DOI: 10.14529/jsfi150304

INMOST Parallel Platform: Framework for Numerical Modeling

A.A. Danilov[1], *K.M. Terekhov*[1,2], *I.N. Konshin*[1], *Yu.V. Vassilevski*[1]

INMOST program platform allows a user to work with distributed data on general meshes. Description of the platform structure, interrelation of mesh elements, work with ghost cells, distribution and redistribution of mesh data is presented, as well as special aspects of the program platform implementation and usage. This paper aims to cover the following research topics: efficient distributed unstructured mesh representation data structure, flexible templates for numerical schemes implementation, convenient framework for parallel linear systems assesment and solution. For one of the specific tasks the exploiting of INMOST program platform is demonstrated on all stages of numerical modeling: distributed meshes construction, attachment of data to mesh elements, use of mesh data for problem discretization, as well as parallel solution of resulting linear systems. INMOST is a newly developed, flexible, and efficient numerical analysis framework that provides scientists with the infrastructure for creating highly scalable high performance computing modeling applications.

Keywords: distributed mesh, polyhedral mesh, parallel framework, numerical modeling.

Introduction

The amount of software for unstructured mesh generation, mesh adaptation, numerical analysis, and graphical visualization is huge. The problem escalates and the computational power of modern computer systems increases rapidly, hence, recent software development requires the use of parallel algorithms with distributed data. All these applications undoubtedly have a common set of needs for representing and manipulating distributed unstructured meshes. The typical infrastructure needed by these applications is a set of data structures for representing mesh and software mechanisms for accessing and modifying mesh data. A mesh representation consists of topological mesh entities (vertices, edges, faces, cells) and topological adjacencies (interentity connections). The combination of a mesh representation and a set of access mechanisms for mesh data is called a mesh framework or meshing infrastructure. Although the need of common infrastructure to enable the rapid and efficient development of mesh based programs is obvious, no single framework is considered flexible enough and commonly accepted. One reason for scarcity of general meshing infrastructures is that the needs of the various meshing and analysis applications vary widely and there is little agreement on computational efficiency of different mesh data representation. Therefore, a large number of mesh representations are in use in the computational community each tailored to a specific application. Some simple numerical analysis programs use only a minimal representation consisting of elements (quads, tetrahedra, etc.) defined by nodes (or points). Other more sophisticated applications like complex finite volume schemes for general polyhedral meshes find it useful to exploit a much wider mesh representation consisting of a full set of mesh entities with an extensive set of topological adjacencies [1]. Therefore, to gain widespread acceptance it is important to have a full mesh framework which allows applications to access all types of mesh entities. At the same time, the infrastructure should be lightweight and efficient to have sufficient utilities for real world applications.

[1]Institute of Numerical Mathematics RAS
[2]Stanford University

Fortunately, the need for general parallel meshing infrastructure is increasingly being recognized and several development efforts have been introduced in the last few years. These include the following packages: MSTK (Mesh Toolkit) [2], STK (Sierra Toolkit) [3], MOAB (A Mesh-Oriented datABase) [4, 5], FMDB (Flexible distributed Mesh DataBase) [6]. Some of these packages do not support dynamic modification of the mesh, some of them do not provide enough parallel functionality like several layers of ghost cells and some of them are still not reliable. Due to lack of appropriate parallel mesh framework for complex numerical analysis applications in early 2010s our group decided to design a convenient framework based on MSTK and MOAB which later evolved to the development of new parallel platform with flexibility and efficiency in mind – INMOST (Integrated Numerical Modeling Object-oriented Supercomputing Technologies).

INMOST is a newly developed, flexible, and efficient numerical analysis framework that provides application developers low-level infrastructure for reading, writing, creating, manipulating, and partitioning distributed unstructured meshes without having to design and implement their own mesh data structures. INMOST provides a convenient infrastructure for matrix assembling hence simplifying rapid development of discretization techniques. INMOST also provides framework for parallel linear system solvers including third party solver packages PETSc [7, 8] and Trilinos [9]. Newly developed parallel linear system solvers may be easily incorporated in or built upon INMOST infrastructure.

