A Global Address Space Framework for Irregular Applications

Kentaro Hara
The University of Tokyo
haraken@logos.ic.i.u-tokyo.ac.jp

Kenjiro Taura
The University of Tokyo
tau@logos.ic.i.u-tokyo.ac.jp

ABSTRACT

Practical parallel scientific applications with domain decompositions, such as finite element methods, require *irregular* domain decompositions of complicated-shaped objects. However, existing PGAS frameworks, such as Global Arrays and XcalableMP, have supported the productive description of exchanging ghost points only for regular domain decompositions. With these backgrounds, we propose, implement and evaluate *Distributed Memory Interface* (*DMI*), a global address space framework for irregular applications. DMI provides highly productive APIs called *read-write-set* for irregular domain decompositions and complicated orderings in practical scientific applications.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; G.1.8 [**Numerical Analysis**]: Partial Differential Equations

General Terms

Algorithms, Languages, Performance

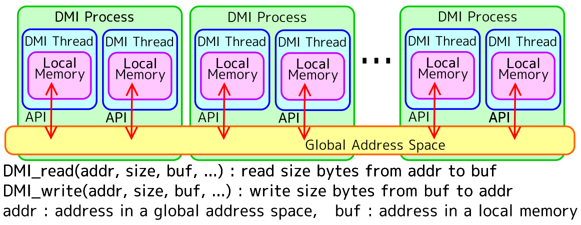
Keywords

Global address space, Finite element method, Productivity

1. INTRODUCTION

Existing PGAS frameworks such as UPC and Global Arrays[5] have enabled the highly productive development of a lot of high performance applications. However, since these PGAS frameworks are designed mainly for applications with regular communication patterns, it is difficult to improve the performance of *practical* applications with *irregular* communication patterns or to describe these applications productively. In particular, in practical parallel scientific applications with domain decompositions such as finite element methods (FEM) and multigrid methods, complicated-shaped objects are decomposed into irregular domains using some graph partitioning algorithm such as METIS. In order to execute these applications on a distributed memory system, updating the element values in each domain requires the values of boundary elements in its neighboring domains (these boundary elements are referred to as *ghost points*[5, 3]), and this exchange of values of ghost points is the primary hurdle in describing these applications. Hence frameworks are required with which a programmer can productively describe the complicated exchange of values of ghost points in scientific applications with irregular domain decompositions. However, most PGAS frameworks such as XcalableMP[3] and Global Arrays have supported only regular domain decompositions, which decompose an n-dimensional rectangular parallelepiped object into m-dimensional rectangular parallelepiped domains. Although Titanium[6, 2] is also a PGAS framework which facilitates highly productive description of parallel scientific applications with domain decompositions, the papers[6, 2] evaluate its performance using multigrid methods and adaptive mesh refinement methods based on only regular domain decompositions. Based on these observations, we propose, implement and evaluate *Distributed Memory Interface* (*DMI*), a global address space framework for irregular applications. DMI introduces *read-write-set*, APIs for the highly productive description of exchanging values of ghost points for not only regular but also irregular domain decompositions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$10.00.



DMI_read(addr, size, buf, ...) : read size bytes from addr to buf
DMI_write(addr, size, buf, ...) : write size bytes from buf to addr
addr : address in a global address space, buf : address in a local memory

Figure 1: System components of DMI.

2. BASIC SYSTEM DESIGNS

DMI is a multi-threaded global address space framework implemented as a shared library for C. DMI distinguishes between a *global address space* and a *local memory* as shown in Fig.1. A local memory is a normal shared memory allocated/deallocated by `malloc()`/`free()` and is read/written normally. In contrast, a global address space is allocated/deallocated by `DMI_mmap()`/`DMI_munmap()` and is read/written by `DMI_read()`/`DMI_write()`. Specifically, a programmer can access to the global address space by calling `DMI_read (int64_t addr,int64_t size,void *buf)`/ `DMI_write (int64_t addr,int64_t size,void *buf)`, which reads/

