Document downloaded from:

http://hdl.handle.net/10251/67965

This paper must be cited as:

Alvarruiz Bermejo, F.; Martínez Alzamora, F.; Vidal Maciá, AM. (2016). Improving the performance of water distribution systems' simulation on multicore systems. Journal of Supercomputing. 1-13. doi:10.1007/s11227-015-1607-5.

The final publication is available at

http://link.springer.com/article/10.1007/s11227-015-1607-5

Copyright Springer

Additional Information

The final publication is available at Springer via http://dx.doi.org/10.1007/s11227-015-1607-5

The Journal of Supercomputing manuscript No.
The final publication is available at Springer via
http://dx.doi.org/10.1007/s11227-015-1607-5

Improving the Performance of Water Distribution Systems Simulation on Multicore Systems

Fernando Alvarruiz ·
Fernando Martínez Alzamora ·
Antonio M. Vidal

Received: date / Accepted: date

Abstract Hydraulic solvers for the simulation of flows and pressures in water distribution systems (WDS) are used extensively, and their computational performance is key when considering optimization problems. This paper presents an approach to speedup the hydraulic solver using OpenMP with two efficient methods for WDS simulation. The paper identifies the different tasks carried out in the simulation, showing their contribution to the execution time, and selecting the target tasks for parallelization. After describing the algorithms for the selected tasks, parallel OpenMP versions are derived, with emphasis on the task of linear system update. Results are presented for four different large WDS models, showing considerable reduction in computing time.

Keywords Water Distribution Systems · simulation · Epanet · multicore · OpenMP · GGA · loop method

1 Introduction

Hydraulic solvers for the simulation of flows and pressures in water distribution systems (WDS) are used extensively to solve a large number of problems, such

This work has been partially supported by *Ministerio de Economía y Competitividad* from Spain, under the project TEC2012-38142-C04-01, and by project PROMETEO FASE II 2014/003 of *Generalitat Valenciana*

F. Alvarruiz
Dept. Sistemas Informáticos y Computación, Universitat Politecnica de Valencia, 46022 Valencia, Spain. E-mail: fbermejo@dsic.upv.es

F. Martínez-Alzamora
Research Institute of Water and Environmental Engineering (IIAMA), Universitat Politecnica de Valencia, 46022 Valencia, Spain

A. M. Vidal
Dept. Sistemas Informáticos y Computación, Universitat Politecnica de Valencia, 46022 Valencia, Spain

as network optimal design, maintenance, model calibration or risk analysis problems. In many cases the simulation needs to be repeated many times within an optimization framework [13], even more when demands are pressure dependent as in head driven models. In the particular case of running water quality models, all pipes must be considered in the model, thus enlarging its size and consequently the computing time. This also applies when models are synchronized with GIS databases. Thus, the computational performance of the hydraulic solver is extremely important.

Epanet [16] is a very efficient public domain WDS simulation software package, considered a reference in this field. Outperforming Epanet is not easy, and efforts to do so by introducing HPC [13,14,6,9] have had limited success, with a reduction in computing time of less than 20%, as we shall see in the following section. A recent work by the authors [4], which focused on using OpenMP to speedup Epanet, achieved a reduction of 36%.

Another way to increase the speed is to improve the simulation method, or to consider an alternative one. Recently, several papers [7,8,10,2,1] have considered the loop method for simulation of WDS, first introduced in [11], as an alternative to the more widely used *Global Gradient Algorithm* (GGA), implemented in Epanet. The authors presented in [5] some contributions to the loop method, with an efficient implementation that is shown to be faster than Epanet for the test networks considered.

This paper explores the use of OpenMP to accelerate the simulation of WDS, extending the work of the authors both in [4] and [5]. While [4] only considers the parallelization of Epanet, this paper also tackles that of the loop method. We present complete results on a set of test networks, showing better results than existing related work. The results also extend those of [5], by considering networks with automatic valves.

After reviewing related work in the next section, an introduction to the problem and the two methods under consideration is made. Then, section 4 identifies the tasks involved and selects the target tasks for parallelization. Section 5 describes the tasks, before parallel algorithms for them are presented in section 6. Finally, results are shown and conclusions are presented.

