

# CityDrain3 - Parallel Computing in Conceptual Urban Drainage Modelling

master thesis in computer science

by

**Gregor Burger**

submitted to the Faculty of Mathematics, Computer  
Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements  
for the degree of Master of Science

supervisor: Dr. Hans Moritsch, Institute of Computer  
Science

**Innsbruck, 17 November 2009**



# **Certificate of authorship/originality**

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Gregor Burger, Innsbruck on the 17 November 2009





## Abstract

The motivating factor for this work is the interest of engineers in long-term effects of urban drainage systems which leads to complex and time consuming simulations. This thesis describes how the runtime of urban drainage modelling simulations can be reduced by applying parallel computing on multi-core computing systems. The reader is introduced to the basic terms and concepts used in urban drainage modelling and parallel computing. Following the introduction an overview of the modelling concepts that provide the mathematical foundations of urban drainage modelling is given.

The thesis describes three parallel strategies that have been developed in order to distribute the computations on several processor cores. The three strategies are: the flow parallel strategy, the pool pipeline strategy and the ordered pipeline strategy. These strategies have been developed with a view on the structure of urban drainage models and have been implemented within a specific simulation environment, called CityDrain3. This urban drainage software tool was used to demonstrate the runtime and speedup effects of the three strategies. A number of different urban drainage systems were analysed to detect shortcomings and limits of the strategies. The benchmark results reveal that the ordered pipeline strategy is capable of, at times significantly, reducing the runtime of all tested sewer systems.

## Kurzfassung

Der motivierende Faktor für diese Arbeit ist das Interesse von Ingenieuren an Langzeitstudien von Siedlungsentwässerungssystemen welche zu komplexen und zeitintensiven Simulationen führen. Diese Masterarbeit beschreibt, wie die Laufzeit für Simulationen von Siedlungsentwässerungsmodellen reduziert werden kann, indem parallele Datenverarbeitung auf Mehrkernsystemen angewendet wird. Der Leser wird in die Grundlagen der Siedlungsentwässerung und der parallelen Datenverarbeitung eingeführt. Weiters wird eine kurzer Überblick über die verwendeten Modellkonzepte der städtischen Entwässerungsmodellierung gegeben.

In dieser Arbeit wurden drei parallele Strategien entwickelt, um die Berechnungen auf mehrere Prozessorkernen zu verteilen. Die drei Strategien sind: die "flow parallel" Strategie, die "pool pipeline" Strategie und die "ordered pipeline" Strategie. Diese Strategien betrachten dabei die Strukturen der urbanen Entwässerungsmodelle und wurden in einer speziellen Simulationsumgebung, genannt CityDrain3, implementiert. Dieses Werkzeug wurde verwendet um das Laufzeit- und das Beschleunigungsverhalten der Strategien zu demonstrieren. Verschiedenste Kanalsysteme wurden analysiert, um die Strategien auf ihre Stärken und Schwächen zu untersuchen. Die Resultate der Benchmarks zeigen, dass die "ordered pipeline" Strategie in allen untersuchten Systemen die Laufzeit der Simulation teils erheblich reduziert.



# Contents

<b>1. Introduction</b>	<b>1</b>
1.1. Motivation . . . . .	1
1.2. Urban Drainage . . . . .	2
1.3. Modelling and Simulation . . . . .	2
1.3.1. System . . . . .	4
1.3.2. Model . . . . .	5
1.3.3. Simulation . . . . .	5
1.4. Parallel Computing . . . . .	6
1.4.1. Overview . . . . .	7
1.4.2. Parallel Algorithm Design . . . . .	8
1.4.3. Communication . . . . .	9
1.4.4. Parallel Architectures . . . . .	12
1.4.5. Multi-core Processors . . . . .	14
1.4.6. Parallel Performance Metrics . . . . .	15
1.4.7. Challenges of Parallel Programming . . . . .	16
<b>2. Urban Drainage Modelling</b>	<b>19</b>
2.1. Modelling Concepts . . . . .	19
2.1.1. Rainfall Runoff . . . . .	20
2.1.2. Hydraulic Transport . . . . .	21
2.2. Integrated Urban Drainage Modelling . . . . .	23
2.3. Conceptual Modelling . . . . .	24
2.3.1. State Space Modelling . . . . .	24
2.3.2. Sewer Structure . . . . .	25
<b>3. Methods and Implementation</b>	<b>27</b>
3.1. Design . . . . .	28
3.1.1. Node . . . . .	28
3.1.2. Model . . . . .	30
3.1.3. Simulation . . . . .	31
3.1.4. Flow . . . . .	32
3.2. Sequential Simulation Run . . . . .	32
3.3. Parallel Implementation . . . . .	34
3.3.1. Flow Parallel Strategy . . . . .	35
3.3.2. Pool Pipeline Strategy . . . . .	37
3.3.3. Ordered Pipeline Strategy . . . . .	39

3.4. Shared Flow . . . . .	45
<b>4. Results</b>	<b>49</b>
4.1. Benchmarked Systems . . . . .	49
4.1.1. Sequential Sewer System . . . . .	50
4.1.2. Parallel Sewer System . . . . .	50
4.1.3. Treelike Sewer Testing System . . . . .	51
4.1.4. Real World Sewer System of Innsbruck . . . . .	52
4.2. The Benchmark Environment . . . . .	54
4.3. Performance Tools . . . . .	56
4.4. Results for the Core2Quad CPU . . . . .	57
4.5. Results for the i7 CPU . . . . .	66
4.6. Shared Flow Comparisons . . . . .	74
4.6.1. Results Core 2 Quad . . . . .	74
4.6.2. Results i7 . . . . .	75
4.6.3. Conclusion . . . . .	76
<b>5. Conclusion</b>	<b>79</b>
<b>Appendices</b>	<b>81</b>
<b>A. CityDrain3 Manuals</b>	<b>83</b>
A.1. Users Manual . . . . .	83
A.1.1. Terms and Concepts . . . . .	83
A.1.2. Starting CityDrain3 . . . . .	87
A.1.3. Writing a Model . . . . .	89
A.1.4. Using cd3modelgen.py . . . . .	95
A.1.5. plugindoc Application . . . . .	98
A.2. Programmers Manual . . . . .	101
A.2.1. Compiling CityDrain3 . . . . .	101
A.2.2. Design Overview . . . . .	102
A.2.3. Extending CityDrain3 . . . . .	106
<b>Submitted Papers</b>	<b>119</b>
<b>List of Figures</b>	<b>137</b>
<b>List of Tables</b>	<b>143</b>
<b>Bibliography</b>	<b>145</b>

# Chapter 1.

## Introduction

### 1.1. Motivation

The aim of CityDrain3 is to speed up the simulations of urban drainage modelling (UDM) by using the currently untapped parallel computing power of modern desktop machines. At the time of writing, consumer class CPUs are composed of up to four cores which is doubled to eight, for a dual CPU system.

Civil engineers are interested in the long-term effects of Urban Drainage systems [DJ97, CKM<sup>+</sup>01]. These long-term effects must be covered by simulations. The runtime of a simulation depends on the time span a simulation wants to cover. If a simulation software is fast longer simulation times are possible in less time.

Another factor influencing the simulation run-time, is the complexity of the system under research. By using parallel computing the complexity of a modelled system can be increased without increasing the simulation run-time.

UDM involves the process of gathering information of a real system, replicating the system by using model formulations and calibrating the reduced view of the system, so that the measured system behaviour fits the numerical modelled system. The last process can be done manually or automatically. In automatic calibration a certain algorithm for determining the best model parameters is used and the outcomes of the simulation run are used to determine the further calibration steps. More parameters in a complex system require more runs until the system is calibrated.

Uncertainty analysis with Monte Carlo simulations are computing intensive, parallel computing is said to help reducing runtimes of these simulations [KGNC02]. CityDrain3 reduces the runtime of a single run and therefore allows to apply Monte Carlo simulations to more complex systems.

Integrated Urban Drainage simulations, described in Chapter 2.2, require an

extensive amount of computing time. Muschalla [Mus08] states several possibilities to reduce the amount of computing time it takes for running IUDM simulations. Although parallel execution of the model simulation is not stated in [Mus08] it fits perfectly into the list of computing time reductions.

## 1.2. Urban Drainage

Artificial drainage systems are essential in urban areas because of the interaction between humans and the natural water cycle. Humans need a great amount of water that is modified while using it in households or industry processes. This denoted waste water includes health harming substances which must be flooded out of the urban area. In order to save the environment the water must be treated before it can get back into the receiving water. Urban areas tend to develop high amounts of impermeable surface areas. Effects of these covered areas are that rainwater may cause harm to the urban area and its inhabitants.

Urban drainage systems have the functionality of draining the following two water types:

1. *Waste water* is water extracted from natural resources used for various kinds, including, drinking, washing, flushing and industrial usages. *Waste water* must be drained in order to reduce health risk and reducing or prohibiting the spread of diseases. *Waste water* contains dissolved material, fine solids and larger solids.
2. *Storm water* is rain that must be drained in order to prevent flooding and damage of urban areas and other inconveniences. *Storm water* contains some pollutants from the air and run-off of built-up areas.

Beside protecting humans from the harm of *waste-* and *storm water* urban drainage is responsible for keeping the environmental pollution caused by urban areas low. This means that the Waste Water Treatment Plant (WWTP) is an essential part of an urban drainage system. [BD04, Guj06]

## 1.3. Modelling and Simulation

This section tries to inform the reader about whats behind modelling and simulation, because it is essential for understanding the rest of the presented material in this thesis.

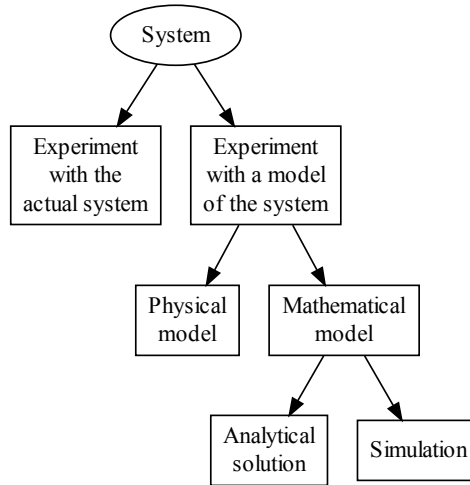


Figure 1.1.: Ways to study a system.(redraw from [LK97])

Modelling and simulation is as old as the human race itself. Even before computers have existed humans tried to theorize, review, and discuss certain ideas to play with them and in the end realize the ideas in some way or another. Modelling and simulation allows to think about a complex system or idea in an abstract way. Sometimes it is not possible to grasp all aspects of a system, sometimes it is just cheaper to model in the first place. It may even be possible that experiments with a system destroy or influence the system itself. [Bos92]

Figure 1.1 shows in which ways a system can be studied and where experiments are able to be carried out. In modelling a system is a collection of entities which interact and have a distinct goal. If one wants to gain insight of a *system* to understand it better or even predict its future states, he can choose between experimenting with the actual system, or *model* the system with regards to the interests of the system. A model is an abstracted replication of a system that allows more ways to experiment and study the system, e.g. destroy it. We can use a physical model, for example a basin full of water to experiment with waves. A model can also be manifested with logical relations and mathematical formula. These kind of models are called mathematical models. Experiments with mathematical models can either be made by using analytical solutions, or if the formula are too complex, numerical solutions are obtained by running simulations.

Modelling is at the heart of the human race, as a child we start by modelling the real world of grownups in order to understand and train for our future lives,

e.g. girls have tea parties with their mates and boys play with model cars. Later in our life we use several models to ease our everyday life, e.g. a map is a model of our overly complex road system, it is an abstracted way to plan (simulate) a way we are going to drive by car. [Bos92]

The vast range of application of modelling formed some common words which allow to identify the characteristics of the modelling process. Some terms have been coined which allow to talk about the certain aspects of modelling and simulation.

### 1.3.1. System

A *system* is the process of interest, the objective of a modelling and simulation process. It is a collection of entities which interact and are also independent from other objects in the world. A *system* can be described by the following aspects:

- a system has a function, that means it has a scope which we can recognise.
- a *system* is build of a collection of *system elements* which are related to each other.
- a *system* has an identity which is lost if we divide the system into separate parts.

These criteria of a *system goal*, *system structure*, and a *system identity* allows us to differ a *system* from plain simple objects. For example a sand heap is not a system because it is still a sand heap if we remove halve of the heap. [Bos92]

A *system* is in a certain state at a certain time, states are a collection of variables. A *system* can either be discrete, which means that the states of the system change at certain times, e.g. the queue of cars in a traffic jam has a certain number of cars. A continuous system is one where the states of the system change gradually, e.g. the position of an air plane heading a certain route. [LK97, Guj08]

The easiest and most precise way to study a system is to experiment with the system itself. But often that is not feasible (e.g. studying the artificial ecosystem of a mixed wood would take decades), not wanted (e.g. large scale experiments with the earth atmosphere) or simply not possible (e.g. landing a space shuttle on the moon). Therefore we build a *model* of such a system so that it is possible to state some predictions despite the unapproachability of the real world system.



### 1.3.2. Model

A *model* can be a *physical* one, which is in most cases a simplified or minimised copy of the original. A map of road systems or a car in a crash test are examples of *physical models*. Another possibility to build a *model* is to use a mathematical description. In this case we talk about a *mathematical model*. These kind of models are especially easy to experiment with by using modern computer technology. The accuracy of such models is constrained by the amount available memory and computing power which gets raised year by year. Because of this mathematical models tend to get detailed year by year and modelling is used in more cases. Ten years ago nobody thought of using computers to simulate car crashes or the wheater.

A model can have various characteristics:

- *deterministic - stochastic*: If a model is free of random parameter or state changes, we speak about a *deterministic* model. Such a models outcome is defined by the mathematical relations, the input and the state of the model. If on the other hand some process involves sudden changes of unknown outcome, we talk about a *stochastic* model.
- *time variant - invariant*: If a model reacts on the same input, and the same environmental parameters exactly the same at a later time, we talk about a *time invariant model*. A human being is *time variant* because he behaves different when he gets older.
- *time discrete - continuous*: Natural systems are always time continuous, but computer systems are only able to calculate in discrete time steps. Although the time steps can be made small enough to talk about a continuous time.
- *structural discrete - continuous*: A *model* is structural discrete if we can calculate parts of the system independently. If we have a large interaction between the components of the system differential equations are needed to be defined. Differential equations are strong signs that a model is structural continuous. [Bos92]

### 1.3.3. Simulation

After a model is found that answers certain scientific questions the next step is to solve the mathematical formulations that represent the system. Often it is possible to obtain informations of a simple model by solving its mathematical formulations analytically. Such a simple model would be newton's first and

second laws of motion, that describe a flying bullet. It is possible to describe the point and time of the impact analytically if it is known where the bullet came from, where it heads to and how fast it is. A model may be too complex to solve it analytically, or no analytical solution of the problem exists. In such a case the model must be studied by using numerical *simulations*. [LK97]

## 1.4. Parallel Computing

Parallel Computing (PC) is a computer science discipline where researchers try to find algorithms for certain problems that are able to run parallel.

As can be seen later parallel computing is not an easy task, neither finding parallel algorithms nor their implementations. The main advantage of using/finding well working parallel algorithms is that one can add extra hardware and find his program working faster. This is especially the case when computational intensive problems are needed to be solved, that would otherwise take an unfeasible amount of time to finish.

Beside the algorithms that need huge computing power, CPU vendors were faced by the fact that scaling the clock rates of the classical single CPU is neither economical nor forever possible [Gee05]. A way out of this crisis was the introduction of the Single-Chip Multiprocessor. A CPU that is equipped with several parallel working execution units [ONH<sup>+</sup>96]. The marketing term for Single-Chip Multiprocessors is multi-core CPU, because of the several execution units named cores. The introduction of multi-core CPUs had several impacts:

1. CPU vendors can further obey Moore's law and double their transistors per year<sup>1</sup>,
2. parallel computers moved into the consumer market and
3. applications won't be faster if they don't get specifically tuned for multi-core CPUs.

The work in this thesis tries to cope with the third impact in the field of urban drainage modelling. More about the multi-core architecture can be found in the Section 1.4.5.

The targets of parallel computing can be summarised by the following key points:

- reduce the time to finish computations,

---

<sup>1</sup>actually every 18 months

- solve large problems in a reasonable time,
- using an array of cheaper computers to reduce costs,
- overcome memory constraints in modern CPUs (see more in Section 1.4.5),
- etc..

Before diving even further into the field of parallel computing one needs to know some concepts and terms of parallel computing that are explained here and used later:

**Task** A task describes a piece of work that needs to be done. It is a unit of work. In parallel computing a problem is divided into tasks that are able to run concurrently.

**Thread** A thread is an active entity that runs tasks independently of other tasks. A thread is part of a process and defines a unit for scheduling.

**Process** A process is an active entity that contains several threads, at least one, and has resources that are shared by these threads.

**Processor** A processor is the physical unit that executes processes. A processor is used as an abstract machine that is able to run processes. It may be a core of a multi-core CPU, a single CPU or even a separate machine in a network of machines.

**Scheduler** The scheduler is a part of the operating system that assigns threads to processors.

### 1.4.1. Overview

Parallel computing formed a new research discipline. With every research discipline comes a completely different set of terms that are used to describe the work in the field. By using a real world example the common terms and some of the problems that arise in parallel computing are described here.

Imagine, two people doing the dishes. We have two "entities" capable of doing work in parallel, i.e. they can do their work at the same time, they can do their dishes concurrent. Parallel working entities are called threads or processes, now we have two of them in form of human beings. The type of work they do is called the task, e.g. one is washing the other is drying the plates and cutlery. Assigning a person to a task is called scheduling, e.g. person *a* is washing, person

$b$  is drying. We have two choices on which way it is possible to parallelize the task of doing the dishes, i.e. two parallelization strategies.

In the first one, everyone gets a staple of dishes and washes and dries them afterwards. The problem here is, that there are two distinct sinks and towels needed. If there is just one, how do they share the sink? What if one is faster than the other, he runs out of work, which is called starving.

We could go after Ford and do it in an assembly line fashion. One grabs a plate, washes it and gives it to the person who dries it off. Handing over the plate is called the communication of the processes. This kind of fashion for doing work is called a pipeline in where the stages of doing the dishes are the pipeline stages and the date that moves from the start to the end of the pipeline is a single plate. If the drying stage is faster than the washing stage the pipeline is unbalanced.

In this thesis the parallel strategy is meant to be the algorithm that is capable of running a prior sequential algorithm in parallel. The strategy is independent of the used API or programming language and is further independent on which hardware the algorithm is running. In other words the strategy describes the following aspects of a parallel algorithm:

- how the work is split up into independent tasks,
- how the work is scheduled onto the processes and
- how synchronisation is happening.

Every parallel strategy has its advantages and disadvantages, coming up with a strategy and solving its disadvantages is at the heart of parallel computing in computer science. This thesis describes parallel strategies and the their implementations for UDM.

### 1.4.2. Parallel Algorithm Design

Classical algorithm design deals with the problem of finding a step-by-step description of how a problem needs to be solved. Parallel algorithms get an extra dimension of concurrency, not just step-by-step but also side-by-side. A parallel algorithm defines sets of steps that run concurrently. A parallel algorithm adds the following [GGK03]:

1. Identify tasks that are able to run concurrently,
2. map these tasks onto threads and processes,

3. manage data exchange between the tasks,
4. manage concurrent accesses to shared resources and
5. synchronise the process and threads at various stages.

Depending on the problem that needs to be solved there are essentially two types of parallelism:

1. Data Parallelism and
2. Task Parallelism.

A problem that can be solved by data parallelism is one where a single algorithm is applied to a certain amount of data. The work for every date is almost identical. This sort of parallelism is easier to implement and has the potential of using more parallel infrastructure.

A task parallel problem is one where different tasks are able to run concurrently. This sort of problems are more difficult to solve. Task parallel problems often require high amounts of concurrent access to shared resources and a high amount of communication between processes.

Problems that are needed to be solved in parallel normally do not belong to a single type, more often they are a mixture of data-parallelism, task-parallelism and many other kinds of parallelism.

### 1.4.3. Communication

Parallel problems without any need of communication between processes are called *embarrassingly parallel* [Fos95]. These problems are easy to solve and do not require a high amount of skills to get implemented. The disadvantage of these problems is that they are rare. Most problems that must be solved parallel, for various reasons, aren't embarrassingly parallel.

That means that most parallel problems have a high amount of communication. Designing a parallel algorithm or a parallelized version of an existing algorithm includes to decide *what*, *when* and *how* to communicate. *What* communication may occur includes:

**Protection** "This data is being worked on, stop until the work has finished".

**Data exchange** "This is the answer that I came up with."

**Synchronisation** "Wait until all data has received." or "Wait for all tasks to be finished."

## Work assignment "Do that!"

When communication happens is certainly restricted by the problem, but *how* to communicate is another big factor that influences the implementation and performance of the algorithm. The most common parallel programming paradigms are:

1. Shared memory model
2. Message passing model
3. Data parallel model

## Shared Memory Model

The first being shared address space is used by programs that are running on shared memory systems [GGK03]. Although the shared memory paradigm has its roots on single computer systems a hybrid form exists where a distributed system uses a shared memory abstraction. More information on distributed shared memory systems can be found in [PTM96, NL91]. The address space is abstracted by a memory model that has a range of consecutive addresses. The exchange of information happens by load and store operations on these addresses. The programmer has to pay attention on things like concurrent memory access of two different processes. A critical section is a program fragment that may access a memory region concurrently with other threads.

Concurrent access control and synchronisation happens by using locking. Different types of locks exist and are used in various situations.

**Mutex** A Mutex protects a critical section and allows just one thread to enter the mutual section. A lock is locked at entry and unlocked at exit of the critical section.

**Semaphore** A Semaphore is a Mutex which allows more than one participant to enter a critical section. A semaphore is initialised by an integer. Every participant decrements at enter and increments at exit. If a participant tries to enter a Semaphore with a counter at zero, it is locked.

**Conditional Variable** A conditional variable allows two operations *wait* and *signal*. It can be used for synchronisation purposes, e.g. waiting for data to arrive.

Locking can be done on various degrees of granularity. Coarse grained locking is easier but destroy performance. For example the python programming language has a global interpreter lock (GIL) that prevents only one thread to be executed at a time [Dav09]. Fine grained locking increases the chances of

programming errors, e.g. forgotten locks, forgotten unlocks, race conditions, deadlocks, livelocks, starvation etc. [Vin07].

An alternative to using locks on a shared memory system is to use software transactional memory (STM). STM offers optimistic locking on shared memory systems without blocking the thread flow. In STM a critical section is marked as atomic. If two threads act upon the same atomic memory region the actions are reverted by a rollback. After a rollback the actions are retried until they succeed. The idea of STM is that locks are most of the time unnecessarily locked, i.e. no concurrent access was prevented by a lock [ST97].

Although STM has some problems that need to be overcome [CBM<sup>+</sup>08], like a relative high overhead [SMSW09]. Beside the imperative programming languages STM is said to be beautiful and highly compositional in functional programming languages like Haskell [BBG09].

## Message Passing Model

The second approach to exchange information is the Message Passing (MP) paradigm. Parallel applications using the MP model are divided into separate processes often called *actors* [Vin07]. These *actors* have private memory which is not shared with other *actors*. *Actors* exchange information based on passing messages to each other. Therefore exchange of information is made explicit.

MP can happen synchronous or asynchronous, the basic building blocks are *send* and *receive* operations [GGK03]. The *send* and *receive* operations can use different blocking and buffering behaviours:

**Blocking Non-Buffered** In this scheme the sender is blocked until the receiver acknowledges that the data was received. In some cases this can be very inefficient, because the receiver waits until the corresponding site got the data. These waiting times are known as *idling overhead*. Another disadvantage of this immediate scheme is that deadlocks appear when two parties send data at the same time. Due to the blocking semantics they never check for new messages and never answer each others messages.

**Blocking Buffered** This scheme solves some of the problems of non-buffered blocking send/receive. The solution is to buffer the send and receive data. The process sending the message is blocked only as long as it takes to append the message to the buffer. The protocol of the underlying network then assures that the messages are transmitted into the receivers buffer. Although buffering solves the idling overhead it doesn't solve all deadlock problems.

**Non-Blocking** Non-blocking operations are accompanied by a *check-status* operations. The *check-status* operation assures that the message has arrived at the receiver or into the receiving buffer, depending on whether the operations are buffered or not. This scheme further reduces the idling overhead and deadlocks are now impossible. The disadvantage of this scheme is that the programmer must now implement more management to assure that messages have been received.

In contrast to shared memory systems, MP is traditionally used in distributed systems [GGK03]. But similar to shared memory left the boundaries of single computers, MP has proven to be very efficient in the use of functional programming languages.

Erlang is such a functional programming language with built-in support for the actor model and MP. Actors run in processes which are essentially user space threads with a very low overhead. An erlang system can easily create one million processes in under a second on a consumer range computer [Vin07]. Actors are able to run locally or in a distributed manner [ADE92].

Programs written using MP are normally distributed and have an exclusive separate address space. They are used in clustered environments where it comes naturally to encapsulate data into messages and send them to other peers over the network. The most widely used API to implement applications using MP is the standardised Message Passing Interface (MPI). MPI is a language independent specification of a MP API, not the implementation itself. Several implementations exist, which conform to the MPI specification.

The biggest advantage of using a MP paradigm is that all data is private, data exchange is made explicit by passing it around. Because of this private data scheme, locking to protect against data corruption is not needed [GGK03].

#### 1.4.4. Parallel Architectures

Depending on the used algorithms and the needed amounts of parallel resources, parallel applications may run on a different range of computers or group of computers. There are two major groups of parallel computing architectures:

**Shared Memory Systems** are systems where the parallel elements share the same address space. In today's computer systems two variations are available. It is possible to combine the two variants to further increase the performance of the system.

1. Multi-core refers to a Single Chip Multiprocessor [ONH<sup>+</sup>96] where



several CPU cores are unified on one die. Single chip performance is going to stagnate because of physical limits that prevent further frequency scaling. Instead of focusing on single chip performance (i.e. scaling in frequency) chip vendors are packing more cores on one die to scale in performance (i.e. scaling in cores).

2. Multiprocessor systems are computers that have more than one identical processor that share the same main memory. Multi-core is a variant of a multiprocessor system. The classic multiprocessor system is a Symmetric Multiprocessing System (SMP) where two or more CPUs are combined in a single system.

Shared memory systems offer fast and low latency data exchange between the parallel elements but are restricted in the amount of parallel execution units. Shared memory systems must explicitly manage concurrent access to memory regions, because of this special attention is needed when programming shared memory systems. More on the problems of parallel programming can be found in Section 1.4.7.

**Distributed/Multi-computer Systems** are systems where parallel elements are connected via networking which allows them to be distributed over a wide range. Again two incarnations of distributed computing systems are used. Distributed systems allow having much more parallel elements. The disadvantage of distributed system is that the exchange of information is depending on the connections slower than on shared memory systems.

1. A Cluster is a group of loosely coupled computers which are connected by fast computer networks. Entities in the cluster are called nodes. A master node is responsible for scheduling the workloads on the working nodes called the slave nodes. Scheduling and data distribution is done manually by using specialised programming APIs and tools like MPI. The most prominent example of a cluster is the one that powers Google's Internet search engine. Google uses a cluster of over 15000 commodity class PCs.

The advantage of using such an architecture is a relative low price for having high performance, compared to system built from fewer but more expensive high-end servers [BDH03]. If more computing power is needed it is easy to just add more nodes to a cluster.

2. A Grid is similar to a cluster in that several computers are connected using the Internet. The big difference between a Grid and a cluster is, that grids are geographically distributed and computing power is considered as a resource which can be bought, has a kind of quality

and should be available in the same sense as the power in a power grid [BV05]. The management of the computing resource is done by a grid middle-ware which controls access and distributes the data to the nodes of the grid. A grid is heterogeneous regarding its nodes which complicates load distribution and management.

**Many Core** is a future architecture which is similar to the shared memory multi-core architectures but the core number is above what can be handled by today's multi-core architectures. The high amounts of cores changes the architectures, because multi-core is not able to scale to a thousands of cores [Bor07]. Many-core had its first incarnation by using Graphical Processing Units (GPU) for General Purpose computing (GPGPU) [LHK<sup>+</sup>04]. GPGPU uses the programmable shader units of modern GPU hardware for general purpose computing [MLG07].

#### 1.4.5. Multi-core Processors

In order to increase the speeds of a processor chip makers are trying to pack more transistors into smaller chips. One can imagine that this process has to end at some point. Chipmakers realised that in order to increase the overall performance of a system they need to pack multiple processing cores into one die [Gee05]. The term multi-core is a marketing term that refers to a single-chip multiprocessor [ONH<sup>+</sup>96]. A multi-core system has several advantages over a single core processor.

Packing multiple cores into a single die solved a lot of problems for chip makers in order to stay true to Moore's law. Although the overall performance of multi-cores is better than the one for single core processors, applications need to be adapted in order to gain from the available processing power.

In order to gain from the multiple cores in a system one needs to partition the problems into several tasks that can run in parallel. The software component that is able to run those tasks is called a thread. Multi-threading is a term that can refer to either the software practise or the hardware platform able to run multiple threads at once, like a multi-core CPU.

Every program contains at least one thread. If it contains more the program is called a multi-threaded program. One or more threads must be managed by the operating system, which is called the operating system thread. The lowest level of a thread is the hardware able to run a software thread.

2005 Intel changed its business and focused on Multi-core processors for its high end CPUs. Intel followed IBM with its Power4 [WKP<sup>+</sup>02] and Sun Mi-

crossystems with its Niagara architecture [KAO05]. The term “multi-core” refers to a CPU which doubles the amount of cores on one die. While Multi-core is a current technology and here to stay, many-core is the future direction which researchers from the University of Berkeley concluded. Many-core is a new architecture type featuring not two, four or eight cores but many thousands in one die [ABC<sup>+</sup>06].

#### 1.4.6. Parallel Performance Metrics

In order to measure the performance of parallel programs and the implemented algorithms a common nomenclature was introduced. For introduction purposes some of these are described here.

The first metric, which is also one that is applicable to sequential algorithms is the *execution time*. The *serial runtime*  $T_S$  is the time frame from the start to the termination of best sequential algorithm. When parallel programs are analysed, we often want to measure the *parallel runtime*  $T_P$  which is the time the first parallel processing happens until the last parallel work item has finished [GGK03]. The parallel runtime is used here as one indication of performance presented in the results chapter.

If the  $T_S$  and  $T_P$  are known and the number of parallel processing units  $p$  we can calculate the *total parallel overhead*  $T_O = pT_P - T_S$ . The *total parallel overhead* allows to show how much additional time was lost because of the parallelization. The lower this overhead is, the better the algorithm performs.

One of the often used parallel performance metric is the speedup. It is a factor which describes how well the parallel hardware was used. The speedup is defined as  $S = \frac{T_S}{T_P}$ . In this thesis the speedup was used to show how well an algorithm adapts to additional added parallel hardware, e.g. cores. Therefore the speedup is slightly modified but reflects its original definition. The speedup is calculated depending on the used parallel elements in use  $S_p = T_{P,1}/T_{P,p}$  and is based on a single parallel item in use instead of the best sequential runtime. The speedup is best when  $S_p = p$  which means that the speedup is equivalent to the number of parallel elements.

A metric which normalises the speedup is the parallel efficiency  $E = \frac{S}{p}$ . The efficiency shows how efficient an algorithm uses the available parallel hardware. It summarises in a value up to 1.0, the ideal case, how efficient a parallel algorithm is [GGK03].

### 1.4.7. Challenges of Parallel Programming

Programming parallel and concurrent systems involves two tasks that are essential for a good working parallel algorithm:

1. making the program correct and
2. making the program fast.

Both tasks demand a completely different set of skills when it comes to program a sequential or a parallel program.

If a sequential program runs several times, the CPU executes the exact same sequences of commands in every run<sup>2</sup>. This is not the case for parallel programs. The execution order is inherently nondeterministic. Making a parallel program correct involves taming this nondeterminism in places where it is needed to assure that no errors can happen. Sources of such errors were never possible in sequential programming. Some notorious and hard to fix problems of parallel programming are [Akh06]:

**Race Conditions** When several threads access the same memory regions it is often the case that the execution order of the threads accessing the memory influences the outcome of the operation. This is called a *race condition*. Even such simple operations as incrementing an Integer number can cause *race conditions*. A solution to race condition is to make the operations atomic by either using hardware supported atomic operations or protecting the memory region with a *Mutex*. [NM92].

**Memory Corruptions** Memory corruptions are the worst case of race conditions in which a process does not only calculate a wrong answer to a problem but also destroys the integrity of data.

**Deadlocks** *Deadlocks* occur if shared resources are locked step wise. For example, if two threads want to transfer money between two accounts. Thread one from *A* to *B* and thread two from *B* to *A*. The both lock their source accounts and they try to lock the target account. Without further information they wait forever to get the target account locked. Potential *deadlocks* happen to lock up rarely and are hard to fix. One solution is to order resources, memory addresses, and lock them in an ordered manner.

**Livelocks** *Livelocks* are variants of *Deadlocks* where two threads aren't waiting for resources but checking the resource first and step back and try it again. An analogy would be, if two persons want to pass but they try it on the

---

<sup>2</sup>If the program is executed in the same context

same side. If one changes the side, the other one does the same. This could go on forever.

Being fast is not an optional feature for parallel algorithms as it is often the case for sequential ones. The only reason to parallelize an algorithm is less runtime. If an algorithm designer has sorted out all problems of the algorithm he is very often faced with sub-optimal speed. Sources of slow downs in parallel programs are [Akh06]:

**Heavily Contented Locks** *Lock contention* is when a thread tries to acquire a lock that is already locked. Coarse grained locking often results in lock contention of the locks, as described in Section 1.4.3. Waiting for contented locks means less progress than possible. One solution to this problem is to use finer grained locking and try to lock as less as possible [Tho94].

**Overheads** Scheduling threads has a small but noticeable overhead, especially if lots of threads are used. A second source of overhead is the creation and destruction time of threads and processes. A solution to both problems is to use a thread/process pool that allows to define an upper limit of threads and processes and allows to reuse already running threads/processes [SV96, Sch98, PSCS01, LML00, KHYP08].

**Priority Inversion** If a process with a low priority holds a lock too long a process with high priority gets blocked. This is called *priority inversion*. It can be compared with a slow car blocking a fast car on a single lane bridge [Nae05].

For this reasons parallel programming is considered hard, and researchers are heavily searching for solutions for some of those problems. One such field is automatic race condition detection. The amount of papers in this field shows that this kind of problems are not easy to handle [BD96, EA03, FF01, NG92].



## Chapter 2.

# Urban Drainage Modelling

*CityDrain3* is a software that allows to simulate the urban drainage waste water cycle. Various ways exist to approach simulation of urban drainage systems. These approaches are called models. A model, as described earlier, is a formulation of mathematical concepts and their connections to analyze system effects virtually. Urban Drainage Modelling (UDM) is the intent of virtually describing the urban waste water processes by such models. This chapter describes the conceptual foundations on which these models are based and the various models researchers have found to calculate water flow and pollutant concentrations. A short overview on the difference between hydrodynamic and hydrological simulations is also given.

*CityDrain3* is a reimplementaion of CITY DRAIN with a focus on speeding up the simulations that use the underlying models from CITY DRAIN. No new models and neither the approach of running the models has changed since CITY DRAIN. Therefore the following chapters are intended as an overview on the matter to introduce the uninformed to the subject of UDM. Informations of the following chapters were taken from [AMR07] and [RGK02] and form the base for further investigation in the field of UDM.

### 2.1. Modelling Concepts

This section describes the mathematical background to UDM. Several processes are used to describe the water and pollutant flow from rain and other flow sources to the Waste Water Treatment Plant (WWTP). *CityDrain3* was developed as the framework to let civil engineers implement the models they are using in their work. Therefore only some simple models were implemented so that it is possible to work on parallel strategies and generate results. Therefore the list of models following is neither thoroughly described nor complete (e.g. pollutant transport and pollutant processes were left out completeley).

The modelling concepts that are needed to be taken account in UDM, which are implemented in CITY DRAIN and are therefore possible to be implemented in CityDrain3 are:

1. Rainfall runoff,
2. Hydraulic transport,
3. Pollutant transport and
4. Pollutant processes.

### 2.1.1. Rainfall Runoff

Rain is one of the sources of flow that an urban drainage system must handle. Rain is the source of high flood peaks, during intensive rain events, in which the quantity of flow sourcing from rain exceeds all other flows. Therefore rain is a decisive factor that an urban drainage system must be able to handle.

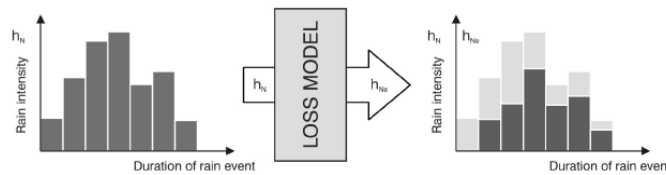


Figure 2.1.: Schematic on the application of rainfall loss model [AMR07]

The amount of rain (i.e. stormwater) actually running on the ground is reduced by various factors including

- wetting,
- depression loss,
- devaporation and
- infiltration.

A loss model transforms rainfall  $h_n$  into the runoff  $h_{En}$ . This process is depicted in Figure 2.1. Various methods exist to calculate the runoff at different levels of details. Figure 2.2 shows three of them which are:

**threshold method** The threshold method simply adheres to an initial loss ( $h_i$ ) in which both, wetting and depression losses are included.

