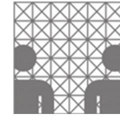
UHH Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG



Infektsimulation

Projekt im Rahmen des Moduls: Parallele Programmierung

von
Alexander Droste & Florian Flachsenberg

8. Oktober 2013

Inhalt

1. Einleitung/Problemstellung 2
2. Lösungsansatz 2 - 9
3. Parallelisierungsschema 9 - 15
4. Laufzeitmessungen/-analyse 15 - 17
5. Skalierbarkeit 17 - 18

Einleitung/Problemstellung

Die Infektsimulation ermittelt wie sich Infektionskrankheiten innerhalb einer “Bevölkerung” ausbreiten und verhalten. Während der Simulation agieren Wirte und Erreger. Dabei bewegen sich Wirte auf der Oberfläche eines Ellipsoids. In Ausnahmen kann das Ellipsoid auch die Form einer Kugel annehmen. Im Verlauf der Simulation können Wirte auf Basis von Position und ihrem Gesundheitstzustand Erreger aufeinander übertragen. Mögliche Stati der Wirte sind: Gesund, immun, infiziert, krank oder tod. Ob und wie sich Wirte anstecken hängt davon ab, wie groß die Ansteckungswahrscheinlichkeiten und Infektionsradii bezüglich verschiedener Übertragungswege des Erregers ausfallen bzw. ob ein Wirt gegen den Erreger zuvor Immunität entwickelt hat. Dabei können Wirte im Laufe der Simulation gegen mehrere Erreger immun werden. Wird ein Wirt infiziert, ergibt sich ein beschränkt variabler Krankheitsverlauf, welcher vom Erregertyp bestimmt ist. Neben Inkubations- und Krankheitsdauer wird zum Zeitpunkt der Infektion festgelegt, ob der Wirt Immunität entwickelt oder der Erreger zu einem tödlichen Verlauf führt.
Obwohl Wirte zu Beginn der Simulation nur mit einem Erregertyp infiziert sind, können Repräsentanten dieses Typs im Laufe der Simulation mutieren. Mutieren Erreger zu einem bestimmten Grad, sind Wirte die gegen den ursprünglichen Typ Immunität besitzen, gegenüber einer mutierten Form nicht mehr immun.
Die Simulation liefert Statistiken über die Anzahl an Erkrankungen, Infektionen und Verläufen mit tödlichem Ausgang insgesamt sowie zu jedem Zeitschritt.

Lösungsansatz

Zu allen Typen des Modells gibt es äquivalente C Typen durch typedefs.

```c
typedef struct {
   Percent mortality; /* percentage of mortality */
   Percent immunogenicity; /* percentage of immunogenicity */
   InfectionProb infection_prob; /* probability of infection */
   Percent outbreak_prob; /* probability infection leads to disease */
   Percent mutation_prob; /* probability of mutation on infection */

   unsigned short incubation_mean_dur; /* mean duration of incubation */
   unsigned short incubation_max_dur; /* incubation max duration */
   unsigned short illness_mean_dur; /* mean duration of illness */
   unsigned short illness_max_dur; /* max duration of illness */
   char infectious_incubation; /* infectious during incubation time (0,1) */

   unsigned short mutation_counter;
   unsigned short mutation_counter_threshold;
   unsigned long immunogenicity_factor; /* in case more than one agent is around they can be identified */
} Agent;
```

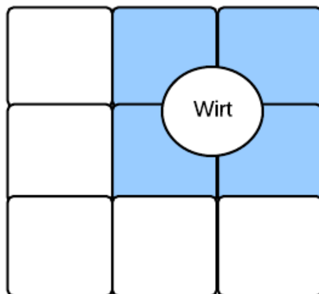
```
typedef struct {
   Agent agent;
   int state;
   unsigned short outbreak_time; /* start point of disease */
   unsigned short infection_time; /* time of infection *

   unsigned short end_of_disease; /* end point of disease */
   char will_ill; /* host will get disease after infection */
   char will_die; /* host will die on end of disease */
   char will_immune; /* host will get immune on end of disease */

   unsigned long immunities; /* agents host is immune against */
   unsigned short disappear_duration; /*time until host disappears after death*/

   Position pos; /* host position: x/y value */
   Field* field; /* field the host is positioned on */
   List_element* list_element; /* pointer to list element so hosts can be deleted from lists easily */
}Host;

typedef struct{
   Size size; //complete size of the world
   FieldMatrix* field_matrix; //matrix containing all fields
   Size full_size_one_field; //size of one field

//...more attributes

   unsigned residents;
   MatrixDimensions process_matrix_dimensions; //how the processes are arranged in the world
   unsigned number_of_neighbours;
}World;
```

Darüber hinaus werden eigens erstellte technisch bedingte Typen wie Linked-List Dynamic-Array und der später erläuterten Field-Matrix verwendet.

Für die Typen gibt es jeweilig Funktionen zum Erzeugen und Entfernen. host_create, host_delete, agent_new_stereotype, agent_delete, world_create, etc.. Einen Sonderfall stellt die agent_new_stereotype Funktion dar. Abhängig vom übergebenen Parameter des Typs “AgentType” können unterschiedliche Erregerarten erzeugt werden. Die Erregerarten unterscheiden sich dabei durch unterschiedliche Krankheitsverläufe, Übertragungswahrscheinlichkeiten und Mutationsverhalten. Die Anwendung der Simulation beschränkt sich zum Beginn auf die Erzeugung eines Typs. Sprich vor dem ersten Zeitschritt können Wirte mit genau einem Erregertyp infiziert sein. Einige Wirte müssen vor Beginn infiziert werden, um eine mögliche Ausbreitung zu simulieren.

