Ausarbeitung

1 - Problemstellung

Im Rahmen des Praktikums Parallele Programmierung (auch PAPO) wurde uns die Aufgabe gestellt ein eigenes Programmierprojekt zu erstellen wobei wir ein Programm schreiben sollten welches zuerst linear ausgeführt werden kann und im späteren Verlauf des Projektes parallelisiert werden sollte. Für ein solches Projekt wird also ein parallel ausführbarer Algorithmus gesucht welcher ein Problem löst. Probleme dieser Art gibt es ausreichend, so zum Beispiel die Berechnung von Verkehr in einer Stadt oder die Berechnung mehrere Partikel in einem physikalischen System. Als Grundlage für unser Problem wollten wir ein Problem aus der Kategorie der Zustandsraumberechnung wählen. Zustandsraum bezeichnet dabei die Menge aller Zustände welche ein System annehmen kann. Solche Probleme treten in vielerlei Gebieten auf aber vor allem im Bereich der Spiele. Das wohl bekannteste Beispiel hierfür wäre Schach, wobei der Zustandsraum nahezu unendlich ist. Auf der Suche nach einem ähnlichen Problem stießen wir auf das Spiel Vier Gewinnt welches für uns im Folgenden die Grundlage für unser zu parallelisierendes Programm bietet.

Bei Vier Gewinnt, auch Die Verflixten Vier, Connect Four oder Captains Mistress genannt, handelt es sich um ein weit verbreitetes Brettspiel für 2 miteinander konkurrierende Spieler, aus der Rubrik der Strategiespiele. Dabei ist das Ziel des Spieles eine ununterbrochene Linie von 4 Spielsteinen der eigenen Farbe zu kreieren Dies kann vertikal, horizontal wie auch diagonal geschehen. Hierbei wird normalerweise auf einem 7x6 Spielfeld gespielt wobei 7 die Breite und 6 die Höhe des Spielfeldes beschreiben.

Der Spielverlauf ereignet sich indem die Spieler abwechselnd einen Stein der eigenen Farbe in das Spielfeld fallen lassen. Dabei nimmt dieser Stein immer den untersten freien Platz der Linie ein. Sollten alle Felder besetzt und noch kein Sieger ermittelt worden sein, endet das Spiel unentschieden.

Bei Vier Gewinnt, handelt es sich zudem um ein Spiel mit perfekter Information. Zudem ist der Zustandsraum endlich. Auch gibt es schon eine bekannte Lösung für das Spiel welche von James D. Allen aufgestellt wurde. Die Regeln die sich dabei ergaben sind wie folgt.

Der erste Spieler kann das Spiel gegen eine perfekte Verteidigung gewinnen, wenn er in der mittleren Spalte beginnt. Beginnt er in der Spalte links oder rechts daneben, endet das Spiel bei beiderseits perfektem Spiel unentschieden.

Wirft er aber seinen ersten Stein in eine der vier restlichen Spalten verliert er gegen einen perfekten Gegner. Für uns ergab sich nun als Aufgabe eine perfekte KI für das Spiel zu erstellen. Diese sollte in der Lage sein mit maximaler Wahrscheinlichkeit gegen einen menschlichen Spieler zu gewinnen.

2 – Lösungsansatz

Um eine KI zu entwickeln welche das von uns gestellte Ziel erreicht, haben wir uns entschieden den gesamten Zustandsraum des Spieles auszurechnen. Dabei wird zunächst auf das Platzieren des ersten Steines des menschlichen Spielers gewartet um danach den gesamten Zustandsraum zu berechnen. Das der menschliche Spieler beginnt hat hierbei zweierlei Relevanz. Zum einen verkleinert dies den Zustandsraum zur Berechnung ungemein. Auf einem freien Feld gibt es für den ersten Spielzug 7 verschiedene Möglichkeiten. Danach ergibt sich der Zustandsraum ungefähr aus 3 hoch den restlichen leeren Feldern. Somit unterscheidet sich der Zustandsraum, wenn wir schon einen gesetzen Stein haben von 42^3 zu 41^3. Dies entspricht ca 7,3*10^19 Zuständen bzw.

verkleinert den Zustandsraum um den Faktor 3. Der zweite Grund leitet sich aus der uns bekannten Lösung ab. Darin wird festgestellt dass der erste Spieler, wenn er perfekt spielt, immer gewinnt wenn er seinen ersten Stein in die Mitte des Feldes wirft. Dies würde bedeuten, ließen wir die KI den ersten Stein setzen würde diese ihren ersten Stein immer in die mittlere Spalte legen und wäre unschlagbar. Wir wollen dem menschlichen Spieler aber eine minimale Siegchance lassen.

Im Folgenden wollen wir den Algorithmus, welchen wir zur Lösung dieses Problems verwenden etwas genauer erläutern.

