Praktikumsbericht Parallele Programmierung, Sommer 2014

Genetischer Algorithmus entwirft Tower Defense Maze

Author: Sascha Schulz

Universität Hamburg, Matr-Nr.: 6434677, 2sschulz@informatik.uni-hamburg.de

Inhalt

Einleitung

Im Modul “Data Mining” wurde ich – parallel zum Theorieteil dieses Praktikums – auf Genetische Algorithmen aufmerksam. Mich faszinierte die Idee, eine Maschine oder ein Programm auf ein Optimierungsproblem anzusetzen, welches für den Menschen eher unzugänglich ist. Um mich auf den Genetischen Algorithmus und dessen Parallelisierung konzentrieren zu können, wählte ich ein triviales Beispiel, bei der die Lösungsbewertung auch dem Laien leichtfällt.

Im Computerspiel “Warcraft III” etablierten sich mit der Zeit Fun-Maps, in denen Monster von einem Spawnpunkt an ein Ziel gelangen möchten. Der Spieler baut nun Türme als Hindernisse, welche den kürzesten Weg vom Start zum Ziel verlängert. Durch einen möglichst langen kürzesten Weg haben die Türme so Zeit, die Kreaturen auf dem Weg durch das so entstehende Labyrinth (Maze) zu töten.

Für das Praktikum wurde dieses Szenario stark vereinfacht und auf die Maximierung des kürzesten Pfades reduziert. Türme haben nicht länger die Aufgabe Monster abzuschießen, sondern stellen ausschließlich ein Hindernis dar, werden daher auch im Weiteren als Wände bezeichnet. Ziel des Genetischen Algorithmus wird sein, aus Wänden und freier Fläche ein solches Maze zu konstruieren und zu bewerten, sodass im Laufe der Entwicklung immer bessere Konstruktionen (im Hinblick auf den kürzesten Weg) entstehen. Dabei interessiert mich besonders, wie nah der Genetische Algorithmus den mir bekannten und gängigen Konstruktionsansätzen kommt.

Das sequentielle Programm

Kernbestandteil des sequentiellen Programms ist neben dem Genetischen Algorithmus eine eigene Implementation des Bellmann-Ford-Algorithmus zum Finden des kürzesten Weges. Während dies für das sequentielle Programm durchaus unnötig kompliziert wirken könnte, macht es zum Zeitpunkt der Parallelisierung Sinn, da sich dieser sehr leicht parallelisieren und dezentralisieren lässt.

Modellierung

Die Spielfläche lässt sich durch ein Raster in Felder unterteilen. Ein jedes Feld kann entweder einen Turm bzw. eine Wand enthalten oder aber leer sein und somit als Weg genutzt werden – ein Feld kann folglich einen von zwei Zuständen annehmen. Kreaturen könnten sich über die freien Felder bewegen um das Maze zu passieren.

Es wird vereinfachend angenommen, dass sich Kreaturen ausschließlich orthogonal durch das Maze bewegen, da es die Umwandlung des Mazes in einen Graphen deutlich vereinfacht und abgeschätzt wurde, dass die grundsätzlichen Funktionsweisen davon nicht schwerwiegend beieinträchtigt werden.

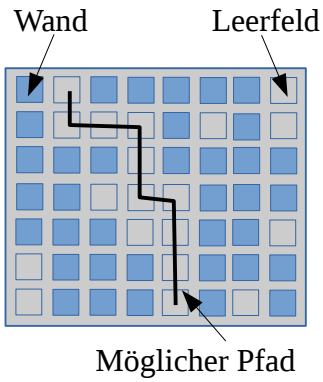

Illustration 1: Maze-Modell

Um den kürzesten Weg durch ein Labyrinth zu finden, lässt sich das Maze in einen Graphen umwandeln, in dem freie Felder einen Knoten darstellen und benachbarte freie Felder durch eine Kante verbunden sind. Die Länge des kürzesten Pfades durch das Maze ist so durch einen beliebigen Algorithmus für kürzeste Pfade in Graphen lösbar.

