Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes

Georg Hager
Erlangen Regional Computing Center (RRZE), Germany
georg.hager@rrze.uni-erlangen.de

Gabriele Jost
Texas Advanced Computing Center (TACC), Austin, TX
gjost@tacc.utexas.edu

Rolf Rabenseifner
High Performance Computing Center Stuttgart (HLRS), Germany
rabenseifner@hlrs.de

Abstract

Hybrid MPI/OpenMP and pure MPI on clusters of multi-core SMP nodes involve several mismatch problems between the parallel programming models and the hardware architectures. Measurements of communication characteristics between cores on the same socket, on the same SMP node, and between SMP nodes on several platforms (including Cray XT4 and XT5) show that machine topology has a significant impact on performance for all parallelization strategies and that topology awareness should be built into all applications in the future. We describe potentials and challenges of the dominant programming models on hierarchically structured hardware. Case studies with the multi-zone NAS parallel benchmarks on several platforms demonstrate the opportunities of hybrid programming.

1. Mainstream HPC architecture

Today scientists who wish to write efficient parallel software for high performance systems have to face a highly hierarchical system design, even (or especially) on "commodity" clusters (Fig. 1 (a)). The price/performance sweet spot seems to have settled at a point where multi-socket multi-core shared-memory compute nodes are coupled via high-speed interconnects. Inside the node, details like UMA (Uniform Memory Access) vs. ccNUMA (cache coherent Non-Uniform Memory Access) characteristics, number of cores per socket and/or ccNUMA domain, shared and separate caches, or chipset and I/O bottlenecks complicate matters further. Communication between nodes usually shows a rich set of performance characteristics because global, non-blocking communication has grown out of the affordable range.

This trend will continue into the foreseeable future, broadening the available range of hardware designs even when looking at high-end systems. Consequently, it seems natural to employ a hybrid programming model which uses OpenMP for parallelization inside the node and MPI for message passing between nodes. However, there is always the option to use pure MPI and treat every CPU core as a separate entity with its own address space. And finally, looking at the multitude of hierarchies mentioned above, the question arises whether it might be advantageous to employ a "mixed model" where more than one MPI process with multiple threads runs on a node so that there is at least some explicit intra-node communication (Fig. 1 (b)–(d)).

It is not a trivial task to determine the optimal model to use for some specific application. There seems to be a general lore that pure MPI can often outperform hybrid, but counterexamples do exist and results tend to vary with input data, problem size etc. even for a given code [1]. This paper discusses potential reasons for this; in order to get optimal scalability one should in any case try to implement the following strategies: (a) Reduce synchronization overhead (see Sect. 3.5), (b) reduce load imbalance (Sect. 4.2), (c) reduce computational overhead and memory consumption (Sect. 4.3), and (d) Minimize MPI communication overhead (Sect. 4.4).

There are some strong arguments in favor of a hybrid model which tend to underline the assumption that it should lead to improved parallel efficiency as compared to pure MPI. In the following sections we will shed some light on

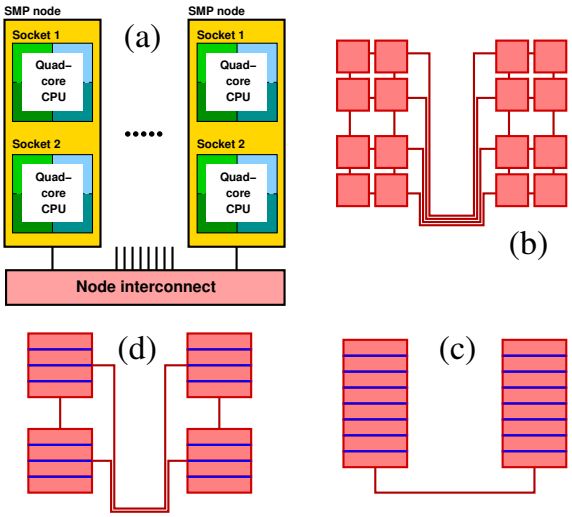

Figure 1. A typical multi-socket multi-core SMP cluster (a), and three possible parallel programming models that can be mapped onto it: (b) pure MPI, (c) fully hybrid MPI/OpenMP, (d) mixed model with more than one MPI process per node.

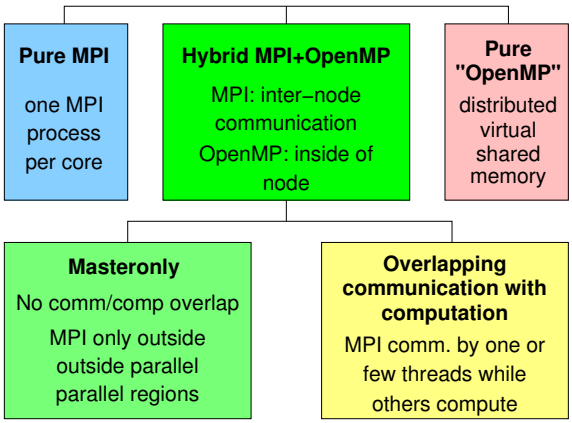

Figure 2. Taxonomy of parallel programming models on hybrid platforms.

most of these statements and discuss their validity.

This paper is organized as follows: In Sect. 2 we outline the available programming models on hybrid/hierarchical parallel platforms, briefly describing their main strengths and weaknesses. Sect. 3 concentrates on mismatch problems between parallel models and the parallel hardware: Insufficient topology awareness of parallel runtime environments, issues with intra-node message passing, and suboptimal network saturation. The additional complications that arise from the necessity to optimize the OpenMP part of a hybrid code are discussed in Sect. 3.5. In Sect. 4 we then turn to the benefits that may be expected from employing hybrid parallelization. In the final sections we address possible future developments in standardization which could help address some of the problems described and close with a summary.

2. Parallel programming models on hybrid platforms

Fig. 2 shows a taxonomy of parallel programming models on hybrid platforms. We have added an "OpenMP only" branch because "distributed virtual shared memory" technologies like Intel Cluster OpenMP [2] allow the use of OpenMP-like parallelization even beyond the boundaries of a single cluster node. See Sect. 2.4 for more information. This overview ignores the details about how exactly the threads and processes of a hybrid program are to be mapped onto hierarchical hardware. The *mismatch problems* which are caused by the various alternatives to perform this mapping are discussed in detail in Sect. 3.

When using any combination of MPI and OpenMP, the MPI implementation must feature some kind of threading support. The MPI-2.1 standard defines the following levels:

- `MPI_THREAD_SINGLE`: Only one thread will execute.
- `MPI_THREAD_FUNNELED`: The process may be multi-threaded, but only the main thread will make MPI calls.
- `MPI_THREAD_SERIALIZED`: The process may be multi-threaded, 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, with no restrictions.

Any hybrid code should always check for the required level of threading support using the `MPI_Thread_init()` call.

