Proceedings of the 10th Annual Water Distribution Systems Analysis Conference WDSA2008, Van Zyl, J.E., Ilemobade, A.A., Jacobs, H.E. (eds.), August 17-20, 2008, Kruger National Park, South Africa.

PARALLEL OPTIMISATION OF PUMP SCHEDULES WITH A THREAD-SAFE VARIANT OF EPANET TOOLKIT

M. López-Ibáñez[1], D.T. Prasad[2] and B. Paechter[3]

[1]Centre for Emergent Computing, School of Engineering and the Built Environment, Napier University, Edinburgh UK; email: m.lopez-ibanez@napier.ac.uk

[2]Centre for Emergent Computing, School of Engineering and the Built Environment, Napier University, Edinburgh UK; email: p.tumula@napier.ac.uk

[3]Centre for Emergent Computing, School of Computing, Napier University, Edinburgh UK; email: b.paechter@napier.ac.uk

Abstract

The optimisation of pump operations in water distribution networks is an ongoing research topic motivated by the great energy savings that a careful scheduling of pumps may achieve. State-of-the-art approaches often combine an optimisation algorithm, such as Evolutionary Algorithms, and full hydraulic simulation. Despite the advances in optimisation techniques and hardware performance, the time required to obtain a near-optimal schedule is still measured in hours. Multiple CPUs are increasingly used to parallelise tasks that require a high computation time. In fact, parallel optimisation algorithms are widely present in the literature. On the other hand, hydraulic simulators that support parallel computation are scarce. The most popular research simulator, EPANET, was not designed with concurrency in mind. In this paper, a new thread-safe variant of the EPANET Toolkit is proposed. As an application example, we propose a parallel variant of an Ant Colony Optimisation (ACO) algorithm for optimal pump scheduling in water distribution networks.. The thread-safe EPANET library is combined with the parallel ACO algorithm to achieve reduced computation time in a multi-core computer. Experimental results show that an initial computation time close to 2 hours may be reduced to less than half an hour without sacrificing the quality of the results. The number of ants is also identified as a parameter that influences execution time in the parallel ACO approach proposed in this paper.

1. INTRODUCTION

From high-performance supercomputers accessible to researchers, to the new generation of multi-core personal computers and laptops, parallel computers are becoming increasingly prevalent nowadays. Parallel computation may reduce the time required to solve a problem. However, our tools need to be adapted in order to take advantage of it. In the context of the problem of finding an optimal schedule of pumps in a water distribution network, the tool may be a combination of an optimisation algorithm and a hydraulic simulator. The optimisation algorithm generates potential schedules of pumps, while the hydraulic simulator evaluates those schedules to calculate its cost and identify violations of system and performance constraints. Although a hydraulic simulator may require just a few seconds to perform an extended period simulation of a particular pump schedule, finding a near-optimal schedule typically requires the evaluation of thousands of different schedules. Some optimisation algorithms, such as Evolutionary Algorithms and Ant Colony Optimisation (ACO), are particularly well-suited for parallel execution, since, at each iteration, they generate a population of candidate solutions that can be independently evaluated. However, one of the most popular research simulators, the EPANET Toolkit (Rossman, 1999), was not designed with parallelism in mind. In this paper we explain the implementation

of a thread-safe variant of the EPANET Toolkit, which involves a more object-oriented design, allowing concurrent multiple simulations within the same application to be performed. This thread-safe variant allows a program to execute multiple simulations in concurrent "threads", which are lightweight processes within the same computer program. The performance benefits of this thread-safe variant of EPANET Toolkit are tested by combining it with an Ant Colony Optimisation (ACO) algorithm. Previously published research shows that this ACO algorithm obtains good results when compared to other techniques, such as Evolutionary Algorithms, for the pump scheduling problem (López-Ibáñez et al., 2008). However, as other algorithms (Atkinson et al., 2000; van Zyl et al., 2004), it requires a long computation time for large real-world networks. We propose a modification of this ACO algorithm to take advantage of multiple CPUs by creating a number of threads that concurrently simulate pump schedules and return the results to the main ACO algorithm. The simulation of the schedules is carried out using the new thread-safe EPANET Toolkit.

