Prospects for Truly Asynchronous Communication with Pure MPI and Hybrid MPI/OpenMP on Current Supercomputing Platforms

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

Gerald Schubert
Institute of Physics
University of Greifswald, Germany
schubert@physik.uni-greifswald.de

Thomas Schoenemeyer
Swiss National Supercomputing Centre (CSCS)
Manno, Switzerland
schoenemeyer@cscs.ch

Gerhard Wellein
Erlangen Regional Computing Center (RRZE), Germany
gerhard.wellein@rrze.uni-erlangen.de

Abstract

We investigate the ability of MPI implementations to perform truly asynchronous communication with nonblocking point-to-point calls on current highly parallel systems, including the Cray XT and XE series. For cases where no automatic overlap of communication with computation is available, we demonstrate several different ways of establishing explicitly asynchronous communication by variants of functional decomposition using OpenMP threads or tasks, implement these methods in a parallel sparse matrix-vector multiplication code, and show the resulting performance benefits. The impact of node topology and the possible use of simultaneous multithreading (SMT) is studied in detail.

1 Introduction

1.1 Asynchronous communication and scope of work

Modern supercomputing systems are almost exclusively built from multicore, multisocket compute "nodes" connected via a high-performance network. Fast networks are mandatory, since highly demanding numerical simulation codes tend to have strong communication requirements. This is partially due to the popular "bulk-synchronous" execution model where (pure) computation phases are interleaved with (pure) communication periods. Often, even the fastest network is unable to deliver the required bandwidth and latency constraints, especially in strong scaling scenarios [1].

In order to alleviate the performance and scalability limitations caused by interprocess communication, application programmers typically try to reduce the amount of data transmitted or, if that is not possible, to hide communication costs by overlapping useful computation with communication. The MPI standard provides some features that might be useful in this respect, first and foremost the nonblocking point-to-point calls `MPI_Irecv()` and `MPI_Isend()`. However, it is known that many current MPI implementations do not support truly asynchronous transfers but perform MPI progress within the `MPI_Wait()` or `MPI_Test()` functions, respectively. This behavior is entirely backed by the MPI standard [2], which does not require nonblocking point-to-point communication to be asynchronous at all. If collective communication is the problem, the current MPI standard does not even provide nonblocking calls (although this is planned for upcoming releases).

Hence, explicit solutions for overlapping computation and communication are required. In [3] the principles of applying OpenMP threads for asynchronous communication were described in detail; in [4] it was demonstrated that OpenMP tasking could provide a convenient

way to perform MPI collectives asynchronously in a hybrid MPI/OpenMP-parallel code. We have discussed MPI-only and hybrid MPI/OpenMP-parallel versions of a sparse matrix-vector multiplication (spMVM) code in [5], with a focus on Intel-based commodity cluster hardware. Hybrid "task mode," i.e., dedicating a separate application thread for communication, has already been used before the advent of multicore processors for optimizing parallel sparse matrix-vector multiplications on vector-parallel and clustered SMP architectures [6, 7].

In this paper we will first present a short survey of the capability of current MPI implementations on Crays and standard cluster systems to perform asynchronous nonblocking point-to-point communication for large messages. After that we turn to the specific application scenario of parallel sparse CRS matrix-vector multiplication (spMVM) with MPI and hybrid MPI/OpenMP. spMVM is the dominant component in many eigenvalue or sparse linear system solvers, so a highly efficient scalable spMVM implementation is fundamental, and complements advancements and new developments in the high-level algorithms. We deliberately ignore formulations where special features of the matrix like symmetries, bands, blocks etc. are exploited. An overview of node-level optimization strategies for spMVM was given in [8], and other recent work [9, 10] deals with distributed-memory parallel implementations, mostly based on MPI-only strategies.

Our hybrid implementations are tested against pure MPI approaches for two application scenarios (i.e., matrices) on a Cray XE6 system as well as an InfiniBand cluster.

1.2 Test machines for the application benchmarks

Intel Westmere EP Cluster The Intel Westmere EP processor (Xeon X5650, 2.66 GHz base frequency, "turbo mode" and simultaneous multithreading [SMT] enabled) accommodates six cores per socket. A socket forms its own ccNUMA locality domain (LD) via three DDR3-1333 memory channels, allowing for a peak bandwidth of 32 GB/s. The dual-socket nodes in RRZE's "LiMa" cluster are connected via a fully nonblocking fat-tree QDR InfiniBand (IB) network. The Intel compiler in version 11.1 and the Intel MPI library in version 4.0.1 were used. Thread-core affinity was controlled with the LIKWID [11] toolkit.

Cray XE6 / AMD Magny Cours The Cray XE6 system ("Palu" at CSCS) is based on dual-socket nodes with AMD Magny Cours 12-core processors (2.1 GHz Opteron 6172) and the latest Cray "Gemini" interconnect. The internode bandwidth of the 2D torus network is beyond the capability of QDR InfiniBand. A 12-core processor package comprises two 6-core chips with separate L3 caches and memory controllers, tightly bound by "1.5" HyperTransport (HT) 16x links. Each 6-core unit forms its own ccNUMA LD via two DDR3-1333 channels, i.e., a two-socket node comprises four ccNUMA locality domains. In total the AMD design uses eight memory channels, allowing for a theoretical main memory bandwidth advantage of $8/6$ over a Westmere node. The Cray compiler in version 7.2.8 was used for the Cray/AMD measurements.

