USING PARALLEL COMPUTING TO INCREASE THE SPEED OF WATER DISTRIBUTION NETWORK OPTIMIZATION

Ehsan Roshani [1], Yves Filion [2]

1. Ph.D. Student, Queen's University, Kingston, ON, CANADA, ehsan.roshani@ce.queensu.ca
2. Assistant Professor, Queen's University, Kingston, ON, CANADA, yves.filion@civil.queensu.ca

ABSTRACT

A high level of computational power is often needed to solve real-world engineering problems. The introduction of multi-core Central Processing Units (CPUs) opened a new window for small companies and academic researchers to benefit from high performance computing. In this paper, the Message Passing Interface (MPI) and Task Parallel Library (TPL) are discussed and applied to find the optimal rehabilitation plane for a real world water distribution system. An optimization framework based on the Non-dominated Sorting Genetic Algorithm II (NSGA-II) is developed and the time required to evaluate all solutions in the last generation were measured for increasing number of cores in both approaches and compared.

INTRODUCTION

The introduction of multi-core Central Processing Units (CPUs) in 2005 revolutionized the software industry. Since then the demand for multi-core computing has increased and it has become progressively less expensive. For example, one can now purchase a desktop computer with 24 cores for less than $3,000. Suddenly, parallel computing has become available to a much broader range of users, including consulting engineers, through new programming interfaces and toolboxes.

Increasingly, engineers are turning to optimization to plan the design, expansion, and rehabilitation of water distribution systems. High computational power is required to find the optimal solution in real networks which tend to have a large number of pipes and components and thus impose long run times on a network solver to evaluate the hydraulic and/or water quality conditions. Optimization is one of the best examples of algorithms in which tasks could be run in parallel. In fact, optimization algorithms are known as "attractively" parallel algorithms (Microsoft Corporation 2008). Since these algorithms are computationally-intensive, they are best solved on multi-cores computers or a cluster machine (Microsoft Corporation 2008).

There are several parallel programming interfaces available in the market and most of them are open source. The objective of this paper is to present the parallelization of code to solve the multi-objective optimization of a real-world water distribution network in Amherstview, Ontario, Canada. Two well-known open source interfaces, TPL and MPI.net, were used to solve the multi-objective problem, with significant increases in computational speed. Both methods will be introduced and discussed in the paper, and the pseudo-code implementation used to parallelize the optimization algorithm will be discussed.

METHODOLOGY/ PROCESS

Parallel Processing

Parallel processing is the ability to do multiple operations or tasks simultaneously. This term is frequently used in the human cognition context to discuss brain functionalities such as vision. The simultaneous use of several CPUs to execute a task or a program is called parallel processing. The idea is to divide a computational task among multiple CPUs to increase the speed of computations. There are well-known problems in water distribution systems analysis that can benefit from parallel computing. Examples include Monte Carlo Simulation, evolutionary algorithms (e.g., genetic algorithms), ant colony optimization, and particle swarm optimization. There are several tools and

programming interfaces available such as Task Parallel Library (TPL) and Message Passing Interface (MPI) to allow users to perform parallel processing operations with a multi-core cluster machine. These tools will be introduced and discussed in the follow section.

Message Passing Interface (MPI)

Message Passing Interface (MPI) is a language-independent communication protocol used to perform parallel computing operations on a multi-core machine. The MPI interface handles all communication protocols and semantic specifications between CPUs in a multi-core machine. MPI was introduced in 1992 by Aoyama and Nakano (1999). Most MPI implementations are open source and consist of a set of routines which are called directly from C, C++, FORTRAN, C#, Java, and Python code.

MPI programs are usually written as Single Program Multiple Data (SPMD) applications. In this application, each CPU (or core in a multi-core machine) is running the same software program but with a different set of data. For example, consider how a single-core and multi-core machine would perform genetic algorithm operations in the least-cost design of a water distribution system. With a population of 1,200 chromosomes, a single-core machine would have to simulate the hydraulic conditions of all solutions (or chromosomes) with EPANET (Rossman 2000). In contrast, a multi-core machine made up of 24 cores would divide the EPANET simulations equally among its 24 cores and so each core would only be expected to perform 50 simulations. Although it is not necessary to divide computational load equally among cores, it often helps achieve a more synchronous code.

