficient MPI programming, which are, to varying degrees, beneficial on all platforms and MPI implementations. Such an overview is necessarily incomplete, since every algorithm has its peculiarities. As in previous chapters on optimization, we will start by a brief introduction to typical profiling tools that are able to detect parallel performance issues in message-passing programs.

10.1 MPI performance tools

In contrast to serial programming, it is usually not possible to pinpoint the root causes of MPI performance problems by simple manual instrumentation. Several free and commercial tools exist for advanced MPI profiling [T24, T25, T26, T27, T28]. As a first step one usually tries to get a rough overview of how much time is spent in the MPI library in relation to application code, which functions dominate, and probably what communication volume is involved. This kind of data can at least show whether communication is a problem at all. IPM [T24] is a simple and low-overhead tool that is able to retrieve this information. Like most MPI profilers, IPM uses the MPI profiling interface, which is part of the standard [P15]. Each MPI function is a trivial wrapper around the actual function, whose name stars with "`PMPI_`." Hence, a preloaded library or even the user code can intercept MPI calls and collect profiling data. In case of IPM, it is sufficient to preload a dynamic library (or link with a static

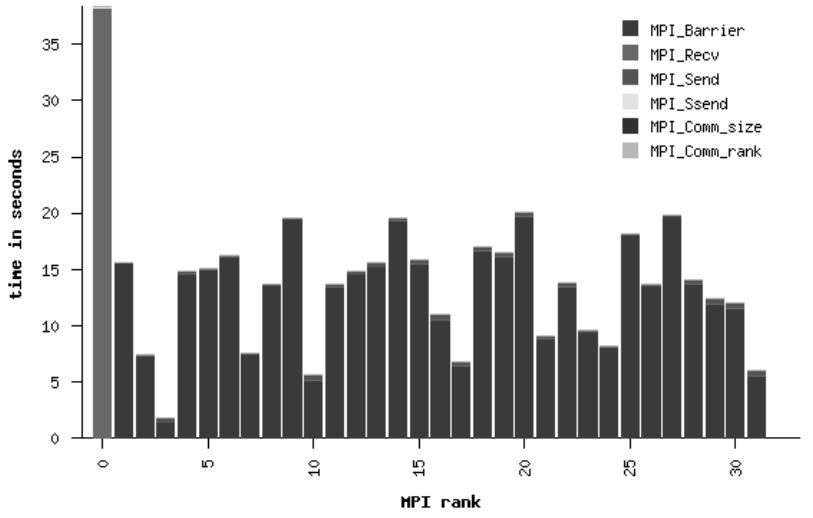


Figure 10.1: (See color insert after page 262.) IPM "communication balance" of a master-worker style parallel application. The complete runtime was about 38 seconds, which are spent almost entirely inside `MPI_Recv()` on rank 0. The other ranks are very load imbalanced, spending between 10 and 50% of their time in a barrier.

version) and run the application. Information about data volumes (per process and per process pair), time spent in MPI calls, load imbalance, etc., is then accumulated over the application's runtime, and can be viewed in graphical form. Figure 10.1 shows the "communication balance" graph of a master-worker application, as generated by IPM. Each bar corresponds to an MPI rank and shows how much time the process spends in the different MPI functions. It is important to compare those times to the overall runtime of the program, because a barrier time of twenty seconds means nothing if the program runs for hours. In this example, the runtime was 38 seconds. Rank 0 (the master) distributes work among the workers, so it spends most of its runtime in `MPI_Recv()`, waiting for results. The workers are obviously quite load imbalanced, and between 5 and 50% of their time is wasted waiting at barriers. A small change in parameters (reducing the size of the work packages) was able to correct this problem, and the resulting balance graph is shown in Figure 10.2. Overall runtime was reduced, quite expectedly, by about 25%.

Note that care must be taken when interpreting summary results that were taken over the complete runtime of an application. Essentially the same reservations apply as for global hardware performance counter information (see Section 2.1.2). IPM has a small API that can collect information broken down into user-definable phases, but sometimes more detailed data is required. A functionality that more advanced tools support is the *event timeline*. An MPI program can be decomposed into very specific events (message send/receive, collective operations, blocking wait,...), and those can easily be visualized in a timeline display. Figure 10.3 is a screenshot from "Intel Trace Analyzer" [T26], a GUI application that allows browsing and analysis of

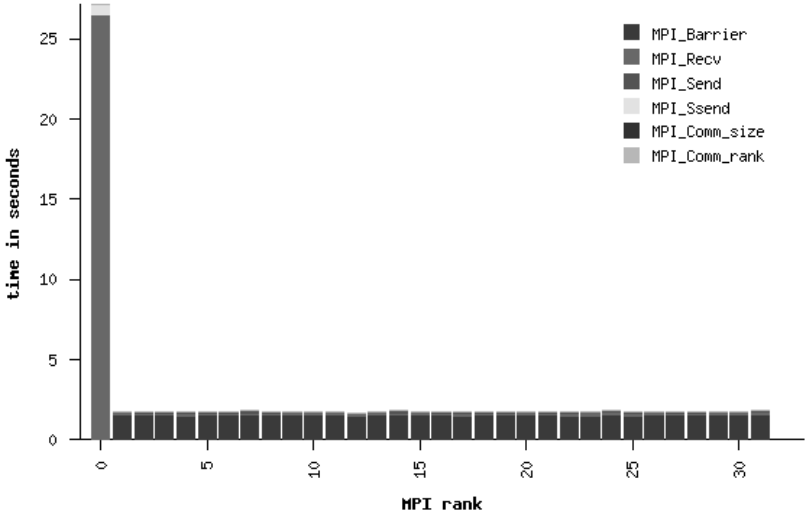


Figure 10.2: (See color insert after page 262.) IPM function profile of the same application as in Figure 10.1, with the load imbalance problem removed.

trace data written by an MPI program (the code must be linked to a special collector library before running). The top panel shows a zoomed view of a timeline from a code similar to the MPI-parallel Jacobi solver from Section 9.3.1. In this view, point-to-point messages are depicted by black lines, and bright lines denote collectives. Each process is broken down along the time axis into MPI (bright) and user code (dark) parts. The runtime is clearly dominated by MPI communication in this example. Pie charts in the lower left panel summarize, for each process, what fraction of time is spent with user code and MPI, respectively, making a possible load imbalance evident (the code shown is well load balanced). Finally, in the lower right panel, the data volume exchanged between pairs of processes can be read off for every possible combination. All this data can be displayed in more detail. For instance, all relevant parameters and properties of each message like its duration, data volume, source and target, etc., can be viewed separately. Graphs containing MPI contributions can be broken down to show the separate MPI functions, and user code can be instrumented so that different functions show up in the timeline and summary graphs.

Note that Intel Trace Analyzer is just one of many commercially and freely available MPI profiling tools. While different tools may focus on different aspects, they all serve the purpose of making the vast amount of data which is required to represent the performance properties of an MPI code easier to digest. Some tools put special emphasis on large-scale systems, where looking at timelines of individual processes is useless; they try to provide a high-level overview and generate some automatic tuning advice from the data. This is still a field of active, ongoing research [T29].

The effective use of MPI profiling tools requires a considerable amount of experience, and there is no way a beginner can draw any use out of them without some