2 Related work

Among the different approaches to introduce HPC in the simulation of WDS, [3] explores the parallelization of Epanet in distributed memory platforms using MPI, considering the simulation of heads and pressures using GGA, and also the simulation of water quality and leakage minimization. In the parallelization of GGA, the key point is the solution of the linear system, which is carried out by means of a multifrontal sparse Cholesky method.

In [13], an analysis is made of the computational blocks involved in the simulation of WDS by means of GGA, using the code of the library CWSNet [12]. It points out that the task of linear system update is the most time consuming one, due to the computation of the coefficients related to the link head

losses (as will be seen in section 4). In the same paper, both vector instructions (SIMD) and Graphics Processing Units (GPU) are applied to accelerate the simulation of WDS using GGA. SIMD instructions are applied mainly to the computation of the link headloss coefficients, obtaining a reduction in computing time between 11% and 28% with respect to CWSNet, or up to 19.5% with respect to Epanet. A 240-core GPU is also used for the computation of the headloss coefficients, obtaining a reduction in computing time of up to 19% with respect to CWSNet, or up to 3.7% with respect to Epanet.

[14] tries to speedup the code of Epanet by using GPUs. In this case the use of GPUs focuses on the linear solver, using a Jacobi-preconditioned Conjugate Gradient method. However, the method fails to converge in some cases when applied to WDS simulation, showing that it is not stable enough for its use with the linear systems produced by Epanet with large networks.

[9] evaluates the use of GPUs for the solution of linear systems in the context of WDS simulation, concluding that GPUs would only reduce the execution time in cases of exceptionally large networks. [6] considers replacing the Epanet linear solver code by different modern multicore-capable linear solvers. They report that "none of the tested solvers was found to perform faster than the original solver for networks with a real-world character".

3 The problem of WDS simulation

The subject of the simulation is a network of m *link* elements (pipes, valves and pumps) through which the water flows. Links connect to *nodes*, n_s of them being *source nodes*, feeding the network, and n of them being *junction nodes*, which consume or demand water. Each node has an associated *head* (or equivalently, pressure), and there is a non-linear relationship between the flow through a link and the *head loss* between its end nodes. Figure 1 shows a simple network with 8 links (P1-P8), 4 junctions (N1-N4) and 2 sources (N5-N6).

The problem of WDS simulation, as considered in this paper, consists of computing the flows q through the links, and the heads h at the nodes, for a given simulation period. This is done through the solution of a sequence of steady-state problems, which is known as *extended period simulation*.

These steady-state problems can be solved using different methods. One of the most effective ones is the GGA [17], which is a Newton-Raphson method requiring at each iteration the solution of a linear system with a sparse, symmetric positive definite $n \times n$ matrix. The GGA is the method used in Epanet [16]. An alternative to the GGA is the loop method [11], which is also a Newton-Raphson process working with a sparse, symmetric positive definite matrix. However, the size of the matrix is in this case $n_l \times n_l$, where $n_l = m - n$ is the number of independent loops that can be found in the network, which is usually much smaller that n. Some recent papers [5,7,10,2,1] have addressed the loop method as a competitive alternative to GGA. The authors developed contributions to this method, with the corresponding implementation [5].

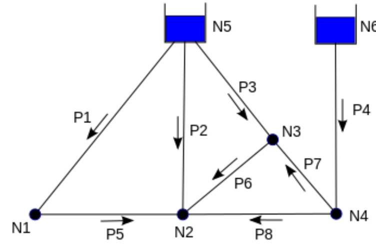

Fig. 1 Sample water supply network

This paper considers the parallelization of both the GGA, as implemented in Epanet, and the loop method, with the implementation described in [5].

4 Identification of computational blocks

In this section we present the different computational tasks involved in the process of WDS simulation, for both the GGA and the loop methods. We discuss their contribution to the execution time and select the tasks to be the subject of parallelization. A similar analysis is done in [13], although only GGA, as implemented in CWSNet, is considered.

Algorithm 1 presents a general view of the Epanet code for simulation of a time step using GGA. Several tasks or computational blocks can be identified, and are shown in parenthesis. The *while* loop corresponds to the Newton-Raphson method, in which an implicit linear system leading to the new values of h and q is reduced to a smaller system (with the unknowns h), and the computation of q is done separately.