Zunächst benötigen wir eine Repräsentation des Spielfeldes um die einzelnen Zustände in unserem System festhalten zu können. Dafür haben wir einen eigenen Variablen-Typen definiert welcher sich Gameboard nennt. Ein Gameboard besteht aus einer Zusammensetzung mehrerer elementarer Datentypen. Zum einen gibt es die Konstanten BoardWidht und BoardHeight welche die Höhe und Breite des Spielfeldes widerspiegeln Zudem besitzt ein Gameboard eine Vielzahl von Methoden. Da diese im Verlaufe des Algorithmus von hoher Wichtigkeit sind sollen diese hier einmal im folgenden genauer erklärt werden..

Die calculateHash Funktion bietet eine Möglichkeit den Hashwert eines Gameboardes zu berechnen. Dadurch haben wir einen Wert mit welchem wir eindeutig Gameboards voneinander unterscheiden können.

UpdateSearchStatus ist eine Hilfsmethode, welche eingesetzt wird um sich zu merken wie viele Steine in einer Reihen vom gleichen Spieler liegen.

CheckRowsForMatch, checkColumnsForMath sowie checkForwardDiagonalsForMatch und checkBackwardsDiagonalsForMatch übernehmen dann die Prüfung für ein Spielfeld, ob einer der Spieler bereits 4 aneinanderliegende Spielsteine besitzt.

Die BoardisFull Methode gibt uns dann Auskunft darüber, ob ein Unentschieden entstanden ist. Mit Hilfe dieser Funktionen kann man schließlich über die Methode updateGameFinishedStatus herausfinden in welchem Zustand sich das Spielfeld zu diesem Zeitunkt befindet.

Die put Methode ist eine der Methoden welche am häufigsten aufgerufen wird. Diese erlaubt uns ein neues Gameboard zu erzeugen, welches ein Nachfolgegameboard des eigentlichen Gameboardes ist, indem in eine der Spalten ein Spielstein des Spielers gelegt wird welcher derzeit als nächstes am Zug ist.

Nun wollen wir im Folgenden den kompletten Zustandsraum ausrechnen um der KI einen Pfad zum Sieg zu ermöglichen. Dabei wollen wir uns der Struktur eines Baumes bedienen um den Zustandsraum darzustellen und den optimalen Pfad einfach ermitteln zu können. Hierfür benötigen wir wieder einen neuen Datentypen um die Knoten eines Baumes repräsentieren zu können.

Jeder dieser Knoten hat eine Menge von Successores also Nachfolgeknoten. Zudem gibt es einen SuccessorsCount welcher die Anzahl der Nachfolgeknoten festhält. Außerdem wird ein Gameboard festgehalten welches einen Spielzustand repräsentiert. Als letzter Wert wird in einem solchen Knot noch die winPercentage in einem Wert von 0 bis 100 festgehalten.

Mit diesen Datentypen ist es uns nun möglich die Aufgabe zu bewältigen.

Die sequentielle Lösung dieses Problems beruht auf einer relativ "alten" Version und unterscheidet sich stark von unserem späteren parallelisiertem Projekt.

Zunächst nehmen wir dabei für unseren Algorithmus den ersten Spielstein des menschlichen Spielers entgegen und generieren uns daraus unser erstes Gameboard. Dieses wird unser erster Knoten welchen wir für unseren Baum nutzen und ist somit die Wurzel des Baumes. Bei der seriellen Bearbeitung unseres Algorithmus wird nun für jede Lane, also jede Spalte unseres Gameboards einmal die put Methode aufgerufen, was bedeutet wir erzeugen eine Menge von Nachfolgezuständen. Diese besitzen nun genau einen Spielstein mehr als zuvor wobei jede Möglichkeit diesen Spielstein einzusetzen berechnet wurde. Diese Zustände werden nun auf ihren Zustand getestet, d. h. Ob das Spiel schon gewonnen, verloren, unentschieden oder noch nicht beendet ist. Aus dieser Menge neuer Spielzustände bzw. Gameboards werden nun Knoten generiert. Dabei werden diese Knoten mit dem vorherigen Knoten verknüpft um so eine Baumstruktur zu generieren. Gleichzeitig wird hierbei aber auf Duplikate überprüft um somit den verbrauchten Speicher und die Menge an Knoten die wir im nächsten Schritt weitergeben so gering wie möglich zu halten und nicht unnötig gleiche Zustände mehrfach ausrechnen. Diese Menge neuer Knoten wird nun in einem weiteren Durchlauf genutzt um über die put Methode alle nachfolgenden Zustände zu berechnen. Dabei werden aber nur diejenigen Knoten verwendet welche in einem noch nicht abgeschlossenen Zustand sich befinden. Anschließend werden diese wieder überprüft und zu Knoten verarbeitet und verknüpft. Dieser Prozess des Baumbauens wird durchgeführt bis es keine Zustände mehr gibt welche berechnet werden können, also der gesamte Zustandsraum ausgerechnet wurde.

