Master Thesis

Parallelization of the Webots simulation loop & the ODE physics engine

Abd-ur-Rehman Mustafa

Cyberbotics Switzerland
Biorobotics Laboratory (Biorob Lab)
École Polytechnique Fédérale de Lausanne

Supervisors
Dr. Olivier Michel
Prof. Auke Jan Ijspeert
Abstract

In this project, we investigate the different ways of parallelizing the simulation loop of Webots, a popular robotic simulation software and the underlying physics engine it uses, the Open Dynamics Engine (ODE).

It builds on the previous parallelization work done at Cyberbotics and starts by investigating OpenMP as a threading alternative to Posix Threads. It then goes on to evolve the existing parallelization algorithm until we reach a satisfactory state where no further optimization is possible. We finish the high-level parallelization with an implementation of a heuristics system to fine-tune the algorithm parameters to attain optimal performances for all simulations in Webots. We finally dive into the LCP solver in ODE and discuss a first-pass parallelization of it.
Acknowledgements

I would like to start by expressing deep gratitude to Dr. Olivier Michel, with whom I worked with at Cyberbotics on this project. Thank you for your continuous support throughout this project and for giving me full autonomy on how to proceed with the work, yet at the same time providing me with the many insightful and highly useful discussions that helped shape the course I charted throughout the project.

I would like to thank my supervisor at EPFL, Prof. Auke Ijspeert at the Biorobotics Lab, for your kind support and understanding throughout the project, in addition to the autonomy availed to me during this time.

I am thankful to Dr. Luc Guyot at Cyberbotics, for helping me with the mathematically tricky parts of the project and for always being there for advice and mentoring, along with the invigorating discussions.

I am thankful to the wonderful team at Cyberbotics that helped make my stay here for the last 4 months very enjoyable, to say the least. It was a great pleasure working with you guys.

I would like to express a special thanks and gratitude to Prof. Simone Deparis, the Deputy of my masters program in Computational Science & Engineering (CSE) at EPFL for his continued support and help throughout these past 2 years at EPFL. You made my stay here unforgettable by being the perfect professor, mentor and program head by selflessly giving both of your time and advice whenever I needed them. I thank you for these.

Finally, no endeavour would be complete without the unconditional love and support of my family. No words on paper can express my gratitude. So I will just end here by dedicating my thesis to them.
## Contents

Abstract ................................................................. ii

Acknowledgements ......................................................... iii

1 Introduction ............................................................. 1
   1.1 Physics Engines ................................................ 1
   1.2 Anatomy of a Rigid-body Physics Engine ...................... 2
   1.3 Webots & ODE .................................................... 3

2 The Open Dynamics Engine ............................................. 7
   2.1 Basic Concepts .................................................. 7
   2.2 Previous Parallelization Work .................................. 8
   2.3 Problems with previous work ................................... 11

3 Project Overview ...................................................... 13
   3.1 Project Goals .................................................... 13
   3.2 Testing Methodology & Benchmarking ......................... 13
   3.3 Timeline ......................................................... 15

4 Threading in ODE_MT .................................................. 17
   4.1 Posix Threads .................................................... 17
   4.2 OpenMP ........................................................ 18
   4.3 OpenMP Vs. Pthreads ........................................... 19
   4.4 Other Threading Options ....................................... 20

5 Clustering 1st Iteration: Static/Dynamic Clustering .............. 21
   5.1 Algorithm ....................................................... 21
   5.2 Benchmarking & Results ....................................... 23
   5.3 Discussion ....................................................... 23

6 Clustering 2nd Iteration: Static/Dynamic Clustering .............. 25
   6.1 Algorithm ....................................................... 25
   6.2 Benchmarking & Results ....................................... 26
   6.3 Discussion ....................................................... 26
## Contents

7 Clustering 3rd Iteration: Duplication Clustering  
7.1 Algorithm  
7.2 Benchmarking & Results  
7.3 Discussion  

8 Clustering 4th Iteration: Declustering  
8.1 Algorithm  
8.2 Benchmarking & Results  
8.3 Discussion  

9 Clustering: Comparison & Conclusion  
9.1 Overview  
9.2 Discussion  
9.3 Benchmarks 1 and 3  
9.4 Benchmarks 2 and 4  

10 Heuristics: The magic of fine-tuning  
10.1 Idea  
10.2 Cell size Heuristic  
10.3 Threading Heuristic  

11 Parallelizing the LCP Solver  
11.1 The Dynamic Response Phase  
11.2 Parallelization Potential of the Dynamic Response Phase  
11.3 Benchmarking & Results  

12 Webots: Putting it all together  
12.1 Profiling in Webots  
12.2 Benchmarking & Results  
12.3 Discussion  
12.4 Performance discrepancies  

13 Conclusion  
13.1 Recap & Discussion  
13.2 Future Work  

A ODE_MT: Code & Concepts  
A.1 Clustering  
A.2 Threading  
A.3 Heuristics  

B Webots: Then & Now  

Bibliography
# List of Figures

1.1 Anatomy of a Physics Engine ................................. 4  
1.2 A screenshot of Webots simulating the Darwin-op robot ............. 4  
2.1 Original Collision Detection Algorithm in ODE ...................... 9  
2.2 Cluster Creation Algorithm .................................... 9  
2.3 Cluster Update Algorithm .................................... 10  
2.4 Multi-threaded scheme of ODE ................................ 11  
3.1 Performance comparison between single-threaded & clustering algorithm ... 16  
4.1 Comparison between OpenMP and Pthreads .......................... 20  
5.1 Static/Dynamic Clustering Algorithm ............................ 24  
5.2 Performance comparison between original clustering and 1st iteration ........ 24  
6.1 Clustering 2nd Iteration Algorithm ............................. 27  
6.2 Performance Comparison between clustering iterations 1 and 2 .......... 27  
7.1 Duplication Clustering Algorithm ............................... 30  
7.2 Performance Comparison between clustering iterations 2 and 3 ........... 31  
8.1 Declustering Algorithm ......................................... 34  
8.2 Performance comparison of iteration 4 with iteration 3 ............... 34  
9.1 Performance Comparison of all clustering iterations ................ 38  
10.1 The heuristic cycle ............................................. 42  
10.2 Effect of hashspace center coordinates on clustering .................. 43  
10.3 Example of a low crossover/declustering frequency world .............. 44  
10.4 Effect of cell size on performance ................................ 45  
10.5 Effect of thread count on performance ............................ 45  
11.1 Performance comparison between single-threaded & LCP parallelization ... 49  
12.1 Intel VTune Amplifier 2013 profile of the 8_khr-2hv.wbt world. ........ 54  
12.2 Intel VTune Amplifier 2013 profile of the blimp_lis.wbt world. ....... 54
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1</td>
<td>The dxClusterManager class</td>
<td>59</td>
</tr>
<tr>
<td>A.2</td>
<td>The dxThreadManager interface class</td>
<td>59</td>
</tr>
<tr>
<td>A.3</td>
<td>The dxHeuristic class</td>
<td>60</td>
</tr>
<tr>
<td>A.4</td>
<td>The Heuristic Alert class and State types</td>
<td>61</td>
</tr>
<tr>
<td>A.5</td>
<td>The dxHeuristicsManager class</td>
<td>62</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Benchmarks for previous ODE parallelization.</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>New Benchmarks for this project.</td>
<td>14</td>
</tr>
<tr>
<td>5.1</td>
<td>Static/Dynamic Clustering Benchmark Visualization</td>
<td>22</td>
</tr>
<tr>
<td>6.1</td>
<td>Clustering 2nd Iteration Benchmark Visualization</td>
<td>26</td>
</tr>
<tr>
<td>12.1</td>
<td>Performance comparison between ODE and ODE_MT in Webots</td>
<td>52</td>
</tr>
<tr>
<td>B.1</td>
<td>Webots sample worlds and their performance</td>
<td>68</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Physics Engines

Rigid-body Vs. Soft-body

Physics Engines, as the name implies are software that can simulate physics on a computer. These most commonly include rigid body dynamics such as simple convex primitives (boxes, spheres, cylinders, etc) colliding with each other and reacting to the resulting forces, but not deforming in the process, i.e. they maintain their shape and size and thus volume throughout the whole simulation. The second type of physics engines are therefore called soft-body physics engines, those which support the simulation of deformable bodies like sponges, cloth, etc. For reasons of computational complexity, the latter are used less-often, at least in real time physics simulations, where speed is of paramount importance.

Real time Vs. Offline

This brings us to a different classification of physics engines: real time vs. offline. Real time engines, most commonly used in video games, are interactive, in that they need to run at a very fast frame rate (30 fps is typical) and still leave enough time on the CPU for other simulation tasks such as Artificial Intelligence (AI), User Input, etc. Examples of these type include PhysX, Havok, ODE and Bullet. On the other hand, offline physics engines may take a very long time to simulate one frame and are most often used in making CGI (Computer Graphics Imagery) animations such as in movies and commercials. Examples of these include the physics engines that come with Autodesk 3D Studio Max, and Maya.

