Parallele Programmierung

Conway's Game of Life 3D

im Sommersemester 2015
Universität Hamburg

Elena Bergmann
Tobias Wesseler

Conway's Game of Life 3D

Inhaltsverzeichnis

1.Problemstellung....3
2.Lösungsansatz....4
2.1.Sequentieller Algorithmus....4
2.2.Ein- / Ausgabe....5
2.3.Datenvisualisierung....5
3.Parallelisierungsschema....5
4.Laufzeitmessungen....7
4.1.Sequentiell....7
4.2.Parallel (MPI)....7
5.Leistungsanalyse....8
5.1.Sequentielles Programm mit oprofile....8
5.2.Paralleles Programm mit Vampir....8
5.3.Ergebnis der Leistungsanalyse....10
6.Skalierbarkeit....11
6.1.Strong Scaling....11
a)Speedup-Diagramm....11
b)Effizienz-Diagramm....11
6.2.Weak Scaling....13
7.Quellen....13

1. Problemstellung

Conway's Game of Life wurde 1970 von John Horton Conway entworfen und stellt einen zweidimensionalen zellulären Automaten dar. Das Spielfeld besteht aus einzelnen Zellen, die quadratisch angeordnet sind und zwischen zwei Zuständen wechseln können. Lebendige Zellen sind aktiv und tote Zellen inaktiv. Die Regeln beschreiben, wann eine tote Zelle lebendig wird, wann eine lebendige Zelle stirbt und wann sie weiterlebt.

Conways Anfangsregeln waren:

- Tote Zellen mit genau drei lebenden Nachbarzellen werden lebendig.
- Lebende Zellen mit weniger als zwei lebenden Nachbarzellen sterben.
- Lebende Zellen mit zwei oder drei lebenden Nachbarn bleiben am Leben.
- Lebende Zellen mit mehr als drei lebenden Nachbarn sterben.

Das Anfangsmuster (fortan „Initialkonfiguration") wurde von dem Spieler gesetzt und durch die Regeln entwickelten sich die Generationen. Statische Objekte bleiben gleich, da ihre leben Nachbarzellen immer gleich bleiben. Sich verändernde Objekte können an einem Ort bleiben oder sich durch das Spielfeld bewegen. Gerade sich bewegende Objekte waren interessant, das sie von den Anfangsregeln nicht vorgesehen waren.

Conway stellte die Herausforderung eine Initialkonfiguration zu finden, die unbegrenztes Wachstum möglich macht (auf einem theoretisch unbegrenztem Spielfeld). Er setzte dafür damals sogar ein Preisgeld von 50$ aus. Bald wurden Gleiterkanonen entdeckt. Diese sind Muster welche Gleiter erzeugen. Gleiter sind Muster welche sich durch das Spielfeld fortbewegen. Inzwischen wurden sogar Gleiterkanonen-produzierende Objekte gefunden – sogenannte Breeder. Die entdeckten Objekte machten schließlich den Beweis der Turing-Mächtigkeit möglich.

Nachdem im zweidimensionalen Spielfeld schon statische, oszillierende, sich bewegende, erzeugende und vernichtende Objekte gefunden wurden, bleibt im dreidimensionalen Bereich noch weiteres zu entdecken. Wir werden daher dreidimensionale Spielfelder verarbeiten und das ganze mit MPI parallelisieren.

Um den gewaltigen Suchraum an Möglichkeiten für ein 3D Game of Life einzuschränken, haben wir für unsere erste Version einige Kompromisse gemacht.

1. Wir konzentrieren uns auf ein einzelnes Regelwerk, für die Bestimmung von Tot und Leben der Zellen.
2. Wir untersuchen keine konstruierten Initialkonfigurationen, sondern nur die sogenannte Primordial Soup. Dies ist eine zufällige Ansammlung lebendiger Zellen.

2. Lösungsansatz

Wir erzeugen viele zufallsgenerierte Spielfelder und lassen diese dann verarbeiten. Nach dem Einlesen im Programm wird die Folgegeneration anhand der Spielregeln berechnet. Wir orientieren uns da an der vielversprechenden Regel 4/5/5/5, bei der vier bis fünf lebendige Nachbarn für lebende Zellen benötigt werden und genau fünf lebende Zellen für tote Zellen, um lebendig zu werden. Diese Regel wurde von Carter Bays als eine interessante Regel für das 3D Game of Life beworben und untersucht.