Listing 1: A simple benchmark to determine the capability of the MPI library to perform asynchronous nonblocking point-to-point communication for large messages (receive variant).

```
if(rank==0) {
  stime = MPI_Wtime();
  MPI_Irecv(rbuf,mcount,MPI_DOUBLE,1,0,
     MPI_COMM_WORLD,&req);
  do_work(calctime);
  MPI_Wait(req, &status);
  etime = MPI_Wtime();
  cout << calctime << "␣" << etime-stime << endl;
} else {
  MPI_Send(sbuf,mcount,MPI_DOUBLE,0,0,
     MPI_COMM_WORLD);
}
```

Cray XT4 / AMD Barcelona For low-level benchmark comparisons we used the older Cray XT4 system "Franklin" at NERSC, which comprises single-socket quad-core Opteron nodes (2.3 GHz).

1.3 Nonblocking point-to-point communication in MPI

Very simple benchmark tests can be used to find out whether the nonblocking point-to-point communication calls in an MPI library do actually support truly asynchronous transfers. Listing 1 shows an example where an `MPI_Irecv()` operation is set off before a function (`do_work()`) performs register-only operations for a configurable amount of time. If the nonblocking message transfer overlaps with computations, the overall runtime of the code will be constant as long as the time for computation is smaller than the time for message transfer. We have used a constant message length of 80 MB in our measurements. Note that, especially for small messages, the results of such tests may depend crucially on tunable parameters like, e.g., the message size for the cross-over from an "eager" to a "rendezvous" protocol. For the application scenarios described later, the assumption of large messages is justified. Figures 1 and 2 show overall runtime versus time for computation on the Intel Westmere cluster and the Cray

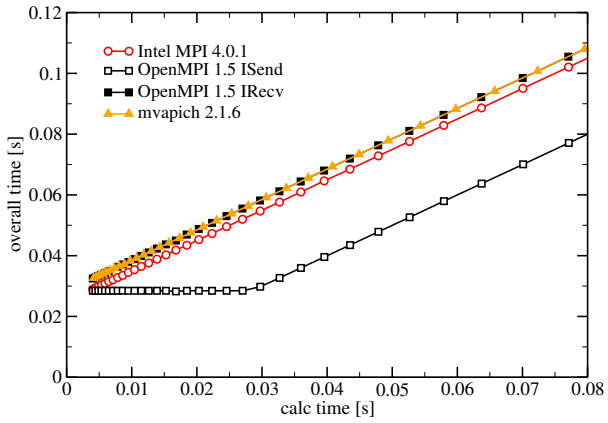


Figure 1: Results for the asynchronous nonblocking MPI benchmark for several MPI implementations on a Westmere-based cluster with QDR InfiniBand interconnect (internode communication). Overlap does generally not work for intranode communication. Unless indicated otherwise, results for nonblocking send and receive are almost identical.

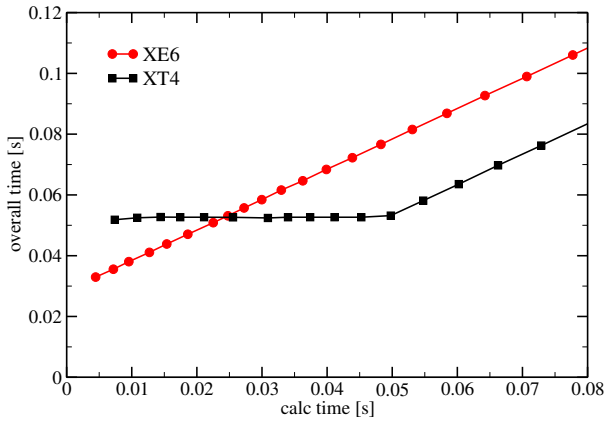


Figure 2: Results for the asynchronous nonblocking MPI benchmark for Cray XT4 and XE6 systems (internode communication). Overlap does generally not work for intranode communication

XT4/XE6 systems, respectively. We only report internode results, since no current MPI implementation on any system supports asynchronous nonblocking intranode communication.

On the Intel cluster we compared three different MPI implementations: Intel MPI, OpenMPI, and MVAPICH2. The latter was compiled with the `--enable-async-progress` flag. OpenMPI 1.5 supports a similar setting, but it is documented to be still under development in the current version (1.5.3), and we were not able to produce a stable configuration with progress threads activated. The results show that only OpenMPI (even without progress threads explicitly enabled) was capable of asynchronous nonblocking communication, albeit only when sending data.

Comparing the Cray XT4 and XE6 systems, it is striking that only the older XT4 has an MPI implementation that supports asynchronous nonblocking transfers for large messages.

In summary, one must conclude that the naive assumption that "nonblocking" and "asynchronous" are the same thing cannot be upheld for most current MPI implementations; as a consequence, overlapping computation with communication is often a matter of explicit programming. We will explore these opportunities using the example of sparse matrix-vector multiplication (spMVM).

1.4 Sparse matrix-vector multiplication and node-level performance model

A possible definition of a "sparse" matrix is that the number of its nonzero entries grows only linearly with the matrix dimension; however, not all problems are easily scaled, so in general a sparse matrix may be defined as containing "mainly" zero entries. Since keeping such a matrix in computer memory with all zeros included is usually out of the question, an efficient format to store the nonzeros only is required. The most widely used variant is "Compressed Row Storage" (CRS) [12]. It does not exploit specific features that may emerge from the underlying physical problem like, e.g., block structures, symmetries, etc., but is broadly recognized as the most efficient format for general sparse matrices on cache-based microprocessors. All nonzeros are stored in one contiguous array `val`, row by row, and the starting offsets of all rows are contained in a separate array `row_ptr`. Array `col_idx` contains the original column index of each matrix entry. A matrix-vector multiplication with a right-hand-side (RHS) vector `B(:)` and a result vector `C(:)` can then be written as follows:

```
do i = 1,Nr
  do j = row_ptr(i), row_ptr(i+1) - 1
    C(i) = C(i) + val(j) * B(col_idx(j))
  enddo
enddo
```

Here N_r is the number of matrix rows. While arrays `C(:)` and `val(:)` are traversed contiguously, access to `B(:)` is indexed and may potentially cause very low spatial and temporal locality in this data stream.