Proprietary Vs. Open-source

The last broad classification of physics engines is like any other software: proprietary vs. open-source. Proprietary engines, built by companies like Nvidia and Havok, are closed-source and have been refined over decades of use and are therefore usually quite expensive to use. Open-source engines, on the other hand, while typically simpler and less feature-packed, offer a viable alternative to proprietary engines since they are free and also have been tested and
improved over years, if not decades. Examples of these include the Open Dynamics Engine (ODE) and Bullet.

1.2 Anatomy of a Rigid-body Physics Engine

Phases

A typical rigid-body physics engine will have 2 phases:

1. Collision Detection: In the first phase, all the geometries in the scene are tested against each other to see which ones collide and a list of colliding objects with their point of contacts is generated.

2. Dynamic Response: Based on the results of the first stage, a series of forces is calculated, one for each contact point, the application of which will realistically simulate the motion of the various rigid-bodies involved in the scene.

Collision Detection

This stage typically involves 2 further phases:

1. Broad-Phase Collision Detection: Instead of directly trying to calculate the contact points between every pair of bodies, we first do a quick test to get a list of possible colliding pairs. This is most commonly done by employing bounding volumes. A bounding volume, is any simple primitive shape (usually a box or a sphere) that is used to enclose a more complicated geometry (say the wheel of a car) to greatly simplify initial collision testing. The key idea is if the bounding volume of 2 objects do not intersect each other, the objects cannot intersect each other.

2. Narrow-Phase Collision Detection: If we detect a collision between the bounding volumes of two geometries, we move to this second phase. This will typically just carry out the full collision detection, with an aim to find out the contact points. Note, however, that at this stage, a pair of geometries might be discarded if found not colliding.

On top of these, most physics engines usually employ optimization techniques that might be worth mentioning here. Instead of using the brute force $O(n^2)$ method of testing each geometry against each other, techniques such as spatial coherence (where the proximity of objects are found by diving up the space) and temporal coherence (where the fact that the results from one time step are usually closely related to the results from the previous timestep). In addition, we can employ hierarchical bounding volumes, i.e. bounding volumes for each part of a larger geometry where we progressively test bounding volumes lower in the hierarchy upon successful collision tests. Finally, for the narrow-phase collision, one simple way is to
test each polygon in one geometry against each in the other geometry. Apart from the obvious optimizations we can employ here, a physics engine will also typically implement different combinations of primitive collisions (ex: sphere vs sphere, box vs. sphere, cylinder vs. box, etc).

**Dynamic Response**

This second phase is typically the more computationally expensive, even for simple primitives like spheres and boxes. This usually involves 3 steps:

1. Data setup: Using the contact point information gathered by the collision detector, we need to formulate a set of constraints as a Linear Complementary Problem (LCP) of the form $Mz + q \geq 0$ with $z \geq 0$ and $z^T(Mz + q) = 0$ where the vectors $M$ and $q$ are built up and the vector $z$ is to be found.

2. Linear Complementary Problem (LCP) Solver: This is the most computationally expensive part. It involves solving the system of equations assembled in the first part. It can be done in 2 ways: using iterative methods such as

3. Force Integrator: At the final stage, we need to use the forces calculated, integrate them to get the velocity and integrate further to get the new positions of each object.

### 1.3 Webots & ODE

**Webots**

Webots, a robot simulation software developed by Cyberbotics Switzerland is a leading software package popular especially amongst universities, mainly used for educational and research purposes. It comes with built-in support for many of today's consumer robots including the e-puck, DARwIn-OP, Lego Mindstorms and the Aldebaran Nao humanoid robot.

Its main features include:

- Multi-platform support on Windows, Linux and Mac OSX.
- Large collection of robots that ship with Webots.
- Robot controller programs can be written in C, C++, Java, Python or Matlab.
- Some robots (like the e-puck and DARwIn-OP) can be directly programmed from Webots.
- Realistic physics simulation of the robots using the Open Dynamics Engine (ODE) physics engine.
Chapter 1. Introduction

• Add all the objects in the scene (robots, obstacles, floor, etc) and set their properties (mass, velocity, etc).

• Find out which objects collide with each other.

• Calculate exact coordinates of the contact-points between objects which collide.

• Calculate the reactionary forces between colliding objects and calculate resulting velocity.

• Using the timestep value, calculate the new coordinates of the colliding objects.

Figure 1.1: Anatomy of a Physics Engine

Figure 1.2: A screenshot of Webots simulating the Darwin-op robot
ODE

The Open Dynamics Engine, or ODE for short, is one of the most popular open-source engines available out there and has been used in many commercially available video games like Call of Jurez, Resident Evil and World of Goo. It also seems to be a favorite when it comes to Robotic Simulators and has been used in software like Gazebo, Webots and V-REP.

Although the next section is dedicated to explaining the Open Dynamics Engine and how it works, here are some of the reasons why it is so popular:

- Open Source: Free and time-tested.
- Excellent at what it does: Simulation of articulated rigid bodies. An articulated structure is created when rigid bodies of various shapes are connected together with joints of various kinds. Examples are ground vehicles (where the wheels are connected to the chassis), legged creatures (where the legs are connected to the body), or stacks of objects.
- Fast and robust with a highly stable integrator: ODE emphasizes speed and stability over accuracy. This might not be the best for robotic simulations, but as we will see throughout this thesis, it works pretty well nonetheless.
- Hard contacts: ODE uses ‘hard contacts’. This means that a special non-penetration constraint is used whenever two bodies collide. The alternative, used in many other simulators, is to use virtual springs to represent contacts. This is difficult to do right and extremely error-prone.
- ODE has a pluggable Collision Detection System! Although it comes with its own collision detection (which provides collision testing between primitives such as spheres, boxes, cylinders, planes, rays and triangular meshes), ODE also provides support for 3rd party colliders.
- ODE supports a range of joint types which make it ideal for robotic simulation software: these include ball-and-socket, hinge, slider, hinge-2, fixed, angular motor and universal.
- Multiple collision spaces and steppers: ODE provides simple, quad-tree and hashspace based spaces for computing the collision detection. In addition, for the LCP phase, it provides 2 stepping methods: the dWorldStep which solves a 'big matrix' and takes more time, and dWorldQuickStep, which employs an iterative method so is faster but less accurate.
2 The Open Dynamics Engine

2.1 Basic Concepts

Before we get started talking about ODE and its parallelization, we need to make a few concepts clear. ODE, like a typical physics engine has the 2 stages mentioned in section 1.3: Collision Detection and Dynamics. Both phases act on geometries involved in the physics scene. What exactly are these and how they are represented in ODE is defined below.

Body

A 'body' in ODE represents the basic rigid-body in a rigid-body simulation. It is represented by the 'dxBody' class in ODE. It is most commonly associated with one or more 'geoms' which define the actual shape of the rigid body. A body is used in the latter stage, i.e. Dynamics.

Geom

A 'geom' in ODE represents the actual shape of an object and is therefore used primarily in the first phase i.e. collision detection. It is defined by the 'dxGeom' class in ODE. Each geom may be assigned to a single or no body. However, a body can have many geoms. Hence the body-geom relationship is single-to-many.

World

A 'world' in ODE represents the collection of rigid bodies. That is, it contains all the objects in the physics simulation which will move. Each world has a list of bodies associated with it and each body has a pointer to the world it belongs to.
Chapter 2. The Open Dynamics Engine

Space

A ‘space’ in ODE represents the collection of geoms. That is, it contains all the objects in the world that have a shape and can therefore collide. Each space has a pointer to a list of geoms associated with it and each geom has a pointer to its parent space.

Joint

A ‘joint’ in ODE represents a relationship that is enforced between two bodies so that they can only have certain positions and orientations relative to each other. There are 8 types of joints: Ball-and-Socket, Hinge, Slider, Universal, Hinge-2, Fixed, Contact, and Angular Motor.

JointGroup

A ‘jointgroup’ in ODE is simply a collection of joints. Although these can be any type of the 8 joints mentioned above, in ODE they mostly consist of contact-point joints. ODE uses a callback mechanism during the collision detection phase that allows the client to add contact points between colliding geometries. Joint groups are an easy way of keeping track of these joints created so they can be removed at the end of each frame.

2.2 Previous Parallelization Work

Original Collision Detection

Figure 2.1 shows the 4 steps involved in the original single-threaded collision detection algorithm implemented in ODE.

ODE employs a spatial partitioning technique as described in section 1.3. It starts by splitting the physics space into a 3-dimensional grid of cells and proceeds to test geoms that occupy the same cell for collision. This does not include geoms that are too ‘big’ to fit into the cells such as planes (used for floors, etc). These geoms are tested separately against each of the geoms in the scene in a brute force O(n^2).

Clustering

Previous work done on ODE implemented a first-pass on the parallelization. This involved a clustering algorithm which is reproduced in Figure 2.2.