MPI supports the SPMD application by allowing the user to execute the same software program on different machines or running several copies of the program (process) on the same machine or a combination of both methods. The only difference among processes is that a specific rank is uniquely assigned to each process. Using this rank, MPI processes can communicate and can behave differently. In the GA example, the process with the first rank (rank 0) is responsible for dividing the population into several smaller sub-populations and for sending them to other processes to evaluate.

There are several types of communications required in parallel processing: (i) one-to-one, (ii) one-to-all, and (iii) all-to-all. All of them are supported in MPI. One-to-one communication is also called point-to-point which is the most basic form of communication. This communication mode allows the process to send a message (e.g., data) to another process. Each message has a source and a target process identified by their rank and it has a tag which is normally used to identify the type of the data being passed. Pseudo code for this mode of communication can be structured as follows:

```
Comm.Send(Data, Target, Tag)
Comm.Receive(Data, Source, Tag)
```

In which Comm is the communicator object, Data is the variable being passed to the target process. Target, is the process rank which receives the data. Source is the process rank which sends the data, and Tag is the integer number which could represent the data type.

Consider a process which in its first rank sends some data to the next process. Pseudo code for this program is written as follows:

```
If(Comm.rank == 0)
  { Comm.Send("Test", Comm.rank + 1 , 0)}
Else
 {Comm.Receive(Data, 0, 0 }
```

In this code, the process with rank 0 (the root process) sends a string “Test” to the ranks 1 and rank 1 receives “Test” form rank 0 and stores it in the variable named “Data”. One-to-all communication or “Broadcast” takes a value and sends it to all other processes. This mode of communication takes two arguments; the first one contains the value to send at the root process and it stores the transmitted value in other processes. The second argument is the rank of the root process. For instance, in the pseudo code indicated below, a variable Data will be broadcast to all other process from the root process which has a rank equal to 3.

```
Comm.Broadcast(Data, 3)
```

There are other versions of Broadcast such as “Scatter” which simply divide the data into smaller pieces and then send each piece to one process. The root process provides an array of values in which the i^{th} value will be sent to the process with rank ith allowing the root to distribute different tasks to each of the other processes.
In the contrary to the Broadcast mode of communication, the “Gather” mode of communication collects data provided by all other processes in the root process. It has two arguments; the first one is the value which is supposed to be collected and the second is the root process rank. In the following pseudo code, a variable named “Data” is collected from all processes and it is stored in the array called “TotalData” in the process with rank 0:

```
Array[] TotalData = Comm.Gather(Data,0)
```

The all-to-all communicator transmits data from every process to every other process. The following pseudo code demonstrates the functionality of this mode of communication:

```
Array[] Results = Comm.AllToAll(Data[])
```

Using these modes of communication, one can easily distribute the computational load for algorithms such as genetic algorithms and Monte Carlo Simulation. The following is the pseudo code required to evaluate all of the solutions in one generation in the genetic algorithm example (mentioned above) to solve the least-cost water distribution network:

```
Comm.scatter(Solutions[], 0)
For I = 0 to n
   Result[I] = evaluate(Solutions[I])
Next I
Array[] Resutls = Comm.Gather(Result,0)
```

From the root process, the first line of the pseudo code sends solutions to other processes. In the loop, each solution is evaluated by the function call “evaluate” and the result for each solution is saved in the array called “Result”. The results from each process are then sent back to the root process and stored in the array called “Results”.
It is obvious from the pseudo code above that this is the MPI responsibility to make sure all the data were transmitted correctly and all the process are working fine and synchronously. But it is still the

programmer's job to assign the variables to send through MPI functions and to handle different processes. Therefore writing a code with MPI needs to have a good understanding of data structure and robust knowledge of algorithm development. MPI codes are designed to work in every environment but they are mainly optimized to work on clusters. Moreover, MPI codes can run on ordinary machines without high performance computing capabilities and multiple cores.

