Chapter 8

Locality optimizations on ccNUMA architectures

It was mentioned already in the section on ccNUMA architecture that, for applications whose performance is bound by memory bandwidth, locality and contention problems (see Figures 8.1 and 8.2) tend to turn up when threads/processes and their data are not carefully placed across the locality domains of a ccNUMA system. Unfortunately, the current OpenMP standard (3.0) does not refer to page placement at all and it is up to the programmer to use the tools that system builders provide. This chapter discusses the general, i.e., mostly system-independent options for correct data placement, and possible pitfalls that may prevent it. We will also show that page placement is not an issue that is restricted to shared-memory parallel programming.

8.1 Locality of access on ccNUMA

Although ccNUMA architectures are ubiquitous today, the need for ccNUMA-awareness has not yet arrived in all application areas; memory-bound code must be designed to employ proper page placement [O67]. The placement problem has two dimensions: First, one has to make sure that memory gets mapped into the locality domains of processors that actually access them. This minimizes NUMA traffic

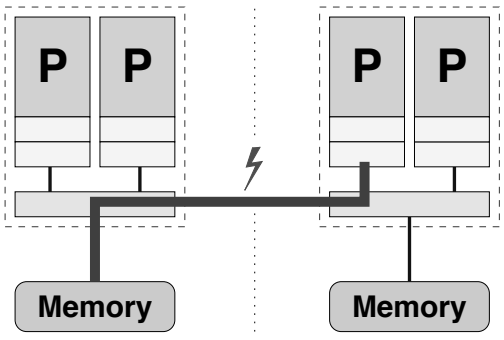


Figure 8.1: Locality problem on a ccNUMA system. Memory pages got mapped into a locality domain that is not connected to the accessing processor, leading to NUMA traffic.

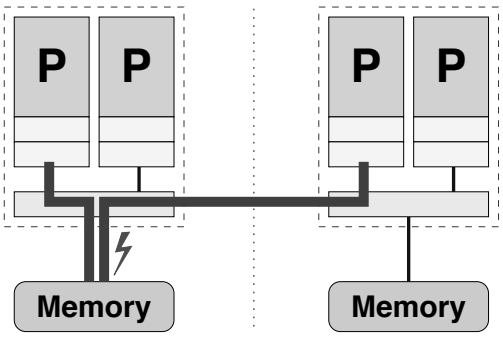


Figure 8.2: Contention problem on a ccNUMA system. Even if the network is very fast, a single locality domain can usually not saturate the bandwidth demands from concurrent local and nonlocal accesses.

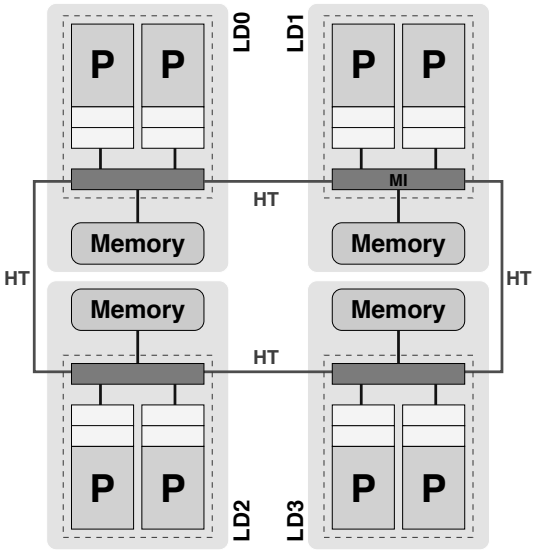


Figure 8.3: A ccNUMA system (based on dual-core AMD Opteron processors) with four locality domains LD0 ... LD3 and two cores per LD, coupled via a HyperTransport network. There are three NUMA access levels (local domain, one hop, two hops).

across the network. In this context, "mapping" means that a page table entry is set up, which describes the association of a physical with a virtual memory page. Consequently, locality of access in ccNUMA systems is always followed on the OS page level, with typical page sizes of (commonly) 4 kB or (more rarely) 16 kB, sometimes larger. Hence, strict locality may be hard to implement with working sets that only encompass a few pages, although the problem tends to be cache-bound in this case anyway. Second, threads or processes must be *pinned* to those CPUs which had originally mapped their memory regions in order not to lose locality of access. In the following we assume that appropriate affinity mechanisms have been employed (see Appendix A).

A typical ccNUMA node with four locality domains is depicted in Figure 8.3. It uses two HyperTransport (HT) links per socket to connect to neighboring domains, which results in a "closed chain" topology. Memory access is hence categorized into three levels, depending on how many HT hops (zero, one, or two) are required to reach the desired page. The actual remote bandwidth and latency penalties can vary significantly across different systems; vector triad measurements can at least provide rough guidelines. See the following sections for details about how to control page placement.

Note that even with an extremely fast NUMA interconnect whose bandwidth and latency are comparable to local memory access, the contention problem cannot be eliminated. No interconnect, no matter how fast, can turn ccNUMA into UMA.

8.1.1 Page placement by first touch

Fortunately, the initial mapping requirement can be fulfilled in a portable manner on all current ccNUMA architectures. If configured correctly (this pertains to firmware ["BIOS"], operating system and runtime libraries alike), they support a *first touch* policy for memory pages: A page gets mapped into the locality domain of the processor that first writes to it. Merely *allocating* memory is not sufficient.

It is therefore the data initialization code that deserves attention on ccNUMA (and using `calloc()` in C will most probably be counterproductive). As an example we look again at a naïve OpenMP-parallel implementation of the vector triad code from Listing 1.1. Instead of allocating arrays on the stack, however, we now use dynamic (heap) memory for reasons which will be explained later (we omit the timing functionality for brevity):

```
1   double precision, allocatable, dimension(:) :: A, B, C, D
2   allocate(A(N), B(N), C(N), D(N))
3  ! initialization
4   do i=1,N
5     B(i) = i; C(i) = mod(i,5); D(i) = mod(i,10)
6   enddo
7   ...
8   do j=1,R
9  !$OMP PARALLEL DO
10    do i=1,N
11      A(i) = B(i) + C(i) * D(i)
12    enddo
13 !$OMP END PARALLEL DO
14    call dummy(A,B,C,D)
15  enddo
```

Here we have explicitly written out the loop which initializes arrays `B`, `C`, and `D` with sensible data (it is not required to initialize `A` because it will not be read before being written later). If this code, which is prototypical for many OpenMP applications that have not been optimized for ccNUMA, is run across several locality domains, it will not scale beyond the maximum performance achievable on a single LD if the working set does not fit into cache. This is because the initialization loop is executed by a single thread, writing to `B`, `C`, and `D` for the first time. Hence, all memory pages belonging to those arrays will be mapped into a single LD. As Figure 8.4 shows, the consequences are significant: If the working set fits into the aggregated cache, scalability is good. For large arrays, however, 8-thread performance (filled circles) drops even below the 2-thread (one LD) value (open diamonds), because all threads access memory in LD0 via the HT network, leading to severe contention. As mentioned above, this problem can be solved by performing array initialization in parallel. The loop from lines 4–6 in the code above should thus be replaced by:

```
1  ! initialization
2  !$OMP PARALLEL DO
3    do i=1,N
4      B(i) = i; C(i) = mod(i,5); D(i) = mod(i,10)
5    enddo
6  !$OMP END PARALLEL DO
```

This simple modification, which is actually a no-op on UMA systems, makes a huge difference on ccNUMA in memory-bound situations (see open circles and inset in Figure 8.4). Of course, in the very large N limit where the working set does not fit into a single locality domain, data will be "automatically" distributed, but not

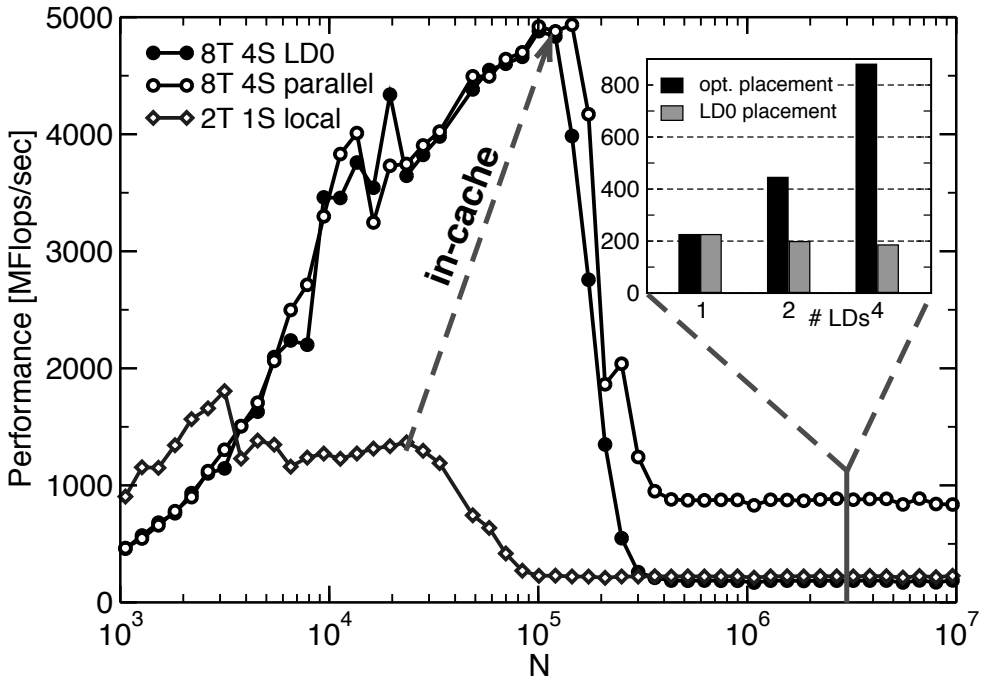


Figure 8.4: Vector triad performance and scalability on a four-LD ccNUMA machine like in Figure 8.3 (HP DL585 G5), comparing data for 8 threads with page placement in LD0 (filled circles) with correct parallel first touch (open circles). Performance data for local access in a single LD is shown for reference (open diamonds). Two threads per socket were used throughout. In-cache scalability is unharmed by unsuitable page placement. For memory-bound situations, putting all data into a single LD has ruinous consequences (see inset).

in a controlled way. This effect is by no means something to rely on when data distribution is key.

Sometimes it is not sufficient to just parallelize array initialization, for instance if there is no loop to parallelize. In the OpenMP code on the left of Figure 8.5, initialization of `A` is done in a serial region using the `READ` statement in line 8. The access to `A` in the parallel loop will then lead to contention. The version on the right corrects this problem by initializing `A` in parallel, first-touching its elements in the same way they are accessed later. Although the `READ` operation is still sequential, data will be distributed across the locality domains. Array `B` does not have to be initialized but will automatically be mapped correctly.

There are some requirements that must be fulfilled for first-touch to work properly and result in good loop performance scalability:

- The OpenMP loop schedules of initialization and work loops must obviously be identical and reproducible, i.e., the only possible choice is `STATIC` with a constant chunksize, and the use of tasking is ruled out. Since the OpenMP standard does not define a default schedule, it is a good idea to specify it explicitly on all parallel loops. All current compilers choose `STATIC` by default, though. Of course, the use of a static schedule poses some limits on possible optimizations for eliminating load imbalance. The only simple option is the choice of

```
integer,parameter:: N=1000000
double precision :: A(N),B(N)



! executed on single LD
READ(1000) A
! contention problem
!$OMP PARALLEL DO
  do i = 1, N
    B(i) = func(A(i))
  enddo
!$OMP END PARALLEL DO
```

→

```
integer,parameter:: N=1000000
double precision :: A(N),B(N)
!$OMP PARALLEL DO
  do i=1,N
    A(i) = 0.d0
  enddo
!$OMP END PARALLEL DO
! A is mapped now
READ(1000) A
!$OMP PARALLEL DO
  do i = 1, N
    B(i) = func(A(i))
  enddo
!$OMP END PARALLEL DO
```

Figure 8.5: Optimization by correct NUMA placement. Left: The `READ` statement is executed by a single thread, placing `A` to a single locality domain. Right: Doing parallel initialization leads to correct distribution of pages across domains.

an appropriate chunksize (as small as possible, but at least several pages of data). See Section 8.3.1 for more information about dynamic scheduling under ccNUMA conditions.

- For successive parallel loops with the same number of iterations and the same number of parallel threads, each thread should get the same part of the iteration space in both loops. The OpenMP 3.0 standard guarantees this behavior only if both loops use the `STATIC` schedule with the same chunksize (or none at all) and if they bind to the same parallel region. Although the latter condition can usually not be satisfied, at least not for all loops in a program, current compilers generate code which makes sure that the iteration space of loops of the same length and OpenMP schedule is always divided in the same way, even in different parallel regions.

- The hardware must actually be *capable* of scaling memory bandwidth across locality domains. This may not always be the case, e.g., if cache coherence traffic produces contention on the NUMA network.

Unfortunately it is not always at the programmer's discretion how and when data is touched first. In C/C++, global data (including objects) is initialized before the `main()` function even starts. If globals cannot be avoided, properly mapped local copies of global data may be a possible solution, code characteristics in terms of communication vs. calculation permitting [O68]. A discussion of some of the problems that emerge from the combination of OpenMP with C++ can be found in Section 8.4, and in [C100] and [C101].

It is not specified in a portable way how a page that has been allocated and initialized can lose its page table entry. In most cases it is sufficient to deallocate memory if it resides on the heap (using `DEALLOCATE` in Fortran, `free()` in C, or `delete[]` in C++). This is why we have reverted to the use of dynamic memory for the triad

benchmarks described above. If a new memory block is allocated later on, the first touch policy will apply as usual. Even so, some optimized implementations of runtime libraries will not actually deallocate memory on `free()` but add the pages to a "pool" to be re-allocated later with very little overhead. In case of doubt, the system documentation should be consulted for ways to change this behavior.

Locality problems tend to show up most prominently with shared-memory parallel code. Independently running processes automatically employ first touch placement if each process keeps its affinity to the locality domain where it had initialized its data. See, however, Section 8.3.2 for effects that may yet impede strictly local access.

8.1.2 Access locality by other means

Apart from plain first-touch initialization, operating systems often feature advanced tools for explicit page placement and diagnostics. These facilities are highly nonportable by nature. Often there are command-line tools or configurable dynamic objects that influence allocation and first-touch behavior without the need to change source code. Typical capabilities include:

- Setting policies or preferences to restrict mapping of memory pages to specific locality domains, irrespective of where the allocating process or thread is running.
- Setting policies for distributing the mapping of successively touched pages across locality domains in a "round-robin" or even random fashion. If a shared-memory parallel program has erratic access patterns (e.g., due to limitations imposed by the need for load balancing), and a coherent first-touch mapping cannot be employed, this may be a way to get at least a limited level of parallel scalability for memory-bound codes. See also Section 8.3.1 for relevant examples.
- Diagnosing the current distribution of pages over locality domains, probably on a per-process basis.

Apart from stand-alone tools, there is always a library with a documented API, which provides more fine-grained control over page placement. Under the Linux OS, the `numatools` package contains all the functionality described, and also allows thread/process affinity control (i.e, to determine which thread/process should run where). See Appendix A for more information.

8.2 Case study: ccNUMA optimization of sparse MVM

It is now clear that the bad scalability of OpenMP-parallelized sparse MVM codes on ccNUMA systems (see Figure 7.7) is caused by contention due to the memory pages of the code's working set being mapped into a single locality domain on

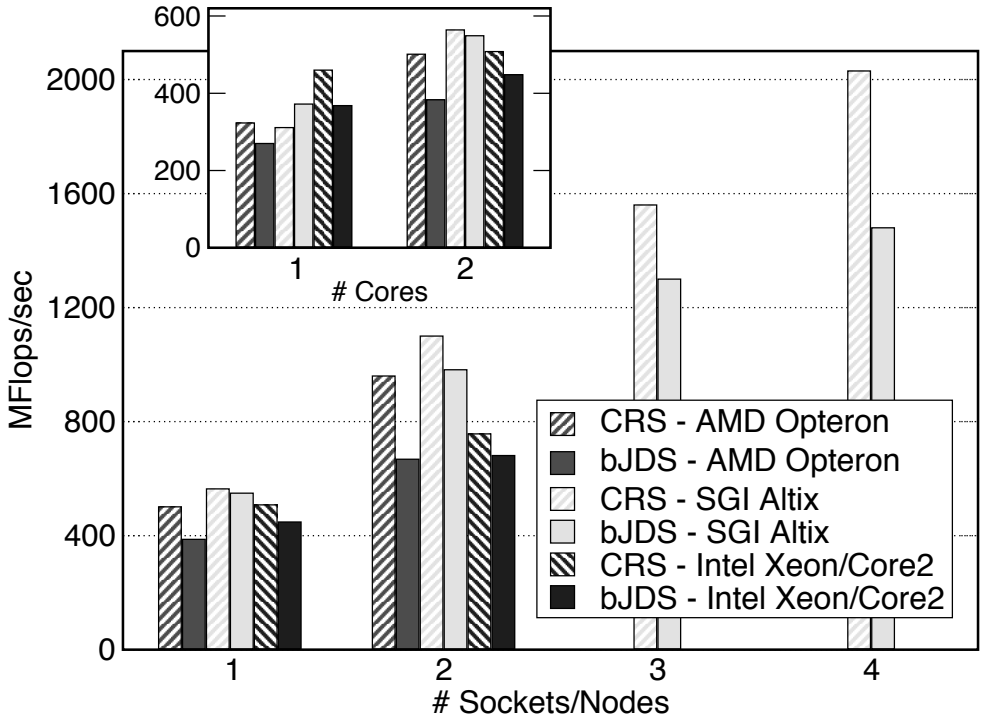


Figure 8.6: Performance and strong scaling for ccNUMA-optimized OpenMP parallelization of sparse MVM on three different architectures, comparing CRS (hatched bars) and blocked JDS (solid bars) variants. Cf. Figure 7.7 for performance without proper placement. The different scaling baselines have been separated (one socket/LD in the main frame, one core in the inset).

initialization. By writing parallel initialization loops that exploit first touch mapping policy, scaling can be improved considerably. We will restrict ourselves to CRS here as the strategy is basically the same for JDS. Arrays `C`, `val`, `col_idx`, `row_ptr` and `B` must be initialized in parallel:

```
!$OMP PARALLEL DO
  do i=1,Nr
    row_ptr(i) = 0 ; C(i) = 0.d0 ; B(i) = 0.d0
  enddo
!$OMP END PARALLEL DO
.... ! preset row_ptr array
!$OMP PARALLEL DO PRIVATE(start,end,j)
  do i=1,Nr
    start = row_ptr(i) ; end = row_ptr(i+1)
    do j=start,end-1
      val(j) = 0.d0 ; col_idx(j) = 0
    enddo
  enddo
!$OMP END PARALLEL DO
```

The initialization of `B` is based on the assumption that the nonzeros of the matrix are roughly clustered around the main diagonal. Depending on the matrix structure it may be hard in practice to perform proper placement for the RHS vector at all.

Figure 8.6 shows performance data for the same architectures and sMVM codes as in Figure 7.7 but with appropriate ccNUMA placement. There is no change in scalability for the UMA platform, which was to be expected, but also on the ccNUMA systems for up to two threads (see inset). The reason is of course that both architectures feature two-processor locality domains, which are of UMA type. On four threads and above, the locality optimizations yield dramatically improved performance. Especially for the CRS version scalability is nearly perfect when going from $2n$ to $2(n+1)$ threads (the scaling baseline in the main panel is the locality domain or socket, respectively). The JDS variant of the code benefits from the optimizations as well, but falls behind CRS for larger thread numbers. This is because of the permutation map for JDS, which makes it hard to place larger portions of the RHS vector into the correct locality domains, and thus leads to increased NUMA traffic.

8.3 Placement pitfalls

We have demonstrated that data placement is of premier importance on ccNUMA architectures, including commonly used two-socket cluster nodes. In principle, ccNUMA offers superior scalability for memory-bound codes, but UMA systems are much easier to handle and require no code optimization for locality of access. One can expect, though, that ccNUMA designs will prevail in the commodity HPC market, where dual-socket configurations occupy a price vs. performance "sweet spot." It must be emphasized, however, that the placement optimizations introduced in Section 8.1 may not always be applicable, e.g., when dynamic scheduling is unavoidable (see Section 8.3.1). Moreover, one may have arrived at the conclusion that placement problems are restricted to shared-memory programming; this is entirely untrue and Section 8.3.2 will offer some more insight.

8.3.1 NUMA-unfriendly OpenMP scheduling

As explained in Sections 6.1.3 and 6.1.7, dynamic/guided loop scheduling and OpenMP `task` constructs could be preferable over static work distribution in poorly load-balanced situations, if the additional overhead caused by frequently assigning tasks to threads is negligible. On the other hand, any sort of dynamic scheduling (including tasking) will necessarily lead to scalability problems if the thread team is spread across several locality domains. After all, the assignment of tasks to threads is unpredictable and even changes from run to run, which rules out an "optimal" page placement strategy.

Dropping parallel first touch altogether in such a situation is no solution as performance will then be limited by a single memory interface again. In order to get at least a significant fraction of the maximum achievable bandwidth, it may be best to distribute the working set's memory pages round-robin across the domains and hope for a statistically even distribution of accesses. Again, the vector triad can serve as a

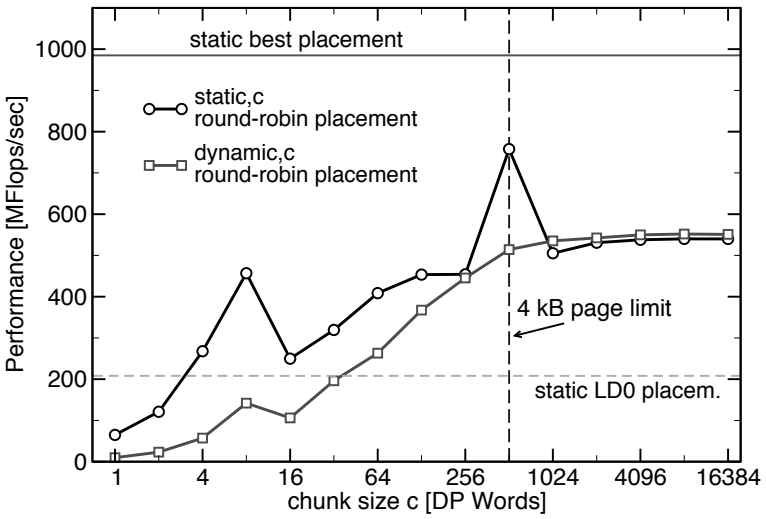


Figure 8.7: Vector triad performance vs. loop chunksize for static and dynamic scheduling with eight threads on a four-LD ccNUMA system (see Figure 8.3). Page placement was done round-robin on purpose. Performance for best parallel placement and LD0 placement with static scheduling is shown for reference.

convenient tool to fathom the impact of random page access. We modify the initialization loop by forcing static scheduling with a page-wide chunksize (assuming 4 kB pages):

```
1  ! initialization
2  !$OMP PARALLEL DO SCHEDULE(STATIC,512)
3    do i=1,N
4      A(i) = 0; B(i) = i; C(i) = mod(i,5); D(i) = mod(i,10)
5    enddo
6  !$OMP END PARALLEL DO
7    ...
8    do j=1,R
9  !$OMP PARALLEL DO SCHEDULE(RUNTIME)
10     do i=1,N
11       A(i) = B(i) + C(i) * D(i)
12     enddo
13 !$OMP END PARALLEL DO
14     call dummy(A,B,C,D)
15   enddo
```

By setting the `OMP_SCHEDULE` environment variable, different loop schedulings can be tested. Figure 8.7 shows parallel triad performance versus chunksize c for static and dynamic scheduling, respectively, using eight threads on the four-socket ccNUMA system from Figure8.3. At large c, where a single chunk spans several memory pages, performance converges asymptotically to a level governed by random access across all LDs, independent of the type of scheduling used. In this case, 75% of all pages a thread needs reside in remote domains. Although this kind of erratic pattern bears at least a certain level of parallelism (compared with purely serial initialization as shown with the dashed line), there is almost a 50% performance penalty versus the ideal case (solid line). The situation at $c = 512$ deserves some attention: With static scheduling, the access pattern of the triad loop matches the placement policy from the initialization loop, enabling (mostly) local access in each LD. The residual discrepancy to the best possible result can be attributed to the ar-

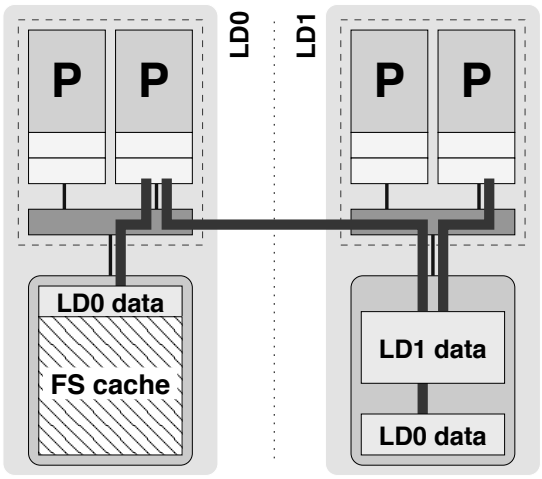


Figure 8.8: File system buffer cache can prevent locally touched pages to be placed in the local domain, leading to nonlocal access and contention. This is shown here for locality domain 0, where FS cache uses the major part of local memory. Some of the pages allocated and initialized by a core in LD0 get mapped into LD1.

rays not being aligned to page boundaries, which leads to some uncertainty regarding placement. Note that operating systems and compilers often provide means to align data structures to configurable boundaries (SIMD data type lengths, cache lines, and memory pages being typical candidates). Care should be taken, however, to avoid aliasing effects with strongly aligned data structures.