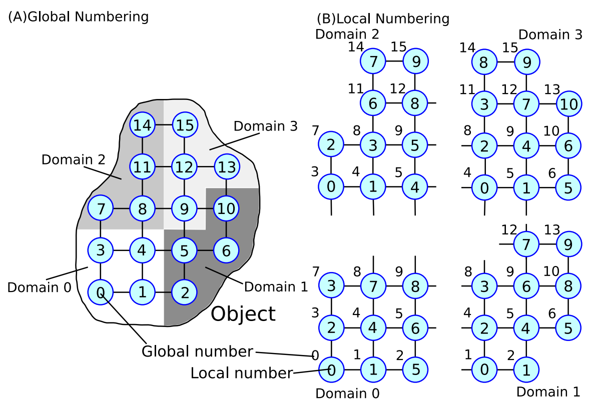


Figure 2: An example FEM problem.

writes `size` bytes from/to the global address space `addr` to/from a local memory `buf`. Like most page-based distributed shared memories (DSM), DMI partitions each global address space into *pages* and guarantees the sequential consistency of `DMI_read()` and `DMI_write()`, the address range [`addr`, `addr+size`) of which is within one page. Here the page size in each global address space can be specified arbitrarily and explicitly by the programmer as with region-based DSMs. Specifically, by calling `DMI_mmap(int64_t page_size,int64_t page_num,)`, the programmer can allocate a global address space with `page_num` pages of `page_size` bytes in size. Furthermore, DMI serves as a remote swap system, supports dynamic joining/leaving of nodes during the execution of a parallel application, provides highly flexible and explicit APIs for improving data locality, but we leave these details for another paper[4].

3. DESIGNS FOR IRREGULAR APPLICATIONS

3.1 A Practical FEM

In practical scientific applications with domain decompositions, complicated-shaped objects are decomposed into irregular domains using some graph partitioning algorithm such as METIS. In addition, in order to improve the convergence of iterative calculations in hard-to-converge problems, the elements in each domain are often ordered based on ordering methods such as RCM ordering.

As an example problem, consider a 2-dimensional FEM with 8 square-shaped elements as shown in Fig.2(A). There are 16 node points [1] numbered as shown in Fig.2(A). These numbers are referred to as *global numbers*. Each point has connectivity to its neighboring 8 points and to the point itself. Assume that there are 4 processors and that we decompose these 16 points into 4 domains. For each domain i, we refer to the points within the domain i as *interior points*, and we refer to the points within other domains but have connectivity to at least one of the interior points of the domain i as *ghost points*. For example, the interior points of domain 1 are 2,5,6 and 10, and the ghost points of domain 1 are 1,4,8,9,12 and 13. At each iterative calculation, each processor must store the values of interior points and ghost points to its local memory according to some ordering. Note that, for solving hard-to-converge problems, it is not sufficient to simply store these values in order of interior points and then ghost points. Although the best ordering depends on problems, here we assume the ordering to be as shown in Fig.2(B). We refer to the numbers obtained by this ordering as *local numbers*. For example, the point in domain 1 for which the local number is 2 has a global number of 4.

In essence, at each iterative calculation, each processor (1) obtains the values of the ghost points from its neighboring processors; (2) stores the values of the interior points and the ghost points to its local memory according to a suitable ordering; and (3) updates the values of the interior points using a given connectivity. Obviously, the most difficult task in describing this program is the exchange of values of ghost points. Describing the exchange of values of ghost points in a local-view programming framework, such as MPI or Co-array Fortran which does not provide a global address space, is a rather complicated task. This complexity in local-view programming is attributed to the fact that since connectivity is given by global numbers, a programmer must calculate the correspondence between the global numbers and the local numbers, and then specify (1) the values of the interior points to be sent and the processor to which these values should be sent, and (2) the number of values that each processor should receive and the processor from which they should be received, as well as location of the local memory of the processor that should store each received value. In contrast, the programmer can easily describe the exchange of values of ghost points in a global-view programming framework with a global address space, using connectivity information given by the global numbers. However, the performance of programs based on global-view programming is generally lower than that based on local-view programming. The primary reason is that since most global-view programming frameworks support only block-cyclic data distribution, these frameworks cannot support data distribution along irregular domains, such as that shown in Fig.2(A) unless very fine-grained data distribution is adopted, which also degrades the performance because of the data management overhead. In summary, local-view programming and global-view programming have a tradeoff between performance and productivity.

3.2 read-write-set

3.2.1 *Characteristics of Scientific Applications*

Based on these analyses, DMI introduces APIs called *read-write-set*, which manages point values efficiently in a manner similar to local-view programming but facilitates highly productive global-view programming. Before explaining the APIs of read-write-set, we summarize the general characteristics of scientific applications with domain decompositions.