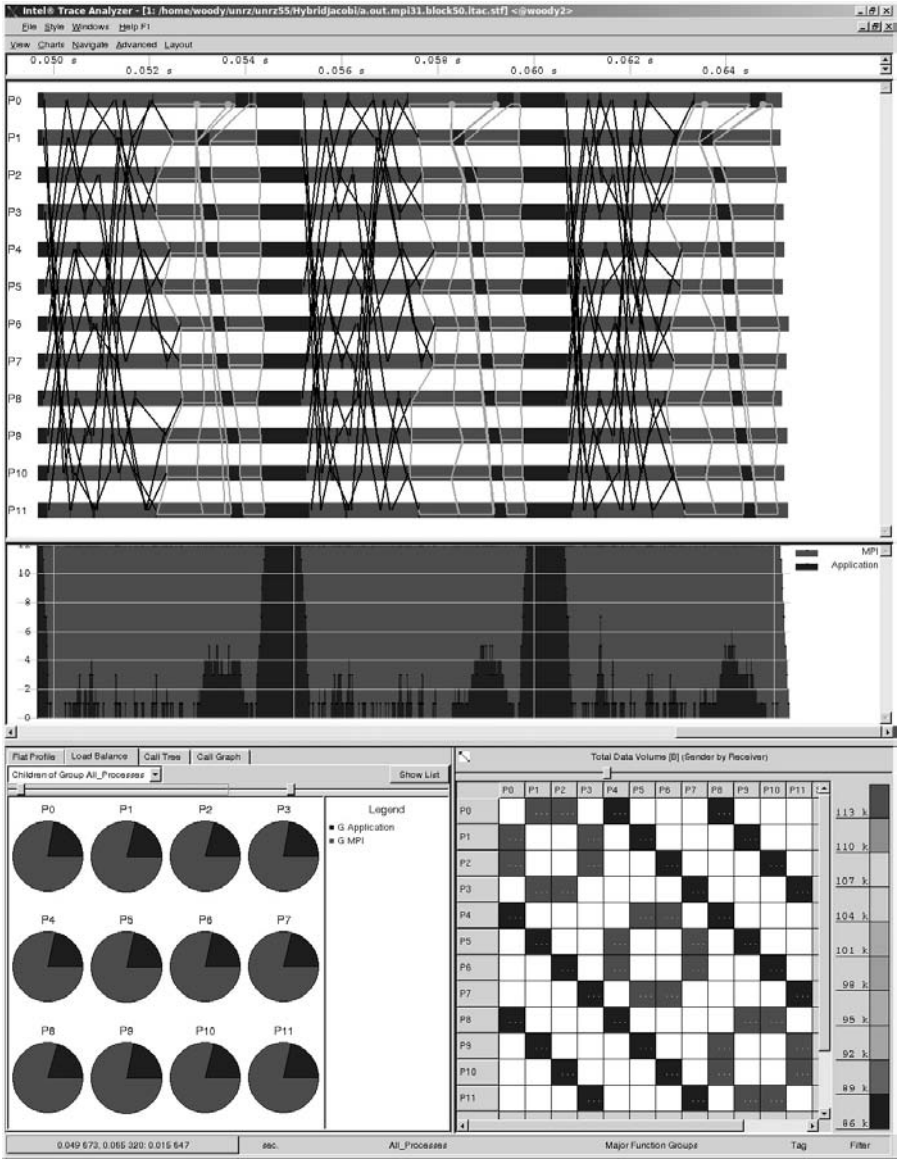


Figure 10.3: (See color insert after page 262.) Intel Trace Analyzer timeline view (top), load balance analysis (left bottom) and communication summary (right bottom) for an MPI-parallel code running on 12 nodes. Point-to-point (collective) messages are depicted by dark (bright) lines in the timeline view, while dark (light) boxes or pie slices denote executed application (MPI) code.

knowledge about the basic performance pitfalls of message-passing code. Hence, this is what the rest of this chapter will focus on.

10.2 Communication parameters

In Section 4.5.1 we have introduced some basic performance properties of networks, especially regarding point-to-point message transfer. Although the simple latency/bandwidth model (4.2) describes the gross features of the effective bandwidth reasonably well, a parametric fit to PingPong benchmark data cannot reproduce the correct (measured) latency value (see Figure 4.10). The reason for this failure is that an MPI message transfer is more complex than what our simplistic model can cover. Most MPI implementations switch between different variants, depending on the message size and other factors:

- For short messages, the message itself and any supplementary information (length, sender, tag, etc., also called the *message envelope*) may be sent and stored at the receiver side in some preallocated buffer space, without the receiver's intervention. A matching receive operation may not be required, but the message must be copied from the intermediate buffer to the receive buffer at one point. This is also called the *eager protocol*. The advantage of using it is that synchronization overhead is reduced. On the other hand, it could need a large amount of preallocated buffer space. Flooding a process with many eager messages may thus overflow those buffers and lead to contention or program crashes.

- For large messages, buffering the data makes no sense. In this case the envelope is immediately stored at the receiver, but the actual message transfer blocks until the user's receive buffer is available. Extra data copies could be avoided, improving effective bandwidth, but sender and receiver must synchronize. This is called the *rendezvous protocol*.

Depending on the application, it could be useful to adjust the message length at which the transition from eager to rendezvous protocol takes place, or increase the buffer space reserved for eager data (in most MPI implementations, these are tunable parameters).

The `MPI_Issend()` function could be used in cases where "eager overflow" is a problem. It works like `MPI_Isend()` with slightly different semantics: If the send buffer can be reused according to the request handle, a sender-receiver handshake has occurred and message transfer has started. See also Problem 10.3.

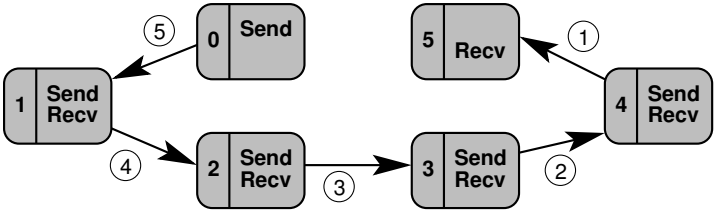


Figure 10.4: A linear shift communication pattern. Even with synchronous point-to-point communication, a deadlock will not occur, but all message transfers will be serialized in the order shown.

10.3 Synchronization, serialization, contention

This section elaborates on some performance problems that are not specific to message-passing, but may take special forms with MPI and hence deserve a detailed discussion.

10.3.1 Implicit serialization and synchronization

"Unintended" frequent synchronization or even serialization is a common phenomenon in parallel programming, and not limited to MPI. In Section 7.2.3 we have demonstrated how careless use of OpenMP synchronization constructs can effectively serialize a parallel code. Similar pitfalls exist with MPI, and they are often caused by false assumptions about how messages are transferred.

The ring shift communication pattern, which was used in Section 9.2.2 to illustrate the danger of creating a deadlock with blocking, synchronous point-to-point messages, is a good example. If the chain is open so that the ring becomes a lin-

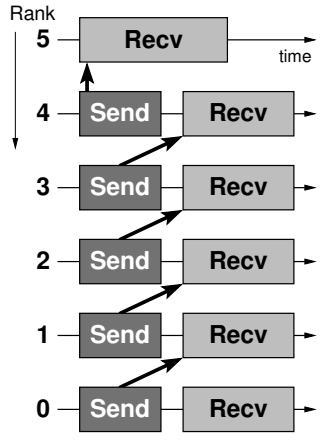


Figure 10.5: Timeline view of the linear shift (see Figure 10.4) with blocking (but not synchronous) sends and blocking receives, using eager delivery. Message transmissions can overlap, making use of a nonblocking network. Eager delivery allows a send to end before the corresponding receive is posted.

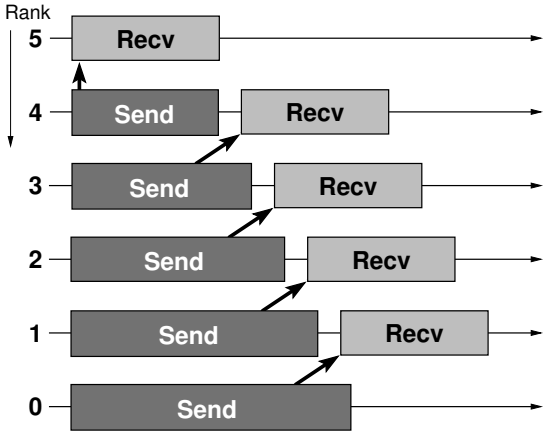


Figure 10.6: Timeline view of the linear shift (see Figure 10.4) with blocking synchronous sends and blocking receives, using eager delivery. The message transfers (arrows) might overlap perfectly, but a send can only finish just after its matching receive is posted.

ear shift pattern, but sends and receives are performed on the processes in the order shown in Figure 10.4, there will be no deadlock: Process 5 posts a receive, which matches the send on process 4. After that send has finished, process 4 can post its receive, etc. Assuming the parameters are such that `MPI_Send()` is not synchronous, and "eager delivery" (see Section 10.2) can be used, a typical timeline graph, similar to what MPI performance tools would display, is depicted in Figure 10.5. Message transfers can overlap if the network is nonblocking, and since all send operations terminate early (i.e., as soon as the blocking semantics is fulfilled), most of the time is spent receiving data (note that there is no indication of where exactly the data is — it could be anywhere on its way from sender to receiver, depending on the implementation).

There is, however, a severe performance problem with this pattern. If the message parameters, first and foremost its length, are such that `MPI_Send()` is actually executed as `MPI_Ssend()`, the particular semantics of synchronous send must be observed: `MPI_Ssend()` does not return to the user code before a matching receive is posted on the target. This does *not* mean that `MPI_Ssend()` blocks until the message has been fully transmitted and arrived in the receive buffer. Hence, a send and its matching receive may overlap just by a small amount, which provides at least some parallel use of the network but also incurs some performance penalty (see Figure 10.6 for a timeline graph). A necessary prerequisite for this to work is that message delivery still follows the eager protocol: If the conditions for eager delivery are fulfilled, the data has "left" the send buffer (in terms of blocking semantics) already before the receive operation was posted, so it is safe even for a synchronous send to terminate upon receiving some acknowledgment from the other side.