In this paper a brief description of INMOST is provided including details on design of the framework and software mechanisms for interacting with it. INMOST is in active development and some of the capabilities are not described here since their functionality may change in near future. However, the core functionality is considered stable and is described in this paper. INMOST is currently used to implement several applications such as safety analysis of nuclear and radioactive facilities [10], free surface computational fluid dynamics modeling [11], and black oil modeling [12] within Institute of Numerical Mathematics RAS Nuclear Safety Institute RAS and Stanford University. INMOST is currently available to general public under Modified BSD License at http://www.inmost.org/.

1. Platform structure

Only one general assumption on the mesh type is in place. The volume mesh is considered to be a conformal mesh with arbitrary polyhedral cells, i.e. any two cells either do not have common points or have one common vertex: either have one common edge or have one common face. INMOST platform have a modular structure allowing development of new modules and interfacing with third party software packages. The two core modules, mesh framework module and linear system solver module, will be covered in the current paper.

INMOST mesh is a dynamic distributed database representing unstructured grid. Operations of mesh modification and parallel mesh redistribution for load balancing require special data structure which is capable of data relocation, pruning, and compactification. Fast data access is one of the key features of INMOST. The direct memory access interface is used avoiding unnecessary data copying. This requires special attention to platform implementation ensuring continuous direct element data access during mesh modification at least until the mesh element is moved or deleted. Memory fragmentation during massive mesh modifications becomes a significant concern. INMOST platform utilizes a set of average size memory blocks for big data arrays. Each block is considered nonrelocatable, hence, ensuring continuous direct access the

blocks may be freed only during data compactification. INMOST implementation of mesh data structure is simple, flexible, and memory efficient with wide range of supported functionality. The detailed analysis of different mesh representations and supporting algorithms was performed by R. Garimella in his work [13]. INMOST mesh framework is based on full mesh representation with circular adjacencies vertex–edge–face–cell–vertex providing the balance between memory requirements and parallel algorithms efficiency (fig. 1).

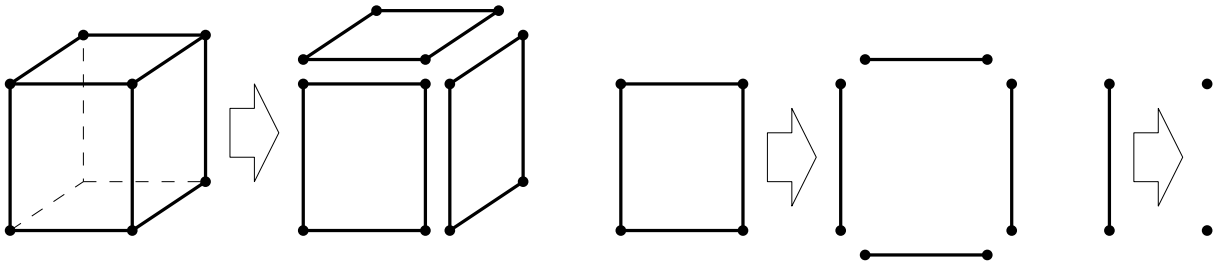

Figure 1. Downstream hierarchical representation of mesh elements

INMOST platform provides infrastructure for parallel grid generation. The user may rely on internal topology consistency checks and geometry calculators. The user decides which type of computed metric data for each type of mesh entities should be precomputed, cached, and updated on modification. The minimalistic parallel structured mesh generator will be presented below. Optionally the user can import and export the mesh in commonly used formats; internal cross architecture portable binary format is used to store the full set of mesh features.

One of the key points in parallel implementation of numerical analysis applications is the idea of domain decomposition and overlapping grids with ghost cells [14]. Ghost cells on one process are cells mirroring the corresponding cells from another process. Different discretization schemes may require different types and width of overlapping layers. In INMOST each mesh entity have exactly one process owner and a specific state for each sharing process: a “owned” one for entities which is only owned by the current process, a “shared” one for own entities which have copies on other processes, and a “ghost” one for copies of entities from other processes. Each shared entity also stores the list of processes it is copied to. INMOST automatically computes and distributes the ghost cells given the number of layers and connectivity type: neighbours through the faces, edges or vertices. Special attention is given to handling of multiple ghost layers, since cross-process transfers occur commonly. The user can also provide an explicit ghost map for specific applications.