Untersucht werden sollen die Welten anhand der Entwicklung der Bevölkerung. Welten mit explodierendem Wachstum. Interessante Spielwelten sind stagnierend oder wachsen gemächlich.

Wenn die Bevölkerungszahl stagniert, dann enthält sie mit hoher Wahrscheinlichkeit statische oder oszillierende, eventuell sogar Gleiter-Objekte. Auch bei Wachstum ist dies gut möglich.

Das entwickeln der Generationen, welche aus der Initialkonfiguration anhand der Regeln deterministisch entstehen, wird dabei parallelisiert. Die Auswertung und Untersuchung findet im Anschluss statt. (Auswertung nicht implementiert).

2.1. Sequentieller Algorithmus

Zuerst werden mit einem separatem Programmaufruf Initialkonfigurationen erzeugt und in Textdateien gespeichert. Diese werden dann eingelesen und Generationen daraus abgeleitet.

1. Welt einlesen aus Textdatei in Array einlesen und zur „Aktuellen Welt“
2. Leere Welt erzeugen in welcher die NachfolgeGeneration hinterlegt wird
dies ist nun die „Nächste Welt“
3. Aktuelle Welt und Bevölkerungzahl in Output-Textdatei vermerken

4. **Für jede Zelle** in aktueller Welt:
   Zähle Nachbarn
   falls Zelle lebendig
   falls Anzahl Nachbarn erlaubt überleben
   setze entsprechende Zelle in nächster Welt als lebendig
   zähle Bevölkerung hoch
   sonst tue nichts
   sonst ist die Zelle tot, dann
   falls Anzahl Nachbarn erlaubt Geburt
   setze entsprechende Zelle in nächster Welt als lebendig
   zähle Bevölkerung hoch
   sonst tue nichts
5. setze die Aktuelle Welt gleich der Nächsten Welt.
   Solange Anzahl gewünschter Generationen nicht erreicht → Gehe zu 2.
6. Schreibe abschliessende Informationen, wie z.B. Stencils/s in Output-Textdatei

2.2. Ein- / Ausgabe

Als Eingabe der Welt gibt es in einer Textdatei z-mal viele x*y Blöcke, daher haben wir die z-Koordinate in Scheiben aufgeteilt. Lebende Zellen werden als eins dargestellt, tote Zellen als null.Die Ausgabe wird um die Generationenanzahl und Populationsgröße ergänzt, ist ansonsten genauso aufgebaut.

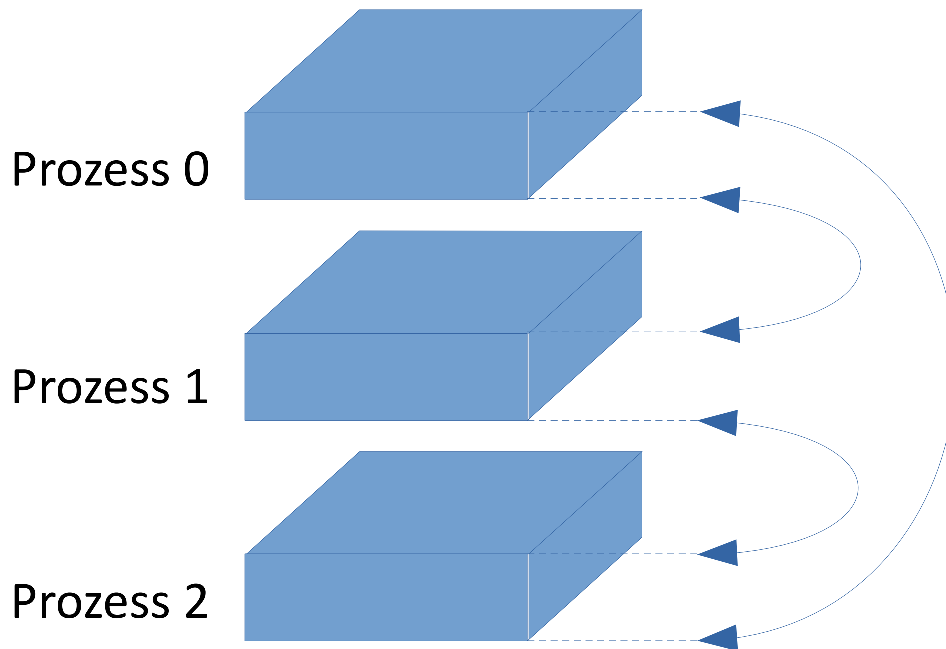
2.3. Datenvisualisierung

Nicht implementiert.