Die Fläche der “World”, welche die Gesamtoberfläche definiert, setzt sich aus “Fields” zusammen, die in ihrer Größe ein Vielfaches des max. Infektionsradius darstellen. Diese werden

in einer Matrix gehalten, der “Field-Matrix”. Ein einzelnes Feld ist neben der Festlegung seiner eigenen Größe aus zwei Typen von Listen zusammengesetzt die getrennt kranke, infizierte und verstorbene oder gesunde und immune Wirte enthalten. Dazu kommen Counter welche die Anzahl der enthaltenen Elementtypen festhalten. Die Trennung in unterschiedliche Listen führt zu diversen Optimierungen, weil sich abhängig vom Status der Wirte Operationen ausschließen lassen, bzw. Wirte nicht betrachtet werden müssen. Soll bspw. für alle gesunden Wirte eines Feldes die Interaktion mit infizierten oder kranken Wirten in Reichweite des max. Infektradius durchgeführt werden, können die Operationen ohne vorzusortieren auf den Elementen der Liste, sprich den Wirten ausgeführt werden. Kranke Wirte können nicht infiziert werden. Diese müssen bezüglich einer möglichen Infektion nicht betrachtet werden. Nur Gesunde können sich an Kranken infizieren. Es werden für eine mögliche Infektion nur Gesunde betrachtet, die nur mit Kranken agieren. Weiter begünstigt wird dies durch kleine Feldgrößen. Im besten Fall muss während der Interaktion zwischen gesunden und kranken Wirten nur ein Feld betrachtet werden. Selbst für den ungünstigsten Fall, dass 4 Felder betrachtet werden müssen bleibt der Aufwand, unter der Bedingung gleicher Populationsdichte, für eine beliebige Anzahl der Welt enthaltenen Wirte konstant. Um dieses Maximum an Feldern vier zu garantieren entspricht die Achsenlänge eines Feldes immer mindestens dem Zweifachen des maximalen Infektionsradius.



[1]

Damit sich Wirte effizient zwischen Feldern bewegen können, bietet unsere Linked-List Implemetation eine “transfer”-Funktion, damit kein Speicher freigegeben und neu allokiert werden muss, falls Wirte im Zuge ihrer Bewegung das Feld wechseln müssen. Stattdessen werden den Pointern der Elemente einfach neue Vor-und Nachfolger zugewiesen. Ein Feld-Wechsel zieht allerdings mehr Operationen als die bloße Änderung der Feldzugehörigkeit nach sich. Abstrahiert zusammengefasst ist der Vorgang in “world_transfer_host_to_field”:

```
void world_transfer_host_to_field(Host* inc_host, Field* dest_field)
{
   List* dest_list;
   List* src_list;
   if(inc_host->state == HEALTHY || inc_host->state == IMMUNE){
      src_list = inc_host->field->healthy;
      inc_host->field->nof_healthy--;
      dest_list = dest_field->healthy;
      dest_field->nof_healthy++;
```

```
    }
    else{
        src_list = inc_host->field->infected;
        inc_host->field->nof_infected--;
        dest_list = dest_field->infected;
        dest_field->nof_infected++;
    }
    //destination field must be assigned to hosts attribute
    inc_host->field = dest_field;
    list_transfer_element(dest_list, src_list, inc_host->list_element);
}
```

Für die in Zeitschritten modellierte Simulation gibt es zwei Funktionen die für jeden Wirt pro Zeitschritt ausgeführt werden. Dabei handelt es sich um die Bewegung der Wirte sowie eine mögliche Änderung ihres Zustandes. Letzteres schließt den Vorgang möglicher Infektionen an Erregern anderer Wirte sowie den Fortschritt einer bereits vorhandenen Infektion und dem Eintreten oder Beenden von Krankheits- oder Todesphase ein.

Werden Wirte auf dem Ellipsoid in einem Zeitschritt bewegt, ist dies in eine beliebige Richtung um die Achsenlänge ihrer einnehmenden Fläche möglich. Um auszuschließen dass zwei Wirte auf der selben Position stehen, wird eine Kollionserkennung verwendet ("world_host_position_collides"). Im Bewegungsschritt wird solange gesucht, bis ein für den Wirt freies Feld in Reichweite gefunden wird. Dabei hat der Wirt auch die Möglichkeit auf seinem Feld stehen zubleiben.

Der Prozess einer möglichen Zustandsänderung ergibt sich wie folgt:
Ist ein Wirt gesund wird errechnet, welche Felder infektiöse Wirte enthalten können, die sich innerhalb der Reichweite des max. Infektionsradius befinden. Die Infektionsradii sind für alle Infektionswege während der Simulation konstant. Nur in diesen Feldern können sich infektiöse Wirte befinden, um einen Erreger auf den gesunden Wirt zu übertragen. Nachdem die Felder gesammelt wurden, werden diese durch eine Zufallsfunktion gemischt. Die Reihenfolge der anschließenden Betrachtung ist also zufällig. Bei der Abarbeitung eines Feldes werden alle Positionen der infektiösen Wirte, ebenfalls in zufälliger Reihenfolge, mit der des gesunden verglichen. Befindet sich ein infektiöser Wirt im Radius eines oder mehrerer Übertragungswege und unterschreitet der errechnete Zufallswert in Prozent den Wahrscheinlichkeitswert einer Infektion, wird der Infektionsprozess gestartet. Dabei kann der potentielle Infektionsvorgang für jeden Übertragungsweg einsetzen, falls die Wirte den Abstand des kleinsten Infektionsradius nicht überschreiten. Ist ein Wirt einmal infiziert ist die Abarbeitung der Phase für diesen abgeschlossen. Besitzt der Wirt Immunität gegenüber dem Erreger hat die Unterschreitung des Wahrscheinlichkeitswertes einer Infektion natürlich keine Folgen. Besitzt dieser allerdings keine Immunität wird der gesamte weitere Krankheitsverlauf zu diesem Zeitpunkt festgelegt und gestartet.
Handelt es sich bei der Betrachtung in dieser Phase hingegen um einen infizierten, kranken oder verstorbenen Wirt, kann abhängig von Status und Zeitpunkt nach einer Infektionsphase die

Krankheitsphase begonnen oder nach Ende der Krankheitsphase beendet werden, was die Möglichkeit nach sich zieht, dass der Wirt verstirbt.