Although not directly related to NUMA effects, it is instructive to analyze the situation at smaller chunksizes as well. The additional overhead for dynamic scheduling causes a significant disadvantage compared to the static variant. If c is smaller than the cache line length (64 bytes here), each cache miss results in the transfer of a whole cache line of which only a fraction is needed, hence the peculiar behavior at $c \leq 64$. The interpretation of the breakdown at $c = 16$ and the gradual rise up until the page size is left as an exercise to the reader (see problems at the end of this chapter).

In summary, if purely static scheduling (without a chunksize) is ruled out, round-robin placement can at least exploit some parallelism. If possible, static scheduling with an appropriate chunksize should then be chosen for the OpenMP worksharing loops to prevent excessive scheduling overhead.

8.3.2 File system cache

Even if all precautions regarding affinity and page placement have been followed, it is still possible that scalability of OpenMP programs, but also overall system performance with independently running processes, is below expectations. Disk I/O operations cause operating systems to set up *buffer caches* which store recently read or written file data for efficient re-use. The size and placement of such caches is highly system-dependent and usually configurable, but the default setting is in most cases, although helpful for good I/O performance, less than fortunate in terms of ccNUMA locality.

See Figure 8.8 for an example: A thread or process running in LD0 writes a large file to disk, and the operating system reserves some space for a file system buffer cache in the memory attached to this domain. Later on, the same or another process in the this domain allocates and initializes memory ("LD0 data"), but not all of those

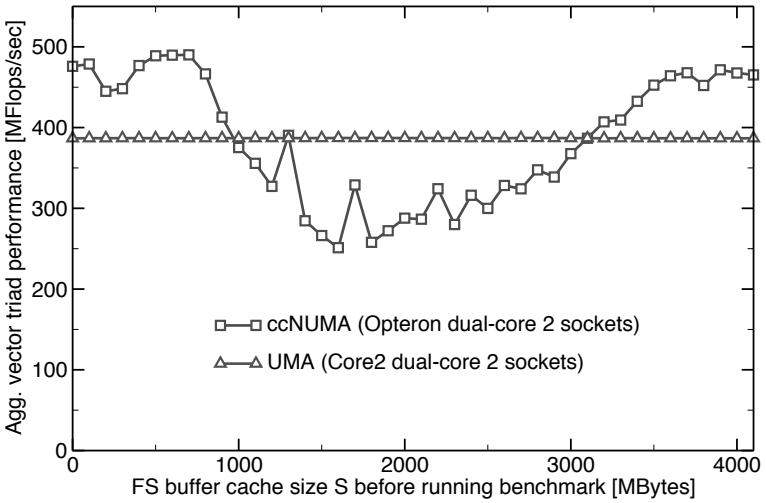


Figure 8.9: Performance impact of a large file system cache on a ccNUMA versus a UMA system (both with two sockets and four cores and 4 GB of RAM) when running four concurrent vector triads. The buffer cache was filled from a single core. See text for details. (Benchmark data by Michael Meier.)

pages fit into LD0 together with the buffer cache. By default, many systems then map the excess pages to another locality domain so that, even though first touch was correctly employed from the programmer's point of view, nonlocal access to LD0 data and contention at LD1's memory interface occurs.

A simple experiment can demonstrate this effect. We compare a UMA system (dual-core dual-socket Intel Xeon 5160 as in Figure 4.4) with a two-LD ccNUMA node (dual-core dual-socket AMD Opteron as in Figure 4.5), both equipped with 4 GB of main memory. On both systems we execute the following steps in a loop:

1. Write "dirty" data to disk and invalidate all the buffer cache. This step is highly system-dependent; usually there is a procedure that an administrator can execute[1] to do this, or a vendor-supplied library offers an option for it.

2. Write a file of size S to the local disk. The maximum file size equals the amount of memory. S should start at a very small size and be increased with every iteration of this loop until it equals the system's memory size.

3. Sync the cache so that there are no flush operations in the background (but the cache is still filled). This can usually be done by the standard UNIX `sync` command.

4. Run identical vector triad benchmarks on each core separately, using appropriate affinity mechanisms (see Appendix A). The aggregate size of all working sets should equal half of the node's memory. Report overall performance in MFlops/sec versus file size S (which equals the buffer cache size here).

The results are shown in Figure 8.9. On the UMA node, the buffer cache has certainly no impact at all as there is no concept of locality. On the other hand, the ccNUMA system shows a strong performance breakdown with growing file size and hits rock

[1] On a current Linux OS, this can be done by executing the command `echo 1 > /proc/sys/vm/drop_caches`. SGI Altix systems provide the `bcfree` command, which serves a similar purpose.

bottom when one LD is completely filled with buffer cache (at around 2 GB): This is when all the memory pages that were initialized by the triad loops in LD0 had to be mapped to LD1. Not surprisingly, if the file gets even larger, performance starts rising again because one locality domain is too small to hold even the buffer cache. If the file size equals the memory size (4 GB), parallel first touch tosses cache pages as needed and hence works as usual.

There are several lessons to be learned from this experiment. Most importantly it demonstrates that locality issues on ccNUMA are neither restricted to OpenMP (or generally, shared memory parallel) programs, nor is correct first touch a guarantee for getting "perfect" scalability. The buffer cache could even be a remnant from a previous job run by another user. Ideally there should be a way in production HPC environments to automatically "toss" the buffer cache whenever a production job finishes in order to leave a "clean" machine for the next user. As a last resort, if there are no user-level tools, and system administrators have not given this issue due attention, the normal user without special permissions can always execute a "sweeper" code, which allocates and initializes all memory:

```
1   double precision, allocatable, dimension(:) :: A
2   double precision :: tmp
3   integer(kind=8) :: i
4   integer(kind=8), parameter :: SIZE = SIZE_OF_MEMORY_IN_DOUBLES
5   allocate A(SIZE)
6   tmp=0.d0
7  ! touch all pages
8  !$OMP PARALLEL DO
9   do i=1, SIZE
10    A(i) = SQRT(DBLE(i))  ! dummy values
11  enddo
12 !$OMP END PARALLEL DO
13 ! actually use the result
14 !$OMP PARALLEL DO
15  do i=1,SIZE
16    tmp = tmp + A(i)*A(1)
17  enddo
18 !$OMP END PARALLEL DO
19  print *,tmp
```

This code could also be used as part of a user application to toss buffer cache that was filled by I/O from the running program (this pertains to reading and writing alike). The second loop serves the sole purpose of preventing the compiler from optimizing away the first because it detects that `A` is never actually used. Parallelizing the loops is of course optional but can speed up the whole process. Note that, depending on the actual amount of memory and the number of "dirty" file cache blocks, this procedure could take a considerable amount of time: In the worst case, nearly all of main memory has to be written to disk.

Buffer cache and the resulting locality problems are one reason why performance results for parallel runs on clusters of ccNUMA nodes tend to show strong fluctuations. If many nodes are involved, a large buffer cache on only one of them can hamper the performance of the whole parallel application. It is the task of system ad-

ministrators to exploit all options available for a given environment in order to lessen the impact of buffer cache. For instance, some systems allow to configure the strategy under which cache pages are kept, giving priority to local memory requirements and tossing buffer cache as needed.

8.4 ccNUMA issues with C++

Locality of memory access as shown above can often be implemented in languages like Fortran or C once the basic memory access patterns have been identified. Due to its object-oriented features, however, C++ is another matter entirely [C100, C101]. In this section we want to point out the most relevant pitfalls when using OpenMP-parallel C++ code on ccNUMA systems.

8.4.1 Arrays of objects

The most basic problem appears when allocating an array of objects of type `D` using the standard `new[]` operator. For simplicity, we choose `D` to be a simple wrapper around `double` with all the necessary overloaded operators to make it look and behave like the basic type:

```
1  class D {
2    double d;
3  public:
4    D(double _d=0.0) throw() : d(_d) {}
5    ~D() throw() {}
6    inline D& operator=(double _d) throw() {d=_d; return *this;}
7    friend D operator+(const D&, const D&) throw();
8    friend D operator*(const D&, const D&) throw();
9    ...
10 };
```

Assuming correct implementation of all operators, the only difference between `D` and `double` should be that instantiation of an object of type `D` leads to immediate initialization, which is not the case for `double`s, i.e., in `a=new D[N]`, memory allocation takes place as usual, but the default constructor gets called for each array member. Since `new` knows nothing about NUMA, these calls are all done by the thread executing `new`. As a consequence, all the data ends up in that thread's local memory. One way around this would be a default constructor that does not touch the member, but this is not always possible or desirable.

One should thus first map the memory pages that will be used for the array data to the correct nodes so that access becomes local for each thread, and then call the constructors to initialize the objects. This could be accomplished by *placement new*, where the number of objects to be constructed as well as the exact memory (base) address of their instantiation is specified. Placement `new` does not call any constructors, though. A simple way around the effort of using placement `new` is to overload

`D::operator new[]`. This operator has the sole responsibility to allocate "raw" memory. An overloaded version can, in addition to memory allocation, initialize the pages in parallel for good NUMA placement (we ignore the requirement to throw `std::bad_alloc` on failure):

```
1  void* D::operator new[](size_t n) {
2    char *p = new char[n];       // allocate
3    size_t i,j;
4  #pragma omp parallel for private(j) schedule(runtime)
5    for(i=0; i<n; i += sizeof(D))
6      for(j=0; j<sizeof(D); ++j)
7        p[i+j] = 0;
8    return p;
9  }
10
11 void D::operator delete[](void* p) throw() {
12  delete [] static_cast<char*>p;
13 }
```

Construction of all objects in an array at their correct positions is then done automatically by the C++ runtime, using placement new. Note that the C++ runtime usually requests a little more space than would be needed by the aggregated object sizes, which is used for storing administrative information alongside the actual data. Since the amount of data is small compared to NUMA-relevant array sizes, there is no noticeable effect.

Overloading `operator new[]` works for simple cases like `class D` above. Dynamic members are problematic, however, because their NUMA locality cannot be easily controlled:

```
1  class E {
2    size_t s;
3    std::vector<double> *v;
4  public:
5    E(size_t _s=100) : s(_s), v(new std::vector<double>(s)) {}
6    ~E() { delete [] v; }
7    ...
8  };
```

`E`'s constructor initializes `E::s` and `E::v`, and these would be the only data items subject to NUMA placement by an overloaded `E::operator new[]` upon construction of an array of `E`. The memory addressed by `E::v` is not handled by this mechanism; in fact, the `std::vector<double>` is preset upon construction inside STL using copies of the object `double()`. This happens in the C++ runtime after `E::operator new[]` was executed. All the memory will be mapped into a single locality domain.

Avoiding this situation is hardly possible with standard C++ and STL constructs if one insists on constructing arrays of objects with `new[]`. The best advice is to call object constructors explicitly in a loop and to use a vector for holding pointers only:

```
1    std::vector<E*> v_E(n);
2
```

```
3 #pragma omp parallel for schedule(runtime)
4   for(size_t i=0; i<v_E.size(); ++i) {
5     v_E[i] = new E(100);
6   }
```

Since now the class constructor is called from different threads concurrently, it must be thread safe.

8.4.2 Standard Template Library

C-style array handling as shown in the previous section is certainly discouraged for C++; the STL `std::vector<>` container is much safer and more convenient, but has its own problems with ccNUMA page placement. Even for simple data types like `double`, which have a trivial default constructor, placement is problematic since, e.g., the allocated memory in a `std::vector<>(int)` object is filled with copies of `value_type()` using `std::uninitialized_fill()`. The design of a dedicated NUMA-aware container class would probably allow for more advanced optimizations, but STL defines a customizable abstraction layer called *allocators* that can effectively encapsulate the low-level details of a container's memory management. By using this facility, correct NUMA placement can be enforced in many cases for `std::vector<>` with minimal changes to an existing program code.

STL containers have an optional template argument by which one can specify the allocator class to use [C102, C103]. By default, this is `std::allocator<T>`. An allocator class provides, among others, the methods (class namespace omitted):

```
1 pointer allocate(size_type, const void *=0);
2 void deallocate(pointer, size_type);
```

Here `size_type` is `size_t`, and `pointer` is `T*`. The `allocate()` method gets called by the container's constructor to set up memory in much the same way as `operator new[]` for an array of objects. However, since all relevant supplementary information is stored in additional member variables, the number of bytes to allocate matches the space required by the container's contents only, at least on initial construction (see below). The second parameter to `allocate()` can supply additional information to the allocator, but its semantics are not standardized. `deallocate()` is responsible for freeing the allocated memory again.

The simplest NUMA-aware allocator would take care that `allocate()` not only allocates memory but initializes it in parallel. For reference, Listing 8.1 shows the code of a simple NUMA-friendly allocator, using standard `malloc()` for allocation. In line 19 the OpenMP API function `omp_in_parallel()` is used to determine whether the allocator was called from an active parallel region. If it was, the initialization loop is skipped. To use the template, it must be specified as the second template argument whenever a `std::vector<>` object is constructed:

```
1   vector<double, NUMA_Allocator<double> > v(length);
```

Listing 8.1: A NUMA allocator template. The implementation is somewhat simplified from the requirements in the C++ standard.

```
template <class T> class NUMA_Allocator {
public:
  typedef T* pointer;
  typedef const T* const_pointer;
  typedef T& reference;
  typedef const T& const_reference;
  typedef size_t size_type;
  typedef T value_type;

  NUMA_Allocator() { }
  NUMA_Allocator(const NUMA_Allocator& _r) { }
  ~NUMA_Allocator() { }

  // allocate raw memory including page placement
  pointer allocate(size_type numObjects,
                   const void *localityHint=0) {
    size_type len = numObjects * sizeof(value_type);
    char *p = static_cast<char*>(std::malloc(len));
    if(!omp_in_parallel()) {
#pragma omp parallel for schedule(runtime) private(ofs)
      for(size_type i=0; i<len; i+=sizeof(value_type)) {
        for(size_type j=0; j<sizeof(value_type); ++j) {
          p[i+j]=0;
      }
    }
    return static_cast<pointer>(m);
  }

  // free raw memory
  void deallocate(pointer ptrToMemory, size_type numObjects) {
    std::free(ptrToMemory);
  }

  // construct object at given address
  void construct(pointer p, const value_type& x) {
    new(p) value_type(x);
  }

  // destroy object at given address
  void destroy(pointer p) {
    p-> value_type();
  }

private:
  void operator=(const NUMA_Allocator&) {}
};
```

What follows after memory allocation is pretty similar to the array-of-objects case, and has the same restrictions: The allocator's `construct()` method is called For each of the objects, and uses placement new to construct each object at the correct address (line 36). Upon destruction, each object's destructor is called explicitly (one of the rare cases where this is necessary) via the `destroy()` method in line 41. Note that container construction and destruction are not the only places where `construct()` and `destroy()` are invoked, and that there are many things which could destroy NUMA locality immediately. For instance, due to the concept of container size versus capacity, calling `std::vector<>::push_back()` just once on a "filled" container reallocates all memory plus a significant amount more, and copies the original objects to their new locations. The NUMA allocator will perform first-touch placement, but it will do so using the container's new capacity, not its size. As a consequence, placement will almost certainly be suboptimal. One should keep in mind that not all the functionality of `std::vector<>` is suitable to use in a ccNUMA environment. We are not even talking about the other STL containers (`deque`, `list`, `map`, `set`, etc.).

Incidentally, standard-compliant allocator objects of the same type must always compare as equal [C102]:

```
1 template <class T>
2 inline bool operator==(const NUMA_Allocator<T>&,
3                         const NUMA_Allocator<T>&) { return true; }
4 template <class T>
5 inline bool operator!=(const NUMA_Allocator<T>&,
6                         const NUMA_Allocator<T>&) { return false; }
```

This has the important consequence that an allocator object is necessarily stateless, ruling out some optimizations one may think of. A template specialization for `T=void` must also be provided (not shown here). These and other peculiarities are discussed in the literature. More sophisticated strategies than using plain `malloc()` do of course exist.

In summary we must add that the methods shown here are useful for outfitting existing C++ programs with *some* ccNUMA awareness without too much hassle. Certainly a newly designed code should be parallelized with ccNUMA in mind from the start.

Problems

For solutions see page 303 *ff*.

8.1 *Dynamic scheduling and ccNUMA*. When a memory-bound, OpenMP-parallel code runs on all sockets of a ccNUMA system, one should use static scheduling and initialize the data in parallel to make sure that memory accesses are mostly local. We want to analyze what happens if static scheduling is not a option, e.g., for load balancing reasons.

For a system with two locality domains, calculate the expected performance impact of dynamic scheduling on a memory-bound parallel loop. Assume for simplicity that there is exactly one thread (core) running per LD. This thread is able to saturate the local or any remote memory bus with some performance p. The inter-LD network should be infinitely fast, i.e., there is no penalty for non-local transfers and no contention effects on the inter-LD link. Further assume that all pages are homogeneously distributed throughout the system and that dynamic scheduling is purely statistical (i.e., each thread accesses all LDs in a random manner, with equal probability). Finally, assume that the chunksize is large enough so that there are no bad effects from hardware prefetching or partial cache line use.

The code's performance with static scheduling and perfect load balance would be $2p$. What is the expected performance under dynamic scheduling (also with perfect load balance)?

8.2 *Unfortunate chunksizes*. What could be possible reasons for the performance breakdown at chunksizes between 16 and 256 for the parallel vector triad on a four-LD ccNUMA machine (Figure 8.7)? Hint: Memory pages play a decisive role here.

8.3 *Speeding up "small" jobs*. If a ccNUMA system is sparsely utilized, e.g., if there are less threads than locality domains, and they all execute (memory-bound) code, is the first touch policy still the best strategy for page placement?

8.4 *Triangular matrix-vector multiplication*. Parallelize a triangular matrix-vector multiplication using OpenMP:

```
1 do r=1,N
2   do c=1,r
3     y(r) = y(r) + a(c,r) * x(c)
4   enddo
5 enddo
```

What is the central parallel performance issue here? How can it be solved in general, and what special precautions are necessary on ccNUMA systems? You may ignore the standard scalar optimizations (unrolling, blocking).

8.5 *NUMA placement by overloading*. In Section 8.4.1 we enforced NUMA placement for arrays of objects of type `D` by overloading `D::operator new[]`. A similar thing was done in the NUMA-aware allocator class (Listing 8.1). Why did we use a loop nest for memory initialization instead of a single loop over `i`?

Chapter 9

Distributed-memory parallel programming with MPI

Ever since parallel computers hit the HPC market, there was an intense discussion about what should be an appropriate programming model for them. The use of explicit *message passing* (MP), i.e., communication between processes, is surely the most tedious and complicated but also the most flexible parallelization method. Parallel computer vendors recognized the wish for efficient message-passing facilities, and provided proprietary, i.e., nonportable libraries up until the early 1990s. At that point in time it was clear that a joint standardization effort was required to enable scientific users to write parallel programs that were easily portable between platforms. The result of this effort was MPI, the Message Passing Interface. Today, the MPI standard is supported by several free and commercial implementations [W125, W126, W127], and has been extended several times. It contains not only communication routines, but also facilities for efficient parallel I/O (if supported by the underlying hardware). An MPI library is regarded as a necessary ingredient in any HPC system installation, and numerous types of interconnect are supported.

The current MPI standard in version 2.2 (to which we always refer in this book) defines over 500 functions, and it is beyond the scope of this book to even try to cover them all. In this chapter we will concentrate on the important concepts of message passing and MPI in particular, and provide some knowledge that will enable the reader to consult more advanced textbooks [P13, P14] or the standard document itself [W128, P15].

9.1 Message passing

Message passing is required if a parallel computer is of the distributed-memory type, i.e., if there is no way for one processor to directly access the address space of another. However, it can also be regarded as a *programming model* and used on shared-memory or hybrid systems as well (see Chapter 4 for a categorization). MPI, the nowadays dominating message-passing standard, conforms to the following rules:

- The same program runs on all processes (Single Program Multiple Data, or *SPMD*). This is no restriction compared to the more general *MPMD* (Multiple Program Multiple Data) model as all processes taking part in a parallel calcu-

lation can be distinguished by a unique identifier called *rank* (see below). Most modern MPI implementations allow starting different binaries in different processes, however. An MPMD-style message passing library is *PVM*, the Parallel Virtual Machine [P16]. Since it has waned in importance in recent years, it will not be covered here.

- The program is written in a sequential language like Fortran, C or C++. Data exchange, i.e., sending and receiving of messages, is done via calls to an appropriate library.

- All variables in a process are local to this process. There is no concept of shared memory.

One should add that message passing is not the only possible programming paradigm for distributed-memory machines. Specialized languages like High Performance Fortran (HPF), Co-Array Fortran (CAF) [P17], Unified Parallel C (UPC) [P18], etc., have been created with support for distributed-memory parallelization built in, but they have not developed a broad user community and it is as yet unclear whether those approaches can match the efficiency of MPI.

In a message passing program, messages carry data between processes. Those processes could be running on separate compute nodes, or different cores inside a node, or even on the same processor core, time-sharing its resources. A message can be as simple as a single item (like a DP word) or even a complicated structure, perhaps scattered all over the address space. For a message to be transmitted in an orderly manner, some parameters have to be fixed in advance:

- Which process is sending the message?

- Where is the data on the sending process?

- What kind of data is being sent?

- How much data is there?

- Which process is going to receive the message?

- Where should the data be left on the receiving process?

- What amount of data is the receiving process prepared to accept?

All MPI calls that actually transfer data have to specify those parameters in some way. Note that above parameters strictly relate to point-to-point communication, where there is always exactly one sender and one receiver. As we will see, MPI supports much more than just sending a single message between two processes; there is a similar set of parameters for those more complex cases as well.

MPI is a very broad standard with a huge number of library routines. Fortunately, most applications merely require less than a dozen of those.