It involves reusing the same ‘hashspace’ as calculated by the collision detection algorithm to divide up the space and world into ‘clusters’. A cluster, in this sense, is just a space and
2.2. Previous Parallelization Work

Figure 2.1: Original Collision Detection Algorithm in ODE

1. Divide scene into hashspace comprising cells
2. Collide objects in a single cell with each other
3. Collide objects in a single cell with objects in the neighboring cells.
4. Collide all objects with the 'big' objects list

<table>
<thead>
<tr>
<th>Hashspace creation</th>
<th>Divide up scene into an I x m x n 3-dimensional grid</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1, m and n are tunable values based on scene size, number of objects, number of threads, etc.</td>
</tr>
<tr>
<td>Assign objects</td>
<td>Assign each object into one or more cells in the grid.</td>
</tr>
<tr>
<td></td>
<td>Based on the corner coordinates and size of the bounding box of an object.</td>
</tr>
<tr>
<td></td>
<td>May be assigned to one or more cells in the grid.</td>
</tr>
<tr>
<td>Find clusters</td>
<td>Start from a cell and recursively go through neighboring cell.</td>
</tr>
<tr>
<td></td>
<td>Group cells together into a single cluster if they contain objects.</td>
</tr>
<tr>
<td></td>
<td>Finalize cluster if no more neighboring cells with objects are left.</td>
</tr>
<tr>
<td>Analyze &amp; Repeat</td>
<td>Once clusters are found, calculate suitability metrics such as number of objects per cluster, number of clusters, etc.</td>
</tr>
<tr>
<td></td>
<td>Adjust parameters (I, m, n, grid size, etc) and repeat algorithm if metrics do not match suitability criteria.</td>
</tr>
</tbody>
</table>

Figure 2.2: Cluster Creation Algorithm
world containing a subset of geoms and bodies respectively. By traversing the hashspace in the manner described, we can separate geoms based on geographical proximity. Each cluster is then simulated independently and load-balancing is employed to distribute the cluster simulation on each thread.

However, the story is not complete here. We still need to update the clusters every frame to account for any ‘crossover’ between them. This, by nature has to be a single-threaded operation since two or more clusters might need to be combined into a single cluster. This is a simple traversal of all geoms in the space and detecting if a geom from a different cluster is occupying the same hashcell. This is done in $O(n)$ time as shown in Figure 2.3.

**Threading**

The clustering creation and update algorithm described above allows us to parallelize the ODE simulation loop. As we have seen, the cluster update part of the loop has to be single-threaded. Figure 2.2 shows this threading scheme. The main thread carries out the non-physics tasks at the beginning of the loop, while the physics threads are idling. The cluster update is then performed on a single-thread before they fork out again to perform the collision detection and dynamics simulation while the main thread waits till the frame end.
2.3. Problems with previous work

The first implementation of the parallelized version of ODE, referred to as 'ODE_MT' from hereon, was tested against 4 benchmarks which were defined specifically for the purposes of the project. These are:

1. **Single-robot simple behavior**: Single e-puck robot moving in an arena.
2. **Many-robot simple behavior**: Many e-puck robots moving in an arena, robot soccer.
4. **Many-robot complex behavior**: Many humanoid robots performing dance routines.

These 4 benchmarks are shown in Table 2.1.
From the benchmarks, we observe two things:

1. All objects are dynamic: This means that the objects in the worlds are all robots, except the plane, which is a static object. Dynamic bodies like robots undergo both the first and the second stage of ODE, i.e. the Collision Detection and LCP solver and are therefore ideally suited for the overall parallelization of ODE. Static bodies, however, only undergo collision detection and therefore do not benefit as much. Having only dynamic bodies in the scene tilts the results in our favor and does not give a fair picture of how Webots will perform in a general world with usually a single robot and a lot of static objects (walls, obstacles, etc).

2. All objects are synchronized: For the many robot benchmarks (2 and 4), the number is a multiple of 4 (the number of CPUs used in the testing) and are also synchronized (ex: the dancing khr robots) and therefore never collide with each other. This again tilts the results in the favor of the benchmarks and is not an indication of how Webots will perform in real life.

For the reasons discussed above, the previous parallelization attempt is not well-suited to a general Webots world. In the next chapter, we define a new set of benchmarks, which are all real-world samples from Webots and show how this previous parallelization is actually slower for these worlds.
3 Project Overview

3.1 Project Goals

The goals of this project are as follows:

1. Investigate ways to speedup the ODE simulation loop.
2. Investigate ways to speedup the Webots simulation loop.

We will thus start by looking at the ODE physics engine and look at ways to make it run faster (mainly by parallelizing it using multi-threaded programming). Once, we have exhausted all areas of investigation in ODE, we will jump a level higher and start looking at ways to parallelize the Webots simulation loop.

3.2 Testing Methodology & Benchmarking

As discussed in the last chapter, the original clustering algorithm, although works quite well for worlds with complicated robots (ex: humanoids), performs quite poorly in worlds with a lot of static objects. To avoid the pitfall of optimizing for benchmarks, we will chose 4 worlds in Webots, 2 of them with a large number of static object and the other 2 with a fair number of dynamic objects. All of these worlds represent typical Webots simulations and will used as-is. They are:

1. **Benchmark 1**: projects/robots/pioneer/pioneer3at/worlds/pioneer3at.wbt
2. **Benchmark 2**: projects/robots/bioloid/worlds/bioloid.wbt
3. **Benchmark 3**: projects/robots/create/worlds/create.wbt
4. **Benchmark 4**: projects/samples/demos/worlds/soccer.wbt
These new benchmarks are shown in Table 3.1.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.jpg" alt="Image 1" /></td>
<td><img src="image2.jpg" alt="Image 2" /></td>
</tr>
<tr>
<td><img src="image3.jpg" alt="Image 3" /></td>
<td><img src="image4.jpg" alt="Image 4" /></td>
</tr>
</tbody>
</table>

Table 3.1: New Benchmarks for this project.

**Testing Methodology**

Profiling was done in code, no 3rd party profiling tool was used. The functions used on the various platforms to measure time are described below:

1. **Linux**: For Linux, the `gettimeofday()` function was used which is defined inside the system header file: `time.h`. It gets the system time to an accuracy of 1 microsecond.

2. **Windows**: For Windows, the `QueryPerformanceCounter()` function was used which is a system function in Windows. It gets the clock ticks so we get an accuracy of a nanosecond on a modern day computer.

3. **Mac OSX**: For OSX, the `Microseconds()` function was used which is defined inside the system header file: `CoreServices.h`. It gets the system time to an accuracy of 1 microsecond.
3.3. Timeline

We measured the timings every frame for the 2 main stages of the physics pipeline: Collision Detection and the LCP solver. In addition, for the clustering algorithms, we also recorded the overhead time incurred for any additional maintenance work done due to the algorithms. The running average for all measurements were recorded. In addition, we saved this average over every 100 frames to disk and calculated the mean and standard deviation using these values. All timings noted in graphs and charts are the times spent in the main ODE function: \texttt{dSpaceCollideAndWorldStep()}. Measurements were stopped when the averages stabilized.

Unless otherwise stated, all tests throughout this report were done on a quad-core i5-2400 CPU running at 3.10 GHz with a memory of 4 GB on Ubuntu 12.04. Thread count was set to 4 for all benchmark testing unless otherwise stated.

**Benchmarks**

We see that Benchmarks 1 and 3 contain a large number of static objects (44 and 62 respectively) whereas Benchmarks 2 and 4 contain a low number of static objects (1 and 8 respectively). Figure 3.1 shows the relative performance of the original clustering algorithm and the single-threaded version of ODE on these benchmarks, using the profiling described before.

We can see that the worlds with a large number of static objects (Benchmarks 1 and 3) perform poorly under the clustering algorithm whereas the one with a fair number of dynamic objects (Benchmarks 2 and 4) perform better. There are a lot of such worlds in Webots, where one robot is roaming a large world with many obstacles. Our first priority will be to optimize the performance for these types of worlds, while still keep the performance gains for the second type of worlds, i.e. the ones with a fair number of dynamic objects, and a low number of static objects.

3.3 Timeline

Before we start looking at making the clustering algorithm faster, we will investigate a different threading library: OpenMP and see how it compares to Posix Threads, and thus decide whether to switch to OpenMP or stick to Pthreads.

We will then look start evolving the clustering algorithm, looking for ways in which we can optimize it, or even parallelize it further to get speed boost for benchmarks 1 and 3, while keeping the performance of benchmarks 2 and 4 steady, if not improved. After exhausting algorithmic optimizations, we will look at other methods of fine-tuning (via heuristics) to make each world perform optimally.

We will finally dig deeper into the LCP solver and attempt to parallelize it starting from a first-pass high-level parallelization going deeper down as far as time permits.
Figure 3.1: Performance comparison between single-threaded & clustering algorithm
4 Threading in ODE_MT

Apart from the algorithmic challenges involved in parallelizing ODE, the second major part was the threading libraries used. Two of the main ones were tested: Posix Threads (Pthreads for short) and OpenMP.

4.1 Posix Threads

Pthreads is a POSIX standard for threads. Therefore, by definition, it is cross-platform and implementations exist on all major operating systems including Microsoft Windows, Linux and Mac OSX. It defines low-level functions for the creation, management and deletion of threads. The pthread concepts used in ODE are described below.