2. LIMITATIONS OF EPANET TOOLKIT FOR PARALLEL ALGORITHMS

The EPANET Toolkit (Rossman, 1999) is an open-source C library that provides an application programming interface (API) for hydraulic and water quality simulations. An optimisation algorithm would call certain functions of EPANET to load a network description, modify the schedule of the pumps, run an extended period simulation and collect information such as the energy consumption of the pumps, tank levels and pressure values. Figure 1 shows the algorithmic schema that a hypothetical sequential optimisation algorithm would follow when interacting with EPANET. Among other functions, EPANET allows an application to load a network instance (**`ENopen`**), obtain information about the network (**`ENgetcount`**), assign schedules to pumps (**`ENsetpattern`**), and run extended period simulations (**`ENopenH`**, **`ENinitH`**, **`ENrunH`**, **`ENnextH`**, **`ENcloseH`**). The simple and straightforward interface is probably one of the reasons why EPANET is widely used for research. However, some aspects of the design of EPANET make difficult its use in parallel applications.

First, the complete status of a particular simulation cannot be easily retrieved and saved. That is, we cannot simply make a copy of a running simulation, then start a new one and, once the new one is finished, restart the first one. In fact, the implementation of the library keeps most of its internal information on global variables that are dynamically allocated with no encapsulation at all. Moreover, data structures related to a particular simulation are often lumped together with data concerning the network description, which typically never changes during the simulation. This lack of encapsulation means that the status of a particular simulation cannot be isolated from another different simulation.

The second issue that precludes the use of EPANET in parallel algorithms is that most API functions are not *reentrant*. A reentrant function only depends on its arguments and it doesn't hold any internal state. It neither calls non-reentrant functions. Therefore, it can be re-entered while it is running. This does not imply thread-safety by itself. If a reentrant function modifies its arguments and multiple threads call the function with the same arguments, there will be a problem of data synchronization. However, a non-reentrant function called by multiple threads will not execute correctly even if the arguments are not shared among threads. In order to take advantage of parallel execution of multiple simulations, the functions called during simulation must be reentrant.

Two main refactoring efforts were undertaken to enable parallelism in EPANET. First, data structures related to a hydraulic simulation were encapsulated within a simulation object, thus multiple simulations can be created and modified concurrently. There are already types in EPANET for pumps, tanks and other elements. However, these objects contain both data that corresponds to the description of the network, such as pump shut-off head and tank maximum volume, and data that is dynamically calculated during simulation, such as pump energy usage and tank head. Thus an important step in our refactoring effort

was the separation of these two kinds of data. Simulation data is encapsulated into a new type of object **`ENsimulation_t`**. This was not the only new type that was created during refactoring. Other internal structures were also encapsulated within objects because these new types will also help us to make crucial EPANET functions reentrant.

```
Initialize ()
{
      ...
      ENopen (network_file);
      ENgetcount (EN_PUMPCOUNT; &num_pumps);
      ...
}

solution_t GenerateSolution ()
{
      ...
      schedule = GenerateSchedules ();
      ...
}

EvaluateSolution (solution_t schedule)
{
      ...
      ENopenH ();
      for (p = 0; p < num_pumps; p++)
            ENsetpattern (pump[p], schedule[p], 24);
      ENinitH (0);
      do {
            ENrunH ();
            ENnextH (&tstep);
      } while (tstep > 0);
      ENgettotalenergycost (&cost);
      ENcloseH ();
      ...
      return cost;
}

main ()
{
      ...
      Initialize ();
      while (not stopping_criteria) {
            for (i = 0; i < population_size; i++) {
                  solution[i] = GenerateSolution ();
                  cost[i] = EvaluateSolution (solution[i]);
            ...
      }
      ...
}
```

Figure 1. Example of optimisation algorithm using EPANET Toolkit

The second refactoring task was the review of all EPANET functions, identifying those which are required in order to perform concurrent hydraulic simulations and converting them into reentrant functions. Since a reentrant function should only call other reentrant functions, the conversion proceeded from the API functions down to the internal functions used only within EPANET. In the original EPANET code, a function would depend on some internal state, reading and writing global variables, thus the same function cannot be executed by multiple threads. A reentrant variant works on a simulation object which is passed as an argument to the function. Hence, two threads can execute the same function concurrently as long as they use different simulation objects.

