NTNU – Trondheim
Norwegian University of
Science and Technology

Towards a Parallel Multiphase Solver Based on Potential Ordering

Henrik Vikøren

Master of Science in Physics and Mathematics
Submission date: June 2015
Supervisor: Helge Holden, MATH
Co-supervisor: Knut-Andreas Lie, SINTEF

Norwegian University of Science and Technology
Department of Mathematical Sciences

Abstract

This thesis presents our work towards developing a parallel multiphase solver based on potential ordering [2, 4, 5, 3]. We begin the thesis by introducing the Fast Multiphase Solver, as developed by Natvig, Lie et al. [3]. Then we, in turn, study the parallel algorithms developed by Fleischer et al. [9] and Bader [10]. As a part of the study we have implemented the algorithms and give an overview of these implementations. Our implementation of the algorithm due to Fleischer [9] confims the serial complexity but are unable to achieve parallel speed-up. Results based on our implementation of Bader's algorithm discourage further development with this approach. Finally we discuss further possibilities and propose our own ideas on how to adapt the parallel algorithms for use in a parallel multiphase solver.

Sammendrag

Denne oppgaven presenterer vårt arbeid mot å utvikle en parallell multifaseløser basert på potensiell reordning [2, 4, 5, 3]. Oppgaven begynner med å introdusere en rask multifase løser utviklet av Natvig og Lie [3]. Deretter studerer vi to parallelle algoritmer utviklet av Fleisher [9] og Bader [10] etter tur. Som en del av dette studiet har vi implementert egne versjoner av disse algoritmene, og presenterer disse. Vår implementasjon av algoritmen utviklet av Flesicher [9] bekrefter den teoretiske kompleksiteten til algoritmen, men gir ingen parallel speed-up. Resultatene fra vår implementasjon av Baders algoritme svekker vår tro på den som et reelt alternativ til å utvikle en parallel multifaseløser. Avsluttningsvis diskuterer vi videre muligheter og kommer med egne forslag til hvordan de overnevnte algoritmene kan tilpasses til bruk i en parallell mulitfaseløser.

Contents

Section 1

Introduction

Fossil resources remains the world's primary energy source. In 2012 it accounted for 81.7% of the world's total energy consumption [1]. Even though the worlds fossil reserves are dwindling and other sources of energy are emerging, it is reasonable to assume that they will be a key source of energy also in the future. Since man began using oil and gas for energy purposes, tremendous amounts have been extracted. The dominating regulator behind this have been economic concerns. As a consequence, the reservoirs which are easy to produce have already been developed. At the same time the worlds energy demand continues to increase. Today's oil and gas companies therefore have to produce more petroleum from ever more challenging locations.

The world's petroleum reserves are found in underground reservoirs, beneath crust, rock, and sometimes sea. Extracting it presents a number of challenges, dependent on where it is located, and mishaps can have devastating consequences. To avoid such mishaps, reservoir engineers are always trying to plan developments in an as detailed manner as possible. With geological information about the reservoir, modern reservoir simulation software can be used to foresee potential danger, and take necessary precautions. Due to the importance of good intelligence to make the right decisions, reservoir simulation has grown into a research field of its own.

One of the contributions within this field is the Fast Multiphase Solver (FMS) [2, 3, 4, 5] developed by Natvig, Lie et al. This is an efficient algorithm for solving the transport equation, which draws benefits from the equations underlying properties. The method considers a discontinuous Galerkin scheme for computing the transport of the fluid. By treating the non-linear system of equations arising out of this as a graph, the elements can be rearranged in a fashion that makes it possible to solve the system by a single sequential traversal of the elements. This approach yields significant performance improvements and has received attention in the field. Kwok an Tchelepi [6] have implemented a related method on non-linear multiphase flow. Shahvali and Tchelepi [7] used a hybrid method to obtain a convergent scheme while also taking counter current flow into account.

Reservoir simulations is a computationally intensive discipline that is closely linked to the development of better and faster computers. Today, reservoir engineers usu-

ally have a better geological description of the reservoir than they can exploit. The seismic data has more detail than what is feasible to use for computing. More efficient methods are therefore always being sought. Algorithms exploiting parallel processors are a way to achieve this. In this thesis we work towards a parallel implementation of the fast multiphase solver developed by Natvig, Lie et al. In a preliminary study done by the author [8], possibilities for this implementation were explored, and this thesis is a natural extension of that work. An important fact that was discovered in that preliminary study was the need for a parallel implementation of both the reordering of grid elements and the sequential solving of the elements. If this is not achieved, the parallel algorithm will not yield satisfactory results in regard to speed.

In our search of a new algorithm we will first take a look at two existing parallel algorithms, with traits that we find favorable to our application. The first algorithm we consider an algorithm developed by Fleischer et al. [9]. This algorithm uses a split and conquer approach to achieve parallelism. It is a well tested algorithm, fully capable of topologically sorting graphs containing cycles. Whether we are able to reorder and solve in the same step is uncertain. We do however, study this algorithm more closely as sorts the graph in parallel, and handles cycles in an appropriate manner.

Furthermore we study a cycle detection algorithm developed by Bader [10]. Cycle detection is an important step in the reordering of the elements used in the FMS. Our interest in this particular algorithm stems partially from its good results in relation to run speed, and partially because of the way it partitions the graphs. Transport in reservoirs have physical features which give favor to a partitioning approach like the one used by Bader [10]. This algorithm lack some important abilities, and the main question is therefore whether it is adaptable to the problem at hand.

This thesis is first and foremost meant as an explanation of what we have done on this problem. Since we were not able to finish the parallelization of the FMS we would like it to ease the work of anyone who might be continuing this endeavor. As a result, we have included a section about the implementation of the algorithms we have been working on. The appendices also include well commented listings of some of our code.

Section 2 describes the physical and mathematical background of the problem. It also includes a short introduction to parallel programming. Section 3 presents the two aforementioned algorithms, and puts them in the context of the FMS. In Section 4 we descirbe our work on implementing these algorithms, hopefully making it easier for anyone wanting to continue this work. Section 5 includes performance tests of the two implementations, and a discussion of the results as well as proposals for further works.

Section 2

Background

This section provides the background which the rest of the thesis builds upon. We start of by explaining the physical realities of reservoir engineering in Section 2.1. In Section 2.2 we describe the mathematical model governing the problem, before we move on to its discretization in Section 2.3. Section 2.4 introduces the reordering procedure at the core of the FMS, and Section 2.4 outlines how the serial version of the FMS works. Finally we give a short introduction to key consepts in parallel programming in Section 2.5.

2.1 Physical explanation of the problem

Oil production is a dangerous endeavour. Drilling wells into reservoirs, sometimes found underneath kilometres of sea and crust, involves great risks. The forces related to this can be immense, and if things get out of control the results can be severe. Extensive planning is therefore needed before developing a new field. An important aspect is to identify how the fluids in the reservoir will flow. Information about fluid movement can prevent dangerous blow-outs and leaks, aside from being used to figure out how to recover as much oil as possible. The first step in this process is to establish an as precise as possible physical model of the reservoir and the fluid contained within. Small differences in terms of mathematical accuracy can have large implications. The following section describes the most important physical parameters considered when developing this model.

Extensive models for describing reservoirs exist within the fields of fluid mechanics and geology . These models are very detailed, and thus requires some pretty involved equations. Since the aim of our work is to start developing a parallel version of the FMS we have chosen to simplify these models some. This allows us to focus on the core of the problem without a lot of complicated notation. We trust that it will be possible to extend our simplified model at a later time, should it be required. In the meantime we refer the reader to [11] for a more thorough introduction, and ask that it be noted that the simplified model presented here would not be sufficient in a mature solver designed for industrial purposes.

The problem

Hydrocarbons trapped in reservoirs underground are usually recovered by drilling one or several wells into the reservoir. As long as the pressure in the reservoir is high enough to push the oil to the surface, oil will flow out of the well by it self. Even so, it is often necessary to increases the pressure in the reservoir to get out as much oil as possible. One way of doing this is to inject water into the reservoir and thereby push the oil out. To achieve this, two types of wells are drilled into the reservoir; injection wells for injecting water and production wells for extracting oil. See Figure 2.2 for an illustration.

Injection of water into a reservoir is expensive and requires a certain amount of return to be profitable. At some point, the water injected into the well will have travelled to production wells, so that a fraction of the produced fluid consists of the injected water. When this fraction gets substantial, profitability decreases rapidly. Naturally we would like the water to push out as much hydrocarbons as possible before reaching the production wells. How fast the water reaches the injection wells is influenced by the well locations, and tools for optimizing well placements are in high demand. The FMS is designed for this purpose. Specifically, it gives an approximate answer to the following question: How much oil and water is contained in a small control volume around the point (x, y, z) at time t?

Mathematically speaking, the FMSr is a highly efficient numerical solver of the nonlinear porous medium equations. This is partly because it utilizes certain properties related to the physical problem, and as such we do the same in the parallel implementation. In this section we give a brief introduction to the physical realities as well as some of the assumptions and simplifications made.

Geological aspects

Fossil resources (oil and gas) are found in underground reservoirs. Common for such reservoirs are that they are found in porous rock formations. A porous rock is a rock with cavities, called pores, that can be filled with a fluid. This is usually referred to as the *porous medium*. In Figure 2.1 an illustration of porous rock is shown. There are two quantities used to characterize the porous medium;:

Porosity The porosity of the rock is the fraction of the rock volume that consists of cavities in which fluids can reside. For a control volume V containing pores of volume V_f, the porosity ϕ is thus defined as

$$\phi = \frac{V_f}{V}.$$

The compressibility of the rock can affect the porosity to a varying degree. Here, we neglect rock compressibility.

Permeability The permeability, k, of the rock indicates its ability to transmit a fluid. That is, it is a measure of how much resistance does a fluid flowing through the porous medium meets. It is defined mathematically through

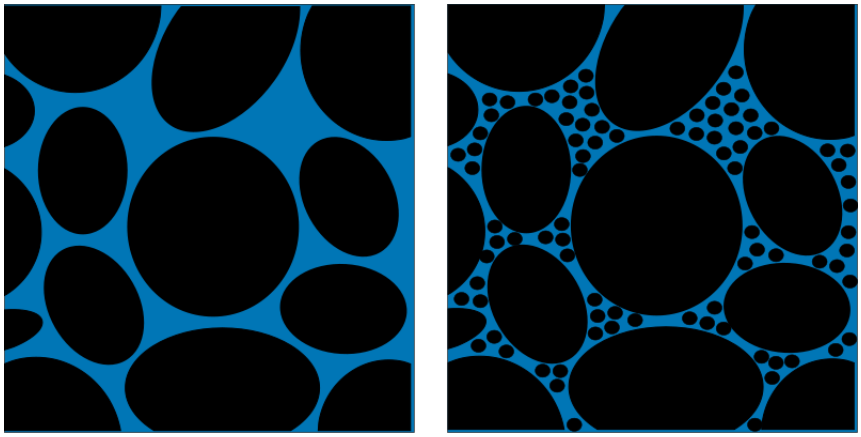


Figure 2.1: Illustration of a porous rock formation. Blue indicates pores in which fluids reside whereas black indicates solid rock. The left picture show a formation with higher porosity than in the right picture.

Darcy's law [12],

$$\mathbf{q} = -k\frac{\nabla p}{\mu}, \tag{2.1}$$

where $\mathbf{q}$ is the flow of the fluid, μ is the fluid viscosity, and ∇p is the pressure gradient.

Fluid description

Petroleum reservoirs contains a mix of fluids. There are water typically and different kinds of hydrocarbons present. Furthermore, the different components occur in one of threes phases; liquid, gaseous or aqueous phase. Different components and phases have different characteristics, and tend to flow individually, though not independently. That is, they flow with different speed but are affected by each other. Throughout this report we will only consider water and black oil in liquid phase. This is done for simplicity, and can easily be extended to any number of phases. To describe the fluids we will use the *saturations* of water and oil. This is a dimensionless quantity, that tells us the portion of water and oil contained in any given control volume inside the reservoir. If we assume that there are no other fluids in the reservoir and the reservoir is completely saturated, the saturation s_α for phase $\alpha \in \{o, w\}$ for oil and water, respectively, has to satisfy

$$s_w + s_o = 1.$$

When calculating the saturations in the reservoir over time there are several factors which come into play. Motivated by this thesis' aim at finding a parallel approach to the FMS, we have chosen to simplify the model quite extensively.

Flux and partial flux The volumetric flux, $\mathbf{q}$, of a fluid is defined as the volumetric amount of fluid passing through a cross section of 1 m^2 per time. We

use partial fluxes to describe water and oil individually, and these are defined analogously, as the amount of water/oil passing through a cross of one square meter per second.

Compressibility The compressibility, β, of a fluid measures its relative volumetric change when put under pressure. It is mathematically defined as:

$$\beta = -\frac{1}{V}\frac{\partial V}{\partial p}.$$

Compressibility affects the fluid pressure and saturations, as it has different values in oil and water. It can also affect the porosity and permeability, since the rock can compress when we apply additional pressure. For simplicity we have chosen to assume no compressibility.

Gravitation Gravitational forces affect the fluid flow, and sometimes cause counter-current flow. This factor is also ignored for simplicity.

Capillary effects Capillary effects can cause the fluid to stick to the wall of the reservoir and hence can affect the flow. Also this effect is ignored in our mathematical description.

Oil recovery

Petroleum reservoirs, whether subsurface or subsea, are all found underground. These have been formed by geological activity over millions of years. For hydrocarbons to be formed there has to be a certain pressure. Because of this, reservoirs are usually under pressure when found in their natural equilibrium. When a well is drilled into it, this pressure will push the oil out of the reservoir until he pressure diminishes and an equilibrium at the top of the well is established. The recovery of oil by means of the natural pressure in the reservoir is called *primary recovery*. Normally around 5-15% of the hydrocarbons in the reservoir can be recovered during the primary recovery.

After the primary recovery we have to force the oil out of the reservoir in order to continue recovering oil. One technique is to inject water in the reservoir. See Figure 2.2 for a simple illustration. The technique has two effects. Firstly it causes the pressure in the reservoir to rise. secondly it serves to displace the oil towards the production well. The use of this technique necessitates the drilling of one or several wells for injection of water, in addition to the wells used for production of oil. The more usual is to have multiple injection wells and multiple production wells. This is especially true for fields situated on land, where the cost of drilling wells are significantly lower than off-shore.

These fields offer possibilities for parallel computations due to the potential of several independent areas. An especially interesting question is how the flow pattern in the reservoir develops in a field with multiple injection and production wells. We expect the areas around the injection wells to be largely dominated by their respective injection wells. These areas should be largely independent of each other, and

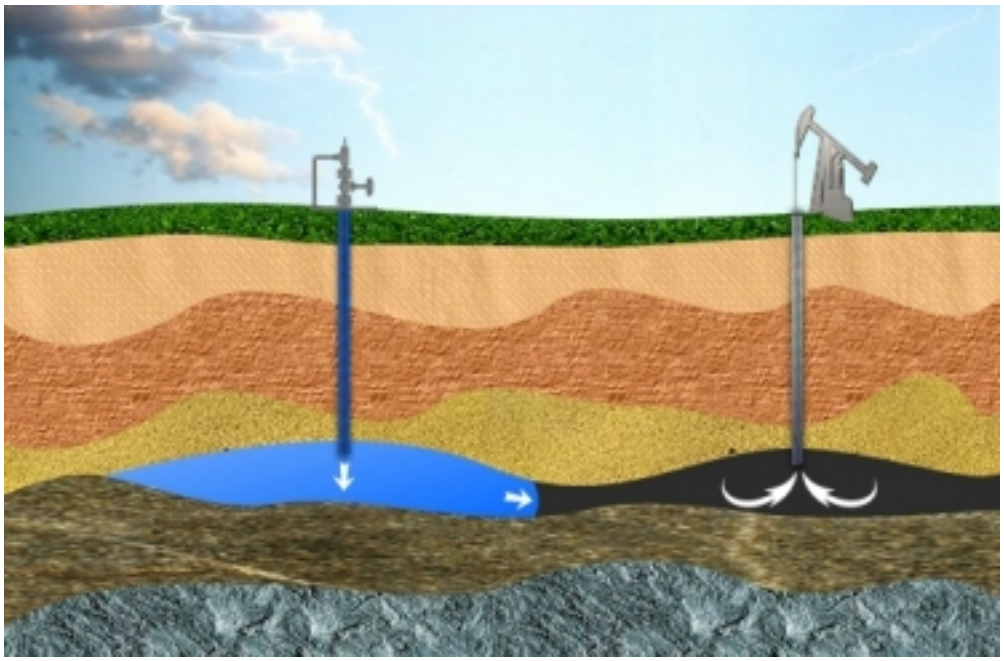


Figure 2.2: Illustration showing how water can be injected into a reservoir to push out hydrocarbons. Image from [13].