Im nächsten Schritt des Algorithmus steht es nun an der Reihe aus dem Baum in welchem uns nur in den Blättern die Gewinnchance des Computers bekannt ist einen Baum zu generieren in welchem uns für jedes Blatt die zu erwartende Gewinnchance bekannt ist. Dafür nehmen wir uns alle Knoten welche im vorletzten Zug berechnet wurden und berechnen deren Winchance. Dies wird erreicht indem wir beim Zug des Spielers den Minimalwert der Nachfolger-Gewinnchancen nehmen und beim Computerzug den Maximalwert. Danach nimmt man sich die Knoten eines Durchlaufs eher und so weiter bis man bei der Wurzel angekommen ist.

Nun ist der gesamte Zustandsraum berechnet und wir sind uns für alle Züge darüber im klaren welcher der Optimale ist. Mit diesem Wissen kann nun die KI Entscheidungen treffen welche mit extremer Wahrscheinlichkeit zum Sieg führen werden.

Daraus lässt sich nun nach jedem Zug des Spielers eine Entscheidung treffen ohne weitere Berechnungen anfertigen zu müssen. Das Berechnen dieses Prozesses ist aber für einen einzelnen Prozess viel zu langsam. Beim Testen verschiedener Größen von Spielfelder haben sich dabei folgenden Werte ergeben:

—

—

Hieran sieht man mehr als ausreichend das ein Spielen praktisch nicht möglich ist bei einem sequentiellen Algorithmus. Mit dieser Erkenntnis wird es nun klar, warum wir eine Parallelisierung dieses Prozesses vornehmen müssen.

3 – Parallelisierungsschema

Das Parallelisierungsschema ist nun aufgrund von vielerlei Optimierungen stark verändert wurden.

Parallelisiert wurden in unserem Algorithmus sowohl der Algorithmus um den Baum zu bauen wie auch der Algorithmus um die Gewinnchancen zu berechenen.

Aufgrund von einer Vielzahl an Optimierungsprozessen gibt es zweierlei parallele Programme. Eines davon ist darauf ausgelegt mit der RAM zu arbeiten und löst für kleine Felder die Aufgabe recht schnell, dafür können größere Felder nicht berechnet werden da die RAM sonst voll läuft. Der andere Algorithmus benutzt direkt den Hauptspeicher indem er in FILES schreibt um die Berechnungen durchzuführen. Dies dauert zwar länger ermöglicht uns aber das Berechnen größerer Felder.

Im folgenden soll erklärt werden wie der Algorithmus funktioniert welcher den Hauptspeicher nutzt. Dabei wird der gesamte Code erklärt. Dies dient als detaillierte Dokumentation. Anschließend wird in einem TL:DR das Schema noch einmal kurz in der reinen Theorie zusammengefasst.

Zunächst nehmen wir dabei für unseren Algorithmus den ersten Spielstein des menschlichen Spielers entgegen und generieren uns daraus unser erstes Gameboard. Danach wird ein Knoten erzeugt, welcher als Ausgangsknoten für unseren Baum dient. Dieses erste Gameboard und der erstellte Knoten werden dann an die Methode weiter gegeben, welche sich um den Bau des "Zustandsbaumes" kümmert.

Beim Aufruf der Methode buildParallelTree werden zunächst einige Variablen wie der Rank gesetzt und anschließend eigene MPI_Datatypes definiert um diese im Verlauf des Prozesses nutzen zu können. Dabei gibt es die Datentypen MPI_GAMEBOARD welches zum Schicken eines Gameboardes benutzt wird, MPI_GAMEBOARD_ARRAY welches und das Schicken von ganzen Arrays an Gameboards ermöglicht und MPI_WINCHANCE_ARRAY welches wir später benötigen um die richtige Winchance zu berechnenen und dabe mehrere Winchance Werte für Knoten enthält.

Danach wird im Master Prozess ein File erstellt im write Binary Modus. Dies hat den Effekt, dass wenn die Datei schon vorhanden ist diese überschreiben wird und wenn nicht kann diese direkt neu angelegt werden. Der Binär Modus führt dazu, dass insgesamt schneller geschrieben und gelesen werden kann und es spart zusätzlich Speicherplatz. Dieses File wird im späteren Verlauf genutzt da wir die Festplatte nutzen um unsere Berechnung auszuführen.

Danach wird einmal setStartTurn ausgeführt um zu schauen in welchem Zustand sich das Spielfeld befindet. Dabei erhalten wir einen Wert, ob auf dem Spielfeld schon ein Stein gesetztz ist oder nicht. Zusätzlich wird noch der TurnCounter initialisiert und setzt für alle Züge welche wir nicht mehr berechnen, da diese schon geschehen sind bis zum jetzigen Spielstand, die TurnSize auf 0. Diese Methode wird benötigt um z.B.ein Spielfeld berechnen zu können welches nicht mehr im Startzustand ist. (Z.B. Wenn man schon eine weile gegen Jemanden spielt und am Verlieren ist kann man einfach das bis dahin enstandene Spielfeld übergeben und darauf rechnen lassen). Anschließend können wir aus diesen Werten berechnen welcher Spieler der erste Spieler war und somit immer wissen welcher Spieler in den von uns nicht berechneten Zügen am Zug war.