Threads

Threads are created when ODE is initialized. A default of 2 threads per CPU are created and the number of threads can be specified by the user by calling a newly defined ODE function ‘dToggleODE_MT’ which takes in the number of threads as the argument. The threads are destroyed at the end when ODE is deinitialized.

Mutexes

Mutexes, short for 'Mutual Exclusion' are a mechanism to implement critical sections. The code enclosed by a mutex is guaranteed to be accessed by one thread at a time. ODE uses mutexes during the collision detection phase to test geoms in different spaces with the common geoms between them (such as floors, etc).

Barriers
Chapter 4. Threading in ODE_MT

A barrier is a thread synchronization mechanism. ODE_MT implements its own barrier called 'ode_barrier_t'. This is because although the pthread specification defines the barrier standard, the Mac OSX does not come with a native implementation of it. The ode_barrier_t thus wraps around the pthread definition for Windows and Linux and the OSX implementation uses conditional variables to implement it. See Appendix C for the code.

4.2 OpenMP

OpenMP, short for 'Open Multi-Processing', is a multi-platform, multi-language, and multi-architecture API. It is therefore supported on both the 32 and 64-bit versions of Windows, Linux and OSX. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. It is a vastly simplified parallelization tool to use compared to pthreads and the features used by ODE_MT are discussed below.

For the most part, multi-platform support for OpenMP is good. One caveat however is on OSX. Since OSX 10.8, Apple switched the native compiler on the Mac to clang, and not gcc which does not yet support OpenMP. The ODE_MT library is therefore compiled using the older llvm-gcc compiler on OSX.

The OMP parallel Directive

The most useful directive of OpenMP is called 'parallel' and is part of a preprocessor directive. There are 2 ways of using it: #omp parallel and #omp parallel for. It is followed by a block if using #omp parallel or a for loop if using #omp parallel for. The former is used when we want to multiple threads to run the block of code in the following block whereas the latter is used when we want to parallelize the execution of the for loop.

Critical Sections

The #omp critical directive is used to define a critical section. It is also a preprocessor statement and is following by a block (enclosed by braces) of code that is executed one thread at a time. It is used in ODE_MT to ensure only one cluster collides with common geoms at one time (similar to the pthread use in the previous section).

Thread Management

OpenMP requires the user to set the number of thread once and it automatically manages thread creation/update/deletion on its own. The user has very limited control over how or when these threads are created, synchronized or deleted.

18
4.3 OpenMP Vs. Pthreads

ODE_MT comes with threading implementations using both OpenMP and pthreads. The differences observed while using both are outlined below.

Ease-of-Use

Probably the biggest difference encountered was ease-of-use. To illustrate, the Pthreads version of the threading module contains 253 lines of code whereas the OpenMP version contains only 44. An interface called 'dxThreadManager' was designed which defines the threading API for ODE and was implemented once for OpenMP and once for Pthreads. The order-of-magnitude difference comes from the fact that pthreads is a low-level library that gives you full control over how threads are created, managed and destroyed whereas OpenMP is high-level API which gives you very little control over these things.

Reliability

A consequence of the simplicity of OpenMP and complexity of Pthreads meant that the Pthread implementation would be more error-prone. This was the main motivation to implement the OpenMP version in addition to the pthread version, i.e. to minimize the number of future bugs encountered.

Control

Posix threads has an advantage in this category. Being a low-level API, it gives you fine-grain control over your threads. Fortunately, the final implementation of ODE_MT does not need very fine-grain control over the threading so both APIs worked just fine for our purposes.

Compatibility

Since Webots is a cross-platform product, being supported on all the major-platforms, compatibility was a big concern. Both OpenMP and Pthreads are cross-compatible, however the caveat mentioned in section 4.2 means that a pthread implementation is favorable in this regard.

Performance

This is where posix threads get a chance to shine. Due to the low-level nature of the API, we would expect them to perform better than OpenMP. However, no major differences in performance were observed between the two versions in ODE_MT.

The differences described above are summarized in Figure 4.1.
Chapter 4. Threading in ODE_MT

4.4 Other Threading Options

Apart from the most popular options available, i.e. OpenMP and Pthreads, there are a couple of threading alternatives available. We will briefly discuss these over here.

- **Message Passing Interface (MPI):** MPI is a standardized and portable message-passing system to facilitate programming on a parallel computer. It is primarily meant as a forms of communication between clusters of computers and so is useful for use in things like cluster farms, used for rendering high-quality computer graphics imagery. Since the goal of the parallelization was on a single computer, MPI is not much use to us.

- **C++11 Threads:** With the introduction of the C++11 standard in 2011, the std::thread class was introduced which can be used to provide low-level threading control, similar to Pthreads. However, we decided not to investigate further due to 2 reasons: the OpenMP version is already working very well and is incredibly simple, and since the rest of ODE and Webots is built on the older C++ standard (C++03), we wanted to keep the same C++ standard throughout our code base. In addition, the new standard does not afford any new features (like thread pools) which we do not already have with Pthreads.
5 Clustering 1st Iteration: Static/Dynamic Clustering

5.1 Algorithm

As discussed, the original clustering algorithm separates objects into 2 categories:

- **Static Objects**: Objects that have a geom but not body anywhere in their hierarchy (ancestral or descendental).
- **Dynamic Objects**: Objects that have both a geom and a body associated with them anywhere in their hierarchy.

In the original clustering algorithm, there are 2 types of collision detections performed:

- **Type I**: Dynamic Objects Vs. Dynamic Objects. In Type I collisions, we test collision detection on multiple threads, one for each cluster created. All dynamic objects within a cluster are tested against each other for collision.
- **Type II**: Dynamic Objects Vs. Static Objects. In Type II collisions, we test collisions on a single thread. We test each dynamic object in each cluster with each static object in the scene.

Clusters were only created using the dynamic objects in a world/space and we ran into the problems discussed in the previous section. For the Clustering 2.0 algorithm, we had to extend the definition of a cluster to static objects, i.e. we define a categorization of clusters as follows:

- **Static Clusters**: A cluster consisting of static objects only. This type of cluster is introduced to reduce Type II collisions and move the remaining number to different threads to get additional speedup.
Chapter 5. Clustering 1st Iteration: Static/Dynamic Clustering

- **Dynamic Clusters:** Dynamic clusters are the same as the old definition of a cluster, i.e. they contain only dynamic objects.

With this new definition of clusters and the corresponding changes to maintain static/dynamic clusters throughout the code, we make the following changes to the crossover algorithm:

1. Dynamic Clusters are merged normally as before.
2. Static Clusters are also merged normally as before.
3. When a dynamic and static cluster come close enough, they are pseudo-merged. This means that the objects in the dynamic cluster are tested against the objects in the static cluster. However, they are not truly merged, like in case 1, i.e. the geoms and bodies are not transferred between spaces.

Table 5.1: Static/Dynamic Clustering Benchmark Visualization
5.2 Benchmarking & Results

For this iteration of the clustering algorithm, we immediately see that not just the dynamic moving objects in the scene such as the robots are clustered, but the static objects like the walls and obstacles are also clustered.

We can see that here Benchmark 1 comes to life in this version since all the pillars are clustered. Benchmark 2 has no change, since there are no static objects in this benchmark. Benchmark 3 and 4 show the walls in the scene belong to the same color, and hence the same cluster. We therefore expect a good performance boost for Benchmarks 1, 3 and 4. The results are shown in Figure 5.2.

5.3 Discussion

We can see that our predictions hold. Benchmarks 1 and 3 have a large number of static objects in them and therefore benefit from static clustering the most. Benchmark 4 has a few static objects whereas Benchmark 2 has no static objects at all. We see that scenes with a single-robot roaming a big world with a lot of objects like obstacles, walls, etc show the largest reduction in collision tests. This potentially goes from n collision tests, where n is the number of static geoms in the scene to 1 collision test (with the floor) if the robot starts off sufficiently far from any surrounding objects.
Chapter 5. Clustering 1st Iteration: Static/Dynamic Clustering

**Cluster Creation**

Static Clusters: automatically cluster each static object into its own static cluster.

Dynamic Clusters: Cluster together dynamic objects together based on physical proximity.

**Crossover**

Static Clusters: Static clusters are merged together if they get sufficiently close together.

Dynamic clusters: Dynamic clusters are merged together if they get sufficiently close to each other.

Static/Dynamic Clusters: If a static and a dynamic cluster come close together, they are pseudo-merged, i.e. the objects are not transferred between them but they are collided against each other.

Figure 5.1: Static/Dynamic Clustering Algorithm

![Performance comparison between original clustering and 1st iteration](image)

Figure 5.2: Performance comparison between original clustering and 1st iteration
6 Clustering 2nd Iteration: Static/Dynamic Clustering

6.1 Algorithm

Although an improvement over the original clustering algorithm, looking at benchmarks 3 and 4, we can immediately notice a room for improvement: the static clusters end up merging together resulting in very few clusters. This is bad since if a static cluster is collision tested against a dynamic cluster, the bigger the static cluster, the more collision tests have to be performed, and hence the slower it is.

