

DBuilder: A Parallel Data Management Toolkit for Scientific Applications

Robert M. Hunter

U.S. Army Engineer Research and Development Center
Major Shared Resource Center
3909 Halls Ferry Road
Vicksburg, MS 39180-6199
Ph: 601-634-377 Fax: 601-634-2324
Robert.M.Hunter@erdc.usace.army.mil

Jing-Ru C. Cheng

U.S. Army Engineer Research and Development Center
Major Shared Resource Center
3909 Halls Ferry Road
Vicksburg, MS 39180-6199
Ph: 601-634-4052 Fax: 601-634-2324
ruth.c.cheng@erdc.usace.army.mil

Presenter: Robert M. Hunter
Conference: PDPTA'05

Abstract

Code migration from serial to parallel platforms using a distributed-memory model involves a number of changes in the serial code. The most significant change is the implementation of a message passing interface (MPI) scheme for domain partitioning, data coherence among processors, and data communication for spatially distinct multicomponent applications. The DBuilder software has been developed as a toolkit for application developers to leverage code migration efforts. This paper details the DBuilder software design, the coordination of objects when partitioning the domain with two existing objects, e.g., vertex and element, the synchronization algorithms, and the coupler setup for multicomponent applications with partially overlapped distinct dimensional domains. Furthermore, the integration of legacy parallel linear solver software is also discussed.

Keywords: parallel algorithms, multidomain, scientific computing, application programming interface, software tool

I. Introduction

A majority of scientific applications require a computational mesh, which is a discretization form of the spatial domain. A variety of data can be associated with the mesh, represented by sets of vertices, edges, or elements. These data can sit on vertices or elements of the mesh. When parallelism is employed, the key is to partition the mesh evenly to processors, maintain data coherence among processors, and implement efficient parallel algorithms to reduce communication overhead. DBuilder has been developed to provide a simple Application Programming Interface (API) for users to sidestep the learning curve of message passing, graph theory, and parallel algorithms. DBuilder is written in C and provides a FORTRAN interface. Facets of parallel programming are implemented by utilizing tools and libraries such as ParMETIS [1] and the Message Passing Interface (MPI). A library similar to DBuilder called libMesh [2] provides a parallel framework for data management, yet it is built on C++ and can be difficult to utilize with legacy codes that are written in FORTRAN. Furthermore, libMesh requires that the global mesh be stored on each processor. DBuilder does not impose such a limitation, which allows for memory to scale as the number of processors is increased (Figure 1).

DBuilder provides support for domain partitioning, parallel data management, coupling coordination, and parallel solver interfacing. Although DBuilder provides higher level functionality such as domain-coupling and parallel solver interfacing, it was designed first and foremost to provide fundamental routines for users to manage shared data between vertices and/or elements of a distributed structured or unstructured mesh. DBuilder is not meant to be a replacement for libraries such as ZOLTAN [3] or DRAMA [4]. The main purpose of these libraries is for workload balancing and adaptive mesh refinement. However, DBuilder can complement such libraries. For example, DRAMA requires that the application provide information on the current distributed mesh and on calculation and communication requirements. DBuilder encapsulates this information for the users, so they need not create new data structures to interface with DRAMA.

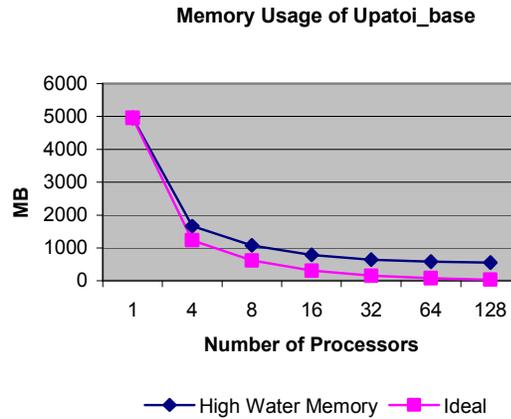


Figure 1. Memory scaling comparison between ideal and actual

II. DBuilder Framework

A mesh can be viewed as a graph $G=(V,E)$, which is a set of vertices (V) and edges (E). When a subgraph on each processor is given to DBuilder, DBuilder partitions the domain (G) to processors. This partition may be manipulated to provide a distribution that is low in communication or a computationally balanced workload. DBuilder can build a vertex domain with a distributed number of vertices, an element domain with a distributed number of elements, or a boundary element domain comprised of boundary elements in the element domain. A vertex domain is favored when the application is to solve vertex-based values such as the finite difference method, while an element domain (or the vertex domain of the dual graph of G) is for solving cell-centered values such as the finite volume method. For a finite element application, coordination between elements and vertices is required because the assembly procedure assembles each elemental matrix to a linear system, and the linear system is then solved for vertex-based values. DBuilder also provides a default rule for such coordination between vertices and elements. A callback approach is supported for users to specify their own rules.