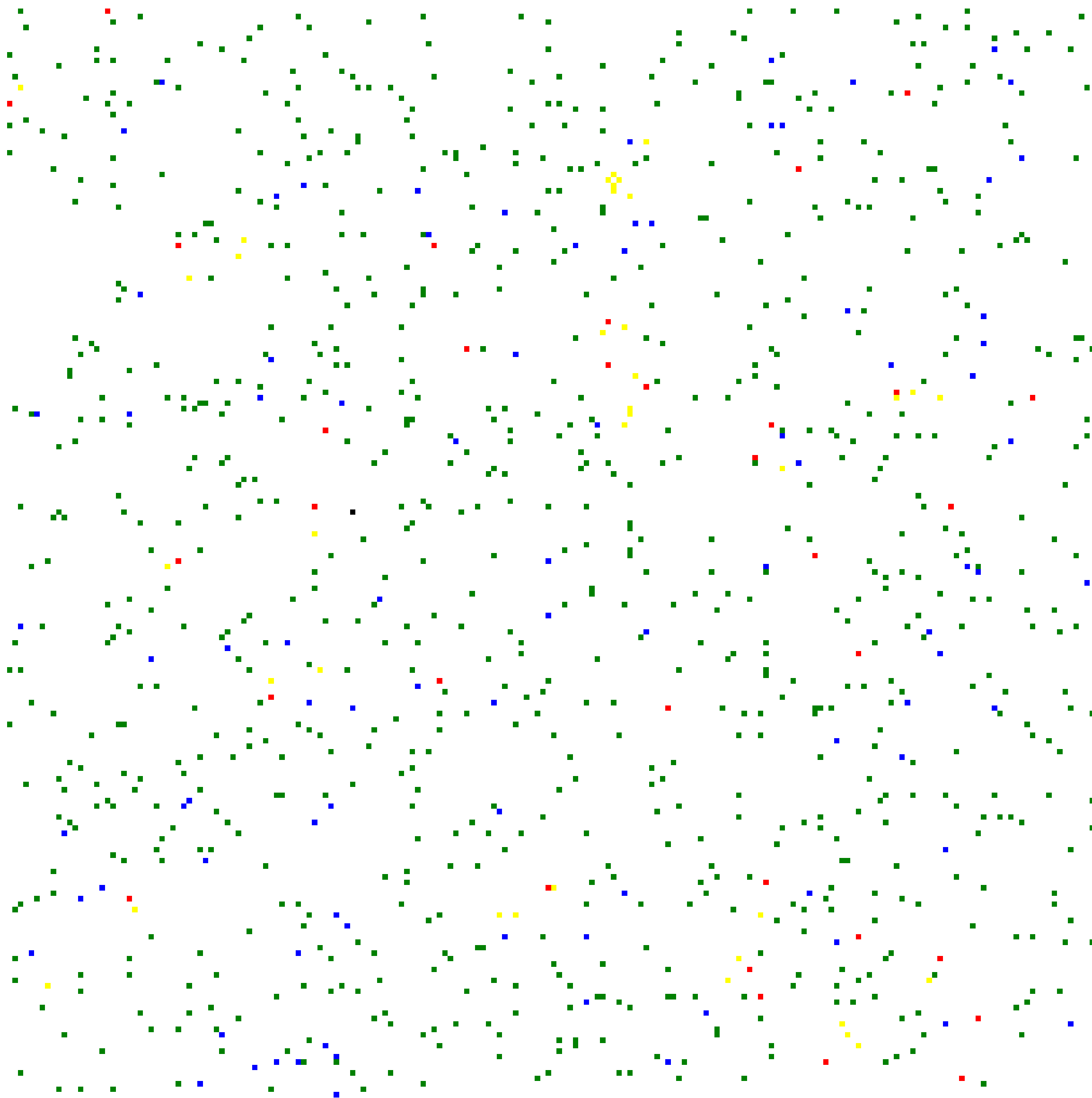
Random-Funktionen

Um realitätsähnliche Varianz in die Berechnungen der Simulation einzubeziehen, werden an vielen Stellen Funktionen verwendet, die ein pseudo-zufälliges Ergebnis produzieren. Eingesetzt wird dies bspw. bei der Bewegung der Wirte, Interaktion zwischen Wirten aus denen sich ein Infekt ergegeben kann, die Reihenfolge der Interaktionen oder bei Krankheitsverläufen. Um mit Funktionen zu arbeiten die eine “hochwertige” Zufallsverteilung produzieren verwendet die Simulation die GSL-Library. Sie bietet für dieses Spektrum die benötigte Funktionalität.