The second key point in effective parallel computation is the load balancing. INMOST provides flexible interface for repartitioning and redistribution of the mesh. The user chooses one of the internal partitioners, the external widely used partitioners Zoltan [15] and Parmetis [16], or provides the partitioning map explicitly. Redistribution process effectively utilizes the ghost layers if any of them have been computed in advance and ensures that the ghost layers structure is preserved after redistribution.

INMOST platform provides flexible infrastructure for organization of mesh entities into hierarchical sets and associate user data with mesh entities. The user data is identified by named tags. Each tag may be associated with vertices, edges, faces, cells, sets, entire mesh or any combinations of them (fig. 2). Each tag stores real, integer or byte data or references to elements. Fixed and dynamic length arrays are natively supported. Tags may be dense, i.e. all appropriate mesh entities have the associated tag, or sparse, i.e. only some of the entities store the tag data. Markers are used in the same way as a boolean type dense tags. The tags data is

mirrored on demand to ghost cells, in addition INMOST supports common reduce operations to gather data from ghost cells back to owner.

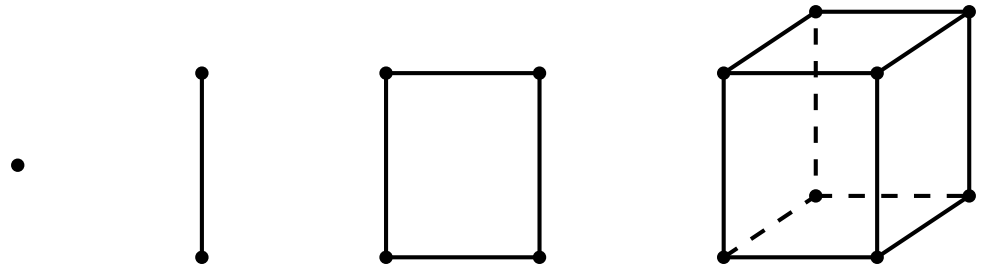

Figure 2. Mesh entities: vertex, edge, face and cell

INMOST solver class provides methods for sparse matrix assembling without prior knowledge of matrix sparsity structure. The linear solver infrastructure may be used to develop and implement specific iterative linear solver with special preconditioners. Several internal solvers are provided natively and convenient interfaces to PETSc and Trilinos solver packages are included as well.

The detailed algorithms and data structures are presented in [17].

2. User interface

```
#include "inmost.h"
using namespace INMOST;
int main(int argc, char *argv[])
{
  // Initialize INMOST activities
  Solver::Initialize(&argc,&argv,"");
  Mesh::Initialize(&argc,&argv);
  Partitioner::Initialize(&argc,&argv);
  ...
  // Finalize INMOST activities
  Partitioner::Finalize();
  Solver::Finalize();
  Mesh::Finalize();
  return 0;
}
```

Figure 3. Initialization and finalization of INMOST modules

The INMOST code is written in C++ language, it is a cross-platform and may be built using wide range of compilers. Modular structure of INMOST code allows the user to enable and disable optional components including interfaces to third party software. The code is organized in a way that the user does not need to exploit any of MPI specific procedures in order to create a parallel application. If any INMOST module is used it should be properly initialized and finalized before and after of its use respectively; see Fig. 3 for examples of several modules usage.

