A Parallel Gauss–Seidel Method for Computing Contact Forces

Author:
Morten Poulsen

Supervisor:
Kenny Erleben

Masters Thesis

Department of Computer Science,
University of Copenhagen

June 21, 2010

Abstract

Picture an empty mug of coffee at rest on a table. In the real world, the mug would not fall through the table. If one side of the table were elevated to a certain height, the mug would start to slide off the table. The problem of computing how two or more bodies in contact affect each other is called the contact problem, which is the subject of this thesis.

Solving the contact problem is an important bottleneck when running large simulations of multiple rigid bodies. Many traditional methods of solving the contact problem, such as the projected Gauss–Seidel method, are inherently sequential. However, with modern hardware advances, improvements in raw computing power have been realized by adding multiple cores to the processing units.

This poses a challenge. Can sequential algorithms be rephrased in a concurrent setting? Or are simpler, but embarrassingly parallel methods better suited? These are some of the questions this thesis aims to address.

Commonly used techniques for parallelizing a sequential algorithm start by expressing the internal dependencies of the problem in a convenient structure. The dependencies of the contact problem can be expressed by a contact graph, where each body is a node and each contact is an edge.

Applying an edge-coloring scheme to the contact graph can detect the internal dependencies of a given scene. The contacts in each color are independent of all contacts in that color, and can be solved in an embarrassingly parallel manner. This raises a number of questions. Which coloring scheme yields the best parallel performance? What sort of problems can a coloring scheme cause? Will the quality of the solution deteriorate when more cores are added? And how does it scale?

A number of parallel physics solvers based on the proximal map formulation of the contact problem have been implemented, analyzed and benchmarked. The solvers utilize Jacobi and Gauss–Seidel based fixed point schemes.

The conclusions are that minimizing the number of colors maximize parallel performance of the Gauss–Seidel schemes, as the need for communication between cores is reduced. An even distribution of colors improves performance, as it makes load balancing easier with lower overhead. This method does not cause a deterioration of the quality of the sequential solver when more cores are added.

A highly unbalanced coloring can be addressed by merging colors with few contacts into a larger color, and apply a naive parallelization scheme to this color. This can, however, cause some deterioration of the quality of the solution.

Benchmarks suggest that the simpler Jacobi-based methods may produce the same results as Gauss–Seidel-based methods, but faster as the number of cores is increased. Note that there are a number of optimizations that can improve the performance of the Gauss–Seidel-based schemes considerably, and may change this result. This could be a topic for future work.

Abstract

Forestil dig et tomt kaffekrus i hvile på et bord. I den virkelige verden ville kruset ikke falde igennem bordet. Hvis man løftede den ene ende af bordet op til en bestemt højde, ville kruset begynde at glide ned af bordet. Beregningen af hvordan to eller flere genstande i kontakt påvirker hinanden kaldes kontaktproblemet og er emnet for dette speciale.

Løsning af kontaktproblemet er en væsentlig flaskehals, når man vil køre store simuleringer, der involverer mange stive legemer. Mange alment benyttede metoder til løsning af kontaktproblemet, såsom den projicerede Gauss–Seidel metode, er iboende sekventielle. Men i moderne hardware bliver forbedringer af rå beregningskraft realiseret ved at tilføje flere kerner til beregningsenhederne.

Dette er en udfordring. Kan sekventielle algoritmer reformuleres i en parallel sammenhæng? Eller er simplere, trivielt paralleliserbare metoder bedre tilpasset til en parallel sammenhæng? Dette er nogle af de spørgsmål, som dette speciale undersøger.

Almindeligt brugte teknikker til at parallelisere en sekventiel algoritme tager udgangspunkt i at udtrykke problemets indre afhængigheder i en smart struktur. Kontaktproblemets afhængigheder kan udtrykkes gennem en kontaktgraf, hvor hvert legeme er en knude og hvert kontakt punkt er en kant.

Brug af graf-farvning på kanterne kan gøre de indre afhængigheder af den givne scene synlige. Kontaktpunkterne i en farve er uafhængige af de andre kontaktpunkter i samme farve, og kan løses med en triviel parallelisering. Dette rejser en række spørgsmål. Hvilken farvningsmetode giver den bedste parallelle ydeevne? Hvilken type problemer kan en farvning give? Vil kvaliteten af løsningen forværres når flere kerner tilføjes? Og hvordan skalerer løseren efterhånden som flere kerner tilføjes?

En række paralleliserede fysik løsere baseret på en nærmeste punk formulering af kontaktproblemet er implementeret, analyseret og deres ydeevne målt. Løserne har brugt Jacobi- og Gauss–Seidel-baserede fixpunktsmetoder.

Konklusionerne er, at minimering af antallet af farver maksimerer den parallelle ydeevne af de Gauss–Seidel-baserede metoder, da behovet for kommunikation mellem kernerne herved reduceres. En jævn fordeling af farver forbedrer også ydeevnen, da det gør det nemmere at fordele beregningernes belastning med lavere spild. Metoden forværrer ikke kvaliteten af løsningen når kerner tilføjes.

Problemet med meget ubalancerede farvninger kan håndteres ved at flette små farver sammen i en større farve og bruge en naiv paralleliseringsmetode på denne farve. Dette kan dog forsage en mindre forværring af løsningens kvalitet.

Målinger af ydeevne og løsningskvalitet antyder, at simplere Jacobi-baserede metoder skalerer bedre med antallet af kerner, og kan producere tilsvarende løsninger hurtigere end Gauss–Seidel-baserede metoder, når antallet af kerner øges. Bemærk dog, at der er en række optimeringer, der kan forbedre Gauss–Seidel metoderne betragteligt og dermed ændre denne konklusion. Dette kan være et emne for fremtidigt arbejde.

Preface

When I started working on this thesis, I had written a few graduate projects on the subject of rigid body simulation with Kenny Erleben as supervisor. One of these projects led to a paper on heuristics to improve the convergence behaviour of the projected Gauss–Seidel method presented at WSCG 2010.

I had also taken a course on Extreme Multiprogramming, and had been toying around with the problem of constructing a massively parallel physics engine – or at least a parallel rigid body solver. My work on this thesis started in December 2009.

If needed, I can be reached at mrtn@diku.dk.

Prerequisites

The reader of this document is expected to have some mathematical maturity. In particular, the reader is expected to have a basic understanding of matrix algebra corresponding to the level in [Mes94]. Knowledge of Newtonian mechanics is not strictly necessary, but the reader may want to keep an introductory textbook on the subject handy. I recommend [KH02].

Acknowledgements

A big thanks to my supervisor, Kenny Erleben, for helpful comments during the work on this thesis, and for patiently answering my questions. Thanks to Brian Vinter for introducing me to MiGrid. Thanks to the proofreaders Sarah Niebe, Manijeh Elsa Modi and Helle Sørensen for their hard work, and to Jesper Dahlkild for suggestions. And finally, thanks to all the people who put up with me while I was working on this thesis.

Contents

1 Introduction **7**
- 1.1 Notation . . . 9
- 1.2 Overview . . . 9

2 The Contact Problem **10**
- 2.1 The Physical Model . . . 12
  - 2.1.1 Discretization of the Newton–Euler Equations . . . 13
  - 2.1.2 Constraints on Contact Impulses . . . 14
  - 2.1.3 Contact Impulses . . . 15
  - 2.1.4 Handling Collisions . . . 19
- 2.2 Complementarity Problems . . . 20
  - 2.2.1 The Linear Complementarity Problem . . . 20
  - 2.2.2 The Nonlinear Complementarity Problem . . . 21
- 2.3 The Prox Formulation . . . 22
  - 2.3.1 Proximal Maps and Variational Inequalities . . . 24
  - 2.3.2 The Contact Problem Using Proximal Maps . . . 26

3 Jacobi and Gauss–Seidel schemes **30**
- 3.1 Blocked Sparse Matrix Structure . . . 31
  - 3.1.1 Block Operations . . . 32
- 3.2 The Structure of $\mathbf{A}$ . . . 33
- 3.3 Jacobi Scheme . . . 35
- 3.4 Gauss–Seidel Schemes . . . 35
  - 3.4.1 $\mathbf{T}$ Scheme . . . 36
  - 3.4.2 Factorized Scheme . . . 37
  - 3.4.3 Comparisons of the schemes . . . 38
- 3.5 Convergence, Divergence and $\mathbf{R}$-factor Strategy . . . 40
- 3.6 Estimating Error . . . 41
- 3.7 Parallelization of the Schemes . . . 43
  - 3.7.1 Previous Work . . . 43
  - 3.7.2 Shared Data . . . 46
  - 3.7.3 Cost of one Contact Point . . . 47
  - 3.7.4 Detecting Divergence . . . 47
  - 3.7.5 Preprocessing for Cache Coherency . . . 47
  - 3.7.6 Real-time Usage . . . 48

3.8 Benchmarking 48
3.8.1 Ball Grid 49
3.8.2 Ball Pile 51

4 The Contact Graph **52**
4.1 Properties of the Contact Graph 53
4.2 Graph Partitioning 54
4.3 Graph Coloring 56
4.3.1 Application to Gauss–Seidel 57

5 Communicating Sequential Processes **59**
5.1 The Basics of CSP 60
5.2 Performance 61
5.2.1 Shutting Down the System 62
5.2.2 Bugs 62

6 Parallel Jacobi **64**
6.1 Extreme Factorized Scheme 64
6.2 Factorized Scheme 66
6.2.1 Performance Analysis 67
6.3 **T** Scheme 70
6.4 OpenMP 72
6.5 Results 75

7 Parallel Gauss–Seidel **76**
7.1 Naive Factorized 76
7.2 CSP Coloring Schemes 77
7.2.1 Rigid Factorized 78
7.2.2 Relaxed Factorized 78
7.2.3 Relaxed T Scheme 79
7.2.4 Results for Greedy Coloring 80
7.3 OpenMP Coloring Schemes 80
7.4 Performance of the Coloring Schemes 82
7.5 Parallel Gauss–Seidel or Parallel Jacobi? 84
7.6 Comparison with Other Solvers 85

8 Conclusion **87**
8.1 Graph Coloring 87
8.2 Graph Partitioning 88
8.3 Contact Sets 88
8.4 Deformable Objects 88

A Thesis Outline **96**

Chapter 1

Introduction

One common problem in physics based animation of rigid bodies is computing contact forces to prevent penetrations as well as model friction in a plausible or even correct manner. The physics is reasonably well understood, but formulating a mathematical and numerical model has proven to be more challenging. Common problems are speed, stability and correctness [PNE10].

State of the art methods used by the graphics community to solve the contact problem are inherently sequential. This poses a problem as expansion of computational power on commodity hardware as well as supercomputers is based on adding more cores. This tendency is unlikely to change any time soon. Can the current state of the art methods, such as the iterative Gauss–Seidel matrix solver, be restated in a concurrent context? Will it perform well compared to simpler, embarrassingly parallel methods, such as the Jacobi method?

These are the questions this thesis aims to investigate and answer. There exist parallel versions of the Gauss–Seidel method, even some applied to the contact problem, but I have only found few results as to how it is made efficient, and I have found no results that compare it to a parallel Jacobi method.

As such, this thesis is a part of a larger move towards concurrent physics engines. One model commonly used to parallelize a library is the *fork-join model*, where the library is profiled and the slowest parts are parallelized [HGS+07]. The fork-join model is illustrated on Figure 1.1.

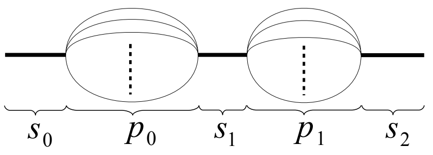


Figure 1.1: *The figure illustrates the fork-join model. The fat line is the controlling process, and the thin lines are worker processes. The portions marked with s_i are sequential parts, and the portions marked with p_j are parallel parts.*

The advantage of this model is that an engine can be parallelized in a modular fashion, and the running time of the worst parts can be improved fast, similar to common performance optimization techniques. The disadvantage is that sequential parts are left, and become the limiting factor on scalability. By Amdahl's Law, if 10% of a problem is sequential, the time usage of the problem can never be reduced below 10%, even if a million CPUs are applied to the problem.

Thus, one may argue that while the fork-join model gives a quick fix, it is not a long term solution. Rather, a new parallel design is likely to scale better in the longer term. This is beyond the scope of this thesis, but may be interesting if one were to construct a distributed physics engine from the ground up.

The enclosed CD contains the developed source code, benchmark data, binaries for Linux and Windows, and an electronic version of this document. Please check out

Variable	**Meaning**
$\mathbf{1}$	$\mathbf{1}$ is the identity matrix
B	B will often refer to a body in the scene
$\vec{c}$, $\vec{C}$	$\vec{c}$ is the product $\mathbf{R}\vec{b}$, $\vec{C}$ denote the contact forces
f, $\vec{f}$, $\vec{F}$	f is a function, when used as a subscript, f is the friction component(s)
	$\vec{f}$ denote forces, $\vec{F}$ denote generalized forces
g, $\vec{g}$	g is often used as a function, $\vec{g}$ represents gravity
$\vec{h}$	$\vec{h}$ are generalized impulses
i, j	i and j are typically used as indices
$\mathbf{I}$	$\mathbf{I}$ is the inertia tensor
$\mathbf{J}$	$\mathbf{J}$ is the Jacobian matrix
k, K	k is a contact point and K is the number of contact points
m, M	m is a body and M is the number of bodies in the scene
$\mathbf{M}$	$\mathbf{M}$ is the generalized mass matrix
n, N	n and N are related index and limit of index
	when used as a subscript, n is the normal component(s)
r, $\vec{r}$, $\mathbf{R}$	r is a factor, $\mathbf{R}$ is a vector of r, $\vec{r}$ is a position vector
$\vec{s}$	$\vec{s}$ is the generalized position and orientation vector,
	but can be a basis vector of the contact coordinate system
$\mathbf{S}$	$\mathbf{S}$ is the position and rotation matrix
t	t is usually the time
$\vec{t}$	$\vec{t}$ is a basis vector for the contact coordinate system
$\mathbf{T}$	$\mathbf{T}$ is $\mathbf{1} - \mathbf{RA}$
$\vec{u}$	$\vec{u}$ is the generalized velocity vector
$\vec{v}$	$\vec{v}$ is a velocity, typically the contact velocity
$\vec{w}$	$\vec{w}$ denote the contact velocity update in the world coordinate system
W, $\mathbf{W}$	W is work, $\mathbf{W}$ is the inverse of $\mathbf{M}$
ε	ε can be the coefficient of restitution or the residual
$\vec{\kappa}$	$\vec{\kappa}$ are the contact impulses in the world coordinate system
μ	μ is the coefficient of friction, both static and kinematic
$\vec{\lambda}$	$\vec{\lambda}$ is the contact impulses in the contact coordinate system
τ, $\vec{\tau}$	τ can be a velocity component, $\vec{\tau}$ is the torque
$\vec{\omega}$	$\vec{\omega}$ is the angular velocity

Table 1.1: *The table shows a non-exhaustive list of variables used in this thesis.*

the readme in the root of the CD file system for more information. The binaries are available online at *http://dl.dropbox.com/u/8130287/binaries.7z*.

1.1 Notation

In general an italic lower-case number a will be a scalar, and an italic upper-case number A will be a scalar constant. A vector will typically be in lower-case $\vec{v}$, and a matrix will typically be in upper-case M. Sets will usually be in upper-case such as $\mathcal{C}$. Specific symbols will be attributed to certain variables as shown in Table 1.1, even though there may be deviations.

1.2 Overview

Chapter 2 introduces the physics of the contact problem and describes some mathematical models based on the physics. Chapter 3 constructs a few numerical methods, based on the prox formulation, and investigates previous work in parallelization of Gauss–Seidel methods. Chapter 4 investigates the contact graph, and Chapter 5 introduces the concurrent CSP framework. Chapter 6 and 7 construct parallel solvers based on Jacobi and Gauss–Seidel methods and investigate their performance. Chapter 8 concludes the thesis.

Chapter 2

The Contact Problem

When two or more objects are in contact with each other, we would like to be able to compute how this contact affects the objects in question. This is called *The Contact Problem*. This thesis will limit itself to investigate the contact problem for rigid bodies, that is, bodies that never change shape. The physical basics of the contact problem are explained in a number of undergraduate textbooks on Newtonian Mechanics, such as [KH02]. However, once one starts working with arbitrary objects and want to simulate it on a computer, the problem becomes somewhat more difficult, particularly as dependencies between the contacting bodies are introduced. This section will introduce the basics of the contact problem and sketch common methods to solve it.

A simple example can be seen on Figure 2.1, where one end of a box is in contact

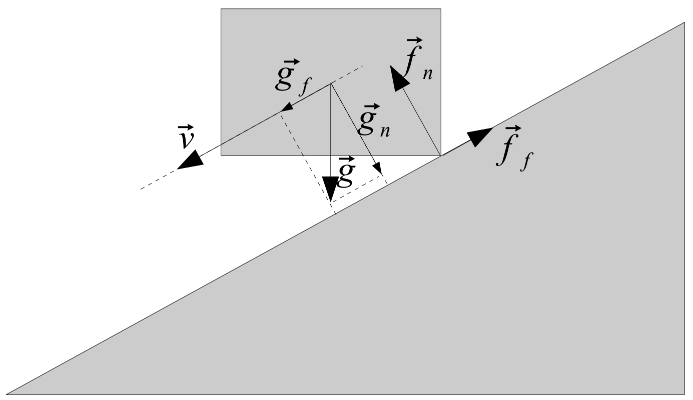


Figure 2.1: *A simple example of the contact problem. The box is placed on a fixed ramp. Gravity $\vec{g}$ acts on the box, and the normal force $\vec{f_n}$ stops the box from falling into the ramp. As gravity also has a tangential component, the surface contact between the box and the ramp applies a friction force to the box.*

with a ramp. The box is affected by gravity $\vec{g}$, and the ramp is at rest and fixed on the ground. What happens? As both bodies are rigid, the box cannot fall into the ramp, so by Newton's 1st Law the ramp must affect the box by a normal force $\vec{f_n}$ equal in magnitude to the gravity force pulling the box into the ramp. By Newton's 3rd Law, the direction of the normal force must be in the direction of the ramps surface normal.

All of the gravity force is not applied in the direction of $\vec{f_n}$, but in the tangential directions, which could cause movement along $\vec{v}$.

However, the motion of an object is not only controlled by which forces affect it. It is also affected by *where* the force is applied. In Figure 2.1, this is simple as there is only one point of contact. However, if the box were rotated so that the bottom side of the box were placed on the ramp as shown on Figure 2.2, there would not be only one point of contact. There would be an entire contact plane. From Amonton's 2nd Law, the magnitude of the friction force between rigid bodies is independent of the surface areas, but the location where it is applied is not. One could try to compute the contact plane, but I have not found any examples of this approach for general rigid body systems. The contact between two bodies is sometimes called the *contact manifold*.

The methods reviewed in this thesis, use a number of *contact points* and assume that the distribution of contact points over the contact surface is a reasonable approximation to the contact plane. The contact points for a given contact plane will be called the *contact set*. As an example, consider Figure 2.2. Placing just one contact point at the closest point between the center of mass of the box and the contact surface would be a bad approximation that could allow the box to rotate into the ramp, whereas a contact point at each corner of the box might be reasonable. It should be apparent that the quality of contact points has significant effect on the quality of the simulation. In the remainder of this thesis, it will be assumed that a reasonable method to generate contact

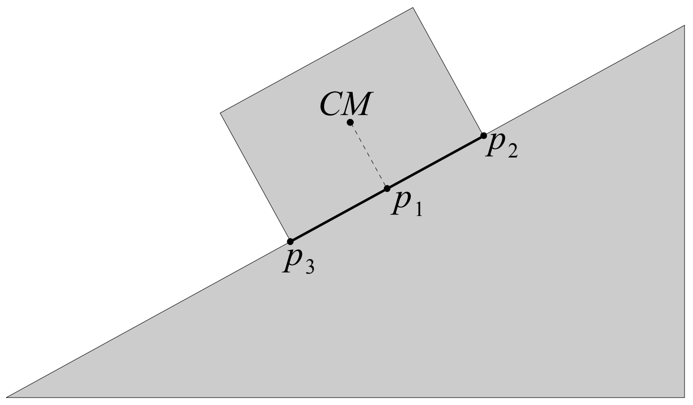


Figure 2.2: *This shows the example from Figure 2.1 with the box rotated so that the bottom side of the box is in contact with the ramp. Where should one place the point where the friction and normal forces are applied? At the entire plane, at the closest point between the center of mass and the ramp p_1 or one or both of the corner points p_2 and p_3?*

points is available.

The acceleration-based models used in the preceding paragraphs have a number of drawbacks such as being unable to handle collisions and instability [MW88]. For these reasons velocity-based methods are often used instead, even though some researchers suggest a number of problems with this model [Cha99]. At the moment, we will continue to argue using the acceleration-based model, but the model will be converted to a velocity-based[1] model in due time.

2.1 The Physical Model

A rigid body is a body that does not deform under any circumstances. Such an object does not exist in the real world – it is an idealization of the real world. However, for a large group of solid objects, this idealization is a reasonable approximation that allows us to ignore the internal structure of a solid. The motion of a single rigid body with mass m can be described by the motion of its center of mass and its rotation around the center of mass, as described by the equations:

$$\frac{d\vec{r}_{cm}}{dt} = \vec{v}_{cm} \tag{2.1a}$$

$$\frac{d}{dt}q = \frac{1}{2}\vec{\omega}q \tag{2.1b}$$

$$\frac{d\vec{v}_{cm}}{dt} = \frac{\vec{f}^{ext}}{m} \tag{2.1c}$$

$$\frac{d}{dt}\vec{\omega} = \mathbf{I}^{-1}\vec{\tau} - \mathbf{I}^{-1}(\vec{\omega} \times \mathbf{I}) \cdot \vec{\omega} \tag{2.1d}$$

where the subscript cm refers to the motion of the center of mass, $\vec{v}$ is the velocity, $\vec{f}$ is the force, $\vec{\omega}$ is the angular momentum, $\vec{\tau}$ is the torque, $\vec{I}$ is the inertia tensor, and $\vec{q}$ is a quaternion describing the orientation of the rigid body. As noted in [ESHD05], equations (2.1) can be written on matrix form, describing a system of M rigid bodies:

$$\frac{d\vec{s}}{dt} = \mathbf{S}\vec{u} \tag{2.2a}$$

$$\mathbf{M}\frac{d\vec{u}}{dt} = \vec{F}_{ext} + \vec{C} \tag{2.2b}$$

where $\vec{s} \in \mathcal{R}^{7M}$ is the *generalized position and orientation vector* defined by

$$\vec{s} = \begin{bmatrix} \vec{r}_1 \\ q_1 \\ \vec{r}_2 \\ q_2 \\ \vdots \\ \vec{r}_M \\ q_M \end{bmatrix}, \tag{2.3}$$

[1]It is actually an impulse-based model, but in some literature impulse-based models refer to collision models. Thus, velocity-based is used to avoid confusion and/or ambiguity.

The position and rotation are controlled by the matrix $\mathbf{S} \in \mathcal{R}^{7M \times 6M}$

$$\mathbf{S} = \begin{bmatrix} \mathbf{1} & & & & \mathbf{0} \\ & \mathbf{Q}_1 & & & \\ & & \ddots & & \\ & & & \mathbf{1} & \\ \mathbf{0} & & & & \mathbf{Q}_M \end{bmatrix} \tag{2.4}$$

where $\mathbf{Q}_i$ is the matrix representation of the quaternion $\vec{q}_i$

$$\mathbf{Q}_i = \frac{1}{2} q_i = \begin{bmatrix} -x_i & -y_i & -z_i \\ s_i & z_i & -y_i \\ -z_i & s_i & x_i \\ y_i & -x_i & s_i \end{bmatrix} \tag{2.5}$$

The *generalized velocity vector* $\vec{u} \in \mathcal{R}^{6M}$ is defined by

$$\vec{u} = \begin{bmatrix} \vec{v}_1 \\ \vec{\omega}_1 \\ \vec{v}_2 \\ \vec{\omega}_2 \\ \vdots \\ \vec{v}_M \\ \vec{\omega}_M \end{bmatrix} \tag{2.6}$$

and the *generalized external force vector* $\vec{F} \in \mathcal{R}^{6M}$ is defined by

$$\vec{F}_{ext} = \begin{bmatrix} \vec{f}_1^{ext} \\ \vec{\tau}_1^{ext} - (\vec{\omega}_1 \times \mathbf{I}_1) \cdot \vec{\omega}_1 \\ \vec{f}_2^{ext} \\ \vec{\tau}_2^{ext} - (\vec{\omega}_2 \times \mathbf{I}_2) \cdot \vec{\omega}_2 \\ \vdots \\ \vec{f}_M^{ext} \\ \vec{\tau}_M^{ext} - (\vec{\omega}_M \times \mathbf{I}_M) \cdot \vec{\omega}_M \end{bmatrix} \tag{2.7}$$

Note how each body has its own block of data in each vector or matrix. This blocked structure will be exploited in the following for convenience and performance. Index subscript i will refer to the ith block.

The vector $\vec{C}$ represents the contact force applied to each body, which is our unknown. Before we investigate ways of computing $\vec{C}$, the Newton–Euler equations needs to be discretized.

2.1.1 Discretization of the Newton–Euler Equations

In Newtonian mechanics, macroscopic movement is usually considered to be continuous. However, computers are not well suited for continuous problems. This leads to the need of discretization of continuous equations. As such, most of our differential calculus assumes that the slope of the function $g(t)$ from t to $t + dt$ is constant if the step dt is infinitely small. As an infinitely small step is not feasible when running a physical simulation, a small stepsize Δt is used. The magnitude of Δt depends on a number of factors, from the physical model used to how much time is available for the simulation.

As the finite stepsize is an approximation, it is reasonable to assume that the ability of the stepper to take variable slopes of $g(t)$ into account has significant impact on the quality of the output from the simulator. In the following, explicit Euler integration will be used for clarity and simplicity, though there are other steppers that may yield a better approximation, such as the Moreau stepper mentioned in [LG03].

In explicit Euler integration, the time derivative of the generalized velocities are approximated by

$$\frac{d\vec{u}}{dt} \approx \frac{\vec{u}^{\,t+\Delta t} - \vec{u}^{\,t}}{\Delta t} \tag{2.8}$$

where the superscripts refer to the time. Rearranging the terms yields

$$\vec{u}^{\,t+\Delta t} \approx \vec{u}^{\,t} + \Delta t \frac{d\vec{u}}{dt} \tag{2.9}$$

If we multiply (2.2b) by $\mathbf{M}^{-1}$, $\frac{d\vec{u}}{dt}$ is isolated and can be inserted into (2.9) which yields

$$\begin{aligned} \vec{u}^{\,t+\Delta t} &\approx \vec{u}^{\,t} + \Delta t \mathbf{M}^{-1}(\vec{F}_{ext} + \vec{C}) &(2.10)\\ &= \vec{u}^{\,t} + \mathbf{M}^{-1}(\Delta t \vec{F}_{ext} + \Delta t \vec{C}) &(2.11) \end{aligned}$$

From Newton's 2nd Law, the impulse $\vec{P}$ can be defined by

$$\vec{P} = \int \vec{F} dt \tag{2.12}$$

If the average force $\vec{F}_{avg}$ applied over a timestep Δt is known, we get

$$\vec{P} = \vec{F}_{avg} \Delta t \tag{2.13}$$

This can be used in (2.11) to go from a acceleration-based formulation to a velocity-based formulation of the contact problem. The advantage is that we no longer need to compute the forces at every instance, nor assume that the force computed at some time t is appropriate for the current timestep. We define the external impulses $\vec{h}_{ext} = \Delta t \vec{F}_{ext}$ and the contact impulses $\vec{\kappa} = \Delta t \vec{C}$ and insert into (2.11):

$$\vec{u}^{\,t+\Delta t} \approx \vec{u}^{\,t} + \mathbf{M}^{-1}(\vec{h}_{ext} + \vec{\kappa}) \tag{2.14}$$

Similarly, explicit Euler integration can be used to compute the approximate position update, based on the approximate velocity. From (2.2a) we get

$$\begin{aligned} \vec{s}^{\,t+\Delta t} &\approx \vec{s}^{\,t} + \Delta t \frac{d\vec{s}}{dt} &(2.15)\\ &= \vec{s}^{\,t} + \Delta t \mathbf{S} \vec{u}^{\,t+\Delta t} &(2.16) \end{aligned}$$

2.1.2 Constraints on Contact Impulses

In the preceding section, $\vec{\kappa}$ was simply treated as an unknown. This section will investigate the physical constraints on the contact impulses. To do this, a contact coordinate system will be defined, and bounds on the contact impulses in this coordinate system will be investigated. This, however, necessitates constructing a matrix to convert the contact impulses from the contact coordinate system into world coordinate system and apply them to the bodies. First, some general ideas on constraints will be refreshed.

In [ESHD05] two types of constraints are mentioned. A *holonomic constraint* can be described by the constraint function $\Phi(\vec{x}) = 0$ and a *non-holonomic constraint* can be described by the constraint function $\Psi(\vec{x}) \geq 0$, where $\vec{x}$ is a vector of arguments. In the case of contact forces, $\vec{x}$ consists of the position and orientation $\vec{s}$ of two bodies at a particular timestep. Taking the time derivative of the constraint yields a *kinematic constraint*,

$$\frac{d}{dt}\Phi(\vec{s}) = \frac{\delta \Phi}{\delta \vec{s}}\frac{d\vec{s}}{dt} \tag{2.17}$$

$$= \frac{\delta \Phi}{\delta \vec{s}}\mathbf{S}\vec{u} \tag{2.18}$$

Using the Jacobian $\mathbf{J}_\Phi = \frac{\delta \Phi}{\delta \vec{s}}\mathbf{S}$ we get

$$\mathbf{J}_\Phi \vec{u} = 0 \tag{2.19}$$

By the principle of virtual work, the generalized impulse from the constraint is

$$\vec{f}_\Phi = \mathbf{J}_\Phi^T \vec{\lambda} \tag{2.20}$$

where $\vec{\lambda}$ is the impulse in constraint coordinates. Similar arguments can be used for non-holonomic constraints to arrive at

$$\mathbf{J}_\Psi \vec{u} \geq 0 \text{ and } \vec{f}_\Psi = \mathbf{J}_\Psi^T \vec{\lambda} \tag{2.21}$$

By the principle of superposition, the normal and frictional impulses can be applied to a body by adding their components. By (2.20) and (2.21) we get that the contact impulse $\vec{\kappa}$ for one body can be expressed as

$$\vec{\kappa} = \mathbf{J}_n^T \lambda_n + \mathbf{J}_f^T \vec{\lambda}_f \tag{2.22}$$

where λ_n and $\vec{\lambda}_f$ are expressed in the contact coordinate system; and $\mathbf{J}_n^T$ and $\mathbf{J}_f^T$ are the corresponding Jacobians. This allows us to apply our knowledge of friction forces in an easy and convenient way. The price is that we will need to derive and compute the blocked Jacobians for the contact forces. The form of the Jacobians depends on the actual constraints that we set up.

2.1.3 Contact Impulses

The preceding section showed that contact impulses can be converted from the contact coordinate system into world coordinates and applied to the correct bodies using Jacobian matrices. This section will investigate the contact forces in the contact coordinate system.

Let us look at one contact between two rigid bodies B_i and B_j where we assume $i < j$. As noted earlier, it follows from Newton's 1st and 3rd Law that non-penetration requires that an impulse equal in magnitude to the impulse pushing B_i into B_j and of opposite direction is applied to B_i. This direction is the surface normal of B_j at the contact point. If an impulse is applied to B_i perpendicular to the contact normal, a friction impulse needs to be applied. This suggests that a coordinate system consisting

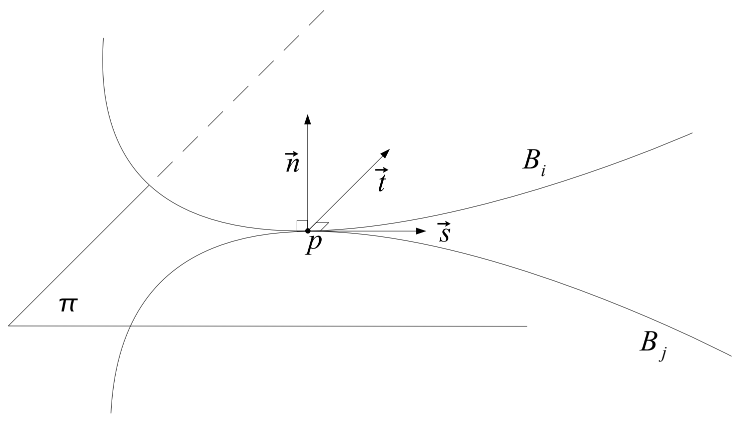


Figure 2.3: *This figure illustrates the contact coordinate system. Two bodies B_i and B_j are in contact. The contact normal $\vec{n}$ is the surface normal of B_j. The contact plane π is placed perpendicular to $\vec{n}$ so that the contact point p is in the plane. The contact point p is the origin of the contact coordinate system.*

of the normal vector $\vec{n}$ and a plane π perpendicular to $\vec{n}$ placed at the contact point is a logical way to express the contact impulses. This is illustrated on Figure 2.3. While it would seem natural to let π be spanned by two vectors, $\vec{s}$ and $\vec{t}$, some methods span this plane differently.

