

# PROJEKTAUFGABE 1

## pixCheckTangle

RAPHAEL DRECHSLER

### INHALTSVERZEICHNIS

1	Problembeschreibung	2
2	Beschreibung der Implementierung	2
2.1	gridGenerator: Generieren der Files . . . . .	2
2.2	pixCheckTangle: Beschreibung des Sequentiellen Algorithmus . . . . .	3
2.3	pixCheckTangle: Beschreibung des Cluster-Algorithmus . . . . .	4
2.4	pixCheckTangle: Funktion für das Abarbeiten eines (Teil-)Rasters . . . . .	6
2.5	pixCheckTangle: Funktion für das Festlegen potentieller Rechtecke . . . . .	7
2.6	Unterschied zum "Gibt es ein Rechteck?"-Algorithmus . . . . .	8
3	Messungen und Interpretation	9
3.1	Vorgehen beim Messen . . . . .	9
3.2	Messergebnisse und Interpretation . . . . .	10
4	Unterschied zum "Gibt es ein Rechteck?"-Algorithmus und Fazit	15
4.1	Fazit . . . . .	15
4.2	Unterschied zum "Gibt es ein Rechteck?"-Algorithmus . . . . .	16
A	Anhang	17
A.1	Code: gridGenerator.c . . . . .	17
A.2	Code: pixCheckTangle.c . . . . .	17
A.3	Code: hostfileGenerator.c . . . . .	17
A.4	Code: bsCreateHostfiles.sh . . . . .	17
A.5	Code: bsRunGrids.sh . . . . .	18
A.6	README: Beispielhafte Ausführung . . . . .	18
A.7	Alle Messergebnisse . . . . .	19

## 1 PROBLEMBESCHREIBUNG

Gegeben ist ein quadratisches Raster von  $n \times n$  Feldern. Jedes der Felder kann schwarz oder weiß gefärbt sein. Es ist ein Algorithmus zu implementieren, welcher

- eine Einfache Eingabe eines solche Rasters erlaubt
- als Rechteck zusammenhängende Felder im Raster erkennt
- die resultierenden Rechtecke ausgibt

Dabei soll der Algorithmus für die Ausführung auf dem Cluster-System implementiert werden.

Anschließend soll mittels Laufzeitmessungen die Effizienz der Parallelisierung betrachtet werden. Dafür ist es erforderlich den Algorithmus als sequentiellen Algorithmus ausführen zu können.

## 2 BESCHREIBUNG DER IMPLEMENTIERUNG

Die Implementierung ist in zwei Programmen umgesetzt.

- *gridGenerator.c*: Programm zum generieren von *.txt*-Dateien, in denen das Raster gespeichert ist.
- *pixCheckTangle.c*: Programm zur Überprüfung des als *.txt*-Datei übergebenen Rasters auf Rechtecke.

Eine beispielhafte Abfolge der Konsolen-Befehle für die Ausführung ist im Anhang A6 aufgezeigt.

In den Folgenden Abschnitten wird die Funktionalität der Programme beschrieben.

### 2.1 gridGenerator: Generieren der Files

Das Programm *gridGenerator.c* wird per mpirun-Befehl über die Konsole aufgerufen.

---

```
mpirun gridGenerator.c [gridfile.txt]
```

---

Wird dabei eine zuvor durch den *gridGenerator* erzeugte *.txt*-Datei als Parameter angegeben, kann diese Datei bearbeitet werden. Andernfalls wird eine neue Datei erstellt. Der Programm-Ablauf ist im Folgenden als Nassi-Shneiderman-Diagramm dargestellt.

Der vollständige Code findet sich im Abgabeordner des Projektes.

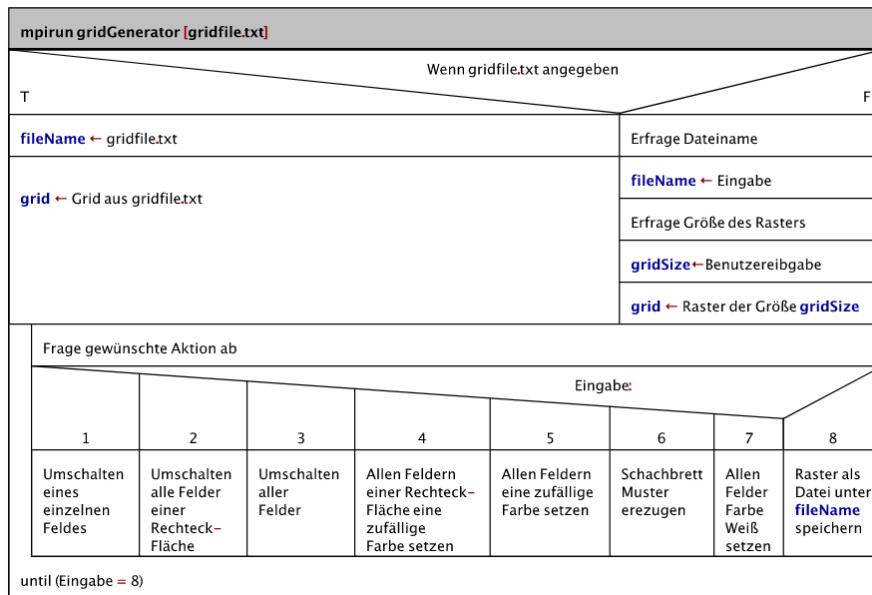


Abbildung 1: Funktionalität von `gridGenerator.c` dargestellt als Nassi-Shneiderman-Diagramm

## 2.2 pixCheckTangle: Beschreibung des Sequentiellen Algorithmus

Das Programm `pixCheckTangle.c` wird per `mpirun`-Befehl über die Konsole aufgerufen. Dabei ist als dem Aufruf eine Raster-Textdatei als Argument zu übergeben. Die übergebene Datei wird dann auf Rechtecke überprüft.

---

`mpirun pixCheckTangle.c gridfile.txt`

---

Für den Fall, dass nur ein Prozessor an der Ausführung des Programmes beteiligt ist, wird die sequentielle Variante des Algorithmus ausgeführt. Diese ist im Folgenden als BPMN-Daigramm dargestellt.

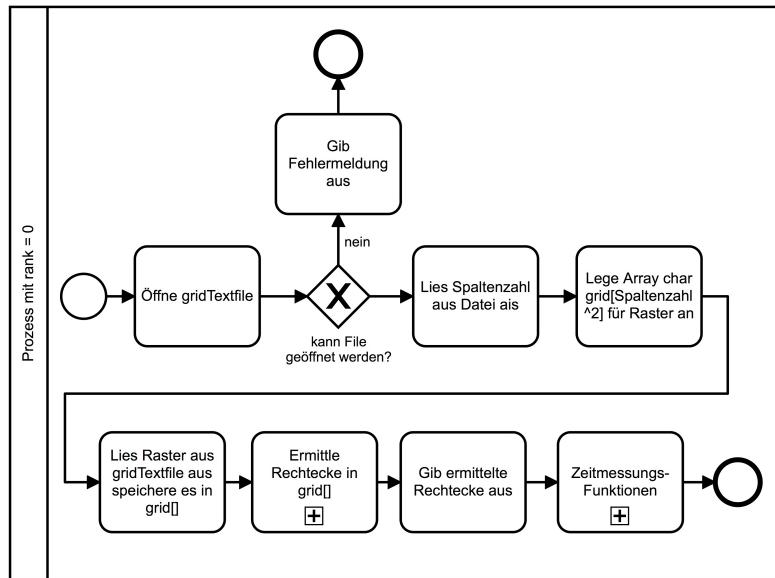


Abbildung 2: Sequentieller Ablauf von `pixCheckTangle.c` dargestellt als BPMN-Diagramm

Der vollständige Code zum Algorithmus ist im Projektordner enthalten. Die Hauptfunktionalität findet sich dabei in dem Codeabschnitt, welcher ausgeführt wird,

wenn der ausführende Prozess den Rang 0 hat und es nur einen Prozess gibt. Die Funktionalität zum Ermitteln der Rechtecke, wird im Kapitel 2.4 näher beschrieben. Nach welchem Prinzip die Zeitmessung erfolgen, wird in Kapitel 3.1 beschrieben.