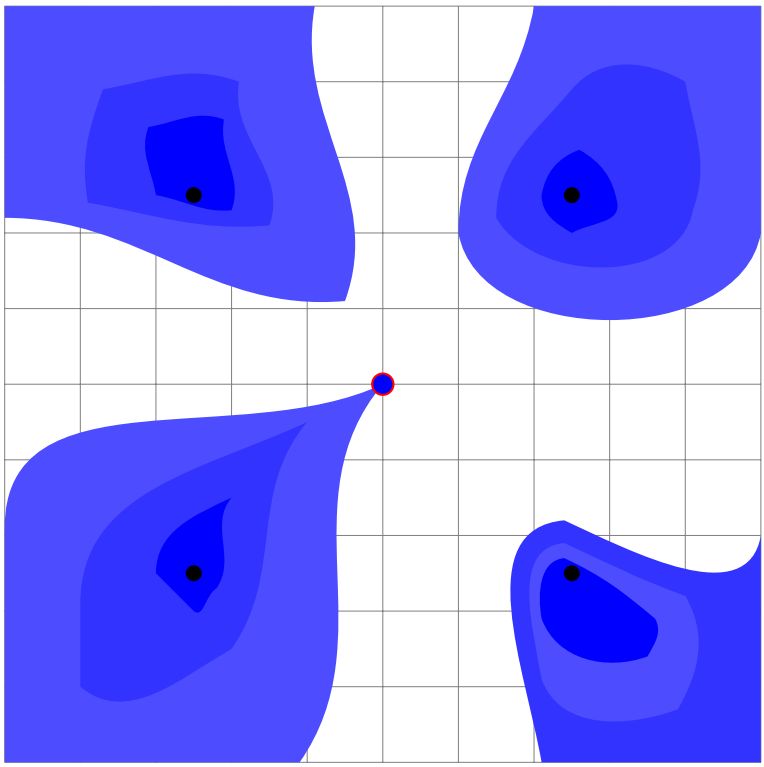


Figure 2.3: Example geometry of field with several injection wells and one production well.

thus well suited for parallelization. Figure 2.3 illustrates how different regions with limited dependence can arise. In a grid with millions of cells, large regions which can be isolated from the rest of the graph can prove a significant advantage.

If we distribute this grid according to the isolated regions we get flow patterns that have few dependencies outside the specific sub region. Exploiting this we should be able to device an algorithm that can do efficient computations in parallel. Some interdependence along sub domain borders will always exists, but with a good partitioning communication costs in these cases should be relatively low. Software packages for partition grids so that the number of edges that have vertices on different processors are minimized are available [14], and should be looked into in due course. Another advantage of this fact is that it gives us a good indication of how we should partition our graphs. Well placements will be known before any computations and can therefore be used as preconditioning for the graph partitioning.

2.2 Mathematical model

We now move on to the mathematical description of the problem. Keep in mind that this is a simple model, not sufficiently accurate to be used in actual simulations for industry purposes. It does, however, convey the key principles of the problem and is therefore sufficiently similar to a full scale model to be used as a basis for developing a parallel solver.

To model the flow of a fluid through a reservoir, we assume conservation of mass and that the fluid will tend two flow in the direction of decreasing pressure, i.e., follow Darcy's law. For a control volume V without any source or sink terms, with boundary ∂V and outward normal vector $\mathbf{n}$, we can formulate the conservation of mass for fluid flow through a porous medium as

$$\frac{\partial}{\partial t}\int_V \rho\phi d\mathbf{x} = -\int_{\partial V} \rho\mathbf{q}\cdot\mathbf{n}\mathrm{d}\mathbf{s}. \tag{2.2}$$

where ρ denotes the density. In layman's terms, Equation (2.2) states that the change in the amount of fluid inside the control volume (the left hand side) is solely due to the flux of the fluid across the boundary of the control volume (the right hand side). Equation (2.2) can also be written on differential form as

$$\frac{\partial}{\partial t}(\rho\phi) + \nabla(\rho\mathbf{q}) = 0.$$

For multiphase flow we can consider the partial flow $\mathbf{q}_\alpha$ of each phase α The partial flows have to satisfy $\sum_\alpha \mathbf{q}_\alpha = \mathbf{q}$. As mention in the previous section this paper assumes two phases: water and oil. This is done for simplicity, but can easily be extended to n phases. Using the saturations s_w and s_o we can write

$$\frac{\partial}{\partial t}(s_\alpha\rho_\alpha\phi) + \nabla\cdot(\rho_\alpha\mathbf{q}_\alpha) = 0, \qquad \alpha\in\{o,w\}. \tag{2.3}$$

Correspondingly for Darcy's law (2.1),

$$\mathbf{q}_\alpha = -k\frac{k_{r\alpha}}{\mu_\alpha}\nabla p, \tag{2.4}$$

which gives

$$\mathbf{q} = -k\left(\frac{k_{rw}}{\mu_w} + \frac{k_{ro}}{\mu_o}\right)\nabla p \tag{2.5}$$

when the two phases are added together. Here we have denoted by k_{rw} and k_{ro} the *relative permeabilities*, which in general depends on s. We also introduce

$$\begin{aligned}\lambda_\alpha &= \frac{k_{r\alpha}}{\mu_\alpha}, \qquad (\alpha \in \{w, o\})\\ \lambda &= \lambda_w + \lambda_o.\end{aligned}$$

Equation (2.5) can now be written

$$\mathbf{q} = -k\lambda\nabla p, \tag{2.6}$$

So far we have considered a control volume without any source or sink terms. We now add a source term r to the model, and make the simplifying assumptions that we introduced in Section 2.1:

1. $\rho_w, \rho_o, \mu_w, \mu_o$ are constant,
2. capillary and gravitational forces are neglible ,
3. $k_{rw} = k_{rw}(s), k_{ro} = k_{ro}(s)$ are known.

Equation (2.3) then becomes

$$\phi\frac{\partial}{\partial t}(s_\alpha) + \nabla\cdot(\mathbf{q}_\alpha) = r_\alpha, \tag{2.7}$$

where r_α denotes the sources contribution to phase α. Combining (2.4) and (2.6) we now get

$$\phi\frac{\partial}{\partial t}(s_\alpha) + \nabla\cdot(\lambda_\alpha\lambda^{-1}\mathbf{q}) = r_\alpha.$$

If we add Equation (2.7) for the two phases together, we get

$$\phi\frac{\partial}{\partial t}(s_w + s_o) + \nabla\cdot(\mathbf{q}_w + \mathbf{q}_o) = r_w + r_o,$$

and since $s_w + s_o = 1$,

$$\nabla\cdot(\mathbf{q}_w + \mathbf{q}_o) = \nabla\cdot\mathbf{q} = r. \tag{2.8}$$

We then have then reached the governing equation for the transport problem:

$$\phi \frac{\partial}{\partial t} s_\alpha = r_\alpha - \mathbf{q} \cdot \nabla f(s_\alpha) - f(s_\alpha) r_\alpha, \tag{2.9}$$

where $f(s_\alpha) = \frac{\lambda_\alpha}{\lambda}$. Note that in the complete system of equations for the entire system the source and sink terms represent the injection and production wells, respectively. As wells are only found in a few cells in the domain, these terms are mostly zero. The physical interpretation of Equation (2.9) is that the change in saturation of phase α, equals the amount of phase α being injected or extracted minus the flow out across the boundaries.

To solve the coupled system consisting of (2.6), (2.8) and (2.9) we need to compute the pressure. Utilizing that $\nabla \cdot \mathbf{q} = r$ we can write (2.6) as

$$-\nabla(k\lambda \nabla p) = r. \tag{2.10}$$

The mathematical model we need to be able to solve is thus established through Equations (2.9) and (2.10). To solve this coupled system operator splitting is used. Since the FMS is an improved solver of the transport equation, we will assume that there exists an efficient solver for the pressure equation (2.10), and focus on the transport equation.

Properties of the transport equation

The hyperbolic nature of Equation (2.9) ensures a well-defined domain of dependence which is essential to the FMS. In particular, the directional derivative $\mathbf{q} \cdot \nabla f(s_\alpha)$ and the the fact that f is a strictly increasing function, ensures that the solution in cell K only depends on the neighbouring cells with flow into K. The total flux through cell K will then be due to the sum of the flux into K from the neighbours in the upwind direction and the flux out of K. Hence, if we already know the saturations of the upwind neighbours of K we can solve the transport equation directly. By choosing an upwind-discretization we can preserve this property and thereby reduce the global transport problem to a series of sub-problems corresponding to each cell K.

2.3 Discretization

Implicit temporal discretization

An efficient numerical solver for equation (2.9), requires a stable and efficient discretization in both time and space. In this section we develop this discretization and show how the resulting system of equations can be manipulated to obtain a faster solver for the transport equation.

A standard one-point upwind scheme is used to approximate the derivative with respect to time. Although implicit temporal discretization leads to a larger computational cost, it is preferred over an explicit version, due to its better stability

and ability to take larger time steps. The method developed by Natvig and Lie [2] provides a solver capable of solving this efficiently. We drop the phase subscripts from here on and introduce the approximated time derivative $\frac{\partial s}{\partial t}$:

$$\frac{\partial s}{\partial t} \approx \frac{s^n - s^{n-1}}{\Delta t},$$

where $s^n = s(n \cdot \Delta t)$. For the majority of the cells, the source term r will be zero, and Equation (2.9) can then be approximated through

$$\phi \frac{s^n - s^{n-1}}{\Delta t} - \mathbf{q}\nabla f(s^n) = 0. \tag{2.11}$$

Spatial discretization through the discontinous Galerkin method

We will introduce the discontinuous Galerkin method by showing four examples with increasing complexity. First, we will look at the simple case of an one-dimensional domain and a basis of constant test functions on a structured grid. In the second step we extend the domain to two-dimensions, but otherwise keep everything as in the one-dimensional case. We proceed to the third example, where we introduce an unstructured grid and discuss how this affects the choice of solver for the pressure equation. Finally, we conclude our discussion of the discretization by extending the method to consider test functions of higher order.

One-dimensional domain, constant test functions

Consider a one dimensional domain. To simplify we assume a unitary porosity, $\phi = 1$ and flow, $q = 1$. Equation (2.9) will in this case read

$$\frac{\partial s}{\partial t} + f_x(s) = 0. \tag{2.12}$$

To find a variational formulation of (2.12) we partition the domain Ω into non-overlapping structured elements $\Omega_K = \{K_i | \cup_i K_i = \Omega\}$ and multiply by an arbitrary test function, v. We then integrate by parts to obtain the weak formulation.

$$\int_K \frac{\partial s}{\partial t} \cdot v + \int_K f(s) \cdot v_x + [f(s) \cdot v]_{\partial K} = 0$$

The next step is to find a finite basis of test functions V_h. For this initial example we choose the the space of element-wise constant functions and allow these to be discontinuous across element boundaries. We denote this space by $V_h^{(0)}$. Since we allow the test functions to be discontinuous across boundaries we also need an approximate flux function $\hat{f}(s)$. We choose this to be the upwind flux $\hat{f}(s^+, s^-) = f(s^+)\max(v, 0) + f(s^-)\min(v, 0)$, in which s^+ and s^- denote the inner and outer approximations at boundaries. This is a consistent and conservative approximation, which preserves the crucial directional dependency. Our problem can

then be written:
find $s_h \in V_h^{(0)}$ such that:

$$a(s_h, v_h) = 0, \qquad v_h \in V_h^{(0)}$$
$$\text{with } a(s,v) = \int_K \frac{\partial s}{\partial t} \cdot v + \int_K f(s) \cdot v_x + \left[\hat{f}(s) \cdot v\right]_{\partial K}.$$

As $V_h^{(0)}$ consists of constant functions, the second term disappears. If we use the temporal discretization from Equation (2.11) we can write $a(s,v)$ as:

$$a(s,v) = \frac{s^n - s^{n-1}}{\Delta t} \cdot |K| + \hat{f}_{out}(s^n) - \hat{f}_{in}(s^n),$$

where $|K|$ is the length of element K. In the special case of $f(s) = s$ we get a linear system $As = b$, where the matrix A will have a diagonal pattern shown below:

$$A = \begin{bmatrix} \bullet & & & & & \\ \bullet & \bullet & & & & \\ & \bullet & \bullet & & & \\ & & \bullet & \bullet & & \\ & & & \bullet & \bullet & \\ & & & & \bullet & \bullet \end{bmatrix} \tag{2.13}$$

which can be solved by forward substitution. In the general case $f(s) \neq s$ we have a set of nonlinear equations $A(s) = b$. In this case the structure of the Jacobi matrix of $A(s)$ will be on the same form as that of A shown in (2.13).

Two dimensional structured grid, constant test functions

We now move on to consider a two-dimensional domain. Equation (2.9) then reads

$$s_t + \mathbf{q} \cdot \nabla f(s) = 0.$$

With constant test functions and numerical flux as in the one-dimensional case we find the weak formulation.
Find $s_h \in V_h^{(0)}$ such that:

$$a(s_h, v_h) = 0 \quad \forall v_h \in V_h^{(0)}$$
$$\text{with } a(s,v) = \int_K s_t \cdot v + \mathbf{q} \cdot \int_K f(s) \cdot v_x + \mathbf{q} \cdot \int_{\partial K} f(s) v.$$

As in the one-dimensional case the second integrand is zero, and thus this term disappears. Since we have moved to two dimensions this formulation will not automatically lead to a lower triangular structure. Each element can be dependent of any of its four neighbours. Based on how we number the elements we get a system

of nonlinear equations $A(s) = b$, where the Jacobi J_A has a structure similar to the one below.

$$J_A = \begin{bmatrix} \bullet & & & & & & & \\ \bullet & \bullet & & \bullet & & & & \\ & \bullet & \bullet & & & & & \\ \bullet & & & \bullet & & & & \\ & & & \bullet & \bullet & & \bullet & \\ & & \bullet & & \bullet & \bullet & & \\ & & & \bullet & & & \bullet & \\ & & & & & \bullet & \bullet & \\ & & & & \bullet & & \bullet & \bullet \end{bmatrix}$$

This system can be rearranged by finding a permutation P such that PJ_AP^T attains a lower triangular structure. The permutation P can be found by a topological ordering of the graph of elements, in which the direction of the flux is viewed as edges. In the above example a topological sort gives a permutation P that yields:

$$PJ_AP^T = \begin{bmatrix} \bullet & & & & & & & \\ \bullet & \bullet & & & & & & \\ & \bullet & \bullet & & & & & \\ & & \bullet & \bullet & & & & \\ & \bullet & & \bullet & \bullet & & & \\ \bullet & & & & \bullet & \bullet & & \\ & & & & & \bullet & \bullet & \\ & & & & \bullet & & \bullet & \bullet \\ & & & \bullet & & & \bullet & \bullet \end{bmatrix}$$

The system can now be solved by forward substitution, where a scalar nonlinear equation is solved for each step. Note that we do not need to assemble any permutation matrices with this technique. The matrices shown above are presented for illustration purposes only. With a simple traversal of the elements, in the order given by the topological sort, we can be sure that we know the dependencies of element K when processing it.

Two dimensional unstructured grid, constant test functions

So far we have looked at a domain divided into finite elements in a structured matter. This is not always practical. When dealing with heterogeneous domains, e.g. large variations in material properties across the domain, one often opts to adapt or refine the grid to fit the underlying geometry, and thus obtain a higher numerical accuracy. This leads to unstructured grids. In the field of reservoir simulation variations in geological properties and well placements, among other things, can motivate the need for unstructured grids.

Introducing unstructured grids adds a complication to our problem which requires special attention. So far we have assumed that there exists an efficient and accurate solver for the pressure equation (2.10). The most common solver for this is the

5-point scheme. However, when introducing unstructured grids we can no longer be sure that the 5-point scheme is convergent. To be sure of convergence we apply multi-point flux approximation methods. These are not in the scope of this thesis and we refer the reader to [15, 16] for an overview. Whereas 5-point schemes give a monotonic flux, there is no guarantee of this when using multi-point flux approximation methods. In the reservoir simulation setting, a non-monotonic flux means circular flow in the domain. Circular flow corresponds to several mutually dependent elements. The resulting system of equations can not be transformed to a strictly lower triangular structure and solved by forward substitution. Instead, the circular flow results in a block-triangular system, with irreducible blocks which needs to be solved using a suitable method.

The algorithm used to obtain a topological ordering now needs to be able to handle cycles in the graph. An example of such an algorithm is Tarjan's algorithm [17].. For every cycle discovered by the algorithm a *supernode* is created, which represents all of the nodes in the cycle. The graph is returned in topological order.