Algorithm 1 Simulation of a time step using GGA.

set demands, valve and pump settings (**demands**)
while not converged **do**
  update linear system (**system update**)
  solve linear system, obtaining node heads h (**linear solver**)
  update flows q (**flow update**)
  update status of valves, pumps and pipes (**status update**)
end while

Algorithm 2 presents the corresponding general view of the simulation using the loop method, in the implementation described in [5]. In the Newton-Raphson method (the *while* loop) the same implicit linear system is reduced to a smaller system with the vector of unknowns $\hat{q}$, consisting of the flow corrections for each network loop.

In order to determine the contribution of each of these tasks to the total execution time of the simulation, a set of four test networks have been selected.

Algorithm 2 Simulation of a time step using the loop method.

set demands, valve and pump settings (**demands**)
balance network flows (**balance**)
while not converged **do**
  update linear system (**system update**)
  solve linear system, obtaining loop flow corrections $\hat{q}$ (**linear solver**)
  update flows q (**flow update**) and heads h (**head update**)
  update status of valves, pumps and pipes (**status update**)
end while

Table 1 Battery of test networks.

Network	nodes	links	S/P/V	Duration/step
net1	12527	14831	4/4/5	26h 55min/5min
net2	26653	29046	26/0/0	48h/20min
net3	4240	4649	4/0/6	96h/5min
net4	25816	29345	3/0/51	24h/1h

Table 2 Contribution of each task to the sequential execution time.

	GGA				loop method			
	net1	**net2**	**net3**	**net4**	**net1**	**net2**	**net3**	**net4**
initialization	7.6%	27.0%	1.2%	27.0%	3.6%	9.3%	0.9%	10.3%
demands	8.4%	5.3%	9.1%	1.0%	8.2%	6.3%	8.2%	1.1%
balance					10.3%	9.9%	10.6%	1.8%
sys update	38.7%	29.2%	42.0%	38.3%	42.3%	41.0%	42.6%	40.5%
linear solver	32.9%	29.7%	34.0%	27.8%	9.8%	8.1%	13.4%	31.5%
flow update	5.8%	4.3%	6.4%	3.7%	10.8%	10.8%	10.4%	6.8%
head update					8.8%	9.6%	8.1%	5.8%
status update	4.3%	2.6%	4.7%	1.6%	4.1%	3.0%	4.0%	1.6%
Total time	0.834	1.001	0.796	0.976	0.864	0.849	0.902	0.880

Table 1 presents the details of the networks, including the number of source nodes (S), pumps (P) and valves (V), the duration of the simulated period, and the time step between successive steady-state problems. Network 1 corresponds to one of the networks used in [15], while the other three networks are real-life hydraulic networks from Spain.

Table 2 presents the execution time (in seconds) for the simulation of the networks, and the contribution to it of each of the tasks, for both methods. Executions were run on a single CPU core with the computer and compiler described in section 7. The task of initialization is done at the beginning of the simulation and includes the reordering and symbolic factorization of the system matrix. It can be seen that GGA is faster for networks 1 and 3, while the loop method is faster for 2 and 4. With respect to the different tasks, the update of the linear system stands out as the most time-consuming. The linear solver has also an important contribution in the case of GGA, while it is less important in the loop method (because of the smaller size of the systems).

The tasks considered for parallelization in this paper are those shown in bold in table 2 (*demands*, *system update* and *flow update*). This selection has

Table 3 Maximum speedup with the proposed parallelization. *max S(p)* indicates the maximum speedup with p cores.

	GGA				loop method			
network	net1	net2	net3	net4	net1	net2	net3	net4
max S(2)	1.36	1.24	1.40	1.27	1.44	1.41	1.44	1.32
max S(4)	1.66	1.41	1.76	1.48	1.85	1.78	1.85	1.57
max S(8)	1.86	1.51	2.01	1.60	2.16	2.04	2.15	1.74
max S(16)	1.98	1.57	2.17	1.68	2.35	2.20	2.35	1.83
max S(32)	2.05	1.60	2.26	1.71	2.46	2.29	2.46	1.88

been done taking into account both the potential benefit which is likely to be gained from the parallelization, and also the difficulty of parallelization.