### 2.3 pixCheckTangle: Beschreibung des Cluster-Algorithmus

Für den Fall, dass mehrere Prozessoren an der Abarbeitung des Programmes beteiligt sind, wird die Cluster-Variante des Algorithmus ausgeführt. Dabei übernimmt der Prozess mit dem Rang 0 die Rolle des Master-Prozesses. Alle übrigen Prozesse übernehmen die Rolle von Worker-Prozessen. Das Raster wird vom Master-Prozess in mehrere Teile zerlegt. Jeder Worker-Prozess übernimmt dann die Abarbeitung eines Teil-Rasters.

#### Beispiel für Verarbeitung durch Cluster-Prozess

Beispielsweise wird das folgende Raster betrachtet. Das Zeichen # repräsentiert in der folgenden Grafik ein schwarzes Feld, weiße Felder werden durch Leerzeichen dargestellt.

	1	2	3	4	5	6	7
1	#	#	#	#	#		
2				#	#		
3	#	#		#	#		
4	#	#		#	#		
5	#	#		#	#		
6	#	#		#	#		
7	#	#		#	#	#	#

Abbildung 3: Ein Raster-Beispiel

Für den Fall, dass an der Abarbeitung des Programmes 4 Prozesse beteiligt sind, wird das Raster vom Master-Prozess in drei Teil-Raster geteilt. Die jeweiligen Teile werden dann den Worker-Prozessen zugewiesen und von diesen abgearbeitet.

	1	2	3	4	5	6	7	
1	#	#	#	#	#			Teil-Raster 1
2			#	#				-> Bearbeitet WorkerProzess 1
3	#	#		#	#			Teil-Raster 2
4	#	#		#	#			-> Bearbeitet WorkerProzess 2
5	#	#		#	#			Teil-Raster 3
6		#	#	#	#			-> Bearbeitet WorkerProzess 3
7	#	#		#	#	#	#	

Abbildung 4: Aufteilung des Beispiel-Rasters bei 4 Prozessen

Die Worker-Prozesse können nun feststellen, ob es sich bei einzelnen/zusammenhängenden Pixeln (im Folgenden als Figur bezeichnet) um Rechtecke handelt oder nicht. In der Folgenden Grafik, werden Figuren, die kein Rechteck darstellen mit einem X dargestellt. Rechtecke werden weiterhin durch das Zeichen # repräsentiert.

WorkerProzess 1								WorkerProzess 2								WorkerProzess 3							
	1	2	3	4	5	6	7		1	2	3	4	5	6	7		1	2	3	4	5	6	7
1	#	#	#	#	#			3	#	#	#	#				5	#	#	#	#			
2			#	#				4	#	#	#	#				6	#	#	X				

Abbildung 5: Worker-Prozesse identifizieren Nicht-Rechteck-Figuren

Wenn ein Rechteck einen Rand des Teil-Rasters berührt und dieses Teil-Raster im gesamten Raster an ein weiteres Teil-Raster angrenzt, so könnte die Figur im angrenzenden Teil-Raster fortgesetzt werden. In diesem Fall ist also durch den Worker-Prozess keine finale Aussage darüber möglich, ob es sich tatsächlich um ein Rechteck handelt. Entsprechende Rechtecke (Im Weiteren als potentielle Rechtecke bezeichnet) werden in der folgenden Grafik als ? gekennzeichnet.

Das Ergebnis der Abarbeitung der Teil-Raster in den Worker-Prozessen sieht also wie folgt aus:

WorkerProzess 1							WorkerProzess 2							WorkerProzess 3									
	1	2	3	4	5	6	7		1	2	3	4	5	6	7		1	2	3	4	5	6	7
1	#	#	#	#	#			3	?	?	?	?	?		5	?	?	?	?	?			
2			?	?				4	?	?	?	?			6	#	#	X					

Abbildung 6: Worker-Prozesse identifizieren Rechtecke und potentielle Rechtecke

Nach dem Verarbeiten der Teil-Raster durch die Worker-Prozesse, muss die finale Überprüfung der potentiellen Rechtecke vom Master-Prozess übernommen werden.

	1	2	3	4	5	6	7		1	2	3	4	5	6	7
1	#	#	#	#	#			1	#	#	#	#	#		
2			?	?				2		#	#				
3	?	?	?	?				3	X	X		#	#		
4	?	?	?	?				4	X	X	#	#			
5	?	?	?	?				5	X	#	#	#			
6	#	#	X					6	#	#	X				
7	#	#	X	X	X			7	#	#	X	X	X		

Abbildung 7: Ein Raster-Beispiel

Damit liegt dem Master-Prozess nun das vollständig überprüfte Raster vor. Die Einzelnen Rechtecke können nun dem Benutzer ausgegeben werden.

### Kommunikation und Ablauf im Cluster-Prozess

Wie in der umgesetzten Implementierung die Kommunikation zwischen den Prozessen verläuft, sei im Folgenden als BPMN-Diagramm dargestellt.

Der vollständige Code findet sich im Projektordner. Die wesentlichen Funktionalitäten für den Master-Prozess finden sich dabei in dem Codeabschnitt, welcher ausgeführt wird, wenn der ausführende Prozess den Rang 0 hat und es mehr als einen aktiven Prozess gibt. Der Code für die Worker-Prozesse findet sich in dem Code-Abschnitt, der ausgeführt wird, wenn der Rang des Prozesses nicht 0 ist.

Die Funktionalität zum Ermitteln der Rechtecke, wird im Kapitel 2.4, die Funktionen zum prüfen der potentiellen Rechtecke in Kapitel 2.5 näher beschrieben. Nach welchem Prinzip die Zeitmessung erfolgen, wird in Kapitel 3.1 beschrieben.