Der Genetische Algorithmus

Genetische Algorithmen werden im allgemeinen verwendet, um auf Basis programmatischer Regeln zur Bewertung bestehender (zufällig erschaffener) Lösungen und deren Kombination neue Lösungen zu erfinden.

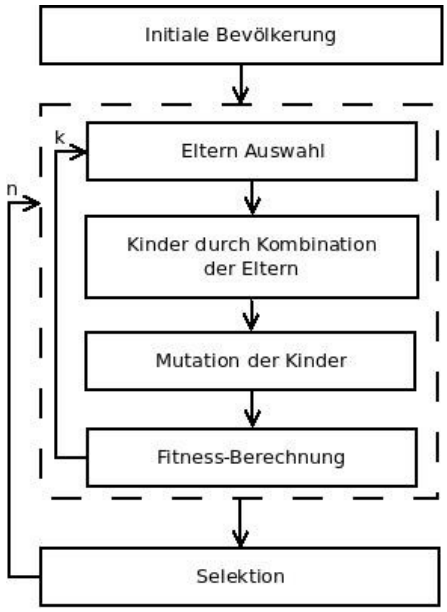

Illustration 2: Phasen des Genetischen Algorithmus mit k Elternpaaren und n Iterationen

Bewertet wird die sogenannte *Fitness* von potentiellen Lösungen, das heißt wie gut eine Lösung ist. Durch die Auswahl der Lösungen für die Rekombination (Fortpflanzung), sowie durch die Selektion von Lösungen für die weitere Berechnung, verbessert sich mit den Iterationen die Fitness der Bevölkerung. Dabei werden lokal gute Lösungen gefunden, globale Optima jedoch eher unwahrscheinlich erreicht.

Der für dieses Praktikum geschriebene Genetische Algorithmus geht dabei wie folgt vor: Initiale Mazes werden zufällig erzeugt. Die Fitness eines Mazes entspricht der Länge des kürzesten Pfades von einem beliebigen Feld der Oberkante zu einem beliebigen Feld der Unterkante. Eltern werden mit einer Wahrscheinlichkeit proportional zur Fitness ausgewählt und zu neuen Mazes rekombiniert – wobei beim Kind Mutationen auftreten können, bei denen der Zustand eines Feldes invertiert wird. Die Fitness von Kindern wird ebenfalls bewertet. Am Ende einer Iteration wird die überlebende Bevölkerung (mögliche Lösungen) selektiert. Hierbei wird *Elitism* verwendet, d.h. die Lösungen mit der höchsten Fitness bleiben erhalten.

Der Bellmann-Ford-Algorithmus

Der Bellmann-Ford-Algorithmus speichert für jeden Vertex den Wert des kürzesten Pfades und den Vorgänger. Zu Beginn haben alle Vertices der Oberkante die Distanz 0, alle anderen verfügen über keine Distanz (im Modell: Der Abstand ist unendlich).

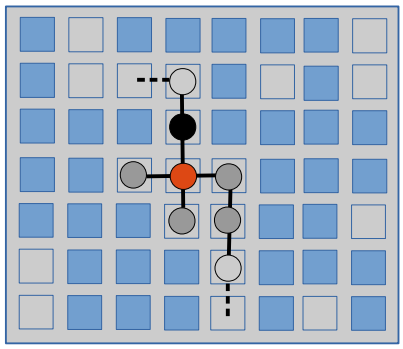

Illustration 3: Für jeden Vertex wird jeder Nachbar geprüft, ob ein kürzerer Weg vorhanden ist.

Für jeden Knoten werden alle direkten Nachbarn geprüft, ob unter ihnen eine kürzere Verbindung vorliegt. Wenn ja wird diese gewählt und entsprechend gespeichert. Dies wiederholt sich, bis ein optimaler Weg gefunden ist, im schlimmsten Fall sind dabei

alle Knoten aufgereiht. Da dies in den seltensten Fällen der Fall ist, bietet es sich an, den Algorithmus terminieren zu lassen, sobald bei einer Iteration für keinen Vertex eine Änderung vorgenommen wurde.

