Parallel Mesh Interface and Ghost Exchange

Timothy J. Tautges[1] and Jason A. Kraftcheck[2]

[1]Argonne National Laboratory, Argonne, IL., USA
[2]University of Wisconsin-Madison, Madison, WI, USA

Abstract. Algorithms are described for the resolution of shared vertices and higher-dimensional entities on inter-processor interfaces on a domain-decomposed parallel mesh, and for ghost exchange between neighboring processors. Performance data is given for large (up to 64M hex element) meshes on up to 16k processors.

Introduction

Most parallel simulation codes solving systems of PDEs use a domain decomposition approach, where the mesh is split into P pieces, one piece per processor. Most of these methods require the knowledge about vertices and other entities shared between processors, and many also require the representation of one or most "ghost" layers of elements. Most mesh formats do not store this information, so it needs to be computed after mesh import. This note describes algorithms for identifying shared entities, and for exchanging ghost entities, that are efficient in both space and time.

MOAB is a library for query and modification of structured and unstructured mesh and field data associated with the mesh[1]. The data model implemented by MOAB references four distinct data types: entities (vertices, triangles, quads, etc.), entity sets (arbitrary collections of entities and other sets), Interface (the object through which all other functions are called, i.e. the database), and tag (information stored on Entity, Entity Set, and Interface objects). This data model can represent most semantic information associated with typical meshes, including boundary conditions, solution fields, geometric associativity, and parallel partitions.

[1] This work was supported by the US DOE's SciDAC program under Contract DE-AC02-06CH11357. Argonne National Laboratory (ANL) is managed by UChicago Argonne LLC under Contract DE-AC02-06CH11357. This work is also sponsored by the DOE NEAMS program.

In parallel meshes, MOAB tags shared entities with the rank of all sharing processor(s) and the remote entity handles, *on all sharing processors, not just the entity's owning processor*. In most cases, a parallel mesh is initialized by reading the mesh from disk, with information about mesh shared between processors established during the mesh reading process.

The algorithms described below make frequent use of the two classes Tuple List and Crystal Router. A Tuple List is a list of tuples, with each tuple having 0 or more int, long int, unsigned, and double datums, numbered 0..(n-1). Tuple List can sort tuples based on a specified datum index. The tuple_transfer() function communicates tuples to destination processors stored in a specified index of each tuple; this function greatly simplifies the routing of messages between processors.

Crystal Router is a class for parallel communication first described by Fox[3]. Starting from a tuple list on each processor with a specified index storing a global id, the Crystal Router returns to each processor a new list of tuples T'[j] = T(ind, np, T[np]), with ind the index of the shared integer value in the original tuple list, np the number of other processors also sharing the global id, and T[np] the original tuples from the sharing processors. gs_init works by partitioning the range of integers over processors such that each processor is responsible for a portion of the range.

Shared Interface Resolution and Ghost Exchange

The starting point for shared interface resolution is after the mesh residing on each processor has been read and initialized in MOAB (Figure 1, left)[2]; we assume a global id space exists for vertices, but not for other entities. To resolve shared vertices, each processor computes the vertices on the mesh skin, and creates a tuple list holding vertex handles and their global ids. This tuple list is passed to gs_init, which returns the vertices shared by other processors and their handles on those processors. At this stage, each processor knows which other processors it also shares higher-dimensional entities with, since sharing an entity also requires sharing the vertices. Next, the higher-dimensional entities on the skin are computed. For each of those entities, the intersection of the vertex sharing lists is computed, and the connectivity of the entity is packed onto a message for each of the resulting processors, along with the entity type and the local handle. Those vertex handles in the connectivity vector (which represent handles local to the sending processor) are replaced with the handles of

[2] The tools for partitioning a mesh and reading it in parallel are not described here, but are available (see the MOAB wiki[2] for further information).

those vertices on the destination processor, using sharing data computed in the first stage of the algorithm. Message are passed to neighbors asynchronously, and incoming messages are received and unpacked. For each entity in the incoming message, the entity type and connectivity list are used to search for a corresponding entity. If one is found, the sharing data is updated with the rank of and the entity handle on the sending processor. Note that for processors A and B sharing an entity, each sends the other a message containing that entity; this allows greater concurrency than if A sent the entity to B, then B returned that entity's handle to A. Figure 1 (right) shows the information after shared interface vertices and higher-dimensional entities have been resolved.

Ghost exchange uses many of the same steps as those used in shared interface exchange, with one important exception. Consider the situation shown in Figure 2, where processor P0 sends quad 0f to processors P1, P2, and P3. Recall that the ghost exchange algorithm must result in each processor knowing remote handles for any shared entity for all other processors sharing the entity. Thus, after receiving a new ghost entity and storing it in the database, a processor must communicate the new local handle for that entity not only to the sending processor, *but to all other processors that were sent that same entity*. This is done by including all destination processors and remote handles in the sharing list for the given entity when it is packed into the message on the sending processor, using zero for remote handles that are not yet known. When those messages are unpacked, the processor notes any new communicating processors (from which it will later receive remote handles). In some cases, the same ghost vertex is sent to a processor from several other processors. The receiving processor must detect that those vertices are in fact the same and create them only once. This is done by keeping a list on each processor of new ghost entities, *indexed by the handles for those entities on the owning processor.* Finally, ghost exchange can also result in new communication partners for a given processor, due to communication between two other processors. For example, in Figure 2, if P0 also sends quad 0f to processor P4, that results in P4 communicating with P1, P2, and P3, due to the vertices that must be sent with the quad. Receives are posted for any new communication partners while messages are unpacked.