The performance of spMVM operations on a single compute node is often limited by main memory bandwidth. Code balance [1] is thus a good metric to establish a simple performance model. We assume the average length of the inner (j) loop to be $N_{nzr} = N_{nz}/N_r$, where N_{nz} is the total

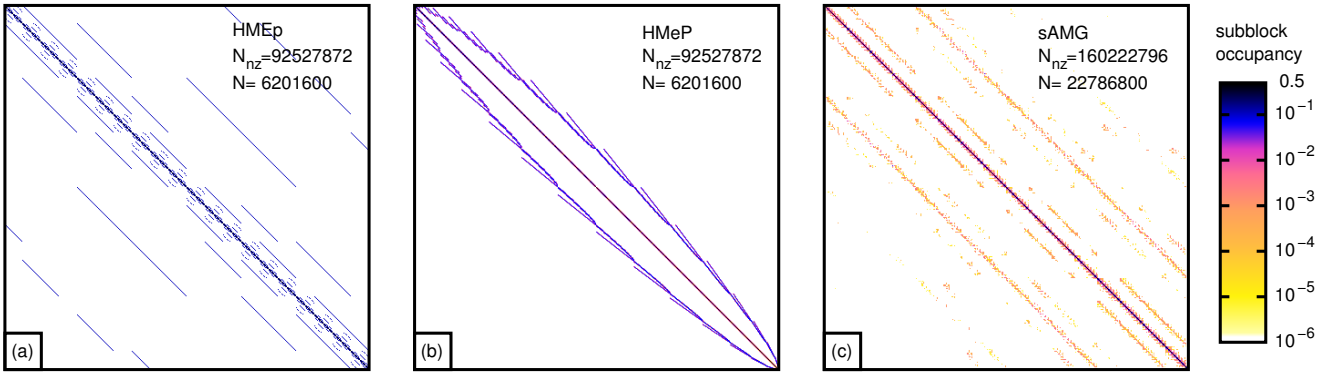


Figure 3: Sparsity patterns of the Hamiltonian matrix described in the text with different numbering of the basis elements ((a) and (b)), and the sAMG matrix (c). Square subblocks have been aggregated and color-coded according to occupancy to improve visibility.

number of nonzero matrix entries. Then the contiguous data accesses in the CRS code generate $(8+4+16/N_{\mathrm{nzr}})$ bytes of memory traffic for a single inner loop iteration, where the first two contributions come from the matrix `val(:)` (8 bytes) and the index array `col_idx(:)` (4 bytes), while the last term reflects the update of `C(i)` (write allocate + evict). The indirect access pattern to `B(:)` is determined by the sparsity structure of the matrix and can not be modeled in general. However, `B(:)` needs to be loaded at least once from main memory, which adds another $8/N_{\mathrm{nzr}}$ bytes per inner iteration. Limited cache size and nondiagonal access typically require loading at least parts of `B(:)` multiple times in a single MVM. This is quantified by a machine- and problem-specific parameter κ: For each additional time that `B(:)` is loaded from main memory, $\kappa = 8/N_{\mathrm{nzr}}$ additional bytes are needed. Together with the computational intensity of 2 flops per j iteration the code balance is

$$B_{\mathrm{CRS}} = \left(\frac{12+24/N_{\mathrm{nzr}}+\kappa}{2}\right)\frac{\text{bytes}}{\text{flop}} = \left(6+\frac{12}{N_{\mathrm{nzr}}}+\frac{\kappa}{2}\right)\frac{\text{bytes}}{\text{flop}}\,. \quad (1)$$

On the node level B_{CRS} can be used to determine an upper performance limit by measuring the node memory bandwidth (e.g., using the STREAM benchmark) and assuming $\kappa = 0$. Moreover, κ can be determined experimentally from the sparse MVM floating point performance and the memory bandwidth drawn by the CRS code (see Sect. 2). Since the matrices used here have $N_{\mathrm{nzr}} \approx 7\ldots15$, each additional access to `B(:)` incurs a nonnegligible contribution to the data transfer. Maximum floating-point performance for large problems is achieved for large N_{nzr}, and/or sparsity patterns that cause good spatial locality when accessing the RHS vector.

Note that this simple model neglects performance-limiting aspects beyond bandwidth bottlenecks, like load imbalance, communication and/or synchronization overhead, and the adverse effects of nonlocal memory access across ccNUMA locality domains (LDs).

1.5 Test matrices

Since the sparsity pattern may have substantial impact on the single node performance and parallel scalability, we have chosen two application areas known to generate extremely sparse matrices.

As a first test case we use a matrix from exact diagonalization of strongly correlated electron-phonon systems in solid state physics. Here generic microscopic models are used to treat both charge (electrons) and lattice (phonons) degrees of freedom in second quantization. Choosing a finite-dimensional basis set, which is the direct product of basis sets for both subsystems (electrons $\otimes$ phonons), the generic model can be represented by a sparse Hamiltonian matrix. Iterative algorithms such as Lanczos or Jacobi-Davidson are used to compute low-lying eigenstates of the Hamilton matrices, and more recent methods based on polynomial expansion allow for computation of spectral properties [13] or time evolution of quantum states [14]. In all those algorithms, sparse MVM is the most time-consuming step.

