# PETSc DMNetwork: A Library for Scalable Network PDE-Based Multiphysics Simulations

SHRIRANG ABHYANKAR, Argonne National Laboratory
GETNET BETRIE, Argonne National Laboratory
DANIEL ADRIAN MALDONADO, Argonne National Laboratory
LOIS C. MCINNES, Argonne National Laboratory
BARRY SMITH, Argonne National Laboratory
HONG ZHANG, Argonne National Laboratory

We present DMNetwork, a high-level package included in the PETSc library for the simulation of multiphysics phenomena over large-scale networked systems. The library aims at applications that have networked structures such as those in electrical, gas, and water distribution systems. DMNetwork provides data and topology management, parallelization for multiphysics systems over a network, and hierarchical and composable solvers to exploit the problem structure. DMNetwork eases the simulation development cycle by providing the necessary infrastructure through simple abstractions to define and query the network components. This paper presents the design of DMNetwork, illustrates its user interface, and demonstrates its ability to solve multiphysics systems, such as an electric circuit, a network of power-grid and water subnetworks, and transient hydraulic systems over large networks with more than 2 billion variables on extreme-scale computers using up to 30,000 processors.

Additional Key Words and Phrases: Networks, multiphysics, extreme-scale computing

## 1. INTRODUCTION

Modeling, simulation, and analysis of critical infrastructures—assets providing essential services that form the backbone of a nation's health, security, and economy, including power distribution systems, water or gas distribution, communication, and transportation—are of paramount importance from several strategically important perspectives, including maintaining sustainability, security, and resiliency and providing key insights for driving policy decisions. Because of their physical and topological nature, infrastructure systems are usually represented mathematically as a network with its elements (nodes, edges) obeying physical laws, for example, energy and matter conservation laws. Furthermore, these systems are often composed of subcomponents with different physics. For instance, in power networks, some power plants will have a gas subsystem as well as a mechanical and an electrical one. Because of the difficulty in performing multiphysics simulations, different network subsystems traditionally have been simulated separately.

In this paper, we present a new library, DMNetwork, for modeling and simulation of network partial differential equation (PDE)-based multiphysics on extreme-scale computers. DMNetwork is seamlessly integrated into the *Portable Extensible Toolkit for Scientific computing* (PETSc) [Balay et al. 2019], thus allowing the use of PETSc hierarchical and composable solvers. Before DMNetwork, managing a network simulation in PETSc was tedious and difficult because the users themselves needed to

DOI: http://dx.doi.org/10.1145/0000000.0000000

(1) maintain the parallel data structures for the network application, including partitioning and setting up needed parallel communication between network physics components; and
(2) completely manage the mapping of their network data structure and physics models to a PETSc solver (e.g., ODE/DAE integrator) application programming interface (API).

DMNetwork provides the underlying infrastructure for managing the network topology and the physics components. It is designed to scale for large networks while facilitating easy and rapid development of applications. DMNetwork can interface with other packages. EPANET, software for simulating water distribution piping systems [Rossman 2000], is an example of the type of network simulation tool that could benefit by incorporating PETSc DMNetwork to provide efficiency and scalability. [1]

Most software packages for network analysis are developed for representing and determining network structures as graphs [Hagberg et al. 2008]. Several tools for network analysis of pure graphs are currently available [NetworKit Development Team 2017; NetworkX Development Team 2018; Leskovec and Sosič 2016] that provide statistical measures, for example centrality and minimum distance. However, they do not pose physics systems modeled by PDEs on the network structure. Packages are available that provide modeling physics to assist scientists and engineers in modeling complex multiphysics networked problems (e.g., Simulink® [Mathworks 2017], Modelica [Association 2017], and LabView [Instruments 2017]), but the problems they can solve are restricted by size. Tools to model and assess the interdependencies between coupled systems are still in an early stage, particularly from a standpoint of high-performance computing. PLASMO [Jalving et al. 2017] is a Julia-based package for coupled optimization of electric and gas networks.

This paper is organized as follows. In Section 2 we introduce DMNetwork, including its design, user interface, and our innovative developments. In Section 3 we present how DMNetwork enables easy use of hierarchical composable solvers, including multilevel domain-decomposition preconditioners based on fieldsplittings [Brown et al. 2012] for multiphysics systems. In Section 4 we demonstrate simulations on hydraulic transient networks with billions of variables on extreme-scale computers. Throughout the paper, we present examples of the application of DMNetwork to the solution of linear electrical circuits, nonlinear power flow, hydraulic networks, and the interconnection of power and hydraulic networks. In the last section we provide a short conclusion and a brief look at future work.

## 2. DMNETWORK: DESIGN AND INTERFACE

The aim of this work is to provide a highly efficient and scalable general framework for expressing network problems based on PDEs and couplings among them. In scientific environments, different parts of complex networks are often modeled by different teams who develop their own specific methods and procedures. Simulation of the overall network is done by using cosimulation, where the interactions between the models are handled by passing data back and forth [HELICS Development Team 2017; OpSim Development Team 2017]. The Functional Mock-up Interface (FMI) [FMI Standard Development team 2017] standard prescribes a set of cosimulation directives for exchanging data between heterogeneous simulators.

Expressing problems of different natures in a common framework is useful for exposing their structure and discovering new ways to exploit it. Often, coupled networks have the form of partial differential-algebraic equations (PDAES) [Bartel and Günther

[1] See *petsc/src/snes/examples/tutorials/network/water/water.c* [Balay et al. 2019].

2018], where the algebraic part arises from Kirchoff-like laws. For the coupling between subsystems, using tailored equations has been shown to lead to faster convergence of the problem [Singer and Cucuringu 2010] .

DMNetwork is developed as a subclass of the DMPlex in PETSc. DMPlex handles general unstructured meshes, the mathematical models on the meshes, and their connection to the PETSc solvers [Lange et al. 2016]. DMNetwork is a subclass of DMPlex that strips away all its mesh-specific items, simplifying it to management of vertices and edges. The design of DMNetwork started from single-network applications (i.e., for a single network only); then it grew to handle multiple coupled networks since many applications require management of a network of networks and/or interactions between different networks. An example of such an application is understanding the interdependencies between different infrastructures, such as the impacts of water shortage on electrical power output or the effect of electrical power outages on natural gas supply.

### 2.1. Basic Building Blocks in DMNetwork

DMNetwork uses three building blocks for managing the network topology and the physics:

— *Vertex*
  A vertex is a connection point in the topology graph connecting one or more edges. From an application point of view, a vertex can be regarded as a physical connection point, such as a power station, factory, or gas network delivery point.

— *Edge*
  An edge represents a connection between two vertices. Edges represent connections between physical assets in a given network, such as a transmission line connecting two electrical substations or a gas/water pipeline.

— *Component*
  A component represents the physics associated with a vertex or an edge. For instance, a component for an edge can be a resistor, and a component for a vertex can be water treatment plant. Components for a network associate the physics of an application with the graph of the network defined by the vertices and edges. A component for DMNetwork is represented by a flat (serialized) data structure that describes the parameters required for describing the component physics. DMNetwork supports having multiple components on any vertex or edge.

New vertices and edges can be easily inserted through the API, and the existing ones can be removed or updated with minimum local changes. We use the term *network point* to refer to either an edge or a vertex. A component can be retrieved and manipulated by querying a network point.

### 2.2. Flow of DMNetwork Application and API Functions

The DMNetwork class contains topological information about the problem as well as physical modeling data. To build an application using DMNetwork, the user needs to carry out the following steps, as illustrated by Figure 1.

(1) *Create network*.
First we create a DMNetwork object.

```
DMNetworkCreate(MPI_Comm comm,DM *network);
```

In the current release of DMNetwork, we assume that the data files that contain the network data are read on the root process. During the network distribution stage, a new distributed network is created and the network data shipped off to