Performance

Parallel performance for interface resolution and ghost exchange was measured with two meshes: a 32 million element hex mesh, and a 64 mil-

lion element tet mesh. All performance measurements were made running in virtual node mode on an IBM BG/P system installed at Argonne. Figure 3 shows times for reading, resolving interface mesh, and exchanging ghost elements for both meshes. The resolve and ghost steps of the process scale nicely with increasing numbers of processors, while the time for reading levels off after about 256 processors.

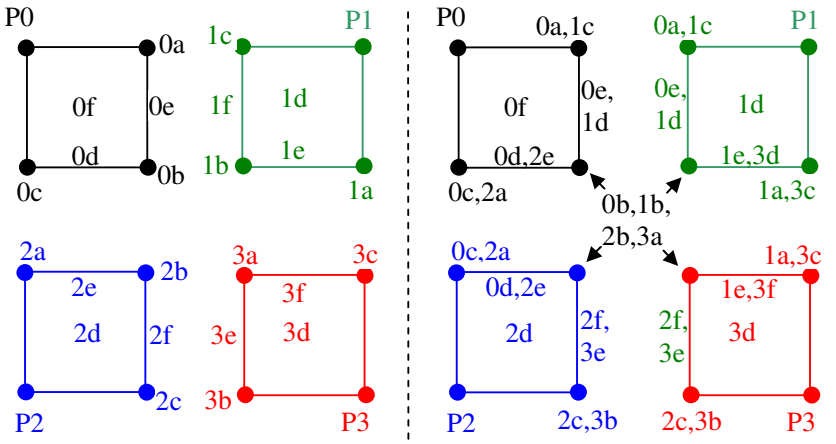

Figure 1: Mesh, with processors/entities labeled with numbers/letters (left); after shared interface resolution, with shared vertices/edges marked (right).

Conclusions

Algorithms for resolving shared interface mesh and for exchanging ghost elements are described which require mostly local, asynchronous communication, with the latter requiring only two rounds of communication. These algorithms scale well out to 16k processors of an IBM BG/P. Efforts are underway to improve scaling to greater numbers of processors, and to reduce the initial read time for these meshes.

REFERENCES

1. Tautges TJ, Meyers R, Merkley K, Stimpson C, Ernst C (2004) MOAB: A Mesh-Oriented Database, Sandia National Laboratories
2. MOAB Wiki Page, http://trac.mcs.anl.gov/projects/ITAPS/wiki/MOAB.

3. G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, "Solving Problems on Concurrent Computers," Prentice-Hall, Englewood Cliffs, New Jersey, vol. 19, p. 88.

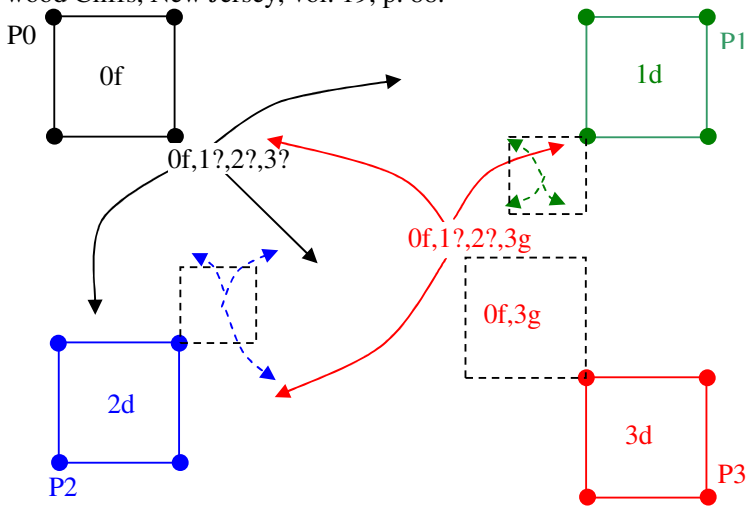

Figure 2: Ghost exchange; P0 sends quad 0f, along with sharing information 0f,1?,2?,3? (black arrows); P3 returns resulting handle 3g to all destination processors (red arrows) ; similarly for P1, P2 (green/blue arrows).

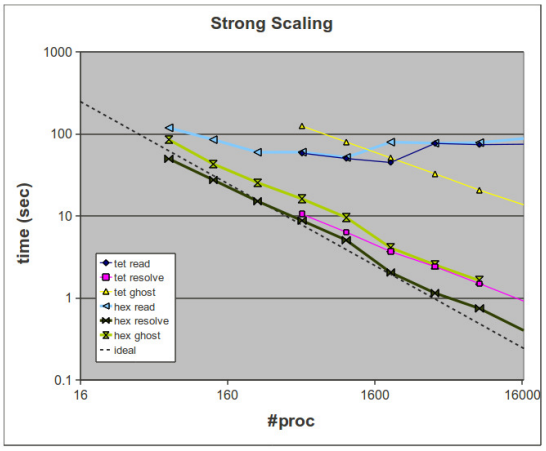

Figure 3: Strong scaling results reading, resolving shared interface mesh, and exchanging one layer of ghost elements. Hex and tet times are for 32M and 64M elements, respectively.