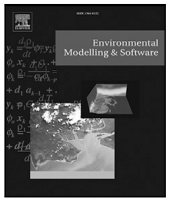
Environmental Modelling and Software 158 (2022) 105554

Contents lists available at ScienceDirect

Environmental Modelling and Software

journal homepage: www.elsevier.com/locate/envsoft



Position Paper

PTSNet: A Parallel Transient Simulator for Water Transport Networks based on vectorization and distributed computing

Gerardo Riaño-Briceño, Ben R. Hodges, Lina Sela *

Department of Civil, Architectural and Environmental Engineering, the University of Texas at Austin, TX 78712, US

ARTICLE INFO

Dataset link: https://github.com/gandresr/PTSNET

Keywords:
Transient flow
Networked water systems
Parallel computing
High-performance computing
Message Passing Interface

ABSTRACT

Modeling transient flow in water transport networks (WTNs) is characterized by hyperbolic partial differential equations. Existing commercial and open-source software for transient modeling in WTNs have limitations, such as lack of scalability and compatibility with high-performance computers, difficulty to systematically execute simulations and analyze results. This work proposes a novel open-source Python package that relies on vectorization and distributed computing to overcome the limitations of existing software. The proposed library, the Parallel Transient Simulator for Water Networks (PTSNet), surpasses in computational performance existing modeling tools and incorporates novel analytics functionalities. PTSNet has been tested on WTNs composed of tens, hundreds, and thousands of hydraulic elements using a personal and a supercomputer, running from tens to hundreds of processors. We show through rigorous analyses that PTSNet is scalable, accurate, and significantly speeds up simulations with sufficiently dense numerical grids.

Software availability

Name of software: PTSNet
Software requirements: Python 3.6, 3.7, 3.8, 3.9 and Python dependencies (i.e., NumPy, tqdm, scipy, h5py, pandas, wntr, mpi4py, numba)
Availability: PTSNet source codes are available from a GitHub repository at https://github.com/gandresr/PTSNET

1. Introduction

Modeling hydraulic processes in environmental systems is critical for management, operation, and control of water transport networks (WTN) in natural and urban environments. Particularly, modeling transient scenarios, such as the surcharge of water mains in drainage systems, or sudden pressure changes in water distribution systems due to the operation of pumps, is essential to prevent failures that may result in high economic losses (Boulos et al., 2005). Failures attributed to rapid flow accelerations and drastic changes in pressure include floods, system fatigue or pipe ruptures, pump and device malfunctioning, and intrusion of pathogens and contaminants through leaks, cracks, and other defects (Boulos et al., 2005; LeChevallier et al., 2003; Fox et al., 2014). This paper focuses on modeling transients in WTNs, which involves solving partial differential equations (PDEs) that describe the water-transport phenomenon in terms of conservation of mass and momentum relationships. PDEs for transient flow modeling can be solved using a variety of numerical methods, including the method of characteristics (MOC) (Wylie et al., 1993; Nault et al., 2018), finite difference methods (Chaudhry and Hussaini, 1985; Kiuchi, 1994; Blanco et al., 2015; Verdugo et al., 2019; Cao et al., 2020), or finite volume methods (Zhao and Ghidaoui, 2004; Castro et al., 2006; Fernández-Pato and García-Navarro, 2014; Mesgari Sohani and Ghidaoui, 2019). Overall, MOC has been predominantly used over other numerical schemes due to its ease of implementation and accuracy. However, since most WTNs are composed of thousands of hydraulic elements and cover extensive areas, modeling transient phenomena for realistic WTNs can be intractable with current simulation tools.

Recent advances in computational fluid dynamics demonstrate that distributed computing and vectorization can significantly speed up computations for modeling water transport in environmental and urban systems. In Lin and Zhang (2021), the authors propose a methodology that divides hydrologic and hydraulic zones for a 2D flood simulation, computing the hydrodynamic behavior of the zones across multiple processors. The model runs 10 times faster than traditional sequential implementations, yet sacrifices accuracy for computational performance. Similarly, in Burger et al. (2014), the authors present a parallel version of the EPA's storm water management model (SWMM) that runs six to ten times faster than sequential SWMM on a twelve-processor system. In Carlotto et al. (2021), the authors developed SW2D-GPU, an open-source 2D shallow water model that parallelizes computations across multiple processing units in a general-purpose

* Corresponding author.
E-mail addresses: griano@utexas.edu (G. Riaño-Briceño), hodges@utexas.edu (B.R. Hodges), linasela@utexas.edu (L. Sela).

https://doi.org/10.1016/j.envsoft.2022.105554
Received 10 August 2022; Received in revised form 28 September 2022; Accepted 10 October 2022
Available online 21 October 2022
1364-8152/

graphic processing units (GPGPUs), achieving 36 times faster running times. Similarly, Anguita et al. (2015) proposes a 3D semi-implicit hydrodynamic model for shallow water that can be executed in parallel. In all the previous works, the improvement in computational efficiency is typically greatest when running highly-dense numerical grids, which are sometimes necessary to capture nonlinearities in large-scale simulations (Abhyankar et al., 2020). Likewise, there is a generalized adoption of partitioning algorithms and distributed computing to accelerate models in water systems (Zhu et al., 2019; Xu et al., 2021; Tiernan and Hodges, 2022). For example, there have been significant advancements in the parallelization of shallow water equations, which are essential for open-flow modeling, e.g., river and drainage modeling, speeding up simulations several orders of magnitude (Carlotto et al., 2021; Burger et al., 2014). However, fewer works have focused on studying the problem of parallelization to solve transient flow in pressurized WTNs.

In the context of urban water systems, packages for transient simulation in WTNs include commercial software, such as Bentley Hammer (Bentley Systems, 2019), KYPipe (KYPipe, 2019), InfoSurge (Innovyze, 2019), and TransAM (McInnis and Karney, 1998). Commercial packages share similar functionalities and are of limited use for large-scale applications. Commercial packages come with limitations in computational efficiency, such as lack of scalability and compatibility with high-performance computers. Moreover, the usability of commercial packages is limited since commercial codes are delivered to users as black boxes and, thus, do not provide opportunities for improving state-of-the-art models, modifying or extending its functionalities, nor running transient simulations systematically, i.e., through program commands rather than interacting with a graphical user interface (GUI). Therefore, it is difficult to extract, analyze, and visualize numerical results.

On the other hand, open-source packages can be categorized as transient-focused and general-purpose. Transient-focused packages, which solve the specific system of hyperbolic PDEs that models transient flow in WTNs, include TSNet (Xing and Sela, 2020) and the free-open-source implementation of MOC presented in Kjerrumgaard Jensen et al. (2018). Transient-focused packages are computationally inefficient with respect to commercial packages when modeling medium- and large-size WTNs. Conversely, general-purpose packages allow users to solve a systems of hyperbolic PDEs defined by the user. Some of the available general-purpose packages include Modelica (Fritzson, 2011), OpenFOAM (Jasak et al., 2007), FEniCSx (Scroggs et al., 2021), and PETSc's DMNetwork (Abhyankar et al., 2020). General-purpose packages incorporate state-of-the-art methods to run distributed computations across multiple processors, but are typically based on implicit numerical schemes whose solution is found via adjoint or iterative methods, e.g., conjugate gradients, which have proven to be computationally inefficient when running in parallel for inherently sparse systems such as WTNs (Burger et al., 2016; Abhyankar et al., 2020). More critically, the usability of general-purpose packages can be cumbersome for users that need to run transient simulations of WTNs, because users need to manually define the topology of the WTN and its properties to formulate the system of PDEs and its boundary conditions. Defining simulation parameters for a large-scale simulation can be highly time-consuming and prone to errors given the multitude of boundary conditions that are present in WTNs.

This study presents and demonstrates a new open-source software package – the Parallel Transient Simulation in Networks (PTSNet) – that addresses the limitations of computational efficiency and code usability of both commercial and open-source packages. For computational efficiency we build upon our DV-MOC work in Riaño-Briceño et al. (2021a), which uses vectorization and distributed computing for efficient computation. In Riaño-Briceño et al. (2021a) we demonstrate: (1) a new vectorized formulation of the transient-flow equations, which is specifically designed to fit an efficient memory allocation scheme and (2) a parallel algorithm that distributes the load among processors to further speed up vectorization and adopts to general WTNs topologies and many boundary conditions. The numerical solution uses an explicit time-marching scheme to reduce the computational overhead of parallel communications. The readers are referred to Riaño-Briceño et al. (2021a) for the theoretical derivations.

This paper contributes to the: (1) definition of the software architecture to support the novel vectorized and parallel scheme for general WTNs, and (2) development of new analytics tools, such as automated input file processing, selection of the best time step and number of processors for a transient simulation running in parallel. The code is written in Python using OpenMPI and HDF5 libraries that are compatible with high-performance computing systems. For code usability, we equip PTSNet with functionalities to automatically extract the WTN properties from standardized input files, i.e., the EPANET .inp file (Rossman, 1994), which have been widely adopted in both industry and academia (Sela and Housh, 2019; Sela et al., 2019). In addition, PTSNet contains a set of functionalities that allows users to extract and analyze simulation data, both quantitatively and visually, facilitating parameter selection and the storage, retrieval, and analysis of results.

The remainder of the paper is organized as follows: Section 2 provides an overview of the DV-MOC transient flow model and PTSNet functionalities. Section 3 describes the underlying architecture of the library and its main data structures for users interested in building new functions. Sections 4 and 5 focus on explaining model setup and execution through code snippets. In Section 6, the package is validated for modeling medium- and large-size networks, by comparing simulation results with widely accepted computational packages for transient modeling. Additionally, PTSNet's performance is compared with that of its counterparts, showing that the DV-MOC offers significantly better simulation times and extends the capabilities of currently available software. Finally, in Section 7, conclusions and future directions are provided.

2. Software description

PTSNet is an open-source Python package capable of computing the dynamics of flowrates $Q(x,t)$ and hydraulic heads $H(x,t)$ in time (t) and space (x) for a pressurized WTN under the effect of transient scenarios, such as valve opening and closure, pump start-up and shut-off, and bursts. Modeling WTN dynamics in PTSNet requires defining the WTN properties using the industry-standard EPANET .inp format defined by Rossman (1994). Input files can be generated systematically using code scripts based on the WNTR package (Klise et al., 2018) or via EPANET's GUI (Rossman, 1994). PTSNet relies on other open-source Python packages that need to be installed through a conda environment following the instructions provided in PTSNet's repository (Riaño-Briceño et al., 2022). In particular, we chose the WNTR package over other available libraries that can systematically extract information from general text files, because WNTR is supported and maintained by the US Environmental Protection Agency (EPA) that is responsible for defining the .inp file standard. Moreover, we use `mpi4py` to handle the open message passing interface (OpenMPI) standard, which enables distributed programming and is broadly used in industry and academia (Graham et al., 2005). Once all the dependencies are installed, PTSNet's installation is done via the PyPi repository using the `pip install ptsnet` terminal command, as described in Riaño-Briceño et al. (2022). We note that, since PTSNet is continuously updated, by using the previous command, users will install the latest version of the package. To install the version presented in this paper, users should use: `pip install ptsnet==0.1.7`.

PTSNet's functionalities can be categorized into four distinct groups: (1) Model Setup, (2) Execution of Simulation, (3) Extraction of Results, and (4) Analytics, as illustrated in Fig. 1. The first three groups of functionalities allow users to run a transient simulation using one or multiple processors and create workspaces to save simulation results. The fourth group are functionalities to help with model setup and results analysis. The remainder of this section presents a description of the underlying transient flow model and describes PTSNet's functionalities.