We can overcome this by changing the crossover rules as follows:

1. Dynamic clusters are always merged, as before.
2. Static clusters are never merged, even when they collide with each other.
3. A static cluster and a dynamic cluster are pseudo-merged, as before.
Chapter 6. Clustering 2nd Iteration: Static/Dynamic Clustering

6.2 Benchmarking & Results

Using these updated rules, we run the benchmarks again. We expect to see Benchmarks 1 and 3 show the most improvement. Figure 6.2 shows the resulting performance.

6.3 Discussion

We can see that Benchmarks 1, 3 and 4 show improvement. The last one is a surprise since the soccer field has only 8 static objects as compared to the dozens in the other two benchmarks. We can explain this improvement if we consider the speed of the robots in the soccer simulation. This is fast enough to have a high frequency of collisions with the soccer field boundaries, i.e. the 8 geoms. Thus, separating these into 8 static clusters instead of 1 has a considerable effect on the total number of collision tests that need to be performed, resulting in a speedup.
6.3. Discussion

Cluster Creation

Static Clusters: automatically cluster each static object into its own static cluster.

Dynamic Clusters: Cluster together dynamic objects together based on physical proximity.

Crossover

Static Clusters: Static clusters are never merged, even if they physically collide with each other.

Dynamic clusters: Dynamic clusters are merged together if they get sufficiently close to each other.

Static/Dynamic Clusters: If a static and a dynamic cluster come close together, they are pseudo-merged, i.e. the objects are not transferred between them but they are collided against each other.

Figure 6.1: Clustering 2nd Iteration Algorithm

![Figure 6.1: Clustering 2nd Iteration Algorithm](image)

Figure 6.2: Performance Comparison between clustering iterations 1 and 2

![Figure 6.2: Performance Comparison between clustering iterations 1 and 2](image)
7 Clustering 3rd Iteration: Duplication Clustering

7.1 Algorithm

The previous clustering algorithm is already a big improvement over the original algorithm. However, we still need to maintain a list of static clusters and handle the crossover of these clusters with regular dynamic clusters.

This has the disadvantage of possibly combining 2 or more dynamic clusters with one or more static clusters into a single cluster, run on a single thread. For example, if two robots are near the same static object, say a boundary wall, all three objects are clustered together, even if the two robots have no chance of colliding with each other (opposite far ends of the wall).

In addition, consider a case where there are 2 sets of dynamic objects: those that touch the floor (wheeled robots, etc) and those that don’t (quadrocopters, etc). In our clustering algorithm thus far, we still maintain a list of ‘large’ objects which are tested against every dynamic object in the scene on a single-thread. Thus, the quadrocopters in the scene are tested against the floor every frame, even though they may never collide with it. We therefore have a possible optimization here.

Thus, the rules are updated as follows:

1. Dynamic clusters are never duplicated, but always merged.

2. Two static clusters are not duplicated or even merged when they come close together.

3. When a dynamic cluster come close enough to a static cluster, we need to crossover as before. However, this time we duplicate the static cluster objects into the dynamic cluster.

7.2 Benchmarking & Results
Chapter 7. Clustering 3rd Iteration: Duplication Clustering

Cluster Creation

Static Clusters: automatically cluster each static object into its own static cluster. No duplication occurs yet.

Dynamic Clusters: Cluster together dynamic objects together based on physical proximity. No duplication occurs yet.

Crossover

Static Clusters: Static clusters are never duplicated nor merged, even if they physically collide with each other.

Dynamic clusters: Dynamic clusters are never duplicated, but always merged if they come close enough to each other.

Static/Dynamic Clusters: If a static and a dynamic cluster come close together, duplication takes place. The static objects in the static cluster are duplicated into the dynamic cluster.

Figure 7.1: Duplication Clustering Algorithm

Due to the nature of this iteration, we do not expect any gains for single-robot simulations like Benchmarks 1 and 3 or simulations with no static clusters like Benchmark 2. Benchmark 4, with 7 dynamic objects and 8 static objects, benefits from this iteration. In terms of visual debugging, this iteration does not add anything new and the cluster cells look the same as the previous iteration. Figure 7.2 shows the resulting performance.

7.3 Discussion

We see from the results that all benchmarks show a slight improvement. This can be explained due to the nature of duplication. Instead of updating the clustering data structures for the static geoms upon crossover, we simply duplicate the geoms. This small reduction in work translates to the small improvement shown in the benchmarks.
Figure 7.2: Performance Comparison between clustering iterations 2 and 3
8 Clustering 4th Iteration: Declustering

8.1 Algorithm
Looking at our clustering algorithm, we realize one last source of unoptimal behavior: declustering. The algorithm only merges clusters based on physical proximity but does not unmerge them upon separation.

This is a bit tricky to do. We need to employ an O(n) time algorithm to achieve declustering, ideally similar to the crossover algorithm. In fact, we can achieve this in the same hashspace traversal as the crossover algorithm. While traversing the hashspace cells looking for crossover conditions, we mark the cells with a tracker in a fashion similar to the original cluster creation algorithm, i.e. neighboring hashspace cells containing geoms are marked with the same id. If we detect that the tracker id needs to be incremented within the same cluster, this is a sufficient condition for declustering.

8.2 Benchmarking & Results
We expect benchmarks with multiple dynamic objects to benefit from this declustering algorithm, especially one that has a lot of merging and separation of clusters. We expect that Benchmarks 1, 2 and 3 do not benefit much from this iteration but Benchmark 4, with its rapidly moving soccer-playing robots should benefit. Results are shown in Figure 8.2.

8.3 Discussion
From the results, we see our predictions were completely false. Benchmarks 1, 2 and 3 benefit from this iteration whereas Benchmark 4 actually suffers.

This can be explained by the additional overhead encountered with the declustering algorithm. Benchmark 4, with its rapidly moving e-puck robots is actually a bad world to benefit from this declustering since the frequency of an e-puck colliding with a wall or another e-puck robot
Chapter 8. Clustering 4th Iteration: Declustering

Cluster Creation

Static Clusters: automatically cluster each static object into its own static cluster. No duplication occurs yet.

Dynamic Clusters: Cluster together dynamic objects together based on physical proximity. No duplication occurs yet.

Crossover

Static Clusters: Static clusters are never duplicated nor merged, even if they physically collide with each other.

Dynamic clusters: Dynamic clusters are never duplicated, but always merged if they come close enough to each other.

Static/Dynamic Clusters: If a static and a dynamic cluster come close together, duplication takes place. The static objects in the static cluster are duplicated into the dynamic cluster.

Figure 8.1: Declustering Algorithm

Figure 8.2: Performance comparison of iteration 4 with iteration 3
and separating is high. At each such separation, there is an overhead of separating the clusters which ultimately slows down the simulation rather than speeding it up. If the e-pucks are slow-moving or if the soccer field is much bigger, we would expect an improvement in speed.

Lastly, we see the first three benchmarks benefit from this iteration mainly because they either have a slow-moving robot in a relatively large world (Benchmarks 1 and 3) or a lot of dynamic objects that are merged and separated in the simulation (Benchmark 2).
9 Clustering: Comparison & Conclusion

9.1 Overview

We started out with an initial clustering implementation which worked well for the defined benchmarks (multiple number of complicated robots in a sparsely populated world) but did not perform as well (or even performed worse) in many real-world Webots worlds.

We then went through 4 iterations of improving the clustering algorithm for 4 benchmarks that we defined, which were real-world Webots worlds with a mix of large number of static/dynamic objects. For each iteration, we refined our algorithm and discussed the performance changes over these benchmarks over the previous iteration. We now give an overall picture of the performance changes over all 4 iterations. This is illustrated in Figure 9.1.

9.2 Discussion

From Chart 9.1, we can see 2 main variations of performance for the set of benchmarks. Benchmarks 1 and 3, which had a single robot and a large number of static objects shows one type of variation whereas Benchmarks 2 and 4 which had a large number of dynamic objects shows a different type of variation.

9.3 Benchmarks 1 and 3

For the first group (Benchmarks 1 and 3), we see that the original clustering algorithm performed quite badly compared to the single-threaded version. This is because the original clustering algorithm was designed to speedup worlds with a large number of dynamic objects, and Benchmarks 1 and 3 each have 1 dynamic object (a pioneer and a create robot respectively) and a large number of static objects. We saw that the overhead of maintaining and updating the hashspace degraded the overall performance considerably.
However, for each iteration of our algorithm, the performance of these benchmarks improved consistently, starting from the static/dynamic clustering algorithm where the amount of collision tests was drastically brought down for these benchmarks to the duplication clustering which had little effect to the final declustering algorithm which improves performance over time.

9.4 Benchmarks 2 and 4

Benchmarks 2 and 4 on the other hand, show consistently good improvements over all iterations of the algorithm. Even in the original clustering algorithm, these worlds showed a considerable improvement, due to the presence of more than a single dynamic object.