Task Parallel Library, TPL

Task Parallel Libraries (TPL) provide the programming infrastructure to use multi-core machines to improve the computational performance of a program. TPL is the task parallelism component to the '.Net' framework developed by Microsoft Research and Microsoft Common Language Runtime (CRL). The library was released in version 4.0 of the '.Net' framework. It exposes parallel constructs like Parallel "For" and "ForEach" loops using regular method calls (D. Leijen and J. Hall 2007).
TPL is designed to use with multi-core machines in the managed codes and it does not support distributed memory architecture usually found in clusters. TPL allows users to develop their code in a sequential manner while still providing the benefit of parallel processing. It should be noted that the functions which are supposed to run in parallel should be independent from each other.
Unlike MPI which works with most programming frameworks, TPL is designed to work within the '.Net' framework. It can be used within all '.Net' languages including C# and Visual Basic. All parallelism is expressed using general call methods. For instance, consider the following sequential code to evaluate all solutions in a generation in GA example (mentioned before) written in C# language:

```
for (int i = 0; i < 100; i++) {
  results[i] = evaluate(Generation[i]) ;
}
```

Since evaluating each solution is independent from others, one can use TPL to benefit from potential parallelism as follows:

```
Parallel.For(0, 100, delegate(int i) {
  results[i] = evaluate(Generation[i]) ;
});
```

Note that "Parallel.For" is a normal call method. It takes three arguments in which the first argument is where the counter begins, the second one is where the counter ends and the last one is the delegate expression which captures the unchanged code from previous sequential code. TPL divides the computational load between available cores and ensures that parallelism is employed to run the program code. The demonstrated codes are the actual codes required to convert a sequential program to a parallel program. These codes illustrate the simplicity of using TPL. TPL includes sophisticated methods to handle dynamic load distribution as well but these methods are beyond the scope of this paper.

PARALLELISIM IN PRACTICE

To examine the effectiveness of MPI and TLP in practice, a water distribution system rehabilitation optimization problem was defined and a genetic algorithm was used to solve it. The problem is developed to find network rehabilitation solutions that minimize the capital costs and minimize the

operational costs simultaneously for a planning period of 20 years. Equation (1 and 2) shows the objective functions in mathematical format.

$$Obj1 = Min(CC) = \sum_{t=0}^{20} \sum_{p=1}^{np} \left(RC_{t,p} + DC_{t,p} + LC_{t,p} + NP_{t,p} \right) \quad (1)$$

$$Obj2 = Min(OC) = \sum_{t=0}^{20} \sum_{p=1}^{np} \left(LkC_{t,p} + BC_{t,p} \right) + \sum_{t=0}^{50} EC_t \quad (2)$$

In which CC is the capital costs, OC is the Operational cost, t is time in year, p is the pipe number, $RC_{t,p}$ is replacement cost for the p^{th} pipe in the t^{th} year, $DC_{t,p}$ is pipe duplication cost, $LC_{t,p}$ is Lining cost, $NC_{t,p}$ is new pipe cost, $LkC_{t,p}$ is the cost of lost water, $BC_{t,p}$ is break repair cost, and finally EC_t is the energy cost. The optimization was subject to operational velocity and pressure constraints. The goal was to find the time, type and the place of rehabilitation activities in the network. Available rehabilitation technologies considered in this problem were new pipe installation, replacement of old pipes, pipe duplication, and pipe re-lining (cement-mortar lining).
The code was developed based on a specific multi-objective genetic algorithm named the non-dominated sorting genetic algorithm version II (NSGA-II) (Deb 2002). The NSGA-II was combined with the EPANet toolkit (Rossman 2000) to simulate the hydraulics of the Amherstview water distribution network in Eastern Ontario, Canada. This network provides drinking water to the Town of Amherstview and the Town of Odessa which have a combined population of approximately 15,000 people (Roshani et al. 2012). The all-pipes model (400 links) of the Amherstview system (Fig. 1) consists of existing pipes and new pipes in future-growth areas.