In particular, the solution of the linear system, which is done by means of a Cholesky factorization process for sparse matrices, could be done in parallel using elimination trees. However, as [13] exposes, the performance gain obtained could be null or minimal, taking into account the relatively small size of the matrices arising in WDS simulation. The *initialization* task has a reduced time contribution for long simulations. The tasks of *balance* and *head update* involve traversing a tree of the network nodes. This is done very efficiently in the sequential code because the order in which the nodes have to be visited has been pre-computed, which means that a parallel version could lead to little or no improvement, except in very large networks.

To end this section, table 3 presents the maximum speedup achievable with the proposed parallelization. These values have been computed according to Amdahl's law, by taking into account the percentages of time corresponding to parallelized and sequential parts.

Next section describes the tasks that are the subject of parallelization.

5 Target computational blocks

5.1 Linear system update

In the case of GGA, as described in [16], the linear system update entails the computation of two coefficients, p_k and y_k, for each link, and the assembly of them in the system matrix and vector. The coefficients are related to the head loss in a link and are computed differently depending on the type of link. For pipes, e. g., they are obtained as:

$$p_k = \frac{1}{\beta r_k |q_k|^{\beta-1} + 2\rho_k |q_k|}, \qquad y_k = p_k q_k (r_k |q_k|^{\beta-1} + \rho_k |q_k|) \tag{1}$$

where k is the link index, r and ρ are pipe resistance coefficients, and β is an exponent with the same value for all the pipes.

Once those values have been obtained, they are assembled in the linear system $Ax = b$. In particular, the matrix A is given by:

$$a_{i,i} = \sum_{k \in I_i} p_k, \qquad a_{i,j} = -\sum_{k \in I_{i,j}} p_k, \quad j \neq i \tag{2}$$

where I_i is the set of links connected to junction i, and $I_{i,j}$ corresponds to the set of links having i and j as end nodes.

Concerning vector b, each of its elements is computed as:

$$b_i = \sum_{k \in I_i^+} q_k - \sum_{k \in I_i^-} q_k - c_i - \sum_{k \in I_i^+} y_k + \sum_{k \in I_i^-} y_k + \sum_{k \in I_i^f} p_k h_j$$

where I_i^+, I_i^- represent the set of links entering and leaving, respectively, junction i, I_i^f is the set of links having node i at one end and a source node at the other end, and j refers to the source node connected to link k.

Taking into account the above considerations, the linear system update can be done as described by algorithm 3. The algorithm takes into account that the matrix A is symmetric, and is stored using a sparse structure, with the diagonal elements kept in a vector (d) and the non-zero off-diagonal elements in another one (v). It also considers that some reordering (such as a fill-reducing ordering) has been performed on the rows/columns.

Algorithm 3 Linear system update in GGA

Input: Link parameters $(r, \rho, \cdots)$, flow vector (q), demands (c). Sparse storage for matrix A, with vectors for diagonal (d) and off-diagonal (v) elements.
Output: Matrix A, vector b

1: initialize A, b, to zeros
2: **for all** link k **do**
3: compute p_k, y_k, e.g. if it's a pipe, use eq. (1)
4: let i, j be the initial and final nodes, respectively, of link k
5: **if** nodes i, j are both junctions **then**
6: $v_{k'} \leftarrow v_{k'} - p_k$, where k' is the index of the non-zero element for link k
7: **end if**
8: let i', j' be the rows of junctions i, j resp. (if nodes i, j are junctions)
9: **if** node i is a junction **then**
10: $d_{i'} \leftarrow d_{i'} + p_k, \quad b_{i'} \leftarrow b_{i'} - q_k + y_k$
11: **else**
12: $b_{j'} \leftarrow b_{j'} + p_k h_i$
13: **end if**
14: ... previous **if** block is repeated, now for node j
15: **end for**
16: **for all** junction i **do**
17: $b_{i'} \leftarrow b_{i'} - c_i$, where i' is the row of junction i
18: **end for**