When the messages are transmitted according to the rendezvous protocol, the situation gets worse. Buffering is impossible here, so sender and receiver must synchronize in a way that ensures full end-to-end delivery of the data. In our example, the five messages will be transmitted in serial, one after the other, because no process can finish its send operation until the next process down the chain has finished its receive. The further down the chain a process is located, the longer its own syn-

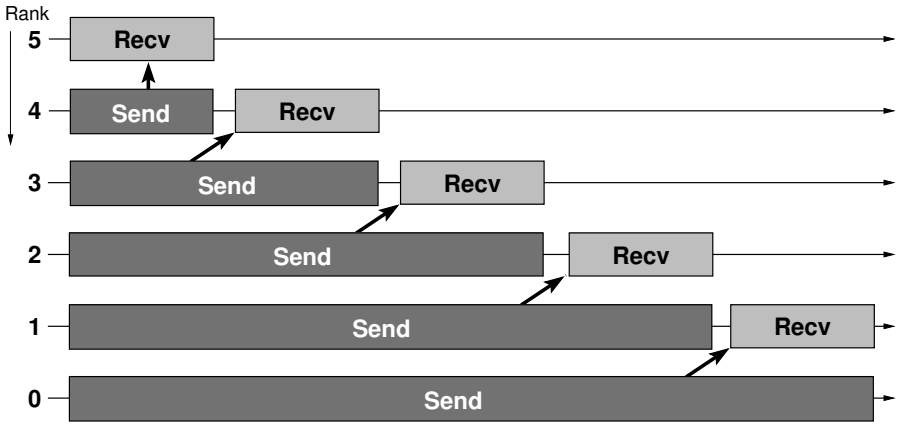


Figure 10.7: Timeline view of the linear shift (see Figure 10.4) with blocking sends and blocking receives, using the rendezvous protocol. The messages (arrows) are transmitted in serial because buffering is ruled out.

chronous send operation will block, and there is no potential for overlap. Figure 10.7 illustrates this with a timeline graph.

Implicit serialization should be avoided because it is not only a source of additional communication overhead but can also lead to load imbalance, as shown above. Therefore, it is important to think about how (ring or linear) shifts, of which ghost layer exchange is a variant, and similar patterns can be performed efficiently. The basic alternatives have already been described in Section 9.2.2:

- Change the order of sends and receives on, e.g., all odd-numbered processes (See Figure 10.8). Pairs of processes can then exchange messages in parallel, using at least part of the available network capacity.

- Use nonblocking functions as shown with the parallel Jacobi solver in Section 9.3. Nonblocking functions have the additional benefit that multiple outstanding communications can be handled by the MPI library in a (hopefully) optimal order. Moreover they provide at least an opportunity for truly asynchronous communication, where auxiliary threads and/or hardware mechanisms transfer data even while a process is executing user code. Note that this mode of operation must be regarded as an optimization provided by the MPI implementation; the MPI standard intentionally avoids any specifications about asynchronous transfers.

- Use blocking point-to-point functions that are guaranteed not to deadlock, regardless of message size, notably `MPI_Sendrecv()` (see also Problem 10.7) or `MPI_Sendrecv_replace()`. Internally, these calls are often implemented as combinations of nonblocking calls and `MPI_Wait()`, so they are actually convenience functions.

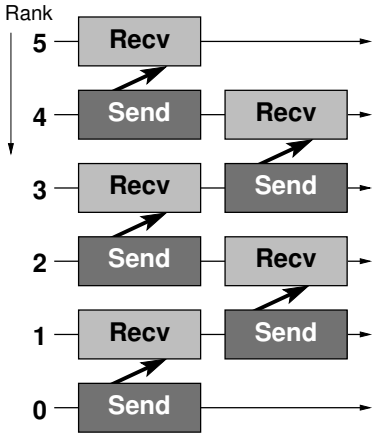


Figure 10.8: Even if sends are synchronous and the rendezvous protocol is used, exchanging the order of sends and receives on all odd-numbered ranks exposes some parallelism in communication.

10.3.2 Contention

The simple latency/bandwidth communication model that we have used so far, together with the refinements regarding message delivery (see Section 10.2) can explain a lot of effects, but it does not encompass contention effects. Here we want to restrict the discussion of contention to network connections; shared resources within a shared-memory multisocket multicore system like a compute node are ignored (see, e.g., Sections 1.4, 4.2, and 6.2 for more information on those issues). Assuming a typical hybrid (hierarchical) parallel computer design as discussed in Section 4.4, network contention occurs on two levels:

- Multiple threads or processes on a node may issue communication requests to other nodes. If bandwidth does not scale to multiple connections, the available bandwidth per connection will go down. This is very common with commodity systems, which often have only a single network interface available for MPI communication (and sometimes even I/O to remote file systems). On these machines, a single thread can usually saturate the network interface. However, there are also parallel computers where multiple connections are required to make full use of the available network bandwidth [O69].

- The network topology may not be fully nonblocking, i.e., the bisection bandwidth (see Section 4.5.1) may be lower than the product of the number of nodes and the single-connection bandwidth. This is common with, e.g., cubic mesh networks or fat trees that are not fully nonblocking.

- Even if bisection bandwidth is optimal, static routing can lead to contention for certain communication patterns (see Figure 4.17 in Section 4.5.3). In the latter case, changing the network fabric's routing tables (if possible) may be an option if performance should be optimized for a single application with a certain, well-defined communication scheme [O57].

In general, contention of *some* kind is hardly avoidable in current parallel systems if message passing is used in any but the most trivial ways. An actual impact on

application performance will of course only be visible if communication represents a measurable part of runtime.

Note that there are communication patterns that are especially prone to causing contention, like all-to-all message transmission where every process sends to every other process; MPI's `MPI_Alltoall()` function is a special form of this. It is to be expected that the communication performance for all-to-all patterns on massively parallel architectures will continue to decline in the years to come.

Any optimization that reduces communication overhead and message transfer volume (see Section 10.4) will most probably also reduce contention. Even if there is no way to lessen the amount of message data, it may be possible to rearrange communication calls so that contention is minimized [A85].

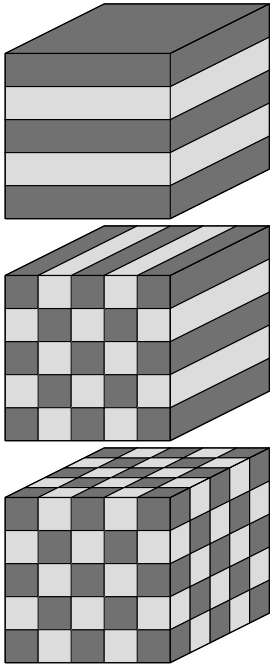
10.4 Reducing communication overhead

10.4.1 Optimal domain decomposition

Domain decomposition is one of the most important implementations of data parallelism. Most fluid dynamics and structural mechanics simulations are based on domain decomposition and ghost layer exchange. We have demonstrated in Section 9.3.2 that the performance properties of a halo communication can be modeled quite accurately in simple cases, and that the division of the whole problem into subdomains determines the communicated data volume, influencing performance in a crucial way. We are now going to shed some light on the question what it may cost (in terms of overhead) to choose a "wrong" decomposition, elaborating on the considerations leading to the performance models for the parallel Jacobi solver in Section 9.3.2.

Minimizing interdomain surface area

Figure 10.9 shows different possibilities for the decomposition of a cubic domain of size L^3 into N subdomains with strong scaling. Depending on whether the domain cuts are performed in one, two, or all three dimensions (top to bottom), the number of elements that must be communicated by one process with its neighbors, $c(L,N)$, changes its dependence on N. The best behavior, i.e., the steepest decline, is achieved with cubic subdomains (see also Problem 10.4). We are neglecting here that the possible options for decomposition depend on the prime factors of N and the actual shape of the overall domain (which may not be cubic). The `MPI_Dims_create()` function tries, by default, to make the subdomains "as cubic as possible," under the assumption that the complete domain is cubic. As a result, although much easier to implement, "slab" subdomains should not be used in general because they incur a much larger and, more importantly, N-independent overhead as compared to pole-shaped or cubic ones. A constant cost of communication per subdomain will greatly harm strong scalability because performance saturates at a lower level, determined by the message size (i.e., the slab area) instead of the latency (see Eq. 5.27).



"Slabs"

$$c_{1d}(L,N) = L \cdot L \cdot w \cdot 2 = 2wL^2$$

"Poles"

$$c_{2d}(L,N) = L \cdot \frac{L}{\sqrt{N}} \cdot w \cdot (2+2) = 4wL^2N^{-1/2}$$

"Cubes"

$$c_{3d}(L,N) = \frac{L}{\sqrt[3]{N}} \cdot \frac{L}{\sqrt[3]{N}} \cdot w \cdot (2+2+2) = 6wL^2N^{-2/3}$$

Figure 10.9: 3D domain decomposition of a cubic domain of size L^3 (strong scaling) and periodic boundary conditions: Per-process communication volume $c(L,N)$ for a single-site data volume w (in bytes) on N processes when cutting in one (top), two (middle), or all three (bottom) dimensions.