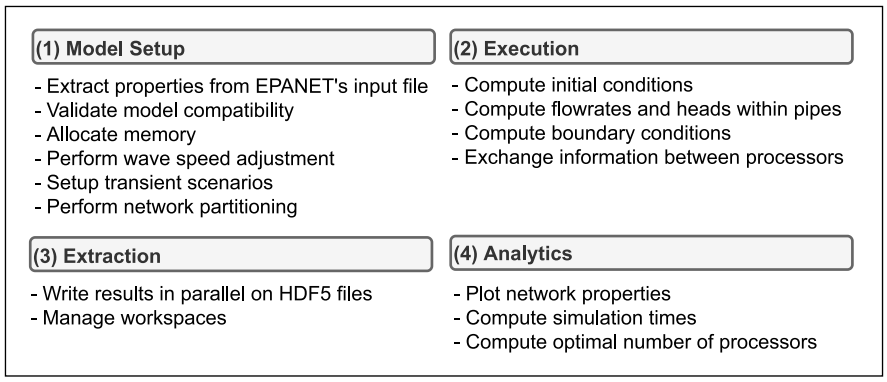


Fig. 1. Main functionalities of the PTSNet library.

2.1. Transient flow model

Transient flow within pipes is characterized by a system of hyperbolic PDEs that describes conservation of mass and momentum relationships (Wylie et al., 1993). These equations are prognostic since they predict the value of flowrates and hydraulic heads for some time in the future based on current-time conditions (Holton, 1973). When pipes are connected to other WTN elements such as valves, pumps, leaks, bursts, surge protections, reservoirs, dead-ends, and demands, transient flow is modeled by coupling prognostic and diagnostic equations, the latter being equations that link flowrates and/or heads at identical times (Holton, 1973). PTSNet is based on the distributed and vectorized method of characteristics (DV-MOC) of Riaño-Briceño et al. (2021a), which has advantages for accuracy, ease of implementation, and suitability for large-scale simulations with significantly lower computational times than sequential implementations. Details of the DV-MOC method are found in Riaño-Briceño et al. (2021a), with an abbreviated description provided below. The DV-MOC time-marching scheme can be executed by one or multiple processors to solve conservation of mass and momentum equations (Wylie et al., 1993):

$$\frac{\partial H(x,t)}{\partial t} + \frac{\omega^2}{ga}\frac{\partial Q(x,t)}{\partial x} = 0, \tag{1}$$

$$\frac{\partial H(x,t)}{\partial x} + \frac{1}{ga}\frac{\partial Q(x,t)}{\partial t} + F(x,t) = 0, \tag{2}$$

where H and Q represent head and flowrate respectively, a is the cross-section area of a pipe, ω is the pressure wave celerity, referred to as wave speed from now on, and g is the gravitational acceleration. The term $F(x,t)$ in Eq. (2) represents the friction head loss per unit length, estimated for steady, quasi-steady, and unsteady conditions (Nault et al., 2018). In this work, we solve Eq. (2) using a steady friction model where $F(x,t) = |Q(x,t)|Q(x,t)f\big/\left(2gda^2\right)$, d is the pipe diameter, and f is the Darcy–Weisbach dimensionless friction factor for pipes. Derivations of prognostic Eqs. (1) and (2) and their numerical solution can be found in Riaño-Briceño et al. (2021a) and Riaño-Briceño et al. (2021b). The formulation of diagnostic and prognostic equations for hydraulic devices (e.g., valves, pumps) is described in Section S1 of the Supporting Information (SI).

The DV-MOC numerical scheme is guaranteed to be linearly stable for model time steps satisfying the Courant–Friedrichs–Lewy (CFL) condition (Courant et al., 1967). This behavior is achieved by discretizing longer pipes lengths into segments and adjusting the local wave speed values for the global time step to ensure that the CFL number remains less than 1 for every pipe segment in the network. This local adjustment of wave speed is a standard approach for MOC solutions (Wylie et al., 1993).

The discrete versions of Eqs. (1) and (2) are solved at the "solution points", which can be classified as interior and boundary points depending on their association with the WTN elements. While interior points are those within pipes, boundary points are those where the coupling between Eqs. (1) and (2) with boundary conditions occur (Riaño-Briceño et al., 2021a). To illustrate, Fig. 2 shows a WTN formed by several pipes in series connecting an upstream reservoir through a pump to supply a downstream demand regulated by a valve. The flow moves across the x axis from left to right. Downstream the pump there is a surge protection device and upstream the valve there is a leak. The pipes of the WTN in Fig. 2 are discretized into pipe segments of length δ, forming interior points marked with black dots, and boundary points associated with the reservoir, general junction, valve, pump, and surge protection device, marked at the ends of the pipes with icons that differentiate the types of the boundary points. The relationship between the system and its model abstraction, as illustrated in Fig. 2, is useful in understanding how simulation settings (time step, wave speed, and the δ segment length) will affect both the computational time and model error. Smaller time steps, faster wave speeds, and smaller pipe segments will provide a more precise solution, but at the cost of increased computational time. We further discuss these settings in Section 4.

2.2. Functionalities

Using PTSNet involves using the functionalities presented in Fig. 1 to set up a transient model, determine simulation settings, such as time step and number of processors, execute a parallel simulation, and extract results. A simulation can be initialized using *Model Setup* functionalities, which extract information from the input file, allocate memory for the model, compute initial conditions, define transient scenarios, and run a compatibility check. The compatibility check validates that the WTN meets PTSNet's connectivity requirements as follows: valves and pumps must be connected to a single pipe upstream and downstream; reservoirs can only be connected to valves or pumps through a pipe; no leaks or demands are allowed at valves or pumps; demand values cannot be negative; and valves cannot have zero head loss if their initial flowrate is greater than zero. As part of the model setup, users must define the number of processors and the simulation time step either manually or using *Analytics* functionalities, as described in Sections 5.1 and 5.2. Once the model is set up, users run a PTSNet simulation, read and plot results after completion using *Execution* and *Extraction* functionalities, respectively. Note that *Model Setup* and *Execution* functionalities are invoked in the same Python script, and separate scripts can be used to invoke *Extraction* and *Analytics* functionalities. The next section describes the software architecture and its underlying data structures, followed by demonstration of PTSNet functionalities through snippets of code.

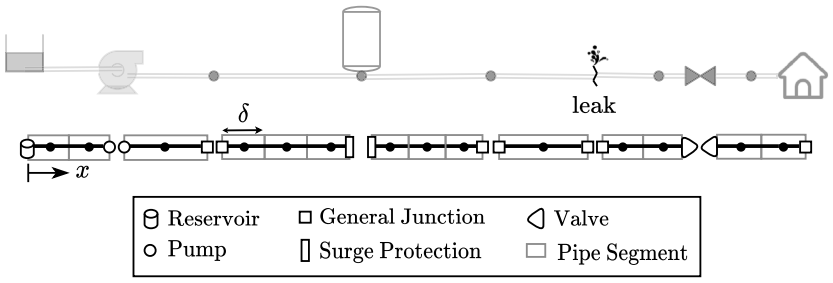


Fig. 2. Discretization of a water transport network. The system schematic is shown on top and its discretization below.

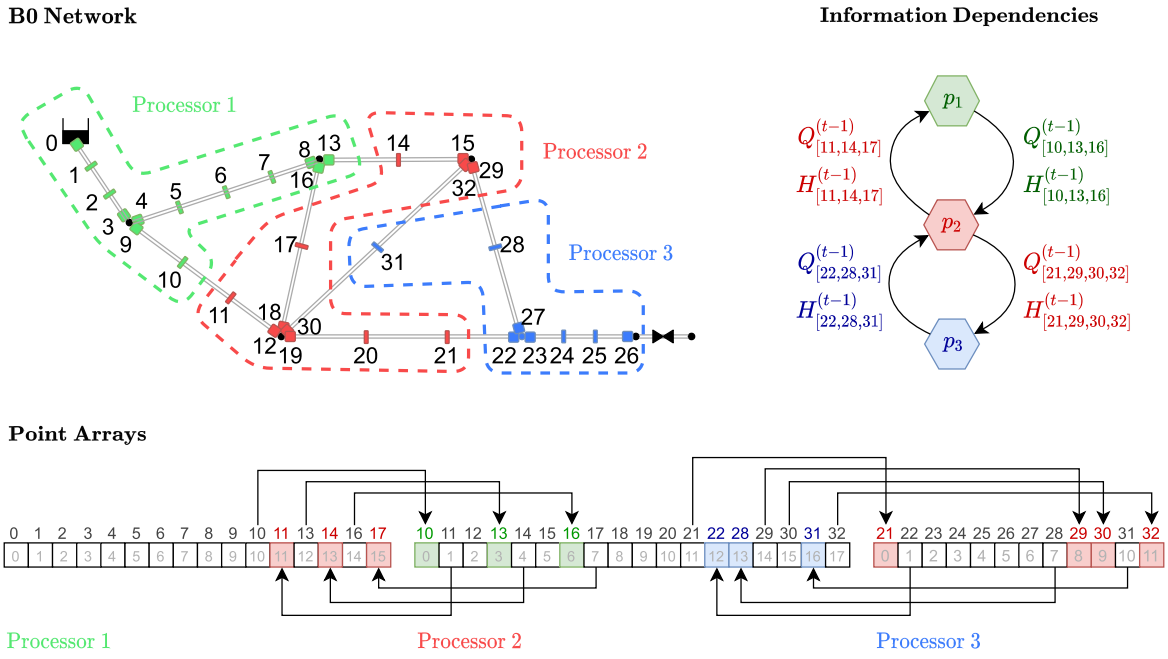


Fig. 3. Data partitioning for the small-scale system.

3. Software architecture

This section describes the underlying mechanisms of vectorization and distributed parallel computing that run within PTSNet, including specifications of the data structures that enable these mechanisms. Using the small-scale example described below and presented in Fig. 3, we show how vector operations take place when solving Eqs. (1) and (2) numerically, how tasks are divided between processors, and how information is exchanged between processors to compute an MOC step. Furthermore, we describe how simulation files are stored and organized within workspaces, and how array data structures are stored in memory using parallel HDF5.

Example (*Small-scale System*). The small-scale test case presented on the top left corner of Fig. 3 comprises a reservoir, nine pipes, five general junctions, and a valve that regulates water demand downstream. As described in Section 2.1, each pipe in the system is divided into segments, such that heads and flowrates are computed at solution points, which are enumerated from 0 to 32. Boundary points at the extreme of the pipes are marked with squares, and interior points within the pipes are marked with ticks. For illustration purposes, the network data is partitioned among three processors (Fig. 3 bottom), thus creating information dependencies between processors (Fig. 3 top right).