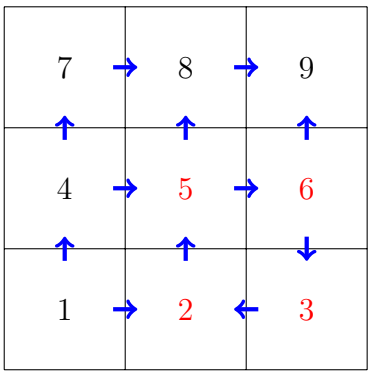
An example showing matrix structures derived from a grid containing cycles is presented in Figure 2.4. When a topological ordering is established we can traverse the sorted graph as we did in the previous examples, solving each cell sequentially. When we encounter a supernode, we compute the values in the mutually dependent elements using a suited nonlinear solver.
The resulting algorithm is shown in Algorithm 1.

Algorithm 1 Solving the coupled system containing cycles using Tarjan's algortihm.

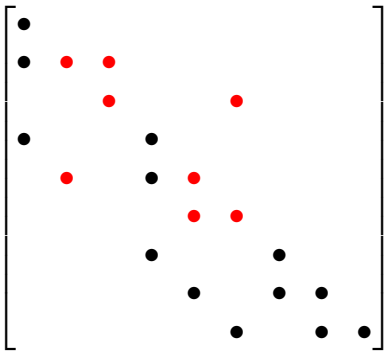
```
Input: Graph G = (V, E) representing the numerical grid
Sort G using Tarjan's algorithm.
for Vertexes v ∈ V in sorted order do
    if v is a supervertex then
        Solve the group of elements with suitable solver
    else
        Solve v with information from previously solved vertices
    end if
end for
```

Two dimensional unstructured grid, non constant test functions

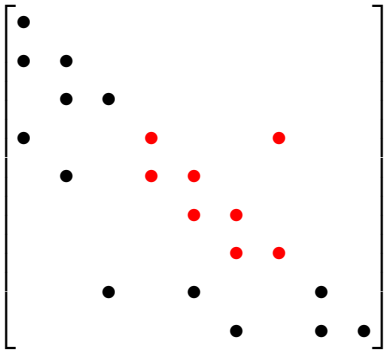
To complete our discussion of discontinuous Galerkin methods we are going to consider one last case. In the previous examples the test functions have always been constant inside each element. We now consider test functions in the polynomial space $\mathbb{Q}^n = \text{span}\{x^p y^q : 0 \leq p, q \leq n\}$. Denoting our space of test functions by $V_h^{(n)} = \{\phi : \phi|_K \in \mathbb{Q}^n\}$. To form a basis for test functions in this space, we use products of Legendre polynomials $L_k(\xi, \eta) = l_r(\xi) l_s(\eta)$. We are thus seeking an



a: Grid showing circular flow.



b: Matrix structure corresponding to the grid
.



c: Matrix structure of the grid after a topological sort. Note that the elements in the cycle are grouped together.

Figure 2.4: An example showing circular flow (a) (indicated with red), the corresponding matrix structure (b) and a topological sorting which groups the cycles together (c).

approximate solution

$$s_h^i(x,y) = \sum_{k=0}^{N} U_{ik} L_k \left(\frac{2(x-x_i)}{\Delta x^i}, \frac{2(y-y_i)}{\Delta y^i} \right),$$

where N is the number of basis functions, (x^i, y^i) is the center of element K_i, and $\{U_{ik}\}$ are the unknown coefficients to be determined. The constant functions we have used so far correspond to $V_h^{(0)}$ and are a first-order accurate scheme. $V_h^{(1)}$ corresponds to a second-order accurate scheme, and so on. We use the notation $dG(n)$ to denote a discontinuous Galerkin scheme of accuracy order $n+1$. The number of unknowns per element for a $dG(n)$ approximation is $(n+1)^2$. This means that each element will correspond to an $(n+1)^2 \times (n+1)^2$ block on the diagonal of our system matrix. The blocks on the diagonal of $A(s)$ will now consist of a system of mutually dependent variables, dependent on the unknowns in the upwind direction of the element K_i. We can thus use the same procedure as in the $dG(0)$-case, with an exception; we now have a block per element that have to be solved using a suited non-linear solver.
If we take $dG(1)$ as an example we will have test functions :

$$\phi_0 = a_0, \quad \phi_1 = a_1 x, \quad \phi_2 = a_2 y, \quad \phi_3 = a_3 xy.$$

In the matrix structure presented in Figure 2.4 each dot will now represent a 4×4 block of unknowns which has to be solved with a nonlinear solver.

The ideas presented in this and the previous sections can easily be extended to three dimensions. The space $\mathbb{Q}^n$ then has dimensions $(n+1)^3$. However, if we instead use the space $\mathbb{P}^n = \{x^p y^q z^r : 0 \leq p+q+r \leq n\}$, we reduce the number of unknowns per element to $((n+1)(n+2)(n+3))/6$, while still obtaining a valid basis.

2.4 Reordering

In Section 2.3 we showed how the matrices stemming from the transport equation can be permuted into a block triangular structure, allowing sequential solving of the cells in the mesh. Finding permutations can be costly, and the strength of the FMS lies in the use of an efficient topological sorting of the finite element mesh to find the permutation. By representing the finite element mesh as a graph we can use graph algorithms to obtain a topological sorting. Towards that end we now introduce some definitions that we need when working with graph problems.

Definitions

Formally stated a graph $G = (V, E)$ is a duple consisting of vertices V and edges E, where an edge is a pair of vertices specifying a relation between the vertices. When the relations have information about the direction of the relationship, it is called a directed graph, otherwise we call it an undirected graph. To represent the finite element mesh as a graph we consider each element as a vertex. Two adjacent

vertices, u and v, with flow going from u across their mutual border into v, are represented as an edge from u to v in the graph.

A topological sorting of a directed graph is an ordering of the vertices such that no edges point backwards in the ordering. In other words, there is no edge which has a start vertex later in the ordering than its terminal vertex. In Definition 1 we have stated a formal definition of a topological ordering.

Definition 1 *Let $G = (V, E)$ be a directed graph and $\hat{V} = \{v_1, v_2, \dots v_n\}$ an ordering of the vertices. If for each edge $(v_i, v_j) \in E, v_i, v_j \in \hat{V}$ we have that $i < j$ we call the ordering $\hat{V}$ a topological ordering.*

Figure 2.5 shows an example of a 3×3 mesh, its representation as a graph and a topological ordering of that graph.

Definition 2 *A path p is an ordering of vertices $p = \{u_1, u_2, \dots, u_n\}$ such that there exist edges $\{(u_1, u_2), (u_2, u_3), \dots (u_{n-2}, u_{n-1}), (u_{n-1}, u_n)\}$.*

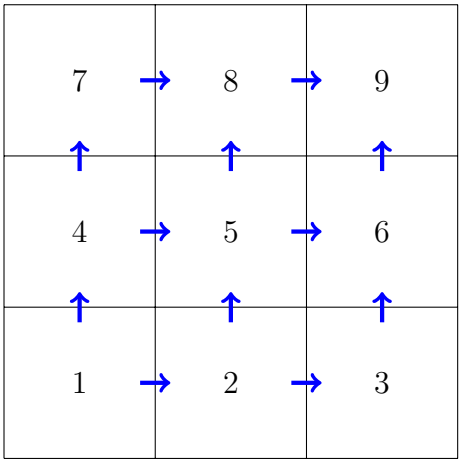
Straightforward topological sorting fails if there exists one or more *strongly connected components* (SCC) in the graph we are trying to sort. An SCC is a group of vertices where all vertices are reachable from all the other vertices in the group. In Figure 2.6a we show an example of a graph with a strongly connected component.

Definition 3 *Let G be a directed graph. If, for any pair of vertices (u, v), there exists paths $p_1 : u \to v$ and $p_2 : v \to u$, we say that the graph is strongly connected.*

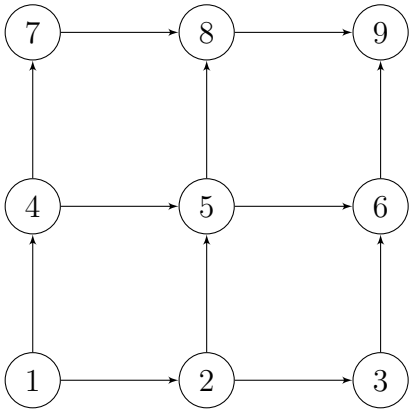
If a strongly connected component exists in a graph, a normal topological sorting is not possible to obtain. As described in Section 2.3, topological ordering is crucial for the FMS. The definite serial algorithm for obtaining a topological ordering for graphs with strongly connected components is Tarjan's algorithm[17]. Built around depth-first search, Tarjan's algorithm marks vertices with a discovery number according to when it was discovered, and uses these numbers to identify vertices which are part of a strongly connected component. Once a strongly connected component is discovered, it is collapsed into a super-vertex, see Figure 2.6b for an illustration. In the resulting topological ordering the super-vertices will contain information about their internal vertices and edges. It is thus possible to obtain a topological ordering without losing any of the information in the graph. See Appendix A for pseudo code of Tarjan's algorithm.

Serial FMS

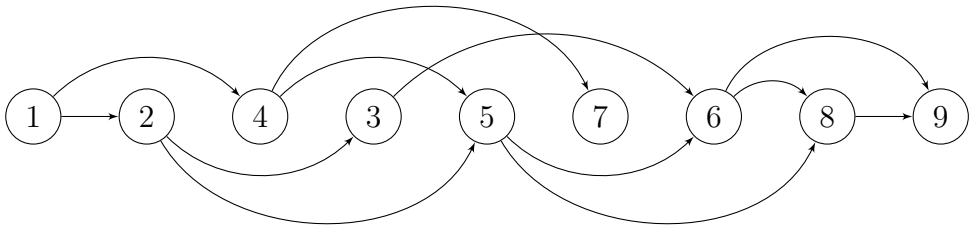
The Fast Multiphase Solver was developed by Natvig, Lie and co-workers [3] at SINTEF and collaborators from other institutions. The program uses the Matlab Reservoir Simulation Toolbox [18] to set up a geometry with corresponding physical quantities described and injection and production wells in place. The reordering procedure and the sequential solver is implemented in C, and MEX is used to integrate with MATLAB. Before the topological solving of the elements can be started,



(a) 3 × 3 example grid. Flow across the cell interfaces is represented with blue arrows.

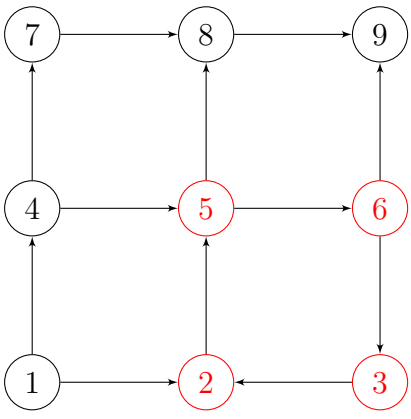


(b) Example grid from Figure 2.5a represented as a directed graph.

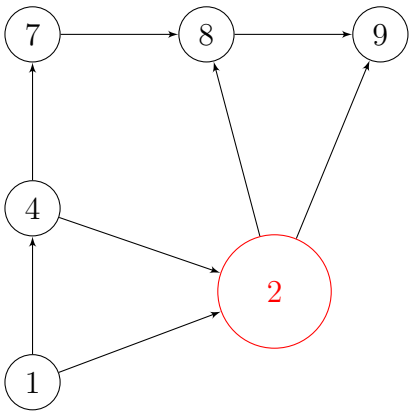


(c) A valid topological ordering of the grid in figure 2.5b

Figure 2.5: An example of a transport problem and its topological sorting.



(a) The vertices coloured red constitutes a strongly connected component.



(b) The strongly connected component from 2.6a collapsed into a super-vertex.

Figure 2.6: Example of a graph with a SCC and how it is collapsed. The sequence 1,4,2,7,8,9 is a valid topological ordering after the collapse of vertices 2,3,5,6 into one super-vertex.

the pressure equation (2.10) is solved. With a pressure field in place, flow directions can be used to represent the numerical grid as a graph. Then Tarjan's algorithm is used to obtain a topological ordering of the resulting graph, collapsing strongly connected components as it goes. Once a topological ordering is established, its elements are sequentially traversed, solving the transport equation (2.9) for each element as it goes. Whenever a super-vertex is encountered, a suitable solver is used to solve the system of equations corresponding to its contained vertices.

2.5 Parallel computing

So far we have presented the problem at hand and its mathematical formulation. Furthermore, we have introduced a fast serial implementation [3]. Although this implementation has shown very good timing results, and outperforms similar solvers by several orders of magnitude, our focus remains on exploiting parallel computation power to further improve the performance of the solver. Before we move on to describing existing parallel algorithms that we propose as candidates for parallelizing the FMS, we introduce some useful notions concerning parallel computation that we will refer to in our later discussions. For a more detailed introduction of parallel programming we refer the reader to [19].

Computation in parallel has such a wide definition that a lot of different approaches fall into this category. A lot of different hardware has been designed with parallel capabilities and a variety of programming techniques exists for exploiting these. Which hardware and programming model one chooses depends on the problem at hand.

Flynn's taxonomy

Categorisation of different computer architectures becomes necessary when working with parallel algorithms. Flynn's taxonomy [20], proposed in 1972, labels machines into one of four categories and is widely used today. It now incorporates two more definitions than when it was first introduced, to better encompass all models of parallel architectures.

SISD Single Instruction Single Data stream. Serial computers with a single stream of instructions working on a single data stream. This architecture exploits no parallelism, essentially doing "one thing at a time". Traditional serial processors fall into this category.

SIMD Single Instruction Multiple Data stream. Exploits parallelism in the data by issuing the same instruction to multiple data at a time. Array processors and GPU's fall into this category. Most modern processors exploit this in some way.

MISD Multiple Instruction Single Data stream. Several different instructions are issued in parallel on the same stream of data. Usually used for systems de-

signed for high fault tolerance. An uncommon architecture unsuitable for high performance computing.

MIMD Multiple Instruction Multiple Data stream. Parallel instructions are issued to multiple streams of data. Distributed systems, either with shared memory or distributed memory, fall into this category. Modern multi-core processors also fall into this category.

Further division of the MIMD category

This thesis focuses on algorithms meant for MIMD architectures, and we will therefore elaborate with a further distinction within this category. Note that these distinctions do not imply anything about the hardware architecture, but rather about how the software is designed.

SPMD Single Program Multiple Data. Used to categorize implementations for which the same program is executed on different data for different processors. Most distributed programs are written in this way.

MPMD Multiple Program Multiple Data. Less common way to design distributed programs. At least two different programs are run on different processors, which then does different tasks for the program. An approach can be to have one or more "manager processors" which control the flow of the program, and distribute tasks to the remaining processors, which run a different program.

Within the MIMD category there are two main ways to handle communication between processors, either by *shared memory* or *distributed memory*

Shared memory

Shared memory machines have several processors working on the same memory. All processors can access any memory location, and explicit message passing is abundant. Avoiding message passing makes this architecture easier to program and usually faster too. Downsides are vulnerability to race-conditions and bad scalability. Modern multi-core processors usually have shared memory.

Distributed memory

Clusters of processors where each processor has its own private memory location are called distributed memory machines. These machines, or clusters of machines, rely on explicit message passing to achieve parallelism. Such machines are organized into nodes, which consists of one or several processors. All the nodes are connected through a high bandwidth, low latency network which facilities the message passing between nodes. Most distributed machines/clusters today also incorporates the advantages of shared memory by having nodes of 4-16 processors which share memory.

Although more tedious to program they are extremely scalable. This allows such machines to operate on data too big to fit on shared memory machines. Programs,

more often than not, have to be completely rewritten to work on distributed memory systems. This facilitates the need for skilled programmers, and a significant improvement has to be plausible for the effort to be worth it.

Measuring performance of parallel algorithms

High performance computing is motivated by the need for faster methods of handling increasingly large problems. To measure how successful a parallel algorithm is, good metrics are needed. Measuring performance of a parallel algorithm based on complexity alone is unsuitable. Parallel programs have the advantage of more resources than their serial counterparts, and can thus do more complex work in the same amount of time.

The primary measure of parallel performance is the *speed-up*. By comparing the time T_s the best serial algorithm needs to solve a given problem, to the time T_p the parallel computer with p processors needs to solve the same problem, the speed-up S_p can be found by the following formula:

$$S_p = \frac{T_s}{T_p}.$$

In other words, it answers the question: how much faster is the parallel algorithm, when run on p processors, than the best serial algorithm? The ideal case is if the parallel implementation is able to execute the problem p times faster than the serial version, this fully utilizing all of the extra resources. This is called *linear speed-up*.

Parallel algorithms normally require a certain degree of communication between the processors. Because of this close to linear speed-up is usually only possible when the problem size is big enough to diminish communication costs in the total run-time. As a consequence, parallel programs only benefit from added computing resources up to a certain point. Since the number of processors are usually limited, it is interesting to note how well a parallel algorithm utilizes the resources available to it. By dividing the speed-up on the number of processors we get a number between 0 and 1, telling us how well the parallel algorithm has made use of the extra processors. This number is called the *efficiency*. Note that linear speed-up corresponds to an efficiency of 1.