Considering now the loop method, the linear system of equations is also built by assembling two coefficients, p'_k, y'_k, for each link. They are also related to the head loss in a link and, in the case of pipes, are computed as:

$$p'_k = \beta r_k |q_k|^{\beta-1} + 2\rho_k |q_k|, \quad y'_k = q_k (r_k |q_k|^{\beta-1} + \rho_k |q_k|) \tag{3}$$

Once obtained, the linear system matrix A is formed as:

$$a_{i,i} = \sum_{k \in I_i^m} p'_k, \qquad a_{i,j} = \sum_{k \in I_{i,j}^m} m_{i,k} m_{j,k} p'_k, \quad j \neq i \tag{4}$$

where I_i^m is the set of links forming the loop i, $I_{i,j}^m$ corresponds to set of links shared by loops i and j, while $m_{i,k}$ is either -1 or $+1$, depending on the orientation of link k with respect to the assumed orientation of loop i. Each element of the right-hand side vector b is obtained as:

$$b_i = -\sum_{k \in I_i^m} m_{i,k} y'_k - \sum_{j \in I_i^{m,f}} m_{i,j}^f h_j$$

where, if i is a pseudo-loop (path going from a source node to another), $I_i^{m,f}$ contains its end nodes, and otherwise is empty. $m_{i,j}^f$ is -1 $(+1)$ if j is the initial (final) node of the pseudo loop.

Valves have been considered following the approach presented in [5]. The resulting process of linear system update is presented in algorithm 4.

Algorithm 4 Linear system update in the loop method

Input: Link parameters $(r, \rho, \cdots)$, flow vector (q). Network loops. Sparse storage for matrix A, with vectors for diagonal (d) and off-diagonal elements (v).
Output: Matrix A, vector b

```
initialize A, b to zeros.
for all link k do
    compute p'_k, y'_k, e.g. if it's a pipe, use eq. (3)
    for all loop i containing link k do
        let δ be the sign of link k in loop i, and i' the row corresponding to loop i
        d_i' ← d_i' + p'_k,    b_i' ← b_i' − δ y'_k
    end for
    for all index i of nonzero coeff. contributed to by link k do
        v_i ← v_i + δ p'_k, where δ is the sign of the contribution
    end for
end for
for all source node j do
    for all loop i with j as an end node do
        let δ be the sign of node j in loop i, and i' the row corresponding to loop i
        b_i' ← b_i' − δ h_j
    end for
end for
```

5.2 Update of demands and flows

The update of demands is done in the same way for both methods. The demands are modeled using demand patterns, which consist of a base value and a sequence of multiplication coefficients that are applied to the base value to obtain a time-based modulation. Each junction can have several demand patterns associated to it, resulting in the following expression to update the demands at each iteration:

$$c_i = \sum_j \hat{c}_{i,j} \mu_{i,j,k}, \qquad 1 \le i \le n \tag{5}$$

where c_i is the demand of junction i, j iterates over the demand patterns of junction i, $\hat{c}_{i,j}$ is the base demand of the pattern and $\mu_{i,j,k}$ is the multiplication coefficient of the pattern at time step k.

With respect to the update of flows, in GGA the following increment is applied to the flow of a link k from node i to j:

$$\Delta q_k = -y_k + p_k(h_i - h_j), \qquad 1 \leq k \leq m \tag{6}$$

Additionally, the following fraction is computed, which corresponds to the relative change of the flows at the current iteration, and is used in the convergence check for the algorithm:

$$\frac{\sum_{k=1}^{m} |\Delta q_k|}{\sum_{k=1}^{m} |q_k|} \tag{7}$$

In the case of the loop method, the flow increment is:

$$\Delta q_k = \sum_i m_{i,k} \hat{q}_i, \qquad 1 \leq k \leq m \tag{8}$$

where i iterates over each of the network loops where link k is contained, and $\hat{q}_i$ is the flow correction associated to the network loop i. The relative change of the flows is also computed, using eq (7).

6 Parallel algorithms

6.1 Parallel linear system update

We first consider the GGA method. Note that in the sequential algorithm 3, a given element of the matrix A or the vector b is modified in several different iterations of the loop that goes over the links. To obtain a parallel version of the algorithm, the idea is to reorganize the code by grouping in the same iteration all the updates that are made to a given element. Then, the iterations can be done in parallel.