Als Letztes ist nur zu ermitteln, welcher Vertex der Unterkante über den kürzesten Pfad durch das Maze verfügt. Gibt es gar keinen Weg durch das Maze, sind ihre Werte seit der Initialisierung unverändert.

Implementierung

Der Genetische Algorithmus ist als Controller realisiert, welcher ausgelagerte Resolver verwendet, um einzelne Teilaspekte wie Auswahl von Eltern, Erzeugen von Kindern, Mutation, Fitness-Bewertung und Selektion vorzunehmen.

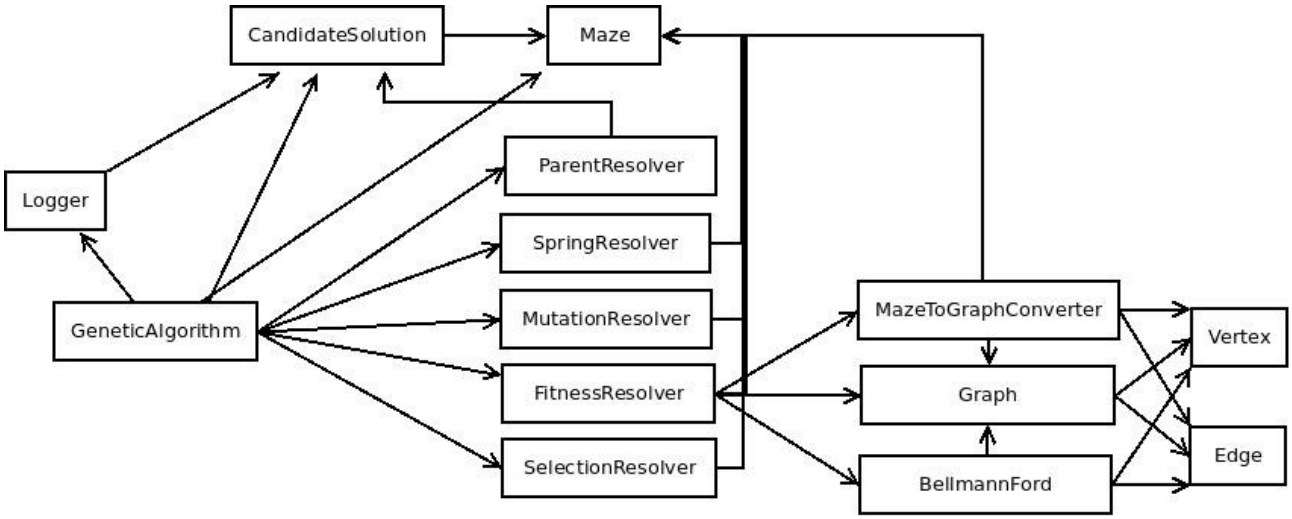

Illustration 4: Klassen der sequentiellen Implementierung und ihre Relationen

Der GeneticAlgorithm hält dabei ein *c++ std::set<CandidateSolution>* vor, welches die Bevölkerung repräsentiert. *Candidate Solutions* sind dabei genau genommen nur Container, welche das Tupel (Maze, Fitness) repräsentieren. Der grobe Ablauf des Controllers sieht wie folgt aus:

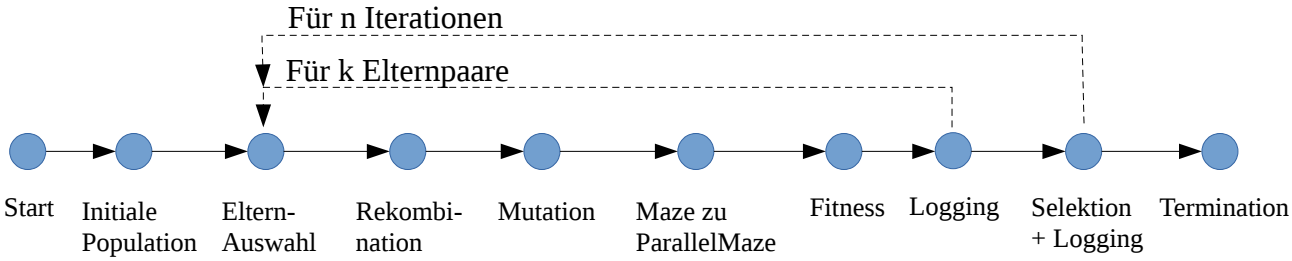

Illustration 5: Sequentieller Progammablauf

Der zentrale Abschnitt des Genetischen Algorithmus ist die Fitness-Berechnung. Diese setzt sich aus der Umwandlung des Mazes in einen Graphen und die Berechnung des kürzesten Weges von einem Vertex eines Feldes der Oberkante zu einem der Unterkante zusammen.

Da für das Modell ausschließlich orthogonale freie Felder von Belang sind, ist die Umwandlung denkbar trivial: Eine Schleife überprüft für jedes Feld reihenweise von oben links nach unten rechts, ob es sich um ein Leerfeld handelt – also passierbar ist. Wenn ja, wird ein Vertex erschaffen, der als ID die jeweilige Feldnummer erhält. Weithin wird geprüft, ob für das Feld oberhalb oder zur Linken ein Repräsentant existiert. Wenn ja, werden diese durch eine Kante verbunden. Der so entstandene Graph wird anschließend in den Bellmann-Ford-Algorithmus gegeben.

Die Parallelisierung

Da mich im Modul “Algorithmen und Datenstrukturen” insbesondere dezentrale Algorithmen faszinierten, entschied ich mich den Fokus auf Parallelisierung mit MPI zu legen. Um den Zeitaufwand im Rahmen zu halten, wählte ich die Umwandlung des Mazes in einen Graphen und die anschließende Berechnung des kürzesten Pfades zu parallelisieren.

Parallelisierungsschema

Ich schätzte initial ab, dass die meisten Teilaufgaben des Genetischen Algorithmus verhältnismäßig schnell erfolgen können und wählte daher ein Master+Slave-Setup mit dem Prozess No. 0 als Master. Zusätzlich erhielt der Master die Kontrolle über das Logging.

Bei der initialen Vorstellung der Projekte ging ich davon aus, dass ich das Spielfeld für die Parallelisierung in quadratische Kacheln unterteile. Betrachten wir Prozesse als Knoten und notwendige Kommunikationen zwischen Prozessen als Kanten, stellt dies lediglich eine Faltung des Mazes dar, der resultierende Graph hat die Form eines 2D-Netzes. Innere Knoten müssten daher mit bis zu vier Nachbarn kommunizieren. Unterteile ich das Spielfeld hingegen in Streifen, erhalte ich eine 1D Struktur und jeder Prozess muss mit maximal zwei Nachbarn kommunizieren.

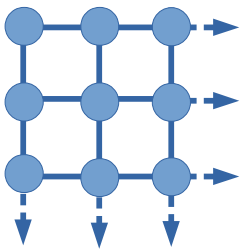

Illustration 6: 2D Prozess-Topologie

Illustration 7: 1D Prozess-Topologie

Da die eigene Implementierung des Bellmann-Ford-Algorithmus synchron arbeitet, bietet sich das 1D-Schema an, um durch die Kommunikation zwischen Prozessen nicht aufgehalten zu werden.

Paralleles Modell

Ein einzelnes Maze ist folglich in Streifen unterteilt. Sofern die Anzahl der Spielfeldreihen sich nicht gleichmäig auf die Prozesse aufteilen lassen, erhält der letzte Prozess den entsprechenden Rest als Zuschlag, da dieser entgegen den anderen Prozessen nur mit einem Nachbarn kommunizieren muss und der erste Prozess als Master bereits zusätzlich ausgelastet ist. Die Prozesse verfügen dabei über keinerlei Wissen über die Felder ihrer Nachbarn.