Here we consider the Holstein-Hubbard model (cf. [15] and references therein) with six electrons (subspace dimension 400) on a six-site lattice coupled to 15 phonons (subspace dimension 1.55×10^4). The resulting matrix of dimension 6.2×10^6 is very sparse ($N_{\mathrm{nzr}} \approx 15$) and can have two different sparsity patterns, depending on whether the phononic or the electronic basis elements are numbered contiguously (see Figs. 3 (a) and (b), respectively). We also applied the well-known "Reverse Cuthill-McKee (RCM)" algorithm [16] to the Hamilton matrix in order to improve spatial locality in the access to the right hand side vector,

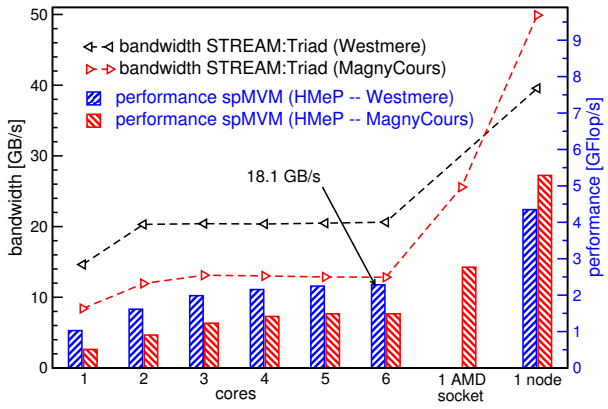


Figure 4: Node-level performance for the Westmere and Cray XE6 test systems. Effective STREAM triads bandwidth[1], and performance for sparse MVM using the HMeP matrix (bars) is shown.

and to optimize interprocess communication patterns towards near-neighbor exchange. Since the RCM-optimized structure showed no performance advantage over the HMeP variant (Fig. 3 (b)) neither on the node nor on the highly parallel level, we will not consider RCM any further here.

The second matrix was generated by the adaptive multigrid code sAMG (see [17, 18], and references therein) for the irregular discretization of a Poisson problem on a car geometry. Its matrix dimension is 2.2×10^7 with an average of $N_{\mathrm{nzr}} \approx 7$ entries per row (see Fig. 3 (c)).

For real-valued, symmetric matrices as considered here it is sufficient to store the upper triangular matrix elements and perform, e.g., a parallel symmetric CRS sparse MVM [10]. The data transfer volume is then reduced by almost a factor of two, allowing for a corresponding performance improvement. We do not use this optimization here for two major reasons. First, the discussion of the hybrid parallel vs. MPI-only implementation should not be restricted to the special case of explicitly symmetric matrices. Second, to our knowledge an efficient shared memory implementation of a symmetric CRS sparse MVM base routine has not yet been presented.

2 Node-level performance analysis

The basis for each parallel program must be an efficient single core/node implementation. Assuming general sparse matrix structures the CRS format presented above is very suitable for modern cache-based multicore processors [19]. Even advanced machine-specific optimizations such as nontemporal prefetch instructions for Opteron processors provide only minor benefits [10] and are thus not considered here. A simple OpenMP parallelization of the outermost loop, together with an appropriate ccNUMA-aware data placement strategy has proven to provide best node-level performance. We choose the HMeP matrix as a reference problem. The results presented hold qualitatively for the other matrix structures as well. Differences will be discussed where relevant.

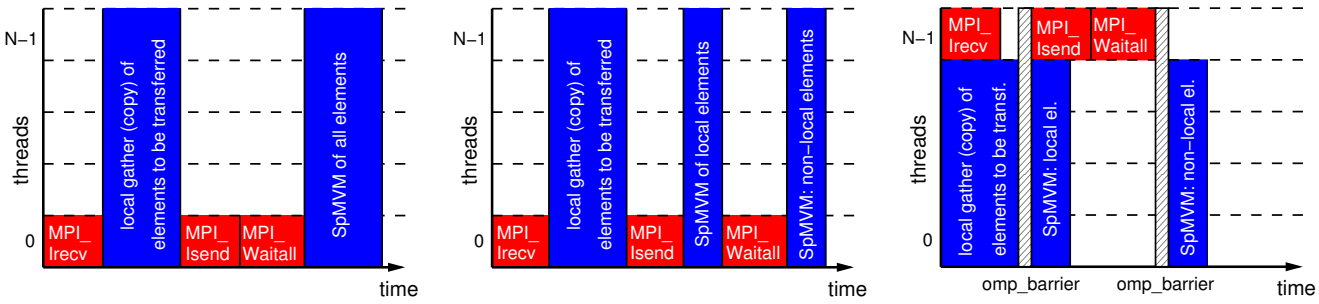
Intrasocket and intranode spMVM scalability should always be discussed together with effective STREAM triad numbers, which form a practical upper bandwidth limit.[1] Figure 4 shows the memory bandwidth on the Cray XE6 and Westmere EP platforms (dashed lines) drawn by the STREAM triad, and the corresponding spMVM performance. While the STREAM bandwidth soon saturates within a socket, the spMVM code needs about four cores to reach saturation. This is typical behavior for codes with (partially) irregular data access patterns. However, the fact that more than 85% of the STREAM bandwidth can be reached with spMVM (as measured by the LIKWID [11] tool) indicates that our CRS implementation makes good use of the resources. The maximum spMVM performance can be estimated by dividing the memory bandwidth by the code balance (1), using $N_{\mathrm{nzr}} = 15$ and $\kappa = 0$. For a single Westmere socket the spMVM draws 18.1 GB/s (STREAM triads: 21.2 GB/s), allowing for a maximum performance of 2.66 GFlop/s (3.12 GFlop/s). Combining the measured performance (2.25 GFlop/s) and bandwidth of the spMVM operation with $B_{\mathrm{CRS}}(\kappa)$ we find $\kappa = 2.5$, i.e., 2.5 additional bytes of memory traffic on `B(:)` per inner loop iteration (37.3 bytes per row) are required due to limited cache capacity. Thus the complete vector `B(:)` is loaded six times from main memory to cache, but each element is used $N_{\mathrm{nzr}} = 15$ times. This ratio gets worse if the matrix bandwidth increases. For the HMEp matrix we found $\kappa = 3.79$, which translates to a 50% increase in the additional data transfers for `B(:)`. The code balance implies a performance drop of about 10%, which is consistent with our measurements.

