Computing (2013) 95:1121–1136
DOI 10.1007/s00607-013-0324-2

MPI + MPI: a new hybrid approach to parallel programming with MPI plus shared memory

Torsten Hoefler · James Dinan · Darius Buntinas · Pavan Balaji · Brian Barrett · Ron Brightwell · William Gropp · Vivek Kale · Rajeev Thakur

Received: 22 December 2012 / Accepted: 25 April 2013 / Published online: 19 May 2013

Abstract Hybrid parallel programming with the message passing interface (MPI) for internode communication in conjunction with a shared-memory programming model to manage intranode parallelism has become a dominant approach to scalable parallel programming. While this model provides a great deal of flexibility and performance potential, it saddles programmers with the complexity of utilizing two parallel

T. Hoefler (✉)
ETH Zurich, Zurich, Switzerland
e-mail: htor@inf.ethz.ch

J. Dinan · D. Buntinas · P. Balaji · R. Thakur
Argonne National Laboratory, Argonne, IL, USA
e-mail: dinan@mcs.anl.gov

D. Buntinas
e-mail: buntinas@mcs.anl.gov

P. Balaji
e-mail: balaji@mcs.anl.gov

R. Thakur
e-mail: thakur@mcs.anl.gov

B. Barrett · R. Brightwell
Sandia National Laboratories, Sandia, NM, USA
e-mail: bwbarre@sandia.gov

R. Brightwell
e-mail: rbbrigh@sandia.gov

W. Gropp · V. Kale
University of Illinois at Urbana-Champaign, Urbana, IL, USA
e-mail: wgropp@illinois.edu

V. Kale
e-mail: vivek@illinois.edu

programming systems in the same application. We introduce an MPI-integrated shared-memory programming model that is incorporated into MPI through a small extension to the one-sided communication interface. We discuss the integration of this interface with the MPI 3.0 one-sided semantics and describe solutions for providing portable and efficient data sharing, atomic operations, and memory consistency. We describe an implementation of the new interface in the MPICH2 and Open MPI implementations and demonstrate an average performance improvement of 40 % to the communication component of a five-point stencil solver.

Keywords MPI-3.0 · Shared memory · Hybrid parallel programming

Mathematics Subject Classification 68N19 other progamming techniques (objects-oriented, sequential, concurrent, automatic, etc.)

1 Introduction

The message passing interface (MPI [1]) has been the dominant parallel programming model since the mid-1990s. One important reason for this dominance has been its ability to deliver portable performance on large, distributed-memory massively parallel processing (MPP) platforms, large symmetric multiprocessing (SMP) machines with shared memory, and hybrid systems with tightly coupled SMP nodes. For the majority of these systems, applications written with MPI were able to achieve acceptable performance and scalability. However, recent trends in commodity processors, memory, and networks have created the need for alternative approaches. The number of cores per chip in commodity processors is rapidly increasing, and memory capacity and network performance are not able to keep up the same pace. Because memory capacity per core is decreasing, mapping a single operating system process to an MPI rank and assigning a rank per core severely limit the problem size per rank. In addition, MPI's single-copy model for both message passing and one-sided communication exacerbate the memory bandwidth problem by using intranode memory-to-memory copies to share data between ranks. Moreover, network interfaces are struggling to support the ability for all cores on a node to use the network effectively. As a result, applications are moving toward a hybrid model mixing MPI with shared-memory models that attempt to overcome these limitations [2,3].

A relatively straightforward and incremental approach to extending MPI to support shared memory has recently been approved by the MPI Forum. Several functions were added that enable MPI ranks within a shared-memory domain to allocate shared memory for direct load/store access. The ability to directly access a region of memory shared between ranks is more efficient than copying and reduces stress on the memory subsystem. Sharing a region of memory between ranks also overcomes the per core memory capacity issue and provides more flexibility in how the problem domain is decomposed. This approach reduces the amount of memory consumed for some data structures such as read-only databases that replicate state across all ranks. From a programming standpoint, providing shared memory supports structured programming, where data is private until it is explicitly shared. The alternative, where data is shared

and must be explicitly made private, introduces more complexity into an existing MPI application and the associated MPI implementation. Shared memory is also nearly ubiquitous, given the prevalence of multicore processors.

This paper describes these recent extensions to the MPI standard to support shared memory, discusses implementation options, and demonstrates the performance advantages of shared memory for a stencil benchmark.

1.1 Motivation and related work

Support for shared memory in MPI has been considered before, but a number of factors have made such support increasingly compelling. In particular, although POSIX shared memory can be used independently from MPI, the POSIX shared-memory model has several limitations that can be overcome by exposing it through MPI. First, POSIX shared-memory allocation is not a collective operation. One process creates a region of memory and allows other processes to attach to it. Making shared-memory creation collective offers an opportunity to optimize the layout of the memory based on the layout of the ranks. Since the MPI implementation has knowledge of the layout of the shared-memory region, it may be able to make message-passing operations using this region more efficient. For example, MPI may be able to stripe messages over multiple network interfaces, choosing the interface that is closest to the memory being sent. Integration between the MPI runtime system and shared memory simplifies shared-memory allocation and cleanup. Relying on an application using POSIX shared memory directly to clean up after abnormal termination has been problematic. Having the MPI implementation be responsible for allocating and freeing shared memory is a better solution. Knowledge of shared memory inside the MPI implementation also provides better support and integration with MPI tools, such as correctness and performance debuggers. Furthermore, nearly all MPI implementations already have the infrastructure for allocating and managing shared memory since it is used for intranode data movement, so the burden on existing implementations is light.