Visualisierung

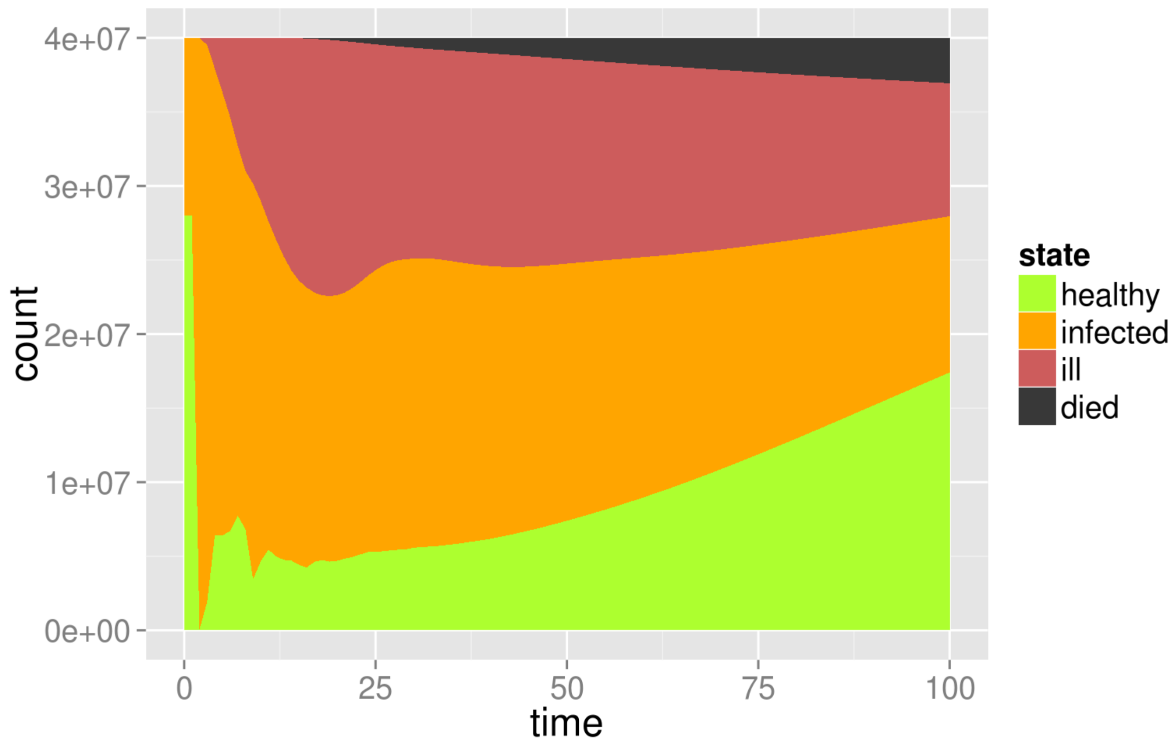
Auf Basis der ImageMagick - Library bietet die Simulation eine Visualisierungsmöglichkeit. Alle Position der Wirte werden dabei in einem Bild festgehalten. Abhängig von ihrem Status sind Wirte farblich unterschiedlich markiert. Durch Aneinanderreihen der Bilder mit FFmpeg lassen sich aus den Bildern die zu jedem Zeitschritt erstellt werden, Videos generieren.

[Legende: grün: gesund, gelb: infiziert, rot: krank, schwarz: verstorben]



[2]

Um den statistischen Verlauf zu visualisieren wurden Plots mit der Programmiersprache R erzeugt. Beispielverlauf:

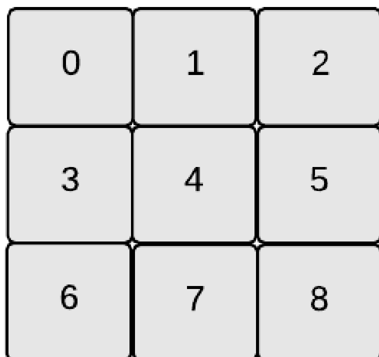


[3]

Parallelisierungsschema

Damit verschiedene Prozesse Aufgaben nebenläufig erledigen können, teilt die Simulation die Gesamtoberfläche in verschiedene Gebiete auf. Die Zuständigkeiten der verschiedenen Gebiete werden auf die Prozesse verteilt. Alle Aufgaben die sich innerhalb eines Gebiets ergeben, werden von dem zugeteilten Prozess bearbeitet. Bei Erzeugung der "World" welche bei mehreren Prozessen mehrfach ausgeführt wird, erstellt sich jeder Prozess den Teil, für welchen er später die Berechnungen durchführt. Sind mehrere Prozesse an der Simulation beteiligt, erfolgt die virtuelle Gebietszerlegung beispielhaft wie folgt:

[für 9 Prozesse]

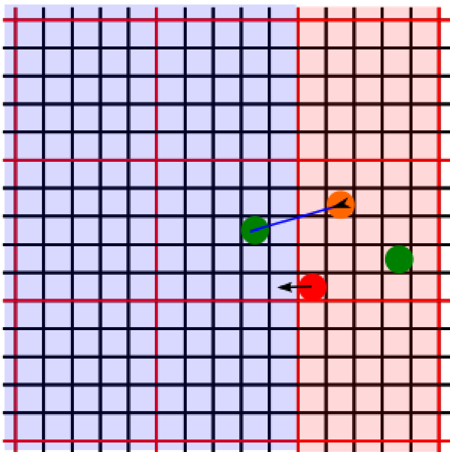


[4]

Beginnend oben links wird über die Spalten gezählt und mit darunterliegenden Reihen fortgesetzt. Die Zahl der Prozesse wird bei der Initialisierung der Simulation in ihre Primfaktoren zerlegt und so verteilt, dass Anzahl an Reihen und Spalten der Prozess-Matrix möglichst nah beieinander liegen. Desweiteren wird zu Beginn errechnet, von welchen Nachbarprozessen ein Prozess umgeben ist. Das ist im Weiteren relevant, um Daten von anderen Prozessen empfangen zu können.