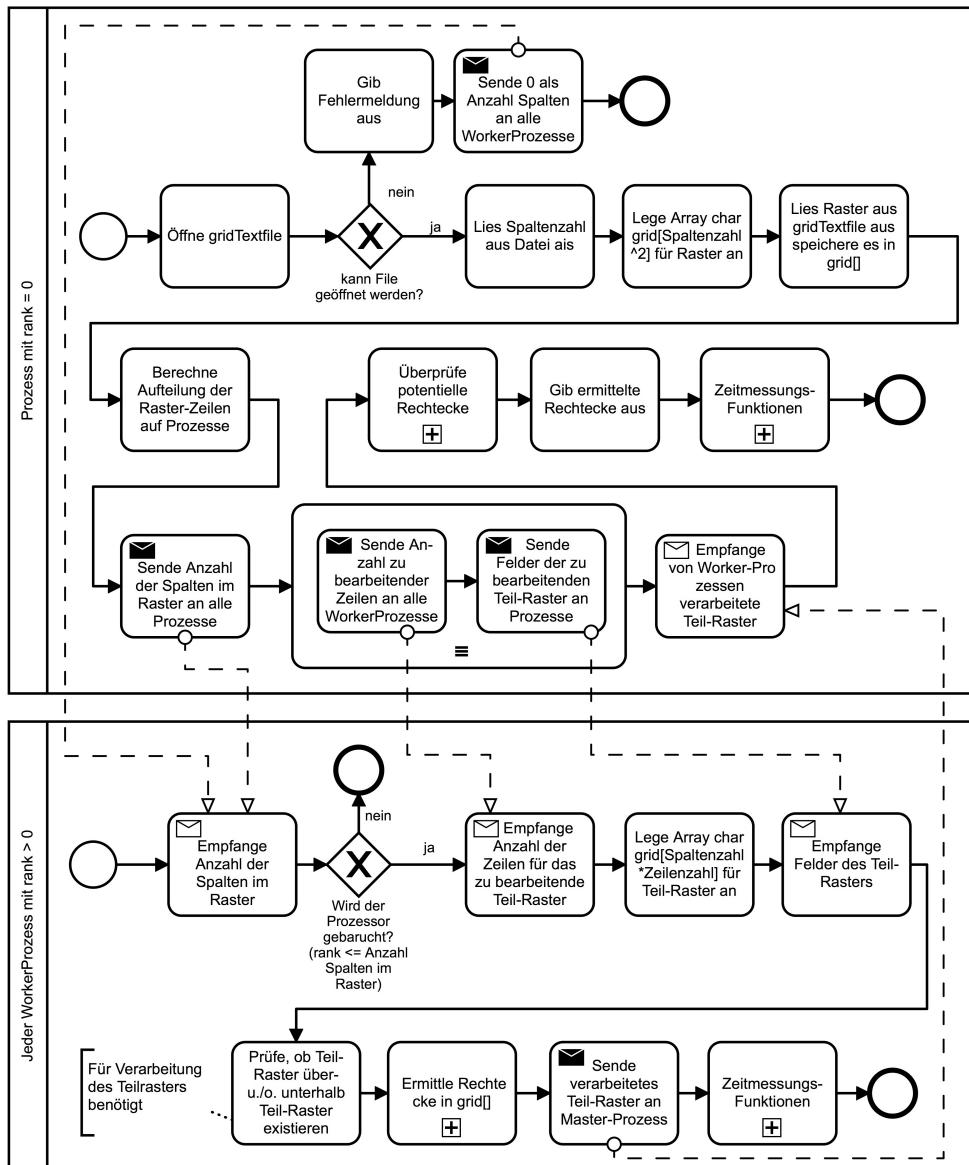


Abbildung 8: Ablauf des Cluster-Prozesses von *pixCheckTangle.c* dargestellt als BPMN-Diagramm

## 2.4 pixCheckTangle: Funktion für das Abarbeiten eines (Teil-)Rasters

Im Wesentlichen folgt das Vorgehen für die Unterscheidung zwischen Rechtecken und Nicht-Rechteck-Figuren dem folgenden Prinzip:

1. Erkenne Spalten-Reichweite der betrachteten Figur anhand der erster Zeile
2. Ist In Zeile über der Figur innerhalb der Spalten-Reichweite ein Feld schwarz, ist die Figur kein Rechteck
3. Enthält eine Zeile unterhalb der ersten innerhalb der Spalten-Reichweite der Figur schwarze und weiße Felder, ist die Figur kein Rechteck
4. Grenzt an eine schwarze Zeile der Figur (von Spalten-Reichweite eingegrenzt) ein schwarzes Feld, ist die Figur kein Rechteck

Zusätzlich ist dabei zu berücksichtigen, ob es sich bei einem erkannten Rechteck, wie weiter oben beschrieben, nur um ein potentielles Rechteck ist.

Die Implementierung der Funktion folgt danach dem folgenden, als Pseudo-Code

dargestellten Algorithmus:

---

```

Daten : Raster, InfoZuAngenzedeteTeilraster
Ergebnis : RasterVerarbeitet

1 für jedes Feld in Raster zeilenweise gelesen tue
2   wenn Feld ist Schwarz und nicht markiert dann
3     FigurIstRechteck ← wahr;
4     startZeile ← aktuelleZeile;
5     startSpalte ← aktuelleSpalte;
6     endeZeile ← aktuelleZeile;
7     endeSpalte ← Spalte von letztem Feld in startZeile, das mit aktuellem Feld
      zusammenhängt;
8     wenn In Zeile über startZeile im Bereich von startSpalte bis endeSpalte ist
       ein schwarzes Feld dann
9       FigurIstRechteck ← falsch;
10    sonst
11      RechteckGeprueft ← falsch;
12      solange FigurIstRechteck und nicht RechteckGeprueft tue
13        wenn Zeile unter endeZeile enthält schwarze und weiße Felder dann
14          FigurIstRechteck ← falsch;
15        sonst
16          wenn Zeile unter endeZeile enthält nur schwarze Felder dann
17            wenn In Zeile unter endeZeile ist feld links von startSpalte
              oder/und rechts endeSpalte von schwarz dann
18              FigurIstRechteck ← falsch;
19            sonst
20              endeZeile ← endeZeile + 1;
21            Ende
22          sonst
23            RechteckGeprueft ← wahr;
24          Ende
25      Ende
26    Ende
27  Ende
28  wenn FigurIstRechteck dann
29    wenn Figur liegt an Raster-Rand, der an weiterem Teil-Raster angrenzt dann
30      Markiere alle schwarzen Felder innerhalb der Reichweite, die von
        startZeile,endeZeile,startSpalte und endeSpalte aufgespannt
        wird als potentielles Rechteck-Feld;
31    sonst
32      Markiere alle schwarzen Felder innerhalb der Reichweite, die von
        startZeile,endeZeile,startSpalte und endeSpalte aufgespannt
        wird als Rechteck-Feld;
33    Ende
34  sonst
35    Markiere alle schwarzen Felder innerhalb der Reichweite, die von
        startZeile,endeZeile,startSpalte und endeSpalte aufgespannt wird
        als Nicht-Rechteck-Feld;
36  Ende
37 Ende
38 Ende

```

---

Der Code für die Funktionalität findet sich im Projektordner im Programm *pixCheckTangle.c* in der Funktion *processSubgrid*.

## 2.5 pixCheckTangle: Funktion für das Festlegen potentieller Rechtecke

Für die Funktionalität zum Überprüfen potentieller Rechtecke wird die im Folgenden per Pseudo-Code beschriebene Vorgehensweise verwendet. Die Kernidee ist

dabei nur noch stichprobenartige Prüfungen zu machen, anstatt die Schritte aus dem oben aufgezeigte Prozess zu wiederholen.