3. Parallelisierungsschema

In MPI teilt der Master jedem Prozess die Länge des Abschnitts mit, den er zu bearbeiten hat. Die Welt ist dabei in einem Array gespeichert. Zur Berechnung holt sich jeder Prozess die angrenzenden Zellen (Siehe Zeichnung 3.1) zweier anderer Prozesse. Dabei ist zu achten, das der erste Prozess vom letzten Daten holt und andersrum. Die Prozesse senden und empfangen abwechselnd – dies geschieht mittels MPI_Ssend und MPI_Recv in 4 Schritten:

1. Prozesse mit gerader Ordnungszahl (im Folgenden „gerade Prozesse“) senden ihre letzte Z-Scheibe (die letzten x mal y Zellen) an den jeweiligen nachfolgenden Prozess mit ungerader Ordnungszahl (im Folgenden „ungerade Prozesse“).
2. Ungerade Prozesse senden ihre letzte Z-Scheibe an den nächsten geraden Prozess.
3. Gerade Prozesse senden ihre erste Scheibe an den vorherigen Prozess.
4. Ungerade Prozesse senden ihre erste Scheibe an den vorherigen Prozesse



Zeichnung 3.1:

Der erste und letzte Prozess sind dabei jeweils der Vorgänger und Nachfolger des anderen. Dieses Schema funktioniert auch mit einer ungeraden Anzahl von Prozessen. Die Grenzen der vier oben genannten Schritte verschieben sich für den ersten und letzten Prozess allerdings etwas. In der Zeichnung unten ist das Beispiel für drei Prozesse grob skizziert.

Nach dem erfolgreichen Austausch der Daten zwischen den Prozessen, beginnt jeder Prozess nun für seinen Block die nächste Generation zu berechnen. Dabei wird gleichzeitig die neue Populationsgröße erfasst. Diese wird in einem nächsten Verarbeitungsschritt mittels der kollektiven Operation MPI_Gather vom Master-Prozess gesammelt, welcher dann die Gesamtbevölkerungszahl der jeweiligen Generation errechnet. Aktuell wird diese nur gesammelt und in der Ausgabe-Datei des Masters mit verzeichnet. Sie könnte aber auch für Analysezwecke und zur Programmoptimierung eingesetzt werden. Siehe hierzu das Kapitel Problemstellung.

4. Laufzeitmessungen

Wir haben Laufzeitdauer bezüglich der „Wallclock“-Zeit (also der Echtzeit) und Verarbeitungsgeschwindigkeit bezüglich Stencils gemessen. Eine Stencil-Operation ist hier definiert als eine einzelne Ausführung der Funktion „countNeighbours(...)“. Diese wiederum beinhaltet 26 mal den Aufruf der Funktion „offset“.

4.1. Sequentiell

Es gibt nur eine Laufzeitmessung für das Sequentielle Programm. Diese dient für unsere SpeedUp- und Effizienzdiagramme als T1 – Grundlage, also als Verarbeitungszeit mit nur einem Prozess.

Bei einer Welt mit 1 000 000 Zellen dauert die Entwicklung von hundert Generationen durch einen einzigen Prozess ca. 76 Sekunden.

4.2. Parallel (MPI)

Abbildung 4.1 zeigt, dass die Verarbeitungszeit ab dem 12. Prozess nicht wirklich weiter sinkt. Dies liegt daran, dass die Ausgabe durch den Master-Prozess länger dauert und die blockierenden MPI-Funktionen, vor allem das kollektive MPI_GATHER, sowie der generelle Overhead durch die Netzwerkkommunikation das Programm ausbremsen.