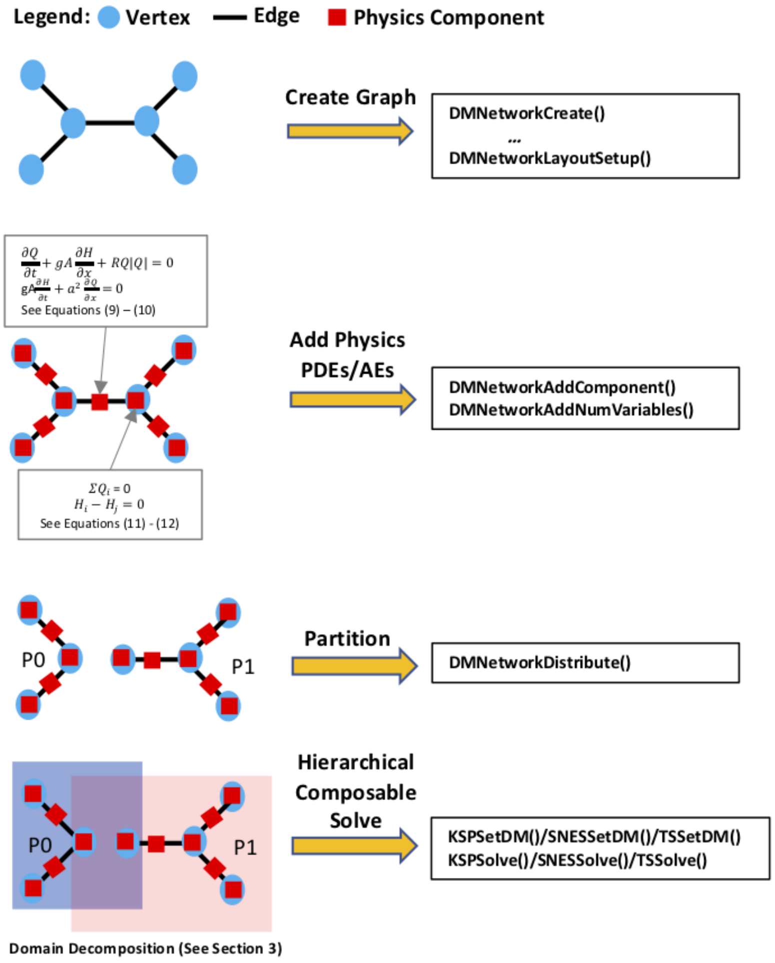

Legend: ● Vertex — Edge 🟥 Physics Component
Create Graph
DMNetworkCreate()
...
DMNetworkLayoutSetup()
$\frac{\partial Q}{\partial t} + gA\frac{\partial H}{\partial x} + RQ|Q| = 0$
$gA\frac{\partial H}{\partial t} + a^2\frac{\partial Q}{\partial x} = 0$
See Equations (9) – (10)
$\Sigma Q_i = 0$
$H_i - H_j = 0$
See Equations (11) - (12)
Add Physics PDEs/AEs
DMNetworkAddComponent()
DMNetworkAddNumVariables()
P0
P1
Partition
DMNetworkDistribute()
P0
P1
Hierarchical Composable Solve
KSPSetDM()/SNESSetDM()/TSSetDM()
KSPSolve()/SNESSolve()/TSSolve()
Domain Decomposition (See Section 3)

Fig. 1. DMNetwork application steps.

an appropriate processor determined by the partitioner. Because the data is read on a single process, the current implementation is obviously limited by the size of the data set. Capability for parallel creation of DMNetwork is under active development and will be released this year.

(2) *Register components*.
DMNetwork requires the user to register each associated physics component:

```
DMNetworkRegisterComponent(DM network,const char* name,size_t size,PetscInt *
    key);
```

*Name* is the user-selected name of the component, and *size* is the size in bytes (obtained through *sizeof()* function) of the component data structure. On successful registration of the component, a *key* is returned that is a unique identifier for that component and can be used subsequently to add this particular component to vertices or edges.

(3) *Set up network graph*.
This step sets up the network topology through the following API functions.

```
DMNetworkSetSizes(DM network,PetscInt Nsubnet,PetscInt nv[],PetscInt ne[],
    PetscInt NsubnetCouple,PetscInt nce[]);
```

*Nsubnet* and *NsubnetCouple* are the numbers of subnetworks and coupling subnetworks, respectively. Users set the sizes of subnetworks through the arrays that contain the number of local (to the given MPI process) vertices (*nv[]*), edges (*ne[]*), and coupling edges (*nce[]*).

```
DMNetworkSetEdgeList(DM network,PetscInt *edgelist[],PetscInt *edgelistCouple
    []);
```

*Edgelist* is an array of integer arrays (one for each subnetwork) describing the vertex connectivity for each edge in that subnetwork. For example, a subnetwork having 3 vertices and 2 edges, with edges connecting vertices 0-1 and 1-2, has the edgelist {0,1,1,2}. *EdgelistCouple* is an array of integer arrays describing the connected vertices between two subnetworks. For example, {0,4,1,0} represents a coupling edge that connects vertex 4 of subnetwork 0 to vertex 0 of subnetwork 1. After setting the network size and connectivity, the function

```
DMNetworkLayoutSetUp(DM network);
```

needs to be called. It constructs the network graph, but no physics is yet associated with the network at this stage.

(4) *Add components and variables*.
A component is added to a given vertex or edge with the following function.

```
DMNetworkAddComponent(DM network,PetscInt point,PetscInt key,void *comp);
```

The number of variables or degrees of freedom for a given edge or vertex is set by the following function.

```
DMNetworkAddNumVariables(DM network,PetscInt point, PetscInt numvar);
```

Here, *point* is the network point (either an edge or a vertex), *key* is the key obtained from *DMNetworkRegisterComponent*, and *comp* is a pointer to component data. The range of points for vertices and edges can be obtained by the following query functions.

```
DMNetworkGetVertexRange(DM network,PetscInt *vStart,PetscInt *vEnd);
DMNetworkGetEdgeRange(DM network,PetscInt *eStart,PetscInt *eEnd);
```

Here, *vStart* and *vEnd* are the starting and end+1 numbers for the vertices, while *eStart* and *eEnd* are corresponding points for edges. The vertices and edges in subnetwork *sub* can be queried by

```
DMNetworkGetSubnetworkInfo(DM network,PetscInt sub,PetscInt *nv, PetscInt *ne,
    const PetscInt **vtx, const PetscInt **edge);
```

returning arrays for vertices (*vtx*) and edges (*edge*) in the subnetwork *sub*.

> *Internal Design Note:*
> DMNetwork uses the following three array/objects to manage the component data (see *petsc/include/petsc/private/dmnetworkimpl.h* [Balay et al. 2019]).
>
> - *header*:
>   This field in the DMNetwork object is an array of structures of size *nv+ne*. Each structure stores information about components at each network point. The stored information is the size of the each component's data structure, the key registered for that component, and the offsets of that component's data in the data section.
> - *DataSection*:
>   DataSection is an object of type *PetscSection* (a PETSc object for managing integer arrays) in the DMNetwork object that is used when distributing the component data.
> - *cvalue*:
>   This is a two-dimensional array of pointers of size $nv \times ne$ to hold the component data. On adding a component to a point $p$, the $cvalue$ array for the corresponding point holds the reference for the component data structure.
>
> When a component is added to a vertex or an edge, the *header* is updated with the size, key, and offset; the *DataSection* adds a *dof* (equal to the size of the component data structure) at the given pointer, and the *cvalue* array is updated to hold a reference to the component data structure.
>
> DMPlex orders edges first, followed by vertices. In the above 3-vertex, 2-edge example, the two edges will have point numbers 0, 1 and the vertices 2, 3, 4. DMNetwork uses a *PetscSection* called *DofSection* to manage the degrees of freedom. This section is later used by DMPlex for creating local and global vectors. On calling *DMNetworkAddNumVariables()*, the number of variables for the given point in *DofSection* gets updated.

(5) *Set up and distribute DMNetwork*.
After adding the components and variables at vertices and edges,

```
DMSetUp(DM network);
```

must be called. It sets up the DMNetwork internal data structures to be used with the PETSc solvers.

> *Internal Design Note:*
> DMSetUp is responsible for performing the following operations:
>
> - A contiguous array is allocated to hold the data for all the components added to the network, and the components are copied over to this array. The component information for each point has the *header* information first, followed by the component data. This array is used for distributing the component data to the appropriate processors when doing the partitioning.
> - Setup of the *DofSection*, which holds the information on number of variables at each network point.

A given network can be then partitioned and distributed to multiple processors to approximately equalize the number of unknowns per process by calling the function

```
DMNetworkDistribute(DM dm, PetscInt overlap};
```

Here, *overlap* is the amount of edge overlap between partitions. DMNetwork has been tested with nonoverlapping partitions (i.e., *overlap*=0). On distribution, each process gets a part of the network that includes its local edges, vertices and ghost vertices. DMNetwork uses graph-partitioning packages, such as ParMetis [Karypis and Kumar 1997] or Chaco [Hendrickson and Leland 1995], to decide what edges and vertices are assigned to each process. The DMPlex code manages the movement of the data to the appropriate processes from the root process where the data was provided by the user code.

(6) *Associate DMNetwork with the PETSc solvers*.
With the above steps the user has created a distributed network object that contains the following:
— A (partitioned) graph representation of the problem.
— Physical data in each vertex and edge, related to the model for that entity.
— A data structure containing ghost vertices and communication data structures.
Now, the user can associate this DMNetwork with a PETSc solver via

```
SolverSetDM(PetscSolver Solver,DM network);
```

and retrieve it during function evaluation with

```
SolverGetDM(PetscSolver Solver,DM *network);
```

Here *Solver* represents a PETSc solver, that is, KSP (linear solver), SNES (nonlinear solver), or TS (time stepper).

### 2.3. Residual Function Evaluation

We use an example to illustrate how DMNetwork facilitates the evaluation of the residual function (or right-hand side vector for linear problems).

*Example: Electric Circuit*

We solve a toy linear electric circuit problem from [Strang 2007]. The topology of the electrical circuit is shown in Fig. 2. The circuit obeys the Kirchhoff laws. Hence, in the vertices of the graph, the energy is not accumulated:

$$\sum_j i^{(j)} = i_{\mathit{source(k)}} \, , \tag{1}$$

where $i^{(j)}$ is the current flowing though the branch (edge) $j$, incident to the node (vertex) $k$. The $i_{\mathit{source(k)}}$ allows one to account for current sources at node $k$. The voltage drop across the edge $k$, from $v_{(i)}$ to $v_{(j)}$, is defined by Ohm's law plus any voltage source $v^{(k)}_{\mathit{source}}$:

$$\frac{i^{(k)}}{r^{(k)}} + v_{(j)} - v_{(i)} = v^{(k)}_{\mathit{source}} \, . \tag{2}$$

We use the superscript and subscript to distinguish between edge and vertex quantities, respectively. In this case $i^{(*)}$ is an edge variable, and $v_{(*)}$ is a vertex variable.