Most practical problems consist of different parts that can be solved individually with specific algorithms. If we can parallelize all parts perfectly and get a linear speed-up we have achieved perfect parallelism. However, in many cases there are parts of a problem that are not possible to execute in parallel, or there exists a specific order in which the different parts have to be executed. Amdahl [21] showed that the possible speed-up S_n of a program with a serial portion of P_s is limited by

$$S_p = \frac{1}{P_s + \frac{1}{p}(1 - P_s)}, \tag{2.14}$$

where p is the number of processors. Observe that as the number of processors goes towards infinity, the speed-up goes towards $\frac{1}{P_s}$. Analysing a problem to identify

inherently serial parts can give a good indication of how much speed-up is to be expected from parallelism.

Amdahl's law puts an unfortunate limitation on the possibilities of parallel programming. However, it has been pointed out that the assumption of a fixed problem size is a weakness in Amdahl's law. Gustafson [22] argued that programmers and scientists decide problem sizes depending on how much computing power they have available. He formulated a law, called Gustafson's law: a program with serial portion P_s can achieve a speed-up of

$$S_p = p - P_s(p - 1).$$

The law proposes that parallel computing is not only about solving existing problems faster, but just as much about being able to solve bigger problems in the same amount of time as before. This makes the limitations of Amdahl's law less severe.

Section 3

Towards a parallel solver

Our aim is to develop a parallel method for solving the transport equation based on reordering, using the same approach as the FMS. Preliminary tests of the serial FMS [8] indicate that the reordering of the elements usually constitutes a small part of the total execution time. Test done on systems using constant test functions show a fractio of 8-15% of the total run time [8]. Potential speed-up is thus limited to 1,1-1,2, according to Amdahl's law (2.14). Systems using test functions of higher order will In other words, if the reordering of the elements is the only thing we are able to run in parallel, we might as well stick to the serial version. The sequential solving of the elements amounted to over 50% of the run time in all the preliminary tests, which means that with no parallelism on this part, we can at best hope for a speed up of 2, according to Amdahl's law. For our algorithm to be successful we therefore have to achieve parallelism on both the topological sorting and the sequential solving.

In Section 2.4 we described how the FMS by Natvig and Lie [2] is able to numerically solve the transport equation with a significant improvement in computational efficiency. The ability to obtain a topological ordering of the elements is of vital importance to their method. In the sequential version this was done using Tarjan's algorithm. Unfortunately Tarjan's algorithm will have to be replaced in a parallel implementation. It is based on depth-first search, which is probably impossible to parallelize [23].

In the current design of the FMS, there is no room for parallelism in the sequential solver. Tarjan's algorithm returns a topologically sorted graph, but gives no information of potentially independent instances. We therefore search for a new approach to the entire FMS. Ideally we would like an algorithm that sorts the graph in parallel and immediately solves the transport equation for vertices whose order has been determined. Since this would require a new approach to both the sorting and the solving, we have approached the problem through finding a new topological solver capable of solving elements simultaneously as sorting the graph, and thus achieving a high degree of parallelism.

Usually there will exist more than one valid topological sort of a graph, implicating that there are instances in the graph whose order does not matter. This opens the

problem up for parallelism. In other words, a graph with sub graphs whose order are irrelevant can be solved in parallel if there are no dependence between them.

3.1 Existing research

The need for fast graph algorithms is evident in a wide range of fields. Modelling of social networks, finite-element meshes, and transport networks are just some examples of relevant fields. Whenever problem sizes get large enough, the use of parallel computing to keep computation times reasonable can become a necessity. Variations of our problem of finding a topological sorting of a directed graph in parallel have already been the subject of some research. Gazit et al. [24, 25] described an algorithm for finding strongly connected components using matrix multiplication, which later was improved by Cole and Vishkin [26], and Amato [27]. Although these algorithms report $O(\log^2 n)$ time, they require an impractical $O(n^{2.376})$ processors. Kao [28] presents an algorithm using $O(n/\log n)$ processors in $O(\log^3 n)$ time for a planar directed graphs.

Bader developed a distributed algorithm for detecting strongly connected components in planar directed graphs [10]. This algorithm, though lacking some important features, is appealing first and foremost because of its highly scalable domain decomposition. If successfully adapted to incorporate the needs of the FMS this is a viable candidate for our parallel FMS.

Fleischer et al. [9] have devised a simple but effective algorithm that finds the topological sorting of a graph containing strongly connected components. The simplicity of this algorithm combined with its adequate treatment of the strongly connected components has made the findings relevant. McLendon et al. [29] developed [9] furter for an implementation on a radiation transport problem. Algorithms by Orzan and Barnat also builds on the work done by Fleischer [9] and [29]. Orzan [30] has devised an algorithm for identifying strongly connected components in parallel used for model checking. Whereas Barnat et al. [31] have modified the latter to work on GPUs. Experiments on GPUs show that it can outperform Tarjan's algorithm by a magnitude of 40 on sufficiently large problem sizes. To limit the scope of this thesis, we have not looked into the use of GPUs, but Barnat's results indicate that this could indeed be the way to go.

The rest of this thesis studies the algorithms developed by Bader [10] and Fleischer et al. [9]. The following sections introduce the algorithms in depth, and discuss possible alterations to meet the FMS' need. After this a detailed account of our own implementation of these algorithms follows. It is our hope that this will help anyone wishing to continue this work.

3.2 Divide and Conquer Strong Components

We now introduce the algorithm called Divide and Conquer Strong Components (DCSC), developed by Fleischer et al. [9]. DCSC is an algorithm that finds the topological sort of a directed graph in parallel, with treatment of strongly connected components. The algorithm is built around two lemmas. Simple and powerful, these lemmas have been used in other research, e.g., the CUDA version proposed in [31]. Furthermore, the algorithm is motivated by finite element meshes, making it a natural candidate for parallelization of the FMS [3]. McLendon et al. [29] have implemented DCSC on a radiation transport problem, which has structural similarities with the porous medium problem. The implementation achieved very good results and in some cases they report linear speed up. Their modified version of the DCSC, fittingly named ModifiedDCSC, includes trimming steps that remove vertices without incoming edges, further improving the efficiency of the algorithm.

Our study includes explanations of Fleischer's [9] original algorithm and McLendon's [29] modified version, as well as a performance study based on our own implementation of the original algorithm.

Definitions

The key to the DCSC is two lemmas presented in [9] which we will repeat here. To do so we have to introduce some definitions, which will help us structure the explanation.

Recall from Section 2.4 that a dipath from v to u is a sequence of edges p such that $p = [(v, v_1), (v_1, v_2), \ldots, (v_n, u)]$. We say that u is *reachable* from v if there exists a dipath from v to u. For a graph $G = (V, E)$ and a v in V, we say that the descendants of v in G, denoted by $Desc(G, v)$, is the set of all vertices in G which are reachable from v. Correspondingly, we say that the predecessors of v in G, denoted by $Pred(G, v)$, is the set of all vertices in G which v is reachable from. The set of vertices in G that belong to neither the predecessors nor the descendants of v is called the remainder of G, denoted by $Rem(G, v)$. The set of all strongly connected components of G is denoted by $SCC(G)$. A specific strongly connected component can be defined through any one of the vertices contained in it, and is denoted by $SCC(G, v)$.

With this established we can state the lemmas proved by Fleischer [9].

Lemma 1 *Let $G = (V, E)$ be a directed graph, with $v \in V$ a vertex in G. Then*

$$Desc(G, v) \cap Pred(G, v) = SCC(G, v).$$

Lemma 1 allows us to reduce the problem of finding strongly connected components to finding unions of descendants and predecessors for the vertices in the graph. Alone, this lemma is not very useful, since finding descendant and predecessor sets

for all vertices would be far slower than alternative algorithms for finding strongly connected components. However, when combined with Lemma 2 we get a very powerful combination:

Lemma 2 *Let G be a graph with a vertex v. Any strongly connected component of G is a subset of Desc(G,v), of Pred(G,v), or of Rem(G,v).*

The problem of finding descendant and predecessor set can now be limited to the three separate instances described in Lemma 2, remarkably reducing the problem size. Here the possibility of parallel recursion also arises, which gives this algorithm its power.

To facilitate the topological sort, we need one more fact.

Lemma 3 *For a directed graph G there exists a numbering π of the vertices from 1 to n for for which the following is true. All elements $u \in Pred(G, v) \setminus Desc(G, v)$ satisfy $\pi(u) < \pi(v)$; and all elements $u \in Desc(G, v) \setminus Pred(G, v)$ satisfy $\pi(u) > \pi(v)$.*

Through Lemma 3 we are able to not only find the strongly connected components, but also topologically sort them. For the proofs of the lemmas we refer the reader to the original article by Fleischer [9].

DCSC explained

With these lemmas established we can explain the DCSC in detail. First a pivot vertex v is chosen at random from the graph. The rest of the graph is sorted into three subsets: descendants of v, predecessors of v and the remainder of the graph. Figure 3.1 shows how a 5×5 grid containing an SSC is divided into predecessor and descendant sets. Due to Lemma 1, vertices belonging to both the predecessors and descendants of v constitute a strongly connected component.

Extracting the union of the descendant and predecessor sets yields one out of two cases. If the pivot vertex alone v makes up the union, we have the trivial case that v is not part of a SCC. If v is part of an SCC we will get all the vertices in the SCC as the union, due to Lemma 1.

In any case, we save the union and then call the algorithm recursively on the three subsets. If the routine is called on an empty graph, it will return immediately. This ensures termination of the algorithm when all vertices have been checked. Pseudo code for DCSC can be found in Algorithm 2.
Divide and Conquer Strong Connect is open to parallelism in two ways. First, the three recursive calls stemming from each identified SCC are fully independent problems, which can be solved on different processors. Secondly, the traversal of the graph to identify predecessors and descendants are open to parallelism [32], however this comes at an extra factor of $\log n$ in run time.

Algorithm 2 The Divide-and-Conquer Strong Connect algorithm, as developed by Fleischer et al.

function DCSC(G)
  if G is empty **then**
    return
  end if
  v = random vertex from G
  $SCC = Pred(G, v) \cap Desc(G, v)$
  Output SCC
  DCSC($Pred(G, v) \backslash SCC$)
  DCSC($Desc(G, v) \backslash SCC$)
  DCSC($G \backslash (Pred(G, v) \cup Desc(G, v))$)
end function

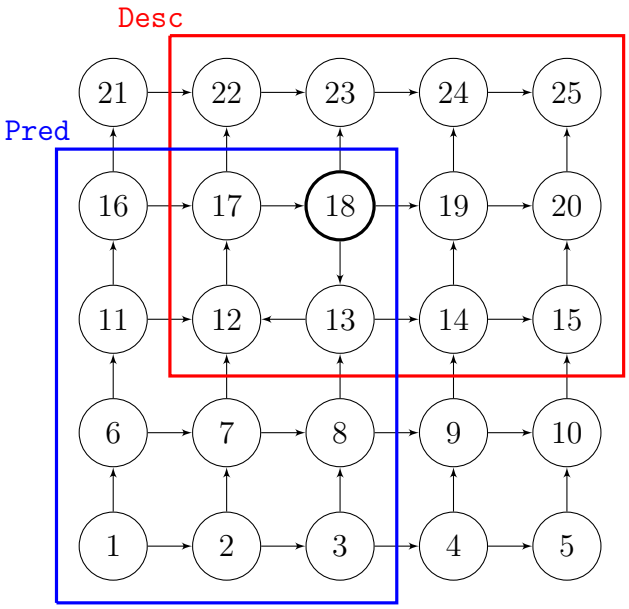


Figure 3.1: Example graph showing predecessor and descendant sets. Pivot vertex is 18.

Modified DCSC

McLendon et al. [29] modified the DCSC by adding a trimming step. Trimming of vertices is especially efficient in graphs where a small portion of the vertices are contained in strongly connected components, which is often the case with reservoir simulation, as mentioned in [2]. Another advantage of the trimming step, is that it allows us to compute the solution in trimmed vertices immediately after having removed them, allowing for further parallelization.

Before the pivot vertex is chosen, the graph is traversed, looking for vertices with no incoming edges. From the definition of a SCC we can conclude that these vertices are not part of any SCCs, and can safely be removed. Analogously we can remove any edges with no outgoing edges. This reduces the problem size, and can have large impacts on run time, especially for problem instances where vertices included in SCCs constitutes a small portion of the total graph.

3.3 Cycle detection due to Bader

The next algorithm we will study is an algorithm due to Bader [10]. This algorithm relies on domain decomposition of a large directed graph, and does a local depth-first search of each sub-domain before merging graphs iteratively to determine whether a cycle exists. It has achieved linear speed-up in previous implementations, and scales very well with respect to both graph size and number of processors.

We have chosen this particular algorithm as a main candidate for developing a parallel version of the FMS because of its use of domain decomposition, as well as its scalable speed-up. Our hope is that we will be able to adapt it to also treat cycles, and sort the graph topologically. If this is successful, it will be perfectly suited to solve the transport equation in parallel with reordering of elements. Before introducing the algorithm, we state some definitions that will make things easier to explain. We also give a brief account of our graph representation and how we partition the graph.

Definitions

For the distributed graphs we define G_z to be the local sub graph assigned to processor p_z, with vertices V_z and edges E_z. Let $f(v_z)$ be a function mapping each vertex $v \in V$ to a processor p_z. All edges can now be categorize as either *local arcs* or *trans arcs*. A *local arc* is an edge that has both its initial and terminal vertex on the same processor, $f(v_i) = f(v_t)$, whereas a *trans arc* has the initial and terminal vertices on different processors, $f(v_i) \neq f(v_t)$. In Figure 3.2 we show an example of trans-arcs and local arcs using 18 vertices distributed across two processors.

The second phase of Bader's algorithm builds a new digraph, called an *express graph*, on each processor p_z. These graphs will hold *exit vertices* (one for each

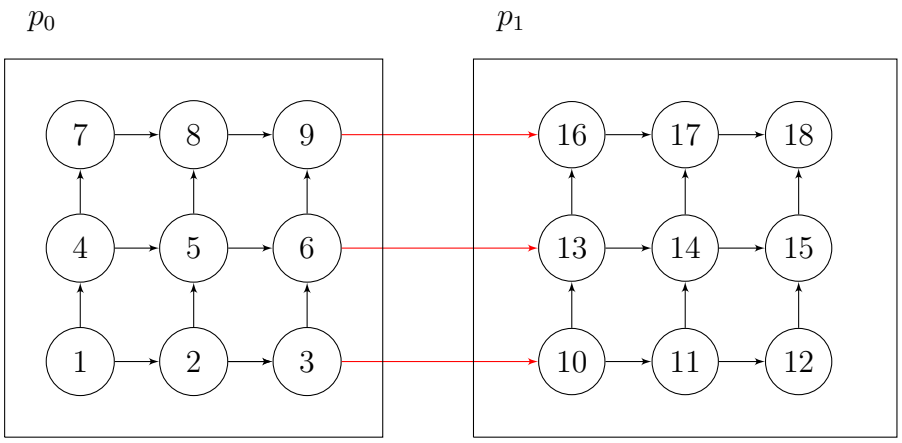


Figure 3.2: Example configuration of 18 nodes distributed across two processors. Black arrows represent local arcs, red arrows represent trans arcs.

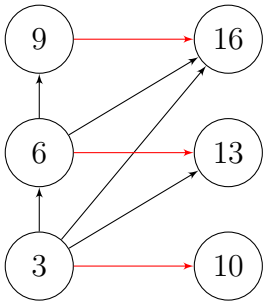


Figure 3.3: The express-graph of the example shown in Figure 3.2. Red arrows represent trans arcs, black arrows represent express arcs.

initial-vertex), and *entrance vertices* (one for each terminal vertex) with respect to the processor p_z. The arcs in the express-graph can be categorised as either *trans arcs*, corresponding to trans arcs in the original graph, or *express arcs* with initial and terminal vertices corresponding to exit- and entrance-vertices. The express graph is explained in detail in Section 3.3 An example of an express graph is shown in Figure 3.3.

Bader's algortihm

Once the graph has been partioned and distributed we can apply Bader's algorithm. The algorithm consists of three steps. A discovery phase, in which the sub graphs are searched for local cycles using depth first search; an express phase, where edges spanning across processor boundaries are indentified and communicated; and a merge phase, where two and two sub-graphs are merged together while looking for cycles. If a cycle is found, the algorithm halts, otherwise it runs until the graph is again located on the root processor and it has been determined that no cycles exist.

Discovery phase

This phase has two goals: to find any local cycles, and to identify edges spanning across subgraphs. In the first case the algorithm halts, and in the latter case the edge is stored as a so-called trans arcs, to be communicated at the end of the phase.

