# Simmulation Ideenverbreitung
## IDK ATM

Arne Struck, Manuel Börries, Jonathan Werner

Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik, DKRZ
Praktikum Paralleles Programmieren

**Zusammenfassung.**
Aufgabenstellung:
Die Erstellung einer parallelen, clusterfähigen Simulation mittels MPI.

Projekt Idee:
Die Untersuchung der Verbreitung von Meinungen, Vorstellungen innerhalb einer Population.

**Schlüsselwörter:** Keywords

# Inhaltsverzeichnis

# 1 Projekt-Idee

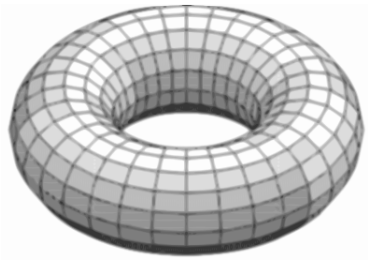
Diese Projekt soll die Verbreitung und Konkurrenz von Ideen im Sinne einer Weltanschauung, eines kontroversen Themas innerhalb einer begrenzten Population untersuchen und deren Mechanismen simulieren. Die Ideen sollen hierbei durch Kommunikation der Individuen und durch Entwicklung veränderbar sein. Des weiteren darf es keine "unrealistischen" Entwicklungen geben. Niemand der gerade erst das Feuer entdeckt hat, baut am nächsten Tag eine Rakete.

# 2 Modelierung

In diesem Abschnitt wird ein Überblick über die Modellierung der Ideen, der Welt und der Menschen in ihr gegeben. Details zur eigentlichen Implementation, insbesondere zur Aufteilung auf mehrere Prozesse sind im Abschnitt Implementation zu finden.

## 2.1 Die Welt

In 2 Dimensionen betrachtet besteht die Welt auf einem Grid, deren jeweilige Enden miteinander verbunden sind. In einer dreidimensionalen Betrachtung entsteht somit ein Torus-Körper.



## 2.2 Die Idee

Eine Idee kann viele Eigenschaften haben, die wichtigen Faktoren um andere Menschen von ihr zu überzeugen lassen sich allerdings grob durch 3 Eigenschaften modellieren. Diese sind zum einen die Qualität, die Komplexität, als auch eine arbiträre durch Zahlen ermöglichte Darstellung des Raumes in dem die Idee angesiedelt ist, im folgenden Weltanschauungswert genannt.
Die Qualität einer Idee soll den Grad ihrer Überzeugungskraft repräsentieren. Die Komplexität hängt mit diesem Wert zusammen, ist allerdings nicht der Selbe, da auch nicht sehr überzeugende Ideen komplex und elaboriert sein können und umgekehrt.

## 2.3 Der Mensch

Ein Mensch besitzt zwei für die Simulation relevante, darstellbare Eigenschaften. Die eine ist die Idee, die andere ist ein Weltanschauungswert, ähnlich der der Idee. Allerdings repräsentiert der Weltanschauungswert des Menschen s restlichen Ideen, welche er besitzt, die allerdings nicht direkt das Themengebiet der untersuchten Ideen-Gruppe berühren, sondern nur indirekte Einflüsse darauf haben. Ein Beispiel hierfür wäre das Mathematikverständnis eines Menschen der Antike betreffend der Frage ob die Erde einer Kugel ähnelt oder nicht.