Figure 2 shows an example of an optimisation algorithm using the new thread-safe variant of EPANET. For our purposes, it is not necessary that two threads are able to concurrently execute all EPANET functions on the same data. In fact, we don't even need all functions to be reentrant, since there is no need in a parallel optimisation algorithm to execute those functions in parallel. An example would be **`ENopen()`**, which is responsible for reading the network description from an input file. This function only needs to be called once. By restricting ourselves to our objective of enabling parallel hydraulic simulation, we obviate the incorporation to the EPANET library of synchronization mechanisms, such as locking and mutual exclusion, that would be required otherwise. As well, this avoids new dependencies in order to build and use the EPANET library. We do not negate that future developments of EPANET may incorporate these characteristics. Nevertheless, as we shall see in the next sections, the current approach is sufficient to implement state-of-the-art optimisation algorithms that make use of parallelism to reduce their execution time.

```
Initialize ()
{
      ...
      ENopen (network_file);
      ENgetcount (EN_PUMPCOUNT; &num_pumps);
      ...
}

solution_t GenerateSolution ()
{
      ...
      schedule = GenerateSchedules ();
      ...
}

double EvaluateSolution (solution_t schedule)
{
      ...
      // A new simulation object is created everytime this function is executed.
      ENsimulation_t simulation;
      ENopenH (&simulation);
      for (p = 0; p < num_pumps; p++)
            ENsetpattern (simulation->pump[p], schedule[p], 24);
      ENinitH (simulation, 0);
      do {
            ENrunH (simulation);
            ENnextH (simulation, &tstep);
      } while (tstep > 0);
      ENgettotalenergycost (simulation, &cost);
      ENcloseH (simulation);
      ...
      return cost;
}

main ()
{
      ...
      Initialize ();
      while (not stopping_criteria) {
            // This "for" can be executed concurrently using i parallel threads.
            for (i = 0; i < population_size; i++) {
                  solution[i] = GenerateSolution ();
                  cost[i] = EvaluateSolution (solution[i]);
            ...
      }
      ...
}
```

Figure 2. An example of optimisation algorithm using the new thread-safe version of EPANET

3. PARALLEL RANDOM SEARCH

In order to assess the potential of the new thread-safe version of EPANET, we run a parallel random search. This random search algorithm simply generates a number of random pump schedules and evaluates them. After reaching a maximum of evaluations, it returns the schedule that generated the lowest electrical cost without violating any constraint. The parallel random search evaluates schedules in parallel by using a number of threads. A candidate schedule of the pumps is assigned to each thread, which evaluates the schedule by performing a hydraulic simulation. As soon as one thread finishes the evaluation of a solution, a new random solution is generated and assigned to it. Therefore, a thread does not wait for other threads to finish.

We apply the random search algorithm to the Richmond network instance (Atkinson et al., 2000; van Zyl et al., 2004). The Richmond network is a real water distribution network located in the United Kingdom. The network comprises 948 links, 836 nodes, 7 pumps, 6 tanks and one reservoir. The efficacy of the algorithm in terms of solution quality is of no interest here. Our only goal is to study how much the execution time is reduced by increasing the number of threads in a multi-core computer. We are also interested in the algorithm *speedup*, which is defined as:

$$S_p = \frac{T_1}{T_p} \tag{1}$$

where S_p is the speedup, p is the number of processors, T_1 is the time required by the sequential algorithm and T_p is the time of the parallel algorithm with p processors. The concept of speedup indicates how well a parallel algorithm scalates with the number of processors. *Ideal speedup* occurs when $S_p = p$. Conversely, an ideal parallel execution time can be calculated, which corresponds to the execution time of the parallel algorithm that result in ideal speedup.