3.1. Vectorization

Modern central processing units (CPUs) have single instruction, multiple data (SIMD) functionalities (Grama et al., 2003), which allow simultaneous computations of basic operations, such as addition and multiplication. This capability, referred to as vectorization, is executed when a loop is replaced in the code by an SIMD instruction so that instead of computing a single operation for a single solution point at every time step, multiple solution points can be processed simultaneously in a fraction of the time (Grama et al., 2003). The first step towards vectorization consists of expressing the numerical solution of Eqs. (1) and (2) in terms of vector operations. To illustrate, we show how flowrates are computed for a partition of the small-scale example. Based on the DV-MOC framework, flowrates at interior points located at x and time (t) are computed based on the previous solution in time, at time (t) minus the time step τ, as follows:

$$Q_x^{(t)} = \underbrace{\frac{W_x^{(t-\tau)} - H_x^{(t)}}{U_x^{(t-\tau)}}}_{C^+} = \underbrace{\frac{H_x^{(t)} - D_x^{(t-\tau)}}{B_x^{(t-\tau)}}}_{C^-} = \frac{W_x^{(t-\tau)} - D_x^{(t-\tau)}}{B_x^{(t-\tau)} + U_x^{(t-\tau)}}, \tag{3}$$

where

$$B_x^{(t)} = \underbrace{\frac{\omega}{ga}}_{\bar{B}_x} + \underbrace{\frac{f\delta}{2gda^2}}_{\bar{R}_x} \text{abs}\left(Q_{x+\delta}^{(t)}\right), \qquad D_x^{(t)} = H_{x+\delta}^{(t)} - \bar{B}_x Q_{x+\delta}^{(t)},$$

$$U_x^{(t)} = \bar{B}_x + \bar{R}_x \text{abs}\left(Q_{x-\delta}^{(t)}\right), \qquad W_x^{(t)} = H_{x-\delta}^{(t)} + \bar{B}_x Q_{x-\delta}^{(t)}.$$

Sequential

```
for i in [22, 23, 24, 25, 26, 27, 28, 31]:
    Q[i] = (W[i] - D[i]) / (B[i] + U[i])
```

$$\frac{W_{22}-D_{22}}{B_{22}+U_{22}} \quad \frac{W_{23}-D_{23}}{B_{23}+U_{23}} \quad \frac{W_{24}-D_{24}}{B_{24}+U_{24}} \quad \frac{W_{25}-D_{25}}{B_{25}+U_{25}} \quad \frac{W_{26}-D_{26}}{B_{26}+U_{26}} \quad \frac{W_{27}-D_{27}}{B_{27}+U_{27}} \quad \frac{W_{28}-D_{28}}{B_{28}+U_{28}} \quad \frac{W_{31}-D_{31}}{B_{31}+U_{31}}$$

Vectorized

```
idx = [22, 23, 24, 25, 26, 27, 28, 31]    # Interior Points
Q[idx] = (W[idx] - D[idx]) / (B[idx] + U[idx])
```

$$\frac{[W_{22},W_{23},W_{24},W_{25},W_{26},W_{27},W_{28},W_{31}]^\top - [D_{22},D_{23},D_{24},D_{25},D_{26},D_{27},D_{28},D_{31}]^\top}{[B_{22},B_{23},B_{24},B_{25},B_{26},B_{27},B_{28},B_{31}]^\top + [U_{22},U_{23},U_{24},U_{25},U_{26},U_{27},U_{28},U_{31}]^\top}$$

Fig. 4. Comparison between sequential an vectorized code for the computation of flowrates by processor 3 for the small-scale system (see Fig. 3).

The information associated with each interior point is stored in custom vector data structures that inherit the properties of a NumPy array. In addition, we store the indexes associated with interior points on a separate array referred to as selector. In the case of the small-scale example, if processor 3 is in charge of computing the points in the network that are marked in blue, the selector of interior points for processor 3 is given by $[22, 23, 24, 25, 26, 27, 28, 31]^\top$. Once the selector is defined, Eq. (3) can be solved either with sequential or vectorized programming, as shown in Fig. 4. When the sequential code is executed in a CPU with eight pipelines, the computation of flowrate vector entries takes eight iterations (one iteration per pipeline, sequentially), whereas the vectorized computation takes a single iteration (eight pipelines computed simultaneously). Vectorization is executed via NumPy's fancy indexing functionality, as shown in Fig. 4, to create new vectors with contiguously allocated data of W, B, D, and U based on the corresponding selector. Once the vectors are created, vector operations take place and flowrates and heads are updated. Note that the number of simultaneous operations is limited and depends on the architecture of the CPU. For example, for Intel's Xeon Phi x200 (Knights Landing) CPUs, which we used to produce some of the results presented in Section 6, the maximum number of simultaneous fused multiply-add operations per cycle is eight when using 64-bit floating-point data (McCalpin, 2022).

One of the main challenges of implementing vectorization and making it practical for efficient numerical operations for networked systems is the proper definition of selectors. Since vector operations are invoked recurrently within PTSNet, we decided to pre-compute and pre-allocate all the necessary selectors to replace any for-loop with array instructions. By doing so, we avoid allocating and deallocating selectors at every iteration. This is possible given that the structure of the numerical grid remains constant throughout the simulation. Table S1 in the SI lists the selectors for points and nodes that can be accessed via the `PTSNetSimulation.worker.where` dictionary. The `PTSNetSimulation.worker.where` data structure is broadly used within the `simulation.funcs` module, as well as within the `results` module, facilitating not only the computations of results, but also the extraction of results.

Data that are used in vectorized operations are stored within a `PTSNETTable`. The `PTSNETTable` was designed to allow data extraction either by label or by a collection of indexes. Users will mostly interact with two types of `PTSNETTable` data structures, i.e., system states and results. System states refer to a collection of tables that store properties of different system elements. Specifically, these are nodes, pipes, valves, pumps, open and closed surge tanks. Each element type is associated with a specific set of states, which can be constant or variable, as defined in the source code (see the `simulation.constants` module). Table properties are stored in ordered arrays such that every position in the array is associated with a specific element label. Since element labels cannot be used to extract information out of ordered arrays in Python, tables internally translate element labels into array indexes, as shown in Fig. 5. Table indexing is done through a Python dictionary that operates as a hash function $f : \mathbb{K} \to \mathbb{Z}_+$ taking a label $k \in \mathbb{K}$ and mapping it to a positive index. The associated collections of states are stored within the `PTSNetSimulation.ss` dictionary. In order to extract information about system states, users need to use a special syntax. To illustrate, reading the initial head (`ihead`) at a specific node involves calling the `PTSNETTable`, extracting the `ihead` property, and using a numerical index or a label to extract the value, as shown in Fig. 5. In the example, the initial head at the node JUNCTION-1 is extracted.

3.2. Distributed parallel computing

PTSNet leverages the computational power of multiple processors using distributed parallel computing. In practice, the overall MOC problem is divided in smaller parts, such that every processor concurrently computes a smaller vectorized MOC problem. When distributed parallel computing is used, each processor has its own private memory making it necessary to exchange information among processors. Once information is passed, processors perform computations independently from each other, spending different amounts of time to run a single time step. Therefore, processors need to synchronize with each other at the end of each time step to ensure that all the solutions of heads and flowrates at time t are available to compute the solutions at time $t + \tau$. Hence, given k processors the networked numerical grid must be partitioned such that the time spent in communication between processors is minimized and the load on each processor is balanced. In general, we assume that boundary conditions can only be computed by one processor, and because of this, boundary points belonging to the same boundary condition are assigned to a single processor. For example, in Fig. 3 (top left) we show the corresponding subgraphs that result from a three-way heuristic partitioning. The graph that represents

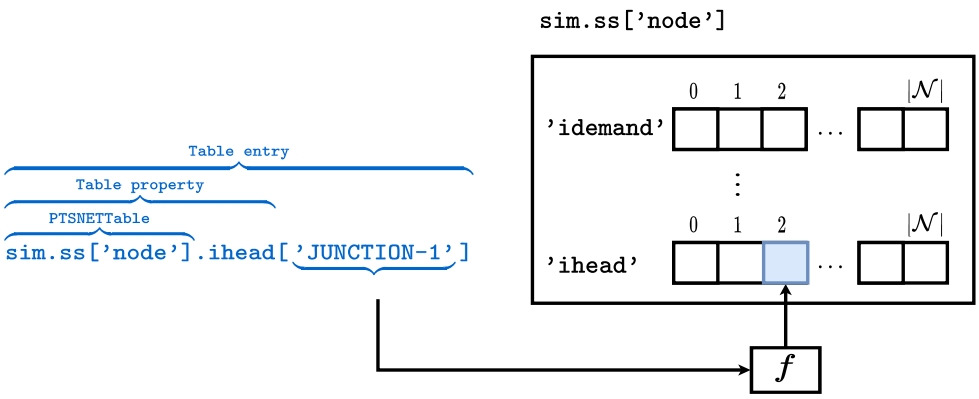


Fig. 5. The PTSNETTable data structure. Values can be extracted using labels, which are internally converted into numerical indexes.

the networked numerical grid consists of thirty-three points that are divided into three sub-graphs, each of them containing 16, 18, and 12 points respectively. Moreover, processors 1 and 2 exchange information from points 11, 14, and 17 coming from processor 2, and points 10, 13, and 16 coming from processor 1. Therefore, information dependencies are bidirectional given that processors not only receive information from their neighbors, but also send information to them.

The partitioning algorithm included within PTSNet achieves balance of work between processors by dividing the number of solution points into approximately equal parts. For example, creating a simulation with the small-scale network using a time step of 0.07 s produces a numerical grid with 33 solution points. This simulation is partitioned to be executed by three processors; hence each processor allocates memory for the points it is in charge of, plus the external information dependencies necessary to compute the MOC equations. Each point in the network receives a global index used to determine information dependencies and export results, and a local index to facilitate local vector operations and information retrieval. Local operations are those executed by an individual processor for a specific region of the WTN, e.g., computation of specific solution points. In contrast, global operations are those executed for the entire WTN by multiple processors, e.g., storing results. For the example presented in Fig. 3, notice that processor 1 is in charge of computing points 0–11, 13, and 16. However, to compute the solution of boundary points 8, 13, and 16, it is necessary to use information from solution points 7, 14, and 17. Therefore, processor 2 needs to communicate information from points 14 and 17 to processor 1, and processor 1 needs to allocate additional space for the incoming information. Indexes pointing to local spaces in memory for the incoming information dependencies are referred to as send-and-receive buffers and can be accessed through the `PTSNetSimulation.worker.send_buffer` and `PTSNetSimulation.worker.recv_buffer` statements. Buffers are used to build the information dependencies graph (IDG), which is a special data structure created with the OpenMPI standard that establishes connections between processors to exchange information. Based on the IDG, PTSNet builds communicator data structures that define groups of processors that need to exchange data. Groups of processors locally exchange data using the `neighbor_all_to_all` communication protocol (Grama et al., 2003).

In summary, each processor independently computes DV-MOC, as presented in Riaño-Briceño et al. (2021a). First, the program is initialized, and then computations of solution points are distributed among processors by performing the partitioning of the networked numerical grid. Once the partitioning is defined, the IDG is built in a distributed fashion, i.e., each processor receives a portion of the network and defines its dependency to other processors. Afterwards, all the processors start running their part of the vectorized step, synchronizing and exchanging information at the end of the time step. This process is repeated iteratively until the duration of the simulation is completed.

3.3. File management

Results are not centrally stored in either memory or written to a file as the simulation is executed. Instead, results on each processor are stored in memory that only that processor can access. However, PTSNet uses the HDF5 library to unify simulation results in a single binary file at the end of the simulation (Collette, 2013). This approach is more efficient than writing either a unified file or separate results files during runtime; however, it has the disadvantage that a model crash will generally produce no written output of results before the crash. The unified output file is concurrently processed and written using the functionalities of the HDF5 library (Collette, 2013). This single, standardized binary file facilitates read operations and reduces the complexity of parsing data when extracting results for analyses. At the end of a simulation, PTSNet stores the results within the `flowrate.h5`, `demand_flow.h5`, `head.h5`, and `leak_flow.h5` files as presented in Fig. 6. Other workspace files include initial conditions, profiler, and simulation properties. Note that within PTSNet jargon "initial conditions" refers to data extracted from the EPANET input file, but rearranged in PTSNet's data structures. "Profiler" files store information of running times for the initialization, computation of interior and boundary points, and communication between processors. Finally, "simulation properties" refers to all the constant values that are used throughout the simulation in order to compute the DV-MOC method. The files composing a workspace are locally stored in the directory from which the user executes the simulation.