### 2.4 Ablauf

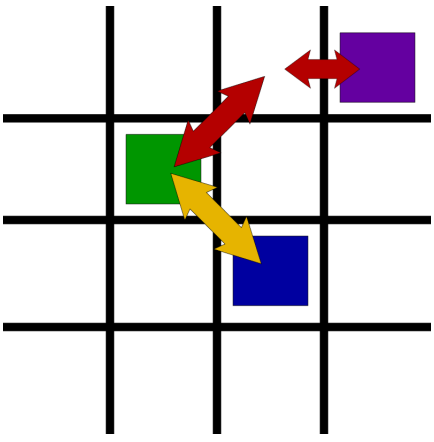
Zuerst wird eine Population von Menschen mit Ideen durchschnittlich geringer Qualität und Komplexität mit zufälligen Weltanschauungswerten in der Welt geschaffen. Die Weltanschauungswerte der Menschen sind in der Nähe ihrer Initialideen angesiedelt.
Nun beginnen die verschiedenen Menschen in runden auf der Welt zu ziehen. Zu erst wird die Möglichkeit der Entwicklung einer neuen Idee und oder die Veränderung der Weltanschauungswerte evaluiert, diese ist relativ gering, tritt aber dem Gesetz der großen Zahlen folgend bei einigen Individuen alle paar Runden ein. Daraufhin werden die Möglichkeiten zur Kommunikation erfasst, sollte ein anderes Individuum sich in Reichweite befinden, wird mit einer gewissen Wahrscheinlichkeit ein Kommunikationsversuch gestartet. Die Wahrscheinlichkeit resultiert aus dem menschlichen Verhalten nicht mit jedem Individuum aus seiner Umgebung Konversation zu betreiben. zum Schluss wird ein Individuum seine Möglichkeiten zur Bewegung evaluieren.

### 2.5 Entwicklung

Die Entwicklung einer Idee erfolgt durch Mutation. Jeder Mensch hat jede Runde eine Mutationschance sowohl für den Qualitäts-, Komplexitätskomplex, als für den Weltanschauungskomplex. Beide Mutationen laufen ähnlich ab, zu erst wird eine Richtung gewählt, um zu gewährleisten dass die Eigenschaften nicht in 2 Richtungen mutieren. Darauf werden die Mutationsraten der einzelnen Elemente berechnet und diese angepasst. Bei dem Weltanschauungskomplex existiert noch die finale Absicherung, dass die Differenz beider Werte nicht einen Schwellwert überschreiten.

### 2.6 Kommunikation, Konkurrenz und Einschränkungen

Bevor es zu einer Kommunikation kommt, muss ein möglicher Kommunikationspartner gefunden werden. Dieser muss sich (wie rechts dargestellt) in einem benachbartem Feld befinden. So können die grüne und die blaue Idee miteinander konkurrieren, der Violetten ist dies allerdings nicht möglich. Sollten mehrere Kommunikationspartner zur Verfügung stehen, wird einer diese ausgewählt.
Die eigentliche Kommunikation ist in drei Phasen geteilt. Nun da ein Kommunikationspartner gefunden ist, wird entschieden ob die beiden sich austauschen.



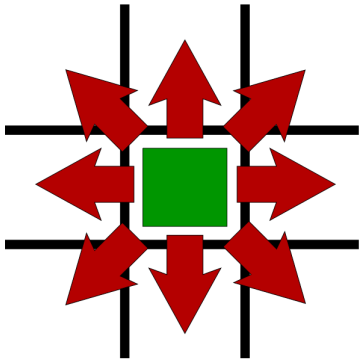
Kommunikationsmöglichkeiten

Ist dies geschehen und positiv ausgefallen muss berechnet werden, ob sie einander überzeugen können. Dies hängt von Faktoren, wie der Komplexitätsdifferenz und den Unterschieden zwischen dem Weltanschauungswert der Zielperson und der Ursprungsidee ab. Je höher die Komplexitätsdifferenz ist, desto größer ist die Wahrscheinlichkeit, dass eine Überzeugung fehlschlägt, da davon auszugehen ist, dass die beiden Menschen auf verschiedenen Ebenen denken und somit nicht einig werden. Die Inkompatibilität sich stark unterscheidender Weltanschauungswerte sollte auf der Hand liegen.
Sollte eine Kompatibilität festgestellt werden, so muss ermittelt werden, welche der beiden Ideen die andere dominiert. Dies geschieht anhand eines Qualitätsabgleichs mit einem zufälligen temporären Aufschlag auf die Qualitätswerte beider Ideen. Dieser Aufschlag repräsentiert mögliche sonstige Einflüsse wie beispielsweise die Eloquenz des Gegenübers.
Wenn nun ein Gewinner feststeht, muss der Verlierer seine Meinung, respektive seine Idee ändern. Dies ist leicht durch die direkte Übernahme zu garantieren. Allerdings muss auch der Weltanschauungswert des verlierenden Menschens an die Idee angepasst werden, da die Idee einen Einfluss auf diesen hat.