2.1. Pure MPI

From a programmer's point of view, pure MPI ignores the fact that cores inside a single node work on shared memory. It can be employed right away on the hierarchical systems discussed above (see Fig. 1 (b)) without changes to existing code. Moreover, it is not required for the MPI library and underlying software layers to support multi-threaded applications, which simplifies implementation (Optimizations on the MPI level regarding the inner topology of the node interconnect, e.g., fat tree or torus, may still be useful or necessary).

On the other hand, a pure MPI programming model implicitly assumes that message passing is the correct paradigm to use for all levels of parallelism available in the application and that the application "topology" can be mapped efficiently to the hardware topology. This may not be true in all cases, see Sect. 3 for details. Furthermore, all communication between processes on the same node goes through the MPI software layers, which adds to overhead. Hopefully the library is able to use "shortcuts" via shared memory in this case, choosing ways of communication that effectively use shared caches, hardware assists for global operations, and the like. Such optimizations are usually out of the programmer's influence, but see Sect. 5 for some discussion regarding this point.

2.2. Hybrid masteronly

The hybrid masteronly model uses one MPI process per node and OpenMP on the cores of the node, with no MPI calls inside parallel regions. A typical iterative domain decomposition code could look like the following:

```
for (iteration = 1...N)
{
  #pragma omp parallel
  {
    /* numerical code */
  }
  /* on master thread only */
  MPI_Send(bulk data to halo areas in other nodes)
  MPI_Recv(halo data from the neighbors)
}
```

This resembles parallel programming on distributed-memory parallel vector machines. In that case, the inner layers of parallelism are not exploited by OpenMP but by vectorization and multi-track pipelines.

As there is no intra-node message passing, MPI optimizations and topology awareness for this case are not required. Of course, the OpenMP parts should be optimized for the topology at hand, e.g., by employing parallel first-touch initialization on ccNUMA nodes or using thread-core affinity mechanisms.

There are, however, some major problems connected with masteronly mode:

- All other threads are idle during communication phases of the master thread which could lead to a strong impact of communication overhead on scalability. Alternatives are discussed in Sect. 3.1.3 and Sect. 3.3 below.
- The full inter-node MPI bandwidth might not be saturated by using a single communicating thread.
- The MPI library must be thread-aware on a simple level by providing `MPI_THREAD_FUNNELED`. Actually, a lower thread-safety level would suffice for masteronly, but the MPI-2.1 standard does not provide an appropriate level less than `MPI_THREAD_FUNNELED`.

2.3. Hybrid with overlap

One way to avoid idling compute threads during MPI communication is to split off one or more threads of the OpenMP team to handle communication in parallel with useful calculation:

```
if (my_thread_ID < ...) {
  /* communication threads: */
  /* transfer halo */
  MPI_Send( halo data )
  MPI_Recv( halo data )
} else {
  /* compute threads: */
  /* execute code that does not need halo data */
}
/* all threads: */
/* execute code that needs halo data */
```

A possible reason to use more than one communication thread could arise if a single thread cannot saturate the full communication bandwidth of a compute node (see Sect. 3.3 for details). There is, however, a trade-off because the more threads are sacrificed for MPI, the fewer are available for overlapping computation.

2.4. Pure OpenMP on clusters

A lot of research has been invested into the implementation of distributed virtual shared memory software [3] which allows near-shared-memory programming on distributed memory parallel machines, notably clusters. Since 2006 Intel offers the "Cluster OpenMP" compiler add-on, enabling the use of OpenMP (with minor restrictions) across the nodes of a cluster [2]. Therefore, OpenMP has literally become a possible programming model for those machines. It is, to some extent, a hybrid model, being identical to plain OpenMP inside a shared-memory node but employing a sophisticated protocol that keeps "shared" memory pages coherent between nodes at explicit or automatic OpenMP flush points.

With Cluster OpenMP, frequent page synchronization or erratic access patterns to shared data must be avoided by all means. If this is not possible, communication can potentially become much more expensive than with plain MPI.

3. Mismatch problems

It should be evident by now that the main issue with getting good performance on hybrid architectures is that none of the programming models at one's disposal fits optimally to the hierarchical hardware. In the following sections we will elaborate on these *mismatch problems*. However, as sketched above, one can also expect hybrid models to have positive effects on parallel performance (as shown in Sect. 4). Most hybrid applications suffer from the former and benefit from the latter to varying degrees, thus it is near to impossible to make a quantitative judgement without thorough benchmarking.

3.1. The mapping problem: Machine topology

As a prototype mismatch problem we consider the mapping of a two-dimensional Cartesian domain decomposition with 80 sub-domains, organized in a 5×16 grid, on a ten-node dual-socket quad-core cluster like the one in Fig. 1 (a). We will analyze the communication behavior of this application with respect to the required inter-socket and inter-node halo exchanges, presupposing that inter-core communication is fastest, hence favorable. See Sect. 3.2 for a discussion on the validity of this assumption.

3.1.1. Mapping problem with pure MPI

We assume here that the MPI start mechanism is able to establish some affinity between processes and cores, i.e. it is not left to chance which rank runs on which core of a node. However, defaults vary across implementations. Fig. 3 shows that there is an immense difference between sequential and round-robin ranking, which is reflected in the number of required inter-node and inter-socket connections. In Fig. 3 (a), ranks are mapped to cores, sockets and nodes (A…J) in sequential order, i.e., ranks 0…7 go to the first node, etc.. This leads at maximum to 17 inter-node and one inter-socket halo exchanges per node, neglecting boundary effects. If the default is to place MPI ranks in round-robin order across nodes (Fig. 3 (b)), i.e., ranks 0…9 are mapped to the first core of each node, all the halo communication uses inter-node connections, which leads to 32 inter-node and no inter-socket exchanges. Whether the difference matters or not depends, of course, on the ratio of computational effort versus amount of halo data, both per process, and the characteristics of the network.

What is the best ranking order for the domain decomposition at hand? It is important to realize that the hierarchical node structure enforces *multilevel domain decomposition* which can be optimized for minimizing inter-node communication: It seems natural to try to reduce the socket "surface area" exposed to the node boundary, as shown in

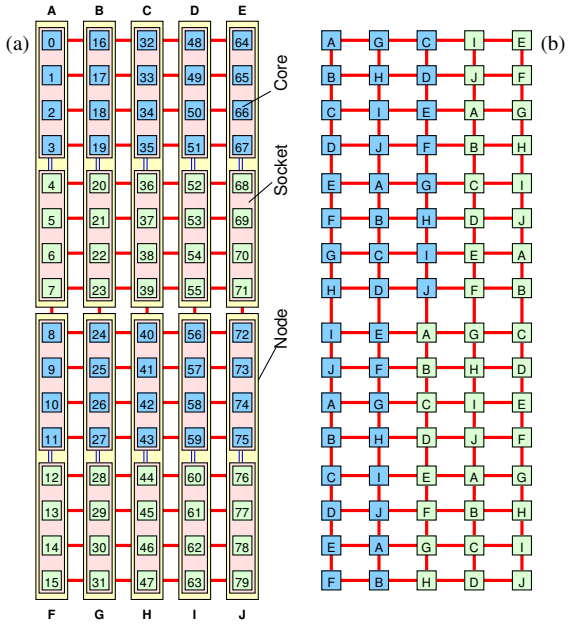

Figure 3. Influence of ranking order on the number of inter-socket (double lines, blue) and inter-node (single lines, red) halo communications when using pure MPI. (a) Sequential mapping, (b) Round-robin mapping.