Listing 9.1: A very simple, fully functional "Hello World" MPI program in Fortran 90.

```
1  program mpitest
2
3  use MPI
4
5  integer :: rank, size, ierror
6
7  call MPI_Init(ierror)
8  call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
9  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)
10
11 write(*,*) 'Hello World, I am ',rank,' of ',size
12
13 call MPI_Finalize(ierror)
14 end
```

9.2 A short introduction to MPI

9.2.1 A simple example

MPI is always available as a library. In order to compile and link an MPI program, compilers and linkers need options that specify where include files (i.e., C headers and Fortran modules) and libraries can be found. As there is considerable variation in those locations among installations, most MPI implementations provide compiler wrapper scripts (often called `mpicc`, `mpif77`, etc.), which supply the required options automatically but otherwise behave like "normal" compilers. Note that the way that MPI programs should be compiled and started is not fixed by the standard, so please consult the system documentation by all means.

Listing 9.1 shows a simple "Hello World"-type MPI program in Fortran 90. (See Listing 9.2 for a C version. We will mostly stick to the Fortran MPI bindings, and only describe the differences to C where appropriate. Although there are C++ bindings defined by the standard, they are of limited usefulness and will thus not be covered here. In fact, they are deprecated as of MPI 2.2.) In line 3, the `MPI` module is loaded, which provides required globals and definitions (the preprocessor is used to read in the `mpi.h` header in C; there is an equivalent header file for Fortran 77 called `mpif.h`). All Fortran MPI calls take an `INTENT(OUT)` argument, here called `ierror`, which transports information about the success of the MPI operation to the user code, a value of `MPI_SUCCESS` meaning that there were no errors. In C, the return code is used for that, and the `ierror` argument does not exist. Since failure resiliency is not built into the MPI standard today and checkpoint/restart features are usually implemented by the user code anyway, the error code is rarely used at all in practice.

The first statement, apart from variable declarations, in any MPI code should be

Listing 9.2: A very simple, fully functional "Hello World" MPI program in C.

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char** argv) {
5    int rank, size;
6
7    MPI_Init(&argc, &argv);
8    MPI_Comm_size(MPI_COMM_WORLD, &size);
9    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10
11   printf("Hello World, I am %d of %d\n", rank, size);
12
13   MPI_Finalize();
14   return 0;
15 }
```

a call to `MPI_Init()`. This initializes the parallel environment (line 7). If thread parallelism of any kind is used together with MPI, calling `MPI_Init()` is not sufficient. See Section 11 for details.

The MPI bindings for the C language follow the case-sensitive name pattern `MPI_Xxxxx...`, while Fortran is case-insensitive, of course. In contrast to Fortran, the C binding for `MPI_Init()` takes pointers to the `main()` function's arguments so that the library can evaluate and remove any additional command line arguments that may have been added by the MPI startup process.

Upon initialization, MPI sets up the so-called *world communicator*, which is called `MPI_COMM_WORLD`. A communicator defines a group of MPI processes that can be referred to by a communicator *handle*. The `MPI_COMM_WORLD` handle describes all processes that have been started as part of the parallel program. If required, other communicators can be defined as subsets of `MPI_COMM_WORLD`. Nearly all MPI calls require a communicator as an argument.

The calls to `MPI_Comm_size()` and `MPI_Comm_rank()` in lines 8 and 9 serve to determine the number of processes (`size`) in the parallel program and the unique identifier (`rank`) of the calling process, respectively. Note that the C bindings require output arguments (like `rank` and `size` above) to be specified as pointers. The ranks in a communicator, in this case `MPI_COMM_WORLD`, are consecutive, starting from zero. In line 13, the parallel program is shut down by a call to `MPI_Finalize()`. Note that no MPI process except rank 0 is guaranteed to execute any code beyond `MPI_Finalize()`.

In order to compile and run the source code in Listing 9.1, a "common" implementation may require the following steps:

```
1 $ mpif90 -O3 -o hello.exe hello.F90
2 $ mpirun -np 4 ./hello.exe
```

This would compile the code and start it with four processes. Be aware that pro-

MPI type	Fortran type
MPI_CHAR	CHARACTER(1)
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_BYTE	N/A

Table 9.1: Standard MPI data types for Fortran.

cessors may have to be allocated from some resource management (batch) system before parallel programs can be launched. How exactly MPI processes are started is entirely up to the implementation. Ideally, the start mechanism uses the resource manager's infrastructure (e.g., daemons running on all nodes) to launch processes. The same is true for process-to-core affinity; if the MPI implementation provides no direct facilities for affinity control, the methods described in Appendix A may be employed.

The output of this program could look as follows:

```
1 Hello World, I am 3 of 4
2 Hello World, I am 0 of 4
3 Hello World, I am 2 of 4
4 Hello World, I am 1 of 4
```

Although the `stdout` and `stderr` streams of MPI programs are usually redirected to the terminal where the program was started, the order in which outputs from different ranks will arrive there is undefined if the ordering is not enforced by other means.

9.2.2 Messages and point-to-point communication

The "Hello World" example did not contain any real communication apart from starting and stopping processes. An MPI message is defined as an array of elements of a particular MPI data type. Data types can either be basic types (corresponding to the standard types that every programming language knows) or *derived types*, which must be defined by appropriate MPI calls. The reason why MPI needs to know the data types of messages is that it supports heterogeneous environments where it may be necessary to do on-the-fly data conversions. For any message transfer to proceed, the data types on sender and receiver sides must match. See Tables 9.1 and 9.2 for nonexhaustive lists of available MPI data types in Fortran and C, respectively.

If there is exactly one sender and one receiver we speak of *point-to-point communication*. Both ends are identified uniquely by their ranks. Each point-to-point message can carry an additional integer label, the so-called *tag*, which may be used to identify the type of a message, and which must match on both ends. It may carry

MPI type	C type
MPI_CHAR	signed char
MPI_INT	signed int
MPI_LONG	signed long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_BYTE	N/A

Table 9.2: A selection of the standard MPI data types for C. Unsigned variants exist where applicable.

any accompanying information, or just be set to some constant value if it is not needed. The basic MPI function to send a message from one process to another is `MPI_Send()`:

```
1 <type> buf(*)
2 integer :: count, datatype, dest, tag, comm, ierror
3 call MPI_Send(buf,      ! message buffer
4               count,    ! # of items
5               datatype, ! MPI data type
6               dest,     ! destination rank
7               tag,      ! message tag (additional label)
8               comm,     ! communicator
9               ierror)   ! return value
```

The data type of the message buffer may vary; the MPI interfaces and prototypes declared in modules and headers accommodate this.[1] A message may be received with the `MPI_Recv()` function:

```
1 <type> buf(*)
2 integer :: count, datatype, source, tag, comm,
3 integer :: status(MPI_STATUS_SIZE), ierror
4 call MPI_Recv(buf,      ! message buffer
5               count,    ! maximum # of items
6               datatype, ! MPI data type
7               source,   ! source rank
8               tag,      ! message tag (additional label)
9               comm,     ! communicator
10              status,   ! status object (MPI_Status* in C)
11              ierror)   ! return value
```

Compared with `MPI_Send()`, this function has an additional output argument, the `status` object. After `MPI_Recv()` has returned, the `status` object can be used to determine parameters that have not been fixed by the call's arguments. Primarily, this pertains to the length of the message, because the `count` parameter is

[1] While this is no problem in C/C++, where the `void*` pointer type conveniently hides any variation in the argument type, the Fortran MPI bindings are explicitly inconsistent with the language standard. However, this can be tolerated in most cases. See the standard document [P15] for details.

only a maximum value at the receiver side; the message may be shorter than `count` elements. The `MPI_Get_count()` function can retrieve the real number:

```
integer :: status(MPI_STATUS_SIZE), datatype, count, ierror
call MPI_Get_count(status,    ! status object from MPI_Recv()
                   datatype,  ! MPI data type received
                   count,     ! count (output argument)
                   ierror)    ! return value
```

However, the `status` object also serves another purpose.The `source` and `tag` arguments of `MPI_Recv()` may be equipped with the special constants ("wildcards") `MPI_ANY_SOURCE` and `MPI_ANY_TAG`, respectively. The former specifies that the message may be sent by anyone, while the latter determines that the message tag should not matter. After `MPI_Recv()` has returned, `status(MPI_SOURCE)` and `status(MPI_TAG)` contain the sender's rank and the message tag, respectively. (In C, the `status` object is of type `struct MPI_Status`, and access to source and tag information works via the "`.`" operator.)

Note that `MPI_Send()` and `MPI_Recv()` have *blocking* semantics, meaning that the buffer can be used safely after the function returns (i.e., it can be modified after `MPI_Send()` without altering any message in flight, and one can be sure that the message has been completely received after `MPI_Recv()`). This is not to be confused with *synchronous* behavior; see below for details.

Listing 9.3 shows an MPI program fragment for computing an integral over some function `f(x)` in parallel. In contrast to the OpenMP version in Listing 6.2, the distribution of work among processes must be handled manually in MPI. Each MPI process gets assigned a subinterval of the integration domain according to its rank (lines 9 and 10), and some other function `integrate()`, which may look similar to Listing 6.2, can then perform the actual integration (line 13). After that each process holds its own partial result, which should be added to get the final integral. This is done at rank 0, who executes a loop over all ranks from 1 to `size` − 1 (lines 18–29), receiving the local integral from each rank in turn via `MPI_Recv()` (line 19) and accumulating the result in `res` (line 28). Each rank apart from 0 has to call `MPI_Send()` to transmit the data. Hence, there are `size` − 1 send and `size` − 1 matching receive operations. The data types on both sides are specified to be `MPI_DOUBLE_PRECISION`, which corresponds to the usual `double precision` type in Fortran (cf. Table 9.1). The message tag is not used here, so we set it to zero.

This simple program could be improved in several ways:

- MPI does not preserve the temporal order of messages unless they are transmitted between the same sender/receiver pair (and with the same tag). Hence, to allow the reception of partial results at rank 0 without delay due to different execution times of the `integrate()` function, it may be better to use the `MPI_ANY_SOURCE` wildcard instead of a definite source rank in line 23.

- Rank 0 does not call `MPI_Recv()` before returning from its own execution of `integrate()`. If other processes finish their tasks earlier, communication cannot proceed, and it cannot be overlapped with computation. The MPI

Listing 9.3: Program fragment for parallel integration in MPI.

```
integer, dimension(MPI_STATUS_SIZE) :: status
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

! integration limits
a=0.d0 ; b=2.d0 ; res=0.d0

! limits for "me"
mya=a+rank*(b-a)/size
myb=mya+(b-a)/size

! integrate f(x) over my own chunk - actual work
psum = integrate(mya,myb)

! rank 0 collects partial results
if(rank.eq.0) then
   res=psum
   do i=1,size-1
      call MPI_Recv(tmp, &   ! receive buffer
                    1,   &   ! array length
                         ! data type
                    MPI_DOUBLE_PRECISION,&
                    i,   &   ! rank of source
                    0,   &   ! tag (unused here)
                    MPI_COMM_WORLD,& ! communicator
                    status,& ! status array (msg info)
                    ierror)
      res=res+tmp
   enddo
   write(*,*) 'Result: ',res
! ranks != 0 send their results to rank 0
else
   call MPI_Send(psum,   &   ! send buffer
                 1,      &   ! message length
                 MPI_DOUBLE_PRECISION,&
                 0,      &   ! rank of destination
                 0,      &   ! tag (unused here)
                 MPI_COMM_WORLD,ierror)
endif
```

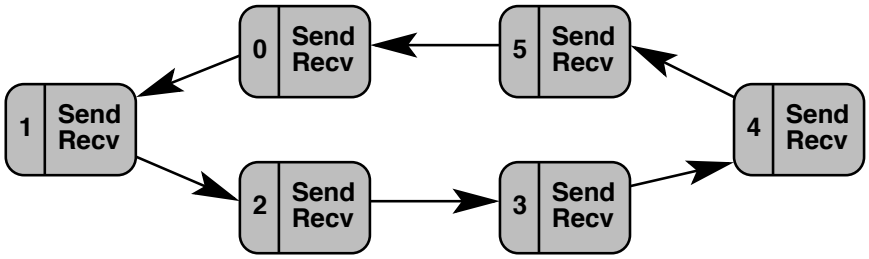


Figure 9.1: A ring shift communication pattern. If sends and receives are performed in the order shown, a deadlock can occur because `MPI_Send()` may be synchronous.

standard provides *nonblocking point-to-point communication* facilities that allow multiple outstanding receives (and sends), and even let implementations support asynchronous messages. See Section 9.2.4 for more information.

- Since the final result is needed at rank 0, this process is necessarily a communication bottleneck if the number of messages gets large. In Section 10.4.4 we will demonstrate optimizations that can significantly reduce communication overhead in those situations. Fortunately, nobody is required to write explicit code for this. In fact, the global sum is an example for a *reduction operation* and is well supported within MPI (see Section 9.2.3). Vendor implementations are assumed to provide optimized versions of such global operations.

While `MPI_Send()` is easy to use, one should be aware that the MPI standard allows for a considerable amount of freedom in its actual implementation. Internally it may work completely synchronously, meaning that the call can not return to the user code before a message transfer has at least started after a handshake with the receiver. However, it may also copy the message to an intermediate buffer and return right away, leaving the handshake and data transmission to another mechanism, like a background thread. It may even change its behavior depending on any explicit or hidden parameters. Apart from a possible performance impact, *deadlocks* may occur if the possible synchronousness of `MPI_Send()` is not taken into account. A typical communication pattern where this may become crucial is a "ring shift" (see Figure 9.1). All processes form a closed ring topology, and each *first* sends a message to its "left-hand" and *then* receives a message from its "right-hand" neighbor:

```
integer :: size, rank, left, right, ierror
integer, dimension(N) :: buf
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)
left = rank+1                ! left and right neighbors
right = rank-1
if(right<0)    right=size-1   ! close the ring
if(left>=size) left=0
call MPI_Send(buf, N, MPI_INTEGER, left, 0, &
                  MPI_COMM_WORLD,ierror)
call MPI_Recv(buf,N,MPI_INTEGER,right,0, &
                  MPI_COMM_WORLD,status,ierror)
```

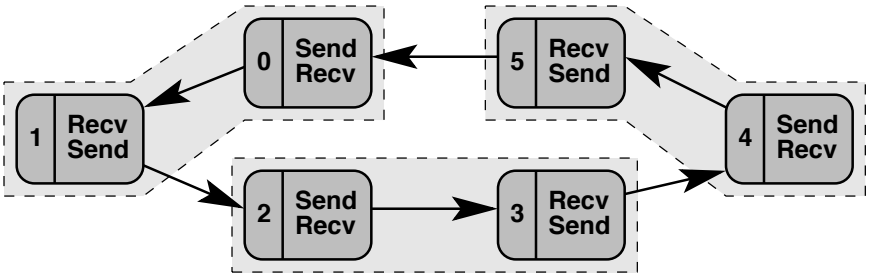


Figure 9.2: A possible solution for the deadlock problem with the ring shift: By changing the order of `MPI_Send()` and `MPI_Recv()` on all odd-numbered ranks, pairs of processes can communicate without deadlocks because there is now a matching receive for every send operation (dashed boxes).

If `MPI_Send()` is synchronous, all processes call it first and then wait forever until a matching receive gets posted. However, it may well be that the ring shift runs without problems if the messages are sufficiently short. In fact, most MPI implementations provide a (small) internal buffer for short messages and switch to synchronous mode when the buffer is full or too small (the situation is actually a little more complex in reality; see Sections 10.2 and 10.3 for details). This may lead to sporadic deadlocks, which are hard to spot. If there is some suspicion that a sporadic deadlock is triggered by `MPI_Send()` switching to synchronous mode, one can substitute all occurrences of `MPI_Send()` by `MPI_Ssend()`, which has the same interface but is synchronous by definition.

A simple solution to this deadlock problem is to interchange the `MPI_Send()` and `MPI_Recv()` calls on, e.g., all odd-numbered processes, so that there is a matching receive for every send executed (see Figure 9.2). Lines 9–12 in the code above should thus be replaced by:

```
1  if(MOD(rank,2)/=0) then
2    call MPI_Recv(buf,N,MPI_INTEGER,right,0, &    ! odd rank
3                  MPI_COMM_WORLD,status,ierror)
4    call MPI_Send(buf, N, MPI_INTEGER, left, 0, &
5                  MPI_COMM_WORLD,ierror)
6  else
7    call MPI_Send(buf, N, MPI_INTEGER, left, 0, & ! even rank
8                  MPI_COMM_WORLD,ierror)
9    call MPI_Recv(buf,N,MPI_INTEGER,right,0, &
10                 MPI_COMM_WORLD,status,ierror)
11 endif
```

After the messages sent by the even ranks have been transmitted, the remaining send/receive pairs can be matched as well. This solution does not exploit the full bandwidth of a nonblocking network, however, because only half the possible communication links can be active at any time (at least if `MPI_Send()` is really synchronous). A better alternative is the use of nonblocking communication. See Section 9.2.4 for more information, and Problem 9.1 for some more aspects of the ring shift pattern.

Since ring shifts and similar patterns are so ubiquitous, MPI has some direct support for them even with blocking communication. The `MPI_Sendrecv()` and `MPI_Sendrecv_replace()` routines combine the standard send and receive in one call, the latter using a single communication buffer in which the received message overwrites the data sent. Both routines are guaranteed to not be subject to the deadlock effects that occur with separate send and receive.

Finally we should add that there is also a blocking send routine that is guaranteed to return to the user code, regardless of the state of the receiver (`MPI_Bsend()`). However, the user must explicitly provide sufficient buffer space at the sender. It is rarely employed in practice because nonblocking communication is much easier to use (see Section 9.2.4).

9.2.3 Collective communication

The accumulation of partial results as shown above is an example for a *reduction* operation, performed on all processes in the communicator. Reductions have been introduced already with OpenMP (see Section 6.1.5), where they have the same purpose. MPI, too, has mechanisms that make reductions much simpler and in most cases more efficient than looping over all ranks and collecting results. Since a reduction is a procedure which all ranks in a communicator participate in, it belongs to the so-called *collective*, or *global communication* operations in MPI. Collective communication, as opposed to point-to-point communication, requires that every rank calls the same routine, so it is impossible for a point-to-point message sent via, e.g., `MPI_Send()`, to match a receive that was initiated using a collective call.

The simplest collective in MPI, and one that does not actually perform any real data transfer, is the barrier:

```
integer :: comm, ierror
call MPI_Barrier(comm,    ! communicator
                 ierror)  ! return value
```

The barrier *synchronizes* the members of the communicator, i.e., all processes must call it before they are allowed to return to the user code. Although frequently used by beginners, the importance of the barrier in MPI is generally overrated, because other MPI routines allow for implicit or explicit synchronization with finer control. It is sometimes used, though, for debugging or profiling.

A more useful collective is the *broadcast*. It sends a message from one process (the "root") to all others in the communicator:

```
<type> buf(*)
integer :: count, datatype, root, comm, ierror
call MPI_Bcast(buffer,    ! send/receive buffer
               count,     ! message length
               datatype,  ! MPI data type
               root,      ! rank of root process
               comm,      ! communicator
               ierror)    ! return value
```

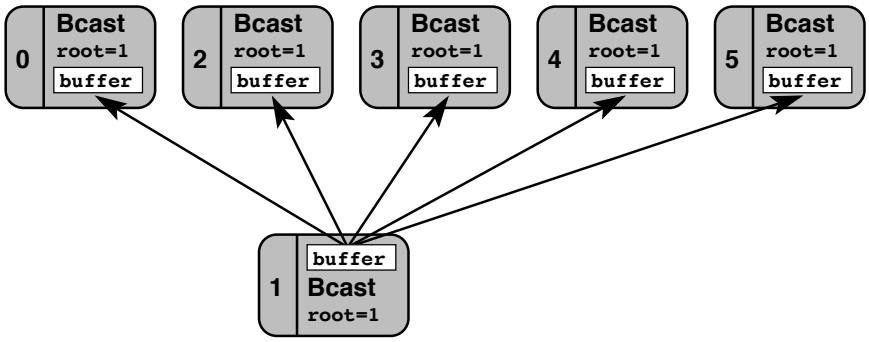


Figure 9.3: An MPI broadcast: The "root" process (rank 1 in this example) sends the same message to all others. Every rank in the communicator must call `MPI_Bcast()` with the same `root` argument.

The concept of a "root" rank, at which some general data source or sink is located, is common to many collective routines. Although rank 0 is a natural choice for "root," it is in no way different from other ranks. The `buffer` argument to `MPI_Bcast()` is a send buffer on the root and a receive buffer on any other process (see Figure 9.3). As already mentioned, every process in the communicator must call the routine, and of course the `root` argument to all those calls must be the same. A broadcast is needed whenever one rank has information that it must share with all others; e.g., there may be one process that performs some initialization phase after the program has started, like reading parameter files or command line options. This data can then be communicated to everyone else via `MPI_Bcast()`.

There are a number of more advanced collective calls that are concerned with global data distribution: `MPI_Gather()` collects the send buffer contents of all processes and concatenates them in rank order into the receive buffer of the root process. `MPI_Scatter()` does the reverse, distributing equal-sized chunks of the root's send buffer. Both exist in variants (with a "v" appended to their names) that support arbitrary per-rank chunk sizes. `MPI_Allgather()` is a combination of `MPI_Gather()` and `MPI_Bcast()`. See Table 9.3 for more examples.

Coming back to the integration example above, we had stated that there is a more effective method to perform the global reduction. This is the `MPI_Reduce()` function:

```
1  <type> sendbuf(*), recvbuf(*)
2  integer :: count, datatype, op, root, comm, ierror
3  call MPI_Reduce(sendbuf,    ! send buffer
4                  recvbuf,    ! receive buffer
5                  count,      ! number of elements
6                  datatype,   ! MPI data type
7                  op,         ! MPI reduction operator
8                  root,       ! root rank
9                  comm,       ! communicator
10                 ierror)     ! return value
```

`MPI_Reduce()` combines the contents of the `sendbuf` array on all processes,

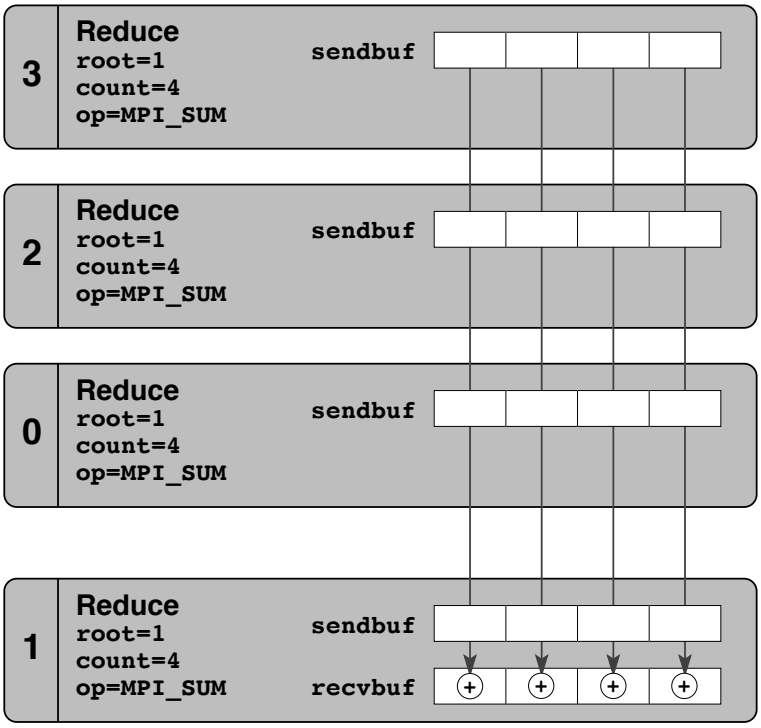


Figure 9.4: A reduction on an array of length `count` (a sum in this example) is performed by `MPI_Reduce()`. Every process must provide a send buffer. The receive buffer argument is only used on the root process. The local copy on root can be prevented by specifying `MPI_IN_PLACE` instead of a send buffer address.

element-wise, using an operator encoded by the `op` argument, and stores the result in `recvbuf` on root (see Figure 9.4). There are twelve predefined operators, the most important being `MPI_MAX`, `MPI_MIN`, `MPI_SUM` and `MPI_PROD`, which implement the global maximum, minimum, sum, and product, respectively. User-defined operators are also supported.

Now it is clear that the whole `if ...else ...endif` construct between lines 16 and 39 in Listing 9.3 (apart from printing the result in line 30) could have been written as follows:

```
1 call MPI_Reduce(psum, &          ! send buffer (partial result)
2                 res,  &          ! recv buffer (final result @ root)
3                 1, &             ! array length
4                 MPI_DOUBLE_PRECISION, &
5                 MPI_SUM, &       ! type of operation
6                 0, &             ! root (accumulate result there)
7                 MPI_COMM_WORLD,ierror)
```

Although a receive buffer (the `res` variable here) must be specified on all ranks, it is only relevant (and used) on root. Note that `MPI_Reduce()` in its plain form requires separate send and receive buffers on the root process. If allowed by the program semantics, the local accumulation on root can be simplified by setting the `sendbuf` argument to the special constant `MPI_IN_PLACE`. `recvbuf` is then used as the send buffer and gets overwritten with the global result. This can be good for performance if `count` is large and the additional copy operation leads to significant overhead. The behavior of the call on all nonroot processes is unchanged.

There are a few more global operations related to `MPI_Reduce()` worth noting.

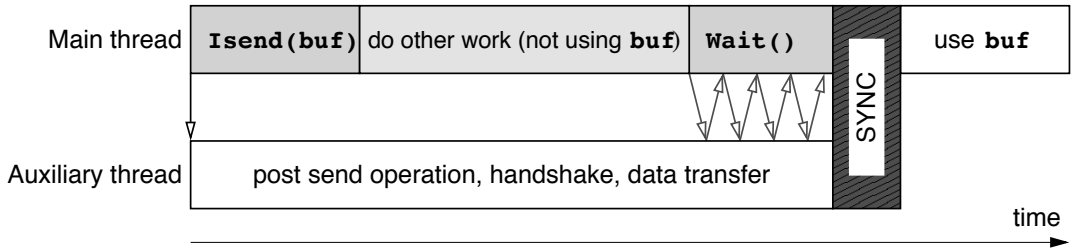


Figure 9.5: Abstract timeline view of a nonblocking send (`MPI_Isend()`). Whether there is actually an auxiliary thread is not specified by the standard; the whole data transfer may take place during `MPI_Wait()` or any other MPI function.

For example, `MPI_Allreduce()` is a fusion of a reduction with a broadcast, and `MPI_Reduce_scatter()` combines `MPI_Reduce()` with `MPI_Scatter()`.

Note that collectives are not required to, but still may synchronize all processes (the barrier is synchronizing by definition, of course). They are thus prone to similar deadlock hazards as blocking point-to-point communication (see above). This means, e.g., that collectives must be executed by all processes in the same order. See the MPI standard document [P15] for examples.

In general it is a good idea to prefer collectives over point-to-point constructs or combinations of simpler collectives that "emulate" the same semantics (see also Figures 10.15 and 10.16 and the corresponding discussion in Section 10.4.4). Good MPI implementations are optimized for data flow on collective communication and (should) also have some knowledge about network topology built in.

9.2.4 Nonblocking point-to-point communication

All MPI functionalities described so far have the property that the call returns to the user program only after the message transfer has progressed far enough so that the send/receive buffer can be used without problems. This means that, received data has arrived completely and sent data has left the buffer so that it can be safely modified without inadvertently changing the message. In MPI terminology, this is called *blocking communication*. Although collective communication in MPI is always blocking in the current MPI standard (version 2.2 at the time of writing), point-to-point communication can be performed with *nonblocking* semantics as well. A nonblocking point-to-point call merely initiates a message transmission and returns very quickly to the user code. In an efficient implementation, waiting for data to arrive and the actual data transfer occur in the background, leaving resources free for computation. Synchronization is ruled out (see Figure 9.5 for a possible timeline of events for the nonblocking `MPI_Isend()` call). In other words, nonblocking MPI is a way in which communication may be overlapped with computation if implemented efficiently. The message buffer must not be used as long as the user program has not been notified that it is safe to do so (which can be checked by suitable MPI calls). Nonblocking and blocking MPI calls are mutually compatible, i.e., a message sent via a blocking send can be matched by a nonblocking receive.

The most important nonblocking send is `MPI_Isend()`:

```
<type> buf(*)
integer :: count, datatype, dest, tag, comm, request, ierror
call MPI_Isend(buf,         ! message buffer
               count,       ! # of items
               datatype,    ! MPI data type
               dest,        ! destination rank
               tag,         ! message tag
               comm,        ! communicator
               request,     ! request handle (MPI_Request* in C)
               ierror)      ! return value
```

As opposed to the blocking send (see page 208), `MPI_Isend()` has an additional output argument, the *request handle*. It serves as an identifier by which the program can later refer to the "pending" communication request (in C, it is of type `struct MPI_Request`). Correspondingly, `MPI_Irecv()` initiates a nonblocking receive:

```
<type> buf(*)
integer :: count, datatype, source, tag, comm, request, ierror
call MPI_Irecv(buf,         ! message buffer
               count,       ! # of items
               datatype,    ! MPI data type
               source,      ! source rank
               tag,         ! message tag
               comm,        ! communicator
               request,     ! request handle
               ierror)      ! return value
```

The `status` object known from `MPI_Recv()` is missing here, because it is not needed; after all, no actual communication has taken place when the call returns to the user code. Checking a pending communication for completion can be done via the `MPI_Test()` and `MPI_Wait()` functions. The former only tests for completion and returns a flag, while the latter blocks until the buffer can be used:

```
logical :: flag
integer :: request, status(MPI_STATUS_SIZE), ierror
call MPI_Test(request,      ! pending request handle
              flag,         ! true if request complete (int* in C)
              status,       ! status object
              ierror)       ! return value
call MPI_Wait(request,      ! pending request handle
              status,       ! status object
              ierror)       ! return value
```

The `status` object contains useful information only if the pending communication is a completed receive (i.e., in the case of `MPI_Test()` the value of `flag` must be `true`). In this sense, the sequence

```
call MPI_Irecv(buf, count, datatype, source, tag, comm, &
               request, ierror)
call MPI_Wait(request, status, ierror)
```

is completely equivalent to a standard `MPI_Recv()`.

A potential problem with nonblocking MPI is that a compiler has no way to know that `MPI_Wait()` can (and usually will) modify the contents of `buf`. Hence, in the following code, the compiler may consider it legal to move the final statement in line 3 before the call to `MPI_Wait()`:

```
1 call MPI_Irecv(buf, ..., request, ...)
2 call MPI_Wait(request, status, ...)
3 buf(1) = buf(1) + 1
```

This will certainly lead to a race condition and the contents of `buf` may be wrong. The inherent connection between the `MPI_Irecv()` and `MPI_Wait()` calls, mediated by the request handle, is invisible to the compiler, and the fact that `buf` is not contained in the argument list of `MPI_Wait()` is sufficient to assume that the code modification is legal. A simple way to avoid this situation is to put the variable (or buffer) into a `COMMON` block, so that potentially all subroutines may modify it. See the MPI standard [P15] for alternatives.

Multiple requests can be pending at any time, which is another great advantage of nonblocking communication. Sometimes a group of requests belongs together in some respect, and one would like to check not one, but any one, any number, or all of them for completion. This can be done with suitable calls that are parameterized with an array of handles. As an example we choose the `MPI_Waitall()` routine:

```
1 integer :: count, requests(*)
2 integer :: statuses(MPI_STATUS_SIZE,*), ierror
3 call MPI_Waitall(count,          ! number of requests
4                  requests,       ! request handle array
5                  statuses,       ! statuses array (MPI_Status* in C)
6                  ierror)         ! return value
```

This call returns only after all the pending requests have been completed. The status objects are available in `array_of_statuses(:,:)`.

The integration example in Listing 9.3 can make use of nonblocking communication by overlapping the local interval integration on rank 0 with receiving results from the other ranks. Unfortunately, collectives cannot be used here because there are no nonblocking collectives in MPI. Listing 9.4 shows a possible solution. The reduction operation has to be done manually (lines 33–35), as in the original code. Array sizes for the status and request arrays are not known at compile time, hence those must be allocated dynamically, as well as separate receive buffers for all ranks except 0 (lines 11–13). The collection of partial results is performed with a single `MPI_Waitall()` in line 32. Nothing needs to be changed on the nonroot ranks; `MPI_Send()` is sufficient to communicate the partial results (line 39).

Nonblocking communication provides an obvious way to overlap communication, i.e., overhead, with useful work. The possible performance advantage, however, depends on many factors, and may even be nonexistent (see Section 10.4.3 for a discussion). But even if there is no real overlap, multiple outstanding nonblocking requests may improve performance because the MPI library can decide which of them gets serviced first.

Listing 9.4: Program fragment for parallel integration in MPI, using nonblocking point-to-point communication.

```
integer, allocatable, dimension(:,:) :: statuses
integer, allocatable, dimension(:) :: requests
double precision, allocatable, dimension(:) :: tmp
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

! integration limits
a=0.d0 ; b=2.d0 ; res=0.d0

if(rank.eq.0) then
  allocate(statuses(MPI_STATUS_SIZE, size-1))
  allocate(requests(size-1))
  allocate(tmp(size-1))
! pre-post nonblocking receives
  do i=1,size-1
    call MPI_Irecv(tmp(i), 1, MPI_DOUBLE_PRECISION, &
                   i, 0, MPI_COMM_WORLD,       &
                   requests(i), ierror)
  enddo
endif

! limits for "me"
mya=a+rank*(b-a)/size
myb=mya+(b-a)/size

! integrate f(x) over my own chunk - actual work
psum = integrate(mya,myb)

! rank 0 collects partial results
if(rank.eq.0) then
   res=psum
   call MPI_Waitall(size-1, requests, statuses, ierror)
   do i=1,size-1
      res=res+tmp(i)
   enddo
   write (*,*) 'Result: ',res
! ranks != 0 send their results to rank 0
else
   call MPI_Send(psum, 1, &
                 MPI_DOUBLE_PRECISION, 0, 0, &
                 MPI_COMM_WORLD,ierror)
endif
```

	Point-to-point	**Collective**

Blocking	`MPI_Send()`
	`MPI_Ssend()`
	`MPI_Bsend()`
	`MPI_Recv()`
		`MPI_Barrier()`
		`MPI_Bcast()`
		`MPI_Scatter()/`
		`MPI_Gather()`
		`MPI_Reduce()`
		`MPI_Reduce_scatter()`
		`MPI_Allreduce()`

Nonblocking	`MPI_Isend()`
	`MPI_Irecv()`
	`MPI_Wait()/MPI_Test()`
	`MPI_Waitany()/`
	`MPI_Testany()`
	`MPI_Waitsome()/`
	`MPI_Testsome()`
	`MPI_Waitall()/`
	`MPI_Testall()`
		N/A

Table 9.3: MPI's communication modes and a nonexhaustive overview of the corresponding subroutines.

Table 9.3 gives an overview of available communication modes in MPI, and the most important library functions.

9.2.5 Virtual topologies

We have outlined the principles of domain decomposition as an example for data parallelism in Section 5.2.1. Using the MPI functions covered so far, it is entirely possible to implement domain decomposition on distributed-memory parallel computers. However, setting up the process grid and keeping track of which ranks have to exchange halo data is nontrivial. Since domain decomposition is such an important pattern, MPI contains some functionality to support this recurring task in the form of *virtual topologies*. These provide a convenient process naming scheme, which fits the required communication pattern. Moreover, they potentially allow the MPI library to optimize communications by employing knowledge about network topology. Although arbitrary graph topologies can be described with MPI, we restrict ourselves to Cartesian topologies here.

As an example, assume there is a simulation that handles a big double precision array `P(1:3000,1:4000)` containing $3000 \times 4000 = 1.2 \times 10^7$ words. The simulation runs on $3 \times 4 = 12$ processes, across which the array is distributed "naturally," i.e., each process holds a chunk of size 1000×1000. Figure 9.6 shows a possible Cartesian topology that reflects this situation: Each process can either be identified by its rank or its Cartesian coordinates. It has a number of neighbors, which depends on

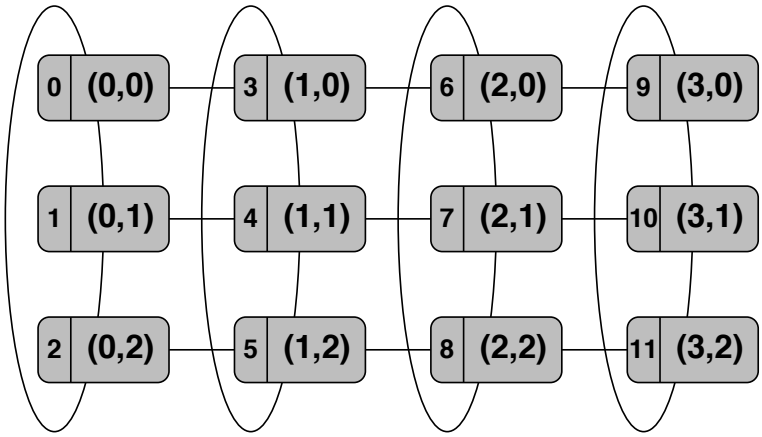


Figure 9.6: Two-dimensional Cartesian topology: 12 processes form a 3×4 grid, which is periodic in the second dimension but not in the first. The mapping between MPI ranks and Cartesian coordinates is shown.

the grid's dimensionality. In our example, the number of dimensions is two, which leads to at most four neighbors per process. Boundary conditions on each dimension can be closed (cyclic) or open.

MPI can help with establishing the mapping between ranks and Cartesian coordinates in the process grid. First of all, a new communicator must be defined to which the chosen topology is "attached." This is done via the `MPI_Cart_create()` function:

```
integer :: comm_old, ndims, dims(*), comm_cart, ierror
logical :: periods(*), reorder
call MPI_Cart_create(comm_old,     ! input communicator
                     ndims,        ! number of dimensions
                     dims,         ! # of processes in each dim.
                     periods,      ! periodicity per dimension
                     reorder,      ! true = allow rank reordering
                     comm_cart,    ! new cartesian communicator
                     ierror)       ! return value
```

It generates a new, "Cartesian" communicator `comm_cart`, which can be used later to refer to the topology. The `periods` array specifies which Cartesian directions are periodic, and the `reorder` parameter allows, if true, for rank reordering so that the rank of a process in communicators `comm_old` and `comm_cart` may differ. The MPI library may choose a different ordering by using its knowledge about network topology and anticipating that next-neighbor communication is often dominant in a Cartesian topology. Of course, communication between any two processes is still allowed in the Cartesian communicator.

There is no mention of the actual problem size (3000×4000) because it is entirely the user's job to care for data distribution. All MPI can do is keep track of the topology information. For the topology shown in Figure 9.6, `MPI_Cart_create()` could be called as follows:

```
call MPI_Cart_create(MPI_COMM_WORLD,        ! standard communicator
                     2,                     ! two dimensions
                     (/ 4, 3 /),            ! 4x3 grid
                     (/ .false., .true. /), ! open/periodic
                     .false.,               ! no rank reordering
```

```
6                       comm_cart,              ! Cartesian communicator
7                       ierror)
```

If the number of MPI processes is given, finding an "optimal" extension of the grid in each direction (as needed in the `dims` argument to `MPI_Cart_create()`) requires some arithmetic, which can be offloaded to the `MPI_Dims_create()` function:

```
1 integer :: nnodes, ndims, dims(*), ierror
2 call MPI_Dims_create(nnodes, ! number of nodes in grid
3                       ndims,  ! number of Cartesian dimensions
4                       dims,   ! input: /=0 # nodes fixed in this dir.
5                               !        ==0 # calculate # nodes
6                               ! output: number of nodes each dir.
7                       ierror)
```

The `dims` array is both an input and an output parameter: Each entry in `dims` corresponds to a Cartesian dimension. A zero entry denotes a dimension for which `MPI_Dims_create()` should calculate the number of processes, and a nonzero entry specifies a fixed number of processes. Under those constraints, the function determines a balanced distribution, with all `ndims` extensions as close together as possible. This is optimal in terms of communication overhead only if the overall problem grid is cubic. If this is not the case, the user is responsible for setting appropriate constraints, since MPI has no way to know the grid geometry.

Two service functions are responsible for the translation between Cartesian process coordinates and an MPI rank. `MPI_Cart_coords()` calculates the Cartesian coordinates for a given rank:

```
1 integer :: comm_cart, rank, maxdims, coords(*), ierror
2 call MPI_Cart_coords(comm_cart,    ! Cartesian communicator
3                       rank,        ! process rank in comm_cart
4                       maxdims,     ! length of coords array
5                       coords,      ! return Cartesian coordinates
6                       ierror)
```

(If rank reordering was allowed when producing `comm_cart`, a process should always obtain its rank by calling `MPI_Comm_rank(comm_cart,...)` first.) The output array `coords` contains the Cartesian coordinates belonging to the process of the specified rank.

This mapping function is needed whenever one deals with domain decomposition. The first information a process will obtain from MPI is its rank in the Cartesian communicator. `MPI_Cart_coords()` is then required to determine the coordinates so the process can calculate, e.g., which subdomain it should work on. See Section 9.3 below for an example.

The reverse mapping, i.e., from Cartesian coordinates to an MPI rank, is performed by `MPI_Cart_rank()`:

```
1 integer :: comm_cart, coords(*), rank, ierror
2 call MPI_Cart_rank(comm_cart,    ! Cartesian communicator
3                     coords,      ! Cartesian process coordinates
```

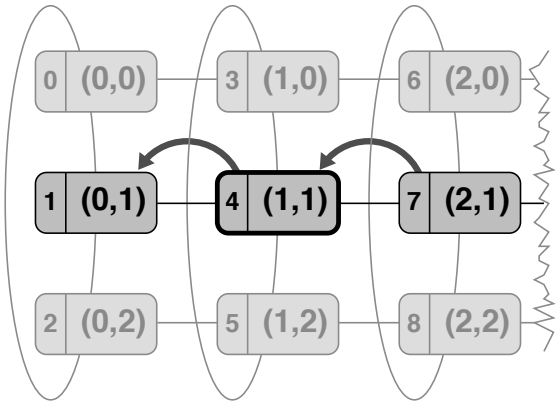


Figure 9.7: Example for the result of `MPI_Cart_shift()` on a part of the Cartesian topology from Figure 9.6. Executed by rank 4 with `direction`=0 and `disp`=−1, the function returns `rank_source`=7 and `rank_dest`=1.

```
4                       rank,          ! return process rank in comm_cart
5                       ierror)
```

Again, the return value in `rank` is only valid in the `comm_cart` communicator if reordering was allowed.

A regular task with domain decomposition is to find out who the next neighbors of a certain process are along a certain Cartesian dimension. In principle one could start from its Cartesian coordinates, offset one of them by one (accounting for open or closed boundary conditions) and map the result back to an MPI rank via `MPI_Cart_rank()`. The `MPI_Cart_shift()` function does it all in one step:

```
1 integer :: comm_cart, direction, disp, rank_source,
2 integer :: rank_dest, ierror
3 call MPI_Cart_shift(comm_cart,   ! Cartesian communicator
4                     direction,   ! direction of shift (0..ndims-1)
5                     disp,        ! displacement
6                     rank_source, ! return source rank
7                     rank_dest,   ! return destination rank
8                     ierror)
```

The `direction` parameter specifies within which Cartesian dimension the shift should be performed, and `disp` determines the distance and direction (positive or negative). `rank_source` and `rank_dest` return the "neighboring" ranks, according to the other arguments. Figure 9.7 shows an example for a shift along the negative first dimension, executed on rank 4 in the topology given in Figure 9.6. The source and destination neighbors are 7 and 1, respectively. If a neighbor does not exist because it would extend beyond the grid's boundary in a noncyclic dimension, the rank will be returned as the special value `MPI_PROC_NULL`. Using `MPI_PROC_NULL` as a source or destination rank in any communication call is allowed and will effectively render the call a dummy statement — no actual communication will take place. This can simplify programming because the boundaries of the grid do not have to be treated in a special way (see Section 9.3 for an example).

9.3 Example: MPI parallelization of a Jacobi solver

As a nontrivial example for virtual topologies and other MPI functionalities we use a simple Jacobi solver (see Sections 3.3 and 6.2) in three dimensions. As opposed to parallelization with OpenMP, where inserting a couple of directives was sufficient, MPI parallelization by domain decomposition is much more complex.

9.3.1 MPI implementation

Although the basic algorithm was described in Section 5.2.1, we require some more detail now. An annotated flowchart is shown in Figure 9.8. The central part is still the sweep over all subdomains (step 3); this is where the computational effort goes. However, each subdomain is handled by a different MPI process, which poses two difficulties:

1. The convergence criterion is based on the maximum deviation between the current and the next time step across all grid cells. This value can be easily obtained for each subdomain separately, but a reduction is required to get a global maximum.

2. In order for the sweep over a subdomain to yield the correct results, appropriate boundary conditions must be implemented. This is no problem for cells that actually neighbor real boundaries, but for cells adjacent to a domain cut, the boundary condition changes from sweep to sweep: It is formed by the cells that lie right across the cut, and those are not available directly because they are owned by another MPI process. (With OpenMP, all data is always visible by all threads, making access across "chunk boundaries" trivial.)

The first problem can be solved directly by an `MPI_Allreduce()` call after every process has obtained the maximum deviation `maxdelta` in its own domain (step 4 in Figure 9.8).

As for the second problem, so-called *ghost* or *halo layers* are used to store copies of the boundary information from neighboring domains. Since only a single ghost layer per subdomain is required per domain cut, no additional memory must be allocated because a boundary layer is needed anyway. (We will see below, however, that some supplementary arrays may be necessary for technical reasons.) Before a process sweeps over its subdomain, which involves updating the $T = 1$ array from the $T = 0$ data, the $T = 0$ boundary values from its neighbors are obtained via MPI and stored in the ghost cells (step 2 in Figure 9.8). In the following we will outline the central parts of the Fortran implementation of this algorithm. The full code can be downloaded from the book's Web site.[2] For clarity, we will declare important variables with each code snippet.

[2] http://www.hpc.rrze.uni-erlangen.de/HPC4SE/

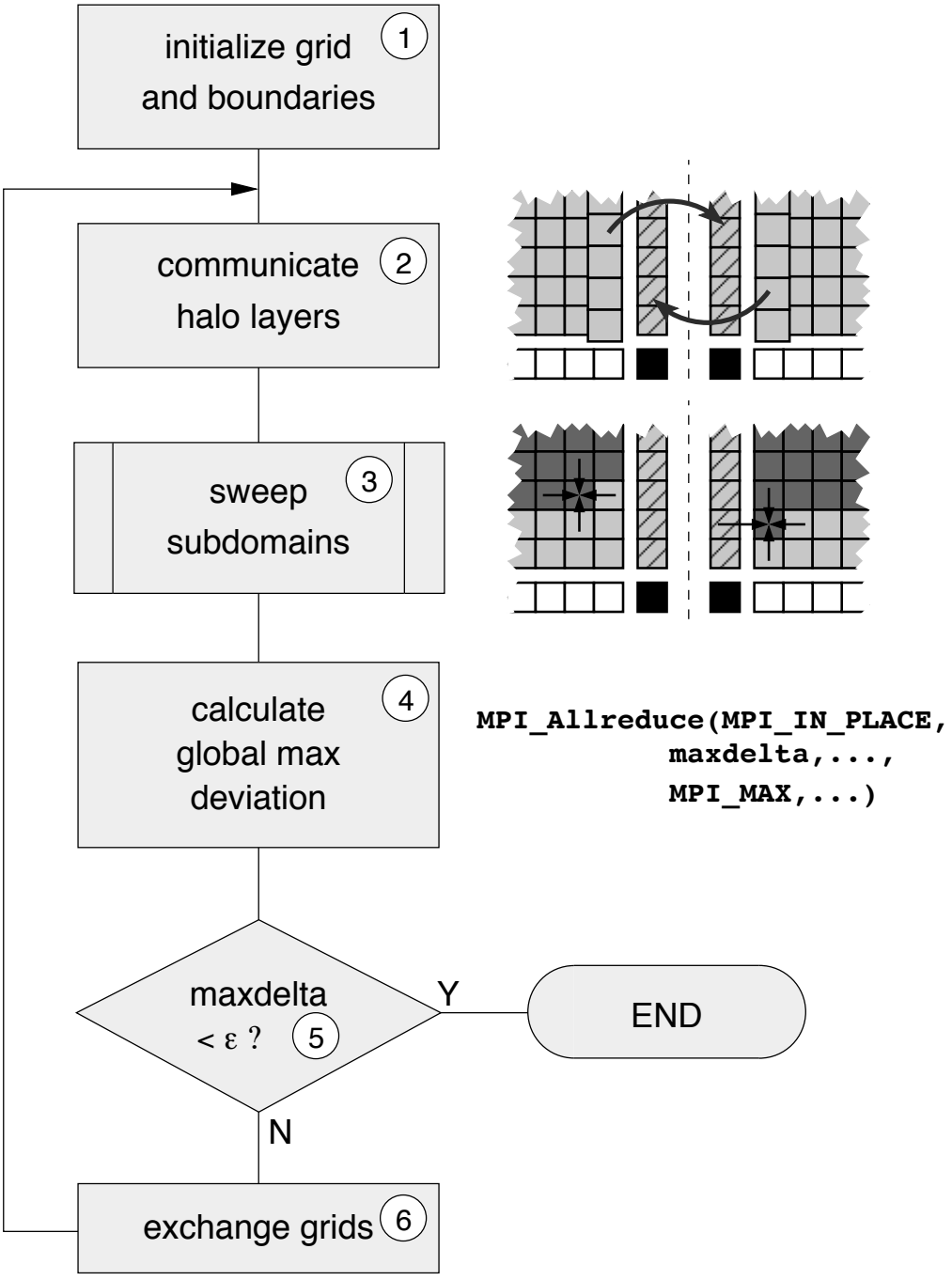


Figure 9.8: Flowchart for distributed-memory parallelization of the Jacobi algorithm. Hatched cells are ghost layers, dark cells are already updated in the $T = 1$ grid, and light-colored cells denote $T = 0$ data. White cells are real boundaries of the overall grid, whereas black cells are unused.

First the required parameters are read by rank zero from standard input (line 10 in the following listing): problem size (`spat_dim`), possible presets for number of processes (`proc_dim`), and periodicity (`pbc_check`), each for all dimensions.