Data migration among processors is encapsulated in DBuilder using MPI data types. DBuilder has the information for local and ghost vertices/elements alive throughout the entire application. This piece of information is mandatory for gather/scatter operations in the parallel paradigm. Two API functions are facilitated for the user to maintain coherent data structures among processors. MPI nonblocking functions are called in DBuilder for data synchronization.

Figure 2 shows a section of code implemented in WASH123D [5] to illustrate the use of the DBuilder API. In this case, both vertex and element domains are built for the finite element method to maintain a balanced number of vertices among processors. The first two function calls pass geometric information to `DBuild_Init` for building vertex and element domains, respectively. DBuilder registers their memory addresses only and does not keep a copy of the geometric information. Then, the function `DBuild_Get_file_part` can be called to retrieve the local count of data to be read on the processor. Two instances of `DBuild_Domains` are called to build the vertex domain followed by the element domain. The element domain's partition rule is based on the constructed vertex domain. These two domains have their own local entities and ghost entities, where the ghost entities are owned by other processors. Two steps are required to update data in the element array (i.e., element indices) that is not owned locally. These are as follows: (1) `DBuild_Set_Type` is called to set the data type for the element array, and (2) `DBuild_Global_update` brings in the ghost values to the element array. Based on the DBuilder constructed element domain, some vertices may need to be available as ghosts on a given processor, because more than one ghost layer may be necessary in the algorithm. `DBuild_Add_ghosts` is available to achieve this purpose. `DBuild_Set_Type` and `DBuild_Global_update` are then called to bring in the vertex coordinates for the vertex domain. `DBuild_Set_type` is then called again to instantiate a `DB_Type` for updating the data on ghost vertices. It should be noted that a `DB_Type` is not interchangeable between domains, even though the data may be of the same primitive type. Once the domains and `DB_Types` are built, the rest of parallelization is basically to place `DBuild_Global_update` routines at appropriate locations in the code.

```

/**/ Initialization for vtxDoamin and elementDomain ***/
ierr = DBuild_Init(num_global_vertices, num_ghost_layer, 0,
                  point2neighbor_list, proc_set, &vtx_neighbor_list,
                  &vtx_neighbor_list, &vtx_coord, total_bytes_of_each_vtx,
                  vtxDomain);
ierr = DBuild_Init(num_global_elements, num_ghost_layer,
                  num_neighbors_per_elm,
                  &num_entries_per_elm, proc_set, &element_array,
                  &elm_neighbor_list, NULL, 0, elementDomain);

/**/ Read mesh from a file --- partial file is read on each processor ***/
ierr = WashRead_geom3(fd, mesh); /** fill in coord and element arrays
                                   in mesh and then create neighbor list **/

/**/ Build domains ***/
ierr = DBuild_Domains(1, NULL, vtxDomain);
ierr = DBuild_Domains(3, vtxDomain, elementDomain);

/**/ update element_array for ghost elements ***/
ierr = DBuild_Set_type(num_entries_per_elm*sizeof(int), &dType,
                      elementDomain);
ierr = DBuild_Global_update(element_array, dType, elementDomain);

/**/ add ghost vertices based on elementDomain ***/
ierr = DBuild_Add_ghosts(element_array, num_local_vertices,
                        num_neighbors_per_elm, num_entries_per_elm, vtxDomain);

/**/ bring in the coordinates for ghost vertices ***/
ierr = DBuild_Set_type(num_dir*sizeof(double), &dType, vtxDomain);
ierr = DBuild_Global_update(coord, dType, vtxDomain);

/**/ build data types for data gathering or scattering ***/
ierr = DBuild_Set_type(sizeof(double), &mesh->doubleType, vtxDomain);

```

Figure 2. Code containing DBuilder APIs

III. DBuilder Coupler

Multiphysics applications on multidomains have become a large focus. Solving these applications requires that multidomain integration be executed to integrate two or more applications. The spatial relationship between computational domains can be adjacent, partially/fully overlapped, or distinct. DBuilder allows for the building of a `coupler` object to avoid the dependency between meshes when partitioning. Figure 3 depicts the concept of this `coupler` implementation. In the figure, all four processors participate in two-dimensional and three-dimensional simulations. Because dependency is not specified between domains when partitioning, a given processor in the 3-D domain exclusively owning a 2-D slice of the domain is not guaranteed. As shown in the figure, processor P3 owns a partition of the 2-D subdomain, which is a partition of the 3-D subdomain owned by processor P2 and P3. Likewise, the top of the 3-D subdomain that processor P1 owns is a partial subdomain partitioned to P0 and P1 in 2-D. A `coupler` formed with DBuilder will provide the message passing from 2- to 3-D and from 3- to 2-D as shown in the red and blue dashed arrow lines, respectively.