The Normal Impulse

One nice feature of this definition of the contact coordinate system is that we can require the magnitude of the normal impulse λ_n to be non-negative. If v_n is the magnitude of the velocity of the contact point in the normal direction, one of two situations must exist:

1. If $\lambda_n > 0$, the bodies cannot be moving away from each other, that is, $v_n = 0$.
2. If the bodies are moving away from each other, i.e. $v_n > 0$, there can be no contact impulse between them, so $\lambda_n = 0$

As the bodies cannot penetrate each other, we get that $v_n \geq 0$. This can be summarized by

$$\lambda_n \geq 0 \qquad v_n \geq 0 \qquad v_n \lambda_n = 0 \tag{2.23}$$

The set of legal values for the normal impulse can be defined by

$$\mathcal{N} = \{\nu \in \mathcal{R} \mid \nu \geq 0\} \tag{2.24}$$

where ν is some legal normal impulse.

Let us assume that $\lambda_n > 0$, from which it follows that $v_n = 0$. If ν_2 is a legal normal impulse larger than or equal to λ_n, the difference $\nu_2 - \lambda_n$ is a non-negative

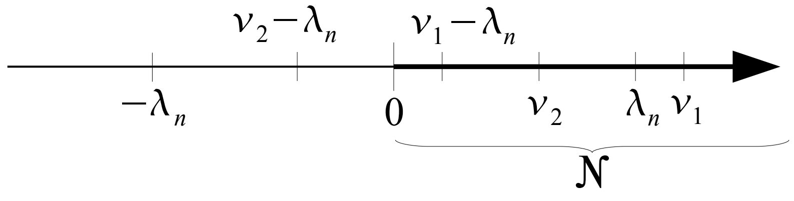


Figure 2.4: *The figure illustrates the value of the difference $\nu - \lambda_n$ for different values of ν.*

number. If ν_1 is smaller than λ_n, then $\nu_1 - \lambda_n$ is a negative number, as illustrated on Figure 2.4. In both cases, however, the product $(\nu - \lambda_n)\, v_n$ is equal to 0.

If $\lambda_n = 0$, the difference $\nu - \lambda_n$ is a non-negative number. Thus, the product $(\nu - \lambda_n)\, v_n \geq 0$. It follows that the complementarity conditions (2.23) are equivalent with the variational inequality

$$(\nu - \lambda_n)\, v_n \geq 0, \text{ for } \forall\, \nu \in \mathcal{N} \text{ and } \lambda_n \in \mathcal{N} \tag{2.25}$$

This property will come in handy when we take a look at the proximal map formulation in Section 2.3.

The Friction Impulses

The friction impulse $\vec{\lambda}_f$ is a bit more complicated. From Coulomb's Friction Law the friction impulse depends on the relative movement of the bodies as well as the surface between them. The magnitude of the friction impulse is bounded by the magnitude of the normal impulse so that

$$\| \vec{\lambda}_f \| \leq \mu\lambda_n \tag{2.26}$$

If B_i is at rest w.r.t. B_j, μ is called *coefficient of static friction* μ_s. If not, μ is called *coefficient of kinematic friction* μ_k. In general, $\mu_s \neq \mu_k$, but for clarity μ will be used for both. It should also be noted that the value of μ depends on both surfaces in a non-linear fashion. If $\mathcal{F}$ is the set of legal values of the friction impulses $\vec{\phi}$ due to Coulomb's Friction Law, we have

$$\mathcal{F}(\mu\vec{\lambda}_n) = \{\vec{\phi} \in \mathbb{R}^2 |\ \| \vec{\phi} \| \leq \mu\lambda_n\} \tag{2.27}$$

If μ is assumed independent of the direction of the external impulse or velocity of B_i at the contact point, the set of legal values for the magnitude of the friction impulse will be a circle on π with radius $\mu\lambda_n$ and centre at the contact point. This type of friction is called *isotropic friction*. In this case, the direction of the friction impulse will be in the opposite direction of the external force or velocity at the contact point.

However, μ may very well be dependent on the direction of the external force or velocity at the contact point due to surface features. This type of friction is called *anisotropic friction*. It follows that the set of legal magnitudes will no longer be circular. It may be elliptic or even an arbitrarily convex set. In this case, we will need to use the principle of maximum dissipation to compute the actual direction of the friction impulse. The *principle of maximum dissipation* states that the friction impulse should dissipate as much energy from the system as possible.

The work W of an impulse $\vec{P}$ is given by

$$W = \vec{P}\vec{v} \tag{2.28}$$

where $\vec{v}$ is the velocity. The work from the friction impulse counteracts the current movement, and is therefore dissipative. Thus, the work of the friction impulse must be smaller than zero. It follows that

$$W_f = \vec{\lambda}_f \vec{v}_f < 0. \tag{2.29}$$

Armed with equation (2.29) the principle of maximum dissipation can be applied. Let $\vec{\phi}$ be a legal friction impulse, and $\vec{\lambda}_f$ the friction impulse that removes the most energy from the system. If W_{λ_f} is the work dissipated from the system by $\vec{\lambda}_f$ and W_ϕ is the work dissipated by $\vec{\phi}$, it follows that

$$\begin{aligned} W_{\lambda_f} &\leq W_\phi \Leftrightarrow & (2.30)\\ 0 &\leq W_\phi - W_{\lambda_f} \Leftrightarrow & (2.31)\\ 0 &\leq \vec{\phi}\vec{v}_f - \vec{\lambda}_f \vec{v}_f \Leftrightarrow & (2.32)\\ 0 &\leq \left(\vec{\phi} - \vec{\lambda}_f\right) \vec{v}_f & (2.33) \end{aligned}$$

Thus, the principle of maximum dissipation can be summarized by

$$\left(\vec{\phi} - \vec{\lambda}_f\right) \vec{v}_f \geq 0, \text{ for } \forall \vec{\phi} \in \mathcal{F} \text{ and } \vec{\lambda}_f \in \mathcal{F} \tag{2.34}$$

This inequality is actually a variational inequality, and in its acceleration-based form it is sometimes called *the maximum power inequality* and hold for both isotropic and anisotropic friction [GRP89].

Interdependency of the Normal and Frictional Impulses

One might be tempted to think that while the normal impulses have a direct influence on the bounds on the friction impulses, the friction impulses do not have any influence on the normal impulses. This, however, is wrong, as illustrated by an example due to [KSJP08].

Consider a house of cards. If we take a look on a card resting on the ground with only one neighbour, as illustrated on Figure 2.5. Aside from gravity, working on the center of mass, it is affected by contact impulses from the cards above and next to it, as well as a normal impulse from the ground. The friction impulse between the ground and the bottom of the card keeps it from slipping.

When the friction impulse is applied at the contact point, a torque is induced around the center of mass of the card. This torque can push the bottom of the card into the ground, so the normal impulse must be increased to satisfy non-penetration. This interdependence of the normal and frictional impulses will have a significant importance when formulating and working with iterative schemes to solve the contact problem.

Contensou Friction

In [LG03] a result by Contensou is noted, where the magnitude of the friction impulse is also affected by the spin of the body. He discovered that in some configurations, the friction impulse could vanish if the ratio $\frac{v}{\omega R}$ converged towards zero, where R is the

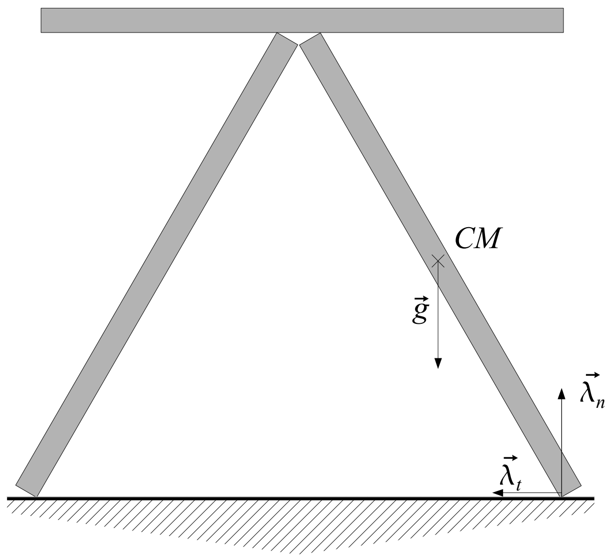


Figure 2.5: *The figure illustrates the coupling between the solution of frictional and normal impulses in the ith iteration. When the normal impulses $\vec{\lambda}_n$ are updated, the bounds on the friction impulses $\vec{\lambda}_f$ change, and when the friction impulses change, the normal impulses may need to change due to a change in torque.*

effective radius of the contact. This is a problem as we have used Amonton's 2nd Law to ignore the effect of the size of the contact. Furthermore, as we use contact points, we have no prior knowledge of the size of the contact. [LG03] sidestep this issue by dividing the slip torque τ_n by R which yields

$$\lambda_\tau = \frac{\tau_n}{R} \tag{2.35}$$

and a spin slip velocity

$$v_\tau = \omega R. \tag{2.36}$$

Thus, the radius of the contact has been absorbed by our unknowns. For isotropic friction, [LG03] shows that λ_τ is bounded by

$$\lambda_\tau \leq \frac{2}{3}\mu\lambda_n. \tag{2.37}$$

The friction impulse can be expanded to include λ_τ. The maximum power inequality holds for friction spin forces [GRP89], so (2.34) still holds.

2.1.4 Handling Collisions

For rigid bodies, collisions can be expressed by Newton's Collision Law. If the bodies B_i and B_j have the velocities v_i and v_j at the contact point along the contact normal, the pre-collision and post-collision velocities are linked by a *coefficient of restitution* ε, defined by

$$(v_i^{t+\Delta t} - v_j^{t+\Delta t}) = -\varepsilon(v_i^t - v_j^t) \tag{2.38}$$

where $0 \leq \varepsilon \leq 1$ can be viewed as a constant that models the internal structure of the rigid body. If $\varepsilon = 0$, it can be seen as modeling a perfect inelastic collision, where B_i and B_j "melt" together, and $\varepsilon = 1$ models a perfect elastic collision, where no kinetic energy is lost in the collision. Like the coefficient of friction, ε is a property of the materials in the collision. From one timestep to the next, $v_i^t - v_j^t$ is the velocity of the contact point in the normal direction v_n, from which it follows that

$$v_n^{t+\Delta t} = -\varepsilon v_n^t \tag{2.39}$$

Note that this velocity update may only be applied in contact coordinates, so we still need to convert it to world coordinates and apply it to the respective bodies.

2.2 Complementarity Problems

A complementarity problem can be defined as follows.

Definition 2.1 (Complementarity Problem) *Given two vectors $\vec{x} \in \mathcal{R}^n$ and $\vec{y} \in \mathcal{R}^n$, the complementarity problem is to find $\vec{x}$ so that the equations*

$$\vec{x} \geq 0 \qquad \vec{y} \geq 0 \qquad \vec{x}^{\,T}\vec{y} = 0 \tag{2.40}$$

are satisfied.

In the complementarity formulations that will be used in this thesis, $\vec{y}$ depends on $\vec{x}$. Furthermore, $\vec{y}$ will describe the non-holonomic constraints on $\vec{x}$.

2.2.1 The Linear Complementarity Problem

A Linear Complementarity Problem (LCP) is a complementarity problem where $\vec{y} = \mathbf{A}\vec{x}+\vec{b}$. $\mathbf{A} \in \mathcal{R}^{n\times n}$ is a known, square matrix and $\vec{b} \in \mathcal{R}^n$ [ST96]. From Definition 2.1 this leads to the definition

Definition 2.2 (Linear Complementarity Problem) *Given two vectors $\vec{x} \in \mathcal{R}^n$, $\vec{b} \in \mathcal{R}^n$ and $\mathbf{A} \in \mathcal{R}^{n\times n}$, the Linear Complementarity Problem (LCP) is to find $\vec{x}$ so that the equations*

$$\vec{x} \geq 0 \qquad \mathbf{A}\vec{x} + \vec{b} \geq 0 \qquad \vec{x}^{\,T}(\mathbf{A}\vec{x} + \vec{b}) = 0 \tag{2.41}$$

are satisfied.

As noted in [CPS09] an LCP has a unique solution if $\mathbf{A}$ is positive definite. If $\mathbf{A}$ is symmetric and positive semi-definite and $\vec{x}$ is a solution to the LCP, the product $\mathbf{A}\vec{x}$ is unique. A matrix is positive definite if all eigenvalues are positive. Thus, if $\mathbf{A}$ can be expressed as $\mathbf{K}^{-1}\mathbf{D}\mathbf{K}$, where $\mathbf{D}$ is a diagonal matrix containing the eigenvalues of $\mathbf{A}$ and $\mathbf{K}$ is a matrix consisting of the corresponding eigenvectors, and $\mathbf{D}$ contains only positive entries, $\mathbf{A}$ is positive definite. If $\mathbf{D}$ contains elements that are zero, $\mathbf{A}$ is positive semi-definite.

The contact problem was formulated as an LCP by Steward and Trinkle [ST96], even though Baraff used an LCP formulation to compute the normal impulses [Bar94].

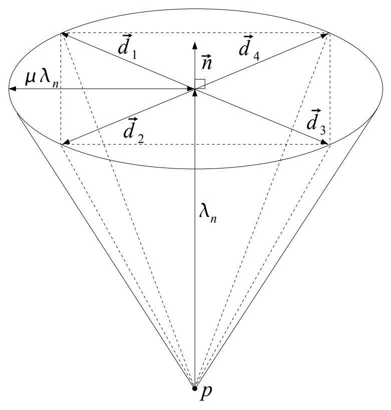


Figure 2.6: *The figure shows the friction cone and the smallest possible LCP polyhedral approximation, using 4 basis vectors $\vec{d}_1, \vec{d}_2, \vec{d}_3, \vec{d}_4$ to span the friction plane. The apex of the cone is placed at the contact point between two bodies. Using Coulomb friction, all legal contact impulses must be inside the cone.*

If the friction is isotropic and obey Coulomb's Friction Law, the friction will be applied in a plane perpendicular to $\vec{n}$. The magnitude of the friction force is bounded by the product $\mu\lambda_n$. The bounds on the contact force can be visualized by a friction cone whose height is the magnitude of the normal force and has base radius of $\mu\lambda_n$. The friction cone can be approximated by a polyhedral cone $\mathcal{F}$ where

$$\mathcal{F} = \{\lambda_n\vec{n} + \mathbf{D}\vec{\beta} \mid c_n \geq 0, \vec{\beta} \geq 0, \vec{e}^{\,T}\vec{\beta} \leq \mu c_n\} \text{ where } \vec{e} = [1, 1, ..., 1]^T \in \mathcal{R}^k \quad (2.42)$$

The matrix $\mathbf{D}$ is the direction vectors that span the approximation of the base of the friction cone, and $\vec{\beta}$ is a vector of non-negative weights. Thus, the product $\mathbf{D}\vec{\beta}$ is the approximated friction force. This last condition in (2.42) expresses that the sum of the weights must be smaller than the bound on the friction force. The friction cone and the polyhedral approximation is illustrated on Figure 2.6.

The velocity-based velocity update (2.14) can be used to derive the linear complementarity formulation. Note that it is necessary to add an extra unknown (aside from the magnitudes in c_n and $\vec{\beta}$) to be able to solve the linear complementarity problem. The LCP for the contact problem can be solved using Lemke's algorithm.

The LCP formulation has the advantage that it can be proved that it has a solution. Another nice feature is that the ability to tune the quality of the friction cone allows the programmer to make a tradeoff between speed and quality. This is, however, also the weakness of the LCP formulation. For each contact point at least four vectors needs to be computed and saved, and at least six unknowns needs to be computed. This is a lot of memory, considering that Coloumb friction is a two dimensional problem, that is, the friction impulse can be described by its magnitude and direction in the contact plane.

2.2.2 The Nonlinear Complementarity Problem

If it were acceptable to use the lowest quality approximation of the friction cone in the LCP, one could accept that the magnitudes $\vec{\beta}$ were negative, and thereby reduce the

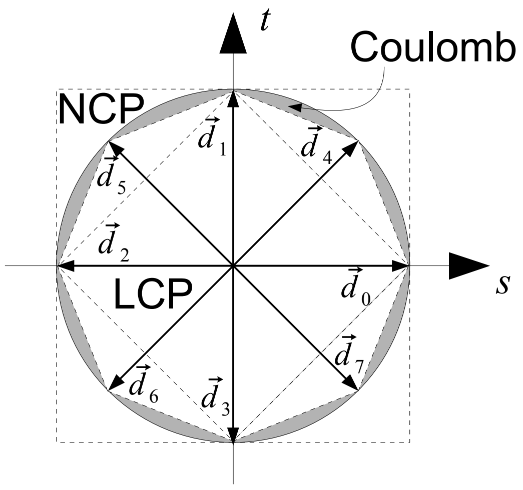


Figure 2.7: *The figure compares the friction model of the LCP and the NCP with Coulomb's Law. Coulomb's Law is shown by the grey circle. The outer white box shows the legal friction values for the NCP, and the white box inside the grey circle show the legal friction values for the LCP. The vectors $\vec{d}_0, \vec{d}_1, \vec{d}_2, \vec{d}_3$ span the minimum LCP region, and vectors $\vec{d}_4, \vec{d}_5, \vec{d}_6, \vec{d}_7$ can be used for an improved approximation.*

memory usage to two vectors and two magnitudes. To couple the generalized contact velocities with the contact forces, the generalized contact velocities must be separated into two variables by their sign. This leads to a Non-linear Complementarity Problem (NCP), where the non-linearity is in the complementarity conditions. A rigorous derivation of the NCP formulation of the contact problem can be found in [Erl07] and [PNE10]. The NCP can be reduced to solving a linear system of equations where the constraints can be handled by a projection. An iterative matrix solver, such as the projected Gauss–Seidel method, can be used to make a very fast implementation, where the number of iterations give a tradeoff between accuracy and time usage.

Unlike the LCP, there are no theoretical results that guarantee existence nor uniqueness of a solution for the NCP. Indeed, results in [SNE09] show slow convergence of the frictional components, and [PNE10] suggests that this may be due to a flawed friction model that allows the friction force to take values that break Coulomb's Law. Figure 2.7 illustrates the differences in the LCP and NCP friction models.

2.3 The Prox Formulation

The problem with the flawed friction model in both the LCP and NCP formulations of the contact problem suggests that it could be an advantage to use a more detailed and correct friction model. In [Ani06] a Cone Complementarity Problem is suggested. However, not all types of friction can be adequately modelled using Coulomb's Law. The surface structure of a body could yield a high coefficient of friction in one direction, but a small one in another direction, or even discontinuous change of μ.

This section will derive another formulation of the contact problem. The focus of this model is a physically realistic friction model with the ability to simulate complex friction forces. This model uses proximal maps, and is dubbed the prox formulation.

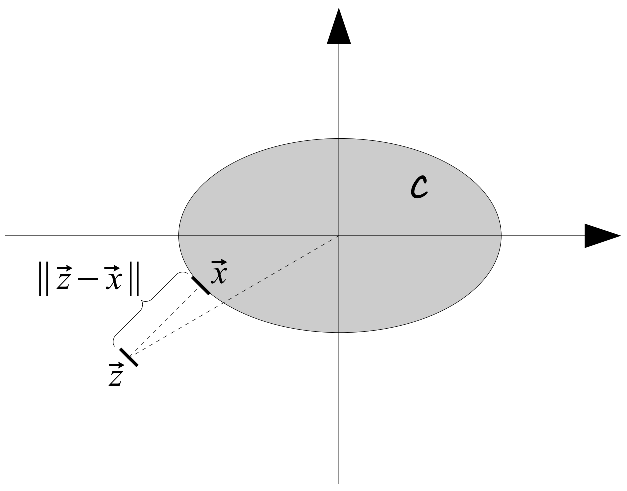


Figure 2.8: *The figure illustrates the notion of a proximal map in 2 dimensions. The convex set $\mathcal{C}$ is bounded by an ellipse. The point $\vec{z}$ is outside $\mathcal{C}$, and the proximal point $\vec{x}$ is the point in $\mathcal{C}$ that minimizes $||\vec{z}-\vec{x}||^2$.*

Proximal maps were originally defined in [Mor65]. Definition 2.3 below is slightly different, but equivalent. A proximal map in 2 dimensions is illustrated on Figure 2.8.

Definition 2.3 (Proximal Map) *If $\mathcal{C}$ is a closed convex set, and the element $\vec{z}$ is an element that may be in or outside $\mathcal{C}$, and the proximal point $\vec{x}$ is the element in $\mathcal{C}$ closest to $\vec{z}$. Thus, a proximal map can be defined by*

$$\mathsf{prox}_{\mathcal{C}}(\vec{z}) = \min_{x\in\mathcal{C}} ||\vec{z}-\vec{x}||^2 \tag{2.43}$$

The proximal point problem is the problem of solving $\vec{x} = \mathsf{prox}_{\mathcal{C}}(\vec{z})$.

In one dimension, the proximal point problem is particularly easy to solve. The set $\mathcal{C}$ of legal values are bounded by an upper and lower bound, y_{min} and y_{max}. The solution x is

$$x = \begin{cases} y_{min} & \text{if } z \leq y_{min} \\ z & \text{if } z \in \mathcal{C} \\ y_{max} & \text{if } z \geq y_{max} \end{cases} \tag{2.44}$$

The one dimensional proximal point problem is illustrated on Figure 2.9.

In higher dimensions, the proximal point problem is significantly harder to solve. If the convex set is an axis parallel box, the one dimensional solution can be applied to each axis. For more complex sets, more complex methods must be employed. In some cases, numerical solutions are needed.

If the convex set $\mathcal{C}$ is known in advance, it may be possible to derive a general solution for a set of that particular type. Let us assume that the border of $\mathcal{C}$ is described by the function f and that $\vec{z}$ is outside of $\mathcal{C}$ at an unknown distance d from $\mathcal{C}$. Thus, $\vec{x}$ is placed on the surface of $\mathcal{C}$, so $f(\vec{x}) = 0$. Note that the form of f may become much

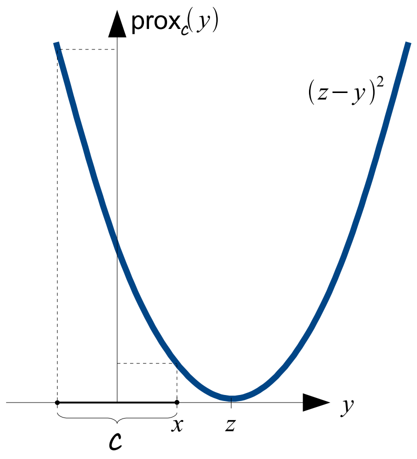


Figure 2.9: *The figure illustrates the proximal point problem in 1 dimension for a set $\mathcal{C}$ with $x = \text{prox}_{\mathcal{C}}(z) = (z-y)^2$.*

simpler if $\mathcal{C}$ is translated so that it contains the origin. As $\vec{x}$ is the point on $\mathcal{C}$ closest to $\vec{z}$, the direction of the vector from $\vec{x}$ to $\vec{z}$ must have the direction of the surface normal of $\mathcal{C}$ at $\vec{x}$. If d is the distance from $\mathcal{C}$ to $\vec{z}$, it follows that

$$\vec{z} - \vec{x} = d\,\nabla f(\vec{x}) \tag{2.45}$$

If $\nabla f(\vec{x})$ is computed in (2.45), it may be possible to isolate $\vec{x}$. In this case, the n dimensional $\vec{x}$ is now described by just one variable. Setting the value of $\vec{x}$ into f noting that $f(\vec{x}) = 0$ allows us to define a function $g(d) = f(\vec{x}) = 0$. Thus, the roots of $g(d)$ can be inserted into (2.45) to yield the solution. It is noted that g may or may not have analytical solutions. It has been assumed that f is continuous and differentiable almost everywhere. It may be needed to handle discontinuities separately.

If $\mathcal{C}$ is not known in advance or has a complex form making an analytical solution difficult or expensive to compute, one can use the Gilbert-Johnson-Keerthi (GJK) algorithm presented in [GJK88]. The GJK algorithm is a general method to compute the distance between two closed convex sets A and B. The GJK algorithm computes the Minkowski difference between A and B, and then computes the distance from the origin to $A - B$. Computing the Minkowski difference can be quite expensive, but the GJK algorithm employs a clever geometric trick to avoid explicit computation of the Minkowski difference.

2.3.1 Proximal Maps and Variational Inequalities

One interesting property of proximal maps is that they can be shown to be equivalent with variational inequalities. Let us start by defining the variational inequality

Definition 2.4 (Variational Inequality) *Let $\mathcal{C}$ be a closed convex set, and let f be a continuous function $f : \mathcal{C} \to \mathcal{R}^n$, the variational inequality VI($\vec{f}, \mathcal{C}$) is given by*

$$\vec{f}(\vec{x})(\vec{y} - \vec{x}) \geq 0 \text{ for } \forall \vec{y} \in \mathcal{C} \tag{2.46}$$

where the unknown $\vec{x} \in \mathcal{C}$.

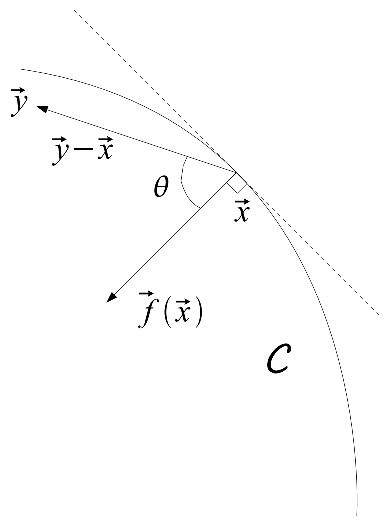


Figure 2.10: *The figure illustrates a variational inequality in two dimensions. The only way the dot product $\vec{f}(\vec{x}) \cdot (\vec{y} - \vec{x})$ can be non-negative is if the angle θ between the vectors is smaller than or equal to $90°$. As $\vec{y}$ can be any point in $\mathcal{C}$, this requires that $\vec{x}$ is on the surface of $\mathcal{C}$ and that $\vec{f}(\vec{x})$ points into $\mathcal{C}$ perpendicular to the tangent of $\mathcal{C}$ at $\vec{x}$.*

The variational inequality is illustrated in two dimensions on Figure 2.10. It will now be shown that for all $r > 0$ the Variational Inequality has the same solution as $\text{prox}_{\mathcal{C}}(\vec{x} - r\vec{f}(\vec{x}))$.

Let us take a look at (2.46). For the dot product of two vectors to be non-negative, the maximum angle between them must be smaller than or equal to $90°$. If $\vec{x}$ is on the inside of $\mathcal{C}$, the vector $\vec{y} - \vec{x}$ can have any direction, and (2.46) cannot be satisfied unless $\vec{f}(\vec{x})$ is zero. Thus, $\vec{x}$ must be on the surface of $\mathcal{C}$. For (2.46) to be true for all $\vec{y} \in \mathcal{C}$, we must require that $\vec{f}(\vec{x})$ is perpendicular to the surface of $\mathcal{C}$ at $\vec{x}$ and pointing into $\mathcal{C}$.

By Definition 2.3 we know that $\vec{z} = \vec{x} - r\vec{f}(\vec{x})$. Thus, the vector from $\vec{x}$ to $\vec{z}$ must be

$$(\vec{z} - \vec{x}) = \vec{x} - r\vec{f}(\vec{x}) - \vec{x} \tag{2.47}$$

$$= -r\vec{f}(\vec{x}) \tag{2.48}$$

As $\mathcal{C}$ is a convex set, the shortest distance from the surface of $\mathcal{C}$ to a point $\vec{z}$ outside $\mathcal{C}$ will be along the normal of $\mathcal{C}$ at the closest point $\vec{x}$. If $\vec{y} \in \mathcal{C}$, the angle between the vector from $\vec{x}$ to $\vec{z}$ and the vector from $\vec{x}$ to $\vec{y}$ will be equal to or larger than $90°$ for all $\vec{y}$. As illustrated on Figure 2.11, it follows that

$$(\vec{z} - \vec{x}) \cdot (\vec{y} - \vec{x}) \leq 0 \tag{2.49}$$

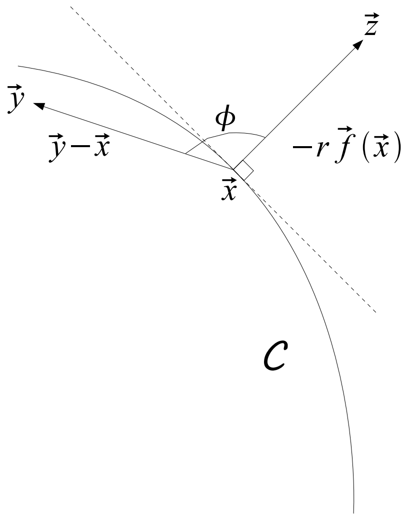


Figure 2.11: *The figure illustrates how a proximal map relates with variational inequalities. As can be seen, the angle ϕ between $\vec{y}-\vec{x}$ and $\vec{z}-\vec{x}$ is equal to or larger than 90° for all $\vec{y}$. Thus, the dot product $(\vec{y}-\vec{x})\cdot(\vec{z}-\vec{x})$ will be larger than or equal to zero.*

Inserting (2.48) into (2.49) yields

$$-r\vec{f}(\vec{x})\cdot(\vec{y}-\vec{x}) \leq 0 \Leftrightarrow \tag{2.50}$$

$$r\vec{f}(\vec{x})\cdot(\vec{y}-\vec{x}) \geq 0 \Leftrightarrow \tag{2.51}$$

$$\vec{f}(\vec{x})\cdot(\vec{y}-\vec{x}) \geq 0 \tag{2.52}$$

Thus, the variational inequality in Definition 2.4 is equivalent with the proximal mapping $\mathsf{prox}_{\mathcal{C}}(\vec{x}-r\vec{f}(\vec{x}))$. This result is summarized by Theorem 2.1.

Theorem 2.1 *The variational inequality VI($\vec{f},\mathcal{C}$) is equivalent with the proximal mapping* $\mathsf{prox}_{\mathcal{C}}(\vec{x}-r\vec{f}(\vec{x}))$ *for $r>0$.*

2.3.2 The Contact Problem Using Proximal Maps

This section will express the contact problem using proximal maps. The normal impulses can be expressed by the variational inequality (2.25). By Theorem 2.1, the normal impulses can be described by the proximal map

$$\lambda_n = \mathsf{prox}_{\mathcal{N}}(\lambda_n - rv_n) \tag{2.53}$$

Similarly, the friction impulses are given by the variational inequality (2.34). By Theorem 2.1, the friction impulses can be described by the proximal map

$$\vec{\lambda}_f = \mathsf{prox}_{\mathcal{F}}(\vec{\lambda}_f - r\vec{v}_f) \tag{2.54}$$

Note that the only assumptions on the form of $\mathcal{F}$ used to derive (2.54) is that $\mathcal{F}$ is a closed convex set and obeys the principle of maximum dissipation. This makes the prox formulation able to handle a large set of possible friction configurations.