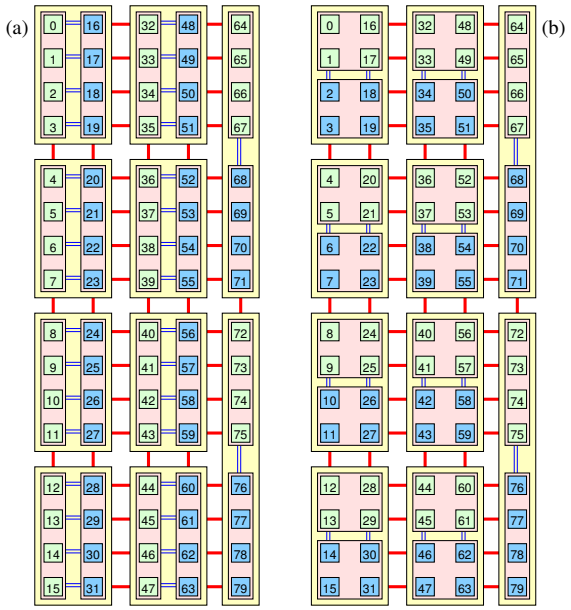

Figure 4. Two possible mappings for multilevel domain decomposition with pure MPI.

Fig. 4 (a), which yields ten inter-node and four inter-socket halo exchanges per node at maximum. But still there is optimization potential, because this process can be iterated to the socket level (Fig. 4 (b)), cutting the number of inter-socket connections in half. Comparing Figs. 3 (a), (b) and Figs. 4 (a), (b), this is the best possible rank order for pure MPI.

Above considerations should make it clear that it can be vital to know about the default rank placement used in a particular parallel environment and modify it if required. Unfortunately, many commodity clusters are still run today without a clear concept about rank-core affinity and even no way to influence it on a user-friendly level.

3.1.2. Mapping problem with fully hybrid MPI+OpenMP

Hybrid MPI+OpenMP enforces the domain decomposition to be a two-level algorithm. On MPI level, a coarse-grained domain decomposition is performed. Parallelization on OpenMP level implies a second level domain decomposition, which may be implicit (loop level parallelization) or explicit as shown in Fig. 5.

In principle, hybrid MPI+OpenMP presents similar challenges in terms of topology awareness, i.e. optimal rank/thread placement, as pure MPI. There is, however, the added complexity that standard OpenMP parallelization is based on loop-level worksharing, which is, albeit easy to apply, not always the optimal choice. On ccNUMA systems, for instance, it might be better to drop the worksharing concept in favor of thread-level domain decomposition in order to reduce inter-domain NUMA traffic (see below). On top of this, proper first-touch page placement is required to get scalable bandwidth inside a node, and thread-core affinity must be employed. Still one should note that those issues are not specific to hybrid MPI+OpenMP programming but apply to pure OpenMP as well.

In contrast to pure MPI, hybrid parallelization of above domain decomposition enforces a 2×5 MPI domain grid, leading to oblong OpenMP subdomains (if explicit domain decomposition is used on this level, see Fig. 5). Optimal rank ordering leads to only three inter-node halo exchanges per node, but each with about four times the data volume. Thus we arrive at a slightly higher communication effort compared to pure MPI (with optimal rank order), a consequence of the non-square domains.

Beyond the requirements of hybrid MPI+OpenMP, multi-level domain decomposition may be beneficial when taking cache optimization into account: On the outermost level the domain is divided into subdomains, one for each MPI process. On the next level, these are again split into portions for each thread, and then even further to fit into successive cache levels (L3, L2, L1). This strategy ensures maximum access locality, a minimum of cache misses,

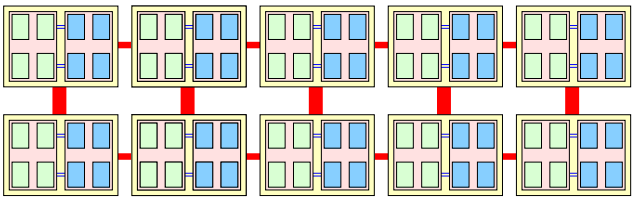

Figure 5. Hybrid OpenMP+MPI two-level domain decomposition with a 2×5 MPI domain grid and eight OpenMP threads per node. Although there are fewer inter-node connections than with optimal MPI rank order (see Fig. 4 (b)), the aggregate halo size is slightly larger.

NUMA traffic, and inter-node communication, but it must be performed by the application, especially in the case of unstructured grids. For portable software development, standardized methods are desirable for the application to detect the system topology and characteristic sizes (see also Sect. 5).

3.1.3. Mapping problem with mixed model

The mixed model (see Fig. 1 (d)) represents a sort of compromise between pure MPI and fully hybrid models, featuring potential advantages in terms of network saturation (see Sect. 3.3 below). It suffers from the same basic drawbacks as the fully hybrid model, although the impact of a loss of thread-core affinity may be larger because of the possibly significant differences in OpenMP performance and, more importantly, MPI communication characteristics for intra-node message transfer. Fig. 6 shows a possible scenario where we contrast two alternatives for thread placement. In Fig. 6 (a), intra-node MPI uses the inter-socket connection only and shared memory access with OpenMP is kept inside of each multi-core socket, whereas in Fig. 6 (b) all intra-node MPI (with masteronly style) is handled inside sockets. However, due to the spreading of the OpenMP threads belonging to a particular process across two sockets there is the danger of increased OpenMP startup overhead (see Sect. 3.5) and NUMA traffic.

As with pure MPI, the message-passing subsystem should be topology-aware in the sense that optimization opportunities for intra-node transfers are actually exploited. The following section provides some more information about performance characteristics of intra-node versus inter-node MPI.

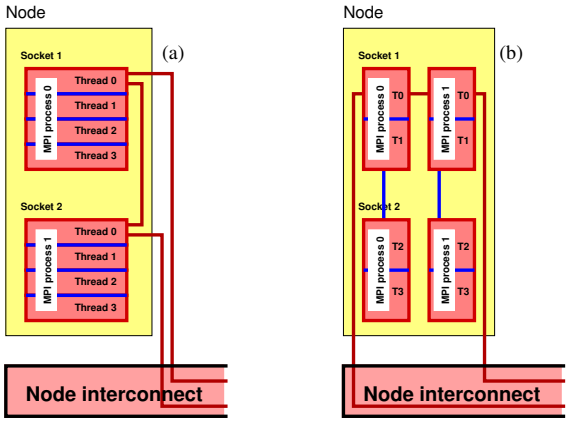

Figure 6. Two different mappings of threads to cores for the mixed model with two MPI processes per eight-core, two-socket node.