While the AMD system is slower on a single LD, its node-level performance is about 25% higher than on Westmere due to its four LDs per node. Within the domains spMVM saturates at four cores on both architectures, leaving ample room to use the remaining cores for other tasks, like communication (see Sect. 3.3).

3 Distributed-memory parallelization

Strong scaling of MPI-parallel sparse MVM is inevitably limited by communication overhead. Hence, it is vital to find ways to hide communication costs as far as possible. A widely used approach is to employ nonblocking point-to-point MPI calls for overlapping communication with useful work. However, as has been shown in Sect. 1.3, most MPI

[1] Nontemporal stores have been suppressed in the STREAM measurements and the bandwidth numbers reported have been scaled appropriately ($\times 4/3$) to account for the write-allocate transfer.



(a) Vector mode without overlap (b) Vector mode with naive overlap (c) Task mode with explicit overlap

Figure 5: Schematic timeline view of the implemented hybrid kernel versions. From left to right: no communication/calculation overlap, naive overlap using nonblocking MPI, and explicit overlap by a dedicated communication thread

implementations support progress, i.e., actual data transfer, only when MPI library code is executed by the user process, although the hardware even on standard InfiniBand-based clusters does not hinder truly asynchronous point-to-point communication. In the following sections we will contrast the "naive" overlap applying nonblocking MPI with an approach that uses a dedicated OpenMP thread for explicitly asynchronous transfers. We adopt the nomenclature from [7] and [1] and distinguish between "vector mode" and "task mode."

MPI parallelization of spMVM is generally done by distributing the nonzeros (or, alternatively, the matrix rows), the right hand side vector `B(:)`, and the result vector `C(:)` evenly across MPI processes. Due to off-diagonal nonzeros, every process requires some parts of the RHS vector from other processes to complete its own chunk of the result, and must send parts of its own RHS chunk to others. Note that it is generally difficult to establish good load balancing for computation and communication at the same time. We use a balanced distribution of nonzeros across the MPI processes here. At a given number of processes, the resulting communication pattern depends only on the sparsity structure, so the necessary bookkeeping needs to be done only once. The actual spMVM computations can be performed either by a single thread or, if multithreading is available, by multiple threads inside the MPI process.

3.1 Vector-like parallelization: Vector mode without overlap

Gathering the data to be sent by a process into a contiguous send buffer may be done after the receive has been initiated, potentially hiding the cost of copying (see Fig. 5 (a)). We call this naive approach "hybrid vector mode," since it strongly resembles the programming model for vector-parallel computers [7]: The time-consuming (although probably parallel) computation step does not overlap with communication overhead. This is actually how "MPI+OpenMP hybrid programming" is still defined in most publications. The question whether and why using multiple threads per MPI process may improve performance compared to a pure MPI version on the same hardware is not easy to answer. Depending on the problem, different aspects come into play, and there is no general rule.

3.2 Vector-like parallelization: Vector mode with naive overlap

As an alternative one may consider hybrid vector mode with nonblocking MPI (see Fig. 5 (b)) to potentially overlap communication with the part of spMVM that can be completed using local RHS elements only. After the nonlocal elements have been received, the remaining spMVM operations can be performed. A disadvantage of splitting the spMVM in two parts is that the local result vector must be written twice, incurring additional memory traffic. The performance model (1) can be modified to account for an additional data transfer of $16/N_{\mathrm{nzr}}$ bytes per inner loop iteration, leading to a modified code balance of

$$B_{\mathrm{CRS}}^{\mathrm{split}} = \left(6 + \frac{20}{N_{\mathrm{nzr}}} + \frac{\kappa}{2}\right) \frac{\mathrm{bytes}}{\mathrm{flop}} \,. \qquad (2)$$

For $N_{\mathrm{nzr}} \approx 7 \ldots 15$ and assuming $\kappa = 0$, one may expect a node-level performance penalty between 15% and 8%, and even less if $\kappa > 0$.

For simplicity we will also use the term "vector mode" for pure MPI versions with single-threaded computation.

3.3 Task mode with explicit overlap

A safe way to ensure overlap of communication with computation is to use a separate communication thread and leave the computational loops to the remaining threads. We call this "hybrid task mode," because it establishes a functional decomposition of tasks (communication vs. computation) across the resources (see Fig. 5 (c)): One thread

executes MPI calls only, while all others are used to copy data into send buffers, perform the spMVM with the local RHS elements, and finally (after all communication has finished) do the remaining spMVM parts. Since spMVM saturates at about 3–5 threads per locality domain (as shown in Fig. 4), at least one core per LD is available for communication without adversely affecting node-level performance. On architectures with SMT, like the Intel Westmere, there is also the option of using one compute thread per physical core and bind the communication thread to a logical core.