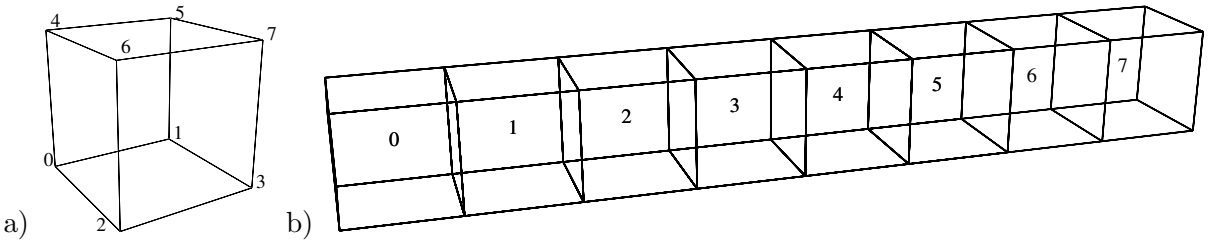

Figure 4. Minimalistic parallel cubic grid: a) one cell with local vertices, b) distributed grid with NP=8

```
Mesh *m = new Mesh();
int rank = m->GetProcessorRank(); // Get process rank
int size = m->GetProcessorsNumber(); // Get number of processes
ElementArray<Node> verts(m); // Create local vertices
for(int k=0; k<2; k++)
  for(int j=0; j<2; j++)
    for(int i=0; i<2; i++)
    {
      double xyz[3]={rank+i, j, k}; // Define node coordinates
      verts.push_back(m->CreateNode(xyz)); // Add new vertex
    }
// Define face connectivity template for cubic cell, rf. Fig. 4a
const int face_nodes[24] = {0,4,6,2, 1,3,7,5, 0,1,5,4,
                            2,6,7,3, 0,2,3,1, 4,5,7,6};
const int num_nodes[6]   = {4, 4, 4, 4, 4, 4};
m->CreateCell(verts,face_nodes,num_nodes,6); // Create the cubic cell
m->ResolveShared(); // Resolve duplicate nodes
m->Save("mesh.pvtk"); // Export mesh to parallel VTK format
delete m;
```

Figure 5. Minimalistic parallel mesh generation

In order to create a parallel mesh the user should set an MPI communicator, an alias INMOST_MPI_COMM_WORLD is provided for convenience by default. Once the communicator is set the process may obtain its rank and the total number of processes becomes known. We demonstrate minimalistic distributed mesh generation process below (see Figs. 4b, 5). One way to create a polyhedral cell is to define all nodes, all edges using nodes, all faces using edges, and a cell using faces. Alternatively short way is to define nodes and create a cell using connectivity template for its faces. In our example we use the second way: we define 8 nodes and create a cell with six faces, each face containing four nodes and their indices are prescribed by face_nodes template (see Fig. 4a). Once local grids are created we use ResolveShared() procedure to resolve duplicate nodes and assign global IDs to mesh entities. The distributed mesh is exported using Save() method.

The same idea is used to construct the distributed rectangular mesh, the only difference is that more vertices are defined and the cells are constructed in loop. The INMOST source package provides “GridGen” example which creates a distributed rectangular mesh for a unit cube.

This example can also be used to create a prismatic mesh. The readers are refered to the example source code for further details.

```
Mesh *m = ... // create or load mesh from file
m->ExchangeGhost(1,FACE);
Mesh::GeomParam table;
table[BARYCENTER] = CELL;
table[NORMAL] = FACE;
table[MEASURE] = CELL | FACE;
m->PrepareGeometricData(table);
Solver::Matrix A;
Solver::Vector b;
// Iterate over all faces
for(Mesh::iteratorFace f = m->BeginFace(); f != m->EndFace(); ++f)
{
  Cell r1 = f->BackCell(), r2 = f->FrontCell();
  // Skip face if cells are ghosts or not available
  if( (!r1->isValid() || r1->GetStatus() == Element::Ghost) &&
      (!r2->isValid() || r2->GetStatus() == Element::Ghost) ) continue;
  if (r1->isValid() && r2->isValid()) { // Internal face case
    double f_area = f->Area(); // Get the face area
    double f_nrm[3], r1_cnt[3], r2_cnt[2];
    f->Normal(f_nrm); // Get the face normal
    r1->Barycenter(r1_cnt); // Get the barycenter of the cell r1
    r2->Barycenter(r2_cnt); // Get the barycenter of the cell r2
    double coef = ... // Compute flux coefficients
                      // using f_area, f_nrm, r1_cnt, r2_cnt
    int id1 = r1->GlobalID(), id2 = r2->GlobalID(); // Get global IDs
    // Fill matrix coefficients only for normal cells
    if( r1->GetStatus() != Element::Ghost )
      A[id1][id1] += -coef, A[id1][id2] += coef;
    if( r2->GetStatus() != Element::Ghost )
      A[id2][id1] += coef, A[id2][id2] += -coef;
  } else {
    ... // Boundary face case
  }
}
// Iterate over all cells
for( Mesh::iteratorCell c = m->BeginCell(); c != m->EndCell(); ++c )
  if( c->GetStatus() != Element::Ghost )
    b[c->GlobalID()] += c->Volume()*c->Mean(rhs, 0); // Integrate rhs()
```