Um festzustellen wo sich ein einzelnes Feld befindet wurden Funktionen implementiert, damit sich Feldkoordintaten der “Field-Matrix” nicht nur zu relativ zu ihrem Prozess sondern auch absolut unter Berücksichtigung aller Prozesse zuordnen lassen. Relative Koordinaten können in absolute und umgekehrt umgerechnet werden.

Kommunikation zwischen Prozessen ist in zwei Fällen nötig, wobei das gesamte Modell auf asynchroner Kommunikation basiert.



[5]

Der erste Fall tritt ein, falls ein Wirt im Zuge seiner Bewegung auf eine Position verschoben wird, für welche nicht der Prozess zuständig ist in dem er sich befindet. (siehe [5]: rot markierter Wirt) Dieser Vorgang ergibt sich im folgenden Ausschnitt von “host_move” wie folgt:

```
void host_move(Host *host)
{
//...
    unsigned target_rank = world_get_process_for_host(host);
    //if needed send host to other process
        if (rank != target_rank) {
            ++world.hosts_send_count;
            MPI_Request req;
            world_remove_host_from_field(host);
            MPI_Isend(host, 1, mpi_host, target_rank, tag_host, MPI_COMM_WORLD, &req);
```

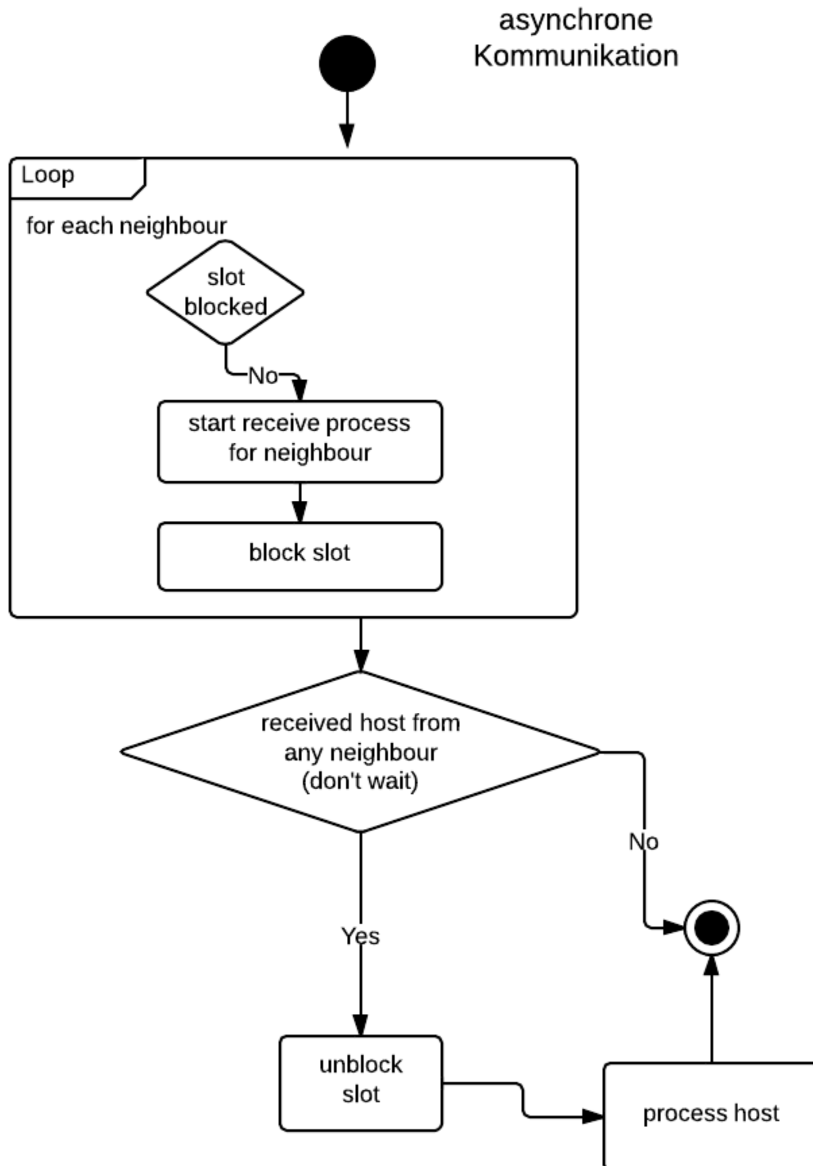
```
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        free(host);
        --world.residents;
         //receiving process checks for position collision
        return;
//..
 }
```

Es wird in “world_get_process_for_host” nach dem zuständigen Prozess des Wirts auf Basis seiner Position ermittelt und mit dem Prozess verglichen, welcher den Wirt gerade bearbeitet. Sind diese ungleich muss der Wirt verschickt werden. Mit der Kombination von MPI_Isend und MPI_Wait wird der Wirt versandt. Dies ist Vorteil gegenüber Kommunikationsmodellen bei denen gewartet werden muss, bis Daten die Daten vollständig im anderen Prozess empfangen werden. Der Grund warum mit MPI_Wait gewartet werden muss liegt darin, dass sonst der vom Wirt benötigte Speicher freigegeben wird, bevor dieser vollständig verschickt wurde.
Während der Bewegungsphase gibt es entsprechend eine Funktion zum Empfangen der Wirte. Sie basiert auf einem Polling-Verfahren und wird jedes mal ausgeführt, wenn ein Wirt innerhalb der Bewegungsphase betrachtet wird.