The contact impulses have been formulated using proximal maps. However, we still need to find the Jacobians. The Jacobians are based on the constraints on the contact impulses. As penetration is illegal, the closest distance between two bodies must be larger than or equal to zero. Similarly, the time derivative of the closest distance must also be larger than or equal to zero, as anything else would imply that one body is moving into another. Let us compute the velocity of the contact point $\vec{v}_k$ between bodies B_i and B_j where we use the convention that $i < j$. If $\vec{v}_i$ and $\vec{v}_j$ are the velocities of the center of mass of the bodies B_i and B_j, $\vec{\omega}_i, \vec{\omega}_j$ are their angular momenta, and $\vec{r}_i, \vec{r}_j$ are the vectors from their center of mass to the contact points, the contact velocity is given by

$$\vec{v}_k = (\vec{v}_j + \vec{\omega}_j \times \vec{r}_j) - (\vec{v}_i + \vec{\omega}_i \times \vec{r}_i) \tag{2.55}$$

in the world coordinate system. We are, however, interested in finding $\vec{v}_k$ in contact coordinates. If basis vectors of the new coordinate system are known in the coordinates of the old coordinate system, the components of a vector $\vec{v}$ in the new coordinate system can be found by computing the dot product of $\vec{v}$ and each of the basis vectors of the new coordinate system. If $\vec{e}$ is a basis vector we note that

$$(\vec{\omega} \times \vec{r}) \cdot \vec{e} = \vec{\omega} \cdot (\vec{r} \times \vec{e}) = (\vec{r} \times \vec{e}) \cdot \vec{\omega} \tag{2.56}$$

Thus, the component of $\vec{v}_{k_e}$ using the basis vector $\vec{e}$ is

$$\begin{aligned}
\vec{v}_{k_e} &= ((\vec{v}_j + \vec{\omega}_j \times \vec{r}_j) - (\vec{v}_i + \vec{\omega}_i \times \vec{r}_i)) \cdot \vec{e} && (2.57)\\
&= (\vec{v}_j + \vec{\omega}_j \times \vec{r}_j) \cdot \vec{e} - (\vec{v}_i + \vec{\omega}_i \times \vec{r}_i) \cdot \vec{e} && (2.58)\\
&= \vec{e} \cdot \vec{v}_j + (\vec{r}_j \times \vec{e}) \cdot \vec{\omega}_j - \vec{e} \cdot \vec{v}_i - (\vec{r}_i \times \vec{e}) \cdot \vec{\omega}_i && (2.59)
\end{aligned}$$

On matrix form (2.59) becomes

$$\vec{v}_{k_e} = \begin{bmatrix} -\vec{e}^T & -(\vec{r}_i \times \vec{e})^T & e^T & (\vec{r}_j \times \vec{e})^T \end{bmatrix} \begin{bmatrix} \vec{v}_i \\ \vec{\omega}_i \\ \vec{v}_j \\ \vec{\omega}_j \end{bmatrix} \tag{2.60}$$

Equation (2.60) holds for the basis of the contact coordinate system $\{\vec{n}, \vec{s}, \vec{t}\}$. The Contenseau friction component, however, acts in a different manner. The contact velocity due to Contenseau friction is simply a spin around $\vec{n}$. Thus, v_{c_τ} is the magnitude of the spin around $\vec{n}$ from which it follows that it is $\vec{\omega}_j - \vec{\omega}_i$ projected onto $\vec{n}$. On matrix form, the velocity of the contact point between bodies B_i and B_j is

$$\vec{v}_k = \begin{bmatrix} -\vec{n}^T & -(\vec{r}_i \times \vec{n})^T & \vec{n}^T & (\vec{r}_j \times \vec{n})^T \\ -\vec{s}^T & -(\vec{r}_i \times \vec{s})^T & \vec{s}^T & (\vec{r}_j \times \vec{s})^T \\ -\vec{t}^T & -(\vec{r}_i \times \vec{t})^T & \vec{t}^T & (\vec{r}_j \times \vec{t})^T \\ \vec{0}^T & -\vec{n}^T & \vec{0}^T & \vec{n}^T \end{bmatrix} \begin{bmatrix} \vec{v}_i \\ \vec{\omega}_i \\ \vec{v}_j \\ \vec{\omega}_j \end{bmatrix} \tag{2.61}$$

The velocity of the contact point is the time derivative of the non-penetration constraint which is known to be non-negative, and the matrix to the left in (2.61) is the generalized velocity vector. Thus, we recognize the non-holonomic constraint (2.21) for the contact point between the bodies B_i and B_j, and note that the matrix to the right in (2.61) is the Jacobian for the contact point. From (2.61) it can be seen that the Jacobian for each contact point can be constructed from two blocks. For convenience, let us define two 4×6 blocks for the kth contact point

$$\mathbf{J}_{ki} = \begin{bmatrix} -\vec{n}^T & -(\vec{r}_i \times \vec{n})^T \\ -\vec{s}^T & -(\vec{r}_i \times \vec{s})^T \\ -\vec{t}^T & -(\vec{r}_i \times \vec{t})^T \\ \vec{0}^T & -\vec{n}^T \end{bmatrix} \tag{2.62}$$

and

$$\mathbf{J}_{kj} = \begin{bmatrix} \vec{n}^T & (\vec{r}_j \times \vec{n})^T \\ \vec{s}^T & (\vec{r}_j \times \vec{s})^T \\ \vec{t}^T & (\vec{r}_j \times \vec{t})^T \\ \vec{0}^T & \vec{n}^T \end{bmatrix} \tag{2.63}$$

With these definitions, the Jacobian $\mathbf{J}$ for the system can be constructed by letting each row in $\mathbf{J}$ correspond to one contact point, with $\mathbf{J}_{ki}$ placed at the ith column and $\mathbf{J}_{kj}$ at the jth column of the kth row. For a system of K contact points and M bodies, $\mathbf{J}$ would be a $K \times M$ matrix with two non-zero 4×6 blocks per row. Equation (2.61) generalizes to

$$\begin{bmatrix} \vec{v}_1 \\ \vdots \\ \vec{v}_k \\ \vdots \\ \vec{v}_K \end{bmatrix} = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 \dots 0 & \mathbf{J}_{ki} & 0 \dots 0 & \mathbf{J}_{kj} & 0 \dots 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} \vec{u}_1 \\ \vdots \\ \vec{u}_i \\ \vdots \\ \vec{u}_j \\ \vdots \\ \vec{u}_M \end{bmatrix} \tag{2.64}$$

From (2.21) it follows that the contact impulses $\vec{\kappa}$ are

$$\vec{\kappa} = \mathbf{J}^T \vec{\lambda} \tag{2.65}$$

where $\vec{\lambda}$ are the contact impulses in the contact coordinate system. The velocity update (2.14) becomes

$$\vec{u}^{t+\Delta t} \approx \vec{u}^t + \mathbf{M}^{-1}(\vec{h}_{ext} + \mathbf{J}^T \vec{\lambda}) \tag{2.66}$$

Thus, the velocity of the contact points at the time $t + \Delta t$ can be approximated by

$$\begin{aligned} \vec{v}^{t+\Delta t} &= \mathbf{J}\vec{u}^{t+\Delta t} & (2.67) \\ &\approx \mathbf{J}\vec{u}^t + \mathbf{J}\mathbf{M}^{-1}(\vec{h}_{ext} + \mathbf{J}^T\vec{\lambda}) & (2.68) \\ &= \underbrace{\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T}_{\mathbf{A}}\vec{\lambda} + \underbrace{\mathbf{J}(\vec{u}^t + \mathbf{M}^{-1}\vec{h}_{ext})}_{\vec{b}} & (2.69) \end{aligned}$$

As we are now looking at the velocity of the contact point, (2.39) can now be applied by noting that the contact velocity update due to collision $\vec{v}_k$ is

$$\vec{v}_c^{t+\Delta t} = -\varepsilon \mathbf{J}\vec{u}^t \tag{2.70}$$

where ε is a diagonal matrix consisting of 4×4 blocks that contain one non-zero element at $(0, 0)$. As the velocity update due to collisions is independent of the contact impulses, the velocity update can be applied to $\vec{b}$. With collision velocity updates $\vec{b}$ becomes

$$\vec{b} = \mathbf{J}(\vec{u}^t + \mathbf{M}^{-1}\vec{h}_{ext}) - \varepsilon\mathbf{J}\vec{u}^t \tag{2.71}$$

$$= \mathbf{J}((\mathbf{1} - \varepsilon)\vec{u}^t + \mathbf{M}^{-1}\vec{h}_{ext}) \tag{2.72}$$

Note that the very peculiar form of ε can be utilized for an efficient implementation to conserve storage and reduce the number of computations. With these results, the proximal map formulation for the kth contact point can be rewritten as

$$\begin{bmatrix} \lambda_{n,k} \\ \vec{\lambda}_{f,k} \end{bmatrix} = \begin{bmatrix} \mathsf{prox}_{\mathcal{N}}(\lambda_{n,k} - r_{n,k}v_{n,k}) \\ \mathsf{prox}_{\mathcal{F}}(\vec{\lambda}_{f,k} - r_{f,k}\vec{v}_{f,k}) \end{bmatrix} \Leftrightarrow \tag{2.73}$$

$$\vec{\lambda}_k = \mathsf{prox}_{\mathcal{N},\mathcal{F}}(\vec{\lambda}_k - \mathbf{R}_k(\mathbf{A}\vec{\lambda} + \vec{b})_k) \tag{2.74}$$

where the r-factors have been gathered into a diagonal matrix $\mathbf{R}$. Equation (2.74) suggests that an iterative fixed point scheme can be constructed to compute $\vec{\lambda}$. For each block we compute

$$\vec{z}_k = \vec{\lambda}_k - \mathbf{R}_k(\mathbf{A}\vec{\lambda} + \vec{b})_k \tag{2.75}$$

Thus, the next guess for $\vec{\lambda}_k$ is

$$\vec{\lambda}_k = \mathsf{prox}_{\mathcal{N},\mathcal{F}}(\vec{z}_k) \tag{2.76}$$

Once $\vec{\lambda}$ has been computed, it can be used to update the system. Using explicit Euler integration, $\vec{\lambda}$ can be inserted into the velocity update in equation (2.66) to compute the new velocity $\vec{u}^{t+\Delta t}$. The updated velocity can be inserted into (2.16) to compute the new positions.

Chapter 3

Jacobi and Gauss–Seidel schemes

In the previous chapter, the contact problem was considered, and a proximal map formulation was derived. The chapter concluded by suggesting that the contact impulses $\vec{\lambda}$ could be computed using an iterative fixed point scheme. For the ith iteration, the contact impulse for the kth contact point $\vec{\lambda}_k^{i+1}$ can be found by computing

$$\vec{z}_k^{\,i+1} = \vec{\lambda}_k^i - \mathbf{R}_k(\mathbf{A}\vec{\lambda}^i + \vec{b})_k \tag{3.1}$$

and

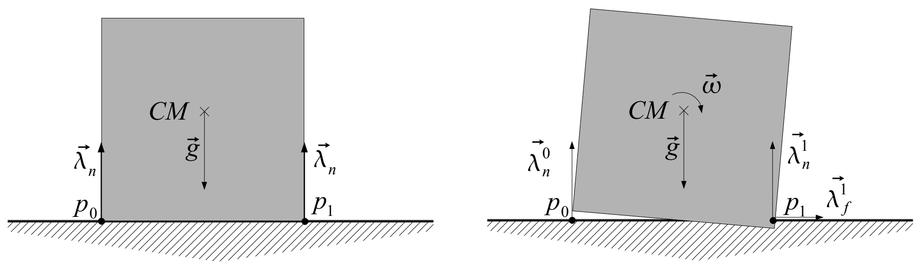
$$\vec{\lambda}_k^{i+1} = \mathsf{prox}_{\mathcal{N},\mathcal{F}}(\vec{z}_k^{\,i+1}) \tag{3.2}$$

Note that the bounds on the friction impulses $\mathcal{F}$ often depend on the magnitude of the normal impulse for that contact point, which has just been computed. One can argue for or against using the updated value of the normal impulse when computing the proximal point for the friction impulse. Using the current normal impulse enforces the friction law in use and may improve convergence behaviour, whereas using the previous normal impulse enforces non-penetration.

In this chapter, two commonly used iterative schemes will be considered. The Jacobi scheme computes updated values of $\vec{\lambda}$, using only results from the previous iteration, whereas the Gauss–Seidel scheme uses results from this iteration as the results are ready. This makes the Jacobi scheme embarrassingly parallel, whereas the Gauss–Seidel scheme is inherently sequential.

The advantage of the Gauss–Seidel scheme is that it takes the coupling between contact points into account to get faster convergence. As a simple example, consider two boxes in a stack placed on the ground. The gravity from the topmost box on the bottom box increases the normal impulse from the ground on the bottom box. If the contact points between the topmost box and the bottom box are considered first, computations on the contact points between the bottom box and the ground can take these results into account. This suggests a coupling between the contact impulses. Note that results from [PNE10] suggest that the relation is a bit more complex than in the above example.

Indeed, the coupling used by the Gauss–Seidel method can cause problems. Consider a box resting on the ground. In two dimensions, there are two contact points, p_0 and p_1 as illustrated on Figure 3.1. If the two points are considered independently of each other, both contact points will yield an upwards normal impulse. However, if one of them is considered before the other, we get stability problems.



(a) *Correct Jacobi result* **(b)** *Incorrect Gauss–Seidel result*

Figure 3.1: *The figure illustrates a situation where the Jacobi method is superior to the Gauss–Seidel method. A box is placed at rest on the ground, with p_0 and p_1 as contact points. The Jacobi method will compute an equal contact impulse at p_0 and p_1 as shown on **(a)**. When the Gauss–Seidel method uses the contact impulse from p_0 to compute the contact impulse at p_1, $\vec{\lambda}_{n,0}$ will induce a rotation around CM, which leads to a different normal impulse $\vec{\lambda}_{n,1}$ and a friction impulse $\vec{\lambda}_{f,1}$ as shown on **(b)**.*

Let us assume that p_0 is considered first, a normal impulse is added to the box at p_0. This will induce a torque and, thus, a rotation around the center of mass. When considering p_1, this causes the need for a larger normal impulse to handle the normal impulse from the rotational movement. Furthermore, the rotational movement causes a change in the direction of movement of p_1, which will cause a friction impulse at p_1. In this simple example, the Jacobi method will converge to the correct solution, whereas the Gauss–Seidel method will converge to an incorrect solution for $\vec{\lambda}$. Note, however, that for the given example both solutions will converge towards the correct contact impulse $\vec{w}$, even though the Gauss–Seidel solution has small errors on the friction components. Stacked structures are likely to exacerbate the problem, and may lead to a situation where the Jacobi method converges, while the Gauss–Seidel method diverges.

On a side note (not investigated further in this thesis), this suggests that a more efficient scheme would be to use a Jacobi method on the set of contact points between two surfaces and a Gauss–Seidel method when iterating over the surfaces. Indeed, this idea could be generalized further to construct heuristics to detect if a particular method, such as Conjugate Gradient or a Newton method, would be well suited on a given portion of a given contact problem. A conversation with Kenny Erleben suggests that such an approach has been explored. However, to our knowledge no publications on the subject has been made.

3.1 Blocked Sparse Matrix Structure

Many of the matrices used by the prox formulation are sparse, and have a clear blocked structure. This section is a short introduction to an efficient blocked sparse matrix structure and discussion of its effects. One such structure was presented and investigated in [Ras09]. The structure is a generalization of the compressed matrix format, where the data is stored in blocks instead of as single elements. The blocks in the *data* array are ordered by row and column, so that all blocks before the ith block are from an earlier row or are in the same row but with lower column index than the ith block.

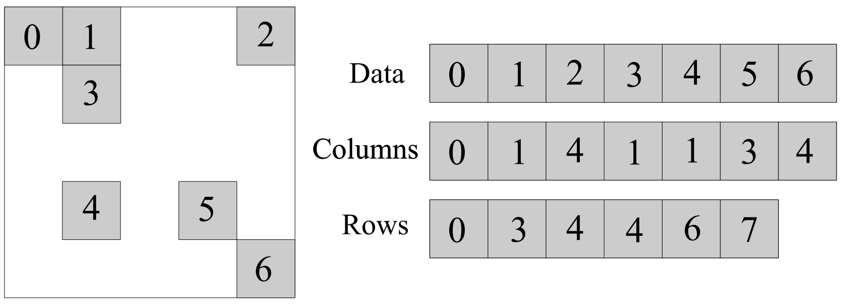


Figure 3.2: *This figure shows an example of the blocked compressed row format. The matrix on the right is the blocked matrix in dense format. The grey areas are non-zero blocks, and the white areas are all zero. The non-zero blocks are saved in the Data array. The column array contains column index of the corresponding block in the data array, and the row array contains at which index in data, the row starts.*

Alongside the data array, there is a *column* array that holds the column index of the block at the same index in the data array. Finally, the ith element of the *row* array contains the index in the data array where the ith row starts. The structure is illustrated on Figure 3.2.

As most matrix-matrix and vector-matrix operations can be implemented by iterating over the rows of the matrix, this data structure has good locality of memory. The overhead is also small - one integer per row and one integer per non-zero block. It has two main weaknesses. The first is that it does not support random access by column. Trying to access a given column requires a binary search through the row in question. The other weakness is that blocks must be added to the matrix in order. Changing the structure of the compressed matrix can be expensive.

The design is implemented by [Ras09] in the Blocked Matrix Library (BML) which is used in this thesis. As a part of the thesis, the BML library has been expanded with row- and column products as well as a compressed vector format.

3.1.1 Block Operations

As the matrices and vectors used by the prox formulation have a very clear block structure, where non-zero elements are organized in blocks, the expense of different block operations is important when comparing different schemes, particular when it comes to the costs inside the loop.

Addition or subtractions of two blocks has a price linear with the number of elements in the block. So the price of adding or subtracting two $m \times n$ blocks is mn additions or subtractions.

Multiplication of two blocks involves one row-column multiplication for each element in the resulting block. If the factors have size $m \times n$ and $n \times p$, the resulting block has $m \times p$ elements, and computation of each element in the resulting block involves n multiplications and $n - 1$ additions. Thus, a block multiplication costs mpn multiplications and $mp(n - 1)$ additions.

Depending on implementation and compiler optimizations, these operations may involve iterating over the indices of the block.

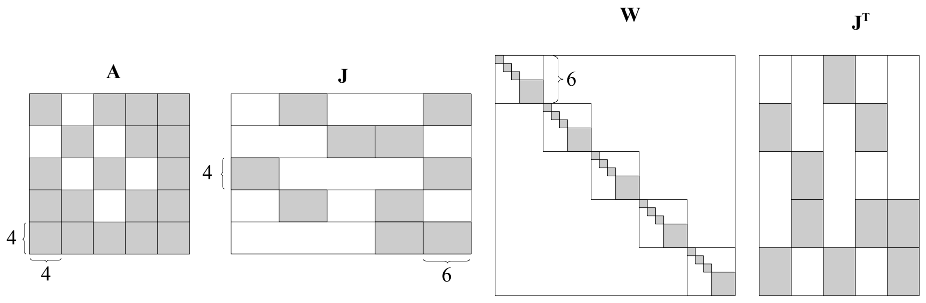


Figure 3.3: *The figure shows the structure of the matrices* $\mathbf{A}$, $\mathbf{J}$, $\mathbf{W}$ *and* $\mathbf{J}^T$. *The grey boxes represent non-zero blocks, and the white areas contain only zeros. The matrices are comparable, so if the product* $\mathbf{A} = \mathbf{J}\mathbf{W}\mathbf{J}^T$ *was computed,* $\mathbf{A}$ *would have the shown form.*

3.2 The Structure of A

In the following it will be shown how the coupling between contact points are encoded into $\mathbf{A}$. From (2.69) we know that

$$\mathbf{A} = \mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T \tag{3.3}$$

The internal blocked structure of the matrices are shown on Figure 3.3. Note that the inverse generalized mass matrix $\mathbf{M}^{-1}$ is often denoted by $\mathbf{W}$. As $\mathbf{M}$ is a diagonal matrix, we know that $\mathbf{W}$ is also a diagonal matrix. Thus, $\mathbf{W}\mathbf{J}^T$ has the same structure as $\mathbf{J}^T$. From Section 2.3.2 we know that $\mathbf{J}$ has a very peculiar form. Each row of $\mathbf{J}$ has exactly two 4×6 blocks. The column index of these blocks are the row indices in $\vec{s}$ of the two bodies that touch at that particular contact point.

As $\mathbf{J}^T$ is the transposed of $\mathbf{J}$, it has exactly two non-zero blocks in each column, whose row indices are the same as the respective body's index in $\vec{s}$. It follows that if K is the number of contact points, $\mathbf{A}$ is a $K \times K$ block matrix consisting of 4×4 blocks. The i, j block of $\mathbf{A}$ is non-zero if the product of the ith row of $\mathbf{J}$ and the jth column of $\mathbf{J}^T$ is non-zero. From this it follows directly that the diagonal block of $\mathbf{A}$ will *always* be non-zero.

The product $\mathbf{J}_{\text{row}i}\mathbf{J}^T_{\text{col}j}$ is non-zero if both $\mathbf{J}_{im}$ and $\mathbf{J}^T_{mj}$ are non-zero for at least one index m. That is, if one of the bodies in the ith contact point is in contact with the mth body. This leads to Theorem 3.1.

Theorem 3.1 *The block at* $\mathbf{A}_{ij}$ *is non-zero if and only if one of the bodies in the* i*th contact point is an active part in the* j*th contact point.*

Theorem 3.1 leads directly to Corollary 3.1.

Corollary 3.1 *Let the* i*th contact point be the contact between bodies* A *and* B*, let* n_a *be the number of contact points between* A *and any other body, and let* n_b *be the number of contact points between* B *and any other body. If there is one contact point between* A *and* B*, then the number of non-zero blocks in the* i*th row of* $\mathbf{A}$ *is equal to* $n_a + n_b - 1$.

Corollary 3.1 has an important consequence: If almost all bodies are in contact with almost all other bodies, the number of non-zero blocks in $\mathbf{A}$ is quadratic with the

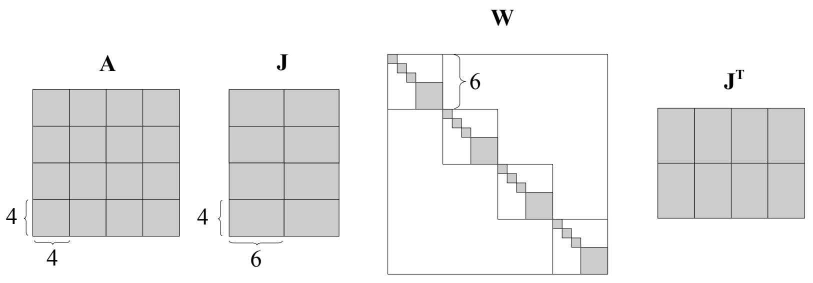


Figure 3.4: *The figure shows the structure of the matrices* $\mathbf{A}$*,* $\mathbf{J}$*,* $\mathbf{W}$ *and* $\mathbf{J}^T$ *for a box placed on the ground. The grey boxes represent non-zero blocks, and the white areas contain only zeros. The matrices are comparable, so if the product* $\mathbf{A} = \mathbf{J}\mathbf{W}\mathbf{J}^T$ *was computed,* $\mathbf{A}$ *would have the shown form.*

number of contact points. While this worst case situation becomes more and more unlikely as K increases, this is the reason that it may be an advantage to not compute $\mathbf{A}$ explicitly, but use the factorization in (3.3) to maintain and update $\vec{w} = \mathbf{W}\mathbf{J}^T\vec{\lambda}$ as suggested in [Erl04].

If the structure of $\mathbf{J}$ and $\mathbf{J}^T$ is known and accessible in some convenient format, it is possible to compute the number of non-zero blocks in $\mathbf{A}$ in linear time. For each row in $\mathbf{J}$, find the column indices of the bodies in contact, and find the length of the rows at those indices in $\mathbf{J}^T$.

The number of non-zero blocks in $\mathbf{A}$ can be computed very fast. This metric can be exploited to chose the fastest method given the configuration.

However, Corollary 3.1 assumes only one contact point between each body. This is often not the case. If the system consists of a box on the ground, there would be 4 contact points between the box and the ground. The form of $\mathbf{A}$, $\mathbf{J}$, and $\mathbf{J}^T$ is shown on Figure 3.4. If we applied Corollary 3.1 directly, the number of nonzero blocks in each row of $\mathbf{A}$ would be $4 + 4 - 1 = 7$, which, clearly, is wrong, as $\mathbf{A}$ is a 4×4 block matrix. This is because each of the contact points between the two bodies are counted twice. It follows that we need to subtract the number of contact points between A and B, from which we get:

Corollary 3.2 *Let the* i*th contact point be the contact between bodies* A *and* B*, let* n_a *be the number of contact points between* A *and any other body, and let* n_b *be the number of contact points between* B *and any other body. If there are* n_{ab} *contact points between* A *and* B*, the number of non-zero blocks in the* i*th row of* $\mathbf{A}$ *is equal to* $n_a + n_b - n_{ab}$.

Corollary 3.2 correctly computes the number of nonzero blocks in a given row of $\mathbf{A}$ in Figure 3.4 to $4 + 4 - 4 = 4$. It is, however, not as computationally cheap to apply Corollary 3.2 using $\mathbf{J}$ and $\mathbf{J}^T$, unless some structure containing the number of contact points between two bodies is available. This is a further indication that grouping contact points into contact sets could be very useful.

The preceding paragraphs have investigated the form of $\mathbf{A}$ given a Jacobian $\mathbf{J}$. What if the two non-zero blocks in each row of $\mathbf{J}$ were randomly distributed? The probability of a non-zero block in the mth column of the ith row would be $p(\mathbf{J}_{i,m}) =$

$2/M$ if M is the number of objects in the scene. Similarly, the probability of a non-zero block in the jth column and mth row in $\mathbf{WJ}^T$ would be $p(\mathbf{WJ}^T_{m,j}) = 2/M$.

The $\mathbf{A}_{ij}$ is non-zero if at least one of the non-zero blocks in the ith row of $\mathbf{J}$ and the jth column of $\mathbf{WJ}^T$ has the same m index. Let us assume that the non-zero blocks of the ith row of $\mathbf{J}$ are placed at m_1 and m_2. The probability that one of the non-zero blocks in the jth column is placed at m_1 or m_2 is $2/M$ for each. It follows that the probability of $\mathbf{A}_{ij}$ being non-zero is $2/M$.

Thus, we expect $2K^2/M$ non-zero elements in $\mathbf{A}$. If M is larger than or comparable to K, $\mathbf{A}$ is sparse with a linear number of non-zero blocks, but if K is large compared to M, $\mathbf{A}$ will be a dense matrix. Note that the assumption of a random distribution of non-zero blocks in $\mathbf{J}$ may be reasonable for nonstructured configurations, but not for structured configurations.

3.3 Jacobi Scheme

In a Jacobi scheme, the solution for the new iteration is computed using the solution from the previous solution. Thus, the full $\vec{z}^{\,i}$ is computed

$$\vec{z}^{\,i+1} = \vec{\lambda}^i - \mathbf{R}(\mathbf{A}\vec{\lambda}^i + \vec{b}) \tag{3.4}$$

And then the proximal map is applied to find $\vec{\lambda}^{i+1}$

$$\vec{\lambda}^{i+1} = \mathsf{prox}_{\mathcal{N},\mathcal{F}}(\vec{z}^{\,i+1}) \tag{3.5}$$

If the previous $\vec{\lambda}^i$ was maintained, it would be possible to compute $\vec{\lambda}^{i+1}$ block by block, without maintaining $\vec{z}$ explicitly. Pseudocode for this scheme is shown on Figure 3.5.

If $\vec{\lambda}^i$ and $\vec{\lambda}^{i+1}$ were placed in an array, it would be possible to roll back by flipping an index. As the Jacobi scheme does not write to shared data in the loop, it is embarrassingly parallel.

For the kth block of $\vec{\lambda}$, $\vec{\lambda}$ is multiplied with a row in $\mathbf{A}$. Each of these block multiplications requires 16 multiplications and 12 additions. Aside from this, we have 2 vector block additions and a multiplication between a diagonal block and a vector block. This has a price of 4 multiplications and 8 additions. As can be seen, the price of multiplying $\mathbf{A}$ and $\vec{\lambda}$ dominates the Jacobi scheme. Note that the cost of computing the proximal point is ignored, but is expected to be constant. The ideas in the following section on Gauss–Seidel schemes can also be used with the Jacobi scheme.

As will become apparent in Section 3.5 it may be necessary to update $\mathbf{R}$ by multiplying it with a scalar ν. This has a cost of 4 multiplications per contact point. Assuming that the product $\mathbf{WJ}^T$ is precomputed, the cost of computing $\mathbf{A}$ is 96 multiplications and 80 additions per non-zero block in $\mathbf{A}$.

3.4 Gauss–Seidel Schemes

The key insight of a Gauss–Seidel method is that the currently computed results for iteration $i+1$ might as well be used directly instead of waiting for the next iteration. If one assumes that the blocks are handled sequentially, the $k-1$ blocks have been updated when updating the kth block. Conceptually, the Gauss–Seidel method can be expressed by

$$\vec{z}_k^{\,i+1} = \vec{\lambda}_k^i - \mathbf{R}_k(\mathbf{L}\vec{\lambda}^{i+1} + (\mathbf{D}+\mathbf{U})\vec{\lambda}^i + \vec{b})_k \tag{3.6}$$

```
NAIVESCHEME($\mathbf{J}, \mathbf{W}, \mathbf{R}, \vec{b}$)
 1  $\mathbf{A} \leftarrow \mathbf{J}\mathbf{W}\mathbf{J}^T$
 2  for $i \leftarrow 0$ to max iterations
 3       do
 4          for $k \leftarrow 0$ to number of contact points
 5              do
 6                  $\vec{z}_k \leftarrow \vec{\lambda}_k^i - \mathbf{R}_k(\mathbf{A}_{\text{row } k}\vec{\lambda}^i + \vec{b}_k)$
 7                  $\vec{\lambda}_k^{i+1} \leftarrow \text{prox}_{\mathcal{N},\mathcal{F}}(\vec{z}_k)$
 8          $\varepsilon^i \leftarrow$compute_residual($\vec{\lambda}^i, \vec{\lambda}^{i+1}$)
 9          if $\varepsilon^i > \varepsilon^{i-1}$
10            then
11                  ▷ handle divergence
12            else
13                  ▷ continue
14  return $\vec{\lambda}$
```

Figure 3.5: *The figure shows pseudocode for a simple sequential Jacobi proximal map scheme. A Jacobi scheme would use one $\vec{\lambda}$ variable in the loop. One could also add notions of absolute convergence as a stop criterion.*

where $\mathbf{L}$ is the strictly lower, $\mathbf{D}$ is the diagonal and $\mathbf{U}$ is the strictly upper matrix of $\mathbf{A}$. However, neither $\mathbf{L}$, $\mathbf{D}$ nor $\mathbf{U}$ need to be computed explicitly. In practice, $\vec{\lambda}$ is updated directly. This can be made explicit in (3.6) by defining the partially updated $\vec{\lambda}^{i'}$ while working on iteration $i+1$ by

$$\vec{\lambda}^{i'} = \begin{cases} \vec{\lambda}_n^{i+1} & \text{if } n < k \\ \vec{\lambda}_n^{i} & \text{if } n \geq k \end{cases}, \tag{3.7}$$

where n is the block corresponding to the nth contact point and k is the contact point currently under consideration. This allows us to rephrase (3.6) as

$$\vec{z}_k^{\,i+1} = \vec{\lambda}_k^{i'} - \mathbf{R}_k(\mathbf{A}_{\text{row}k}\vec{\lambda}_k^{i'} + \vec{b}_k) \tag{3.8}$$

The cost of updating $\vec{\lambda}_k$ using (3.8) is equal to the price of (3.4). The following sections will investigate different ways of improving the performance of (3.8).

3.4.1 T Scheme

One common way of optimizing a method is to move computations out of the loop. In this case, equation (3.8) is the central part of the loop, performed in each iteration, for each contact point. Let us rearrange the terms

$$\begin{aligned} \vec{z}_k^{\,i+1} &= \vec{\lambda}_k^{i'} - \mathbf{R}_k(\mathbf{A}_{\text{row}k}\vec{\lambda}_k^{i'} + \vec{b}_k) && (3.9)\\ &= \vec{\lambda}_k^{i'} - \mathbf{R}_k\mathbf{A}_{\text{row}k}\vec{\lambda}_k^{i'} + \mathbf{R}_k\vec{b}_k && (3.10)\\ &= (\mathbf{1} - \mathbf{R}_k\mathbf{A}_{\text{row}k})\vec{\lambda}_k^{i'} + \mathbf{R}_k\vec{b}_k && (3.11)\\ &= \mathbf{T}_{\text{row}k}\vec{\lambda}_k^{i'} + \vec{c}_k && (3.12) \end{aligned}$$

where $\mathbf{1}$ is the identity matrix. As $\mathbf{R}$ is a diagonal matrix, the distribution of non-zero blocks in $\mathbf{T}$ is the same as the distribution of non-zero blocks in $\mathbf{A}$ almost everywhere, though $\mathbf{T}$ *may* have zeroes in the diagonal. Thus, the price of computing $\mathbf{T}_{\text{row}k}\vec{\lambda}_k^{i'}$ is the same as computing $\mathbf{A}_{\text{row}k}\vec{\lambda}_k^{i'}$, that is, 16 multiplications and 12 additions per non-zero block in the kth row of $\mathbf{T}$. There is, however, only one vector addition, which costs 4 additions. Thus, this scheme saves 4 additions and multiplications in the innermost loop. The price is that computing $\mathbf{T}$ is more expensive. As $\mathbf{R}$ is a diagonal matrix, the product $\mathbf{R}\ \mathbf{A}$ costs 16 multiplications per non-zero block and 4 additions to subtract the results from $\mathbf{1}$. These operations are performed once, when $\mathbf{T}$ is computed.