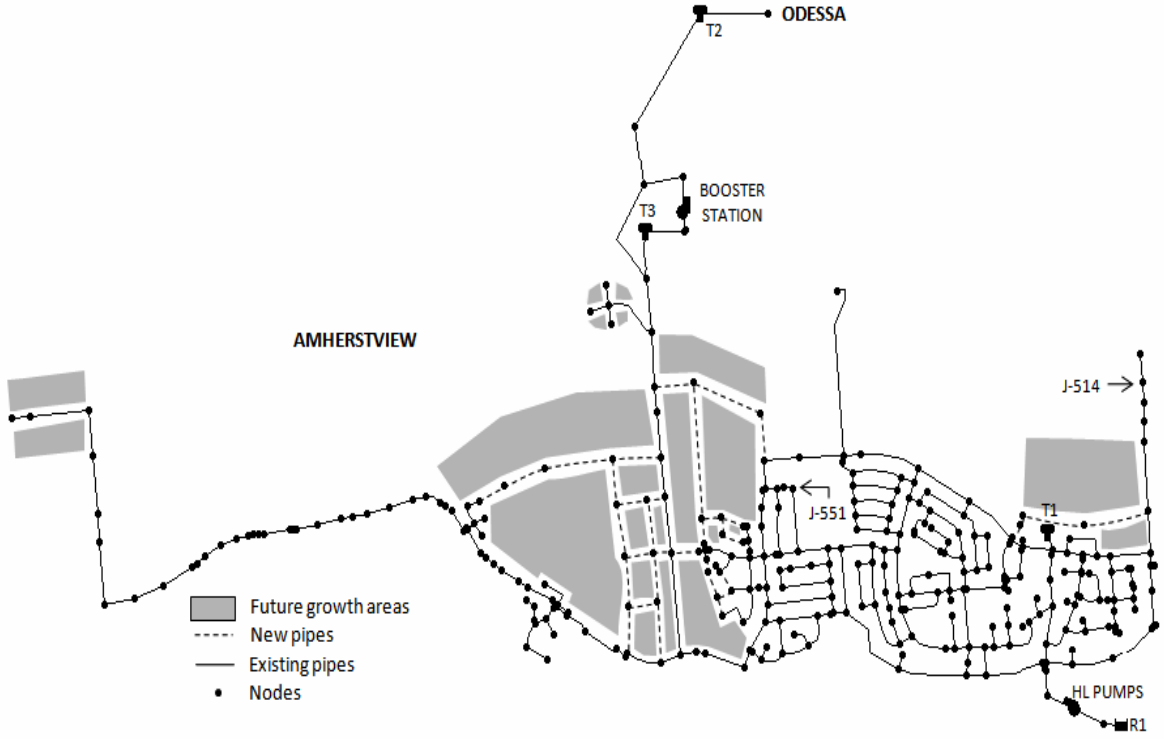

Figure 1: Amherstview water distribution system.

In the optimization application, pipe rehabilitation events were applied in each year of the 20-year study period. For this reason, the hydraulic conditions in the Amherstview network were simulated in each year of the 20-year period. Energy costs were characterized by running a 24-hour extended period simulation in each year of the 20-year period. A total of 20 extended period simulations were performed for each rehabilitation solution considered in the optimization. Additional steady-state

simulations were run each 10 years to characterize pressures during maximum day demand and fire flows at three critical locations. The details of the hydraulic model, pipe break model, leak model, and pipe aging model are discussed in Roshani and Filion (2012).
The parallelized optimization algorithm was solved with both TPL and MPI interfaces and applied to the Amherstview network. The problem was solved with an increasing number of cores (1, 2, 4, 6, 8, 10, 12, 16, 20, 24, 48, 60, 80 and 120) to show the effect of parallelization in increasing computational speed. A GA population of 1,200 chromosomes was adopted in the optimization problem. Since GA operators such as crossover, mutation, and selection are random operators, it is impossible to obtain the same exact chromosomes (solutions) in different generation therefore computational load in each scenario would be different. The authors ensured that each run (with a specified number of cores) had a similar computational load by comparing the run time of the 1000th generation in the optimization problem.

RESULTS/ OUTCOMES

A '.Net' implementation of MsMPI was used and C# was chosen as the preferred programming language in the optimization application. A graphical user interface for data entry and GA progress monitoring was developed (Fig. 2). Task Parallel Library in Visual Studio 2010 was used directly in C#. It should be noted that although Visual Studio is proprietary but both of MPI.net and TPL are free and could be used with other .net language including Visual Basic.