---

```

Daten : Raster,zuPruefendeZeile InfoZuRasterumbrueche
Ergebnis : RasterVerarbeitet
1 für Jedes schwarze Feld in zuPruefendeZeile mit Markierung =
  potentielles-Rechteck-Feld tue
2   startZeile ← aktuelleZeile;
3   startSpalte ← aktuelleSpalte;
4   endeZeile ← aktuelleZeile;
5   endeSpalte ← Spalte von letztem Feld in startZeile, das mit aktuellem Feld
     zusammenhängt;
6   wenn zuPruefendeZeile ist Zeile oberhalb von Umbruch dann
7     wenn In Zeile oberhalb startZeile ist in Reichweite von startSpalte bis
       endeSpalte mindestens ein schwarzes Feld dann
8       zeilenMarker ← MIN(End-Zeile der Figur, Zeile vor folgendem
          Teil-Raster-Umbruch) ;
9       Markiere alle schwarzen Felder innerhalb der Reichweite, die von
          startZeile,zeilenMarker,startSpalte und endeSpalte aufgespannt
          wird als Nicht-Rechteck-Feld;
10      Fahre bei Feld nach der endeSpalte fort.
11   wenn hinter zuPruefendeZeile folgt min 1 Teil-Raster-Umbruch dann
12     solange Ein Teilaraster-Bruch folgt tue
13       wenn Figur setzt unter Teilaraster-Bruch fort dann
14         wenn In erster Zeile nach Teilaraster-Bruch ist die Laenge der Figur
           ungleich nicht mit der bisherigen Spalten-Reichweite dann
15           FigurIstRechteck ← falsch;
16     Ende
17     startZeile ← Anfang der Figur;
18   wenn FigurIstRechteck dann
19     zeilenMarker ← Ende der Figur;
20     Markiere alle schwarzen Felder innerhalb der Reichweite, die von
       startZeile,zeilenMarker,startSpalte und endeSpalte aufgespannt
       wird als Rechteck-Feld;
21     Fahre bei Feld nach der endeSpalte fort.
22   sonst
23     zeilenMarker ← Letzte Zeile in der Rechteck-Bedingug fuer Figur
       erfüllt war;
24     Markiere alle schwarzen Felder innerhalb der Reichweite, die von
       startZeile,zeilenMarker,startSpalte und endeSpalte aufgespannt
       wird als Nicht-Rechteck-Feld;
25     Fahre bei Feld nach der endeSpalte fort.
26   Ende
27 sonst
28   zeilenMarker ← Ende der Figur;
29   Markiere alle schwarzen Felder innerhalb der Reichweite, die von
       startZeile,zeilenMarker,startSpalte und endeSpalte aufgespannt wird
       als Rechteck-Feld;
30   Fahre bei Feld nach der endeSpalte fort.
31 Ende
32 Ende

```

---

Der Code für die Funktionalität findet sich im Projektordner im Programm *pix-CheckTangle.c* in der Funktion *checkPotentialRectangle*.

## 2.6 Unterschied zum "Gibt es ein Rechteck?"-Algorithmus

Betrachtet man als Problemstellung die Frage "*Enthält das Raster exakt ein Rechteck?*", gestaltet sich der Algorithmus in den folgenden Punkten anders.

1. Die Worker-Prozesse müssen auf dem Teil-Raster prüfen, ob alle schwarzen Felder zusammenhängen und ob für Figuren auf mehreren untereinanderliegenden Zeilen die Spalten-Reichweite der schwarzen Felder identisch ist
  2. Folglich können Worker-Prozesse ggf. vorzeitig abbrechen und müssen dann nicht alle Felder des Teil-Rasters betrachten (Bestes Beispiel wäre hier ein Schachbrett-Muster)
  3. Worker müssen an den Master-Prozess nur die Information, ob es exakt ein Rechteck im Teilraster gibt und ggf. die Lage-Informationen des Rechtecks übermitteln (dann, wenn Rechteck eine Teil-Raster-Grenze berührt, an die ein weiteres Teil-Raster grenzt)
  4. Die Abschließende Prüfung durch den Master-Prozess beurteilt, ob für mehrere auseinanderliegende Teil-Raster ein Rechteck zurückgegeben wurde und dazwischen kein Rechteck existierte.
- Sollten nur für mehrere zusammenhängende Teil-Raster Rechtecke zurückgegeben worden sein, ist die Übereinstimmung der Spalten-Reichweite der schwarzen Felder zu prüfen.

Es bleibt also gedanklich festzuhalten, dass der kommunikative Aufwand (Bei Rückkommunikation) und der Aufwand für die abschließende Prüfung durch  $P_0$  im "Gibt es ein Rechteck?"-Algorithmus geringer ist.

Zudem kann der gesamte Algorithmus bei entsprechenden Mustern durch vorzeitige Abbrüche schneller ausgeführt sein (Beispiel Schachbrett-Muster).

### 3 MESSUNGEN UND INTERPRETATION

#### 3.1 Vorgehen beim Messen

##### Gemessene Zeitpunkte

Für die Ermittlung des Speedups sollen Zeitmessungen durchgeführt werden. Da ein großer Anteil der Laufzeit für die Netzwerkkommunikation benötigt wird, soll hier zunächst eine Vorüberlegung angestellt werden, wie die Netzwerk-Kommunikation aus der Ausführungszeit herauszurechnen ist.

Die Folgende schematische Darstellung zeigt die von den einzelnen Prozessen benötigte Zeit, in einem Cluster-Prozess mit 3 Worker-Prozessen.

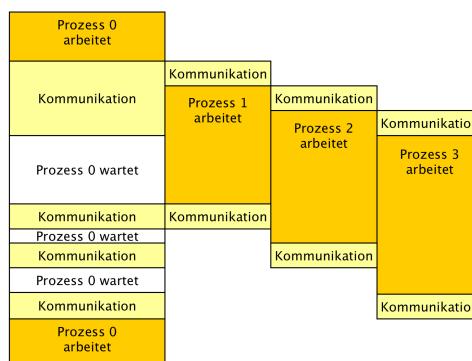


Abbildung 9: Skizze: Zeitbedarf mit Netzwerkkommunikation

Nimmt man hierbei an, dass die Netzwerkkommunikation keine Zeiteinheiten kostet, so erhält man näherungsweise die folgende schematische Darstellung.

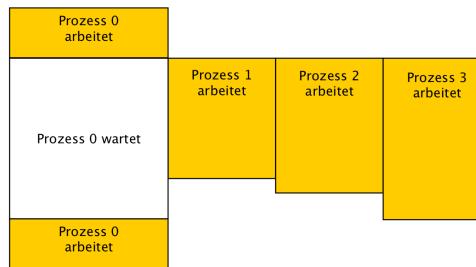


Abbildung 10: Skizze: Zeitbedarf ohne Netzwerkkommunikation

Daraus geht hervor, dass sich die zu messende Zeit unter Ausschluss der Netzwerkkommunikation näherungsweise aus der von Prozess 0 benötigten aktiven Arbeitszeit und der benötigten Arbeitszeit, des am längsten laufenden Worker-Prozesses zusammensetzt.

Die Zeit-Messungen werden im Code mittels **`MPI_Wtime()`**-Befehl realisiert. Zum ermitteln der maximalen Arbeitszeit aller Worker-Prozesse wird im Anschluss an die eigentliche Programm-Ausführung der **`MPI_Reduce()`**-Befehl genutzt.

Die Resultate der Zeitmessungen im Code werden mittels der Funktion `writeTimeToFile` in eine Datei geschrieben.

### Ausführung der Messreihen

Für die Ausführung von mehreren Messungen in einem Lauf werden die folgenden Hilfs-Skripte/Programme verwendet. Diese sind im Anhang einzusehen und im Projektordner enthalten.

- `hostfileGenerator.c`: Generiert eine hostfile-Datei mit n Prozessoren.
- `bsCreateHostfiles.sh`: ruft `hostfileGenerator.c` auf, um mehrere hostfiles zu erzeugen.
- `bsRunGrids.sh`: Ruft `pixCheckTangle` per MPI für eine Raster-Datei und mehrere Prozesse auf.