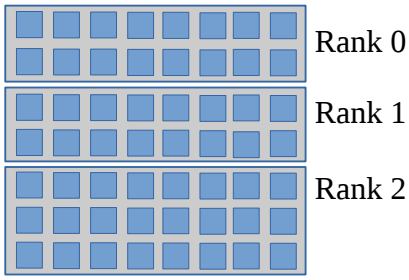

Illustration 8: Exemplarische Aufteilung eines Mazes zweischen drei Prozessen

Der Graph wurde für die parallele Version nicht modifiziert. Jeder Prozess erzeugt den Teilgraphen für seinen Abschnitt. Dabei erhält jeder Vertex die Nummer des Feldes, welches er *global* repräsentiert. Anschließend werden die Vertices für Felder der Ober- und Unterkanten an den jeweiligen Nachbarn kommuniziert, sodass dieser eine Verbindung über die Prozessgrenze hinweg herstellen kann. Es werden also Kanten “ins Nichts” erzeugt, denn der lokale Graph kennt den kommunizierten Vertex nicht, da er nicht für diesen zuständig ist. Allerdings ist somit die Kante und die Adjazenz des Ghost-Vektors bekannt, sodass der Bellmann-Ford-Algorithmus auf Basis dessen wie gewohnt operieren kann.

Paralleler Ablauf

Im Folgenden sind die Aufgaben des Master-Prozesses sowie eines exemplarischen Slave-Prozesses noch einmal aufgeschlüsselt:

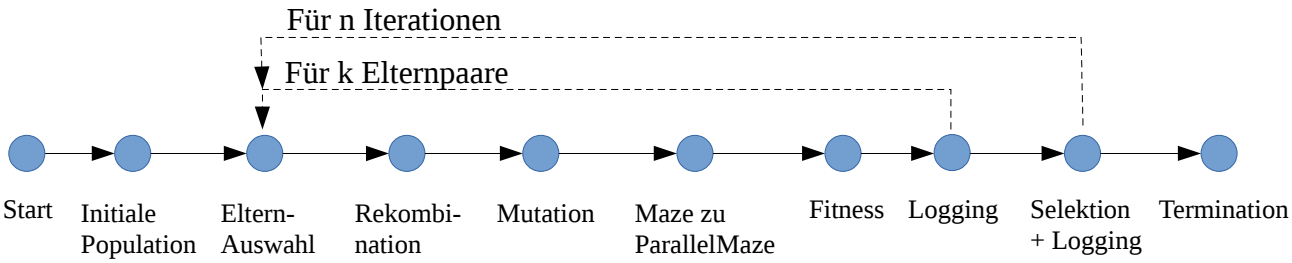

Illustration 9: Aufgaben des Masters

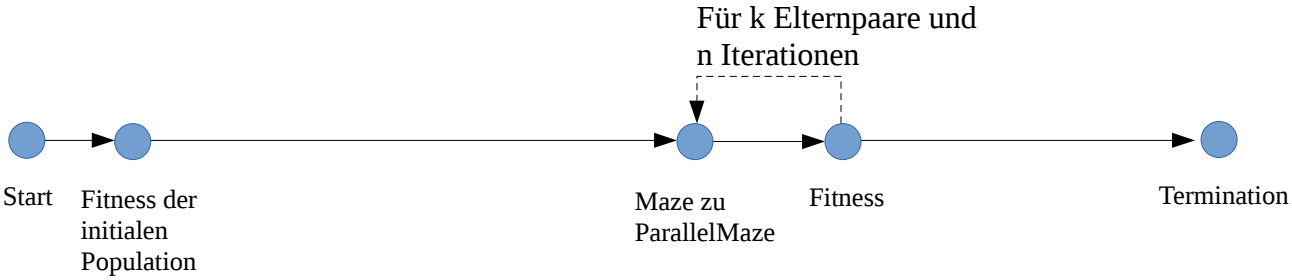

Illustration 10: Aufgaben eines Slaves

Erfahrungswerte mit MPI

Ich durfte bereits zu Beginn der Parallelisierung feststellen, dass es zwar sehr leicht fällt auf einem Blatt Papier Grenzen für die Zuständigkeit von Prozessen zu ziehen, es jedoch sehr viel schwerer fällt, das sequenzielle Programm entsprechend zu transformieren.