### 2.7 Bewegung

Nicht alle Menschen gehen einer missionarischen Tätigkeit (welche Anschauung sie auch immer vertreten) sondern vielen anderen Beschäftigungen nach und verbreiten ihre Ideen eher ungerichtet an andere.
Nun gäbe es die Möglichkeit diese Tätigkeiten zu simulieren, dies erscheint allerdings in Anbetracht der Projekt-Idee wenig zielführend, daher ist die Wahl der Bewegungsart auf eine zufällige Wahl aus den maximal 9 erreichbaren Feldern gefallen, welche als eine ausreichende Abstraktion der anderen Tätigkeiten erscheint. Natürlich existieren auch Missionare in dem Sinn, dass sie ihre Idee einer breiten Masse zugänglich machen, allerdings muss ein zu überzeugender Mensch erst einmal (zum Beispiel durch Mundpropaganda oder zufällige Entdeckung, wenn der Missionar noch unbekannt ist) auf diese Missionare stoßen. Somit stellt dies kein Argument für die Einschränkung einer zufälligen Bewegung dar.

Die möglichen Bewegungen sind hier noch einmal grafisch in rot zu sehen dargestellt (die Möglichkeit still zu stehen bleibt bestehen, obwohl nicht markiert).



# 3 Implementation

## 3.1 Logik und Idee & Mensch

Die im Abschnitt Modellierung geschilderten Modellierungen sind eins zu eins in Software umgesetzt, mit der Ausnahme, dass ein Mensch/ eine Idee durch einen struct mit 4 Feldern repräsentiert wird, wobei noch ein weiteres Feld als Flag für eine empty-idea hinzukommt.

## 3.2 Parallelisierungsschema

Das Feld wird implementiert durch ein 2D-Array über den struct Ideas.

```
#define malloc_idea_matrix(name)
    Idea **name = (Idea **)malloc(num_rows * sizeof(Idea *));
    for (int i = 0; i < num_rows; ++i)
        name[i] = (Idea *)malloc(num_cols * sizeof(Idea));
```

Es werden *zwei* Felder erstellt:

```
malloc_idea_matrix(field)
malloc_idea_matrix(field_new)
```

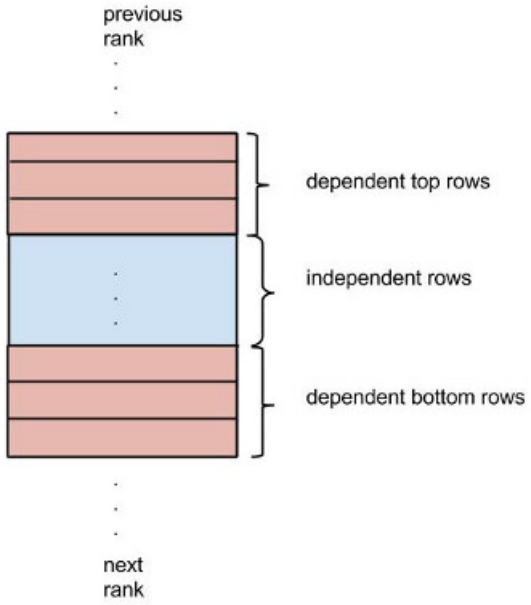
`field` wird daraufhin mit leeren Ideen initialisiert und an zufälligen Positionen werden Ideen gespawnt. Der Inhalt von `field` wird dann in `field_new` kopiert. Man braucht diese zwei Felder, damit eine Idee nur ein mal pro Runde ziehen kann: es wird über den Inhalt von `field` iteriert, der Zug Ideen wird in `field_new` vermerkt. Somit ist erreicht, dass eine die Idee nach rechts zog, im nächsten Iterationsschritt nicht noch einmal zum Zuge kommen. Das Kopieren der Felder ist wie folgt implementiert:

```
#define copy_field_into_field_new()
    for_every(i, num_rows, {
        for_every(j, num_cols, {
            field_new[i][j] = field[i][j];
          });
    });
```