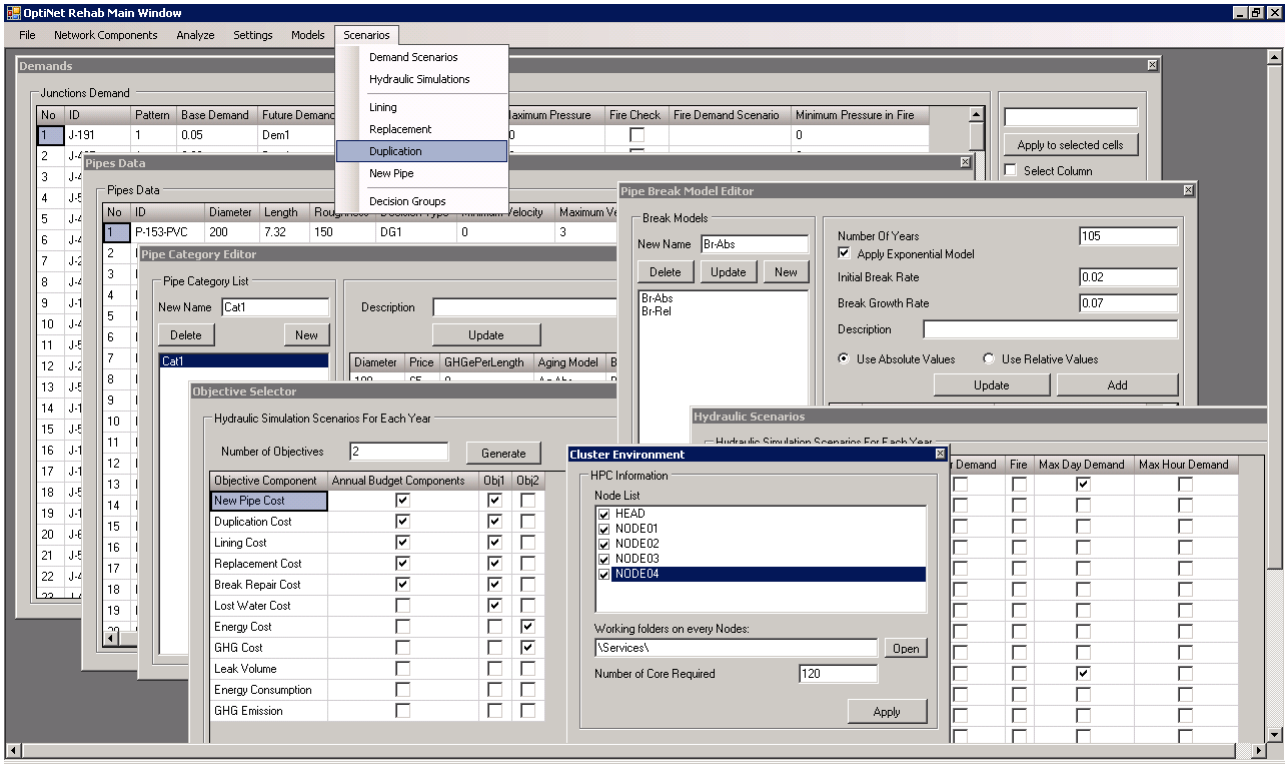

Figure 2: Part of the Graphical User Interface, GUI, developed to prepare the optimization models and to control the cluster machine

A cluster machine with five nodes was used to run the optimization program. Each machine has 2 sockets and each socket has 12 cores. Therefore each node has 24 cores and there are a total of 120 available cores. Windows HPC Server 2008 R2 serves as the operating system on all of the machines. Two networking switches have been used to transfer data. A 10 Gigabit switch serves as an application network between nodes exclusively for MPI application. And a one gigabit switch was used to manage nodes by the operating system. Separating application network and operating

network reduced the networking overload in the cluster (Juethner 2011). It should be noted that these network switches were used in MPI model and not in TPL model.
The run times to evaluate all 1,200 solutions in the 1000th generation were measured and are indicated in Figure 3. Solutions were evaluated with 1 to 120 cores with the MPI model and with 1 to 24 cores with the TPL model. As previously mentioned, the TLP model is designed to utilize multi-core CPUs on one machine and since there are only 24 cores in one node, it was not possible to run the TLP model with more than 24 cores.