An object O is composed of a finite number of points. The connectivity between the points in O is given. The object O is decomposed into n domains. Each domain i is composed of a set of points W_i, a set of interior points. If $i \neq j$, then $W_i \cap W_j = \emptyset$ holds. For each domain i, a set of points R_i is defined by connectivity as a set of points $x \in O$ such that there exists a point $x' \in W_i$ which has connectivity to the point x. Namely, R_i is a set of interior points and ghost points. Each processor i handles a domain i, that is, at each iterative calculation each processor i reads the values of the points in R_i and updates the values of the points in

[1]To distinguish a machine node from a node point in an FEM, we refer to the former as a *node* and refer to the latter as a *point*.

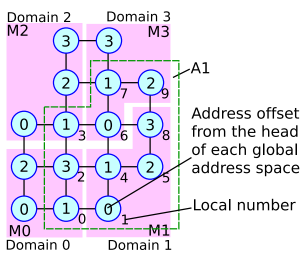


Figure 3: Global address spaces in read-write-set.

W_i. Both W_i and R_i are ordered sets because some ordering of the points in W_i and R_i must be defined. For example, $W_1 = \{2, 5, 6, 10\}$ and $R_1 = \{1, 2, 4, 8, 5, 6, 9, 12, 10, 13\}$ in Fig.2.

3.2.2 *Programming Interfaces*

Based on the above characteristics, read-write-set provides the following APIs which enable global-view programming of irregular applications. Fig.4 shows a pseudo code for solving an FEM problem by the CG method using read-write-set APIs.

First, a programmer calls rwset_decompose(O,i,W_i,R_i) for each domain i in an object O. This API defines the fact that the ordered set of the interior points of the domain i is W_i and that the ordered set of the interior points and the ghost points of the domain i is R_i. In other words, the programmer defines the interior points and the ghost points of each domain i by global numbers using this API. Second, the programmer calls rwset_alloc(O,i) for each domain i in O. This API returns a communication handle H_i to read/write values of the points in R_i and W_i efficiently. Note that rwset_decompose() for all domains must be called before any rwset_alloc() is called. In subsequent iterations, by calling rwset_write(H_i,*wbuf*), the programmer can store a value of the j-th ($0 \leq j < |W_i|$) point in W_i from a local memory *wbuf*[j] to a global address space. In addition, by calling rwset_read(H_i,*rbuf*), the programmer can load a value of the j-th ($0 \leq j < |R_i|$) point in R_i from the global address space to a local memory *rbuf*[j]. Thus, the programmer can load/store values of the points in R_i/W_i from/to a global address space to/from a local memory *rbuf*/*wbuf* through a communication handle H_i.

In essence, since one domain is assigned to one processor in most cases, most programs can be described as follows: (1) each processor i calls rwset_decompose() for a domain i; (2) all processors are synchronized; (3) each processor i calls rwset_alloc() for the domain i and obtains a communication handle H_i; (4) at each iteration each processor i reads/writes values of the points in R_i/W_i from/to the global address space through H_i using rwset_read()/rwset_write().

3.2.3 *Implementations*

Although read-write-set facilitates highly productive global-view programming as mentioned above, read-write-set manages point values efficiently in a manner similar to local-view programming by internally transforming global numbers specified by the programmer into local numbers, by which read-write-set manages point values.

