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].

Downloaded by [Yale University] at 06:31 23 July 2014

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.

Downloaded by [Yale University] at 06:31 23 July 2014

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
```

Downloaded by [Yale University] at 06:31 23 July 2014

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

Downloaded by [Yale University] at 06:31 23 July 2014

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.

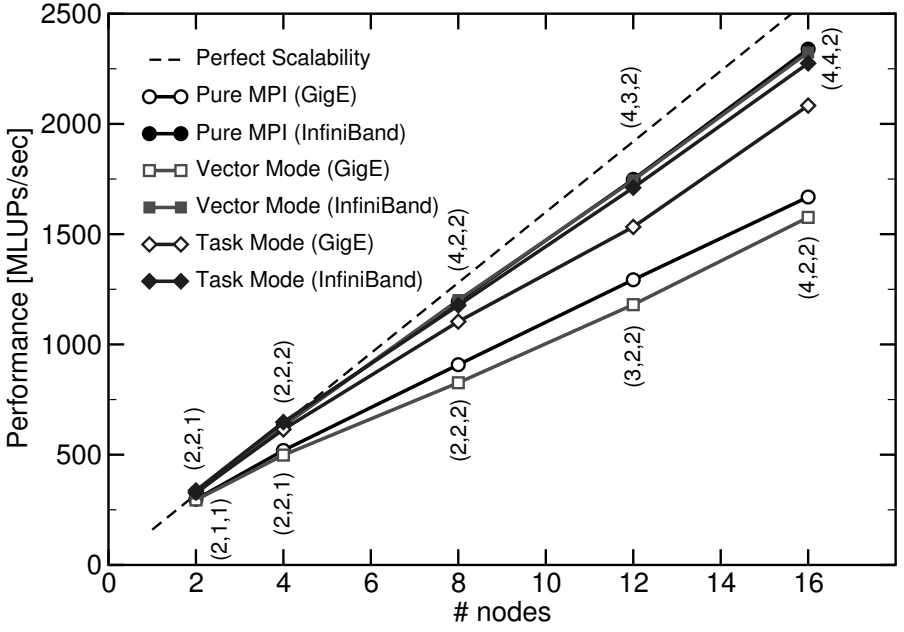

Figure 11.1: Pure MPI (circles) and hybrid (squares and diamonds) parallel performance of a 3D Jacobi solver for strong scaling (problem size 480^3) on the same single-socket dual-core cluster as in Figure 9.10, using DDR-InfiniBand (filled symbols) vs. Gigabit Ethernet (open symbols). The domain decomposition 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.

Downloaded by [Yale University] at 06:31 23 July 2014

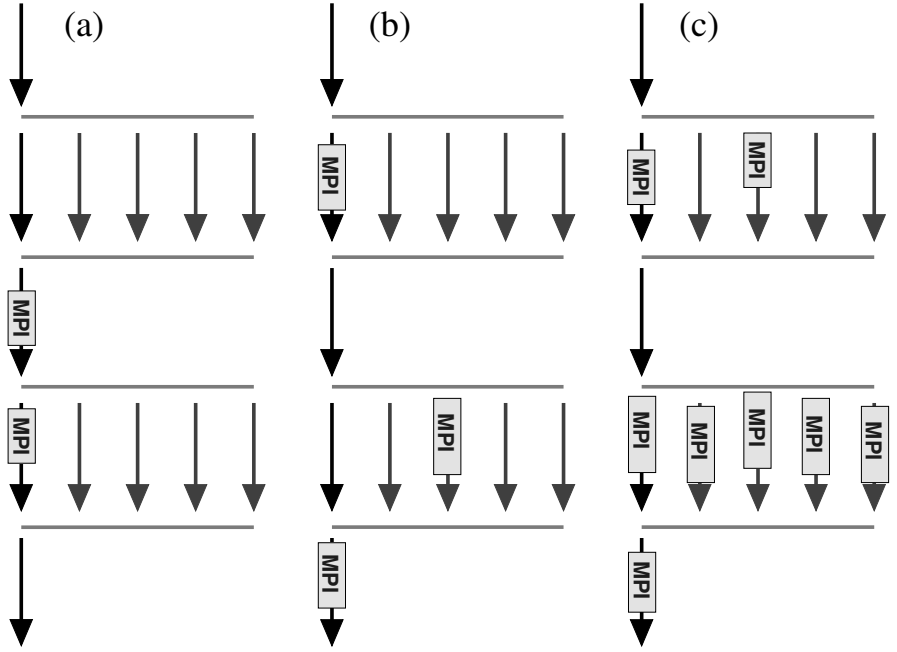

Figure 11.2: Threading levels supported by MPI: (a) `MPI_THREAD_FUNNELED`, (b) `MPI_THREAD_SERIALIZED`, and (c) `MPI_THREAD_MULTIPLE`. The plain MPI mode `MPI_THREAD_SINGLE` is omitted. Typical communication patterns as permitted by the respective level of thread support are depicted for a single multithreaded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