Danach wird initializeQueues aufgerufen. Dabei werden wieder Variablen initiallisiert und in den currentGameboardBuffer wird das erste Gameboard geladen. Anschließend wird ein sogenanntes CurrentGameboardFile erstellt in welches wir direkt unser erstes Gameboard abspeichern und ein NextGameboardFile. Diese Files werden später benutzt um die berechneten Gameboards abzuspeichern.

Als Nächstes folgt der Aufruf makeFirstTurn. Wir öffnen das SaveFile im AppendBinary Modus um weitere Daten in das File zu schreiben. Dann wird der erste erstellte Knoten in das File geschrieben. Es wird dann die TurnSize für den aktuellen Zug gesetzt, welcher in diesem Fall 1 ist. Anschließend wird das TurnDisplacement gesetzt. Diese Variable wird verwendet um abzuspeichern wie viele Knoten bisher vor dem ersten Knoten des derzeitigen Zuges berechnet wurden. Nun wird der Resultbuffer initialisiert und an die calculateBoardSuccessors weitergeben. Diese Methode ist in der

Lage für eine Menge von Gameboards alle NachfolgeBoards zu berechnen. Die Methode lässt dabei nur das Erstellen von validen Gameboards zugelassen. Für einen invaliden Versuch ein Gameboard zu erstellen, wenn z. B. die Lane in welche wir versuchen einen Stein zu werfen voll ist, wird ein Gameboard zurück gegeben welches in allen seinen Werten eine 0 hat. Somit ist auch der NextPlayer auf 0 gesetzt, womit wir diese invaliden Boards Problemlos von validen Boards unterscheiden können. Diese ausgerechneten Gameboards werden dann auf der Festplatte abgespeichert im NextGameboardsFile. Gleichzeitig wird während dessen der NextGameboardsCount aktualisiert anschließend wird Speicher im RAM wieder freigegeben und auf den nächsten Zug gestellt. Dies bedeutet, dass das CurrentGameboardsFile gelöscht werden kann und das NextGameboardsFile an dessen Stelle tritt. Das Turndisplacement wird neu berechnet und der Turncounter inkrementiert

Dieser Ablauf entspricht dem ersten Zug. Dieser ist noch vollständig sequentiell da bisher noch keine Daten zur Parallelisierung erstellt wurden. Nun wird noch der Wert der Variable firstplayer an alle Prozesse geschickt damit jeder der Prozesse diese Information besitzt. Damit ist die Initiallisierungsphase abgeschlossen und der eigentliche parallele Algorithmus wird in Gang gesetzt über den Aufruf der Funktion calculateTurns.

Zunächst werden eine Menge an Variablen gesetzt.

TreeFinished gibt an, ob das Erstellen des Baumes beendet wurde.
Der TaskSendBuffer speichert die CurrentGameboards für welche als Nächstest die NextGameboards ausgerechnet werden sollen.
Der TotalSendCount merkt sich wie viele Gameboards im aktuellen Zug berechnet werden müssen.
SendCnts, Displacements, taskRecvBuffer und recvCnt sowie ResultSendBuffer und resultRecvBuffer werden später für das verschicken über MPI benötigt.

Nun befinden wir uns in einer Schleife bis wir den Baum vollständig berechnet haben.
Der Master kann nun den totalSendCount berechnen. Dabei werden die CurrentGameboards überprüft welcher noch berechnet werden muss und welches dieser Gameboards schon genutzt wird. Dabei wird nur der NextPlayer Wert aus dem File geladen, da dies zur Überprüfung genügt und die genutzte Menge an RAM minimal hält. Calculate SendCounts berechnet dann die sendcnts Werte und displacements.

Nun können wir die sendcnts und den totalsendcnt an alle Prozesse verschicken damit diese den benötigten Speicherplatz allokieren können. In der Methode calculateTurnSteps wird nun berechnet in wie vielen Schritten der aktuelle Zug berechnet werden muss. Das Zerlegen eines Zuges in verschiedene Schritte liegt darin begründet das es aufgrund des limitieren RAMS nicht möglich ist unendlich große Züge auf einmal zu berechnen. Somit wird dieses Problem umgangen. Dabei kann jeder Prozess über eine "pessimistische" Schätzung ausrechnen wie viel RAM er wahrscheinlich benötigen wird für die Berechnung. Dann versuchen alle Prozesse diesen Speicher zu allokieren. Sollte einer dieser Prozesse Fehlschlagen wird die Menge an Schritten in welcher ein Zug berechnet wird erhöht von dem initialwert 1 bis alle Prozesse in der Lage sind ihren Speicher zu allokieren. Nun wird noch einmal turnSize auf dem derzeitig korrespondierenden Wert auf 0 initialisiert.