Apart from the additional memory traffic due to writing the result vector twice (see Sect. 3.2), another drawback of hybrid task mode is that the standard OpenMP loop worksharing directive cannot be used, since there is no concept of "subteams" in the current OpenMP standard. Work distribution is thus implemented explicitly, using one contiguous chunk of nonzeros per compute thread. It is possible to simplify the code by using OpenMP tasking along the lines of [4], but ccNUMA placement issues have to be taken into account if a team of threads spans more than a single LD [20].

4 Internode performance results and discussion

Figures 6 and 8 show strong scaling results for the two chosen matrices (HMeP and sAMG). We compare the hybrid kernel versions presented in Fig. 5 on the Cray XE6 and present the best result obtained on the Westmere cluster for reference (see [5] for a thorough discussion of those results). For the considered matrix sizes and largest node counts, a single spMVM takes a couple of milliseconds.

4.1 Testcase HMeP (see Fig. 6)

At one MPI process per physical core (left panel), there is hardly any discernible difference between the two vector modes (task mode is ruled out here, since no resources are left for communication threads). The overhead for the additional data transfer on the result vector does not seem to have much impact on the performance, which leads to the conclusion that load imbalance effects play an important role at large process counts. The Westmere cluster delivers roughly the same performance per node as the XE6, with task mode being the best variant. In this case, one SMT thread per core is used for computation and the other performs communication. This could also be a usable model for MPI libraries that support progress threads, given that thread-core affinity issues can be appropriately resolved.

With one process per ccNUMA locality domain (middle panel) there is a slight advantage for task mode (one out of six cores per domain runs a communication thread), which can be attributed to the smaller number of MPI processes; now there is a chance for the hidden communication to pay off. The Westmere system clearly outperforms the XE6 by about 25% at larger node counts, so the per-node performance advantage of the Magny Cours cannot be saved into the highly parallel domain. Since the memory bus of a Westmere socket is already saturated with four threads, it does not make a difference whether six worker threads are used with one communication thread on a virtual core, or whether a physical core is devoted to communication.

Best results are obtained on the XE6 when using a single MPI process with 24 threads per node (right panel). The performance advantage of task mode versus the other variants is close to 30%. The symbols in Fig. 6 indicate the 50% parallel efficiency point (with respect to the best single-node XE6 performance of 5.3 GFlop/s, as reported in Fig. 4) on each data set. In practice one would not go beyond this number of nodes because of bad resource utilization. For the matrix and the system under investigation it is clear that task mode allows strong scaling to much higher levels of parallelism than any variant of vector mode.

Contrary to expectations based on the single-node performance numbers (Fig. 4), the Cray XE6 can generally not match the performance of the Westmere cluster at larger node counts, with the exception of pure MPI where both are roughly on par (left panel, Westmere results for task mode with one communication thread per physical core). When using threaded MPI processes (middle and right panel), task mode performs best on the Cray system. The advantage over the other kernel variants is by far not as pronounced as on Westmere, however. We have observed a strong influence of job topology and machine load on the communication performance over the 2D torus network. Figure 7 shows two scaling runs with the HMeP matrix and pure MPI (one process per core) up to 32 nodes. In one case, only the required 32 nodes were allocated through the batch system; in the other case, 150 nodes (almost the full machine) were requested. Since other user jobs were running one the rest of the machine, the influence of machine load on performance is evident. Also, as sparse MVM requires significant non-nearest-neighbor communication with growing process counts, the nonblocking fat tree network on the Westmere cluster seems to be better suited for this kind of problem. The results presented in Figs. 6 and 8 are best values obtained on a dedicated machine.

Interestingly, the hybrid vector mode variants with one MPI process per LD or per node show almost the same performance as pure MPI on the XE6. There is also a universal drop in scalability beyond about six nodes, which is largely independent of the particular hybrid mode. This can be ascribed to a strong decrease in overall internode communication volume when the number of nodes is small. The effect is somewhat less pronounced for pure MPI, since the overhead of intranode message passing cannot be neglected.

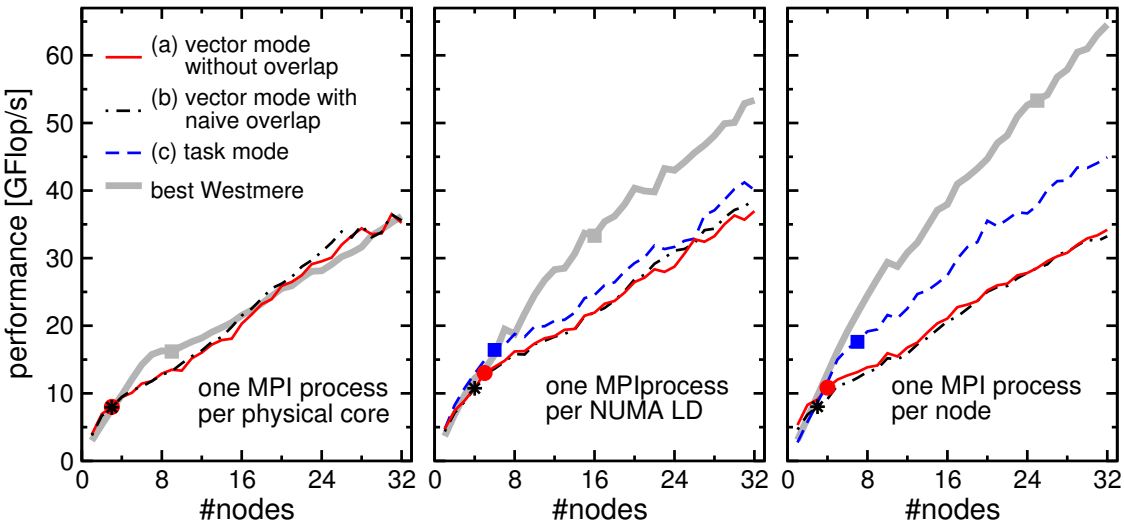


Figure 6: Strong scaling performance data for spMVM with the HMeP matrix on the Cray XE6 for different pure MPI and hybrid variants (kernel version (a) – (c)) as in Fig. 5). Symbols on each data set indicate the 50% parallel efficiency point with respect to the best single-node version. The best variant on the Westmere cluster system (task mode in all cases) is shown for reference (see text for details).