**percentage method** The percentage method contributes a permanent loss ( $h_p$ ) in percentage to the threshold method.



**limit value method** The limit value introduces a depression loss ( $h_D$ ) in form of an exponential function.

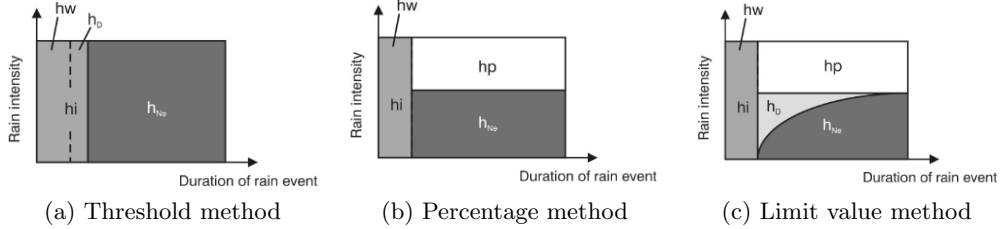


Figure 2.2.: Lossmodels [AMR07]

After the amount of runoff is known the effective flow of an urban area can be calculated:

$$Q_{Ne} = \frac{h_{Ne} \cdot A_{EFF}}{\Delta t}$$

where the effective area  $A_{EFF}$  is influenced by the run off coefficient  $\varphi$

$$A_{EFF} = \varphi \cdot A_{TOT}$$

## 2.1.2. Hydraulic Transport

After knowing what is the remaining rain, we need to transport this remaining flow away. Two methods for calculating a flow are known, one is based on the physical principals of continuity and the preservation of energy and the other is based on conceptual relations between cause and effect.

The first one is known as the *pyhsical flow model* and involves solving the one dimensional flow equations of St. Venant:

$$\frac{1}{b} \cdot \frac{\partial Q}{\partial x} + \frac{\partial h}{\partial t} = 0$$

$$\frac{\partial Q}{\partial t} + \frac{\partial}{\partial x} \cdot \left( \frac{Q^2}{A} \right) + g \cdot A \cdot \frac{\partial h}{\partial x} + g \cdot A \cdot (I_E - I_S) = 0$$

The flow  $Q(x, t)$  and cross section  $A(x, t)$  are the unknowns in this equation. The equation can be approximated by several layer of details with:

$$\begin{array}{rcl}
 g \cdot (I_E - I_S) = 0 & \frac{\partial Q}{\partial x} + b \cdot \frac{\partial h}{\partial t} = 0 & \dots \text{kinematic wave} \\
 g \cdot \frac{\partial h}{\partial x} + g \cdot (I_E - I_S) = 0 & \frac{\partial Q}{\partial x} + b \cdot \frac{\partial h}{\partial t} = 0 & \dots \text{diffuse wave} \\
 \frac{\partial v}{\partial t} + v \cdot \frac{\partial v}{\partial x} + g \cdot \frac{\partial h}{\partial x} g \cdot (I_E - I_S) = 0 & \frac{\partial Q}{\partial x} + b \cdot \frac{\partial h}{\partial t} = 0 & \dots \text{dynamic wave}
 \end{array}$$

The kinematic wave is only able to model translation due to the approximation in which friction and energy slope equality ( $I_E = I_S$ ). The diffuse wave is able to calculate backwater and attenuation. The wave models damping and translation. The diffuse wave is of second order, because of this two boundary conditions (up- and downstream) are needed.

The second way to calculate the transport of flow is the *conceptual modelling approach*. Conceptual models (also known as hydrological modelling) are applied to various fields of hydraulic routing, some of these are now described, where the mathematical formulations are just given for the musking routing method because of its importance in this thesis:

**Time area method (Isochronous Method)** The time area method is used for catchments where at each point the water requires a specific time to reach the outlet. The catchment is divided into isochronous areas separated by lines after which a point needs the required time to reach the outlet (see Figure 2.3a).

$$Q_i = \sum_{j=i}^n A_j \cdot R_{i-(j-1)}$$

**Linear hydraulic retention** The linear hydraulic retention basin (see Figure 2.3b) is the simplest type of hydraulic retention where the retention effect depends linearly on the stored volume.

$$Q_E(t) = \frac{1}{K_s} \cdot V(t)$$

The outflow  $Q_E \left[ \frac{m^3}{s} \right]$  depends on a storage constant  $K_s [s]$  and the stored volume  $V(t) [m^3]$  at the time  $t$ .

**Cascading linear hydraulic retention** An extension of the linear hydraulic retention is the cascading linear hydraulic extensions where several linear storages are combined as depicted in Figure 2.3c.

**Muskingum method** The muskingum method is the most important one in

this thesis, because it is used in many nodes that were implemented in the version of CityDrain3 that is used for this thesis. The method is used frequently in river and sewer hydraulics and was first applied to flow control at the river muskingum. The storage is described as a function of inflow  $Q_I$  and outflow  $Q_E$  where it may be shaped like a prisma and like a prisma with a wedge atop (see Figure 2.4a and 2.4b)

$$V = K \cdot Q_E + K \cdot X \cdot (Q_I - Q_E) \quad (2.1)$$

with the constants  $K$  (time for a unit discharge wave to travel through the reach) and  $X$  (wedge storage factor) describe the muskingum formular. The first term ( $K \cdot Q_E$ ) of Equation 2.1 is responsible for the prismatic storage whereas the second term ( $K \cdot X \cdot (Q_I - Q_E)$ ) describes the wedge storage.

Its possible to extend the muskingum method in the same way the linear storage retention method was extended by using arranging several sub-reachers in a row as shown in Figure 2.4c.

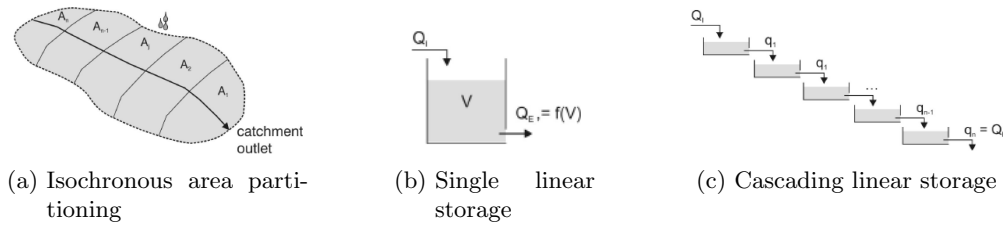


Figure 2.3.: Routing Methods [AMR07]

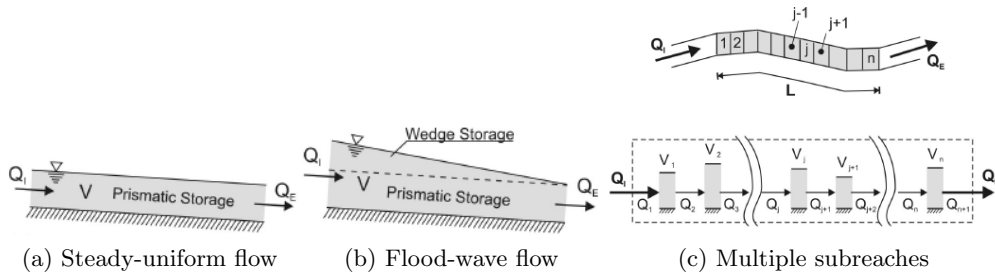


Figure 2.4.: Muskingum Routing Method [AMR07]

## 2.2. Integrated Urban Drainage Modelling

Integrated modelling is the intent to combine the traditionally separated components of the waste water cycle into a unified model that is able to indicate the

performance of the whole system [BS05]. The components and their interaction are shown in Figure 2.5.

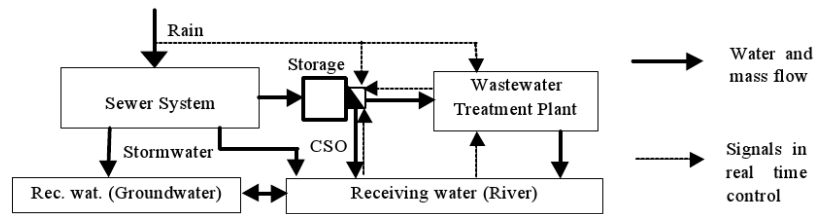


Figure 2.5.: Integrated drainage system (redrawn from [WJP<sup>+</sup>02])

These components were traditionally considered separately because of different responsibilities for the management and planning of sewers, treatment plants and rivers [WJP<sup>+</sup>02]. However a separated view of these components can not describe the full effects of the waste water system and can result in misleading conclusions in various scenarios [BS05].

Because of the importance shown here, CityDrain3 has the goal to be able to implement such integrated approaches. To make such an integrated approach reasonable and show long-term effects an efficient framework for running IUDM simulations is needed.

## 2.3. Conceptual Modelling

As stated earlier conceptual modelling is different from using physical based models where the exact physical mechanics are applied. These physical modelling known as hydrodynamic approaches, often involve solving complex formulations in form of differential equations as it is the case for the St. Venant equations.

Instead conceptual modelling focuses on cause effect relations to reduce the computational complexity which makes running long-term simulations unfeasible. Section 2.1 showed the mathematical background of some conceptual models, this section shows how such models are integrated into a framework for running such model simulations.

### 2.3.1. State Space Modelling

The whole CityDrain3 model is based on the state space modelling approach of CITY DRAIN. In CITY DRAIN a component of the waste water system is

called a block. Blocks have an input  $u$ , an output  $y$  and an internal state  $x$  as depicted in Figure 2.6.

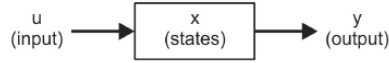


Figure 2.6.: Schematic description of a block

The output of a node is calculated by solving the discrete versions of the differential equations shown in Section 2.1 with numerical methods. This type of linear state space is described by the following equations

$$\begin{aligned}\frac{\partial x}{\partial t} &= A \cdot X + B \cdot u \\ y &= C \cdot x + D \cdot u\end{aligned}$$

The block notation comes from the roots of CITY DRAIN - Matlab. In the work of this thesis lots of graph theory was needed and used. Because of this it is natural to speak about nodes not blocks. The output of a node is the input of a downstream node. The whole system begins at the catchment and forms a graph structure that ends at the treatment plant.

### 2.3.2. The Graph Structure of a Sewer System

A graph is a pair  $G = (V, E)$  of the set of vertices ( $V$ ) and the set of edges ( $E$ ). The set of edges are pairs so that  $E \subseteq [V]^2$ , which means that the members of the edges must be in the set of the vertices. Edges short notations of  $(x_i, x_j)$  is  $x_i x_j$ . The set of vertices are the nodes and the edges represent a connection of two nodes. A graph is directed if we distinguish between the start and the end of an edge. In other words a undirected graph that contains an edge must also contain the reversed edge [Die05]:

$$(a, b) \in E \rightarrow (b, a) \in E | a, b \in V$$

A path is a nonempty graph  $P = (V, E)$  of the form  $V = \{x_0, x_1, \dots, x_k\}$   $E = \{x_0 x_1, x_1 x_2, \dots, x_{k-1} x_k\}$ . The length of the path is the number of edges which is denoted by  $P^k$  with a  $k$ -length. If  $P = x_0 \dots x_{k-1}$  is a path the  $C = P + x_{k-1} x_0$  is called a cycle. A graph with no cycles is called acyclic. A graph is called connected if there does not exist a pair of vertices which is not connected by a path [Die05].

If a graph is directed and is acyclic (DAG) there exists a non unique relation  $R$  over the nodes of the graph such that  $xRy$  exists iff there exists a path from  $x$  to  $y$ . The relation  $R$  is known as the topological order of a graph [Kah62].

In the case of an urban drainage simulation the simulation scenario is a graph, the nodes are the conceptual entities of the waste water cycle and the receiving water body. Because of the rather high abstraction of the urban drainage model some entities form a group of elements. For example a catchment is a very highly abstracted element, it represents an area where rain is caught and drained to a single effluent sewer. In reality an area represented by a catchment contains high amounts of buildings and sewer systems that must be treated differently.

The edges of the graph are the connections of the nodes. These connections don't exist in reality but allow to reuse certain algorithms with different parameters. A connection from a node  $x$  to a node  $y$  means that water is flowing from  $x$  to  $y$ .

The graph representing urban drainage systems may be cyclic, directed graphs but most of the time they are DAGs. In this thesis the focus was on speeding up cycle free urban drainage systems, although the software is able to calculate them without using parallel algorithms.

## Chapter 3.

# Methods and Implementation

CityDrain3 is a reimplementation of CITYDRAIN developed by [AMR07] with a focus on speed. CITY DRAIN was based on a combination of Matlab/Simulink which is broadly used by civil engineers. Simulink is a simulation system which can be used to analyse multi-domain dynamic systems. It features a graphical interface where a user can easily connect and arrange the blocks which represent subsystems of the simulation system. Because of the Simulink based implementation of CITYDRAIN it is very easy to model systems and extend CITYDRAIN by implementing new blocks in Simulink. Due to the prototypical nature of Matlab the speed of CITYDRAIN is slower than a native implementation with a simulation framework that is targeted at the needs of conceptual models used in IUDM simulations.

CityDrain3 is implemented in C++ which should bring runtime advantages over the Matlab based predecessor CITYDRAIN. Furthermore C++ was chosen because it offers a more flexible memory model and allows to use various ways to exploit modern multi-core CPU technologies.

The Matlab/Simulink environment provides a well performing simulation system that must be reimplemented in C++. Although, the simulation system needed by IUDM simulations is simple, various prototypes were needed to find the best solution that fits IUDM and allows to parallelize the system.

An object oriented design (OOD) was chosen to allow extending and separated the concerns of the simulation. The following chapter shows the most central parts of this OOD. Following the OOD design the algorithms and implementations of the parallel simulations are described.

## 3.1. Design

OOD allows to divide the code and data on which the code is working to be split up into separate logical units called classes. A class normally represents a noun of a software, for example *Simulation*, *Node* and *Model*. A *class* can be seen as a template, whereas the *object* is an instance of a *class*. e.g. a human being can be seen as a *class* whereas the instance *gregor* is an object of the type human (i.e. *class Human*).

Operations which belong to a class are called methods and are typically associated to the verbs belonging to a software system. e.g.: we *start* a *Simulation*, or we *init* and *execute* a *Node*. So by just talking about a piece of software it is very easy to identify the classes and operations on the classes. The OOD of a software are the identified *classes*, their relationships and how they interact. This chapter shows briefly how CityDrain3 is designed in order to reach its goal, usage of several cores on a modern CPU.

OOD can be seen as the low level design of a software, a common practice when designing complex software systems is to group classes into a highlevel architecture called Model View Controller (MVC). The model classes are used to hold and change the data of a software system. The controller classes encapsulate the business logic of an application, i.e. they act on the model and transform the data. The view is responsible for displaying the data. The design of CityDrain3 focused on the controller and model aspects of the architecture.

Figure 3.1 shows a very high level overview of the classes and their relationships used in CityDrain3.

### 3.1.1. Node

The *Node* class is the most central class in CityDrain3. A *Node* in CityDrain3 can be seen as a block in the Simulink based CITYDRAIN. In the modelling jargon a *Node* can be seen as an identified and separable part of the system we want to gain insights. But depending on the needs of the modelling process a *Node* can also be seen as the combination of several distinct parts of the system in order to have a more abstract view on the system, i.e. the *Catchment* often contains various sub components of the waste water cycle, which would be too complex to identify separately. Examples of Nodes implemented in CityDrain3 are:

- Catchment,
- Combined Sewer Overflow (CSO),



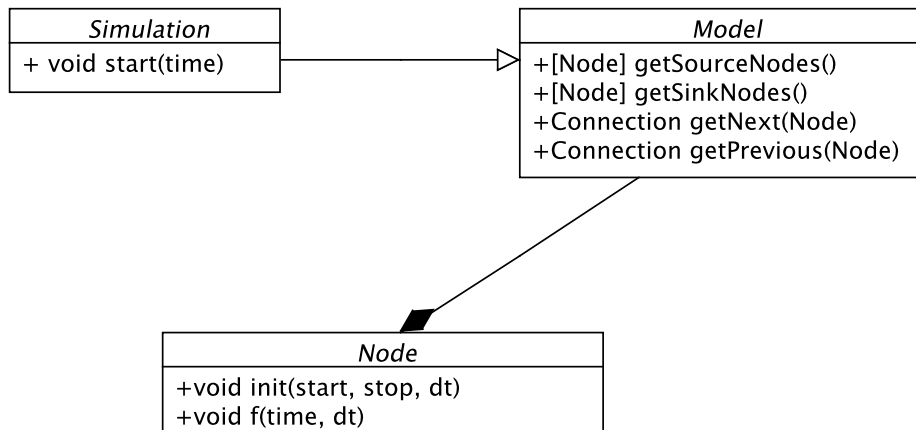


Figure 3.1.: Class Overview

- Sewer,
- Waste Water Treatment Plant (WWTP),
- but also virtual elements like a rain reader.

A Node has essential two methods - “init” and “f”. The method “init()” is used to initialize the node, which involves setting internal states to a predefined value, loading data, setting input- and output ports and naming the internal states so that the simulation can access them for serialization or controlling aspects.

The second method “f()” is responsible for updating the calculation based on inputs and internal states. It is called at every time step. Before “f()” is run the input ports are updated to represent the output values of preceding nodes, in other words upstream elements of the urban drainage water cycle. After “f()” is called the node needs to update the output ports for downstream nodes. As an example: A mixer node has up to n input ports. The “f()” of a mixer mixes these input flows and updates a single output port for the downstream nodes.

Inputs and outputs are distributed by ports. A port is tuple of a name and a *Flow* (described in Section 3.1.4). It represents the connection with other nodes. A node can have any number of input ports and any number of output ports.

Internal states of nodes are represented as a list of name value tuples. Values are typed and can therefore be programmatically enumerated, read and updated. This abstract implementation of internal states is used to implement controlling behaviours and serialization of simulation states. Serialization of simulations

allows to restart a simulation with updated or calibrated values which in turn allows the application to be controlled online [SSTW09].

To extend CityDrain3 with a new Node-type one needs to extend the abstract Node class and implement the “f()” and “init()” methods. The exact procedure to add new nodes to CityDrain3 is described in Chapter A.2.3.

<i>Node</i>
+void init(start, stop, dt) +void f(time, dt) +void addParameter(name) +addState() +getState(name) +addInPort() +addOutPort()
+[(name, value)] states +[(name, Flow)] in_ports +[(name, Flow)] out_ports

Figure 3.2.: The *Node* Class

### 3.1.2. Model

As said earlier the term model is used in an ambiguous way. A model used in “modelling and simulating” is the mathematical abstracted description of a complex system that somebody wants to experiment with by using simulations. But a model is also known as the data definition in MVC. In which the model is the data layer, the view the representation of the data and the controller the executing logic that transforms and uses the data model. In this chapter term model is used as in the MVC pattern.

A model represents the structure of a simulated system, it describes the used nodes, how they are connected and what parameters are used for the model nodes. In other words a model represents the urban drainage system which researchers want to examine. From a computer science point of view the model is a node container that allows to traverse the graph structure of an urban drainage system.

A model is the input of a simulation run. It is loaded into the memory from an external file which is structured by an xml file format. The model is loaded on startup and handed over to an instance of the simulation class(see Chapter 3.1.3).

Figure 3.3 shows a simplified overview of the *Model* class. *addNode* and *addConnection* are used by the *XmlLoader* class. They are used to setup and initialize the model. The next four operations are used by the various implementations of the *Simulation* interface. *getSource/SinkNode* are used to begin the traversal of the graph structure at the sink or source nodes. Source nodes are nodes with no input whereas sink nodes are nodes with no outputs. Using *getNext/getPrevious* the *Simulations* are able to traverse the graph step by step. This frees a simulation to choose whatever direction it wants to traverse the model.

Another feature of the model class not included in the class diagram in Figure 3.3 is the ability to serialize the whole model including the states of the nodes. This allows to restart a simulation run from a given time. The serialization format is presented in a human and machine readable format. This allows various online and offline calibration scenarios and the ability to stop and restart long running simulations.

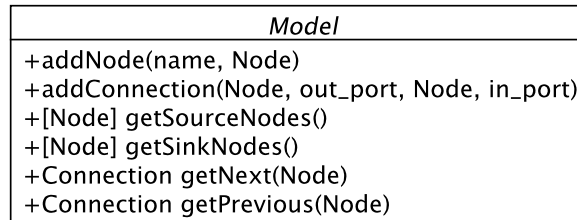


Figure 3.3.: The *Model* Class

### 3.1.3. Simulation

The *Simulation* class defines an interface used to drive a simulation. It is kept minimum so that the implementations aren't too restricted by the interface. The most important method is *start*, it is used to start and restart simulations.

The *Simulation* interface is implemented to use various simulation scenarios. This includes the parallel strategies described in Chapter 3.3 and simulations with varying time steps which are able to handle cycles in the graph structure. *Simulation* implementations get an instance of the *Model* class. Based on the implementation it traverses the *Model* graph, executes the "f" functions of the *Node* instances and distributes the data exchange between the connected *Nodes*. The data exchanged by *Nodes* is called a flow.

<i>Simulation</i>
+ void start(time) + setModel(Model) + setSimulationParameters(SimParams) + Signal progress()

Figure 3.4.: The *Simulation* Class

### 3.1.4. Flow

The *Flow* class represents the simulation data which is exchanged by the nodes. It represents the amount of water plus the concentrations in terms of pollutants, sediments, etc.. In CITYDRAIN the flow as a separate unit does not exist. The exchange of flow was handled by using lists of values. The definition of a flow and its concentrations was at a fixed position defined globally. In CityDrain3 the *Flow* class is a little bit more flexible and allows each node to add and remove concentrations on the fly. The elements of a flow are typed, have a unit and are accessed by names. This catches errors and detects them early whereas in CITYDRAIN they were left undetected.

*Flow* is exchanged between nodes on every timestep. Each node needs a separate copy of the *Flow* because it is not known what happens with the *Flow* in the various *Node* implementations, e.g. the *FileOut* node just writes the values of a *Flow* into a file whereas the *Sewer* node modifies its input. The exchange and update of flow happens very often in the simulation framework. An efficient implementation for the exchange of *Flow* is essential to a well performing parallel implementation.

A copy on write (COW) mechanism was introduced to keep the copying of a flow's data and its definition at the minimum. The values and meta information on the values, i.e. type, name and unit, were kept in separate shared instances of the class. If a *Flow* is exchanged the values and meta information are shared first. The values and meta information are only copied if they are altered. Chapter 4.6 shows the influences of such an implementation on the overall and parallel performance.

## 3.2. Sequential Simulation Run

Knowing how the process of running a simulation takes place is essential for understanding how this process can be broken up into different independent

parallel execution steps. This section describes a simple algorithm that runs a simulation sequentially. This should make it clear what a simulation needs to do.

In Line 1 of Algorithm 1 a typical main loop of a discrete time step simulation can be seen. We run from `starttime` up to `endtime` in `dt` time steps. One step of this `for` loop is called a *timestep*.

In each *timestep* a set  $N$  is initialized to contain all the nodes defined and in the model of this particular simulation. We begin by choosing a single  $n$  out of  $N$ . This  $n$  is then handed over to the recursive `execute n` procedure shown in Algorithm 2.

When `execute` is finished  $n$  is removed from the set of  $N$ . Choosing a  $n$  from  $N$  is done as long as the set  $N$  is not empty. If the set  $N$  is empty the simulation is finished with this timestep and it is assured that all nodes were executed in the right order, which gets assured by the `execute` algorithm.

---

**Algorithm 1** The main loop of a simulation run.

---

```
1: for  $t \leftarrow t_{start}$  to  $t_{stop}$  in  $dt$  steps do  
2:    $N \leftarrow$  all nodes in model  
3:   while  $N$  not empty do  
4:     choose  $n \in N$   
5:     execute  $n$   
6:   end while  
7: end for
```

---

The procedure `execute n` shown in Algorithm 2 is a recursive algorithm which executes the node  $n$ . But before that can happen all nodes on which  $n$  depends, i.e. which precede  $n$  in the model, must be executed first. This assures that  $n$  runs only if all nodes upstream are already done. After  $n$  is executed it gets removed from the set  $N$  which is the set of nodes that are not yet executed.

$PN$  is the set of nodes that precede the node  $n$  i.e. the ones from which  $n$  gets an input.  $PN$  is intersected with  $N$  so that  $PN$  is free of nodes that may have already been executed. This is because the underlying structure of a model is not a perfect Tree but a DAG (see Chapter 2.3.2).

The here presented algorithms are highly simplified and are only presented that the reader may get a better understanding of how a simple simulation software running IUDM simulations is carried out.

---

**Algorithm 2** The execute  $n$  procedure.

---

- 1:  $PN \leftarrow$  predecessor of  $n$
  - 2:  $PN \cap N$
  - 3: **for all**  $pn \in PN$  **do**
  - 4:   execute  $pn$
  - 5:   set output of  $pn$  to input  $n$
  - 6: **end for**
  - 7: execute  $n$
  - 8:  $N \leftarrow N \setminus n$
- 

### 3.3. Parallel Implementation

CityDrain3 was designed to be a very modular and flexible framework for experimenting with different parallel simulation execution strategies. Loading of a *Simulation* class is done at runtime so that different parallel implementations are easily tested and benchmarked against each other. This section describes three different parallel strategies:

1. flow parallel,
2. pool pipeline and
3. ordered pipeline strategy.

The Flow Parallel Strategy (FPS) was implemented first. It tries to fit a data parallel approach onto the tree like structure of the input sewer systems. The idea behind this strategy was to keep the data of the parallel flows in a single thread in order to reduce memory transfers between the cores. Figure 3.5 shows a conceptual overview of this strategy. It shows how the data stays in the CPU cores and in turn in the caches of the cores.

The second strategy the Pool Pipeline Strategy (PPS) tries to use a task/pipelined parallel approach [GGK03]. A pool of nodes is used and the time is pipelined through the threads. Figure 3.7 shows a conceptual overview of this strategy.

The PPS showed very bad performance in initial tests. In order to overcome this shortcomings the Ordered Pipeline Strategy (OPS) was implemented. It tries to eliminate all randomness of PPS. Figure 3.8 shows an overview of the OPS.

As stated earlier *Simulations* are loaded at runtime which makes them easily exchangeable. This is done by specifying the simulation which should be used

in the input XML system. More information on how to define the simulation parameters and the *Simulation* class can be found in Chapter A.1.2 and A.1.3.

### 3.3.1. Flow Parallel Strategy

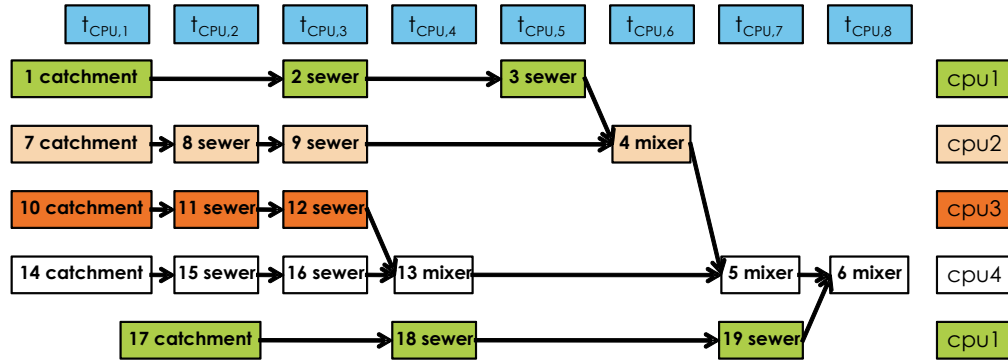


Figure 3.5.: Flow Parallel Strategy

Before any of the parallel strategies were implemented a standard straight-forward simulation was written. In order to assure that the parallel implementations are correct the results were compared with the standard simulation implementation. Also this standard implementation acted as a base for the first parallel strategy the FPS.

The standard simulation starts a loop over the simulation time. Each node has a counter which is initialized by the number of input connections. In case of the “Mixer” node in Figure 3.6 this counter is set to two because it has two input connections. Another loop is started over the source nodes (i.e. “Source 1” and “Source 2” in Figure 3.6). Before a node is able to be executed, i.e. running the “f()” function, the counter must be decremented and it must have the value zero in order to preserve the simulation defined calculation sequence. After the “f()” function ran the simulation loops over all output connections and again decrements the counter by one and checks if the downstream nodes are able to be executed. This algorithm assures that all depending nodes, i.e. the one with a downstream connection, are calculated first.

Figure 3.6 shows a sequence and the updated counter. The red node is always the next node that runs.

The FPS is heavily based on the standard simulation. Algorithms 3, 4 and 5 show a pseudocode on how the FPS algorithm works. The FPS introduced two changes to the standard implementation:

1. the loop over the source nodes was changed to a *OpenMP parallel for* loop,

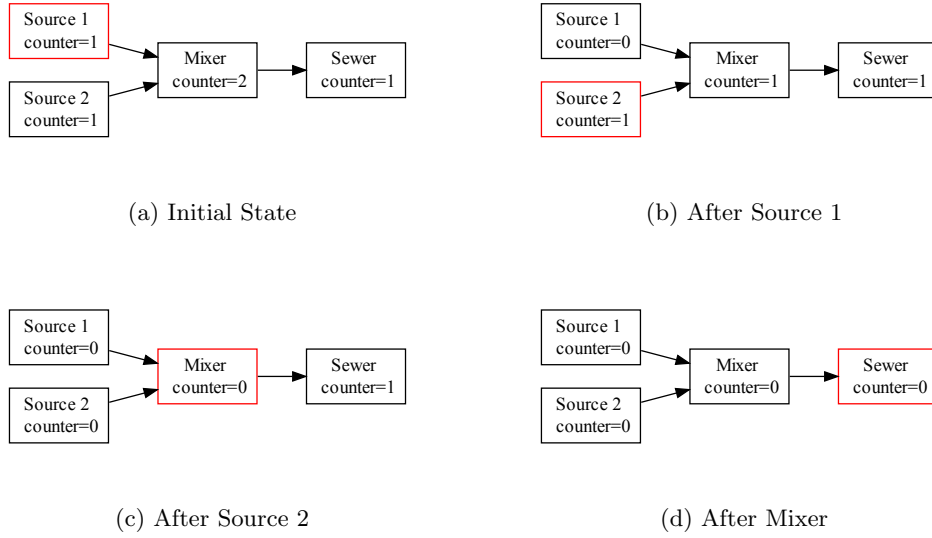


Figure 3.6.: Input Counter

---

**Algorithm 3** flow\_parallel\_simulation

---

```

1: for  $t \leftarrow t_{start}$  to  $t_{stop}$  in  $dt$  steps do
2:    $PD \leftarrow \text{get\_predecessors}$ 
3:   for all  $sn \in \text{sinknodes}$  do
4:     threaded(run_flow( $sn$ ))
5:   end for
6: end for

```

---



---

**Algorithm 4** run\_flow( $sn$ )

---

```

1: execute  $n$ 
2:  $NS \leftarrow \text{successors of } n$ 
3: for all  $n\_succ \in NS$  do
4:   set output of  $n$  into  $n\_succ$ 
5:    $PD(n\_succ) \leftarrow PD(n\_succ) \setminus \{n\}$ 
6:   if  $PD(n\_succ) = \emptyset$  then
7:     threaded(run_flow( $n\_succ$ ))
8:   end if
9: end for

```

---



---

**Algorithm 5** get\_predecessors

---

```
1:  $N \leftarrow$  all nodes
2:  $PN \leftarrow \emptyset$ 
3: for all node  $n$  in all nodes do
4:    $PN(n) \leftarrow PN(n) \cup PRED(n)$ 
5: end for
6: return  $PN$ 
```

---

2. and the counter update was protected by a *OpenMP critical section*

The greatest advantage of this strategy is the rather low number of changed lines of code. The second advantage is that the exchanged *Flows* between the nodes stay in the thread, and therefore in the core, as long as the thread does not reach a node with a mixing behaviour. The fact that the data stays thread local reduces the data migration costs from one core to another and allows to take advantage of the available cache memory.

The FPS should run particularly fast on simulation systems with parallel streams that offer long sequential flows. Because of this fact a testing sewer system was introduced from which this parallel simulation implementation should benefit the most. This simulation system shows the upperbound of what is the maximum expected performance of the FPS.

Although the data transfer between the cores is lower when using the flow parallel strategy the amount of parallelism which is usable by this parallel implementation is limited by the parallel streams in the sewer system. Depending on the drainage system this can have critical impacts on the parallel performance. The worst model is one where there are no parallel streams at all. This also means that the overall amount of threads to which this implementation can scale is limited by the number of parallel streams. Because of this limitations the set of testing systems includes such a sequential testing system.

Testing systems with a tree like structure offer lots of parallel streams at the source nodes. But they collapse downstream into just a handful of nodes and moreoften into one - the WWTP. Therefore the parallel performance drops when the simulation advances downstream.

### 3.3.2. Pool Pipeline Strategy

FPS is not able to scale on a sequential testing system and only allows to calculate parallel flows concurrently. Because of this limitations of FPS further investigations were made into finding ways to exploit more parallelism out of

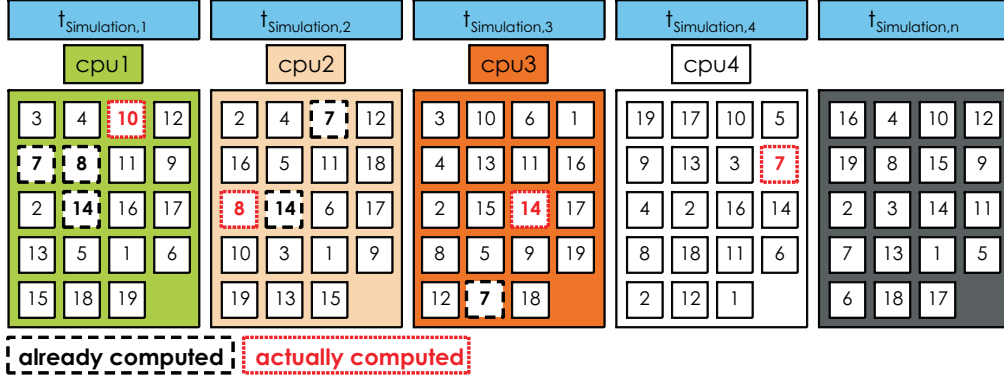


Figure 3.7.: Pool Pipeline Strategy

---

**Algorithm 6** pps\_main

---

- 1:  $N \leftarrow$  all nodes
  - 2: **for**  $t \leftarrow t_{\text{start}}$  to  $t_{\text{stop}}$  in  $dt$  steps **do**
  - 3:  $POOL_t \leftarrow N$
  - 4: thread\_pool\_add\_task(pps\_run\_timestep( $POOL_t$ ))
  - 5: **end for**
- 

the available sewer system. The characteristic of the FPS is that one time step is completely finished before the next starts. This shortcoming manifests itself into the limited scalability described before.

Searching for other ways to parallelize simulations of urban drainage systems, it was discovered that another timestep can begin even if the first one is not yet finished. This pipelining of timesteps through the sewer system gave the PPS the name.

---

**Algorithm 7** pps\_run\_timestep( $POOL_t$ )

---

- 1:  $PD \leftarrow$  get\_predecessors
  - 2: **while**  $POOL_t \neq \emptyset$  **do**
  - 3:  $n \leftarrow$  choose random  $n \in POOL_t$
  - 4: **if**  $PD(n) = \emptyset$  **then**
  - 5: execute( $n$ )
  - 6: **for all**  $ns \in SUCC(n)$  **do**
  - 7:  $PD(ns) \leftarrow PD(ns) \setminus \{n\}$
  - 8: **end for**
  - 9:  $POOL_t \leftarrow POOL_t \setminus \{n\}$
  - 10: **end if**
  - 11: **end while**
-

The (PPS) was the first *Simulation* which used this knowledge. Algorithm 6 shows the main loop of the simulation. Each parallel task is now responsible for a single timestep. A pool ( $POOL_t$ ) is used to track which nodes are not yet worked out. A thread randomly chooses a node from the pool and checks if all dependencies are satisfied. The dependencies from the pool pipeline strategy differ from the ones in the flow parallel, because nodes can now be in every state. Algorithm 7 is the part of the algorithm which runs in a thread managed by a thread pool.

All input depending nodes must be in timestep  $t + dt$ . And all output connecting nodes must be in timestep  $t$  and not less than  $t$ . This extended dependencies must be checked because the nodes can be in any timestep. If all the dependencies are satisfied the nodes “f()” function is called and the node gets removed from the threads pool. A thread is finished if its pool is empty.

The PPS uses simple threading, mutexes for critical regions and a thread pool to reduce the startup and teardown costs of threads [Sch98]. The advantage of this strategy is that it is able to gain from extra cores even if there are no parallel streams at all. The limiting factor is the number of sequential nodes. A big disadvantage of this strategy is that it randomly chooses from a pool. Despite this weakness it was at first easy to implement. Another disadvantage is that more data exchange happens between cores which reduces the memory throughput of the CPU [Dre07].