Gemessen werden:

- Laufzeit mit und ohne Netzwerkkommunikation
- Für Raster mit  $n = \{100, 1000, 10.000\}$  Feldern Breite.
- Pro Rastergröße: Messungen mit je  $p = \{1, 2, \dots, 50\}$  Prozessen.
- Pro Rastergröße und Prozess-Anzahl: drei verschiedene Probleminstanzen:
  - Alle Felder Weiß
  - Alle Felder Schwarz
  - Schachbrett-Raster

Jede Messung wird drei mal durchgeführt. Als Messwert wird der resultierende Mittelwert betrachtet.

## 3.2 Messergebnisse und Interpretation

### Was wird betrachtet?

Im folgenden sollen die gemessenen Daten analysiert und ausgewertet werden. Dabei werden direkt die errechneten Verläufe von Speedup und Effizienz über die einzelnen Probleminstanzen bei steigender Anzahl der aktiven Prozesse  $p$  betrachtet. Die zugrundeliegenden Berechnungen für Speedup und Effizienz sind hierbei:

Speedup:

$$S(p) = \frac{T(1)}{T(p)} \quad (1)$$

Effizienz:

$$E(p) = \frac{S(p)}{p} \quad (2)$$

### Kleine Probleminstanz mit Netzwerkkommunikation.

Um besser nachvollziehen zu können, aus welchen Gründen die Kurven welche Charakteristika aufweisen, soll zunächst eine kleine Probleminstanz mit Rasterbreite  $n = 100$  betrachtet werden. Speedup und Effizienz bei der Messung einschließlich der Netzwerkkommunikation stellen sich dabei wie folgt dar:

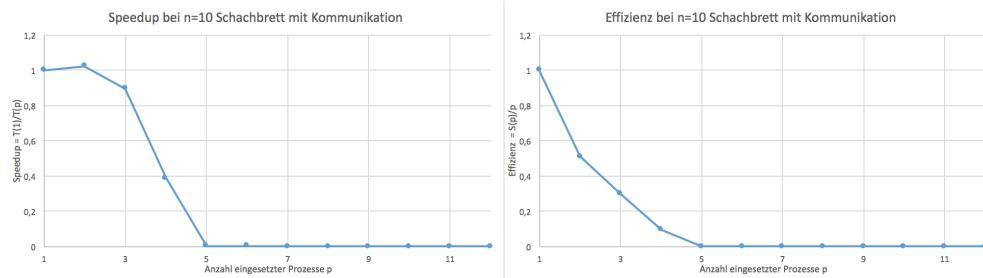


Abbildung 11:  $S(p)$  und  $E(p)$  für  $n=10$  Schachbrett mit Netzwerkkommunikation

### Feststellung #1

Betrachtet man das erste Diagramm, ist dabei zu sehen, dass ab dem Einsatz eines fünften Prozessors der Speedup gegen Null geht. Die Ursache hierfür ist zu erkennen, wenn man sich die generierten und genutzten hostfiles vor Augen führt. Als Beispiel, sei hier das generierte hostfile *myhostfile15* gezeigt.

---

```
simson01 slots = 4
simson02 slots = 4
simson03 slots = 4
simson04 slots = 3
```

---

Pro Rechner werden alle Prozessoren genutzt. Sobald alle vier Prozessoren eines Rechners genutzt werden, wird bei Erhöhung von  $p$  ein weiterer Rechner zum Abarbeiten des Programmes hinzugezogen.

Im Diagramm ist zu erkennen, dass die Kommunikation über das Netzwerk mit weiteren Rechnern einen großen Teil der Arbeitszeit einnimmt.

In allen untersuchten Fällen bewirkt dieser Umstand, dass eine effiziente Parallelisierung (unter Berücksichtigung des Kommunikationsaufwandes) bestenfalls nur dann möglich ist, wenn keine Kommunikation über das Netzwerk stattfindet. (Siehe hierzu sämtliche Graphen, die die Netzwerkkommunikation mit betrachten im Anhang.)

### Kleine Probleminstanz ohne Netzwerkkommunikation.

Im Weiteren soll nach dieser Erkenntnis der Blick auf die reine Parallelisierung ohne Betrachtung des Netzwerk-Kommunikationsaufwandes gelenkt werden. Speedup und Effizienz für die oben betrachtete Probleminstanz ohne Netzwerkkommunikation stellen sich wie folgt dar.

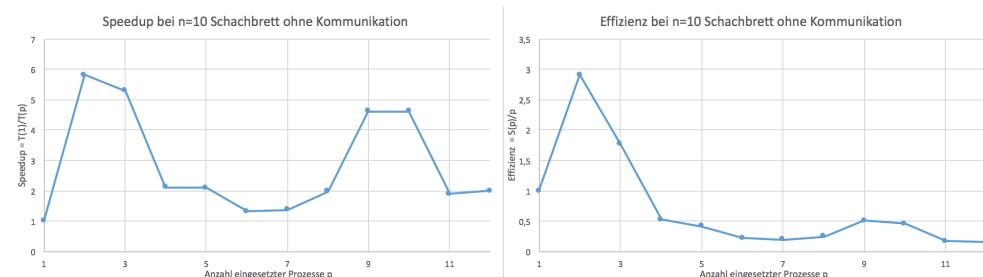


Abbildung 12:  $S(p)$  und  $E(p)$  für  $n=10$  Schachbrett ohne Netzwerkkommunikation

Daraus lassen sich die folgenden zwei Erkenntnisse gewinnen.

### Feststellung #2

Die Abarbeitung gleicher Teil-Raster erfolgt auf verschiedenen Prozessoren teilweise unterschiedlich schnell. Theoretisch betrachtet, müssten bei der vorliegenden Implementierung der sequentielle Algorithmus und die Ausführung mit zwei Prozessen ohne Betrachtung der Netzwerkkommunikation gleich schnell sein. Der Speedup für  $p = 2$  müsste dabei folglich 1 sein. Dies ist, wie oben zu sehen, nicht der Fall. Für die Laufzeit mit Betrachtung der Kommunikation müsste sich ergeben, dass  $T(1) < T(2)$ . Wie in Abbildung 11 zu erkennen, gilt jedoch  $T(1) > T(2)$ . Im Prozess mit kleineren Feldern (Raster-Breite  $n = 10, 100$ ) ist dieser Unterschied besonders erkennbar, bei der Erhöhung der Feldgröße, wirkt sich dieser Effekt nicht mehr so stark aus. (Vergleiche Diagramme im Anhang.)

### Feststellung #3

Im Diagramm für den Speedup ist ein deutlicher Abfall des Speedups im Bereich  $p = 4, \dots, 8$  festzustellen. In diesem Bereich hätte man eine bogenförmige Speedup-Kurve (monoton steigend, dann monoton fallend) vermuten können. Woher der Einbruch kommt, wird klar, wenn man sich ansieht, was bei welcher Prozessoranzahl in dieser Probleminstanz passiert.

Im Folgenden ist in einer Tabelle dargestellt, wie sich die Aufteilung des Rasters auf die Worker-Prozesse gestaltet und wie viele Zeilen vom Master-Prozess auf potentielle Rechtecke geprüft werden müssen.

Anzahl Prozesse p	1	2	3	4	5	6	7	8	9	10	11
Zeilen pro $P_1 \dots P_{p-1}$	-	-	5	3	2	1	1	1	1	1	1
Zeilen pro $P_p$	-	10	5	4	4	2	5	4	3	2	1
Prüfung Zeilen in $P_0$	-	0	2	4	6	8	6	7	8	9	10

Es wird klar, dass der Einbruch der Speedup-Kurve zwei Gründe hat.