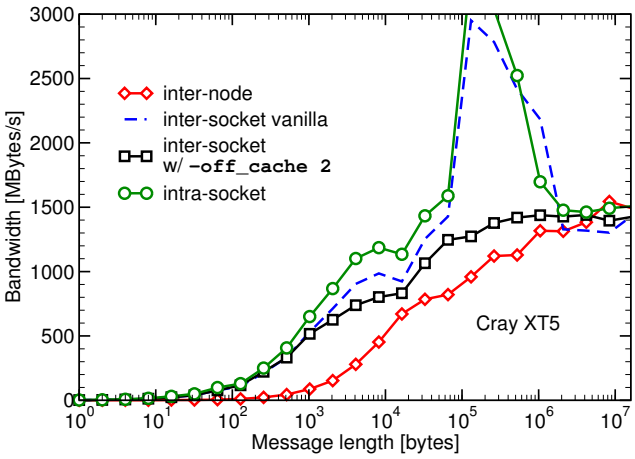

Figure 7. IMB PingPong bandwidth versus message size for inter-node, inter-socket, and intra-socket communication on an XT5. The "vanilla" variant (dashed line) uses no special options and shows pathological behavior for intermediate message sizes (see text).

3.2. Issues with intra-node MPI communication

The question whether the benefits or disadvantages of different hybrid programming models in terms of communication behavior really impact application performance cannot be answered in general since there are far too many parameters involved. Even so, knowing the characteristics of the MPI system at hand, one may at least arrive at an educated guess. As an example we choose the well-known PingPong benchmark from the Intel MPI benchmark (IMB) suite, performed on a Cray XT5 at ARSC [17] (see Sect. 4.1 for more information on the system). Fig. 7 shows that there are significant differences in achievable bandwidths for message sizes below 2 MB, which happens to be one processor's L3 cache size. If the two processes run on the same socket (circles), the shared L3 cache provides superior communication bandwidth until the aggregate message sizes exceed the cache size. For inter-socket communication (squares), the "revolving MPI buffers" feature of the IMB must be activated (using the `-off_cache` command line option) in order to avoid meaningless results [7]. The `-off_cache` option forces the use of a new send buffer on each `MPI_Send()` until the aggregate size of all buffers exceeds a configurable limit (the option's parameter). Choosing the limit to be at least the L3 cache size ensures 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. If the option is omitted (dashed line), the first `MPI_Send()` writes the message to the cache of the processor it is executed on. Subsequent `MPI_Send()` calls (which are always performed with PingPong to get accurate timing measurements) then perform in-cache copy operations. As the receive buffer is never used by the other process, it never gets evicted to main memory, leading to similar measurements as in the intra-socket case. This is a situation that would never occur in a real application.

All intra-node and inter-node performance numbers become identical at message lengths above 2 MB. This may be surprising because memory bandwidth as measured with, e.g., STREAM benchmarks on the AMD Barcelona processor yields results beyond 5 GB/s for a single core. However, if we assume that a single intermediate buffer is used for intra-node point-to-point transfers, the two required copy operations send the message between four and six times over the processor bus(es) (depending on whether non-temporal moves are employed or not [8]), which easily explains at least the observed order of magnitude. The slight disadvantage with inter-socket communication may be attributed to the unavoidable NUMA traffic across the HyperTransport links in this case.

This behavior is not only typical for the XT5 but also for commodity cluster systems, see Figs. 8 and 9 [4, 13, 15]. It shows that simplistic assumptions about superior performance of intra-node connections may be false, at least in terms of bandwidth. Comparing Figs. 8 and 9, one can see that the results not only depend on the hardware characteristics of a given system, but also significantly on optimizations within the MPI library. There is a growing awareness of those problems in the community and efforts towards more efficient intra-node MPI communication are visible [16].

At small message sizes, MPI communication is latency-

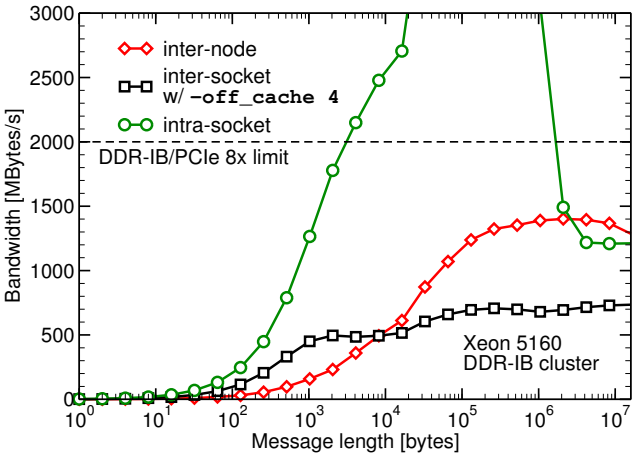

Figure 8. IMB PingPong bandwidth versus message size for inter-node, inter-socket, and intra-socket communication on a dual-socket dual-core Xeon 5160 cluster with DDR InfiniBand [4], using Intel MPI 3.0.43.

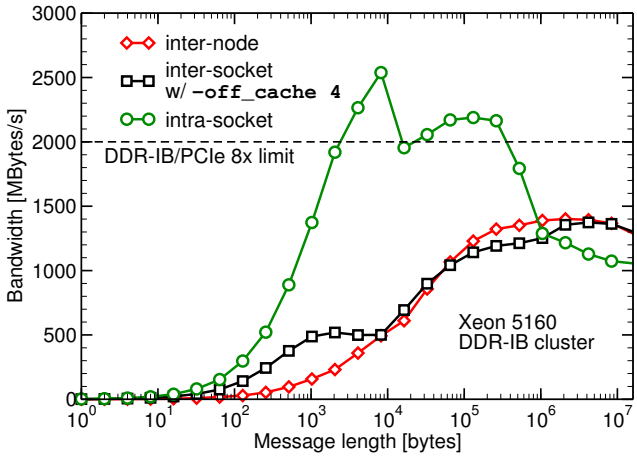

Figure 9. IMB PingPong bandwidth versus message size for inter-node, inter-socket, and intra-socket communication on a dual-socket dual-core Xeon 5160 cluster with DDR InfiniBand [4], using Intel MPI 3.1.038.

dominated. For the setup described above we measure the following latency numbers:

	Latency [μs]		
		Xeon-IB	
Mode	XT5	IMPI 3.0	IMPI 3.1
inter-node	7.40	3.13	3.24
inter-socket	0.63	0.76	0.55
intra-socket	0.49	0.40	0.31

In strong scaling scenarios it is often quite likely that one "rides the PingPong curve" towards a latency-driven regime as processor numbers increase, possibly rendering the carefully tuned process/thread placement useless.