The negative power of N appearing in the halo volume expressions for pole- and cube-shaped subdomains will dampen the overhead, but still the surface-to-volume ratio will grow with N. Even worse, scaling up the number of processors at constant problem size "rides the PingPong curve" down towards smaller messages and, ultimately, into the latency-dominated regime (see Section 4.5.1). This has already been shown implicitly in our considerations on refined performance models (Section 5.3.6, especially Eq. 5.28) and "slow computing" (Section 5.3.8). Note that, in the absence of overlap effects, each of the six halo communications is subject to latency; if latency dominates the overhead, "optimal" 3D decomposition may even be counterproductive because of the larger number of neighbors for each domain.

The communication volume per site (w) depends on the problem. For the simple Jacobi algorithm from Section 9.3, $w = 16$ (8 bytes each in positive and negative coordinate direction, using double precision floating-point numbers). If an algorithm requires higher-order derivatives or if there is some long-range interaction, w is larger. The same is true if one grid point is a more complicated data structure than just a scalar, as is the case with, e.g., lattice-Boltzmann algorithms [A86, A87]. See also the following sections.

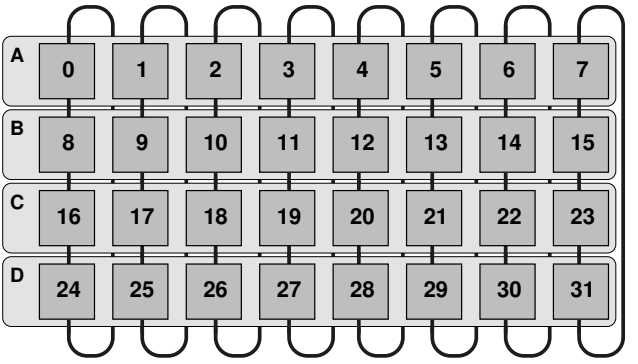


Figure 10.10: A typical default mapping of MPI ranks (numbers) to subdomains (squares) and cluster nodes (letters) for a two-dimensional 4×8 periodic domain decomposition. Each node has 16 connections to other nodes. Intranode connections are omitted.

Mapping issues

Modern parallel computers are inevitably of the hierarchical type. They all consist of shared-memory multiprocessor "nodes" coupled via some network (see Section 4.4). The simplest way to use this kind of hardware is to run one MPI process per core. Assuming that any point-to-point MPI communication between two cores located on the same node is much faster (in terms of bandwidth and latency) than between cores on different nodes, it is clear that the mapping of computational subdomains to cores has a large impact on communication overhead. Ideally, this mapping should be optimized by `MPI_Cart_create()` if rank reordering is allowed, but most MPI implementations have no idea about the parallel machine's topology.

As simple example serves to illustrate this point. The physical problem is a two-dimensional simulation on a 4×8 Cartesian process grid with periodic boundary conditions. Figure 10.10 depicts a typical "default" configuration on four nodes (A ... D) with eight cores each (we are neglecting network topology and any possible node substructure like cache groups, ccNUMA locality domains, etc.). Under the assumption that intranode connections come at low cost, the efficiency of next-neighbor communication (e.g., ghost layer exchange) is determined by the maximum number of internode connection per node. The mapping in Figure 10.10 leads to 16 such connections. The "communicating surface" of each node is larger than it needs to be because the eight subdomains it is assigned to are lined up along one dimension. Choosing a less oblong arrangement as shown in Figure 10.11 will immediately reduce the number of internode connections, to twelve in this case, and consequently cut down network contention. Since the data volume per connection is still the same, this is equivalent to a 25% reduction in internode communication volume. In fact, no mapping can be found that leads to an even smaller overhead for this problem.

Up to now we have presupposed that the MPI subsystem assigns successive ranks to the same node when possible, i.e., filling one node before using the next. Although this may be a reasonable assumption on many parallel computers, it should by no means be taken for granted. In case of a *round-robin*, or *cyclic* distribution, where successive ranks are mapped to successive nodes, the "best" solution from Figure 10.11 will be turned into the worst possible alternative: Figure 10.12 illustrates that each node now has 32 internode connections.

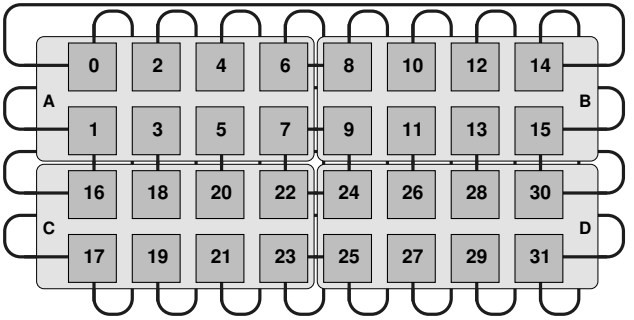


Figure 10.11: A "perfect" mapping of MPI ranks and subdomains to nodes. Each node has 12 connections to other nodes.

Similar considerations apply to other types of parallel systems, like architectures based on (hyper-)cubic mesh networks (see Section 4.5.4), on which next-neighbor communication is often favored. If the Cartesian topology does not match the mapping of MPI ranks to nodes, the resulting long-distance connections can result in painfully slow communication, as measured by the capabilities of the network. The actual influence on application performance may vary, of course. Any type of mapping might be acceptable if the parallel program is not limited by communication at all. However, it is good to keep in mind that the default provided by the MPI environment should not be trusted. MPI performance tools, as described in Section 10.1, can be used to display the effective bandwidth of every single point-to-point connection, and thus identify possible mapping issues if the numbers do not match expectations.

So far we have neglected any intranode issues, assuming that MPI communication between the cores of a node is "infinitely fast." While it is true that intranode latency is far smaller than what any existing network technology can provide, bandwidth is an entirely different matter, and different MPI implementations vary widely in their intranode performance. See Section 10.5 for more information. In truly hybrid programs, where each MPI process consists of several OpenMP threads, the mapping problem becomes even more complex. See Section 11.3 for a detailed discussion.

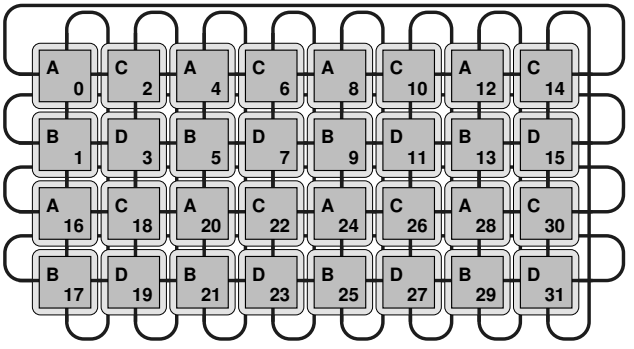


Figure 10.12: The same rank-to-subdomain mapping as in Figure 10.11, but with ranks assigned to nodes in a "round-robin" way, i.e., successive ranks run on different nodes. This leads to 32 internode connections per node.

10.4.2 Aggregating messages

If a parallel algorithm requires transmission of a lot of small messages between processes, communication becomes latency-bound because each message incurs latency. Hence, small messages should be *aggregated* into contiguous buffers and sent in larger chunks so that the latency penalty must only be paid once, and effective communication bandwidth is as close as possible to the saturation region of the PingPong graph (see Figure 4.10 in Section 4.5.1). Of course, this advantage pertains to point-to-point and collective communication alike.

Aggregation will only pay off if the additional time for copying the messages to a contiguous buffer does not outweigh the latency penalty for separate sends, i.e., if

$$(m-1)T_\ell > \frac{mL}{B_\mathrm{c}}\,, \tag{10.1}$$

where m is the number of messages, L is the message length, and B_c is the bandwidth for memory-to-memory copies. For simplicity we assume that all messages have the same length, and that latency for memory copies is negligible. The actual advantage depends on the raw network bandwidth B_n as well, because the ratio of serialized and aggregated communication times is

$$\frac{T_\mathrm{s}}{T_\mathrm{a}} = \frac{T_\ell/L + B_\mathrm{n}^{-1}}{T_\ell/mL + B_\mathrm{c}^{-1} + B_\mathrm{n}^{-1}}\,. \tag{10.2}$$

On a slow network, i.e., if B_n^{-1} is large compared to the other expressions in the numerator and denominator, this ratio will be close to one and aggregation will not be beneficial.