The result is shown in algorithm 5, where the updates are organized in four different loops. The first one simply computes the coefficients p_k and y_k and can be readily parallelized. The second loop, starting at line 5, computes the elements of the matrix diagonal and the vector b. Each iteration of the loop computes a different diagonal element $d_{j'}$ and its corresponding vector element $b_{j'}$, so that there is no conflict between iterations. The last two loops compute the off-diagonal coefficients. We take into account that a given off-diagonal coefficient can be affected by several links, although this happens rarely. To cope with this, the approach taken is to initially consider only the contribution of the first link for each off-diagonal coefficient, which is done in parallel in the third loop (line 16), and then consider the rest of the links in the last sequential loop.

Considering now the system update in loop method, the idea is again to reorganize algorithm 4 in order to group in a single iteration all the different updates affecting a given system element.

Algorithm 5 Parallel linear system update in GGA

Input: Link parameters $(r, \rho, \cdots)$, flow vector (q), demands (c). Sparse storage for matrix A, with vectors for diagonal (d) and off-diagonal (v) elements.
Output: Matrix A, vector b

1: initialize A, b, to zeros
2: **parallel for all** link k **do**
3: compute p_k, y_k, e.g. if it's a pipe, use eq. (1)
4: **end for**
5: **parallel for all** junction j **do**
6: let j' be the row of junction j
7: **for all** link k connected to junction j **do**
8: let $\delta = +1$ $(\delta = -1)$ if link k enters (leaves) junction j
9: $d_{j'} \leftarrow d_{j'} + p_k, \quad b_{j'} \leftarrow b_{j'} + \delta(q_k - y_k)$
10: **if** the other end of link k is a source node **then**
11: $b_{j'} \leftarrow b_{j'} + p_k h_i$, where i is the source node
12: **end if**
13: **end for**
14: $b_{j'} \leftarrow b_{j'} - c_j$
15: **end for**
16: **parallel for all** non-zero off-diagonal coefficient k' **do**
17: $v_{k'} \leftarrow v_{k'} - p_k$, where k is the first link of the nonzero coefficient
18: **end for**
19: **for all** link k not considered in the previous loop **do**
20: $v_{k'} \leftarrow v_{k'} - p_k$, where k' is the index of the off-diagonal element for link k
21: **end for**

That is what algorithm 6 does. The first loop in the algorithm, as in the case of the GGA counterpart, corresponds to the computation of the headloss-related coefficients for each link. The second loop (line 5) computes the matrix off-diagonal elements, each of them corresponding to an adjacency of two network loops (i.e. a pair of loops with some links in common). Since each iteration modifies a different element, this can be done in parallel. Then the third loop in the algorithm (line 11) performs updates on the matrix diagonal and the vector b. Since each iteration affects a different element of the diagonal and its corresponding vector element, this too can be done in parallel.

In order to balance the load of some parallelized loops, a preprocessing step is done to determine the best distribution of iterations among threads.

6.2 Parallel update of demands and flows

The task of demand update corresponds to eq. (5) in both methods. Since each demand c_i can be computed independently, the parallelization is straight-forward. The update of other pattern-controlled source node heads and pumps is also done using a formula similar to (5), and has also been parallelized.

Concerning the task of flow update, which corresponds to eqs. (6) and (7) in GGA, and to eqs. (8) and (7) in the loop method, the parallel version is also quite straight-forward, since the computation of each of the flow increments Δq_k is independent of the others. Computation of eq. (7) requires the use of a `reduction` clause to obtain the values of the sums.

Algorithm 6 Parallel linear system update in the loop method.