The discovery phase consists of a recursive search of the vertices on each processor. This is done by using a color-coding scheme: all vertices are first coded as white (not visited). Then all of the vertices are visited in turn.
When a vertex v is visited, the following steps are conducted:

1. v is color coded red, indicating that it is currently being visited.
2. All of the neighbouring vertices of v are traversed.
    - If a neighbour is not yet visited (color-coded white), a recursive visit is made immediately. Any descendants found in this visit are added to the descendants of v.
    - If a neighbour has already been visited previously, its descendants are added to the descendants of v.
    - If a neighbour is color coded red, it means that a predecessor of v is also a descendant of v, implying a cycle. The algorithm halts.
    - If a neighbour of v does not belong to this sub graph, a special trans-arc is saved for later communication, and the neighbour is also saved as a descendant.
3. After all neighbours have been checked, v is color coded green if it has no descendants and black if it has descendants. The method returns the descendants of v.

After all the vertices have been visited, the trans-arcs are exchanged with the neighbouring processes. Since this thesis only considers Cartesian 2D domains, each processor needs to do at most four exchanges (north, south, west, east).

Express-graph phase

During this phase the express-graph is constructed. The express-graph is a compact data-structure designed to hold information about the trans-arcs, spanning processor boundaries, and their dependants. As described in the above definitions the express-graph consists of an exit vertex wherever there exists an edge to a node contained on another processor. This exit vertex points two an entrance vertex, representing the node on which the edge enters another process. In addition to the trans-arcs, express-arcs are added wherever there exist a dipath between an exit vertex and an entrance vertex. See Figure 3.3 for an illustration.

Merge phase

The last phase iteratively merges pairs of subgraphs until one of two things happen:

- a cycle is found, in which case the algorithm halts.
- the entire graph is left on the root process, in which case we have determined that no cycles exists.

Which processors merge in what order is in the original implementation governed by a bit-wise manipulation of processor ranks. This approach makes sense for a graph that is not mapped over a specific geometry. For our problem, an approach using the spatial locality in the Cartesian mesh that the processors are organized in could prove more sensible, as there will only be trans-arcs between cells that lie in adjacent processors in this grid.

Merging of two sub-graphs $ExG1$ and $ExG2$ starts by creating a new express-graph $ExG0$ from the union of vertices in the two existing express-graphs. All express-arcs are also transferred directly to this new express-graph. Next, all the trans-arcs in the two original express-graphs are traversed. For each initial trans-arc v_i we have two possibilities:

$v_t \notin V(ExG0)$ This means that the trans-arc ends on another process. The corresponding trans-arc is then transferred to $ExG0$.

$v_t \in V(ExG0)$ This means the trans-arc ends on one of the two processes being merged. We now have to do one of the following:

If an express-arc exist between the two: a cycle has been found. The algorithm halts.

If an express-arc does not exist between the two: Create express-arcs between all pairs of predecessors and descendants of v_i and v_t, then remove v_i and v_t from $ExG0$.

The merge phase ends when either all the express-graphs have been merged into one, or one of the processors discovers a cycle, in which case the algorithm terminates.

Section 4

Implementation

The following section describes choices we have made when implementing the two algorithms described in Section 3. Fleischer's divide-and-conquer approach [9] has been implemented and tested on a shared memory architecture. Bader's algorithm for identifying strongly connected components [10] has been implemented using an explicit message passing approach. This section describes technicalities around our work towards this, with the aim that someone wishing to continue this work will have an easier start than we did.

4.1 Parallel software

OpenMP

OpenMP is an API for C, C++ and Fortran built for ease of use multi-threading on shared memory architecture. OpenMP consists of compiler directive, run time library functions and environment variables. To do simple parallelization only a few are needed, but the API is powerful enough to support a wide array of functionalities, although we do not go into depth on that functionality here. However, we do take time to dwell over some key features that are of importance to our implementation. For a more thorough introduction we refer the reader to [33]. Introduced in OpenMP 3.0 [34], the task clause makes it possibility to parallelize tasks instead of the traditional thread approach. Tasks are independent instances of a program. If several tasks do not have to be performed in any specific order the possibility for parallel execution of these tasks arises. The task clause in OpenMP takes care of this for the programmer.

MPI

Message Passing Interface (MPI) is a standardized portable library for message passing. It can be used in programs written in C/C++, Fortran and Java, on both shared and distributed systems. We have used the Open-MPI implementation [35].

As with OpenMP we give no detailed account of the inner workings of MPI, but introduce some key routines. For the interested reader Pacheco gives a good in-

troduction to MPI [19]. For a more comprehensive text, see [36]. MPI_send is the basic routine for sending a message from one processor to another, MPI_recv is the corresponding receive routine. MPI is developed for SPMD programming, and the programmer has to make sure that if a processor calls a send routine, the receiving processor calls a receive routine. As MPI_send defines a unique sender and receiver of the message, it is categorized as so called one-to-one communication. MPI includes a wide range of sending routines, including one-to-all, all-to-one and all-to-all. We will not describe them all here, and refer to the above mentioned literature for further details.

4.2 Implementation of DCSC

We have implemented DCSC in C using our own data-structures. Although this was a tedious endeavour, it gave us better control over the process than if we had used an existing library. Source files for the program can be found on https://github.com/henvik/DCSC.git. In Appendix B we have included parts of the program for quick referencing while reading.

Data structures

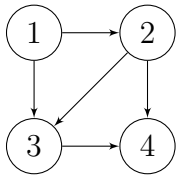
As the DCSC requires forward and backward traversals of the graph, as well as the ability to easily add and remove vertices, we chose linked list. In the implementation this consists of three separate structures that together make up the linked list.

Node is the meat and bone of the lists. Each node has an unique vertex number, identifying it. It also holds pointers to the previous and next nodes in the linked list. Note that the previous and next nodes only point to the nodes which lie adjacent in the linked list, and have nothing to do with the actual structure of the graph. To keep track of the structure of the graph, each node contains a pointer to two sets of edges: children edges and parent edges, representing edges directed out and in of the node, respectively. The children and parent edges are implemented as linked lists.

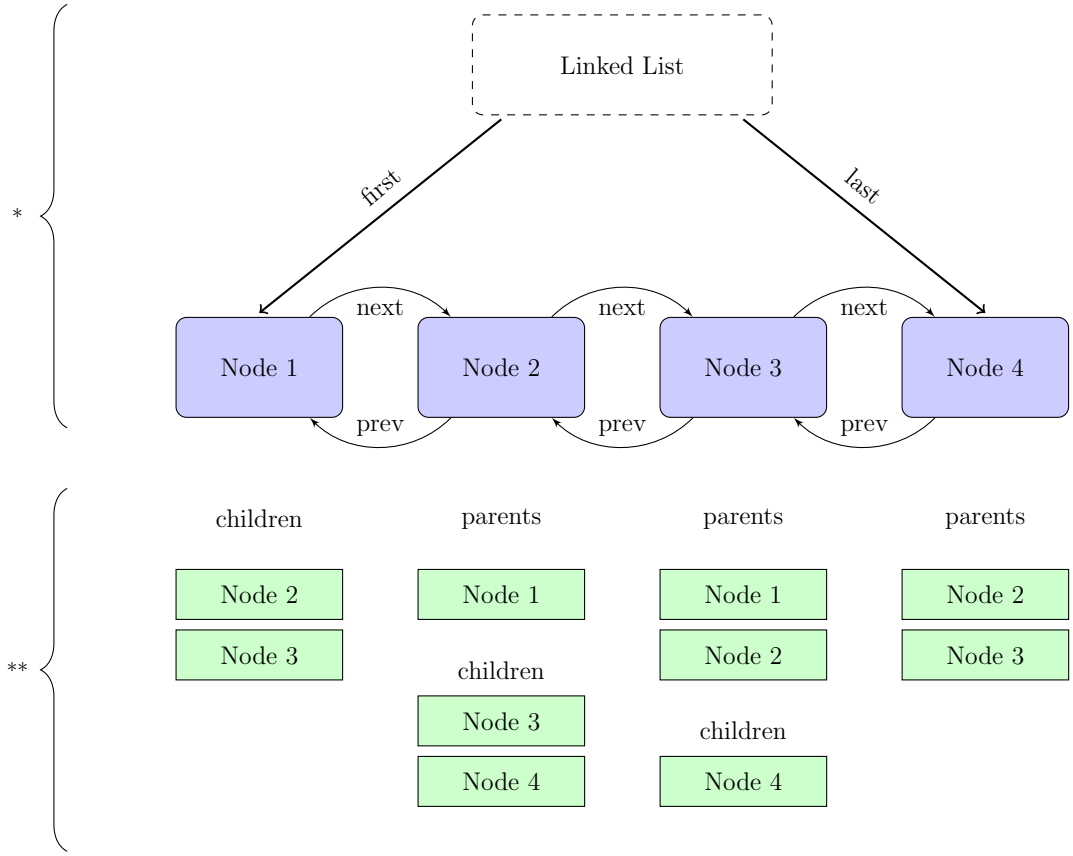
In addition, the node has three statuses that are used in the descendant and predecessor search. These statuses are pointers, which are also used for storing addresses of copies made of the node. Null pointers mean not visited yet.

Arc holds information about the destination node of the arc, in the form of a pointer to the node's memory location. It also holds a pointer to the next arc in the children/parent list. The initial vertex of the arc is implicitly saved in the node which the arc belongs to.

Linkedlist is a container for all the nodes in the list. It holds information about the number of nodes in the graph, and pointers to the first and last node in the list.



(a) Example graph.



(b) Illustration of the data structures used for implementing the DCSC. Nodes are blue and arcs are green.
* Memory structure. To store the information in the list and allow for easy insertion and removal.
** Graph structure. Represents the geometry of the graph. Allows for forward and backwards traversal.

Figure 4.1: Example graph (4.1a) and its representation in the implementation of the DCSC (4.1b).

Around these three structures, the program is built. The choice of graph representation stems from our need to efficiently traverse the graph both forwards and backwards. We also need to be able to remove and add vertices quite frequently, without a high cost. This does however, come at cost. Search and access will cost more than if we were to use a adjacency list.

Core procedures

Our implementation of Divide and Conquer Strongly Connect is build around a few core procedures.

DCSC is the main procedure. It finds a pivot vertex, randomly chosen from all of the vertices in the graph, and in turn finds the descendant and predecessor sets as wells as the union of theses The search for descendants and predecessors is done in parallel.

Once the graph is divided into three sub graphs according to descendants, predecessors, and the remainder of the graph, DCSC is called recursively on each of the three sets. To parallelize these three independent sub-problems, we use the parallel task clause in OpenMP. Available processors will then work in parallel on the tasks being created by the recursion.

When the graphs get sufficiently small, executing the recursion in parallel creates more overhead than it gains. Therefore, at a given cutoff, the recursive calls are done serially on one processor.

Find descendants traverses the descendants of a specified pivot vertex. For each descendant found it marks the descendant status of that node, indicating that this node has been identified as a descendant of the pivot. If it is the first time that descendant has been visited, it is copied into the graph of descendants. The status is also a pointer to the copy in the descendant graph. This is used when encountering a descendant that we have already visited. The status of that node is then used to create an arc to it in the descendant graph.

Find predecessors traverses the predecessors of the pivot vertex. The same procedure as in the descendant search is used to avoid adding more than one copy of each predecessor, and to add the necessary arcs.

Remove marked removes the marked nodes from graph. We use this procedure to remove descendants or predecessors of the pivot node from the original graph. Any vertices found in both graphs are copied to a new graph, as they make up a strongly connected component.

Topological ordering of the sub graphs are obtained by the order in which we returns the graph. Due to Lemma 3 we can be sure that there are no nodes in the graph which depend on any node in the descendant graph. We can thus return this at the end of the graph. Furthermore we note that the nodes in the predecessor set can

have no dependencies outside of the set itself. What order we return the SCC and the remainder in is irrelevant as long as they are placed in between the descendant and predecessor set, as they can not be dependent on each other.

4.3 Baders algorithm

Implementing Bader's algorithm represents a considerable amount of the work done on this thesis. It includes a considerably more complicated algorithm than the DCSC. Furthermore we have implemented it for use on a distributed system, which is more tedious to program. Even though it is not a complete algorithm for the problem we want to solve, we consider it an interesting prospect for alteration, due to its scalability. The distribution of sub graphs also allows us to exploit parallelism due to independent regions as mentioned in Section 2.1.

Source code for this implementation can be found on:
https://github.com/henvik/BaderCycle.git.

Graph representation

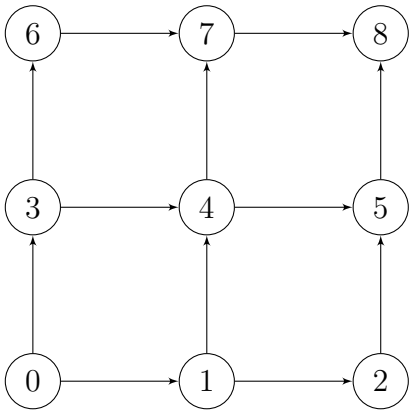
The algorithm assumes input in the form of an already partitioned graph represented in some sensible format. Our aim is to develop a fully functional method for solving the transport equation in parallel. We have thus chosen a graph representation and partitioned it into sub graphs. To represent the graph we use an adjacency list. This approach conserves memory and allows for easy traversal of the graph.

We implement the adjacency list by using using two arrays: `ia` and `ja`. Here `ja` holds all the edges of the graph, and `ia[i]` holds the index of vertex i's first edge in `ja`. This means that { `ja[ia[i]]`, `ja[ia[i]+1]`, `ja[ia[i]+2]`, ..., `ja[ia[i+1]-1]` } constitutes all the vertexes which have an edge from the i'th vertex. In Figure 4.2 we show an example graph and its corresponding adjacency list.

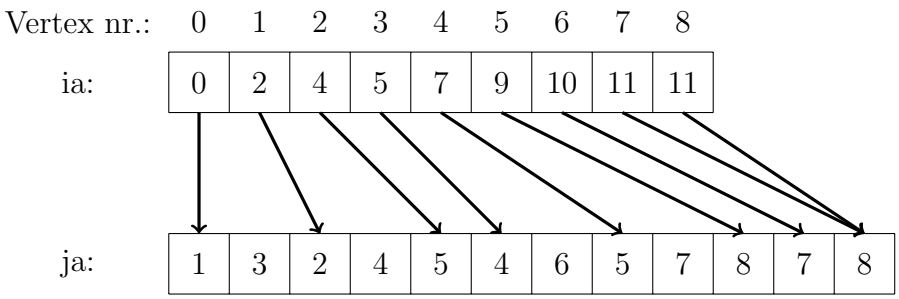
Domain decomposition

Bader's algorithm [10] works on subgraphs and we therefore have to decompose our graph into as many sub graphs as there are processors. For simplicity we have used rectangular grids when developing this algorithm. This makes the process of partitioning the graph straight forward, and should suffice for testing. There exists frameworks for partitioning graphs, such as [14], which could be applied later, but we deem this an unnecessary complication at this stage.

In Figure 4.3 we show how an 8×8 grid can be partitioned across 4 processors. This partitioning is done on the root processor and distributed to the other processors. To maintain the topological structure of the grid we arrange the processors in a Cartesian coordinate system and distribute the graph according to this system. In Figure 4.3 the axis' denotes the Cartesian coordinates of each sub graph in the grid.



(a) Example graph



(b) Conceptual illustration of the adjecency list corresponding to the graph in 4.2a

Figure 4.2: Example graph and its adjacency list.

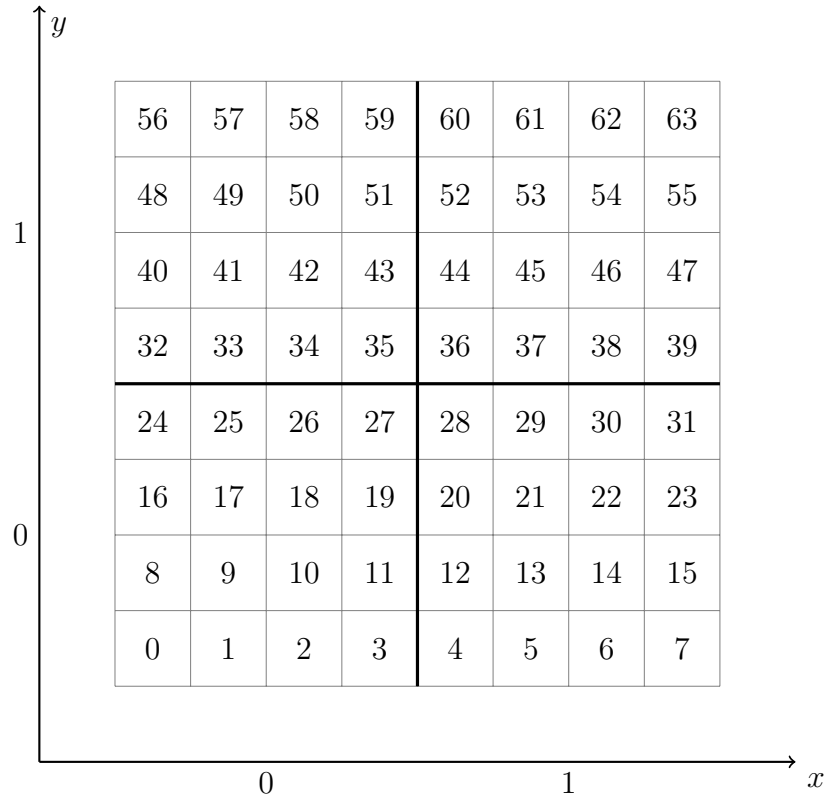


Figure 4.3: The figure illustrates how a 8×8 domain is split between four processors.