### 3.3.3. Ordered Pipeline Strategy

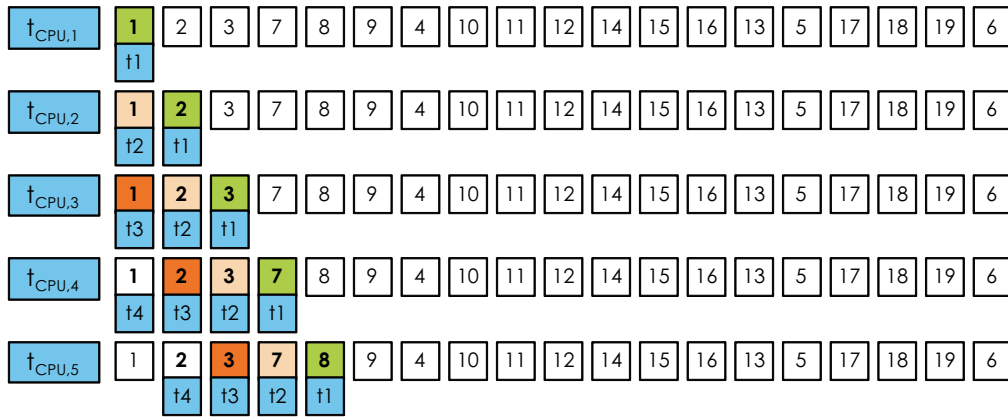


Figure 3.8.: Ordered Pipeline Strategy

To overcome the weaknesses of the PPS another pipelined simulation was implemented. The idea was to force the random aspects of the PPS into an ordered, linear structure, hence the “ordered” in the name. The idea of OPS

is to not randomly search for a next node to calculate, but grab it from a linear container which was already set up prior the simulation. Each thread, responsible for a timestep, then gets the nodes from this linear structure, runs them and puts them in the same order into another instance of this linear structure which is connected to the next thread (i.e. the next timestep).

The first problem which needs to be solved is to transform the DAG representing the model into a linear shape with all dependencies preserved. To get a linear structure the DAG is sorted by introducing a node relation  $R$ . A node  $i$  is said to be smaller than  $j$ ;  $i \leq j$  if the nodes are connected and  $i$  is the source node and  $j$  is the sink node. In graph theory: we have a directed Graph  $G(V, E)$  with  $V$  vertices and  $E \subseteq [V]^2$  edges, we say  $i, j \in V$  are smaller if  $(i, j) \in E$ . Topological sorting is the problem of finding a linear extension of the relation  $R$ , where  $iRj \iff (i, j) \in E$  for  $i, j \in V$  [CLRS01, Kah62].

Algorithm 8 shows a way to find the topological sorting of a DAG  $G = (E, V)$ .  $S$  is first initialized to be the set of the source nodes in  $G$ , i.e. all nodes which have no incoming connections:  $S = \{n \mid \nexists m, (m, n) \in E\}$ .  $L$  is initialized to be the empty linear structure holding the resulting sorted nodes, called a list. Figure 3.9 shows an example DAG, the red nodes are the source nodes, Table 3.1

---

**Algorithm 8** topological.sort( $G$ )

---

```

 $L \leftarrow \emptyset$  {list structure}
 $S \leftarrow$  set of source nodes of  $G$ 
 $E \leftarrow$  set of edges of  $G$ 
while  $S \neq \emptyset$  do
   $n \leftarrow n \in S$ 
   $S \leftarrow S \setminus \{n\}$ 
   $L \ll n$  {appends  $n$  to the back of the list}
  for all  $m$  where  $(n, m) \in E$  do
     $E \leftarrow E \setminus \{(n, m)\}$ 
    if  $m$  has no more incoming edges in  $E$  then
       $S \leftarrow S \cup \{m\}$ 
    end if
  end for
end while
if  $E = \emptyset$  then
  return  $L$ 
else
  return  $\emptyset$ 
end if

```

---

shows the steps to get to the final sort of  $L = \{1, 3, 7, 2, 4, 5, 6\}$ .

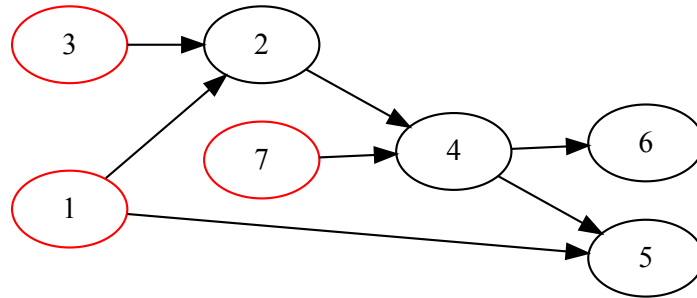


Figure 3.9.: An example DAG

step	n	S	update of $E$	L
1		$S = \{1, 3, 7\}$		$L = ()$
2	$n = 1$	$S = \{3, 7\}$	$E \leftarrow E \setminus \{(1, 2), (1, 5)\}$	$L = (1)$
3	$n = 3$	$S = \{7, 2\}$	$E \leftarrow E \setminus \{(3, 2)\}$	$L = (1, 3)$
4	$n = 7$	$S = \{2\}$	$E \leftarrow E \setminus \{(7, 4)\}$	$L = (1, 3, 7)$
5	$n = 2$	$S = \{4\}$	$E \leftarrow E \setminus \{(2, 4)\}$	$L = (1, 3, 7, 2)$
6	$n = 4$	$S = \{6, 5\}$	$E \leftarrow E \setminus \{(4, 6), (4, 5)\}$	$L = (1, 3, 7, 2, 4)$
7	$n = 5$	$S = \{6\}$	$E \leftarrow E \setminus \emptyset$	$L = (1, 3, 7, 2, 4, 5)$
8	$n = 6$	$S = \emptyset$	$E \leftarrow E \setminus \emptyset$	$L = (1, 3, 7, 2, 4, 5, 6)$

Table 3.1.: Example sort of DAG

The linear structure between the threads is an adapted queue from the Standard Template Library (STL). The STL queue was wrapped by a class *tqueue* in order to be thread-safe and synchronize the threads if the queue becomes empty. A thread which is blocked on an empty queue wakes up if a new node arrives. Depending on underlying thread implementation a core can be retargeted to another thread while waiting for data in the queue.

```

template <typename T>
class tqueue {
public:
    tqueue(){}
    virtual ~tqueue(){}

    void enqueue(T t) {
        {
            unique_lock<boost::mutex> lock(mut);
            q.push(t);
        }
        wait.notify_one();
    }

    T dequeue() {
        unique_lock<boost::mutex> lock(mut);
        while (q.empty()) {
            wait.wait(lock);
        }
        T front = q.front();
        q.pop();
        return front;
    }

private:
    std::queue<T> q;
    boost::mutex mut;
    boost::condition_variable wait;
};

```

Listing 3.1: The thread-safe queue

Listing 3.1 shows the implementation of the thread-safe queue. The data held in the class are:

- **The STL queue** used as the underlying data structure. It has a First In First Out (FIFO) semantic. This means that the order that one “pushes” data into the queue is the same as the one “popping” the data out of the queue.

- **A mutex lock** used to guarantee that “pushing” and “popping” is race condition free if more then one thread accesses the queue.
- **A wait condition** used to make the “popping” thread sleep if the queue is empty. A “push” into the queue is always followed by a notify so that a waiting thread will be waken up.

The thread-safe lock has several advantages that benefit its performance and also the elegance of the implementation of OPS. The greatest advantage is that locking and synchronization is done in one place - in the queue. This means that only the queue is responsible for locking and synchronization and synchronization is not distributed throughout the code. Code using *tqueue*<sup>1</sup> doesn’t need to care about race-conditions or deadlocks, they just call *push* and *pop* the rest is handled by *tqueue*. A queue is shared by only two threads, one responsible for *timestep<sub>t</sub>* and the other for *timestep<sub>t+1</sub>*. This fact reduces the propability of lock-contentions.

The ordered execution poses a more subtle problem when exchanging the flow data. Figure 3.10 shows a simple DAG with five nodes. A topological sort of this DAG is shown in Figure 3.11. If a node is executed it updates its input ports and translates the data by using the internal node semantik into an output that is provided at the output ports. The problem emerges when the data from the input ports is fetched, it can not be assured that the data is from the current timestep. This is because the fetched node could already be in the next timestep due to the pipelined nature of this strategy. For example we run nodes *A*, *B* and *C*, when we want to execute node *D* it fetches data from *A*, *B* and *C* but this nodes could be in *timestep<sub>t+1</sub>* because they come earlier in the queue. The problem also exists if instead of fetching the flows from an executing node the nodes push the data after execution, because you can’t know in which timestep the node is in, when “pushing”. The solution was to use a buffering datastructure able to handle concurrent access to its elements and that preserves the order in which data is “pushed” in. This sounds pretty familiar to the *tqueue* which was used to solve the problem.

---

**Algorithm 9** ops\_main

---

```

G ← input Graph
L ← topological_sort(G)
Qstart-dt ← init with L
for t ← tstart to tend in dt steps do
  Qt ← init empty queue
  thread_pool_add_task(ops_run_timestep(t))
end for

```

---

<sup>1</sup>*tqueue* was also usefull in other places.

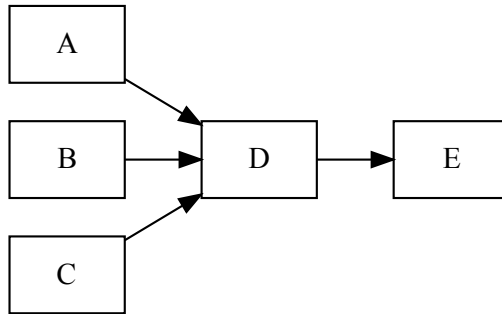


Figure 3.10.: Flow exchange problem DAG

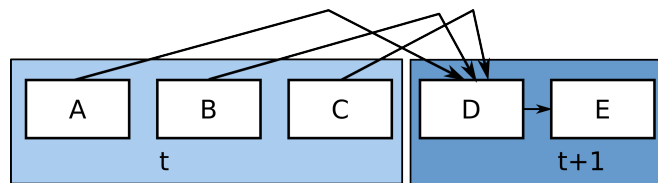


Figure 3.11.: Flow exchange problem ordered



Understanding *tqueue* and its uses is essential for understanding the inner workings of the OPS. Algorithm 9 shows the main loop of the OPS. First a queue is set up with the contents of the topological sorted nodes of  $G$ . This queue is special because it is the connection from the mainloop to the first timestep/thread. The use of this special queue allows the first timestep to be exactly like any other timestep. For each timestep there is a queue, denoted as  $Q_{t-dt}$  and  $Q_t$  for the timestep  $t$ , from which the thread gets its nodes and to which it pushes the finished nodes.

---

**Algorithm 10** ops\_run\_timestep( $t$ )

---

```

 $n \leftarrow$  pop from  $Q_{t-dt}$ 
while  $n \neq \perp$  do
    fetch inputs for  $n$ 
    execute  $n$ 
    output results from  $n$ 
    push  $n$  into  $Q_t$ 
     $n \leftarrow$  pop from  $Q_{t-dt}$ 
end while

```

---

Algorithm 10 shows the steps done in each thread, i.e. each timestep. A node is “popped” from the queue of the preceding timestep, the inputs are updated and the “f” function of the node is called. After the update of the output ports the node is “pushed” into the queue that is connected to the successor timestep.

The advantage of the OPS is that it does not care of which shape the sewer system is, as long as there are enough nodes in the system it is able to parallelize up to the number of nodes in the system. Another advantage is that synchronization and locking is handled in one piece of code, the *tqueue*. The threads are loosely decoupled and connected by the *tqueues* which are used to communicate the data between the threads. This makes the OPS a pretty elegant solution. Locking of OPS is very fine grained, as shown earlier, a fine grained locking solution has the potential to be more scalable.

### 3.4. Shared Flow

CityDrain3 simulates the flow of water including the concentrations of pollution in the water. The compound of concentration and water is called the flow. Each node then uses its input flow and transforms it. These transformations are designed to require low computational power in order to run longterm simulations, see Chapter 2.3 for more information on the characteristics of UDM simulations. This means the flow of water is at the heart of the simulation. Nodes were chosen to be the parallel units of work. The transfer of water between the nodes

is the most essential communication effort and it is not a minor one. Before and after every calculation step, data has to be brought up to date at the nodes ports.

Efficient handling of this communication was thought to be the key for good performance in general and for good parallel performance in specific. A large part of the design phase was dedicated into finding an efficient way to transfer the data between the nodes.

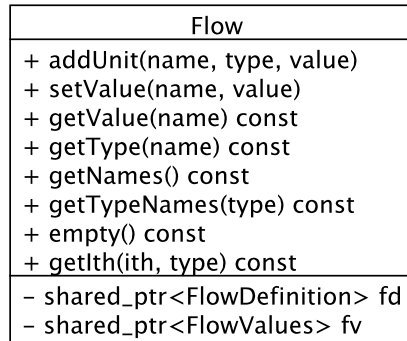


Figure 3.12.: UML of the Flow class

The *Flow* class had to fulfill two conditions which influenced its design:

1. Accessing the elements of the flow should be easy and straight forward. CITY DRAIN used lists to transfer the flow. A global table was used to index into the array and find the elements of the flow. This is inconvenient at best.
2. It should be fast to transfer, without worrying too much about data copying.

The first step was to get rid of a simple array based implementation. The array to index the flow elements is now part of the *Flow* class. The *Flow* class is a dynamic class which allows to add elements and update their values <sup>2</sup>. An overview of the class is shown in Figure 3.12.

The second problem was solved by introducing a Copy on Write (COW) semantics to the *Flow* class. COW is essentially a delayed copy of a resource. If an involved party alters the resource, the delayed copy is executed to be a real copy. The COW semantic in the flow is handled by using an already proposed standard extension to the *C++ STL* called *shared\_ptr* [Aus05].

<sup>2</sup>A flow element is essentially the class *CalculationUnit* with a value

A *shared\_ptr* is a wrapper class around a pointer that holds a use counter. If the use counter becomes zero the data to which the pointer points to will be deleted. Listing 3.2 shows the COW specific implementations of the *Flow* class<sup>3</sup>.

```

Flow::Flow() {
#ifdef SHARED_FLOW
    f = shared_ptr<FlowPriv>(new FlowPriv());
    fd = shared_ptr<FlowDefinition>(new FlowDefinition());
#else
    f = new FlowPriv();
    fd = new FlowDefinition();
#endif
}

Flow::Flow(const Flow &other) {
#ifdef SHARED_FLOW
    f = other.f;
    fd = other.fd;
#else
    f = new FlowPriv(*other.f);
    fd = new FlowDefinition(*other.fd);
#endif
}

Flow Flow::nullFlow() {
    Flow f;
    f.addUnit("flow", CU::flow, 0.0);
    return f;
}

Flow::~~Flow() {
#ifdef SHARED_FLOW
    delete f;
    delete fd;
#endif
}

Flow &Flow::operator =(const Flow &other) {
#ifdef SHARED_FLOW
    f = other.f;
    fd = other.fd;
#else
    delete f;
    delete fd;
    f = new FlowPriv(*other.f);
    fd = new FlowDefinition(*other.fd);
#endif
}

```

<sup>3</sup>The `#ifdef SHARED_FLOW` preprocessor directives are used to change the implementation with on that does not use COW semantics

```

#endif
    return *this;
}

#ifdef SHARED_FLOW
void Flow::copyData() {
    if (!f.unique()) {
        FlowPriv *old = f.get();
        f = shared_ptr<FlowPriv>
            (new FlowPriv(*old));
    }
}

void Flow::copyDefinition() {
    if (!fd.unique()) {
        FlowDefinition *old = fd.get();
        fd = shared_ptr<FlowDefinition>
            (new FlowDefinition(*old));
    }
}
#endif

```

Listing 3.2: Delayed copy in *Flow*

In the constructor a *shared\_ptr* member is initialized. The copy constructor of *Flow* is overloaded and calls the assignment operator of the *shared\_ptr* class, which in turn increases the usage count. Deleting the resources is handled by *shared\_ptr* so the destructor of the shared flow is empty. Assigning a flow is equivalent to copying it with the copy constructor, i.e. assigning the *shared\_ptr*. Every method altering the internals of the class must call one of the two *copy* methods, depending on what they change, i.e. *FlowDefinition* or *FlowPriv*. In the *copy* method the flow checks if the *shared\_ptr* is actually shared by other classes. If it is shared a copy of the member data is allocated and assigned. After the copy function is called the data can be changed without disturbing any other partys that have shared the data.

*FlowDefinition* and *FlowPriv* are sperated because the data of a flow, i.e. *FlowPriv*, gets changed more often than the definition of the data. This further reduces amounts of copied data.

Section 4.6 shows the difference of an implementation with and without a shared flow. Further improvements could be made by introducing perfect hashing as described by [Spr77]. The identification part of the flow is not changed very often, therefore static perfect hashing could be a way to lower the lookup of the flow and keep the comfortable string based lookups for the flow entities.

# Chapter 4.

## Results

This section presents the results of applying the identified parallel algorithms, described in Chapter 3 to IUDM simulations. The different algorithms were run on a variety of urban drainage systems, artificially generated for benchmarking purposes. These systems are introduced in Section 4.1 by means of their shape and purpose of showing effects of the algorithms. Beside showing what has been benchmarked the hardware on which the tests were run is as important as displaying their results. Chapter 4.2 gives an overview of the computer systems used to gain the results which are presented in a per hardware split in Chapter 4.5 and 4.4.

### 4.1. Benchmarked Systems

The algorithms presented in Section 3 have different characteristics which means they perform better or worse on the posed input sewer system. The nodes used in this systems are simple catchements which emit a constant dry wheather flow, mixing nodes which combine several input flows into a single flow, CSOs which emit waste water into the river if the input is beyond the capacity of the CSO, simple sewer nodes using the muskinggum routing method and a fileout node which writes the results of the simulation into a file.

The emphasis of these benchmark systems lies in the shape of the systems and not which nodes were used. As long as there are parallel to calculate nodes the runtime of a single node does not influence the parallel performance of the strategies. Although the worst case would be if far downstream a single node would block the whole timestep. OPS and PPS are not influenced by these downstream slow nodes because they are able to pipeline the timesteps of the slow nodes. FPS instead would block until this single, slow downstream node is finished to begin a new timestep. This kind of situations were not reviewed in the following tests.

Several different shapes were chosen in order to show the strengths and weaknesses of the algorithms. It is possible to divide the testing systems into two categories with different purposes:

1. *Artificially shaped* sewer systems in order to show the best or worst of an algorithm and
2. *Natural shaped* sewer systems to show how the algorithms would behave on sewer systems that are more or less structurally akin to real world systems.

Both categories are now described in detail before the runtimes of these testing systems are presented in the following sections.

#### 4.1.1. Sequential Sewer System

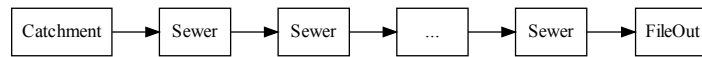


Figure 4.1.: Sequential Testing System

The sequential sewer system shown in Figure 4.1 can be seen as the simplest and basic sewer system used for benchmarking. The system has only one sink (an artificial source node emitting some values without meanings) and source node (a sink node which writes the inputs to a file. In between source and sink nodes a different number of sewers were placed, ranging from ten up to a 1000 sequential nodes.

The purpose of this testing system is to show that the FPS is not able to scale to more than one processor. This sounds pretty uninteresting but it also shows the overhead of running more threads than can be used by the FPS. The OPS should even gain in this scenario performance even though there are no parallel flows.

#### 4.1.2. Parallel Sewer System

The parallel sewer system, shown in Figure 4.2, consists of a number of parallel flows. Benchmarked were four and eight parallel streams. Each parallel stream of nodes consists of a number of sequential sewers like in the earlier introduced

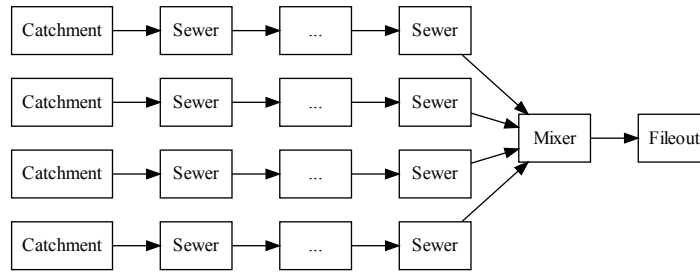


Figure 4.2.: Parallel Sewer Testing System

sequential sewer system. At the right end of the system a mixer combines the parallel flows and a file output node writes the results into a file.

The purpose of this system is to show what the maximum performance gain is by using the FPS. It can be seen as the counterpart to the sequential sewer system in that the sequential was used to demonstrate the weaknesses of FPS and the parallel sewer system is there to show its strenghts.

#### 4.1.3. Treelike Sewer Testing System

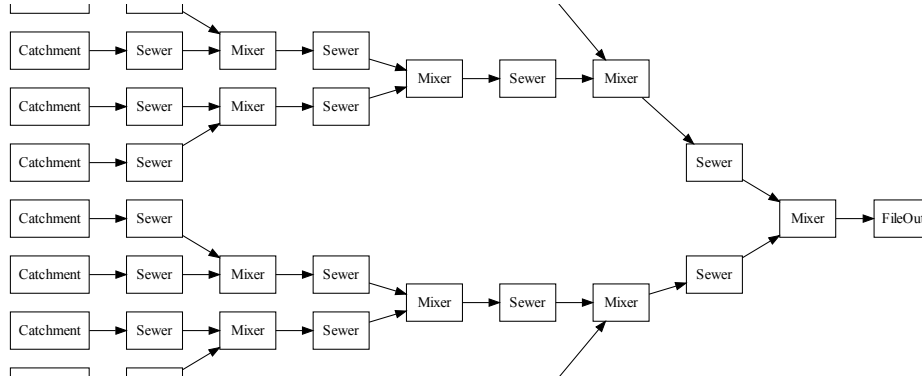


Figure 4.3.: Treelike Sewer Testing System

The treelike sewer system, shown in Figure 4.3, is again an artificial system although one that should be more natural than the two define earlier. Natural suply system have always a shape like a tree. They start very small at the households then the sewers get bigger and combine into the main sewer until they reach the WWTP. This shape of a tree like structure was tried to replicate by a, although, perfect tree. A script was able to generate trees at various

depths, two are included into the results one with small depth of four and one big system with a depth of ten.

The purpose of this system is to show the effects of different parallelization strategies on a system which is more or less shaped like real-world sewer system. The advantage of this artificial real world shaped system is that it is possible to change it in size.

#### **4.1.4. Real World Sewer System of Innsbruck**

The last system that was used for benchmarking purpose is the model of Innsbruck sewer system. The system was first modelled in Karen by Kleidorfer in [KMFRed]. The system was then converted into the native CityDrain3 format. This combined sewer system consists of 53 catchments, a total runoff effective area of 915 ha and a total basin volume of  $5100m^3$ . In the city of Innsbruck live 165,000 population equivalents (PE) with a daily dry weather flow (DWF) of  $200 \frac{l}{(PEd)}$ . [KMFRed]



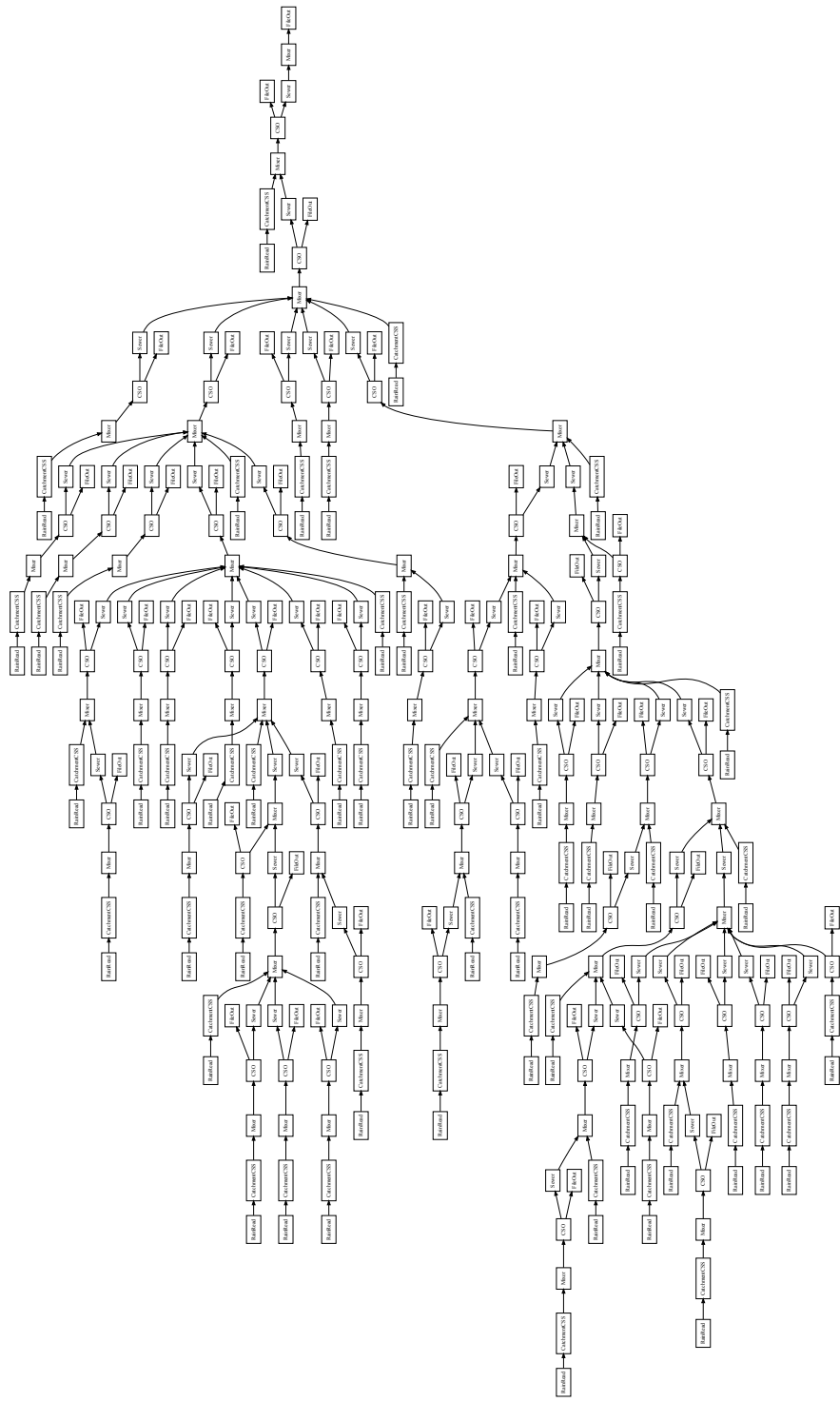


Figure 4.4.: Real World Testing System

## 4.2. The Benchmark Environment

The benchmarks were run on an up-to-date Linux (Ubuntu Intrepid Ibex). The distribution was targeted at the x86-64 architecture which has 64-bit wide registers. The CityDrain3 binaries were compiled to run on the 64-bit architecture. Two compilers were used to develop and test CityDrain3. The Gnu Compiler Collection (GCC) at version 4.4.0 and 4.3.1 and the Intel Compiler Suite at version 11. Both compilers are able to automatically parallelize some parts of the source code as part of the compiler optimizations. These features were turned off because this would have biased the outcomes of the benchmarks.

The compilers which were used for developing and testing was the GCC shipped with Ubuntu Intrepid Ibex at version 4.3.1. This compiler features OpenMP version 2.5 [RS09]. The compilers used for the benchmarking were the Intel Compiler Suite at Version 11. The OpenMP compiler of Intel conforms to the OpenMP 3.0 specification [Int09]. The Intel compilers were chosen because internal test showed that their OpenMP implementation performs better than the GNU OpenMP (GOMP).

Two different machines were used for running the benchmarks, both in the higher end consumer class available at that time. This kind of machines are normally used by civil engineers, the target users of the software. The first machine was an Intel(R) Core(TM)2 Quad CPU @ 2.40GHz with four GB of DDR2 main memory. This CPU has four cores and a 4MB L2 cache. The second processor was an Intel(R) Core(TM) i7 CPU 920 @ 2.67GHz with six GB DDR3 ram and eight MB L2 cache. This CPU has four hyperthreaded cores, which means up to eight hardware threads four being completely independent. The i7 CPU has two major advantages which should help boost parallel performance:

- *L3 shared cache* should lower the latency and amount of memory transfer between the cores and the mainmemory in case of cache misses.
- three memory channels allows more parallel access of the main memory.
- a faster DDR3-capable memory controller.

The model which was provided to the *cd3* binary has a fixed simulation time from 0 to 7200 with  $dt = 300$ . This is a simulation of two hours with a five minute timestep. The simulation time is constant because all the parallelization strategies are based on a structural decomposition of the urban drainage systems, therefore the simulation time does not influence the parallel behaviour of the strategies and was chosen small enough to reduce the runtime of a benchmark run.

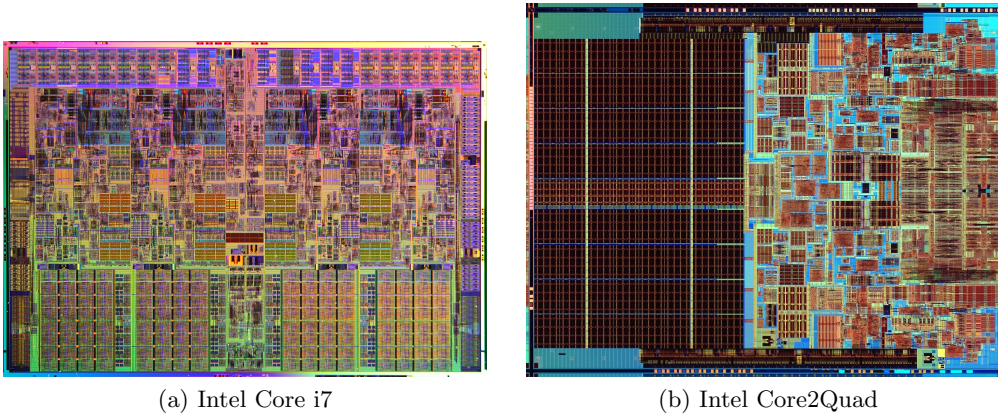


Figure 4.5.: Die Shots of the Two Processors used for Benchmarking

Each simulation system is run with a different number of allowed maximum threads. This was done in order to show how much time is used if more threads are added and allows in a certain way to predict the scalability if more threads are possible to run, maybe with future manycore CPUs having 32 cores [KAO05, ABC<sup>+</sup>06]. The benchmarks are repeated eight times and the minimum of the eight is the resulting runtime presented in the graphs in Section 4.5 and 4.4.

For each simulation system two graphs are shown. The x-axis is always the maximum number of threads the strategy is allowed to use. The left side graph is the runtime graph. It shows the time in milliseconds it takes a strategy to simulate a given drainage system. In the graph on the right hand side the speedup chart is shown. This chart shows the amount of speedup a strategy is able to gain by adding more threads. The base case of the speedup calculation is the single thread performance of the strategy. The speedup is calculated using the following formulas:

$$speedup_{n\ threads} = \frac{runtime_{1\ thread}}{runtime_{n\ threads}}$$

The best speedup of using  $n$  threads is  $n$  and is also shown in the charts.

The measured time on which all the runtime and speedup charts are based is the time a strategy takes to run the simulation, only the simulation. This time does not include setting up the simulation, like reading the XML-Model files. The time measurement start just before the first node is called and just after the last node finished the calculation of the last timestep.

### 4.3. Performance Tools

Throughout the course of programming CityDrain3 various tools were used to analyze the performance of the application. This section lists them and notes on which aspect of performance they were used.

*Valgrind* uses a virtual machine approach for performance measurements. This approach allows various applications of the tool and allows them to be very exact. The disadvantage of this is that the application runs several degrees slower [NS07, NS03]. *Valgrind* was used for profiling which allows to find weak performance spots, measure cache behaviour and it was used to find memory leaks.

A second set of tools which are not as heavy weight as *Valgrind* are the Google *perf-tools*. *perf-tools* offer a CPU profiler, a heap checker to detect memory leaks and a heap profiler for detecting excessive allocations. The *perf-tools* are targeting exact thread profiling.

The last tool is relatively new and was released as CityDrain3 was already finished. Because of that the tool was only used for result interpretation. The tool is called *mutrace* and is available at <http://0pointer.de/blog/projects/mutrace.html>. It shows the contention, and other information, of locks used in multithreaded programs. An output of the program can be seen in Listing 4.1. It shows the number of contentions, i.e. how often a lock was tried to lock when it was already locked, the total wait time for the lock, etc.

```
$ mutrace ./build/cd3 -v 1 data/models/paper/long-100-ordered.xml
...
mutrace: Showing 10 most contended mutexes:
  Mutex #  Locked  Changed  Cont.  tot. Time [ms]  avg. Time [ms]  max. Time [ms]  Flags
  196      237    203      9      0.126          0.001          0.001  Mx.--
  118      222    204      3      0.114          0.001          0.002  Mx.--
  192      204    111      3      0.116          0.001          0.002  Mx.--
   92      207    131      2      0.133          0.001          0.005  Mx.--
   80      204    131      2      0.139          0.001          0.003  Mx.--
  157      264    203      1      0.188          0.001          0.002  Mx.--
   97      298    197      1      0.703          0.002          0.517  Mx.--
   96      276    167      1      0.177          0.001          0.002  Mx.--
  148      211    144      1      0.141          0.001          0.002  Mx.--
  159      204    123      1      0.132          0.001          0.002  Mx.--
  ...      ...      ...      ...      ...          ...          ...      |
  Object:                                     M = Mutex, W = RWLock /|
  State:                                       x = dead, ! = inconsistent /|
  Use:                                         R = used in realtime thread /|
  Mutex Type:                                r = RECURSIVE, e = ERRORCHECK, a = ADAPTIVE /|
  Mutex Protocol:                            i = INHERIT, p = PROTECT /|
  RWLock Kind:  r = PREFER_READER, w = PREFER_WRITER, W = PREFER_WRITER_NONREC /|
  ...
```

Listing 4.1: Output of *mutrace*

The Google *perf-tools* and *mutrace* use an interception library approach, see [God02, NT05] for more information on interception libraries.

## 4.4. Results for the Core2Quad CPU

The results presented here ran on an Intel Core 2 Quad presented in Chapter 4.2. This processor is able to run four threads independently. Although two threads share a L2 cache. The benchmark results ran up to ten threads in total. This was done in order to show what happens if the algorithm is allowed to use more threads than the hardware is able to run in parallel.

### Sequential Sewer Systems

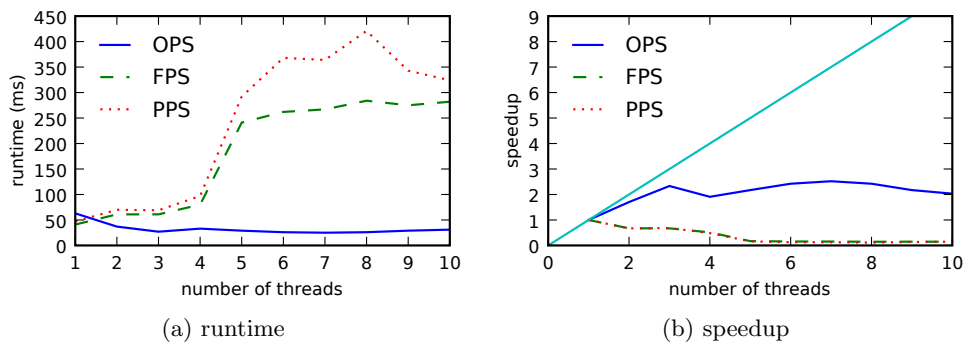


Figure 4.6.: Sequential (10)

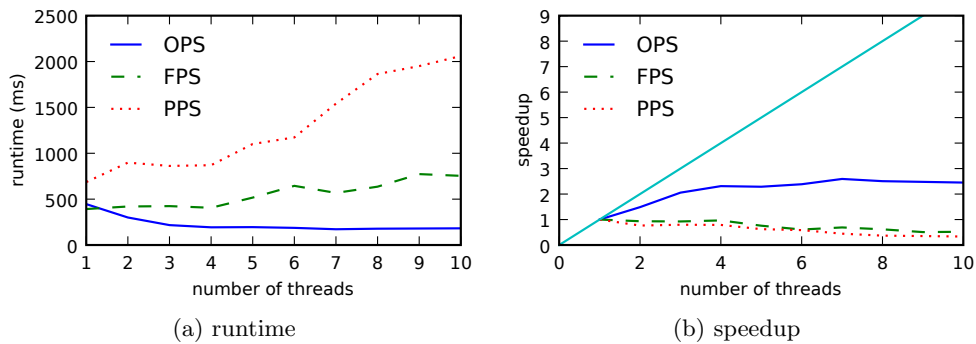


Figure 4.7.: Sequential (100)

Figure 4.6, 4.6 and 4.8 show the results of the sequential sewer system with ten, 100 and 1000 consecutive nodes. FPS has a good single thread performance, better than OPS, in the small systems of ten and hunderet nodes. But FPS is per definition not able to scale on sequential systems because it tries to calculate parallel sewer sub-systems in parallel. The sequential testing system don't offer parallel sewer sub-systems.