Nun begibt sich das Programm in eine For schleife welche sozusagen die einzelnen Schritte simuliert für diesen Zug. Dafür muss zuerst der CurrentStepSendCount auf 0 gesetzt werden. Dieser Wert entspricht der Anzahl an Gameboards welche in diesem Schritt an die Prozessoren verteilt werden müssen um die NachfolgeGameboards auszurechnen. Dafür speichert zuerst der Master die currentGamebaords in den dafür vorgesehenen Buffer. Anschließend wird das Senden vorbereitet wobei der currentStepSendCount aktualisiert wird und der Sendbuffer wird dann vorbereitet indem nur die zu versenden Gameboards gebuffert werden. Dann müssen wieder die SendCounts und

Displacements neu berechnet werden. Nun können diese Counts Gescattert werden. Anschließend wird der recvBuffer allokiert. Jeder Prozess erfährt also wie viele Gameboards er erhalten wird und anschließen werden die Gameboards versendet. Nun wird ersteinmal im Master der TaskSendbuffer freigegeben. Nun kann der resultSendBuffer allokiert werden. Anschließend werden wie schon bekannt über die Methode calculateBoardsSuccessors die NachfolgeGameboards berechnet. Während der Berechnung kann sobald der Master fertig ist die Methode addCurrentGameboardsTurn aufrufen.

Diese beginnt damit, dass für die gerade versendeten Gameboards Knoten angelegt werden. Dafür muss zunächst der Speicherplatz für diese Knoten allokiert werden. Dann muss die StepSize initialisiert werden. Diese Variable gibt an wie viele Knoten in diesem Step gespeichert wurden. Dies wird später benötigt um uns zu merken an welche Stelle wir in den succesorKnots Buffer die neuen Knoten speichern müssen. Der danach berechnete SuccessorIndexIndex ist eine Hilfsvariable die wir später für das Setzen der Successorindexe benötigen. Das SaveFile in welchem sich alle Knots befinden wird nun geöffnet um ein Seek darauf auszuführen. Dadurch wird der Cursor im File auf den ersten Index gesetzt im letzten Zug der bisher noch nicht gesetzt wurde.

Die folgende For Schleife läuft einmal für jedes gebufferte Gameboard. Der successorGameboardIndex gibt den Index des derzeitigen Knoten an welcher gespeichert werden soll. Wenn der Knoten nicht gespeichert wird aufgrund eines invalid Gameboards so wird dieser Wert auf -1 gesetzt, weshalb er auch so initialisiert wird. Wenn das Gameboard valide ist dann wird der Index gesetzt auf die TurnSize der vorherigen Steps plus die aktuelle Stepsize gesetzt. Danach wird der Pointer der auf dem nächsten Knoten in Successorknots zeigt zwischengespeichert und wenn das Gameboard gefinished ist, wird die Winchance gesetzt. Anschließend muss nur noch die StepSize inkrementiert werden.

Nun wird der Index ins SaveFile geschrieben an der Stelle wo vorher der Cursor extra platziert wurde. Falls in der folgenden Kalkulation eine 0 berechnet wird muss der Cursor zum ersten SuccessorIndex des nächsten Vorgängerknotens gesetzt werden.

Damit haben wir für alle gebufferten Gameboards im vorherigen Zug die Indexe der aktuellen Knots gesetzt und diese initialisiert. Nun muss die Turnsize des aktuellen Zuges aktualisert werden und die SuccesorKnots welche gerade berechnet wurden werden abgespeichert.

Zurück in die Hauptmethode. Hier können wir nun den taskRecvBuffer freigeben. Als Nächstes wird für den Master der ResultRecvBuffer allokiert und die Results zusammengesammelt. Dabei sind im RecvBuffer GameboardArrays. Dabei sind die Nachfolger des TaskSendBuffer[i] exakt die Werte des GameboardsArrays im RecvBuffer[i]. Nun werden diesem Empfangen Gameboards wieder über die bereits bekannte Methode saveNextGameboards gespeichert und der resultRecvBuffer kann freigegeben werden.

Als Nächstes wird davon ausgegangen das der Tree fertig gebaut wurde. Dann öffnen wir erneut das nextGameboardsFile und lesen nacheinander einzeln die Gameboards aus und es wird überprüft, ob alle Felder invalide oder zueende gespielt ist. Sollte ein Feld gefunden werden welches dem nicht entspricht wird Treefinished wieder auf 0 gesetzt und die Schleife verlassen. Dann können wir wieder NextTurn aufrufen. Sollte es bis dahin dazu gekommen sein das der Baum fertig gebaut ist muss noch einmal "manuell" auf den "leaves" des Baumes addCurrentGameboardsTurn aufgerufen werden um die Indexe des vorletzten Zuges über unsere Methoden zu berechnen und Knoten zu den letzten Gameboards abzuspeichern. Das Displacement für den letzten Zug wird gesetzt und die Successorindizies des letzten Zuges werden alle auf -1 gesetzt. Danach können wir die beiden GameboardFiles löschen um wieder mehr Speicherplatz freizugeben. Abschließend muss nur noch einmal der treeFinished Status an alle Prozesse verteilt werden damit am Ende alle Prozesse

gemeinsam die Whileschleife verlassen können. Weiterhin werden noch der Speicherplatz für die sendCnts und die Displacment freigegeben.