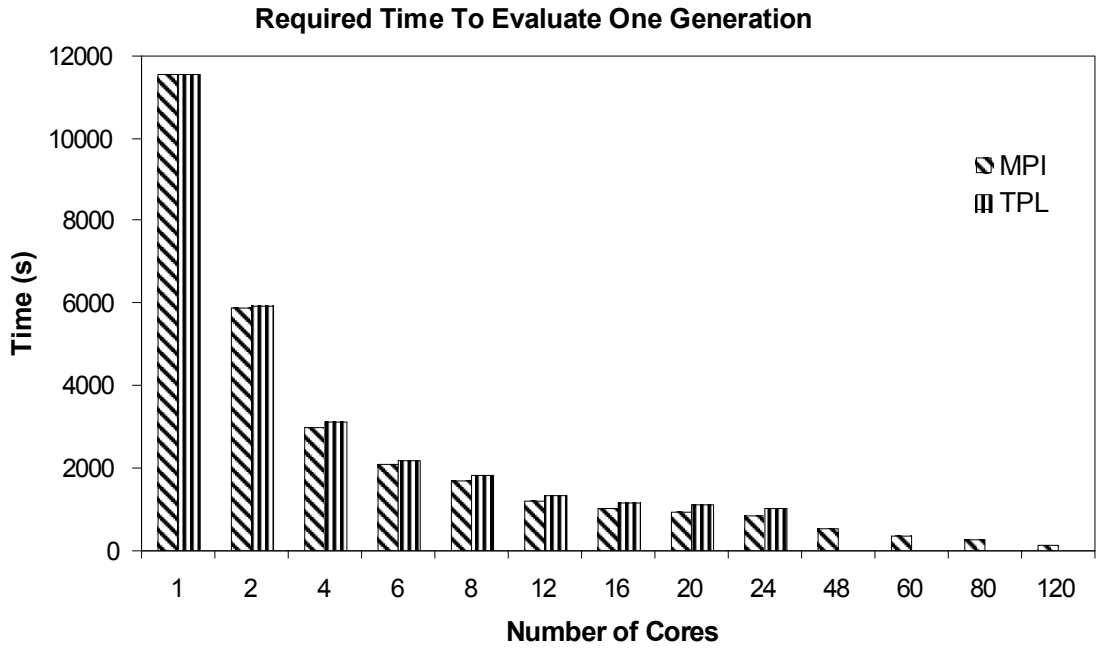

Figure 3: Required time to evaluate all solutions in one generation with a different number of cores.

As shown in Figure 3, the MPI and TPL models used with one core (sequential mode) produced run times of 11,572s and 11,575s. Doubling the cores to 2 and then to 4 decreased the run time to almost half and then to a quarter of the run time of a single core. Figure 4 indicates that the run time observed with 24 cores was almost 14 times less than the run time observed with a single core. There is a small difference between MPI and TPL run times owing to difference in their architecture. This difference increases by adding the number of cores which is probably caused by increasing the operating system overload.
Although it is expected that using 120 cores should reduce the required time by a factor of 120, this is not likely to be observed in practice. This is mainly because of networking and operating system overload. Our platform accelerated the running time by a factor of 76. A computing time of 151 seconds was needed to evaluate one generation. A computing time of 1.75 days was needed to simulate 1,000 generations to find the near-optimal Pareto front as compared to a computing time of 134 days had sequential programming been used.