Der Polling-Mechanismus arbeitet so:

```
void world_polling_move(DynArray* neigh_processes)
{
  static MPI_Request request_host[8] =
  {MPI_REQUEST_NULL,MPI_REQUEST_NULL,MPI_REQUEST_NULL,MPI_REQUEST_NULL,
   MPI_REQUEST_NULL,MPI_REQUEST_NULL,MPI_REQUEST_NULL,MPI_REQUEST_NULL};

  static Host* recv_buff[8];
  int received_message = 0;
  static int check_for_message[8] = {1,1,1,1,1,1,1,1};
  int rank; MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  //check for all neighbours if host sent
  for (int i = 0; i < world.number_of_neighbours; ++i) {
    if (check_for_message[i]) {
      recv_buff[i] = calloc(1, sizeof(Host));
      MPI_Irecv(recv_buff[i], 1, mpi_host, neigh_processes->memory[i],
             tag_host, MPI_COMM_WORLD, &request_host[i]);
      check_for_message[i] = 0;
    }
  }
  //check if host was received
  static int i;
  MPI_Testany(8, request_host, &i, &received_message, MPI_STATUS_IGNORE);

  if (received_message) {
```

```
        ++world.hosts_received_count;
        Host* received_host = recv_buff[i];
        recv_buff[i] = NULL;
         world_add_host_to_field(received_host);

        request_host[i] = MPI_REQUEST_NULL;
        check_for_message[i] = 1;
        //received_message is set dyn. in MPI_Test

        ++world.residents;

        //in case received host is not on a free position
        if (world_host_position_collides(received_host)) {
            host_move(received_host);
        }
    }
}
```



[6]

Im ersten Schritt werden MPI_Irecv “Aufträge” ausgeführt um im Weiteren von allen Nachbar-Prozessen Wirte empfangen zu können. Nachdem zum Empfangen eines Wirts MPI_Irecv gerufen wurde, werden weitere Receive-Aufrufe für diesen Nachbarprozess solange geblockt, bis ein Wirt von diesem vollständig empfangen wurde. Konkret erfolgt das Blocken durch check_for_message[i] = 0. Würde pausenlos MPI_Irecv gerufen werden, könnte die Empfangsoperation niemals abschließen. Jeder neue Aufruf würde den vorigen überschreiben ohne dass dieser abgeschlossen wäre. Aufgrund 8 möglicher Nachbarprozesse wurden die Arraygrößen entsprechend gewählt. Static hält die Informationen auch zwischen Aufrufen der Polling-Funktion permanent im Speicher.
Hat der “world_polling_move” rufende Prozess von mindestens einem Nachbarn einen Wirt empfangen, wird dies von MPI_Testany erkannt. MPI_Testany hält mit Hilfe des Index i

desweiteren fest, von welchem Nachbarn der Wirt empfangen wurde. Die notwendigen Operationen zum Eingliedern in den eigenen Prozess schließen sich an. Beim nächsten Aufrufen von “world_polling_move” kann der nächste MPI_Irecv-Aufruf für diesen Nachbarprozess gestartet werden.

Vorteil dieses Kommunikationsmodells ist, dass Wirte fortlaufend dynamisch empfangen werden können. Es werden keine Annahmen über den genauen Zeitpunkt oder die Anzahl der versendeten Wirte gemacht. Zwischen dem Einleiten des Empfangsaufrufs und dem vollständigen Empfangen können Ressourcen weiterhin zur Berechnung der eigentlichen Simulationsaufgaben eingesetzt werden.

Der zweite Fall in dem Kommunikation zwischen Prozessen benötigt wird liegt vor, falls für einen gesunden Wirt Felder anderer Prozesse betrachtet werden müssen, um die Interaktion mit infizierten oder kranken Wirten zu ermöglichen, die einen Erreger übertragen könnten. (siehe [5]: grün & orange markierte Wirte) Im Kontext nebenläufiger Prozesse wird in “host_prepare_infection_process” ein Array mit absoluten Feldkoordinaten aller in Reihenfolge zu betrachtenden Felder spezifiziert. Stellt ein bearbeitender Prozess fest, dass der Wirt Interaktion in einem Feld benötigt, das in einem anderen Prozess liegt, wird dieser versendet.

```
void host_for_infection_process(HostForInfection* host_for_infection, unsigned time)
{
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  Coordinates local;
  int new_rank;

  while (host_for_infection->curr_field_coord > 0) {
     new_rank = world_get_process_for_absolute_coord(
       host_for_infection->field_coord[host_for_infection->curr_field_coord-1].x,
       host_for_infection->field_coord[host_for_infection->curr_field_coord-1].y);

     //if we stay => perform infection HERE
     if (rank == new_rank) {
       local = world_get_rel_for_abs_coord(
         host_for_infection->field_coord[host_for_infection->curr_field_coord-1].x,
           host_for_infection->field_coord[host_for_infection->curr_field_coord-1].y);

       host_infection_process_for_field(&(host_for_infection->host),
       get_field_from_matrix_for_coord(local.x, local.y), time);
       --host_for_infection->curr_field_coord;
       if (host_for_infection->host.state == INFECTED) {break;}
    }

    else {
       ++world.infects_send_count;
```

```
            MPI_Request req;
            MPI_Isend(host_for_infection, 1, mpi_hostForInfection, new_rank, tag_infect, MPI_COMM_WORLD, &req);
            // wait, otherwise we may leave the scope and the infection struct is gone
            MPI_Wait(&req, MPI_STATUS_IGNORE);
            --host_for_infection->curr_field_coord;
            return;
        }
}
```

Das Empfangen von Wirten während der Zustandsverarbeitungs-Phase verhält sich äquivalent zur Bewegungs-Phase. Für beide Phasen wurden MPI-Typen erstellt, um Wirte typisiert senden und empfangen zu können. Da für die Zustandsverarbeitungs-Phase ein Array der abzuarbeitenden Felder in Form der Koordinaten mit versendet werden muss, sind zwei Typen zum Versenden der Wirte notwendig. Innerhalb der Bewegungs-Phase ist das Koordinatenarray nicht nötig und würde zusätzlichen Kommunikationsaufwand bedeuten.