Benchmark 2 showed little improvement over the first three iterations and showed a good improvement over the last iteration, when declustering was implemented. This is due to the absence of any static geoms in the scene, apart from the floor, which prevented this benchmark from reaping the fruits of any of the iterations. The declustering algorithm works for both static and dynamic clusters, because of which we saw an improvement in the last iteration.

Benchmark 4 benefited more from all iterations, except the last one, which came as a surprise. Since it has an almost equal number of static and dynamic objects, we saw it benefit from the static/dynamic, and duplication clustering iterations. To begin with, benchmark 4 already showed the best improvement going from single-threaded to the original clustering algorithm, mainly due to the 6 e-puck robots in the world. Due to the simple geometry of the
9.4. Benchmarks 2 and 4

e-puck robots, we saw the declustering iteration add more overhead to this benchmark, thus degrading overall performance. We finally noted that an increase in the size of the soccer field or a reduction in speed of the e-puck robots would favor a higher speedup.
10 Heuristics: The magic of fine-tuning

10.1 Idea

After having perfected our clustering algorithm and leaving no room for improvement, we move on to another technique in optimization: heuristics. Although usually used in a slightly different context, here we will define a heuristic as an experience-based technique for problem-solving.

We have seen that the clustering algorithm uses some parameters which can be fine-tuned across different simulations for better results. Some of these parameters include:

- **Hashspace cell size**: The hashspace is divided into an $l \times m \times n$ grid. The higher the values of $l$, $m$ and $n$ are the more computationally expensive the clustering algorithm is going to be. However, the lower the values of $l$, $m$ and $n$ are, the more the hashspace resembles a single space, and hence we end up with one giant cluster and therefore lose all the speedup gained with our work. Therefore, for each world in webots, there must be an optimal value of the cell size to get maximum performance.

- **Hashspace center**: The hashspace is defined by its center coordinates, the size of each hash cell and the number of such cells. The coordinates of the center, although they don’t affect the total number of hashcells, might still affect the overall performance. This is most evident in the stewart_platform.wbt world in Projects->Samples->Demos where a number of boxes are stacked over rotating platform. Figure 10.2 shows how if the hashspace center coordinates are ill-defined, everything is clustered into a single cluster, even if the hashcell size is matched to the size of the box. If we move the center coordinates such that there is a cell boundary between the rotating contraption and the stack of boxes, we can get 2 clusters at least. In addition, if the hashcell size is small enough, we can cluster all 6 dynamic objects separately.

- **Crossover Frequency**: Even in the final algorithm, there is a single-threaded part where the clusters need to be checked for merging. If they need to be merged, the objects in
one cluster are migrated to the other cluster. The frequency of checking this condition can be controlled and depends highly on the nature of the world. For example, a world with a lot of robots moving around and colliding at relatively high speeds will need to be checked more often as compared to a world where the robots are spaced far in-between and are moving at a relatively slow pace. A good example of a world where this applies is the 8_khr-2hv.wbt located in Projects->Robots->khr->khr-2hv folder. It consists of 8 khr robots dancing in perfect synchronization. We see that the robots, once clustered need not be checked for merging at all since they will never collide. Figure 10.3 illustrates this.

- **Declustering Frequency**: The idea is the same as the crossover frequency. Each cluster needs to be checked for a possible separation into 2 or more clusters. The same rules apply as for the crossover frequency, i.e. worlds with fast-moving robots need to be checked more often for declustering than worlds with slow-moving or synchronized robots.

- **Thread Load-balancing**: Although not part of the clustering algorithm, load-balancing is an important aspect of any parallelization effort. This means controlling things like the total number of threads to how the threads are managed (threadpooling vs. static load assignment) to which clusters are assigned to which threads. Part of the problem is also platform-dependent, for example Windows manages threads differently than Linux on the kernel level so the thread heuristic rules might be different in Windows than in
10.2 Cell size Heuristic

The first heuristic we implemented was the cell size heuristic: changing the cell size to get an optimal number of clusters for each world. The basic outline follows:

1. We consider the hashspace to be divided up into equal number of cells in each dimension, i.e. \( l = m = n \) when dividing up the hashspace into \( l \times m \times n \) cells.

2. We define 2 sizes, a maximum and a minimum. This means the hashspace is initially divided up into \( 5 \times 5 \times 5 = 125 \) cells.

3. The optimal number of clusters are found by running the collision detection once and finding the number of separate 'entities'. Here we define an 'entity' as a number of geoms/bodies connected with fixed or temporary joints.

4. After a fixed amount of time, the number of clusters formed are tested against the optimal number of clusters and if they do not match, the cell count is increased by 1 in each dimension.
5. We run the simulation again for a fixed amount of time and repeat steps 3 and 4 until we either get the optimal number of clusters or the maximum size is reached.

6. The maximum cell count in each dimension is set to 20, i.e. the maximum number of cells in the hashspace will be $20 \times 20 \times 20 = 8000$.

7. The minimum cell count of 5 and maximum number of 20 were chosen based on testing out the heuristics algorithm on all the worlds that come with Webots and choosing these values for optimal performance.

### 10.3 Threading Heuristic

This is another simple heuristic that we investigated. The idea was to do automatic load-balancing by setting the number of threads to the number of clusters every time the cluster count changed (upon merging or separation of clusters). This way, we can let the Operating System handle the scheduling of the threads and this rids us of the responsibility of balancing the load on the threads. As it turned out, there is no real benefit to doing this and the drawback is possibly creating and destroying threads on the fly, depending on which library we use. The results of the benchmarks with this heuristic turned on show that there is no real benefit to it so we decided to scratch it at the end.
10.3. Threading Heuristic

Figure 10.4: Effect of cell size on performance

Figure 10.5: Effect of thread count on performance
Parallelizing the LCP Solver

After parallelizing the whole physics pipeline from a high-level, and looking at ways of optimizing it for different worlds in Webots, we finally move on to the nitty gritty details of the LCP solver.

11.1 The Dynamic Response Phase

The collision detection phase outputs a list of contact joints between pairs of bodies. The second phase (dynamic response) is broadly divided into these steps:

1. **Island Creation:** Based on the pairs of bodies which collide, we separate out the list of bodies in the world to ‘islands’. An island is simply a list of connected bodies through joints.

2. **Data preparation:** All the matrices and values needed for the 3rd stage is computed here. This includes things like the inertial tensor for each body, and the constraint and Jacobian matrices for the LCP solver.

3. **LCP stepper:** This is the meat of the dynamics phase. It takes the LCP matrices equated in the previous step and solves it to get the resulting forces on each body through the joints.

4. **Force Integrator:** The last step involves taking the reaction forces calculated in the previous step and integrating them for each body to get the velocity (both angular and linear). This is further integrated to get the position updates for each body.

The parallelization of each step is discussed now.

11.2 Parallelization Potential of the Dynamic Response Phase
Chapter 11. Parallelizing the LCP Solver

Island Creation

The algorithm for the island creation goes traverses the list of bodies in the world through the joints. Therefore, it might possibly need to traverse all the bodies in the world and assign them to a single island. This makes parallelizing this part not trivial since 2 threads might end up accessing the same body or joint and writing to the same location. We could try to investigate splitting up the work among the threads based on the islands created in the previous frame, since they are not likely to change very much in a single time step.

Data Preparation

This part contains 2 types of traversals: fixed-length traversal of the list of bodies and joints, and variable length traversal. The fixed-length traversal can be parallelized easily whereas the variable-length will take a bit of work. Even the fixed-length traversals can be tricky since each iteration may depend on the results of a previous iteration.

LCP Stepper

The LCP stepper takes the most amount of time in the dynamic response phase (typically 70% of the time) and so we will benefit the most by parallelizing this. Unfortunately, it is not trivial. ODE has 2 stepper functions ‘dWorldStep’ and ‘dWorldQuickStep’. DworldStep uses an iterative Dantzep solver which is slow but accurate. dWorldQuickStep, on the other hand uses a much faster iterative solver but sacrifices accuracy. For Webots, dWorldQuickStep is good for simple simulations like stacks of boxes but is unusable for anything more complicated (like a robot!). Due to the iterative nature of the algorithm, this will be the most difficult to parallelize.

Force Integrator

The last step should be the easiest to parallelize. It just traverses the list of bodies and uses the forces calculated to update the angular/linear velocity and hence the orientation/position of the body. It should be simple to parallelize this on multiple threads.

11.3 Benchmarking & Results

We only had only enough time to investigate a high-level parallelization of the dynamics phase. After calculating the islands at each time step, we parallelize the rest of the dynamics phase vertically, i.e. after performing step 1 on a single-thread, steps 2, 3 and 4 are run, as they are, on different threads.

Although not a big breakthrough, we do expect to get significant gains by this vertical parallelization. It should be on par with our clustering algorithm for some worlds (large number of complex robots where the dynamics phase overshadows the collision detection phase), and not have any gains at all for other worlds (scenes with a single robot in them). We tested it on our usual benchmarks and the results are discussed below.
11.3. Benchmarking & Results

Figure 11.1: Performance comparison between single-threaded & LCP parallelization

11.4 Discussion

We can observe that the LCP parallelization is a clear win. Benchmarks 1 and 3 show no little or no improvement. This is to be expected due to the fact that there is only one robot in Benchmark 1 and one robot and three stationary dynamic objects in Benchmark 3.