Figure 6. Assemble matrix of linear system

Different discretization techniques result in matrix assembling. We demonstrate the template for finite volume (FV) scheme shown in Fig. 6. The detailed example is packed within

```
Solver S(Solver::INNER_ILU2); // Specify the linear solver
S.SetMatrix(A); // Compute the preconditioner for the original matrix
S.Solve(b,x);   // Solve the linear system with the preconditioner
Tag phi = m->CreateTag("Solution", DATA_REAL, CELL, NONE, 1);
for(Mesh::iteratorCell c = m->BeginCell(); c != m->EndCell(); ++c)
  if( c->GetStatus() != Element::Ghost )
    c->Real(phi) = x[c->GlobalID()];
```

Figure 7. Solve linear system and attach solution to mesh cells

INMOST source package as "FVDiscr" example. In general case matrix elements depend on the neighboring ones, thus, a layer of ghost cells is usually needed. ExchangeGhost() method is used to create one or several layers of ghost cells. We use two point flux approximation in FV scheme, hence, for each internal face we need its area and barycenters of neighboring cells. We also need cell volumes for right-hand side calculation. INMOST can precompute and cache the needed geometric information using PrepareGeometricData() method. Several entity iterators are available in INMOST. In our FV scheme we iterate over all faces and compute matrix coefficients for neighboring cells BackCell() and FrontCell(). Global cell identificators GlobalID() are used as indices to create matrix and right-hand side vector.

Once the matrix is assembled one can utilize internal BiCGStab(L) solver with second order ILU factorization as preconditioner. The code example is presented in Fig. 7. The computed solution is stored in tag "Solution", a single real value is attached to all cells. A more detailed example of linear system solution is presented in "MatSolve" example from INMOST source code.

More complicated examples and test cases are bundled in INMOST package and demonstrate in more detail mesh generation, mesh partitioning and redistribution, matrix assembling for FV scheme of diffusion problem and linear system solution using different solvers and packages. The user is advised to consult with online documentation available on project site and take a look at unit tests for mesh and solver modules.

3. Numerical experiments

In this section several performance tests are presented using the code introduced above and included as code examples in INMOST package.

The code is used to solve the problem $-\nabla \cdot (K \nabla U) = f$ with Dirichlet boundary conditions, where K is unit tensor and the right-hand side f is computed from the exact solution: $U = \sin(\pi x)\sin(\pi y)\sin(\pi z)$. The parallel code creates a rectangular grid in unit cube, this stage is called "GridGen" below. The simplest two-point FVM scheme is used to assemble local matrices. Using ghost cells effectively links local matrices in a global matrix. This matrix assembly stage is refered to as "Assemble" below. At the final stage the linear system is solved, namely "MatSolve" stage below.

To perform parallel numerical experiments, two parallel computer systems were used: the INM RAS computer cluster and the "Lomonosov" computer cluster. We exploited the nodes of the "x6core" queue of the INM RAS cluster with the total of 12 nodes. These are Xeon X5650@2.67GHz nodes with 24 GB of memory and 12 cores per node. We also exploited the nodes of "regular4" queue of Lomonosov cluster with the total of 64 nodes. These are Xeon