A typical application of message aggregation is the use of *multilayer halos* with stencil solvers: After multiple updates (sweeps) have been performed on a subdomain, exchange of multiple halo layers in a single message can exploit the "PingPong ride" to reduce the impact of latency. If this approach appears feasible for optimizing an existing code, appropriate performance models should be employed to estimate the expected gain [O53, A88].

Message aggregation and derived datatypes

A typical case for message aggregation comes up when separate, i.e., noncontiguous data items must be transferred between processes, like a row of a (Fortran) matrix or a completely unconnected bunch of variables, possibly of different types. MPI provides so-called *derived datatypes*, which support this functionality. The programmer can introduce new datatypes beyond the built-in ones (`MPI_INTEGER` etc.) and use them in communication calls. There is a variety of choices for defining new types: Array-like with gaps, indexed arrays, n-dimensional subarrays of n-dimensional arrays, and even a collection of unconnected variables of different types scattered in memory. The new type must first be defined using `MPI_Type_XXXXX()`, where "`XXXXX`" designates one of the variants as described above. The call returns the new type as an integer (in Fortran) or in an `MPI_Datatype` structure (in C/C++). In

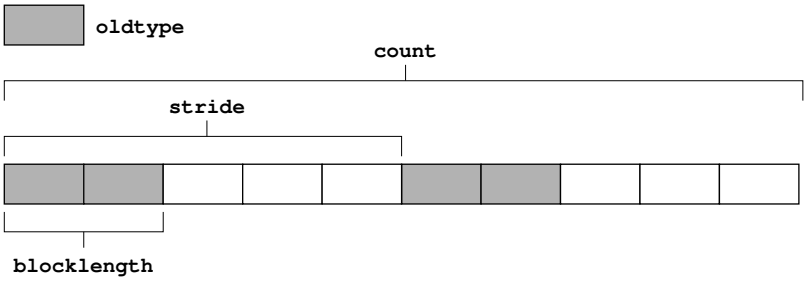


Figure 10.13: Required parameters for `MPI_Type_vector`. Here `blocklength`=2, `stride`=5, and `count`=2.

order to use the type, it must be "committed" with `MPI_Type_commit()`. In case the type is not needed any more, it can be "freed" using `MPI_Type_free()`.

We demonstrate these facilities by defining the new type to represent one row of a Fortran matrix. Since Fortran implements column major ordering with multidimensional arrays (see Section 3.2), a matrix row is noncontiguous in memory, with chunks of one item separated by a constant stride. The appropriate MPI call to use for this is `MPI_Type_vector()`, whose main parameters are depicted in Figure 10.13. We use it to define a row of a double precision matrix of dimensions `XMAX`×`YMAX`. Hence, `count`=`YMAX`, `blocklength`=1, `stride`=`XMAX`, and the old type is `MPI_DOUBLE_PRECISION`:

```
1  double precision, dimension(XMAX,YMAX) :: matrix
2  integer newtype                                  ! new type
3
4  call MPI_Type_vector(YMAX,                       ! count
5                       1,                          ! blocklength
6                       XMAX,                       ! stride
7                       MPI_DOUBLE_PRECISION,       ! oldtype
8                       newtype,                    ! new type
9                       ierr)
10 call MPI_Type_commit(newtype, ierr)              ! make usable
11 ...
12 call MPI_Send(matrix(5,1),                       ! send 5th row
13               1,                                 ! sendcount=1
14               newtype,...)                       ! use like any type
15 ...
16 call MPI_Type_free(newtype,ierr)                 ! release type
```

In line 12 the type is used to send the 5th row of the matrix, with a `count` argument of 1 to the `MPI_Send()` function (care must be taken when sending more than one instance of such a type, because "gaps" at the end of a single instance are ignored by default; consult the MPI standard for details). Datatypes for simplifying halo exchange on Cartesian topologies can be established in a similar way.

Although derived types are convenient to use, their performance implications are unclear, which is a good example for the rule that performance optimizations are not portable across MPI implementations. The library could aggregate the parts of

the new type into an internal contiguous buffer, but it could just as well send the pieces separately. Even if aggregation takes place, one cannot be sure whether it is done in the most efficient way; e.g., nontemporal stores could be beneficial for large data volume, or (if multiple threads per MPI process are available) copying could be multithreaded. In general, if communication of derived datatypes is crucial for performance, one should not rely on the library's efficiency but check whether manual copying improves performance. If it does, this "performance bug" should be reported to the provider of the MPI library.

10.4.3 Nonblocking vs. asynchronous communication

Besides the efforts towards reducing communication overhead as described in the preceding sections, a further chance for increasing efficiency of parallel programs is overlapping communication and computation. Nonblocking point-to-point communication seems to be the straightforward way to achieve this, and we have actually made (limited) use of it in the MPI-parallel Jacobi solver, where we have employed `MPI_Irecv()` to overlap halo receive with copying data to the send buffer and sending it (see Section 9.3). However, there was no concurrency between stencil updates (which comprise the actual "work") and communication. A way to achieve this would be to perform those stencil updates first that form subdomain boundaries, because they must be transmitted to the halo layers of neighboring subdomains. After the update and copying to intermediate buffers, `MPI_Isend()` could be used to send the data while the bulk stencil updates are done.

However, as mentioned earlier, one must strictly differentiate between nonblocking and truly *asynchronous* communication. Nonblocking semantics, according to the MPI standard, merely implies that the message buffer cannot be used after the call has returned from the MPI library; while certainly desirable, it is entirely up to the implementation whether data transfer, i.e., MPI progress, takes place while user code is being executed outside MPI.

Listing 10.1 shows a simple benchmark that can be used to determine whether an MPI library supports asynchronous communication. This code is to be executed by exactly two processors (we have omitted initialization code, etc.). The `do_work()` function executes some user code with a duration given by its parameter in seconds. In order to rule out contention effects, the function should perform operations that do not interfere with simultaneous memory transfers, like register-to-register arithmetic. The data size for MPI (`count`) was chosen so that the message transfer takes a considerable amount of time (tens of milliseconds) even on the most modern networks. If `MPI_Irecv()` triggers a truly asynchronous data transfer, the measured overall time will stay constant with increasing delay until the delay equals the message transfer time. Beyond this point, there will be a linear rise in execution time. If, on the other hand, MPI progress occurs only inside the MPI library (which means, in this example, within `MPI_Wait()`), the time for data transfer and the time for executing `do_work()` will always add up and there will be a linear rise of overall execution time starting from zero delay. Figure 10.14 shows internode data (open symbols) for some current parallel architectures and interconnects. Among those, only the Cray

Listing 10.1: Simple benchmark for evaluating the ability of the MPI library to perform asynchronous point-to-point communication.

```
1  double precision :: delay
2  integer :: count, req
3  count = 80000000
4  delay = 0.d0
5
6  do
7    call MPI_Barrier(MPI_COMM_WORLD, ierr)
8    if(rank.eq.0) then
9      t = MPI_Wtime()
10     call MPI_Irecv(buf, count, MPI_BYTE, 1, 0, &
11                    MPI_COMM_WORLD, req, ierr)
12     call do_work(delay)
13     call MPI_Wait(req, status, ierr)
14     t = MPI_Wtime() - t
15   else
16     call MPI_Send(buf, count, MPI_BYTE, 0, 0, &
17                   MPI_COMM_WORLD, ierr)
18   endif
19   write(*,*) 'Overall: ',t,' Delay: ',delay
20   delay = delay + 1.d-2
21   if(delay.ge.2.d0) exit
22 enddo
```


XT line of massively parallel systems supports asynchronous internode MPI by default (open diamonds). For the IBM Blue Gene/P system the default behavior is to use polling for message progress, which rules out asynchronous transfer (open squares). However, interrupt-based progress can be activated on this machine [V116], enabling asynchronous message-passing (filled squares).

One should mention that the results could change if the `do_work()` function executes memory-bound code, because the message transfer may interfere with the CPU's use of memory bandwidth. However, this effect can only be significant if the network bandwidth is large enough to become comparable to the aggregate memory bandwidth of a node, but that is not the case on today's systems.