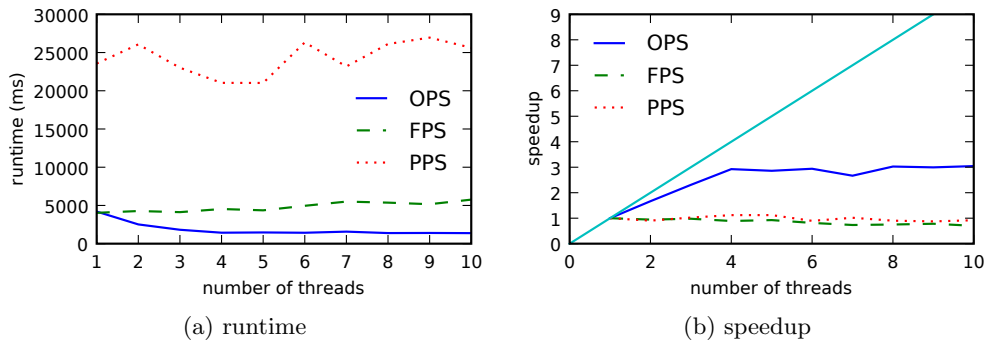


Figure 4.8.: Sequential (1000)

After exceeding the thread count the hardware is able to run natively the performance of FPS seems to degrade. This effect could be caused by memory stalls due to cache misses which arise when the OpenMP thread scheduler decides to put a thread on a core which was previously calculated on a different core. A CPU with a shared L3 cache like the i7 system offers could compensate this effect. And indeed, this effect is not as dramatically on the i7 system.

The longer, i.e. bigger, the system the less this effect carries weight. Figure 4.8 shows the system with 1000 sequential nodes in which FPS stays steady in the runtime with more threads added.

OPS is not as good in single thread performance than FPS. This is because the management of the nodes is done by a queue and even in single threading the queue is protected by a lock in order to prevent corruption of the internal data structures. OPS holds a queue between two timesteps. The testing systems have all the same simulation time from  $t = 0 \dots 7200$  in  $dt = 300$  steps. This results in 24 simulation runs and therefor 24 queues.

Beside the fact that OPS is a little slower in single thread performance it scales even on the most sequential structure a sewer system can be. Even on small systems (Figure 4.6) OPS offers very good performance on this quad-core system. The bigger the systems get the more stable the results get.

PPS was mentioned to be not that efficient due to its random nature and the coarse grain locking. It gets even worse when the node count increases, because there are even more nodes to check for satisfied preconditions. Preconditions that need to be satisfied in order to run a selected node. It shows some scaling in Figure 4.8 but the best runtime at four threads does not even reach single thread performance of FPS or OPS.

## Parallel Sewer Systems

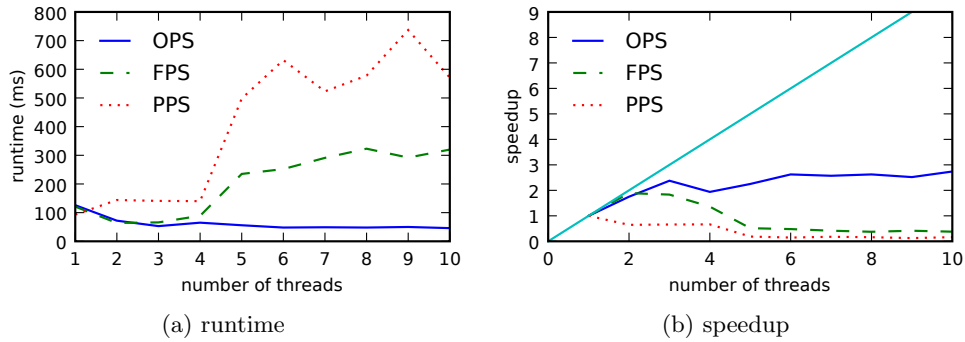


Figure 4.9.: Parallel (2-10)

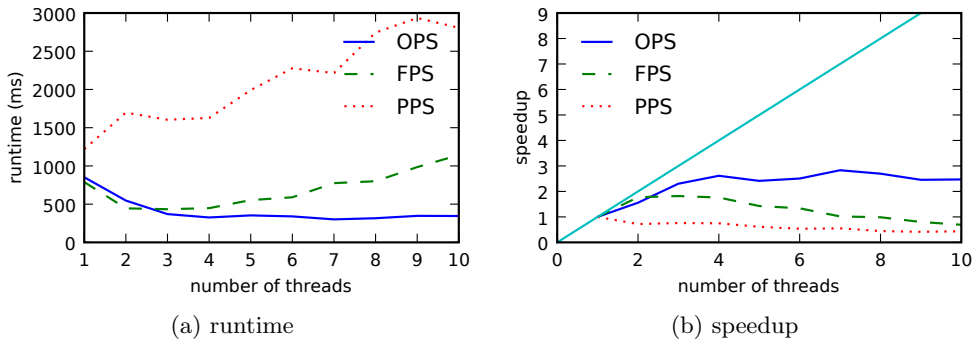


Figure 4.10.: Parallel (2-100)

Figure 4.9, 4.10, 4.11, 4.12, 4.13 and 4.14 show the parallel sewer systems with two parallel ten sequential, two parallel 100 sequential, four parallel ten sequential, four parallel 100 sequential, eight parallel ten sequential and eight parallel 100 sequential. The number of nodes in the systems are  $s * p + 2$  because there are  $p$  parallel streams, each parallel stream contains  $s$  consecutive nodes plus one mixer and one fileout node.

Figures 4.9 and 4.10 show the expected result that the FPS scales only for two threads. The performance plateaus for the third and fourth thread added. After the fourth thread the same effect as in the sequential systems shows up, the performance drops.

Figure 4.11 and 4.12 are the most optimal simulation system for FPS. Four parallel streams every one able to be computed by the four available cores of the CPU. In the small system (Figure 4.11) the performance is well but degrades again after four threads. In the big systems shown in Figures 4.12 and 4.14 FPS

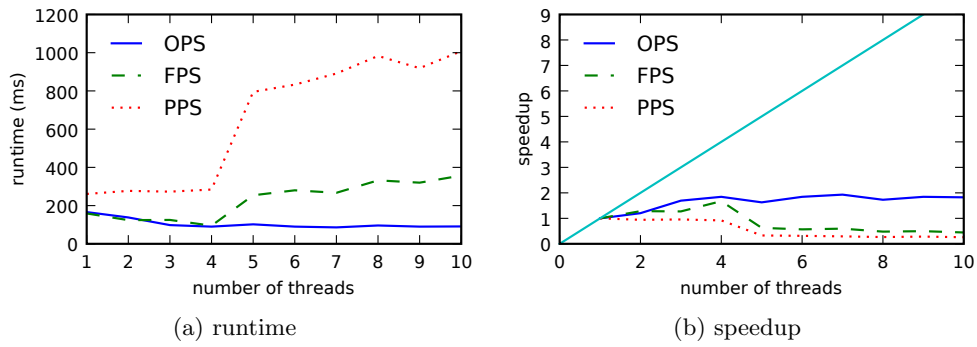


Figure 4.11.: Parallel (4-10)

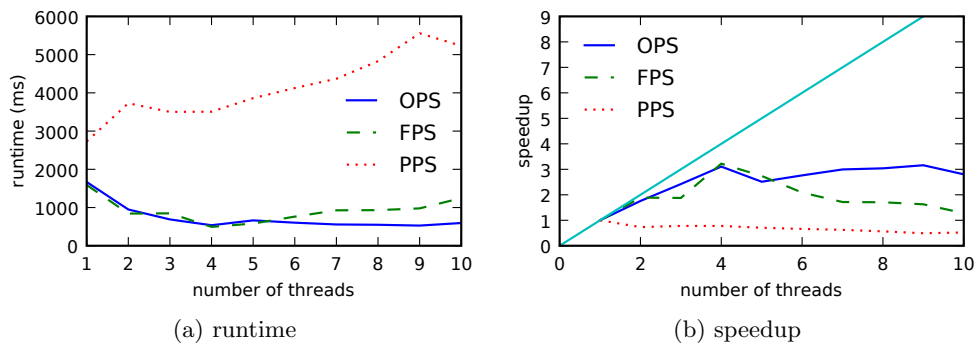


Figure 4.12.: Parallel (4-100)

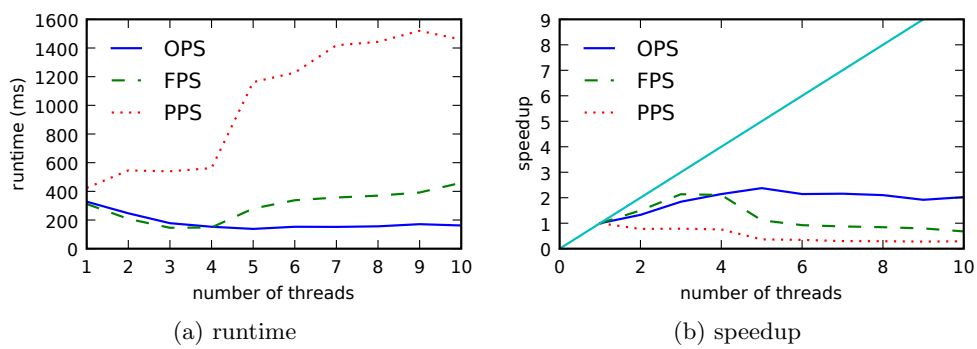


Figure 4.13.: Parallel (8-10)



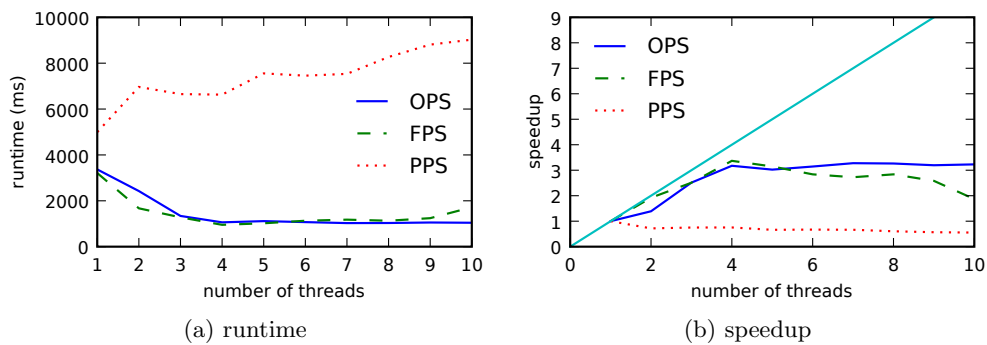


Figure 4.14.: Parallel (8-100)

shows its strengths on the parallel testing system and beats or is at least as fast as OPS up until four threads. Although the performance is exceptional well for this system, chances are small that such a system is ever researched beside in this work.

OPS scales very well on all ranges of parallel sewer systems. But the more nodes the system contains the better OPS scales, because the overhead of the queues gets neglectable. If a system of parallel sewers is as large as in Figure 4.14 the difference of FPS and OPS are minor and they both scale very well.

PPS is again not able to gain any speedup by adding more threads and parallel hardware. The parallel sewer systems show the same results as the sequential ones, the performance rises exponentially after three additional threads.

### Treelike Sewer Systems

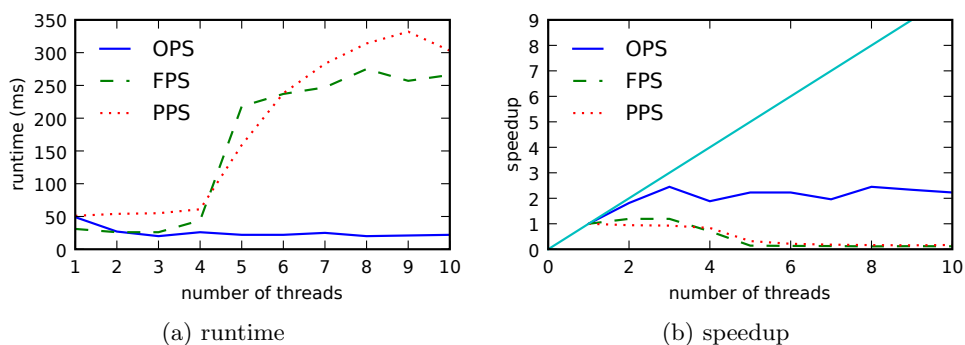


Figure 4.15.: Tree (two generations)

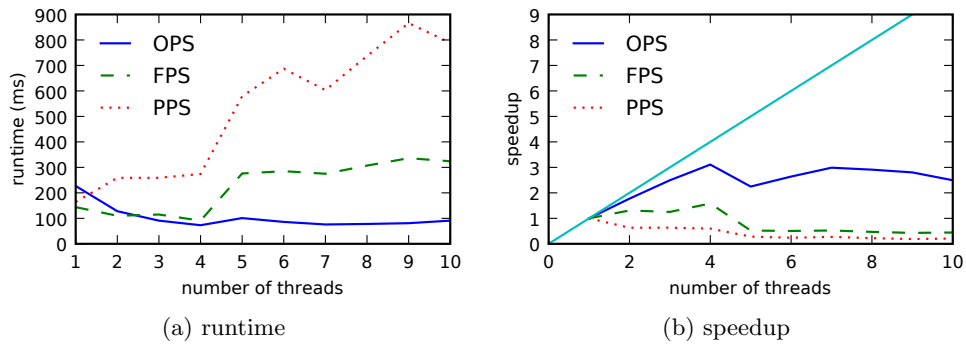


Figure 4.16.: Tree (four generations)

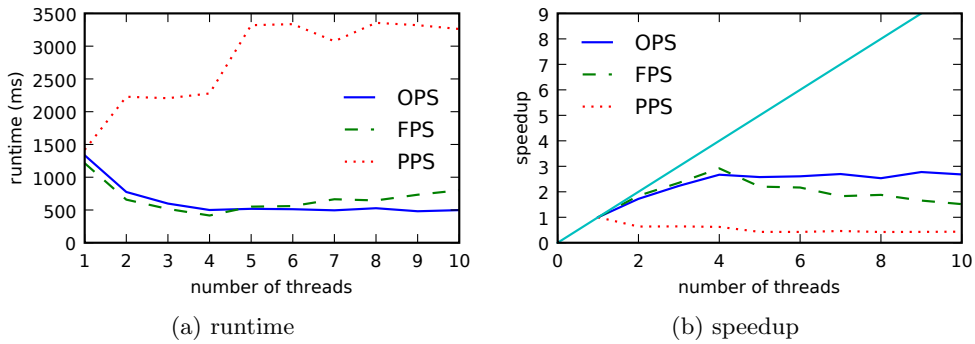


Figure 4.17.: Tree (seven generations)

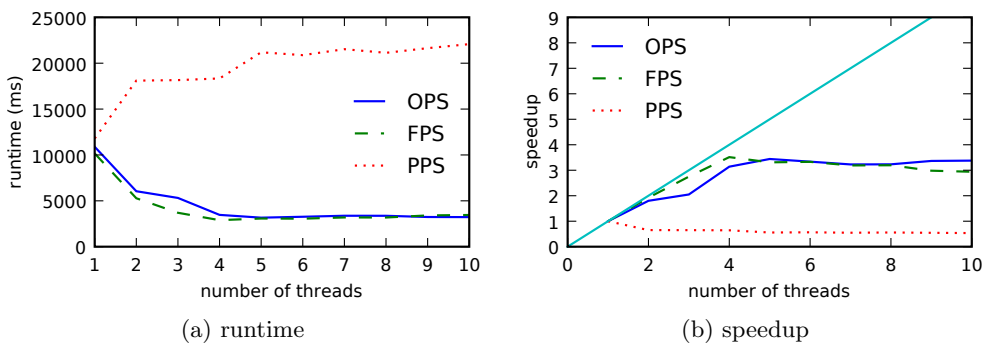


Figure 4.18.: Tree (ten generations)

Figure 4.15, 4.16, 4.17 and 4.18 show tree sewer systems with two, four, seven and ten generations. The number of nodes in a system is  $2^{g+1} - 1$  for  $g$  generations. This results in  $2^3 - 1 = 7$  nodes for Figure 4.15 and  $2^{11} - 1 = 2047$  nodes for Figure 4.18. The number of parallel streams starts with the number of leaf nodes in a perfect binary tree and halves at each level down to one. The number of leaf nodes can be calculated using  $2^g$  for  $g$  generations which is  $2^2 = 4$  for two generations and  $2^{10} = 1024$  for ten generations.

FPS performs well on all ranges of trees. The bigger the tree the better the scaling of FPS. On the small system FPS has again better single thread performance but performance degrades after four threads. At five threads it is even worse than PPS which is otherwise the worst in all tests. The performance drop after four threads is noticeable at all ranges of tree sizes, but it gets better if the tree is bigger.

OPS is again in all tests the fastest with the most stable results throughout the tested tree systems. Even the small systems show good scaling on more threads. Figure 4.17 and 4.18 show better results for FPS in the speedup charts, a view on the left side reveals that the overall performance of OPS is better than FPS. On thread count from one to four FPS and OPS are on par.

PPS is again the worst with no increase at all in speed at neither tree system.

## Realworld Testingsystem Innsbruck

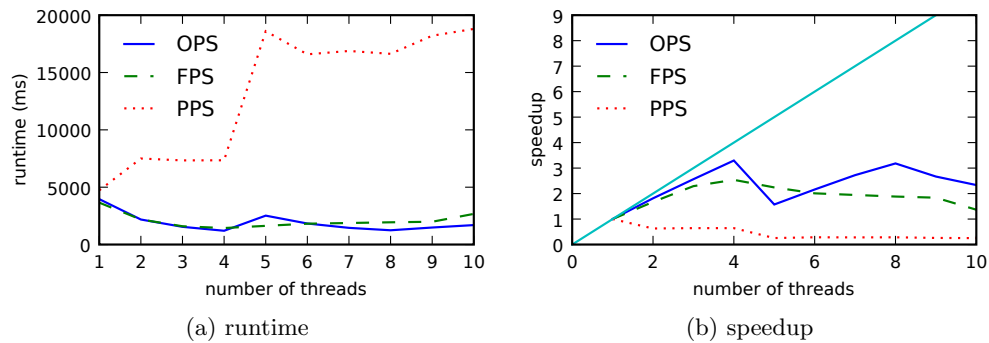


Figure 4.19.: Innsbruck

Figure 4.19 shows the real world sewer system of Innsbruck. It has enough parallel streams to get a speedup with the FPS. The OPS performs best at this sewer system and reaches a speedup of up to 3.3. OPS has the better single thread performance and the better peak performance. OPS shows a performance

degrade after four threads which it does not in other testing systems. PPS does not scale in this test either.

## OpenMP Scheduling Effects

As mentioned earlier OpenMP shows weak performance on the quad core system when the thread count rises above the physical available core count of four. Test results for small systems are highly influenced by this effect. It is weaker on the bigger testing systems and almost gone on the i7 system because of a shared L3 cache.

The described effect arises because chances are high that a thread that has previously been executed on a core will be scheduled on another core. If that happens the thread must be migrated from one core to another. Table 4.1 shows a round robing scheduling of five threads on four cores. Even at the first step migration of thread one from core one to four is needed. Table 4.2 shows

step	core 1	core 2	core 3	core 4
1	thread 1	thread 2	thread 3	thread 4
2	thread 5	thread 1	thread 2	thread 3
3	thread 4	thread 5	thread 1	thread 2
⋮	⋮	⋮	⋮	⋮

Table 4.1.: Thread Scheduling with three threads

the same scheduling but now with an optimal thread count of four. There are no migrations at all. The same is true for two threads (optimal thread scheduling assumed). Figure 4.12 shows the results of a not optimal thread-

step	core 1	core 2	core 3	core 4
1	thread 1	thread 2	thread 3	thread 4
2	thread 1	thread 2	thread 3	thread 4
3	thread 1	thread 2	thread 3	thread 4
⋮	⋮	⋮	⋮	⋮

Table 4.2.: Thread Scheduling with four threads

scheduling performed by the OpenMP implementation of the Intel Compilers. At first sight the results are very good because they show a small super linear speedup. Super linear speedups are possible due to an effective bigger cache size from the additional cores [AJS07], but this effect is not based on these facts. Examined closely, Figure 4.12 unveils the cause of the superlinear speedup to be an exceptional slow single thread execution time. Because the speedup is

calculated with the single thread runtime at the base case, this system shows a superlinear speedup. This effect occurs only at some systems, parallel systems with four parallel streams and 100 sequential (Figure 4.11) and the big tree systems (Figure 4.17 and 4.18). The slow performance is caused by excessive scheduling of the single thread on all available cores. CPU usage is at 25% of all cores instead of 100% on a single core.

Disabling the thread affinity of the OpenMP scheduler solves this problem. This can be done by setting the environment variable *KMP\_AFFINITY* to the value *"nooverbose , disabled"*. Figure 4.20, 4.21, 4.22 show the results of the effected systems with the OpenMP scheduler disabled.

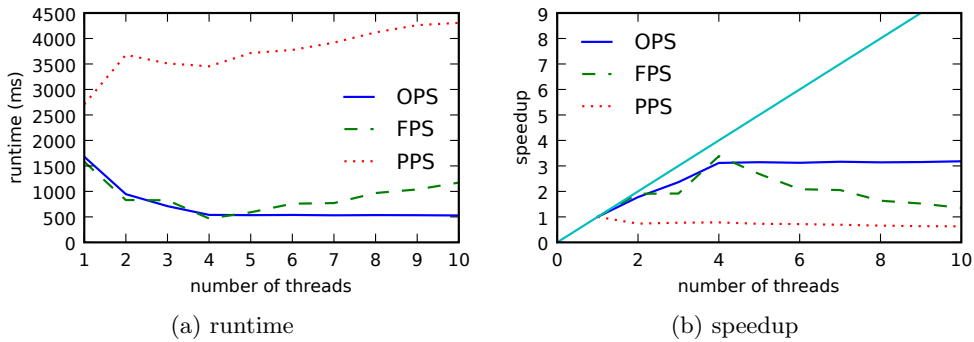


Figure 4.20.: Parallel system with OpenMP scheduler disabled(4-100)

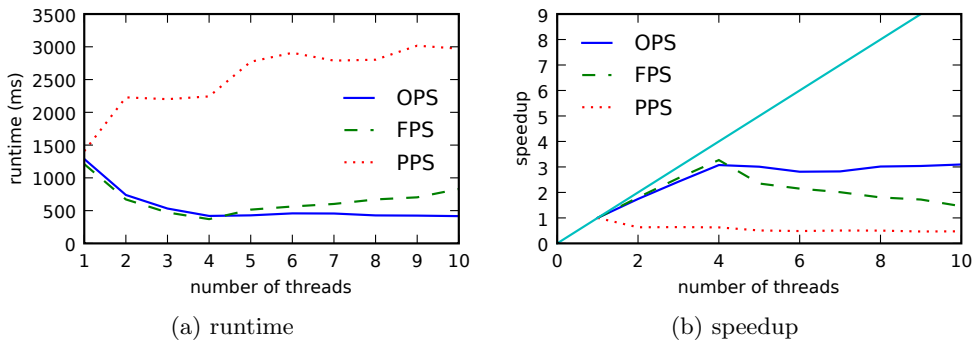


Figure 4.21.: Tree system with OpenMP scheduler disabled (seven generations)

Disabling thread affinity influences the overall performance of FPS in a positive way, FPS offers better performance than OPS at four threads with thread affinity disabled. Although the performance is better on this system, it is a totally different situation on other CPUs with different core layouts and is not further explored in this work.

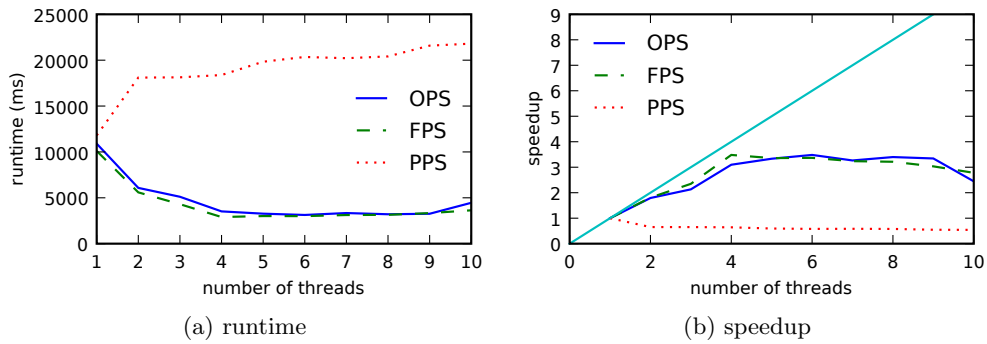


Figure 4.22.: Tree system with OpenMP scheduler disabled (ten generations)

The scheduling effects that were described are not effecting OPS because OPS does not use OpenMP and the threads are scheduled by the operating system instead of OpenMP. The underlying reasons for this regressions are not known, they could as well be caused by the linux thread scheduler used in desktop systems.

## 4.5. Results for the i7 CPU

This section describes the results for the Intel Core i7 (i7) hardware testing system. The i7 is a little bit different compared to the Core2Quad system. This differences should benefit the parallel performance of CityDrain3. The biggest advantage of this processor is a 8MB big shared L3 cache. The cache forms a third level in the memory hierarchy. The cache is shared and used by all four cores. This shared characteristics of the cache should speedup the inter-core communication and should reduce the costs of migrating a thread. Cache related impacts of processors on the performance of computer programs can be further read in [Dre07].

The tests ran with up to ten threads because this processor provides eight hardware threads, although each pair of thread shares the ressource of a core.

### Sequential Sewer Systems

Figure 4.23, 4.24 and 4.25 show the result for the ten, 100 and 1000 sequential testing system running on the i7 system.

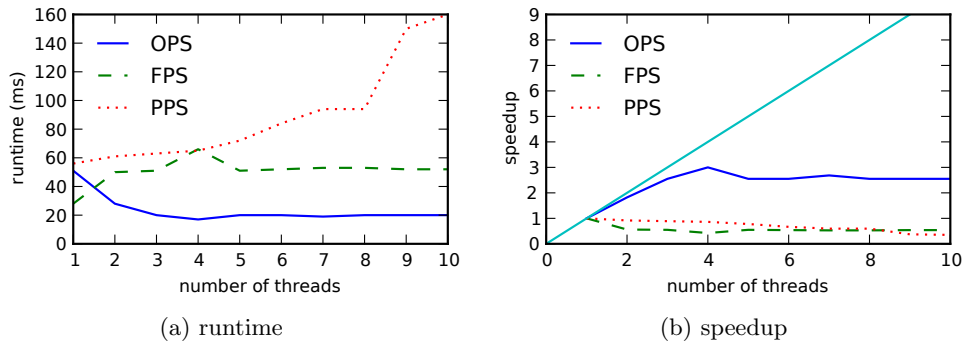


Figure 4.23.: Sequential (10)

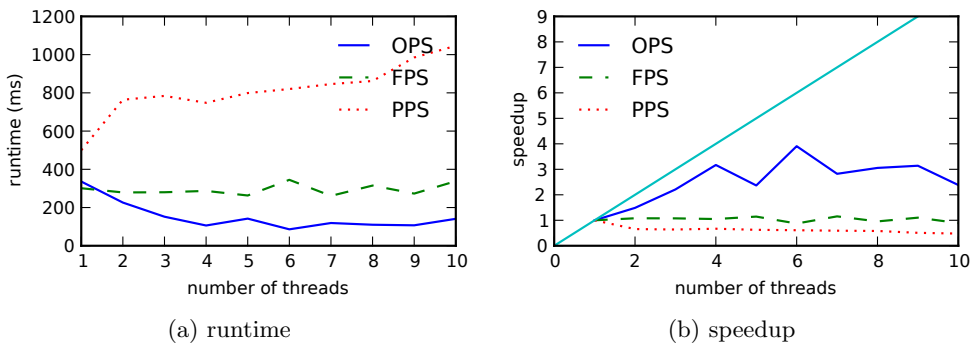


Figure 4.24.: Sequential (100)

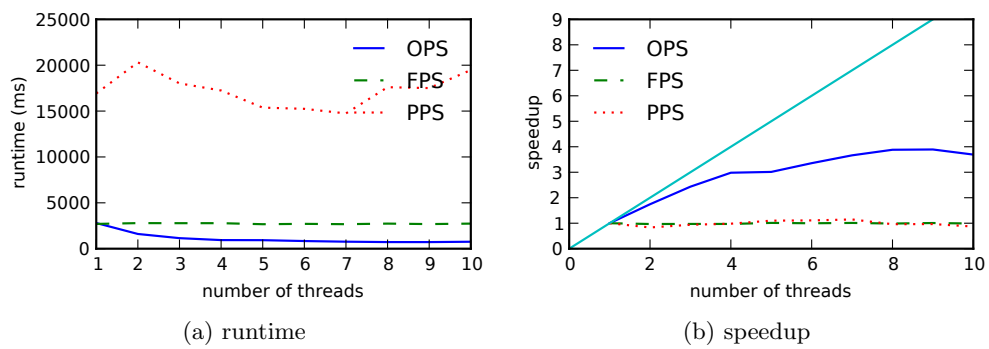


Figure 4.25.: Sequential (1000)

As already mentioned before, FPS can not scale when executed on sequential testing systems. Although it does not scale on this architecture it shows less overhead compared to the Core2Quad system shown in Figure 4.6. In the small system (in Figure 4.23) a little peak can be seen at four threads. FPS is the fastest strategy at single thread performance for the small system, the bigger the systems get the less bumpy the runtime curve of FPS becomes and the single-thread performance difference between OPS and FPS diminishes. This means that the overhead from using more threads than FPS can handle is getting lesser the bigger the system becomes. For the long sewer system of 1000 nodes the overhead is not even recognizable, see Figure 4.25.

OPS and PPS have almost the same single thread performance at the small system of ten nodes (Figure 4.23). FPS has the best single thread performance. This changes in the system with 100 nodes (Figure 4.24) and has the worst outcome for PPS at the testing system with 1000 nodes (Figure 4.25), where PPS is six times slower on a single thread than OPS.

The reason PPS is that slow on large sequential systems is because the probability of randomly finding a node that is able to run is getting smaller and smaller the bigger the systems get. In case of the sequential systems this can be easily calculated. At the first node the probability of finding a node that is able to run is to select one node out of the pool of the un-run nodes. This means  $\frac{1}{np}$  for  $np$  as pool size which is  $\frac{1}{1000} = 0.001$  for the large system. Finding the right node is the overhead of this strategy. This overhead is too big for achieving a speedup, in fact the overhead slows the strategy down enormously.

OPS performed best in this testing systems. Even the small system shows good results. The results are getting better the bigger the systems get. The speedups are 3.0 for ten nodes, 3.5 for 100 nodes and 3.8 for 1000 nodes. The performance of OPS is best at four threads for the small systems, at six threads for 100 nodes and eight threads for 1000 nodes.

## Parallel Sewer Systems

The parallel testing systems for the i7 are shown in Figure 4.26, 4.27, 4.28, 4.29, 4.30 and 4.31.

Figure 4.26 and 4.27 show the expected effect of FPS not getting faster after two threads because these system don't provided more than two parallel streams. The thread scheduling effects described in Section 4.4 are not recognizable on this machine because of a shared L3 cache that is available for the i7 CPU.

At the system with four parallel streams FPS performs very good and is on



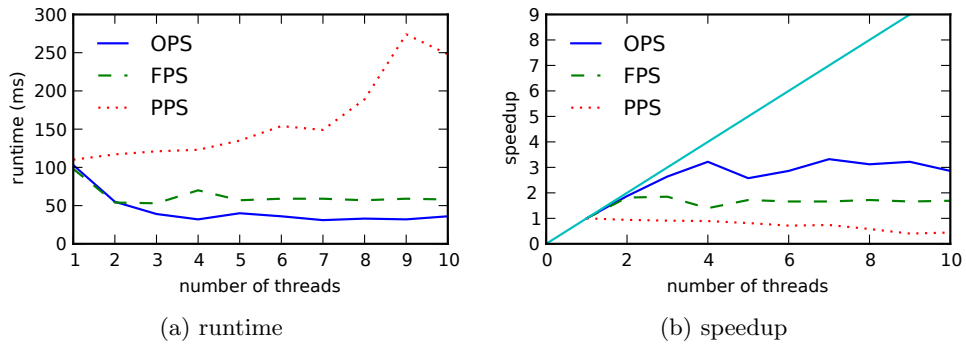


Figure 4.26.: Parallel (2-10)

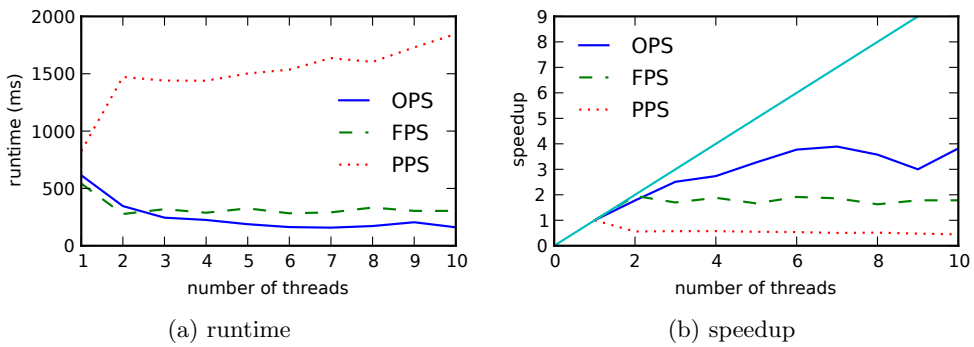


Figure 4.27.: Parallel (2-100)

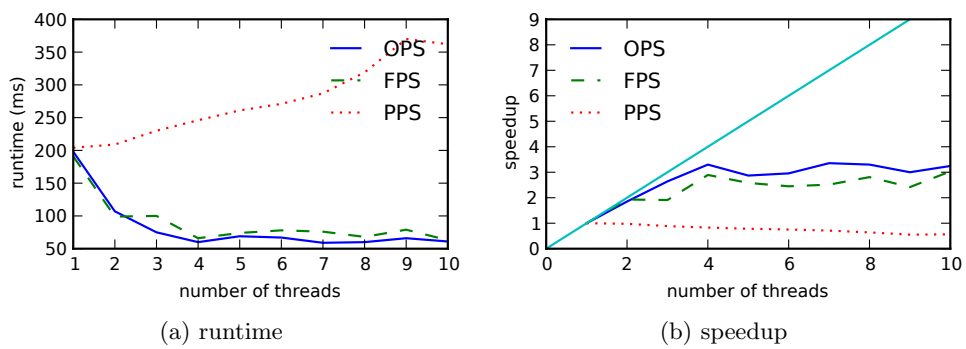


Figure 4.28.: Parallel (4-10)

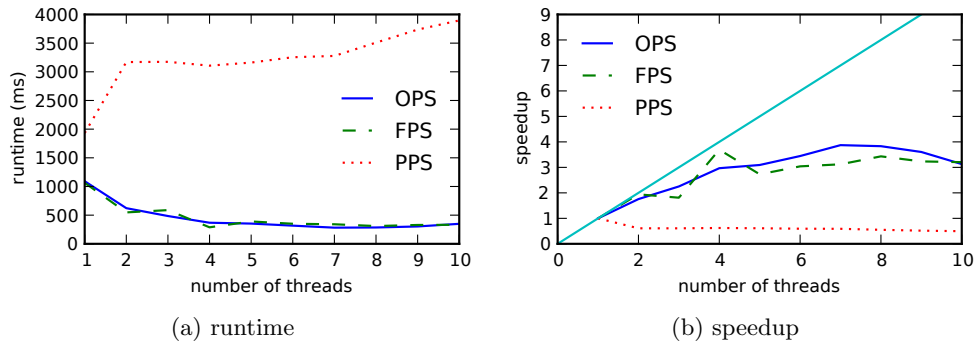


Figure 4.29.: Parallel (4-100)

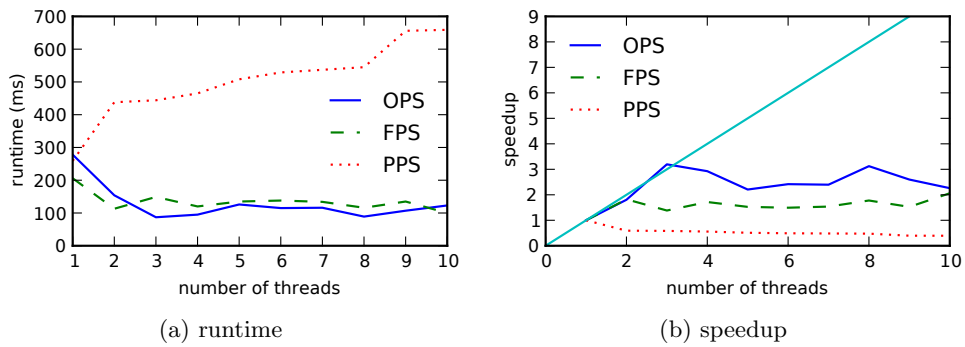


Figure 4.30.: Prallel (8-10)

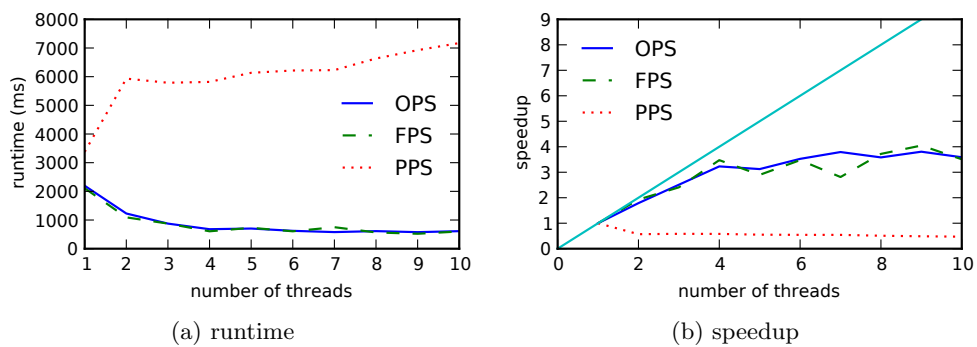


Figure 4.31.: Parallel (8-100)

par with OPS. In the systems with eight parallel streams FPS needs the bigger system to get good results comparable to OPS.