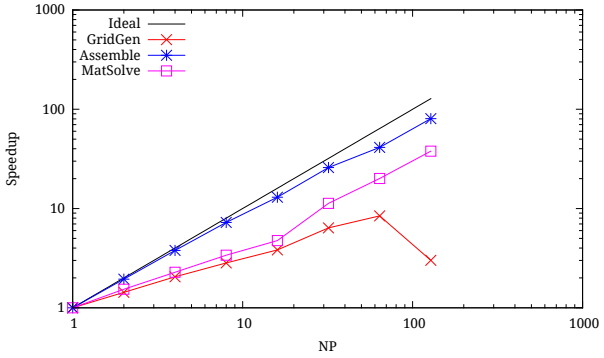

Figure 8. Strong scalability test on INM RAS computer cluster, speedup graph

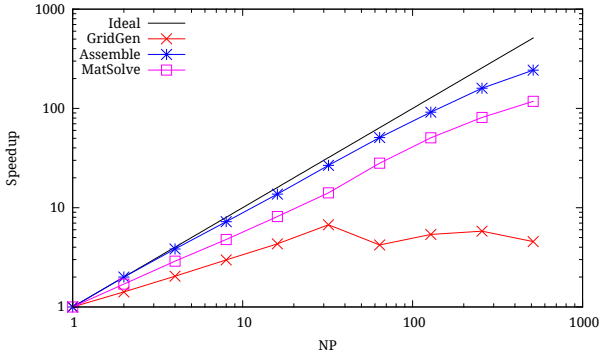

Figure 9. Strong scalability test on "Lomonosov" computer cluster, speedup graph

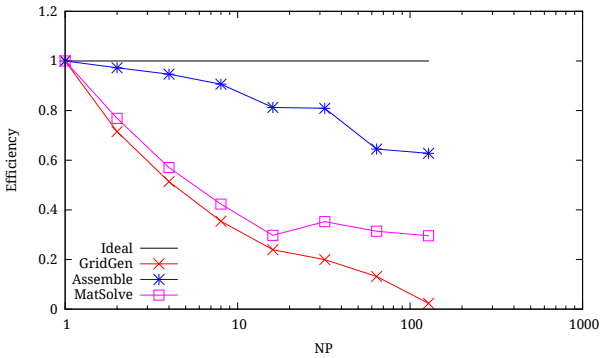

Figure 10. Strong scalability test on INM RAS computer cluster, efficiency graph

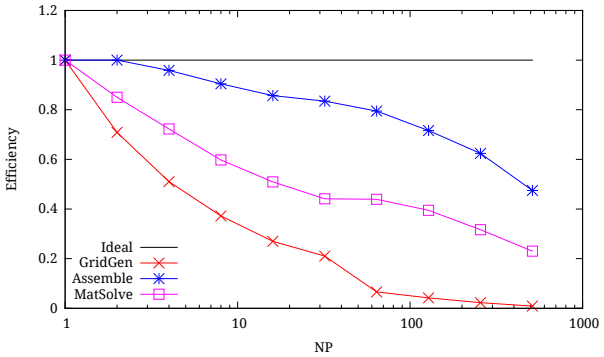

Figure 11. Strong scalability test on "Lomonosov" computer cluster, efficiency graph

X5570@2.93Ghz nodes with 12 GB of memory and 8 cores per node. In both cases Intel Compilers with Intel MPI were used.

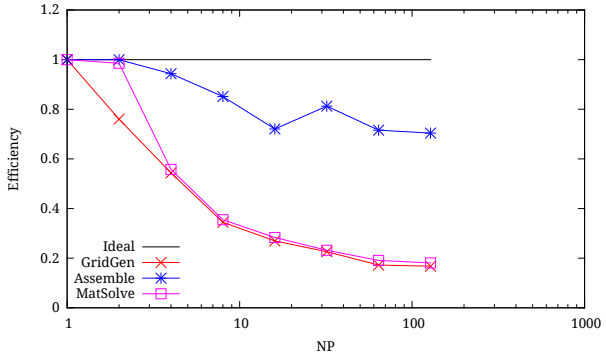

Figure 12. Weak scalability test on INM RAS computer cluster, efficiency graph

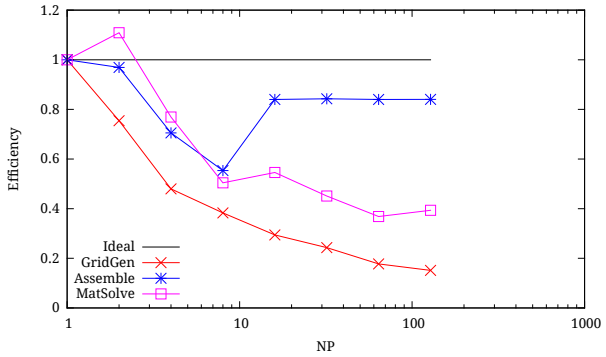

Figure 13. Weak scalability test on "Lomonosov" computer cluster, efficiency graph