Der Versuch dies mittels `memcpy` effizienter zu gestalten schlug fehl, da `memcpy` bei 2D-Arrays nur die Pointer der Arrays der zweiten Ebene kopiert werden. Dies ist allerdings nicht im Interesse des Programmes, da die Ideen by value kopiert werden müssen.

Die Aufteilung des 2D-Arrays auf die MPI-Prozesse funktioniert per horizontaler Spaltung des Feldes. Dies bringt die Vorteile einer schnellen Aufteilung und Reduktion der Anzahl an Kanten, welche kommuniziert werden müssen. Die Ränder bestehen aus *ghost rows*, wobei eine *ghost row* des aktuellen *ranks* der vorletzten Reihe des benachbarten *ranks* entspricht. Die Kommunikation besteht im Austausch der *real* und der *ghost rows* mit den jeweiligen Nachbarn.


Dies ermöglicht, jedem Prozess seine Züge unabhängig von den anderen Prozessen durchzuführen. Wenn eine Idee auf einer *real row* nach oben respektive nach unten zieht, so hat sie garantiert die Informationen, welche Werte sich in der Zielzelle aus dem Nachbarprozess befinden.
Aus diesem Vorgehen resultiert das Konzept der *dependent rows* an den Rändern und der *independent rows* in der Mitte.
*Dependend* bedeutet, dass die Ereignisse betreffend Ideen die anliegenden *ranks* betreffen können. Die drei äußersten *rows* an den Rändern sind jeweils *dependent*: Die beiden äußeren müssen direkt mit dem Nachbarn kommunizieren und die Drittäußerste kann durch Interaktion von Ideen mit der zweitäußersten *row* ebenso *dependet* sein.
Die restlichen *rows* hingegen tangieren die anderen *ranks* nicht innerhalb einer Runde, denn pro Runde kann eine Idee nur ein Feld weiter ziehen. Somit können diese *rows* als *independent* gelten.




Daraus folgt, dass die Berechnung der Ereignisse im Feldabschnitt der *independent rows* unabhängig von anderen *ranks* erfolgen kann, die der *dependent rows* allerdings nicht und somit zeitlich orchestriert werden muss. Diese Orchestration erfolgt in drei Schritten, diese Schritte sehen wie folgt aus und gelten für jeden *rank*.

1. Bewege die Ideen die sich auf den *independent rows* befinden.
2. Bewege die Ideen, die sich auf den *dependent top rows* befinden:
   1. Sende *top real* und *top ghost row* zu oberem Nachbarn.
   2. Empfange *top real* und *top ghost row* von unterem Nachbarn.
3. Analog wie Schritt 2 für die *bottom dependent rows*.

Die Kommunikation zwischen den *ranks* wird mittels `MPI_Isend` und `Irecv` realisiert. Dies hat den Vorteil, dass real und *ghost row* nacheinander versendet werden, ohne erst auf die Empfangsbestätigung für das erste Paket zu warten.