Please note that more elaborate low-level benchmarks than PingPong may be advisable to arrive at a more complete picture about communication characteristics.

3.3. Network saturation and sleeping threads with the masteronly model

The masteronly variant, in which no MPI calls are issued inside OpenMP-parallel regions, can be used with fully hybrid as well as the mixed model. Although being the easiest way of implementing a hybrid MPI+OpenMP code, it has two important shortcomings:

1. In the fully hybrid case, a single communicating thread may not be able to saturate the node's network connection. Using a mixed model (see Sect. 3.1.3) with more than one MPI process per node might solve this problem, but one has to be aware of possible rank/thread ordering problems as described in Sect. 3.1. On flat-memory SMP nodes with no intra-node hierarchical structure, this may be an attractive and easy to use option [5]. However, the number of systems with such characteristics is waning. Current hierarchical architectures require some more effort in terms of thread-/core affinity (see Sect. 4.1 for benchmark results in mixed mode on a contemporary cluster).

2. While the master thread executes MPI code, all other threads sleep. This effectively makes communication a purely serial component in terms of Amdahl's Law. Overlapping communication with computation may provide a solution here (see Sect. 3.4 below).

One should note that on many commodity clusters today (including those featuring high-speed interconnects like InfiniBand), saturation of a network port can usually be achieved by a single thread. However, this may change if, e.g., multiple network controllers or ports are available per node. As for the second drawback above, one may argue that MPI provides non-blocking point-to-point operations which should generally be able to achieve the desired overlap. Even so, many MPI implementations allow communication progress, i.e., actual data transfer, only inside MPI calls so that real background communication is ruled out. The non-availability of non-blocking collectives in the current MPI standard adds to the problem.

3.4. Overlapping communication and computation

It seems feasible to "split off" one or more OpenMP threads in order to execute MPI calls, letting the rest do the actual computations. Just as the fully hybrid model, this requires the MPI library to support at least the `MPI_THREAD_FUNNELED`. However, work distribution across the non-communicating threads is not straightforward with this variant, because standard OpenMP worksharing works on the whole team of threads only. Nested parallelism is not an alternative due to its performance drawbacks and limited availability. Therefore, manual worksharing must be applied:

```
if (my_thread_ID < 1) {
  MPI_Send( halo data )
  MPI_Recv( halo data )
} else {
  my_range = (high-low-1) / (num_threads-1) + 1;
  my_low   = low + (my_thread_ID+1)*my_range;
  my_high  = high+ (my_thread_ID+1+1)*my_range;
  my_high  = max(high, my_high)
  for (i=my_low; i<my_high; i++) {
     /* computation */
  }
}
```

Apart from the additional programming effort for dividing the computation into halo-dependent and non-halo-dependent parts (see Sect. 2.3), directives for loop worksharing cannot be used any more, making "dynamic" or "guided" schemes that are essential to use in poorly load-balanced situations very hard to implement. Thread subteams [6] have been proposed as a possible addition to the future OpenMP 3.x/4.x standard and would ameliorate the problem significantly. OpenMP tasks, which are part of the recently passed OpenMP 3.0 standard, also form an elegant alternative but presume that dynamic scheduling (which is inherent to the task concept) is acceptable for the application.

See Ref. [5] for performance models and measurements comparing parallelization with masteronly style versus overlapping communication and computation on SMP clusters with flat intra-node structure.

3.5. OpenMP performance pitfalls

As with standard (non-hybrid) OpenMP, hybrid MPI+OpenMP is prone to some common performance pitfalls. Just by switching on OpenMP, some compilers refrain from some loop optimizations, causing a significant performance hit. A prominent example is SIMD vectorization of parallel loops on x86 architectures, which gives best performance when using 16-byte aligned load/store instructions. If the compiler cannot apply dynamic loop peeling [8], a loop parallelized with OpenMP can only be vectorized using unaligned loads and stores. The situation seems to improve gradually, though.

On ccNUMA architectures correct first-touch page placement must be employed in order to achieve scalable performance across NUMA locality domains. In this respect one should also keep in mind that communicating threads, inside or outside of parallel regions, may have to partly access non-local MPI buffers (i.e. from other NUMA domains).

Due to, e.g., limited memory bandwidth, it may be preferential in terms of performance or power consumption to use fewer threads than available cores inside of each MPI process [9]. This leads again to several affinity options (similar to Fig. 6 (a) and (b)) and may impact MPI inter-node communication.

Thread creation/wakeup overhead and frequent synchronization are further typical sources of performance problems with OpenMP, because they add to serial execution and thus contribute to Amdahl's Law on the node level. In order to estimate these overheads we propose a simple benchmark setup: We use the vector triad with short vector lengths so that the parallel run scales across threads if each core has its own L1 (performance would not scale with larger vectors as shared caches or main memory usually present bottlenecks):

```
!$OMP PARALLEL PRIVATE(j)
  do j=1,NITER
!$OMP DO SCHEDULE(static) NOWAIT ! NOWAIT optional
    do i=1,N
      A(i) = B(i) + C(i) * D(i)
    enddo
!$OMP END DO
  enddo
!$OMP END PARALLEL
```

`NITER` is chosen so that overall runtime can be accurately measured and one-time startup effects (loading data into cache the first time etc.) become unimportant. The `NOWAIT` clause is optional and is used here only to demonstrate the impact of the implied barrier at the end of the loop worksharing construct (see below).

The performance model assumes that overall runtime with a problem size of N on t threads can be split into computational and setup/shutdown contributions:

$$T(N,t) = T_{\mathrm{c}}(N,t) + T_{\mathrm{s}}(t) \quad . \tag{1}$$

Further assuming that we have measured purely serial performance $P_{\mathrm{s}}(N)$ we can write

$$T_{\mathrm{c}}(N,t) = \frac{N}{tP_{\mathrm{s}}(N/t)} \ , \tag{2}$$

which accounts for any N-dependent performance behavior unconnected to OpenMP overhead. As mentioned above, setup/shutdown time is composed of a constant latency and a per-thread component:

$$T_s(t) = T_l + T_p(t) \quad . \tag{3}$$

Now we can calculate parallel performance on t threads at problem size N:

$$P(N,t) = \frac{N}{T(N,t)} = \frac{N}{N\left[tP_s(N/t)\right]^{-1} + T_s(t)} \tag{4}$$

Fig. 10 shows performance data for the small-N vector triad on an XT5 node. We distinguish several important cases. First, there is a measurable overhead for running with a single OpenMP thread (squares) versus purely serial mode (circles). However, the purely serial saturation performance of 2300 MFlops/s is met for $N \lesssim 1000$ even with OpenMP switched on. This leads to the conclusion that, although the compiler could do a better job in reducing overhead for starting a single-thread "parallel" construct, the scalar loop-level optimizations (software pipelining, SIMD vectorization, etc.) are unharmed by OpenMP.