Input: Link parameters $(r, \rho, \cdots)$, flow vector (q). Network loops. Sparse storage for matrix A, with vectors for diagonal (d) and off-diagonal (v) elements.
Output: Matrix A, vector b
1: initialize A, b to zeros.
2: **parallel for all** link k **do**
3: compute p'_k, y'_k, e.g. if it's a pipe, use eq. (3)
4: **end for**
5: **parallel for all** adjacency between two loops **do**
6: let j' the index of the off-diagonal element for the adjacency
7: **for all** link k associated to the adjacency **do**
8: $v_{j'} \leftarrow v_{j'} + \delta p'_k$, where δ is the sign of the link in the adjacency
9: **end for**
10: **end for**
11: **parallel for all** loop i **do**
12: let i' be the row of loop i
13: **for all** link k in loop i **do**
14: $d_{i'} \leftarrow d_{i'} + p'_k, \quad b_{i'} \leftarrow b_{i'} - \delta y'_k$, where δ is the sign of link k in loop i
15: **end for**
16: **if** loop i is a pseudo-loop from a source node (j) to another (j') **then**
17: $b_{i'} \leftarrow b_{i'} + h_j - h_{j'}$
18: **end if**
19: **end for**

Table 4 Sequential time **tseq** and speedup **S(#cores)** for complete simulation.

	GGA				loop method			
	net1	**net2**	**net3**	**net4**	**net1**	**net2**	**net3**	**net4**
tseq	0.837	1.005	0.797	0.981	0.771	0.747	0.814	0.808
S(2)	1.21	1.14	1.17	1.20	1.31	1.30	1.31	1.24
S(4)	1.37	1.27	1.32	1.34	1.60	1.54	1.55	1.42
S(8)	1.55	1.36	1.50	1.46	1.80	1.74	1.75	1.54
S(14)	1.63	1.40	1.59	**1.52**	**1.92**	1.81	**1.82**	1.59
S(20)	**1.65**	**1.41**	**1.60**	1.52	1.89	**1.82**	1.79	**1.60**
S(28)	1.56	1.36	1.53	1.46	1.75	1.73	1.67	1.56

7 Results and conclusions

Results are presented for the set of test networks described previously in table 1. All simulations have been carried out on a computer with 2 Intel® Xeon® E5-2697 v3 processors at 2.60GHz and 14 cores each (28 cores in total). The Intel® compiler has been used for all programs. Times are in seconds.

Table 4 presents the sequential time and the speedup for the complete simulation for both GGA and the loop method. Table 5 focuses on the performance of the tasks that have been parallelized.

We can see that the speed-up achieved for the complete simulation is not far from the ideal one shown in table 3. Table 4 shows speedups of up to 1.65 in the case of GGA and up to 1.92 in the case of the loop method, with respect to the corresponding sequential codes. The reduction in computing time is of up to 39% for GGA and up to 48% for the loop method. Comparing the time of the parallel loop method with respect to Epanet (sequential GGA),

Table 5 Sequential time **tseq** and speedup **S(#cores)** for parallelized part.

	GGA				loop method			
	net1	**net2**	**net3**	**net4**	**net1**	**net2**	**net3**	**net4**
tseq	0.440	0.386	0.456	0.420	0.443	0.404	0.467	0.371
S(2)	1.57	1.58	1.42	1.71	1.81	1.88	1.83	1.88
S(4)	2.41	2.58	1.98	2.79	3.27	3.35	3.10	3.38
S(8)	4.05	4.38	3.02	4.95	5.70	6.09	4.95	6.04
S(14)	5.81	6.41	3.83	7.38	8.73	9.45	6.08	8.98
S(20)	**6.84**	**7.73**	**4.14**	**8.77**	**10.37**	11.80	**6.09**	11.30
S(28)	6.20	6.91	3.92	6.91	8.88	**13.67**	4.95	**13.10**

the reduction in computing time goes from 44% (for net3) to 59% (for net2). These numbers are obtained from table 4, e.g for net2 the time using sequential GGA is 1.005, and the best time using the parallel loop method can be obtained as $0.747/1.82 = 0.410$, yielding a reduction of 59%. These results are clearly better that the related work reviewed in section 2, where the maximum reduction in computing time with respect to Epanet was 19.5%.

We also note that the sequential time for the loop method reduces with respect to table 2, which is due to the fact that reorganizing the code for algorithm 6 also led to a faster sequential code.