Although the selection of systems is by no means an exhaustive survey of current technology, the result is representative. Within the whole spectrum from commodity clusters to expensive, custom-made supercomputers, there is hardly any support for asynchronous nonblocking transfers, although most computer systems do feature hardware facilities like DMA engines that would allow background communication. The situation is even worse for intranode message passing because dedicated hardware for memory-to-memory copies is rare. For the Cray XT4 this is demonstrated in Figure 10.14 (filled diamonds). Note that the pure communication time roughly matches the time for the intranode case, although the machine's network is not used and MPI can employ shared-memory copying. This is because the MPI point-to-point bandwidth for large messages is nearly identical for intranode and internode situa-

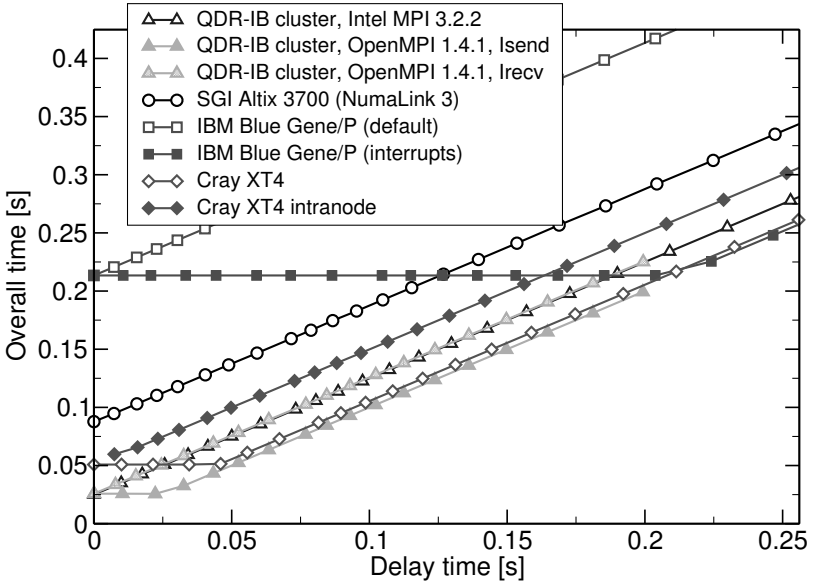


Figure 10.14: Results from the MPI overlap benchmark on different architectures, interconnects and MPI versions. Among those, only the MPI library on a Cray XT4 (diamonds) and OpenMPI on an InfiniBand cluster (filled triangles) are capable of asynchronous transfers by default, OpenMPI allowing no overlap for nonblocking receives, however. With intranode communication, overlap is generally unavailable on the systems considered. On the IBM Blue Gene/P system, asynchronous transfers can be enabled by activating interrupt-driven progress (filled squares) via setting `DCMF_INTERRUPTS`=1.

tions, a feature that is very common among hybrid parallel systems. See Section 10.5 for a discussion.

The lesson is that one should not put too much optimization effort into utilizing asynchronous communication by means of nonblocking point-to-point calls, because it will only pay off in very few environments. This does not mean, however, that nonblocking MPI is useless; it is valuable for preventing deadlocks, reducing idle times due to synchronization overhead, and handling multiple outstanding communication requests efficiently. An example for the latter is the utilization of full-duplex transfers if send and receive operations are outstanding at the same time. In contrast to asynchronous transfers, full-duplex communication is supported by most interconnects and MPI implementations today.

Overlapping communication with computation is still possible even without direct MPI support by dedicating a separate thread (OpenMP, or any other variant of threading) to handling MPI calls while other threads execute user code. This is a variant of *hybrid programming*, which will be discussed in Chapter 11.

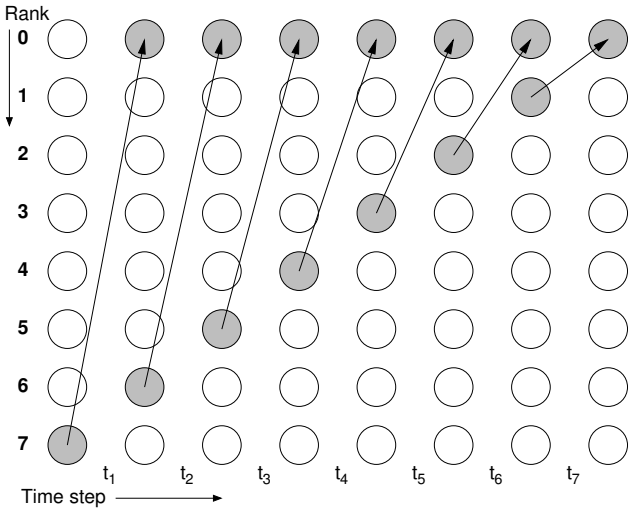


Figure 10.15: A global reduction with communication overhead being linear in the number of processes, as implemented in the integration example (Listing 9.3). Each arrow represents a message to the receiver process. Processes that communicate during a time step are shaded.

10.4.4 Collective communication

In Section 9.2.3 we modified the numerical integration program by replacing the "manual" accumulation of partial results by a single call to `MPI_Reduce()`. Apart from a general reduction in programming complexity, collective communication also bears optimization potential: The way the program was originally formulated makes communication overhead a linear function of the number of processes, because there is severe contention at the receiver side even if nonblocking communication is used (see Figure 10.15). A "tree-like" communication pattern, where partial results are added up by groups of processes and propagated towards the receiving rank can change the linear dependency to a logarithm if the network is sufficiently nonblocking (see Figure 10.16). (We are treating network latency and bandwidth on the same footing here.) Although each individual process will usually have to serialize all its sends and receives, there is enough concurrency to make the tree pattern much more efficient than the simple linear approach.

Collective MPI calls have appropriate algorithms built in to achieve reasonable performance on any network [137]. In the ideal case, the MPI library even has sufficient knowledge about the network topology to choose the optimal communication pattern. This is the main reason why collectives should be preferred over simple implementations of equivalent functionality using point-to-point calls. See also Problem 10.2.

10.5 Understanding intranode point-to-point communication

When figuring out the optimal distribution of threads and processes across the cores and nodes of a system, it is often assumed that any intranode MPI communica-

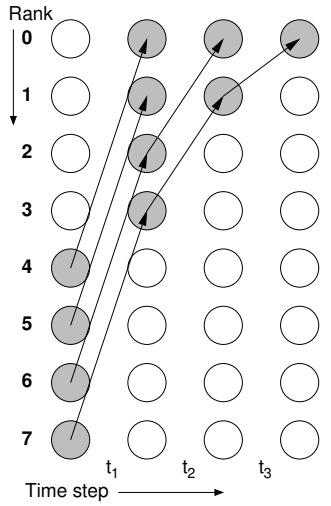


Figure 10.16: With a tree-like, hierarchical reduction pattern, communication overhead is logarithmic in the number of processes because communication during each time step is concurrent.

tion is infinitely fast (see also Section 10.4.1 above). Surprisingly, this is not true in general, especially with regard to bandwidth. Although even a single core can today draw a bandwidth of multiple GBytes/sec out of a chip's memory interface, inefficient intranode communication mechanisms used by the MPI implementation can harm performance dramatically. The simplest "bug" in this respect can arise when the MPI library is not aware of the fact that two communicating processes run on the same shared-memory node. In this case, relatively slow network protocols are used instead of memory-to-memory copies. But even if the library does employ shared-memory communication where applicable, there is a spectrum of possible strategies:

- Nontemporal stores or cache line zero (see Section 1.3.1) may be used or not, probably depending on message and cache sizes. If a message is small and both processes run in a cache group, using nontemporal stores is usually counterproductive because it generates additional memory traffic. However, if there is no shared cache or the message is large, the data must be written to main memory anyway, and nontemporal stores avoid the write allocate that would otherwise be required.

- The data transfer may be "single-copy," meaning that a simple block copy operation is sufficient to copy the message from the send buffer to the receive buffer (implicitly implementing a synchronizing *rendezvous* protocol), or an intermediate (internal) buffer may be used. The latter strategy requires additional copy operations, which can drastically diminish communication bandwidth if the network is fast.

- There may be hardware support for intranode memory-to-memory transfers. In situations where shared caches are unimportant for communication performance, using dedicated hardware facilities can result in superior point-to-point bandwidth [138].

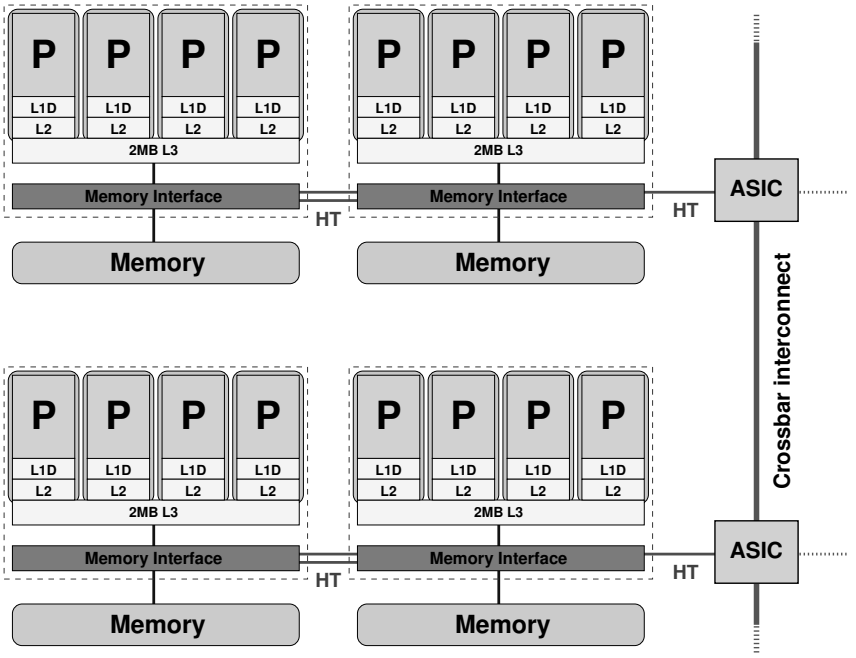


Figure 10.17: Two nodes of a Cray XT5 system. Dashed boxes denote AMD Opteron sockets, and there are two sockets (NUMA locality domains) per node. The crossbar interconnect is actually a 3D torus (mesh) network.