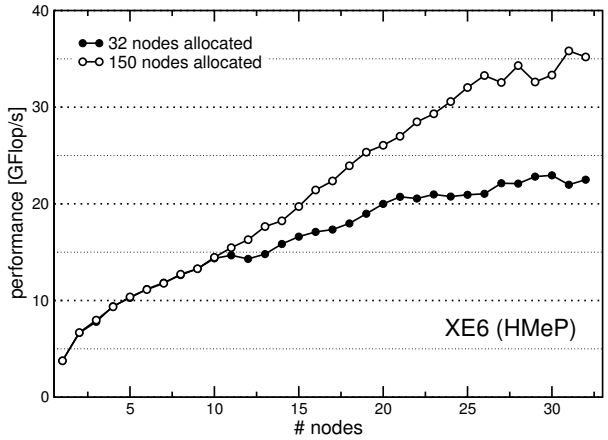


Figure 7: Scaling behavior of HMeP on the Cray XE6 on up to 32 nodes with different numbers of nodes allocated on the 176-node machine.

4.2 Testcase sAMG (see Fig. 8)

The sAMG matrix has much weaker communication requirements than HMeP, and the impact of load imbalance is very small. On the Cray XE6, vector mode without overlap performs best across all hybrid modes, with a measurable advantage when running one MPI process with six threads per LD. Surprisingly, vector mode with naive overlap is significantly slower than the other variants; a close inspection of message sizes, communication timings, and the influence of eager mode would be in order to fully understand this behavior. This aspect is still to be investigated. Comparison with the Westmere performance data (vector mode without overlap in all cases) shows that the Cray XE6 is able to maintain its node-level performance advantage also in the parallel case.

These observations support the general rule that it makes no sense to consider MPI+OpenMP hybrid programming if the pure MPI code already scales well and behaves in accordance with a single-node performance model.

5 Summary and outlook

Starting from the observation that truly asynchronous nonblocking point-to-point communication is not possible in most current MPI implementations (although the hardware is often able to support it), we have investigated the performance properties of different pure MPI and MPI+OpenMP hybrid variants of sparse matrix-vector multiplication on Cray XE6 and Westmere-based InfiniBand cluster systems, using two matrices with significantly different sparsity patterns. The single-node performance model and analysis on Intel Westmere and AMD Magny Cours processors showed that memory-bound sparse MVM saturates the memory bus of a ccNUMA locality domain already at about four threads, leaving free resources for implementing explicit computation/communication overlap. Explicit overlap enables substantial performance gains in strong scaling scenarios for communication-bound problems, especially when running one process per ccNUMA domain or per node. Since the communication thread can run on

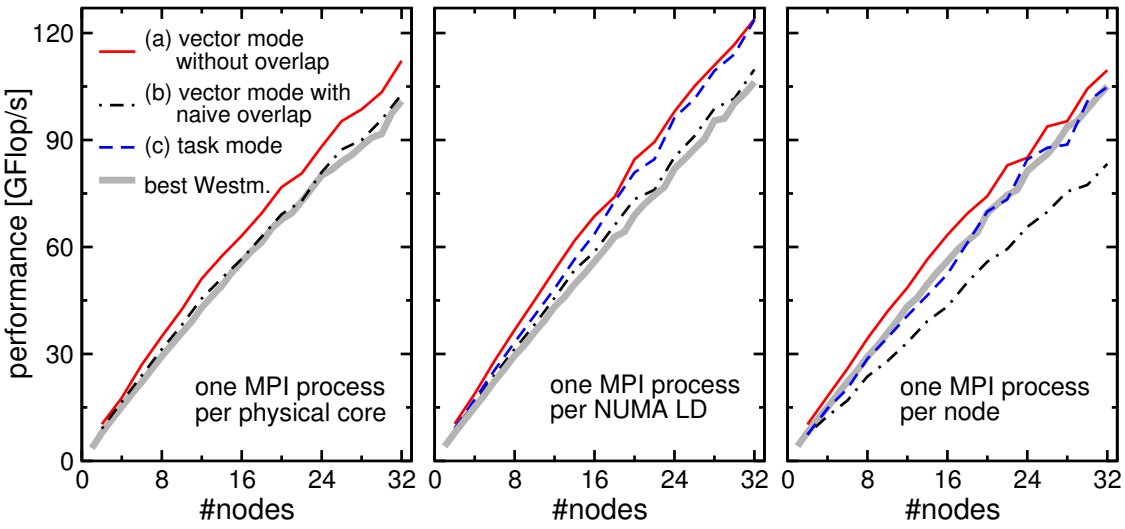


Figure 8: Strong scaling performance data for spMVM with the sAMG matrix (same variants as in Fig. 6). Parallel efficiency is above 50% for all versions up to 32 nodes. The Westmere cluster performed best in vector mode without overlap for all cases, just like the Cray XE6.

a virtual core, MPI implementations could use the same strategy for internal "progress threads" and so enable asynchronous communication without changes in MPI-only user code. This leaves some potential to be exploited in upcoming AMD processors ("Bulldozer"), since the current AMD architecture does not feature hardware threads.