OPS performs best at all systems and has a small edge ahead of FPS. In all systems even the small ones with ten sequential nodes the OPS performs well and scales well to the four available cores.

### Treelike Testing Systems

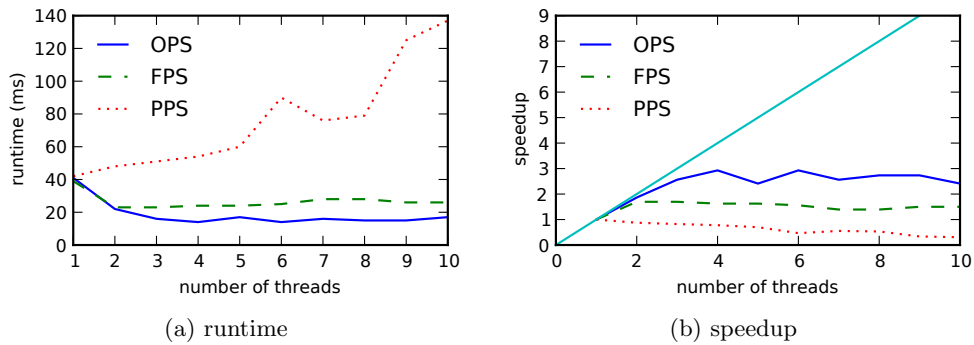


Figure 4.32.: Tree (two generations)

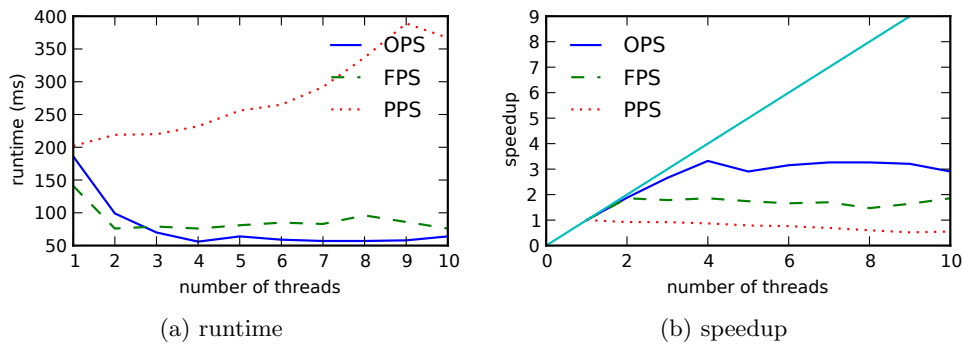


Figure 4.33.: Tree (four generations)

The tree testing systems are shown in Figure 4.32, 4.33, 4.34 and 4.35 for two, four, seven and ten generations.

FPS, due to its limitations, can only scale at the big systems, above four generations. FPS shows a small peak around five threads in all the big systems. This is due to the scheduling of the OpenMP threads.

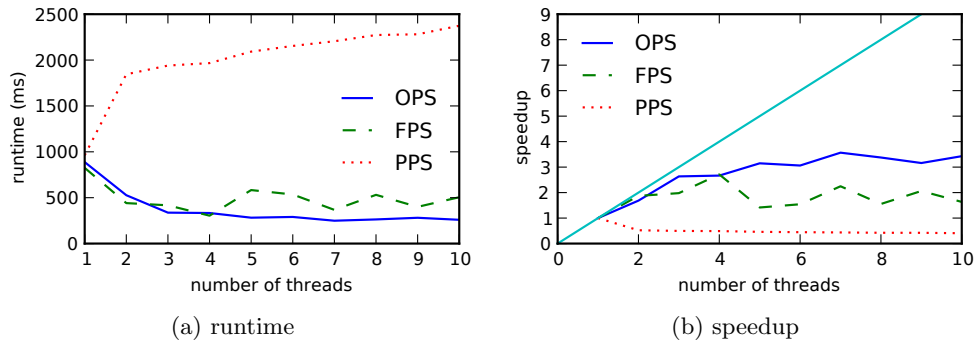


Figure 4.34.: Tree (seven generations)

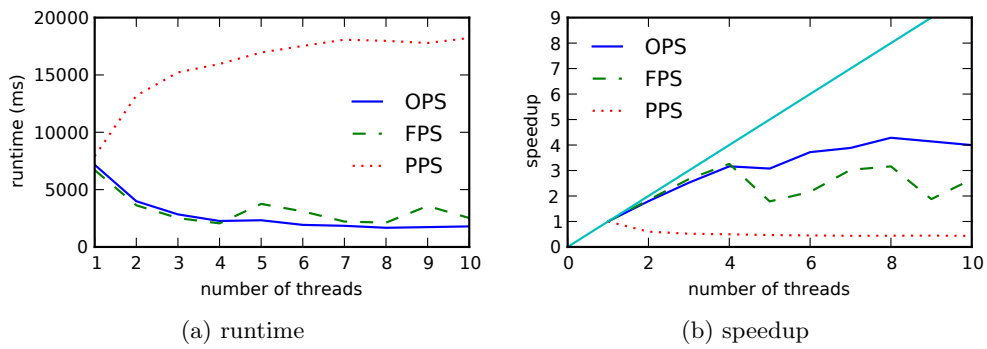


Figure 4.35.: Tree (ten generations)

OPS shows good results on all systems, but also shows a small degrade, although not as big as in FPS, around five threads. The speedups of OPS are 2.9 for two, 3.3 for four, 3.5 for seven and 4.2 for ten generations.

The speedup of 4.2 of OPS is above the physical core count which means that OPS benefits of the extra parallelism provided by the hyper-threaded cores in the i7. Although there is a degrade in performance at four threads with the small system performance recovers at higher thread counts, this is in contrast to the Core2Quad system shown in Figure 4.15.

### Realworld Sewersystem Innsbruck

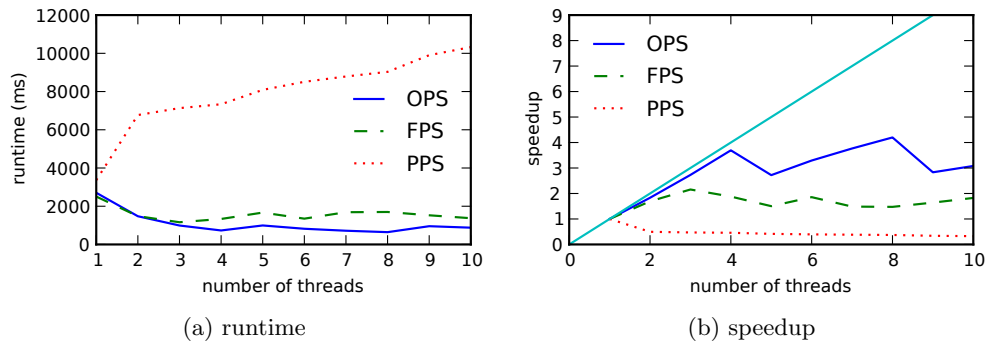


Figure 4.36.: Innsbruck

Figure 4.36 shows the results for the real-world testing system of Innsbruck. This system has essentially a shape of an existing sewer system which can be found under the city of Innsbruck. This system represents one statistical sample of a real sewer system. That is why this system is the most essential one and the results of this system are there to show how much speedup is able on systems engineers need to work with everyday.

FPS and OPS are on par up until three threads. From there on OPS scales up to 4.19 at eight threads and FPS up to 2.15 at three threads. Both perform very well although OPS has a slight edge ahead. FPS is not as fast as OPS because this model which can be seen in Figure 4.4 has lots of mixing and splitting elements which forces to tear down and start up new OpenMP threads in FPS. This heavy switching of threads causes an overhead that leads to a less optimal performance of FPS in real-world systems that are similar to that found in Innsbruck.

## 4.6. Shared Flow Comparisons

Section 3.4 showed the implementation details of the *Flow* class and in special how it offers a lazy copy on write semantics which is called the shared flow. This section compares simulation runs with the shared flow semantics enabled and disabled. Whether it is beneficial for the overall runtime and/or the speedup to use a shared flow in urban drainage modelling.

The results are presented with charts showing how the implementations scale by using more threads. On the left side are the runtime charts and on the right side the speedup charts. In a chart a single system ran by a single parallel strategy with shared flow enabled and disabled.

This section deliberately shows only two systems (Innsbruck and the tree system with ten generations) and the two good working strategies (FPS and OPS).

### 4.6.1. Shared flow results for Core2Quad CPU

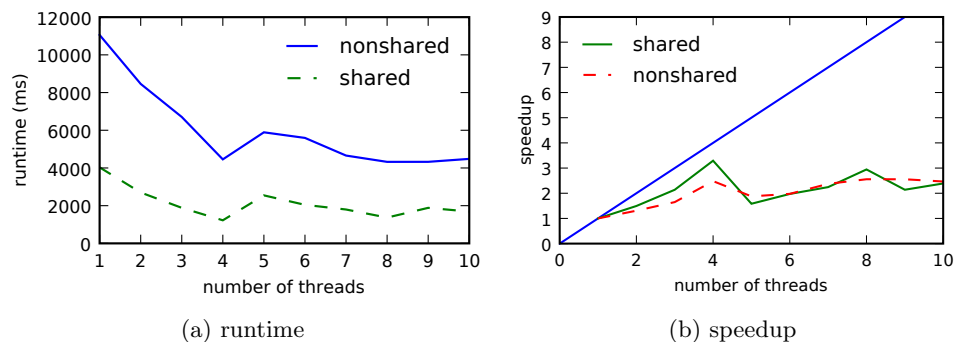


Figure 4.37.: Innsbruck OPS

In the Innsbruck testing system, shown in Figure 4.37 and 4.38, both strategies show a massive runtime gain and a small speedup gain when the shared flow is enabled. OPS shows more fluctuations when the shared flow is enabled after exceeding the four cores. OPS benefits more with a speedup from around 2 to over 3, while FPS improved from 2 to nearly 3.

Figure 4.39 and 4.40 show the shared flow comparisons with the big tree system of ten generations. These graphs are a lot smoother than the one of the Innsbruck system. At both strategies the non shared variants are able to speed up after the four threads but aren't able to reach the speedups when shared flow enabled. FPS with a shared flow shows almost linear speedup until four

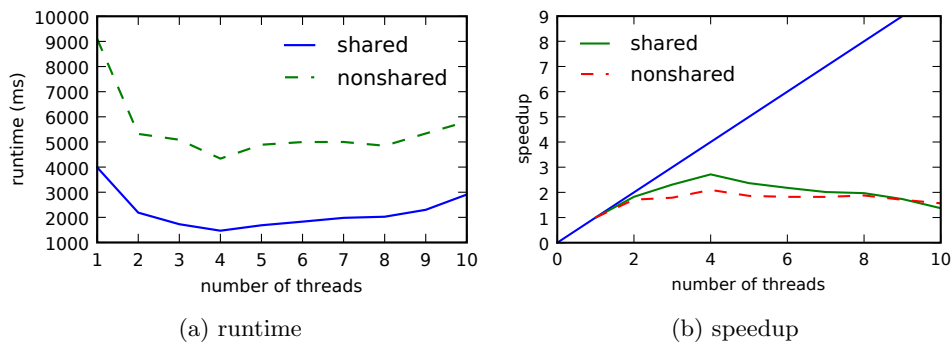


Figure 4.38.: Innsbruck FPS

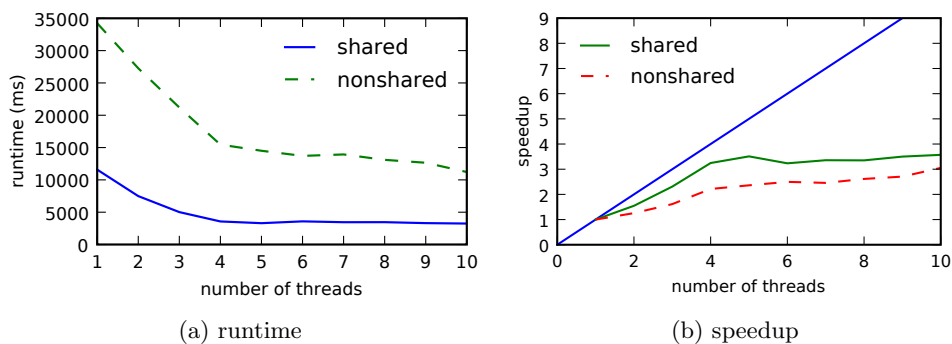


Figure 4.39.: Tree (10 Generations) OPS

threads when the shared flow is used and gains a lot more by using a shared flow. Speedup doubled from 2 to 4 at FPS and from 2 to over 3 with OPS.

#### 4.6.2. Shared flow results for the i7 CPU

This section describes the shared flow results for the i7 CPU which features a large L3 cache that is shared by the four cores. Because of this the effects of copying around large amounts of flow should be not as dramatically as in a system without a L3 cache.

Figure 4.41 and 4.42 show the results for Innsbruck respectively for OPS and FPS. The overall runtime has been reduced dramatically by using a shared flow. The runtime more than halves by using a shared flow. The speedup is almost identical although OPS showed a speedup improvement at eight threads. FPSs speedup is identical for shared vs. nonshared flows.

Figure 4.43 and 4.44 show the results for the tree testing system with ten

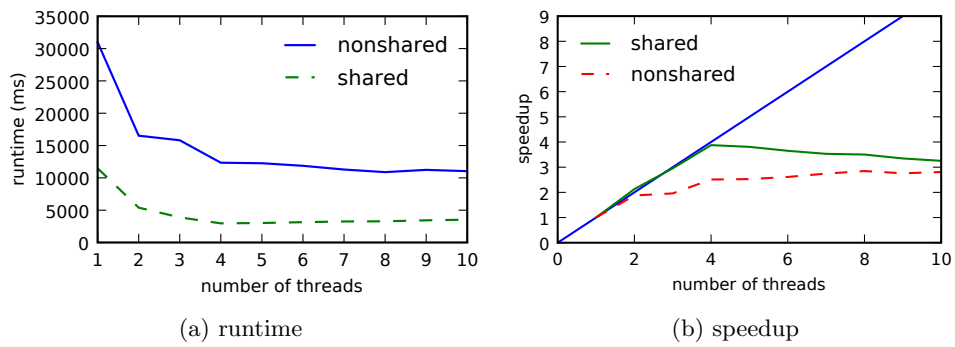


Figure 4.40.: Tree (10 Generations) FPS

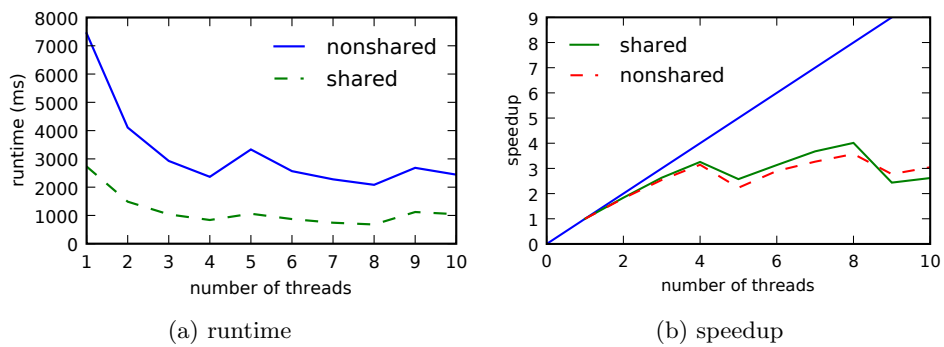


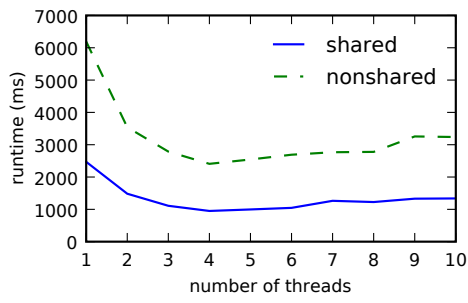
Figure 4.41.: Innsbruck OPS

generations. Again the overall runtime is more than cut in halve. On this testing system both FPS and OPS benefit from using a shared flow with a better speedup.

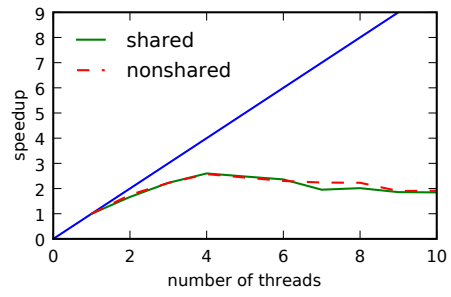
### 4.6.3. Conclusion

It seems that the overall runtime of the systems is halved, even more in some tests. This concludes that using a shared flow reduces the runtime even on single thread performance. The speedup gains also from using a shared flow, although not as dramatically as the runtime. The i7 system doesn't speed up a lot more on shared flow whereas the Core2Quad system doubled the speedup in the tree system with ten generations.



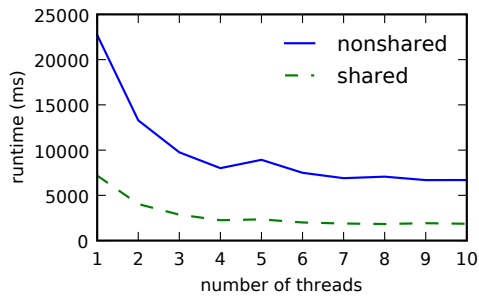


(a) runtime

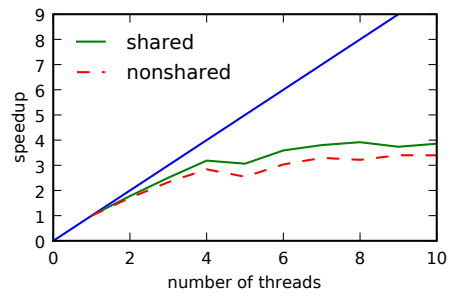


(b) speedup

Figure 4.42.: Innsbruck FPS

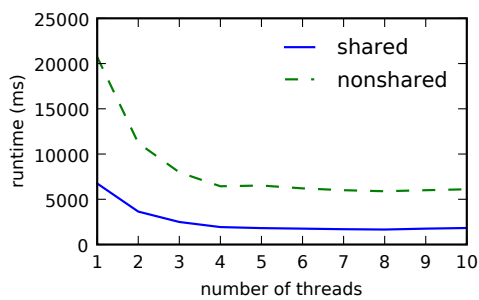


(a) runtime

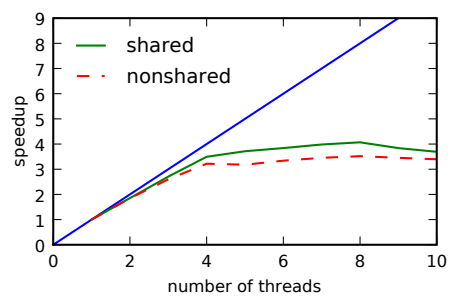


(b) speedup

Figure 4.43.: Tree (10 Generations) OPS



(a) runtime



(b) speedup

Figure 4.44.: Tree (10 Generations) FPS



## Chapter 5.

### Conclusion

CityDrain3 was developed with the aim of speeding up urban drainage simulations by exploiting the available multi-core architectures. CityDrain3 steps into the footsteps of CITY DRAIN, a Matlab based urban drainage modelling toolkit. With regards to urban drainage simulations CITY DRAIN and CityDrain3 are feature equivalent.

Three different parallelization strategies have been found and put into tests by various different testing systems. These testing systems were chosen such that the weak and sweet spots of the strategies were hit hard. The testing systems even included a real world system, representing the sewer system of Innsbruck, developed for CITY DRAIN and converted into CityDrain3s native input format. Beside the real world system, a sequential system, a parallel system and a tree shaped system were used to analyse the strategies. Sewer systems that included waste water treatment plants were not included. Results of this kind of systems are interesting in the future especially if the processes in the treatment plant are complex.

The three parallel strategies are the flow parallel, pool pipeline and the ordered pipeline strategy. The flow parallel strategy was used to investigate the possibility of calculating parallel flow streams concurrently. The pool pipeline strategy exploits the fact the time discrete simulations may be split up on a timely based manner. This allows to even parallelize sequential sewer systems which aren't able to be sped up by the flow parallel strategy. The pool pipeline strategy maintains a pool of nodes for each discrete timestep. A single thread is responsible for randomly choosing a node out of the pool, checking for satisfied dependencies based on the sewer system and performing the nodes calculations. A timestep is finished if his pool is empty. Initial tests showed very weak performance of this strategy. An improved version of the pool pipeline strategy is the ordered pipeline strategy, as the name suggests it attempts to order the calculations. The sewer system is ordered in a topological manner. Topological sorting of the sewer system preserves a correct sequential execution order of the sewer system. This sequential ordering is fed into thread safe queues maintaiend between threads calculating different timesteps.

The different testing systems were used to compare the runtime and speedup behaviour of the three strategies. The flow parallel strategy exposed good results for all testing systems except for the sequential system. The pool pipeline strategy showed poor performance in initial tests. On all tests the pool pipeline strategy showed severe performance losses when more threads were used. The ordered pipeline strategy exposed very good results on all range of input sewer systems including the sequential ones. It showed the best performance of all three strategies without any losses in speed.

Parallel computing in urban drainage is possible and shows good results if the used parallel strategy does not employ any kind of nondeterminism. The ordered pipeline strategy could be used without any worrying of a slow down on some obscure sewer system. Due to its time pipelined fashion it allows to calculate a lot more in parallel than the hardware parallelism that was available at the time of the writing. It should also be possible to scale to the many-core CPUs if the input sewer system is large enough.

# Appendices



# Appendix A.

## CityDrain3 Manuals

### A.1. Users Manual

This manual describes the usage of CityDrain3. It first describes common terms and concepts which are also interesting for extenders of CityDrain3. It also guides through defining new XML model files, and how to start CityDrain3 to run the newly created models in a simulation. At the end it shows the usage of *plugindoc* and how one can extract the needed informations from this application. The python module *cd3modelgen.py* is described and how it is used the effort to create a model XML file.

#### A.1.1. Terms and Concepts

##### Node

A node is the central processing unit. Each node has its own algorithm implemented. A node is a one to one mapping of the natural urban drainage elements. Currently there are several nodes implemented:

**Null** does nothing (demo usages)

**ConstSource** source node which emits a constant static flow.

**FileOut** writes the values of a flow per time step into a specified file.

**Mixer** mixes  $n$  flows.

**Splitter** splits one flow into two by a factor.

**CSO** combined sewer overflow (simple implementation).

**Catchment** combined sewer catchment.

**Sewer** simple sewer implementing muskingum flow routing.

**RainRead** read ixr rain files.

Its possible to extend CityDrain3 by providing new node implementations. Section A.2.3 and A.2.3 shows how this can be done.

In CityDrain3 a node is a pretty abstract construct. It gets input in shape of a flow. The implemented algorithm takes this inputs and produces a output also in shape of a flow. A node can have internal states which are accumulated by the run of the simulation. The nodes algorithm can also be parametrized. All these concepts are now described in detail.

**Parameters** are static values which are loaded before the node is initialized. The values are set in the XML model file. Depending on these parameters a node can behave differently, can have different kind of internal states or can have a different number of inputs or outputs. Therefore node parameters are set before the node initializes itself and are static throughout the simulation. Parameters can have default values which are set in the nodes constructor. Parameters have names, types and default values.

Examples of parameters are:

- in *FileOut* a parameter specifies where to write the results.
- in the *Mixer* node a parameter specifies how many Flows will be mixed, which influences the number inputs.
- the *Sewer* node allows to specify how many sub reaches the sewer has and the muskingum parameters like  $X$  and  $K$ .
- the *RainRead* gets the path of the rain file by a parameter.
- the catchment has lots of parameters defining the area of the catchment or the dry weather flow, etc..

**Internal States** are used as intermediate results of the nodes calculations. They must be specified by the node in order to get written out if a simulation needs to save the states. Section A.1.2 shows how to save the states of a simulation. States have names, types and values. Examples of internal states are:

- The fill level of a CSO.
- The volumes of the sub reaches stored in the a sewer.
- The fill level of the loss basin in a catchment.

some node don't have internal states like the Splitter, Mixer and ConstSource node. These nodes are therefore referential transparent and the output of their



calculation only depends in inputs.

**Inputs and Outputs** are named ports. A port is essentially a named flow. Inputs and output ports need to be specified by the node. They can be dynamically created, e.g. in the *Mixer* node. A *Mixer* node can have an unlimited number of input ports and has one output port. A flow travels from the output port of a node (i.e. the source) to the input port of another node (i.e. the sink).

## Connection

Connections are used to describe the flow exchange of nodes. A Flow travels from one node to the other by the sources output port into the sink input port.

A connection is a collection of references:

1. A reference to the source node,
2. a reference to the sink node,
3. the name of the source nodes port and
4. the name of the sink nodes port.

Connections can behave differently depending on the used simulation type. For example a node can buffer flows if nodes implement variable time steps. More information on the different Simulation types can be found in Section A.1.1.

## Flow

The flow is the manifestation of the data exchange between the nodes. It represents an exchange of water including concentrations between nodes per time step. A flow is a list of values. Each value (floating point) has a name and a corresponding unit (e.g. the flow of water named *flow* with the unit  $\frac{l}{\delta t}$  )

At the moment the following three units are allowed to be part of a flow:

1. *flow*: Is the amount of water travelling through the sewer system with the unit  $\frac{l}{\delta t}$ .
2. *rain*: Is the amount of rain in  $\frac{mm}{\delta t}$ .

3. *concentration*: Is the amount of pollutant travelling in the water in  $\frac{g}{m^3}$ .

Internally its a C++ class which allows to enumerate the concentrations and get/ set the values of the items. All values have names, an a value of double precision floating point type.

## Simulation

A simulation is a class which controls the run of the simulation. Because various scenarios have been identified in where different simulations may be used, simulations can be loaded dynamically by the plug-in system.

Various Simulation types are already implemented:

- VarDt
- Standard
- FlowParallel
- OrderedPipeline

The last two are parallel implementations which are able to use the extra power of multi-core processors.

## Controller

A controller is a function which is called every time step. It allows to override/extend the behaviour of the chosen simulation class. There are two controllers implemented: The first can be activated to write the internal states of the nodes at every time step, the second one is used to write the progress of the simulation to the terminal.

Controllers may either be written as C++ code (described in Section A.2.3) or it can be written as a collection of ECMAScript functions.

## Model

A model is the collection of all nodes and their parameters, connections and the additional information needed to run a single simulation instance. A model describes the structural properties of a sewer system under research.

A model is loaded into CityDrain3 by specifying a XML model file at the command line prompt. After the model is loaded the simulation is, run and depending on the model the results are written to dedicated files on the hard disk, ready for further processing and analyses.

Section A.1.3 gives a step by step guide how to write such XML model files, ready to be used and consumed by CityDrain3.

### A.1.2. Starting CityDrain3

CityDrain3 is a terminal application, this means that you need to be comfortable starting applications in the terminal environment. If you haven't done that, its not a big deal.

*cd3* used several libraries. These libraries must be available for the application. On Windows all the libraries are in the *win32* sub directory. Starting the application from here should work.

On Linux the *Qt* libraries should be installed globally in your system <sup>1</sup>. Setting the right paths to find the non-Qt libraries is done by sourcing the *source-me.sh* file:

```
$ source source-me.sh
```

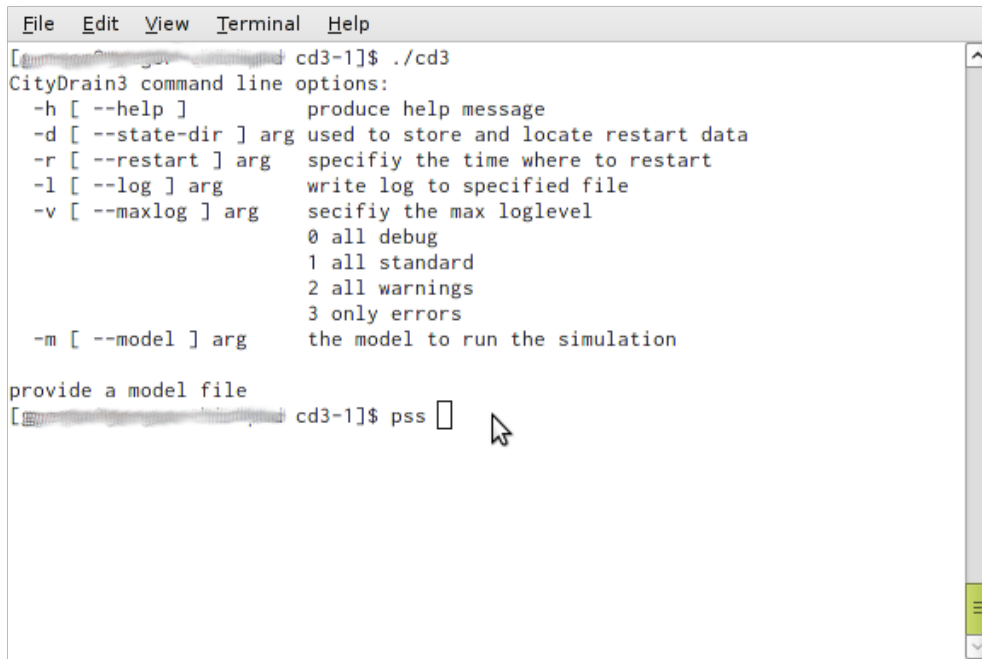
CityDrain3s executable is named *cd3*. If it gets started without a parameter it shows a simple help message displayed in Figure A.1. A user must at least provide a path to the model file which should be calculated. CityDrain3 has several examples which show how to write models but also allows to experiment with the binary. Section A.1.3 shows how to write models.

CityDrain3 has several parameters which allows the application to behave differently. these are:

- h** Shows the help message with a rough overview of the parameters.
- m** This is the flag which allows to provide the model path. This parameters can be omitted if its placed on the end of the command line as the last parameter.
- d** Following this flag the user can provide a directory name (must not exist) where CityDrain3 stores the internal states of the nodes per time steps. (described in Section A.1.1). This behaviour can also be implemented by

---

<sup>1</sup>They are in almost all cases

A terminal window titled "Terminal" with a menu bar containing "File", "Edit", "View", "Terminal", and "Help". The terminal shows the command `./cd3` being executed in a directory named `cd3-1`. The output displays the "CityDrain3 command line options:" followed by a list of flags and their descriptions: `-h [ --help ]` (produce help message), `-d [ --state-dir ] arg` (used to store and locate restart data), `-r [ --restart ] arg` (specify the time where to restart), `-l [ --log ] arg` (write log to specified file), `-v [ --maxlog ] arg` (specify the max loglevel, with sub-options: 0 all debug, 1 all standard, 2 all warnings, 3 only errors), and `-m [ --model ] arg` (the model to run the simulation). Below the options, it says "provide a model file" and shows the command `pss` being entered at the prompt. A mouse cursor is visible over the terminal window.

```
File Edit View Terminal Help
[cd3-1]$ ./cd3
CityDrain3 command line options:
-h [ --help ]          produce help message
-d [ --state-dir ] arg used to store and locate restart data
-r [ --restart ] arg   specify the time where to restart
-l [ --log ] arg       write log to specified file
-v [ --maxlog ] arg   specify the max loglevel
                      0 all debug
                      1 all standard
                      2 all warnings
                      3 only errors
-m [ --model ] arg     the model to run the simulation

provide a model file
[cd3-1]$ pss
```

Figure A.1.: Starting CityDrain3 without parameters.

using a hand crafted controller, see Section A.1.1 and A.2.3. For example one can dump the states only at predefined time steps.

- r This flag can only be used in combination with `-d`. A user specifies a time step on where the simulation restarts. The time step must be available in the specified states directory.
- l This flag is used if the log needs to be saved into a file. The name following the `-l` flag must be a path to a non existing file.
- v Is used to specify the maximum loglevel. Log message below the specified level are ignored. The levels range from 0 - all messages to 3 - silent mode.

For example if one wants to store all the node states in a `states` sub directory and run a model called `thesis-model.xml`, `cd3` is started by:

```
$ ./cd3 -d states thesis-model.xml
```

Which would be the same as:

```
$ ./cd3 -m thesis-model.xml -d states
```

restarting at time step 100000 is done by:

```
$ ./cd3 -m thesis-model.xml -d states -r 100000
```

### A.1.3. Writing a Model

As described earlier a model is a XML file containing all the information for running a simulation. A XML model file is the least parameter CityDrain3 needs in order to run the simulation. This sections gives a very short introduction into the XML format in general and after that it shows the structure of the CityDrain3 XML file format.

XML is a simple textual file format. It allows to structure complex information into a format which can be consumed and interpreted by machines and human beings. Although its said to be easier to read by machines.

Elements are the containing pieces in a XML file. A element has a name which comes directly after the opening of the tag. Listing A.1 shows a real basic XML example. The containing elements of the document are:

1. book, which is the root element,
2. person,
3. surname,
4. familyname and
5. married.

*book* is the root element. A root element must be single on its level, this means after the closing `</book>` there is nothing more allowed. *person* and *married* have attributes. attributes are named values separated by an equal sign which are placed into the start tag of an element. An element can contain other elements of *CDATA* which stands for character data (i.e. text). *surname* and *familyname* are such elements only containing text.

```
<book>
  <person sid="3227">
    <surname>Gregor</surname>
    <familyname>Burger</familyname>
    <married state="false" />
  </person>
</book>
```

Listing A.1: simple xml file

A XML document is said to be well formed if it follows the XML format, informally described above. Well formedness alone is not enough, because *sure-name* can also be chosen to be *sname*. A designer of an XML format has to choose the structure of the XML document which should be machine readable. In order to fix the structure and names, XML allows to specify a document type description (DTD). A DTD is a contract which specifies which elements and attributes are allowed and in which arrangement. A XML document is said to be valid if it full fills the contract setup by a DTD.

To check an XML file for well formdness and validate it against a DTD several tools exist. One is *xmllint*, it comes with almost all Linux distributions.

```
$ xmllint --path dtd --valid --noout models/test-sewer.xml
```

Listing A.2 displays a simple model on which the following descriptions to write your own model are based.

```
<?xml version="1.0"?>
<!DOCTYPE citydrain SYSTEM "../dtd/model.dtd">

<citydrain version="1.0">

  <pluginpath path="nodes" />

  <simulation class="DefaultSimulation">
    <time start="0" stop="7200" dt="300" />
  </simulation>

  <model>
    <nodelist>
      <node id="constsource" class="ConstSource">
        <parameter name="const_flow" kind="complex" type="Flow">
          <flow>
            <unit name="Q" kind="Flow" value="234.0" />
            <unit name="C0" kind="Concentration" value="0.1" />
            <unit name="C1" kind="Concentration" value="0.2" />
          </flow>
        </parameter>
      </node>

      <node id="fileout" class="FileOut">
        <parameter name="out_file_name" type="string"
          value="tmp/sewerout.txt" />
      </node>

      <node id="musk1" class="Sewer" />
    </nodelist>
  </model>
</citydrain>
```

```

<connectionlist>

  <connection id="con1">
    <source node="constsource" port="out" />
    <sink node="musk1" port="in" />
  </connection>

  <connection id="con2">
    <source node="musk1" port="out" />
    <sink node="fileout" port="in" />
  </connection>

</connectionlist>
</model>
</citydrain>

```

Listing A.2: a simple model

The first two lines describe some general aspect about the XML file. Its commonly known as the XML header. The second line describes the name and path of the file containing the DTD. In the CityDrain3 source code the DTD is contained in the path *cd3-1/dtd*.

```

<citydrain version="1.0">

  <pluginpath path="nodes" />

  <simulation class="DefaultSimulation">
    <time start="0" stop="7200" dt="300" />
  </simulation>

```

Listing A.3: xml header of model file

The root node of a model XML document is *citydrain*. Its only attribute is a version. This version id may be used in upcoming releases to identify changes in model file. Zero or more *pluginpath* nodes are the first child of the *citydrain* root node. They provide a single attribute named *path* on where CityDrain3 can find plug ins. These plug ins contain all the implemented node and simulation types. The path to the *dll* or shared object *so* must be specified either in portal format like in the example or as an fully specified path. The portable notation adds *.dll* on windows an *libname.so* on linux. Next comes the simulation tag. The single attribute *class* states the name of the simulation class to load and run the model. The *time* element defines the simulation time. The simulation starts at the time *start* runs each time step with a length of *dt* and stops if the simulation time reaches the value of *stop*.

```

<node id="constsource" class="ConstSource">
  <parameter name="const_flow" kind="complex" type="Flow">

```

```

    <flow>
      <unit name="Q" kind="Flow" value="234.0" />
      <unit name="C0" kind="Concentration" value="0.1" />
      <unit name="C1" kind="Concentration" value="0.2" />
    </flow>
  </parameter>
</node>

<node id="fileout" class="FileOut">
  <parameter name="out_file_name" type="string"
    value="tmp/sewerout.txt" />
</node>

<node id="musk1" class="Sewer" />

```

Listing A.4: the node list

The node *odelist* contains all the nodes and their parameters. Every node has a unique id. This id can be found in the *node* elements start tag as the *id* attribute. This id is important for connecting nodes. Two nodes forming a connection are referenced by these ids. *class* is the same as in the simulation tag it specifies which C++ class must be loaded. Example classes are: *CSO*, *Sewer*, *Catchment* etc.. *node* accepts *parameter* elements as childrens. A parameter is per default a *simple* parameter if one wants to specify basic data types like *int*, *double*, *bool* or *string*. Specifying a *flow* like in the *ConstSource* node requires to state *complex* as the *kind* value. In case of a complex parameter the *value* attribute is not necessary. Parameters must not be stated if the default value are satisfied by the needs of the model. An example is the *Sewer* node with *id=musk1*.

```

<connection id="con1">
  <source node="constsource" port="out" />
  <sink node="musk1" port="in" />
</connection>

<connection id="con2">
  <source node="musk1" port="out" />
  <sink node="fileout" port="in" />
</connection>

```

Listing A.5: the connection list

The last part of the document is the list of connections in the *connectionlist*. A connection has, similar to the node, a unique id. The first child of *connection* is *source* it references the *id* and a port of connections source. *sink* behaves equally, it just describes the sink part of the connection.