TL;DR:
Der Vorgang in seiner Theorie ohne zu sehr auf das Technische einzugehen wäre wie folgt:
Es wird für die Spielfelder aus dem aktuellen Zug die Nachfolger berechnet. Dies geschieht indem an jeden Prozess Spielfelder geschickt werden, für welche die Nachfolger berechnet werden. Sollte es dabei dazu kommen, dass der RAM zu klein ist um alle Spielfelder in einem Prozessor mit einmal zu berechnen wird die Berechnung in mehrere Schritte aufgeteilt. Die ausgerechneten Nachfolger werden dann beim Master gesammelt. Dieser speichert diese dann ab. Dann werden für die Spielfelder aus dem aktuellen Zug Knoten angelegt. Die Knoten werden mit den Knoten aus dem vorherigen Zug vernetzt. Abschließend werden die Knoten abgespeichert und die neuen Spielfelder geladen. Der Vorgang wiederholt sich dann bis keine Nachfolger mehr berechnet werden können.

Nun können wir die Winpercenate berechnen.
Wieder folgt zuerst eine genaue Beschreibung der Funktionsweise des Algorithmus und dannach eine TLDR Version, welche den technischen Teil auslässt.

Zu Beginn werden zwei Queues initialisiert, beide mit den Werten Null. Danach wird wieder für jeden Prozess der recvCnt sendCnts die displacements der taskSendBuffer sowie der taskRecvBuffer gesetzt. Anschließend wird der turnCounter um 1 dekrementiert und befindet sich somit im vorletzten Zug. Anschließend wird dieser Wert an alle Prozessoren verteilt.

Nun folgt eine Vorschleife welche rückwärts alle Züge durchgeht. Also vom Vorletzten bis zum Ersten. Dabei wird zuerst für den taskSendBuffer Speicherplatz allokiert. Der Master lädt nun die relevanten Knoten, also die Knoten des aktuellen und des folgenden Zuges.

Folgend wird in der prepareWinpercentageArraySend für jeden Knoten in den Predeccessorknots ein Array von Winpercantage Werten im taskSendBuffer vorbereitet. Diese werden dabei gefüllt mit den Winpercentages der Nachfolgenden Knoten. Werte die im Winpercentage Array nicht gefüllt werden können, da z.B das Array eine Größe X hat aber es nur X-4 Nachfolger gibt so werden die dabei entstehenden freien Werte mit dem Wert -1 belegt.

Anschließend können diese Arrays an die Prozesse verteilt werden über einen Scatter-Befehl. Dabei allokiert jeder Prozess den Platz für seinen TaskRecvBuffer und der TaskSendBuffer kann freigegeben werden. Dann kann der ResultSendBuffer allokiert werden. Anschließend wird für jedes Array von Winpercanteges der Wert der Winpercentage für den dazu gehörigen Knoten ausgerechnet. Die ausgerechneten Winpercanteges werden nun empfangen und der resultSendBuffer kann freigegeben werden. Nun kann der Master den entsprechenden Knoten die entsprechenden Winpercentages zuordnen. Anschließend werden die alten abgespeicherten Knoten von den neu berechneten Knoten überschrieben. Nun können alle weiterhin nicht benötigten Ressourcen freigegeben werden.

TL;DR:
Der Vorgang in seiner Theorie ohne zu sehr auf das Technische einzugehen wäre wie folgt:
Es wird für jeden Knoten im Vorletzten Schritt ein Array an Winpercentages der Nachfolgeknoten erstellt. Diese werden an die Prozessoren verteilt um daraus die Winpercantages des Knotens zu berechnen. Anschließend wird dieser gesetzt und im saveFile abgespeichert. Der Vorgang wird für den Zug davor wiederholt bis alle Knoten ihre Winpercantge berechnet haben.

4.Laufzeitmessungen

Beim sequentiellen Algorithmus ergeben sich bei Feldern kleiner als 4x4 Laufzeiten von weniger als einer Sekunde. Ein 4x4 Feld dauert im Durchschnitt ~38 Sekunden, 5x4 dauert knapp mehr als 2 Stunden und 5x5 dauert länger als das Zeitlimit auf dem Cluster von 6 Stunden.

Die Version die nur den Arbeitsspeicher verwendet erhält vergleichbare Laufzeiten bei Feldern kleiner als 4x4. Aber andere Felder sind zu groß, da diese Version im Gegensatz zur Sequentiellen keine doppelten Spielfelder aussortiert, reicht der Arbeitsspeicher nicht.

Die finale Version benötigt für ein 4x3 Feld 2 Minuten und für 4x4 eine knappe Stunde. Diese Werte hängen stark von der Lese- und Schreibgeschwindigkeit der Festplatte des Systems ab und stammen von Tests auf dem Cluster.

5.Leistungsanalyse

In der Version die nur den Arbeitsspeicher verwendet kann man sehen, dass das Programm fast ausschließlich mit der Hauptaufgabe des Berechnens und Auf-Sieger-Überprüfens von nachfolgenden Spielfeldern ist. Allerdings ergibt sich bei einem 4x4 Spielfeld bereits das Problem, dass der Arbeitsspeicher voll ist, was das Programm zum Abstürzen bringt.