```
1  logical, dimension(1:3) :: pbc_check
2  integer, dimension(1:3) :: spat_dim, proc_dim
3
4  call MPI_Comm_rank(MPI_COMM_WORLD, myid, ierr)
5  call MPI_Comm_size(MPI_COMM_WORLD, numprocs, ierr)
6
7  if(myid.eq.0) then
8    write(*,*) ' spat_dim , proc_dim, PBC ? '
9    do i=1,3
10     read(*,*) spat_dim(i), proc_dim(i), pbc_check(i)
11   enddo
12 endif
13
14 call MPI_Bcast(spat_dim , 3, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
15 call MPI_Bcast(proc_dim , 3, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
16 call MPI_Bcast(pbc_check, 3, MPI_LOGICAL, 0, MPI_COMM_WORLD, ierr)
```

Although many MPI implementations have options to allow the standard input of rank zero to be seen by all processes, a portable MPI program cannot rely on this feature, and must broadcast the data (lines 14–16). After that, the Cartesian topology can be set up using `MPI_Dims_create()` and `MPI_Cart_create()`:

```
1  call MPI_Dims_create(numprocs, 3, proc_dim, ierr)
2
3  if(myid.eq.0) write(*,'(a,3(i3,x))') 'Grid: ', &
4      (proc_dim(i),i=1,3)
5
6  l_reorder = .true.
7  call MPI_Cart_create(MPI_COMM_WORLD, 3, proc_dim, pbc_check, &
8      l_reorder, GRID_COMM_WORLD, ierr)
9
10 if(GRID_COMM_WORLD .eq. MPI_COMM_NULL) goto 999
11
12 call MPI_Comm_rank(GRID_COMM_WORLD, myid_grid, ierr)
13 call MPI_Comm_size(GRID_COMM_WORLD, nump_grid, ierr)
```

Since rank reordering is allowed (line 6), the process rank must be obtained again using `MPI_Comm_rank()` (line 12). Moreover, the new Cartesian communicator `GRID_COMM_WORLD` may be of smaller size than `MPI_COMM_WORLD`. The "surplus" processes then receive a communicator value of `MPI_COMM_NULL`, and are sent into a barrier to wait for the whole parallel program to complete (line 10).

Now that the topology has been created, the local subdomains can be set up, including memory allocation:

```
1  integer, dimension(1:3) :: loca_dim, mycoord
2
3  call MPI_Cart_coords(GRID_COMM_WORLD, myid_grid, 3,
4      mycoord,ierr)
5
```

```
6  do i=1,3
7    loca_dim(i) = spat_dim(i)/proc_dim(i)
8    if(mycoord(i) < mod(spat_dim(i),proc_dim(i))) then
9      local_dim(i) = loca_dim(i)+1
10   endif
11 enddo
12
13 iStart = 0 ; iEnd = loca_dim(3)+1
14 jStart = 0 ; jEnd = loca_dim(2)+1
15 kStart = 0 ; kEnd = loca_dim(1)+1
16
17 allocate(phi(iStart:iEnd, jStart:jEnd, kStart:kEnd,0:1))
```

Array `mycoord` is used to store a process' Cartesian coordinates as acquired from `MPI_Cart_coords()` in line 3. Array `loca_dim` holds the extensions of a process' subdomain in the three dimensions. These numbers are calculated in lines 6–11. Memory allocation takes place in line 17, allowing for an additional layer in all directions, which is used for fixed boundaries or halo as needed. For brevity, we are omitting the initialization of the array and its outer grid boundaries here.

Point-to-point communication as used for the ghost layer exchange requires consecutive message buffers. (Actually, the use of derived MPI data types would be an option here, but this would go beyond the scope of this introduction.) However, only those boundary cells that are consecutive in the inner (i) dimension are also consecutive in memory. Whole layers in the i-j, i-k, and j-k planes are never consecutive, so an intermediate buffer must be used to gather boundary data to be communicated to a neighbor's ghost layer. Sending each consecutive chunk as a separate message is out of the question, since this approach would flood the network with short messages, and latency has to be paid for every request (see Chapter 10 for more information on optimizing MPI communication).

We use two intermediate buffers per process, one for sending and one for receiving. Since the amount of halo data can be different along different Cartesian directions, the size of the intermediate buffer must be chosen to accommodate the largest possible halo:

```
1  integer, dimension(1:3) :: totmsgsize
2
3  ! j-k plane
4  totmsgsize(3) = loca_dim(1)*loca_dim(2)
5  MaxBufLen=max(MaxBufLen,totmsgsize(3))
6  ! i-k plane
7  totmsgsize(2) = loca_dim(1)*loca_dim(3)
8  MaxBufLen=max(MaxBufLen,totmsgsize(2))
9  ! i-j plane
10 totmsgsize(1) = loca_dim(2)*loca_dim(3)
11 MaxBufLen=max(MaxBufLen,totmsgsize(1))
12
13 allocate(fieldSend(1:MaxBufLen))
14 allocate(fieldRecv(1:MaxBufLen))
```

At the same time, the halo sizes for the three directions are stored in the integer array `totmsgsize`.

Now we can start implementing the main iteration loop, whose length is the maximum number of iterations (sweeps), `ITERMAX`:

```
1  t0=0 ; t1=1
2  tag = 0
3  do iter = 1, ITERMAX
4    do disp = -1, 1, 2
5      do dir = 1, 3
6
7        call MPI_Cart_shift(GRID_COMM_WORLD, (dir-1), &
8                            disp, source, dest, ierr)
9
10       if(source /= MPI_PROC_NULL) then
11         call MPI_Irecv(fieldRecv(1), totmsgsize(dir), &
12              MPI_DOUBLE_PRECISION, source, &
13              tag, GRID_COMM_WORLD, req(1), ierr)
14       endif   ! source exists
15
16       if(dest /= MPI_PROC_NULL) then
17         call CopySendBuf(phi(iStart, jStart, kStart, t0), &
18                          iStart, iEnd, jStart, jEnd, kStart, kEnd, &
19                          disp, dir, fieldSend, MaxBufLen)
20
21         call MPI_Send(fieldSend(1), totmsgsize(dir), &
22                       MPI_DOUBLE_PRECISION, dest, tag, &
23                       GRID_COMM_WORLD, ierr)
24       endif   ! destination exists
25
26       if(source /= MPI_PROC_NULL) then
27         call MPI_Wait(req, status, ierr)
28
29         call CopyRecvBuf(phi(iStart, jStart, kStart, t0), &
30                          iStart, iEnd, jStart, jEnd, kStart, kEnd, &
31                          disp, dir, fieldRecv, MaxBufLen)
32       endif   ! source exists
33
34     enddo   ! dir
35   enddo   ! disp
36
37   call Jacobi_sweep(loca_dim(1), loca_dim(2), loca_dim(3), &
38                     phi(iStart, jStart, kStart, 0), t0, t1, &
39                     maxdelta)
40
41   call MPI_Allreduce(MPI_IN_PLACE, maxdelta, 1, &
42                      MPI_DOUBLE_PRECISION, &
43                      MPI_MAX, 0, GRID_COMM_WORLD, ierr)
44   if(maxdelta<eps) exit
45   tmp=t0; t0=t1; t1=tmp
46 enddo   ! iter
47
48 999 continue
```

Halos are exchanged in six steps, i.e., separately per positive and negative Cartesian direction. This is parameterized by the loop variables `disp` and `dir`. In line 7, `MPI_Cart_shift()` is used to determine the communication neighbors along the

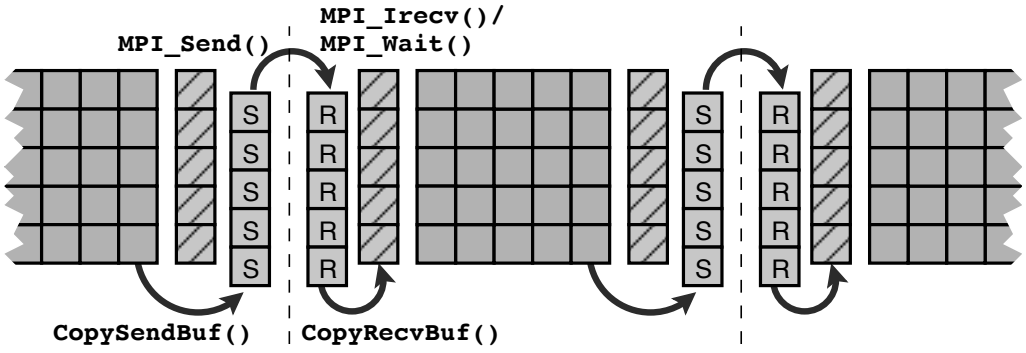


Figure 9.9: Halo communication for the Jacobi solver (illustrated in two dimensions here) along one of the coordinate directions. Hatched cells are ghost layers, and cells labeled "R" ("S") belong to the intermediate receive (send) buffer. The latter is being reused for all other directions. Note that halos are always provided for the grid that gets read (not written) in the upcoming sweep. Fixed boundary cells are omitted for clarity.

current direction (`source` and `dest`). If a subdomain is located at a grid boundary, and periodic boundary conditions are not in place, the neighbor will be reported to have rank `MPI_PROC_NULL`. MPI calls using this rank as source or destination will return immediately. However, as the copying of halo data to and from the intermediate buffers should be avoided for efficiency in this case, we also mask out any MPI calls, keeping overhead to a minimum (lines 10, 16, and 26).

The communication pattern along a direction is actually a ring shift (or a linear shift in case of open boundary conditions). The problems inherent to a ring shift with blocking point-to-point communication were discussed in Section 9.2.2. To avoid deadlocks, and possibly utilize the available network bandwidth to full extent, a nonblocking receive is initiated before anything else (line 11). This data transfer can potentially overlap with the subsequent halo copy to the intermediate send buffer, done by the `CopySendBuf()` subroutine (line 17). After sending the halo data (line 21) and waiting for completion of the previous nonblocking receive (line 27), `CopyRecvBuf()` finally copies the received halo data to the boundary layer (line 29), which completes the communication cycle in one particular direction. Figure 9.9 again illustrates this chain of events.

After the six halo shifts, the boundaries of the current grid `phi(:,:,:,t0)` are up to date, and a Jacobi sweep over the local subdomain is performed, which updates `phi(:,:,:,t1)` from `phi(:,:,:,t0)` (line 37). The corresponding subroutine `Jacobi_sweep()` returns the maximum deviation between the previous and the current time step for the subdomain (see Listing 6.5 for a possible implementation in 2D). A subsequent `MPI_Allreduce()` (line 41) calculates the global maximum and makes it available on all processes, so that the decision whether to leave the iteration loop because convergence has been reached (line 44) can be made on all ranks without further communication.

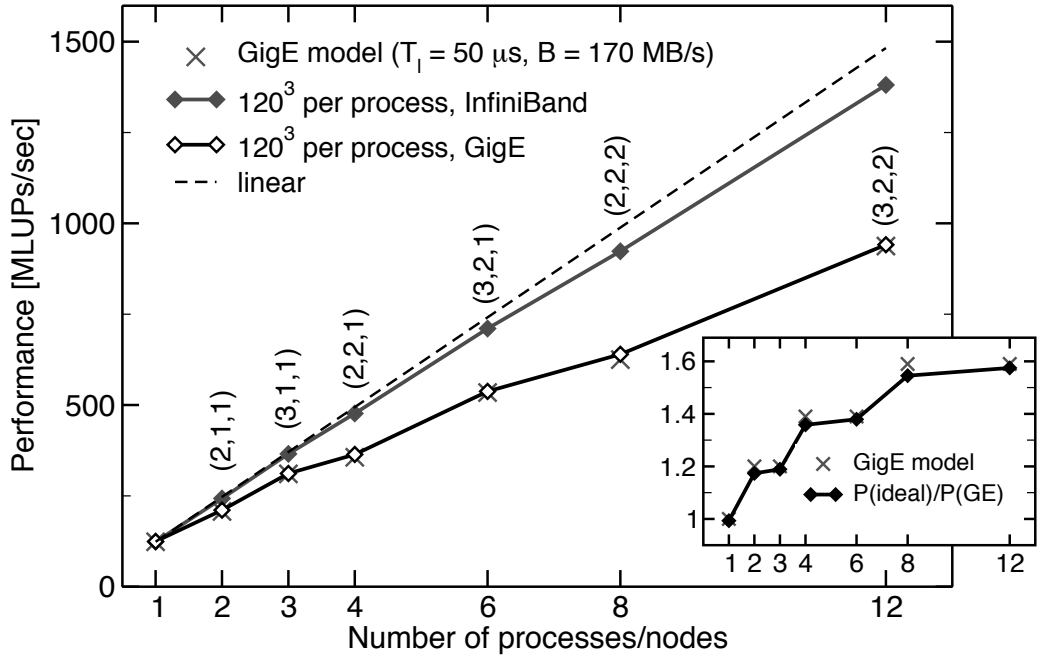


Figure 9.10: Main panel: Weak scaling of the MPI-parallel 3D Jacobi code with problem size 120^3 per process on InfiniBand vs. Gigabit Ethernet networks. Only one process per node was used. The domain decomposition topology (number of processes in each Cartesian direction) is indicated. The weak scaling performance model (crosses) can reproduce the GigE data well. Inset: Ratio between ideally scaled performance and Gigabit Ethernet performance vs. process count. (Single-socket cluster based on Intel Xeon 3070 at 2.66 GHz, Intel MPI 3.2.)

9.3.2 Performance properties

The performance characteristics of the MPI-parallel Jacobi solver are typical for many domain decomposition codes. We distinguish between weak and strong scaling scenarios, as they show quite different features. All benchmarks were performed with two different interconnect networks (Intel MPI Version 3.2 over DDR InfiniBand vs. Gigabit Ethernet) on a commodity cluster with single-socket nodes based on Intel Xeon 3070 processors at 2.66 GHz. A single process per node was used throughout in order to get a simple scaling baseline and minimize intranode effects.

Weak scaling

In our performance models in Section 5.3.6 we have assumed that 3D domain decomposition at weak scaling has constant communication overhead. This is, however, not always true because a subdomain that is located at a grid boundary may have fewer halo faces to communicate. Fortunately, due to the inherent synchronization between subdomains, the overall runtime of the parallel program is dominated by the slowest process, which is the one with the largest number of halo faces if all processes work on equally-sized subdomains. Hence, we can expect reasonably linear scaling behavior even on slow (but nonblocking) networks, once there is at least one subdomain that is surrounded by other subdomains in all Cartesian directions.

The weak scaling data for a constant subdomain size of 120^3 shown in Figure 9.10 substantiates this conjecture:

Scalability on the InfiniBand network is close to perfect. For Gigabit Ethernet, communication still costs about 40% of overall runtime at large node counts, but this fraction gets much smaller when running on fewer nodes. In fact, the performance graph shows a peculiar "jagged" structure, with slight breakdowns at 4 and 8 processes. These breakdowns originate from fundamental changes in the communication characteristics, which occur when the number of subdomains in any coordinate direction changes from one to anything greater than one. At that point, internode communication along this axis sets in: Due to the periodic boundary conditions, every process always communicates in all directions, but if there is only one process in a certain direction, it exchanges halo data only with itself, using (fast) shared memory. The inset in Figure 9.10 indicates the ratio between ideal scaling and Gigabit Ethernet performance data. Clearly this ratio gets larger whenever a new direction gets cut. This happens at the decompositions (2,1,1), (2,2,1), and (2,2,2), respectively, belonging to node counts of 2, 4, and 8. Between these points, the ratio is roughly constant, and since there are only three Cartesian directions, it can be expected to not exceed a value of ≈ 1.6 even for very large node counts, assuming that the network is nonblocking. The same behavior can be observed with the InfiniBand data, but the effect is much less pronounced due to the much larger ($\times 10$) bandwidth and lower ($/20$) latency. Note that, although we use a performance metric that is only relevant in the parallel part of the program, the considerations from Section 5.3.3 about "fake" weak scalability do not apply here; the single-CPU performance is on par with the expectations from the STREAM benchmark (see Section 3.3).

The communication model described above is actually good enough for a quantitative description. We start with the assumption that the basic performance characteristics of a point-to-point message transfer can be described by a simple latency/bandwidth model along the lines of Figure 4.10. However, since sending and receiving halo data on each MPI process can overlap for each of the six coordinate directions, we must include a maximum bandwidth number for full-duplex data transfer over a single link. The (half-duplex) PingPong benchmark is not accurate enough to get a decent estimate for full-duplex bandwidth, even though most networks (including Ethernet) claim to support full-duplex. The Gigabit Ethernet network used for the Jacobi benchmark can deliver about 111 MBytes/sec for half-duplex and 170 MBytes/sec for full-duplex communication, at a latency of 50 μs.

The subdomain size is the same regardless of the number of processes, so the raw compute time T_s for all cell updates in a Jacobi sweep is also constant. Communication time T_c, however, depends on the number and size of domain cuts that lead to internode communication, and we are assuming that copying to/from intermediate buffers and communication of a process with itself come at no cost. Performance on $N = N_x N_y N_z$ processes for a particular overall problem size of $L^3 N$ grid points (using cubic subdomains of size L^3) is thus

$$P(L,\vec{N}) = \frac{L^3 N}{T_\mathrm{s}(L) + T_\mathrm{c}(L,\vec{N})} \ , \tag{9.1}$$

where

$$T_c(L,\vec{N}) = \frac{c(L,\vec{N})}{B} + kT_\ell \ . \tag{9.2}$$

Here, $c(L,\vec{N})$ is the maximum bidirectional data volume transferred over a node's network link, B is the full-duplex bandwidth, and k is the largest number (over all subdomains) of coordinate directions in which the number of processes is greater than one. $c(L,\vec{N})$ can be derived from the Cartesian decomposition:

$$c(L,\vec{N}) = L^2 \cdot k \cdot 2 \cdot 8 \tag{9.3}$$

For $L = 120$ this leads to the following numbers:

N	(N_z,N_y,N_x)	k	$c(L,\vec{N})$ [MB]	$P(L,\vec{N})$ [MLUPs/sec]	$\frac{NP_1(L)}{P(L,\vec{N})}$
1	(1,1,1)	0	0.000	124	1.00
2	(2,1,1)	2	0.461	207	1.20
3	(3,1,1)	2	0.461	310	1.20
4	(2,2,1)	4	0.922	356	1.39
6	(3,2,1)	4	0.922	534	1.39
8	(2,2,2)	6	1.382	625	1.59
12	(3,2,2)	6	1.382	938	1.59

$P_1(L)$ is the measured single-processor performance for a domain of size L^3. The prediction for $P(L,\vec{N})$ can be seen in the third column, and the last column quantifies the "slowdown factor" compared to perfect scaling. Both are shown for comparison with the measured data in the main panel and inset of Figure 9.10, respectively. The model is clearly able to describe the performance features of weak scaling well, which is an indication that our general concept of the communication vs. computation "workflow" was correct. Note that we have deliberately chosen a small problem size to emphasize the role of communication, but the influence of latency is still minor.

Strong scaling

Figure 9.11 shows strong scaling performance data for periodic boundary conditions on two different problem sizes (120^3 vs. 480^3). There is a slight penalty for the smaller size (about 10%) even with one processor, independent of the interconnect. For InfiniBand, the performance gap between the two problem sizes can mostly be attributed to the different subdomain sizes. The influence of communication on scalability is minor on this network for the node counts considered. On Gigabit Ethernet, however, the smaller problem scales significantly worse because the ratio of halo data volume (and latency overhead) to useful work becomes so large at larger node counts that communication dominates the performance on this slow network. The typical "jagged" pattern in the scaling curve is superimposed by the communication volume changing whenever the number of processes changes. A simple predictive model as with weak scaling is not sufficient here; especially with small grids, there is

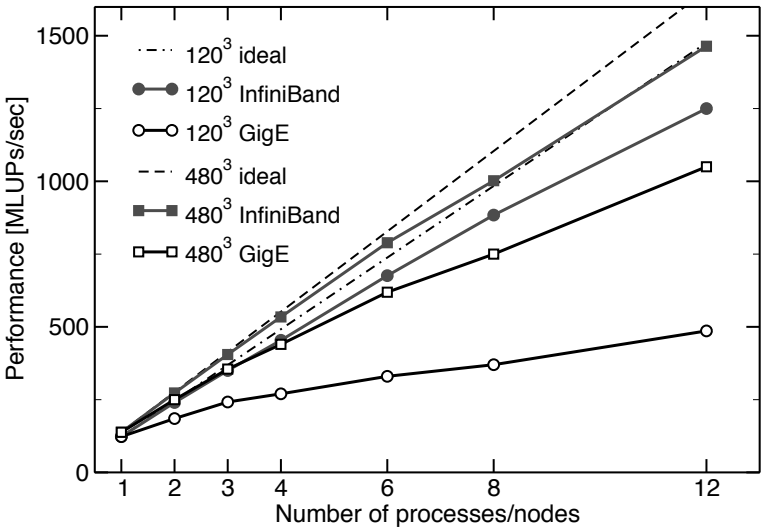


Figure 9.11: Strong scaling of the MPI-parallel 3D Jacobi code with problem size 120^3 (circles) and 480^3 (squares) on IB (filled symbols) vs. GigE (open symbols) networks. Only one process per node was used. (Same system and MPI topology as in Figure 9.10.)

a strong dependence of single-process performance on the subdomain size, and intranode communication processes (halo exchange) become important. See Problem 9.4 and Section 10.4.1 for some more discussion.

Note that this analysis must be refined when dealing with multiple MPI processes per node, as is customary with current parallel systems (see Section 4.4). Especially on fast networks, intranode communication characteristics play a central role (see Section 10.5 for details). Additionally, copying of halo data to and from intermediate buffers within a process cannot be neglected.

Problems

For solutions see page 304 *ff*.

9.1 *Shifts and deadlocks*. Does the remedy for the deadlock problem with ring shifts as shown in Figure 9.2 (exchanging send/receive order) also work if the number of processes is odd?

What happens if the chain is open, i.e., if rank 0 does not communicate with the highest-numbered rank? Does the reordering of sends and receives make a difference in this case?

9.2 *Deadlocks and nonblocking MPI*. In order to avoid deadlocks, we used nonblocking receives for halo exchange in the MPI-parallel Jacobi code (Section 9.3.1). An MPI implementation is actually not required to support overlapping of communication and computation; MPI progress, i.e., real data transfer, might happen only if MPI library code is executed. Under such conditions, is it still guaranteed that deadlocks cannot occur? Consult the MPI standard if in doubt.

9.3 *Open boundary conditions*. The performance model for weak scaling of the Jacobi code in Section 9.3.2 assumed periodic boundary conditions. How would the model change for open (Dirichlet-type) boundaries? Would there still be plateaus in the inset of Figure 9.10? What would happen when going from 12 to 16 processes? What is the minimum number of processes for which the ratio between ideal and real performance reaches its maximum?

9.4 *A performance model for strong scaling of the parallel Jacobi code*. As mentioned in Section 9.3.2, a performance model that accurately predicts the strong scaling behavior of the MPI-parallel Jacobi code is more involved than for weak scaling. Especially the dependence of the single-process performance on the subdomain size is hard to predict since it depends on many factors (pipelining effects, prefetching, spatial blocking strategy, copying to intermediate buffers, etc.). This was no problem for weak scaling because of the constant subdomain size. Nevertheless one could try to establish a partly "phenomenological" model by measuring single-process performance for all subdomain sizes that appear in the parallel run, and base a prediction for parallel performance on those baselines. What else would you consider to be required enhancements to the weak scaling model? Take into account that T_s becomes smaller and smaller as N grows, and that halo exchange is not the only internode communication that is going on.

9.5 *MPI correctness*. Is the following MPI program fragment correct? Assume that only two processes are running, and that `my_rank` contains the rank of each process.