Um die Kommunikation während beider Phasen so früh wie möglich abzuschließen wird jeweilig immer mit der Abarbeitung der äußeren Felder begonnen. Liegt innerhalb der Phasen genügend Arbeitsaufwand zur Abarbeitung innerhalb der Prozesse vor, kann die Kommunikation mit Hilfe der Polling-Funktion im Zuge dessen praktisch in vollem Lauf ohne Wartezeiten bezüglich der Kommunikation ablaufen.

Um die verschiedenen Prozesse am Ende jeder Phase synchron abzuschließen, wird eine sog. “catch-phase” verwendet. In dieser wird solange iteriert, bis die Anzahl versendeter Wirte gleich der empfangenen Wirte ist. Die Kommunikations- und somit auch Verarbeitungsoperationen für diese Phase sind anschließend abgeschlossen und die nächste Phase kann beginnen.

Laufzeitmessungen

Seriell

2*10^6 Wirte, 30% infiziert (Influenza), Welt 2240x2240, 100 Schritte, 1 CPU

*excl. time	incl. time	calls	name
0.275s	3714.989s	1	main
83.397s	3696.286s	1	world_simulate
92.766s	1815.282s	206306163	host_move
749.474s	1622.773s	329216571	world_host_position_collides
62.287s	1712.964s	45874130	host_get_infection

870.169s	1590.330s	126822893	host_infection_process_for_field
974.675s	974.675s	14641878108	list_element_next
246.112s	246.112s	3864656587	list_element_get_value
205.159s	205.159s	3056162979	sq_euclidean

Wie ersichtlich wird beim seriellen Ablauf der Simulation am meisten Zeit in der Zustandsverarbeitung sowie in der Kollisionsdetektion während der Bewegungsphase benötigt. Letzteres wurde nach dem letzten Stand des Profilings nochmals überarbeitet. Statt einen Abgleich mit jedem anderen Wirt durchzuführen, ob sich dieser auf der gleichen Position befindet, wird nun direkt das “Field” nach der Belegtheit einer Position befragt. Jedes “Field” enthält dazu ein Array, um für jede Position festzuhalten, ob diese belegt ist.

Parallel (Vampirtrace - Profiling)
8*10^7 Wirte, 30% infiziert (Influenza), Welt 14140x14140, 100 Schritte, 40 CPU

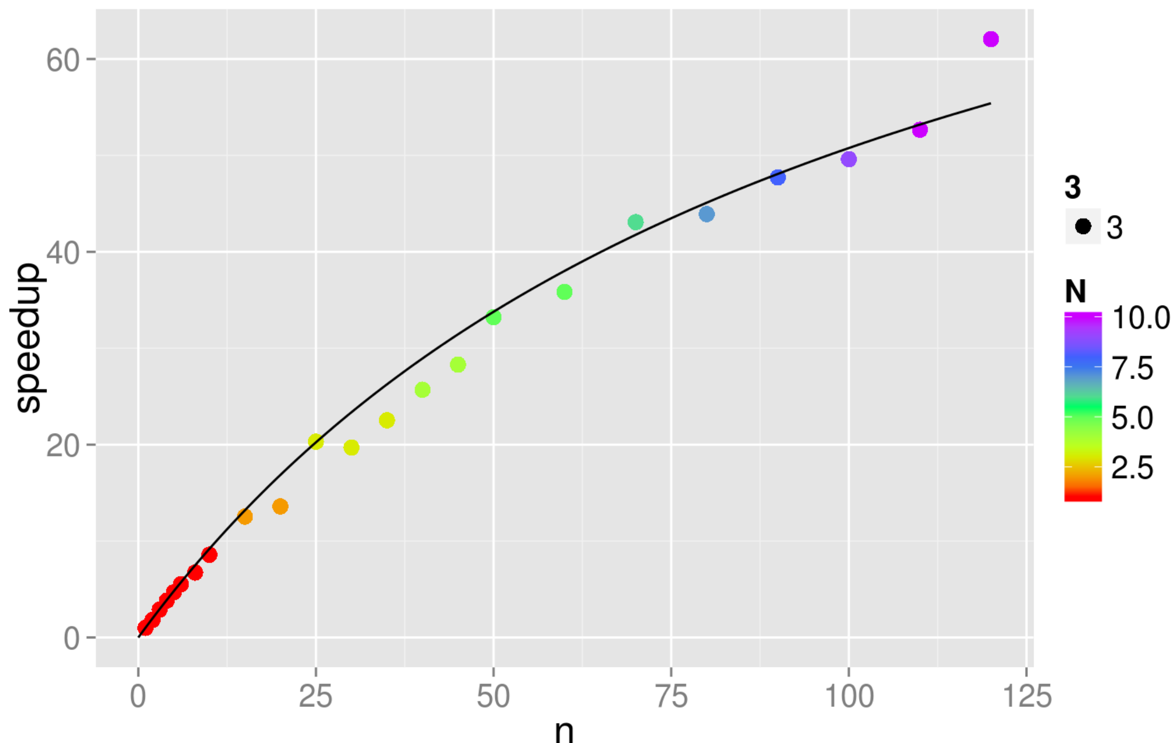
*excl. time	incl. time	name
0.315s	7049.633s	main
0.626s	7023.799s	world_simulate
2812.619s	2812.619s	MPI_Allreduce
0.919s	2222.823s	world_move_both_types_in_field
12.785s	552.348s	world_move_healthy_in_field
44.976s	1669.556s	world_move_infected_in_field
127.571s	1995.912s	host_move
768.119s	1683.817s	world_host_position_collides
0.929s	1980.854s	world_process_both_types_in_field
18.102s	1805.114s	world_process_healthy_in_field
46.598s	1745.561s	host_prepare_infection_process
41.494s	1664.284s	host_for_infection_process
871.535s	1573.085s	host_infection_process_for_field

Bei 40 Prozessen und gegebenem Aufwand fällt neben den zuvor gennanten Funktionen, welche die meiste Zeit benötigen, der dazugekommene Aufwand für die MPI-Kommunikation auf.