Figure 4 shows a section of code used to build a vertex coupler associated with the vertex domain and an element coupler associated with the element domain. The API function `DBuild_Coupler_init` is called twice to create these two couplers. The function `DBuild_Get_coupler_size` is called for a request of the size of a coupler. The argument `DB_D2TOD1` accounts for the size of the coupler in Domain 2 (D2)

when updating Domain 1(D1). From the code in Figure 4, DBuilder internally represents the 2-D domain as D1 and the 3-D domain as D2. This is determined by the ordering of the domains in the call to `DBuild_Coupler_init`. The function `DBuild_Coupler_update` is called to maintain consistent data on two different domains (e.g., D2 and D1) among processors. In this example, the data on D1 named `mtyp_temp`, which is a source, updates the destination on D2 shown in the first argument based on the coupler's element domain mentioned at the last argument. According to the third argument, it is known that each entry is composed of two integers.

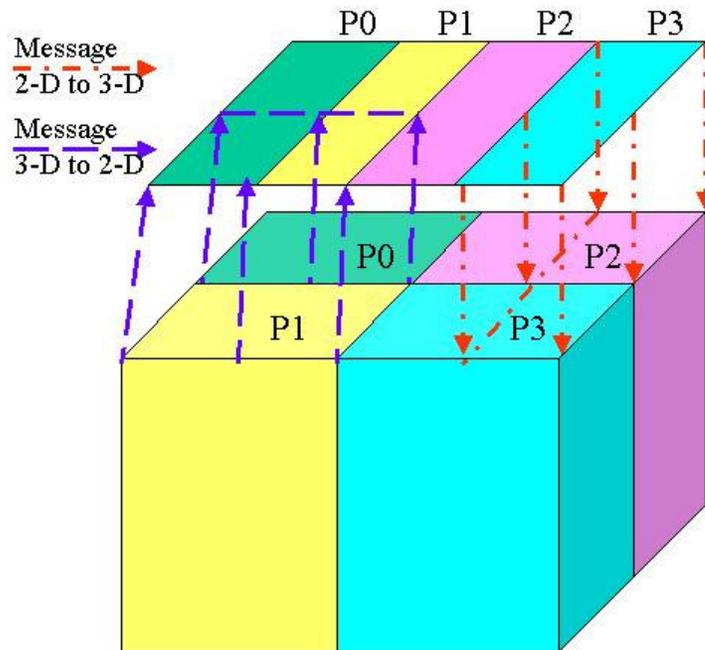


Figure 3. Concept of the coupler implementation

```

/** initialization and creation of coupler's vtx domain and element
    domain */
ierr = DBuild_Coupler_init(&mesh2->vtxDomain,&mesh3->vtxDomain,
    coupler->npxz, mesh2->vtxDomain.numberLocalElements,
    &coupler->vtx_coupler);
ierr = DBuild_Coupler_init(&mesh2->elementDomain,&mesh3->elementDomain,
    coupler->mtopxz,mesh2->elementDomain.numberLocalElements,
    &coupler->elm_coupler);

/** get the size of coupler's element domain */
ierr = DBuild_Get_coupler_size(&couplerSize,&coupler->elm_coupler,
    DB_D2TOD1);

/** update D1's vector to D2's vector on the element domain */
ierr = DBuild_Coupler_update(coupler->mtypxz, mtyp_temp, 2*sizeof(int),
    DB_D1TOD2,&coupler->elm_coupler);

```

Figure 4. Code building coupler using DBuilder

IV. DBuilder and Linear Solvers

DBuilder has an interface to linear solvers such as BlockSolve95 [6] and pARMS [7]. As shown in Figure 5, only two functions are required to use the parallel linear solver BlockSolve95. First, `DBuild_Solver_reset_co` passes the matrix and associated geometric domain to DBuilder. Second, `DBuild_Solver_solve` along with the right-hand side vector, initial guess, and the associated mesh domain is passed to DBuilder to solve the linear system, with the result stored in the second argument and the residual information at the third argument.