Table 1. Strong scalability test on INM RAS computer cluster

	GridGen			Assemble			MatSolve		
NP	T	S	E	T	S	E	T	S	E
1	11.435	1.00	1.000	19.029	1.00	1.000	28.841	1.00	1.000
2	7.994	1.43	0.715	9.785	1.94	0.972	18.759	1.54	0.769
4	5.566	2.05	0.514	5.025	3.79	0.947	12.641	2.28	0.570
8	4.035	2.83	0.354	2.624	7.25	0.906	8.519	3.39	0.423
16	2.985	3.83	0.239	1.464	13.00	0.813	6.060	4.76	0.297
32	1.793	6.38	0.199	0.735	25.89	0.809	2.557	11.28	0.352
64	1.356	8.43	0.132	0.461	41.29	0.645	1.436	20.09	0.314
128	3.803	3.01	0.023	0.237	80.37	0.628	0.762	37.84	0.296

Both strong and weak scalability tests were performed on both clusters. The strong scalability test was performed using uniform $96 \times 96 \times 96$ cubic grid on INM RAS cluster (fig. 8, fig. 10), and uniform $100 \times 100 \times 100$ cubic grid on "Lomonosov" cluster (fig. 9, fig. 11). Total times, speedups, and efficiencies are presented in tab. 1 and tab. 2. Weak scalability test was performed using $64 \times 64 \times 64$ cubic grid on each process on INM RAS cluster (fig. 12), and $50 \times 50 \times 50$ cubic grid on each process on "Lomonosov" cluster (fig. 13). Total times and efficiencies are presented in tab. 3 and tab. 4.

Numerical experiments have showen reasonable scalability of "Assemble" and "MatSolve" examples during strong scalability tests. The "Assemble" example also results in high efficiency during weak scalability tests. Minor reduction in efficiency observed with $NP = 8$ on

Table 2. Strong scalability test on "Lomonosov" computer cluster

	GridGen			Assemble			MatSolve		
NP	T	S	E	T	S	E	T	S	E
1	15.882	1.00	1.000	28.552	1.00	1.000	64.849	1.00	1.000
2	11.204	1.42	0.709	14.279	2.00	1.000	38.129	1.70	0.850
4	7.783	2.04	0.510	7.446	3.83	0.959	22.438	2.89	0.723
8	5.335	2.98	0.372	3.946	7.24	0.904	13.561	4.78	0.598
16	3.677	4.32	0.270	2.082	13.71	0.857	7.960	8.15	0.509
32	2.359	6.73	0.210	1.069	26.71	0.835	4.594	14.12	0.441
64	3.768	4.22	0.066	0.561	50.85	0.795	2.308	28.10	0.439
128	2.956	5.37	0.042	0.311	91.68	0.716	1.283	50.55	0.395
256	2.737	5.80	0.023	0.179	159.76	0.624	0.800	81.03	0.317
512	3.488	4.55	0.009	0.117	243.08	0.475	0.550	117.94	0.230

Table 3. Weak scalability test on INM RAS computer cluster

	GridGen		Assemble		MatSolve	
NP	T	E	T	E	T	E
1	3.484	1.000	5.589	1.000	7.469	1.000
2	4.583	0.760	5.592	0.999	7.580	0.985
4	6.406	0.544	5.925	0.943	13.393	0.558
8	10.106	0.345	6.566	0.851	21.050	0.355
16	12.924	0.270	7.757	0.720	26.350	0.283
32	15.409	0.226	6.876	0.813	32.199	0.232
64	20.233	0.172	7.809	0.716	39.109	0.191
128	20.767	0.168	7.938	0.704	41.319	0.181

Table 4. Weak scalability test on "Lomonosov" computer cluster

	GridGen		Assemble		MatSolve	
NP	T	E	T	E	T	E
1	2.042	1.000	3.364	1.000	7.037	1.000
2	2.706	0.754	3.471	0.969	6.346	1.109
4	4.254	0.480	4.770	0.705	9.149	0.769
8	5.329	0.383	6.078	0.554	13.953	0.504
16	6.935	0.294	4.005	0.840	12.892	0.546
32	8.396	0.243	3.993	0.843	15.606	0.451
64	11.507	0.177	4.004	0.840	19.111	0.368
128	13.485	0.151	4.004	0.840	17.883	0.394

"Lomonosov" cluster (fig. 13) can be attributed to the limited memory access bandwidth on one node. Starting with $NP = 16$ the number of physical nodes increases and the overall

bandwidth increases as well. Poor performance of "GridGen" example can be attributed to mesh generator design flaw. The final step of ResolveShared() procedure involves multiple communications for huge grids. A better approach would be to generate coarse grid first, resolve shared entities, and only then refine the mesh on each processor.