- Die Aufteilung des Rasters in Teil-Raster ist im Falle einer geringen Raster-Größe nicht optimal.
- Der Aufwand des nicht Parallelisierbaren Anteils am Prozess (die Prüfung potentieller Rechtecke in  $P_0$ ) wird mit steigender Anzahl genutzter Prozesse größer.

Den ersten Punkt kann man an der Anzahl der Zeilen erkennen, die von  $P_p$  abzuarbeiten sind. Bei steigender Raster-Größe wirkt sich der Effekt nicht mehr so stark aus (wie später zu sehen), ist aber dennoch als Schwankung in den nachfolgenden Diagrammen zu erkennen.

Der zweite Punkt, das Wachstum des nicht-parallelisierbaren Anteils, wird in den folgenden Betrachtungen eine wesentliche Rolle spielen.

### Vergleich der Muster

Nachdem diese Vorbetrachtung abgeschlossen ist, sollen nun für ein größeres Raster (Raster-Breite  $n = 100$ ) die drei verschiedenen Probleminstanzen (Schachbrett, alle Felder schwarz, alle Felder weiß) in puncto Speedup und Effizienz ohne Berücksichtigung der Netzwerkkommunikation betrachtet werden. Die entsprechenden Graphen sind in Abbildung 13 untereinander dargestellt.

### Feststellung #4

In den Graphen ist der vermutete bogenförmige Verlauf der Speedup-Kurve besser zu erkennen. Des weiteren wird ersichtlich, dass der Speedup umso höher ist, je weniger und kleinere Rechtecke im Raster enthalten sind. Besonders beachtlich ist daher der Speedup für die Probleminstanz ohne schwarze Felder.

Der Speedup verringert sich, je mehr potentielle Rechtecke vom Master-Prozess geprüft werden müssen, wobei der Prüf-Aufwand besonders groß wird, wenn die



Abbildung 13:  $S(p)$  und  $E(p)$  für  $n=100$  Alle drei Muster ohne Netzwerkkommunikation

Rechtecke sehr groß sind und über mehrere Teil-Raster-Grenzen hinweg reichen.

Im Hinblick auf diese Überlegung ist es damit erklärbar, dass gilt:

$Speedup_{Allesweiss} > Speedup_{Schachbrett} > Speedup_{Allesschwarz}$   
Für die Effizienz gilt die entsprechende Relation.

Bei den Probleminstanzen für die Raster-Breite  $n = 100$  ist anzumerken, dass die Effizienz auch unter Berücksichtigung des Zeitaufwandes für die Kommunikation für alle drei Instanzen so hoch ist, dass sich eine Parallelisierung lohnt (Hier wirkt sich der in Feststellung #2 beschriebene Effekt besonders sichtbar aus). Nach Überschreiten der Rechner-Grenze gilt jedoch für die Effizienz  $E(p) < 1$ , wie in Feststellung #1 beschrieben.

### Erhöhen der Raster-Größe

Als nächstes soll untersucht werden, wie sich Speedup und Effizienz der Probleminstanzen bei steigender Raster-Größe verhalten.

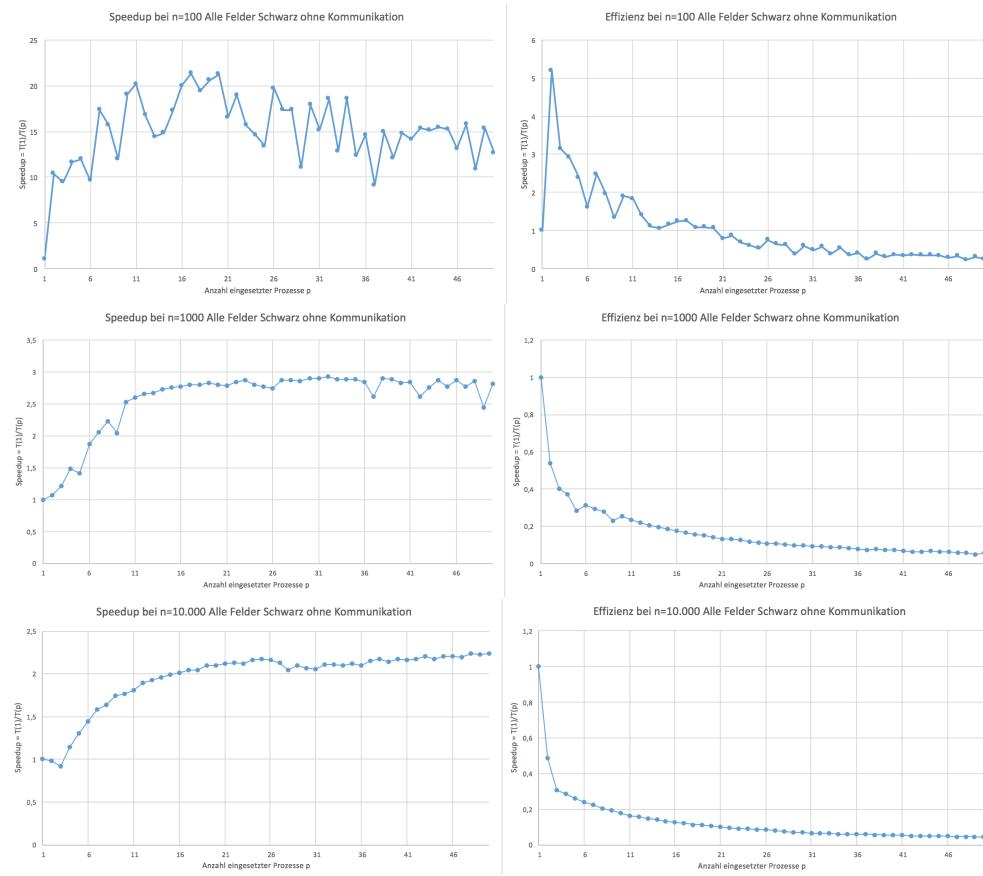
Dem Maximum der Raster-Größe sind dabei zwei Grenzen gesetzt.

Zum einen verhindert der für den User begrenzte Festplattenspeicher ab einer hinreichend großen Raster-Breite das Anlegen des Grid-Files durch den *gridGenerator.c*. Zum Anderen wird im Programm *pixCheckTangle.c* zur Kalkulation der Indizes für die Feld-Adressierung ein long int Typ verwendet. Dessen Wertebereich wird ab einer bestimmten Raster-Breite durch die Anzahl der Raster-Felder überschritten. Die Grenze des Festplattenspeichers ist dabei eher erreicht. So waren das Anlegen von Rastern mit Raster-Breite  $n \geq 40.000$  nicht möglich.

Im Folgenden sei das Wachstum der Grid-Files und das Überschreiten des long-int-Bereiches verdeutlicht.

Raster-Breite n	Elemente im Raster	$n * n \geq \text{Long int?}$	Größe txt-File
10.000	100.000.000	nein	200 MB
20.000	400.000.000	nein	800 MB
30.000	900.000.000	nein	1,8GB
40.000	1.600.000.000	nein	3,2GB
50.000	2.500.000.000	ja	5GB

Zur Messung sollen nun die Raster-Breiten  $n = 100/1000/10.000$  betrachtet werden. Exemplarisch sind in **Abbildung 14** die Kurven für das Muster "Alle Felder Schwarz" ohne Betrachtung des Kommunikationsaufwandes dargestellt. Die Übrigen Diagramme finden sich im Anhang.