Distribution

Our implementation reads the graph from a formatted text file. This allows for a certain degree of mobility and makes it possible to migrate graphs generated in MRST [18]. We also devised a simple program for generating semi-random grids for testing.
We have not focused on parallel I/O in our implementations, and have instead chosen an approach where the grid is read from file by one of the processors, and then distributed to all the other processors in the network. Our implementation of this includes a method for reading a CSV file and converting it into either an adjacency list or a linked list. After this is done, we build a send buffer for each of the processors in the grid and send it to the respective processor using MPI Send. To support this work a series of subroutines are implemented. These are included in Appendix C.4.

Local cycle discovery

During the discovery phase the local sub graphs are examined for local cycles. In this phase, the reachability lists containing information about which trans arc are reachable from the border vertices, are also created. This is done through the following routines:

discovery Initializes required variables and calls the three other routines. Code can be found in Listing C.1.

visit Performs the visiting routine as described in Section 3.3. Every trans arc that is found, is explicitly saved in both the express graph and the reachability list of that vertex. If a cycle is found this is communicated to all the other processes, and the algorithm is halted. Code can be found in Listing C.2

comm_transArcs Processors communicate the trans arcs found during the discovery phase with their neighbours using non-blocking sends. Code can be found in Listing C.3

complete_ExprGraph After all trans arcs have been found, the express graph is completed by adding express arcs to the graph, as described in Section 3.3. Code can be found in Listing C.4.

Pairwise merging

The merging phase is controlled by a main routine called simply `merge`, which controls which processors receive and send their graphs, such that the entire graph ends up on one process. This is done by bitwise manipulation of the ranks, as shown in Listing C.5.
Merge uses several sub routines to send, receive and merge sub graphs. The main ones are as follows:

SendExp/ReceiveExp These routines control the sending and receiving of the express graphs.

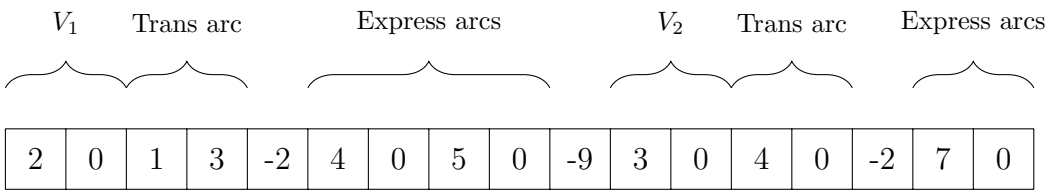


Figure 4.4: Illustration showing how an express graph is packed using the packReceiveBuffer routine. The -2 is used as a delimiter, signifying that there are no more trans-arcs. The -9s signifies that all the express-arcs of a node has been listed and a new node is about to begin.

packReceivebuffer/unpackReceivebuffer To save communication time we pack the express graphs in a condensed format before sending them. Each graph therefore has to be packed and unpacked before being sent and after having been received. An illustration of how the express graphs are packed is shown in Figure 4.4

MergeGraphs In this routine we do the actual merging of the graphs as described in Section 3.3.

Section 5

Results and discussion

Thus far we have presented the theory behind the algorithms and our own implementations. In this section we present numerical result, discuss these, and make propositions for future development towards a parallel multiphase solver.

Testing

To test our implementations we initially used grids exported from MRST [18]. However, to do performance testing we require larger grids than were conceivable to generate with MRST. We therefore made our own program for generating pseudo-random grids for test purposes. This program can be found on https://github.com/henvik/GraphGen

Our program generates vertices in either a two dimensional square or a three dimensional cube. All grids imitate a reservoir with flow from an injection well in one corner of the grid to a production well in the other corner. The simplest grid are a two-dimensional grid with flow strictly diagonal, as shown in Figure 5.1. All vertices has edges to its right- and above neighbour.

To get cycles in our graphs we have assigned edges based on probabilities. To keep the overall flow going from one corner to another we have assigned a 90% probability for a vertex to have an edge to its right neighbour, a 75% probability for an edge to the above neighbour. In addition we have made edges backwards and downwards

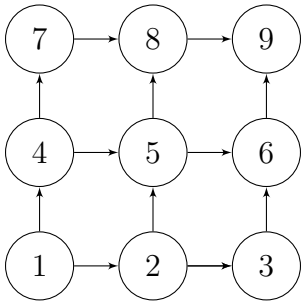


Figure 5.1: Example grid showing the structure of the simplest test case.

with probability of 10%. In that way we get a graph with 15-20 % of the vertices included i a strongly connected component, evenly distributed throughout the graph.

Our implementation of Bader's algorithm has been developed for two-dimensional geometries, as introduced by Bader [10]. It is thus not capable of handling three dimensional geometries, and have therefore only been tested with two-dimensional grids. For DCSC we have also tested with three dimensional geometries, generated the same way as their two-dimensional counterparts. We have tested on a Intel Core i7-4770 CPU with 3.40Ghz×4 and 16 Gb of memory, running Ubuntu 14.04 LTS. The processor has 4 hyper-threaded cores capable of a maximum of 8 logical threads.

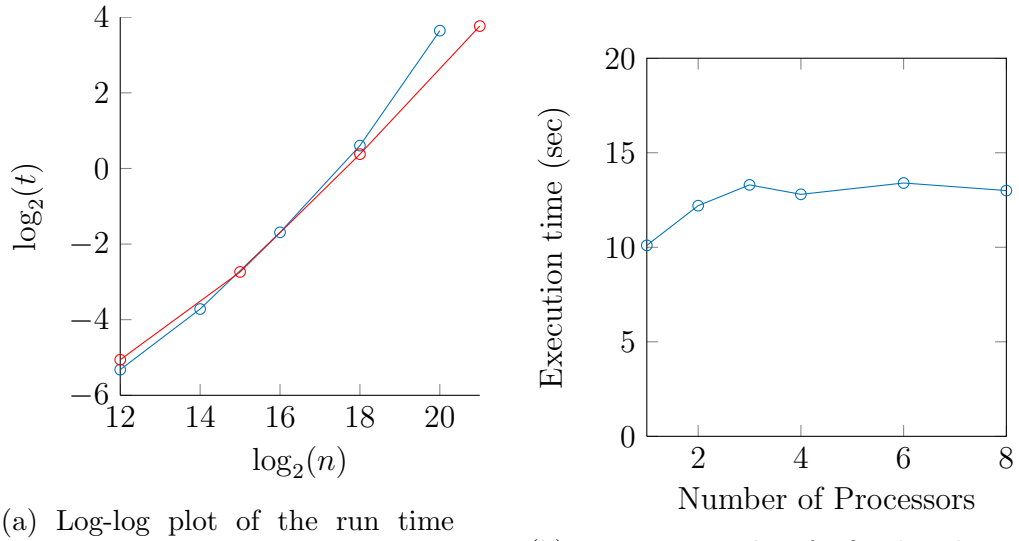
5.1 Shared memory DCSC

We now present test results from our implementation of DCSC. Our implementations of DCSC is developed for use on shared memory machines. To verify the claimed complexity of $O(n\log(n))$ stated in [9], we have tested our program on a single processor, with varying problem sizes. Figure 5.2a shows execution times for different problem sizes. We see that the run time is proportional to $n\log(n)$, where n is the problem size. This is in accordance with the theoretical serial complexity presented by Fleischer [9]. Furthermore, the run time is not affected by the dimension of the underlying grid. Three dimensional grids have the same run time as two-dimensional grids, as long as the number of nodes are equal.

Additionally we have tested the effect of the parallelism by comparing run-times to the number of processors. Figure 5.2b shows execution time for a fixed problem size of $n = 2^{20}$ nodes, for different number of processors.
The implementation of DCSC for shared memory architecture was fruitless. Figure 5.2b shows that there is nothing to gain by sorting the graph in parallel, as one processor gives the fastest execution time. This indicates that the overhead created by solving in parallel exceeds the gains in execution time, discouraging further developments using only shared memory. McLendon et al. [29] reported linear speed-up on their distributed implementation. Although their application differ from ours, a distributed implementation should be explored.

5.2 Using DCSC to parallelize the FMS

As mentioned in Section 3 a criteria for a successful parallel FMS is parallelism in both the reordering and the sequential solving of the elements. We propose two possible adaptations that can be made in order to achieve this with DCSC. Of the two proposals we make, the first is the most substantiated, whereas the second is a notion that we would like to explore further.



(a) Log-log plot of the run time on one processor for different grid sizes. Blue line indicates two-dimensional grids, red line indicates three-dimensional grids.

(b) Run time results of a fixed problem size for different number of processors.

Figure 5.2: Run time test results for DCSC.

Firstly we suggest an approach were we keep track of the dependencies of all the subset DCSC is called recursively on. Consider the first time we find predecessor, descendant, and remainder sets. It follows from Lemma 3 that we can start solving the predecessors elements as soon as that set is sorted, since it has no dependencies elsewhere in the graph. When this set is solved, we can start solving the SCC, and then in turn the remainder and descendant sets. As a consequence, we can do solving of the predecessors parallel to sorting descendants and remainders.

To keep all processors busy, we propose a load balancing scheme built around two queues. One for ordered sets that are ready to be solved and one for sub-graphs which are yet to be sorted. The queue containing sub-graphs for sorting should have a priority system according to how many dependencies they have. In that way we work on the problems that will make it possible to start computations on already ordered sets. For a pivot vertex, the sub problems stemming from the predecessor set will have higher priority than the sub problems stemming from the descendant set, as the descendants can not be solved before the predecessors in any case. To avoid idle processors, work should always be pulled from the longest queue. Ideally, only a small amount of solving should remain when the whole graph has been solved.

Secondly we propose looking into the work by Barnat [31], in which modifications to parallel algorithms are introduced to make them suitable for GPUs. Barnat reports performance results that makes it feasible to obtain over 40 topological orderings in the same time as one serial version needs. As we have mentioned before, there may exist several valid topological orderings of any graph. This is true if there are several sections in the graph whose ordering is irrelevant in the topological ordering.

The fact that DCSC choose pivot vertices randomly, ensures that there will be variations in the different topological orderings returned by several executions on the same graph. By considering these variations in the topological orderings, it should be able to identify which regions are independent of each other, and thus can be solved in parallel. However, this is just a notion and should be given more consideration before attempted.

5.3 Bader's algorithm

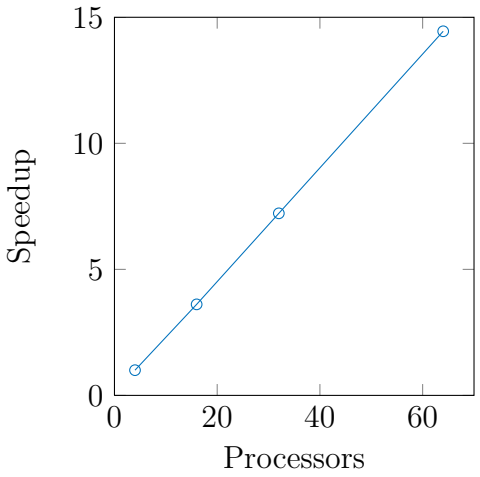
The implementation we present is still immature and needs further work to be a viable alternative to the DCSC . Bader's algorithm is quite involved and has many technical pitfalls, making it particularly hard to implement. We have not succeeded in developing an efficient implementation that incorporates features needed to be of value to FMS. In particular, our current implementation lacks an efficient merging procedure. Our merging procedure has to search through the entire express-graph several times per merge, making the algorithm unscalable. The following test are based on this implementation and does therefore not necessarily reflect the full potential of the algorithm.

Figure 5.3a shows speed up tests for a graph representing a 1000×1000 grid, which were randomly generated with our graph generator. As this graph contains a lot of small cycles evenly distributed, the discovery procedure will detect local cycles and the algorithm terminates before reaching the merge step. We see that this phase scales well, showing a substantial speed-up. This is expected, as each processor gets less data as the number of processors increase.

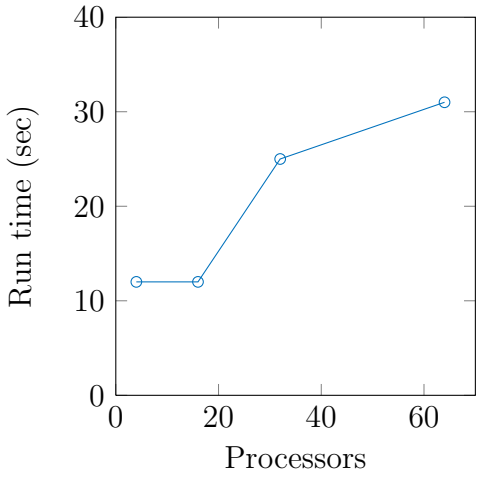
In Figure 5.3b we have tested the implementation on a graph without cycles. The merging then continues until all express graphs are contained on one processor. This creates a bottleneck, since the size of the merges do not decrease as we add more processors. Consider an increase from 4 to 16 processors. After having merged until there are only 4 express-graphs left, we are left with the same amount of work as when we only had 4 processors in total. Here we see an increase in run time for added processors. Because of our inefficient merging procedure, any gain in speed from splitting up the problem is eaten up by the added overhead due to the extra communication.

5.4 Using Bader's algorithm to parallelize the FMS

Bader's algorithm is appealing because of the domain decomposition it is built around. As explained in 2.1, reservoirs with several wells offers possibilities for parallelization. Bader's algorithm is an excellent candidate to exploit this parallelism. The question that remains is whether the advantages of the algorithm will diminish when additional functionality are added. We now present our thoughts on the alterations needed to adapt Bader's algorithm, making it suitable for parallelizing FMS.



(a) Speedup of Bader's algorithm on a graph with cycles.



(b) Execution time of Bader's algorithm on a graph without cycles versus number of processors.

Handling SCCs

For Bader's algorithm to be able to solve the transport equation by way of reordering, it has to treat cycles, not only detect them. In the original algorithm, once a cycle is found, the execution of the program ends. If instead, a predecessor and descendant search like the one used in DCSC were done, one could identify and collapse the vertices in the strongly connected component. This would allow the algorithm to continue and identify any other cycles present.

Topological sorting

Another problem with the existing Bader's algorithm is its incapability to topologically sort the graph. Again we propose to use elements of the DCSC to extend Bader's algorithm. In the modified DCSC proposed by McLendon et al. [29], they add a trimming step to the algorithm. This step examines vertices for incoming and outgoing edges and removes vertices lacking either kind. Due to the definition, such vertices can not be part of a strongly connected component, and the step is thus valid. Drawing from the same wisdom, we propose a scheme were vertices which have no unresolved vertices are marked ready for solving. By this we mean that a vertex for which all predecessors have been solved, is ready to be solved.

Advanced geometries

Our current implementation consider two dimensional squares. This is of course useless for any practical implementation. An extension therefore has to be made in order to treat realistic domains, including irregular geometries in three dimensions.

Section 6

Conclusion

We have in this thesis given a general introduction to the FMS developed by Natvig, Lie et al. [3] and explored how parallel computing can be exploited to speed it up. In particular we have studied the Divide and Conquer Strong Connect (DCSC) algorithm developed by Fleischer et al. [9], and the distributed algorithm due to Bader [10]. For both of the algorithms mentioned above, we have given a thorough account of our own implementations.

DCSC has been implemented for a shared memory architecture. Run time analysis show serial performance in accordance with theoretical complexity. Speed-up test however, are disappointing; showing no benefit from parallelism on shared memory. This indicates that the overhead created by parallelism outweighs the advantages gain by doing computations in parallel on a shared memory system. Tests done by McLendon et al. [29] show that their distributed implementation achieved linear speed-up. We thus propose distributed DCSC as the best candidate for parallelizing FMS and in Section 5 we suggest two possible approaches that could be used for this purpose.

Bader's algorithm has been implemented on a distributed system. We have been unable to develop this algorithm to fit FMS' needs, but do propose several improvements we deem necessary for this to become a capable solver. Although Bader's algorithm has properties that are tractable to a parallel FMS, it is still uncertain whether it can be adapted to meet FMS' needs. Another uncertainty is whether the adapted version, if obtainable, will still be faster than the serial alternative. As mentioned in Section 5, the merging phase has to be improve for this to be a viable alternative.