Im Code sieht das wie folgt aus:

```
#define send_ideas(ideas_arr, to, tag, req) \
  MPI_Isend(ideas_arr, num_cols, mpi_idea_type, to, tag, MPI_COMM_WORLD, &req)

#define receive_ideas_into(ideas_arr, from, tag, req) \
  MPI_Irecv(ideas_arr, num_cols, mpi_idea_type, from, tag, MPI_COMM_WORLD, &req)

#define send_top_rows(field) \
  /* send our first ghost row into the bottom real row of the previous rank */ \
  send_ideas(field[0], prev_rank, GHOST, req);                                 \
  /* send our first real row into the bottom ghost row of the previous rank */ \
  send_ideas(field[1], prev_rank, REAL, req2);                                 \

#define receive_into_bottom_rows(field) \
  /* receive first ghost row from next rank into our bottom real row */ \
  receive_ideas_into(field[num_rows-2], next_rank, GHOST, req);         \
  /* receive first real row from next rank into our bottom ghost row */ \
  receive_ideas_into(field[num_rows-1], next_rank, REAL, req2);         \
```

Der `mpi_idea_type` wird hierbei auf folgende Art definiert:

```
#define mpi_define_idea_type()                                                \
  int          blocklengths[5] = {1,1,1,1,1};                                 \
  MPI_Datatype types[5] = {MPI_INT,MPI_INT,MPI_INT, MPI_INT, MPI_INT};        \
  MPI_Datatype mpi_idea_type;                                                 \
  MPI_Aint     offsets[5];                                                    \
  offsets[0] = offsetof(Idea, a);                                             \
  offsets[1] = offsetof(Idea, b);                                             \
  offsets[2] = offsetof(Idea, c);                                             \
  offsets[3] = offsetof(Idea, h);                                             \
  offsets[4] = offsetof(Idea, empty);                                         \
    MPI_Type_create_struct(5, blocklengths, offsets, types, &mpi_idea_type); \
    MPI_Type_commit(&mpi_idea_type);                                          \
```

Die Kommunikation benutzt `field_new` zum Austausch der Daten. Am Ende einer Runde wird dieses pro Prozess in `field` kopiert.
Es ist entscheidend, die einzelnen Schritte per `MPI_Barrier` voneinander zu trennen, da ansonsten *race conditions* auftreten können.

### 3.3 Visualisierung

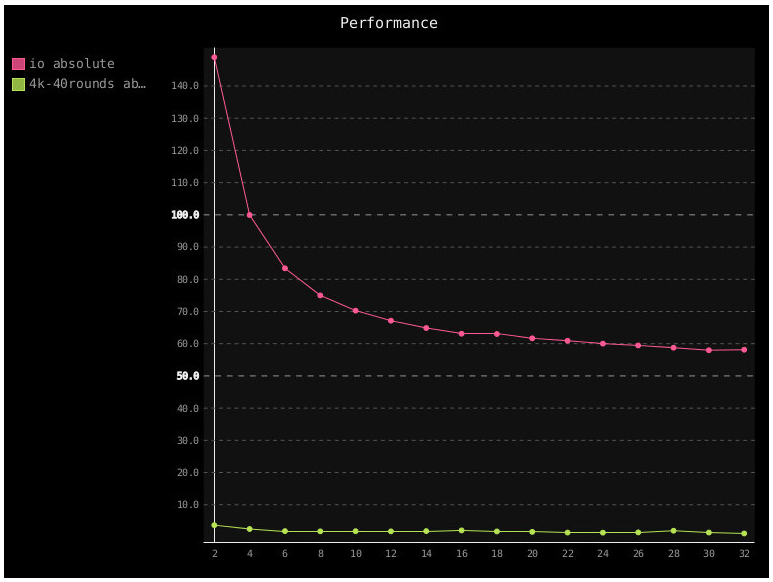
Die Visualisierung erfolgt lokal in einer mit Pygame-Implementation. Die Daten werden von C, sofern `#define DRAW` gilt, pro Prozess pro Runde in `src/draw/out/$round-$proc`

exportiert und im Nachhinein von Python eingelesen. Dies hat den entscheidenden Nachteil, dass zum einen der Nutzen des Clusters eingeschränkt ist: um diesen zu nutzen und gleichzeitig eine Visualisierung zu ermöglichen, müssen die Output-Files auf dem Cluster generiert und dann nach local kopiert werden. Dies geschieht per automatisiertem Deploy script, dass den Quellcode auf den cluster rsync, dort das Programm laufen lässt und darauf die Output-Files wieder per rsync zurück kopiert. Leider ist die Praktikabilität hiervon sehr eingeschränkt, da größere Dateimengen bewegt werden müssen. Somit wird der Performance-Gewinn durch den Cluster unterminiert. Ein weiterer Nachteil ist, dass es sich hierbei um keine Realtime-Visualisierung handelt, sondern nur ein Replay nach der Berechnung abgespielt wird.