Future work will cover a more complete investigation of load balancing effects, and a careful analysis of the performance properties of the Cray XE6 system. We will also investigate further the asynchronous communication capabilities of MPI implementations for small messages.

Acknowledgments

We thank J. Treibig and R. Keller for valuable discussions, A. Basermann for providing the RCM transformation, and K. Stüben and H. J. Plum for providing and supporting the AMG test case. We acknowledge financial support from KONWIHR II (project HQS@HPC II) and thank NERSC for granting access to their Cray systems.

About the authors

Georg Hager holds a PhD in computational physics, and is now a senior research scientist at Erlangen Regional Computing Center (RRZE), which is part of the University of Erlangen-Nuremberg, Germany. Gerald Schubert holds a PhD in theoretical physics and is a postdoctoral researcher with the Institute of Physics at the University of Greifswald, Germany. Thomas Schoenemeyer is associate director of the Swiss National Supercomputing Centre (CSCS) in Manno, Switzerland, and has served for SGI and NEC before. Gerhard Wellein holds a PhD in theoretical physics and a professorship for high performance computing in the department for computer science at the University of Erlangen-Nuremberg. He heads the HPC services group at RRZE.

References

[1] G. Hager and G. Wellein: *Introduction to High Performance Computing for Scientists and Engineers.* CRC Press, ISBN 978-1439811924, (2010).

[2] *MPI: A Message-Passing Interface Standard. Version 2.2*, September 2009. `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`

[3] G. Hager, G. Jost, and R. Rabenseifner: *Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes.* In: Proceedings of the Cray Users Group Conference 2009 (CUG 2009), Atlanta, GA, USA, May 2009.

[4] A. Koniges, R. Preissl, J. Kim, D. Eder, A. Fisher, N. Masters, V. Mlaker, S. Ethier, W. Wang, M. Head-Gordon: *Application Acceleration on Current and Future Cray Platforms.* In: Proceedings of the Cray Users Group Conference 2010 (CUG 2010), Edinburgh, Scotland, May 2010. `http://www.nersc.gov/news/reports/technical/Cug2010Alice.pdf`

[5] G. Schubert, G. Hager, H. Fehske and G. Wellein: *Parallel sparse matrix-vector multiplication as a test case for hybrid MPI+OpenMP programming.* Accepted for the Workshop on Large-Scale Parallel Processing (LSPP 2011), May 20th, Anchorage, AK, USA.

[6] G. Wellein, G. Hager, A. Basermann, and H. Fehske: *Fast sparse matrix-vector multiplication for TFlop/s computers.* In: High Performance Computing for Computational Science — VECPAR2002, LNCS 2565, Eds. J. Palma et al., Springer Berlin (2003). DOI 10.1007/3-540-36569-9_18

[7] R. Rabenseifner and G. Wellein: *Communication and Optimization Aspects of Parallel Programming Models on Hybrid Architectures.* International Journal of High Performance Computing Applications **17**, 49–62, February 2003. DOI 10.1177/1094342003017001005

[8] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, J. Demmel: *Optimization of sparse matrix-vector multiplications on emerging multicore platforms.* Parallel Comput. **35**, 178 (2009)

[9] R. Geuss and S. Röllin: *Towards a fast parallel sparse symmetric matrix-vector multiplication.* Parallel Computing **27** (1), 883–896 (2001). DOI 10.1016/S0167-8191(01)00073-4

[10] M. Krotkiewski and M. Dabrowski: *Parallel symmetric sparse matrix-vector product on scalar multi-core CPUs.* Parallel Computing **36** (4), 181–198 (2010). DOI 10.1016/j.parco.2010.02.003

[11] `http://code.google.com/p/likwid`

[12] R. Barrett *et al.*: *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, ISBN 978-0898713282, (1993).

[13] A. Weiße, G. Wellein, A. Alvermann, and H. Fehske: *The kernel polynomial method.* Rev. Mod. Phys. 78, 275 (2006).

[14] A. Weiße and H. Fehske: *Chebyshev expansion techniques.* In: Computational Many Particle Physics, Lecture Notes in Physics 739, pp. 545–577, Springer Berlin Heidelberg (2008).

[15] H. Fehske, G. Wellein, G. Hager, A. Weiße, and A. R. Bishop: *Quantum lattice dynamical effects on the single-particle excitations in 1D Mott and Peierls insulators.* Phys. Rev. B **69**, 165115 (2004).

[16] E. Cuthill and J. McKee: *Reducing the bandwidth of sparse symmetric matrices.* In: Proceedings of the 1969 24th national conference (ACM '69), ACM, New York, NY, USA, 157–172. DOI 10.1145/800195.805928

[17] K. Stüben: *An Introduction to Algebraic Multigrid.* In: Multigrid: Basics, Parallelism and Adaptivity, Eds. U. Trottenberg et al., Academic Press (2000).

[18] `http://www.scai.fraunhofer.de/en/business-research-areas/numerical-software/products/samg.html`

[19] G. Schubert, G. Hager, and H. Fehske: *Performance limitations for sparse matrix-vector multiplications on current multicore environments.* In: High Performance Computing in Science and Engineering, Garching/Munich 2009, Eds. S. Wagner et al., pp. 13–26, Springer Berlin Heidelberg (2010).

[20] M. Wittmann and G. Hager: *Optimizing ccNUMA locality for task-parallel execution under OpenMP and TBB on multicore-based systems.* Unpublished technical report. Preprint: `http://arxiv.org/abs/1101.0093`