Based on this we propose that further research towards a parallel FMS should be focused on DCSC. This is a tested algorithm built on simple but powerful method, sporting all the capabilities required for use with the FMS. As we propose in Section 5, it should be possible to use DCSC as a basis for developing a method that sorts and solves in one step. Furthermore, the DCSC does not put any restrictions on the geometry of the input graph, and can thus be used more or less as it is.

Bibliography

[1] International Energy Agency. Key world energy statistics. 2014.

[2] Jostein R Natvig, Knut-Andreas Lie, and Birgitte Eikemo. Fast solvers for flow in porous media based on discontinuous galerkin methods and optimal reordering. In *Proceedings of the XVI International Conference on Computational Methods in Water Resources, Copenhagen, Denmark*, 2006.

[3] Jostein R Natvig and Knut-Andreas Lie. Fast computation of multiphase flow in porous media by implicit discontinuous galerkin schemes with optimal ordering of elements. *Journal of Computational Physics*, 227(24):10108–10124, 2008.

[4] Jostein R Natvig, Knut-Andreas Lie, Birgitte Eikemo, and Inga Berre. An efficient discontinuous galerkin method for advective transport in porous media. *Advances in water resources*, 30(12):2424–2438, 2007.

[5] Jostein R Natvig and Knut-Andreas Lie. On efficient implicit upwind schemes. In *11th European Conference on the Mathematics of Oil Recovery*, 2008.

[6] Felix Kwok and Hamdi Tchelepi. Potential-based reduced newton algorithm for nonlinear multiphase flow in porous media. *Journal of Computational Physics*, 227(1):706–727, 2007.

[7] Hamdi Tchelepi, Mohammad Shahvali, et al. Efficient coupling for nonlinear multiphase flow with strong gravity. In *SPE Reservoir Simulation Symposium*. Society of Petroleum Engineers, 2013.

[8] Henrik Vikøren. Exploring a Parallel Fast Multiphase Solver Based on Potential Ordering. Norwegian University of Science and Technology, 2015.

[9] Lisa K Fleischer, Bruce Hendrickson, and Ali Pınar. On identifying strongly connected components in parallel. In *Parallel and Distributed Processing*, pages 505–511. Springer, 2000.

[10] David A Bader. A practical parallel algorithm for cycle detection in partitioned digraphs. 1999.

[11] Jørg E Aarnes, Tore Gimse, and Knut-Andreas Lie. An introduction to the numerics of flow in porous media using matlab. In *Geometric Modelling, Numerical Simulation, and Optimization*, pages 265–306. Springer, 2007.

[12] Henry Darcy. *Les fontaines publiques de la ville de Dijon*. 1856.

[13] Secondary recovery. http://www.amerexco.com/recovery.html. Accessed: 2015-06-15.

[14] George Karypis and Vipin Kumar. A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. *University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN*, 1998.

[15] Jaime Ambrus, CR Maliska, FSV Hurtado, and AFC da Silva. Finite volume methods with multi-point flux approximation with unstructured grids for diffusion problems. In *Defect and Diffusion Forum*, volume 297, pages 670–675. Trans Tech Publ, 2010.

[16] Ivar Aavatsmark. Multipoint flux approximation methods for quadrilateral grids. In *9th International forum on reservoir simulation, Abu Dhabi*, 2007.

[17] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[18] Knut-Andreas Lie. An introduction to reservoir simulation using matlab. *SINTEF ICT*, 2014.

[19] Peter Pacheco. *An introduction to parallel programming*. Elsevier, 2011.

[20] Michael Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972.

[21] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[22] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[23] John H Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[24] Hillel Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 492–501. IEEE, 1986.

[25] Hillel Gazit and Gary L Miller. An improved parallel algorithm that computes the bfs numbering of a directed graph. *Information Processing Letters*, 28(2):61–65, 1988.

[26] Richard Cole and Uzi Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and computation*, 81(3):334–352, 1989.

[27] Nancy Amato. Improved processor bounds for parallel algorithms for weighted directed graphs. *Information Processing Letters*, 45(3):147–152, 1993.

[28] Ming-Yang Kao. Linear-processor nc algorithms for planar directed graphs i: Strongly connected components. *SIAM Journal on Computing*, 22(3):431–459, 1993.

[29] William Mclendon Iii, Bruce Hendrickson, Steven J Plimpton, and Lawrence Rauchwerger. Finding strongly connected components in distributed graphs. *Journal of Parallel and Distributed Computing*, 65(8):901–910, 2005.

[30] T Sminia and Simona Mihaela Orzan. On distributed verification and verified distribution. 2004.

[31] Jiri Barnat, Petr Bauch, Lubos Brim, and Milan Ceska. Computing strongly connected components in parallel on cuda. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 544–555. IEEE, 2011.

[32] Michael J Quinn and Narsingh Deo. Parallel graph algorithms. *ACM Computing Surveys (CSUR)*, 16(3):319–348, 1984.

[33] Leonardo Dagum and Rameshm Enon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[34] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 20(3):404–418, 2009.

[35] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer, 2004.

[36] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

Appendix A

Tarjans Algorithm

```
index:=0
WorkStack := empty
OutputGraph :=empty
for all v in V do
    if index of v is undefined then
        STRONGCONNECT(v)
    end if
end for
return OutputGraph
```

```
function STRONGCONNECT(v)
    v.index = index
    v.lowlink=index
    index = index +1
    PUSH(WorkStack,v)
    for each (v, w) in E do
        if w.index is undefined then
            STRONGCONNECT(w)
            v.lowlink = min(v.lowlink,w.lowlink)
        end if
        if w is in WorkStack then
            v.lowlink = min(v.lowlink,w.index)
        end if
    end for
    if v.lowlink = v.index then
        start new SCC
        w = POP(W)orkStack
        while w ≠ v do
            add w to SCC
        end while
        add SCC to OutputGraph
    end if

end function
```

Appendix B

Code listings for DCSC

B.1 Main program

```
LinkedList* DCSC_parallel(LinkedList* G, int cutoff){
/*If DCSC is called on an graph smaller than the cutof it calls the serial version*/
  if(G->num_vert<=cutoff){
    return DCSC_serial(G);
  }

/*Finding pivot node. */
  Node *pivot=get_pivot(G,rand()%G->num_vert);
  if(!pivot){
    printf("get_Node_error.\n");
    exit(0);
  }

/*Finding descendants and predecessors of the pivot node. */
  VisitStack *desc_stack=new_VisitStack();
  VisitStack *pred_stack=new_VisitStack();
  LinkedList *desc=new_LinkedList();
  LinkedList *SCC=new_LinkedList();
  LinkedList *pred=new_LinkedList();
/*Finding descendants and predecessors in parallel*/
  #pragma omp task
    FindDescendants(pivot,desc,desc_stack);
  #pragma omp task
    FindPredecessors(pivot,pred,pred_stack);
  #pragma omp taskwait
  free(desc_stack),free(pred_stack);

/* Remove the nodes that have been identified as part of a subset */
  removeMarked(G,desc,pred,SCC);


//  Recursion is done in parallel using tasks, allowing for run-time load balancing
  LinkedList *listOfLists[4];
  LinkedList *predReturn, *remReturn, *descReturn;
    #pragma omp task shared(predReturn)
    predReturn =DCSC_parallel(pred,cutoff);
    #pragma omp task shared(remReturn)
    remReturn =DCSC_parallel(G,cutoff);
  #pragma omp task shared(descReturn)
    descReturn=DCSC_parallel(desc,cutoff);
  #pragma omp taskwait
  listOfLists[0]=predReturn;
  listOfLists[1]= remReturn;
  listOfLists[2]=SCC;
  listOfLists[3]=descReturn;

  return mergeLinkedLists(listOfLists,4);


}

LinkedList* DCSC_serial(LinkedList* G){
/*If DCSC is called on an empty graph it returns immediately*/
  if(G->num_vert==0){
    return G;
  }

/*Finding pivot node. */
  Node *pivot=get_pivot(G,rand()%G->num_vert);
  if(!pivot){
    printf("get_Node_error\n");
    exit(0);
  }
```

```
/*Finding descendants and predecessors of the pivot node.
*/
  LinkedList *SCC=new_LinkedList();
  LinkedList *desc=new_LinkedList();
  LinkedList *pred=new_LinkedList();
  VisitStack *stack=new_VisitStack();

  FindDescendants(pivot,desc,stack);
  FindPredecessors(pivot,pred,stack);
  free(stack);

/* Remove the nodes which have been determined to belong to a subset */
  removeMarked(G,desc,pred,SCC);


//  Recursion

  LinkedList *listOfLists[4];
  listOfLists[0]=DCSC_serial(pred);
  listOfLists[1]= DCSC_serial(G);
  listOfLists[2]=SCC;
  listOfLists[3]=DCSC_serial(desc);

  return mergeLinkedLists(listOfLists,4);


}
```

Listing B.1: DCSC routines

B.2 Core routines

```
/* Finds the descendants of the pivot node
  Input:
    pivot: pivot node whose descendants we seek
    stack: workstack to control the flow of the program
  Output:
    desc: linked list where we save the subgraph that make up the descendants

 */
void  FindDescendants(Node *pivot, LinkedList *desc, VisitStack *stack){
  /* We add a copy of th pivot node to the list of descendants.  Current is our iterator, which
       we use to iterate through all the descendants of the nodes we visit.*/
  arc_t *current;
  pivot->desc_status=new_Node(pivot->vert_num);
  add_node(desc,pivot->desc_status);
  while(pivot){

    /* We check all the descendants of the pivot node. */
    current=pivot->children;
    while(current){
      /* If the descendant already have been visited through another node, we simply add and
           edge to it. */
      if(current->head->desc_status){
        add_edge(pivot->desc_status,current->head->desc_status);
        current=current->next;
        continue;
      }
      /* If it is the first time we encounter the descendant we add copy of it to the list, and
           adds an edge to it from the current node.*/
      current->head->desc_status=new_Node(current->head->vert_num);
      add_node(desc,current->head->desc_status);
      add_edge(pivot->desc_status,current->head->desc_status);
      /* We put the discovered node on the workstack, indicating that we will visit it later.*/
      push_VisitStack(stack,new_StackNode(current->head));
      current=current->next;
    }
    /* We pop a new pivot node to visit after having checked all the descendants.*/
    pivot=pop_VisitStack(stack);
  }

}
/* Finds the predecessors of the pivot node
  Input:
    pivot: pivot node whose predecessors we seek
    stack: workstack to control the flow of the program
  Output:
    pred: linked list where we save the subgraph that make up the predecessors

  Implementation analogous to FindDescendants
 */
void FindPredecessors(Node *pivot, LinkedList *pred, VisitStack *stack);

/* Finds the union of the descendants and predecessors. Also removes these nodes from the
     subgraphs.
```

```
  Input:
    G: the graph we have searched
    desc: the descendants found
    pred: the predecessors
  Output:
    SCC: nodes which make up a strongly connected component.

*/
void removeMarked(LinkedList *G, LinkedList *desc, LinkedList *pred, LinkedList *SCC){
  Node *current=G->first;
  Node *tmp;
  /* We iterate through all the  nodes in the graph*/
  while(current){
    /* If a node is marked as both descendant and predecessors it is added to the SCC set. The
        node is then removed from the three other subgraphs.  */
    if(current->desc_status && current->pred_status){
      add_node(SCC,new_Node(current->desc_status->vert_num));
      tmp=current->next;
      remove_node(desc,current->desc_status);
      remove_node(pred,current->pred_status);
      remove_node(G,current);
      free_node(&current->desc_status);
      free_node(&current->pred_status);
      free_node(&current);
      current=tmp;
      continue;
    }
    /* If a node is a predecessors or descendant it is removed from the graph */
     if(current->pred_status || current->desc_status){
      tmp=current->next;
      remove_node(G,current);
      free_node(&current);
      current=tmp;
      continue;
    }
    current=current->next;
  }
}
```

Listing B.2: Functions used to support the DCSC routine

B.3 Graph utilities

For support functions used in the implementations we include function declarations only.

```
typedef struct arc_t{
  struct Node* head;    //Pointer to node which the arc points to
  struct arc_t* next;   //pointer to the next arc
}arc_t;


typedef struct Node{
  int vert_num;     //the vertex number in the global graph
  struct Node *next;      //the next node in the list
  struct Node *prev;      //the prev node in the list

  arc_t *parents;   //list of the direct predecessors
  arc_t *children;    //list of the direct descendants

  struct Node *desc_status; //indicator of already discovered descendants
  struct Node *pred_status; //indicator of already discovered predecessors
}Node;


typedef struct LinkedList{
  int num_vert;     //number of vertices in graph

  Node *first;      //pointer to the first node in the graph
  Node *last;       //pointer to the last node in the graph
}LinkedList;

typedef struct Node_pointers{
        struct Node** list;
        int size;
        int capasity;
}Node_pointers;


/* Creates a new instance of the structure LinkedList on the heap. Returns a pointer to it's
    location. */
LinkedList* new_LinkedList();

/* Creates a new instance of the structure Node on the heap. Returns a pointer to it's location.
      */
```

```
Node* new_Node(int vert_num);

/*Adds a node to a graph*/
void add_node(LinkedList *graph,Node *node);

/* Adds an edge from the source to the terminal */
void add_edge(Node *source, Node *terminal);

/* Copies the contents of a node to a new instance */
Node* copy_Node(Node *node);

/* Frees a node, including all its edges */
void free_node(Node **node);

/* Frees a LinkedList, including all its nodes */
void free_LinkedList(LinkedList **G);

/* Imports a grid from a file and saves it as a LinkedList*/
LinkedList* importGridLinked(char* file);

/* Converts a grid from adjacency list representation to LinkedList */
LinkedList* convertGrid(int *ia,int size_ia , int *ja);

// imports grid from file, saves it as an adjecency list in ia and ja. nv is the number of
    vertices. ne is the number of edges
void importGrid(char* file, int **ia, int **ja, int *nv, int *ne);

//Printing
void printNode(Node *node);
void printLinkedList(LinkedList *graph);
void printNodeParents(Node *node);
void printLinkedListPredecessors(LinkedList *graph);
void printNode_pointers(Node_pointers* pointers);
void printLinkedListSequence(LinkedList *graph);

/* Checks if a certain node is a part of a graph */
bool isIn(LinkedList *G, Node *node);

/* Finds a specified node based on the vertex number */
Node* get_Node(LinkedList *graph,int vert_num);

/* Returns the num_in_graph node in the graph*/
Node* get_pivot(LinkedList *graph, int num_in_graph);


//Remove nodes and edges
/* Removes a node from a graph*/
void remove_node(LinkedList *G,Node* node);

/* removes an edge between two nodes */
void remove_edge(Node* source, Node* terminal);

/* Removes all the forward edges of a node*/
void remove_forward_edges(Node* node);

/*Removes all the backward edges of a node */
void remove_backwards_edges(Node* node);

/*  removes a subset of nodes from a graph */
LinkedList* remove_from_graph(LinkedList* graph, Node_pointers* sub_graph);

/* Removes nodes from the graph based on their statuses*/
void removeMarked(LinkedList *G, LinkedList *desc, LinkedList *pred, LinkedList *SCC);

//Appends the second list onto the first
void appendLinkedLists(LinkedList *first, LinkedList **second);

/*Merges n LinkedLists into one*/
LinkedList* mergeLinkedLists(LinkedList **listOfLists, int n);
```

Listing B.3: Headers for graph utilities used in the DCSC

Appendix C

Code listings for Bader's algortihm

C.1 Discovery phase

```
ExpGraph* discovery(int num_vert){
/*
...initializing exluded for brievity

*/
/*Colorcoding of all the vertices are set to WHITE, meaning not yet visited/
  for(int i=0; i<num_vert;i++){
    color[i]=WHITE;
  }
/*All vertices are visited in turn*/
  for(int i=num_vert-1; i>=0;i--){
    if(color[i]==WHITE){
      adjecent[i]=visit(i);
    }
  }
/*Communicate whether any of the processes have found a cycle. All processes end if any cycles
    are found.*/
  MPI_Alltoall(found,1,MPI_INT,recv,1,MPI_INT, cart_comm);
  for(int i=0; i<size;i++){
    if(recv[i]){
      MPI_Finalize();
      exit(0);
    }
  }

/* Communicate information about trans arcs to neighbouring processors, and add information
    recieved to the processors express*/
  comm_transArcs(expGraph);
  completeExpGraph(expGraph,adjecent);


  return expGraph;
}
```