Previous work on efficiently supporting MPI on shared-memory systems has concentrated mostly on mapping an MPI rank to a system-level or user-level thread [4–8]. This approach allows MPI ranks to share memory inside an operating system process, but it requires program transformation or knowledge on the part of the programmer to handle global and static variables appropriately. Systems specifically aimed at mixing MPI and shared memory have been developed, effectively augmenting MPI with shared-memory capabilities as the new extensions do. LIBSM [9] and the Unified Parallel System [10] are two such systems developed to support the ability for applications to use both MPI and shared memory efficiently. However, neither of these systems actually made internal changes to the MPI implementation; rather, they provided an application-level interface that abstracted the capabilities of message passing and shared memory.

The need for shared memory in MPI was brought up at the Forum by R. Brightwell, who proposed a malloc/free interface that did not define synchronization semantics. T. Hoefler later proposed to merge this functionality into the newly revamped one-sided communication interface. Hoefler and J. Dinan brought forward a concrete proposal, which was further developed by the MPI Forum remote memory access (RMA) work-

ing group. The Forum eventually voted for inclusion in MPI-3. The interface described in this paper is the interface in MPI-3.

This paper is a more detailed version of [11].

Using shared memory between processes is not new, of course. Unix System V defined a way for processes to share memory, and this has been used by applications (including MPI implementations). One example where the sharing memory between processes was the key part of the programming interface was MLP (multi-level parallelism) [12], developed at NASA and used for production computational fluid dynamics codes. This work showed that direct access to shared memory by the application processes provides a significant performance benefit. However, this model and ones like it never achieved the ubiquity of MPI and hence are rarely used directly by computational science applications.

2 Extending MPI with integrated shared memory

MPI's remote memory access interface defines one-sided communication operations, data consistency, and synchronization models for accessing memory regions that are exposed through MPI windows. The MPI-2 standard defined conservative but highly portable semantics that would still guarantee correct execution on systems without a coherent memory subsystem. In this model, the programmer reasons about the data consistency and visibility in terms of separate private (load/store access) and public (RMA access) copies of data exposed in the window.

The MPI-3 RMA interface extends MPI-2's *separate* memory model with a new *unified* model, which provides relaxed semantics that can reduce synchronization overheads and allow greater concurrency in interacting with data exposed in the window. The unified model was added in MPI-3 RMA to enable more efficient one-sided data access in systems with coherent memory subsystems. In this model, the public and private copies of the window are coherent, and updates to either "copy" automatically propagate. Explicit synchronization operations can be used to ensure completion of individual or groups of operations.

The unified memory model defines an efficient and portable mechanism for one-sided data access, including the needed synchronization and consistency operations. We observe that this infrastructure already provides several important pieces of functionality needed to define a portable, interprocess shared-memory interface. We now discuss the additional functionality, illustrated in Fig. 1, that is needed to extend the RMA model to support load/store accesses originating from multiple origin processes to data exposed in a window. In addition, we discuss new functionality that is needed to allow the user to query system topology in order to identify groups of processes that communicate through shared memory.

2.1 Using the RMA interface for shared memory

In the MPI-2 one-sided communication interface, the user first allocates memory and then exposes it in a window. This model of window creation is not compatible with the interprocess shared-memory support provided by most operating systems, which

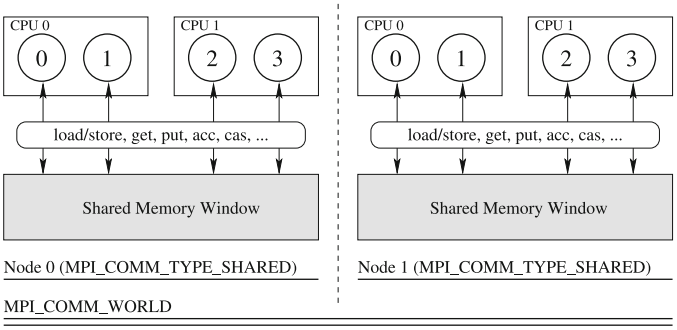


Fig. 1 Interprocess shared-memory extension using MPI RMA; an execution with two nodes is shown, and a shared memory window is allocated within each node. The *circles* represent MPI ranks running on two dual-core dual-socket nodes

require the use of special routines to allocate and map shared memory into a process's address space. Therefore, MPI-3 defines a new routine, MPI_Win_allocate_shared, that collectively allocates and maps shared memory across all processes in the given communicator.

CPU load and store instructions are similar to one-sided get and put operations. In contrast with get/put, however, load/store operations do not pass through the MPI library; and, as a result, MPI is unaware of which locations were accessed and whether data was updated. Therefore, the separate memory model conservatively defines store operations as updating to full window, in order to prevent data corruption on systems whose memory subsystem is not coherent. However, an overwhelming majority of parallel computing systems do provide coherent memory, and on these systems this semantic is unnecessarily restrictive. Therefore, MPI-3 defines a unified memory model where store operations do not conflict with accesses to other locations in the window. This model closely matches the shared-memory programming model used on most systems, and windows allocated by using MPI_Win_allocate_shared are defined to use the unified memory model.