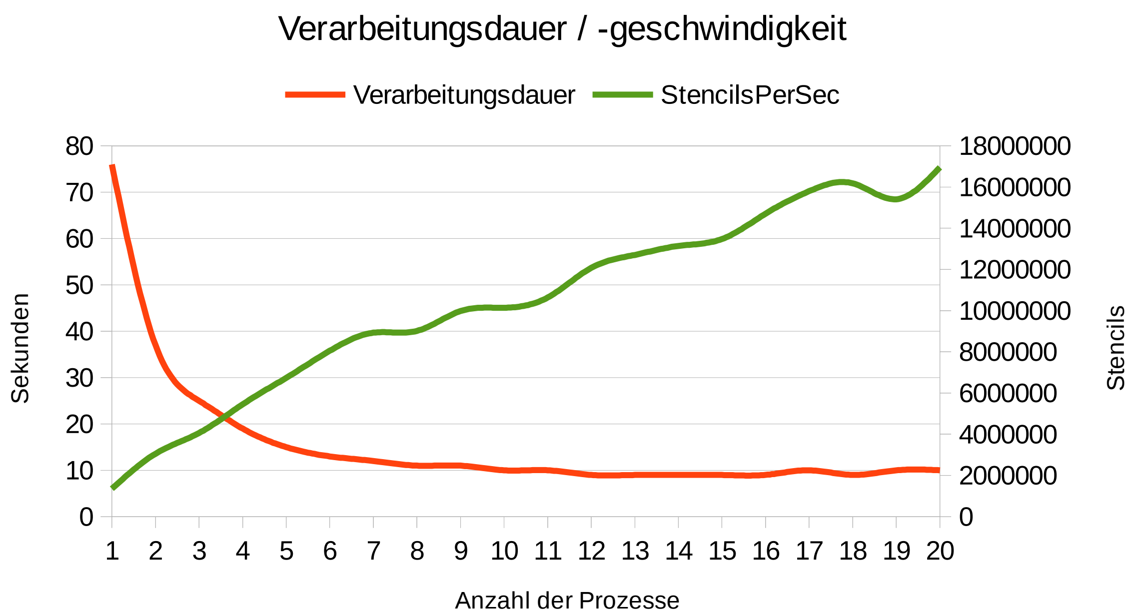


Abbildung 4.1

Allerdings kann man auch beobachten, dass die Stencils pro Sekunde weiter zunehmen. Dies

kommt der Bewertung beim weak scaling zu Gute. Mit einem wachsenden Problem könnte eine größere Anzahl von Prozessen effizient genutzt werden. Diese Laufzeitmessung wurde auf dem Cluster-Frontend ausgeführt, daher stehen nicht mehr als 12 Prozesse zur Verfügung.

5. Leistungsanalyse

In diesem Kapitel werden wir sowohl die sequentielle Version als auch die parallele Version des Programms mittels der Werkzeuge oprofile bzw. VampirTrace überprüfen, um festzustellen ob es wie erwartet funktioniert.

5.1. Sequentielles Programm mit oprofile

```
the sample file was created.
warning: the last modified time of the binary file does not match that
 of the sample file for /lib/x86_64-linux-gnu/libc-2.15.so
samples  %        image name               symbol name
159013   75.9665  singol                   count_neighbours
39787    19.0077  singol                   offset
4228      2.0199  singol                   evolveWorld
2264      1.0816  no-vmlinux               /no-vmlinux
1906      0.9106  libc-2.15.so             vasprintf
1674      0.7997  singol                   outputTXT
256       0.1223  libc-2.15.so               __strcasecmp_l_sse2
```

Abbildung 5.1

Abbildung 5.1 zeigt einen Ausschnitt der Ausgabe des Befehls *opreport -l* nachdem eine Welt mit 500 000 Zellen für 20 Generationen iteriert wurde. Gemessen wurden die Clockticks wenn die CPU gearbeitet hat, also das Default-Event CPU_CLK_UNHALTED.

Schnell wird deutlich, dass die Funktion countNeighbours einen Löwenanteil der Ausführungszeit für sich beansprucht. Als erstes Zeichen ist dies positiv zu werten, da es dem gewünschten Verhalten entspricht. Allerdings muss auch beobachtet werden ob die Zeit innerhalb der Funktion sinnvoll genutzt wird und/oder gegebenenfalls optimiert werden kann.

5.2. Paralleles Programm mit Vampir

Die Analyse des Programms mit dem Tracingtool Vampir hat ergeben, dass das Programm gemäß den Erwartungen verläuft. Der Auszug in Abbildung 5.1 zeigt einen recht großen roten Abschnitt im oberen rechten Bereich. Dies ist die Funktion MPI_Init. Der Bereich ist recht groß, da die Gesamtbeobachtungszeit nur über einige wenige Iterationen statt fand. Im linken großen Teil von Abbildung 5.1 kann man schwarze vertikale Striche über den Prozessbalken sehen. Dies sind die Kommunikationslinien des Datenaustausches. Sie markieren also jeweils ungefähr den Beginn

einer Iteration. Etwas weiter rechts ist einmal ein größerer roter Abschnitt. Hier wartet der zweite Prozess darauf, dass der erste im Daten schickt. Hier könnte man ansetzen und optimieren, in dem der zweite Prozess bereits den Teil des nächsten Iteration berechnet, für den er nicht die Daten aus dem ersten braucht.