```
1 if(my_rank.eq.0) then
2   call MPI_Bcast(buf1, count, type, 0, comm, ierr)
3   call MPI_Send(buf2, count, type, 1, tag, comm, ierr)
4 else
5   call MPI_Recv(buf2, count, type, 0, tag, comm, status, ierr)
6   call MPI_Bcast(buf1, count, type, 0, comm, ierr)
7 endif
```

(This example is taken from the MPI 2.2 standard document [P15].)

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

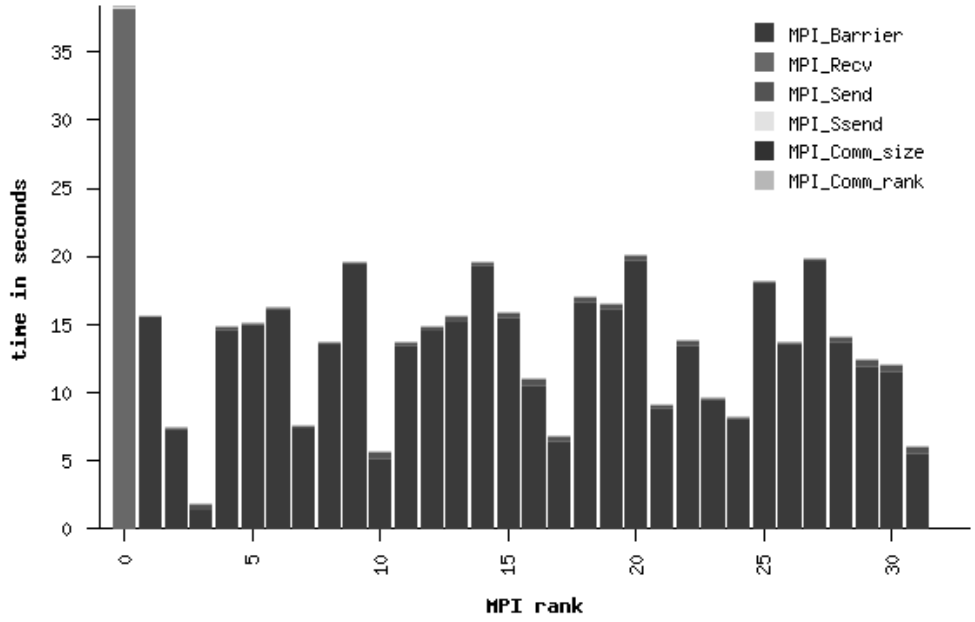


Figure 10.1: (See color insert after page 262.) IPM "communication balance" of a master-worker style parallel application. The complete runtime was about 38 seconds, which are spent almost entirely inside `MPI_Recv()` on rank 0. The other ranks are very load imbalanced, spending between 10 and 50% of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

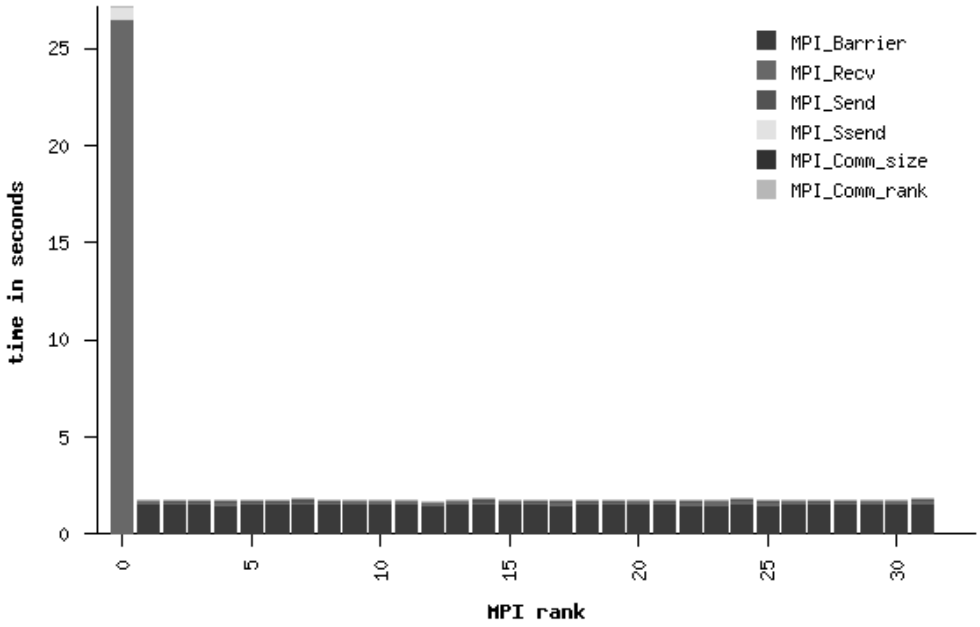


Figure 10.2: (See color insert after page 262.) IPM function profile of the same application as in Figure 10.1, with the load imbalance problem removed.

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

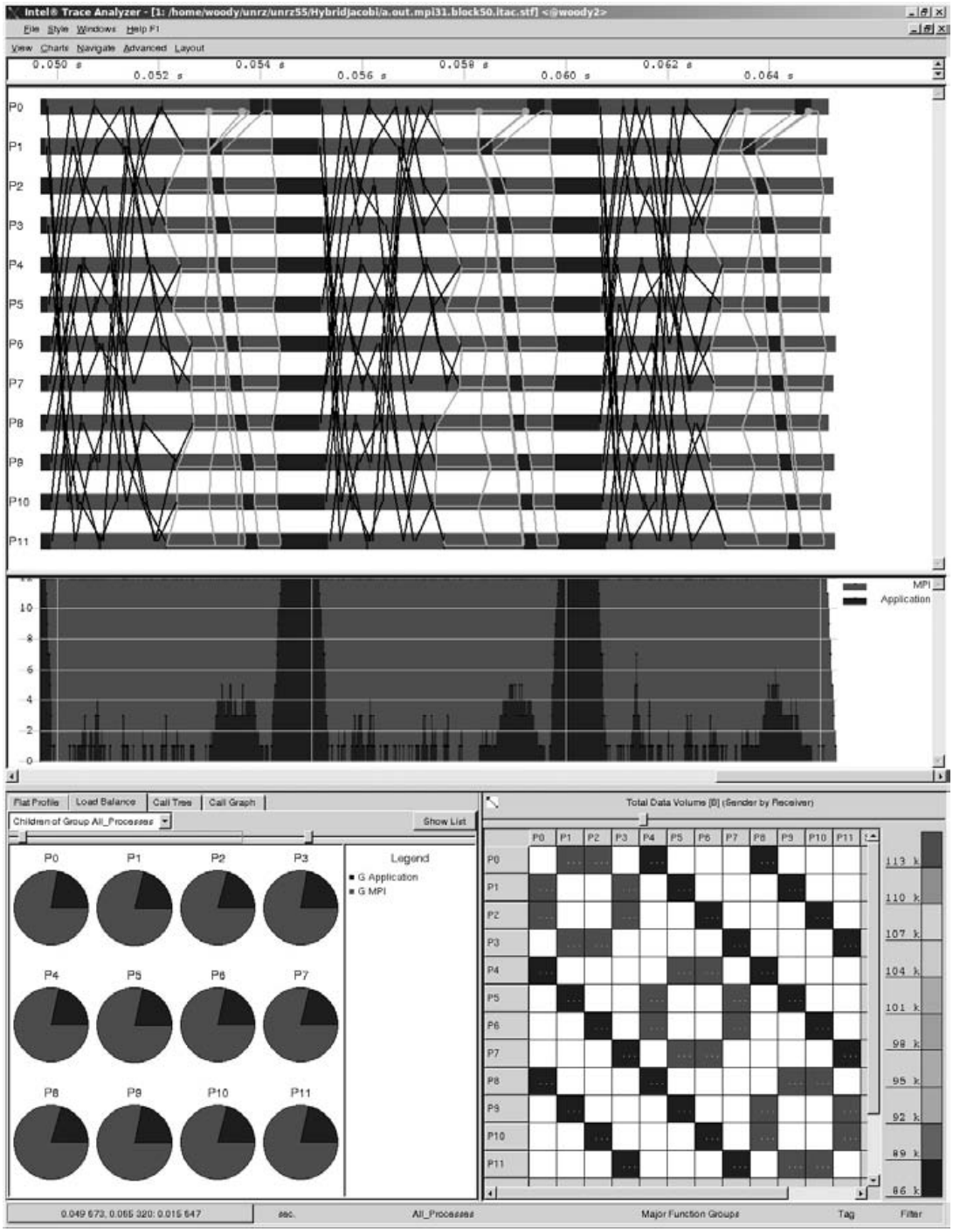


Figure 10.3: (See color insert after page 262.) Intel Trace Analyzer timeline view (top), load balance analysis (left bottom) and communication summary (right bottom) for an MPI-parallel code running on 12 nodes. Point-to-point (collective) messages are depicted by dark (bright) lines in the timeline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

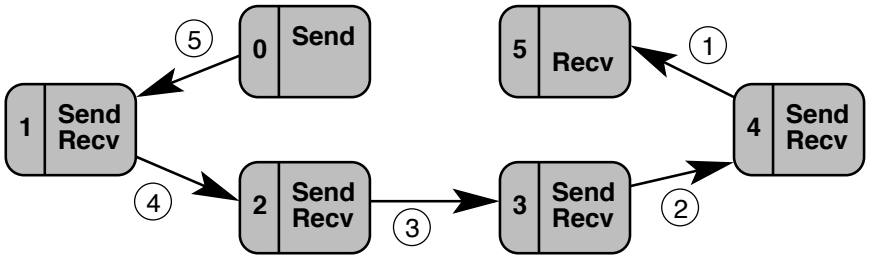


Figure 10.4: A linear shift communication pattern. Even with synchronous point-to-point communication, a deadlock will not occur, but all message transfers will be serialized in the order shown.

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-

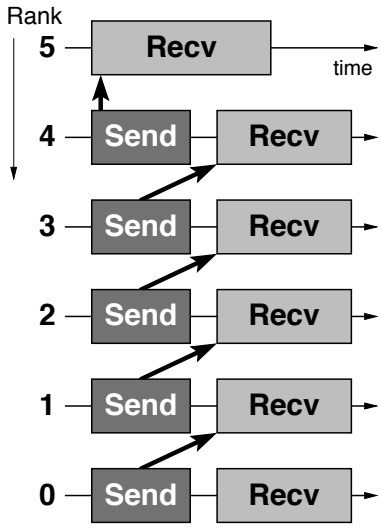


Figure 10.5: Timeline view of the linear shift (see Figure 10.4) with blocking (but not synchronous) sends and blocking receives, using eager delivery. Message transmissions can overlap, making use of a nonblocking network. Eager delivery allows a send to end before the corresponding receive is posted.

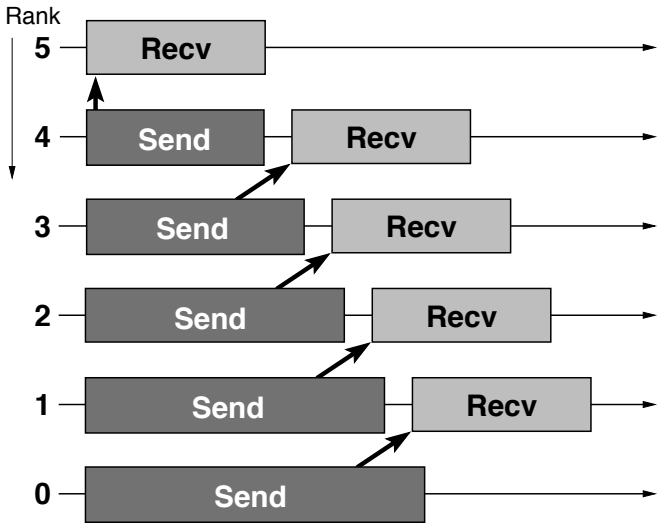


Figure 10.6: Timeline view of the linear shift (see Figure 10.4) with blocking synchronous sends and blocking receives, using eager delivery. The message transfers (arrows) might overlap perfectly, but a send can only finish just after its matching receive is posted.

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-

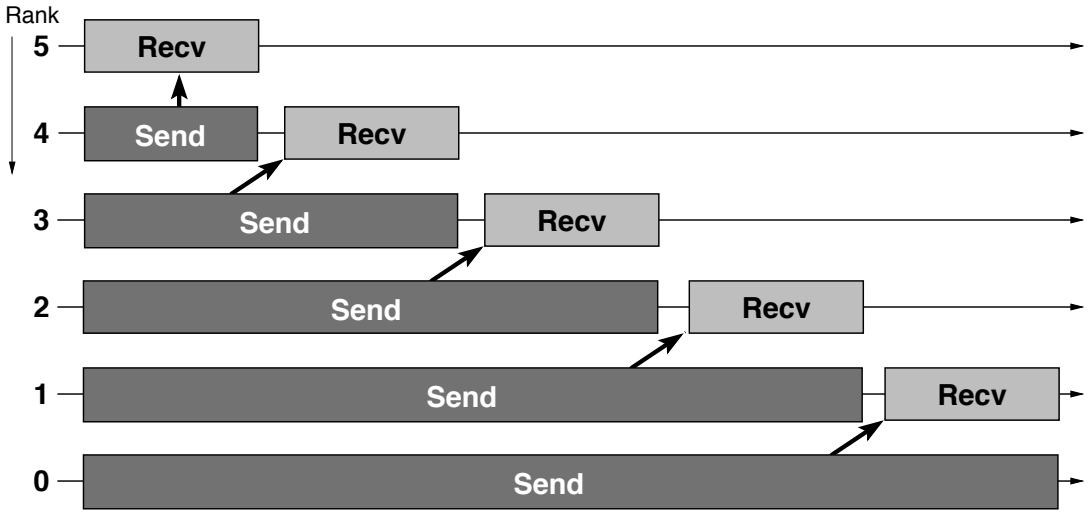


Figure 10.7: Timeline view of the linear shift (see Figure 10.4) with blocking sends and blocking receives, using the rendezvous protocol. The messages (arrows) are transmitted in serial because buffering is ruled out.

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.

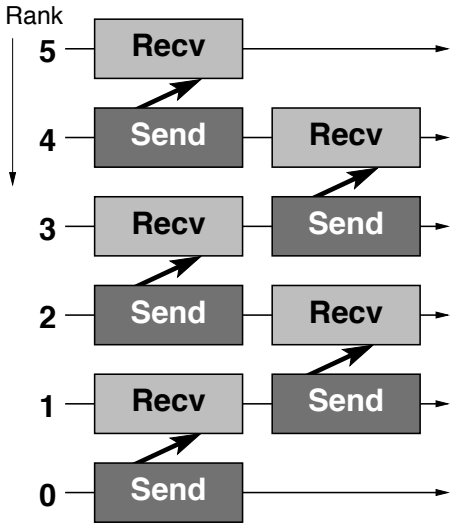


Figure 10.8: Even if sends are synchronous and the rendezvous protocol is used, exchanging the order of sends and receives on all odd-numbered ranks exposes some parallelism in communication.

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].

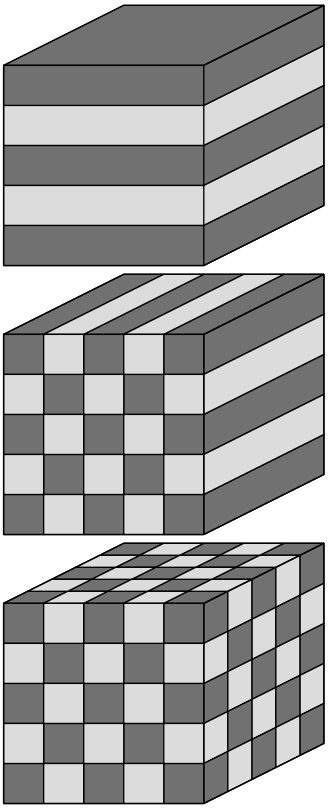
10.4 Reducing communication overhead

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"

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

"Poles"

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

"Cubes"

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

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.

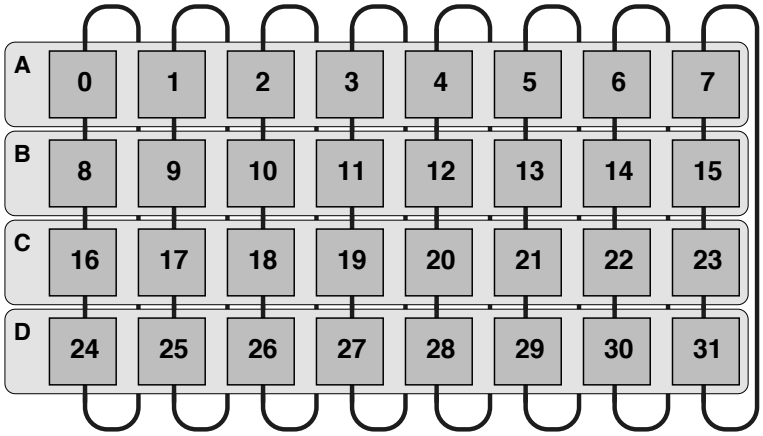


Figure 10.10: A typical default mapping of MPI ranks (numbers) to subdomains (squares) and cluster nodes (letters) for a two-dimensional 4×8 periodic domain decomposition. Each node has 16 connections to other nodes. Intranode connections are omitted.

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.

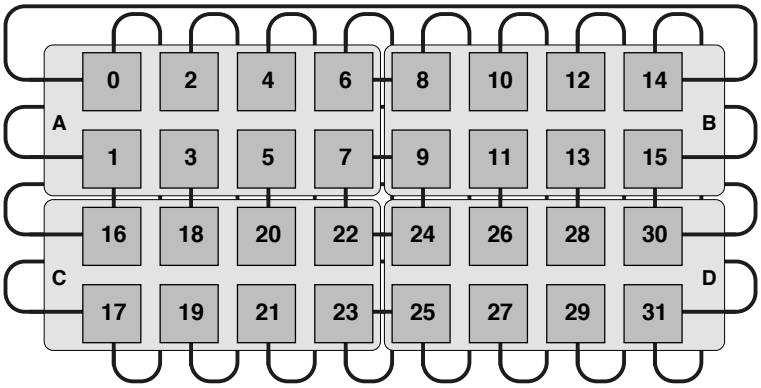


Figure 10.11: A "perfect" mapping of MPI ranks and subdomains to nodes. Each node has 12 connections to other nodes.

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.

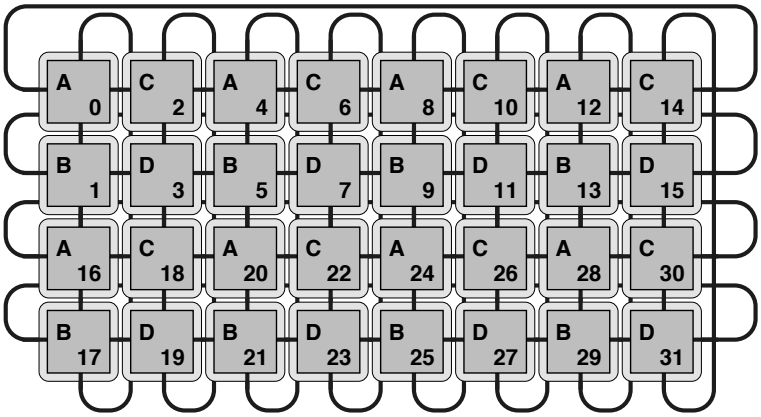


Figure 10.12: The same rank-to-subdomain mapping as in Figure 10.11, but with ranks assigned to nodes in a "round-robin" way, i.e., successive ranks run on different nodes. This leads to 32 internode connections per node.

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 Ping-Pong 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

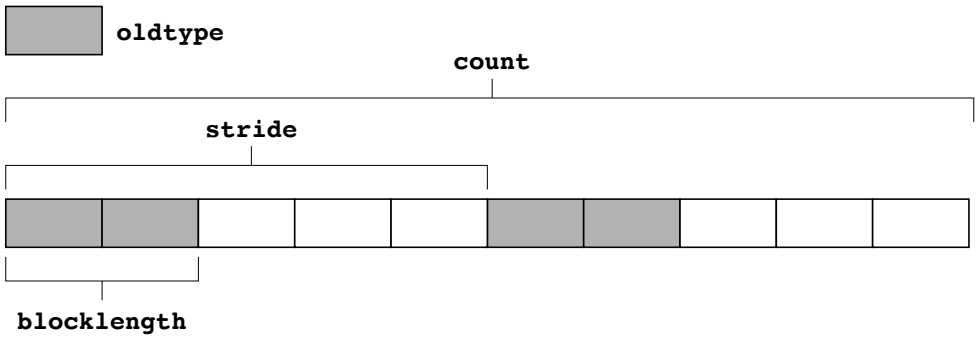


Figure 10.13: Required parameters for `MPI_Type_vector`. Here `blocklength`=2, `stride`=5, and `count`=2.

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-

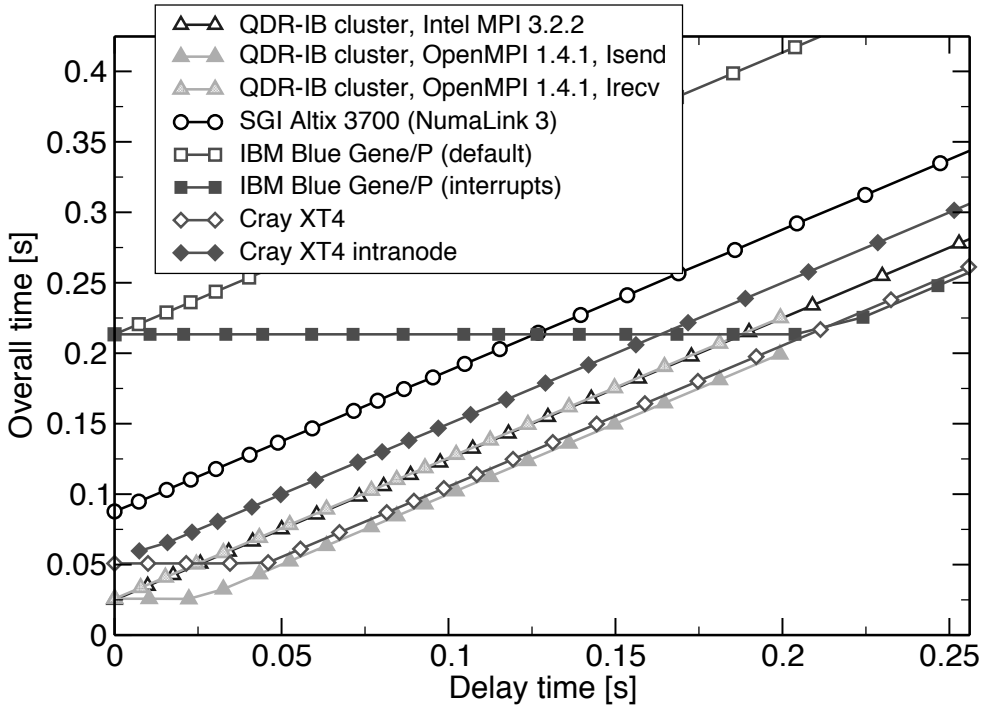


Figure 10.14: Results from the MPI overlap benchmark on different architectures, interconnects and MPI versions. Among those, only the MPI library on a Cray XT4 (diamonds) and OpenMPI on an InfiniBand cluster (filled triangles) are capable of asynchronous transfers by default, OpenMPI allowing no overlap 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.

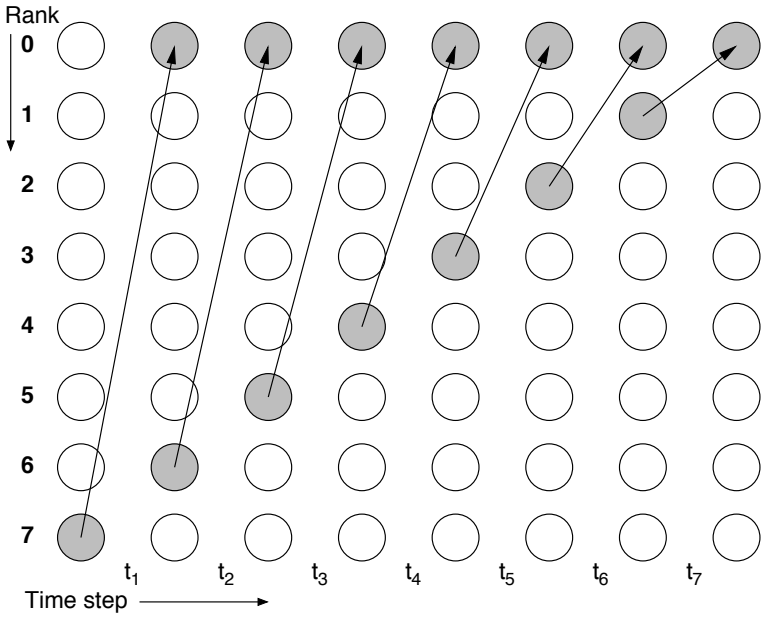


Figure 10.15: A global reduction with communication overhead being linear in the number of processes, as implemented in the integration example (Listing 9.3). Each arrow represents a message to the receiver process. Processes that communicate during a time step are shaded.

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-

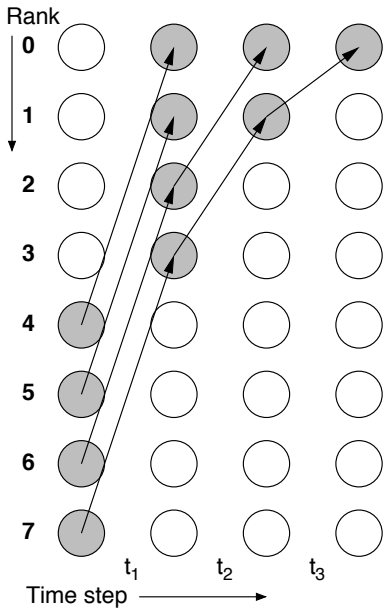


Figure 10.16: With a tree-like, hierarchical reduction pattern, communication overhead is logarithmic in the number of processes because communication during each time step is concurrent.

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].

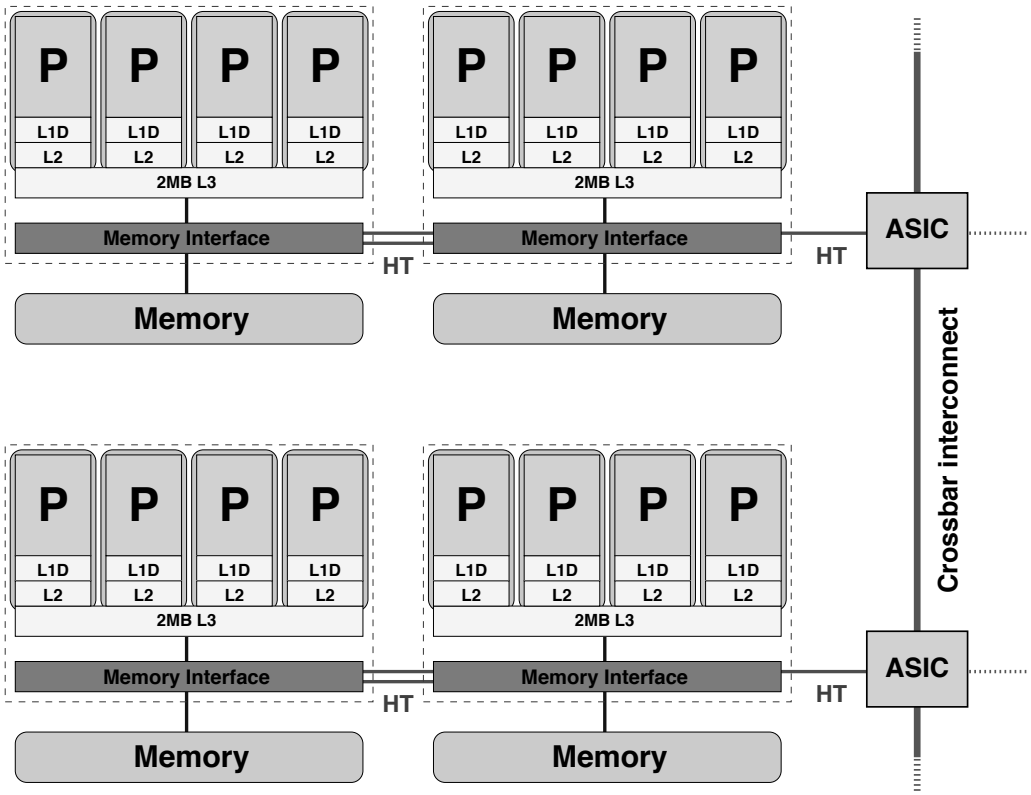


Figure 10.17: Two nodes of a Cray XT5 system. Dashed boxes denote AMD Opteron sockets, and there are two sockets (NUMA locality domains) per node. The crossbar interconnect is actually a 3D torus (mesh) network.

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

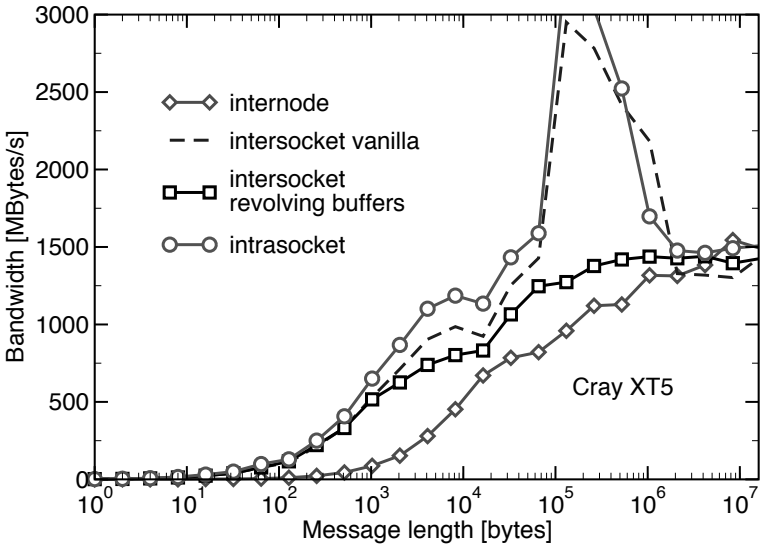
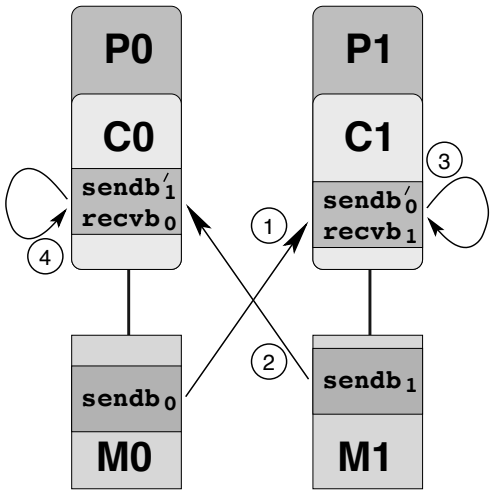


Figure 10.18: IMB PingPong performance for internode, intranode but intersocket, and pure intrasocket communication on a Cray XT5 system. Intersocket "vanilla" data was obtained without using revolving buffers (see text for details).

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-

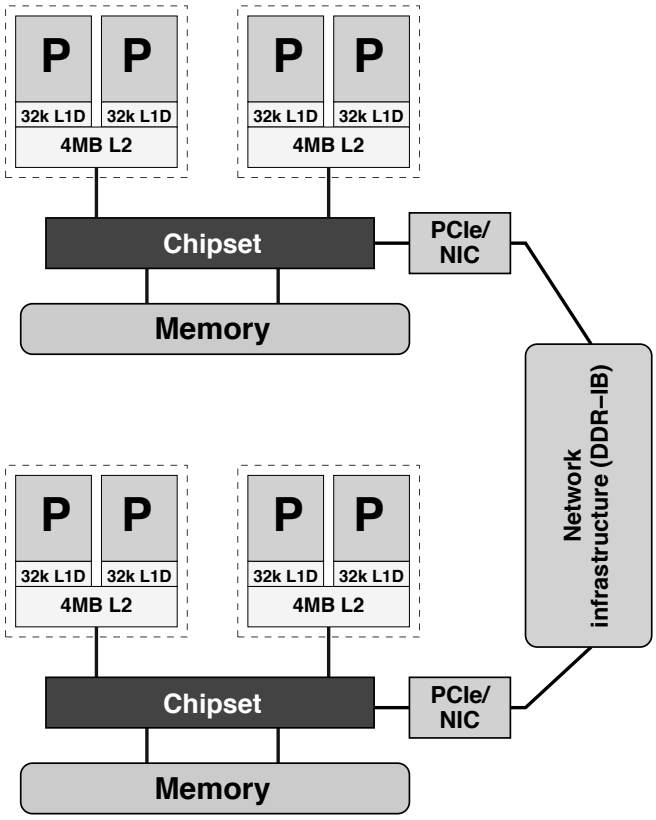


Figure 10.20: Two nodes of a Xeon 5160 dual-socket cluster system with a DDR-InfiniBand interconnect.

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

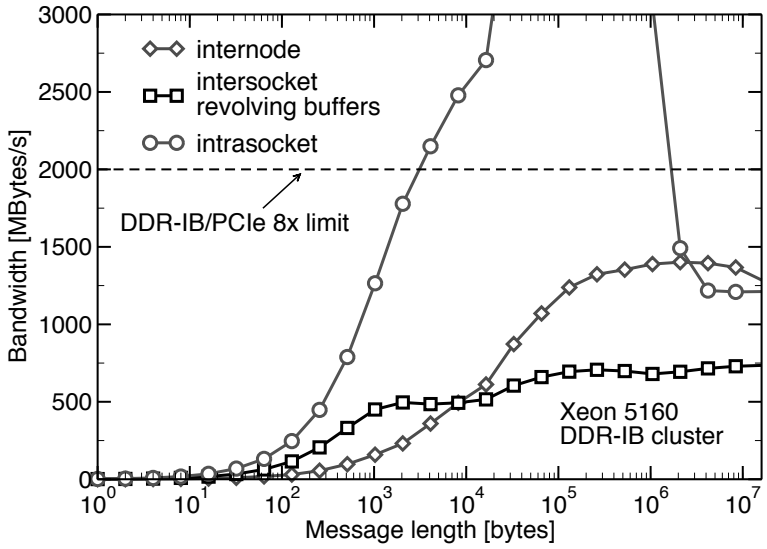


Figure 10.21: IMB PingPong performance for internode, intranode but intersocket, and pure intrasocket communication on a Xeon 5160 DDR-IB cluster, using Intel MPI 3.0.

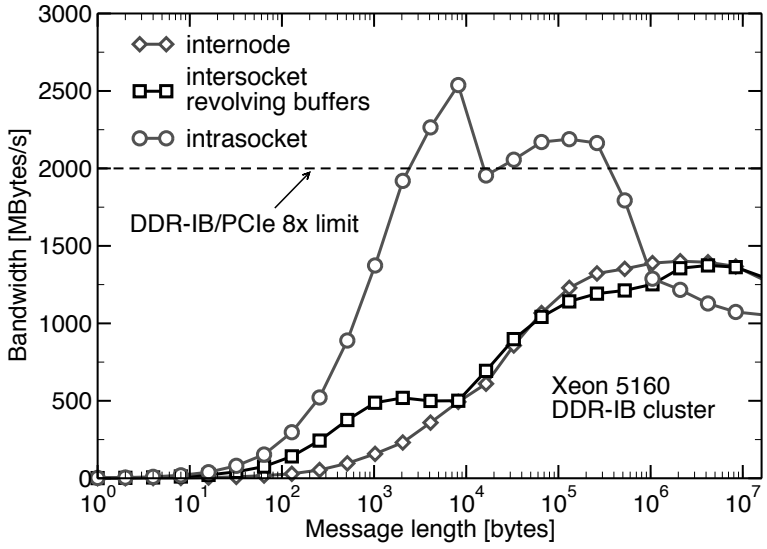


Figure 10.22: The same benchmark as in Figure 10.21, but using Intel MPI 3.1. Intranode behavior has changed significantly.

	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 $B_{\text{eff}}(N, L, w, T_\ell, B)$. 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?

Chapter 11

Hybrid parallelization with MPI and OpenMP

Large-scale parallel computers are nowadays exclusively of the distributed-memory type at the overall system level but use shared-memory compute nodes as basic building blocks. Even though these hybrid architectures have been in use for more than a decade, most parallel applications still take no notice of the hardware structure and use pure MPI for parallelization. This is not a real surprise if one considers that the roots of most parallel applications, solvers and methods as well as the MPI library itself date back to times when all "big" machines were pure distributed-memory types, such as the famous Cray T3D/T3E MPP series. Later the existing MPI applications and libraries were easy to port to shared-memory systems, and thus most effort was spent to improve MPI scalability. Moreover, application developers confided in the MPI library providers to deliver efficient MPI implementations, which put the full capabilities of a shared-memory system to use for high-performance intranode message passing (see also Section 10.5 for some of the problems connected with intranode MPI). Pure MPI was hence implicitly assumed to be as efficient as a well-implemented hybrid MPI/OpenMP code using MPI for internode communication and OpenMP for parallelization within the node. The experience with small- to moderately-sized shared-memory nodes (no more than two or four processors per node) in recent years also helped to establish a general lore that a hybrid code can usually not outperform a pure MPI version for the same problem.

It is more than doubtful whether the attitude of running one MPI process per core is appropriate in the era of multicore processors. The parallelism within a single chip will steadily increase, and the shared-memory nodes will have highly parallel, hierarchical, multicore multisocket structures. This section will shed some light on this development and introduce basic guidelines for writing and running a good hybrid code on this new class of shared-memory nodes. First, expected weaknesses and advantages of hybrid OpenMP/MPI programming will be discussed. Turning to the "mapping problem," we will point out that the performance of hybrid as well as pure MPI codes depends crucially on factors not directly connected to the programming model, but to the association of threads and processes to cores. In addition, there are several choices as to how exactly OpenMP threads and MPI processes can interact inside a node, which leaves significant room for improvement in most hybrid applications.

11.1 Basic MPI/OpenMP programming models

The basic idea of a hybrid OpenMP/MPI programming model is to allow any MPI process to spawn a team of OpenMP threads in the same way as the master thread does in a pure OpenMP program. Thus, inserting OpenMP compiler directives into an existing MPI code is a straightforward way to build a first hybrid parallel program. Following the guidelines of good OpenMP programming, compute intensive loop constructs are the primary targets for OpenMP parallelization in a naïve hybrid code. Before launching the MPI processes one has to specify the maximum number of OpenMP threads per MPI process in the same way as for a pure OpenMP program. At execution time each MPI process activates a team of threads (being the master thread itself) whenever it encounters an OpenMP parallel region.

There is no automatic synchronization between the MPI processes for switching from pure MPI to hybrid execution, i.e., at a given time some MPI processes may run in completely different OpenMP parallel regions, while other processes are in a pure MPI part of the program. Synchronization between MPI processes is still restricted to the use of appropriate MPI calls.

We define two basic hybrid programming approaches [O69]: *Vector mode* and *task mode*. These differ in the degree of interaction between MPI calls and OpenMP directives. Using the parallel 3D Jacobi solver as an example, the basic idea of both approaches will be briefly introduced in the following.

11.1.1 Vector mode implementation

In a vector mode implementation all MPI subroutines are called outside OpenMP parallel regions, i.e., in the "serial" part of the OpenMP code. A major advantage is the ease of programming, since an existing pure MPI code can be turned hybrid just by adding OpenMP worksharing directives in front of the time-consuming loops and taking care of proper NUMA placement (see Chapter 8). A pseudocode for a vector mode implementation of a 3D Jacobi solver core is shown in Listing 11.1. This looks very similar to pure MPI parallelization as shown in Section 9.3, and indeed there is no interference between the MPI layer and the OpenMP directives. Programming follows the guidelines for both paradigms independently. The vector mode strategy is similar to programming parallel vector computers with MPI, where the inner layer of parallelism is exploited by vectorization and multitrack pipelines. Typical examples which may benefit from this mode are applications where the number of MPI processes is limited by problem-specific constraints. Exploiting an additional (lower) level of finer granularity by multithreading is then the only way to increase parallelism beyond the MPI limit [O70].

Listing 11.1: Pseudocode for a vector mode hybrid implementation of a 3D Jacobi solver.