Second, the two-thread data (diamonds) labeled "1S" and "2S" has been obtained by binding the threads to the same sockets (solid line) and different sockets (dashed line), respectively. An additional 20–40 % overhead for cross-socket synchronization of a pair of threads can be clearly observed. This substantiates the need for accurate thread placement.

And finally we have fitted the model (4) to measured data for the eight-thread cases when using the `NOWAIT` clause (filled triangles) and without it (open triangles). Instead of the real performance values for P_s we have used a fit function (dashed line) to a simple latency/bandwidth model (latency=1.1 ns, performance=2300 MFlops/s) up to a loop length of $N = 1000$. Measurements get erratic above $N = 1000$ because of L1 associativity conflicts, but the range considered here is sufficient to get decent estimates for the overhead. The indicated fit parameters in nanoseconds denote $T_s(t)$, as defined in (3). Obviously the barrier strongly dominates OpenMP overhead. The eight-thread data labeled "inner" (stars) was obtained by using a combined `parallel do` directive, so that the team of threads is woken up each time the inner loop gets executed:

```
  do j=1,NITER
!$OMP PARALLEL DO SCHEDULE(static)
    do i=1,N
      A(i) = B(i) + C(i) * D(i)
    enddo
!$OMP END PARALLEL DO
  enddo
```

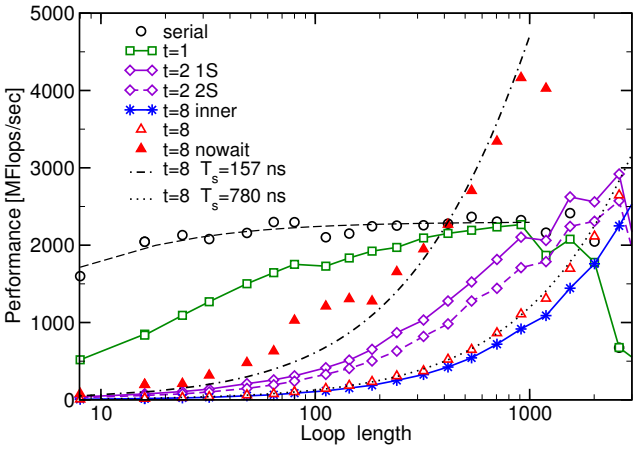

Figure 10. Impact of OpenMP loop overhead on an XT5 node for the small-N vector triad. For two cases (triangles) the model (4) was fitted to measured performance data. Note the large impact of the implied barrier (removed by the `nowait` clause). The stars indicate data obtained by restarting the parallel region with 8 threads on every triad loop.

This makes it possible to separate the thread wakeup time from the barrier and worksharing overheads: Although small compared to the barrier time, wakeup still contributes in a measurable way, which proves that it is desirable to minimize the number of parallel regions in an OpenMP program. We must stress that these overhead measurements can only be rough guidelines, and that the resulting numbers in nanoseconds denote orders of magnitude rather than exact figures. However, they show that thousands of cycles can be spent for setting up a parallel work-sharing construct. This effect will worsen with the advent of many-core chips.

4. Expected hybrid parallelization benefits

We have made it clear in the previous section that the parallel programming models described so far do not really fit onto standard hybrid hardware. Consequently, one should always try to optimize the parallel environment, especially in terms of thread/core mapping and the correct choice of hybrid execution mode, in order to minimize the mismatch problems.

On the other hand, as pointed out in the introduction, several real benefits can be expected from hybrid programming models as opposed to pure MPI. We will elaborate on the most important aspects in the following sections.

4.1. Additional levels of parallelism

In some applications, there is a coarse outer level of parallelism which can be easily exploited by message passing, but is strictly limited to a certain number of workers. In such a case, a viable way to improve scalability beyond this limit is to use OpenMP in order to speed up each MPI process, e.g. by identifying parallelizable loops at an inner level. A prominent example is the BT-MZ benchmark from the NPB (Multi-Zone NAS Parallel Benchmarks) suite.

Benchmark results on Cray XT4 and Cray XT5 System

Here we present some performance results that were obtained on Cray XT4 and Cray XT5 systems. The Cray XT4 used for the tests consists of 2,151 compute nodes. Each node comprises one AMD "Budapest" 2.1 GHz quad-core processor with 8 GB of memory, which makes for a total number of 8608 cores. The Cray XT5 has 432 compute nodes. Each node comprises two AMD "Barcelona" 2.3 GHz quad-core processors and 32 GB of memory, for a total of 3456 cores. The nodes are connected to a 3D torus network by HyperTransport links, using the Cray Seastar2 Interconnect on both systems. The MPI implementations are based on MPICH-2. Results were obtained by courtesy of the HPCMO Program, the Engineer Research and Development Center Major Shared Resource Center, Vicksburg, MS, and the Arctic Region Supercomputer Center in Fairbanks, Alaska. For compiling the benchmarks we used the Cray ftn compiler which is based on the PGI Fortran compiler pgf90 7.1. We used the options `ftn -fastsse -tp barcelona-64 -r8 -mp`. The `aprun` command was used for execution. On the Cray XT4 the executable was started as

```
export OMP_NUM_THREADS={4,2,1}
aprun -n NPROCS -N {1,2,4} -d {4,2,1}
```

This places 1, 2, or 4 MPI processes per node and employs 4, 2, or 1 core per MPI process, corresponding to the number of threads per process which is being used. On the Cray XT5 the executable was started as

```
export OMP_NUM_THREADS={8,4,2,1}
aprun -n NPROCS -N 1 -d 8
aprun -n NPROCS -S {1,2,4} -d {4,2,1}
```

The first command places 1 MPI process per node and employs all 8 cores for OpenMP threads. The second command places 1, 2, or 4 MPI processes per socket and employs 4, 2, or 1 core per MPI process. One processor socket (4 cores) corresponds to one NUMA node.