To take advantage of (3.12), $\mathbf{T}$ and $\vec{c}$ must be precomputed. This can be a problem if $\mathbf{R}$ is changed. Section 3.5 will explain that in some cases, $\mathbf{R}$ must be multiplied by the scalar ν so that

$$\mathbf{R}_1 = \nu\mathbf{R}_0 \tag{3.13}$$

The new $\vec{c}_1$ is thus

$$\vec{c}_1 = \nu\mathbf{R}_0\vec{b} = \nu\vec{c}_0 \tag{3.14}$$

Updating $\vec{c}$ costs 4 multiplications per contact point. The new $\mathbf{T}_1$ is

$$\mathbf{T}_1 = \mathbf{1} - \nu\mathbf{R}_0\mathbf{A} \tag{3.15}$$

Noting that $\mathbf{T}_0 = \mathbf{1} - \mathbf{R}_0\mathbf{A}$ it follows that $\mathbf{R}_0\mathbf{A} = \mathbf{1} - \mathbf{T}_0$, so

$$\mathbf{T}_1 = \mathbf{1} - \nu(\mathbf{1} - \mathbf{T}_0) \tag{3.16}$$

At face value, the price of computing $\mathbf{T}_1$ from $\mathbf{T}_0$ is 16 multiplications and 32 subtractions per block. However, $\mathbf{1}$ is zero outside the diagonal, so it is only in the diagonal that we actually need to perform any subtraction. Thus, for the diagonal blocks 16 multiplications and 8 subtractions are needed. For non-diagonal blocks, 16 multiplications are needed. This leads to the expression

$$\mathbf{T}_1^{ij} = \begin{cases} \nu\mathbf{T}_0^{ij} & \text{if } i \neq j \\ \mathbf{1} - \nu(\mathbf{1} - \mathbf{T}_0^{ij}) & \text{if } i = j \end{cases}, \tag{3.17}$$

The expensive part of the $\mathbf{T}$ scheme is the computation of $\mathbf{T}\vec{\lambda}$. When the kth contact point is done, the product can be updated using the change in $\vec{\lambda}_k$ and multiply it with the kth column in $\mathbf{T}$. However, when the kth contact point has been considered, we only really need to update the rows in $\mathbf{T}\vec{\lambda}$ after the kth row to continue in that iteration. After such an iteration, $\mathbf{T}\vec{\lambda}$ would need to be recomputed. This is exactly what happens in a regular $\mathbf{T}$ scheme iteration. Thus, one can alternate between normal iterations updating $\mathbf{T}\vec{\lambda}$ and cheap iterations, only updating the lower part of $\mathbf{T}\vec{\lambda}$. Such a scheme would reduce the overall number of block multiplications by approximately one quarter. Due to time constraints this optimization has not been implemented.

3.4.2 Factorized Scheme

The price of computing $\mathbf{A}$ or $\mathbf{T}$, as well as the product $\mathbf{T}\vec{\lambda}$ are high, particularly when one takes into consideration that $\mathbf{A}$ and $\mathbf{T}$ *could* be dense $K \times K$ matrices. One way around this problem is to use the factorization of $\mathbf{A}$ in equation (3.3). We know that $\mathbf{J}$ has exactly two 4×6 non-zero blocks in each row. Similarly, $\mathbf{W}\mathbf{J}^T$ has exactly two

```
FACTORIZEDSCHEME($\mathbf{J}, \mathbf{W}, \mathbf{R}, \vec{b}$)
 1  $\mathbf{WJT} \leftarrow \mathbf{WJ}^T$
 2  for $i \leftarrow 0$ to max iterations
 3      do
 4         for $k \leftarrow 0$ to number of contact points
 5             do
 6                $\vec{z}_k \leftarrow \vec{\lambda}_k - \mathbf{R}_k(\mathbf{J}_{\text{row } k}\vec{w} + \vec{b}_k)$
 7                $\Delta\vec{\lambda}_k \leftarrow \vec{\lambda}_k$          ▷ save old value
 8                $\vec{\lambda}_k \leftarrow \text{prox}_{\mathcal{N},\mathcal{F}}(\vec{z}_k)$
 9                $\Delta\vec{\lambda}_k \leftarrow \vec{\lambda}_k - \Delta\vec{\lambda}_k$
10                $\vec{w} \leftarrow \vec{w} + \mathbf{WJT}_{\text{col } k}\Delta\vec{\lambda}_k$
11         $\varepsilon^i \leftarrow$compute_residual($\vec{\lambda}^i, \vec{\lambda}^{i+1}$)
12         if $\varepsilon^i > \varepsilon^{i-1}$
13           then
14                ▷ handle divergence
15           else
16                ▷ continue
17  return $\vec{\lambda}$
```

Figure 3.6: *The figure shows pseudocode for a sequential factorized Gauss–Seidel scheme. Memory usage can be improved by computing the residual inside the loop.*

non-zero 6×4 blocks in each column. If $\vec{w}$ is the contact velocity update to the bodies in the scene due to the contact impulses $\vec{\lambda}$, we have

$$\vec{w} = \mathbf{WJ}^T\vec{\lambda} \tag{3.18}$$

If $\vec{\lambda}_k$ is updated, this affects exactly two bodies, which correspond to two blocks in $\vec{w}$. If $\mathbf{WJ}^T$ is precomputed, $\vec{w}$ can be updated by

$$\vec{w} = \vec{w} + (\mathbf{WJ}^T)_{\text{col } k}(\vec{\lambda}_k^{i'} - \vec{\lambda}_k^{i'-1}) \tag{3.19}$$

The pseudocode of this scheme is shown on Figure 3.6.

The price of updating $\vec{w}$ is 52 additions and 48 multiplications as there are exactly two non-zero blocks in each column of $\mathbf{WJ}^T$. The multiplication $\mathbf{J}_{\text{row}k}\vec{w}$ also requires exactly two vector-matrix multiplications and one block addition. Thus, 40 additions and 48 multiplications are performed. Additionally, 8 additions and 4 multiplications need to be performed to compute $\vec{z}_k$. So, the price to compute $\vec{z}_k$ and update $\vec{w}$ is 100 additions and 100 multiplications.

On the flip side, we will need to maintain the $\vec{w}$ vector, which uses one 6×1 block per body in the scene. This complicates matters, if we would need to roll back the solution. We would need to either maintain the previous $\vec{w}$, which requires some copying, or recompute $\vec{w}$ using the previous value of $\vec{\lambda}$.

3.4.3 Comparisons of the schemes

Table 3.1 compares the time complexity of the schemes. It is noted that the initialization costs for the Naive and $\mathbf{T}$ schemes are high, comparable to performing ~ 7 iterations

Scheme	**Loop cost per iteration**	**Initialization cost**	**R update cost**
Naive scheme	$16N + 8K$ multiplications $12N + 4K$ additions	$96N$ multiplications $80N$ additions	$4K$ multiplications
$\mathbf{T}$ scheme	$16N$ multiplications $12N + 4K$ additions	$112N$ multiplications $84N$ additions	$4K + 16N$ multiplications $8K$ additions
Factorization scheme	$100K$ multiplications $100K$ additions	-	$4K$ multiplications

Table 3.1: *The table shows an overview of the time complexity of the different schemes. K is the number of contact points and N is the number of non-zero blocks in $\mathbf{A}$. Note that the Naive and $\mathbf{T}$ Scheme is linear with the number of non-zero blocks in $\mathbf{A}$ and the number of contact points, whereas the factorized scheme is linear with the number of contact points. Initialization cost assumes that $\mathbf{WJ}^T$ and $\vec{b}$ are precomputed, as all of the three schemes will need to compute them.*

with these schemes. In the loop itself, the number of non-zero blocks N in $\mathbf{A}$ or $\mathbf{T}$ is seen to have a large impact. The factorized scheme seems to have the advantage if $N > 6K$.

One disadvantage of the factorized scheme is that the memory usage in the loop is linear with the number of bodies in the scene M. This can be alleviated by using a temporary data structure that only contain the number of bodies that touch another body which is at most $2K$. Worse, lookups in $\vec{w}$ and $\mathbf{WJ}^T$ are likely to have bad cache coherency. This could be improved if $\mathbf{WJ}^T$ was reorganized as a compressed column matrix.

So far, we have ignored the cost of computing $\vec{b}$ and $\mathbf{WJ}^T$. Let us take a look at that cost. First, we would need to invert the mass matrix. As $\mathbf{M}$ is blocked diagonal matrix, where each block consists of one 3×3 diagonal matrix, and one possibly dense 3×3 matrix, which we will need to invert. This is dominated by computing the determinant of the 3×3 matrix, yielding a total cost of approximately $44M$ operations. However, this matrix is likely to be reusable across iterations, so the cost can be amortized.

Computation of $\mathbf{J}$ involves 3 vector cross products per block, yielding a total cost of approximately $53K$ operations. Transposing it is simple, pretty much just a copy, which costs some $48K$ operations. The product $\mathbf{WJ}^T$ is, however, expensive. All of the $2K$ blocks in $\mathbf{J}^T$ must be multiplied with the corresponding 6×6 blocks in $\mathbf{W}$, with each operation costing 264 operations, or $528K$ operations in all.

Computing $\vec{b}$ involves computing the product $\mathbf{J}\vec{u}$, which involves $2K$ block multiplications between a 4×6 block and a 6×1 block, corresponding to $88K$ operations. The product $\mathbf{W}d\vec{th}$ involves $6K$ multiplications and 66 operations per block multiplications, yielding $72K$ operations. The subsequent adding operation costs some $24K$ operations.

The total initialization costs are on the order of $44M + 813K$ operations. If $\mathbf{A}$ and $\mathbf{T}$ are very sparse, on the order of $N = 5K$, the price of computing $\mathbf{A}$ and $\mathbf{T}$ would be comparable to the computation of $\vec{b}$ and $\mathbf{WJ}^T$. However, if $\mathbf{A}$ and $\mathbf{T}$ are somewhat more dense, just on the order of $N = 10K$, the construction of $\mathbf{A}$ and $\mathbf{T}$ would significantly increase the initialization time.

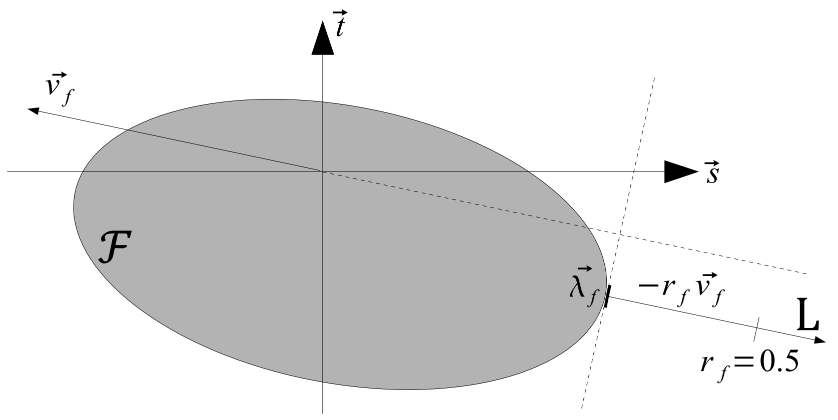


Figure 3.7: *The figure illustrates that for all $r_f > 0$, the point in $\mathcal{F}$ closest to any point on the halfline $\mathbf{L} = \{\vec{\lambda}_f - r_f\vec{v}_f \mid r_f > 0\}$ will be $\vec{\lambda}_f$ in two dimensions.*

3.5 Convergence, Divergence and R-factor Strategy

In equation (2.74) a condition on the correct contact impulse $\vec{\lambda}$ was shown. There is only one bound on the r-factors – they need to be greater than zero. For the correct solution $\vec{\lambda}$, all $r > 0$ can be said to be equivalent as illustrated on Figure 3.7. However, when using an iterative fixed point scheme, the solution is unknown. In each iteration, the currently known solution is used to approximate the contact velocity and the size of the set of legal friction components $\mathcal{F}$. Different values of r may have an impact on how fast the solution converges, or even if it converges at all. This has been investigated by [FGNU06].

Consider a sphere of mass m resting on the ground. Gravity causes a downward impulse $-g$, which results in a base contact velocity of $v_n^0 = -g/m$. To ensure against penetrations, the normal impulse must be $\lambda_n = g$. The first iteration yield

$$\lambda_n^1 = \mathsf{prox}_{\mathcal{N}}(\lambda_n^0 - rv_n^0) = \mathsf{prox}_{\mathcal{N}}(0 - r(-g/m)) = \mathsf{prox}_{\mathcal{N}}(rg/m) = \rho g \tag{3.20}$$

where $\rho = r/m$. Note that for $r = m$, the correct λ_n is found in one iteration. The updated contact velocity is

$$v_n^1 = v_n^0 + \frac{\lambda_n^1}{m} = -\frac{g}{m} + \frac{\rho g}{m} = \frac{(\rho - 1)g}{m} \tag{3.21}$$

The next λ_n is

$$\lambda_n^2 = \mathsf{prox}_{\mathcal{N}}(\lambda_n^1 - rv_n^1) = \mathsf{prox}_{\mathcal{N}}(\rho g - r\frac{(\rho - 1)g}{m})) = \rho(2 - \rho)g \tag{3.22}$$

If $\rho = 1$, the product $\rho(2 - \rho) = 1$, and λ_n have converged to the correct solution.

The derivative of $h(\rho) = \rho(2-\rho)$ w.r.t. ρ is $h'(\rho) = 2-2\rho$. So, if $0 < \rho < 2$, $h(\rho)$ converges towards 1, and if $\rho \geq 2$, $h(\rho)$ does not converge. Thus, the value of r can have significant effect on the speed with which $\vec{\lambda}$ converges, or even *if* it converges. In [FGNU06] two strategies to find a good r-factor are discussed.

The *global strategy* is based on minimizing the spectral radius of $\mathbf{T}$. This is similar to the results derived in [BF97] where it is shown that when using the Jacobi or the Gauss–Seidel iterative matrix solvers on a linear system of equations of the form $\vec{\lambda}^{i+1} = \mathbf{T}\vec{\lambda}^{i} + \vec{c}$ converges if the spectral radius of $\mathbf{T}$ is smaller than 1, and that a smaller spectral radius implies faster convergence. The spectral radius of $\mathbf{T}$ is the largest eigenvalue of $\mathbf{T}$. The eigenvalues of $\mathbf{T}$ can be computed from the eigenvalues of $\mathbf{A}$

$$\tau_i = 1 - r\alpha_i \tag{3.23}$$

As the largest eigenvalue of $\mathbf{T}$ depend on the largest and the smallest eigenvalue of $\mathbf{A}$, the optimal global r-factor r_{opt} is

$$r_{opt} = \frac{2}{\alpha_{min} + \alpha_{max}} \tag{3.24}$$

The *local strategy* is motivated by noting that for a diagonal $\mathbf{A}$, choosing

$$r_i = \frac{1}{A_{ii}} \tag{3.25}$$

results in $\mathbf{T}$ becoming zero, and $\vec{\lambda} = \vec{c}$. This special case yields convergence in one iteration. This strategy is also likely to yield good results if $\mathbf{A}$ is diagonally dominant.

[FGNU06] notes that while the local strategy is cheaper to compute and overall performs better than the global strategy, neither strategy ensures convergence. However, if divergence is detected, they recommend rolling back and multiplying r_i by $\nu_i < 1$, as a smaller r-factor may converge. Even though small r-factors are more likely to converge, larger r-factors converge faster *if* they converge.

Unfortunately, [FGNU06] does not consider how a good ν-factor strategy can be formulated. However, if divergence is detected at iteration $i + 1$, it is reasonable to assume that the locally optimal solution is placed between $\vec{\lambda}^i$ and $\vec{\lambda}^i - r_k\vec{v}^i$. As the mid point is likely to be the closest to the locally optimal solution, one would assume that choosing $\nu_i = 0.5$ would yield good results. If a good ν-factor strategy were available, it could be interesting to construct a diagonal matrix of ν-factors or even compute it on the fly.

A related omission is how many times one should recompute $\mathbf{R}$ before giving up. It makes no sense to continue indefinitely, but stopping too soon might mean we miss the r-factor that causes convergence. Trying 5 times seems reasonable, as r at that point will be reduced to 0.03125 of the original value.

3.6 Estimating Error

An important part of the r-factor strategy is to have a fast and reliable way to discover if the solution is converging or diverging. We would also like to be able to estimate the quality of a given result, that is, how well does it satisfy the equations of motion.

A measure of the residual $\vec{\varepsilon}_k^{\,i}$ of the kth block in the ith iteration can be found by noting that for (2.74) we have

$$\vec{\lambda}_k = \mathsf{prox}_{\mathcal{N},\mathcal{F}}(\vec{\lambda}_k - \mathbf{R}_k(\mathbf{A}\vec{\lambda} + b)_k) \Leftrightarrow \tag{3.26}$$

$$0 = \mathsf{prox}_{\mathcal{N},\mathcal{F}}(\vec{\lambda}_k - \mathbf{R}_k(\mathbf{A}\vec{\lambda} + b)_k) - \vec{\lambda}_k \tag{3.27}$$

Thus, the residual can be defined by

$$\vec{\varepsilon}_k^{\,i} = \mathsf{prox}_{\mathcal{N},\mathcal{F}}(\vec{\lambda}_k^{i-1} - \mathbf{R}_k(\mathbf{A}\vec{\lambda}^{i-1} + b)_k) - \vec{\lambda}_k^{i-1} \tag{3.28}$$

Equation (3.28) is equivalent with the difference between the current and the previous solution, from which it follows that

$$\vec{\varepsilon}^{\,i} = \vec{\lambda}^{i} - \vec{\lambda}^{i-1} \tag{3.29}$$

This suggests that a merit function Θ could be defined by

$$\Theta = \vec{\varepsilon}^{\,i} \cdot \vec{\varepsilon}^{\,i} \tag{3.30}$$

This estimate requires squaring and summing each unknown, which is expensive, and may have a number of numeric problems. A common alternative is to use the infinity norm, where one estimates the residual by the largest change in an unknown, that is

$$\varepsilon_{\infty}^{i} = \max_j(\ \mathrm{abs}(\vec{\lambda}_j^{i} - \vec{\lambda}_j^{i-1})\ ) \tag{3.31}$$

where j is an element index (not a block index). If $\varepsilon_{\infty}^{i} < \varepsilon_{\infty}^{i-1}$, $\vec{\lambda}$ is converging.

In case of convergent behaviour, (3.31) can be viewed as a measure of how much the solution has improved over the last iteration. Thus, in each iteration, we wish to maximize ε.

When (3.31) is used in an r-factor scheme, we need to consider the possibility that the improvement per iteration could increase. Indeed, test runs have suggested that this actually happens in some cases, particularly in the first few iterations. Ignoring early divergent behaviour may wrongly ignore real divergent behaviour, so that is not an option. We just have to *hope* that it is not the dominant residual that accelerates. If this problem could be solved, $\mathbf{R}$ could be updated locally, improving overall convergence behaviour.

Another possible use of the residual is to compare the quality of two proposed solutions. However, the residual in (3.31) depends on the value of the r-factors. If $r = 10^{-100}$, the residual is likely to be very small. This alone makes (3.31) unsuitable as a measure of the quality of a proposed solution.

An alternative is to use a constant $\mathbf{R}$ to estimate the quality of a solution. If these constant r-factors are too small, the speed of convergence will be dominant, and if they are to large, they may cause numerical instability or mask small differences. However, if a constant $\mathbf{R}$ was used by the solver to estimate residual, it would require performing an extra iteration to measure the residual, effectively doubling the cost of each iteration.

This makes (3.31) with a constant $\mathbf{R}$ unsuitable for use by the solver, but could be a reasonable measure to compare two proposed solutions to the same problem. Let us define the quality measure γ of the solution in the ith iteration by

$$\gamma^{i} = \frac{1}{2}||(\mathsf{prox}_{\mathcal{N},\mathcal{F}}(\vec{\lambda}^{i} - \mathbf{R}^{const}(\mathbf{A}\vec{\lambda}^{i} + b)) - \vec{\lambda}^{i}||^2 \tag{3.32}$$

where $\mathbf{R}^{const}$ is a matrix of large and constant r-factors. Some tests suggest that the constant r-factors need to be significantly larger than the value of the r-factors used by the solver. A factor of at least 10 is advisable. Thus, setting the constant r-factors to 100 or similar seems a reasonable value.

3.7 Parallelization of the Schemes

The preceding sections have discussed the contact problem and formulated a number of schemes to solve it. These schemes have been based on Jacobi and Gauss–Seidel methods. This section will start by investigating how similar schemes have been parallelized. After investigating previous work, the section will investigate general issues with parallelizing Gauss–Seidel schemes. In the following, p will be the number of CPUs available.

3.7.1 Previous Work

A number of articles have investigated how to parallelize the Gauss–Seidel iterative matrix solver, that is, solve $\mathbf{A}\vec{x} + \vec{b} = 0$ for $\vec{x}$.

Naive Parallelization

[LM99] split the data into $p - 1$ sets, where p is the number of available processors. The first processor is used as a master, the others handle one of the subsets. It is similar to Processor Block Gauss–Seidel mentioned in [ABHT03], but the authors claim to obtain (super) linear scalability with the number of processors to a certain point, where the improvements in time usage fall off. Results by [ABHT03] note that this naive splitting has problems of non-convergence, mirroring the performance of the Jacobi method. These differences may be due to their test data yields $\mathbf{A}$ matrices with different structures.

A refinement of this scheme is proposed in [TD07]. $\mathbf{A}$, $\vec{x}$ and $\vec{b}$ are split into p parts and placed on their own CPU. Each part is solved from the bottom or the top so that neighbours are solved from different directions. This exploits the Gauss–Seidel property that the computed values from this iteration has an impact on multiplications above or below the diagonal, respectively. After each iteration, the direction is changed and some fixing is made at the boundaries. Test data suggest a reasonable performance, but scalability is not investigated.

[RDA04] investigate the problem of constructing a parallel contact dynamics algorithm to solve for granular media, that is a media made up of many grains, such as sand or rice, and conform to the shape of a container, but has some stability. They use a parallel version of a nonlinear Gauss–Seidel, but use precious little time to describe their method, noting that *splitting the contact loop between P threads which may be related to different processors* may generate race conditions, which suggests a naive approach.

Their experiments suggest that their method scales linearly up to 5-6 processors where a falloff can be seen. It is unlikely that their method scales linearly beyond this point, which implies that their method is not useful in the context of a massively parallel version of Gauss–Seidel.

Graph Partitioning

The naive parallelization can be refined further by considering another approach to splitting the problem into smaller subproblems. All bodies in the scene can be partitioned into sets depending on their spatial location, and each set controlled by one process.

This idea has been investigated by [IR09a], where each process handled the entire physics simulator for the set in question, including construction of the LCP conditions, solving the LCP and advancing the animation. The authors benchmarked their implementation and discovered that their method scales so that doubling the number of cores reduces time usage by more than a factor 2, likely due to better cache coherency. The communication between the processes can lead to significant slowdowns, as they discovered when changing the shape of the sets: Fewer neighbours resulted in less slowdown. They also worked on a physics engine, *pe*, introduced in a tech report [IR09b].

If one were to relax the usage of rigid bodies a little, results by [LSH10] might be interesting. The key insight is that when using *deformable* objects, applying a contact force to the centre of a bar will not have an immediate influence on objects placed on the far ends of the object. Thus, spatially distant contact points can be considered to be independent. Separating the problem into smaller parts yields significant speedups.

Independent Batches

[ABHT03] note that a parallel version of Gauss–Seidel can be obtained by identifying groups of unknowns that are independent of one another. The unknowns of an independent group can be updated concurrently. Their Parallel Block Multi-color Gauss–Seidel is based on this insight. $\mathbf{A}$ is partitioned into a number of blocks. The border nodes are processed first, and inner nodes are processed while results from border nodes are communicated to the other CPUs. However, they note that such a method can be very difficult to implement, and that there is a significant reduction in number of floating point operations per second, as the number of processors increase.

An adaption of this scheme is discussed in [Tsu09] where the contact graph is partitioned into areas (called cells) that are considered in parallel. First, the inner part of each cell is considered. In the next batch, they consider contact points on the border between 2 cells (membranes), and in the 3rd batch, they consider contact points on the border between 3 cells (strings) etc. In practice they note that only 3-4 synchronizations are needed, compared to 8-11 for their test setups.

The idea of identifying independent unknowns and using this to define a splitting is investigated by [MOIS05] by using graph coloring. Note that they use it as a pre- and post-processor for the Conjugate Gradient method. However, the coloring is constructed by iterating sequentially over the strictly lower part of the $\mathbf{A}$ matrix, ensuring that the color of the jth unknown is different from the colors of dependent unknowns by setting the color of the ith unknown to $c_i = \mathrm{mod}(c, n_{color})$ where n_{color} is the total number of colors and c is incremented if an unknown with that color is found in the ith row. This ensures a maximum number of used colors and a reasonable but not total independence between unknowns in the same color. They show that their method scales well up to 16 processors (they do not show data for more processors). No data is presented on the quality of the coloring, nor if they had any problems with race conditions.

Graph-coloring is also utilized for parallelization in [CCH$^+$07]. The contact graph is colored, and batches are constructed so that the contact points in each batch are independent. With this reordering they get a $\times 35$ speedup with 64 CPUs. Unfortunately, they do not note if the speedup includes the duration of the preprocessing, how different graph coloring strategies may affect the performance, nor which type of problem they used for benchmarking. Given their comment on broadphase computations, however, one would expect they used benchmarks where few of the objects are in contact at any given time.

Cache Efficiency

In [DHK$^+$00] and [SCF01] it is shown that cache efficiency has a significant impact on how fast a Gauss–Seidel iteration can be performed for dense and sparse matrices, respectively. As processor speeds have grown much faster than main memory speeds in the previous decade, the importance of cache efficiency is likely to grow [GBP$^+$07]. [SCF$^+$05] try to apply these techniques in a parallel shared memory system. Given the system, a dependency graph is built – for the contact problem this would be a contact graph where each vertex is a body and each contact point is an edge. Computations on a contact point can proceed to the next iteration when the dependent contact points have finished. Tiles are sets of neighbours in the dependency graph, and the interior of such a tile can be computed concurrently with independent tiles. When a tile has finished an iteration, the border can be removed from consideration and computation on the next iteration for the remaining points can proceed. The tiles are ordered in a dependency graph, and when the dependencies of a tile has finished computation, the tile can start. The advantage of this method is that it supports parallelism, intra iteration locality (results from previous iteration steps are available locally) and inter iteration locality (results from previous iterations are available locally).

A nice framework for considering and comparing parallel methods is used, and they have the nice feature of showing the ”raw” speedup (ignoring preprocessing). Unfortunately, there is an error in the article, where the table showing preprocessing times was not there, but they do show that $\sim 80\%$ of execution time is used on graph partitioning. Overall, their sparse tiling method performs well, but depends heavily on how well the graph is partitioned and how much parallelism can be pressed into it. Their results seem to suggest that the height of the tile should not increase beyond 3 or 4.

It should be noted that the contact graph can be ill suited for this type of cache based optimization, but it should be possible to devise a scheme to detect if this method can be used.

Utilization of the GPU

The problem of parallelizing the contact problem on the GPU is investigated by [TNA08] and [TNA$^+$10] using a Cone Complementarity Problem (CCP) formulation. GPUs have a very high potential throughput in terms of floating point operations per second, particularly due to a large number of concurrent floating point processors. One of the challenges when working on the GPU is that there is no mutex, which gives problems with concurrent updates of shared data. This problem is solved by ignoring it, arguing that when the number of contact points is very high, it is very unlikely that there are two contact points in the pipe that update the same variable.

Another important problem is that communication between GPUs must use the CPU, slowing the solver down, so parallelization on the GPU can only utilize one GPU at the time. Their results show that a GPU with $\sim \times 7$ Gflops more than a CPU, performs $\sim \times 7$ faster on problems with 1000, 2000 and 8000 bodies. This is not surprising as all the problems of shared data is ignored.

Another approach to using the GPU to perform Gauss–Seidel iterations on dense matrices is presented by [CA09]. A number of parallel schemes are discussed (very high level). Their primary method is a blocked approach, where they maintain the invariant that in each block row, the diagonal element is the last to be updated. This is used to schedule the block computations, and a counter can be used to detect if $\vec{x}_i$ is up to date. This eliminates the need for global synchronization. Furthermore, the computations of the blocks to the left of the diagonal can be carried out at any time, as they are independent of the updated blocks of $\vec{x}$.

Bullet

Bullet Physics is an open source physics engine, developed and maintained by Erwin Coumans [Cou10a]. A recent article in Game Developer Magazine rated Bullet as the 3rd most popular physics engine, and the most popular open source physics engine [DeL09]. This makes it interesting to look at how it handles rigid body dynamics and parallelism. In a post on the Bullet Physics Forums, Coumans explain that Bullet uses Projected Gauss–Seidel as described in [Erl04], with improvements from the Sequential Impulse method due to Erin Catto [For06]. Code review shows that this is still the case. Similar to ideas by [KSJP08], normal constraints are solved before friction constraints, though joint constraints are solved first.

Parallelism has recently been introduced into Bullet. The parallelism centers on the Cell SPU architecture, which is not surprising given that Coumans works for Sony, who produce PS3 that use the Cell SPU architecture. However, thread support for Windows and pthreads is also included. It seems, however, that only the contact detection methods have been parallelized [Cou10b], and code review of the most recent 2.76 release suggests that while parts of the parallel constraint solver are in the makings, a functional parallel solver has not yet been implemented.

3.7.2 Shared Data

When computing $\vec{\lambda}_k^{i'}$, both the naive $\mathbf{A}$ scheme and the $\mathbf{T}$ scheme will need to use dependent values of $\vec{\lambda}_k^{i'}$. When threads are running concurrently, this may cause one thread to try to write to one variable while another thread is trying to read from it. This race condition can cause the second thread to get corrupt data. For the Jacobi method this is not a problem as it reads from and writes to different vectors.

For the $\mathbf{T}$ scheme this race condition would never become a problem if it could be enforced that only independent contact points are in the pipe at any time. This insight is a central argument for using a coloring scheme.

If the contact points are distributed among processors by some coloring scheme, these race conditions only become a problem for the border nodes, and it may be possible to handle them by communication between neighbouring processors.

If one decide to use the factorized scheme of Section 3.4.2, maintaining $\vec{w}$ presents a challenge, both for a Jacobi and for a Gauss–Seidel method, as it is shared among different contact points.

For the Jacobi method, one solution could be to maintain a $\vec{w}^i$ and $\vec{w}^{i+1}$, saved in a $\vec{w}$-array, similar to $\vec{\lambda}^i$ and $\vec{\lambda}^{i+1}$, or simply recompute $\vec{w}$ in case of convergence.

The situation is a bit more complex for the Gauss–Seidel method. To exploit the Gauss–Seidel property, we need to ensure that a given block in $\vec{w}$ is updated as fast as possible – or at least before the updated block is to be used.

Each worker could compute a local update to $\vec{w}$ that could be applied to their local copy of $\vec{w}$ during the iteration, and by a controller thread to the global $\vec{w}$ between iterations or at another convenient time. If a graph partitioning is used, inner nodes can be updated directly.

If a coloring scheme is used, updates to $\vec{w}$ could be applied directly, as no other contact points under consideration could use that contact point.

3.7.3 Cost of one Contact Point

It follows from Theorem 3.1 that the number of non-zero blocks in a row of $\mathbf{A}$ or $\mathbf{T}$ varies. With the performance analysis in Section 3.4.1, this implies that the number of computations needed to solve one contact point varies as well. Thus, if an efficient parallel scheme is to be constructed, one may need to include weights of the contact point into the scheme. Fortunately, the cost of solving a contact point in one iteration is linear with the number of non-zero blocks in the corresponding row in $\mathbf{A}$ or $\mathbf{T}$.

If the number of non-zero blocks is N, and p processors are available, a greedy strategy could be to add contact points to a processor until the corresponding number of non-zero blocks reach N/p. Such a strategy is simple and fast to implement, but may yield some imbalance as one contact point could have a lot of non-zero blocks in $\mathbf{A}$.

As shown by the performance analysis in Section 3.4.2, the factorized scheme does not have this problem. All contact points have approximately the same cost, so a distribution is simple to compute and will in the worst case yield an imbalance of one contact point.

3.7.4 Detecting Divergence

As discussed in Section 3.5 and 3.6, a good r-factor strategy requires the ability to detect and handle divergence. Handling divergence is embarrassingly parallel, as $\mathbf{R}$ (or $\mathbf{T}$ and $\vec{c}$) can be updated in parallel. Computing the residual ε as discussed in Section 3.6 is embarrassingly parallel, but it has a number of side effects on the design.

To be able to detect and handle divergence, an iteration needs to be a well defined entity in the parallel environment with a clear start and end. Each worker can only perform one iteration and must send results of this iteration to a controller process and *wait* for a signal from the controller to proceed or roll back to the previous state. If the problem with accelerating residuals mentioned in Section 3.6 could be solved, the workers could handle local divergence or convergence directly. This would likely improve parallel performance.

