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.

```
void shmemx_<T>_aput(T * dst, const T * src,
                     ptrdiff_t dst_str,
                     ptrdiff_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 $blockelems = 1$ and $blockcount = nelems$.

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.

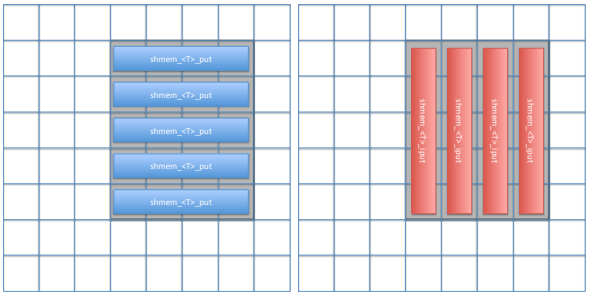

Figure 1: Pictorial representation of the PUT (left) and IPUT (right) operations acting on a 4-by-5 submatrix of an 8x8 matrix. The blue bars represent *contiguous* put operations while the red bars represent *indexed* (element-wise strided) put operations.

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 *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 *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]. For clarity in the code, we show only the case of type double, as the extension to other types is obvious.

3.1 Reference Implementation

To illustrate that APUT is inherently compatible with OpenSHMEM in its current form, it is useful to map from the proposed features to existing ones. One reason to show this implementation is to prove that there is no special burden on implementers to change the internal design of their OpenSHMEM runtime to support APUT. The second reason is that the obvious performance deficiences in the reference implementation motivate the addition of APUT to OpenSHMEM to allow implementers to optimize for it.

```
void shmemx_double_aput(double * dest,
                        const double * src,
                        ptrdiff_t dstr,
                        ptrdiff_t sstr,
                        size_t blksz,
                        size_t blkct, int pe)
{
  double       *dtmp = dest;
  const double *stmp = src;
  if (blksz<blkct) /* may require tuning */ {
    for (size_t i=0; i<blksz; i++) {
      shmem_double_iput(dtmp, stmp, dstr, sstr,
                        blkct, pe);
      dtmp++; stmp++;
    }
  } else {
    for (size_t i=0; i<blkct; i++) {
      shmem_double_put(dtmp, stmp, blksz, pe);
      dtmp += dstr; stmp += sstr;
    }
  }
}
```

Figure 2: Implementation of APUT in terms of PUT and IPUT.

The implementation shown in Figure 2 attempts to minimize the number of SHMEM calls, which may not be the best metric. The overhead of moving data via IPUT may be higher than PUT, in which case a different threshhold is appropriate. In any case, this detail is not significant here; we merely aspire to illustrate the semantics of this function by mapping to ones with which the reader is surely familiar.

3.2 Optimized Implementation

The implementation shown in Figure 3 leverages 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 (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 *blksz* and *blkct*) handles and their individual testing – DMAPP does not provide a bulk handle completion function akin to `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.

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

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

We hope that inclusion of these features in OpenSHMEM spurs the development of OpenSHMEM-based dense linear algebra libraries similar to Global Arrays and otherwise enables both performance and productivity for this class of computations.

In the future, we will explore the mapping of the Global Arrays programming model to OpenSHMEM both as end in its own right and as a vehicle for understanding the utility of APUT in important scientific applications (e.g. NWChem). An obvious issue with this task is the lack of remote accumulate in SHMEM APIs. However, it has already been discussed how to [12] how one can implement the accumulate at the initiator using a lock-get-accumulate-put-unlock implementation, which can be implemented using `shmem_set_lock`, `shmem_clear_lock`, and the appropriate put and get operations for the datatype. An implementation of Global Arrays using OpenSHMEM has the potential to be implemented without an asynchronous agent (i.e. polling thread), which obviates many of the issues associated with such a design [5].

Acknowledgment

The author thanks Jim Dinan for feedback.

5. REFERENCES

[1] Using the GNI and DMAPP APIs. Technical Report S-2446-52, Cray, 2014.

[2] D. Bonachea. Proposal for extending the UPC memory copy library functions and supporting extensions to GASNet. Technical Report LBNL-56495, Lawrence Berkeley National Lab, 2004.

[3] E. J. Bylaska et. al. NWChem, a computational chemistry package for parallel computers, version 6.1.1, 2012.

[4] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. In *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*, pages 95–106. Springer, 1996.

[5] J. R. Hammond, J. Dinan, P. Balaji, I. Kabadshow, S. Potluri, and V. Tipparaju. OSPRI: An optimized one-sided communication runtime for leadership-class machines. In *The 6th Conference on Partitioned Global Address Space Programming Models*, Santa Barbara, CA, 04/2011 2011.

[6] S. Hirata. Tensor Contraction Engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *J. Phys. Chem. A*, 107:9887–9897, 2003.

[7] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[8] MPI Forum. MPI: A message-passing interface standard. Version 3.0., Nov. 2012.

[9] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Parallel and Distributed Processing*, pages 533–546, London, UK, 1999. Springer-Verlag.

[10] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: a portable "shared-memory" programming model for distributed memory computers. In *Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 340–349, New York, 1994. ACM.

[11] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A non-uniform-memory-access programming model for high-performance computers. *The Journal of Supercomputing*, 10:10–197, 1996.

[12] J. Nieplocha, V. Tipparaju, and E. Apra. An evaluation of two implementation strategies for optimizing one-sided atomic reduction. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 7–pp. IEEE, 2005.

[13] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. Panda. High performance remote memory access communications: The ARMCI approach. *International Journal of High Performance Computing and Applications*, 20(2), 2006.

[14] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998.

[15] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel. Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 813–824, Washington, DC, USA, 2013. IEEE Computer Society.