**JavaScript Nodes** C++ is a compiled language designed for high performance and not for ease of use. Because of this CityDrain allows to implement nodes in JavaScript programming language. JavaScript is interpreted and therefore there is no need to compile it before running it.

A JavaScript node is somehow special. The first thing different is that you need to specify a *script* attribute and use the node class *QSWNode*. The second thing is you need to write a script file which replaces the *init* and *f* functions of a standard C++ node. Beside that a script node behaves exactly like any other node, has states and accepts predefined parameters. Section A.2.3 shows how to write a script that can be specified to be used by *QSWNode*.

```
<node id="fileout2" class="FileOut">
  <parameter name="out_file_name" type="string"
    value="tmp/jssplitter2.txt" />
</node>
```

Listing A.6: using a JavaScript node

**Controller** A controller is a JavaScript or C++ code which is called before and after a time step is completed. A controller can:

- set internal states of nodes,
- write the states to a file,
- load the states from a file and
- can even stop the whole simulation if needed.

Listing A.7 shows how to specify such a JavaScript controller. How to write a controller script can be found in the programmers manual in section A.2.3.

```
<simulation class="DefaultSimulation">
  <time start="0" stop="72000" dt="300" />
</simulation>

<controller script="scripts/controller.js" />

<model>
```

Listing A.7: specifying a controller script

**Cycles** are handled by adding a *cycle\_break* attribute to a connection where its needed to break up a cycle. The *cycle\_break* connection is typically the one

where the the flow enters the cycle back into the beginning. Listing A.8 shows how one specifies such a cycle break connection.

```

<model>
  <nodelist>

    <node id="ConstSource-0" class="ConstSource">
      <parameter name="const_flow" kind="complex" type="Flow">
        <flow>
          <unit name="Q" kind="Flow" value="100.000000" />
          <unit name="C2" kind="Concentration" value="3.000000" />
          <unit name="C1" kind="Concentration" value="5.000000" />
          <unit name="C0" kind="Concentration" value="1.000000" />
        </flow>
      </parameter>
    </node>

    <node id="Mixer-0" class="Mixer">
      <parameter name="num_inputs" type="int" value="2" />
    </node>

    <node id="Splitter-0" class="Splitter">
      <parameter name="ratio" type="double" value="0.5" />
    </node>

    <node id="FileOut-0" class="FileOut">
      <parameter name="out_file_name" type="string"
        value="tmp/genout.txt" />
    </node>

    <node id="Sewer-0" class="Sewer">
      <parameter name="N" type="int" value="11" />
      <parameter name="K" type="int" value="300" />
      <parameter name="X" type="double" value="0.1" />
    </node>

    <node id="Sewer-1" class="Sewer">
      <parameter name="N" type="int" value="11" />
      <parameter name="K" type="int" value="300" />
      <parameter name="X" type="double" value="0.1" />
    </node>

  </nodelist>
  <connectionlist>

```

```

<connection id="con-ConstSource-0-Mixer-0">
  <source node="ConstSource-0" port="out" />
  <sink node="Mixer-0" port="inputs [0]" />
</connection>

<connection id="con-Mixer-0-Splitter-0">
  <source node="Mixer-0" port="out" />
  <sink node="Splitter-0" port="in" />
</connection>

<connection id="con-Splitter-0-Sewer-0">
  <source node="Splitter-0" port="out1" />
  <sink node="Sewer-0" port="in" />
</connection>

<connection id="con-Splitter-0-Sewer-1">
  <source node="Splitter-0" port="out2" />
  <sink node="Sewer-1" port="in" />
</connection>

<connection id="con-Sewer-1-Mixer-0" cycle_break="true">
  <source node="Sewer-1" port="out" />
  <sink node="Mixer-0" port="inputs [1]" />
</connection>

<connection id="con-Sewer-0-FileOut-0">
  <source node="Sewer-0" port="out" />
  <sink node="FileOut-0" port="in" />
</connection>

</connectionlist>

```

Listing A.8: specifying a `cycle_break` attribute in a connection

#### A.1.4. Using `cd3modelgen.py`

`cd3modelgen.py` is a python module which allows to easy create XML module files programmatically. This approach has several advantages:

- converter scripts may be written which read other file formats and convert them into the CityDrain3 model file,

- models maybe generated using stochastic methods generating artificial systems,
- the tedious work of writing XML files is covered by just a few python method calls,
- python scripts are more readable than XML files,
- etc..

The python module allows to define in an object oriented manner the structure of the file. It is completely independent of the CityDrain3 C++ source code. It just prints the XML file based in its input. No checking or other sanitizing of the input is done.

Python is a scripted programming language. It is famous for its ease of programming and productivity. The language is easy to learn and lots of information regarding python is on the Internet.

Listing A.9 shows a simple script which generates a model with a *ConstSource*, *Sewer* and a *FileOut* to show the results. Each C++ node has a python pendant with the same name. <sup>2</sup> Connections are handled by the *Connection* class. The parameters of the Constructor are:

1. the source node
2. the sink node
3. the source port (optional, default "out")
4. the sink port (optional, default "in")

*Simulation* is the class which collects the nodes into the *nodes* array and the connections into the *cons* array. After all nodes and connections have been added to the simulation the *render* method prints the XML to the terminal. The output can then simply written in a file by redirecting the output of the script.

```
$ python gentest.py > gentest.xml
```

```
from cd3modelgen import *
constsource=ConstSource()
sewer=Sewer()
```

<sup>2</sup>As the *cd3modelgen.py* does not use the C++ code it must be extended if new nodes are added later on.

```

fileout=FileOut("tmp/testout.txt")

con1=Connection(constsource, sewer)
con2=Connection(sewer, fileout)

sim=Simulation()
sim.nodes += [constsource, sewer, fileout]
sim.cons += [con1, con2]

sim.render()

```

Listing A.9: simple python script

Listing A.10 shows the output of the script in Listing A.9.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE citydrain SYSTEM "file:../dtd/model.dtd">

<citydrain version="1.0">

  <pluginpath path="./libnodes.so" />
  <simulation class="DefaultSimulation">
    <time start="0" stop="7200" dt="300" />
  </simulation>
  <model>
    <nodelist>

      <node id="ConstSource-0" class="ConstSource">
        <parameter name="const_flow" kind="complex"
          type="Flow">
          <flow>
            <unit name="Q" kind="Flow" value="100.000000" />
            <unit name="C2" kind="Concentration"
              value="3.000000" />
            <unit name="C1" kind="Concentration"
              value="5.000000" />
            <unit name="C0" kind="Concentration"
              value="1.000000" />
          </flow>
        </parameter>
      </node>

      <node id="Sewer-0" class="Sewer">
        <parameter name="N" type="int" value="11" />

```

```

    <parameter name="K" type="int" value="300" />
    <parameter name="X" type="double" value="0.1" />
</node>

<node id="FileOut-0" class="FileOut">
  <parameter name="out_file_name" type="string"
    value="tmp/testout.txt" />
</node>

</nodelist>
<connectionlist>

  <connection id="con-ConstSource-0-Sewer-0">
    <source node="ConstSource-0" port="out" />
    <sink node="Sewer-0" port="in" />
  </connection>

  <connection id="con-Sewer-0-FileOut-0">
    <source node="Sewer-0" port="out" />
    <sink node="FileOut-0" port="in" />
  </connection>

</connectionlist>
</model>
</citydrain>

```

Listing A.10: output of simple python script

A full documentation of the python module can be generated by typing:

```
$ pydoc cd3modelgen
```

### A.1.5. *plugindoc* Application

Due to the high abstraction level in CityDrain3s design its possible to generate a documentation for the nodes which a plug-in contains.

*plugindoc* is such an application which uses the CityDrain3 infrastructure to generate the list of nodes, their parameter, their states and the input and output ports. The applications parameter is the portable library name of the plug ins which should be inspected:

```
$ ./plugindoc/plugindoc nodes
```

The output of the documentation application with the default node plug-in is shown in Listing A.11.

```
Nodes:
CSO:
  parameter: Q_Max
  parameter: V_Max
  state: stored_volume
  in_port: in
  out_port: out
  out_port: overflow

CatchmentCSS:
  parameter: n_rain_conc
  parameter: K
  parameter: N
  parameter: permanent_loss
  parameter: X
  parameter: run_off_coeff
  parameter: initial_loss
  parameter: area
  state: V[1]
  state: V[0]
  state: V[2]
  state: loss_basin
  in_port: rain_in
  in_port: parasite_in
  in_port: q_upstream
  in_port: dwf_in
  out_port: out

ConstSource:
  parameter: const_flow
  out_port: out

FileOut:
  parameter: out_file_name
  in_port: in

Mixer:
  parameter: num_inputs
  in_port: inputs[0]
  in_port: inputs[1]
  out_port: out

Null:
  out_port: out
```

```

RainRead:
  parameter: file_name
  parameter: base_date
  state: rain
  out_port: out

Sewer:
  parameter: K
  parameter: N
  parameter: X
  state: V[6]
  state: V[1]
  state: V[0]
  state: V[5]
  state: V[4]
  state: V[8]
  state: V[3]
  state: V[7]
  state: V[9]
  state: V[2]
  state: V[10]
  in_port: in
  out_port: out

Splitter:
  parameter: ratio
  in_port: in
  out_port: out2
  out_port: out1

TestNode:
  state: test
  state: int_value
  state: double_value
  state: string_value
  in_port: in
  out_port: out

Simulations:
  DefaultSimulation
  OrderedPipeSimulation
  ParallelSimulation
  PipelinedSimulation
  VarDTSimulation

```

Listing A.11: Node documentation



## A.2. Programmers Manual

This section describes the internals of CityDrain3 to allow a user to extend and implement further features into CityDrain3. The first chapter deals with compiling CityDrain3. The second chapter deals with the design and implementation details which are needed to extend CityDrain3. Extending CityDrain3 is covered in the last chapters.

### A.2.1. Compiling CityDrain3

#### Tested compilers

CityDrain was mainly developed on Linux, which always ship with high quality and up-to-date compilers. The compilers used on Linux were gcc and Intel, which both work very well. Compiler that are known to not work are Visual Studio 6 and gcc 3.4, both in Windows. Unfortunately the compiler that is shipped with the Qt SDK is gcc 3.4 that does not work.

Compiler	Platform	status
gcc 3.4	Windows	fails
gcc 4.4.0	Windows	works
Intel C++ 11	Windows	works
Visual Studio 2008 <sup>3</sup>	Windows	works
gcc 3.4	Linux	works
gcc 4.4	Linux	works
Intel C++ 11	Linux	wokrs

#### Toolset

Several steps (on Windows) are needed in order to get CityDrain3 compiled:

1. Install a git client. Windows binaries are at <http://code.google.com/p/msysgit/>
2. Install the Qt libraries. For windows users just download the Qt SDK from <http://www.qtsoftware.com>.
3. Install an up to date MinGW GCC compiler (found at: <http://www.tdragon.net/recentgcc/>).

4. Get the source code by cloning the git repository. Open the Git Bash and type:

```
$ git clone git://138.232.95.43/cd3-1.git
```

5. Download and install CMake from <http://www.cmake.org/>
6. Start the CMake Gui from within the “MinGW Command Prompt”, found in the MinGW folder of the start menu after installation.
7. Point the cmake in the gui to the source of cd3.
8. Hit the “configure” button twice, then the “generate” button.
9. switch into the “MinGW Command Prompt” windows and start *mingw32-make* in the cmake build directory (typically the source directory of cd3).

You now have the up to date CityDrain3 source code, hopefully error free compiled. If you want to update to the latest commit use these commands in the “git bash” of the *cd3* top level directory:

```
$ git pull origin master
```

The preferred compiler on windows is either Visual Studio 2008, in this case the Visual Studio Qt integration tools are very handy, or a MinGW gcc version 4.4 which can be downloaded from <http://www.tdragon.net/recentgcc/>. If a different compiler than the one included in Qt SDK is used the path to the compiler must be specified on the Project view in the “Build Environment” settings. Point the path variable as first entry to your installed compiler bin directory, as shown in Figure A.2.

CityDrain3 uses Qt for handling JavaScript integration and XML file reading. Qt 4.5 got further feature enhancements in the QtScript module on which CityDrain3 heavily depends, therefore Qt 4.5 is needed.



The minimum version of Qt is 4.5 and MinGW gcc 3.4 on windows shipped with Qt SDK does not work.

## A.2.2. Design Overview

### Directory Layout

The first thing a new developer of CityDrain3 needs to know is where to find stuff in the directory tree of CityDrain3. This section describes the directories

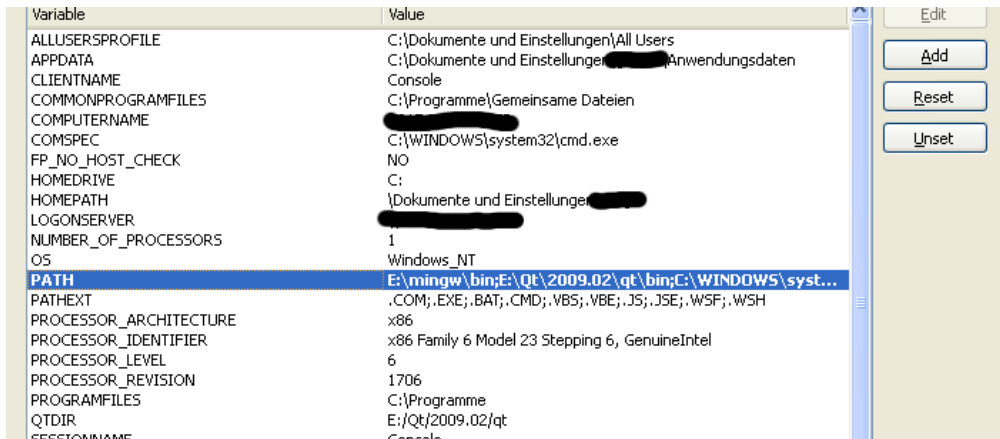


Figure A.2.: Setting the Path for the MinGW compiler

and their contents used in CityDrain3.

The root of the directory *cd3-1* contains the following directories:

**3rdparty** contains all 3rdparty libraries which are shipped with CityDrain3.

Currently only the boost library is contained here. Qt must be installed globally.

**src/app** contains the main of the *cd3* executable.

**bench** contains scripts to run benchmarks with the different simulation classes.

The scripts also create charts to show the results of the benchmarks graphically.

**src/cd3core** is the most important directory. It contains the framework of CityDrain3. It contains the code and interfaces which are at the heart of the application.

**doc** contains a *doxygen* script for generating a navigable reference documentation of the C++ code.

**data/dtd** contains the *model.dtd* file used to check the XML model files for validity and well formedness.

**data/models** is the directory where example and test models are stored. The folder also contains the python scripts used by *cd3modelgen.py*.

**src/nodes** is the home of all implemented nodes and simulation classes source code.

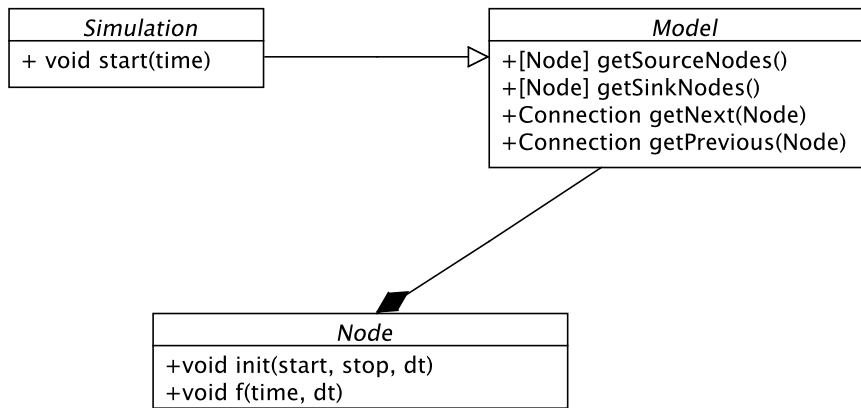


Figure A.3.: Overview of the essential interfaces

**src/plugindoc** contains the source code of the *plugindoc* application.

**data/scripts** contains all the JavaScript scripts.

**win32** contains the executable and libraries of a *Windows* build.

**tmp** contains compiled intermediate files and results of the demo models are stored here typically.

## Class Overview

Figure A.3 shows an overview of the essential interface classes and their interaction. The *Model* (see Figure A.4) class is a collection of *Nodes* (see Figure A.6) and the *Connections* between nodes. The *Simulation* (see Figure A.5) interface is used by an application to start the simulation and add controllers if needed. The *Simulation* class used the *Model* to navigate through the Directed Graph (DG) that represents the model under research. Classes which extend from *Node* are then responsible for calculating urban drainage algorithms based on the inputs. They produce side effects called states and output their results on the output ports.

The *Model* class has the notion of source and sink nodes. The collection of source nodes are all node which don't have input ports. Equivalently, sink nodes are all nodes which have no source port. By using the *getNext* method one can query all connection start by the specified node. *getSinkNodes* and *getNext* allows to iterate the whole graph from sinks to sources. The other way around is done by using *getSinkNodes* and *getPrevious*.

<i>Model</i>
+addNode(name, Node) +addConnection(Node, out_port, Node, in_port) +[Node] getSourceNodes() +[Node] getSinkNodes() +Connection getNext(Node) +Connection getPrevious(Node)

Figure A.4.: The *Model* class

<i>Simulation</i>
+ void start(time) + setModel(Model) + setSimulationParameters(SimParams) + Signal progress()

Figure A.5.: The *Simulation* class

<i>Node</i>
+void init(start, stop, dt) +void f(time, dt) +void addParameter(name) +addState() +getState(name) +addInPort() +addOutPort()
+[(name, value)] states +[(name, Flow)] in_ports +[(name, Flow)] out_ports

Figure A.6.: The *Node* class

The class diagrams shown in this Section are heavily simplified in order to focus on the important stuff.

### A.2.3. Extending CityDrain3

CityDrain3 was designed with a focus on extensibility. An extension writer can choose between an easy to deploy JavaScript extension mechanism or write native plug ins in C++. The latter involves more knowledge of the surrounding tools and the C++ language, but allows to control and inspect all most all aspects of a simulation. Although the the JavaScript extension are easier to develop they are somehow limited in their scope and application. For example one can not inspect the Graph representing the connections and nodes of a sewer system. The big advantage of JavaScript is that it is interpreted and allows to have short turnover times. This fits perfectly for a rapid prototyping approach. Once the algorithms are settled it is easy to port the JavaScript code to C++ because the names and conventions are almost the same.

This section describes in a step by step manner how one can adapt CityDrain3 to the modelling needs of the simulation with both C++ and JavaScript methods.

#### JavaScript notes

JavaScript is an interpreted, prototype based dynamic, weakly typed programming language with first class functions. It was invented to allow dynamic contents of web pages. It is known as the scripting language of the web. Due to the Web 2.0 wave dynamic web applications and therefore JavaScript got hyped a lot and drove inventions in the language.

JavaScript has objects which are associative arrays: that means that the statements `obj.x = 10` and `obj['x'] = 10` are equivalent. Every script has a global object associated. This is the name space where all variables and functions get defined. This means that there does not exist an explicit *main* function, the execution begins at the glob name space. dynamic weakly typing means that types are bound to values not to variables. A variable `x` can at one line point to a value of an *int* and at the next line to a *string*. Support for first class functions means that a function is just another value, although callable, which can be passed around. Prototype based objects are different than class based objects in that they completely miss classes. Objects are instantiated and functions and properties are added. If one wants to inherit in a prototype based

language he just clones the parent, the parent is then the prototype of the child object.

In CityDrain3 most features are ignored. There exist only a few objects, namely *Flow*, *Node* and *CalculationUnit*<sup>4</sup>. Every script is compiled first and all the global variables defined here can be used later if the callback functions are invoked. A callback function can be *init*, *f*, *controllBefore* or *controllAfter* depending on what aspect of CityDrain3 is extended.

## Implementing a JavaScript Node

Implementing a node in CityDrain3 using the JavaScript language involves basically two things. Define a node in the model with the class *QSWNode* and pass it a path to a JavaScript script file. The second thing to do is to implement the script file. Listing A.12 shows the minimum code needed for a script file. A script must implement a *init* function which is called once, and a *f* function which is called at least once per time step.

```
//define parameters, in/out ports, states and globals here.
function init(start, stop, dt) {
    //init node based on parameters
}

function f(time, dt) {
    //calculate outputs based on inputs
    //from ports, states and time
    return dt
}
```

Listing A.12: JavaScript node template

Before these functions are called the script gets loaded and interpreted by the JavaScript engine. After this step all globals (defined before *init*) and the functions in the script are known. Globals have a lifetime from the first interpretation of the script until the simulation is destroyed (i.e. the simulation is stopped). This is different to the behaviour in the older CITY DRAIN implementations where states and globals were passed around via function arguments.

```
x=0
function init(start, stop, dt) {
    x = dt
}
```

<sup>4</sup>A calculation unit is just a triple of a name, a unit and a description (e.g. (flow,  $m^2/s$ , the flow of water)).

```

function f(time, dt) {
  x = x + (dt / 10)
  return dt
}

```

Listing A.13: global state example

The code in Listing A.13 shows this effect with a global variable  $x$ .  $x$  gets initialized with zero, after `init` it has the value of  $dt$ . At every time step the value of  $x$  is incremented by  $\frac{dt}{10}$ . If the value of  $x$  influences the outputs it must be added to the states of the node. If for example the previously defined  $x$  is added to the states one needs to call the `addState()` function as shown in Listing A.14. The parameter must be a string with the variable name one wants to add. Now  $x$  gets saved and loaded if a simulation is restarted.

```

x=0
addState('x')
function init(start, stop, dt) {
  x = dt
}

function f(time, dt) {
  x = x + (dt / 10)
  return dt
}

```

Listing A.14: adding a state

Adding input and output ports is done by calling `addInPort` or `addOutPort` respectively. The node in Listing A.15 leads a flow through the node without touching it. Such a behaviour could be used if one wants to protocol the flow reaching through the node.

```

flow = Flow()
function init(start, stop, dt) {
  addInPort('in', flow)
  addOutPort('out', flow)
}

function f(time, dt) {
  return dt
}

```

Listing A.15: adding ports



## Implementing a simple Splitter in JavaScript

The node we want to develop, as an example implementation of a working JavaScript node, is a simple Splitter. It splits two streams into halve, leaving the concentrations untouched.

```
var in_flow = Flow()
var out1 = Flow()
var out2 = Flow()

addInPort('in', in_flow)
addOutPort('out1', out1)
addOutPort('out2', out2)

function init(start, stop, dt) {
  //nothing here
  //note behaves pretty static
}

function f(time, dt) {
  out1.copy(in_flow)
  out2.copy(in_flow)
  qhalve = in_flow.getIth(flow, 0) / 2.0
  out1.setIth(flow, 0, qhalve)
  out2.setIth(flow, 0, qhalve)
  return dt
}
```

Listing A.16: A simple Splitter in JavaScript

As we can see in Listing A.16 we need one in-port (*in*) and two out port (*out1*, *out2*). *init* is empty because the node doesn't need a parameter nor does it use internal states. In the *f* function we have to copy the values of input to the two outputs. This must be done to leave the concentrations. Then we grab the water amount per seconds of the flow (i.e. the flow unit of the flow), halve it and set it in the outputs.

## Implementing a JavaScript Controller



Controllers may not behave as expected if a parallel simulation is used. This is because the timestep are started and stop not sequentially and not synchronously. This could mean that a timestep has already began before the last one has stopped.

Implementing a controller is similar to implementing node. The first thing to do is to point to a script file in the model using the *controller* element. The *controller* has a single attribute *path* which points to the script file. The element must be after *simulation* and before the *model* element.

```
<simulation class="DefaultSimulation">
  <time start="0" stop="72000" dt="300" />
</simulation>

<controller script="scripts/controller.js" />

<model>
```

Listing A.17: placement of the controller element

The script file must contain two functions with the names

- *controllAfter(time)* and
- *controllBefore(time)*

which are called either before each time step and after a time step. A sample script is depicted in Listing A.18. It shows basically what a controller can do.

```
//globals
function controllBefore(time) {
  //stops the simulation after time 5100
  if (time >= 5100) {
    stopSimulation()
  }
}

function controllAfter(time) {
  //gets the flow with name "V[0]" of node "musk1"
  v0 = getFlow("musk1", "V[0]")
  //and prints the concentration "C1"
  print(v0.getValue("C1"))
  //write state into the directory "/tmp/states"
  serialize("/tmp/states")
}
```

Listing A.18: JavaScript controller example

## JavaScript API Reference

The JavaScript Application Interface (API) includes four classes at the time of writing the manual:

1. Node
2. Controller
3. Flow
4. CalculationUnit

A *Node* class which is implicitly inherited if one implements a JavaScript node can call the following functions:

**addInPort(name, flow)** adds a flow as input port,  
**addOutPort(name, flow)** adds a flow as output port,  
**addParameter(name, value)** adds a parameter with initial value,  
**addState(name)** adds a state,  
**print(value)** prints a value.

All *names* must be of type string, values can have all basic types + *Flow*.

A *Controller* class which is implicitly inherited if one implements a JavaScript controller can call the following functions:

**stopSimulation()** stops a simulation,  
**serialize(dir)** writes internal states to dir,  
**deserialize(dir, time)** loads internal states from dir,  
**setInt(node, state, value)** sets state of node to a value,  
**getInt(node, state)** returns value of state from node,  
**setDouble(node, state, value)**  
**getDouble(node, state)**  
**setString(node, state, value)**  
**getString(node, state)**  
**setBool(node, state, value)**  
**getBool(node, state)**  
**setFlow(node, state, value)**  
**getFlow(node, state)**  
**print(value)** prints a value.

A value of type *Flow* can be instantiated using the *Flow* constructor function. The *Flow* class has the following methods:

**clear()** clear the flow to an initial state,  
**addUnit(name, unit, value)** adds a unit with name, unit and a value,  
**setValue(name, value)** sets a unit value,  
**getValue(name)** returns a unit value,  
**setIth(unit, i, value)** sets the value of the ith unit (when you don't know the name),  
**getIth(unit, i)** return the ith value of unit,  
**getNames()** get all added names,  
**getUnitNames(unit)** get all names with the unit.  
**copy(flow)** copy the values from a flow (assignment workaround)

*Node* and *Controller* have globally defined names for the units of *CalculationUnit* (i.e.. *flow*, *calculation* and *rain*).

Adding more API calls is as easy as adding them to the corresponding C++ classes (prefixed with *QSW*). If a method should be callable from JavaScript it must be added into the "*public Q\_SLOTS:*" section in order to be picked up by *QtScript* engine. More information on *QtScript* is available at <http://doc.qtsoftware.com/4.5/qtscript.html>.

## Implementing a C++ Controller

*Controllers* in C++ are handled differently than in *JavaScript*. They do not even follow the implement a subclass extension paradigm. Instead, a controller is just a receiver of a *boost::signal* which is comparable to a *Qt* signal. A receiver of such a signal is called a *slot*. It must be callable with a fixed set of parameters. A callable is either a function pointer or a standard C++ class with the "*()*" operator, the function call operator, overloaded. Listing A.19 shows how an implementation looks, acceptable as a *slot*.

```

struct PerStateHandler {
    PerStateHandler(const std::string dir) {
        state_dir = dir;
    }
    void operator()(ISimulation *s, int time) {
        (void) time;
        s->serialize(state_dir);
    }
private:
    std::string state_dir;
};

struct ProgressHandler {
    ProgressHandler(ISimulation *sim) {
        sp = sim->getSimulationParameters();
        lastp = 0;
        length = sp.stop - sp.start;
        count = 0;
        t = QTime::currentTime();
    }
    void operator()(ISimulation *s, int time) {
        (void) s;
        int newp = (time / length) * 100;
        count++;
        if (newp <= lastp)
            return;
    }
};
  
```

```

    QTime tmp_t(QTime::currentTime());
    Logger(Standard) << "Progress:" << newp << "%" << count << "dt:" << t.msecsTo(tmp_t);
    lastp = newp;
    count = 0;
    t = tmp_t;
}
double pfactor;
int lastp;
int count;
float length;
QTime t;
SimulationParameters sp;
};

```

Listing A.19: Two example slots

The simulation class provided two signals: *timestep\_after* and *time step\_before*. Signals and slots are connected by using the *connect* method of the signal. Listing A.20 shows how a *signal* and a *slot* are connected.

```

s->timestep_before.connect(ProgressHandler(s));

```

Listing A.20: connecting *signals* and *slots*

A slot receives the current *Simulation* instance and the current time step value. Using these two parameters a *Controller* can access all informations and control all aspects of the simulation.

The API a controller can use are actually the internals of CityDrain3. This allows greater flexibility but also has several risks of unknown unintentional side effects.

## Implementing a C++ Node

Listing A.21 and A.22 show the header and implementation of a node implemented in C++. The macro *CD3\_DECLARE\_NODE* does all the sub classing and class header declaration which are needed for the plug-in handling. The macro is just a convenient and less error prone way to declare a node, beside that everything is standard C++.

The node interface offers two virtual methods that a subclass must override. The first one is *init*, it is called once before the simulation starts. Before *init* is called values of parameters are loaded from the XML model file. The second method a new node must override is *f*. This is the method where the calculations of the node should happen. It is called once per time step and all input ports are brought up to date before *init*. After *f* has finished the output ports are picked up and forwarded to the next node.

```

#include <node.h>
#include <flow.h>

```

```

CD3_DECLARE_NODE( Null )
public:
    Null ();
    int f( int time , int dt );
private:
    Flow out ;
};

```

Listing A.21: Null node header

```

CD3_DECLARE_NODE_NAME( Null )

Null::Null() {
    addOutPort( "out" , &out );
    out = Flow::nullFlow ();
}

int Null::f( int time , int dt ) {
    (void) time ;
    return dt ;
}

```

Listing A.22: Null node implementation

```

#include <cd3globals.h>

extern "C" {
    void CD3_PUBLIC registerNodes( NodeRegistry *registry ) {
        registry ->addNodeFactory( new NodeFactory<ConstSource > ( ) );
        registry ->addNodeFactory( new NodeFactory<FileOut > ( ) );
        registry ->addNodeFactory( new NodeFactory<Mixer > ( ) );
        registry ->addNodeFactory( new NodeFactory<Sewer > ( ) );
        registry ->addNodeFactory( new NodeFactory<RainRead > ( ) );
        registry ->addNodeFactory( new NodeFactory<TestNode > ( ) );
        registry ->addNodeFactory( new NodeFactory<QSWNode, true > ( ) );
        registry ->addNodeFactory( new NodeFactory<CSO > ( ) );
        registry ->addNodeFactory( new NodeFactory<Splitter > ( ) );
        registry ->addNodeFactory( new NodeFactory<CatchmentCSS > ( ) );
    }
}

```

Listing A.23: Register nodes in a dynamic library

**Parameters** are set by calling the *addParameter* method defined in *Node*. The macro *ADD\_PARAMETERS* is again a convenient way of declaring a param-

eter and stating its name in one step. If a pointer parameter is used then `ADD_PARAMETERS_P` macro must be used.

```
in = new Flow();
```

Listing A.24: Registering parameters

Parameters are ordinary class members of the node. They are filled between the constructor call and the `init` call. Parameters were introduced to allow a more dynamic behaviour of the nodes (e.g. the *Mixer* node uses a parameter to specify how many inputs it must mix). The values of the parameters come from the XML model file, in which they are defined in the `node/parameter` element. The name of the parameters in the XML file and the C++ class are the same if the `ADD_PARAMETERS` macros are used.

**Ports** are registered using the `addInPort` and `addOutPort` methods. Again using the `ADD_PARAMETERS` or `ADD_PARAMETERS_P` macros so that the names can be picked up. Ports can be registered in the constructor or in the `init` method. Adding ports in the constructor is the preferred way, because then they show up in the `plugindoc` application. If ports depend on parameters then they belong into the `init` method. Listing A.25 shows both static and dynamic ports which depend on the `num_inputs` parameter.

```
Mixer::Mixer() {
    num_inputs = 2;
    addParameter(ADD_PARAMETERS(num_inputs));
    out = new Flow();
    addOutPort(ADD_PARAMETERS_P(out));
}
void Mixer::init(int start, int end, int dt) {
    (void) start;
    (void) end;
    (void) dt;
    for (int i = 0; i < num_inputs; i++) {
        Flow *tmp = new Flow();
        std::ostringstream name;
        name << "inputs[" << i << " ]";
        addInPort(name.str(), tmp);
        inputs.push_back(tmp);
    }
}
```

Listing A.25: Registering ports

**States** are pretty similar to ports and parameters. They are registered using the `addState` method. Again if static the registering calls belong in the constructor and if dynamic in the `init` method. In Listing A.26 the *Sewer* node registers the volumes stored in the sewer sub reaches. This state depends on the parameter  $N$  which states the number of sub reaches in the sewer.

### Step by Step

1. create `newnode.h` and `newnode.cpp`
2. declare node with `CD3_DECLARE_NODE` in header (see `nodes/null.h`)
3. declare name with `CD3_DECLARE_NODE_NAME` (see `nodes/null.cpp`)
4. implement `f` and `init`
5. register static ports, parameters and states in the constructor
6. register dynamic ports, parameters and states in `init`
7. create dll entry points like in `nodes/nodes.cpp` (see Listing A.23)
8. register node in dll entry points (see Listing A.23).

```

for (int i = 0; i < N; i++) {
    V.push_back(new Flow());
    addState(str(format("V[%1%]" ) % i), V[i]);
}

```

Listing A.26: Registering states

### Implementing a Simulation

Implementing a simulation is similar to a node. The difference is that one now inherits from *ISimulation* class instead of *Node*. A simulation implementation can choose to just implement the run method:

```

virtual int run(int time, int dt) = 0;

```

This one is called by the *ISimulation* class. If more is needed then the start method must be overridden:

```

virtual void start(int time);

```



In the second case more attention has to be paid, for example the signals of the controllers must be called. Registering of a simulation is shown in Listing A.27 and comparable to registering a node.