In the average case, the worker would be allowed to proceed, so it could be allowed to start the next iteration, anticipating an all-clear message from the controller. However, such a design would require the worker to be able to roll back not to the previous state, but the one before. Another problem is that a given contact point could be updated while its neighbours are two iterations behind. This could cause issues of divergence or slow convergence. If a graph partitioning scheme was applied, this problem could be reduced by only updating inner contact point.

Note that for some designs, such as using graph coloring to determine independent contact points and only have independent contact points under consideration at any time, an iteration needs to be well defined. So, nothing is lost. On the other hand, for other designs this is a strong constraint.

3.7.5 Preprocessing for Cache Coherency

A Jacobi or Gauss–Seidel scheme can be constructed so that it has a good cache coherency, that is, the memory is packed so that it is accessed sequentially when an iteration runs. The major exception is $\vec{w}$, which is likely to be accessed randomly, yielding a bad locality.

However, if one uses a preprocessing scheme, such as graph coloring or partitioning, we do not know the order in which contact points are processed. The sequence of memory accesses could be random, which is an exceptionally bad locality. This can cause performance to plummet. However, the sequence is likely to be the same from iteration to iteration.

This suggests that it would be a good idea to apply a permutation of the contact points so that the data is placed sequentially in memory w.r.t. order of execution on different processors. While this would likely improve the overall performance of the parallel schemes, such permutations are not likely to affect the scaling behaviour. Due to time constraints, the proposed permutation has not been implemented.

An interesting variation on this idea is to let each core construct its own variables, such as $\mathbf{J}$, $\mathbf{R}$, $\vec{b}$ and so on. This has the additional advantage that initialization costs are parallelized and gives an easy improvement of cache coherency, but it does require a static splitting into subproblems.

3.7.6 Real-time Usage

If one wishes to use a physics engine at interactive rates or even in real time, there are a number of additional issues to consider. To be interactive, there are some hard bounds on how much time the engine can use. At least 15 frames must be shown to the user each second, and one would prefer $30 - 60$ or even more if the refresh rate of the monitor is higher than 60 Hz.

At a framerate of 15 frames per second (FPS), each frame can use approximately 67 ms, including calls to a rendering function. At a framerate of 60 FPS less than 17 ms are available. Assuming latencies in the ms range for distributed systems, it should be clear that distributed physics computations are unlikely to be feasible if the subproblems need to communicate during computations. Thus, a shared memory architecture is better suited for a real-time application than a distributed system is.

3.8 Benchmarking

When assessing the parallel performance of a method, one would like to get an idea of how the method scales with the number of CPUs available. [IR09a] uses the concepts of strong and weak scaling to get a measure of performance. *Strong scaling* investigates how a problem of fixed size scales when the number of CPUs is increased. Usually the interesting metric is the speedup, that is, how much extra CPUs decrease the amount of time used. *Weak scaling* investigates how changing the size of a problem changes parallel performance. An example could be that splitting a large problem on p CPUs may scale better than a similar, small problem on p CPUs. This could be due to initial overhead, or many communications compared to actual work due to a large proportion of the tasks having external dependencies.

When looking at the scaling behaviour, there are two interesting metrics:

1. How does time usage per iteration scale w.r.t. the number of CPUs?
2. How fast can we get a result of the same quality w.r.t. the number of CPUs?

The first metric can be seen as the raw scalability of the scheme – if time usage per iteration is not reduced in a reasonable fashion, the method is unlikely to scale well with the other metric. The second metric is of particular interest for parallel Gauss–Seidel methods as we would like to get the convergence behaviour of the sequential Gauss–Seidel method, while getting the scaling behaviour of the Jacobi method.

To measure the second metric on a benchmark demo we use the quality measure γ defined in Section 3.6. We run the sequential Gauss–Seidel scheme for 50 iterations, and the quality is noted. This is the target quality for the other solvers.

We find the number of iterations needed for the sequential Jacobi scheme to reach similar quality by trial.

Actual measurements require construction of a number of demos, preferably demos that can be scaled up or down in size and complexity. The following sections will describe a few of these demos and present results for sequential factorized implementations of the Jacobi and Gauss–Seidel methods. These results will be used as base lines for assessing the quality of the parallel implementations.

Many of the demos have a large number of bodies, which cause contact generation to take up a lot of time. Because of this, the contact points have been pre-generated using a Bullet Physics compatibility layer, and are loaded at runtime. Pseudo randomness is used to generate some of the demos. Using pseudo random functions ensure that we get the same actual configuration, if the same randomness seed is used. Unfortunately, the implementations of pseudo random functions differ from system to system, so to ensure consistency between demo and contact data, these demos are loaded from a data file as well.

Aside from measuring the time used by the solver, the time used for preprocessing is also measured. The preprocessing timer is started when the contact points have been generated and includes construction of $\mathbf{W}$, $\mathbf{J}$, $\vec{b}$ and so on.

All benchmarks have been run through Minimum intrusion Grid (MiG) using the *amigos50.diku.dk.0* server (a.k.a. EightByEight) on the DIKU VGrid. The server has an AMD64 architecture with 8 cores and $8,192$ MB memory. The executable has been compiled with gcc $4.4.1$ on Ubuntu 9.10 32 bit with the `-static` flag and `-O2` optimizations. Most benchmarks are run 5 times, and the standard deviation σ of a series of time measurements $t_1, t_2, ...t_N$ is computed by

$$\sigma = \sqrt{\frac{\sum_i^N (t_i - t_{avg})^2}{N}} \tag{3.33}$$

where N is the number of measurements, and t_{avg} is the average value of t.

3.8.1 Ball Grid

The ball grid consists of a number of unit spheres placed in a grid on a large, fixed box (the ground). Two grid layouts are supported. In the first layout, the spheres are placed on top of and next to each other, so that all spheres touch their neighbours as illustrated

Name	**Dimension**	**Bodies**	**Contact Points**	$\mathbf{T}_{nnz}$
Small Ball Grid	$8 \times 8 \times 8$	513	1,408	17,728
Medium Ball Grid	$24 \times 24 \times 24$	13,825	40,320	757,824
Large Ball Grid	$40 \times 40 \times 40$	64,001	188,800	4,587,840

Table 3.2: *The table shows the three ball grid demos used for benchmarking.* $\mathbf{T}_{nnz}$ *is the number of non-zero blocks in* $\mathbf{T}$.

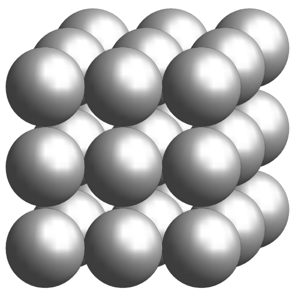


Figure 3.8: *The figure shows a simple* $3 \times 3 \times 3$ *ball grid in the standard configuration.*

on Figure 3.8. The second layout is a cannonball grid that reduces the unoccupied volume between the spheres. The demo also supports increasing the distance between the spheres along all three axis, as well as applying pseudo random velocities to the spheres. Finally, it is possible to set the height, width and depth of the grid.

The version used for benchmarking uses the layout illustrated on Figure 3.8, and no distance between the spheres, nor is any velocity applied to the spheres. Thus, the ball grid can be viewed as a dense structured demo. Benchmarking will be performed on three variations shown in Table 3.2. The benchmark demos are somewhat similar to the compaction granular media in used in [RDA04]. [TNA08] used a wall of parallelepipeds to benchmark their application, and ran a simulation where the wall collapsed.

The results for benchmarks are shown in Table 3.3. As expected, the sequential Jacobi scheme needs more iterations to reach the same quality as the sequential Gauss–Seidel scheme, and the sequential Jacobi scheme has a lower residual. The quality measure has been rounded to three significant digits, as the quality measure of some of the benchmarks change the 3rd figure from one iteration to the next.

Note that the execution time of the Jacobi and the Gauss–Seidel methods is almost identical. Profiling with Intel Parallel Studio 9.0 has shown that the factorized Gauss–Seidel schemes spend a significant amount of time in the function to update $\vec{w}$. There are two performance problems. First, the column product $\mathbf{WJT}_{\text{col }k}\Delta\vec{\lambda}_k$ has horrible cache coherency. Second, $\mathbf{WJT}$ is saved in compressed *row* format. As

Method	**Demo Size**	**Iterations**	**Quality**	**Residual**	**Solver Duration**	σ
Jacobi	Small	63	249	0.068	0.0933 s	0.0003
Jacobi	Medium	66	2790	0.035	3.20 s	0.003
Jacobi	Large	65	7890	0.035	14.3 s	0.003
Gauss–Seidel	Small	50	247	0.088	0.105 s	0.00015
Gauss–Seidel	Medium	50	2850	0.090	3.24 s	0.003
Gauss–Seidel	Large	50	7920	0.090	13.1 s	0.005

Table 3.3: *The table shows the performance of the sequential factorized Jacobi and Gauss–Seidel methods on the ball grid demos. Note that the Gauss–Seidel method uses significantly fewer iterations than the Jacobi method to get to a solution of the same quality.*

Name	Dimension	Bodies	Contact Points	$\mathbf{T}_{nnz}$
Small Ball Pile	$40 \times 40 \times 40$	64,001	6,605	7,415
Large Ball Pile	$80 \times 80 \times 80$	512,001	52,909	58,631

Table 3.4: *The table shows the three ball grid demos used for benchmarking. $\mathbf{T}_{nnz}$ is the number of non-zero blocks in $\mathbf{T}$.*

noted in Section 3.1 iterating over a column requires a binary search in all rows that have a non-zero block at that index. Both problems could be solved by implementing a compressed column format, but this has not been done due to time constraints.

The result that incremental recomputation of $\vec{w}$ is slower than computing it in one go is bad news for a factorized parallel Gauss–Seidel scheme, as it will need to update $\vec{w}$ when a contact point is updated.

3.8.2 Ball Pile

The ball pile constructs a pseudo random distribution of spheres. The spheres are placed in evenly spaced planes, but the distance between each sphere and its velocity is pseudo random, with the twist that the probability of a contact between two spheres can be configured. As such, this demo is not really a pile – the name is a reflection of its unstructured nature.

In the configurations used for benchmarking, the chance of contact is set at 25%. Note, however, that the velocity of the spheres is likely to make the actual number of contact points smaller than 25% of the number of bodies. The ball pile can be viewed as a largely unstructured demo. Benchmarking will be performed on two variations shown on Table 3.4. These unstructured benchmarks are somewhat similar to the benchmarks used by [IR09a] and the mixed granular media used in [RDA04], and likely similar to the benchmarks used by [CCH$^+$07].

The results of running the demos above are shown in Table 3.5. The Gauss–Seidel method converges significantly faster than the Jacobi method, and has a better quality. It should be noted that the large number of bodies in the large ball pile causes a preprocessing cost of approximately 2.6 seconds.

An interesting point is that the Gauss–Seidel iterations are faster than the Jacobi iterations. Profiling suggests that this is caused by the Jacobi scheme recomputing $\vec{w}$ in each iteration. To follows that the Jacobi scheme could be improved by using compressed vectors if $M \gg K$.

Method	Demo Size	Iterations	Quality	Residual	Solver Duration	σ
Jacobi	Small	63	0.000235	0.00026	1.63 s	0.0016
Jacobi	Large	65	0.0137	0.0049	14.1 s	0.005
Gauss–Seidel	Small	50	0.000219	0.00036	0.897 s	0.0009
Gauss–Seidel	Large	50	0.0148	0.0073	7.26 s	0.0018

Table 3.5: *The table shows the performance of the sequential factorized Jacobi and Gauss–Seidel methods on the ball pile demos. Note that the Gauss–Seidel method converges significantly faster than the Jacobi method both in terms of iterations needed and duration.*

Chapter 4

The Contact Graph

When parallelizing a task with inner dependencies, one common strategy is to partition the problem into tasks that are distributed among the available CPUs. This may require some additional communication, as local results may need to be forwarded to other CPUs. Different strategies can be formulated to construct such a partition, with the goal of reducing the *cost* of communication. Note that the cost of communication may depend on both the volume of communication as well as the number of CPUs that need to intercommunicate [IR09a].

It should also be noted that the cost of computing dependencies and dividing the problem into smaller tasks may be significant, particularly if performed sequentially when the number of available CPUs increase. Thus, the parallel performance of these methods have an effect. While parallel versions of the described methods will not be implemented, it may be interesting to do so in the future.

For the contact problem, the dependencies between the contact impulses of one contact point and another can be described by the contact graph. The contact graph is constructed by letting each body be a vertex and each contact point be an edge between the bodies in contact, as illustrated on Figure 4.1.

The contact graph in Figure 4.1 is not very general. There will often be multiple contacts between two objects, e.g. there can easily be 4-8 contact points between two boxes in a stack. These extra edges introduce a significant redundancy and possible overhead in the graph algorithms below that should be considered. One option could be to collapse the edges between two bodies into one weighted edge, that is, use contact manifolds as the basis for the contact graph instead of contact points. For convenience and clarity, we will assume that there is at most one contact point between each body.

A contact point is directly dependent on the edges from the two vertices at each end of the corresponding edge. Before we proceed, some general results from graph theory will be reviewed in the light of the contact graph, as most classic formulations of graph algorithms such as [CLRS01] centers on vertices, using edges as weights.

Definition 4.1 (Graph) *A graph G consists of a set of vertices V and edges E. Such a graph is written $G = (V, E)$. The number of vertices is $|V|$ and the number of edges is $|E|$. The neighbours of a vertex are all the vertices at the other end of edges incident to the vertex. Similarly, the neighbours of an edge are all the edges incident to the end-vertices of the edge. The number of neighbours of a vertex is called the degree of the vertex d_v, and the number of neighbours of an edge is called the degree d_e of the*

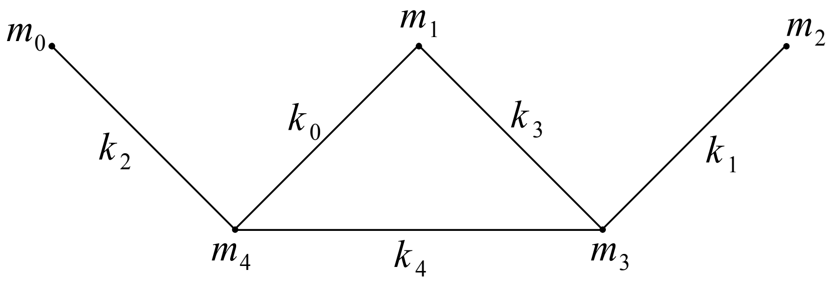


Figure 4.1: *The figure shows an example of a contact graph corresponding to the matrices in Figure 3.3. The vertices are the bodies in the scene, and the edges are the contact points. The subscripts correspond to the indices in* $\mathbf{J}$ *where* k_i *correspond to the* i*th row, and* m_j *is the* j*th column.*

edge.

4.1 Properties of the Contact Graph

From Definition 4.1 it follows that if an edge e_k has v_i and v_j as its endpoints,

$$d_{e_k} = d_{v_i} + d_{v_j} - 2 \tag{4.1}$$

as we do not want to count e_k when computing its degree. The neighbours of an edge e_k depend on the edge. Thus, d_{e_k} is the number of contact points that depend on e_k. By Corollary 3.1 this is exactly the information encoded into $\mathbf{A}$. Noting that we need to count e_k once for the diagonal of $\mathbf{A}$, (4.1) leads to Theorem 4.1:

Theorem 4.1 *Let the contact graph G have at most one edge between each vertex. If* e_k *is the edge in the contact graph that correspond to the* k*th contact point and* d_{e_k} *is the degree of* e_k*, the* k*th row in* $\mathbf{A}$ *has* $d_{e_k} + 1$ *non-zero blocks.*

This has the very important effect that the number of non-zero blocks in $\mathbf{A}$ is given by

$$\mathbf{A}_{nnz} = \sum_{k=0}^{K} d_{e_k} + 1 = K + \sum_{k=0}^{K} d_{e_k} \Leftrightarrow \tag{4.2}$$

$$\sum_{k=0}^{K} d_{e_k} = \mathbf{A}_{nnz} - K \tag{4.3}$$

where $\mathbf{A}_{nnz}$ is the number of non-zero blocks in $\mathbf{A}$. It follows from (4.3) that if the number of non-zero blocks in $\mathbf{A}$ is quadratic, then so is the sum of edge degrees in the contact graph. This is important as many of the graph algorithms that can be used to compute dependencies run in $\mathcal{O}(\sum_{k=0}^{K} d_{e_k})$ time.

[CLRS01] describe two typical representations of a graph – an adjacency list where each vertex has an entry with a list of outgoing edges or an $M \times M$ adjacency matrix where a non-zero element at i, j imply an edge between the ith and the jth vertex.

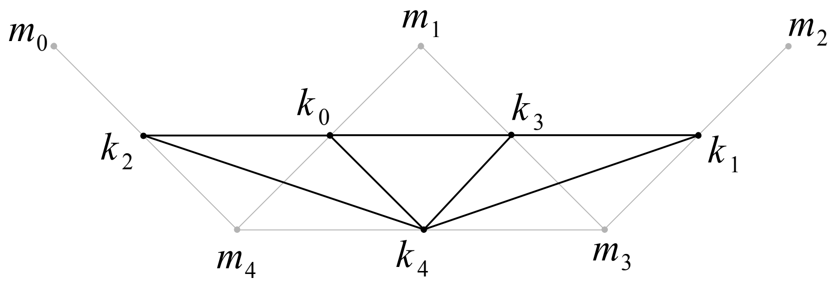


Figure 4.2: *The figure shows the vertex-centric representation of the contact graph in Figure 4.1. The vertex-centric representation is the dual of the edge-centric representation. Note that* $\mathbf{A}$ *in Figure 3.3 is the adjacency matrix of the dual.*

Due to our knowledge of $\mathbf{J}$ and $\mathbf{J}^T$, we have a convenient representation of the contact graph, on the form of an edge table.

Each row in $\mathbf{J}$ is an edge, and the column index of the two non-zero blocks describe which vertices are at the end points. Each row in $\mathbf{J}^T$ is a vertex, and the row index of the non-zero blocks describes which edges are incident to the vertex. Using the compressed row matrix representation in Section 3.1, this allows constant time lookup into the contact graph, even though there are some complications compared to a ”pure” graph structure. Given that $\mathbf{J}$ and $\mathbf{J}^T$ will have to be computed anyway, this representation of the contact graph has *no* overhead. The drawback is that the graph cannot be constructed before $\mathbf{J}$ and $\mathbf{J}^T$ has been constructed. This is a drawback as one may want to start manipulating a partial graph, or start as early as possible, using the list of contact points directly.

Much of the following work on graphs is centered on *vertices*, were the contact graph implies an emphasis on *edges*. This leads to the convenience definition:

Definition 4.2 *A graph represented by its edges is called an edge-centric graph, and a graph represented by its vertices is called a vertex-centric graph.*

As such, one could convert the contact graph to a vertex-centric representation where the contact points are the vertices, and there is an edge between the vertices v_i, v_j if the ith and jth contact point depend on each other. As previously noted, a vertex-centric representation of the contact graph is encoded into $\mathbf{J}^T$. Thus, a vertex-centric representation can be found at no additional cost.

This representation is, however, ill suited for computing dependencies of the contact problem. To get a vertex-centric representation of the contact graph that conserves the dependencies between the contact points, the dual of the contact graph will have to be computed. This information is encoded into $\mathbf{A}$. The dual of the graph in Figure 4.1 is shown on Figure 4.2.

4.2 Graph Partitioning

Intuitively, the graph partitioning problem can be described as finding a way to partition the graph into p sets so that the sets are roughly of the same size and has the

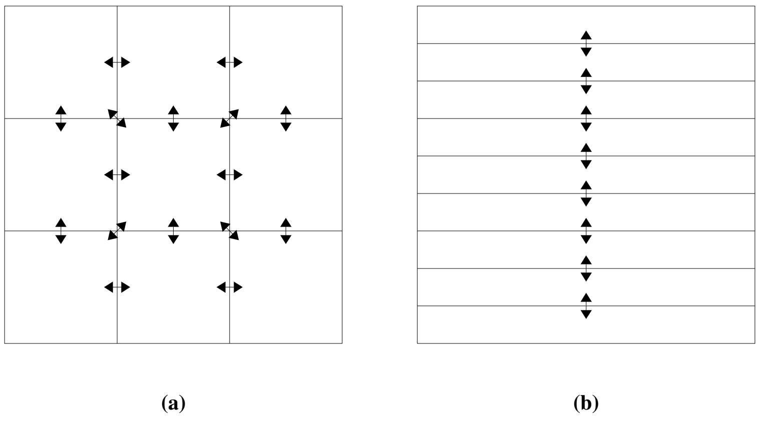


Figure 4.3: *The figure illustrates graph partitioning results by [IR09a] in 2 dimensions. In (a) the center box needs to communicate with all 8 neighbours, whereas each slice in (b) at most will need to communicate with two neighbours.*

least number of edges from one set to the other sets. A graph partitioning could be used to divide the contact problem into sets, where each set is placed on a CPU and computed separately. Results from the border points of one set could be forwarded to sets that contain dependent contact points between iterations – or one could relax or ignore the dependencies between sets altogether to get better parallel scaling. Some sort of checkerboard scheme could be used to further partition the sets into batches of border and inner vertices, similar to the scheme in [IR09a]. Before we define the graph partitioning, we define the notion of an *edge cut*:

Definition 4.3 (Edge cut) *Given an unweighted graph $G = (V, E)$ and a line L partitioning the graph, the edge cut of the partitioning L is the number of edges cut by L. If the graph is weighted, the edge cut is the sum of the weights of the edges cut by L.*

The notion of an edge cut can be used to define the p-way graph partitioning:

Definition 4.4 (p-way graph partitioning) *Given a graph G = (V,E), partition V into p subsets $V_1, V_2, ..V_p$ so that each set is disjoint and the edge-cut is minimized.*

In [KK98] it is noted that finding the optimal p-way graph partitioning is NP-hard. Standard methods of partitioning graphs can be grouped into three groups: Spectral partitioning, that are slow due to having to compute eigenvalues, geometric methods, that leverage geometric knowledge such as the coordinates of the vertices, and multilevel methods that partition a coarsed graph.

Multilevel methods are investigated in [KK98] and [ARK05] which show solutions of good quality and parallel scalability. A multilevel scheme is a scheme where the original problem iteratively is made more coarse in a manner where the more coarse level retains the structure of the finer problem in some way. This makes the problem easier to work with, and at the coarsest level, one can compute a good p-way partition. Once computed, one can iteratively go to a finer level and use the result from the previous level as a starting point of a refinement, hopefully getting a better solution.

A number of methods were investigated in [KK98], and experimental results suggest that Heavy Edge Matching (HEM - vertices with heaviest edges were merged) was the best method for coarsening, Greedy graph growing partitioning (GGGP - vertex with largest decrease in edge cut inserted first) for partitioning and Boundary KL (BKL - refinement along the boundaries) for refining the solution from the previous level. The authors also suggest that this method can be parallelized reasonably well, citing their own research that allowed them to gain a $\times 57$ speedup with 128 cores in some cases.

Due to time constraints, graph partitioning will not be investigated further in this thesis. It is worth noting that one could use the Metis library for graph partitioning methods. However, the notion of optimal partitioning may not be similar to the edge cut defined above. Rather, it is likely that we would prefer to minimize the number of neighbours, while keeping the number of contact points in each subset reasonably close [IR09a]. Two partitioning strategies are illustrated in Figure 4.3. Note that the price of the thin slice design is that the proportion of internal nodes may decrease. Indeed, some of their results suggest that the slices should be relatively large for that design to perform well.

4.3 Graph Coloring

Let us start by defining the graph coloring problem for an edge-centric graph.

Definition 4.5 (Graph Coloring) *Given a graph $G = (V, E)$, the graph coloring problem is to give a color c to all edges E so that the color of the kth edge is different from the colors of its neighbours. If n colors are used, the coloring is called an n-coloring of G. If n is the minimum number of colors needed to color a graph, this coloring is considered optimal.*

Computing the optimal coloring (or even the optimal number of colors) for an arbitrarily graph is known to be NP-hard [LC94]. It is, however, possible to compute a decent coloring very fast using a greedy method. The color is often represented by an integer number. The greedy method considers each edge in turn. For each edge e_k it

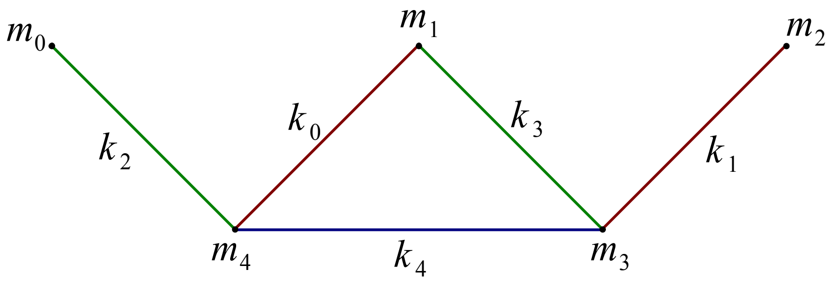


Figure 4.4: *The figure shows an example of a coloring of the contact graph from Figure 4.1 that uses 3 colors. If a greedy method is used, the ordering of the contact points yields an optimal coloring. If k_0 and k_2 were swapped, a greedy method would use 4 colors, which is the highest edge degree.*

investigates the colors (if any) of the neighbours of e_i and assigns the lowest color not taken. If the degree of e_k is d_{e_k}, the cost of this algorithm is $\mathcal{O}(\sum_{k=0}^{K} d_{e_k})$. As it must be possible to assign a color to any edge, the highest number of colors assigned will be equal to the highest edge degree $d_e^{max} + 1$.

A greedy method may construct a highly unbalanced coloring, where the first colors are used much more than the later colors. An alternative to the greedy coloring is to define a balanced scheme [ABC$^+$95], where the color picked is the available color that is in least usage. This can, however, pose a problem as we would need to know the number of colors to be used before starting. This can be handled by a two-phase pass: The first computes the highest edge degree, and the second pass perform the actual coloring. From Corollary 3.1 it should be clear that the highest edge degree can be found in $\mathcal{O}(|E|)$ time. A somewhat more expensive option is to compute a tentative coloring and use this number of colors for the second pass. One could also require that the number of colors used is a multiple of the number of CPUs available.

A number of parallel methods have been proposed, but many of them, such as [LC94], [ABC$^+$95], and [GM00] either scales horribly or does not provide information on scaling. [BBC$^+$05] proposes a method that scales for a low number of processors, but a significant falloff can be seen at 16 processors.

Their approach is to partition the graph into subsets, that are colored concurrently, handled by each processor. Instead of iteration, they use a term ”round”. Each round is partitioned into two phases:

1. Tentative coloring
2. Conflict detection

The first phase is subpartitioned into supersteps. During each superstep, the processor handles a number of vertices. In the end of each superstep, the processor exchange information on coloring of border vertexes, and it is decided which of the processors need to update their vertex. This allows reduction of communication between the processors. Note that a graph partitioning is needed, which one would like to be able to parallelize as well.

A similar method is used in [BCG$^+$05] on a variation of the graph coloring problem, where the color must be unique among the neighbours and the neighbours of the neighbours on a distributed system, which scaled reasonably well up to 16 processors.

4.3.1 Application to Gauss–Seidel

A colored contact graph can be used to tell which contact points are independent of each other. Contact points of the same color are independent of each other, from which it follows that they can be computed embarrassingly parallel.

Batches of one color

If contact points are considered in batches, where each batch consists of one color, each batch can be distributed among the available processors, an updated solution computed and the workers are ready for the next batch. This has one important drawback: The need for synchronization is high. Batch c cannot be started before batch $c - 1$ is done, or problems with race conditions will ensue.

More colors means more waits, which implies that the number of colors can have a significant impact on the parallel scaling of a Gauss–Seidel scheme. The potentially

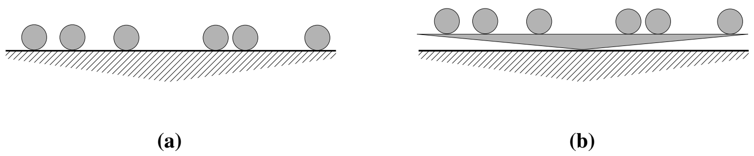


Figure 4.5: *The figures show (a) six balls on the fixed ground and (b) six balls on a rotated pyramid on the ground. In the first example, the contact points between the balls and the ground can be viewed as independent. In the second example the contact points between the balls and the pyramid are dependent.*

large number of synchronizations suggests that a graph coloring scheme is ill suited for a distributed solver, but may work well on a shared memory architecture.

Unbalanced colors

As noted earlier, a highly unbalanced coloring may be produced. Consider a setup where m balls are shot onto a box from m cannons. If they all hit the box in the same timestep, all m contact points will be contacts between the box and a ball. Thus, all contact points depend on each other. It follows that such a setup will require m colors. In this case, each batch will consist of 1 contact point, which will cause the parallel solver to become very slow. Similar issues may surface in other setups.

If the object(s) causing the unbalanced coloring is fixed, the contact points in the unbalanced colors may very well be independent. Consider m spheres at rest on the ground as shown on Figure 4.5 (a). If the ground is fixed, the spheres will not affect the movement of the ground. Indeed, if the spheres do not touch each other, all contact points are independent, even if a naive graph coloring will view them as being dependent. This suggests that custom handling of fixed objects can yield a coloring with a better balance. Due to time constraints such a handling has not been implemented.

However, if the spheres bounce up and down on a rotated pyramid on the ground, as illustrated on Figure 4.5 (b), the contact points between the pyramid and the spheres will be dependent on each other, and will need to be handled in a reasonable manner.

A simple way of detecting if a color is unbalanced is to compare it to the number of available processors. If there are more processors than contact points in the color, the color is highly unbalanced and some action is needed. One could also define some multiple n on the number of processors, so that if there are fewer than np contact points in the color, the color is assumed to be unbalanced.

One option is to merge unbalanced colors into an *unsafe* color. The new color is unsafe, as any contact point in the new color could depend on another contact point in that color. This new color will likely need to be handled by a Jacobi method or by a naively parallelized Gauss–Seidel method. The advantage of this solution is that it has good parallel performance, the drawback is that we are unable to leverage the interdependency of the contact points in that color for faster convergence.

This can be viewed as a relaxation of the color-based parallelization, and solvers based on this heuristic will be called relaxed.

Chapter 5

Communicating Sequential Processes

Ever since the invention of multi-threaded programming, often just called multiprogramming, the issue of writing efficient, multi-threaded programs has been a challenge for many programmers. Common errors include deadlocks, starvation, livelocks and race conditions, all of which are linked to performance issues, such as overzealous synchronizations on shared resources.

Initially, multiprogramming was used to ensure that the performance of a program would not be hindered by waiting for a particular event, such as user input, while ensuring the experience of a responsive system, or allowing the appearance of multiple programs being active ”at once”. Users would often have one available CPU, so usually each process would be charged with solving one task. However, when systems with multiple CPUs became available, distributing one task over multiple CPUs became very interesting.

Techniques or frameworks to assist the programmer in the construction of multi-threaded programs were very much needed. Communicating Sequential Processes (CSP) is a theoretical framework for the construction of multi-threaded programs, introduced by Hoare in 1978 [Hoa78]. A detailed description of CSP is beyond the scope of this thesis, but a thorough introduction by Hoare is available online [Hoa04].

Other commonly used multiprogramming frameworks include MPI and OpenMP.

Open Multi-Processing (OpenMP) uses compiler macros to generate parallel code. This has the advantage that it is easy to parallelize embarrassingly parallel problems directly in the sequential code. If the problem can be divided into embarrassingly parallel tasks, OpenMP can also be used with great ease. This ease comes at a cost, however. It is difficult to leverage the structure of subproblems, and OpenMP focus on shared memory architectures.

The *Message Passing Interface* (MPI) is somewhat similar to CSP in the sense that it relies on passing messages between processes. The interface is fairly low level, which makes message passing somewhat complex. On the other hand, it is widely used, and is well suited for communication in a distributed system.

An alternative to using the CPU is to use the GPU with OpenCL or NVIDIA’s CUDA. Both have the drawbacks inherrent to GPU solutions, notably that GPUs cannot intercommunicate – the CPU has to assign work to the GPUs. It is also noted that

CUDA is only supported on NVIDIA GPUs, and OpenCL is still a fairly new standard.

5.1 The Basics of CSP

A CSP system consists of processes and channels. The processes perform the actual work, and all communication between processes uses channels. As such, processes are similar to functions or procedures in programming languages, and channels can be viewed as function calls.

A process is often viewed as a black box, and assumed to be correct. If two processes execute at the same time, they are said to *run in parallel*. If one process can execute before, after or at the same time as another process, these processes are said to *run concurrently*.

A CSP process has an alphabet that describes the legal actions of the process, as well as a state diagram, describing which actions can be taken after each other. An example could be a vending machine, whose alphabet could be to *accept payment* and *dispense chocolate*, where it is only allowed to dispense chocolate after receiving payment.