Benchmarks 2 and 4 however show a clear speedup, with the latter benefiting the most, as expected. It is almost twice as fast! We see this as a consequence of the fact that there are 7 dynamic objects moving around the scene at any one point, and if each of them is simulated on its own thread, we can get up to an ideal speedup of 3 in the LCP solver, without the additional overhead of the clustering algorithm. Since the LCP solver takes around 80% of the time in the soccer simulation, we have an overall speedup of about 2.0 in the ideal case.
Now that we have finished evolving our clustering algorithm to a point where we are satisfied, all that is left is to plug in ODE_MT into Webots and see how the different worlds perform.

### 12.1 Profiling in Webots

Webots simulations can be run in 2 modes:

1. **Realtime Mode:** The simulations are run in 'realtime', which means they take the same amount of time to move as they would if they were running in real life. This means, the physics stepping is made to synchronize with the controller input. For example, if a timestep of 32 ms is specified and the physics simulation is done in 10 ms, it will have to wait 22 ms to run the next step.

2. **Fast Mode:** In fast mode, Webots is run without regards to controller timing consideration. This means the physics simulations and controller code is run at full speed, without even a graphical output, to enable Webots to 'fast-forward' time. This is extremely useful if the user wants to see how their simulations perform in the long-run.

Webots comes with a timer which measure simulation time along with a 'realtime factor' which is a number that shows how fast the virtual simulation is running compared to if it were running in realtime. The expected value of this factor should therefore be 1.0 for the realtime mode and (hopefully) greater than 1.0 for the fast mode. It therefore gives us a good measure of how fast the underlying physics and controller code is capable of performing.

### 12.2 Benchmarking & Results

We will now use this profiling process to run our 4 benchmarks using the single-threaded version of ODE and the mult-threaded version and compare performances. The realtime
factor was updated once every second and the value saved to disk for a minute of simulation. These values were then used to calculate the mean and standard distribution, for both the single-threaded and multi-threaded versions. The normal distributions thus obtained were plotted on the same graph for comparison. The results are shown in Table 12.1. The red lines represent the single-threaded version and the blue lines represent the multi-threaded version.

![Graphs showing performance comparison between ODE and ODE_MT in Webots](image)

**Table 12.1: Performance comparison between ODE and ODE_MT in Webots**

### 12.3 Discussion

From the results we can see that the performance difference is not as pronounced as when measured inside ODE. We can see that the means are higher, and the standard deviations are lower, but the general frame-to-frame speedup is marginal. Benchmark 2 shows the least improvement. This is due to the fact that most of the dynamic objects in the Bioloid simulation are balls, which after rolling around for a short time, stop and are therefore not factored into the simulation time. The declustering algorithm here thus helps little. However, the declustering does stop the simulation of these rolling balls earlier (by assigning them to a
different thread) and therefore the spread is smaller than the single-threaded version. A full list of the performance comparison between the single-threaded and multi-threaded versions of the sample worlds in Webots is given in Appendix B.

12.4 Performance discrepancies

In this section, we discuss the apparent performance discrepancies between the profiling done in ODE and the profiling done in Webots. We start by looking at the Webots simulation loop. This consists of 3 main parts:

1. **Controller**: In Webots, each controller is a separate process that is compiled and run independently of Webots. Each robot has a controller process attached to it and each controller process communicates with Webots using pipes.

2. **Physics**: The physics step is performed once every controller time step in normal mode, even if it finishes earlier (which it usually does). This is done on the main Webots thread.

3. **Graphics**: The rendering part is also done on the main Webots thread and is responsible for displaying the simulation on screen. It uses the Ogre graphics engine which uses the OpenGL rendering library. In fast mode, this step is not performed.

To visually see what happens in a typical Webots simulation every frame, we used a profiling software from Intel, called the Intel VTune Amplifier 2013. It is a very nice way to see what is going on at the kernel level in Webots. Figure 12.1 shows this. We see one thread at the top, which is the main Webots thread and 8 more threads below it, one for each controller associated with the khr robots. Dark green areas show code running, light green means the thread was idling at that stage. We immediately see that most of the work is done on the main thread and the controller threads are usually very light. Their running time is typically in the microseconds. This is not a problem for heavy simulations like the 8_khr-2hv.wbt world where the physics simulation time is in the milliseconds.

However, for lighter simulations (like the benchmarks in this report), we look more closely at the threading performance. An extremely light world, like the blimp_lis.wbt in the samples folder is considered. The resulting thread profile is shown in Figure 12.2. We see that the controller takes a total of 180 us (controller code + piping time). The physics simulation, on the other hand takes a mere 600 to 800 microseconds. Therefore, at this scale, the threading effects become noticeable. They are even more pronounced if the controller threads are interrupted when the user is running other applications in the background since the controller thread is a low-priority one, resulting in context-switches. Finally, these effects are multiplied when we have worlds with several simple robots and therefore several controller processes like the soccer.wbt world.
Chapter 12. Webots: Putting it all together

Figure 12.1: Intel VTune Amplifier 2013 profile of the 8_khr-2hv.wbt world.

Figure 12.2: Intel VTune Amplifier 2013 profile of the blimp_lis.wbt world.
Conclusion

13.1 Recap & Discussion

We started off with an initial clustering implementation inside the Open Dynamics Engine physics engine with an aim to improve general performance in Webots, not just worlds where the clustering algorithm is suited. We first scoured the different worlds that come with Webots and looked for 4 benchmarks, 2 of which fared quite poorly under the original algorithm and 2 of which fared quite well. The former had a lot of static objects in them and the latter a fair amount of dynamic objects. We did not define our own benchmarks to avoid falling into the same pitfall of optimizing for benchmarks, as in the original parallelization attempt.

The first thing we did was to switch the threading implementation from Pthreads to OpenMP to compare the two and select the better one. We made the choice to go with OpenMP due its vastly greater ease-of-use, considering they both fair equally on the performance front.

With the aim to make Webots faster, we started with looking at why some worlds in Webots performed so poorly under simulation and proceeding to evolve the clustering algorithm, step-by-step and measuring the impact on performance on the 4 benchmarks we selected. After 4 such iterations, we found the performance to be ‘good-enough’ with our final algorithm outperforming the single-threaded version in the weakest performing benchmarks and still working just as well in the better performing benchmarks.

With no room left to evolve the algorithm, we investigated the effects of tuning different parameters in the algorithm on performance and implemented a heuristics system, which allows ODE to dynamically fine-tune these parameters to different values for different worlds to get optimal performance for each world.

Lastly, we looked at the last remaining piece of the puzzle: the LCP solver. Our first attempt at high-level parallelization was a success with the dynamic benchmarks showing the same level of improvement as our original clustering algorithm, but without the small overhead of clustering. We ran out of time here and ended by noting the different areas of the LCP solver
that could be parallelized.

13.2 Future Work

Although having exhausted the clustering algorithm and the collision detection phase of the ODE, we have barely scratched the surface of the complicated LCP solver. The LCP solver is the meat of any physics engine and is quite challenging and difficult to get right due to the need to create and solve a large system of linear equations. This means that any serious parallelization attempt will need a thorough understanding of the LCP algorithm implementation inside ODE. Even then, undertaking the parallelization will be error-prone and we will need to tread carefully.

The good news, however, is that the LCP solver shows a lot of potential for parallelization, since it involves a lot of operations on the rows and columns of matrices. This is ideally suited to porting it to the GPU where we can hope to get speedups of an order-of-magnitude over the CPU. Today's modern GPUs contain hundreds of so-called 'stream processors' which do not have a lot of logic/code execution control but a fair amount of arithmetic hardware which enables it to do a lot of number crunching. We have seen in Webots that in most worlds, the LCP solver takes 70-80% of the time and with complicated robots, this number can go up to 95%. Therefore, getting a speedup of even 2.0 in the LCP solver will benefit the overall simulation considerably.

Lastly, at the moment, Webots is severely limited by the number of complicated robots it can simulate at once, especially if they collide with each other a lot. A typical example would be the case of a humanoid soccer match, with 22 players overall playing at any one point. If the humanoids are slow-moving (which they typically are), even the super-fast mode in Webots will need to be run for hours before we can simulate the whole match. Therefore, there is a certain value in taking these simulations to a cluster of computers where we have hundreds of cores available. This should enable Webots to complete the simulation in a matter of minutes rather than hours. This is especially useful for universities, which have access to supercomputing infrastructures that might be interested in simulating many complex robots in large worlds.
The implementation details of the multi-threaded version of ODE, referred to as 'ODE_MT' in this document is discussed in this chapter.

The multi-threading component of ODE_MT is divided into 3 main parts:

1. **Clustering**: Most of the algorithms discussed in this report fall under this category. It handles the clustering of geoms and bodies in the physics world/space and thereafter maintaining and deleting these clusters.