The NAS Parallel Benchmark (NPB) Multi-Zone (MZ) [10] codes BT-MZ and SP-MZ (class E) were chosen

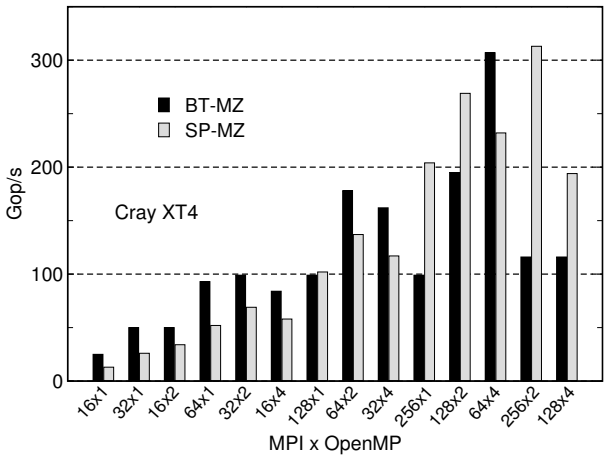

Figure 11. NPB BT-MZ and SP-MZ (class C) performance on Cray XT4 for mixed hybrid and pure MPI modes (see text for details on the mixed setup). There is no pure MPI data for 512 cores as the number of MPI processes is limited to 256 (zones) in that case.

to exemplify the benefits and limitations of hybrid mode. The purpose of the NPB-MZ is to capture the multiple levels of parallelism inherent in many full scale applications. Each benchmark exposes a different challenge to scalability: BT-MZ is a block tridiagonal simulated CFD code. The size of the zones varies widely, with a ratio of about 20 between the largest and the smallest zone. This poses a load balancing problem when only coarse-grained parallelism is exploited on a large number of cores. SP-MZ is a scalar pentadiagonal simulated CFD code with equally sized zones, so from a workload point of view the best performance should be achieved by pure MPI. A detailed discussion of the performance characteristics of these codes is presented in [11]. The class C problem size for both benchmarks comprises an aggregate grid size of $480\times320\times28$ points and a total number of 256 zones. Each MPI process is assigned a set of zones to work on, according to a bin-packing algorithm to achieve a balanced workload. Static worksharing is used on the OpenmMP level. Due to the implementation of the benchmarks the maximum number of MPI processes is limited to the number of zones for SP-MZ as well as BT-MZ to 256 for the class C benchmark.

Figures 11 and 12 show results at 16 to 512 cores. For both BT-MZ and SP-MZ the mixed hybrid mode enables scalability beyond the number of zones. In the case of BT-MZ, reducing the number of MPI processes and using OpenMP threads allows for better load balancing while maintaining a high level of parallelism. On the Cray XT4 we observe that BT-MZ does not scale to 512 cores. The

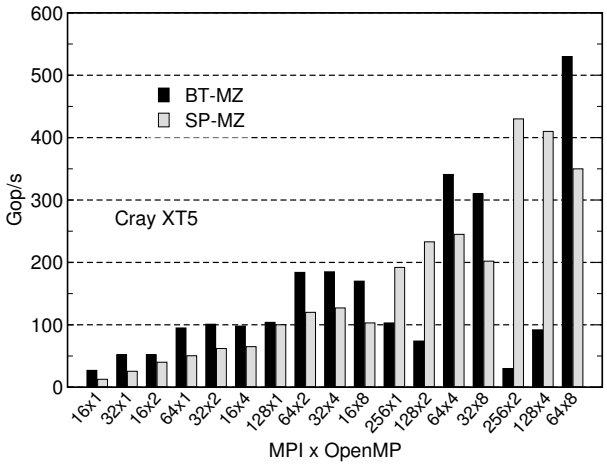

Figure 12. NPB BT-MZ and SP-MZ (class C) performance on Cray XT5 for mixed hybrid and pure MPI modes (see text for details on the mixed setup). There is no pure MPI data for 512 cores as the number of MPI processes is limited to 256 (zones) in that case.

reason is that optimal load balancing is achieved by employing 64 MPI processes using 8 threads each. This combination is not possible on the Cray XT4. On the Cray XT5 BT-MZ does scale for 512 cores. SP-MZ generally scales well with pure MPI. However, reducing the number of MPI processes cuts down on the amount of data to be communicated and the total number of MPI calls. For some cases this leads to the hybrid version outperforming pure MPI. On the Cray XT5, for example, SP-MZ 64×4 outperforms 256×1. The communication overhead depends on the exact communication pattern and placement of the MPI processes. Thus, for both benchmarks, hybrid MPI+OpenMP can outperform pure MPI. The best mixed hybrid mode for BT-MZ depends on the coarse-grained load balancing that can be achieved and varies with the number of available cores.

We must emphasize that the use of affinity mechanisms (built into `aprun` in this particular case) is absolutely essential for getting good performance and reproducibility on any ccNUMA architecture, including the XT5.

4.2. Improved load balancing

If the problem at hand has load balancing issues, some kind of dynamic balancing should be implemented. In MPI, this is a problem for which no generic recipes exist. It is highly dependent on the numerics and potentially requires significant communication overhead. It is therefore hard to implement in production codes.

One big advantage of OpenMP over MPI lies in the possible use of "dynamic" or "guided" loop scheduling. No additional programming effort or data movement is required. However, one should be aware that non-static scheduling is suboptimal for memory-bound code on ccNUMA systems because of unpredictable (and non-reproducible) access patterns; if guided or dynamic schedule is unavoidable, one should at least employ round-robin page placement for array data in order to get *some* level of parallel data access.

For the hybrid case, simple static load balancing on the outer (MPI) level and dynamic/guided loop scheduling for OpenMP can be used as a compromise. Note that if dynamic OpenMP load balancing is prohibitive because of NUMA locality constraints, a mixed model (Fig. 1 (d)) may be advisable where one MPI process runs in each NUMA locality domain and dynamic scheduling is applied to the threads therein.

4.3. Reduced memory consumption

Although one might believe that there should be no data duplication or, more generally, data overhead between MPI processes, this is not true in reality. E.g., in domain decomposition scenarios, the more MPI domains a problem is divided into, the larger the aggregated surface and thus the larger the amount of memory required for halos. Other data like buffers internal to the MPI subsystem, but also lookup tables, global constants, and everything that is usually duplicated for efficiency reasons, adds to memory consumption. This pertains to redundant computation as well.

One the other hand, if there are multiple (t) threads per MPI process, duplicated data is reduced by a factor of t (this is also true for halo layers if not using domain decomposition on the OpenMP level). Although this may seem like a small advantage today, one must keep in mind that the number of cores per CPU chip is constantly increasing. In the future, tens and even hundreds of cores per chip may lead to a dramatic reduction of available memory per core.

It should be clear from the considerations in the previous sections that it is not straightforward to pick the optimal number of OpenMP threads per MPI process for a given problem and system. Even assuming that mismatch/affinity problems can be kept under control, using too many threads can have negative effects on network saturation, whereas too man MPI processes might lead to intolerable memory consumption.

4.4. Further opportunities

Using multiple threads per process may have some benefits on the algorithmic side due to larger physical domains inside of each MPI process. This can happen whenever a larger domain is advisable in order to get improved numerical accuracy or convergence properties. Examples are:

- A multigrid algorithm is employed only per MPI domain, i.e. inside each process, but not between domains.
- Separate preconditioners are used inside and between MPI processes.
- MPI domain decomposition is based on physical zones.