Users extract data via PTSNet's *persistent* mode, which accesses: (i) initial conditions computed with EPANET, (ii) simulation times, (iii) flowrates at the start and end of pipes, (iv) demand and leak flowrates at nodes, and (iv) the head at every node in the network. Users do *not* have to directly manipulate these files; instead the workspace data is accessed through PTSNet objects whose information is internally loaded from the files — if the *persistent* mode is active. Using this approach, results are extracted invoking the `PTSNETSimulation` object; initial conditions are found within the system `PTSNETSimulation.ss` dictionary; and running times are stored in the `PTSNETSimulation.worker.profiler`. Note that information stored in `.pkl` files is internally used by PTSNet and stored in the `PTSNETSimulation`. The workspace folders can occupy a large block of memory during the simulation, depending on the size of the problem. Therefore, users should carefully consider how much simulation data to store in the workspace. PTSNet provides flexible control of storage through functionalities to read results, list workspace information, and delete a specific workspace. These functionalities are found within the `results.workspaces` module.

To allocate time series results, PTSNet uses the `PTSNETTable2D` data structure, which is an extension of a `PTSNETTable` with time series as table entries. Results are allocated within the `sim.results`

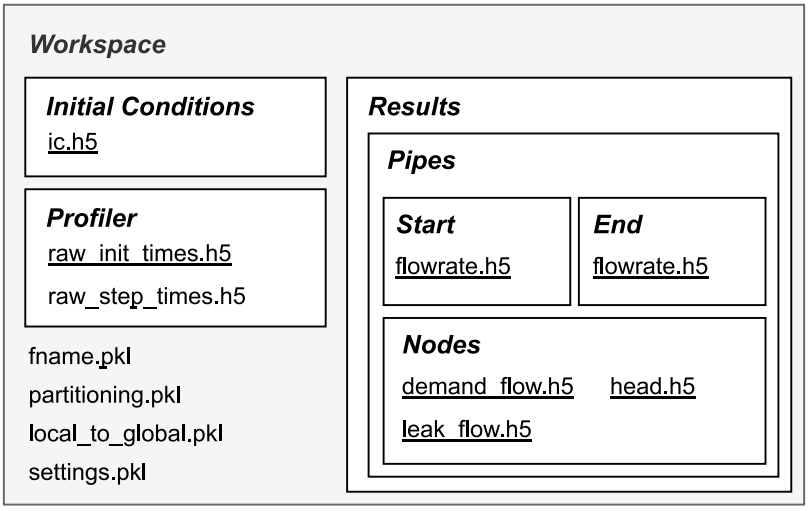


Fig. 6. PTSNet workspace file structure.

dictionary and then unified and updated at the end of the simulation through the `PTSNETSimulation` object. Internal manipulations simplify the syntax to extract results. For example, whenever `PTSNetSimulation['node']` is invoked, information is either loaded from RAM or disk, and extracted from the `PTSNetSimulation.results['node']` data structure. Results can be extracted for nodes and pipe extremes, i.e., pipe start and end segments. In the case of nodes, users can extract heads, leak and demand flowrates, and for pipes users can extract flowrates. Pipe flowrates will always be given with positive values. If users require the sign of the flowrate to match the flow convention of the EPANET model used as input, the flowrate time series can be multiplied by `sim.ss['pipe'].direction`.

Note that a PTSNet simulation can run out of memory if the results for all time steps and all solution points are stored. To minimize the use of memory resources and ensure scalability, the time-marching algorithm of PTSNet routinely stores point data for the minimum number of time steps necessary to advance in time (i.e, two consecutive time levels). These are referred to as the "memory pool of points" (`mem_pool_points`) and their data are stored in `PTSNETTable2D`. As the simulation advances PTSNet switches between table rows in order to compute the next time step, i.e., initial conditions at time $t = 0$ are stored in the first row of the table, then, a step is taken and the solution of time $t = \tau$ is stored on the second row. When the third step $t = 2\tau$ is computed, the solution is stored on the first row, using the solution of $t = \tau$. Therefore, accessing high-resolution point data at every time step requires manually extracting the necessary results from the "memory pool of points", making use of proper selectors, such as the ones described in Section S2 of the SI. Even though point data is only stored for two time steps, data at the extreme of pipes is available for every time step. Results at pipe extremes can be accessed either directly from RAM or from the HDF5 disk file via the `PTSNETSimulation`, as shown in Fig. 7.

4. Model setup

This section illustrates different use cases through snippets of code that exercise PTSNet's functionalities. These snippets can be used to model specific transient scenarios, extract results, and determine time step and number of processors. Additional example codes for using PTSNet are available in Riaño-Briceño et al. (2022). Setting up a transient model with PTSNet involves: (i) defining simulation settings in the form of a Python dictionary, (ii) creating transient model, and (iii) defining the transient scenario, as described next.

Defining simulation settings: PTSNet provides default settings (see lines 7 through 20 in Fig. 8). Users can adjust these values to increase the temporal and spatial resolution of the model, extend the duration of the simulation, set up wave speed values for each pipe, turn on and off secondary processes, such as displaying messages on the terminal, running a compatibility check, measuring simulation times, and saving results. Secondary processes (see lines 15 through 19 in Fig. 8) can also be turned off to reduce simulation times. For instance, once PTSNet determines that a WTN is compatible with the transient flow model, there is no need to run the compatibility check again if the same input file is executed multiple times. The temporal settings, such as the `time_step` and `duration`, should be given in seconds. PTSNet transient analysis initiates from a steady-state condition computed by EPANET. In the event that the EPANET .inp file includes an extended period simulation (EPS), i.e., the steady-state simulation spans multiple hours, users need to select the time of the steady-state condition to initialize the transient simulation. The `period` setting is the index associated with the time period within the EPS that will be used to calculate the initial condition for the transient simulation.

To define the wave speed values for pipes, users can either (i) set the same value for all pipes using the `default_wave_speed` setting, or (ii) define specific wave speed values using a text file containing pipe labels on the first column and wave speed values on the second. The disk path to access the text file with wave speed information must be defined via the `wave_speed_path` setting. Note that in any MOC method wave speed values are adjusted by the MOC algorithm by a factor ϕ, i.e., an adjusted wave speed $\hat{\omega}_i = \phi_i \omega_i$ is applied for each i pipe. Thus the user input values (or the default values) are merely starting points for the adjusted wave speeds. Three common wave speed adjustment methods are included in PTSNet: the `'user'` (Bentley Systems, 2022a), the `'critical'` (Wylie et al., 1993), and the `'optimal'` (Misiūnas, 2008) methods. Note that regardless of the method, at least two segments (n_i) are required for each pipe to obtain a valid numerical grid. The `'user'` method prioritizes the user's choice of simulation time step, τ, and sets the number of segments for the ith pipe as required by the MOC method for that τ; specifically, $n_i = \text{round}(\bar{n}_i)$ where $\bar{n}_i \equiv \ell_i/(\hat{\omega}_i \tau)$ and ℓ_i is the length of the pipe. The `'critical'` method allows the user to identify a *critical pipe*, i.e., the user (or default) wave speed values would require the smallest time step to meet the CFL condition. Then, the simulation time step is computed based on the critical pipe i as $\tau = \ell_i/(2\omega_i)$ with two segments, and the number of segments for all the other pipes in the WTN is computed as in the `'user'` method. For the `'user'` and the `'critical'` methods, the wave speed adjustment is given by $\phi_i = \bar{n}_i/n_i$. The `'optimal'` method simultaneously adjusts the

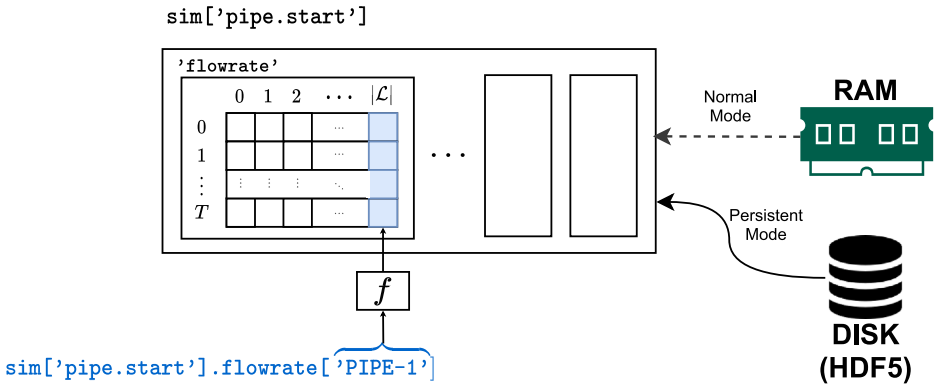


Fig. 7. The results data structure can be read from RAM or from an HDF5 file.

```python
from ptsnet.simulation.sim import PTSNETSimulation
from ptsnet.utils.io import get_example_path

# ------------------ (1) Model Setup -----------------

default_settings = {
    "time_step" : 0.01, # Simulation time step in [s]
    "duration" : 20, # Simulation duration in [s]
    "period" : 0, # Simulation period for EPS
    "default_wave_speed" : 1000, # Wave speed value for all pipes in [m/s]
    "wave_speed_file_path" : None, # Text file with wave speed values
    "delimiter" : ',', # Delimiter of text file with wave speed values
    "wave_speed_method" : 'optimal', # Wave speed adjustment method
    "save_results" : True, # Saves numerical results in HDF5 format
    "skip_compatibility_check" : False, # Dismisses compatibility check
    "show_progress" : False, # Shows progress (Warnings should be off)
    "profiler_on" : False, # Measures computational times of the simulation
    "warnings_on" : False, # Warnings are displayed if True
}

# Create a simulation
sim = PTSNETSimulation(workspace_name = "TNET3_SIM",
    inpfile = get_example_path('TNET3'),
    settings = default_settings
    # If settings are not defined, default settings are loaded automatically
)

# Define transient scenario
sim.define_valve_operation('VALVE-179',
    initial_setting=1, final_setting=0, start_time=0, end_time=1)
# sim.define_valve_settings('VALVE-179',
#   X=[start_time, end_time], Y=[initial_setting, final_setting])

# ------------------ (2) Execution -----------------

sim.run()
# while not sim.is_over():
#     sim.run_step()
```

Fig. 8. Code template with *Model Setup* and *Execution* functionalities.

wave speeds for all the pipes in WTN by solving the least squares problem. First, the number of pipe segments for each pipe are computed based on the smallest permissible time step that satisfies the CFL condition, i.e., $\tau = \min_i \ell_i/(2\omega_i)$. Second, the wave speed adjustments and the time step are computed by minimizing the sum of squared wave speed adjustments, which has an analytical solution based on the normal equation, i.e., $\boldsymbol{\phi}, \tau = \text{argmin}_{\boldsymbol{\phi},\tau} \left\{ \|\boldsymbol{\phi}\|^2 : \tau = \ell_i / \left(\omega_i \phi_i n_i\right) \right\}$, where $\boldsymbol{\phi}$ denotes the vector of wave speed adjustments for every pipe in the WTN. More details about the `'optimal'` method can be found in Misiūnas (2008). Users select the wave speed adjustment method using the `wave_speed_method` setting.