Figure 3 shows the wall-clock time required for 8,000 evaluations of the random search algorithm in a 4-CPU machine (2 dual-core AMD64 Opteron 275, 2.2 GHz and 64KB/1MB of cache memory per core) running GNU/Linux. The algorithm is implemented in C and uses POSIX threads (Kerrisk, 2005). With one thread the algorithm is sequential and only makes use of one CPU. In this case, Fig. 3 shows that the runtime for the 8,000 evaluations is close to one hour. By using two threads, the load is shared between two CPUs and therefore, the computation time is halved. For 4 or more threads, the *speedup* obtained is practically ideal, that is, close to 4. Therefore, there is little to no overhead in the parallel implementation of the EPANET library with respect to the sequential version.

Admittedly, the random search algorithm discussed here is of little practical interest. The next step is to perform the same experiment in a state-of-the-art optimisation algorithm, where a smaller speedup is expected due to the sequential parts of the algorithm.

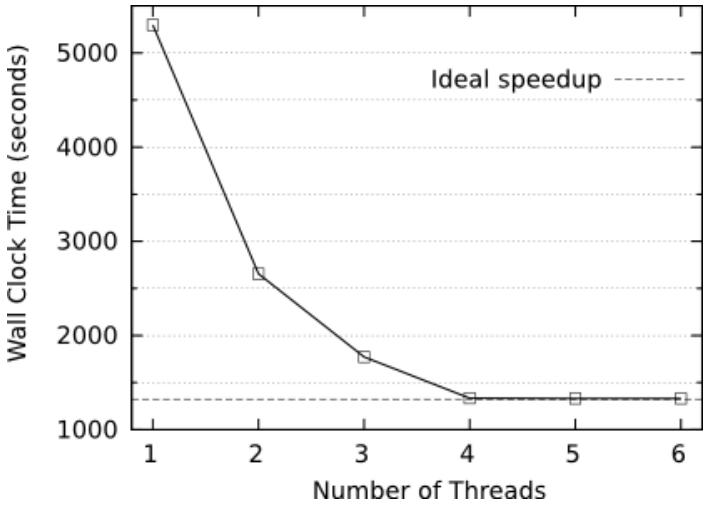

Figure 3. Runtime in seconds for random search algorithm

4. PARALLEL ACO FOR THE PUMP SCHEDULING PROBLEM

Ant Colony Optimisation (ACO) (Dorigo and Stützle, 2004) is an optimisation technique that mimics the behaviour of real ants when finding the optimal path between food and their nest. ACO has been successfully applied to both the design of water distribution networks (Maier et al., 2003) and the optimisation of pump schedules in water distribution networks (López-Ibáñez et al., 2008). In the context of the pump scheduling problem, the ACO algorithm proposed by López-Ibáñez et al. (2008) obtained state-of-the-art results. However, computation time was still high (over one hour). Our proposal in this section is to apply parallelism to this algorithm in such a way that the behaviour of the algorithm stays the same but, in the presence of multiple CPUs, the computation time can be notably reduced.

At every iteration of ACO, a fixed number of artificial *ants* iteratively construct candidate solutions to a problem. Each ant constructs a solution by stochastically adding *solution components* to its partial solution. The probability of an ant choosing a particular solution component is influenced by numerical information called *pheromone*, which is associated to each solution component.. After being constructed, each solution is evaluated to calculate its objective function cost and possible constraint violations.. The ant that constructed the best solution is allowed to deposit pheromone along its path. That is, the pheromone information associated to solution components that are part of the best solution is increased. Thus, in subsequent iterations, those solution components will have a higher probability of being chosen by an ant to complete a partial solution. This is a very general schema of the ACO framework. However, it suffices to describe how parallelism can be introduced in an ACO algorithm.