Some of these actions may require synchronization with another process. In the example of the vending machine above, the *accept payment* action can only be performed if a customer is *presenting payment*. These synchronizations are performed over channels, that is, if one process wants to send a message to another process, it has to wait until the other process is ready to receive the message. As unneeded synchronization can considerably slow down a system, CSP includes the notion of a buffered channel, where one process can write to a buffer and proceed. At some later time, the other process can read from the buffer. In the previous example, this could be viewed as the vending machine being able to continuously accept payment and dispense chocolate while enough payment has been received.

Most CSP implementations support a number of channel types. The simple channel is a one-to-one channel, where two processes communicate. Communication is one-way, so one process can read from the channel and one process can write to it. However, CSP also support multiple readers or writers on one channel, yielding the any-to-one, one-to-any, and any-to-any channels. If two-way communication is needed, two channels must be used.

These processes and channels can be described using the CSP formalism, and it is possible to formally prove that the system is free from deadlocks, livelocks and race conditions – or that it suffers from those exact problems. It is also possible to make a specification for a system, and prove that the system satisfies these specifications.

Another interesting point is that the formalism allows programmers to design a multi-threaded system from processes and channels alone. Further, a system of processes and channels could be viewed as a process itself. These processes could be used as building blocks in a larger, more complex system. Note that this is not without its dangers, as the internal structure of a block could cause unexpected behaviour.

However, this suggests that a multi-threaded system can be designed quite easily, one could even use a GUI to design the system. CSPBuilder [FW08,CSP09] provides a GUI for PyCSP, which is a Python implementation of CSP [BVA07,PyC]. Aside from Python, CSP has been implemented in a number of programming languages, such as Occam [KRo], Java [JCS], C++ [C++], C# and Delphi [Jib].

An example of a CSP design is shown in Figure 5.1. The boxes are processes, and the lines are channels. The arrows show the direction of the transaction: The process

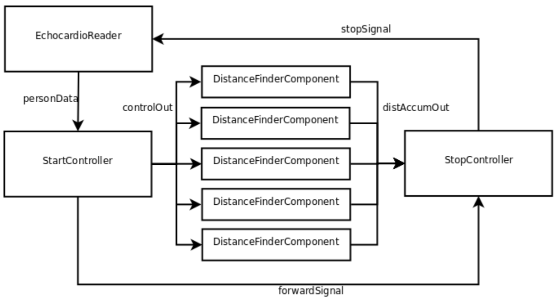


Figure 5.1: *An example of a CSP design. The boxes are processes, and the lines are channels. This particular design is used to read a number of data points, and use these points to guess missing data for other points. It was developed as a part of an earlier assignment.*

without an arrow writes to the channel, and the process with an arrow pointing in reads from the channel. As such, all types of data structures can be communicated over channels, but with C++ templates, it is possible to specify which types a particular channel accept. Thus, the compiler can be used to assist the programmer.

As the formalism is independent of operating system and underlying hardware, CSP can also be viewed as an abstraction layer. This is desirable as it becomes possible to implement a highly portable multi-threaded system – all we need is to ensure that the CSP framework is implemented on the target OS or hardware. This makes CSP very interesting compared to MPI.

In this thesis, we are primarily interested in performance, portability and ease of design, and not so much in formally proving the design correct. Both C++CSP and Jibu for C++ are distributed under the GNU Lesser General Public Licence, versions 2.1 and 2.0, respectively.

5.2 Performance

Operating Systems (OS) have different types of threads, often referred to as *kernel-level* and *user-level* threads. A kernel-level thread runs in the kernel, and if a process has two kernel-level threads, each thread can run on separate CPUs. A user-level thread lives inside a kernel-level thread, but all user-level threads in a kernel-level thread must run on the same processing unit as the kernel-level thread. User-level threads use less memory than kernel-level threads and context switches from one thread to another are significantly cheaper.

The design and implementation of threads are different from OS to OS, but the most widely used OS's, such as Windows, Mac and Linux, support the above notions in one form or another [SGG04]. There is a performance hit whenever a context shift is performed, particularly between kernel-level threads. Why is this important? Well, the differences between kernel-level threads and user-level threads are one of the things that C++CSP does not abstract from. By default it builds kernel-level threads, but one can specify the need for user-level threads.

As noted earlier, context shifting has a price. In CSP, communication between two processes requires a synchronization between the processes. This is particularly expensive between kernel-level threads, but we need kernel-level threads to take advantage of multiple CPUs. This can be reduced somewhat by adding a buffer to the channels, as the need for explicit synchronization between the processes is reduced, leading to fewer context shifts.

However, we still need to ensure that the time between each communication is large compared to the time needed to perform the communication. And we really would like to minimize the time used for context shifts, that is, minimize the number of threads running concurrently.

As will be shown empirically in the next chapter, having many user-level threads running concurrently inside kernel-level threads collaborating on the same task is significantly less efficient than few kernel-level threads, each with its own large chunk of the problem.

5.2.1 Shutting Down the System

An important part of any algorithm is to know when to terminate. For most functions this is handled by using a `return` statement. However, when the CSP network is done, it will need to be shut down. Notably, the processes will need to shut down. A standard method of handling this is to *poison* the channels in the network. Whenever a process tries to read from a poisoned channel, it should poison all available channels and shut itself down.

The notion of poisoning channels is supported by many CSP implementations, including C++CSP. There is, however, one significant performance problem. Often poisoning is implemented by letting the channel throw an exception whenever a process tries to read from or write to it. This exception can be caught and handled by the process.

Using exceptions comes at a performance cost, depending on the system. Some implementations introduce an overhead on a try-catch-finally loop, but catch exceptions cheaply, whereas others handle try-catch-finally loops cheaply, but at a large cost when exceptions are caught. Some simple benchmarks in Windows 7 with the Visual Studio 2008 compiler suggested an overhead of 10 % longer execution times, compared to handling shutdown via signals over channels. Naturally, it is a minor complication of the design to handle shutdowns explicitly, but the performance advantage makes it worth it.

5.2.2 Bugs

Usage of C++CSP showed a number of bugs that needed to be handled when using C++CSP on Windows 7 with Visual Studio 2008, as well as Ubuntu 9.10 with gcc 4.4.1. The changes made to C++CSP 2.0.5 are

- To make the project compile, `time.h` had to be renamed to `time_util.h`.
- Changed line 167 of `cppcsp.h` from `#define _WIN32_WINNT 0x0403` to `#define _WIN32_WINNT 0x0503`, in order to ensure that `winbase.h` is loaded. If `winbase.h` is not loaded, usage of `ConvertFiberToThread` caused a compile error in the C++CSP library.
- `WhiteHoleChannel` had unimplemented abstract components causing a compile error when the constructor was called. The abstract components have been implemented. `BlackHoleChannel` is unlikely to have the same problem.

The `WhiteHoleChannel` mentioned above is important when it comes to making *skip guards*. A skip guard can be placed on a channel to allow a process to investigate if there is something in a channel without halting the process. If there is something, the process will read from the channel, and if there is not, the process can continue working.

For convenience, the C++CSP project files have been edited to use CMake instead of make so that it is easier to use with different IDEs. Note that Linux/Mac support is hardcoded to Ubuntu 9.10 in the supplied cmake project files. The CD contains the original C++CSP 2.0.5 make files.

Chapter 6

Parallel Jacobi

In previous chapters, we have investigated the contact problem, its solutions and discussed parallelism. This chapter will investigate how to make an efficient parallel implementation of the Jacobi method. There are two good reasons to start by investigating the parallel Jacobi method.

First, it is embarrassingly parallel, which makes it both fairly easy to implement, and easy to test different parallel designs.

Second, we would really like to know if the overhead of a parallel Gauss–Seidel scheme eats up its performance advantage over a Jacobi scheme. If a parallel Gauss–Seidel scheme is unable to outperform a parallel Jacobi scheme, there is no reason to use a parallel Gauss–Seidel scheme as it will be both more complex and slower.

Two Jacobi schemes will be investigated. The first is the factorized scheme, where we maintain $\vec{w} = \mathbf{W}\mathbf{J}^T\vec{\lambda}$, as described in Section 3.4.2, and the other is the $\mathbf{T}$ scheme, as described in Section 3.4.1.

As noted in Section 3.4.3, the strength of the $\mathbf{T}$ scheme is that the loop is likely to be fast if there are few dependencies between contact points, and that the only variable to synchronize between iterations is $\vec{\lambda}$. On the other hand, we have to compute $\mathbf{T}$, which is likely to be expensive, and, in case of diverging behaviour, it is expensive to update. As the rows of $\mathbf{T}$ have different lengths, we will also need to compute a reasonable division of contact points between the workers.

The factorized scheme has the strength that the work performed for each contact point is likely to be very similar, which makes division of work very easy. On the other hand, we have to update $\vec{w}$ with the results of each iteration, so it is ready for the next iteration. While we know that each contact point at most updates two entries in $\vec{w}$, one entry in $\vec{w}$ can be dependent on all contact points.

In the following, $\vec{x}$ will be the current guess of the correct solution $\vec{\lambda}$.

6.1 Extreme Factorized Scheme

In this scheme, the full problem is subdivided into the smallest tasks possible: Each contact point is one task. It is in this sense that the scheme is extreme.

A pipeline is constructed, where a dispatcher sends x_k and w_i, w_j corresponding to the kth contact point to the workers. The workers solve the problem and send the new x_k, the updates to w_i, w_j and the residual to an accumulator. The accumulator updates the local copy of $\vec{w}$, $\vec{x}$ and the residual. When all contact points have been considered, the iteration is done. The accumulator updates $\vec{x}$ and $\vec{w}$, and investigate if the solution is converging, diverging or if we are done. This information, along with the updated $\vec{w}$ and $\vec{x}$ are sent to the dispatcher, and a new iteration can start. This design is illustrated on Figure 6.1. Note that each worker is initialized with a pointer to $\mathbf{J}$, $\mathbf{J}$WT, $\mathbf{R}$, ν, $\vec{\mu}$ and $\vec{b}$.

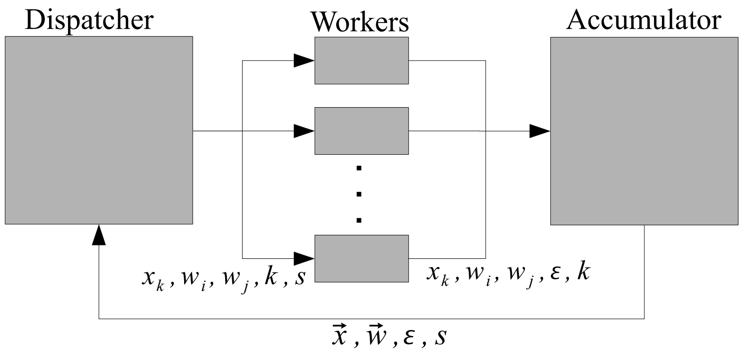

Dispatcher
Workers
Accumulator
x_k, w_i, w_j, k, s
$x_k, w_i, w_j, \varepsilon, k$
$\vec{x}, \vec{w}, \varepsilon, s$

Figure 6.1: *A graphic representation of the extreme factorized Jacobi method, using a producer-consumer model. The dispatcher makes tasks that are handled by the workers, who send their results to the accumulator. The variable s is a status variable, noting if the network is to shut down.*

Test runs show that this design performs horribly: The time usage per iteration is an order of magnitude slower than the sequential implementation, and the initial implementation did not scale beyond the second CPU. This is not entirely unexpected, as communication between processes on different CPUs is significantly more expensive than communication between processes on the same CPU. The design also has the problem that a global lock is instituted whenever an iteration has finished.

The problem of communication costs can be addressed by reducing the volume of communication. If pointers to the blocks are used instead of raw data, the volume can be significantly reduced. This, however, does not improve performance much. This suggests that C++CSP channels and processes do not perform pre-fetching. So, the full cost of a communication is incurred on every, single contact point. It follows that either C++CSP must be expanded with a clever pre-fetching scheme, the workers must implement pre-fetching, or a design that reduce the number of communications must be implemented.

The problem of global locks can be alleviated by placing the dispatcher and the accumulator in the same kernel-level thread. This also has the effect of allowing the model to scale to two CPUs, as shown on Figure 6.2. One could also merge the two components into one. This makes the design of the component slightly more complex, but did not improve the performance much.

Figure 6.2 suggests that in order to get good parallel performance, we need to maximize the work performed by each worker, that is, use as few workers as possible, with a large workload.

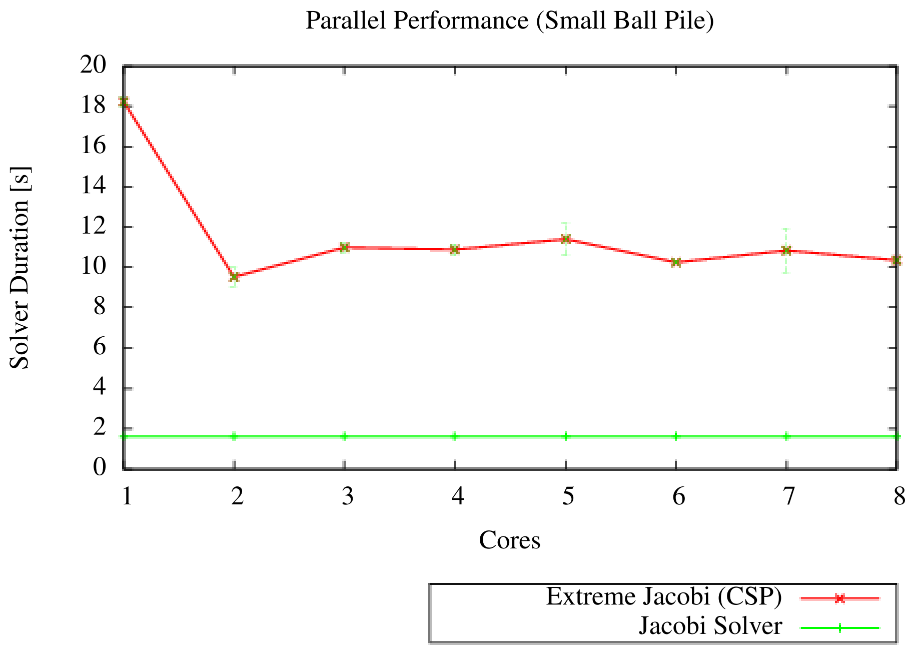


Figure 6.2: *The figure shows the performance of the extreme Jacobi solver compared to the sequential Jacobi solver. Note that the extreme Jacobi solver perform significantly worse than the sequential solver, and that adding cores seems to make little difference beyond the second.*

6.2 Factorized Scheme

In this design, each worker is initialized with a pointer to the constant data, the result vectors $\vec{x}_{in}$ and $\vec{x}_{out}$, and the range of contact points, the worker is responsible for. Each worker has one in-channel where it gets a status message, which tells it to continue, handle divergence or shut down. It sends the residual of the locally handled contact points and the *update* to $\vec{w}$. The value of $\vec{x}$ can be updated locally, as the worker is the only process using it.

The out-channel of the worker is read by the accumulator. The accumulator sums the updates to $\vec{w}$, and when all workers have finished, the iteration is finished. Between iterations, the current residual is compared to the previous residual to investigate if the solution is converging or diverging. If we are converging, the updates are applied to $\vec{w}$. The status of the solution is sent to the workers, along with a pointer to the updated $\vec{w}$. Note that the workers are started and perform the first iteration without waiting for the accumulator.

The pointer to $\vec{w}$ is included to ensure that values from $\vec{w}$ are reloaded by each worker. The `volatile` keyword could be used instead, but would require that the BML library was expanded with separate volatile functions. It would also reduce performance, as we only need to reload $\vec{w}$ once per iteration.

This has the nice effect of heavily reducing the weight of communication, as only the pointer to the updated $\vec{w}$, and the two status variables (residual and status) need to be passed over the network.

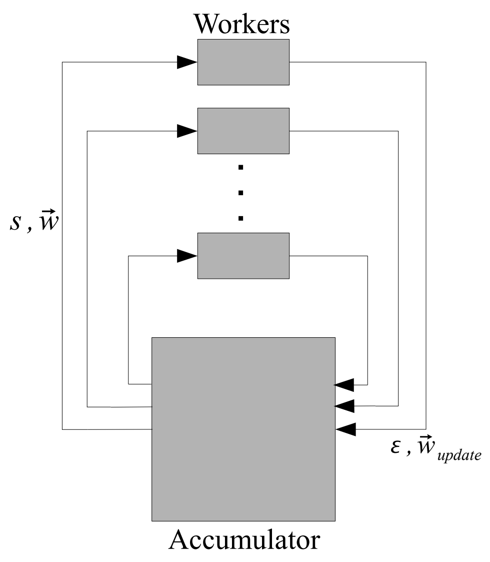


Figure 6.3: *A graphic representation of the factorized Jacobi method. The workers are initialized with a portion of the task, and send updates to $\vec{w}$ and the residual to the accumulator. The accumulator investigates convergence behaviour, and in case of convergence, the accumulated updates to $\vec{w}$ are applied to $\vec{w}$ before the next iteration starts. When the next iteration starts, the accumulator sends a status s message to the workers and a pointer to $\vec{w}$.*

For performance reasons, the channel from the workers to the accumulator is buffered. If we had only one `One2AnyChannel` from the accumulator to the workers, the system is prone to starvation and race conditions, where one worker could deliver its results and "steal" the go-ahead signal from another worker. One range of contact points would be run twice in that iteration, while another would not be run at all. This could cause starvation and race conditions that could yield divergent behaviour. To protect the system from this problem, the accumulator has an explicit channel to each worker. To alleviate the performance hit due to this design, this channel is buffered with a memory capacity of one signal. The design is illustrated on Figure 6.3.

6.2.1 Performance Analysis

The usage of pointers to non-constant data is a violation of the CSP design pattern, but it yields a good performance. The design protects against synchronization issues. There is, however, still one major problem: The entire system is locked while $\vec{w}$ is updated, which will limit the parallel performance boost as more processors are added. Indeed, benchmarking suggests that this causes scaling to plummet when 4–5 processors are used.

To alleviate this issue, a compressed block vector data structure has been implemented. The point is that only the elements of the updates to $\vec{w}$ that can be non-zero are explicitly represented. This reduces the time needed to update $\vec{w}$, the data transmission time when the workers transfer data to the accumulator, and the time used by the accumulator to compute the total update to $\vec{w}$.

However, it does not fix the issue that causes the problem: The total lockdown of the system between each iteration. A further improvement is to maintain two copies of $\vec{w}$. One of them, $\vec{w}_{out}$, is updated by the accumulator as the results are in from the workers, while the workers use the other, $\vec{w}_{in}$. Aside from having to allocate more memory, this solution has the drawback of needing to copy from $\vec{w}_{out}$ to $\vec{w}_{in}$, but this can be performed while the workers are working on their tasks. Thus, the lockdown between each iteration can be reduced significantly.

While this optimization reduces the lockdown time, it does not remove it. Section 3.7.4 suggests that in the average case, the workers will get a go-ahead signal from the accumulator. So, if a local $\vec{w}_{local}$ is maintained by each worker, the workers can start the next iteration. Indeed, with incremental updates of $\vec{w}_{local}$ using the local updates, non-exhaustive tests showed that with 8 cores a lookahead scheme could handle up to $15 - 20\%$ of the total number of contact points during the lockdown. Unfortunately, this is not a golden grail: Using local updates often showed divergent behaviour. This is likely due to the fact that the contact points on a worker may be highly dependent on contact points controlled by other workers. If some graph partitioning scheme were used, using contact points on the inside of the contact graph allocated to the worker might yield good performance. However, such a scheme does not generalize to Gauss–Seidel schemes using graph coloring. Worse, it incurs some overhead as data consistency needs to be maintained.

Looking more closely at the problem reveals some interesting features. The accumulator is mostly unused during execution, but when the workers finish, they communicate their results to the accumulator, pretty much at the same time. This may clog

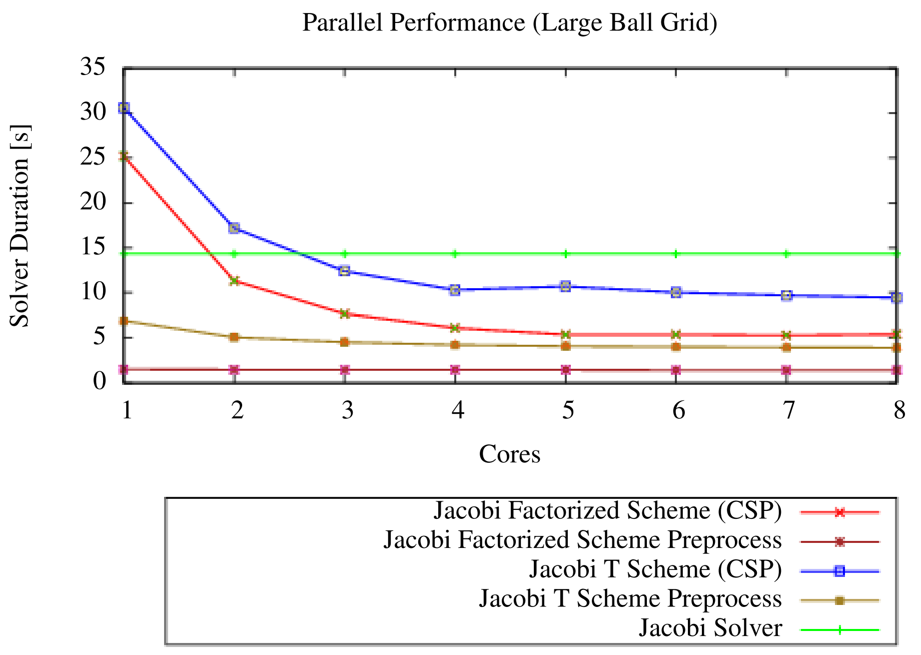


Figure 6.4: *The figure shows how the factorized scheme scale by iteration for the large ball grid. Each extra cores reduce time usage per iteration until the 5th core, where extra cores add little improvement. Note that 3 cores are needed for the* $\mathbf{T}$ *scheme to perform faster than the sequential factorized scheme.*

Method	**Demo**	**Speedup w.r.t 1 core**	**Speedup w.r.t. sequential**	**Optimal Cores**
Factorized Scheme	Small ball pile	2.4	2.6	8
Factorized Scheme	Large ball pile	2.8	2.8	8
Factorized Scheme	Small ball grid	3.1	2.6	4
Factorized Scheme	Medium ball grid	3.7	2.9	7
Factorized Scheme	Large ball grid	4.8	2.7	7

Table 6.1: *The table investigate weak scaling of the factorized scheme using CSP by looking at the speedup compared to the size of the problem. The fastest time is used and compared to the time usage using one core and the time usage of the sequential Jacobi method.*

the accumulator, causing a longer lockdown. Profiling the program for locks and waits using Intels Parallel Studio 9 showed that the workers spend a lot of time waiting for the go-ahead signal from the accumulator, which support this theory.

One response to this insight may be to use a technique from data bases: *Paging*. The point is that instead of sending the entire result, one could send portions of the entire results, called pages. This would increase the number of communications, so we would prefer a large number of contact points in each communication. Test runs suggest that paging does not improve parallel performance, but it does have a significant influence on how much time is spent on each iteration, and for well chosen page sizes, the time usage per iteration can be reduced with more than 10%. Simple benchmarking suggests that a good page size lies between 50 and 100 contact points.

The failure of paging to improve parallel performance suggests that either the bottleneck is elsewhere or there is another bottleneck. One possible bottleneck could be the channels themselves. As such the channels can be viewed as task queues who are prone to being bottlenecks in multithreaded applications [KHN07].

So far, `Any2OneChannels` have been used to send results from the workers to the accumulator. Thus, whenever one worker needs to broadcast a signal to the accumulator over the buffered channel, it will need to acquire a lock on the channel to ensure that it is the only process to work on it. Profiling with Intel Parallel Studio showed that significant time is used by workers to wait to be able to write to the shared channel, even if the buffer on the channel has not used its capacity. This problem could be reduced by letting each worker have its own channel to the accumulator, and place an input guard on the accumulator, so that it reads from whichever channel is ready.

Performance plots for the large ball grid and ball pile are shown in Figure 6.4 and Figure 6.5, respectively. In both cases, the factorized scheme scale well until the 5th core is added, and stagnates thereafter. It is interesting to note that the time usage when using one core is 1.76 times as high as the sequential method. Some testing show that the parallel scheme has a high startup cost, and each iteration costs 1.5 times as much as the sequential scheme.

The higher per iteration cost is partly due to updates to $\vec{w}$ being computed by the workers. The high startup costs are the costs of computing the form of the compressed vectors to handle the paged updates of $\vec{w}$. This cost can be reduced significantly by using a more clever approach. Note that the workers construct the compressed vectors they need, so the startup costs are performed concurrently, and tests suggest that these computations scale well.

Weak scaling is illustrated in Table 6.1, where it can be seen that larger demos

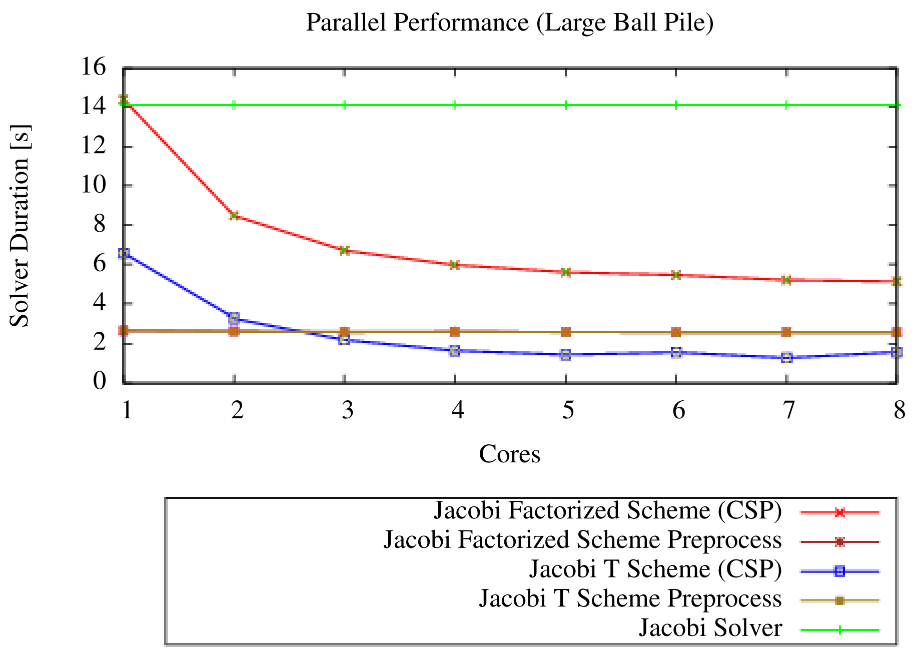


Figure 6.5: *The figure shows how the factorized and* $\mathbf{T}$ *scheme scales by iteration for the large ball pile. Each extra cores reduce time usage per iteration until the 5th core, where extra cores add little improvement. The preprocessing costs are dominated by the large number of objects in the scene and the preprocessing costs of the two schemes is similar. Note that even with only one core available the* $\mathbf{T}$ *scheme is significantly faster than the sequential factorized scheme for a similar number of iterations.*

scale better than smaller demos when looking at the per iteration price. Compared to the sequential cost, there is little change.

6.3 T Scheme

The $\mathbf{T}$ scheme has the advantage that there is no need to neither compute nor update $\vec{w}$. We do, however, need to ensure that no value of $\vec{x}$ is written while another process is reading from it. This is enforced by giving the workers pointers to two $\vec{x}$ vectors. Each worker will read from one of the vectors and write to the other. Which one to use is enforced by the accumulator. When the worker is done, it sends the residual to the accumulator and waits for further instructions.

When all workers have sent their residuals to the accumulator, the accumulator computes the status of the iteration, and send a status message to each of the workers, similarly to the factorized design. The design is shown on Figure 6.6.

While all workers have to wait between iterations, the time used waiting is very short, and independent of the size of the problem. This suggests that the $\mathbf{T}$ scheme will scale better than the factorized scheme.

On the other hand, we will need to compute $\mathbf{T}$. In the worst case, $\mathbf{T}$ can have a quadratic number of non-zero blocks as noted in Section 3.2. However, the size and

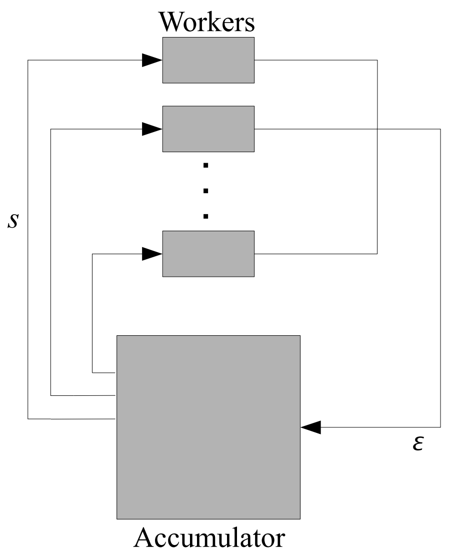


Figure 6.6: *A graphic representation of the a Jacobi* $\mathbf{T}$ *scheme. The workers are initialized with a portion of the task and send their local residual to the accumulator, who in turn investigate convergence behaviour. The status is sent to the workers in the variable* s*. The workers handle convergence updates, as* $\mathbf{T}$ *and* $\vec{c}$ *can be updated embarrassingly parallel.*

structure of $\mathbf{T}$ can be computed very fast, and the expensive parts (computing each block in $\mathbf{T}$) are embarrassingly parallel. Computing $\mathbf{T}$ row by row is likely to have better performance, but as discussed in Section 3.7.3 requires some scheduling, as the rows can have different lengths.

This uneven structure of $\mathbf{T}$ also has a significant effect on how the contact points can be distributed among the workers in an efficient and balanced manner. The greedy solution described in Section 3.7.3 has been found to work well.

However, a greedy scheme may still produce an unbalanced division, where one worker gets significantly more work than the others. A greedy paging scheme could be added, where each page is a certain percentage of the work load per worker. A reasonable size of the workload could be found by dividing the number of contact points per worker into a number of subtasks. The scheduling performance will increase with the number of subtasks, but overall performance will deteriorate. Some benchmarking suggests that using 100 subtasks yield a good performance if there are enough contact points. If the number of contact points is low, setting the minimum number of subtasks at 10 may be reasonable.

The parallel performance of the $\mathbf{T}$ scheme is shown on Figure 6.4 and Figure 6.5 for the large ball grid and ball pile, respectively. As can be seen the per iteration cost scales reasonably well up until 4 cores are added, where a significant falloff can be seen.

Another interesting point is that, as expected, the $\mathbf{T}$ scheme performs significantly better on the unstructured ball pile than the factorized scheme, but performs significantly worse on the structured ball grid. Weak scaling is shown in Table 6.2, where it can be seen that larger unstructured demos scale better than smaller demos, but the $\mathbf{T}$ scheme scales badly for larger structured problems.

Method	**Demo**	**Speedup w.r.t 1 core**	**Speedup w.r.t. sequential**	**Optimal Cores**
$\mathbf{T}$ Scheme	Small ball pile	3.6	7.5	4
$\mathbf{T}$ Scheme	Large ball pile	5.1	11.0	7
$\mathbf{T}$ Scheme	Small ball grid	2.7	2.0	4
$\mathbf{T}$ Scheme	Medium ball grid	3.1	1.8	6
$\mathbf{T}$ Scheme	Large ball grid	3.2	1.5	8

Table 6.2: *The table investigate weak scaling of the* $\mathbf{T}$ *scheme using CSP by looking at the speedup compared to the size of the problem. The fastest time is used and compared to the time usage using one core and the time usage of the sequential Jacobi method.*

This could be explained by the size of $\mathbf{T}$ shown on Table 3.2. Each non-zero block in $\mathbf{T}$ uses 4×4 doubles or 128 byte. For the medium ball grid, $\mathbf{T}$ uses 757,824 non-zero blocks, corresponding to 97 MB, and for the large ball grid, 587 MB are used. Even if spread out on multiple cores, this is significantly more than most CPUs have in their caches. Running the medium ball grid using floats instead of doubles improved the speedup w.r.t. one core to 3.6. For the large ball grid, the speedup changed from 3.2 to 5.2, improving the speedup w.r.t. the sequential Jacobi to 2.6.