2.2 Mapping of interprocess shared memory

Each rank in the shared-memory window provides an allocation size, and a shared memory segment of at least the sum of all sizes is created. Specifying a per rank size rather than a single, global size allows implementations to optimize data locality in nonuniform memory architectures. By default, the allocated shared-memory region is required to be contiguous. That is, the memory region associated with rank N in a given window must be located directly before the memory region associated with rank $N + 1$. The info key alloc_shared_noncontig allows the user to relax this allocation constraint. When this key is given, MPI can map the segments belonging to each process into noncontiguous locations. This approach can enable better performance by allowing MPI to map each segment on a page boundary, potentially eliminating negative cache and NUMA effects.

Many operating systems make it difficult to ensure that shared memory is allocated at the same virtual address across multiple processes. The MPI one-sided interface,

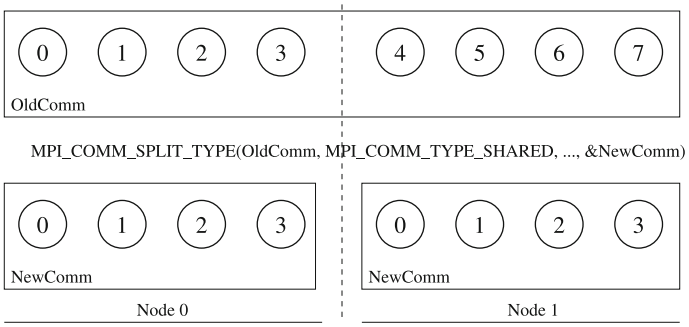


Fig. 2 MPI_Comm_split_type can be used to create communicators suitable for use by MPI_Win_allocate_shared from any intracommunicator

which encourages the dynamic creation of shared-memory regions throughout an application's life, exacerbates this problem. MPI_Win_allocate_shared does not guarantee the same virtual address across ranks, and it returns only the address of the shared-memory region for the local rank. MPI_Win_shared_query provides a query mechanism for determining the base address in the current process and size of another process's region in the shared-memory segment. The address of the absolute beginning of the window can be queried by providing MPI_PROC_NULL as the rank argument to this function, regardless of whether rank zero specified a size greater than zero.

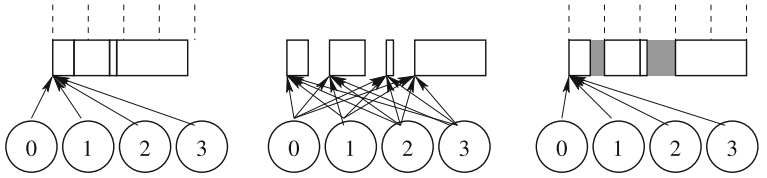
2.3 Querying machine topology

The MPI_Win_allocate_shared function expects the user to pass a communicator on which a shared-memory region can be created. Passing a communicator where this is not possible is erroneous. In order to facilitate the creation of such a "shared memory capable" communicator, MPI-3 provides a new routine, MPI_Comm_split_type. This function is an extension of the MPI_Comm_split functionality, with the primary difference being that the user passes a type for splitting the communicator instead of a color. Specifically, the MPI-3 standard defines the type MPI_COMM_TYPE_SHARED, which splits a communicator into subcommunicators on which it is possible to create a shared-memory region, as shown in Fig. 2.

The MPI_Comm_split_type functionality also provides an info argument that allows the user to request architecture-specific information that can be used to restrict the communicator to span only a NUMA socket or a shared cache level, for example. While the MPI-3 standard does not define specific info keys, most implementations are expected to provide NUMA and cache management capabilities through these info keys.

3 Implementation of shared-memory RMA

The shared-memory RMA interface has been implemented in both MPICH and Open MPI by using similar techniques. In this section we describe the steps required for the MPI library to allocate a shared window; we also provide implementation details.



(a) Contiguous **(b)** Noncontig Separate **(c)** Noncontig Padded

Fig. 3 Shared-memory window allocation strategies. *Dotted lines* in **a** and **c** represent page boundaries. In **b** each window segment is allocated in a separate shared-memory region and is page aligned **a** contiguous **b** noncontig separate **c** noncontig padded

The *root* (typically the process with rank 0 in the associated communicator) allocates a shared-memory region that is large enough to contain all the window segments of all processes sharing the window. Once the shared-memory region has been created, information identifying the shared-memory region is broadcast to the member processes, which then attach to it. At any process, the base pointer of a window segment can be computed by knowing the size and base pointer of the previous window segment: the base pointer of the first window segment, segment 0, is the address of where the shared-memory segment was attached; and the base pointer of segment i is $base\_ptr_i = base\_ptr_{i-1} + seg\_size_{i-1}$.

Scalability needs to be addressed for two implementation issues: (1) computing the sum of the shared window segments in order to determine the size of the shared-memory segment and (2) computing the base pointer of a window segment. For windows with a relatively small number of processes, an array of the segment size of each process can be stored locally at each process by using an all-gather operation. From this array, the root process can compute the size of the shared-memory segment, and each process can compute the base pointer of any other segment. For windows with a large number of processes, however, the offsets may be stored in a shared-memory segment, with scalable collectives (reduce, broadcast, exscan) used to compute sizes and offsets.