First, when rwset_decompose(O,i,W_i,R_i) is called, DMI calls `DMI_mmap()` and allocates a global address space M_i with one page, the size of which is $|W_i| \times$ (each point size) bytes to store values of the points in W_i. The address in the global address space M_i to which each value of the points in

```
void fem_code_of_processor_i(int i /* a processor i */
     , int n /* the number of processors */
     , int64_t o_addr /* an object to be decomposed */ ) {
  int rn, wn, iter, u, v;
  int *ri, *rbuf, *wi, *wbuf;
  matrix_t *mat;
  DMI_rwset_t rwset;

  mat = a matrix of a domain i in CRS format;
  wn = the size of W_i;
  wi = malloc(sizeof(int) * wn);
  wbuf = malloc(sizeof(double) * wn);
  /* define W_i as an ordered set of global numbers */
  if(i == 0) wi[0..wn-1] = {0,1,3,4};
  else if(i == 1) wi[0..wn-1] = {2,5,6,10};
  else if(i == 2) wi[0..wn-1] = {7,8,11,14};
  else if(i == 3) wi[0..wn-1] = {9,12,13,15};
  rn = the size of R_i;
  ri = malloc(sizeof(int) * rn);
  rbuf = malloc(sizeof(double) * rn);
  /* define R_i as an ordered set of global numbers */
  if(i == 0) ri[0..rn-1] = {0,1,3,7,4,2,5,8,9};
  else if(i == 1) ri[0..rn-1] = {1,2,4,8,5,6,9,12,10,13};
  else if(i == 2) ri[0..rn-1] = {3,4,7,8,5,9,11,14,12,15};
  else if(i == 3) ri[0..rn-1] = {4,5,8,11,9,6,10,12,14,15,13};
  for(u = 0; u < wn; u++) /* define initial values of W_i */
    wbuf[u] = 0.00;
  DMI_rwset_decompose(o_addr, i, wi, wn, ri, rn);
  barrier between n processors;
  DMI_rwset_alloc(&rwset, o_addr, i);

  for(iter = 0; /* until convergence */; iter++) {
    ...; /* the CG method */
    barrier between n processors;
    DMI_rwset_write(&rwset, wbuf);
    barrier between n processors;
    DMI_rwset_read(&rwset, rbuf);
    for(u = 0; u < wn; u++) {
      wbuf[u] = 0;
      for(v = mat->row[u]; v < mat->row[u + 1]; v++) {
        wbuf[u] += mat->val[v] * rbuf[mat->col[v]];
      }
    }
    ...; /* the CG method */
  }
  DMI_rwset_free(&rwset);
}
```

Figure 4: A pseudo code for solving an FEM problem by the CG method using read-write-set.

W_i is stored depends on the order of the points in W_i. For example, if the programmer specifies $W_0 = \{0, 1, 3, 4\}, W_1 = \{2, 5, 6, 10\}, W_2 = \{7, 8, 11, 14\}, W_3 = \{9, 12, 13, 15\}$, then four global address spaces M_0, M_1, M_2, M_3 are allocated in the address order as shown in Fig.3. Each R_i, W_i and M_i are preserved in O. Second, when rwset_alloc(O,i) is called, for each point x in R_i, DMI calculates which address in global address spaces $M_0, \cdots, M_{n-1}$ should be accessed in order to obtain a value of the point x. In other words, for each point x in R_i, DMI calculates the location of the point x in the global address spaces. As a result, an ordered set of addresses A_i is constructed, the j-th address of which indicates the address at which a value of the j-th point in R_i is stored. Here A_i, W_i and M_i are preserved in a communication handle H_i, which is a return value of this API. Third, when rwset_write(H_i,*wbuf*) is called at each iteration, DMI calls `DMI_write(`the head address of M_i`,`$|W_i|\times$(each point size)`,`*wbuf*`)`. Note that the order of the points in W_i indicates the address in a global address space M_i to which each value in *wbuf* is stored. In other words, the programmer can specify the correspondence between *wbuf* and points in the global address space by the order of the points in W_i. Incidentally, since DMI can locate M_i to be local to the processor which issues the first `DMI_write()` for M_i using APIs for improving data locality explicitly,

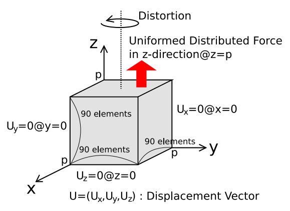


Figure 5: Stress analysis using an FEM.

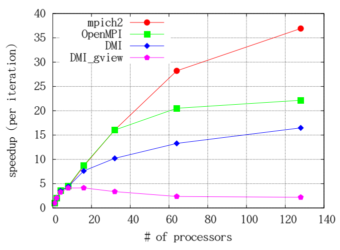


Figure 6: The scalability of DMI and MPI for a stress analysis with an FEM.

the second or later `DMI_write()` for M_i is completed locally when this API is called several times. Forth, when rwset_read(H_i,*rbuf*) is called at each iteration, DMI calls `DMI_read(`the j-th address in A_i`,`each point size`,`*rbuf*[j]`)` for each j ($0 \leq j < |A_i| = |R_i|$). Note that the order of the points in R_i indicates from which address in the global address spaces each value in *rbuf* is loaded. In other words, the programmer can specify the correspondence between *rbuf* and points in the global address spaces by the order of the points in R_i. Here calling `DMI_read()` for each address in A_i requires too much overhead. Not only FEMs, most irregular applications issue such discrete memory accesses. Hence DMI provides a mechanism to explicitly group multiple put/get operations for discrete global address space regions. For example, when the programmer issues discrete get operations for a set of addresses A_1 as shown in Fig.3, DMI groups these $|A_1|$ discrete regions into 4 regions so that each region belongs to one processor, and then sends 4 get requests to 4 processors in parallel (one of these get requests is handled locally). To put it more general, when the programmer issues discrete put/get operations for y discrete global address space regions of arbitrary size across x pages, DMI groups these y regions into x regions so that each region belongs to one page, and then sends x put/get requests (regardless of y) to the owner of each page in parallel. This read-write-set can be regarded as a form of inspector/executor[6].