6.4 OpenMP

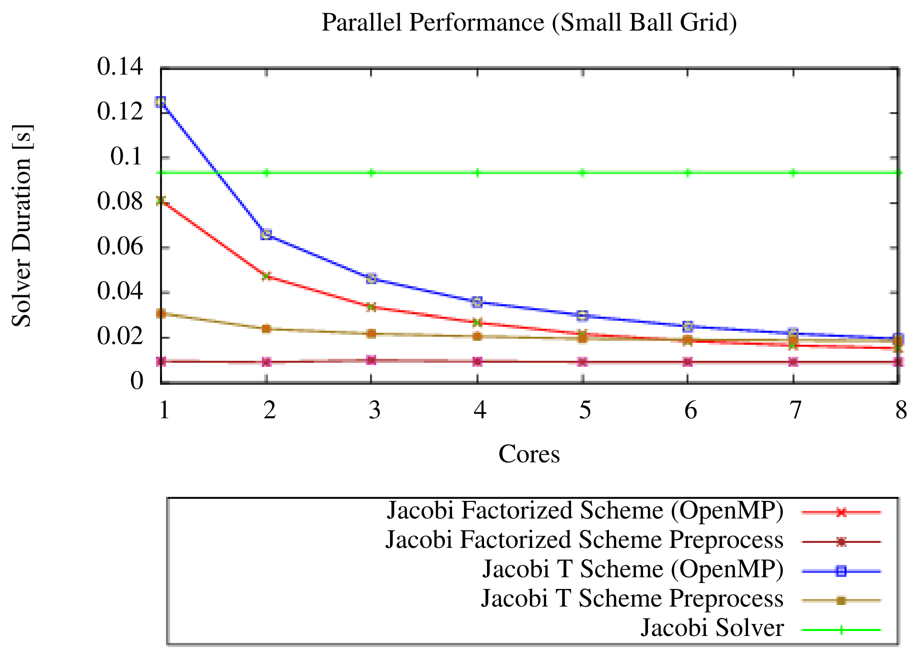
The strong scaling of the CSP-based parallel Jacobi methods is not overwhelmingly good. Because of this, a simple OpenMP implementation of the factorized and the $\mathbf{T}$ schemes has been made. This showed that OpenMP is pretty simple to work with, even though the sequential code may have to be rewritten to expose the parallism. One interesting result is that it is faster to recompute $\vec{w}$ in case of convergence than it is to update it incrementally in the loop. This is likely due to better cache coherency when computing $\vec{w}$ in one go.

Let us investigate the parallel performance of the two solvers. The strong scaling is shown on Figure 6.7 and Figure 6.8.

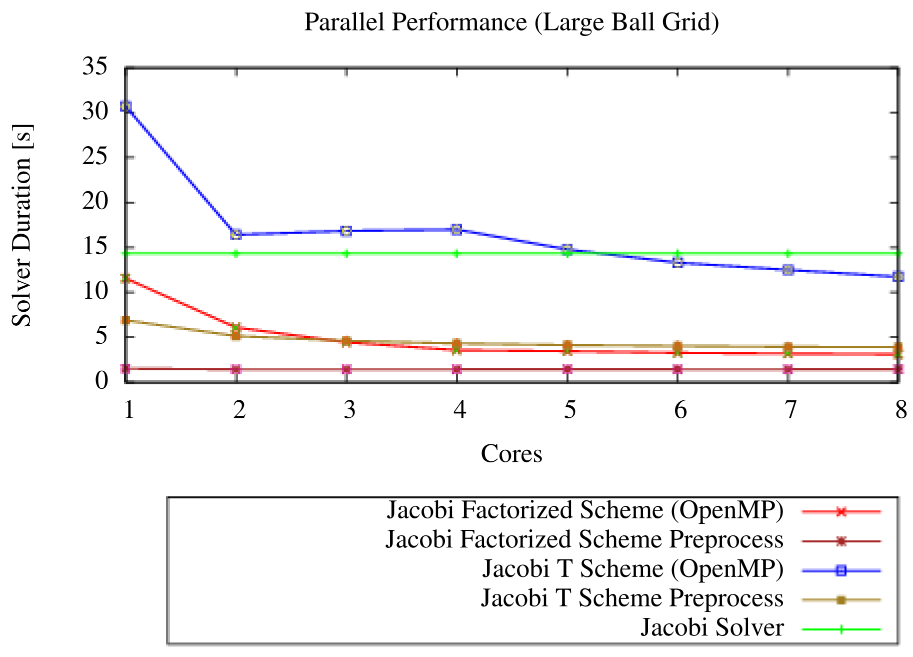
For the structured ball grid demo on Figure 6.7, it is clear that the factorized solver is the fastest. Indeed, on the large ball grid, the $\mathbf{T}$ solver seems not to improve from 2 to 3 and 4 cores. This is likely due to the structure of the subproblems, where one row in $\mathbf{T}$ can be significantly more expensive to handle than the next one. This could be improved by levering the structure of the subproblem. The $\mathbf{T}$ scheme also has trouble with the preprocessing costs.

Both schemes perform well on the unstructured ball grid demos on Figure 6.8. As expected, the $\mathbf{T}$ scheme performs best as there are few dependencies, and, hence, $\mathbf{T}$ is very sparse. The parallel factorized Jacobi scheme is faster than the factorized Jacobi scheme, even when only one core is available. This performance advantage is caused by the parallel factorized Jacobi scheme reuses the previously allocated $\vec{w}$, were as the sequential version reallocates $\vec{w}$ as a temporary variable.

Looking a bit closer at the data shows some interesting results for weak scaling. Table 6.3 shows the speedup of the factorized and $\mathbf{T}$ schemes when using 8 cores. It can be seen that as the problem size increases, the speedup *decrease* in most cases. This is likely due to the cost of memory transfers. Another interesting point is that both schemes have very good scaling for the ball pile demos. In particular, the $\mathbf{T}$ scheme

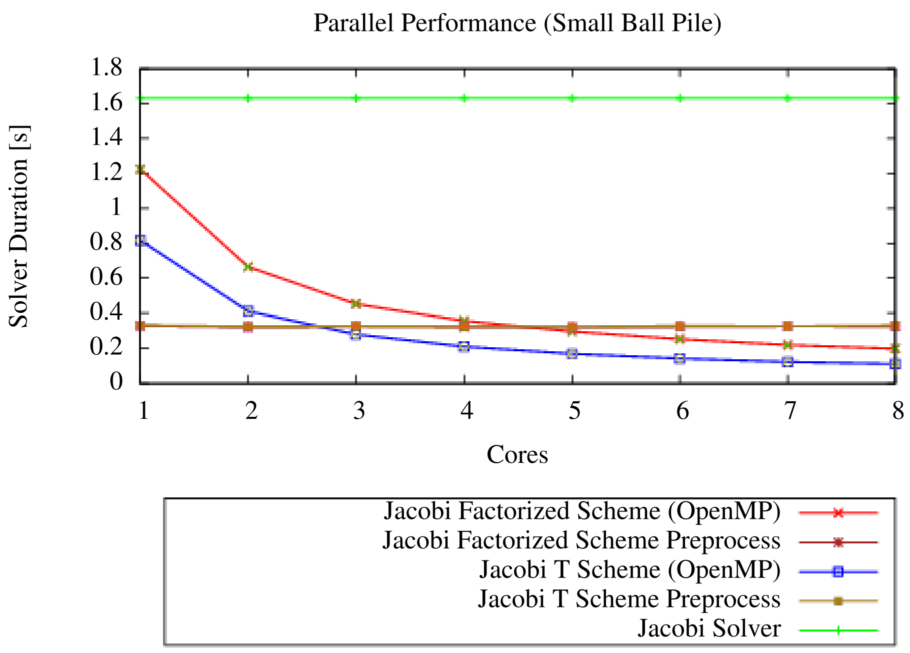


(a)

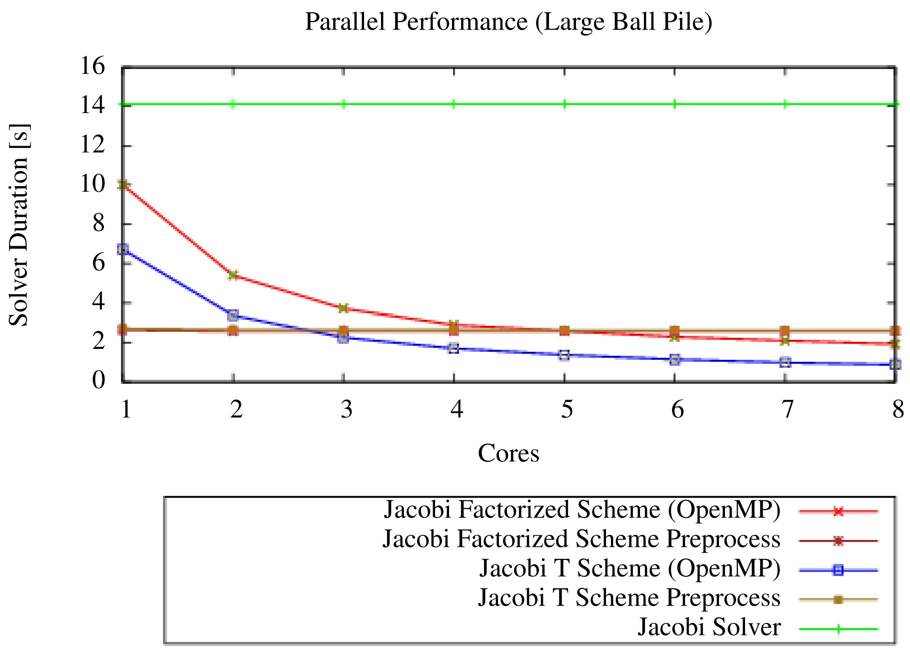


(b)

Figure 6.7: *The curves show how the OpenMP versions of the parallel factorized and* $\mathbf{T}$ *scheme performs on the (a) small and (b) large ball grid demos. On the small ball grid demo, both perform well even though the factorized version clearly is faster and has shorter preprocess time. On the large ball grid demo, the* $\mathbf{T}$ *scheme performs significantly worse, partly due to high preprocess time and unbalanced subproblems.*



(a)



(b)

Figure 6.8: *The curves show how the OpenMP versions of the parallel factorized and* $\mathbf{T}$ *scheme performs on the (a) small and (b) large ball pile demos. Note how both solvers scale well, even if the* $\mathbf{T}$ *scheme clearly is faster than the factoized scheme.*

Method	**Demo**	**Speedup w.r.t 1 core**	**Speedup w.r.t. sequential**
Factorized Scheme	Small ball pile	6.2	8.4
Factorized Scheme	Large ball pile	5.2	7.3
Factorized Scheme	Small ball grid	5.3	6.1
Factorized Scheme	Medium ball grid	6.3	4.9
Factorized Scheme	Large ball grid	3.7	4.6
$\mathbf{T}$ Scheme	Small ball pile	7.3	14.6
$\mathbf{T}$ Scheme	Large ball pile	7.7	16.1
$\mathbf{T}$ Scheme	Small ball grid	6.4	4.7
$\mathbf{T}$ Scheme	Medium ball grid	3.4	2.0
$\mathbf{T}$ Scheme	Large ball grid	2.6	1.2

Table 6.3: *The table shows the speedup of the OpenMP factorized and* $\mathbf{T}$ *schemes using 8 cores, ignoring preprocessing costs. Note how larger problem sizes reduces the speedup, particularly on the structured benchmarks.*

manages a speedup much larger than the number of cores added, as it is better suited for a problem of this type than the factorized scheme. The factorized scheme scales reasonably well on the ball grid benchmarks, were the $\mathbf{T}$ scheme performs horribly on the medium and large ball grid benchmarks.

6.5 Results

Reducing the problem into one task per contact point is very inefficient. Rather, the tasks should be large. This, however, introduces another problem: Scheduling. While it is fully possible to partition the contact problem into tasks of equal size and distribute the tasks over the workers, the time needed to finish a task may vary a little bit, which will cause some workers to wait for other workers. Profiling showed that the time usage of the same task could vary with up to 10% from one iteration to the next.

The results from the OpenMP solvers show that this can be offset by using smaller tasks, but this has the price of increasing communication which may decrease performance for large problems. Switching from task to task may also decrease cache coherency.

Usage of paging to distribute communications improved overall performance, but did not improve scaling for the CSP-based schemes.

A lot of computational power is wasted between iterations. This is the price of using a model that requires each iteration to have a well defined start and end. The cost could be decreased if workers were able to perform some work while waiting. If a graph partitioning were computed, the contact points could be distributed among the workers based on this partitioning, and during waits, the workers could start working on the internal points of the partition. This is reasonable as the internal contact points are independent of contact points handled by other workers. Indeed, such a design could be used to relax the notion of an iteration by allowing the workers themselves to swap results.

Chapter 7

Parallel Gauss–Seidel

In Chapter 6, different parallel designs were investigated for the Jacobi method. One important conclusion was that the classic consumer-producer method comes with a significant overhead. This section will investigate different parallel designs of the Gauss–Seidel schemes proposed in Section 3.4, using the discussion in Section 3.7.

Aside from a naive approach, the work in this section will center on graph coloring. Two coloring strategies will be investigated – a greedy strategy and a balanced strategy. As discussed in Section 4.3, the coloring may be highly unbalanced. Even the balanced strategy could yield a coloring with only few contact points within each color. Unbalanced colorings can be handled by merging them into unsafe colors. The performance of merging the unsafe colors is tested by the relaxed schemes, where the rigid scheme only uses safe colors.

The relaxed schemes are built for correctness, so unlike [TNA08] race conditions are not ignored. Note that some of the CSP optimizations mentioned in Chapter 6 have not been implemented due to an impending deadline.

7.1 Naive Factorized

The naive factorized scheme utilizes a Jacobi-like splitting, where the contact points are distributed evenly among the workers without respect for internal dependencies. To exploit cache coherency, each worker gets a sequential range of contact points. Additionally, each worker gets an index that is passed to the accumulator along with results and a pointer to the local updates of $\vec{w}$. In the loop, a local compressed $\vec{w}$ and the updates of this iteration are maintained. After computing a new $\vec{x}_k$, the updates w_i, w_j are computed and applied to the update vector $\vec{w}_{update}$ and the local variable $\vec{w}_{local}$.

When a worker sends its results to the accumulator, the accumulator applies the $\vec{w}_{update}$ from the worker to its global $\vec{w}_{update}$. When all workers are done, the global $\vec{w}_{update}$ is used to update $\vec{w}$. The updated $\vec{w}$ is copied into the $\vec{w}_{update}$ pointers supplied by the worker, and a status message is returned to the worker. To ensure against the problem mentioned in Section 6.2 where one worker ran multiple times in the same iteration, the accumulator has a channel to each worker. For performance reasons, the channel is buffered with a capacity of one.

When this message is received by the worker, it uses $\vec{w}_{update}$ as the new $\vec{w}_{local}$, clear the old $\vec{w}_{local}$ and use this as $\vec{w}_{update}$. The design is illustrated on Figure 7.1.

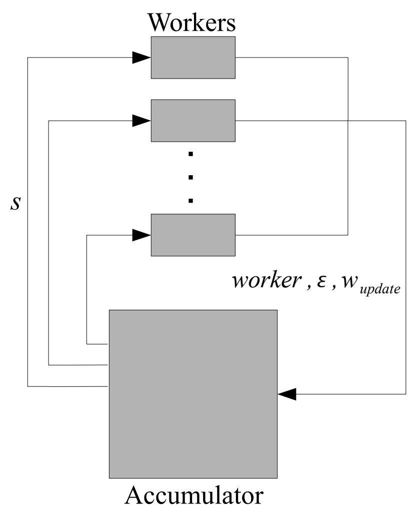


Figure 7.1: *A graphic representation of the design of the naive factorized Gauss–Seidel scheme. The workers are initialized with a portion of the task, and send locally computed updates to $\vec{w}$ to the accumulator. The accumulator investigates global convergence, update $\vec{w}$ and send a status message s to the workers.*

One nice feature of this design is that it is easy to roll back an iteration in case of divergence. If the workers maintain $\vec{w}_{update}$ and $\vec{w}_{local}$ in an array, it can be reduced to flipping an index. Another nice feature is that the updates to $\vec{w}$ can be performed as soon as a worker returns. This reduces the amount of time the workers have to wait at the end of each iteration.

Running benchmarks on the naive factorized scheme shows that as it is split into smaller parts, it mirrors the performance of the Jacobi method as noted by [ABHT03]. For the small ball grid the quality measure with one core is 246, but with 8 cores it is 306 when run for 50 iterations. For the large ball grid, the relative change is pretty small – from 7921 to 7976. An interesting note is that the naive factorized Gauss–Seidel manages to solve the large ball grid in 3.3 seconds, $\times 4$ faster than sequential Gauss–Seidel when using 8 cores.

For largely unstructured problems, the quality measure changes little with the number of cores.

7.2 CSP Coloring Schemes

As noted in Section 4.3, the dependencies of each contact point can be described by a coloring of the contact graph. Contact points of one color are independent and can be updated without fear of race conditions. However, this requires that only the contact points corresponding to one color are considered at any time. It follows that colors will need to be considered sequentially, adding internal synchronizations during the loop. This section will investigate different CSP-based coloring schemes.

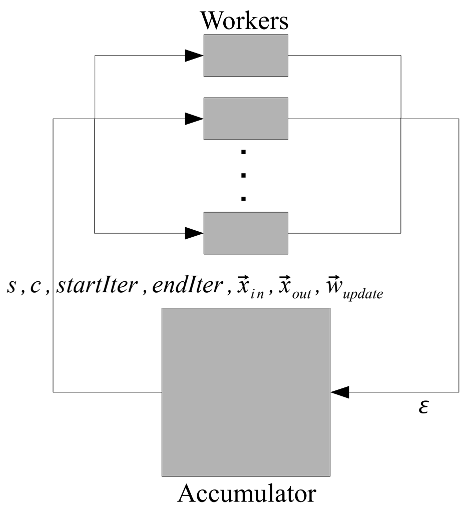

Workers
s, c, startIter, endIter, $\vec{x}_{in}$, $\vec{x}_{out}$, $\vec{w}_{update}$
ε
Accumulator

Figure 7.2: *A graphic representation of the design of the rigid factorized Gauss–Seidel scheme. In each iteration, a number of batches of independent contact points are distributed among the workers so that only one batch is under consideration at one time.*

7.2.1 Rigid Factorized

The rigid factorized design assumes that a graph coloring has been applied to the contact graph, and all contact points in each color are independent of each other. Each iteration is partitioned into batches, where each batch corresponds to one color. The contact points in each color are pre-partitioned among the workers. The design allows each worker to be applied to any part of the contact problem, so this design is closer to the consumer-producer model than the others.

As the contact points sent to each worker in a batch are independent, both $\vec{x}$ and $\vec{w}$ can be updated directly by the worker without dangers of race conditions.

Each worker receives a start and end iterator, specifying which contact points are to be considered in this batch. Note that the contact points are unlikely to have decent cache coherency as the variables are not rebuilt for cache coherency, which is a major performance concern. Apart from the iterators, a status signal and a color is sent to the worker, along with pointers to $\vec{x}_{in}$, $\vec{x}_{out}$ and $\vec{w}$.

The output from each worker is simply a residual that is sent to the accumulator. This allows building of a very light-weight accumulator that is unlikely to become a performance bottleneck. This is important, as one might fear that the many communications needed inside the iteration may result in a lot of overhead due to synchronizations. Another possible problem is that a highly unbalanced coloring may cause the batches to be so small that only one worker can be active during the batch, causing the system to mimic sequential behaviour with parallelization overhead.

7.2.2 Relaxed Factorized

The relaxed factorized design is intended to address the possible problem of very small batches as outlined earlier. This requires using two types of workers – rigid workers and relaxed workers, where the accumulator will use the rigid workers if the color is safe, and the relaxed workers if the color is unsafe.

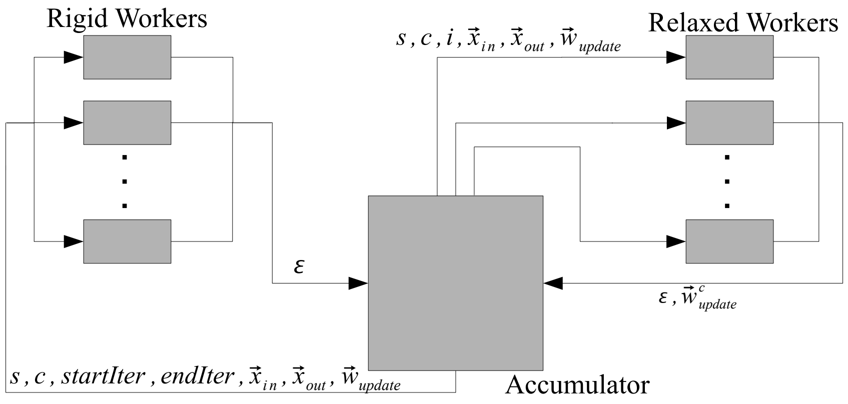

Rigid Workers
Relaxed Workers
$s, c, i, \vec{x}_{in}, \vec{x}_{out}, \vec{w}_{update}$
ε
$\varepsilon, \vec{w}^{c}_{update}$
$s, c, startIter, endIter, \vec{x}_{in}, \vec{x}_{out}, \vec{w}_{update}$
Accumulator

Figure 7.3: *An overview of the relaxed factorized Gauss–Seidel scheme. The rigid workers handle safe colors, and the relaxed workers handle unsafe colors in a correct manner. The accumulator distributes the tasks among the workers so that only one color c is being handled at a time.*

In each iteration, the accumulator investigates the colors in turn. If the color is marked safe, the accumulator distribute the batch with that color between the rigid workers as discussed in Section 7.2.1, reads the results and proceeds. If an unsafe color is encountered, the accumulator sends the status, iteration and color along with $\vec{x}_{in}$, $\vec{x}_{out}$ and $\vec{w}$ to the relaxed workers, and waits for them to finish.

The relaxed workers are assigned a particular part of the subproblem and when they get the signal from the accumulator, they start working on that task. For each unsafe color, the worker maintains a compressed $\vec{w}_{local}$ that contains the locally updated values, and a $\vec{w}_{update}$ that contains the accumulated updates for that color. When it has finished with a color c, $\vec{w}^{c}_{update}$ is sent to the accumulator along with the residual. The design is shown on Figure 7.3.

It is noted that at any given time either the relaxed or rigid workers will be busy. Thus, it makes sense to group the workers so that there is one rigid and one relaxed worker on each processor.

7.2.3 Relaxed T Scheme

The relaxed T Gauss–Seidel scheme has many similarities with the relaxed factorized Gauss–Seidel scheme. The accumulator considers each color in turn, and dispatch safe colors to the rigid worker, and unsafe colors to the relaxed worker.

The rigid worker gets status, iteration, color and worker, along with a pointer to $\vec{x}_{in}$ and $\vec{x}_{out}$. It updates $\vec{x}$ directly, and forward the residual to the accumulator.

The relaxed worker reads the same data from its in-channel, but returns a compressed $\vec{x}^{c}_{update}$ with updated values of $\vec{x}$. It cannot write them directly, as one or more workers could be working on a dependent contact point, which could cause a race condition. Note that the relaxed worker performs Jacobi iteration on the unsafe color.

The accumulator loads the updates into a local structure, and when all relaxed workers are done, the accumulated updates are loaded into $\vec{x}$. The design is shown on Figure 7.4.

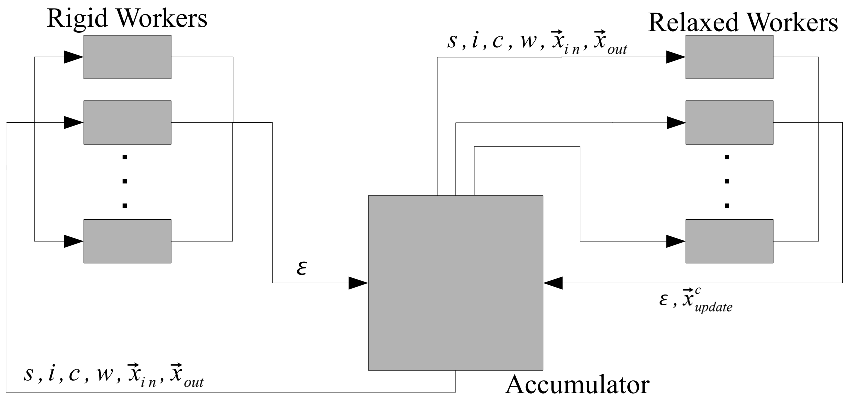

Rigid Workers
Relaxed Workers
$s, i, c, w, \vec{x}_{in}, \vec{x}_{out}$
ε
$\varepsilon, \vec{x}^{c}_{update}$
$s, i, c, w, \vec{x}_{in}, \vec{x}_{out}$
Accumulator

Figure 7.4: *An overview of the relaxed Gauss–Seidel* $\mathbf{T}$ *scheme. The rigid workers handle safe colors, and the relaxed workers handle unsafe colors in a correct manner. The accumulator distribute the tasks among the workers so that one color* c *is being handled at one time.*

7.2.4 Results for Greedy Coloring

The plots on Figure 7.5 and Figure 7.6 illustrate scaling results for the greedy CSP solvers described in the previous sections. Both plots show the time that the solvers ran before a given quality were reached.

The Gauss–Seidel schemes does not perform well on the medium ball grid on Figure 7.5. Depending on the scheme used, 2-3 cores are needed to outperform a sequential Jacobi method. Neither of the schemes scale well beyond the 4th core, even if the relaxed factorized scheme continue to improve slightly up to the 6th core with a speedup of 2.6 compared to using one core. Regardless of the number of cores in use, the relaxed parallel Gauss–Seidel methods needed to run for 57 iterations before reaching the quality of the sequential Gauss–Seidel scheme, most likely due to usage of unsafe colors. The relaxed $\mathbf{T}$ scheme solver needed to run for 65 iterations – almost as many as the sequential Jacobi scheme, most likely because it uses Jacobi iterations on the unsafe colors.

The strong scaling behaviour of the parallel Gauss–Seidel methods are significantly better on the large ball pile benchmark. Even when using one core, they are faster than the sequential Gauss–Seidel. The three schemes scale well up to four cores, and continue to improve until 6-7 cores.

7.3 OpenMP Coloring Schemes

When using OpenMP, it is somewhat more difficult to leverage the structure of the subproblems, which is of particular interest for the coloring schemes. Application of naive Gauss–Seidel comes with a large overhead, and, thus, is not applied to unsafe colors.

For the $\mathbf{T}$ based schemes, the basic strategy is to compute a coloring of a graph, and iterate through the colors. If the color is safe, $\vec{x}_{out}$ can be used and updated directly. This, however, requires that the previous result $\vec{x}_{in}$ is maintained in a separate variable.

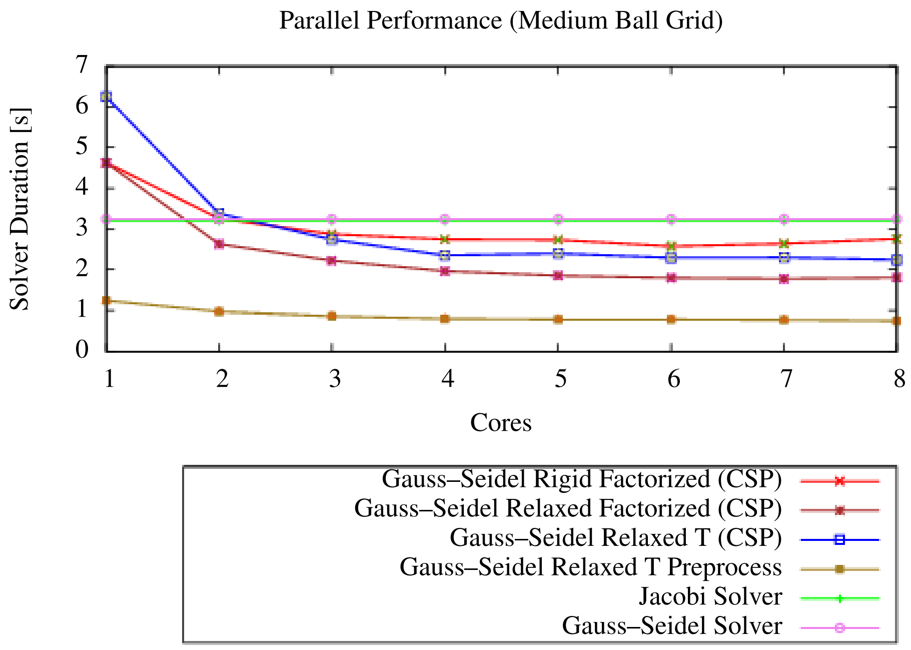


Figure 7.5: *The curves show how different CSP-based greedy coloring schemes perform on the medium ball grid when more cores are added. The solvers have been run to reach the same quality. At two cores, only the relaxed factorized is better than the sequential methods.*

If the color is unsafe, $\vec{x}_{out}$ cannot be updated directly, so the updated values are saved in a vector, and are applied in parallel if the solution converges.

A graph coloring is computed for the factorized schemes as well, and the schemes iterate through the colors. If a color is safe, $\vec{w}$ is used and updated directly. If the color is unsafe, only $\vec{x}_{out}$ is updated. If the solution converges, $\vec{x}_{out}$ is used to update $\vec{w}$, exploiting that we know which contact points are unsafe. In case of divergence, a new $\vec{w}$ is computed.

For the medium ball grid, the greedy coloring constructs a 7-coloring, where the 6 safe colors contain 6912 – 5329 contact points and the unsafe color contains 570 contact points. The balanced coloring constructs a 580 coloring where each color is safe and contains 69 or 70 colors.

Running the colored OpenMP solvers on the medium ball grid yielded the results shown on Figure 7.7. As expected, the factorized solvers handle this problem much better than the $\mathbf{T}$ solvers. Nor is it surprising that the greedy solvers perform better than the balanced solvers on this problem when many cores are used – the cost of the many synchronizations in the loop simply eat the effect of extra cores.

The strong scaling of the colored OpenMP solvers on the medium ball grid is not impressive – a $\times 3$ speedup compared to their 1 core time usage and a maximum $\times 2$ improvement compared to sequential Gauss–Seidel.

The results for the large ball pile benchmark are shown on Figure 7.8. The Greedy Coloring constructs three colors with 50125, 2781 and 3 contact points, respectively. These are merged into two colors with 50125 and 2784 contact points, where the last

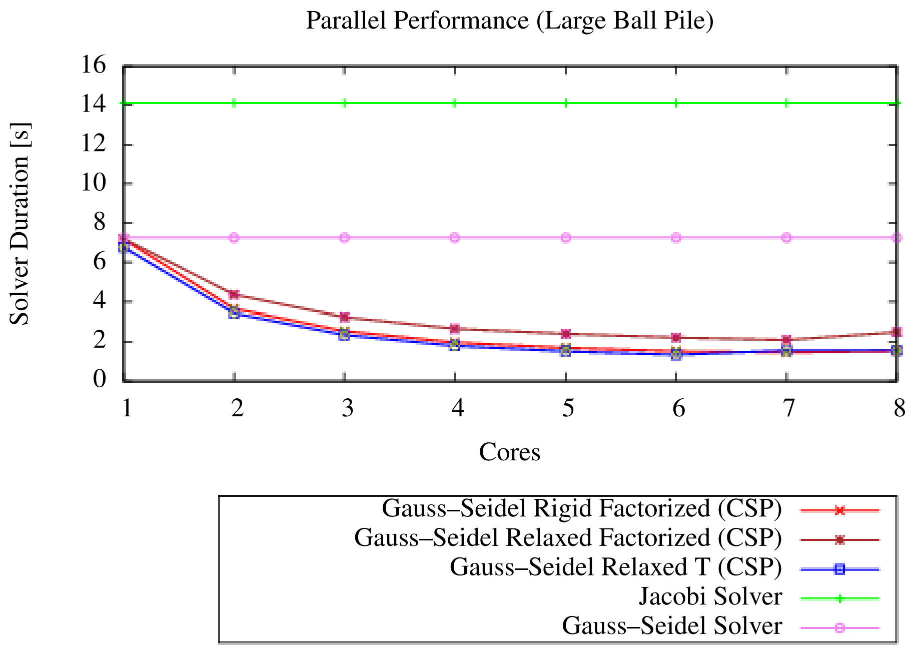


Figure 7.6: *The curves show how different CSP-based greedy coloring schemes perform on the large ball pile when more cores are added, targeting the same quality.*

color is unsafe. The Balanced Coloring produce three safe colors, with 17637, 17636 and 17636 contact points each and no unsafe colors.

As such, one would expect the balanced coloring to produce slightly better results per iteration, but have a slightly higher synchronization cost. Indeed, the balanced colorings finish in 50 iterations, where the greedy colorings need 51 iterations.

The **T** schemes perform almost identically regardless of coloring scheme used, with the greedy scheme being slightly faster than the balanced scheme. For the factorized scheme, the balanced coloring performs significantly better than the greedy scheme.

Another notable feature is that adding extra cores decrease the solver duration for all the OpenMP schemes. On 8 cores, the greedy **T** scheme solves the problem 8.4 times faster than the sequential Gauss–Seidel, and 6.6 times faster than the greedy **T** scheme performs on one core.