Die Version welche auch den Hauptspeicher verwendet ist extrem ineffizient, da die Speicher- und Ladeoperationen von dem Master-Prozess übernommen werden.

Die Leistung ließe sich also stark verbessern indem jeder Prozess die für ihn relevanten Daten selbst lädt und speichert. Das zu koordinieren würde die Komplexität des Programms noch einmal wesentlich steigern und wir haben es innerhalb der gegebenen Zeit nicht geschafft eine solche Lösung zu implementieren.

Eine weitere Möglichkeit die Leistung zu steigern wäre das Zusammenführen von doppelt vorkommenden Spielfeldern parallel zu bewältigen. In der ersten parallelen Implementation des Programmes wurde diese Aufgabe sequentiell vom Master-Prozess erledigt was zu erheblichen Wartezeiten bei den anderen Prozessen führte.
Wenn dies aber parallel implementiert wäre, würde sich die Anzahl der errechneten Spielfelder enorm verringern, was die Laufzeit dementsprechend verringern würde.

6.Skalierbarkeit

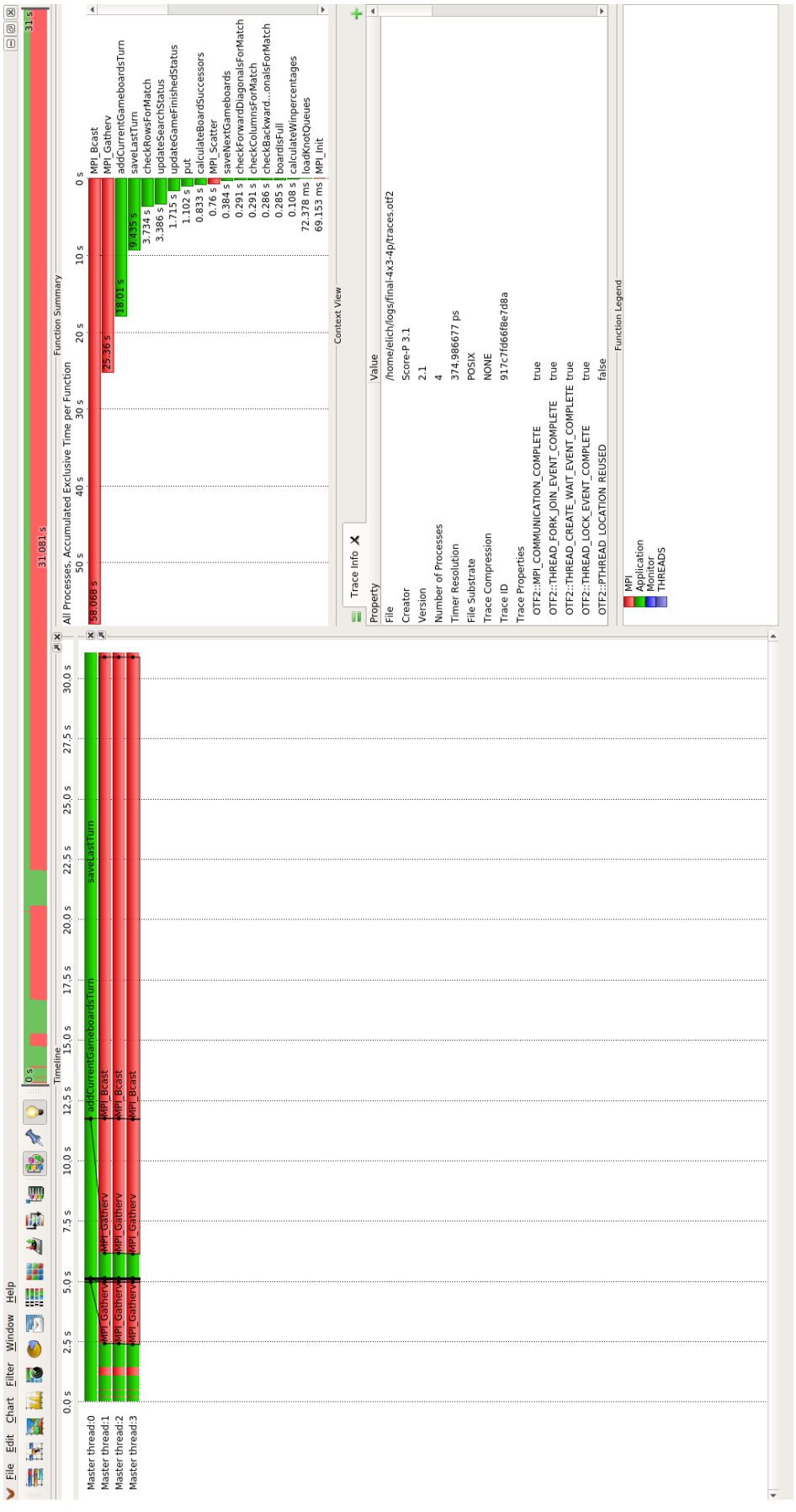
Es ist uns leider nicht gelungen ein Programm zu entwickeln, das besonders gute Skalierbarkeit besitzt.