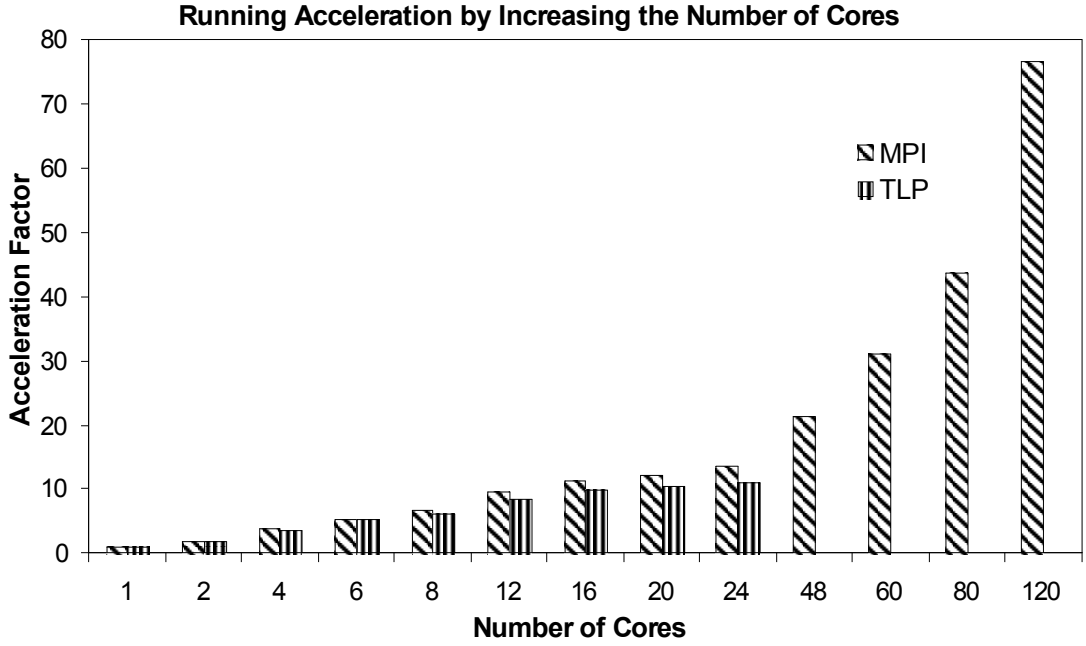

Figure 4: Running acceleration achieved with the parallelism.

SUMMARY

The paper presented a method of code parallelization to solve the multi-objective optimization of a real-world water distribution network in Amherstview, Ontario, Canada. Two well-known open source interfaces, TPL and MPI.net, were used to solve the multi-objective problem, with significant increases in computational speed. Both TPL and MPI methods were introduced, discussed, and compared against each other and against a serial computing control example. Two models were developed based on established parallelism concepts. The advantages and disadvantages of each method were discussed. Although it is more difficult to develop an algorithm for MPI codes, such a code can be run on a cluster of machines with almost unlimited computational power. While the TLP method is simpler to develop, it is limited to one machine. The results indicated that using MPI and running the model on 120 cores could speed up the calculation time by a factor of 76 as compared to sequential programming. TLP can speed up calculations by a factor of 11 on a singe machine with 24 cores.

REFERENCES

Alvisi S., Franchini M., (2009) "Multiobjective Optimization of Rehabilitation and Leakage Detection Scheduling in Water Distribution Systems", Journal of water resources planning and management, 135(6): 426-436

Aoyama Y., Nakano J., (1999) "RS/6000 SP: Practical MPI Programming", IBM international technical support organization, www.redbooks.ibm.com, 238p

Deb, K., Pratap, A., Agarwal, S., Meyarivan, T. (2002). "A fast and elitist multiobjective genetic algorithm: NSGA-II." IEEE Transactions on Evolutionary Computation, 6(2), 182-197.

Juethner, K., (2011) "Dramatically Improve Compute-Intense Applications in the Supercomputing Cloud", White paper, COMSOL Inc.

Leijen D., Hall J., (2007) "Optimize Managed Code For Multi-Core Machines", D, MSDN Magazine Issue 2007 Oct.

Microsoft Corporation, (2008), “Windows HPC Server 2008, Using Windows HPC Server 2008 Job Scheduler”, Published: June 2008, Revised September 2008.

Roshani, E., MacLeod, S.P., Filion, Y.R. (2012) “Evaluating the Impact of Climate Change Mitigation Strategies on the Optimal Design and Expansion of the Amherstview, Ontario Water Network: A Canadian Case Study” Journal of Water Resources Planning and Management, ASCE, Vol 138, Issue 2, P 100-110.

Roshani, E., Filion, Y.R. (2012), “Event based network rehabilitation planning and asset management” 14th annual Water Distribution Systems Analysis conference (WDSA), September, 2012, Adelaide , Australia

Rossman, L. A. (2000). EPANET2 User’s Manual. US EPA, Washington, D.C.

Shamir, U., Howard, C. D. D., (1979), “An analytic approach to scheduling pipe replacement.” J. Am. Water Works Assoc., 71(5), 248–258.

Sharp, W. W., Walski, T. M., (1988), “Predicting internal roughness in water mains”, J. Am. Water Works Assoc., 80(11), 34–40.