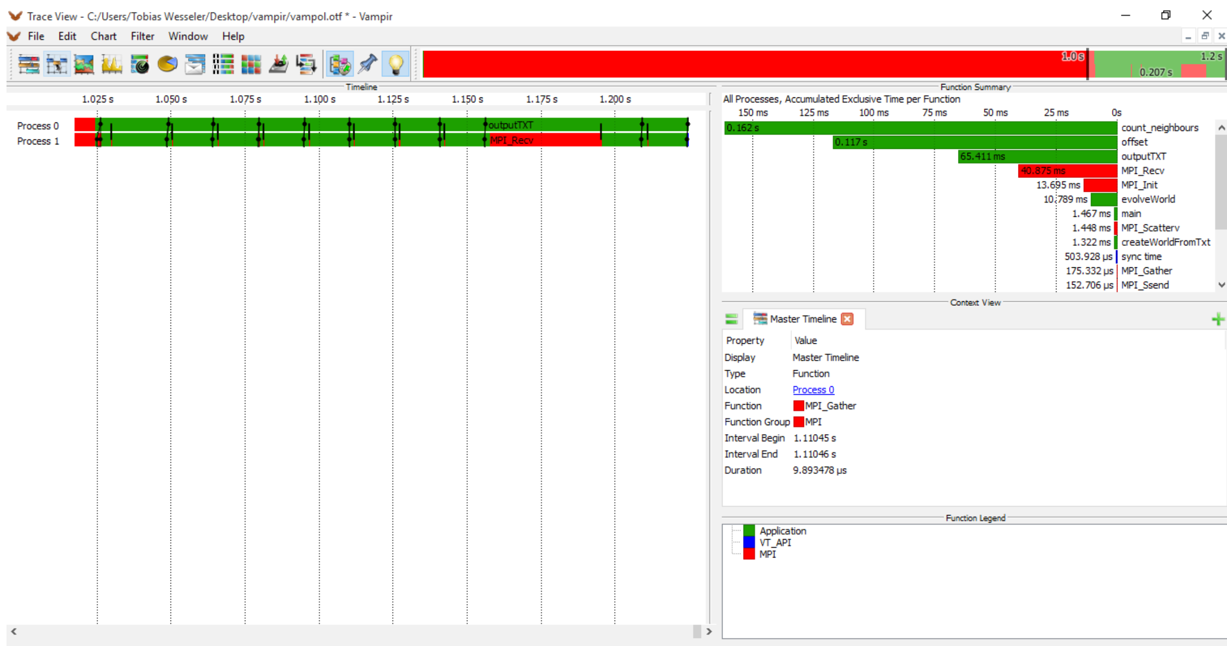


Abbildung 5.2

In der folgenden Abbildung (Abb. 5.2) ist der Datenaustausch zwischen den Prozessen höher aufgelöst. Zwischen den Iterationen kann man hier gut erkennen, wie das Kommunikationsmuster nach dem Parallelisierungschema funktioniert. Siehe dazu auch Zeichnung 3.1 und Kapitel 3. Abwechselnd senden und empfangen die Prozesse Daten.