In typical ACO algorithms, the number of ants is a parameter that does not change during runtime. Moreover, it is frequent that, after each ant constructs one solution, all solutions must be evaluated before updating the pheromones. Ants evaluate their solutions sequentially as shown in Fig. 4. However, these evaluations can be performed in parallel by using as many threads as the number of ants. Therefore, the maximum speedup will be limited by the number of ants. However, the number of ants, ranging from 10 to a few hundreds, is typically larger than the number of CPUs available in a multi-core computer, so this is not a limitation in practice. Assuming the number of CPUs is smaller than the number of ants and, in order to maximise the speedup, we must take into account the fact that. some schedules may require more simulation time than others. This may be due to several factors, such as the time required to find a

solution to the hydraulic equations. As well, system constraints, such as pressure constraints, may be violated early into the simulation, and, thus, the schedule would be considered infeasible without requiring a complete simulation. Whatever the reason, the fact is that some schedules will require more computation time than others and, hence, depending on the assignment of schedules to threads, some threads will take considerably more time to finish. The effect can be minimised by making the assignment dynamic. That is, instead of equally distributing the solutions among the threads, one solution is assigned to each thread and the rest of solutions are assigned as threads finish evaluating previous solutions. A higher ratio of solutions per thread would also tend to minimise the impact of particularly long simulations. Figure 5 shows a timeline of the execution of one iteration of the parallel ACO algorithm using 3 threads. In this example, ant 5 is assigned to the third thread because the other threads are still busy evaluating the other ants' solutions.

Another issue that affects the maximum speedup is the time required by the sequential code, which, by definition, is not reduced by using a higher number of parallel threads. If the stopping criteria of ACO are a maximum number of evaluations or a time limit, a smaller number of ants would increase the number of iterations of the algorithm. This, in turn, increases the time required by the sequential part of the algorithm.

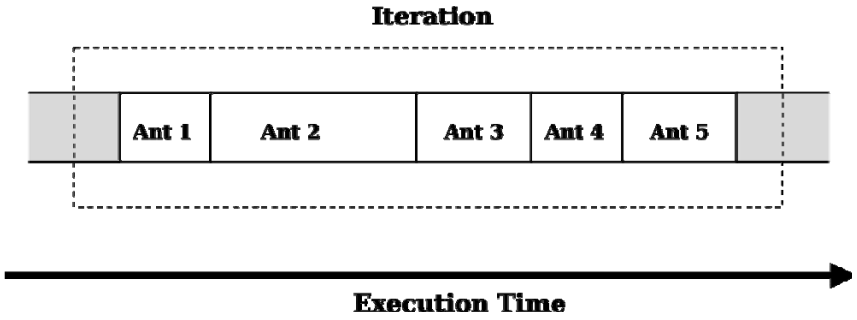

Figure 4. Execution schema of sequential ACO

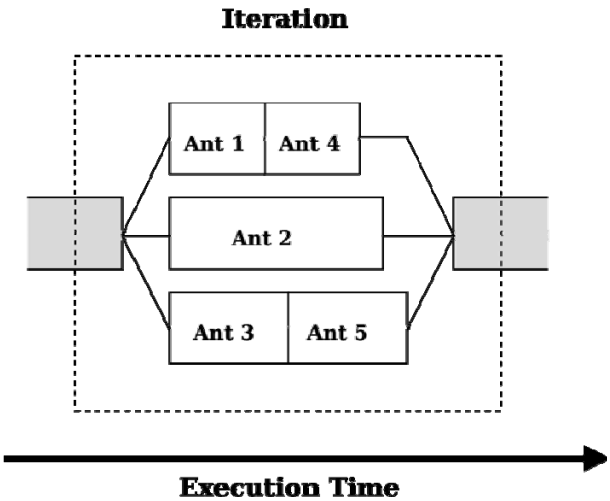

Figure 5. Execution schema of parallel ACO using 3 threads

5. EXPERIMENTS WITH PARALLEL ACO USING THREAD-SAFE EPANET

We empirically test the benefits of the parallel ACO approach described above. The underlying ACO algorithm is the one described by López-Ibáñez et al. (2008). This algorithm is modified to incorporate the parallel evaluation of solutions by dynamically assigning solutions to a number of threads. This algorithm is linked to the new thread-safe version of EPANET proposed in this paper. The goal of our empirical study is to analyse the performance, in terms of wall-clock time, of the algorithm. The performance in terms of solution quality is not considered here because the parallel variant generates the same sequence of solutions as the sequential ACO algorithm. In other words, given the same parameters, the quality of solutions generated by both algorithms is the same and only the computation time is smaller in the parallel variant. Our objective is to determine how much computation time is reduced by the use of multiple concurrent threads. In the previous section, we argued that the number of ants may have some effect on computation time. Thus, several values for the number of ants are tested.