4. PERFORMANCE EVALUATION

The experimental platform is a cluster composed of 16 nodes interconnected by 1Gbit Ethernet. Each node contains two Intel Xeon E5410 2.33GHz (4 cores) CPUs, 32 GB memory, running Linux OS with the 2.6.18-6-amd64 kernel. We used gcc 4.1.2 with an -O3 option for DMI, and OpenMPI 1.3.3 and mpich2-1.1.1p1 with an -O3 option for MPI.

We analyzed the stress of a 3-dimensional cubic object using the FEM with 90^3 cubic elements and the force and boundary conditions shown in Fig.5. This is a practical and very hard-to-converge problem used in the parallel programming contest on cluster systems in Japan[1] and powerful preconditioning and complicated orderings are essential. We compared the performance of the champion MPI program of the contest with a DMI program using read-write-set with the same algorithm as the MPI program.

In the result, we confirmed that we can easily describe this complicated program using read-write-set in DMI although we must calculate the complicated correspondence between global numbers and local numbers in MPI. Fig.6 compares the scalability of DMI with that of MPI. The line labeled DMI_gview in Fig.6 shows the result obtained when programming was performed using one simple global address space of 4KB pages as mentioned in section 3.1, not using read-write-set. The communication time and computation time per iteration were 0.2603 seconds and 0.2872 seconds in mpich2, 0.6369 seconds and 0.2798 seconds in OpenMPI, and 0.9890 seconds and 0.2666 seconds in DMI, respectively. Here the communication time includes the time to wait for data. This result indicates that read-write-set can achieve much larger scalability than simple global-view programming but that the scalability of DMI is inferior to that of MPI because of the large communication time of DMI. Another analysis showed that this large communication time of DMI is mainly due to the slowness of a point-to-point synchronization of DMI in the matrix-vector multiplication in each iteration. Thus optimization of the point-to-point synchronization is crucial.

5. CONCLUSIONS

In this paper we proposed, implemented and evaluated Distributed Memory Interface (DMI), a global address space framework for irregular applications. In particular, we proposed read-write-set, APIs which enable the highly productive description of exchanging values of ghost points in practical scientific applications with irregular domain decompositions and complicated orderings.

6. ACKNOWLEDGMENTS

We wish to thank the committee of the parallel programming contest on cluster systems, including Kengo Nakajima, for providing us with datasets of the FEM used in section 4. We also wish to thank to Jun Nakashima for providing us with the champion MPI program of the contest.

7. REFERENCES

[1] The Second Parallel Programming Contest on Cluster Systems.
https://www2.cc.u-tokyo.ac.jp/procon2009-2/.

[2] Kaushik Datta1, Dan Bonachea1, and Katherine Yelick1. Titanium Performance and Potential: An NPB Experimental Study. *18th International Workshop on Languages and Compilers for Parallel Computing*, Vol. 4339, pp. 200–214, 2006.

[3] XcalableMP Specification Working Group. XcalableMP Application Program Interface Version 1. Technical report, Center for Computational Sciences, University of Tsukuba, Nov 2009.

[4] Kentaro Hara, Kenjiro Taura, and Takashi Chikayama. DMI: A Large Distributed Shared Memory Interface Supporting Dynamically Joining/Leaving Computational Resources. *IPSJ Transactions on Programming*, Vol. 3, No. 1, pp. 1–40, Mar 2010.

[5] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edo Apra. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *International Journal of High Performance Computing Applications*, Vol. 20, No. 2, pp. 203–231, 2006.

[6] Jimmy Su, Tong Wen, and Katherine Yelick. Compiler and Runtime Support for Scaling Adaptive Mesh Refinement Computations in Titanium. Technical report, Electrical Engineering and Computer Sciences University of California at Berkeley, Jun 2006.