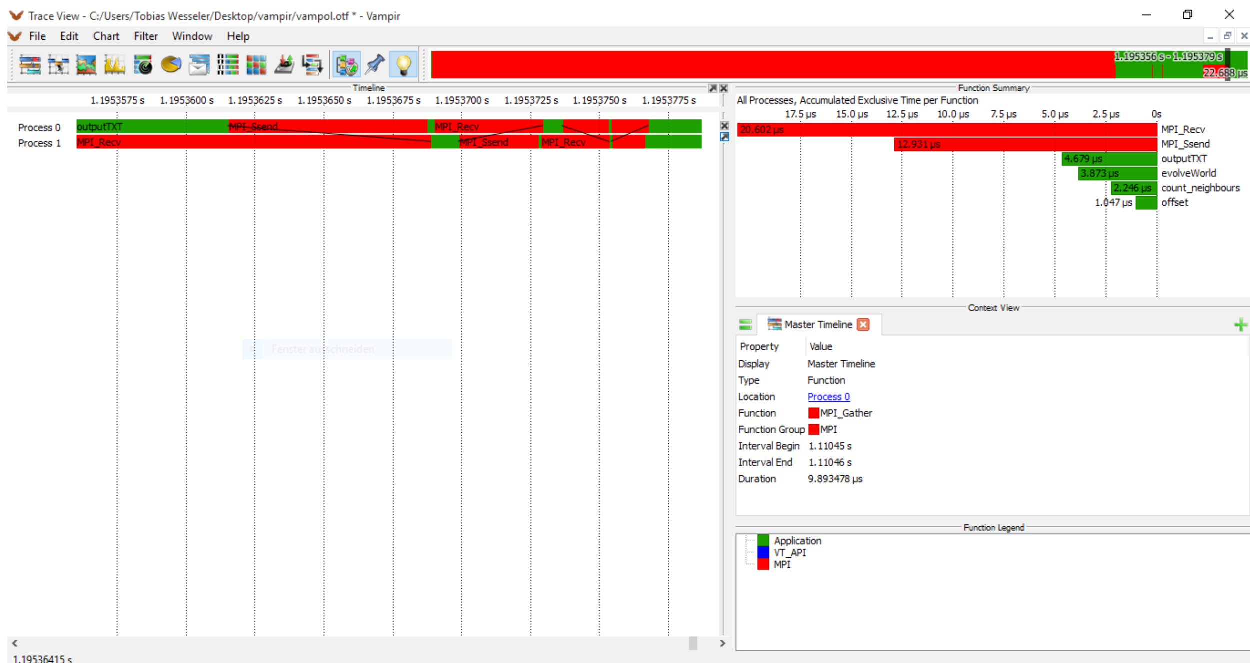
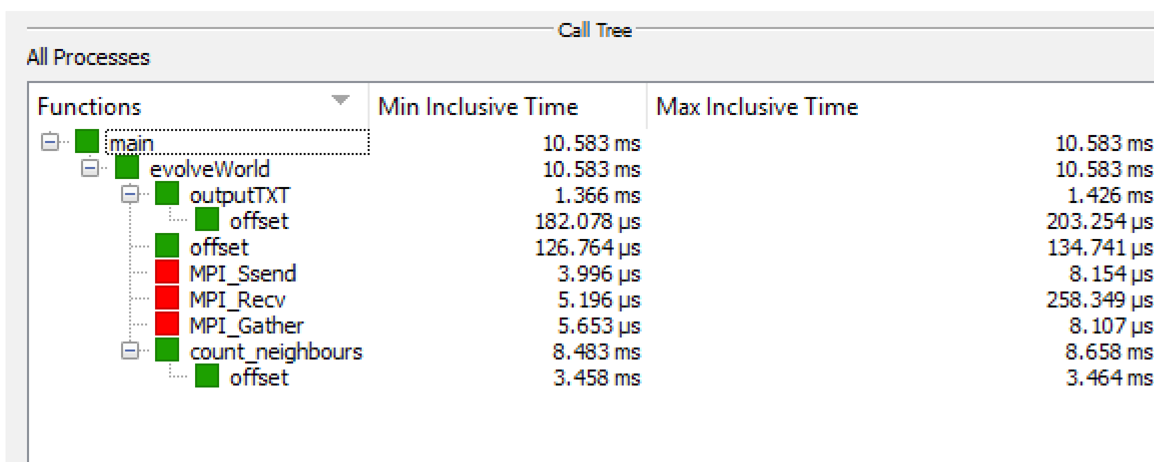


Abbildung 5.3

Der CallTree (Abb. 5.4) zeigt dass die Funktion „offset“ mit Abstand am meisten aufgerufen wird. Auch dies ließe sich möglicher reduzieren und optimieren. Da wir unseren dreidimensionalen Würfel in einem eindimensionalen Array ablegen, benötigen wir die offset-Funktion um xyz-Koordinaten in Arraypositionen umzuwandeln. An einigen Stellen wo wir via x, y und z-Werten durch den gesamten Würfel (bzw. Teilblock des Würfels) iterieren, könnten wir stattdessen auch direkt durch das Array wandern indem wir einfach ein Zeiger immer um eine Position nach vorn verschieben.


Call Tree

All Processes

Functions	Min Inclusive Time	Max Inclusive Time
main	10.583 ms	10.583 ms
evolveWorld	10.583 ms	10.583 ms
outputTXT	1.366 ms	1.426 ms
offset	182.078 µs	203.254 µs
offset	126.764 µs	134.741 µs
MPI_Ssend	3.996 µs	8.154 µs
MPI_Recv	5.196 µs	258.349 µs
MPI_Gather	5.653 µs	8.107 µs
count_neighbours	8.483 ms	8.658 ms
offset	3.458 ms	3.464 ms

Abbildung 5.4

5.3. Ergebnis der Leistungsanalyse

Wie man am CallTree erkennen kann, verbringt das Programm tatsächlich die meiste Zeit in der