Creating transient model: Defining a transient model requires creating a `PTSNETSimulation` object, whose constructor function is imported in line 1 and executed in lines 22 through 27 in Fig. 8. When the `PTSNETSimulation` is created, PTSNet creates a workspace folder in the current working directory which is identified by the parameter `workspace_name`, as shown in line 22 in Fig. 8. While files associated with results of a specific simulation are allocated in the workspace folder, PTSNet internally allocates RAM memory for the simulation and extracts WTN properties from the input file. The path to the input file can be declared, as shown in line 24 in Fig. 8, either explicitly or using the function `get_example_path` to select one of the 12 examples included in the library.

Defining transient scenarios: After the `PTSNETSimulation` is created, users can define the transient scenario. Transient scenarios must be defined after the creation of the `PTSNETSimulation` object and before the simulation is executed, i.e., between lines 28 and 36 in Fig. 8. For example, in lines 30 and 31 in Fig. 8, a valve closure operation is defined using the `define_valve_operation` function, which takes as inputs the valve label, the initial and final settings, and the maneuver start and end times in seconds. Valve settings are defined as fractional opening from zero (closed) to one (fully open). Note that setting changes are linear from start to end times. Users requiring more control over behaviors can define custom operational maneuvers using time series, specifying setting values `Y` for specific times `X`, as shown in lines 32 and 33 in Fig. 8. However, time-series operations are slightly different than standard linear open/close operations in that a time series is interpreted as a step function, i.e., setting changes occur instantaneously at a given time rather than linearly between times.

Other transient scenarios that can be modeled with PTSNet include the operation of pumps, pipe bursts, leaks, variable demands, and surge protections. The commands to operate pumps (see Fig. 9(a)) are related to pump speed, and are defined similarly to valve settings. Pump setting equal to one represents a pump operating at its full initial speed, and zero means that the pump is off. Pump settings can be defined as linear functions with start and end times using the `define_pump_operation` command, or with time series with the `define_pump_settings` function, analogous to the `define_valve_settings` functionality. The operation of the pump will be determined based on the characteristic pump curve defined in the .inp input file, which is used as a diagnostic equation in the model. The equations for modeling transient scenarios are based on Riaño-Briceño et al. (2021a) and presented in Section S1 in the SI.

Users can also add a burst to the model using the `add_burst` command shown in Fig. 9(b). Bursts in PTSNet are modeled using the orifice equation with outflow $\hat{Q} = \kappa\sqrt{H - z}$, where κ is the time-varying discharge coefficient and z is the elevation at the location of the burst (Wylie et al., 1993). The outflow associated with a burst is computed by coupling the orifice equation with the general junction equations introduced in Riaño-Briceño et al. (2021a) (see Section S1.2 in the SI). The size of the orifice produced by the burst grows linearly between start and end times until it reaches the discharge coefficient defined in the `add_burst` command. Orifices with constant discharge coefficients, referred to as leaks, can also be modeled with PTSNet, yet they are not introduced through PTSNet commands. Instead, users must define leaks in the EPANET .inp input file before executing the simulation with PTSNet and adjust the value of the emitter coefficient in the input file (Rossman, 1994). PTSnet can also model variable demands, as shown in Fig. 9(c), by changing the discharge coefficient associated with the demand at a specific node. Variable demands need to be changed iteratively, thus it is necessary to execute the simulation using a while loop step by step.

Finally, surge protection devices can also be modeled in PTSNet, as shown in Fig. 9(d). Surge protection devices include open and closed surge tanks that absorb the wave shocks by compressing the air contained in the vessel. In general, open surge tanks are open to the atmosphere, have infinite storage capacity, and uniform cross-section area, which needs to be specified. When modeling open surge tanks, users need to specify the location and the cross-section area of the tank, as shown in line 2 of Fig. 9(d). Closed surge tanks are covered on top and have limited storage capacity. When modeling the closed surge tank, users need to specify the location of the tank, its cross-section area and height, and the initial water level inside the tank, as shown in line 4 of Fig. 9(d). The equations for surge tanks are based on Larock et al. (1999) and presented in Section S1.5 of the SI.

5. Model execution

A PTSNet simulation can be executed with one or multiple processors by saving a script similar to the one presented in Fig. 8, and then using the terminal command `mpiexec -n 2 python main_script.py`, where the `-n` flag is used to specify the number of processors that will execute the PTSNet simulation (in this case, two). Within the script the execution is triggered either by invoking the `sim.run()` command (as shown in lines 38 and 39 in Fig. 8) or using a `while` loop that advances the simulation step-by-step with the command `sim.run_step()`, as shown in line 6 in Fig. 9(c). With respect to the initial conditions, PTSNet always starts the WTN transient solution from a steady-state condition developed using EPANET with the users network configuration and initial settings. PTSNet automatically runs the EPANET model prior to the transient simulation without any prompting from the user. However, users should be aware that the EPANET steady-state solution has some limitations relative to the discretization of the DV-MOC network. In particular, since EPANET only provides heads at the extremes of pipes and a single flowrate per pipe, heads for the smaller pipe segments of DV-MOC are linearly interpolated from the pipe end values and a single flowrate is enforced along all segments of a single pipe. When running PTSNet in parallel, a single processor executes the EPANET simulation and broadcasts steady-state simulation results to the other processors. Once the processors receive the steady-state results from EPANET, processors interpolate initial heads and populate initial flowrate values for the pipe segments assigned to them only. After PTSNet has setup the initial conditions using EPANET, the DV-MOC transient simulation of heads and flowrates is computed from the user-specified start to end time. Note that results are only extracted by setting `save_results` to `True`, in which case results will be extracted at every time step an will be temporarily allocated in a data structure during the simulation. The results are converted and saved permanently in HDF5 files at the end of the simulation, as shown in line 14 in Fig. 8. Recall that HDF5 files are stored within a workspace folder whose name is defined when creating the `PTSNETSimulation` object, as done in line 22 in Fig. 8. The Jupyter notebook `1_simulate_scenarios` available on GitHub demonstrates how to setup, execute, and extract simulation results.

For *Analytics* functionalities that involve multiprocessing, such as those that allow time step selection and determining the number of processors, the execution requires a two-step process. First, the function is invoked using a single processor, then, PTSNet automatically generates a script that is used in step two for running a suite of simulations, and prints a line of code that needs to be executed in the terminal by the user. Finally, the user executes the command in

```
1  sim.define_pump_operation('PUMP-172', # Pump shut-off
2      initial_setting=1, final_setting=0, start_time=0, end_time=2)
```

(a)

```
1  sim.add_burst('JUNCTION-90', burst_coeff=0.02, start_time=0, end_time=1)
```

(b)

```
1  coeff = np.sin(np.linspace(0, 2*np.pi, 100))
2  ii = 0
3  while not sim.is_over:
4      if ii < 100:
5          sim.ic['node'].demand_coefficient['JUNCTION-23'] = coeff[i]
6      sim.run_step()
7      ii += 1
```

(c)

```
1  # Add surge protection
2  sim.add_surge_protection('JUNCTION-34', 'open', tank_area=0.1)
3  sim.add_surge_protection('JUNCTION-34', 'closed',
4      tank_area=0.1, tank_height=1, water_level=0.2)
```

(d)

Fig. 9. PTSNet commands to model: (a) pump shut-off scenario, (b) burst, (c) variable demand, and (d) open and closed surge tanks.

the terminal, which triggers the suite of simulations via the OpenMPI standard. The two-step execution process for *Analytics* functionalities is necessary in order to execute customized parallel simulations, given that simulations with multiple processors can only be executed through the command line. The Jupyter notebook `3_analytics` available on GitHub demonstrates the analytics functionalities, including analyzing simulation time step, number of processors, and wave speeds.

5.1. Time step selection

The simulation time step determines the resolution, accuracy, and computational burden of a PTSNet model. This is because the time step determines how much the wave speeds need to be adjusted to fulfill the CFL condition. Therefore, the wave speed adjustment, i.e., $|\hat{\omega}_i/\omega_i - 1|$, can be used as a proxy to estimate the effect of the time step selection on simulation results. PTSNet incorporates functionalities within the `graphics` module that facilitate the users to visualize the network topology and wave speed adjustments. More specifically, the `plot_wave_speed_error` functionality in the `graphics.static` module generates a plot of the network and creates a colored map with the wave speed adjustment values. In order to execute this functionality users must previously define a `PTSNETSimulation` object and specify the name of the file for saving the plot, as shown in lines 1–27 in Fig. 8 and line 8 in Fig. 10, respectively. This method is executed prior to the transient simulation and only takes a few seconds to run even for networks with thousands of pipes. Hence, it provides the users with a rapid selection of a time step that fits their modeling needs.

5.2. Determining the number of processors

The *Analytics* functionalities in PTSNet allow users to determine the optimal number of processors for their simulations. Considering that the performance of the library depends on the characteristics of the transient model (e.g., model resolution, duration, number of boundary conditions) and hardware specifications (Riaño-Briceño et al., 2021a), estimating the number of processors that maximize speedup and from which there is no improvement in computational performance is not straightforward. As more processors are used in a simulation, less time is spent on computing simulation steps, but the time spent on communication increases. Also, the sequential part of the program limits the speedup, as shown in Riaño-Briceño et al. (2021a). Furthermore, running a simulation multiple times using a different number of processors might become cumbersome when using a supercomputer or when a simulation takes too long to run. To help the user manage these issues, PTSNet includes functionalities to (i) estimate computational times and (ii) determine the optimal number of processors for a specific application. First, the `compute_simulation_times` functionality (shown in line 7 in Fig. 10) allow users to run a set of simulations with different time step values and a given number of processors, such that only a fraction of the simulation is executed and profiled. Based on the average computation time per step, the `compute_simulation_times` functionality estimates the total computation time that will be required to execute the simulation. Second, the `compute_num_processors` functionality (shown in line 6 in Fig. 10) allows users to determine the number of processors that best fits their application by running a fraction of the simulation with various processor numbers and a fixed time step defined by the user. Users can specify the number of steps that they want to compute for the partial simulations, and the average running times per step are computed for each processor. This allows the user to determine the number of processors that provides the minimum running time per step for their application. The resulting computational times are stored in temporary binary files within the workspace, such that users can consult them even after the simulation is terminated (see Section 3.3 for more details).

5.3. Exporting results

At the end of the simulation, users can access simulation results (including head and discharge at the nodes and, flowrates at the start and end nodes of the links) for all time steps of the simulation. Results associated with nodes and pipe extremes can be extracted from the

```
# Analytics
from ptsnet.utils.analytics import compute_num_processors
from ptsnet.utils.analytics import compute_simulation_times
from ptsnet.graphics.static import plot_wave_speed_error

compute_num_processors(sim, steps=1000, max_num_processors=8)
compute_simulation_times(inpfile, time_steps=[0.01, 0.001], max_num_processors=8)
plot_wave_speed_error(sim, 'WSError.pdf')
```

Fig. 10. PTSNet *Analytics* functionalities to compute optimal number of processors, estimate simulation times, and plot wave speed errors before running a simulation.

```
# ------------------ (3) Extraction ------------------

import matplotlib.pyplot as plt
from ptsnet.simulation.sim import PTSNETSimulation

with PTSNETSimulation("TNET3_SIM") as sim:
    plt.plot(sim['time'], sim['node'].head['JUNCTION-23'], label='JUNCTION-23')
    plt.xlabel('Time [s]'); plt.ylabel('Head [m]'); plt.legend()
    plt.show()
```

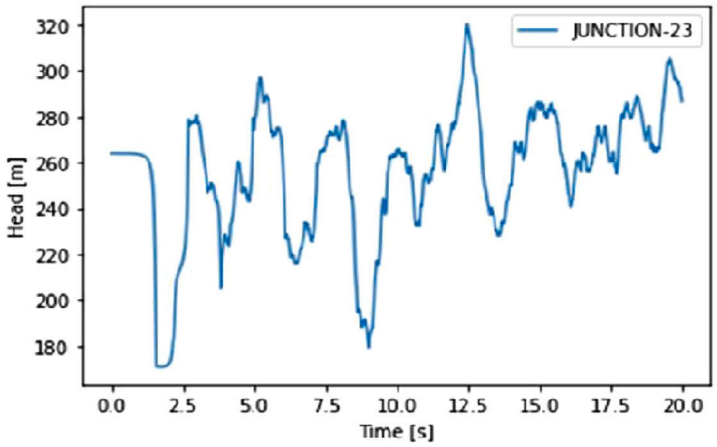


Fig. 11. *Extraction* functionalities in PTSNet to read simulation results after execution; and resulting plot showing the head at JUNCTION-23.