```
1   do iteration=1,MAXITER
2   ...
3  !$OMP PARALLEL DO PRIVATE(..)
4     do k = 1,N
5  ! Standard 3D Jacobi iteration here
6  ! updating all cells
7        ...
8     enddo
9  !$OMP END PARALLEL DO
10
11 ! halo exchange
12    ...
13    do dir=i,j,k
14
15      call MPI_Irecv( halo data from neighbor in -dir direction )
16      call MPI_Isend( data to neighbor in +dir direction )
17
18      call MPI_Irecv( halo data from neighbor in +dir direction )
19      call MPI_Isend( data to neighbor in -dir direction )
20    enddo
21    call MPI_Waitall( )
22  enddo
```

11.1.2 Task mode implementation

The task mode is most general and allows any kind of MPI communication within OpenMP parallel regions. Based on the thread safety requirements for the message passing library, the MPI standard defines three different levels of interference between OpenMP and MPI (see Section 11.2 below). Before using task mode, the code must check which of these levels is supported by the MPI library. Functional task decomposition and decoupling of communication and computation are two areas where task mode can be useful. As an example for the latter, a sketch of a task mode implementation of the 3D Jacobi solver core is shown in Listing 11.2. Here the master thread is responsible for updating the boundary cells (lines 6–9), i.e., the surface cells of the process-local subdomain, and communicates the updated values to the neighboring processes (lines 13–20). This can be done concurrently with the update of all inner cells, which is performed by the remaining threads (lines 24–40). After a complete domain update, synchronization of all OpenMP threads within each MPI process is required, while MPI synchronization only occurs indirectly between nearest neighbors via halo exchange.

Task mode provides a high level of flexibility but blows up code size and increases coding complexity considerably. A major problem is that neither MPI nor OpenMP has embedded mechanisms that directly support the task mode approach. Thus, one usually ends up with an MPI-style programming on the OpenMP level as well. The different functional tasks need to be mapped to OpenMP thread IDs (cf. the `if` statement starting in line 5) and may operate in different parts of the code.

Listing 11.2: Pseudocode for a task mode hybrid implementation of a 3D Jacobi solver.