Funktion countNeighbours. Es verhält sich also wie erwartet. Auffallend ist aber auch, dass knapp die Hälfte dieser Zeit wiederum in der Funktion offset landet. Diese lässt sich hier nicht so einfach optimieren wie oben vorgeschlagen, da hier keine rein sequentiellen Indizes berechnet werden, sondern die der 26 Nachbarn einer Zelle. Von denen liegen allerdings zwei mal 8 mal drei Stück hintereinander, was wieder Optimierungspotenzial bietet. Allerdings stellen Randgebiete wieder eine Ausnahme dar, was die Findung eines Algorithmus etwas erschwert.

Als Optimierung könnte man MPI_SendRecv statt einzelner MPI_Send und MPI_Recv verwenden, um den Overhead weiter zu verringern.

6. Skalierbarkeit

Als Aufgabenstellung definieren wir das berechnen von Generationen für Initialkonfigurationen und das ausgeben dieser in Textdateien. Diese lässt sich sowohl mit einer festen Aufgabengröße (im Weiteren auch „strong scaling“) als auch mit wachsender Aufgabengröße (im Weiteren auch „weak scaling“) gut analysieren.

6.1. Strong Scaling

Pargol kommt in seiner aktuellen, wenig optimierten Version beim strong scaling schnell an seine Grenzen. Die nachfolgenden Diagramme und Erläuterungen veranschaulichen dies.

a) Speedup-Diagramm

Das Speedup-Diagramm (Abb. 6.1) veranschaulicht die Parallele Beschleunigung des Programms. Die zugrunde liegende Aufgabe war, die Berechnung und Ausgabe von 100 Generation einer Initialkonfiguration mit 1 Million Zellen.

Für bis zu 8 Prozesse skaliert die Beschleunigung ganz ordentlich. Ab Prozess Nr 9 wird der Overhead durch die Netzwerkkommunikation, mit jedem neuen Knoten (also bei ungeraden Prozesszahlen) so hoch, dass es kleine Leistungseinbußen gibt. Bis zu 18 Prozesse können dies ausgleichen, in dem auf jedem Knoten auch 2 Prozesse laufen, anstatt nur einem. Danach wird der Overhead allerdings deutlich zu groß, weil das Problem dann in zu kleine Teilprobleme aufgesplittet wurde.

b) Effizienz-Diagramm

Im Effizienz-Diagramm wird der SpeedUp-Wert durch die Anzahl der Prozesse geteilt. Dadurch kann man mit Hilfe eines Effizienz-Diagramms darstellen, wie sinnvoll der Einsatz einer größeren

Anzahl an Prozessen ist bzw wie gut jeder einzelne Prozess dann noch ausgelastet ist. Eine wachsende Anzahl an Prozessen bedeutet nämlich in aller Regel auch eine schrumpfende Problemgröße pro Kern (zumindest beim häufig auftretenden strong scaling), sowie mehr Kommunikation zwischen den Prozessen und damit mehr Overhead. Abbildung 6.2 ist das zu 6.1 gehörige Effizienzdiagramm.