When the alloc_shared_noncontig info key is set to "true," the implementation is not constrained to allocate the window segments contiguously; instead, it can allocate each window segment so that its base pointer is aligned to optimize memory access. Individual shared-memory regions may be exposed by each rank, an approach that can be used to provide optimal alignment and addressing but requires more state. An alternative implementation is to allocate the window as though it was allocated contiguously, except that the size of each window segment is rounded up to a page boundary. In this way each window segment is aligned on a page boundary, and shared state can be used to minimize resource utilization. Both MPICH and Open MPI use the latter approach.

Figure 3 shows the three shared-memory allocation strategies discussed above. In Fig. 3a we see the contiguous memory allocation method. The figure shows four processes each of which has the entire memory region attached. The shared-memory region contains four window segments of different sizes. Figure 3b and c show noncontiguous allocations. In Fig. 3b each window segment is allocated in a separate

shared-memory region. Each process attaches all the memory regions. In Fig. 3c a single shared-memory region is attached by each process. Each window segment is padded out to a window boundary. The first and third segments do not end on a page boundary; thus, we see that those segments are padded so that the next window segment starts on a page boundary.

Synchronization operations must provide processor memory barriers to ensure consistency semantics but otherwise are straightforward to implement. Because of the direct memory access available for all target operations, communication calls may be implemented as memory copies performed during the communication call itself. While an implementation could choose to implement the accumulate operations by using processor atomics, locks and memory copies can also provide the required semantics. Both MPICH and Open MPI use a spinlock per target memory region to implement accumulate operations, because of the simplicity of implementation and greater portability.

4 Issues with a library interface to shared memory

Several issues can arise when implementing shared-memory semantics as a library [13, 14]. Like Pthreads, MPI does not specify memory semantics for load/store accesses to the shared-memory window within an epoch. The user must be aware of this feature and must guarantee the required consistency using techniques outside the scope of MPI (e.g., language features or inline assembly). Note that, like on traditional MPI windows, MPI RMA synchronization operations (e.g., fence, lock, or sync) have clearly specified memory semantics and can be used to synchronize memory accesses.

The main issue arises from the fact that the compiler is not aware that programs are executed in a multithreaded context and that serial optimizations may break anticipated parallel semantics. For example, Boehm et al. [14] assume the following code to synchronize the access of two processes to a shared-memory window:

```
// assuming two processes in shmcomm, each allocating one int, contiguously
MPI_Win_allocate_shared(sizeof(int), info, shmcomm, &mem, &win);
// query address of process 0 in my local memory
MPI_Win_shared_query(win, 0, &sz, &ptr);
int *x = &ptr[0]; // assign variables
int *done = &ptr[1];

if(r == 0) {          // process 0
  *x = ...            // initialize x
  *done = 1;          // set flag
}

if(r == 1) {          // process 1
  while(!(*done)) {}  // wait for flag
  ... = *x;           // read x
}
```

Similar to the example in [14], a good compiler could use a simple points-to-analysis [15], determine that *done is not changed in the loop body, and optimize

process 1's loop to tmp=*done; while(!tmp) { } to avoid the memory references. More examples, such as register promotion and performance issues, can be found in [13,14].

A possible solution would be to make the compiler aware of the fact that the thread is executed in a *parallel environment* or to prohibit certain optimizations. For example, a compiler would not be allowed to add additional reads or writes to and from a shared memory segment, and it would also not be allowed to perform any optimization affecting memory in a shared-memory segment across any MPI (or generally any) function call. Those two simple rules would prevent most of the erroneous transformations known today. However, to define the sets of correct and incorrect compiler transformations, one would need to define a memory model for accessing MPI shared-memory windows.

In this model, MPI synchronization calls act as optimization barriers that synchronize memory, and MPI_Win_sync can be used similar to a memory fence because of its semantics that close and open an epoch. The reason rests with the opaque nature of MPI calls (they are often simply linked in as object files) for today's existing C/C++ compilers, which have to assume that MPI functions have arbitrary effects on the global state of the program. However, standard Fortran compilers, and C/C++ compilers using whole program analysis and interprocedural optimizations (IPO), may still cause problems when applying heavy optimizations assuming a serial execution of the code (we suggest that compiler and library writers cooperate to avoid such problems).

The MPI Forum decided to omit this specification because of missing practical experience (the Java [16] and C++ [17] memory models are rather new) and the resulting doubts about the impact on performance of such a specification. Another limitation of MPI, as a library, is that a fully specified memory model may require many additional, often expensive, library calls to emulate strong models such as sequential consistency (cf. *volatile* in the Java memory model [16] or *atomic* in the C++ memory model [17]).

5 Use cases and evaluation

Shared-memory windows in MPI programs have multiple effects on future parallel programming techniques. Current scientific applications often use OpenMP to enable sharing of large data structures (e.g., hash tables or lookup tables or databases) among cores inside a compute node. This approach requires using two different models of parallelization: MPI and a carefully crafted OpenMP layer that enables scalability to the large core counts (32–64) in today's architectures. Doing so often requires an "MPI-style" domain decomposition of the OpenMP parts, effectively leading to a complex two-stage parallelization of the program. Shared-memory windows allow a structured approach to this issue in that OpenMP can be used where it is most efficient (e.g., at the loop level) and shared memory can be shared across different MPI processes with a single level of domain decomposition.