**Abbildung 14:**  $S(p)$  und  $E(p)$  für  $n=100/1000/10.000$  Alle Felder schwarz ohne Netzwerkkomunikation

### Feststellung #5

Bei der Betrachtung der Kurven wird zunächst deutlich, dass die bogenförmigen Verläufe der Speedup-Kurve mit zunehmender Raster-Größe länger und deutlich flacher werden.

So "wandert" das Maximum für das Raster "Alle Felder schwarz" wie folgt:

- Für Raster-Breite  $n = 100$ :  $S(p = 17) \approx 22$
- Für Raster-Breite  $n = 1000$ :  $S(p = 32) \approx 2,9$
- Für Raster-Breite  $n = 10.000$ :  $S(p > 50) \approx 2,3$

Der Effekt, dass die Bögen des Speedups in den Graphen länger werden, wird ebenfalls in den Raster-Instanzen "Schachbrett" und "alle Felder weiß" ersichtlich,

wobei jedoch der maximale Speedup nicht stetig fällt.

Die Maxima der Speedup-Graphen ohne Netzwerkkommunikation stellen sich konkret wie folgt dar:

Raster-Breite n	Alles weiß	Schach	Alles schwarz
100	$S(p = 12) \approx 120$	$S(p = 11) \approx 34$	$S(p = 17) \approx 22$
1000	$S(p = 26) \approx 13$	$S(p = 26) \approx 14$	$S(p = 32) \approx 2,9$
10.000	$S(p >> 50) > 28$	$S(p = 47) \approx 25$	$S(p > 50) \approx 2,3$

Ein langsam anwachsender Speedup bedeutet eine geringe Effizienz der Parallelisierung. Dies spiegelt sich in den entsprechenden Effizienz-Graphen wieder. Für eine Raster-Breite  $n \geq 1000$  ist eine Parallelisierung bereits nicht mehr effizient möglich. Es gilt  $E(p) < 1 \mid p > 1$ .

Zu Begründen sind diese Beobachtungen durch den stark wachsenden nicht-parallelisierbaren Anteil des Programmes, der Prüfung potentieller Rechteck im Master-Prozess. Die Werte zeigen, dass dieser Anteil mit wachsender Raster-Größe schneller wächst als der zeitliche Gewinn, der durch das Parallelisieren entsteht.

### Feststellung #6

Die oben dargestellten Maxima der Speedup-Kurven bestätigen, dass aufgrund dieses Effektes die Relationen

$\text{Speedup}_{\text{Allesweiss}} > \text{Speedup}_{\text{Schachbrett}} > \text{Speedup}_{\text{Allesschwarz}}$   
und

$\text{Effizienz}_{\text{Allesweiss}} > \text{Effizienz}_{\text{Schachbrett}} > \text{Effizienz}_{\text{Allesschwarz}}$   
bei steigender Raster-Größe erhalten bleiben.

## 4 UNTERSCHIED ZUM "GIBT ES EIN RECHTECK?"-ALGORITHMUS UND FAZIT

### 4.1 Fazit

Es sind zwei Aspekte deutlich geworden, welche bei dem betrachteten Problem einer effizienten Parallelisierung entgegenwirken:

- Die notwendige Übertragung der großen Datenmengen für die Teil-Raster von Prozessor zu Prozessor.
- Der mit steigender Raster-Größe wachsende, nicht-parallelisierbare Aufwand.

Der zweiter Aspekt bedingt, dass eine Parallelisierung bei den betrachteten Problemfällen nur bei kleiner Raster-Größe effizient ist. (Im untersuchten Fall für  $n = 100$ .)

Der erste Aspekt bedingt, dass auch im Falle einer kleineren Raster-Größe die Parallelisierung nur dann effizient ist, wenn die Parallelisierung auf die Prozesse eines Rechners beschränkt ist und keine Rechner-übergreifende Kommunikation stattfinden muss.

Generell liegt für das gesamte Problem der Umstand vor, dass auf einer großen Datenmenge nur wenige und teils schnelle Rechenoperationen (in den meisten Fällen Kalkulation von Indizes und Vergleichs-Operationen) ausgeführt werden müssen. Wäre das Problem so geartet, dass aufwändige Operationen auf kleinen Datenmengen ausgeführt werden müssen, hätte hier eine Parallelisierung (wenn dann möglich) eine größere Effizienz.

#### 4.2 Unterschied zum "Gibt es ein Rechteck?"-Algorithmus

Wie zuvor in 2.6 festgestellt, besteht ein wesentlicher Unterschied zwischen der umgesetzten Implementierung und dem "Gibt es ein Rechteck?"-Algorithmus darin, dass die Worker-Prozesse nur geringe Informationsmengen an den Master-Prozess zurück-übertragen müssen. Die Übertragung der Teil-Raster vom Master- an die Worker-Prozesse bleibt jedoch erforderlich.

Das Abfallen der Effizienz, welches einsetzt, sobald zwischen mehreren Rechnern kommuniziert werden muss, sollte sich daher in geringerem Maße auswirken. Da jedoch somit optimistisch geschätzt nur die Hälfte der notwendigen Kommunikation wegfällt, sollte der Effekt jedoch weiterhin stark bemerkbar sein und vermutlich auch zu einer Ineffizienz der Parallelisierung für  $p > 4$  sorgen.

Ein weiterer wesentlicher Unterschied besteht darin, dass der Master-Prozess sehr viel weniger Prüf-Aufwand im nicht-parallelisierbaren Anteil des Algorithmus hat. Zudem ist dieser Anteil im wesentlichen unabhängig von der Raster-Größe und nur abhängig von der Anzahl der eingesetzten Prozessoren. Hier sollte daher kein fallender Speedup für identisches  $p$  mit steigender Raster-Größe zu erwarten sein.

Das oben resumierte Problem bezüglich der großen Datenmenge und den wenigen Rechenoperationen gilt jedoch auch in diesem Fall.

## A ANHANG

### A.1 Code: gridGenerator.c

Auf eine vollständige Auflistung des Codes wird hier aus Gründen der schlechten Lesbarkeit verzichtet. Der Code befindet sich als Datei "gridGenerator.c" im Angabe-Ordner.

### A.2 Code: pixCheckTangle.c

Auf eine vollständige Auflistung des Codes wird hier aus Gründen der schlechten Lesbarkeit verzichtet. Der Code befindet sich als Datei "pixCheckTangle.c" im Angabe-Ordner.