These equations can be represented with the KKT matrix and the graph Laplacian. This structure is shared by many network flow problems, such as water networks:

$$\begin{bmatrix} R^{-1} & A \\ A^T & \end{bmatrix} \begin{bmatrix} i \\ v \end{bmatrix} = \begin{bmatrix} v_{\mathit{source}} \\ i_{\mathit{source}} \end{bmatrix} \, . \tag{3}$$

The practical implementation of this problem requires knowing the topology or connectivity of the network (a list of vertices and a list of edges defined by vertex pairs) and the physics (resistance, values of voltage source, and current source).

The following is a code fragment from *petsc/src/ksp/ksp/examples/tutorials/network/ex1.c* [Balay et al. 2019]. In this code, we define two physics components, *Node* and *Branch*

```
typedef struct {
```

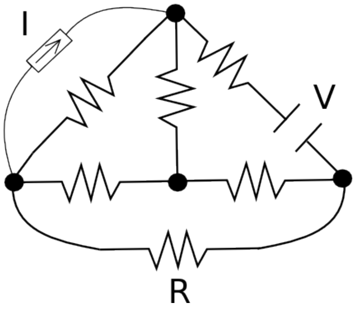


Fig. 2. Network diagram. Resistances are represented by zig-zag lines ($R$), voltage sources by parallel lines ($V$), and current sources by an arrow ($I$). The vertices of the graph are potentials, and the current flows across the edges.

```
  PetscInt    id;  /* node id */
  PetscScalar inj; /* current injection (A) */
  PetscBool   gr;  /* boundary node */
} Node;

typedef struct {
  PetscInt    id;  /* branch id */
  PetscScalar r;   /* resistance (ohms) */
  PetscScalar bat; /* battery (V) */
} Branch;
```

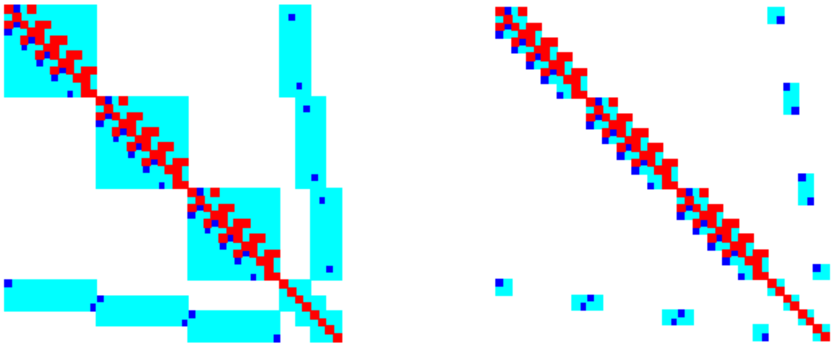
The subroutine below illustrates how to access DMNetwork information, iterate over the graph structure, query network elements and associated physics components, and then define entries of the matrix and right-hand side vector.

```
PetscErrorCode FormOperator(DM dmnetwork, Mat A, Vec b)
{
  DMNetworkGetEdgeRange(dmnetwork,&eStart,&eEnd);
  DMNetworkGetVertexRange(dmnetwork,&vStart,&vEnd);

  VecGetArray(b,&barr);
  for (e = eStart; e < eEnd; e++) { /* loop over edges */
    DMNetworkGetComponent(dmnetwork,e,0,NULL,(void**)&branch);
    DMNetworkGetVariableOffset(dmnetwork,e,&lofst);

    DMNetworkGetConnectedNodes(dmnetwork,e,&cone);
    DMNetworkGetVariableOffset(dmnetwork,cone[0],&lofst_fr);
    DMNetworkGetVariableOffset(dmnetwork,cone[1],&lofst_to);

    /* set rhs b for Branch equation (2) */
    barr[lofst] = branch->bat; /* battery value */

    /* set Branch equation (2) */
    row[0] = lofst;
    col[0] = lofst; val[0] =  1./branch->r;
    col[1] = lofst_to; val[1] =  1;
    col[2] = lofst_fr; val[2] = -1;
    MatSetValuesLocal(A,1,row,3,col,val,ADD_VALUES);

    /* set Node equation (1) */
    /* from node */
    DMNetworkGetComponent(dmnetwork,cone[0],0,NULL,(void**)&node);
    if (!node->gr) { /* not a boundary node */
      row[0] = lofst_fr;
      col[0] = lofst;   val[0] = -1;
      MatSetValuesLocal(A,1,row,1,col,val,ADD_VALUES);
32       }
33
34      /* to node */
35      DMNetworkGetComponent(dmnetwork,cone[1],0,NULL,(void**)&node);
36      if (!node->gr) { /* not a boundary node */
37        row[0] = lofst_to;
38        col[0] = lofst;   val[0] = 1;
39        MatSetValuesLocal(A,1,row,1,col,val,ADD_VALUES);
40      }
41    }
42
43    /* set rhs b for Node equation (1) */
44    for (v = vStart; v < vEnd; v++) { /* loop over vertices */
45      DMNetworkIsGhostVertex(dmnetwork,v,&ghost);
46      if (!ghost) {
47        DMNetworkGetComponent(dmnetwork,v,0,NULL,(void**)&node);
48        DMNetworkGetVariableOffset(dmnetwork,v,&lofst);
49
50        if (node->gr) {
51          row[0] = lofst;
52          col[0] = lofst;   val[0] =  1;
53          MatSetValuesLocal(A,1,row,1,col,val,ADD_VALUES);
54        } else {
55          barr[lofst] += node->inj;
56        }
57      }
58    }
59    VecRestoreArray(b,&barr);
60    MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
61    MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
```

First we retrieve the ranges of edges and vertices (Lines 3–4). We experimented with a higher-level iterator construct to loop over the entities but found that the simple data structures provide higher performance and a simplicity of debugging. Of course developers are free to layer an iterator abstraction of their liking directly on top of the DMNetwork interface if they value the syntax it offers. We then iterate over the edges (Line 7). For each edge we retrieve the component *branch* (that is, a pointer to the application-specific data for this edge), the edge variable offset, and the offsets for the variables in the boundary vertices (Lines 8–13). Next we write the Kirchhoff voltage law (Lines 19–23); and then, for each boundary vertex, we check whether the vertex is not a ghost value and write its contribution to the Kirchhoff current law (Lines 27–40). We then iterate over each vertex and add the contribution of each current injection to the corresponding equation (Lines 44–58).

### 2.4. Scalable Finite-Difference Jacobian Approximation with Coloring

In engineering fields, from which many of the network problems arise, the models often have a complicated structure: they may include control logic and react in a discrete way to transients in the network. Writing an analytic Jacobian matrix evaluation subroutine is a time-consuming, error-prone, and often prohibitive task. When the user does not provide a routine for analytic Jacobian evaluation, PETSc offers tools to calculate a finite-difference approximation of the Jacobian matrix suitable for some classes of problems. Note that for solving nonlinear problems, inexact Newton-like methods are often used, in which outer Newton iterations determine the convergence criteria while inner linear iterations ensure the asymptotic convergence of the method [Balay et al. 2019]. The inner solutions are approximations. Thus, when analytical Jacobian matrices are not available, approximate Jacobian matrices, that is, finite-difference Jacobian approximations, produce satisfying solutions for many applications.



(a) Jacobian matrix generated by DMPlex. Number of colors used: 40.

(b) Jacobian matrix generated by DMNetwork. Number of colors used: 6.

Fig. 3. Comparison of Jacobian preallocation with DMPlex (a) and DMNetwork (b).

DMPlex and DMNetwork contain information about the connectivity of the vertices and edges, which enables building a sparse Jacobian matrix structure and using matrix coloring schemes [Coleman and Moré 1983] for finite-difference Jacobian approximation. For DMNetwork, we made two new developments:

(1) Because the ghost points occur only on the vertices and the edges do not directly interact with each other, the columns associated with an edge do not share any row element of other edges in the Jacobian matrix. Based on this information, we implemented a new routine, which ensures that the number of colors of the Jacobian matrix is independent of the total number of edges in the network and that the column indices for the interior points of edges are not sent to the other computer processes. Our experiments show that this new routine minimizes the number of colors and the amount of interprocessor data communication for DMNetwork Jacobian computation.