A second use-case is to use shared-memory windows for fast intranode communications. Here, the user employs a two-level parallelization in order to achieve the highest possible performance using true zero-copy mechanisms (as opposed to MPI's mandated single-copy from send buffer to receive buffer). This has the advantage over a purely threaded approach that memory is explicitly shared and heap corruption due

to program bugs is less likely (cf. [18]). An example of this benefit explored with an early prototype of the shared-memory extensions can be found in [19]. This work demonstrates the incremental approach of incorporating shared memory into an MPI application in order to reduce the iteration count of the linear solver portion of an application. The rest of the application, which performs and scales well, can remain unchanged and largely unaware of the use of shared memory.

Shared-memory regions can also help better support the use of accelerators within an MPI application. For example, if an application is running with one MPI rank per core and all ranks wish to transfer data to a GPU, it can be challenging to coordinate the transfer of data between the host memory of each rank and GPU memory. Using shared memory, one rank can be responsible for transferring data between the host and the device, reducing the amount of coordination among ranks.

5.1 Finite-volume mini-application

Preconditioned iterative methods for solving linear systems can benefit from a hybrid MPI plus shared-memory approach. Solver implementations based on an MPI-everywhere model often suffer from poor scalability for large numbers of MPI processes, because the number of iterations per linear solve step increases significantly as the number of MPI ranks, and thus the number of subdomains, increases. Extra solver iterations typically consume an increasingly large percentage of overall application runtime as the number of MPI ranks grows. An approach to addressing this issue is to use shared memory to reduce the number of subdomains in the solver. Although the rest of the application would continue to operate using a single subdomain per core, the solver library could organize the data into fewer subdomains so that multiple cores work together on a single larger subdomain. Mathematically, this approach improves the convergence rate and robustness of the solver.

To illustrate this approach, we use a mini-application from the Mantevo project [20] called HPCPCG (HPC preconditioned conjugate gradient). This miniapp is designed to encapsulate some of the performance characteristics of an unstructured finite-volume application. It partitions a three-dimensional domain in the z-dimension and stores data in an unstructured fashion. It implements a conjugate-gradient iterative method preconditioned by a symmetric Gauss-Seidel sweep, implemented as one lower triangular solve and one upper triangular solve [21] using a previously developed level-set triangular solver implementation [22,23]. A level set is calculated after expressing the data dependencies as a directed acyclic graph (DAG). The level sets of this DAG represent sets of row operations in the triangular solve that can be done independently by using threads with synchronization barriers occurring after each level. This approach is most beneficial for solving triangular systems resulting from incomplete factorizations, where the resulting matrix factors are sparse enough to yield sufficiently large levels and mitigate the synchronization costs.

The original implementation of HPCPCG was MPI-only. Like most codes that use domain decomposition, the sweep is restricted to each subdomain assigned to each MPI rank. The sweep is done independently in each rank without any communication, which leads to iteration inflation. Using the shared memory extensions, the preconditioner

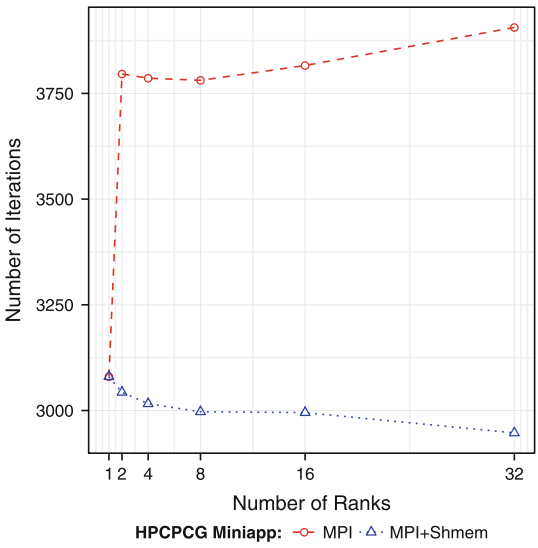


Fig. 4 Comparison of iteration count for the HPCPCG mini-application using MPI-only and MPI plus shared memory

can operate on fewer, larger domains. HPCPCG was modified so that one rank per node is responsible for the preconditioning using a larger domain corresponding to the combined domains of all the ranks on a node. During the preconditioning step, all ranks can operate on a single domain directly, leading to a reduction in the iteration count over each rank operating on the domain independently. A multithreaded approach would lead to the same reduction in iteration count, but using the shared memory extensions allows for an incremental approach to improving the performance of the solver without impacting the rest of the application. Applications that use the solver can continue to use an MPI-everywhere approach, while only the solver needs minor changes to exploit the shared memory capability.

Figure 4 shows the iteration count for HPCPCG running on a 2 GHz Intel Xeon Nehalem X7550 with 4 sockets and 8 cores per socket, 512 GB of memory, running RHEL 6 with a 256x256x1024 grid size using strong scaling. HPCPCG reports the number of iterations needed for convergence to a specified tolerance for the MPI-only versus MPI plus shared-memory version. As expected, the MPI-only version requires more iterations to converge than does the shared-memory variant that uses larger subdomains in the preconditioning phase.