The behavior of MPI libraries with respect to above issues can sometimes be influenced by tunable parameters, but the rapid evolution of multicore processor architectures with complex cache hierarchies and system designs also makes it a subject of intense development.

Again, the simple PingPong benchmark (see Section 4.5.1) from the IMB suite can be used to fathom the properties of intranode MPI communication [O70, O71]. As an outstanding example we use a Cray XT5 system. One XT5 node comprises two AMD Opteron chips with a 2 MB quad-core L3 group each. These nodes are connected via a 3D torus network (see Figure 10.17). Due to this structure one can expect three different levels of point-to-point communication characteristics, depending on whether message transfer occurs inside an L3 group (intranode intrasocket), between cores on different sockets (intranode intersocket), or between different nodes (internode). (If a node had more than two ccNUMA locality domains, there would be even more variety.) Figure 10.18 shows internode and intranode PingPong data for this system. As expected, communication characteristics are quite different between internode and intranode situations for small and intermediate-length messages. Two cores on the same socket can really benefit from the shared L3 cache, leading to a peak bandwidth of over 3 GBytes/sec. Surprisingly, the characteristics for intersocket communication are very similar (dashed line), although there is no shared

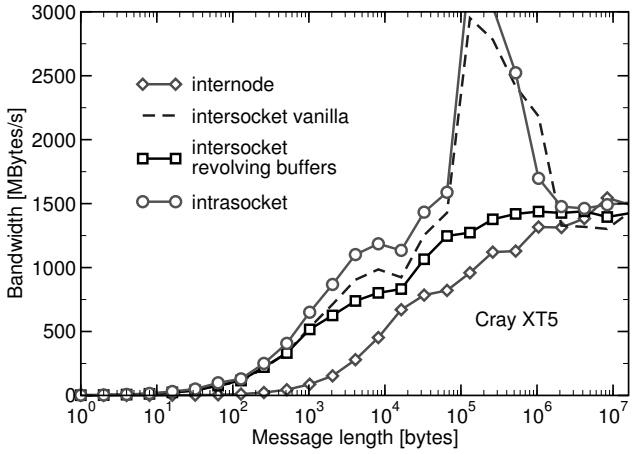
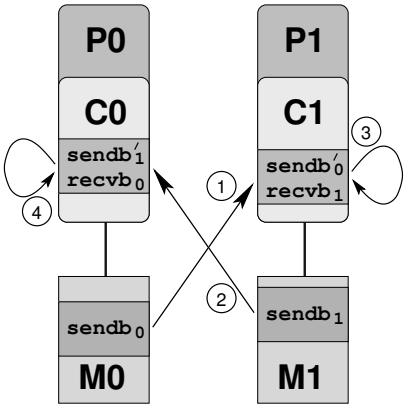


Figure 10.18: IMB PingPong performance for internode, intranode but intersocket, and pure intrasocket communication on a Cray XT5 system. Intersocket "vanilla" data was obtained without using revolving buffers (see text for details).

cache and the large bandwidth "hump" should not be present because all data must be exchanged via main memory. The explanation for this peculiar effect lies in the way the standard PingPong benchmark is usually performed [A89]. In contrast to the pseudocode shown on page 105, the real IMB PingPong code is structured as follows:

```
1  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
2  if(rank.eq.0) then
3    targetID = 1
4    S = MPI_Wtime()
5    do i=1,ITER
6      call MPI_Send(buffer,N,MPI_BYTE,targetID,...)
7      call MPI_Recv(buffer,N,MPI_BYTE,targetID,...)
8    enddo
9    E = MPI_Wtime()
10   BWIDTH = ITER*2*N/(E-S)/1.d6      ! MBytes/sec rate
11   TIME   = (E-S)/2*1.d6/ITER        ! transfer time in microsecs
12                                     ! for single message
13 else
14   targetID = 0
15   do i=1,ITER
16     call MPI_Recv(buffer,N,MPI_BYTE,targetID,...)
17     call MPI_Send(buffer,N,MPI_BYTE,targetID,...)
18   enddo
19 endif
```

Most notably, to get accurate timing measurements even for small messages, the Ping-Pong message transfer is repeated a number of times (`ITER`). Keeping this peculiarity in mind, it is now possible to explain the bandwidth "hump" (see Figure 10.19): The transfer of $\texttt{sendb}_0$ from process 0 to $\texttt{recvb}_1$ of process 1 can be implemented as a *single-copy* operation on the receiver side, i.e., process 1 executes $\texttt{recvb}_1\texttt{(1:N)} = \texttt{sendb}_0\texttt{(1:N)}$, where `N` is the number of bytes in the message. If `N` is sufficiently small, the data from $\texttt{sendb}_0$ is located in the cache of process 1 and there is no need to replace or modify these cache entries unless $\texttt{sendb}_0$ gets



1. First ping: P1 copies $\texttt{sendb}_0$ to $\texttt{recvb}_1$, which resides in its cache.
2. First pong: P0 copies $\texttt{sendb}_1$ to $\texttt{recvb}_0$, which resides in its cache.
3. Second ping: P1 performs in-cache copy operation on its unmodified $\texttt{recvb}_1$.
4. Second pong: P0 performs in-cache copy operation on its unmodified $\texttt{recvb}_0$.
5. ... Repeat steps 3 and 4, working in cache.

Figure 10.19: Chain of events for the standard MPI PingPong on shared-memory systems when the messages fit in the cache. C0 and C1 denote the caches of processors P0 and P1, respectively. M0 and M1 are the local memories of P0 and P1.

modified. However, the send buffers are not changed on either process in the loop kernel. Thus, after the first iteration the send buffers are located in the caches of the receiving processes and in-cache copy operations occur in the subsequent iterations instead of data transfer through memory and the HyperTransport network.

There are two reasons for the performance drop at larger message sizes: First, the L3 cache (2 MB) is to small to hold both or at least one of the local receive buffer and the remote send buffer. Second, the IMB is performed so that the number of repetitions is decreased with increasing message size until only one iteration — which is the initial copy operation through the network — is done for large messages.

Real-world applications can obviously not make use of the "performance hump." In order to evaluate the true potential of intranode communication for codes that should benefit from single-copy for large messages, one may add a second PingPong operation in the inner iteration with arrays $\texttt{sendb}_i$ and $\texttt{recvb}_i$ interchanged (i.e., $\texttt{sendb}_i$ is specified as the receive buffer with the second `MPI_Recv()` on process number i), the sending process i gains exclusive ownership of $\texttt{sendb}_i$ again. Another alternative is the use of "revolving buffers," where a PingPong send/receive pair uses a small, sliding window out of a much larger memory region for send and receive buffers, respectively. After each PingPong the window is shifted by its own size, so that send and receive buffer locations in memory are constantly changing. If the size of the large array is chosen to be larger than any cache, it is guaranteed that all send buffers are actually evicted to memory at some point, even if a single message fits into cache and the MPI library uses single-copy transfers. The IMB benchmarks allow the use of revolving buffers by a command-line option, and the resulting performance data (squares in Figure 10.18) shows no overshooting for in-cache message sizes.