2. **Threading**: Independent of how the clusters are created, updated and destroyed, ODE_MT uses a threading component to run the simulation of one or more clusters on one or more threads. This part comprises not just the simulation, but also the load distribution (how many clusters are run on each thread) before that.

3. **Heuristics**: The heuristics mechanism is ODE_MT handles the on-the-fly fine-tuning of the clustering algorithm parameters, and therefore has strong dependence on the clustering component.

These different components are discussed in more detail now.

### A.1 Clustering

The Clustering component consists of 2 main classes:

1. **dxClusteredWorldAndSpace**: This is the main class that contains most of the clustering code. It is defined inside cluster.h and cluster.cpp.

2. **dxClusterManager**: This is the manager class that manages all the clusters for a particular world and space. It is a singleton and is defined inside clusterManager.h and clusterManager.cpp. The dxClusterManager class is reproduced in Figure A.1.
A.2 Threading

The Threading component consists of an interface and 2 main classes, one for the OpenMP implementation, and the other one for the Pthread implementation:

1. **dxThreadManager**: This is the interface class which defines the functions any thread manager has to implement. It is defined inside the file threadManager.h. The code has been reproduced in Figure A.2.

2. **dxThreadManagerOpenMP**: This is the implementation of the dxThreadManager interface class for OpenMP. It is defined inside the file threadManagerOpenMP.h and threadManagerOpenMP.cpp.

3. **dxThreadManagerPthread**: This is the implementation of the dxThreadManager interface class for Posix Threads. It is defined inside the file threadManagerPthread.h and threadManagerPthread.cpp.

A.3 Heuristics

The heuristics component consists of 2 main classes: the dxHeuristicsManager and the dxHeuristic class:

1. **dxHeuristic**: This defines the basic heuristic, which is just an input, a rule, followed by an output. That is, a heuristic keeps track of one or more values over time, and based on a rule and these input values, changes something in the clustering algorithm (the output value of the heuristic). It is defined inside heuristic.cpp and heuristic.h, the code of which is reproduced in Figure A.3. Observe that the dxHeuristic class has a list of heuristic alerts. The heuristic will keep track of a value (say the cluster count) and report back if for example the value has changed, or fallen below a certain value, etc. A complete list of these alert types is given in Figure A.4.

2. **dxHeuristicsManager**: This is the manager class for all the heuristics defined inside ODE_MT. The code is reproduced in Figure A.5. As you can see, any heuristic has to be defined here and updated every frame. It also inherits from a callback class which allows it to be triggered every time one of the heuristic rules is fulfilled.
A.3. Heuristics

class dxClusterManager
{
private:
    class CWASNode *clusters;
    clusterChangeCallbackFunc* clusterChangeCallback;

private:
    dxClusteredWorldAndSpace *getCWAS2(dWorldID world, dSpaceID space);
    dxClusteredWorldAndSpace *addCWAS(dWorldID newWorld, dSpaceID newSpace);
    static dxClusterManager* mInstance;

public:
    dxClusteredWorldAndSpace* currentCWAS;
    static dxClusterManager* instance() { return mInstance; }
    dxClusterManager(clusterChangeCallbackFunc*);
    ~dxClusterManager();
    dxClusteredWorldAndSpace* getCWAS(dWorldID, dSpaceID);
    void removeCWAS(dWorldID world, dSpaceID space);
    void simLoop(bool);
};

Figure A.1: The dxClusterManager class

typedef void* voidFunc(int, void*);

struct threadInfo
{
    int tid;
    voidFunc* threadFunc;
    void* threadArgs;
};

class dxThreadManager
{
protected:
    int threadCount;
    struct threadInfo threadArgs[MAX_THREADS];
    voidFunc* threadFunc;

public:
    virtual void init(int numThreads, voidFunc threadFunc) = 0;
    virtual void simLoop() = 0;
    virtual void reinit(int numThreads) = 0;
    int getThreadCount() { return threadCount; }
};

Figure A.2: The dxThreadManager interface class
Appendix A. ODE_MT: Code & Concepts

class HeuristicAlertCallback;

template<typename T>
class dxHeuristic
{
    private:
        HeuristicState mState;

        T oldValue;
        T newValue;
        int frameCount;
        int frameCounter;

        int recordedSamples;
        T average;
        T speed;

        std::list<heuristicAlert> alertsList;

    public:
        dxHeuristic(T _value);
        ~dxHeuristic();

        void simLoop();
        void update(int _frameCount, T _value);
        void pause();
        void reset();

        T getNewValue() { return newValue; }
        T getOldValue() { return oldValue; }
        void addAlert(AlertType _alertType, class HeuristicAlertCallback
            * _alertFunc, T _value);

    };

Figure A.3: The dxHeuristic class
A.3. Heuristics

```c
struct heuristicAlert
{
    AlertType alertType;
    T threshold;
    HeuristicAlertCallback* alertFunc;
};

enum AlertType
{
    ALERTTYPE_FIXEDINTERVAL,
    ALERTTYPE_VALUECHANGE,
    ALERTTYPE_VALUEBELOWTHRESHOLD,
    ALERTTYPE_VALUEABOVETHRESHOLD,
    ALERTTYPE_SPEEDBELOWTHRESHOLD,
    ALERTTYPE_SPEEDABOVETHRESHOLD,
};

enum HeuristicState
{
    STATE_INIT,
    STATE_RESET,
    STATE_RUNNING,
    STATE_PAUSED,
    STATE_STOPPED
};
```

Figure A.4: The Heuristic Alert class and State types
Appendix A. ODE_MT: Code & Concepts

class HeuristicAlertCallback
{
public:
    virtual void alertCallback(dxHeuristic<int>* _heuristic) = 0;
};

class dxHeuristicsManager : public HeuristicAlertCallback
{
private:
    int userThreadCount;
    int recommendedThreadCount;
    int frameCount;
    int syncInterval;

    class dxClusterManager* clusterManager;
    class dxThreadManager* threadManager;

    class dxHeuristic<int>* objectCountHeuristic;
    class dxHeuristic<int>* clusterCountHeuristic;
    class dxHeuristic<int>* clusterCheckHeuristic;

    void alertFunc();
    void alertCallback(dxHeuristic<int>* _heuristic);

public:
    dxHeuristicsManager(int _numThreads, int _syncInterval, class dxClusterManager* _clusterManager, class dxThreadManager* _threadManager);
    ~dxHeuristicsManager();

    void reset();
    void simLoop(class dxWorld* _world, class dxSpace* _space);
};

Figure A.5: The dxHeuristicsManager class
In this appendix, we look at the overall performance of the sample worlds that come with Webots 7.3.1 and see how they fare in the old single-threaded version and the new multi-threaded versions of ODE. In the table below, the overall speedup in ‘Fast Mode’ in Webots was sampled once a second over a period of a minute, giving 60 samples which were then used to calculate the mean and standard deviation. These mean and standard deviations were used to plot the normal distribution of the speedup values and are superimposed for the 2 versions: ODE and ODE_MT. The red line represents the original version (ODE) and the blue line represents the new version (ODE_MT).

<table>
<thead>
<tr>
<th>World</th>
<th>Snapshot</th>
<th>Performance Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Darwin-Op</td>
<td><img src="image1" alt="Darwin-Op Snapshot" /></td>
<td><img src="image2" alt="Darwin-Op Performance" /></td>
</tr>
<tr>
<td>2. Pioneer3at</td>
<td><img src="image3" alt="Pioneer3at Snapshot" /></td>
<td><img src="image4" alt="Pioneer3at Performance" /></td>
</tr>
</tbody>
</table>
Appendix B. Webots: Then & Now

3. Bioloid

4. KHR

5. Youbot

6. Nao Demo
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7. Stewart Platform</td>
<td><img src="image7.png" alt="Image" /></td>
</tr>
<tr>
<td>8. Salamander</td>
<td><img src="image8.png" alt="Image" /></td>
</tr>
<tr>
<td>9. Shrimp</td>
<td><img src="image9.png" alt="Image" /></td>
</tr>
<tr>
<td>10. GhostDog</td>
<td><img src="image10.png" alt="Image" /></td>
</tr>
<tr>
<td>11. Surveyor</td>
<td><img src="image11.png" alt="Image" /></td>
</tr>
</tbody>
</table>
Appendix B. Webots: Then & Now

12. IPR Factory

13. Uneven Terrain

14. Gantry

15. Hoap2 Walking

16. Hexapod
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>17. E-puck Line Demo</td>
<td><img src="image" alt="E-puck Line Demo" /></td>
</tr>
<tr>
<td>18. iRobot Create</td>
<td><img src="image" alt="iRobot Create" /></td>
</tr>
<tr>
<td>19. Sojourner</td>
<td><img src="image" alt="Sojourner" /></td>
</tr>
<tr>
<td>20. Autonomous Vehicle</td>
<td><img src="image" alt="Autonomous Vehicle" /></td>
</tr>
<tr>
<td>21. Yamor</td>
<td><img src="image" alt="Yamor" /></td>
</tr>
</tbody>
</table>
Appendix B. Webots: Then & Now

Table B.1: Webots sample worlds and their performance
Bibliography