Schließt ein Prozess eine Phase früher ab als andere iteriert dieser in einer while-Schleife in welcher geprüft wird, ob die Zahl der versendeten gleich der empfangenen Wirte ist. Haben Prozesse mehr Aufwand müssen die anderen entsprechend warten, um eine Phase synchron abzuschließen, wodurch die benötigte Zeit für MPI_Allreduce zu erklären ist. Der benötigte Aufwand des ständigen Aufrufens der Polling-Funktion ist im Verhältnis zum Gesamtaufwand marginal und somit in der Statistik vernachlässigbar.

Strong-Scaling

4*10^7 Wirte, 30% infiziert (Influenza), Welt 10000x10000, 100 Schritte

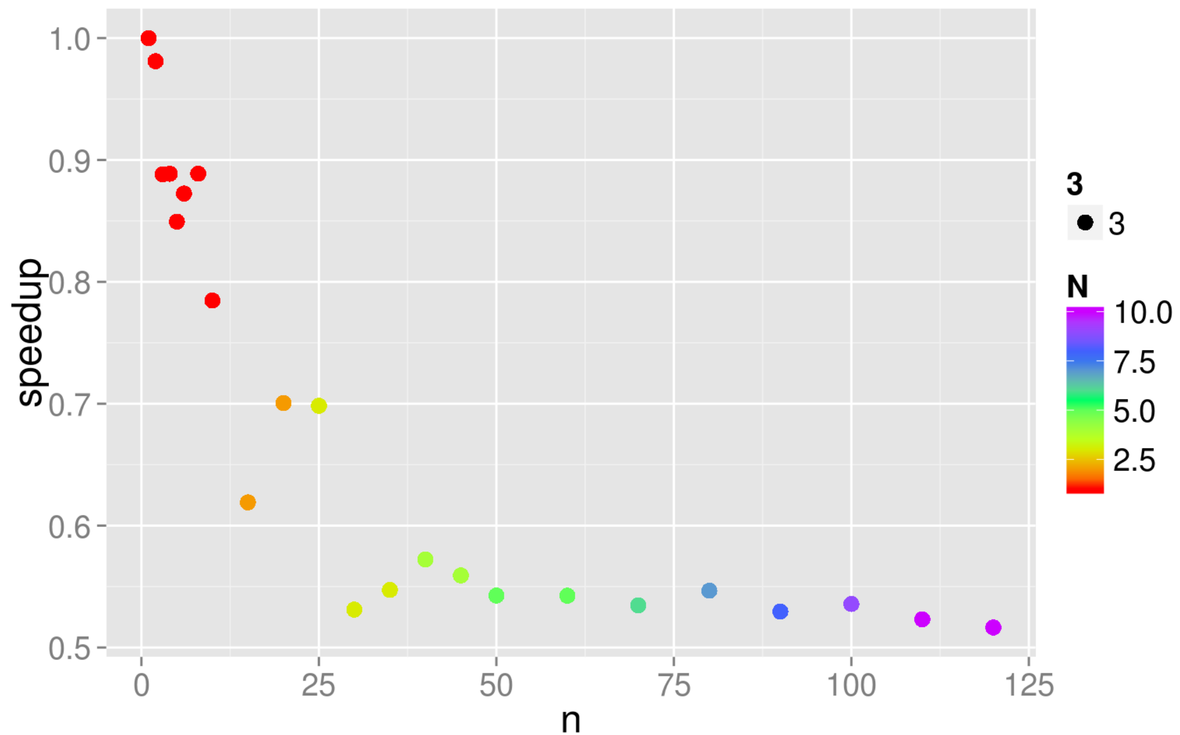


[7]

Die parallele Version der Simulation skaliert bis 10 Prozessoren linear. Ab ~11 Prozessoren nimmt der Speedup im Verhältnis ab. Zu erklären ist dies dadurch, dass die Dauer zur Abarbeitung der eigentlichen Aufgagen geringer wird als die, welche zur MPI-Kommunikation benötigt wird. Ungeachtet dessen skaliert die Simulation, wenn auch in geringerem Maße, weiterhin für n > 100 und kann dann selbst Größenordnungen wie 4*10^7 Wirte bei 100 Zeitschritten in unter 10min verarbeiten. Würde das Modell weiter ausgebaut werden, so dass sich für die Simulation mehr Arbeitsaufwand für die einzelnen Phasen ergeben würde, wäre der Speedup auch für größere n linear. Mit steigendem Aufwand verschiebt sich also entsprechend der Punkt bis für welche n die Simulation linear skaliert.

Weak-Scaling

2*10^6 Wirte je Prozess, Dichte 0.4 Wirte/Feld, 30% infiziert (Influenza), 100 Schritte



[8]

Mit steigender Prozessorzahl bei gleichbleibender Arbeitsmenge, nimmt die Performance aufgrund erhöhter Kommunikation am stärksten bis 26 n ab. Die Performance sinkt bis dahin um ~50%. Ab n > 25 sinkt der Performanceverlust nur noch marginal. Von 26 -125 beträgt dieser weniger als 5%. Man könnte also sagen, dass die Performance aufgrund erhöhter Kommunikation ab ~26 Prozessen stabil bleibt.