Using the DBuilder interface, the parallel solver can be changed from run to run though environment variables. Figure 6 shows the speedup of WASH123D when the native solver of the code was replaced using BlockSolve95. With minimal effort, a speedup of 1.5 to 2.5 was achieved for various processor counts.

```
ierr = DBuild_Solver_reset_co(matrix, &mesh3->vtxDomain);
iter = DBuild_Solver_solve(rhs, x, &residual, &mesh3->vtxDomain);
```

Figure 5. Code call DBuilder API to interface BlockSolve95

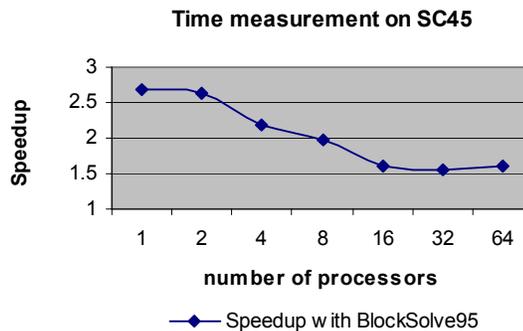


Figure 6. Speedup with BlockSolve95

V. Conclusion and Future Work

DBuilder provides a parallel software toolkit to implement parallelism in application codes. Experimental results show that the overhead for building domains can be considered insignificant. However, the time spent in communication to maintain data coherence becomes dominant as the number of processors increases (Figure 7). This may occur if the problem size is too small for a large number of processors, or the parallel solver is not efficient, i.e., too much communication time spent in the solver. Thus, more sophisticated parallel solvers, such as BlockSolve95 and pARMS, are integrated into DBuilder, with future work to support the PETSC interface. To learn more about DBuilder visit www.ercd.hpc.mil/scs/dbuilder/Dbuilder.html.

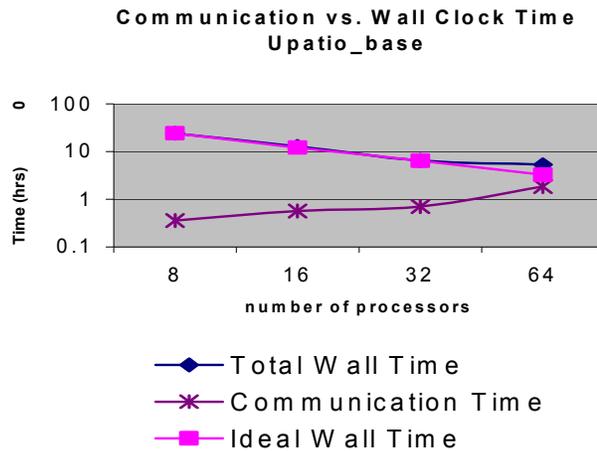


Figure 7. Performance measurement on Compaq SC45

Acknowledgments

This work was supported in part by an allocation of computer time from the DoD High Performance Computing Modernization Program at the Engineer Research and Development Center (ERDC) Major Shared Resource Center (MSRC). The ERDC Coastal and Hydraulics Laboratory, Vicksburg, MS, also provided support.

References

- [1] Karypis, George, et al., "ParMETIS: Parallel Graph Partitioning," <http://www-sers.cs.umn.edu/~karypis/metis/parmetis/>.
- [2] Kirk, Benjamin S. and Peterson J. W., "libMesh: A C++ Finite Element Library," <http://libmesh.sourceforge.net>, University of Texas at Austin.
- [3] Boman, Erik, et al., "Zoltan: Data-Management Services for Parallel Applications," <http://www.cs-sandia.gov/Zoltan/>.
- [4] Merten, Bart, et al., "DRAMA: A Library for Parallel Dynamic Load Balancing of Finite Element Applications," P. Amestoy, P. Berger. M. Daydé, I. Duff, V. Frayssé, L. Giraud and D. Ruiz, (eds), EuroPar'99 Parallel Processing. Lecture Notes in Computer Science, No. 1685, Springer-Verlag, 1999, pp 1-22.
- [5] Cheng, Jing-Ru C., et al., "Parallelization of the WASH123D code—Phase I: 2-Dimensional Overland and 3-Dimensional Subsurface Flows," to appear, the 2004 International Conference of Computational Methods in Water Resources.
- [6] Jones, Mark T. and Paul E. Plassmann, "BlockSolve95: Scalable Library Software for the Parallel Solution of Sparse Linear Systems," <http://www-unix.mcs.anl.gov/sumaa3d/BlockSolve/>.
- [7] Yousef, Saad, et al., "pARMS: parallel Algebraic Recursive Multilevel Solvers," <http://www-unix.mcs.anl.gov/sumaa3d/BlockSolve/>.