`sim['node']`, `sim['pipe.start']`, and `sim['pipe.end']` data structures respectively (see Section 3.3). When operating with results, users can use PTSNet in *normal* or *persistent* modes. Depending on the mode, results will be retrieved from RAM or disk memory. Under the *normal* mode, users can extract results directly from RAM after the transient simulation is completed, without closing the current Python session, i.e., calling the results data structures after line 37 in Fig. 8 using lines 7–9 in Fig. 11.

Users operating under *persistent* mode can read results saved from previous simulations. To use the persistent mode, a copy of the workspace and results are saved in HDF5 files via simulation settings (line 14 in Fig. 8). The *persistent* mode is activated by opening a simulation using Python's `with` statement, which ensures a safe manipulation of the workspace files, as shown in line 6 in Fig. 11. In the code presented in Fig. 11, results are extracted via *persistent* mode by invoking the `PTSNetSimulation()` constructor, which retrieves data from the workspace saved with the name passed as an argument. As a result, the head for `JUNCTION-23` is plotted, as shown at the bottom of Fig. 11. Workspaces persist in memory only if the user specifies so through the simulation settings when simulating for the first time under the *normal* mode. After that, users can assign a name to the workspace to differentiate results from future simulations. The Jupyter notebook `2_get_results` available on GitHub demonstrates how to load results saved in previous simulations.

6. Results

We assess the performance of the PTSNet package by: (i) comparing the accuracy and computational times with other transient software using different transient scenarios, (ii) performing scalability tests with large-scale WTN and analyze the performance on a personal computer (PC) and high performance computer (HPC), and (iii) showcase the implementation of *Analytics* functionality to explore wave speed adjustment and the effect of time step selection.

6.1. Transient software comparison

The following example demonstrates the capability of PTSNet to model a pump shut-off scenario in a mid-scale network. The simulation results of the PTSNet library executed on a single processor are compared against the TSNet Python package, an open-source library (Xing and Sela, 2020), and Bentley Hammer v8i (Bentley Systems, 2019), a commercial software broadly used in both academia an industry to analyze transient phenomena in pipes. We validate the physical results of the model by comparing the transient response of the system to different scenarios in terms of heads at a set of junctions in different locations in the network (labeled in Fig. 12). The Jupyter notebook `4_SI_figures` available on GitHub demonstrates how to replicate the results in this section and in the SI.

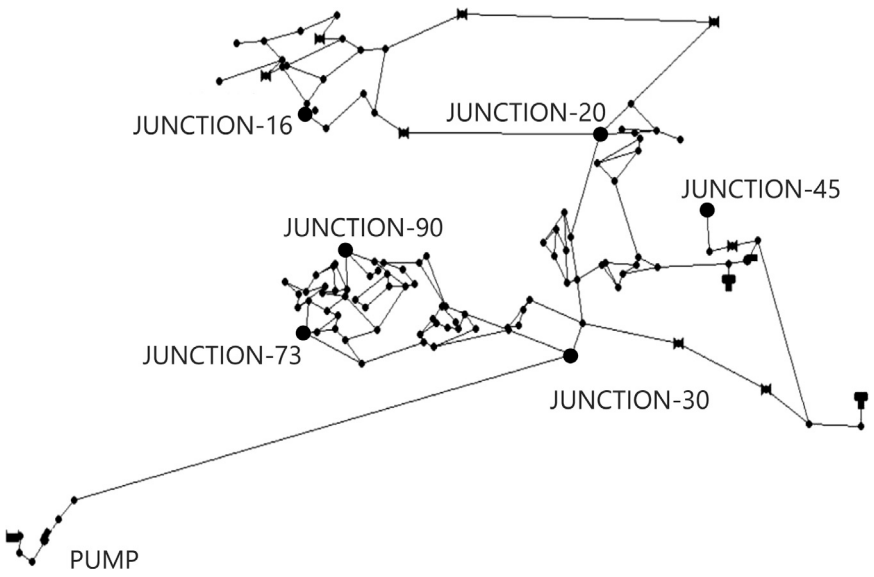


Fig. 12. BWSN-I network.

Example (*Mid-scale System*)**.** Consider the BWSN-I network presented in Fig. 12 and adapted from Ostfeld et al. (2008). BWSN-I comprises 126 nodes, one reservoir, 168 pipes, two pumps, and eight valves. Different transient events are generated, including pump shut-off, burst, and valve closure, resulting pressure waves that propagate throughout the entire WTN. For illustration purposes, all pipes are assumed to have a wave speed value of 1200 m/s, and the transient event is simulated for 20 s, using a time step of 5 ms.

6.1.1. Numerical results

The pump shut-off scenario illustrates how PTSNet models a transient event resulting from a controlled pump shut-off at PUMP-172 by decreasing the pump rotational speed to zero starting at 0 and until 1 s into the simulation. Fig. 13 shows the pressure at five different nodes in the network in response to pump shut-off, where (a), (b), and (c) correspond to simulation results using PTSNet, Hammer, and TSNet, respectively. The results from the three solvers closely resemble attenuation and phase shift throughout the simulation period, capturing down- and up-surges in pressure. The minor discrepancies are attributable to the different wave speed adjustment schemes and boundary condition computation methods adopted by the three packages, e.g., PTSNet adopts flow-based equations whereas TSNet are velocity-based, and Hammer's wave speed adjustment method differs from that of both PTSNet and TSNet.

Overall, all the methods show pressure waves generated by the pump propagating through the system. As waves propagate their amplitude and shape changes according to the network topology. The amplitude of waves attenuates or amplifies as a result of wave reflections and transmissions, and the transient pressure observed at different locations of a WTN is an aggregated signal of multiple pressure waves. Hence, locations farther from the origin of the transient do not necessarily exhibit lower amplitudes. For example, in Fig. 13, JUNCTION-30 senses the transient first, while JUNCTION-16 experiences it last. The node with the largest change in pressure is JUNCTION-90, which experiences a pressure drop of over 50 m after 7 s, indicating that the pump shut-off, can generate significant transients in the WTN when operated quickly. Therefore, it is essential to evaluate the impacts of and design appropriate procedures to guide pumping operations. Additional results comparing simulation results between the three solvers for burst, valve closure, and pump shut-off with open and closed surge tanks are shown in Figs. S1–S4 in the SI.

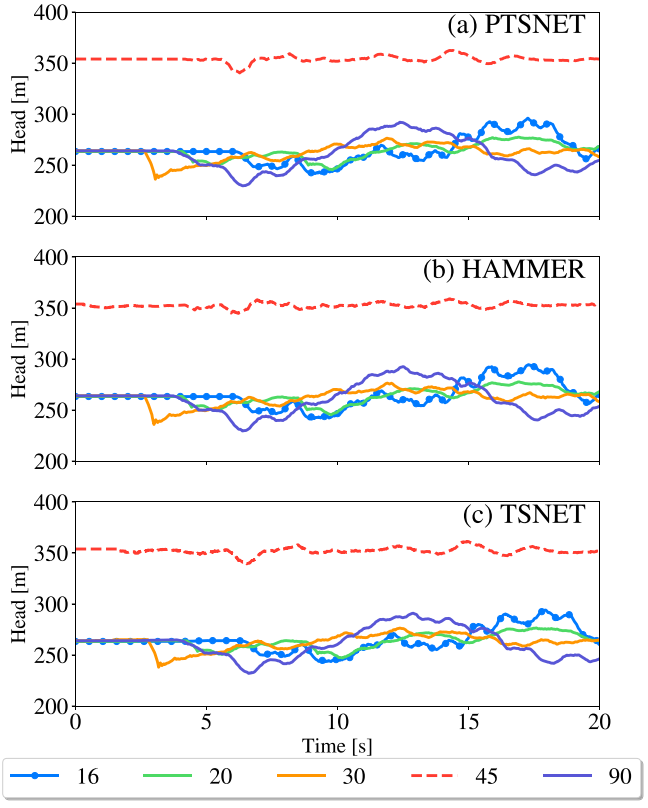


Fig. 13. Simulation results for pump shut-off scenario in the BWSN-I network: (a) PTSNet, (b) Bentley Hammer v8i, and (c) TSNet.

6.1.2. Computational performance results

Table 1 lists the running times for the different test cases executed by each package, i.e., pump shut-off, valve closure, and burst. It is evident that PTSNet surpasses in performance both TSNet and Hammer packages. On average, simulating the mid-scale network takes around 6 s with PTSNet, 700 s with TSNet, and 45 s with Hammer. Overall, PTSNet ran roughly seven times faster than Hammer in all the test cases and 116 times faster than TSNet. Notably, for a fair comparison, PTSNet

Table 1
Comparison of simulation times between transient models.

Case	PTSNet [s]	TSNet [s]	Hammer [s]
Valve closure	6.01	729.50	53.85
Pump shut-off	6.09	709.51	40.89
Burst	6.23	711.80	41.66

Table 2
Simulation parameters for large-scale system.

Time step	Value [s]	Solution points	Steps
τ_1	0.0015	1.2×10^6	5.3×10^4
τ_2	0.0010	1.8×10^6	8×10^4
τ_3	0.0005	3.7×10^6	1.6×10^5

simulations were executed using a single processor, hence the speedups are largely attributed to vectorization of the MOC equations.

6.2. Scalability test

In this section, we show performance tests for a large-scale test case to illustrate the benefits of PTSNet over other transient modeling tools. We focus on the computational aspect of the software, showing simulation times when executing PTSNet in parallel. We introduce a large-scale test case using BWSN-II water system, which is a well-known benchmark in the water systems research community (Ostfeld et al., 2008), and compare running times for three different numerical grid resolutions. We also compare average running times per time step for the denser numerical grid using both PC and HPC systems. We show that the optimal number of processors depends on simulation properties and on hardware specifications. All the reported running times are wall-clock times (Grama et al., 2003). PC simulations ran on a 16 GB RAM computer with an AMD Ryzen 7 5700U 4.5 GHz processor. Simulations performed on a supercomputer ran on the `Stampede 2` system (Stanzione et al., 2017) and its Intel Xeon Phi 7250 1.6 GHz computing nodes, whose hardware specifications can be found in TACC - Texas Advanced Computing Center (2020). The results presented in this section are not compared with commercial or open-source software, since it was not possible to execute the simulations for the large-scale system. Initializing the BWSN-II network with TSNet takes several hours and Hammer v8i crashes when executing the simulation with small time steps.