Downloaded by [Yale University] at 06:31 23 July 2014

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

Downloaded by [Yale University] at 06:31 23 July 2014

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

Downloaded by [Yale University] at 06:31 23 July 2014

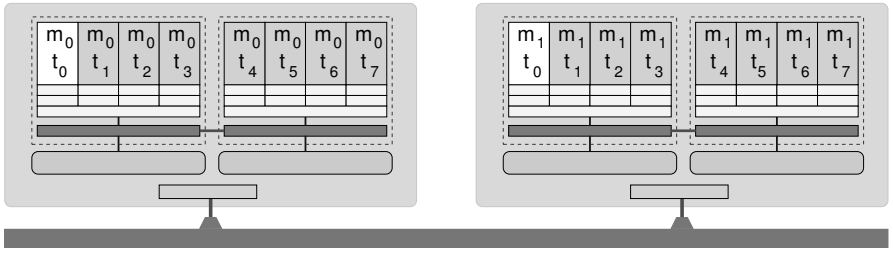

Figure 11.3: Mapping a single MPI process with eight threads to each node.

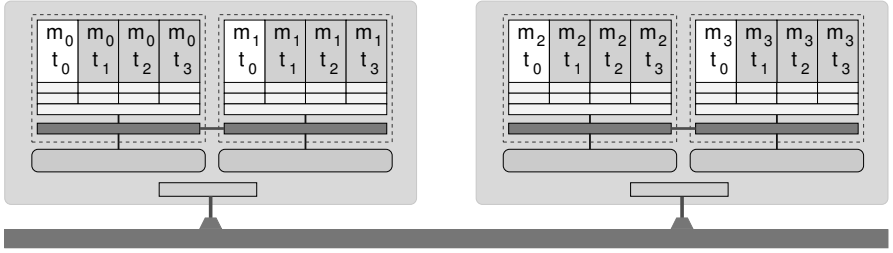

Figure 11.4: Mapping a single MPI process with four threads to each socket (L3 group or locality domain).

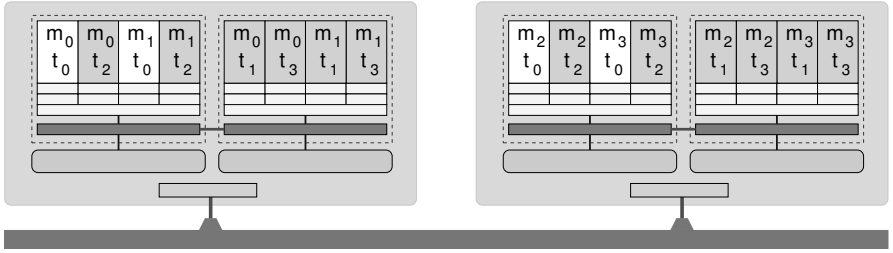

Figure 11.5: Mapping two MPI processes to each node and implementing a round-robin thread distribution.

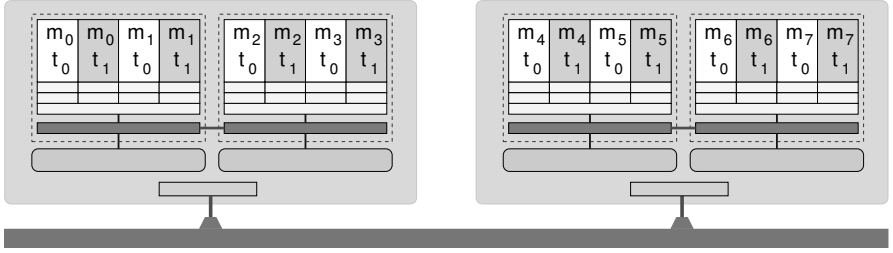

Figure 11.6: Mapping two MPI processes with two threads each to a single socket.

Downloaded by [Yale University] at 06:31 23 July 2014

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.

Downloaded by [Yale University] at 06:31 23 July 2014

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.

Downloaded by [Yale University] at 06:31 23 July 2014

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.

Downloaded by [Yale University] at 06:31 23 July 2014