Interestingly, intranode and internode bandwidths meet at roughly the same asymptotic performance for large messages, refuting the widespread misconception that intranode point-to-point communication is infinitely fast. This observation, al-

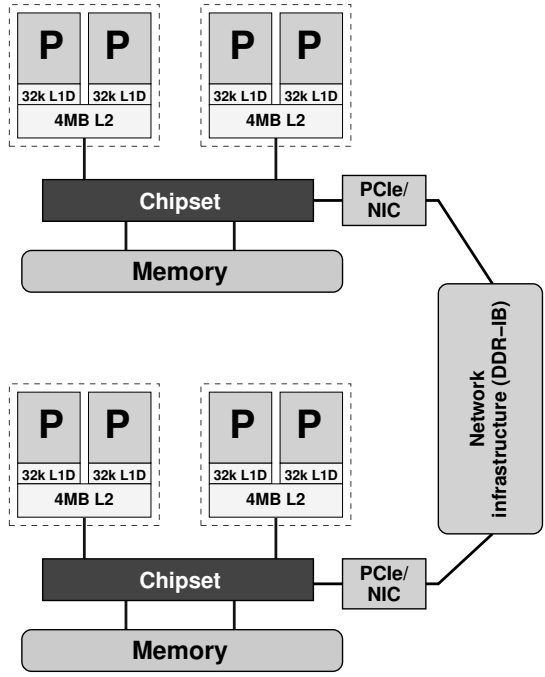


Figure 10.20: Two nodes of a Xeon 5160 dual-socket cluster system with a DDR-InfiniBand interconnect.

though shown here for a specific system architecture and software environment, is almost universal across many contemporary (hybrid) parallel systems, and especially "commodity" clusters. However, there is a large variation in the details, and since MPI libraries are continuously evolving, characteristics tend to change over time. In Figures 10.21 and 10.22 we show PingPong performance data on a cluster comprised of dual-socket Intel Xeon 5160 nodes (see Figure 10.20), connected via DDR-InfiniBand. The only difference between the two graphs is the version number of the MPI library used (comparing Intel MPI 3.0 and 3.1). Details about the actual modifications to the MPI implementation are undisclosed, but the observation of large performance variations between the two versions reveals that simple models about intranode communication are problematic and may lead to false conclusions.

At small message sizes, MPI communication is latency-dominated. For the systems described above, the latencies measured by the IMB PingPong benchmark are shown in Table 10.1, together with asymptotic bandwidth numbers. Clearly, latency is much smaller when both processes run on the same node (and smaller still if they share a cache). We must emphasize that these benchmarks can only give a rough impression of intranode versus internode message passing issues. If multiple process pairs communicate concurrently (which is usually the case in real-world applications), the situation gets much more complex. See Ref. [O72] for a more detailed analysis in the context of hybrid MPI/OpenMP programming.

The most important conclusion that must be drawn from the bandwidth and latency characteristics shown above is that process-core affinity can play a major role

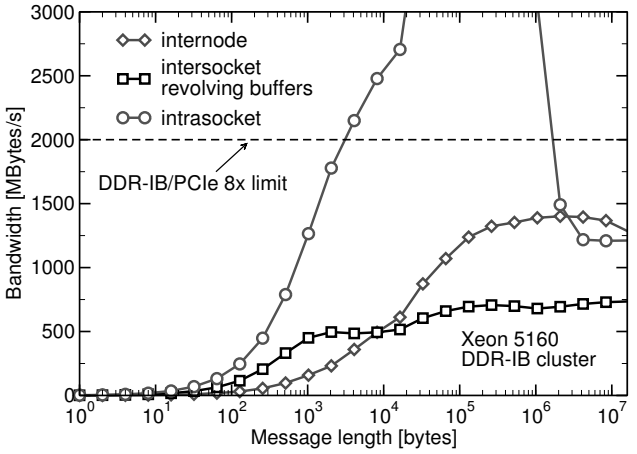


Figure 10.21: IMB PingPong performance for internode, intranode but intersocket, and pure intrasocket communication on a Xeon 5160 DDR-IB cluster, using Intel MPI 3.0.

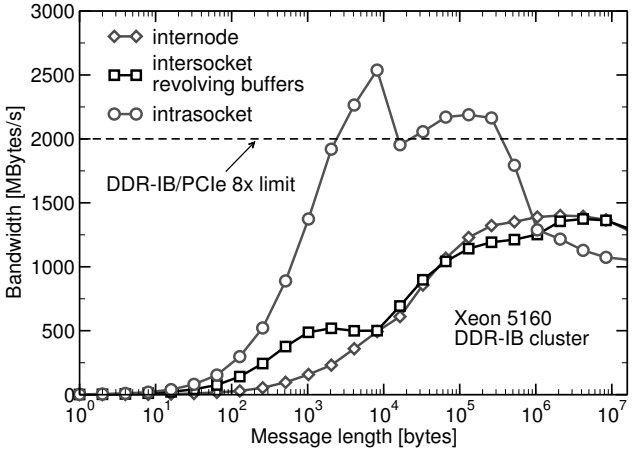


Figure 10.22: The same benchmark as in Figure 10.21, but using Intel MPI 3.1. Intranode behavior has changed significantly.

	Latency [μs]			Bandwidth [MBytes/sec]		
	XT5	Xeon-IB		XT5	Xeon-IB	
Mode	MPT 3.1	IMPI 3.0	IMPI 3.1	MPT 3.1	IMPI 3.0	IMPI 3.1
internode	7.40	3.13	3.24	1500	1300	1300
intersocket	0.63	0.76	0.55	1400	750	1300
intrasocket	0.49	0.40	0.31	1500	1200	1100

Table 10.1: Measured latency and asymptotic bandwidth from the IMB PingPong benchmark on a Cray XT5 and a commodity Xeon cluster with DDR-InifiniBand interconnect.

for application performance on the "anisotropic" multisocket multicore systems that are popular today (similar effects, though not directly related to communication, appear in OpenMP programming, as shown in Sections 6.2 and 7.2.2). Mapping issues as described in Section 10.4.1 are thus becoming relevant on the intranode topology level, too; for instance, given appropriate message sizes and an MPI code that mainly uses next-neighbor communication, neighboring MPI ranks should be placed in the same cache group. Of course, other factors like shared data paths to memory and NUMA constraints should be considered as well, and there is no general rule. Note also that in strong scaling scenarios it is possible that one "rides down the PingPong curve" towards a latency-driven regime with increasing processor count, possibly rendering the performance assumptions useless that process/thread placement was based on for small numbers of processes (see also Problem 10.5).

Problems

For solutions see page 306 *ff.*

10.1 *Reductions and contention.* Comparing Figures 10.15 and 10.16, can you think of a network topology that would lead to the same performance for a reduction operation in both cases? Assuming a fully nonblocking fat-tree network, what could be other factors that would prevent optimal performance with hierarchical reductions?

10.2 *Allreduce, optimized.* We stated that `MPI_Allreduce()` is a combination of `MPI_Reduce()` and `MPI_Bcast()`. While this is semantically correct, implementing `MPI_Allreduce()` in this way is very inefficient. How can it be done better?

10.3 *Eager vs. rendezvous.* Looking again at the overview on parallelization methods in Section 5.2, what is a typical situation where using the "eager" message transfer protocol for MPI could have bad side effects? What are possible solutions?

10.4 *Is cubic always optimal?* In Section 10.4.1 we have shown that communication overhead for strong scaling due to halo exchange shows the most favorable dependence on N, the number of workers, if the domain is cut across all three coordinate axes. Does this strategy always lead to minimum overhead?

10.5 *Riding the PingPong curve.* For strong scaling and cubic domain decomposition with halo exchange as shown in Section 10.4.1, derive an expression for the effective bandwidth $B_{\mathrm{eff}}(N, L, w, T_\ell, B)$. Assume that a point-to-point message transfer can be described by the simple latency/bandwidth model (4.2), and that there is no overlap between communication in different directions and between computation and communication.

10.6 *Nonblocking Jacobi revisited.* In Section 9.3 we used a nonblocking receive to avoid deadlocks on halo exchange. However, exactly one nonblocking request was outstanding per process at any time. Can the code be reorganized to use multiple outstanding requests? Are there any disadvantages?

10.7 *Send and receive combined.* `MPI_Sendrecv()` is a combination of a standard send (`MPI_Send()`) and a standard receive (`MPI_Recv()`) in a single call:

```
<type> sendbuf(*), recvbuf(*)
integer :: sendcount, sendtype, dest, sendtag,
           recvcount, recvtype, source, recvtag,
           comm, status(MPI_STATUS_SIZE), ierror
call MPI_Sendrecv(sendbuf,      ! send buffer
                  sendcount,    ! # of items to send
                  sendtype,     ! send data type
                  dest,         ! destination rank
                  sendtag,      ! tag for receive
                  recvbuf,      ! receive buffer
                  recvcount,    ! # of items to receive
                  recvtype,     ! recv data type
                  source,       ! source rank
                  recvtag,      ! tag for send
                  status,       ! status array for recv
                  comm,         ! communicator
                  ierror)       ! return value
```

How would you implement this function so that it is guaranteed not to deadlock if used for a ring shift communication pattern? Are there any other positive side effects to be expected?

10.8 *Load balancing and domain decomposition.* In 3D (cubic) domain decomposition with open (i.e., nontoroidal) boundary conditions, what are the implications of communication overhead on load balance? Assume that the MPI communication properties are constant and isotropic throughout the parallel system, and that communication cannot be overlapped with computation. Would it make sense to enlarge the outermost subdomains in order to compensate for their reduced surface area?