Following the work by López-Ibáñez et al. (2008), we apply the parallel ACO algorithm to the optimisation of pump schedules in the Richmond network and the algorithm is stopped at 8,000 evaluations. Experiments are performed on the same 4-CPU machine described above. The implementation of the parallel ACO algorithm uses the C language and POSIX threads (Kerrisk, 2005). We conduct several runs of ACO with different number of ants (5, 10, 20, 40, 80) and different number of threads (1, 2, 3, 4, 5, 6).

The left plot in Fig. 6 shows the wall-clock time taken by ACO for each combination of parameters, while the plot on the right gives the corresponding speedup. The left plot shows that the sequential ACO (corresponding to using 1 thread) requires almost 2 hours of computation time. The differences in computation time for the sequential case are explained by the different sequence of solutions generated when using different number of ants. As explained above, for the same number of ants, the same sequence of solutions is generated independently of the number of threads. However, different number of ants will generate different results and, thus, there will be variations in the computation time. On the other hand, the speedup for each value of the number of ants is calculated with respect to the computation time required when using 1 thread and the same number of ants. Therefore, variations in the time required by the sequential algorithm do not translate into variations in the speedup. Thus, it is an interesting result that the speedup decreases with the number of ants. This indicates that parallelism is better exploited by using a high number of ants. The explanation for this result was already given in the previous section. The higher ratio of ants to threads allows a better utilization of the multiple CPUs and minimises the impact of schedules that require particularly long simulation time. We also hinted that a low number of ants will increase the time spent on the non-parallelised parts of ACO. However, our tests showed that less than 10 seconds of computation time are spent on these parts of the algorithm independently of the number of ants used. Thus, differences in the time spent on the sequential parts of the algorithm cannot have a significant influence on the observed differences in speedup. The overall, conclusion is that, in the parallel ACO algorithm, a higher number of ants reduces computation time. This is an encouraging result because, as reported by López-Ibáñez et al. (2008), the ACO algorithm obtains the best schedules, in terms of quality, when using a high number of ants.