(2) DMPlex creates the Jacobian matrix structure using the connectivity of the network graph with dense blocks representing edges and vertices, which becomes memory prohibitive when edges or vertices are built with components having large numbers of variables (see Fig. 3(a)). We enable the user to input application-specific analytical subblocks or sparse nonzero substructures to replace the default dense blocks at the edge and vertex points.

```
DMNetworkEdgeSetMatrix(dmnetwork, e, Juser);
DMNetworkVertexSetMatrix(dmnetwork, v, Juser);
```

The finite-difference Jacobian approximation with customized coloring via DMNetwork then involves the following steps:

(1) The user provides a subroutine for local function evaluation.
(2) The user provides $Juser$, the problem-specific sparse matrix nonzero structures at the network point $e$ or $v$; if $Juser$ is not provided, dense matrix subblocks are used.
(3) DMNetwork builds the global sparse Jacobian matrix structure by assembling the overall networks and application-specific sparse matrix blocks (when provided).
(4) The Jacobian matrix is computed by finite-difference approximation using the newly developed matrix coloring scheme in an efficient and scalable fashion.

*Example: A Small Water Pipe Network*

The example *petsc/src/ts/examples/tutorials/network/wash/pipes1.c* simulates a network of three water pipes (edges) connected linearly by four junctions (vertices). Figures 3(a) and 3(b) show the nonzero structure of a finite-difference Jacobian matrix of size 44 × 44 using DMPlex and DMNetwork. DMPlex generates dense blocks for edges and vertices, while DMNetwork employs user-provided block tridiagonal submatrices for water pipes (edges) and tiny sparse blocks for junctions (edge-vertex couplings). By taking advantages of network structure and application input, DMNetwork computes the Jacobian matrix using much less storage and a significantly smaller number of colors and interprocessor data communication.

## 3. HIERARCHICAL AND COMPOSABLE SOLVERS ON A NETWORK OF NETWORKS

The design of DMNetwork introduced in the preceding section decouples the user-specific physics models from the library solvers and the parallel computer implementation. DMNetwork provides an abstraction that allows users to express their problem concisely, hide cumbersome data management operations, and use flexible and efficient PETSc solvers. By querying DMNetwork, users can retrieve a

- subnetwork, for example, an electric power-grid network or a hydraulic network;
- network point, that is, an edge or a vertex;
- component, for example, a water pipe, a power generator, or a power transmission line; or
- coupling element, for example, a link between a power generator in the power-grid network and a water pump in a hydraulic network.

Through the retrieved elements, users directly access their application data and transparently apply their own operations, for example, function evaluations or Jacobian matrix approximations.

A user-friendly API and robust execution are critical aspects of software libraries. While the entire DMNetwork is distributed among multiple computer processors, our current design of DMNetwork assigns an individual network point (an edge or a vertex), together with its components, to a single process. Thus users actually write sequential code at these network points, for example, an algebraic equation at a voltage point (Equation (1)) and a PDE system for a single water pipe (Equations (9 - 10)). The PETSc library assembles these sequential network points into a parallel network distributed over multiple computer processors. We may add support for parallel processing single network points if the load imbalance becomes an issue.

On the library side, PETSc views DMNetwork as a single instance/object of its DM class and applies its hierarchical composable solvers to the DMNetwork. For networked system composition and decomposition, the following methods are particularly useful.

(1) *Domain decomposition* [Smith and Tu 2013]:
Decomposition of a computational domain into smaller subdomains. Each subnetwork is defined by a subregion of the entire region on which the original network is defined, as illustrated in Fig. 4.
(2) *Fieldsplit* [Brown et al. 2012; Smith et al. 2012]:
Splitting of a multiphysics system into multiple single-physics subsystems. Fig. 5 shows a network composed of two subnetworks that represent distinct physics (for example, electrical and water distribution).
(3) *Hierarchical and composable solvers for network of networks*:
Splitting a network into a hierarchy of global and local networks. These are then associated with PETSc composable solvers.

We have further developed and customized these methods for applications that use DMNetwork.

As illustrated by Fig. 1, we first construct a network or a composite network by creating a DMNetwork object. We then add a series of physical elements (for example,

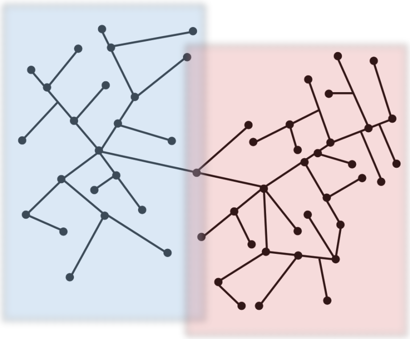


Fig. 4. Network partitioned into two subnetworks, with regard to its topology.

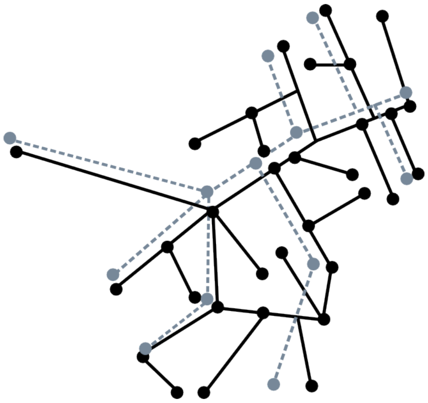


Fig. 5. Network composed of two subnetworks of distinct physics (represented by solid and dashed lines) (for example, electrical and water distribution). Customized solvers can be applied to the individual subnetworks.

water pipes, gas pipes, and power transmission lines) as the edges of the graphs and we add pipe junctions, electrical generators, and load systems as the vertices of the graph. From this information, DMNetwork, through the user-provided residual function evaluation, builds a mathematical model for the entire physical system. The model is generally either a nonlinear algebraic equation or a differential algebraic equation (DAE). The Jacobian operator of such a system often has the following structure:

$$N = \begin{bmatrix} P_1 & & & \dots & C_1 \\ & P_2 & & \dots & C_2 \\ & & P_3 & \dots & C_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ D_1 & D_2 & D_3 & \dots & J \end{bmatrix}, \tag{4}$$

where the submatrices $P_i$ represent individual elements of the network: a water pipe or a power transmission line, a water reservoir or an electrical generator. These elements can have dissimilar structures and time scales. For instance, in power systems each vertex represents a mechanical generator that produces electrical energy, each of which will have different parameters and even different equations. In water networks, the edges represent water conduits modeled with partial differential equations, whereas the vertices represent either simple boundary conditions of pressure and flow or complex mechanical regulators such as valves or variable height reservoirs. Furthermore, their physical meaning is independent of the network. The matrices $J$, $C_i$, and $D_i$ contain information about boundary conditions of the individual system, such as continuity or energy conservation; that is, they provide constraints for systems $P_i$. They also provide information about the topological structure of the system (what is connected to what). The PETSc *FieldSplit* preconditioner can take advantage of this structure by employing customized preconditioners for a group of $P_i$'s that represent the same physical components. Section 4.1.1 provides an example of how DMNetwork eases the application of the *FieldSplit* preconditioner.

Domain decomposition can be considered under various contexts:

— computer-based;
— network-based; or
— physics-based.

PETSc *domain decomposition* preconditioners (for example, block Jacobi and the overlapping additive Schwarz method [Balay et al. 2019]) support computer-based domain decomposition. These preconditioners distribute a computing domain, a network here, to multiple computer processors for parallel execution.

If DMNetwork is created by more than one subnetwork, each representing a local region of the entire network, then a network-based domain decomposition happens naturally. Users can query an individual subnetwork for customized operation or observation.

DMNetwork is developed by targeting a network of networks with subneworks arising from different physics. DMNetwork is intended to enable field scientists to independently develop their own model and write their physics-based function evaluations, facilitate an integration of the field developments into a composite multiphysics model, and then build the PETSc solvers on its top. PETSc solvers, such as KSP (linear), SNES (nonlinear), and TS (time-stepper), can be split and composed at all levels of the network and physics. Below, we provide a simple example to demonstrate these capabilities.

*Example: A Network of a Power-Grid Subnetwork and a Hydraulic Subnetwork* [2]

In this example, we build a network composed of two subnetworks: a power-grid subnetwork obtained from [Zimmerman et al. 2011] (see Power Subnetwork in Fig. 6), and a hydraulic subnetwork from EPANET (see Water Subnetwork in Fig. 6).

Two subnetworks are coupled by an edge linking a power load in the power-grid subnetwork to a water pipe junction in the hydraulic subnetwork. We treat the coupling as a third subnetwork, as indicated by the dashed line in Fig. 6.

Let

$$X = \begin{bmatrix} X_{power} \\ X_{water} \end{bmatrix}, \quad F(X) = \begin{bmatrix} F_{power}(X_{power}) \\ F_{water}(X_{water}) \\ F_{couple}(X) \end{bmatrix}. \tag{5}$$

The nonlinear mathematical system we wish to solve is

$$F(X) = 0. \tag{6}$$

[2] See *petsc / src / snes / examples / tutorials / network / ex1.c* [Balay et al. 2019].

We create three nonlinear solver objects: SNES, SNES_power, and SNES_water. SNES solves the coupled Equation (6). SNES_power solves

$$\begin{bmatrix} F_{power}(X_{power}) \\ X_{water} - X_{water_{old}} \end{bmatrix} = 0\,, \tag{7}$$

and SNES_water solves

$$\begin{bmatrix} F_{water}(X_{water}) \\ X_{power} - X_{power_{old}} \end{bmatrix} = 0\,, \tag{8}$$

where $X_{power_{old}}$ and $X_{water_{old}}$ represent the initial guess or previous approximation of $X_{power}$ and $X_{water}$, respectively. Equations (7) and (8) are the power grid and hydraulic subsystems using the global solution vector $X$, but they update only their own entries represented by $X_{power}$ and $X_{water}$.