Es bereitete mir am Anfang einige Schwierigkeiten zu verstehen, wie ich Master und Slaves die Applikation durchlaufen lassen muss, sodass alle Prozesse zur richtigen Zeit im richtigen Abschnitt sind. Letztlich ließ sich dies sehr einfach durch Fallunterscheidung auf Basis des Ranks realisieren und erscheint rückwirkend als sehr trivial.

Eine weitere große Hürde stellte die Kommunikation von Nachrichten dar, deren Größe im Voraus nicht absehbar ist. Bei der Ausführung des Bellmann-Ford-Algorithmus werden pro Iteration die Werte für jeden Vertex der lokalen Ober- und Unterkante übertragen. Allerdings ist ungewiss, wie viele Werte übertragen werden. Ich löste dies, indem ich einmalig zu Beginn des Algorithmus die Größe der späteren Nachrichten kommunizierte.

Rückwirkend stellte ich fest, dass es deutlich einfacher gewesen wäre, einfach die bekannte Obergrenze (Maze-Breite) zu verwenden. Dennoch führte mich dies zu der Fragestellung, inwieweit bisher für derartige Probleme Lösungen abstrahiert und verglichen wurden. Insbesondere mit Blick auf die Patterns in der objektorientierten Programmierung, welche für viele ständig auftretende Probleme mögliche Lösungen aufzeigen, erscheint dies als naheliegend. Ein kurzes Suchen führte jedoch zu keinen Ergebnissen, vermutlich da ich noch nicht tief genug in der Materie stecke, um die richtigen Suchbegriffe zu kennen.

Parallelität als eigenständiger Belang

In der Diskussion um *Clean Code* in der objektorientieren Programmierung sind die Themen *Separation of Concerns* sowie die *Cohesion* von Methoden sehr zentral. Da ich mich auch in den vergangenen (Berufs-/Privat-/Studien-)Projekten stets dafür interessierte, wie sich Strukturen legen lassen und welche Auswirkungen dies hat, nutze ich die Gelegenheit um Auszuprobieren, was passiert, wenn ich Parallelität als eigenen Concern betrachte und daher parallele Ausprägungen einer Klasse als eigenständige Spezialisierung ansehe.

In der unten stehenden Illustration ist gut zu erkennen, dass ich zu jeder Klasse, die eine Abweichung zum sequentiellen Verhalten aufweist, jeweils eine spezialisierende Klasse schrieb, welche mit *“Parallel”* geprefixed ist. Ferner gibt das unten stehende UML-Diagramm Aufschluss darüber, welche Methoden mit der Spezialisierung überschrieben wurden und wie viele hinzugekommen sind. Ignorieren wir Konstruktoren, wurden neun Methoden überlagert, 18 Methoden und sechs Attribute kamen neu hinzu – primär, um den Austausch von Daten zu realisieren.