```
!$OMP PARALLEL PRIVATE(iteration,threadID,k,j,i,...)
  threadID  = omp_get_thread_num()
  do iteration=1,MAXITER
  ...
  if(threadID .eq. 0) then
  ...
! Standard 3D Jacobi iteration
! updating BOUNDARY cells
  ...
! After updating BOUNDARY cells
! do halo exchange

    do dir=i,j,k
      call MPI_Irecv( halo data from neighbor in -dir direction )
      call MPI_Send( data to neighbor in +dir direction )
      call MPI_Irecv( halo data from neighbor in +dir direction )
      call MPI_Send( data to neighbor in -dir direction )
    enddo

    call MPI_Waitall( )

  else ! not thread ID 0

! Remaining threads perform
! update of INNER cells 2,...,N-1
! Distribute outer loop iterations manually:

    chunksize = (N-2) / (omp_get_num_threads()-1) + 1
    my_k_start = 2 + (threadID-1)*chunksize
    my_k_end = 2 + (threadID-1+1)*chunksize-1
    my_k_end = min(my_k_end, (N-2))

! INNER cell updates
    do k = my_k_start , my_k_end
      do j = 2, (N-1)
        do i = 2, (N-1)
          ...
        enddo
      enddo
    enddo
  endif ! thread ID
!$OMP BARRIER
  enddo
!$OMP END PARALLEL
```

Hence, the convenient OpenMP worksharing parallelization directives can no longer be used. Workload has to be distributed manually among the threads updating the inner cells (lines 28–31). Note that so far only the simplest incarnation of the task mode model has been presented. Depending on the thread level support of the MPI library one may also issue MPI calls on all threads in an OpenMP parallel region, further impeding programmability. Finally, it must be emphasized that this hybrid approach prevents incremental hybrid parallelization, because substantial parts of an existing MPI code need to be rewritten completely. It also severely complicates the maintainability of a pure MPI and a hybrid version in a single code.

11.1.3 Case study: Hybrid Jacobi solver

The MPI-parallel 3D Jacobi solver developed in Section 9.3 serves as a good example to evaluate potential benefits of hybrid programming for realistic application scenarios. To substantiate the discussion from the previous section, we now compare vector mode and a task mode implementations. Figure 11.1 shows performance data in MLUPs/sec for these two hybrid versions and the pure MPI variant using two different standard networks (Gigabit Ethernet and DDR InfiniBand). In order to minimize affinity and locality effects (cf. the extensive discussion in the next section), we choose the same single-socket dual-core cluster as in the pure MPI case study in Section 9.3.2. However, both cores per node are used throughout, which pushes the overall performance over InfiniBand by 10–15% as compared to Figure 9.11 (for the same large domain size of 480^3). Since communication overhead is still almost negligible when using InfiniBand, the pure MPI variant scales very well. It is no surprise that no benefit from hybrid programming shows up for the InfiniBand network and all three variants (dashed lines) achieve the same performance level.

For communication over the GigE network the picture changes completely. Even at low node counts the costs of data exchange substantially reduce parallel scalability for the pure MPI version, and there is a growing performance gap between GigE and InfiniBand. The vector mode does not help at all here, because computation and communication are still serialized; on the contrary, performance even degrades a bit since only one core on a node is active during MPI communication. Overlapping of communication and computation is efficiently possible in task mode, however. Parallel scalability is considerably improved in this case and GigE performance comes very close to the InfiniBand level.

This simple case study reveals the most important rule of hybrid programming: Consider going hybrid only if pure MPI scalability is not satisfactory. It does not make sense to work hard on a hybrid implementation and try to be faster than a perfectly scaling MPI code.

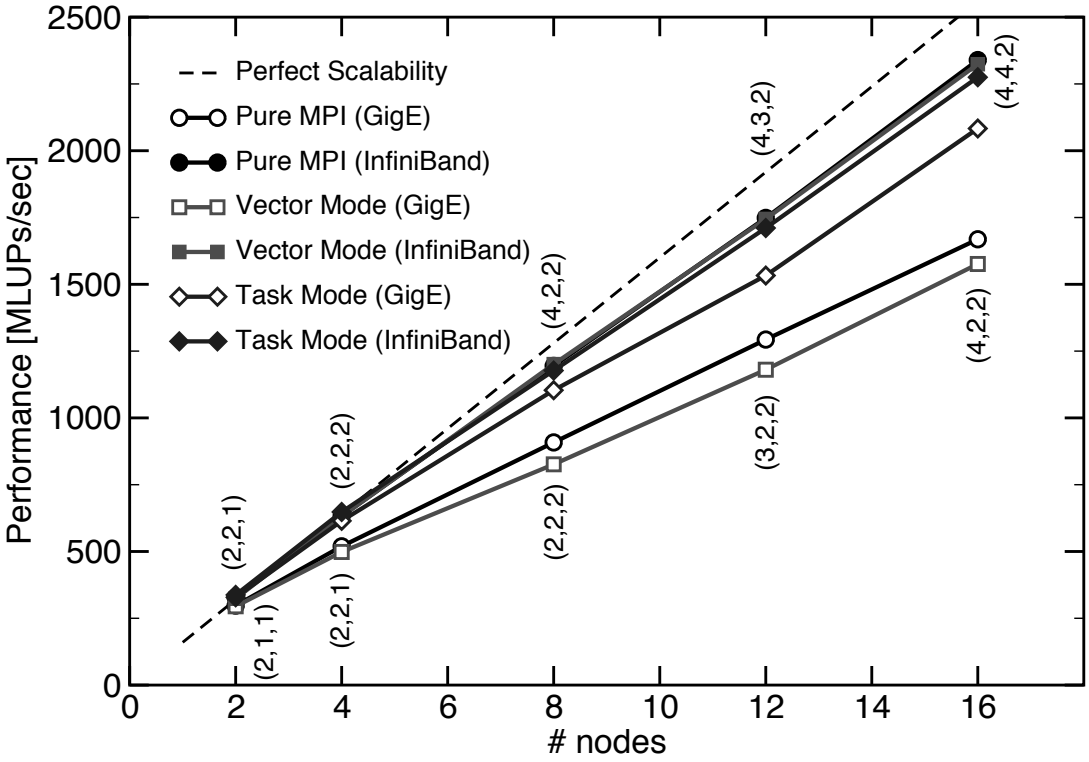


Figure 11.1: Pure MPI (circles) and hybrid (squares and diamonds) parallel performance of a 3D Jacobi solver for strong scaling (problem size 480^3) on the same single-socket dual-core cluster as in Figure 9.10, using DDR-InfiniBand (filled symbols) vs. Gigabit Ethernet (open symbols). The domain decomposition topology (number of processes in each Cartesian direction) is indicated at each data point (below for hybrid and above for pure MPI). See text for more details.

11.2 MPI taxonomy of thread interoperability

Moving from single-threaded to multithreaded execution is not an easy business from a communication library perspective as well. A fully "thread safe" implementation of an MPI library is a difficult undertaking. Providing shared coherent message queues or coherent internal message buffers are only two challenges to be named here. The most flexible case and thus the worst-case scenario for the MPI library is that MPI communication is allowed to happen on any thread at any time. Since MPI may be implemented in environments with poor or no thread support, the MPI standard currently (version 2.2) distinguishes four different levels of thread interoperability, starting with no thread support at all ("`MPI_THREAD_SINGLE`") and ending with the most general case ("`MPI_THREAD_MULTIPLE`"):

- `MPI_THREAD_SINGLE`: Only one thread will execute.
- `MPI_THREAD_FUNNELED`: The process may be multithreaded, but only the main thread will make MPI calls.

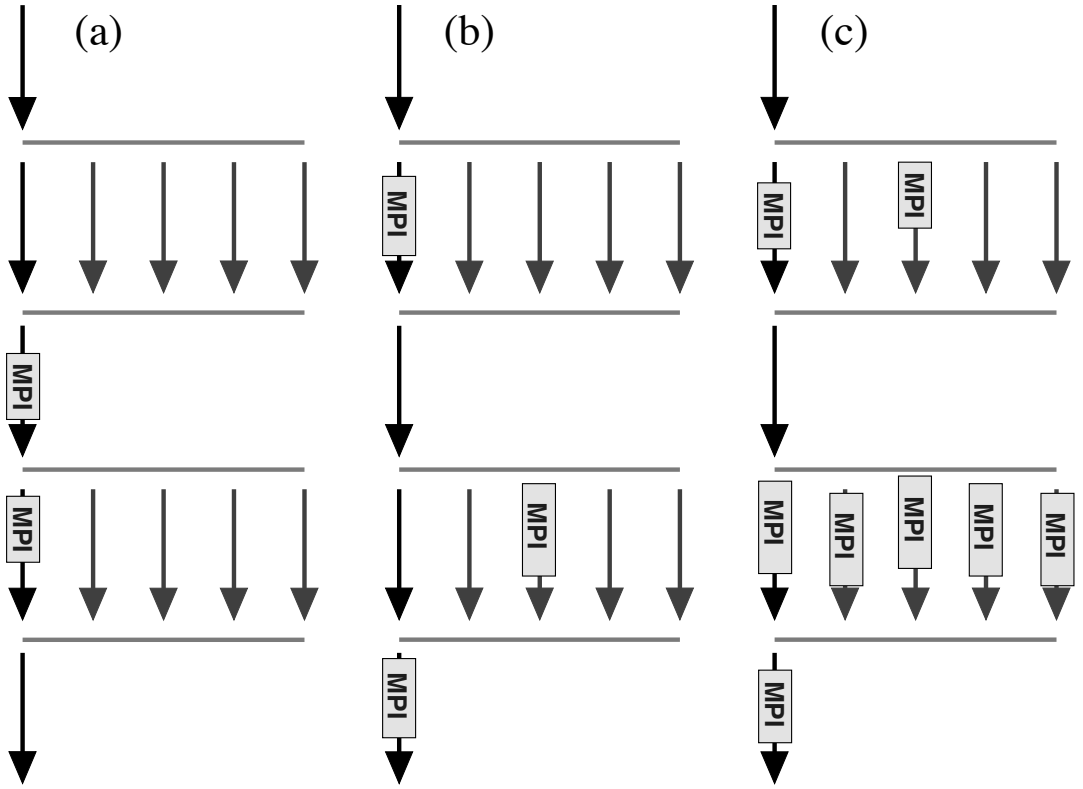


Figure 11.2: Threading levels supported by MPI: (a) `MPI_THREAD_FUNNELED`, (b) `MPI_THREAD_SERIALIZED`, and (c) `MPI_THREAD_MULTIPLE`. The plain MPI mode `MPI_THREAD_SINGLE` is omitted. Typical communication patterns as permitted by the respective level of thread support are depicted for a single multithreaded MPI process with a number of OpenMP threads.

- `MPI_THREAD_SERIALIZED`: The process may be multithreaded, and multiple threads may make MPI calls, but only one at a time; MPI calls are not made concurrently from two distinct threads.
- `MPI_THREAD_MULTIPLE`: Multiple threads may call MPI anytime, with no restrictions.

Every hybrid code should always check for the required level of threading support using the `MPI_Init_thread()` call. Figure 11.2 provides a schematic overview of the different hybrid MPI/OpenMP modes allowed by MPI's thread interoperability levels. Both hybrid implementations presented above for the parallel 3D Jacobi solver require MPI support for `MPI_THREAD_FUNNELED` since the master thread is the only one issuing MPI calls. The task mode version also provides first insights into the additional complications arising from multithreaded execution of MPI. Most importantly, MPI does not allow explicit addressing of different threads in a process. If there is a mandatory mapping between threads and MPI calls, the programmer has to implement it. This can be done through the explicit use of OpenMP thread IDs and potentially connecting them with different message tags (i.e., messages from different threads of the same MPI process are distinguished by unique MPI message tags).

Another issue to be aware of is that synchronous MPI calls only block the calling thread, allowing the other threads of the same MPI process to execute, if possible. There are several more important guidelines to consider, in particular when fully exploiting the `MPI_THREAD_MULTIPLE` capabilities. A thorough reading of the section "MPI and Threads" in the MPI standard document [P15] is mandatory when writing multithreaded MPI code.

11.3 Hybrid decomposition and mapping

Once a hybrid OpenMP/MPI code has been implemented diligently and computational resources have been allocated, two important decisions need to be made before launching the application.

First one needs to select the number of OpenMP threads per MPI process and the number of MPI processes per node. Of course the capabilities of the shared memory nodes at hand impose some limitations on this choice; e.g., the total number of threads per node should not exceed the number of cores in the compute node. In some rare cases it might also be advantageous to either run a single thread per "virtual core" if the processor efficiently supports simultaneous multithreading (SMT, see Section 1.5) or to even use less threads than available cores, e.g., if memory bandwidth or cache size per thread is a bottleneck. Moreover, the physical problem to be solved and the underlying hardware architecture also strongly influence the optimal choice of hybrid decomposition.

The mapping between MPI processes and OpenMP threads to sockets and cores within a compute node is another important decision. In this context the basic node characteristics (number of sockets and number of cores per socket) can be used as a first guideline, but even on rather simple two-socket compute nodes with multicore processor chips there is a large parameter space in terms of decomposition and mapping choices. In Figures 11.3–11.6 a representative subset is depicted for a cluster with two standard two-socket quad-core ccNUMA nodes. We imply that the MPI library supports the `MPI_THREAD_FUNNELED` level, where the master thread (t_0) of each MPI process assumes a prominent position. This is valid for the two examples presented above and reflects the approach implemented in many hybrid applications.

One MPI process per node

Considering the shared-memory feature only, one can simply assign a single MPI process to one node (m_0, m_1) and launch eight OpenMP threads ($t_0, \ldots, t_7$), i.e., one per core (see Figure 11.3). There is a clear asymmetry between the hardware design and the hybrid decomposition, which may show up in several performance-relevant issues. Synchronization of all threads is costly since it involves off-chip data exchange and may become a major bottleneck when moving to multisocket and/or hexa-/octo-core designs [M41] (see Section 7.2.2 for how to estimate synchronization overhead in OpenMP). NUMA optimizations (see Chapter 8) need to

be considered when implementing the code; in particular, the typical locality and contention issues can arise if the MPI process (running, e.g., in LD0) allocates substantial message buffers. In addition the master thread may generate nonlocal data accesses when gathering data for MPI calls. Using less powerful cores, a single MPI process per node could also be insufficient to make full use of the latest interconnect technologies if the available internode bandwidth cannot be saturated by a single MPI process [O69]. The ease of launching the MPI processes and pinning the threads, as well as the reduction of the number of MPI processes to a minimum are typical advantages of this simple hybrid decomposition model.

One MPI process per socket

Assigning one multithreaded MPI process to each socket matches the node topology perfectly (see Figure 11.4). However, correctly launching the MPI processes and pinning the OpenMP threads in a blockwise fashion to the sockets requires due care. MPI communication may now happen both between sockets and between nodes concurrently, and appropriate scheduling of MPI calls should be considered in the application to overlap intersocket and internode communication [O72]. On the other hand, this mapping avoids ccNUMA data locality problems because each MPI process is restricted to a single locality domain. Also the accessibility of a single shared cache for all threads of each MPI process allows for fast thread synchronization and increases the probability of cache re-use between the threads. Note that this discussion needs to be generalized to groups of cores with a shared outer-level cache: In the first generation of Intel quad-core chips, groups of two cores shared an L2 cache while no L3 cache was available. For this chip architecture one should issue one MPI process per L2 cache group, i.e., two processes per socket.

A small modification of the mapping can easily emerge in a completely different scenario without changing the number of MPI processes per node and the number of OpenMP threads per process: If, e.g., a round-robin distribution of threads across sockets is chosen, one ends up in the situation shown in Figure 11.5. In each node a single socket hosts two MPI processes, potentially allowing for very fast communication between them via the shared cache. However, the threads of different MPI processes are interleaved on each socket, making efficient ccNUMA-aware programming a challenge by itself. Moreover, a completely different workload characteristic is assigned to the two sockets within the same node. All this is certainly close to a worst-case scenario in terms of thread synchronization and remote data access.

Multiple MPI processes per socket

Of course one can further increase the number of MPI Processes per node and correspondingly reduce the number of threads, ending up with two threads per MPI process. The choice of a block-wise distribution of the threads leads to the favorable scenario presented in Figure 11.6. While ccNUMA problems are of minor importance here, MPI communication may show up on all potential levels: intrasocket, intersocket and internode. Thus, mapping the computational domains to the MPI processes in a way that minimizes access to the slowest communication path is a

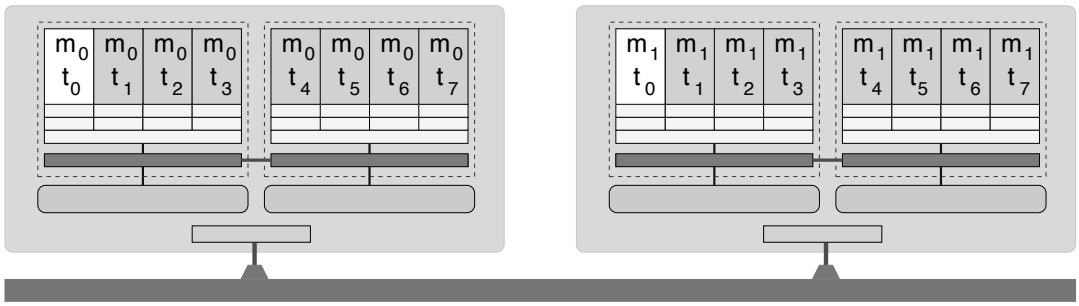


Figure 11.3: Mapping a single MPI process with eight threads to each node.

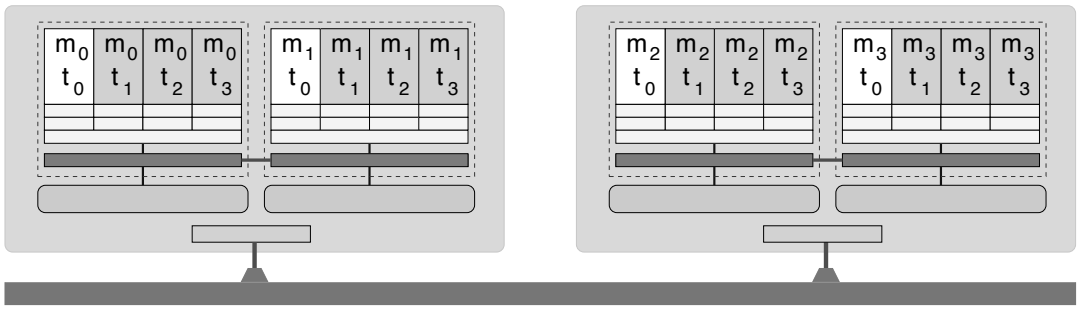


Figure 11.4: Mapping a single MPI process with four threads to each socket (L3 group or locality domain).

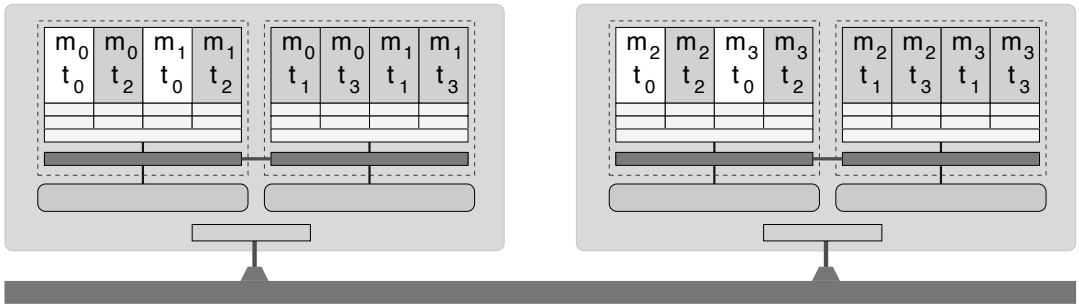


Figure 11.5: Mapping two MPI processes to each node and implementing a round-robin thread distribution.

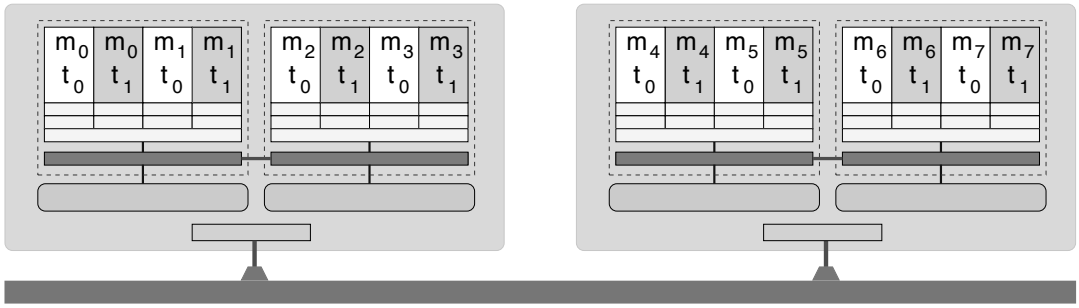


Figure 11.6: Mapping two MPI processes with two threads each to a single socket.

potential optimization strategy. This decomposition approach can also be beneficial for memory-intensive codes with limited MPI scalability. Often half of the threads are already able to saturate the main memory bandwidth of a single socket and the use of a multithreaded MPI process can add a bit to the performance gain from the smaller number of MPI processes. Alternatively a functional decomposition can be employed to explicitly hide MPI communication (see the task mode implementation of the Jacobi solver described above). It is evident that a number of different mapping strategies is available for this decomposition as well. However, due to the symmetry arguments stressed several times above, the mapping in Figure 11.6 should generally provide best performance and thus be tested first.

Unfortunately, at the time of writing most MPI implementations only provide very poor support for defining different mapping strategies and pinning the threads correctly to the respective cores. If there is some support, it is usually restricted to very specific combinations of MPI and compiler. Hence, the correct launch of a hybrid application is mostly the programmer's responsibility, but indispensable for a profound performance study of hybrid codes. See Section 10.4.1 and Appendix A for further information on mapping and affinity issues.

11.4 Potential benefits and drawbacks of hybrid programming

The hybrid parallel MPI/OpenMP approach is, for reasons mentioned earlier, still rarely implemented to its full extent in parallel applications. Thus, it is not surprising that there is no complete theory available for whether a hybrid approach does pay back the additional costs for code restructuring or designing a complete hybrid application from scratch. However, several potential fundamental advantages and drawbacks of the hybrid approach as compared to pure MPI have been identified so far. In the following we briefly summarize the most important ones. The impact of each topic below will most certainly depend on the specific application code or even on the choice of input data. A careful investigation of those issues is mandatory if *and only if* pure MPI scalability does not provide satisfactory parallel performance.

Improved rate of convergence

Many iterative solvers incorporate loop-carried data dependencies like, e.g., the well-known lexicographical Gauss–Seidel scheme (see Section 6.3). Those dependencies are broken at boundary cells if standard domain decomposition is used for MPI parallelization. While the algorithm still converges to the correct steady-state solution, the rate of convergence typically drops with increasing number of subdomains (for strong scaling scenarios). Here a hybrid approach may help reduce the number of subdomains and improve the rate of convergence. This was shown, e.g., for a CFD application with an implicit solver in [A90]: Launching only a single MPI process per node (computing on a single subdomain) and using OpenMP within the node improves convergence in the parallel algorithm. This is clearly a case where parallel

speedups or overall floating-point performance are the wrong metrics to quantify the "hybrid dividend." Walltime to solution is the appropriate metric instead.

Re-use of data in shared caches

Using a shared-memory programming model for threads operating on a single shared cache greatly extends the optimization opportunities: For iterative solvers with regular stencil dependencies like the Jacobi or Gauss–Seidel type, data loaded and modified by a first thread can be read by another thread from cache and updated once again before being evicted to main memory. This trick leads to an efficient and natural parallel temporal blocking, but requires a shared address space to avoid double buffering of in-cache data and redundant data copying (as would be enforced by a pure MPI implementation) [O52, O53, O63]. Hybrid parallelization is mandatory to implement such alternative multicore aware strategies in large-scale parallel codes.

Exploiting additional levels of parallelism

Many computational tasks provide a problem-immanent coarse-grained parallel level. One prominent example are some of the multizone NAS parallel benchmarks, where only a rather small number of zones are available for MPI parallelization (from a few dozens up to 256) and additional parallel levels can be exploited by multithreaded execution of the MPI process [O70]. Potential load imbalance on these levels can also be addressed very efficiently within OpenMP by its flexible loop scheduling variants (e.g., "guided" or "dynamic" in OpenMP).

Overlapping MPI communication and computation

MPI offers the flexibility to overlap communication and computation by issuing nonblocking MPI communication, doing some computations and afterwards checking the MPI calls for completion. However, as described in Section 10.4.3, most MPI libraries today do not perform truly asynchronous transfers even if nonblocking calls are used. If MPI progress occurs only if library code is executed (which is completely in line with the MPI standard), message-passing overhead and computations are effectively serialized. As a remedy, the programmer may use a single thread within an MPI process to perform MPI communication asynchronously. See Section 11.1.3 above for an example.

Reducing MPI overhead

In particular for strong scaling scenarios the contribution to the overall runtime introduced by MPI communication overhead may increase rapidly with the number of MPI processes. Again, the domain decomposition approach can serve as a paradigm here. With increasing process count the ratio of local subdomain surface (communication) and volume (computation) gets worse (see Section 10.4) and at the same time the average message size is reduced. This can decrease the effective communication bandwidth as well (see also Problem 10.5 about "riding the Ping-Pong curve"). Also the overall amount of buffer space for halo layer exchange increases

with the number of MPI processes and may use a considerable amount of main memory at large processor counts. Reducing the number of MPI processes through hybrid programming may help to increase MPI message lengths and reduce the overall memory footprint.

Multiple levels of overhead

In general, writing a truly efficient and scalable OpenMP program is entirely nontrivial, despite the apparent simplicity of the incremental parallelization approach. We have demonstrated some of the potential pitfalls in Chapter 7. On a more abstract level, introducing a second parallelization level into a program also brings in a new layer on which fundamental scalability limits like, e.g., Amdahl's Law must be considered.

Bulk-synchronous communication in vector mode

In hybrid vector mode, all MPI communication takes place outside OpenMP-parallel regions. In other words, all data that goes into and out of a multithreaded process is only transferred after all threads have synchronized: Communication is *bulk-synchronous* on the node level, and there is not a chance that one thread still does useful work while MPI progress takes place (except if truly asynchronous message passing is supported; see Section 10.4.3 for more information). In contrast, several MPI processes that share a network connection can use it at all possible times, which often leads to a natural overlap of computation and communication, especially if eager delivery is possible. Even if the pure MPI program is bulk-synchronous as well (like, e.g., the MPI-parallel Jacobi solver shown in Section 9.3), little variations in runtime between the different processes can cause at least a partial overlap.