The results show important improvements in the performance of WDS simulation with respect to Epanet, which come as a result of different contributions. One of them is to consider the loop method, with the improvements described by the authors in [5], while reducing the sequential execution time as a result of code restructuring, and extending the results of [5] by considering networks with automatic valves. The other one is the parallelization over multicore systems of the simulation both for GGA and the loop method, where the loop method is shown to have better speedup.

Taking into account that problems such as network design or network optimization make intensive use of simulation, the increase of speed shown in this paper can have great impact on the time needed to solve such problems.

References

1. Abraham, E., Stoianov, I.: Efficient preconditioned iterative methods for hydraulic simulation of large scale water distribution networks. Procedia Engineering **119**, 623 – 632 (2015)
2. Abraham, E., Stoianov, I.: Sparse null space algorithms for hydraulic analysis of large-scale water supply networks. Journal of Hydraulic Engineering pp. 04015,058–04015,058 (2015). DOI 10.1061/(ASCE)HY.1943-7900.0001089
3. Alonso, J.M., Alvarruiz, F., Guerrero, D., et al: Parallel computing in water network analysis and leakage minimization. J. Water Resour. Plann. Manage. **126**(4), 251–260 (2000)
4. Alvarruiz, F., Martínez-Alzamora, F., Vidal, A.M.: Efficient simulation of water distribution systems using openmp. In: 15th Int. Conf. Math. Methods in Sci. and Eng., pp. 125–129 (2015)
5. Alvarruiz, F., Martínez-Alzamora, F., Vidal, A.M.: Improving the efficiency of the loop method for the simulation of water distribution systems. J. Water Resour. Plann. Manage. **141**(10), 04015,019 (2015)

6. Burger, G., Sitzenfrei, R., Kleidorfer, M., Rauch, W.: Quest for a new solver for EPANET 2. Journal of Water Resources Planning and Management (2015). DOI 10.1061/(ASCE)WR.1943-5452.0000596
7. Creaco, E., Franchini, M.: Comparison of Newton-Raphson global and loop algorithms for water distribution network resolution. J. Hydraul. Eng. **140**(3), 313–321 (2014)
8. Creaco, E., Franchini, M.: The identification of loops in water distribution networks. Procedia Engineering **119**, 506 – 515 (2015). Computing and Control for the Water Industry (CCWI2015) Sharing the best practice in water management
9. Crous, P.A., van Zyl, J.E., Roodt, Y.: The potential of graphical processing units to solve hydraulic network equations. Journal of Hydroinformatics **14**, 603–612 (2012)
10. Elhay, S., Simpson, A., Deuerlein, J., Alexander, B., Schilders, W.: Reformulated co-tree flows method competitive with the global gradient algorithm for solving water distribution system equations. J. Water Resour. Plann. Manage. **140**(12), 04014,040 (2014)
11. Epp, R., Fowler, A.G.: Efficient code for steady-state flows in networks. Journal of the Hydraulics Division **96**(1), 43–56 (1970)
12. Guidolin, M., Burovskiy, P., Kapelan, Z., Savić, D.: Cwsnet: An object-oriented toolkit for water distribution system simulations. In: Proc. 12th Water Distribution System Analysis Symp., ASCE, Reston, VA (2010)
13. Guidolin, M., Kapelan, Z., Savic, D.: Using high performance techniques to accelerate demand-driven hydraulic solvers. Journal of Hydroinformatics **15**(1), 38–54 (2013)
14. Guidolin, M., Kapelan, Z., Savic, D., Giustolisi, O.: High performance hydraulic simulations with epanet on graphics processing units. In: Proc. 9th Int. Conf. on Hydroinformatics (2010)
15. Ostfeld, A., Uber, J., Salomons, E., et al: The battle of the water sensor networks (BWSN): A design challenge for engineers and algorithms. J. Water Resour. Plann. Manage. **134**(6), 556–568 (2008)
16. Rossman, A.L.: Epanet 2 Users manual. Water Supply and Water Resources Division, US Environment Protection Agency (2000)
17. Todini, E., Pilati, S.: A gradient algorithm for the analysis of pipe networks. In: B. Coulbeck, C.H. Orr (eds.) Computer Applications in Water Supply: Vol. 1—Systems Analysis and Simulation. Research Studies Press Ltd., Letchworth, Hertfordshire, UK (1988)