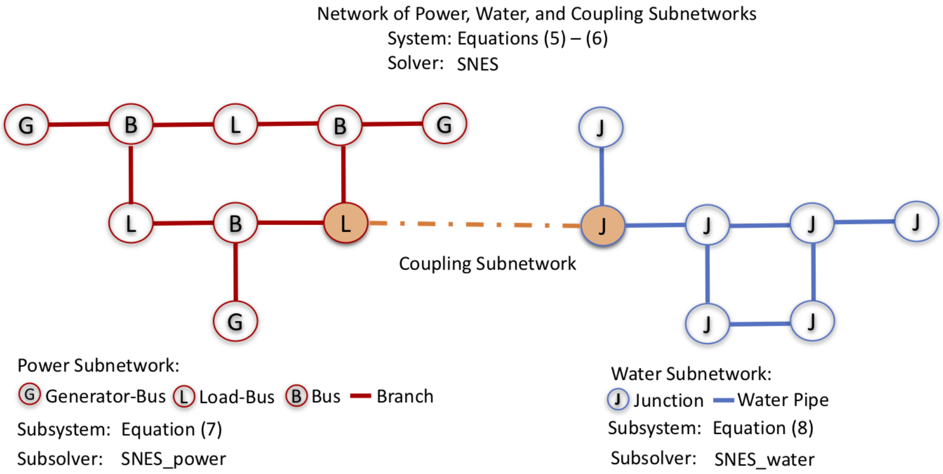


Fig. 6. Network of subnetworks and the associated solvers in *ex1.c*.

Figure 6 shows the detailed layout of the networks, associated systems and the solvers in the example. Below are its computational steps.

(1) Read data for the electric power grid, water pipe, and coupling subnetworks.
(2) Create a (single composite) DMNetwork:
- Register power, water, and coupling components in the DMNetwork;
- Add edge connectivity for all subnetworks;
- Set up the network layout;
- Retrieve subnetworks; add components and variables;
- Set up and then distribute the DMNetwork to multiple processes.

(3) Set up solvers:
- Create nonlinear solvers SNES, SNES_power, and SNES_water;
- Write subphysics function/Jacobian routines in the subdirectories: (a) power/pf-functions.c (*Func_power()*); (b) water/waterfunctions.c (*Func_water()*);
- Write the top-level function routine that calls *Func_power()* and *Func_water()*.

(4) Solve:
while (it $<$ it_max and !converge)

— $k1$ iterations of SNES_power (update $X_{power}$);
— $k2$ iterations of SNES_water (update $X_{water}$);
— $k3$ iterations of coupled composite system for X.

Running this code with run-time options, the users can experiment with domain decomposition and solver composition/spliting at various levels of physics and computation without modifying ex1.c. For example, for the overall coupled system (6), we use options

```
-coupled_snes_fd
-coupled_ksp_type gmres
-coupled_pc_type bjacobi
-coupled_sub_pc_type lu
-coupled_sub_pc_factor_mat_ordering_type qmd
```

That is, we use the finite-difference Jacobian approximation with the coloring scheme introduced in Section 2.4 for the Newton iterations and GMRES Krylov iteration with block Jacobi as preconditioner. Each block uses LU direct factorization with QMD matrix ordering. For the power subsystem (7), we call the user-provided subroutine for analytic Jacobian evaluation and use options

```
-power_pc_type asm
-power_sub_pc_type lu
-power_sub_pc_factor_mat_ordering_type qmd
```

for its inner-linear iterations, namely, the additive Schwarz method with subdomain overlapping preconditioner [Abhyankar et al. 2011]. Similarly, various options can be applied to the hydraulic subsystem (8).

## 4. HYDRAULIC TRANSIENT NETWORK SIMULATIONS ON EXTREME-SCALE COMPUTERS

Hydraulic transient network simulations are performed for problems such as water distribution, oil distribution, and hydraulic generation. In this section, we demonstrate the scalability of the DMNetwork through two hydraulic transient simulations with millions to billions of variables on extreme-scale computers.

### 4.1. Water Pipe Network

We focus on the water transients on closed conduits such as an urban distribution system. Simulation of hydraulic transients involves the calculation of pressure changes induced by a sudden change of velocity of the fluid [Chaudhry 1979; Wylie and Streeter 1978]. These velocity changes create a pressure wave that propagates proportionally to the speed of sound in the fluid media and the friction of the conduits. The modeling of this problem involves the solution of a set of PDEs. The disturbance of the system is introduced through a perturbation on the boundary conditions and results in a stiff differential equation, traditionally solved by the method of characteristics.

To facilitate discussion, we introduce the following notation:

— $nv$, $ne$: number of junctions (vertices) and pipes (edges) of the network;
— $Q^k$, $H^k$: water flow and pressure for pipe $k$;
— $Q_i$, $H_i$: boundary values of water flow and pressure adjacent to junction $i$;
— $ne_i$: number of connected pipes at junction $i$.

For pipe $k$, the water flow and pressure are described by the momentum and continuity equations:

$$\frac{\partial Q^k}{\partial t} + gA\frac{\partial H^k}{\partial x} + RQ^k \left|Q^k\right| = 0\,, \tag{9}$$

$$gA\frac{\partial H^k}{\partial t} + a^2\frac{\partial Q^k}{\partial x} = 0\,, \tag{10}$$

where $g$ is the gravity constant, $A$ is the area of the conduit, $R = \frac{f}{2DA}$ with $f$ being the friction of the conduit and $D$ its diameter, and $a$ is the velocity of the pressure wave in the conduit. At an interior junction $i$, $ne_i > 1$, the boundary conditions satisfy

$$\sum_{j=1}^{ne_i} Q_i^{k_j} = 0\,, \tag{11}$$

$$H_i^{k_j} - H_i^{k_1} = 0, \quad j = 2 \ldots ne_i\,. \tag{12}$$

Equations (9)–(12) are built over a network or a network of subnetworks. Note that the physical meaning of these equations corresponds to conservation of energy and mass. Special boundaries such as the connection to a reservoir, valve, or pump provide application-specific boundary conditions.

*4.1.1. Steady State.* Systems engineers customarily divide the simulation of a dynamic system into two stages: steady state and transient state. Most systems are assumed to operate in steady-state conditions until some disturbance perturbs the system. In steady state the magnitudes do not vary with time. Thus Equations (9)–(10) are reduced to

$$gA\frac{\partial H^k}{\partial x} + RQ^k \left|Q^k\right| = 0\,, \tag{13}$$

$$\frac{\partial Q^k}{\partial x} = 0\,. \tag{14}$$

We can make two assertions for steady state: (1) along an individual pipe, the flow $Q^k$ is constant, and (2) the drop of pressure $H^k$ can be described with a linear function. Hence values (Q, H) inside a pipe can be uniquely determined by their boundary values, which satisfy the interior junction equations (11)–(12) and special boundary conditions. Equations (11) and (12) represent global connectivity of the network. Adding special boundary conditions, they form an algebraic differential subsystem that we call the *junction subsystem*. We name the rest of the system the *pipe subsystem*. The junction subsystem has an ill-conditioned Jacobian matrix in general; its size is determined by the numbers of vertices, edges, and the network layout but is independent of the level of refinement used within the pipes for the differential equations (13)–(14). Thus it forms a small, but difficult to solve, subsystem requiring a strong preconditioner for its solution.

The Newton-Krylov method [Kelley 2003] is used to solve the nonlinear steady-state problem. For its linear iterations, we use the *FieldSplit* preconditioner to extract the junction equations from the entire system, apply a direct linear solver (e.g., MUMPS parallel LU solver [Amestoy et al. 2001]), and use block Jacobi with ILU(0) in each subblock of the Jacobian for the rest of the system.

To use the *FieldSplit* preconditioner, we must provide the index set for the junctions. DMNetwork facilitates an easy collection of the indices: iterate over the local vertices, query Junction components, and retrieve variable offsets (see Section 2.3).

The command-line options for this execution are as follows.

- Preconditioner for entire system:

```
-initsol_pc_type fieldsplit
```

- Preconditioner for junction subsystem:

```
-initsol_fieldsplit_junction_pc_type lu
-initsol_fieldsplit_junction_pc_factor_mat_solver_type mumps
```

- Preconditioner for pipe subsystem:

```
-initsol_fieldsplit_pipe_pc_type bjacobi
-initsol_fieldsplit_pipe_sub_pc_type ilu
```

The prefix *initsol* is used to distinguish the steady-state system from the transient system, which is solved next.

*4.1.2. Transient State.* In the transient state we calculate the pressure wave that arises after a perturbation is applied to the steady-state solution. We solve Equations (9)-(12), which are a set of hyperbolic partial differential and algebraic equations. We have simulated a case appearing in [Wylie and Streeter 1978, p. 38] to benchmark the accuracy of our solution. The case consists of a single pipe connected to a reservoir and a valve. At time $t = 0^+$ the valve is closed instantaneously, creating a pressure wave that propagates through the pipe back and forward. In Figure 7, we plot the pressure profiles (hydraulic or piezometric head, in water column meters) for a set of equally spaced points along the water pipe. In particular, the dark blue curve in the figure represents the pressure at the water reservoir. The water reservoir is treated as a constant pressure source, and hence the pressure is constant through time. At the other extreme, in light blue, we can see the pressure wave next to the valve, which sharply increases after its closure. The point right next to the valve (in yellow) will not be perturbed until the pressure wave has reached it, at a time that is proportional to the speed of the wave. This example provides physical intuition of the importance of the Courant number in computing hyperbolic PDEs. Several methods to solve such systems are described in the literature. In this section we describe two of the most common ones.