Example (*Large-scale System*)**.** The BWSN-II network is adapted from Ostfeld et al. (2008). BWSN-II comprises 12,526 nodes, two reservoirs, 14,824 pipes, a single pump, and six valves. A transient event was generated by rapidly closing a valve that controls the flow in one of the main pipelines in the system within 10 s, thus producing a pressure wave that propagates throughout the entire piped network. For illustration purposes, we assume that all pipes have a wave speed value of 1000 m/s.

6.2.1. Comparing different numerical grid resolutions

We model the large-scale network using three different numerical-grid resolutions to study the performance of PTSNet when dealing with highly computationally-expensive simulations. In general, the model resolution is highly sensitive to the time step value, given that small changes in time step result in large changes in problem size due to the CFL condition. Considering that the problem size grows exponentially for high-resolution models, simulating with PTSNet using a single processor can be computationally restrictive due to memory and running-time limitations. Since PTSNet adopts the DV-MOC framework, it ensures scalability for high-resolution models by dividing the transient flow problem among multiple processors. We report detailed results for three numerical-grid resolutions. Table 2 lists the time steps, and the corresponding number of solution points in the numerical grid and the number of time steps in the simulation.

In Fig. 14(a), we present running times from 1, 22, and 64 processors, with simulation times categorized by subprocess, i.e., initialization, computation of interior points, computation of boundary conditions, and communication among processors. With a single-core system the simulation takes 879, 2815, and 10,188 s for the numerical-grid resolutions given by τ_1, τ_2, and τ_3, respectively. Overall, as the number of processors increases the marginal improvement in running times decreases until no further improvement is observed.

Analyzing the computational time invested in computing the different subprocesses, we observe that the majority of the time running simulations with one processor is spent computing interior points for all the different numerical grid resolutions. However, once the number of processors increases the time spent in communication dominates the simulation time. The computational time on interior points decreases, since the number of interior points per processor decreases as the number of processors increases, thus reducing the computational burden per processor. However, as the number of processors increases more partitions are generated, thus increasing the number of information dependencies between processors, which increases the cost of communication. Regardless of the time step, it is demonstrated that communication becomes the bottleneck of the simulation for any sufficiently large set of processors. Simultaneously, the time spend in the computation of interior and boundary points becomes minimal as the number of processors increases for a problem of fixed size. Note that initialization is almost negligible compared to the rest of the subprocesses across all cases. Furthermore, the time spent computing boundary conditions accounts for roughly 5% of the total simulation time and remains virtually constant for 22 cores or more.

A speedup metric can be defined as $S = t_1/t_k$, where t_1 is the time spent simulating with DV-MOC on a single processor, and t_k is the time spent simulating with k processors. In general, the speedups are more significant as the number of interior points in the numerical grid increases. If we compare the running times for 1 and 64 processors when the time step is equal to τ_3, the speedup is 3; the speedup with τ_2 is 1.85, and with τ_1 the speedup is 1.43. Thus, it is not as efficient to use multiple processors for τ_1 since the communication overhead outweighs the reduction in the computation of solution points. Additionally, higher speedups are achieved with denser numerical grids, which can be attributed to the fact that cache mechanisms operate more efficiently as the number of contiguously allocated solution points increases (Riaño-Briceño et al., 2021a).

6.2.2. Comparing PC with HPC systems

In Fig. 14(b), we show the performance in terms of average running-time per time-step for the large-scale system BWSN-II running on both PC and HPC systems. Simulations running on both systems were identical with respect to simulation settings, numerical-grid resolution, and the number of processors. Specifically, we ran 2500 time steps with a time step of $\tau_3 = 0.0005$ s. We measured running times using PTSNet's profiler module and averaged running times per step, including computation of solution points and communication between processors. As evidenced by Fig. 14(b), running-times per time-step differ significantly between the PC and HPC simulations. The PC running times are faster than the HPC as long as the number of processors is less than 8. Additionally, for the HPC the computational time decreases as the number of processors increases. However, for PC an initial decrease in computational times is observed when increasing the number of processors from one, but as more processors are added, the overall computational times increase. The latter can be attributed to the different computational burdens between PC and HPC systems. It might be surprising that the PC runs faster than the HPC, yet if we compare the core specifications of both machines, the PC cores run almost three times faster than the supercomputer cores due to the differences in frequencies, i.e., 4.5 GHz vs 1.6 GHz. Nevertheless, the HPC performance surpasses the PC as

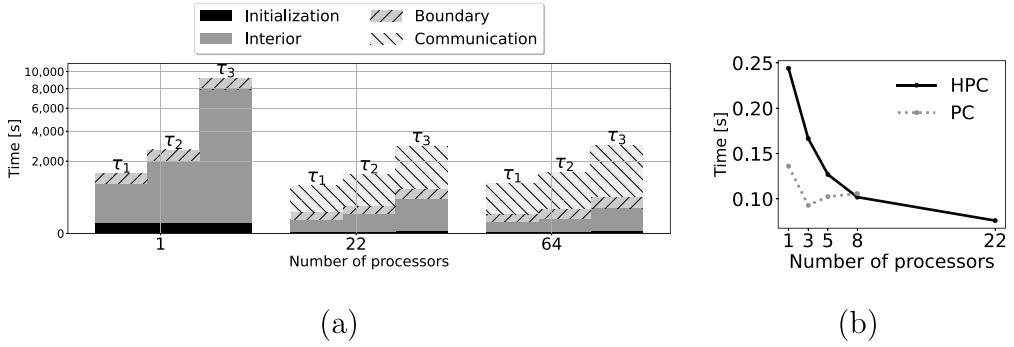


Fig. 14. Running times for the BWSN-II network: (a) total simulation time on Stampede 2, using $\tau_1 = 0.0015$ s, $\tau_2 = 0.001$ s, and $\tau_3 = 0.0005$ s; (b) Average running times per time step for Stampede 2 and personal-use PC with $\tau = 0.0005$ s.

the number of processors is increased, given that the HPC architecture has been optimized to minimize the communication burden between processors. Communication rapidly becomes a bottleneck for the PC with far fewer cores than for the HPC system. The optimal number of processors for the PC is three, while the optimal number of processors for the HPC is estimated to be eight using the elbow of the curve as the selection criterion (see Fig. 14(b)). The elbow point is a commonly used heuristic to determine the optimal balance between computational cost and speedup (Grama et al., 2003).

6.3. Wave speed error

Referring back to Section 5.1, the simulation time step determines the resolution, accuracy, and computational burden of a transient model. Selecting a time step that results in a grid resolution that ensures high accuracy and fast running times is ideal, yet these goals are conflicting: finer grid resolutions provide higher accuracy, but also demand higher computational resources and are likely to lead to communication bottlenecks and slower execution. PTSNet provides functionalities to estimate the time step automatically, balancing the trade-off between accuracy and computational burden (Bentley Systems, 2022b). For example, for large-scale networks, Hammer v8i assigns more weight to the computational cost, hence the automatic time step selection results in high simulation errors. In such cases, users need to manually tune the time step, testing different grid resolutions by iteratively running very long simulations.

To overcome this manual process, we propose using the wave speed adjustment error as a proxy for the simulation error, allowing the user to visualize the error before running the simulation. Visualizing the wave speed error produced by the resolution of the numerical grid facilitates identifying parts of the network where the error will be higher and determining whether or not a time step selection is satisfactory for a particular application. For example, a user might find it acceptable to have a time step that results in acceptable wave speed error in the study region and localized high wave speed error far from the area of study. Also, users can define a time step that minimizes the problem size, such that the majority of the network pipes remain below a certain wave speed error threshold.

To illustrate, we used PTSNet *Analytics* functionalities to plot the wave speed error for two time steps: $\tau_1 = 0.01$ s and $\tau_2 = 0.1$ s. The resulting numerical grids have 199,923 and 46,642 solution points for τ_1 and τ_2, respectively. Thus τ_1 provides a higher resolution and accuracy compared to τ_2. In Fig. 15 we show the relative wave speed error, defined as in Riaño-Briceño et al. (2021a), for both time steps, zooming on a section of the BWSN-II network. The plot shows that error remains below 10% for almost every pipe in the network when using the smaller time step for τ_1, whereas for τ_2, the error is greater than 10% for almost every pipe in the system. Even though the number of solution points became 3.2 times greater for τ_1 compared to τ_2, which can result in longer simulation times, users may prefer using the smaller time step and exploit PTSNet parallel capabilities to run this test case.

7. Conclusion

This paper presents PTSNet, an open-source Python package for parallel transient simulation in WTNs. All the source code, software documentation, and multiple examples, including input files and codes, are provided within the package and can be downloaded from the GitHub repository (Riaño-Briceño et al., 2022). The capability and user interaction with PTSNet are demonstrated through the detailed simulation examples of pump shut-off, valve closure, burst, and surge tank. The PTSNet package is proven to be computationally efficient compared to other open-source and commercial packages. Unlike other packages, we demonstrate that PTSNet is scalable and capable of running large-scale transient simulations, providing significant speedups to its users, thanks to the adoption of vectorization and distributed computing. In addition, the package offers essential analytic tools for users to quickly determine the best time step and number of processors for their application. PTSNet does not include all the modeling capabilities of the commercial software; instead, it is designed to provide simulation capabilities for transient modeling in WTNs for the research community that is currently not available in open-source software, including PETSc (Abhyankar et al., 2020) and TSNet (Xing and Sela, 2020). PTSNet is under continuous maintenance, improvement, and development. Future work includes extending the solver to generalize to other PDE-based models, e.g., open-flow channel modeling, 2D and 3D dynamics (Li and Hodges, 2021; Morales-Hernández et al., 2020).

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

All data and codes are available in a public github repository that was referenced in the manuscript https://github.com/gandresr/PTSNET.

Acknowledgments

The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this publication. This work was supported in part by the National Science Foundation, US under award 2015658 and Cooperative Agreement No. 83595001 awarded by the U.S. Environmental Protection Agency to The University of Texas at Austin. It has not been formally reviewed by EPA. The views expressed in this presentation are solely those of the authors, and do not necessarily reflect those of the Agency. EPA does not endorse any products or commercial services mentioned in this publication.

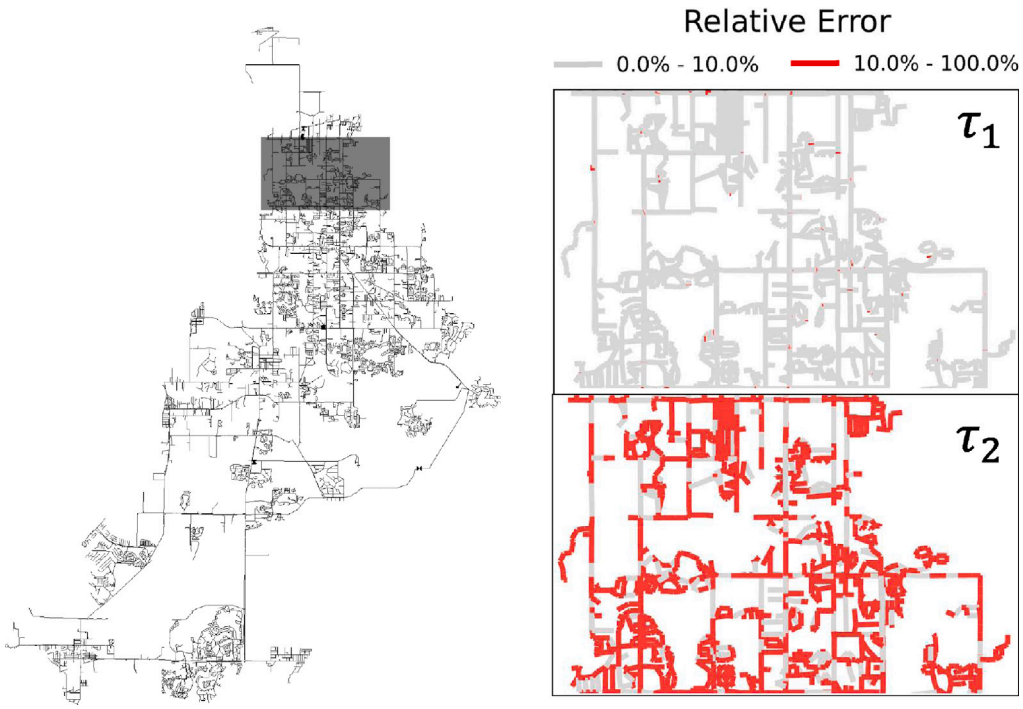


Fig. 15. Relative error for wave speed after adjustment for the BWSN-II test case using time steps $\tau_1 = 0.01$ s and $\tau_2 = 0.1$ s.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.envsoft.2022.105554.