5.2 Quantum Monte Carlo code example

The quantum Monte Carlo method (QMC) is a highly scalable method for solving the many-body Schrödinger equation numerically, with a limited set of approximations relative to density-functional or diagrammatic many-body methods. QMC provides a general approach, which can be applied to problems relevant to chemistry, biology, materials science, and physics, particularly for systems with strong electron correlation. Starting from a reference wavefunction, QMC iteratively refines the wavefunc-

tion using a random walk in the high-dimensional space defined by the many-electron wavefunction, which is represented in a basis of one-electron functions.

In the popular QMCPack [24] and QWalk [25] quantum Monte Carlo applications, the reference wavefunction is captured by using the Einspline library [26], which uses the cubic B-spline basis. The resulting data, referred to as the ensemble data, is stored as a four-dimensional table of coefficients, where the first three dimensions are spatial, and the fourth dimension corresponds to the number of single particle orbitals in the system under simulation. Thus, the size of this table is proportional to the number of electrons in the physical system under analysis and the number of grid points in each spatial dimension.

Initially, it was possible to store one copy of the ensemble data per process on a node. But as memory per hardware execution thread is no longer increasing and scientific objectives have targeted larger systems, this is no longer possible [27]. Thus, developers of several QMC applications—notably QMCPack—have invested significant time and effort into hybridizing existing MPI code with shared-memory libraries, such as OpenMP, in order to share the coefficients table [28]. While the hybrid MPI+OpenMP model provides many useful capabilities to enable node-level parallelism, QMC applications implemented with only MPI are already relatively scalable and load balanced; thus, the shared-memory capability of this model is the primary feature needed by such applications.

Algorithm 5a shows pseudocode for the computational kernel at the core of the diffusion Monte Carlo algorithm and includes markup indicating what must be added to the existing MPI implementation in order to extend it to the hybrid MPI+OpenMP model. From this code, we see that significant effort must be invested to convert the computation to a two-level parallel structure that incorporates two parallel programming systems. A key challenge in the Hybrid MPI+OpenMP or MPI+threads models is the transition from the MPI process private-by-default model to the threaded shared-by-default model for global, heap, common block, and library data. This transition has resulted in the addition of two critical sections to regulate updates to data that is shared across walkers. Developers report that significant effort was required to duplicate unintentionally shared data in thread-private storage and to ensure safe access to shared data [28].

For comparison, we show the MPI with shared-memory windows pseudocode in Algorithm 5b. Because this code preserves MPI's private-by-default data model, we see that the DMC algorithm requires no changes. Instead, changes are localized to the allocation and creation of the ensemble data structure. A node communicator is created, followed by a shared-memory window, and finally one process per node populates the ensemble data table.

5.3 Five-point stencil kernel evaluation

We now evaluate the performance improvements that can be achieved with shared-memory windows using an application kernel benchmark. We prefer not to show the usual ping-pong benchmarks because they would simply show the MPI overhead versus the performance of the memory subsystem while hiding important effects caused

```
create_ensemble_data()

for generation g = 1 ... N_g do
  #pragma omp parallel for private(w,
      p, r_i', R', ...)
  for walker w = 1 ... N_w do
    Let R: position vector of N electrons
    R = {r_1 ... r_N}
    for particle p = 1 ... N_p do
      #pragma omp critical
      { δ = compute_new_position() }
      r_i' = r_i + δ
      R' = {r_1 ... r_i' ... r_N}
      ρ = Ψ_T(R')/Ψ_T(R)
      if accept(r → r') then
        #pragma omp critical
        { Update solution }
      end if
    end for
  end for
end for
```

(a) Hybrid MPI+OpenMP

```
node = Comm_split_type(SHR, WORLD)
buf = Win_allocate_shared(node, ...)
if Comm_rank(node) == 0 then
  create_ensemble_data(buf)
end if
Barrier(node)

for generation g = 1 ... N_g do
  for walker w = 1 ... N_w do
    Let R: position vector of N electrons
    R = {r_1 ... r_N}
    for particle p = 1 ... N_p do
      r_i' = r_i + δ
      R' = {r_1 ... r_i' ... r_N}
      ρ = Ψ_T(R')/Ψ_T(R)
      if accept(r → r') then
        Update solution
      end if
    end for
  end for
end for
```

(b) MPI+MPI

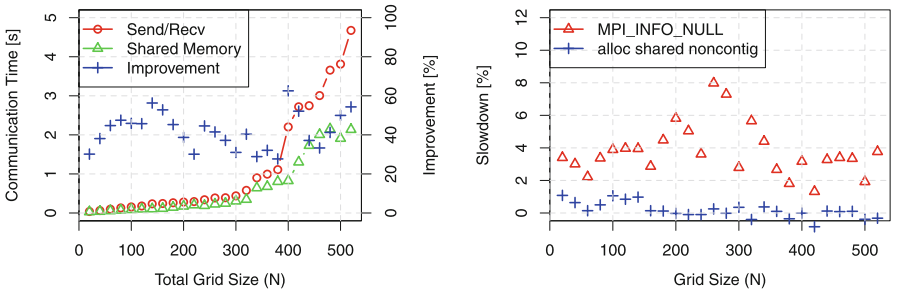
Fig. 5 Diffusion Monte Carlo kernel pseudocode, courtesy of [29], in the hybrid MPI+OpenMP and MPI with shared-memory window models **a** Hybrid MPI+OpenMP **b** MPI+MPI