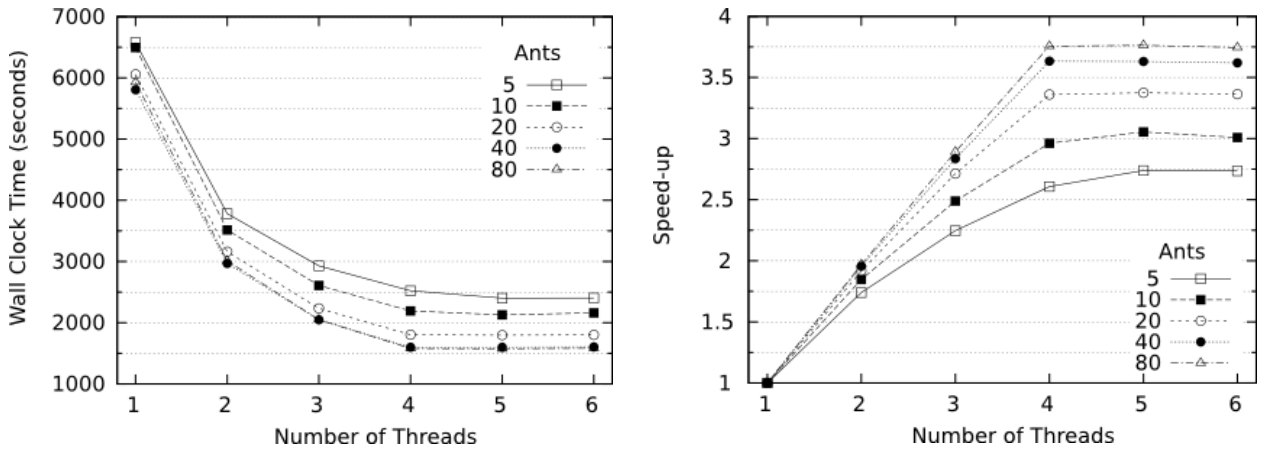

Figure 6. Runtime in seconds (left) and speedup (right) of ACO algorithm

6. CONCLUSIONS AND FURTHER WORK

We have discussed the limitations of EPANET for parallel computing. Thereby, we have proposed a thread-safe version of EPANET that can be used for the implementation of parallel optimisation algorithms for water engineering problems. We tested the new library on the problem of optimal pump scheduling in water distribution networks. In particular, we modified a state-of-the-art ACO algorithm to take advantage of multiple CPUs. The resulting parallel ACO algorithm shows significant savings in computation time when using 4-CPUs: from almost two hours to slightly less than half an hour. The performance in terms of quality is not affected by our modifications and the algorithm generates the exact same schedules. Moreover, from our experiments we conclude that using a large number of ants and more threads than CPUs usually gives the shortest computation time. The reduced time can be beneficial in several ways. In real-world applications, it will provide a faster response when an engineer is designing, testing or operating a water distribution network. As well, it enables to tackle larger and more complex networks that would require excessive computation time with a non-parallel approach. Researchers may also take advantage of the thread-safe EPANET, which does not depend on any parallel paradigm or implementation, to perform massive experimental tests on high-performance computers.

The experimental code of the thread-safe version of EPANET developed during this work is available at *http://sbe.napier.ac.uk/~manuel/epanetlinux* and we would like to encourage researchers to use it and improve it. Further research should apply the thread-safe EPANET to other optimisation algorithms, such as evolutionary algorithms, more elaborated parallel ACO approaches, and alternative parallelization techniques. Closely related to the pump scheduling problem are water quality problems, which can be also evaluated through EPANET. Water quality simulation requires even longer execution times, thus the benefits of extending EPANET to handle water quality simulation in parallel are potentially greater. Therefore, further developments should extend the thread-safe variant of EPANET to be useful for other optimisation problems.

Acknowledgement. This work was carried out under the HPC-EUROPA project (RII3-CT-2003-506079), with the support of the European Community - Research Infrastructure Action under the FP6 “Structuring the European Research Area” Programme.

7. REFERENCES

Atkinson, R., van Zyl, J.E., Walters, G.A., and Savic, D.A. (2000) “Genetic algorithm optimisation of level-controlled pumping station operation”. In *Water network modelling for optimal design and management*, Centre for Water Systems, Exeter, U.K., 79–90.

Dorigo, M., and Stützle, T. (2004) *Ant Colony Optimization*, MIT Press.

Kerrisk, M. (2005) “pthreads – POSIX threads” Section 7 of *Linux Programmer's* Manual. <http://www.linux-man-pages.org/man7/pthreads/> (accessed May 15, 2008).

López-Ibáñez, M., Prasad, T.D., and Paechter, B. (2008) “Ant colony optimisation for the optimal control of pumps in water distribution networks”. *Journal of Water Resources Planning and Management, ASCE,* (In press).

Maier, H.R., Simpson, A.R., Zecchin, A.C., Foong, W.K., Phang, K.Y., Seah, H.Y., Tan, C.L. (2003) “Ant colony optimization for design of water distribution systems”. *Journal of Water Resources Planning and Management, ASCE,* **129**(3), 200–209.

Rossman, L.A. (1999) “The EPANET Programmer’s Toolkit for analysis of water distribution systems”. *Proceedings of the Annual Water Resources Planning and Management Conference, ASCE*, Reston, USA.

van Zyl, J.E., Savic, D.A., and Walters, G.A. (2004) “Operational optimization of water distribution systems using a hybrid genetic algorithm”. *Journal of Water Resources Planning and Management, ASCE*, 1**30**(2),160–170.