7.4 Performance of the Coloring Schemes

The usage of balanced solvers compared with greedy solvers is shown on Figure 7.7. At first glance, they seem similar, even if the relaxed greedy solvers are somewhat faster than their balanced counterparts. Under the hood, however, it is noted that the balanced solvers ran for 49 iterations and managed to get results similar to what the relaxed greedy solvers got in 57 iterations.

Thus, there is a clear per iteration cost of using many colors. This suggests that the need for synchronizations inside the loop cause a serious deterioration of strong

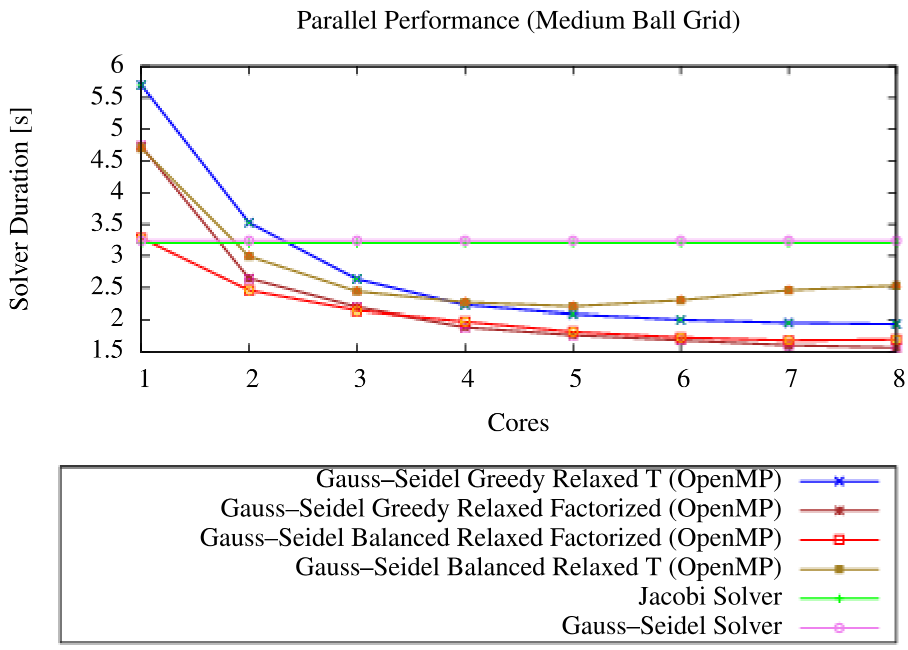


Figure 7.7: *The curves show how different OpenMP-based coloring schemes perform on the medium ball grid when more cores are added, targeting the same quality. Note that the bottom axis is* 1.5 *not* 0*.*

per iteration scalability compared to the Jacobi schemes. Thus, a coloring scheme is ill suited for an environment where communication costs are significant, such as a distributed system.

Improving the handling of unsafe colors could improve the performance of the relaxed schemes. The unsafe colors in the ball grid benchmarks primarily contain the contact points between the ground and the bottom spheres. In general, one would expect the contact points in an unsafe color to be dependent on a lot of other contact points. This suggests that these contact points may be the contact points that are slowest to converge. In the ball grid benchmark, each iteration will apply the weight of another layer of spheres to the bottom layer – where most of the unsafe contact points reside. Some testing suggests that the contact points in the unsafe colors converge faster if they are handled last in an iteration.

One possible improvement in the handling of unsafe colors is to compute the interdependency of the unsafe contact points and group them so that dependent contact points are placed on the same relaxed worker running naive Gauss–Seidel iterations.

The results in Figure 7.8 suggests that there is an advantage to grouping contact points in few colors of similar size, rather than one big color and a few smaller safe or unsafe colors.

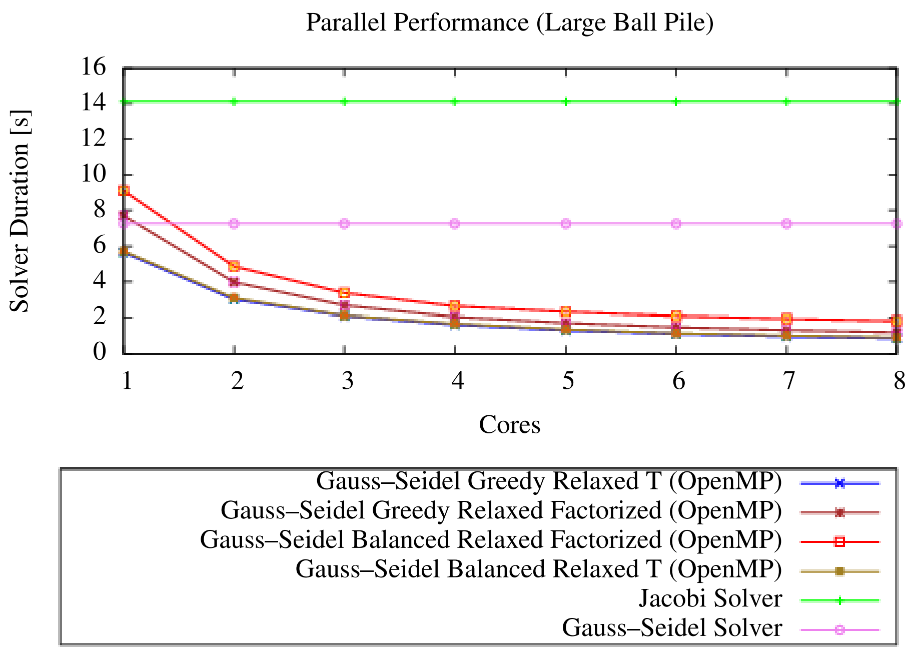


Figure 7.8: *The curves show how different OpenMP-based coloring schemes perform on the large ball pile when more cores are added, targeting the same quality.*

7.5 Parallel Gauss–Seidel or Parallel Jacobi?

If we take a look at the large ball pile benchmark, the best Gauss–Seidel scheme is the OpenMP-based greedy relaxed $\mathbf{T}$ scheme shown on Figure 7.8, which manages a speedup of $\times 8.4$ compared with the sequential Gauss–Seidel. The best parallel Jacobi method is the $\mathbf{T}$ scheme shown on Figure 6.8. The schemes are compared on Figure 7.9 for a common target quality.

The fastest Gauss–Seidel scheme is a bit faster than the fastest Jacobi scheme, particularly for a small number of cores, but the difference decreases as more cores are added.

Note, however, that the Gauss–Seidel schemes perform significantly fewer iterations – 50 iterations compared to the 65 iterations performed by the Jacobi method. Thus, if the per iteration cost of the Gauss–Seidel schemes were improved a clear difference would be visible.

As can be seen on Figure 7.5 and Figure 7.7 neither of the Gauss–Seidel based schemes perform well on the medium ball grid benchmark, whereas the Jacobi schemes perform reasonably well, as shown on Figure 6.7. These benchmark results are collected on Figure 7.10. As can be seen, the parallel Jacobi solver is significantly faster than the Gauss–Seidel solvers. However, the factorized Jacobi scheme uses 66 iterations and the Gauss–Seidel schemes use 57 iterations for greedy colorings and 50 iterations for balanced colorings.

The benchmarks clearly show that parallel Jacobi schemes are competitive with parallel Gauss–Seidel schemes. Naturally, one must note that the number of benchmarks is small, and consists of very limited subset of the real world problems that a

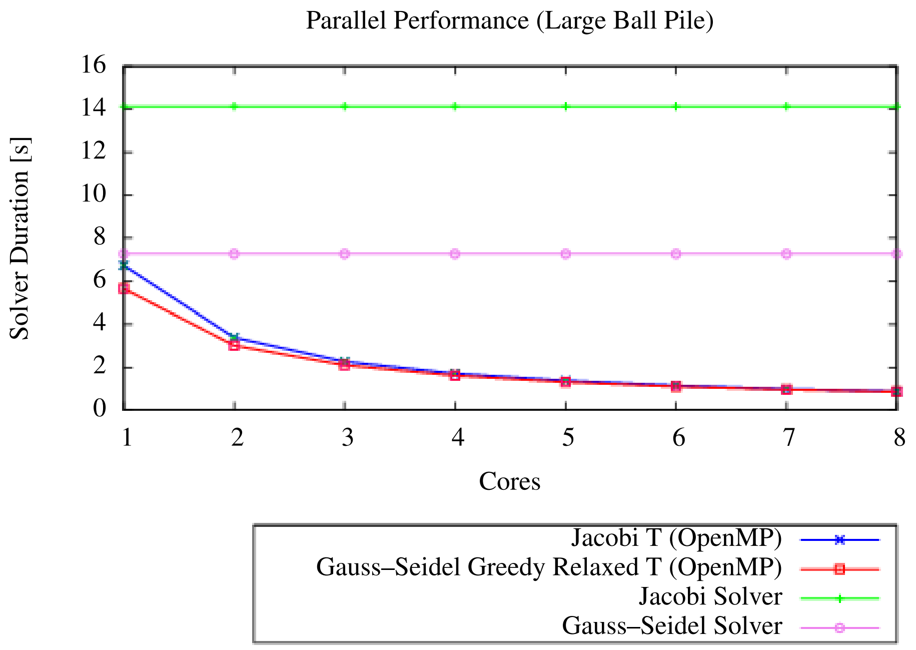


Figure 7.9: *The curves show how different parallel solvers perform on the large ball pile when more cores are added, targeting the same quality*

physics solver could encounter, and that there are a number of optimizations for the Gauss–Seidel schemes that can improve their performance.

7.6 Comparison with Other Solvers

As earlier noted, [CCH$^+$07] likely used unstructured benchmarks for their graph coloring scheme. With 8 cores their parallel LCP solver managed to get a speedup of approximately $\times 6.3$, compared to a sequential Gauss–Seidel implementation. This is somewhat similar to the $\times 6.6$ speedup the $\mathbf{T}$ scheme managed on 8 cores compared to its 1-core performance.

[IR09a] does not supply 1-core or sequential benchmark results, but from 4 to 8 cores their graph partitioning scheme manage a $\times 2.3$ speedup. The closest benchmark in our collection is the small ball pile, where the greedy relaxed Gauss–Seidel $\mathbf{T}$ scheme mange a $\times 1.9$ speedup. I would have liked to run my benchmarks on their solver, but the *pe* engine does not seem to be available online.

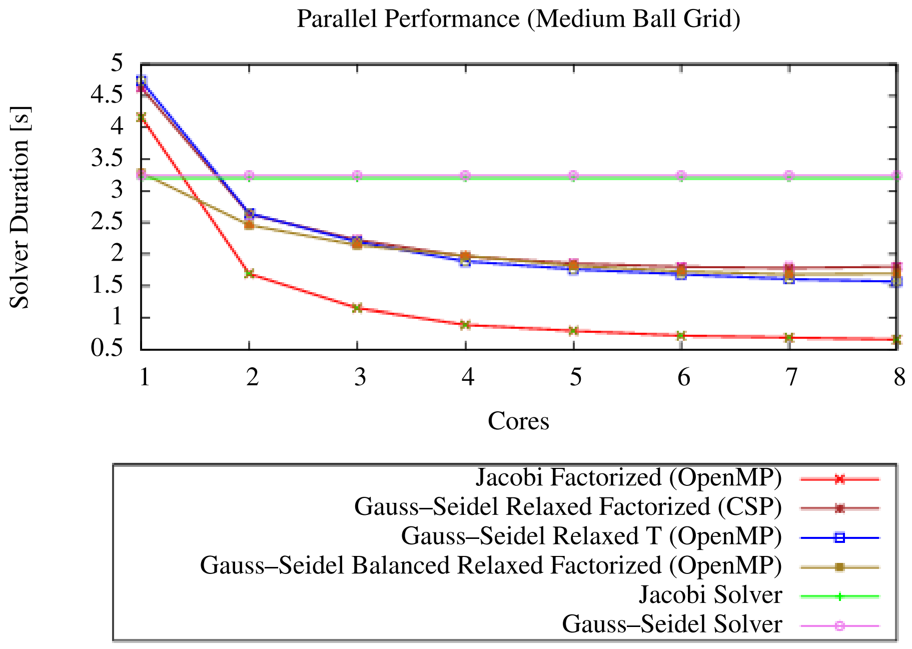


Figure 7.10: *The curves show how different parallel solvers perform on the medium ball grid when more cores are added, targeting the same quality. Note that the bottom axis is* 0.5 *and not* 0.

Chapter 8

Conclusion

In Chapter 2, the contact problem was introduced and a mathematical model was derived. Chapter 3 developed a number of numerical schemes to solve the contact problem, and considered issues of a parallel version of the numerical schemes. This led to an investigation of the contact graph in Chapter 4, and in Chapter 7 it was demonstrated that the contact graph can be used to detect dependencies between contact points.

Chapter 4 suggested that one could use either graph partitioning or graph coloring to split the contact problem over multiple CPUs, but due to time constraints only graph coloring has been investigated in depth.

8.1 Graph Coloring

For graph coloring, we noted that the number of synchronizations was increased, which suggests that a small number of colors is preferred. Many colors with a small number of contact points leads to a deterioration of the per iteration parallel scaling, which could be alleviated by merging the small colors into a larger, unsafe color.

An interesting point is that if the number of colors is low, it is an advantage to use a balanced coloring strategy, whereas if the number of colors is high, a greedy strategy where small colors are merged is better. A small number of colors of approximately the same size allows the task of updating contact points in a given color to be partitioned into larger batches. These larger batches can be distributed among the available CPUs using dynamic scheduling. This reduces the likely overhead in terms of waits, where one CPU could finish before another, which is consistent with the results from Section 6.5.

Using benchmarks, it was demonstrated that construction of a parallel scheme using graph coloring conserved the Gauss–Seidel properties. However, most CSP-based coloring schemes did not improve much beyond the 4th CPU. The OpenMP implementations fared better in some cases, but the better scaling properties and lower per iteration cost of the Jacobi-based schemes made them more and more advantageous. Indeed, the Jacobi schemes consistently outperformed the Gauss–Seidel schemes on structured benchmarks, and were competitive on unstructured benchmarks.

It would seem that the embarrassingly parallel Jacobi is better suited for parallel solvers. I am, however, not ready to call it just yet. The number of benchmarks used is very small, and the configurations cannot be said to present a reasonable subset of the configurations that a physics solver may encounter. Additionally, there are a number of optimizations of the Gauss–Seidel schemes that are likely to reduce the per iteration cost. Furthermore, graph partitioning has not been investigated in depth.

8.2 Graph Partitioning

One of the weak points of the coloring schemes is the potentially large number of synchronizations. Partitioning the contact graph into parts, where each CPU handle one or more parts, and only communicates between iterations could decrease the number of synchronizations considerably.

While waiting for updates on border nodes from its neighbours, each CPU could proceed by working on inner nodes, thereby reducing waiting times in the system. As shown by [IR09a], an efficient graph partitioning would need to minimize the number of neighbours of each CPU. However, a reasonable distribution of workload is also needed, and minimizing the edge cut can improve performance further.

Most of the variables used by the graph partitioned solver can be computed locally, which has the advantage of further parallelization as well as improved cache coherency. The drawback is that the contact graph will need explicit representation.

A graph partitioning scheme can also be used to distribute the contact points over nodes, where each node is a collection of processing units. Consider the problem of utilizing multiple GPUs for a parallel solver. Each CPU can only control 4 GPUs, and communication between GPUs has to go over the CPU. So, a graph partitioning scheme can be used to distribute the task over a system of CPUs, who can divide their subtasks between the available GPUs. Distribution over GPUs could use the method of [TNA08] or one could investigate usage of a local coloring scheme.

Such a design is, however, mostly interesting for off-line solvers, as data transmission times are likely to be high.

8.3 Contact Sets

Let a contact set be the set of contact points between two bodies. The contact set can be seen as an approximation of the contact plane or even a contact manifold between the two bodies. Some of the analysis and results suggest that using contact sets as the atomic unit may be more efficient than using contact points. This would simplify the contact graph, reduce the number of colors needed for a graph coloring and allow us to use a better suited method to solve the contact problem for the contact plane in question.

Usage of contact sets, however, comes at a cost. The blocked matrix library may have to be customized to be convenient with such a structure, and scheduling becomes somewhat more complicated.

8.4 Deformable Objects

The rigid body idealization is utilized to be able to ignore the internal structure of an object, but it has another advantage. Consider two boxes in a stack, where the top box is rotated along the up direction. This simple configuration will use 8 contact points. For a contact between two arbitrary meshes, the number of contact points is likely to be very large. However, for rigid bodies it is possible to add simpler bounding volumes, as the bodies do not deform.

For deformable objects it is not that simple. The position of each contact influence the shape of the object and thereby its movement. Deformable bodies have one advantage over rigid bodies: Updating one contact point does not necessarily spread to all contact points of the bodies in question. A little lag is acceptable on physical grounds, as applying an impulse at one point of an object does not spread instantaneously to all other points.

This insight, noted by [LSH10], can be leveraged to make a very simple and very efficient parallel scheme. A graph partitioning could be computed by partitioning the world into disjoint parts using cuts that maximize the number of internal contact points.

Bibliography

[ABC$^+$95] J. R. Allwright, R. Bordawekar, P. D. Coddington, K. Dincer, and C. L. Martin. A Comparison of Parallel Graph Coloring Algorithms, 1995.

[ABHT03] Mark Adams, Marian Brezina, Jonathan Hu, and Ray Tuminaro. Parallel Multigrid Smoothing: Polynomial versus Gauss-Seidel. *Journal of Computational Physics*, 188:593–610, 2003.

[Ani06] Mihai Anitescu. Optimization-based Simulation of Nonsmooth Rigid Multibody Dynamics. *Mathematical Programming*, 105:113–143, 2006.

[ARK05] A. Abou-Rjeili and G. Karypis. Multilevel Algorithms for Partitioning Power-Law Graphs. Technical report, Department of Computer Science and Engineering, University of Minnesota, October 2005.

[Bar94] David Baraff. Fast Contact Force Computation for Nonpenetrating Rigid Bodies. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 23–34, New York, NY, USA, 1994. ACM.

[BBC$^+$05] Erik G. Boman, Doruk Bozdag, Umit Catalyurek, Assefaw H. Gebremedhin, and Fredrik Manne. A Scalable Parallel Graph Coloring Algorithm for Distributed Memory Computers. In *in Proceedings of Euro-Par 2005 Parallel Processing*, pages 241–251. Springer, 2005.

[BCG$^+$05] Doruk Bozdag, Umit Catalyurek, Assefaw H. Gebremedhin, Fredrik Manne, Erik G. Boman, and Füsun Özgüner. A Parallel Distance-2 Graph Coloring Algorithm for Distributed Memory Computers. In *High Performance Computation Conference (HPCC), Lecture Notes in Computer Science*, pages 796–806. Springer-Verlag Berlin Heidelberg, 2005.

[BF97] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. Books/Cole Publishing Company, 1997.

[BVA07] John Markus Bjørndalen, Brian Vinter, and Otto J. Anshus. PyCSP – Communicating Sequential Processes for Python. In *Communicating Process Architectures 2007 : WoTUG-30, Proceedings of the 30st WoTUG Technical Meeting (CPA-07)*, volume 65 of *Concurrent Systems Engineering Series*, pages 229–248. IOS Press, 2007.

[C++] C++CSP2. http://www.cs.kent.ac.uk/projects/ofa/c++csp/.

[CA09] Hadrien Courtecuisse and Jérémie Allard. Parallel Dense Gauss-Seidel Algorithm on Many-Core Processors. In *High Performance Computation Conference (HPCC)*. IEEE CS Press, jun 2009.

[CCH$^+$07] Yen-Kuang Chen, Jatin Chhugani, Christopher J. Hughes, Daehyun Kim, Sanjeev Kumar, Victor Lee, Albert Lin, Anthony D. Nguyen, Eftychios Sifakis, and Mikhail Smelyanskiy. High-Performance Physical Simulations on Next–Generation Architecture with Many Cores. *Intel Technology Journal*, 11:251–262, 2007.

[Cha99] Anindya Chatterjee. On the Realism of Complementarity Conditions in Rigid Body Collisions. *Nonlinear Dynamics*, 20:159–168, 1999.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[Cou10a] Erwin Coumans. Bullet physics, http://bulletphysics.org/wordpress/, 2010.

[Cou10b] Erwin Coumans. *Bullet Users Manual – Bullet 2.76 Physics SDK Manual*. 2010.

[CPS09] Richard Cottle, Jong-Shi Pang, and Richard E. Stone. *The Linear Complementarity Problem*. Philadelphia: Society for Industrial and Applied Mathematics, 2009.

[CSP09] CSPBuilder. http://www.migrid.org/vgrid/CSPBuilder/. 2009.

[DeL09] Mark DeLoura. Middleware Showdown. *Game Developer Magazine*, August 2009.

[DHK$^+$00] C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss. Cache Optimization for Structured and Unstructured Grid Multigrid. *Electronic Transaction on Numerical Analysis, 10:21–40, 2000*, 10:21–40, 2000.

[Erl04] Kenny Erleben. *Stable, Robust, and Versatile Multibody Dynamics Animation*. PhD thesis, University of Copenhagen, April 2004.

[Erl07] Kenny Erleben. Velocity-Based Shock Propagation for Multibody Dynamics Animation. *ACM Transactions on Graphics, Vol. 26, No. 2, Article 12*, 2007.

[ESHD05] Kenny Erleben, Jon Sporring, Knud Henriksen, and Henrik Dohlmann. *Physics-Based Animation*. Charles River Media, 2005.

[FGNU06] Martin Förg, Thomas Geier, Lutz Neumann, and Heinz Ulbrich. R-Factor Strategies for the Augmented Lagrangian Approach in Multi-body Contact Mechanics. In *III European Conference on Computational Mechanics, Solids, Structures and Coupled Problems in Engineering, Lisbon, Portugal*. Springer Netherlands, June 2006.

[For06] Physics Simulation Forum.
http://www.bulletphysics.org/Bullet/phpBB3/viewtopic.php?p=&f=4&t=534.
2006.

[FW08] Rune Møllegaard Friborg and Brian Winter. CSPBuilder – CSP based Scientific Workflow Modelling. In *Communicating Process Architectures 2008 : WoTUG-31, Proceedings of the 31st WoTUG Technical Meeting (CPA-08)*, volume 66 of *Concurrent Systems Engineering*, pages 347–362. IOSPress, 2008.

[GBP$^+$07] Amol Ghoting, Gregory Buehrer, Srinivasan Parthasarathy, Daehyun Kim, Anthony Nguyen, Yen-Kuang Chen, and Pradeep Dubey. Cache-conscious Frequent Pattern Mining on Modern and Emerging Processors. *The VLDB Journal*, 16:77–96, January 2007.

[GJK88] E.G. Gilbert, D.W. Johnson, and S.S. Keerthi. A Fast Procedure for Computing the Distance Between Complex Objects in Three-dimensional Space. *IEEE Journal of Robotics and Automation, Volume: 4, Issue: 2, April 1988*, pages 193–203, 1988.

[GM00] Assefaw Hadish Gebremedhin and Fredrik Manne. Scalable Parallel Graph Coloring Algorithms, 2000.

[GRP89] Suresh Goyal, Andy Ruina, and Jim Papadopoulos. Limit Surface and Moment Function Descriptions of Planar Sliding. *Proceedings of IEEE International Conference on Robotics and Automation, Volume: II*, pages 794–800, 1989.

[HGS$^+$07] Christopher J. Hughes, Radek Grzeszczuk, Eftychios Sifakis, Daehyun Kim, Sanjeev Kumar, Andrew P. Selle, Jatin Chhugani, Matthew Holliman, and Yen kuang Chen. Physical Simulation for Animation and Visual Effects: Parallelization and Characterization for Chip Multiprocessors. In *SIGARCH Comput. Archit. News*, 2007.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. *Communnications of the ACM*, 21(8):666–677, 1978.

[Hoa04] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 2004. Available online at http://www.usingcsp.com/.

[IR09a] Klaus Iglberger and Ulrich Rüde. Massively Parallel Rigid Body Dynamics Simulations. *Computer Science - Research and Development*, 23:159–167, June 2009.

[IR09b] Klaus Iglberger and Ulrich Rüde. The pe Rigid Multi-Body Physics Engine. Technical report, Friedrich-Alexander-Universiät Aterlangen-Nürnberg Institute Für Informatic(Matematische Maschinen und Datenverarbeitung), May 2009.

[JCS] JCSP. http://www.cs.ukc.ac.uk/projects/ofa/jcsp/.

[Jib] Jibu. http://www.axon7.com.

[KH02] J. M. Knudsen and P. G. Hjorth. *Elements of Newtonian Mechanics*. Springer, 2002.

[KHN07] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: Architectural Support for Fine-grained Parallelism on Chip Multiprocessors. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 162–173, New York, NY, USA, 2007. ACM.

[KK98] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[KRo] KRoC. http://www.cs.kent.ac.uk/projects/ofa/kroc/.

[KSJP08] Danny M. Kaufman, Shinjiro Sueda, Doug L. James, and Dinesh K. Pai. Staggered Projections for Frictional Contact in Multibody Systems. *ACM Trans. Graph.*, 27(5):1–11, 2008.

[LC94] Gary Lewandowski and Anne Condon. Experiments with Parallel Graph Coloring Heuristics. In *Johnson & Trick*, pages 309–334. American Mathematical Society, 1994.

[LG03] R.I. Leine and Ch. Glocker. A Set-valued Force Law for Spatial Coulomb-Contensou Friction. *European Journal of Mechanics and Solids*, 22:193–216, 2003.

[LM99] Martin Larsen and Per Madsen. A Scalable Parallel Gauss-Seidel and Jacobi Solver for Animal Genetics. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 356–363, London, UK, 1999. Springer-Verlag.

[LSH10] Olexiy Lazarevych, Gabor Szekely, and Matthias Harders. Decomposing the Contact Linear Complementarity Problem into Separate Contact Regions. In *18th International Conference on Computer Graphics, Visualization and Computer Vision, WSCG 2010*. University of West Bohemia, Czech Republic, 2010.

[Mes94] Robert Messer. *Linear Algebra: Gateway to Mathematics*. HarperCollins College Publishers, 1994.

[MOIS05] T. Mifune, N. Obata, T. Iwashita, and M. Shimasaki. A Parallel Algebraic Multigrid Preconditioner Using Algebraic Multicolor Ordering for Magnetic Finite Element Analyses. In *Parallel Computing 2005: Current and Future Issues of High-End Computing, NICSeries*, volume 33, pages 237–244, 2005.

[Mor65] J.J. Moreau. Proximité et dualité dans un espace Hilbertien. *Bulletin de la Société Mathématique de France 93*, pages 273–299, 1965.

[MW88] Matthew Moore and Jane Wilhelms. Collision Detection and Response for Computer Animation. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 289–298, New York, NY, USA, 1988. ACM.

[PNE10] Morten Poulsen, Sarah Niebe, and Kenny Erleben. Heuristic Convergence Rate Improvements of the Projected Gauss-Seidel Method for Frictional Contact Problems. In *18th International Conference on Computer Graphics, Visualization and Computer Vision, WSCG 2010*. University of West Bohemia, Czech Republic, 2010.

[PyC] PyCSP. http://www.cs.uit.no/∼johnm/code/PyCSP/.

[Ras09] Jens Rasmussen. Blocked Sparse Matrix Library. Master's thesis, Department of Computer Science, University of Copenhagen, June 2009.

[RDA04] Mathieu Renouf, Frédéric Dubois, and Pierre Alart. A Parallel Version of the Non Smooth Contact Dynamics Algorithm Applied to the Simulation of Granular Media. *Journal of Compututational and Applied Mathematics*, 168(1-2):375–382, 2004.

[SCF01] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. Rescheduling for Locality in Sparse Matrix Computations. In *ICCS 2001, Lecture Notes in Computer Science*, pages 137–146. Springer-Verlag Berlin Heidelberg, 2001.

[SCF$^+$05] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, Jonathan Freeman, and Barbara Kreaseck. Combining Performance Aspects of Irregular Gauss-Seidel Via Sparse Tiling. In *Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science*, pages 90–110. Springer-Verlag Berlin Heidelberg, 2005.

[SGG04] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts with Java*. John Wiley & Sons, Inc., 6th edition, 2004.

[SNE09] Morten Silcowitz, Sarah M. Niebe, and Kenny Erleben. Nonsmooth Newton Method for Fischer Function Reformulation of Contact Force Problems for Interactive Rigid Body Simulation. In *Virtual Reality Interactions and Physical Simulations (VRIPhys)*, 2009.

[ST96] David Stewart and J. C. Trinkle. An Implicit Time-Stepping Scheme for Rigid Body Dynamics with Coulomb Friction. *International Journal of Numerical Methods in Engineering*, 39:2673–2691, 1996.

[TD07] Rohallah Tavakoli and Parviz Davami. A New Parallel Gauss-Seidel Method Based on Alternating Group Explicit Method and Domain Decomposition Method. *Applied Mathematics and Computation*, 188:713719, 2007.

[TNA08] A. Tasora, D. Negrut, and M. Anitescu. Large–scale Parallel Multi-body Dynamics with Frictional Contact on the Graphical Processing Unit. In *Proceedings of the Institution of Mechanical Engineers, Part K: Journal of Multi-body Dynamics*, pages 315–326. Professional Engineering Publishing, 2008.

[TNA$^+$10] Alessandro Tasora, Dan Negrut, Mihai Anitescu, Hammad Mazhar, and Toby David Heyn. Simulation of Massive Multibody Systems using GPU Parallel Computation. In *18th International Conference on Computer Graphics, Visualization and Computer Vision, WSCG 2010*, pages 1–1. University of West Bohemia, Czech Republic, 2010.

[Tsu09] Jumpei Tsuda. Practical Rigid Body Physics for Games. In *SIGGRAPH ASIA '09: ACM SIGGRAPH ASIA 2009 Courses*, pages 1–83, New York, NY, USA, 2009. ACM.

Appendix A
Thesis Outline

The following pages contain the original thesis outline.

A Parallel Gauss–Seidel Method for Computing Contact Forces

Supervisor: Kenny Erleben
Student: Morten Poulsen

Description

In the field of physics based animation of rigid bodies,a common problem is to compute contact forces. Contact forces are used to ensure against penetrations between the bodies in the scene, as well as model friction and collisions. While the physics is reasonably well understood, building a mathematical and numerical model that is fast, robust and precise has proved to be a challenge.

Advances in hardware in the last decade has made multi-core machines affordable for the public, and the current tendencies imply that a significant portion of the hardware improvements in the future will be based on adding more cores. This puts the methods of the graphics community under some stress, as many of the methods are sequential in nature, and were constructed assuming that they were to be run sequentially. To which degree can these methods be restated in concurrent contexts? Will simpler methods be better suited for concurrency?

One of these sequential methods is the iterative Gauss–Seidel method, usually used as a matrix solver. The Gauss–Seidel method is commonly used to solve the contact force problem. This thesis will investigate how methods inspired by the iterative matrix solver Gauss–Seidel can be parallelized. While parallel versions already exist, there seems to be little in the literature on how these versions perform when used on the contact force problem. One may also expect that our knowledge of the geometry and physics of the problem could be used to devise a clever and fast scheme when formulating a parallel version.

Project Goals

The goals of this project are to:

1. Identify and analyze problems when parallelizing Gauss–Seidel based methods, such as interaction between batches, or the problem of visualization when using many CPUs.

2. Investigate if Graph Coloring of the contact graph can be used to devise a fast scheme.

3. Select a solution to implement, noting side effects.

4. Implement the selected solution.

5. Compare the selected solution with solutions from other frameworks, such as BULLET Physics Library. As other frameworks may use a different contact model, the comparison will focus on the Gauss–Seidel part of the solution. This comparison could be algorithmic, measurement of load balance of Gauss–Seidel or something similar.

6. Evaluate the differences w.r.t other solutions.

Milestones

- 1 month: Literature study done.
- 3 months: Basic prototype working.
- 6 months: Hand in thesis.

Risk Assessment

To accurately evaluate the performance of the implemented method, a rigid body physics engine will be needed. Implementation of an entire rigid body physics engine is beyond the scope of this thesis. Instead, the PROX framework will be expanded to facilitate implementation of a parallel Gauss–Seidel scheme.

The core parts of the PROX framework lacks the following features:

1. Gauss-Seidel prox solver. This solver will be implemented as a part of this thesis, and a parallel version will be added.
2. The Blocked Matrix Library (BML) lacks some features needed for an efficient implementation of a Gauss–Seidel scheme. These features will be implemented as a part of this thesis.
3. The broadphase collision detector currently uses an all-pairs search, which limits our ability to handle large problems.
4. Only one contact point is generated per body touching each other. This limits our engine to handle geometries that can be represented by spheres.

The first two points will be implemented as a part of this thesis, the two latter points may be implemented if enough time is available, and no one else has taken ownership of it.

The PROX framework also lacks some other useful features, such as a visualizer to visualize a scene, the ability to load a scene specified externally, and the ability to output profiling data. It should be noted that functionality to output data to Matlab scripts do exist.