(1) Method of characteristics
The method of characteristics [Chaudhry 2014] involves applying a change of variables to Equations (9)–(10). Taking (10), scaling it by a term $\lambda$, and adding it to (9), we obtain

$$(\frac{\partial Q^k}{\partial t} + \lambda a^2 \frac{\partial Q^k}{\partial x}) + \lambda g A(\frac{\partial H^k}{\partial t} + \frac{1}{\lambda}\frac{\partial H^k}{\partial x}) + RQ^k \left|Q^k\right| = 0\,. \tag{15}$$

Using the chain rule, we obtain

$$\frac{dQ}{dt} = \frac{\partial Q}{\partial t} + \frac{\partial Q}{\partial x}\frac{dx}{dt}\,, \tag{16}$$

and

$$\frac{dH}{dt} = \frac{\partial H}{\partial t} + \frac{\partial H}{\partial x}\frac{dx}{dt}\,. \tag{17}$$

By defining

$$\frac{1}{\lambda} = \frac{dx}{dt} = \lambda a^2\,, \tag{18}$$

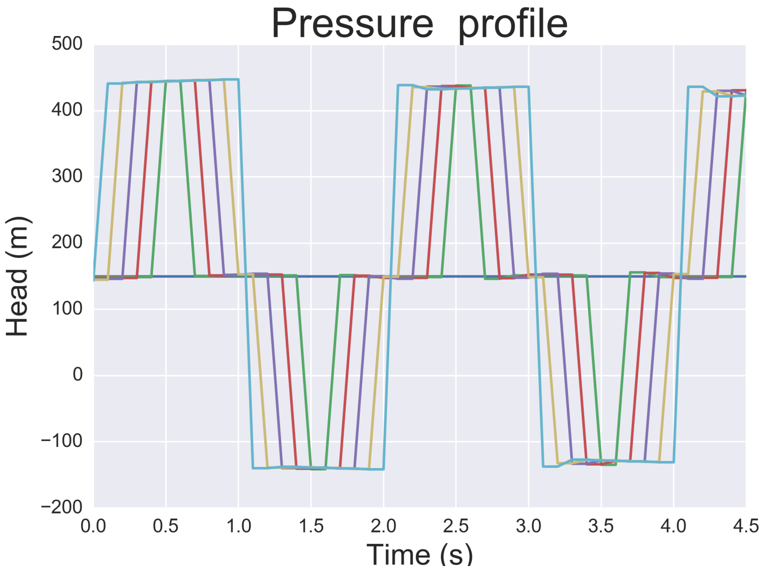


Fig. 7. Pressure wave on a conduit. This pressure wave is created in a single pipe with a conduit, when the end of the conduit is instantaneously closed. The vertical axis measures the piezometric head in water column meters, and each different color shows the pressure of a point in the pipe.

that is $\lambda = \pm\frac{1}{a}$, we obtain the ODEs in which the independent variable $x$ has been eliminated:

$$\frac{dQ}{dt} \pm \frac{gA}{a}\frac{dH}{dt} + RQ\,|Q| = 0\,. \tag{19}$$

We use the characteristic equations at the boundaries.

(2) Lax scheme
An alternative to the method of characteristics is the Lax scheme [Chaudhry 2014], an explicit first-order scheme. We approximate the partial derivatives as

$$\frac{\partial H}{\partial t} = \frac{H_i^{j+1} - \bar{H}_i}{\Delta t}, \qquad \frac{\partial Q}{\partial t} = \frac{Q_i^{j+1} - \bar{Q}_i}{\Delta t}\,, \tag{20}$$

$$\frac{\partial H}{\partial x} = \frac{H_{i+1}^j - H_{i-1}^j}{2\Delta x}, \qquad \frac{\partial Q}{\partial x} = \frac{Q_{i+1}^j - Q_{i-1}^j}{2\Delta x}\,, \tag{21}$$

$$\bar{H}_i = 0.5(H_{i-1}^j + H_{i+1}^j), \qquad \bar{Q}_i = 0.5(Q_{i-1}^j + Q_{i+1}^j)\,. \tag{22}$$

We apply these equations to the interior points.

*4.1.3. Experimental Results.* We conducted experiments on two computer systems: Cetus, an IBM Blue Gene/Q supercomputer in the Argonne Leadership Computing Facility [ALCF 2018a], and Edison, a Cray XC30 system in the National Energy Research Scientific Computing Center [NERSC 2017]. Cetus has 4,096 nodes, each with 16 1600 MHz PowerPC A2 cores with 16 GB RAM per node, resulting in a total of 65,536 cores. Edison has 5,576 compute nodes. A node has 64 GB of memory and two sockets, each with a 12-core Intel processor at 2.4 GHz, giving a total of 133,824 cores.

Our test network was obtained from [de Corte and Sörensen 2014] in EPANET format [Rossman 2000]. It consists of 926 vertices and 1,109 edges of various lengths (see Fig. 8). Equations (9)–(12) were discretized via the finite volume method into a sys-

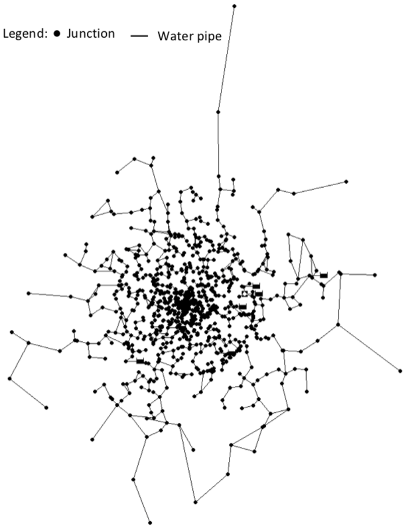


Fig. 8. Water network provided by [de Corte and Sörensen 2014].

tem over this single network with a total of 3,949,792 variables. We call the resulting DMNetwork *sub-dmnetwork*.

In Table I we compare the performance and strong scalability of the LU and *FieldSplit* preconditioners for solving the steady-state system (13)–(14) on this sub-dmnetwork. Its result will be used to construct an initial solution for the transient-state system. As discussed in Section 4.1.1, most execution time is spent in the LU factorization. Hence, when we use *FieldSplit* employing LU for the small junction subdomain and block Jacobi on the larger pipe subdomain, computation time is greatly reduced.

Table I. Execution Time of Steady State on Edison (sec)

| No. of Cores | LU | *FieldSplit* |
|---|---|---|
| 24 | 52.0 | 3.91 |
| 48 | 45.9 | 3.84 |
| 72 | 42.3 | 2.92 |

The steady-state solution was compared with the EPANET software by using a benchmark case from [de Corte and Sörensen 2014] consisting of 74 junctions, 102 pipes, and one reservoir. The solution of the problem was consistent with EPANET's solution, minding the differences in the approach.

We then built a large network by duplicating *sub-dmnetworks* and composing them into a composite *dmnetwork*. Artificial edges were added between the *sub-dmnetworks* without introducing new variables over these edges. To perform weak-scaling[3] studies of the simulation when the number of *sub-dmnetworks* was doubled, we increased the number of processor cores proportionally.

The current implementation of DMNetwork reads the user input data file and sets up the initial DMNetwork data structure on processor zero (see Sec. 2.2); then the

[3] See *https://en.wikipedia.org/wiki/Scalability#Weak_versus_strong_scaling.*

structure is distributed to the multiple processes for building a parallel DMNetwork, which is used for the simulation.

DMNetwork setup and the initial steady-state solution computation (13)–(14) occur only once. The focus of our simulation is the repeated time integration of the transient-state system. All the experiments were done by using the PETSc DAE solver, that is, the Lax scheme for each pipe and backward Euler time integration for the entire composite network system (9)–(12). The Jacobian matrix was approximated by finite differencing with coloring, as discussed in Section 2.4, and the Krylov GMRES iterations with selected preconditioner were implemented for the linear solves. Since the Jacobian matrix structure does not change with the time, the computing time spent at each time step remains approximately constant. We run only 10 time steps for each test case.

Tables II and III show the scalability of the simulation on Cetus and Edison. The linear solvers dominate the computation with efficiency determined by the selected preconditioners. Among all the preconditioners provided by PETSc, we found the block Jacobi and the additive Schwarz method (ASM) with subdomain overlapping 1 (ov.1) and 2 (ov.2) to be the most efficient for our application. In the tables, we compare these three preconditioners using the total simulation time and cumulative number of linear iterations (given in parentheses).