References

Abhyankar, S., Betrie, G., Maldonado, D.A., Mcinnes, L.C., Smith, B., Zhang, H., 2020. PETSc DNNetwork: A library for scalable network PDE-based multiphysics simulations. ACM Trans. Math. Softw. 46 (1), 1–24.

Anguita, M., Acosta, M., Fernández-Baldomero, F.J., Rueda, F.J., 2015. Scalable parallel implementation for 3D semi-implicit hydrodynamic models of shallow waters. Environ. Model. Softw. 73, 201–217.

Bentley Systems, 2019. Water hammer and transient analysis software. URL https://www.bentley.com/en/products/product-line/hydraulics-and-hydrology-software/hammer.

Bentley Systems, 2022a. Time Step and Computational Reach Length. https://docs.bentley.com/LiveContent/web/Bentley%20HAMMER%20SS6-v1/en/GUID-40CBDEA4-E74C-4A6D-9846-8E9522278B2B.html. (Accessed 29 Oct 2022).

Bentley Systems, 2022b. Selection of the Time Step. https://docs.bentley.com/LiveContent/web/Bentley%20HAMMER%20SS6-v1/en/GUID-A04E4B2C49E94CB9BADAAC7AB81DA6EF.html. (Accessed 29 Oct 2022).

Blanco, P.J., Mansilla Alvarez, L.A., Feijóo, R.A., 2015. Hybrid element-based approximation for the Navier–Stokes equations in pipe-like domains. Comput. Methods Appl. Mech. Engrg. 283, 971–993.

Boulos, P.F., Karney, B.W., Wood, D.J., Lingireddy, S., 2005. Hydraulic transient guidelines for protecting water distribution systems. Journal of American Water Works Association 97 (5), 111–124.

Burger, G., Sitzenfrei, R., Kleidorfer, M., Rauch, W., 2014. Parallel flow routing in SWMM 5. Environ. Model. Softw. 53, 27–34.

Burger, G., Sitzenfrei, R., Kleidorfer, M., Rauch, W., 2016. Quest for a new solver for EPANET 2. J. Water Resour. Plan. Manag. 142 (3), 04015065.

Cao, H., Mohareb, M., Nistor, I., 2020. Finite element for the dynamic analysis of pipes subjected to water hammer. Journal of Fluids and Structures 93, 102845.

Carlotto, T., Chaffe, P.L.B., dos Santos, C.I., Lee, S., 2021. SW2D-GPU: A two-dimensional shallow water model accelerated by GPGPU. Environ. Model. Softw. 145, 105205.

Castro, M.J., García-Rodríguez, J.A., González-Vida, J.M., Parés, C., 2006. A parallel 2d finite volume scheme for solving systems of balance laws with nonconservative products: Application to shallow flows. Comput. Methods Appl. Mech. Engrg. 195 (19–22), 2788–2815.

Chaudhry, M.H., Hussaini, M.Y., 1985. Second-order accurate explicit finite-difference schemes for water hammer analysis. J. Fluids Eng. 107 (4), 523–529.

Collette, A., 2013. Python and HDF5: Unlocking Scientific Data. O'Reilly Media, Inc..

Courant, R., Friedrichs, K., Lewy, H., 1967. On the partial difference equations of mathematical physics. IBM J. Res. Dev. 11 (2), 215–234.

Fernández-Pato, J., García-Navarro, P., 2014. Finite volume simulation of unsteady water pipe flow. Drinking Water Engineering and Science 7 (2), 83–92.

Fox, S., Shepherd, W., Collins, R., Boxall, J., 2014. Experimental proof of contaminant ingress into a leaking pipe during a transient event. Procedia Eng. 70, 668–677.

Fritzson, P., 2011. Introduction to Modeling and Simulation of Technical and Physical Systems with Modelica. John Wiley & Sons.

Graham, R.L., Woodall, T.S., Squyres, J.M., 2005. Open MPI: A flexible high performance MPI. In: International Conference on Parallel Processing and Applied Mathematics. Springer, pp. 228–239.

Grama, A., Kumar, V., Gupta, A., Karypis, G., 2003. Introduction to Parallel Computing. Pearson Education.

Holton, J.R., 1973. An introduction to dynamic meteorology. Amer. J. Phys. 41 (5), 752–754.

Innovyze, 2019. Infosurge users guide. URL https://www.innovyze.com/en-us/products/infowater/infosurge.

Jasak, H., Jemcov, A., Tukovic, Z., et al., 2007. OpenFOAM: A C++ library for complex physics simulations. In: International Workshop on Coupled Methods in Numerical Dynamics, Vol. 1000. IUC Dubrovnik Croatia, pp. 1–20.

Kiuchi, T., 1994. An implicit method for transient gas flows in pipe networks. Int. J. Heat Fluid Flow 15 (5), 378–383.

Kjerrumgaard Jensen, R., Kær Larsen, J., Lindgren Lassen, K., Mandø, M., Andreasen, A., 2018. Implementation and validation of a free open source 1D water hammer code. Fluids 3 (3), 64.

Klise, K.A., Murray, R., Haxton, T., 2018. An overview of the water network tool for resilience (WNTR).

KYPipe, 2019. Kypipe 2018 users guide. URL http://kypipe.com/.

Larock, B.E., Jeppson, R.W., Watters, G.Z., 1999. Hydraulics of Pipeline Systems. CRC Press.

LeChevallier, M.W., Gullick, R.W., Karim, M.R., Friedman, M., Funk, J.E., 2003. The potential for health risks from intrusion of contaminants into the distribution system from pressure transients. Journal of Water and Health 1 (1), 3–14.

Li, Z., Hodges, B.R., 2021. Revisiting surface-subsurface exchange at intertidal zone with a coupled 2D hydrodynamic and 3D variably-saturated groundwater model. Water 13 (7), 902.

Lin, Q., Zhang, D., 2021. A scalable distributed parallel simulation tool for the SWAT model. Environ. Model. Softw. 144, 105133.

McCalpin, J.D., 2022. A peculiar throughput limitation on Intel's Xeon Phi x200 (Knights Landing). https://sites.utexas.edu/jdm4372/tag/xeon-phi/. (Accessed 19 May 2022).

McInnis, D., Karney, D., 1998. TransAM reference manual. URL http://hydratek.com/expertise/transient-analysis-model.

Mesgari Sohani, S., Ghidaoui, M.S., 2019. Formulation of consistent finite volume schemes for hydraulic transients. Journal of Hydraulic Resesearch 57 (3), 353–373.

Misiūnas, D., 2008. Failure Monitoring and Asset Condition Asssessment in Water Supply Systems. Vilniaus Gedimino technikos universitetas.

Morales-Hernández, M., Sharif, M.B., Gangrade, S., Dullo, T.T., Kao, S.-C., Kalyanapu, A., Ghafoor, S., Evans, K., Madadi-Kandjani, E., Hodges, B.R., 2020. High-performance computing in water resources hydrodynamics. J. Hydroinform. 22 (5), 1217–1235.

Nault, J.D., Karney, B.W., Jung, B.-S., 2018. Generalized flexible method for simulating transient pipe network hydraulics. Journal of Hydraulic Engineering 144 (7), 04018031.

Ostfeld, A., Uber, J.G., Salomons, E., Berry, J.W., Hart, W.E., Phillips, C.A., Watson, J., Dorini, G., Jonkergouw, P., Kapelan, Z., 2008. The battle of the water sensor networks (BWSN): A design challenge for engineers and algorithms. J. Water Resour. Plan. Manag. 134 (6), 556–568.

Riaño-Briceño, G., Hodges, B.R., Sela, L., 2022. PTSNet. http://dx.doi.org/10.18738/T8/CKIOSF, URL https://github.com/gandresr/ptsnet.

Riaño-Briceño, G., Sela, L., Hodges, B.R., 2021a. Distributed and vectorized method of characteristics for fast transient simulations in water distribution systems. Comput.-Aided Civ. Infrastruct. Eng. 37 (2), 163–1846.

Riaño-Briceño, G., Sela, L., Hodges, B.R., 2021b. Supporting Information - Distributed and Vectorized Method of Characteristics for Fast Transient Simulations in Water Distribution Systems. http://dx.doi.org/10.18738/T8/OTC0OO.

Rossman, L.A., 1994. EPANET users manual.

Scroggs, M.W., Dokken, J.S., Richardson, C.N., Wells, G.N., 2021. Construction of arbitrary order finite element degree-of-freedom maps on polygonal and polyhedral cell meshes. arXiv preprint arXiv:2102.11901.

Sela, L., Housh, M., 2019. Increasing usability of water distribution analysis tools through plug-in development in EPANET. J. Hydraul. Eng. 145 (5), 02519001.

Sela, L., Salomons, E., Housh, M., 2019. Plugin prototyping for the EPANET software. Environ. Model. Softw. 119, 49–56.

Stanzione, D., Barth, B., Gaffney, N., Gaither, K., Hempel, C., Minyard, T., Mehringer, S., Wernert, E., Tufo, H., Panda, D., 2017. Stampede 2: The evolution of an XSEDE supercomputer. In: Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact. pp. 1–8.

TACC - Texas Advanced Computing Center, 2020. Stampede 2 User Guide. URL https://portal.tacc.utexas.edu/user-guides/stampede2.

Tiernan, E.D., Hodges, B.R., 2022. A topological approach to partitioning flow networks for parallel simulation. Journal of Computing in Civil Engineering 36 (4), 1–13. http://dx.doi.org/10.1061/(ASCE)CP.1943-5487.0001020.

Verdugo, F., Martín, A.F., Badia, S., 2019. Distributed-memory parallelization of the aggregated unfitted finite element method. Comput. Methods Appl. Mech. Engrg. 357, 112583.

Wylie, E.B., Streeter, V.L., Suo, L., 1993. Fluid Transients in Systems, Vol. 1. Prentice Hall Englewood Cliffs, NJ.

Xing, L., Sela, L., 2020. Transient simulations in water distribution networks: Tsnet python package. Adv. Eng. Softw. 149, 102884.

Xu, Z., Tang, G., Jiang, T., Chen, X., Chen, T., Niu, X., 2021. An automatic partition-based parallel algorithm for grid-based distributed hydrological models. Environ. Model. Softw. 144, 105142.

Zhao, M., Ghidaoui, M.S., 2004. Godunov-type solutions for water hammer flows. J. Hydraul. Eng. 130 (4), 341–348.

Zhu, L.-J., Liu, J., Qin, C.-Z., Zhu, A.-X., 2019. A modular and parallelized watershed modeling framework. Environ. Model. Softw. 122, 104526.