by the memory allocation strategy. Instead, we use a simple, two-dimensional Poisson solver, which computes a heat propagation problem using a five-point stencil. The $N \times N$ input grid is decomposed in both dimensions by using `MPI_Dims_create` and `MPI_Cart_create`. The code adds one-element-deep halo zones for the communication. The benchmark utilizes nonblocking communication of $8 \cdot N$ Bytes in each direction to update the halo zones and `MPI_Waitall` to complete the communication. It then updates all local grid points before it proceeds to the next iteration.

The shared-memory implementation utilizes `MPI_Comm_split_type` to create a shared-memory communicator and allocates the entire work array in shared memory. Optionally, it provides the `alloc_shared_noncontig` info argument to allow the allocation of localized memory. The communication part of the original code is simply changed to `MPI_Win_fence` in order to ensure memory consistency and direct memory copies from remote to local halo zones. To simplify the example code, we assume that all communications are in shared memory only. The following listing shows the relevant parts of the code (variable declarations and array swapping are omitted for brevity).

```
MPI_Comm_split_type(comm, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, &shmcomm);

MPI_Win_allocate_shared(size*sizeof(double), info, shmcomm, &mem, &win);
MPI_Win_shared_query(win, north, &sz, &northptr);
MPI_Win_shared_query(win, south, &sz, &southptr);
MPI_Win_shared_query(win, east, &sz, &eastptr);
MPI_Win_shared_query(win, west, &sz, &westptr);
```



(a) Communication Overhead **(b)** Computation Slowdown

Fig. 6 Communication and computation performance for the five-point stencil kernel

```
for(iter=0; iter<niters; ++iter) {
  MPI_Win_fence(0, win); // start new access and exposure epoch
  if(north != MPI_PROC_NULL) // the north "communication"
    for(int i=0; i<bx; ++i) a2[ind(i+1,0)] = northptr[ind(i+1,by)];
  if(south != MPI_PROC_NULL) // the south "communication"
    for(int i=0; i<bx; ++i) a2[ind(i+1,by+1)] = southptr[ind(i+1,1)];
  if(east != MPI_PROC_NULL) // the east "communication"
    for(int i=0; i<by; ++i) a2[ind(bx+1,i+1)] = eastptr[ind(1,i+1)];
  if(west != MPI_PROC_NULL) // the west "communication"
    for(int i=0; i<by; ++i) a2[ind(0,i+1)] = westptr[ind(bx,i+1)];

  update_grid(&a1, &a2); // apply operator and swap arrays
}
```

We ran the benchmark on a six-core 2.2 GHz AMD Opteron CPU with two MPI processes and recorded communication and computation times. The domain was decomposed in the x (contiguous) direction, and both MPI processes ran in the same NUMA domain without internode communication. Open MPI and MPICH perform similarly because of the similar implementations; we focus on experimentation with the MPICH implementation.

Figure 6a shows the communication times of the send/recv version (red line with dots) and the shared-memory window versions (green line with triangles), as well as the communication time improvement of the shared-memory window version (blue crosses). In general, we show that the communication overhead for the shared-memory window version is 30–60 % lower than for the traditional message-passing approach. This is due to the direct memory access and avoided matching queue and function call costs.

Figure 6b shows the computation time of the shared-memory window version, that is, the time to update the inner grid cells relative to the computation time of the send/recv version. We observe a significant slowdown (up to 8 %) of the computation without the alloc_shared_noncontig argument. This is partially due to false sharing and the fact that the memory is local to rank 0. Indeed, the slowdown of the computation eliminated any benefit of the faster communication and made the parallel code slower.

Specifying alloc_shared_noncontig eliminates the overhead down to the noise (< 1.7 %) and leads to an improvement of the overall runtime.

6 Conclusions and outlook

In this work, we described an MPI standard extension to integrate shared-memory functionality into MPI-3.0 through the remote memory access interface. We motivated this new interface through several use-cases where shared-memory windows can result in improved performance, scaling, and capabilities. We discussed the design space for this new functionality and provided implementations in two major MPI implementations, which will both be available shortly in the official releases.

To evaluate the application-level impact of shared memory windows, we conducted a performance study using a heat-propagation 5-point stencil benchmark. The benchmark illustrated two important aspects: (1) an average 40 % reduction in data movement time compared with a traditional send/recv formulation and (2) the potentially detrimental slowdown of computation if false sharing and NUMA effects are ignored. By allowing the MPI implementation to automatically adjust the shared-memory mapping, we showed that these negative performance effects can be eliminated.

We plan to further investigate NUMA-aware allocation strategies, direct mapping of shared memory (e.g., XPMEM), and the effective use of the info argument to `MPI_Comm_split_type` to expand this routines topology querying capabilities. We also plan to apply the shared-memory extensions to incomplete factorization codes, as well as to a human heartbeat simulation code.