An often used argument in favor of hybrid programming is the potential reduction in MPI communication in comparison to pure MPI. As shown in Sect. 3.1 and 5, this point deserves some scrutiny because one must compare optimal domain decompositions for both alternatives. However, the number of messages sent and received per node does decrease which helps to reduce the adverse effects of MPI latency. The overall aggregate message size is diminished as well if intra-process "messages", i.e. NUMA traffic, are not counted. In the fully hybrid case, no intra-node MPI is required at all, which may allow the use of a simpler (and hopefully more efficient) variant of the message-passing library, e.g., by not loading the shmem device driver. And finally, a hybrid model enables incorporation of functional parallelism in a very straightforward way: Just like using one thread per process for concurrent communication/computation as described above, one can equally well split off another thread for, e.g., I/O or other chores that would be hard to incorporate into the parallel workflow with pure MPI. This could even reduce the non-parallelizable part of the computation and thus enhance overall scalability.

5. Aspects of future standardization efforts

In Sect. 3 we have argued that mismatch problems need special care, not only with hybrid programming, but also under pure MPI. However, correct rank ordering and the decisions between pure and mixed models cannot be optimized without knowledge about machine characteristics. This includes, among other things, inter-node, inter-socket and intra-socket communication bandwidths and latencies, and information on the hardware topology in and between nodes (cores per chip, chips per socket, shared caches, NUMA domains and networks, and message-passing network topology). Today, the programmer is often forced to use non-portable interfaces in order to acquire this data (examples under Linux are libnuma/numactl and the Intel "cpuinfo" tool; other tools exist for other architectures and operating systems) or perform their own low-level benchmarks to figure out topology features.

What is needed for the future is a standardized interface with an abstraction layer that shifts the non-portable programming effort to a library provider. In our opinion, the right place to provide such an interface is the MPI library, which has to be adapted to the specific hardware anyway. At least the most basic topology and (quantitative) communication performance characteristics could be done inside MPI at little cost. Thus we propose the inclusion of a topology/performance interface into the future MPI 3.0 standard, see also [12].

As mentioned in Sect. 3.3, there are already some efforts to include a subteam feature into upcoming OpenMP standards. We believe this feature to be essential for hybrid programming on current and future architectures, because it will greatly facilitate functional parallelism and enable standard dynamic load balancing inside multi-threaded MPI processes.

6. Conclusions

In this paper we have pinpointed the issues and potentials in developing high performance parallel codes on current and future hierarchical systems. Mismatch problems, i.e. the unsuitability of current hybrid hardware for running highly parallel workloads, are often hard to solve, let alone in a portable way. However, the potential gains in scalability and absolute performance may be worth the significant coding effort. New features in future MPI and OpenMP standards may constitute a substantial improvement in that respect.

Acknowledgements

We express our thanks to the HPCMO Program and the Engineer Research and Development Center Major Shared Resource Center, Vicksburg, MS (`http://www.erdc.hpc.mil/index`), who provided compute time and support to obtain our benchmark results. We greatly appreciate the excellent user support provided by the Arctic Region Supercomputing Center. Fruitful discussions with Rainer Keller and Gerhard Wellein are gratefully acknowledged.

References

[1] R. Loft, S. Thomas, J. Dennis: *Terascale Spectral Element Dynamical Core for Atmospheric General Circulation Models.* Proceedings of SC2001, Denver, USA.

[2] *Cluster OpenMP for Intel compilers.* `http://software.intel.com/en-us/articles/cluster-openmp-for-intel-compilers`

[3] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, W. Zwaenepoel: *TreadMarks: Shared Memory Computing on Networks of Workstations.* IEEE Computer 29(2), 18–28 (1996).

[4] http://www.hpc.rrze.uni-erlangen.de/systeme/woodcrest-cluster.shtml

[5] R. Rabenseifner, G. Wellein: *Communication and Optimization Aspects of Parallel Programming Models on Hybrid Architectures.* International Journal of High Performance Computing Applications 17(1), 49–62 (2003).

[6] B. M. Chapman, L. Huang, H. Jin, G. Jost, B. R. de Supinski: *Toward Enhancing OpenMP's Work-Sharing Directives.* In W. E. Nagel et al. (Eds.): Proceedings of Euro-Par 2006, LNCS 4128, 645–654. Springer (2006).

[7] G. Hager, H. Stengel, T. Zeiser, G. Wellein: *RZBENCH: Performance evaluation of current HPC architectures using low-level and application benchmarks.* In: S. Wagner et al. (Eds.), High Performance Computing in Science and Engineering, Garching/Munich 2007, 485–501, Springer (2009). http://arxiv.org/abs/0712.3389

[8] M. Stürmer, G. Wellein, G. Hager, H. Köstler, U. Rüde: *Challenges and potentials of emerging multicore architectures.* In: S. Wagner et al. (Eds.), High Performance Computing in Science and Engineering, Garching/Munich 2007, 551–566, Springer (2009).

[9] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, M. Schulz: *Prediction Models for Multi-dimensional Power-Performance Optimization on Many Cores.* In D. Tarditi, K. Olukotun (Eds.), Proceedings on the Seventeenth International Conference on Parallel Architectures and Compilation Techniques (PACT08), Toronto, Canada, Oct. 25–29, 2008.

[10] R. F. Van Der Wijngaart, H. Jin: *NAS Parallel Benchmarks, Multi-Zone Versions.* NAS Technical Report NAS-03-010, NASA Ames Research Center, Moffett Field, CA, 2003.

[11] H. Jin, R. F. Van Der Wijngaart: *Performance Characteristics of the multi-zone NAS Parallel Benchmarks.* Journal of Parallel and Distributed Computing, Vol. 66, Special Issue: 18th International Parallel and Distributed Processing Symposium, pp. 674–685, May 2006.

[12] MPI Forum: *MPI-2.0 Journal of Development (JOD)*, Sect. 5.3 "Cluster Attributes", http://www.mpi-forum.org, July 18, 1997.

[13] R. Rabenseifner, G. Hager, G. Jost, R. Keller: *Hybrid MPI and OpenMP Parallel Programming.* Half-day Tutorial No. M-09 at SC08, Austin, TX, Nov. 15–21, 2008.

[14] R. Rabenseifner: *Some Aspects of Message-Passing on Future Hybrid Systems.* Invited talk at 15th European PVM/MPI Users' Group Meeting, EuroPVM/MPI 2008, Sep. 7–10, 2008, Dublin, Ireland. LNCS 5205, pp 8–10, Springer (2008).

[15] R. Rabenseifner, G. Hager, G. Jost: *Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes.* In Didier El Baz et al. (Eds.), Proceedings of the 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2009), Weimar, Germany, Feb. 16–18, 2009, pp. 427–236, Computer Society Press (2009).

[16] B. Goglin: *High Throughput Intra-Node MPI Communication with Open-MX.* In Didier El Baz et al. (Eds.), Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2009), Weimar, Germany, Feb. 16–18, 2009, pp. 173–180, Computer Society Press (2009).

[17] http://www.arsc.edu/support/howtos/usingxt5.html