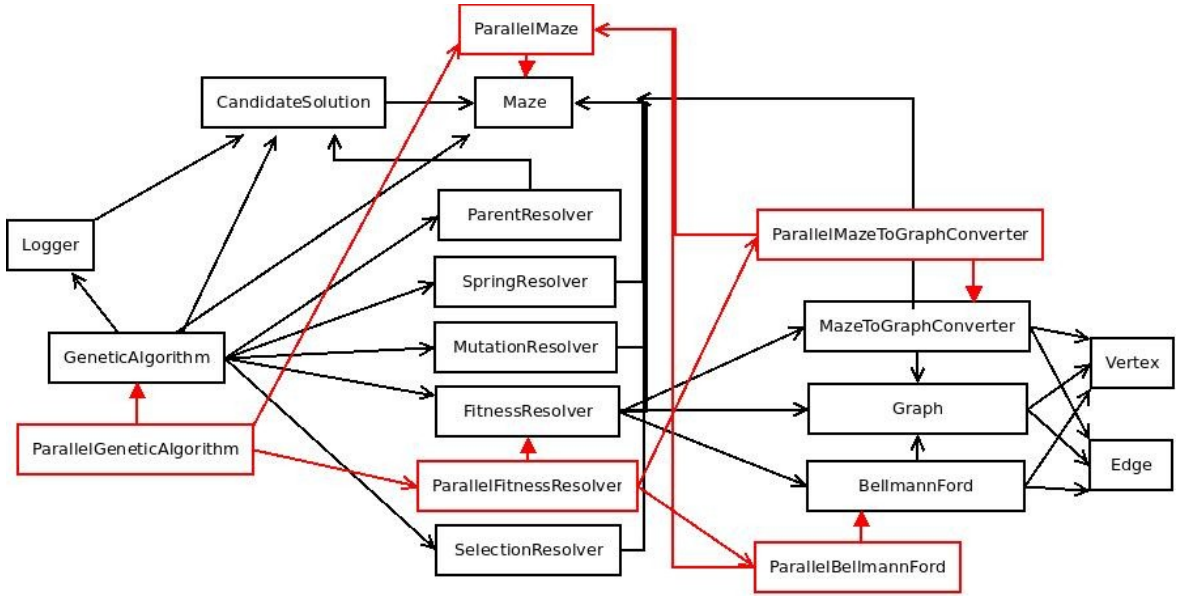

Illustration 11: Klassen des parallelen Programms. Parallele Aspekte sind als Spezialisierungen realisiert (hier rot hervorgehoben)

Obwohl sich dies recht gut liest und auch in Diagrammen angenehm übersichtlich erscheint, erschien mir der Ansatz letztlich als künstlich erzwungen. Im Kontext des Projektes war es tatsächlich hilfreich, sehr klar abgegrenzt zu haben, was paralellisiert ist und was nicht – insbesondere, da das sequentielle Programm jederzeit schnell wieder erzeugbar ist. Doch es erfordert, dass Methoden eine sehr große *Cohesion* aufweisen, sodass tatsächlich nur der durch die Parallelisierung betroffene Abschnitt überschrieben wird. Aus dem Blickwinkel der sauberen