```
void CD3_PUBLIC registerTypes(TypeRegistry *registry) {
    (void) registry;
}

void CD3_PUBLIC registerSimulations(SimulationRegistry *registry) {
    registry->addSimulationFactory(new SimulationFactory<DefaultSimulation>());
    //registry->addSimulationFactory(new SimulationFactory<VarDTSimulation>());
}
```

Listing A.27: Register simulations in a dynamic library

## C++ API Reference

An API reference is not included in this document, but it can be generated by using the doxygen automatic documentation generation tool. It can be generated by calling *doxygen* in the *docs* sub directory.



## Submitted Papers

- [BFKR09] Gregor Burger, Stefan Fach, Heiko Kinzel, and Wolfgang Rauch. Parallel computing in integrated urban drainage simulations. In *8UDM & 2RWHM*, pages 366–367. IWA, September 2009.
- [BFKR10] Gregor Burger, Stefan Fach, Heiko Kinzel, and Wolfgang Rauch. Parallel computing in conceptual sewer simulations. *Water Science and Technology*, 2010. accepted.

# Parallel computing in conceptual sewer simulations

G. Burger<sup>\*)</sup>, S. Fach<sup>\*\*)</sup>, H. Kinzel<sup>\*\*\*)</sup> and W. Rauch<sup>\*\*)</sup>

*\*Institute of Computer Science, University of Innsbruck,  
Technikerstr. 21A, A-6020 Innsbruck, Austria (E-mail: gregor.burger@uibk.ac.at)*

*\*\* Unit of Environmental Engineering, University of Innsbruck, Technikerstr. 13,  
A-6020 Innsbruck, Austria (E-mail: stefan.fach@uibk.ac.at, wolfgang.rauch@uibk.ac.at)*

*\*\*\* hydro-IT GmbH, Technikerstr. 13, A-6020 Innsbruck, Austria  
(E-mail: Kinzel@hydro-it.com)*

## ABSTRACT

Integrated urban drainage modelling is used to analyze how existing urban drainage systems respond to particular conditions. Based on these integrated models, researchers and engineers are able to e.g. estimate long-term pollution effects, optimize the behaviour of a system by comparing impacts of different measures on the desired target value or get new insights on systems interactions. Although the use of simplified conceptual models reduces the computational time significantly, searching the enormous vector space that is given by comparing different measures or that the input parameters span, leads to the fact, that computational time is still a limiting factor. Due to the stagnation of single thread performance in computers and the rising number of cores one needs to adapt algorithms to the parallel nature of the new CPUs to fully utilize the available computing power. In this work a new developed software tool named CD3 for parallel computing in integrated urban drainage systems is introduced. From three investigated parallel strategies two showed promising results and one results in a speedup of up to 4.2 on an eight-way hyperthreaded quad core CPU and shows even for all investigated sewer systems significant run-time reductions.

## KEYWORDS

CD3, conceptual model, parallel strategy, integrated urban drainage modelling, multi-core, parallel computing

## INTRODUCTION

Whilst in the past the processors (CPUs) got significantly more powerful (and thus faster), nowadays it is not the single CPU that can be improved further but instead the number of processors is increased. Multi-core systems are the future of desktop computing. Single thread performance is stagnating, but the number of cores is rising (Kongetira et al., 2005). To fully utilize that available computing power one needs to adapt algorithms to the parallel nature of these new CPU-architectures. Therefore a framework for integrated urban drainage models named CITY DRAIN 3 (CD3) was developed that exploits these additional computational resources of multi-core CPU-architectures. CD3 is a further development of the existing non multi-core capable CITY DRAIN 2 (Achleitner et al., 2007). The objective was to program a framework capable to be extended for all kind of integrated urban drainage models, like waste water treatment plant processes and river quality models. In a first step CD3 is limited to conceptual sewer systems.

## The need of computational power for urban drainage simulations

Integrated urban drainage modelling (IUDM) combines the main subsystems of urban drainage systems (e.g. natural and urban catchments, sewers, receiving water bodies and waste water treatment plants) of the urban (waste) water cycle into one single model (Rauch *et al.*, 2002; Butler and Schütze, 2005). The main use of those models is to analyze how existing urban drainage

systems respond to particular conditions (Butler and Davies, 2004). Generally, deterministic models are used which always produce the same output for a specific set of input data. With these models engineers and scientists are able to fully reason about sewer system performance, discharge to the receiving water body and river water quality. Based on these integrated models, researchers are able to e.g. estimate long-term pollution effects (Rauch *et al.*, 1998), optimize the behaviour of a system by comparing impacts of different measures on the desired target value or get new insights on systems interactions.

IUDM models are often formulated in a simplified manner applying e.g. hydrological routing instead of hydrodynamic wave equations to calculate the waste water transport in the sewer. Supplementary the originally complex conversion process of a rainfall hyetograph into a surface runoff hydrograph is often reduced to a simplified model with initial and continuing losses. The resulting effective rainfall hyetograph is then transformed into a surface runoff hydrograph using also simple models, e.g. (synthetic) unit hydrographs, time-area diagrams or reservoir models. Using these simplified conceptual models reduces the computing time significantly. A simulation run of a moderate sized system, over several decades with time steps in the order of minutes, only takes few seconds on recent computers. Nevertheless searching the enormous vector space, that is given by comparing different measures (e.g. spatial configuration of CSOs or tanks or stormwater infiltration devices) or that the input parameters span (e.g. for auto calibration or Monte Carlo simulation for uncertainty analysis), leads to the fact, that computing time is still a limiting factor, even with sophisticated searching algorithms. So speed is the limiting factor for an efficient use of existing auto calibration tools, such as CALIMERO (Kleidorfer *et al.*, 2009a) or PEST (Doherty *et al.*, 1994). Hence the development of simulation code that can be executed faster is still an important issue in integrated urban drainage modelling. For example Feyen *et al.* (2007) used parallel computing to implement a conceptual rainfall runoff model named LISFLOOD which simulates the river discharge in large drainage basins as a function of spatial information on topography, soils and land cover.

### Parallel computing

Parallel Computing is a term used in computer science which describes a way to solve a computationally expensive problem by dividing it into subtasks. These subtasks are then distributed on different independent computational units and run concurrently (in parallel). Splitting up problems into parallel parts is called decomposition. There exists a huge variety of decomposition techniques, like recursive decomposition, data decomposition, exploratory decomposition and speculative decomposition (Grama *et al.*, 2003).

The performance of parallel implementations is calculated using speedups. Speedup is defined as the ratio between the computational time needed for the best sequential algorithm divided by the computational time required for the parallel algorithm (Akhter, 2006):

$$speedup(n_i) = \frac{time_{bestseqalg}}{time_{parallelalg}(n_i)} \quad (1)$$

Scalability of an algorithm is how far it can be parallelized and how well it works on more parallel entities. Linear scalability is the theoretically best achievable condition, it means adding n entities makes the run-time n-times better.

Different parallel computing entities exist depending on the parallel computing environment, e.g. cores of a multi-core CPU or servers in a clustered environment, for which the algorithm was designed for. Martins *et al.* (2001) give an overview and compare several common used parallel computing environments. Choosing the suited platform is generally critical and depends on several impact factors. Urban drainage simulations are characterized by many small computations with a high amount of dependencies. The architecture of single chip multiprocessors (commonly known as

multi-core processors) fulfils the hope of having the best outcome with respect to parallel performance (Olukotun et al., 1996). Furthermore these multi-core systems are cheap available at the consumer market. With regard to the software OpenMP (OpenMP Architecture Review Board, 2008) and the standardized portable operating system interface [for Unix] threads (POSIX-threads) were used to implement the parallel strategies. POSIX is set of standardized libraries and tools that allow writing portable applications.

Communication and synchronization in parallel computing is the exchange of information between concurrent tasks. Depending on the need of synchronization, a programmer can choose between locks, semaphores, monitors, conditional-variables, messages, fences and barriers. Synchronization errors are subtle, hard to find and hard to fix, because a near infinite number of situations can occur depending on the race of the threads. These errors are mostly unknown in classical sequential programming and have names like “race condition”, “dead locks” and “live locks“ (Akhter, 2006).

Finding parallel strategies with good communications and avoiding concurrency errors are critical for high performance of parallel systems.

## METHODS

In this chapter the parallel strategies developed for conceptual sewer systems are described that were used to accelerate the computation of the processes. For demonstration purposes the conceptual sewer system consisted of a reduced set of nodes: a catchment for the constant dry weather flow, a sewer for the routing process, a mixer and a file-out node for writing the simulation results into a file. In total three strategies were found. The first one is the flow parallel strategy (FPS) which uses a data parallel decomposition. The second, pool pipeline strategy (PPS) and third one, ordered pipeline strategy (OPS) are based on a pipelined method in which the nodes are pipelined through the threads.

### Flow parallel strategy

The flow parallel strategy combines data parallel and task parallel model described in Grama et al. (2003). The flow chart of Fig. 1 illustrates the depending computations of the sewer system for one simulation time step. On each input flow originated by a catchment a new thread is started (see Fig. 1). Each sequential node after the input node is then calculated by this thread. If several flows are merged due to a mixer node all threads except one are shut down. This thread continues to compute the merged downstream flow.

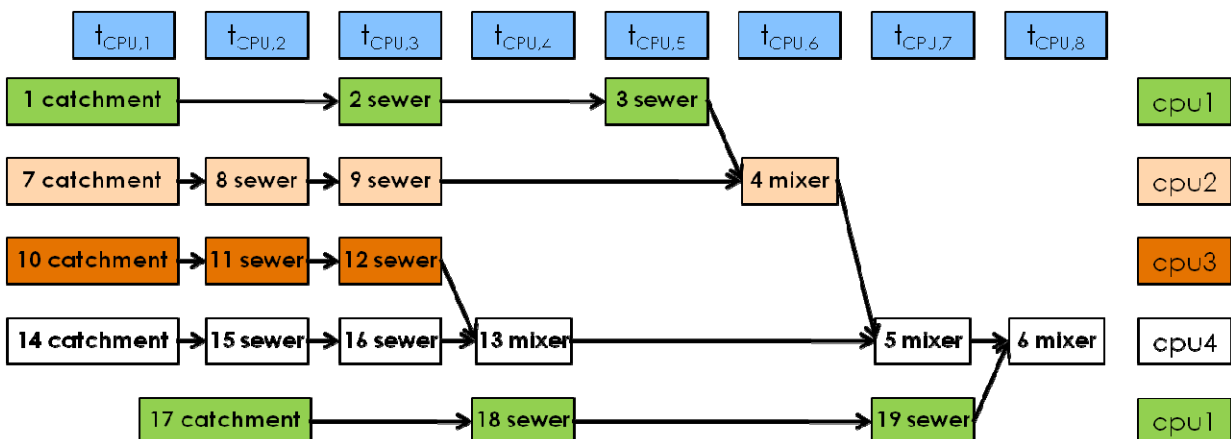


Fig. 1. Flow parallel strategy realized on a quad core CPU

At the mixer node which functions as a junction, synchronization is required. Each mixer node contains a counter starting with the number of input flows, i.e. number of connected links. If a

thread reaches a mixer node its counter is decremented by one. The thread which decrements the counter to zero continues the calculation of the nodes following the mixer node downstream.

The number of threads is limited by the number of input flows. At each mixer node at least two flows are merged. This effect impacts the possible parallel streams to be calculated, i.e. the more the flow of sewer sections downstream is already calculated the less parallelization is possible. Due to this fact this strategy is not able to fully utilize all cores over the entire sewer system.

The arrows in Fig. 1 are symbolizing the data transfer between the nodes. The ones which start and end in different colours are data transfers between the CPU cores. The ones which start and end in the same colour are data transfers in the cores. Data transfers between CPU cores causes CPU flushes and memory stalls which are expensive with regard to CPU cycles (Drepper, 2007). The advantage of this strategy is that the data needed for downstream computations is more likely to remain in the cores, as can be seen in Fig. 1.

### Pool pipeline strategy

The second strategy starts a thread per simulation time step (see Fig. 2). The goal for the thread is to get all nodes of the sewer system executed. A node can execute the implemented algorithms, e.g. Muskingum routing, if all its upstream nodes are in the same time step. Each thread maintains a private set of nodes, called a pool, which were not yet processed. The next node to be processed is chosen randomly from this pool and gets executed if the dependencies are figured out. If the pool is empty the simulation time step is finished and the thread computing the step can be reallocated for the next time step.

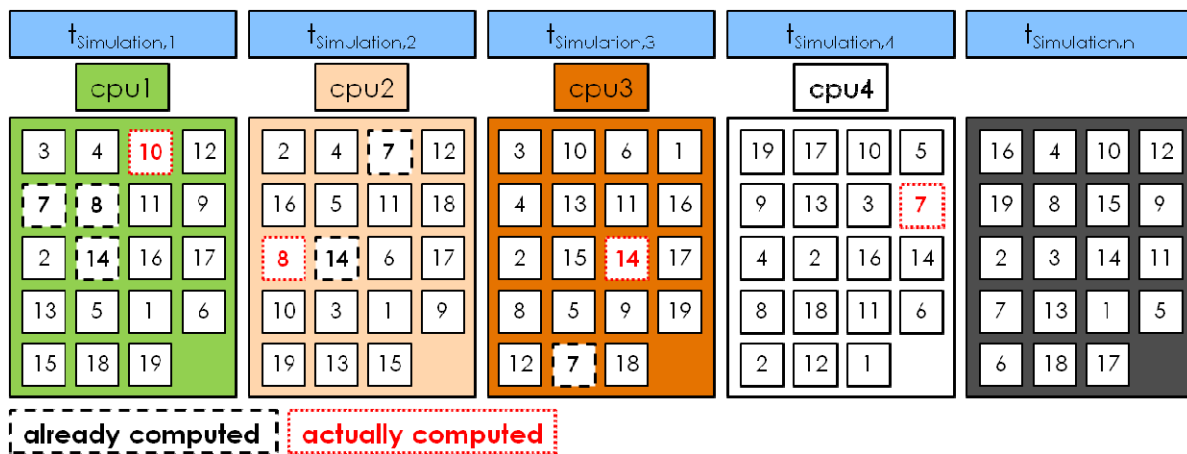


Fig. 2. Pool pipeline strategy realized on a quad core CPU

### Ordered pipeline strategy

The goal of the OPS was to get rid of the non deterministic behaviour of the PPS. As in the PPS a thread is responsible for a time step. Instead of randomly choosing the next node to be executed, the execution order is determined prior the simulation run (see Fig. 3). The execution order is calculated by applying a topological sorting algorithm onto the sewer system (Kahn, 1962). The topological sorting assures that the nodes are executed with all dependencies satisfied. Each thread is connected to the thread executing the next time step by a first in first out (FIFO) queue. If a node is finished in time step  $dt$  it is fed into the queue of the time step  $dt+1$ . An empty queue means that the responsible thread is finished with this time step. The thread can then move on to calculate a new pending time step.

In the pipelined strategies PPS and OPS the number of threads to be utilized is the highest number of sequential nodes in the sewer system. OPS can even run all nodes in parallel by intelligently

buffering the nodes data exchange. This means that the pipelined strategies are able to handle more cores than the FPS. The disadvantage of these pipelined strategies is that every single data exchange between nodes is carried out between threads and therefore CPU cores.

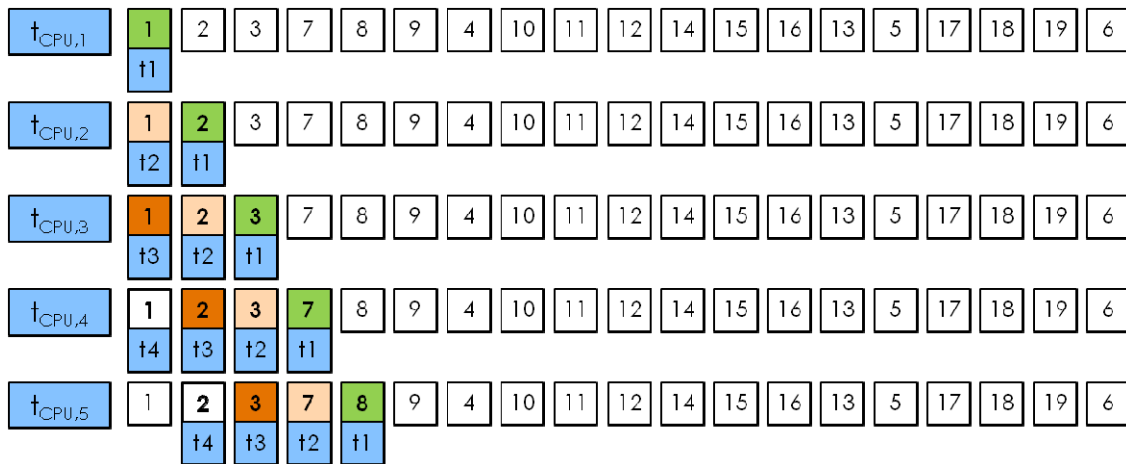


Fig. 3. Ordered pipeline strategy realized on a quad core CPU

### URBAN DRAINAGE SYSTEM USED FOR TESTING

CD3 is a reimplementation of the MATLAB based CITY DRAIN software tool (Achleitner et al., 2007) with a focus on performance. Therefore in CD3 the same mathematical models are used as in CITY DRAIN. On the other hand the benchmark of CD3 against CITY DRAIN is meaningless due to the fact that CD3 has been realised outside the MATLAB environment which has a significant impact on the computational performance. On the other hand CD3 cannot be benchmarked with other software tools because of the different model approaches, i.e. runoff generation, surface routing and sewer flow implemented. Artificial testing systems were generated to highlight the advantages and disadvantages of the different parallel strategies. Furthermore, one converted and adapted system of the city of Innsbruck was used to show how well the parallel implementation can handle real world scenarios. The software was benchmarked to demonstrate the increase of computational performance with the above mentioned testing systems.

### Artificial urban drainage systems

Generally the structure of urban drainage systems complies with an inverted tree. The artificial testing systems chosen differ in complexity and degree of reality. The first artificial testing system is a sequential line of sewer sections starting with an input node and ending in a file out node delivering the results in a CSV format (see Fig. 4). With this testing system it is possible to demonstrate the benefits of parallelization even for simple conceptual models. The sequential nature of this testing system is well suited to show the communication overhead of the FPS. The second testing system is used to point out the theoretical upper bounds reached by the FPS. Several sequential sewer sections in parallel are mixed together at the end (see Fig. 5). The third testing system is chosen due to its close scheme to natural drainage structures which have the shape of an inverted binary tree (see Fig. 6). Two sewers sections are mixed up and connected to a sewer section downstream.

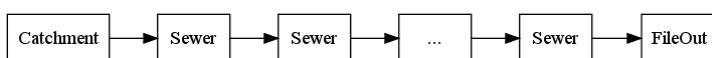


Fig. 4. Artificial testing system consisting of a sequential line of sewer sections





## RESULTS AND DISCUSSION

### Benchmark Environment

The hardware on which CD3 was benchmarked is an Intel (R) Core(TM) i7 CPU 920 @ 2.67GHz equipped with four hyperthreaded cores. The software was compiled with the Intel (R) Compiler Suite v11. Each testing system was run with a different count of allowed threads from one to ten. As benchmark results the minimum of four simulation runs was taken. The processor has eight virtual threads, two hyperthreaded per core. Due to parallel strategies are based on structural decomposition and not on time decomposition the simulation time of the runs was two hours with five minutes time steps. The results are presented using two kinds of diagrams. The diagrams on the left depict the total time the simulation takes. The diagrams on the right show the speedups. For speedup calculations the single thread performance is equivalent to the best sequential algorithm as is apparent from equation 1. The x-axis is always the number of threads that a strategy is allowed to use.

### Computational calculation time for artificial urban drainage systems

The testing system of Fig. 8 consists of 1000 sequential sewer sections. Despite the fact that this sewer system is intrinsic sequential the implementation of the OPS seems to gain a speedup using more threads. Although the implementation of the FPS cannot scale in this testing system the communication overhead in such situations is insignificant.

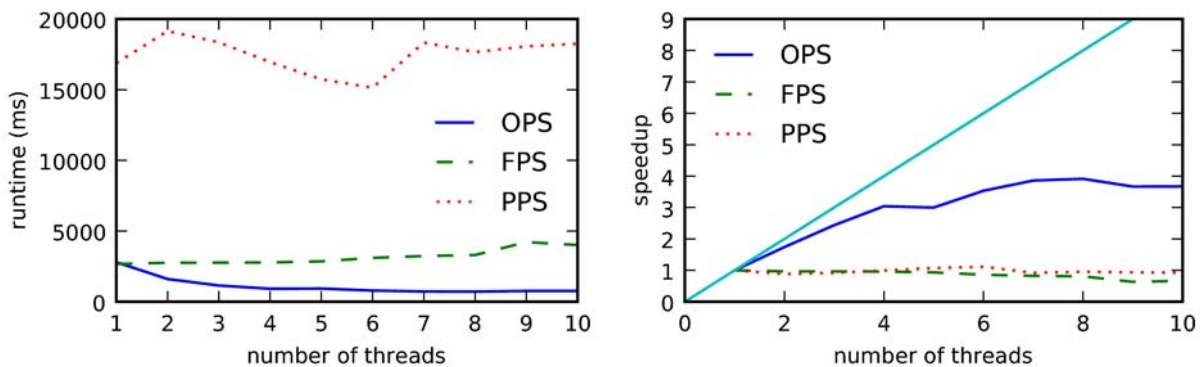


Fig. 8. Results of 1000 sequential sewer sections

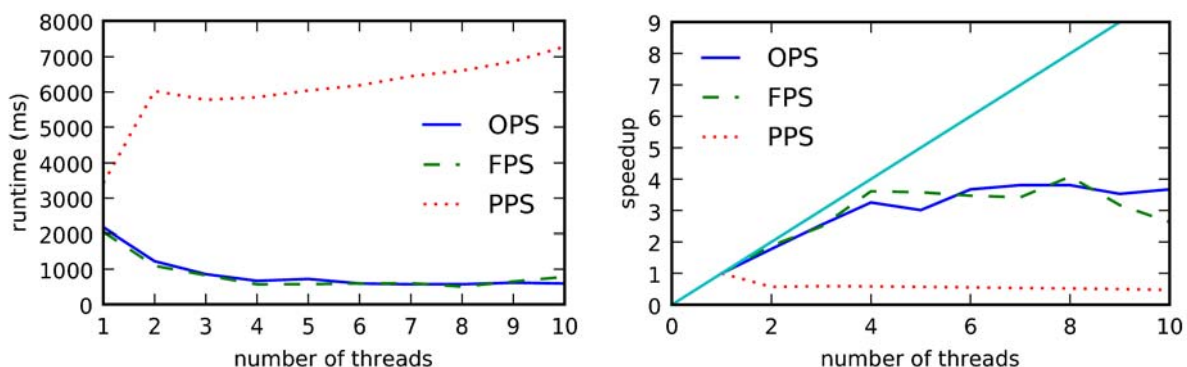
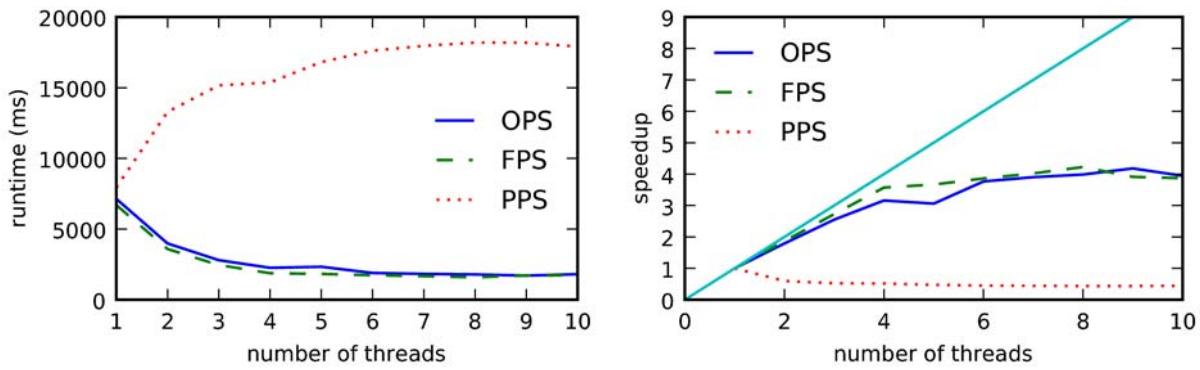


Fig. 9. Results of eight parallel streams each 100 sewer sections long

Fig. 9 shows the results of the parallel testing system with eight parallel streams each 100 sewer sections long. This testing system should give particularly good results for the FPS, but also the implementation of the OPS seems to scale pretty well to the four available cores. Because of the unusable bad computational performance of the PPS the results are not explained further in the discussion chapter.

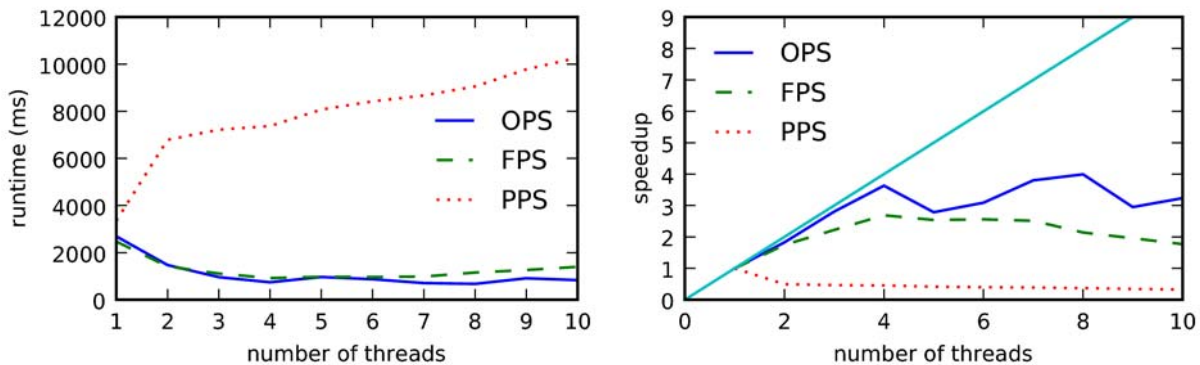


**Fig. 10. Results of a binary tree system with ten generations**

In Fig. 10 the results of the binary tree with ten generations and 2047 nodes are depicted. This tree offers enough parallel streams that the FPS and OPS are able to reduce the run-time on more cores, although the FPS has a slight edge ahead. Even with a smaller testing system consisting of two generations and seven nodes (not depicted here) the OPS showed significant time reductions by using more cores.

### Computational calculation time for real world case study of Innsbruck

Fig. 11 shows the results from the real world case study of the sewer system of Innsbruck. The ordered pipelined strategy performs obviously better than the other strategies. On the right hand side of Fig. 11 the implementation of the ordered pipelined strategy is given separately.



**Fig. 11. Results of the real world sewer system of the city of Innsbruck**

### CONCLUSION AND OUTLOOK

This paper demonstrates the possibility of parallel computing to decrease the run time of urban drainage simulations. Three parallel strategies were tested by means of different benchmark systems. The implementation of the PPS was the one which performed worst. No speedup could be seen in a single test. The randomized characteristic of the PPS was identified to be the weak spot. Getting rid of this randomization resulted in the well performing OPS. The OPS achieved good results throughout all testing systems, from small binary tree to sequential and real world testing systems. In the sequential testing system the implementation of the OPS had the highest speedup of 3.9 on eight threads. In the tree testing system with ten generations the FPS and OPS were on par with a maximum speedup of around 3.8. In the real word sewer system of Innsbruck OPS gained a maximum speedup of 4.1.

The sewer testing systems used in the benchmarks included also worst case scenarios to emphasize the weak spot of the parallel strategies, i.e. their worst performance. Therefore it can be concluded

that OPS due the good results on all testing systems should also perform good on other sewer systems not investigated in this paper.

OPS should benefit from an optimized lock-free queue implementation as described in Fober et al. (2001) and Fober et al. (2002) because this strategy has a high usage of queues for communication. Although the FPS performed well on some testing systems, the performance was lower than expected. A parallel computing trace tool revealed high amounts of lock usages implicitly emitted by OpenMP.

## REFERENCES

- Achleitner, S.; Möderl, M.; Rauch, W. (2007): CITY DRAIN©—An open source approach for simulation of integrated urban drainage systems. In: *Environmental Modelling and Software*, Vol. 22(8), p. 1184-1195.
- Akhter, S. (2006): *Multi-core Programming: Increasing Performance Through Software Multi-threading*. Intel Press
- Butler, D.; Davies, J. (2004): *Urban drainage*. Spon Press, London
- Butler, D.; Schütze, M. (2005): Integrating simulation models with a view to optimal control of urban wastewater systems. In: *Environmental Modelling and Software*, Vol. 20(4), p. 415-426
- Doherty, J.; Brebber, L.; Whyte, P. (1994): PEST manual. In: *Watermark Computing, Corinda, Australia*
- Drepper, U. (2007): What every programmer should know about memory. In: *Proceedings of the Red Hat Summit*, Nashville, USA, 21.11.2007
- Feyen, L.; Vrugt, J.; Nualláin, B.; van der Knijff, J.; De Roo, A. (2007): Parameter optimisation and uncertainty assessment for large-scale streamflow simulation with the LISFLOOD model. In: *Journal of Hydrology*, Vol. 332(3-4), p. 276-289
- Fober, D.; Letz, S.; Orlarey, Y. (2002): Lock-Free Techniques for Concurrent Access to Shared Objects. In: *Proceedings of the JIM Actes des Journées d'Informatique Musicale*, Marseille, France, p. 143-150
- Fober, D.; Orlarey, Y.; Letz, S. (2001): *Optimised Lock-Free FIFO Queue*. Technical Report-01-01-01 Grame
- Grama, A.; Gupta, A.; Karypis, G. (2003): *Introduction to Parallel Computing*. 0002. Ed., Addison Wesley Pub Co Inc
- Kahn, A. B. (1962): Topological sorting of large networks. In: *Communications of the ACM*, Volume 5(11), p. 558-562
- Kleidorfer, M.; Leonhardt, G.; Mair, M.; McCarthy, D.T.; Kinzel, H.; Rauch, W. (2009a): CALIMERO - A model independent and generalized tool for autocalibration. In: *Proceedings of the 8th International Conference on Urban Drainage Modelling*, Tokyo, Japan, 7. - 11.09.2009
- Kleidorfer, M.; Möderl, M.; Fach, S.; Rauch, W. (2009b): Optimization of measurement campaigns for calibration of a conceptual sewer model. In: *Water Science and Technology*, Vol. 59(8), p. 1523-1530
- Kongetira, P.; Aingaran, K.; Olukotun, K. (2005): Niagara: A 32-Way Multithreaded Sparc Processor. In: *IEEE Micro*, Vol. 25(2), p. 21-29
- Martins, S.D.L.; Ribeiro, C.C.; Rodriguez, N. (2001): Parallel Computing Environments. In: *Proceedings of the Handbook of Applied Optimization*
- Olukotun, K.; Nayfeh, B.A.; Hammond, L.; Wilson, K.; Chang, K. (1996): The Case for a Single-Chip Multiprocessor. In: *Proceedings of the IEEE Computer*, p. 2-11
- OpenMP Architecture Review Board (2008): *OpenMP application program interface 3.0*. <http://www.openmp.org>
- Rauch, W.; Aalderink, H.; Krebs, P.; Schilling, W.; Vanrolleghem, P. (1998): Requirements for integrated wastewater models – driven by receiving water objectives. In: *Water Science and Technology*, Vol. 38(11), p. 97-104
- Rauch, W.; Bertrand-Krajewski, J.L.; Krebs, P.; Mark, O.; Schilling, W.; Schütze, M.; Vanrolleghem, P.A. (2002): Deterministic modelling of integrated urban drainage systems. In: *Water Science and Technology*, Vol. 45(3), p. 81-94.

# Parallel computing in integrated urban drainage simulations

G. Burger<sup>\*)</sup>, S. Fach<sup>\*\*)</sup>, H. Kinzel<sup>\*\*\*)</sup> and W. Rauch<sup>\*\*)</sup>

*\*Institute of Computer Science, University of Innsbruck,  
Technikerstr. 21A, A-6020 Innsbruck, Austria (E-mail: gregor.burger@uibk.ac.at)*

*\*\* Unit of Environmental Engineering, University of Innsbruck, Technikerstr. 13,  
A-6020 Innsbruck, Austria (E-mail: stefan.fach@uibk.ac.at, wolfgang.rauch@uibk.ac.at)*

*\*\*\* hydro-IT GmbH, Technikerstr. 13, A-6020 Innsbruck, Austria  
(E-mail: Kinzel@hydro-it.com)*

## ABSTRACT

Integrated urban drainage modelling is used to analyze how existing urban drainage systems respond to particular conditions. Based on these integrated models, researchers and engineers are able to e.g. estimate longterm pollution effects, optimize the behaviour of a system by comparing impacts of different measures on the desired target value or get new insights on systems interactions. Although the use of simplified conceptual models reduces the computational time significantly, searching the enormous vector space that is given by comparing different measures or that the input parameters span, leads to the fact, that computational time is still a limiting factor. Due to the stagnation of single thread performance in computers and the rising number of cores one needs to adapt algorithms to the parallel nature of the new CPUs to fully utilize the available computing power. In this work a new developed software tool named CD3 for parallel computing in integrated urban drainage systems is introduced. From three investigated parallel strategies two showed promising results and one results in a speedup of up to 4.2 on an eight-way hyperthreaded quad core CPU and shows even for all investigated sewer systems significant run-time reductions.

## KEYWORDS

CD3, conceptual model, flow parallel strategy, integrated urban drainage modelling, parallel computing, pipeline strategy

## INTRODUCTION

While in former days the processors (CPU's) got significantly more powerful (and thus faster), nowadays it is not the single CPU that is developed further but instead the number of processors is increased. To fully utilize that available computing power one needs to adapt algorithms to the parallel nature of these new CPU-architectures.

### **The need of computational power for urban drainage simulations**

Integrated urban drainage modelling (IUDM) combines the main subsystems of urban drainage systems (e.g. natural and urban catchments, sewers, receiving water bodies and waste water treatment plants) of the urban (waste) water cycle into one single model (Rauch *et al.*, 2002; Butler and Schütze, 2005). The main use of those models is to analyze how existing urban drainage systems respond to particular conditions (Butler and Davies, 2004). Generally, deterministic models are used which always produce the same output for a specific set of input data. With these models engineers and scientists are able to fully reason about sewer system performance, discharge to the receiving water body and river water quality. Based on these integrated models, researchers are able to e.g. estimate longterm pollution effects (Rauch *et al.*, 1998), optimize the behaviour of a system by comparing impacts of different measures on the desired target value or get new insights on systems interactions.

IUDM models are often formulated in a simplified manner applying e.g. hydrological routing instead of hydrodynamic wave equations to calculate the waste water transport in the sewer. Supplementary the originally complex conversion process of a rainfall hyetograph into a surface runoff hydrograph is often reduced to a simplified model with initial and continuing losses. The resulting effective rainfall hyetograph is then transformed into a surface runoff hydrograph using also simple models, e.g. (synthetic) unit hydrographs, time-area diagrams or reservoir models. Using these simplified conceptual models reduces the computing time significantly. A simulation run of a moderate sized system, over several decades with timesteps in the order of minutes, only takes few seconds on recent computers. Nevertheless searching the enormous vector space, that is given by comparing different measures (e.g. spatial configuration of CSOs or tanks or stormwater infiltration devices) or that the input parameters span (e.g. for auto calibration or Monte Carlo simulation for uncertainty analysis), leads to the fact, that computing time is still a limiting factor, even with sophisticated searching algorithms. So speed is the limiting factor for an efficient use of existing auto calibration tools, such as CALIMERO (Kleidorfer et al., 2009a) or PEST (Doherty et al., 1994). Hence the development of simulation code that can be executed faster is still an important issue in integrated urban drainage modelling. For example Feyen et al. (2007) used parallel computing to implement a conceptual rainfall runoff model named LISFLOOD which simulates the river discharge in large drainage basins as a function of spatial information on topography, soils and land cover.

### Parallel computing

Parallel Computing is a term used in computer science which describes a way to solve a computational expensive problem by dividing it into subtasks. These subtasks are then distributed on different independent computational units and run concurrently (parallel). Splitting up problems into parallel parts is called decomposition. There exists a huge variety of decomposition techniques, like recursive decomposition, data decomposition, exploratory decomposition and speculative decomposition (Grama et al., 2003).

$$speedup(n_i) = \frac{time_{bestseqalg}}{time_{parallelalg}(n_i)} \quad (1)$$

$$max\ speedup = \frac{1}{S + \frac{1-s}{S}} \quad (2)$$

