Towards a matrix-oriented strided interface in OpenSHMEM

Jeff R. Hammond
Extreme Scalability Group and Parallel Computing Lab
Intel Corporation
jeff_hammond@acm.org

ABSTRACT
New communication routines are proposed for OpenSHMEM to allow the efficient implementation of distributed matrix computations.

1. INTRODUCTION
Matrix computations are fundamental to many domains of science and are therefore ubiquitous in high-performance computing. Distributed-matrix algorithms frequently use a two-dimensional decomposition (e.g. the block-cyclic distribution used in ScalAPACK [4]), wherein both the row and column index spaces are tiled and wrapped cyclically. Global Arrays [10, 11] uses a two-dimensional distribution by default and supports user-defined blocking in both dimensions. For this reason, multidimensional subarray operations are an essential and highly optimized feature in ARMCI [9, 13]. Co-array Fortran [14], which is now standardized in ISO Fortran 2008, also supports multidimensional arrays distributed in each of the dimensions.

Vendor SHMEM implementations and the OpenSHMEM 1.1 specification support two types of data layouts: contiguous and indexed. The indexed PUT and GET routines take a source and target stride but only support striding in a single data element. For example, one can communicate the even elements of a vector of doubles using a stride of two, but there is no way – at least within a single function call – to communicate four doubles in a row at a stride of eight doubles.

The inability to send multiple elements with a stride means that communicating with submatrices, as is often required for distributed matrix computations, cannot be done efficiently. For example, if one wants to send an \( M \) by \( N \) submatrix corresponding to row-major layout\(^1\), either \( M \) calls to a contiguous operation acting on \( N \) elements or \( N \) calls to an indexed operation acting on \( M \) elements is required. The overhead of multiple function calls is exacerbated by the absence of true nonblocking operations in OpenSHMEM, although these are an active topic for inclusion in a future version of the standard. However, even if nonblocking operations were available, preventing the user from exposing the semantics of their application, i.e. that they are communicating a submatrix, prevents optimizations at the network level for noncontiguous data that may exist due to the support of these features in MPI [8], ARMCI [13], and GASNet [2], among others.

In the case of a tall, skinny submatrix, the use of indexed operations reduces the number of function calls but may require touching cache lines more times than would otherwise be required, which is undesirable on all modern memory systems. For example, if a submatrix of dimension 4 by \( n \) doubles of a matrix of \( m \) by \( n \) (\( m \geq 8 \)) is communicated via indexed puts, each cache line will be touched four times, once each to load the first, second, third and fourth elements out of eight (we assume a 64-byte cache line, but observes the same behavior for a 128-byte cache line if \( m \geq 16 \)). Of course, this assumes CPU-like access to memory – a network that can address DRAM directly at word granularity may avoid such issues.

2. PROPOSED FEATURES
We propose to add array-oriented communication operations for PUT and GET, generalizing the existing contiguous and indexed operations for all types. The syntax of array-put (APUT) is given below; the corresponding GET operation can be inferred.

\[
\text{void shmemx}_{\langle T \rangle}\_aput(T *\ dst, const T *\ src,\\ pptrdiff_t\ dst\_str,\\ pptrdiff_t\ src\_str,\\ size_t\ blockelems,\\ size_t\ blockcount,\\ int\ pe);\\]

Just as for the indexed PUT (IPUT) operation, there are arguments for the source and destination strides. The nelems argument is replaced with two arguments in order to capture the fact that these operations moved blocks of data rather than single elements. The mapping from APUT to IPUT is trivial; one merely sets \( \text{blockelems} = 1 \) and \( \text{blockcount} = \text{nelems} \).

In the event that the user calls APUT such that there are overlapping writes to the same memory location, the result

\(^1\)We will use this convention throughout the paper. Unless said otherwise, it is implied.
is undefined. Implementations may be able to write atomically, in which case one of these writes will succeed and the others will fail, but this behavior cannot be assumed, as other implementations may not be able to write atomically. Other semantics should be inferred from the mapping of APUT to IPUT noted above.

It is worth asking whether it is worthwhile to generalize the APUT operation for dimensions higher than two to support tensor operations (for some applications, see [7] and [15]). There are two arguments against this. First, operations on subarrays of dimension greater than two can be expressed in terms of a single APUT operation by combining the strides; for example, a three-dimensional subarray operation can be cast in terms of a two-dimension subarray computation if the stride over $x$ and $y$ are multiplied together (here we assume $z$ is the contiguous dimension that is captured by \textit{blockelems}). Regardless of the number of dimensions associated with the strides, the key efficiency gain with APUT is accomplished by operating on blocks of contiguous data rather than single elements, as is the case for IPUT. Second, the myriad of applications involving tensor operations include many cases where cartesian subarrays are not useful. For example, in the domain of quantum chemistry, most tensors have permutation (anti-)symmetry and thus cannot make use of operations designed for non-symmetric subarrays. Such is the complexity of tensor data in the NWChem [3] Tensor Contraction Engine [6] that block-sparse and permutation-(anti)symmetric tensors are mapped to \textit{one-dimensional} global arrays with an application-defined hashing scheme.

3. PROTOTYPE IMPLEMENTATION
We describe an implementation of the proposed additions to OpenSHMEM using the OpenSHMEM 1.1 functionality and an optimized one using the Cray® DMAPP interface [1], which closely resembles SHMEM but has a number of additional features, including nonblocking one-sided calls corresponding to the existing blocking one-sided operations in OpenSHMEM. These come in both implicit and explicit form, which is to say, one set of nonblocking operations takes a handle that can be tested or waited upon in a similar manner to MPI nonblocking Send and Receive, whereas the other (implicit) is associated with bulk completion via global synchronization (\textit{gsync}) operations.

Because DMAPP does not support an unlimited number of outstanding nonblocking operations of the implicit type, we must periodically wait on their completion. An explicit nonblocking implementation requires the creation of numerous (the minimum of $blkzs$ and $blckts$) handles and their individual testing – DMAPP does not provide a bulk handle completion function akin to \textit{MPI_Waitall} and friends) – hence they are not a good match for APUT.

4. CONCLUSION AND FUTURE WORK
In this short paper, we have described a new set of features – proposed for inclusion in a future version of OpenSHMEM – that would allow the efficient expression of distributed matrix algorithms involving two-dimensional blocked data distributions.
References

The author thanks Jim Dinan for feedback.

Acknowledgment

The author thanks Jim Dinan for feedback.

5. REFERENCES


void shmem_double_aput (double * dest,
const double * src,
ptrdiff_t dstr,
ptrdiff_t sstr,
size_t blksz,
size_t blckt, int pe)
{
    int maxnbi = DMAPP_DEF_OUTSTANDING_NONBLOCKING/2;
    double *dtmp = dest;
    const double *stmp = src;
    if (blksz<blckt) /* may require tuning */ {
        for (size_t i =0; i<blksz; i++) {
            dmapp_put_nb (dtmp, _sheap, pe, (double*)stmp,
                          dstr, sstr, blksz, DMAPP_QW);
            if (i%maxnbi==0) dmapp_gsync_wait();
            dtmp ++; stmp ++;
        }
    } else {
        for (size_t i =0; i<blckt; i++) {
            dmapp_put_nbi (dtmp, _sheap, pe, (double*)stmp,
                           blkct, DMAPP_QW);
            if (i%maxnbi==0) dmapp_gsync_wait();
            dtmp += blksz; stmp += sstr;
        }
    }
}

Figure 3: Implementation of APUT in terms of non-blocking PUT and IPUT from the Cray® DMAPP interface.