Die Version welche nur den Arbeitsspeicher verwendet hätte theoretisch und laut der Analyse mit Scorep eine sehr gute Skalierbarkeit, da der Overhead im Vergleich zu der Rechenzeit extrem gering ist. Aber diese Version lässt keine Berechnungen zu die lang genug dauern um aussagekräftige Daten zur Skalierbarkeit zu sammeln.

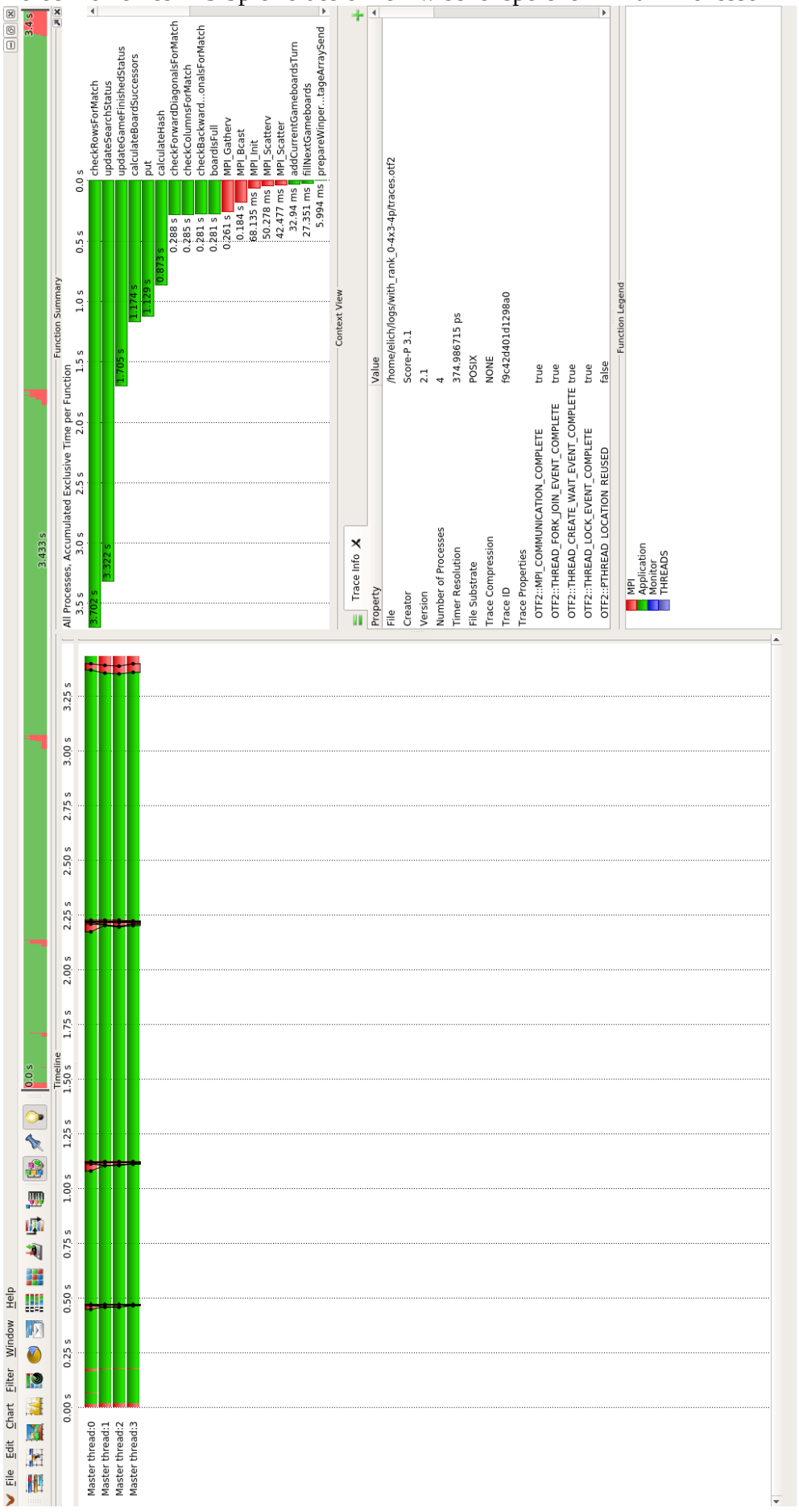
Die andere Version lässt solche Berechnungen zwar zu, es ist aber sehr schnell Erkennbar dass die Skalierbarkeit minimal ist, da das Speichern und Laden einen extremen Overhead-Aufwand darstellt.

7.Anhang

Berechnen eines 4x3 Spielfeldes mit Speichern aller Zwischenergebnisse mit 4 Prozessen



Berechnen eines 4x3 Spielfeldes ohne Zwischenspeichern mit 4 Prozessen



Berechnen eines 4x3 Spielfeldes ohne Zwischenspeichern mit 3 Prozessen