The performance of parallel implementations is calculated using speedups. Speedup is defined as the quotient of the computational time needed for the best sequential algorithm divided by the computational time required for the parallel algorithm (equation 1). Both algorithms deliver the same result. An upper limit of the speedup can be calculated using Amdahls Law (equation 2) (Akhter, 2006). Scalability of an algorithm is how far it can be parallelized and how well it works on more parallel entities. Linear scalability is the theoretically best achievable condition, it means adding n entities makes the runtime n-times better.

Different parallel computing entities exist depending on the parallel computing environment, e.g. cores of a multi-core CPU or servers in a clustered environment, for which the algorithm was designed for. Martins et al. (2001) give an overview and compare several common used parallel computing environments. Choosing the suited platform is generally critical and depends on several impact factors. Urban drainage simulations are characterized by many small computations with a high amount of dependencies. The architecture of single chip multiprocessors (commonly known as multi-core processors) fulfils the hope of having the best outcome with respect to parallel

performance (Olukotun et al., 1996). Furthermore these multi-core systems are cheap available at the consumer market.

Communication and synchronization in parallel computing is the exchange of information between concurrent tasks. Depending on the need of synchronization, a programmer can choose between locks, semaphores, monitors, conditional-variables, messages, fences and barriers. Synchronization errors are subtle, hard to find and hard to fix, because a near infinite number of situations can occur depending on the race of the threads. These errors are mostly unknown in classical sequential programming and have names like “race condition”, “dead locks” and “live locks“ (Akhter, 2006).

Finding parallel strategies with good communications and avoiding concurrency errors are critical for high performance of parallel systems.

**Multi-core**

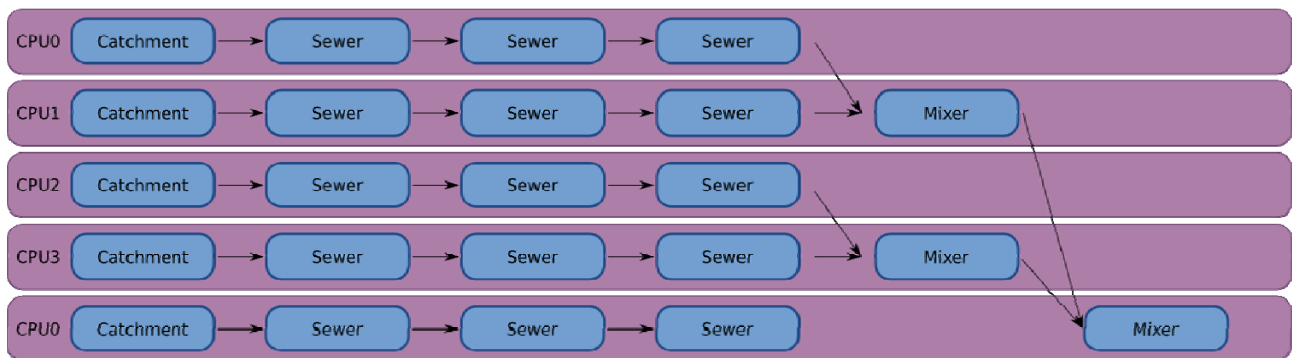
The efforts of this work focuses on implementing CD3 on a multi-core system in the consumer class. PCs of the consumer class are cheap, available and usable from the users of the simulation community. Beside that multi-core systems are the future of desktop computing. Single thread performance is stagnating, but the number of cores is rising (Kongetira et al., 2005). OpenMP (OpenMP) and POSIX threads were used to implement the parallel strategies.

**METHODS**

In this chapter the parallel strategies used are described. The following strategies were identified to possibly accelerate the computational time, implemented in the newly developed software tool CD3 and benchmarked to demonstrate the increase of computational performance. The first strategy can be classified as a data parallel model. The second and third strategies are pipeline models.

**Flow parallel strategy**

The flow parallel strategy combines data parallel and task parallel model. On each input flow originated by a catchment a new thread is started (see Fig. 1). Each sequential node after the input node is then calculated by this thread. If several flows of different characteristics are mixed up to a single flow due to a mixer node one thread waits until all others are finished. The thread waiting for the other threads to be finished continues to compute the nodes after the mixer node.



**Fig. 1. Flow parallel strategy realized on a quad core CPU**

At the mixer node which functions as a junction, synchronization is required. Each mixer node contains a counter starting with the number of input flow, i.e. number of connected links. If a thread reaches a mixer node its counter is decremented by one. The thread which decrements the counter to zero continues the calculation of the nodes following the mixer node downstream.

The number of threads is limited by the number of input flows. At each combining element, e.g. mixer at least two flows are merged. This effect impacts the possible parallel streams to be calculated, i.e. the more the flow of sewer sections downstream is already calculated the less parallelization is possible. Due to this fact this model is not able to fully utilize all cores over the entire model.

The advantage of this model is that the data needed remains in the cores, i.e. in the caches as can be seen in Fig. 1. The arrows symbolizing the data transfer, the ones which cross the CPU boundaries are transfers between the CPU cores. Data transfer between CPU cores causes CPU flushes which are expensive with regard to CPU cycles (Drepper, 2007).

### Pool pipeline strategy

The second strategy starts a thread per time step. The goal for the thread is to get all nodes of the model into the specified state, i.e. time step. A node can execute the implemented algorithms, e.g. Muskingum mixing, if all the nodes connected are in the same state needed to compute (see Fig. 2). Each thread has a private set of nodes, called a pool, which were not processed before. The next node to be processed is chosen randomly from this pool and gets executed if the dependencies are figured out. If the pool is empty the thread will be finished.

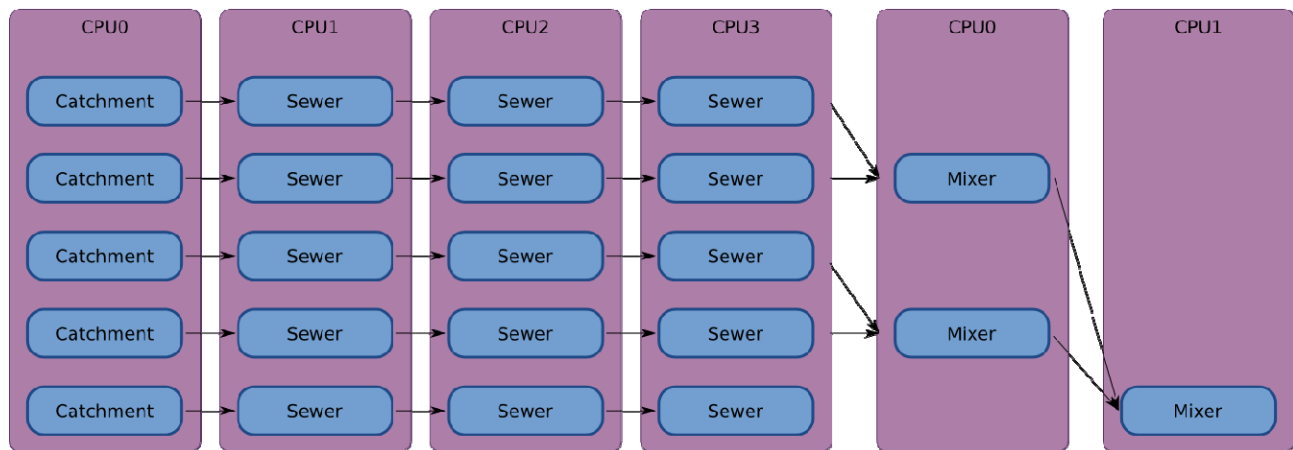


Fig. 2. Ordered pipeline strategy realized on a quad core CPU

### Ordered pipeline strategy

The ordered pipeline strategy is implemented to get the non deterministic characteristics out of the pool pipeline strategy. As in pool pipelining a thread is responsible for each time step, but instead of randomly choosing the next node to be executed, the execution order is determined prior the simulation run. First in first out (FIFO) queues are spanned between threads executing neighbouring timesteps, i.e. timestep (dt) and timestep (dt+1). If a node is finished in timestep (dt) it is fed into the queue of the timestep (dt+1). Due to the fact that in IUDM the models are directed acyclic graphs, the ordering is determined by the sorting of topology. In the pipelined strategies, i.e. pool pipeline strategy and ordered pipeline strategy the number of threads to be utilized is the highest number of sequential nodes in the model. Theoretically, the pipelined strategies are able to handle more cores than the flow parallel strategy.

The disadvantage of these parallel strategies is that every single data exchange between nodes is carried out between threads and therefore CPU cores. This effect can be seen in Fig. 2 by the amount of arrows crossing the CPU boundaries.



## URBAN DRAINAGE SYSTEM USED FOR TESTING

CD3 is a reimplemention of the MATLAB based CITY DRAIN software tool (Achleitner et al., 2007) with a focus on performance. Therefore in CD3 the same mathematical models are used as in CITY DRAIN. On the one hand side the benchmark of CD3 against CITY DRAIN is meaningless due to the fact that CD3 has been realised outside the MATLAB environment which has a significant impact on the computational performance. On the other hand CD3 cannot be benchmarked with other software tools because of the different model approaches, i.e. runoff generation, surface routing and sewer flow implemented. Artificial testing systems were generated to highlight the advantages and disadvantages of the different parallel implementations. Furthermore, one converted and handcrafted system of the city of Innsbruck was used to show how well the parallel implementation can handle real world scenarios.

### Artificial urban drainage systems

Generally the structure of urban drainage systems complies with an inverted tree. The artificial testing systems chosen differ in complexity and degree of reality. The first artificial testing system is a sequential line of sewer sections starting with an input node and ending in a file node which delivers the results in a CSV format (see Fig. 3). With this testing system it is possible to demonstrate the benefits of parallelization even for simple conceptual models. This testing system is also suited to show how much communication overhead the implementation of the flow parallel strategy produces. The second testing system is used to point out the theoretically upper bounds reached by the flow parallel strategy. Several sequential sewer sections in parallel are mixed together at the end (see Fig. 4). The third testing system is chosen due to its close scheme to natural drainage structures which have the shape of an inverted binary tree (see Fig. 5). Two sewers sections are mixed up and connected to a sewer section downstream.

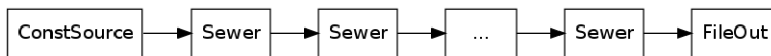


Fig. 3. Artificial testing system consisting of a sequential line of sewer sections

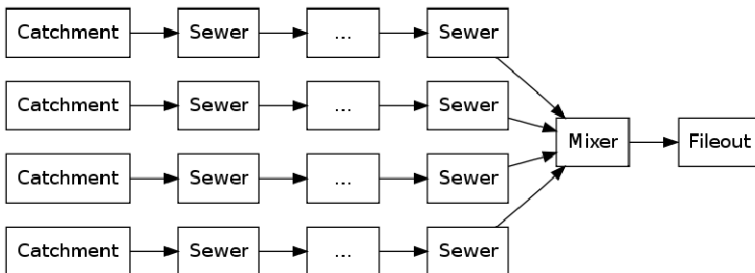


Fig. 4. Artificial testing system consisting of a several sequential sewer sections in parallel

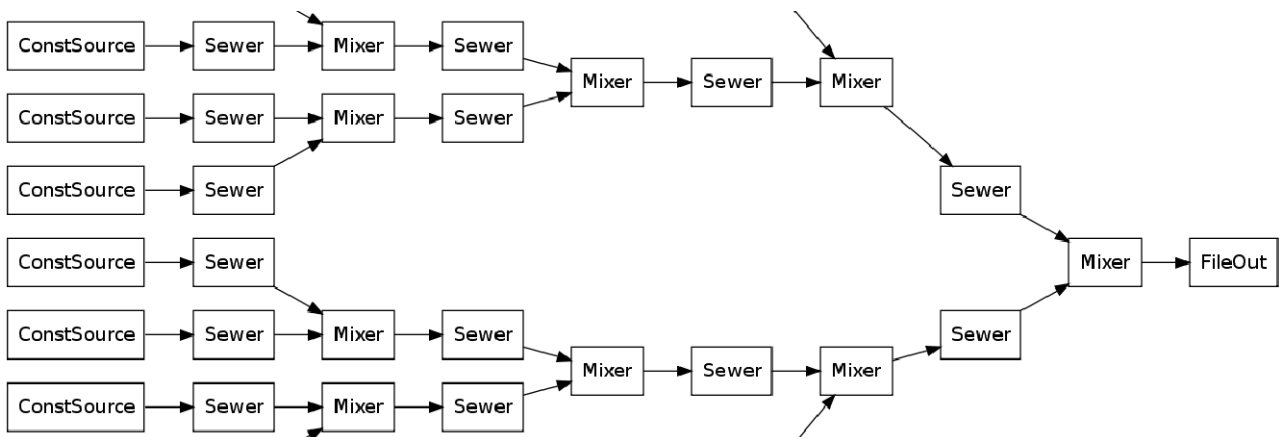


Fig. 5. Detail of the artificial testing system with a binary tree structure

## Real world case study of Innsbruck

To have one real world example as testing system the conceptual sewer system of the city of Innsbruck developed by Kleidorfer et al. (2009b) was used and converted into the native CD3 input format. This combined sewer system consists of 53 catchments, a total runoff effective area of 915 ha and a total basin volume of 5100 m<sup>3</sup>. In the city of Innsbruck live 165,000 population equivalents (PE) with a daily dry weather flow (DWF) of 200 l/(PE d).

## RESULTS AND DISCUSSION

### Benchmark Environment

The hardware on which CD3 was benchmarked is an Intel (R) Core(TM) i7 CPU 920 @ 2.67GHz Quad Core. The software was compiled with the Intel (R) Compiler Suite v11. Each model was run with threads limited from one to ten. As benchmark results the average of four simulation runs are taken. The processor has eight virtual threads, two per core hyperthreaded. Due to parallel strategies are based on structural decomposition and not on time decomposition the simulation time of the runs was two hours with five minutes time steps.

### Computational calculation time for artificial urban drainage systems

The left side of Fig. 6 shows the results of the parallel testing system with eight parallel streams each 100 sewer sections long. This testing system should give particularly good results for the flow parallel strategy, but also the implementation of the ordered pipelined strategy seems to scale pretty well to the four available cores. Because of the unusable bad computational performance of the pipelined parallel strategy the results are not explained further in the discussion chapter.

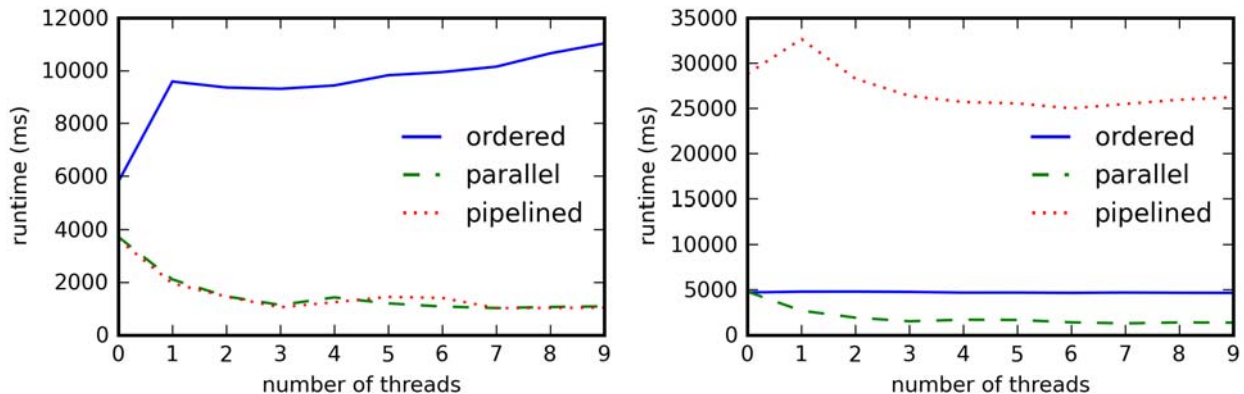


Fig. 6. Run-time for eight parallel streams each 100 sewer sections long (left side) and for 1000 sequential sewer sections (right side)

The second testing system (see right side of Fig. 6) consists of 1000 sequential sewer sections. Despite the fact that this model is intrinsic sequential the implementation of the ordered pipelined strategy seems to gain a speedup using more threads. Although the implementation of the flow parallel strategy cannot scale in this testing system the communication overhead in such situations is insignificant.

In Fig. 7 two binary tree testing systems were benchmarked. These systems differ in the number of generations. The results of the tree with two generations and four nodes is depicted on the left, whereas the results of a tree with ten generations and 1024 nodes is illustrated on the right. The small tree is used to examine if all strategies also perform well for small systems. Even with this small testing system the ordered pipelined strategy showed significant time reductions by using more cores. The big tree offers enough parallel streams that the implementation of the flow parallel strategy is able to reduce the run-time on more cores.

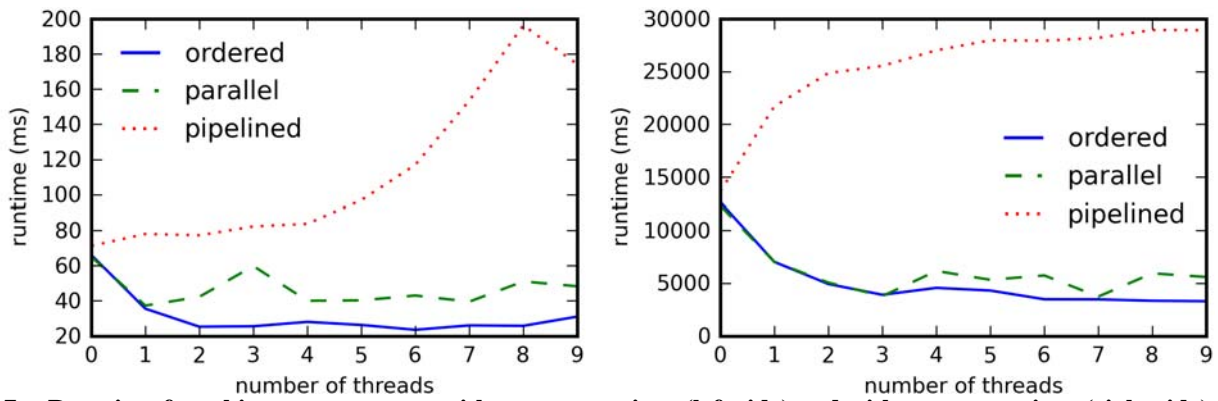


Fig. 7. Run-time for a binary tree system with two generations (left side) and with ten generations (right side)

### Computational calculation time for real world case study of Innsbruck

Fig. 8 shows the results from the real world case study of the sewer system of Innsbruck. The ordered pipelined strategy performs obviously better than the other strategies. On the right hand side of Fig. 8 the implementation of the ordered pipelined strategy is given separately.

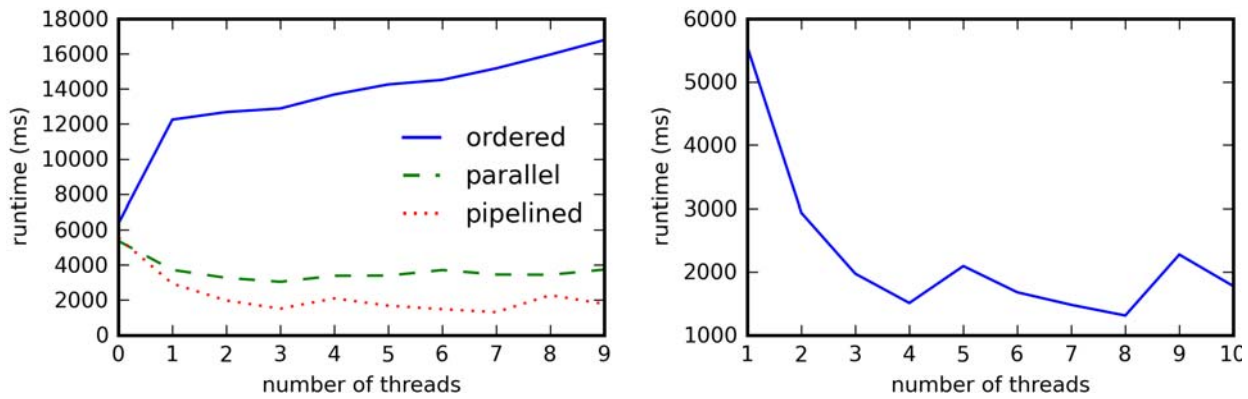


Fig. 8. Run-time of the real world sewer system of the city of Innsbruck (left side) for all strategies and for the ordered pipeline strategy (right side)

### Maximum achieved speedups

In the sequential testing system (see Fig. 6 left side) the implementation of the ordered pipeline strategy has the highest speedup of 3.9 times on eight threads. On the binary tree system with ten generations the ordered pipelined strategy has the maximum speedup of 3.8 followed by the flow parallel strategy with a speedup of 3.3. In the real world scenario of the sewer system of Innsbruck the implementation of ordered pipeline strategy has a maximum speedup of 4.2.

### CONCLUSION AND OUTLOOK

In this paper we investigate the option of parallel computing to increase the computational speed in urban drainage simulations. Several parallelisation strategies are tested by means of benchmark systems. The implementation of the pooled pipeline strategy is the one which performed worst, as no speedup is seen in a single test. The randomized characteristics of the pool pipelined algorithm have been identified to be the weak spot. This randomization was removed and restricted to ordered calculations which lead to the ordered pipeline implementation. The ordered pipeline strategy performed well on all testing systems, from small binary tree to sequential sewer and real world systems. Due to the shared calculation units of two hyperthreads on a core, e.g. floating point unit (FPU) and arithmetic and logic unit (ALU) etc. and the almost exclusive floating point calculations used in IUDM simulations, scaling above four threads is almost impossible. Nevertheless, the ordered pipeline strategy reaches a speedup of 4.2 in some testing systems.

Further enhancements can be achieved by using lock-free algorithms (Fober et al., 2002). Especially, the implementation of the ordered pipeline strategy should benefit from an optimized lock-free queue implementation as described in Fober et al. (2001). Although the flow parallel strategy performed well on some testing systems, the performance was lower than expected. A hand crafted interception library revealed high amounts of lock usages implicitly emitted by OpenMP (OpenMP).

## REFERENCES

- Achleitner, S.; Möderl, M.; Rauch, W. (2007): CITY DRAIN©—An open source approach for simulation of integrated urban drainage systems. In: *Environmental Modelling and Software*, Vol. 22(8), p. 1184-1195.
- Akhter, S. (2006): *Multi-core Programming: Increasing Performance Through Software Multi-threading*. Intel Press
- Butler, D.; Davies, J. (2004): *Urban drainage*. Spon Press, London
- Butler, D.; Schütze, M. (2005): Integrating simulation models with a view to optimal control of urban wastewater systems. In: *Environmental Modelling and Software*, Vol. 20(4), p. 415-426
- Doherty, J.; Brebber, L.; Whyte, P. (1994): PEST manual. In: *Watermark Computing, Corinda, Australia*
- Drepper, U. (2007): What every programmer should know about memory. In: *Proceedings of the Red Hat Summit*, Nashville, USA, 21.11.2007
- Feyen, L.; Vrugt, J.; Nualláin, B.; van der Knijff, J.; De Roo, A. (2007): Parameter optimisation and uncertainty assessment for large-scale streamflow simulation with the LISFLOOD model. In: *Journal of Hydrology*, Vol. 332(3-4), p. 276-289
- Fober, D.; Letz, S.; Orlarey, Y. (2002): Lock-Free Techniques for Concurrent Access to Shared Objects. In: *Proceedings of the JIM Actes des Journées d'Informatique Musicale*, Marseille, France, p. 143-150
- Fober, D.; Orlarey, Y.; Letz, S. (2001): *Optimised Lock-Free FIFO Queue*. Technical Report-01-01-01 Grame
- Grama, A.; Gupta, A.; Karypis, G. (2003): *Introduction to Parallel Computing*. 0002. Ed., Addison Wesley Pub Co Inc
- Kleidorfer, M.; Leonhardt, G.; Mair, M.; McCarthy, D.T.; Kinzel, H.; Rauch, W. (2009a): CALIMERO - A model independent and generalized tool for autocalibration. In: *Proceedings of the 8th International Conference on Urban Drainage Modelling*, Tokyo, Japan, 7. - 11.09.2009
- Kleidorfer, M.; Möderl, M.; Fach, S.; Rauch, W. (2009b): Optimization of measurement campaigns for calibration of a conceptual sewer model. In: *Water Science and Technology*, Vol. 59(8), p. 1523-1530
- Kongetira, P.; Aingaran, K.; Olukotun, K. (2005): Niagara: A 32-Way Multithreaded Sparc Processor. In: *IEEE Micro*, Vol. 25(2), p. 21-29
- Martins, S.D.L.; Ribeiro, C.C.; Rodriguez, N. (2001): Parallel Computing Environments. In: *Proceedings of the Handbook of Applied Optimization*
- Olukotun, K.; Nayfeh, B.A.; Hammond, L.; Wilson, K.; Chang, K. (1996): The Case for a Single-Chip Multiprocessor. In: *Proceedings of the IEEE Computer*, p. 2-11
- OpenMP, C. C++ application program interface. In: *For further details see <http://www.openmp.org>*
- Rauch, W.; Aalderink, H.; Krebs, P.; Schilling, W.; Vanrolleghem, P. (1998): Requirements for integrated wastewater models – driven by receiving water objectives. In: *Water Science and Technology*, Vol. 38(11), p. 97-104
- Rauch, W.; Bertrand-Krajewski, J.L.; Krebs, P.; Mark, O.; Schilling, W.; Schütze, M.; Vanrolleghem, P.A. (2002): Deterministic modelling of integrated urban drainage systems. In: *Water Science and Technology*, Vol. 45(3), p. 81-94.

# List of Figures

1.1. Ways to study a system.(redraw from [LK97]) . . . . .	3
2.1. Schematic on the application of rainfall loss model [AMR07] . . .	20
2.2. Lossmodels [AMR07] . . . . .	21
2.3. Routing Methods [AMR07] . . . . .	23
2.4. Muskingum Routing Method [AMR07] . . . . .	23
2.5. Integrated drainage system (redrawn from [WJP+02]) . . . . .	24
2.6. Schematic description of a block . . . . .	25
3.1. Class Overview . . . . .	29
3.2. The <i>Node</i> Class . . . . .	30
3.3. The <i>Model</i> Class . . . . .	31
3.4. The <i>Simulation</i> Class . . . . .	32
3.5. Flow Parallel Strategy . . . . .	35
3.6. Input Counter . . . . .	36
3.7. Pool Pipeline Strategy . . . . .	38
3.8. Ordered Pipeline Strategy . . . . .	39
3.9. An example DAG . . . . .	41
3.10. Flow exchange problem DAG . . . . .	44
3.11. Flow exchange problem ordered . . . . .	44
3.12. UML of the Flow class . . . . .	46

4.1. Sequential Testing System . . . . .	50
4.2. Parallel Sewer Testing System . . . . .	51
4.3. Treelike Sewer Testing System . . . . .	51
4.4. Real World Testing System . . . . .	53
4.5. Die Shots of the Two Processors used for Benchmarking . . . . .	55
4.6. Sequential (10) . . . . .	57
4.7. Sequential (100) . . . . .	57
4.8. Sequential (1000) . . . . .	58
4.9. Parallel (2-10) . . . . .	59
4.10. Parallel (2-100) . . . . .	59
4.11. Parallel (4-10) . . . . .	60
4.12. Parallel (4-100) . . . . .	60
4.13. Parallel (8-10) . . . . .	60
4.14. Parallel (8-100) . . . . .	61
4.15. Tree (two generations) . . . . .	61
4.16. Tree (four generations) . . . . .	62
4.17. Tree (seven generations) . . . . .	62
4.18. Tree (ten generations) . . . . .	62
4.19. Innsbruck . . . . .	63
4.20. Parallel system with OpenMP scheduler disabled(4-100) . . . . .	65
4.21. Tree system with OpenMP scheduler disabled (seven generations) . . . . .	65
4.22. Tree system with OpenMP scheduler disabled (ten generations) . . . . .	66
4.23. Sequential (10) . . . . .	67
4.24. Sequential (100) . . . . .	67

4.25. Sequential (1000) . . . . .	67
4.26. Parallel (2-10) . . . . .	69
4.27. Parallel (2-100) . . . . .	69
4.28. Parallel (4-10) . . . . .	69
4.29. Parallel (4-100) . . . . .	70
4.30. Parallel (8-10) . . . . .	70
4.31. Parallel (8-100) . . . . .	70
4.32. Tree (two generations) . . . . .	71
4.33. Tree (four generations) . . . . .	71
4.34. Tree (seven generations) . . . . .	72
4.35. Tree (ten generations) . . . . .	72
4.36. Innsbruck . . . . .	73
4.37. Innsbruck OPS . . . . .	74
4.38. Innsbruck FPS . . . . .	75
4.39. Tree (10 Generations) OPS . . . . .	75
4.40. Tree (10 Generations) FPS . . . . .	76
4.41. Innsbruck OPS . . . . .	76
4.42. Innsbruck FPS . . . . .	77
4.43. Tree (10 Generations) OPS . . . . .	77
4.44. Tree (10 Generations) FPS . . . . .	77
A.1. Starting CityDrain3 without parameters. . . . .	88
A.2. Setting the Path for the MinGW compiler . . . . .	103
A.3. Overview of the essential interfaces . . . . .	104
A.4. The <i>Model</i> class . . . . .	105

A.5. The <i>Simulation</i> class . . . . .	105
A.6. The <i>Node</i> class . . . . .	105



## List of Algorithms

1.	The main loop of a simulation run. . . . .	33
2.	The execute $n$ procedure. . . . .	34
3.	<code>flow_parallel_simulation</code> . . . . .	36
4.	<code>run_flow(sn)</code> . . . . .	36
5.	<code>get_predecessors</code> . . . . .	37
6.	<code>pps_main</code> . . . . .	38
7.	<code>pps_run_timestep(POOL<sub>t</sub>)</code> . . . . .	38
8.	Topological sorting . . . . .	40
9.	<code>ops_main</code> . . . . .	43
10.	<code>ops_run_timestep(t)</code> . . . . .	45



## List of Tables

3.1. Example sort of DAG . . . . .	41
4.1. Thread Scheduling with three threads . . . . .	64
4.2. Thread Scheduling with four threads . . . . .	64



# Bibliography

- [ABC<sup>+</sup>06] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, and S. W. Williams. The landscape of parallel computing research: A view from berkeley. *Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, December*, 18(2006-183):19, 2006.
- [ADE92] J. L. Armstrong, B. O. Da, and S. S. Erlang. IMPLEMENTING a FUNCTIONAL LANGUAGE FOR HIGHLY PARALLEL REAL TIME APPLICATIONS. *SETSS*, 1992.
- [AJS07] Ayaz Ali, Lennart Johnsson, and Jaspal Subhlok. Scheduling fft computation on smp and multicore systems. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 293–301, New York, NY, USA, 2007. ACM.
- [Akh06] S. Akhter. *Multi-core Programming: Increasing Performance Through Software Multi-threading*. Intel Press, June 2006.
- [AMR07] Stefan Achleitner, Michael Möderl, and Wolfgang Rauch. CITY DRIN © - an open source approach for simulation of integrated urban drainage systems. *Environmental Modelling & Software*, 22(8):1184–1195, August 2007.
- [Aus05] M. Austern. Proposed draft technical report on c++ library extensions. Technical report, Technical Report PDTR 19768, n1745 05-0005, ISO/IEC, 2005.
- [BBG09] Johannes Borgström, Karthikeyan Bhargavan, and Andrew D. Gordon. A compositional theory for stm haskell. In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 69–80, New York, NY, USA, 2009. ACM.
- [BD96] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing systems*, 2(2):131–152, 1996.
- [BD04] D. Butler and J. W. Davies. *Urban drainage*. Spon Pr, 2004.

- [BDH03] L. A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The google cluster architecture. *IEEE micro*, 23(2):22–28, 2003.
- [Bor07] Shekhar Borkar. Thousand core chips: a technology perspective. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 746–749, New York, NY, USA, 2007. ACM.
- [Bos92] H. Bossel. *Modellbildung und Simulation. Konzepte, Verfahren und Modelle zum Verhalten dynamischer Systeme: Ein Lehr- und Arbeitsbuch mit Simulations-Software*. Vieweg Braunschweig, 1992.
- [BS05] David Butler and Manfred Schütze. Integrating simulation models with a view to optimal control of urban wastewater systems. *Environmental Modelling & Software*, 20(4):415 – 426, 2005. Vulnerability of Water Quality in Intensively Developing Urban Watersheds.
- [BV05] R. Buyya and S. Venugopal. A gentle introduction to grid computing and technologies. *CSI Communications*, 29(1):9–19, 2005.
- [CBM<sup>+</sup>08] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, 2008.
- [CKM<sup>+</sup>01] B. Chocat, P. Krebs, J. Marsalek, W. Rauch, and W. Schilling. Urban drainage redefined: from stormwater removal to integrated management. *Water Science & Technology*, 43(5):61–68, 2001.
- [CLRS01] T. H. Corman, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press Cambridge, MA, USA, Cambridge, MA, USA, 2001.
- [Dav09] Beazley David. Inside the python GIL, June 2009.
- [Die05] R. Diestel. Graph theory, volume 173 of graduate texts in mathematics. *Springer, Heidelberg*, 91:92, 2005.
- [DJ97] Butler D and Parkinson J. Towards sustainable urban drainage. <http://www.iwaponline.com/scripts/dtSearch/dtisapi6.dll?cmd=getdoc&DocId=9512&Index=E%3a%5cdtIndex%5cIW%5fWST&HitCount=4&hits=17b+17c+17d+17e+&SearchForm=D%3a%5ciwaponline%5csearch%5csearch%2ehtm>, May 1997.
- [Dre07] U. Drepper. *What every programmer should know about memory*. 2007.

- [EA03] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. *ACM SIGOPS Operating Systems Review*, 37(5):237–252, 2003.
- [FF01] C. Flanagan and S. N. Freund. Detecting race conditions in large programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 90–96. ACM New York, NY, USA, 2001.
- [Fos95] I. Foster. *Designing and building parallel programs: concepts and tools for parallel software engineering*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.
- [Gee05] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.
- [GGK03] Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing*. Addison Wesley Pub Co Inc, 0002 edition, February 2003.
- [God02] A. Godber. *Linux Function Interception*. 2002.
- [Guj06] W. Gujer. *Siedlungswasserwirtschaft*. Springer, 2006.
- [Guj08] Willi Gujer. *Systems Analysis for Water Technology*. Springer, October 2008.
- [Int09] Intel Corporation. *Intel® C++ Compiler Professional Edition 11.1 for Linux, In Depth*, 2009.
- [Kah62] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, 1962.
- [KAO05] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparcc processor. *IEEE Micro*, 25(2):21–29, 2005.
- [KGNC02] H. Korving, P. Van Gelder, J. Van Noortwijk, and F. Clemens. Influence of model parameter uncertainties on decision-making for sewer system management. In *Proc. 5th Int. Conf. on Hydroinformatics, ID Cluckie, D. Han, JP Davis and S. Heslop, eds*, volume 2, pages 1361–1366, 2002.
- [KHYP08] DongHyun Kang, Saeyoung Han, SeoHee Yoo, and Sungyong Park. Prediction-based dynamic thread pool scheme for efficient resource usage. In *CITWORKSHOPS '08: Proceedings of the 2008 IEEE*

*8th International Conference on Computer and Information Technology Workshops*, pages 159–164, Washington, DC, USA, 2008. IEEE Computer Society.

- [KMFRed] Manfred Kleidorfer, Michael Möderl, Stefan Fach, and Wolfgang Rauch. Optimization of measurement campaigns for calibration of a conceptual sewer model. *Water Science and Technology*, accepted.
- [LHK<sup>+</sup>04] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. Gpgpu: general purpose computation on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, page 33, New York, NY, USA, 2004. ACM.
- [LK97] A. M. Law and W. D. Kelton. *Simulation modeling and analysis*. McGraw-Hill Higher Education, 1997.
- [LML00] Yibei Ling, Tracy Mullen, and Xiaola Lin. Analysis of optimal thread pool size. *SIGOPS Oper. Syst. Rev.*, 34(2):42–55, 2000.
- [MLG07] D. Manocha, M. C. Lin, and N. Govindaraju. GPGPU to Many-Core processing: Higher performance for mass market applications. In *Manycore Computing Workshop*, 2007.
- [Mus08] D. Muschalla. Optimization of integrated urban wastewater systems using multi-objective evolution strategies. *Urban Water Journal*, 5(1):59–67, 2008.
- [Nae05] Gustaf Naeser. Priority inversion in multi processor systems due to protected actions. *Ada Lett.*, XXV(1):43–47, 2005.
- [NG92] R. Netzer and S. Ghosh. *Efficient race condition detection for shared-memory programs with post/wait synchronization*. University of Wisconsin-Madison, Computer Sciences Dept., 1992.
- [NL91] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, 1991.
- [NM92] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):74–88, 1992.
- [NS03] N. Nethercote and J. Seward. Valgrind a program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2):44–66, 2003.



- [NS07] N. Nethercote and J. Seward. Valgrind: A framework for heavy-weight dynamic binary instrumentation. In *Proceedings of the 2007 PLDI conference*, volume 42, pages 89–100. ACM New York, NY, USA, 2007.
- [NT05] M. Nichols and D. Taylor. A mechanism for visualizing TCP-socket interactions. In *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pages 212–224. IBM Press, 2005.
- [ONH<sup>+</sup>96] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *IEEE Computer*, pages 2–11, 1996.
- [PSCS01] Irfan Pyarali, Marina Spivak, Ron Cytron, and Douglas C. Schmidt. Evaluating and optimizing thread pool strategies for real-time corba. In *LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, pages 214–222, New York, NY, USA, 2001. ACM.
- [PTM96] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Parallel Distrib. Technol.*, 4(2):63–79, 1996.
- [RGK02] W. Rauch, W. Gujer, and P. Krebs. Grundlagen der siedlungsentwässerung. *Skript zur Vorlesung Siedlungsentwässerung der ETH Zürich*, 2002.
- [RS09] The GCC community RM Stallman. *The GNU OpenMP implementation*. 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA, 4.3.3 edition, 2009. accessed at <http://gcc.gnu.org/onlinedocs/>.
- [Sch98] D. C. Schmidt. Evaluating architectures for multithreaded object request brokers. 1998.
- [SMSW09] Michael F. Spear, Maged M. Michael, Michael L. Scott, and Peng Wu. Reducing memory ordering overheads in software transactional memory. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 13–24, Washington, DC, USA, 2009. IEEE Computer Society.
- [Spr77] Renzo Sprugnoli. Perfect hashing functions: a single probe retrieving method for static sets. *Commun. ACM*, 20(11):841–850, 1977.
- [SSTW09] Achleitner S, Fach S, Einfalt T, and Rauch W. Nowcasting of

rainfall and of combined sewage flow in urban drainage systems. <http://www.iwaponline.com/wst/05906/wst059061145.htm>, March 2009.

- [ST97] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, February 1997.
- [SV96] D. Schmidt and S. Vinoski. Comparing alternative programming techniques for multi-threaded CORBA servers: Thread-per-Object. *C++ Report*, 8(6), 1996.
- [Tho94] Alexander Thomasian. On a more realistic lock contention model and its analysis. In *Proceedings of the Tenth International Conference on Data Engineering*, pages 2–9, Washington, DC, USA, 1994. IEEE Computer Society.
- [Vin07] Steve Vinoski. Concurrency with erlang. *IEEE Internet Computing*, 11(5):90–93, 2007.
- [WJP<sup>+</sup>02] Rauch W, Bertrand-Krajewski J, Krebs P, Mark O, Schilling W, Schtze M, and Vanrolleghem P. Deterministic modelling of integrated urban drainage systems. *Water science and technology*, February 2002.
- [WKP<sup>+</sup>02] J. D. Warnock, J. M. Keaty, J. Petrovick, J. G. Clabes, C. J. Kircher, B. L. Krauter, P. J. Restle, B. A. Zoric, and C. J. Anderson. The circuit and physical design of the POWER4 microprocessor. *IBM Journal of Research and Development*, 46(1):27–51, 2002.