Listing C.1: Discovery

```
AdjLst visit(int v){
//  printf("Visit called on local vertex %d, globCellNr %d.\n",v,local_map[v]);
  adjecent[v]=new_AdjLst();
  color[v]=RED;
  for(int w=local_ia[v]; w<local_ia[v+1]; w++){
    int procNr=procNrFromCell(local_dims, gridDims,local_ja[w]);
    int locCellNr=globalCellNr2localCellNr(local_ja[w],gridDims,local_dims);
    if(procNr==rank){
  //    printf("Internal edge: %d, on proc %d. Local cellNr is %d\n", local_ja[w],rank,
      globalCellNr2localCellNr(local_ja[w],gridDims,local_dims));
      switch(color[locCellNr]){
        case WHITE:
          adjecent[locCellNr]=visit(locCellNr);
          merge_AdjLst(&adjecent[v],&adjecent[locCellNr]);
          break;
        case BLACK:
          merge_AdjLst(&adjecent[v],&adjecent[locCellNr]);
          break;
        case RED:
          printf("Cycle found %d, im outta here!\n",local_map[v]);
          int *found=malloc(size*sizeof(int));
          for(int i=0; i<size;i++){
            found[i]=1;
          }
          int *recv=malloc(size*sizeof(int));
```

```
            MPI_Alltoall(found,1,MPI_INT,recv,1,MPI_INT, cart_comm);
            MPI_Finalize();
            exit(0);
        }
      }
      else{
      //  printf("External edge: %d, on proc %d, to %d\n", local_ja[w],rank,procNr);
        //add_trans_arc( v,  local_ja[w], &trans_arcs, &trans_arcs_count, adjecent);
        addExternalEdge(expGraph, local_map[v], local_ja[w],rank,procNr, &adjecent[v]);
      }
    }
    if(adjecent[v].size==0){
      color[v]=GREEN;
    }
    else{
      color[v]=BLACK;
    }
    return adjecent[v];
}
```

Listing C.2: Visit subroutine of Bader's algorithm

C.2 Express graph phase

```
void comm_transArcs(ExpGraph *expGraph){

  int *buffSizes=(int *)calloc(NEIGHBOURS, sizeof(int));
  for(int i=0; i<NEIGHBOURS;i++){
    //pre allocating memory for send/recv buffers. reallocating later if needed.
    trans_buffer[i]=new_EdgeLst(INITSIZE);
  }
/* Subroutine assembling the information in a condensed array suitable for sending */
  build_transBuffer(buffSizes,trans_buffer, expGraph);

/* Commiting self made MPI_Datatype */
  MPI_Datatype MPI_Edge;
  def_datatypes(&MPI_Edge);

/* Requests for communication */
  MPI_Request send_req[NEIGHBOURS],recv_req[NEIGHBOURS];


/*Sending the buffer to the processor to the north*/
  if(north!=-2){
    MPI_Isend(trans_buffer[0].list,buffSizes[0],MPI_Edge,north,0,cart_comm,&send_req[0]);
  }
/*
...
Analogusly for sout,east, and west
...
*/

/*Reciving buffer from the processor to the north */
  Edge *recv_buff;
  if(north!=-2){
    MPI_Status recv_status;
    int recv_size;
    MPI_Probe(north,1,cart_comm,&recv_status);
    MPI_Get_count(&recv_status,MPI_Edge,&recv_size);

    recv_buff =malloc(recv_size*sizeof(Edge));

    MPI_Recv(recv_buff,recv_size,MPI_Edge,north,1,cart_comm,MPI_STATUS_IGNORE);
    recvTransArcs(recv_buff, recv_size, expGraph, north);
    free(recv_buff);

  }

/*
...
 Analogously for south, east, and west
...
*/

  /* Waiting for all sends to complete */
  if(north!=-2){
    MPI_Wait(&send_req[0],MPI_STATUS_IGNORE);
  }
/*
...
 Analogously for south, east, and west
...
*/

  free(buffSizes);
  for(int i=0; i<NEIGHBOURS;i++){
    free(trans_buffer[i].list);
```

```
  }

}

/* Routine for putting the information of the received message into the ExpGraph */
void recvTransArcs(Edge* list, int size, ExpGraph *expGraph, int procNr){

  for(int i=0; i<size;i++){
    ExVert *new=new_ExVert(list[i].v,procNr);
    addExVert(expGraph,new);
    addTransArc(new, newTransArc( rank, list[i].w));
  }

}
```

Listing C.3: Routine handling communication of trans arcs with neighbouring processes. Part of Bader's algorithm.

```
void completeExpGraph(ExpGraph *G, AdjLst *adjecent){
  //Add express acrs
  ExVert *current=G->first;
  while(current){
    if(current->procNr!=rank){
      TransArc *cTarc=current->trans_arcs;
      while(cTarc){
        int index=globalCellNr2localCellNr(cTarc->vert_num,gridDims,local_dims);
        for(int i=0; i<adjecent[index].size;i++){
          addExArc(current,newExArc(adjecent[index].list[i], cTarc->proc_nr));
        }
        cTarc=cTarc->next;
      }
      current=current->next;
      continue;
    }
    ExVert *iterater=G->first;
    while(iterater){
      if(iterater==current){
        iterater=iterater->next;
        continue;
      }
      TransArc *cTarc=iterater->trans_arcs;
      while(cTarc){
        int index=globalCellNr2localCellNr(current->vert_num,gridDims,local_dims);
        if(isReachable(adjecent[index],cTarc->vert_num)){
          addExArc(current,newExArc(cTarc->vert_num, cTarc->proc_nr));
        }
        cTarc=cTarc->next;
      }
      iterater=iterater->next;
  }
  current=current->next;
  }
}
```

Listing C.4: Routine for adding express arcs to the expressgraph.

C.3 Merge phase

```
void merge(ExpGraph* expGraph){
  EdgeLst* graphBuff;
  for(int h=0;h<log2(size);h++){
    if(last(rank,h)==0){
      if(test(rank,h)==0){
        int procNr=set(rank,h);
        ExpGraph *expRecieved;
        RecieveExp(procNr,&expRecieved);

        MergeGraphs(expGraph, expRecieved,rank,procNr);

      }else{
         int procNr=clear(rank,h);
        SendExp(procNr, expGraph);
      }
    }
  }

}

/*Returns the h least-significant bits of z */
int last(int z, int h);

/*Returns the h least-significant bit of z */
int test(int z, int h);
```

```
/* Returns z with the h least-significant bit set to 1 */
int set(int z, int h);

/* Returns z with the h-least significant bit set to 0 */
int clear(int z, int h);

/*Sends express graph to the specified processor
  Input:
    procNr: rank of receiving processor
    exp: express graph
*/
void SendExp(int procNr, ExpGraph* exp){

    int *send_buffer=(int *)malloc(sizeof(int)*exp->num_vert*12); //magic number 12 is semi-
        arbitrary to allocate enough memory for each vertex, reallocated later if needed.
    int send_count=packGraph(exp, &send_buffer);

    MPI_Send(send_buffer,,MPI_INT,procNr,0,cart_comm);
    free(send_buffer);
}

/*Packs the Express graph in the following manner:
All the vertices are stored sequentially with a -1 signaling the end of one vertice and the
    start of the next.
Within each vertice the info is stored as following. The first element contains the vertice
    number, the second contians the procNr.
The trans arc are stored in pairs of two integers. The first is the vertice, the second the proc
    nr.
A -2 signals the end of transArcs and the start of express arcs. These are stored in pairs of
    two, like the trans arcs.
A -9 signals the end of the express graph.
*/
int packGraph(ExpGraph *exp, int **send_buffer):

/* Receives graph from specified processor
  Input:
    procNr: rank of sending processor
  Outpur:
    exp: express graph received
*/
void RecieveExp(int procNr, ExpGraph** exp){
  MPI_Status recv_status;
  int recv_size;
  MPI_Probe(procNr,0,cart_comm,&recv_status);
  MPI_Get_count(&recv_status,MPI_INT,&recv_size);

  int *recv_buff =malloc(recv_size*sizeof(int));
  MPI_Recv(recv_buff,recv_size,MPI_INT,procNr,0,cart_comm,MPI_STATUS_IGNORE);
  *exp=unPackRecvBuffer(recv_buff);
}
/* Unpacks the receivebuffer
  Input:
    recv_buffer: receive buffer
  Output:
    express graph corresponding to the received information
*/
ExpGraph *unPackRecvBuffer(int *recv_buffer);

/* Merges express graphs
  Input:
    exp1: express graph to be merged
    exp2: express graph to be merged
    origin1: originating processor for exp1
    origin2: originating processor for exp2
  Output:
    merged express graph

*/
ExpGraph* MergeGraphs(ExpGraph* exp1, ExpGraph* exp2, int origin1, int origin2);
```

Listing C.5: Routine for pairwise merging of subgraphs. Used in Bader's algortihm.

C.4 Graph utilites

For support functions used in the implementations we include function declarations only.

```
/*Reads the grid from a .csv file and saves it as an adjecancy list in ia and ja.
  Input:
    - file : char string containing the path to the csv file
  Output: ia: pointer to a string of integers. The i'th element of the lists contains the index
      in ja of the first edge for node i,
      ja: pointer to a string of integer. The ia[i]'th element of ja conatins the first edge of
          the i'th vertex.
*/
void importGrid(char* file, int **ia, int **ja, int *nv);
```

```
/*Prints an adjecancy graph
  Input:
    ia: pointer to a string of integers. The i'th element of the lists contains the index in ja
        of the first edge for node i,
    ja: pointer to a string of integer. The ia[i]'th element of ja conatins the first edge of
        the i'th vertex.
*/

void printGraph(int* ia, int *ja, int nv);

/*Prints an adjecancy graph, where the indices in ia have been maped to global coordinates
    contained in map
  Input:
    ia: pointer to a string of integers. The i'th element of the lists contains the index in ja
        of the first edge for node map[i],
    ja: pointer to a string of integer. The ia[i]'th element of ja conatins the first edge of
        the map[i]'th vertex.
 */
void printMappedGraph(int* ia, int *ja,int *map, int nv);

/*Calculates the rank of the processor owning a cell with a certain cell number
  Input:
    LocalGridDims: array specifying the dimensions of the local grids, distributed across all of
        the processors in the network
    CartGridDims:  array specifying the dimensions of the global grid.
    CellNr:      the global cell number of the cell
  Output:
    Integer specifying the rank of the processor which owns the cell.
*/
int procNrFromCell(int LocalGridDims[], int CartGridDims[], int CellNr);


/* Takes the global coordinates of a cell and returns its local coordinates
  Input:
    globalCoords: the coordinates of the cell in reference to the global grid
    localDims: the dimension of the local grid
  Output:
    localCoords: the coordinates of the cell with respect to the local grid
*/
void globalCoords2localCoords(int globalCoords[],int localDims[],  int* localCoords );

/*Converts local coordinates to local cell nr
  Input:
    loocalCoords: the local coordinates
    localDims: dimension to the local cartesian grid
  Output:
    local cellnr
 */
int  localCoords2localCellNr(int localCoords[], int localDims[]);

/*Converts global cell nr to global cartesian coordinates
  Input:
    cellNr: the global cell number
    GlobalDims: array containing the dimensions of the global cartesian grid
  Output:
    output: pointer to an array containing the cartesian coordinates in the global grid
*/
void cellNr2cartCoord(int cellNr, int* GlobalDims, int* output);

/*Converts cartesian coordinates to global cell nr
  Input:
    x: x-coordinate in the global grid
    y: y-coordinate in the global grid
    dims: dimensions of the global grid
  Output:
    the cellNr
*/
int cartCoord2cellNr( int x, int y, int* dims);

/*Converts global cell nr to local cell nr
  Input:
    globalCellNr: the cell nr of the cell in the global grid
    globalDims: the dimensions of the global grid
    localDims: the dimension of the local grids
  output:
    the cell nr in the local grid
*/
int globalCellNr2localCellNr(int globalCellNr, int globalDims[],int localDims[]);

/*Concatenates the two arrays
  Input:
    A: array of integer
    B: array of integers
    lengthA: number of elements in A
    lengthB: number of elements in B
  output:
    A: contains the concatenated array containing the elements of A followed by the elements of
        A
    lengthA: pointer to integer specifying the number of elements in the new array
*/
void concatenate(int* A, int* B, int* lengthA, int* lengthB);


Edge new_edge(int v, int w);
```

```
void add_trans_arc(int v, int w,EdgeLst *trans_arcs, int *trans_arcs_count,AdjLst* adjecent);

/*Creates a new instance of the structure AdjLst
  Input:
    NONE
  Output:
    new AdjLst allocated on the heap
*/
AdjLst new_AdjLst();

/*Merges two AdjLsts into one.
  Input:
    A: AdjLst
    B: AdjLst
  Output:
    A: New AdjLst now containing the elments of A and B.

*/
void merge_AdjLst(AdjLst *A, AdjLst *B);

/*Adds an edge to an AdjLSt
  Input:
    A: the AdjLst that we want to add an edge to
    edge: the cellNr to the edge that we want to add
  Output:
    A: the AdjLst now containing the new edge
*/
void add_Edge(AdjLst *A, int edge);

/* Searches an AdjLst for a certain edge
  Input:
    A: the AdjLst that we want to search
    x: the cellNr of the edge that we are searching for
  Output:
    boolean value. True if x is in A. False otherwise
*/
bool isInAdjLst(AdjLst A,int x);

/*Creates a new instance of the structure EXpGraph
  Input:
    takes no input
  Output:
    new pointer to a new ExpGraph element
*/
ExpGraph* new_ExpGraph();

/*Creates a new instance of the structure ExVert
  Input:
    vert_num: unique vertex number identifying the vertex
    procNr: rank of the processor on which the vertex currently resides.
  Output:
    new instance of an ExVert
*/
ExVert* new_ExVert(int vert_num, int procNr);

/*Adds a vertex two an express graph
  Input:
    G: express graph
    vert: vertex
  Output:
    G: express graph now containing the new vertex
*/
void addExVert(ExpGraph *G, ExVert *vert);

/*Adds a trans arc to an express vertex
  Input:
    vert: express vertex
    trans_arc: the trans arc
  Output:
    vert: express vertex now containing new trans arc
 */
void addTransArc(ExVert *vert, TransArc *trans_arc);

/*Adds Express arc to express vertex
  Input:
    vert: Express vertex
    ex_arc: express arc
  Output:
    vert: express vertex now containing new express arc
 */
void addExArc(ExVert *vert, ExArc *ex_arc);

/*Creates new instance of the structure TransArc
  Input:
    proc_nr: rank of the processor owning the terminal vertex of the trans arc
    vert_num: the vertex number of the terminal vertex.
  Output:
    pointer to a new TransArc
*/
TransArc* newTransArc(int proc_nr, int vert_num);


/*Creates a new instance of the structure ExArc
  Input:
    vert_num: vertex number of the terminal vertex.
```

```
    proc_nr: rank of the processor owning the terminal vertex of the express arc.
  Output:
    pointer to a new ExArc
*/
ExArc* newExArc(int vert_num, int proc_nr);

/*Searches through an ExVerts express arcs looking for certain express arc
  Input:
    vert: the vertex to be searched
    ex_arc: express arc
  Output:
    Returns true if ex_arc is found in vert. False otherwise.
*/
bool isInExp_arcs(ExVert *vert,ExArc *ex_arc);

/*Searches an express graph for a vertex
  Input:
    G: express graph
    v: vertex number of the sought vertex
  Output:
    Pointer to vertex. NULL if vertex is not found.
*/
ExVert* FindExVert(ExpGraph *G, int v);

/*Prints the express graph in a formatted matter
  Input:
    G: express graph
*/
void printfExpGraph(ExpGraph *G);


/*Adds a trans arc to the express graph G, and a new edge to the adjacency list of exit the
    vertex
  Input:
    G: Express graph
    internal_vert_num: vertex number of the exit vertex
    external_vert_num: vertex number of the entrance vertex
    in_procNr: the rank of the processor owning the exit vertex
    external_procNr: rank of the processor owning the entrance vertex
    adjacent: adjacency list of the exit vertex.
  Output:
    G: express graph with the new trans arc added
    adjacent: adjacency list with the new edge added
*/
void addExternalEdge(ExpGraph *G, int internal_vert_num, int external_vert_num, int in_procNr,
    int external_procNr, AdjLst *adjecent);

/*Checks if a certain vertex is reachable from another
  Input:
    adjecent: Adjacency list of the initial vertex
    vert_num: vertex number of vertex in question
  Output:
    True if vert_num is reachable. False otherwise.
*/
bool isReachable(AdjLst adjecent,int vert_num);

/*Merges the vertices of exp1 and exp2 into one ExpGraph
  Input:
    exp1: express graph
    exp2: express graph
  Output:
    new express graph which vertices are the union of the vertices in exp1 and exp2
*/
ExpGraph* mergeVertices(ExpGraph *exp1,ExpGraph *exp2);

/*Removes vertex from express graph
  Input:
    G: express graph
    vert: vertex to be removed
  Output:
    G: express graph with vert removed
*/

void removeExVert(ExpGraph *G,ExVert *vert);

/*Removes express arc from express vertex
  Input:
    vert: express vertex
    exarc: express arc to be removed
  Output:
    vert: express vertex with exarc removed
*/
void removeExArc(ExVert *vert, ExArc *exarc);
```

Listing C.6: Graph utilities for Bader's algorithm.