Acknowledgments We thank the members of the MPI Forum and the MPI community for their efforts in creating the MPI 3.0 specification. In addition, we thank Jeff RHammond for reviewing a draft of this article. This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357, under award number DE-FC02-10ER26011 with program manager Sonia Sachs, under award number DE-FG02-08ER25835, and as part of the Extreme-scale Algorithms and Software Institute (EASI) by the Department of Energy, Office of Science, U.S. DOE award DE-SC0004131. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energys National Nuclear Security Administration, under contract DE-AC-94AL85000.

References

1. MPI Forum (2012) MPI: a message-passing interface standard. version 3.0
2. Smith L, Bull M (2001) Development of mixed mode MPI/OpenMP applications. Sci Program 9(2,3):83–98
3. Rabenseifner R, Hager G, Jost G (2009) Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In: Proceedings of the 17th Euromicro international conference on parallel, distributed and network-based processing
4. Demaine E (1997) A threads-only MPI implementation for the development of parallel programs. In: Proceedings of the 11th international symposium on HPC systems. pp 153–163
5. Bhargava P (1997) MPI-LITE: multithreading support for MPI. http://pcl.cs.ucla.edu/projects/sesame/mpi_lite/mpi_lite.html
6. Shen K, Tang H, Yang T (1999) Adaptive two-level thread management for fast MPI execution on shared memory machines. In: Proceedings of the ACM/IEEE conference on supercomputing

7. Tang H, Shen K, Yang T (2000) Program transformation and runtime support for threaded MPI execution on shared memory machines. ACM Trans Program Lang Syst 22:673–700
8. Pérachec M, Carribault P, Jourdren H (2009) MPC-MPI: an MPI implementation reducing the overall memory consumption. In: Proceedings of EuroPVM/MPI 2009, Springer, pp 94–103
9. Shirley D (2000) Enhancing MPI applications through selective use of shared memory on SMPs. In: Proceedings of the 1st SIAM conference on CSE
10. Los Alamos National Laboratory (2001) Unified parallel software users' guide and reference manual. http://public.lanl.gov/ups/Doc_Directory/UserGuide/UserGuide.pdf
11. Hoefler T, Dinan J, Buntinas D, Balaji P, Barrett B, Brightwell R, Gropp W, Kale V, Thakur R (2012) Leveraging MPIs one-sided communication interface for shared-memory programming. In: Träff J, Benkner S, Dongarra J (eds) Recent advances in the message passing interface. vol 7490, pp 132–141
12. Taft JR (2001) Achieving 60 GFLOP/s on the production CFD code OVERFLOW-MLP. Parallel Comput 27(4):521–536
13. Boehm HJ (2005) Threads cannot be implemented as a library. In: Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation. PLDI '05, New York, NY, USA, ACM pp 261–268
14. Boehm HJ, Adve SV (2012) You do not know jack about shared variables or memory models. Commun. ACM 55(2):48–54
15. Aho AV, Sethi R, Ullman JD (1986) Compilers: principles, techniques, and tools. Addison-Wesley Longman Publishing Co. Inc., Boston
16. Manson J, Pugh W, Adve SV (2005) The Java memory model. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on principles of programming languages. POPL '05, New York, ACM pp 378–391
17. Boehm HJ, Adve SV (2008) Foundations of the C++ concurrency memory model. SIGPLAN Not 43(6):68–78
18. Lee EA (2006) The problem with threads. Computer 39(5):33–42
19. Heroux MA, Brightwell R, Wolf MM (2011) Bi-modal MPI and MPI+threads computing on scalable multicore systems. IJHPCA (Submitted)
20. Sandia National Laboratories (2012) Mantevo project. http://www.mantevo.org
21. Saad Y (2003) Iterative methods for sparse linear systems. Society for Industrial and Applied Mathematics
22. Saltz JH (1990) Aggregation methods for solving sparse triangular systems on multiprocessors. SIAM J Sci Stat Comput 11(1):123–144
23. Wolf MM, Heroux MA, Boman EG (2010) Factors impacting performance of multithreaded sparse triangular solve. Technical report SAND2010-0331 presented at VECPAR'10
24. Esler KP, Kim J, Ceperley DM, Purwanto W, Walter EJ, Krakauer H, Zhang S, Kent PRC, Hennig RG, Umrigar C, Bajdich M, Koloren J, Mitas L, Srinivasan A (2008) Quantum monte carlo algorithms for electronic structure at the petascale; the endstation project. J Phys 125(1):012057
25. Wagner LK, Bajdich M, Mitas L (2009) Qwalk: a quantum monte carlo program for electronic structure. J Comput Phys 228(9):3390–3404
26. Esler KP Einspline libaray. Online: http://einspline.svn.sourceforge.net/
27. Niu Q, Dinan J, Tirukkovalur S, Mitas L, Wagner L, Sadayappan P (2012) A global address space approach to automated data management for parallel quantum Monte Carlo applications. In: Proceedings 19th international conference on high performance computing. HiPC'12
28. Smith L, Kent P (2000) Development and performance of a mixed OpenMP/MPI quantum Monte Carlo code. Concurr Pract Exp 12(12):1121–1129
29. Esler KP, Kim J, Ceperley DM, Shulenburger L (2012) Accelerating quantum Monte Carlo simulations of real materials on GPU clusters. Comput Sci Eng 14(1):40–51