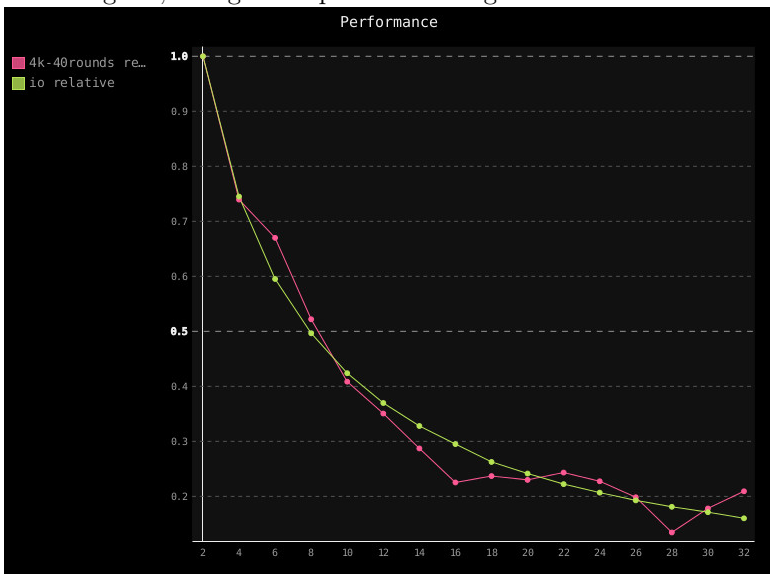
Das Problem eine Realtime-Visualisierung scheint aber kein triviales zu sein: Schließlich müssten die Daten in einem master rank pro Runde gesammelt werden um sie dann grafisch auszugeben. Somit wäre hier ein Bottleneck während der Berechnung geschaffen. Des weiteren war nicht klar, wie und ob so etwas überhaupt auf dem Cluster - per X-Forwarding - funktionieren würde, weshalb wir uns schlussendlich für die lokale Scripting-Variante entschieden haben.

# 4 Performance

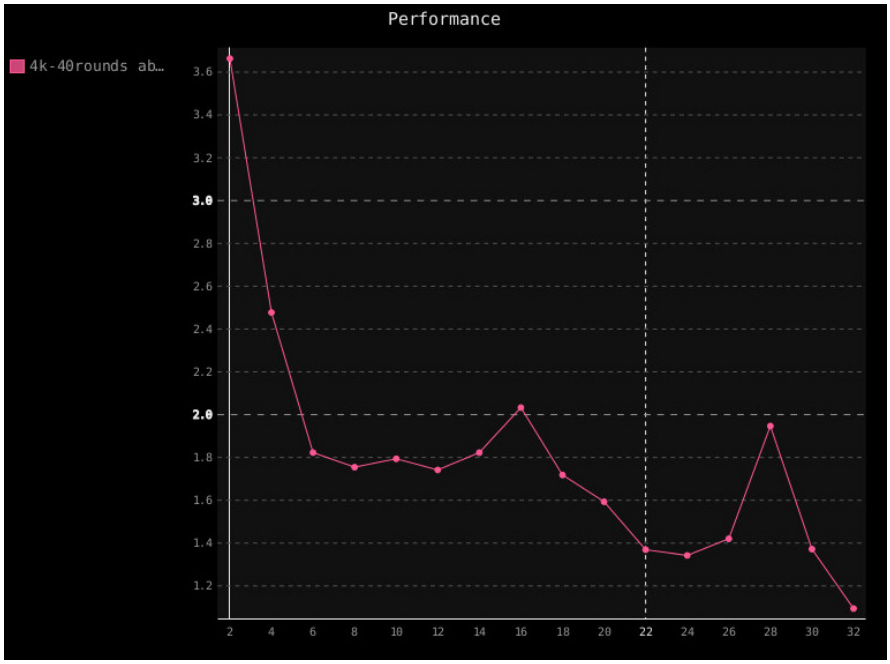
Wie zu erwarten, ist die Performance mit aktiviertem IO ca. um den Faktor 80 schlechter als ohne:




Der Speedup der beiden Varianten ähnelt sich aber dennoch. Ein Wert von 1 hieße eine lineare Relation zwischen Anzahl von Prozessen und absoluter Geschwindigkeit, weniger entsprechend weniger effektiv:



Der absolute Verlauf ohne IO sieht wie folgt aus:




Die Ausschläge sind hier relativ stark, sie treten allerdings immer und dann auf, wenn ein zusätzlicher Rechner innerhalb des Clusters hinzugeschaltet wird, da dann ein Overhead an Network-Kommunikation hinzukommt.
Die relativen Speedups und die Abnahme der Effizienz ist damit zu erklären, dass das Verhältnis von MPI-Kommunikation zu realer Rechenzeit steigt und insofern der Anteil an "teuren" Instruktionen ständig steigt.

## 5 Optimierung und andere Probleme

Folgende Optimierungen haben wir unter anderem im Laufe der Entwicklung vorgenommen:

1. `MPI_Isend/Recv` anstelle von `Send/Recv` einsetzen. Dies brachte ca. 10% Speedup.
2. Den `field_new` -> `field` Kopiervorgang nur einmal ausführen. Dies war Bestandteil eines tieferen Problems, welches uns viele Stunden durch Bugs beschäftigt hat, auch wenn es im Nachhinein trivial wirkt, die ersten Wochen hatten wir das `field` (und nicht `field_new` in der MPI-Kommunikation verschickt. Dies führte zu Korruptionen im Feld, die Ideen vermehrten sich manchmal, bei einer genügend hohen Ideendichte. Durch die Zufälligkeit der Simulation war es quasi unmöglich zu debuggen. Erst als wir eine Drop-In

Unit-test-artige Bewegungsmethode nutzten, welche Ideen immer nur nach unten gehen ließ, konnten wir den Fehler aufspüren.
3. Den schon zuvor angesprochenen Versuch, das Kopieren von `field_new` in `field` per `memcpy` umzusetzen. Auch dies funktionierte leider erstmal aus uns unbekannten Gründen, sodass der Eindruck entstand, die Simulation tue das, was sie tuen solle. Leider wurden auf diese Art aber nur Pointer auf die einzelnen rows kopiert. Dies führte zu weiteren schwerwiegenden Bugs.
4. Genauere und frühzeitige Differenzierung der einzelnen Logik-Fälle und daraus folgende Optimierung der Laufzeit der inneren Schleife.

Generell kann man den Entwicklungsprozess des MPI-Parts anging hauptsächlich als eine Bugsuche, die den Großteil der Zeit extrem in Anspruch war, beschreiben. Leider waren Debug-Tools hier nicht wirklich hilfreich, zum einen wegen der Involvierung des Zufalls, die zu nicht reproduzierbaren Ergebnissen führten und somit deterministische Fehlersuche unterminierte, zum anderen, weil erste spezifische Konstellationen des Feldes zu korrupierten Ergebnissen führten.

Dementsprechend kann man unsere Strategie bezüglich MPI, als die Funktionsfähigkeit hergestellt war als defensiv beschreiben. Das System erschien fragil und wir wollten endlich eine funktionierende Basis für das Testen der Simulation mit mehreren Prozessoren an sich haben. Daher ist eine Optimierung des MPI-Parts mehr als nicht ausgeschlossen.