Table II. Execution Time of Transient State on Cetus (sec)

| No. of | Variables | Linear Preconditioner | | |
|---|---|---|---|---|
| Cores | (in millions) | Block Jacobi | ASM | ASM |
| | | | ov. 1 | ov. 2 |
| 256 | 16 | 60.0 (43) | 50.5 (24) | 45.3 (20) |
| 1,024 | 63 | 63.4 (49) | 50.6 (24) | 45.4 (20) |
| 4,096 | 253 | 86.1 (54) | 72.8 (34) | 58.7 (20) |
| 16,384* | 1,012 | 94.1 (54) | 81.2 (34) | 65.3 (20) |

* We set the number of cores per node to 8 (instead of 16) to double the memory available per core.

Table III. Execution Time of Transient State on Edison (sec)

| No. of | Variables | Maximum Variables | Linear Preconditioner | | |
|---|---|---|---|---|---|
| Cores | | per Core | Block Jacobi | ASM | ASM |
| | (in millions) | (in thousands) | | ov. 1 | ov. 2 |
| 240 | 16 | 106 | 9.9 (48) | 7.3 (25) | 6.4 (20) |
| 960 | 63 | 106 | 10.6 (55) | 7.0 (24) | 6.2 (20) |
| 3,840 | 253 | 106 | 10.4 (53) | 7.3 (24) | 6.7 (20) |
| 15,360 | 1,012 | 104 | 11.9 (53) | 11.4 (26) | 9.9 (20) |
| 30,720 | 2,023 | 117 | 20.0 (53) | 17.6 (26) | 17.2 (20) |

The water pipe network has pipes of various lengths that give rise to pipes with varying degrees of freedom. Since each pipe is restricted to a single process, some job imbalance results. As the number of cores increases, the job imbalance worsens, as indicated by the maximum number of variables per core in column 3 of Table III. This imbalance affects the scaling with increased number of cores.

To investigate this, we then doubled the number of *sub-dmnetworks* for each test and listed the results in Table IV. As the work pool increases for each core, the job balance as well as the scalability improve.

Table IV. Execution Time of Transient State on Edison (sec)

| No. of Cores | Variables (in millions) | Maximum Variables per Core (in thousands) | Linear Preconditioner | | |
|---|---|---|---|---|---|
| | | | Block Jacobi | ASM ov. 1 | ASM ov. 2 |
| 240 | 32 | 151 | 14.1 (40) | 11.8 (24) | 10.4 (20) |
| 960 | 126 | 152 | 14.5 (47) | 11.1 (24) | 10.1 (20) |
| 3,840 | 506 | 157 | 16.2 (50) | 12.1 (24) | 11.2 (20) |
| 15,360 | 2,023 | 162 | 18.7 (50) | 15.7 (24) | 16.9 (20) |

Table V. Execution Time of Residual and Jacobian Evaluation on Edison (sec)

| No. of Cores | Variables (in millions) | Residual Function | Jacobian Matrix |
|---|---|---|---|
| 240 | 16 | 0.9 (7 %) | 0.9 (9%) |
| 960 | 63 | 0.9 (6 %) | 1.0 (9 %) |
| 3,840 | 253 | 0.9 (7 %) | 1.0 (9 %) |
| 15,360 | 1,012 | 1.4 (6 %) | 1.5 (12 %) |
| 30,720 | 2,023 | 2.8 (5 %) | 3.1 (14 %) |

Table V shows the weak scalability of the residual function evaluation and Jacobian evaluation using the matrix-coloring scheme developed for DMNetwork as described in Section 2.4. The data is taken from the cases using the block Jacobi preconditioner. Other cases give similar scalabilities. Along with the total execution time spent on these evaluations, we list their percentage of total time in the transient-state simulation.

Because of the initial sequential setup of DMNetwork, the size of applications with which we can experiment is limited by the local memory of the computer systems. Cetus has 1 GB or 2 GB of memory per core when 16 cores or 8 cores are used per node, respectively; and Edison has 2.67 GB of memory per core. The largest problems we are able to run on these machines are approximately 1 billion variables using 16,384 cores on Cetus and 2 billion variables using 30,720 cores on Edison, respectively. Parallel DMNetwork is under active development and will be released this year.

The results demonstrate that, using DMNetwork, we can achieve weak scalability for the simulation with 2 billion variables using up to 30,000 cores. Overall, for most test cases, ASM, with an overlap of 2, is the fastest with the fewest number of linear iterations.

### 4.2. Mississippi River Network

Applying DMNetwork to a real application network is our ongoing project. This section reports preliminary results in simulating the Mississippi river network.

We start with the shallow water equations, which consist of conservation of mass equation (23) and momentum equation (24). The equations describe the flow behavior in open channels, such as river, canal, and pipe networks [Abbott and Basco 1989; Te Chow 1959; Guinot 2012]. The assumptions for the equations include that the water is incompressible, transverse and vertical accelerations are negligible, the flow regime is turbulent, and the change in bed slope is very small. A single river is modeled as

$$\frac{\partial H}{\partial t} + \frac{\partial (HU)}{\partial x} = 0\,, \tag{23}$$

$$\frac{\partial (HU)}{\partial t} + \frac{\partial (HU^2 + \frac{1}{2}gH^2)}{\partial x} = gH(S_b - S_f)\,, \tag{24}$$

where $H$ is flow depth, $U$ is flow velocity, $g$ is the gravity constant, $S_b$ is the bed slope, and $S_f$ is the bed friction. Let $Q = HU$, Equations (11)-(12) are used to model the junctions of the rivers in the network.

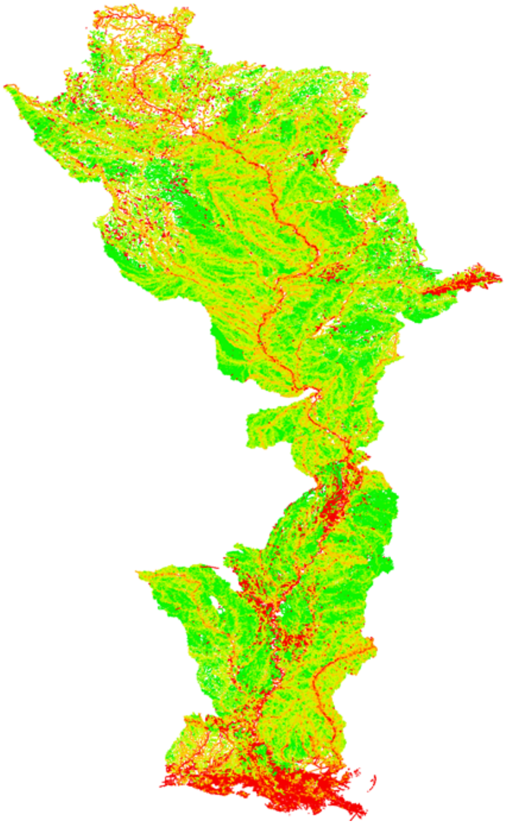


Fig. 9. Mississippi river network.

DMNetwork is applied to simulate river flow in the Mississippi river [Betrie et al. 2018], which is the largest river in North America. The Mississippi river system consists of one-eighth of the total river segements in the conterminous United States [Kammerer 1987]. The data for the river network and the physical properties for each river segement (e.g., length, width, slope, flow depth and velocity) are obtained from the NHDPlus dataset [McKay et al. 2012].

Figure 9 depicts the Mississippi river network modeled with 263,531 edges and 256,437 vertices. A finite volume method is used to discretize Equations (23)-(24) with a total of 28,894,804 variables for the network.

Numerical simulation is conducted on the Theta supercomputer at Argonne Leadership Computing Facility [ALCF 2018b]. The strong-scaling[4] result depicted in the Fig. 10 shows that successive doubling the number of processor cores (from 64 to 128, 256, and 512) decreases the computation time by approximately half each time. We note

[4] See *https://en.wikipedia.org/wiki/Scalability#Weak_versus_strong_scaling.*

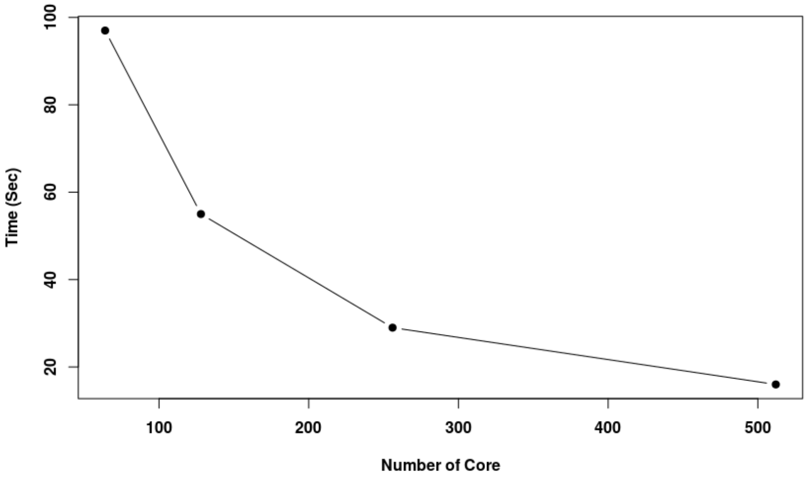


Fig. 10. Strong-scaling result of the Mississippi river network simulation.

that the scaling result from this study is satisfactory compared with the Routing Application for Parallel Computation of Discharge model [David et al. 2011] that scales up to 16 cores for the simulation of the upper Mississippi river.