Conclusions

INMOST program platform is presented which allows a user to work with distributed data on general meshes. Internal platform structure is addressed to showing flexibility and efficiency of the infrastructure. Several user interface examples are used for minimalistic demonstration purposes of mesh generation, matrix assembling and linear system solution. Numerical results demonstrate reasonable scalability of matrix assembly and linear system solution stages.

The work is partially supported by the Russian Foundation for Basic Research (RFBR) grants No.14-01-00830 and No.15-35-20991.

References

1. Danilov AA, Vassilevski YV. A monotone nonlinear finite volume method for diffusion equations on conformal polyhedral meshes. Russian Journal of Numerical Analysis and Mathematical Modelling. 2009;24(3):207–227.

2. Garimella RV. MSTK – A Flexible Infrastructure Library for Developing Mesh Based Applications. In: 13th International Meshing Roundtable, September 19-22, 2004, Williamsburg, Virginia, USA, Proceedings; 2004. p. 203–212.

3. Edwards HC, Williams AB, Sjaardema GD, Baur DG, Cochran WK. SIERRA Toolkit Computational Mesh Conceptual Model. Technical Report SAND2010-1192, Sandia National Laboratories; 2010.

4. Tautges TJ. MOAB-SD: Integrated Structured and Unstructured Mesh Representation. Engineering With Computers. 2004;20(3):286–293.

5. Tautges TJ, Meyers R, Merkley K, Stimpson C, Ernst C. MOAB: A Mesh-Oriented Database. Technical Report SAND2004-1592, Sandia National Laboratories; 2004.

6. Seol ES. FMDB: flexible Distributed Mesh Database for Parallel Automated Adaptive Analysis, Ph.D. Thesis, Rensselaer Polytechnic Institute; 2005.

7. Balay S, Abhyankar S, Adams MF, Brown J, Brune P, Buschelman K, et al. PETSc users manual. Argonne National Laboratory, ANL-95/11 - Revision 3.5; 2014.

8. Balay S, Gropp WD, McInnes LC, Smith BF. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In: Modern Software Tools in Scientific Computing, Birkhäuser Press; 1997. p. 163–202.

9. Heroux MA, Phipps ET, Salinger AG, Thornquist HK, Tuminaro RS, Willenbring JM, et al. An overview of the Trilinos project. ACM Transactions on Mathematical Software. 2005;31(3):397–423.

10. Kapyrin I, Konshin I, Kopytov G, Nikitin K, Vassilevski Y. Hydrogeological modeling in radioactive waste disposal safety assessment using the GeRa code. Russian Supercomputing Days: Proceedings of the international conference (September 28-29, 2015, Moscow, Russia). Moscow State University; 2015. p. 122–132. Russian.

11. Terekhov KM, Nikitin KD, Olshanskii MA, Vassilevski YV. A semi-Largangian method on dynamically adapted octree meshes, to appear in Rus.J.Num.Anal.Math.Model.

12. Konshin I, Kaporin I, Nikitin K, Vassilevski Y. Parallel linear systems solution for multiphase flow problems in the INMOST framework. Russian Supercomputing Days: Proceedings of the international conference (September 28-29, 2015, Moscow, Russia). Moscow State University; 2015. p. 96–103.

13. Garimella RV. Mesh Data Structure Selection for Mesh Generation and FEA Applications. International Journal of Numerical Methods in Engineering. 2002;55(4):451–478.

14. Bertsekas DP, Tsitsiklis JN. Parallel and Distributed Computation: Numerical Methods. Prentice-Hall; 1989.

15. Boman EG, Catalyurek UV, Chevalier C, Devine KD. The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: partitioning, ordering, and coloring. Scientific Programming. 2012;20(2):129–150.

16. Schloegel K, Karypis G, Kumar V. Parallel static and dynamic multi-constraint graph partitioning. Concurrency and Computation: Practice and Experience. 2002;14(3):219–240.

17. Vassilevski Y, Konshin I, Kopytov G, Terekhov K. INMOST – a software platform and a graphical environment for development of parallel numerical models on general meshes. Moscow State Univ. Publ., Moscow; 2013. Russian.