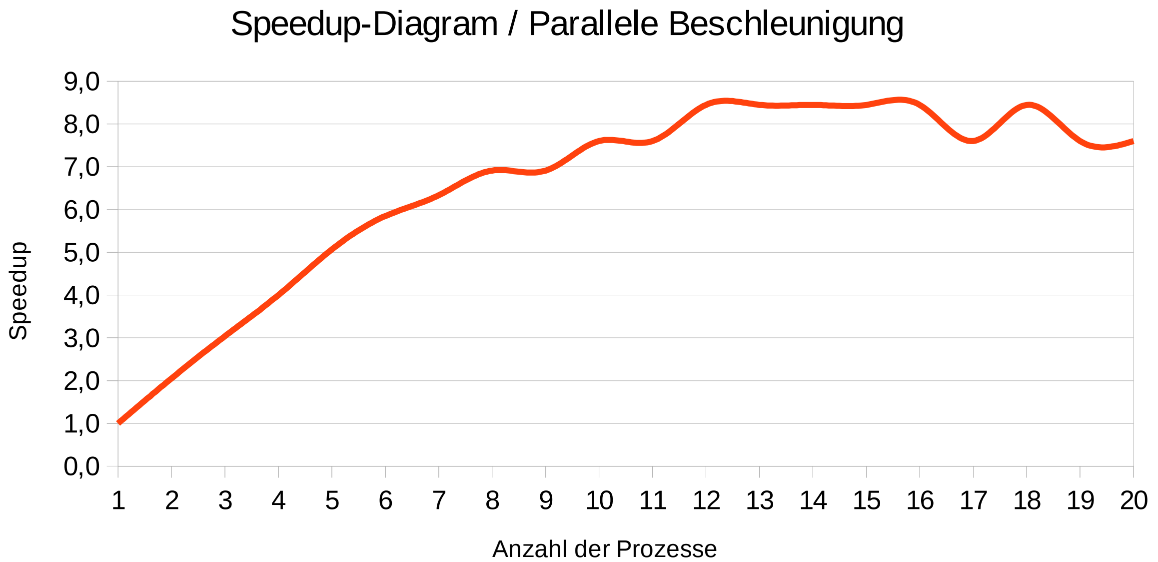


Abbildung 6.1

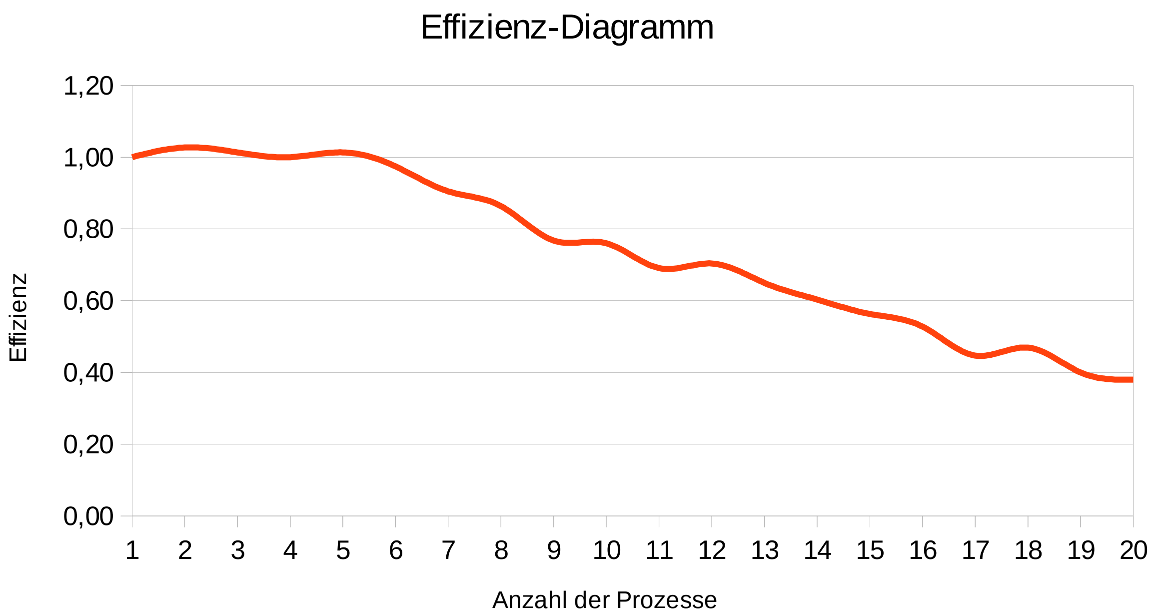


Abbildung 6.2

6.2. Weak Scaling

Dieses Kapitel kann gerne in Zukunft ergänzt werden. Da Conway's Game of Life 3D mit der anfänglich genannten Problemstellung ein forschendes Experiment mit einem gewaltig großen Suchraum ist, eignet es sich besonders gut für weak scaling. Es könnten also theoretisch sehr viele Prozesse mit verschiedenen Initialkonfigurationen und Regelwerken beschäftigt werden, oder man könnte die Größe der Welten einfach erhöhen. Der aktuelle Stand der Implementation integriert solche Möglichkeiten allerdings noch nicht, da dies den Rahmen der Praktikumsarbeit in einem Semester gesprengt hätte, weshalb hier nun dieser Platzhalter steht.

Sollten andere Studenten dieses Projekt fortsetzen wollen, ist dies ein interessanter zu erforschender Aspekt.

7. Quellen

Bays, Carter, “Candidates for the Game of Life in Three Dimensions,” Complex Systems, 1 (1987) 373–400.

Abbildungen 5.x: Screenshots aus Vampir 8
Zeichnung 3.1: Mittels LibreOffice selbst gezeichnet

speedup und effizienzdiagramme: verständnis von http://www.math.tu-cottbus.de/~kd/parallel/vorl/vorl/node20.html