## 5. CONCLUSIONS AND FUTURE WORK

This paper introduces DMNetwork, a new package in PETSc for the simulation of large networked PDE-based multiphysics applications. DMNetwork enables scientists to build physics models independently, assemble their models into an overall system, and then apply PETSc hierarchical and composable solvers to it on extreme-scale computers.

The design and interfaces of DMNetwork enable users to switch between various solvers with minimal effort and offer an effective test base for network-structured applications. DMNetwork greatly simplifies programming parallel code to solve potentially complicated network multiphysics problems.

Early users of DMNetwork reported satisfactory results for power flow simulations [Werner et al. 2019; Rinaldo and Ceresoli 2018]. Applications presented in this paper include an electric circuit, a network of a power-grid subnetwork and a hydraulic subnetwork, the Mississippi river network, and a composite water pipe network.

Numerical experiments for the large-scale water pipe network show the robustness and the scalability of the PETSc solvers with DMNetwork on computers using up to 30,000 processor cores.

Our future work will include parallel creation of DMNetwork, coupling of distinct physics, and multiscale-time-step integration over extremely large networks.

## ACKNOWLEDGMENTS

We thank Mathew Knepley for his help in designing and implementing DMNetwork. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under Contract DE-AC02-06CH11357. This paper was initiated under Argonne's Resilient Infrastructure Initiative and Laboratory Directed Research and Development (LDRD) project number 2017-088-R1. This research used resources of the National Energy Research Scientific Computing Center, a DOE

Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

## REFERENCES

Michael Barry Abbott and David R Basco. 1989. *Computational fluid dynamics: an introduction for engineers*. Longman Pub Group.

Shrirang Abhyankar, Barry Smith, Hong Zhang, and A. Flueck. 2011. Using PETSc to develop scalable applications for next-generation power grid. In *Proceedings of the 1st International Workshop on High Performance Computing, Networking and Analytics for the Power Grid*. ACM. http://www.mcs.anl.gov/uploads/cels/papers/P1957-0911.pdf

ALCF. 2018a. Cetus supercomputer. https://www.alcf.anl.gov/cetus-and-vesta. (2018).

ALCF. 2018b. Theta supercomputer. https://www.alcf.anl.gov/theta. (2018).

Patrick R. Amestoy, Ian S. Duff, Jean-Yves L'Excellent, and Jacko Koster. 2001. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal. Appl.* 23, 1 (2001), 15–41.

Modelica Association. 2017. Modelica web page. (2017). https://www.modelica.org/.

Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Todd Munson, Karl Rupp, Patrick Sanan, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. 2019. *PETSc Users Manual: Revision 3.12*. Technical Report ANL-95/11 - Rev 3.12. Argonne National Laboratory.

Andreas Bartel and Michael Günther. 2018. PDAEs in refined electrical network modeling. *SIAM Rev.* 60, 1 (jan 2018), 56–91. DOI:http://dx.doi.org/10.1137/17M1113643

Getnet Betrie, Hong Zhang, Barry Simith, and Eugene Yan. 2018. A scalable river network simulator for extereme scale computers using the PETSc library. In *AGU Fall Meeting*.

Jed Brown, Matthew G. Knepley, David A. May, Lois C. McInnes, and Barry F. Smith. 2012. Composable linear solvers for multiphysics. In *Proceeedings of the 11th International Symposium on Parallel and Distributed Computing (ISPDC 2012)*. IEEE Computer Society, 55–62. DOI:http://dx.doi.org/10.1109/ISPDC.2012.16

M. Hanif Chaudhry. 1979. *Applied hydraulic transients*. Van Nostrand Reinhold Company.

M. Hanif Chaudhry. 2014. *Applied hydraulic transients, 3rd edition*. Springer.

Thomas F. Coleman and Jorge J. Moré. 1983. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM J. Numer. Anal.* 20, 1 (Feb. 1983), 187–209.

Cédric H David, David R Maidment, Guo-Yue Niu, Zong-Liang Yang, Florence Habets, and Victor Eijkhout. 2011. River network routing on the NHDPlus dataset. *Journal of Hydrometeorology* 12, 5 (2011), 913–934.

Annelies de Corte and Kenneth Sörensen. 2014. HydroGen: an artificial water distribution network generator. *Water Resources Management* 28, 2 (2014), 333–350.

FMI Standard Development team. 2017. FMI: Functional mock-up interface. (2017). https://fmi-standard.org/

Vincent Guinot. 2012. *Wave propagation in fluids: models and numerical techniques*. John Wiley & Sons.

Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2008. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008), Gaël Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA)*. 11–15.

HELICS Development Team. 2017. HELICS web page. (2017). https://github.com/GMLC-TDC/HELICS-src

Bruce Hendrickson and Robert Leland. 1995. A multilevel algorithm for partitioning graphs. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (CDROM)*. ACM Press, New York, 28. DOI:http://dx.doi.org/10.1145/224170.224228

National Instruments. 2017. LabVIEW web page. (2017). http://www.ni.com/labview/.

Jordan Jalving, Shrirang Abhyankar, Kibaek Kim, Mark Herald, and Victor Zavala. 2017. A graph-based computational framework for simulation and optimization of coupled infrastructure networks. *IET Generation, Transmission, and Distribution* 11, 12 (2017), 3163–3176.

John C Kammerer. 1987. *Largest rivers in the United States (water fact sheet)*. Technical Report. U.S. Geological Survey,.

George Karypis and V. Kumar. 1997. *ParMETIS: Parallel graph partitioning and sparse matrix ordering library*. Technical Report 97-060. Department of Computer Science, University of Minnesota. http://www.cs.umn.edu/ metis.

Carl T. Kelley. 2003. *Solving nonlinear equations with Newton's method*. SIAM.

Michael Lange, Lawrence Mitchell, Matthew G. Knepley, and Gerard J. Gorman. 2016. Efficient mesh management in Firedrake using PETSc-DMPlex. *SIAM Journal on Scientific Computing* 38, 5 (2016), S143–S155. http://epubs.siam.org/doi/abs/10.1137/15M1026092

Jure Leskovec and Rok Sosič. 2016. SNAP: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1.

Mathworks. 2017. SIMULINK web page. (2017). https://www.mathworks.com/products/simulink.html.

L McKay, T Bondelid, T Dewald, J Johnston, R Moore, and A Rea. 2012. NHDPlus Version 2: user guide. *National Operational Hydrologic Remote Sensing Center, Washington, DC* (2012).

NERSC. 2017. Edison supercomputer. https://www.nersc.gov/users/computational-systems/edison/. (2017).

NetworKit Development Team. 2017. NetworKit web page. (2017). http://network-analysis.info

NetworkX Development Team. 2018. NetworkX web page. (2018). https://networkx.github.io/

OpSim Development Team. 2017. OpSim: Test and Simulation Environment for grid control and aggregation strategies. (2017). https://www.iee.fraunhofer.de/en/schnelleinstieg-wirtschaft/themen/opsim-homepage.html

Stefano Guido Rinaldo and Andrea Ceresoli. 2018. *Newton–Krylov–Schwarz Methods for Distributed Power Flow and Related Applications*. Master's thesis. Politecnico di Milano.

Lewis A. Rossman. 2000. *EPANET 2 Users Manual*. Technical Report. Water Supply and Water Resources Division, National Risk Management Research Laboratory, Cincinnati, OH 45268.

Amit Singer and Mihai Cucuringu. 2010. Uniqueness of low-rank matrix completion by rigidity theory. *SIAM J. Matrix Anal. Appl.* 31, 4 (Jan 2010), 1621–1641. DOI: http://dx.doi.org/10.1137/090750688

Barry Smith, Lois Curfman McInnes, Emil Constantinescu, Mark Adams, Satish Balay, Jed Brown, Matthew Knepley, and Hong Zhang. 2012. *PETSc's software strategy for the design space of composable extreme-scale solvers*. Preprint ANL/MCS-P2059-0312. Argonne National Laboratory. DOE Exascale Research Conference, April 16-18, 2012, Portland, OR.

Barry F. Smith and Xumin Tu. 2013. *Encyclopedia of Applied and Computational Mathematics*. Springer, Chapter Domain Decomposition.

Gilbert Strang. 2007. *Computational science and engineering*. Wellesley-Cambridge Press.

Ven Te Chow. 1959. *Open-channel hydraulics*. McGraw-Hill.

Alex Werner, Kapil Duwadi, Nicholas Stegmeier, Timothy M. Hansen, and Jung-Han Kimn. 2019. Parallel implementation of AC optimal power flow and time-constrained optimal power flow using high-performance computing. In *IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC 2019)*. Best Paper in Track.

E. Benjamin Wylie and Victor L. Streeter. 1978. *Fluid transients*. McGraw-Hill Book Company.

Ray D. Zimmerman, Carlos E. Murillo-Sánchez, and Robert J. Thomas. 2011. MATPOWER: Steady-state operations, planning and analysis tools for power systems research and education. *IEEE Transactions on Power Systems* 26, 1 (2011), 12–19. DOI: http://dx.doi.org/10.1109/TPWRS.2010.2051168

**Disclaimer.** The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan. http://energy.gov/downloads/doe-public-access-plan.