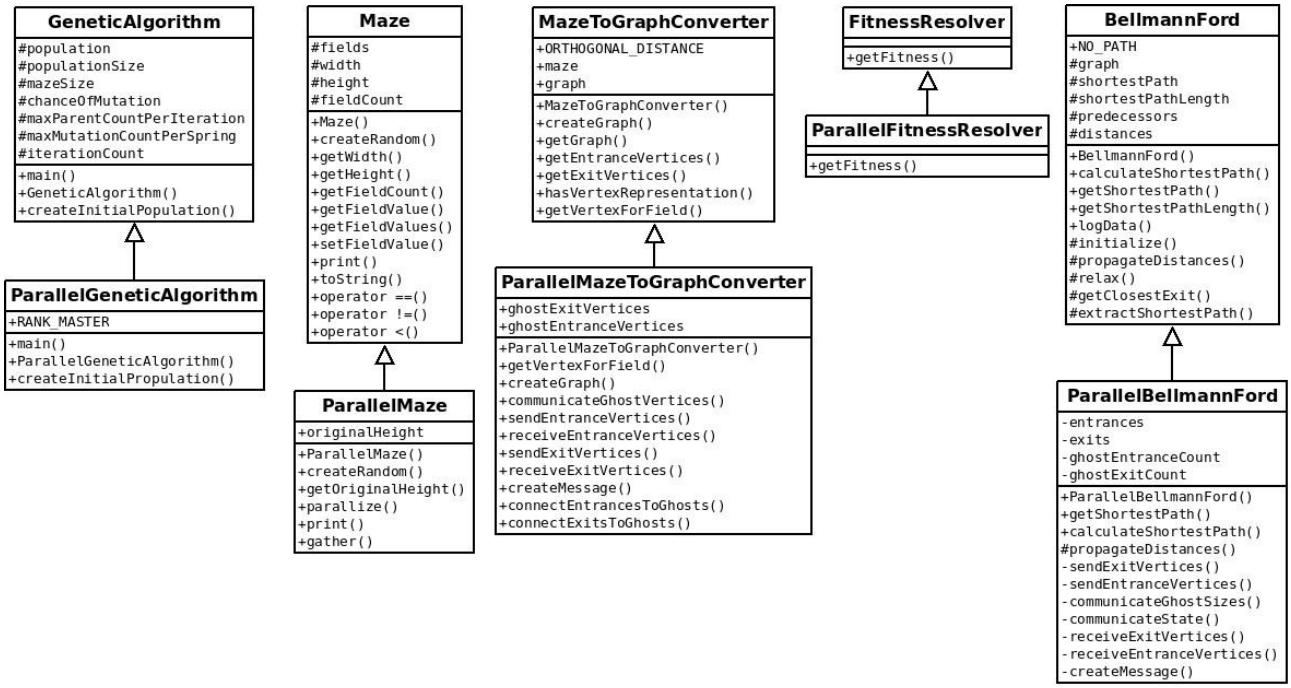

Illustration 12: Einblick in die Basisklassen und deren parallele Spezialisierung. Parameter und Typen wurden hier weggelassen um den Fokus rein auf Attribute und Operationen zu legen.

Programmierung mag das wichtig sein, allerdings steht dies (ebenso wie mein Separierungsansatz) der Optimierung des Programms entgegen: Es ist durch die starke Struktur nur schwer möglich, durch die Reihenfolge von Instruktionen und die Verschränkung unabhängiger Berechnungen die CPU optimaler Latenzen in der Kommunikation zu nutzen.

Werkzeuge und Entwicklungsumgebung

- mpi++ / g++ als Compiler
- Eclipse CDT als IDE und zum sequentiellen debuggen
- gtest für automatisierte Tests
- MPICH2 als MPI-Implementation in C++
- GDB zum parallelen debuggen
- Valgrind zum Auffinden eines Segmentation Faults
- d3.js zum rendern der Visualisierung

Testing

Um die Korrektheit der Implementierung zu überprüfen, wurden Tests im gtest Environment geschrieben. Mitunter fragte ich mich während der Implementierung, wie viel Debugging und Beten wohl notwendig wären, würde ich diese Tests nicht geschrieben haben – und dabei handelt es sich nicht einmal um viele Abhängigkeiten oder viel Code. Dennoch gaben sie mir eine gute Gelegenheit, meine Annahmen über Funktionsweisen zu bestätigen.

Es interessierte mich insbesondere, wie sich das Testen in der parallelen Umgebung verhält. Es stellte sich heraus, dass es grundsätzlich wenig Unterschiede gibt. Es kommt lediglich eine zusätzliche Dimension für *Expectations* hinzu, da die Annahmen oftmals an den Prozess gebunden sind. Durch Fallunterscheidungen auf Basis des MPI ranks lässt sich dies jedoch einfach lösen.

Fazit

Inhaltlich hat mich das Thema sehr interessiert und ich würde gern deutlich mehr Zeit investieren, um mich mit dem Thema Parallelisierung weiter zu befassen, MPI noch intensiver zu verwenden und auch mit OpenMP Erfahrungen zu sammeln.

Auch ohne bisher vorliegende Analysen konnte ich feststellen, dass die Fitness-Berechnung mit dem parallelen Programm deutlich schneller wurde.

Rückblickend bin ich mit meiner Ausarbeitung für das Praktikum mäßig zufrieden. Durch den Umfang der sequentiellen Implementierung bin ich schließlich mit dem eigentlichen Fokus, der parallelen Optimierung, stark in Bedrängnis geraten. Da mir diese jedoch sehr wichtig war, nahm ich mir die Zeit – worunter schließlich die Ausarbeitung und das User Interface litt. Auch hatte ich kaum Zeit, mich inhaltlich mit dem von mir geschaffenen Programm auseinanderzusetzen, um mit der Wahl möglichst guter Parameter den Genetischen Algorithmus gute Lösungen finden zu lassen. Folglich: Meine Projektplanung war leider unzureichend und ich sollte dies in künftigen Arbeiten stark ausbauen.