### A.3 Code: hostfileGenerator.c

---

```
#import <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    //get inputed number as int
    int hosts = 0;
    int i;
    char *value;
    for(i = 1; i < argc; i++)
    {
        value = argv[i];
        hosts += atoi(value);
    }

    char name[30];
    sprintf(name, "myhostfile%d", hosts);
    printf("%s\n",name);

    FILE *fp;
    fp = fopen(name, "w");

    div_t fullClients = div(hosts,4);

    for (i = 0;i<fullClients.quot;i++){
        //fprintf(fp, "simson%02d slots = 4\n",i+1);
        fprintf(fp, "imunix%02d slots = 4\n",i+1);
    }
    if (fullClients.rem != 0){
        //fprintf(fp, "simson%02d slots = %d\n",fullClients.quot
        +1,fullClients.rem);
        fprintf(fp, "imunix%02d slots = %d\n",fullClients.quot +1,fullClients.rem);
    }

    fclose(fp);
}
```

---

### A.4 Code: bsCreateHostfiles.sh

---

```
for run in {1..50}
```

---

```

do
./hostfileGenerator $run
done

```

---

#### A.5 Code: bsRunGrids.sh

---

```

for run in {1..50}
do
mpirun -hostfile myhostfile$run pixCheckTangle grid1.txt
done

```

---

#### A.6 README: Beispielhafte Ausführung

Example of Execution:

---

Single run example:

- compile gridGenerator and create a Grid

```

$ mpicc -o gridGenerator gridGenerator.c
$ mpirun gridGenerator
Enter filename i.e. "grid1"
Enter the Grid-Size i.e. 30
Generate a rectangle using option 2 i.e following Entries:
4
4
24
10
Use option 8 to save grid to file

```

- compile pixCheckTangle and run it with grid-file:

```

$ mpicc -o pixCheckTangle pixCheckTangle.c
$ mpirun pixCheckTangle grid1.txt

```

---

Full Cluster Measurement example:

- generate myhost-files:

```

Change the host-name in hostfileGenerator.c
$ clang -o hostfileGenerator hostfileGenerator.c
$ chmod +x ./bsCreateHostfiles.sh
$ ./bsCreateHostfiles.sh 50

```

- compile gridGenerator and create a Grid

```

$ mpicc -o gridGenerator gridGenerator.c
$ mpirun gridGenerator
Enter filename i.e. "grid1"
Enter the Grid-Size i.e. 30
Generate a rectangle using option 2 i.e following Entries:
4
4

```

24

10

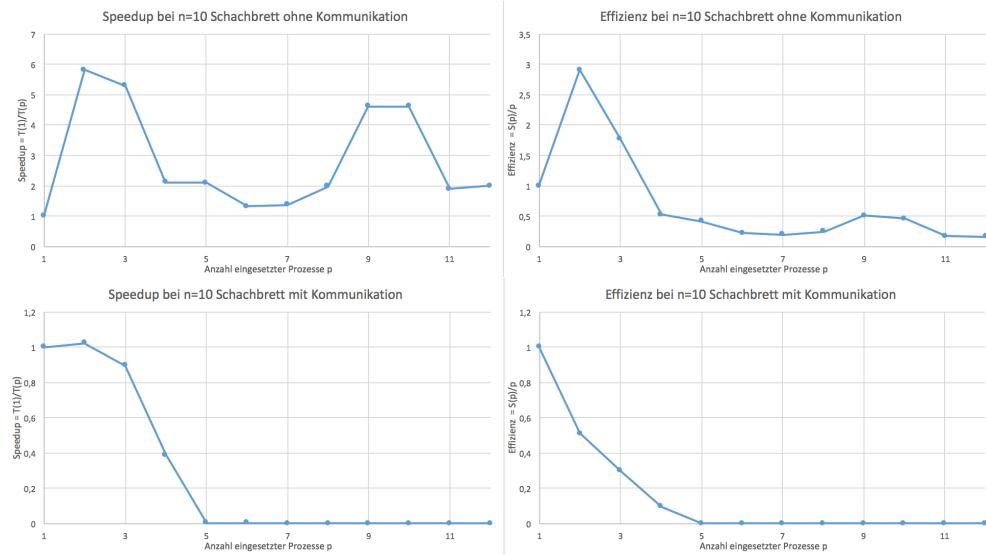
Use option 8 to save grid to file

3. compile pixCheckTangle and run full measurement:

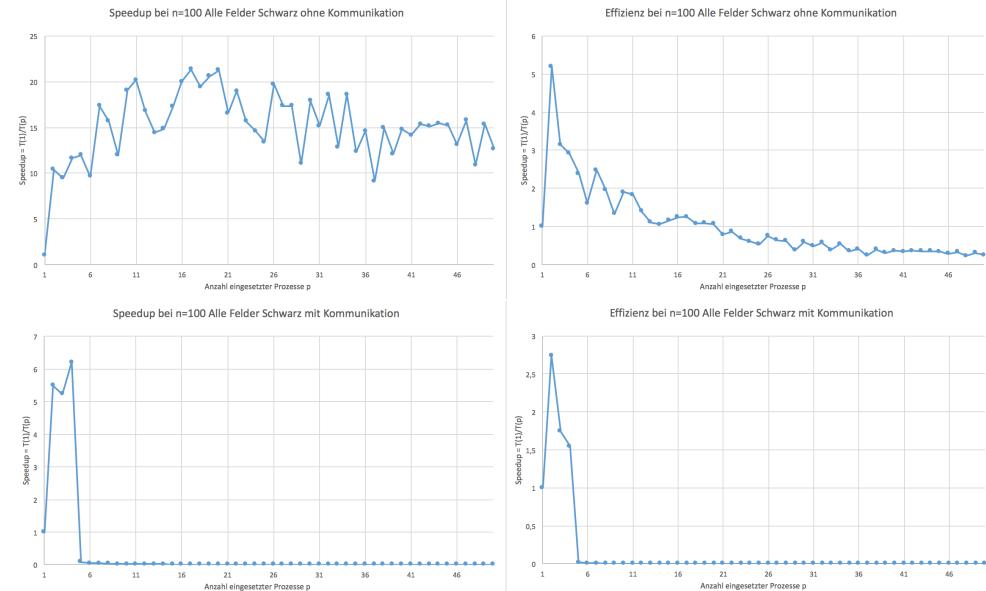
```
$ mpicc -o pixCheckTangle pixCheckTangle.c
Change the grid-File-Name in bsRunGrids.sh
$ chmod +x ./bsRunGrids.sh
$ ./bsRunGrids.sh
```

## A.7 Alle Messergebnisse

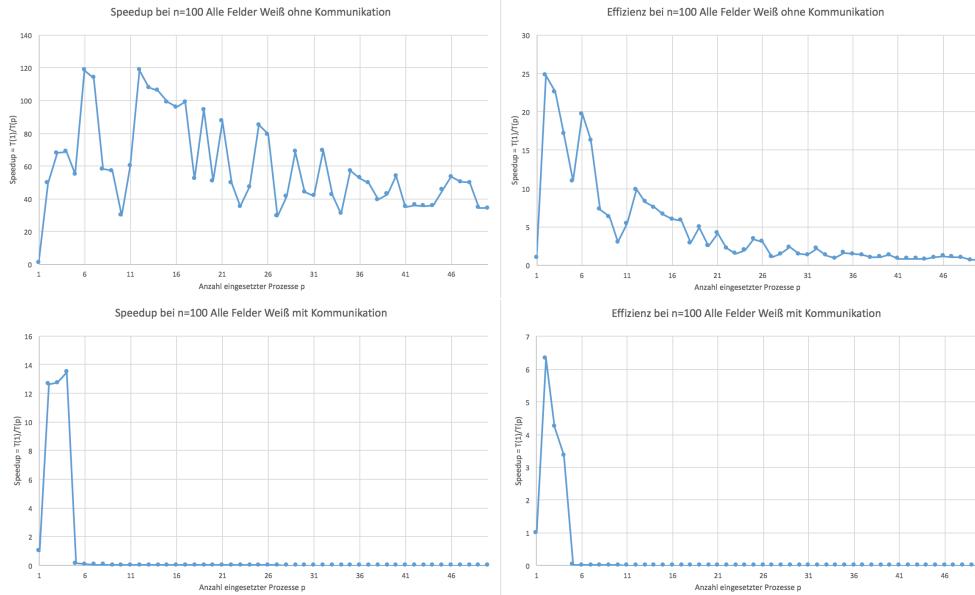
### Feldgröße 10x10 - Schachbrett



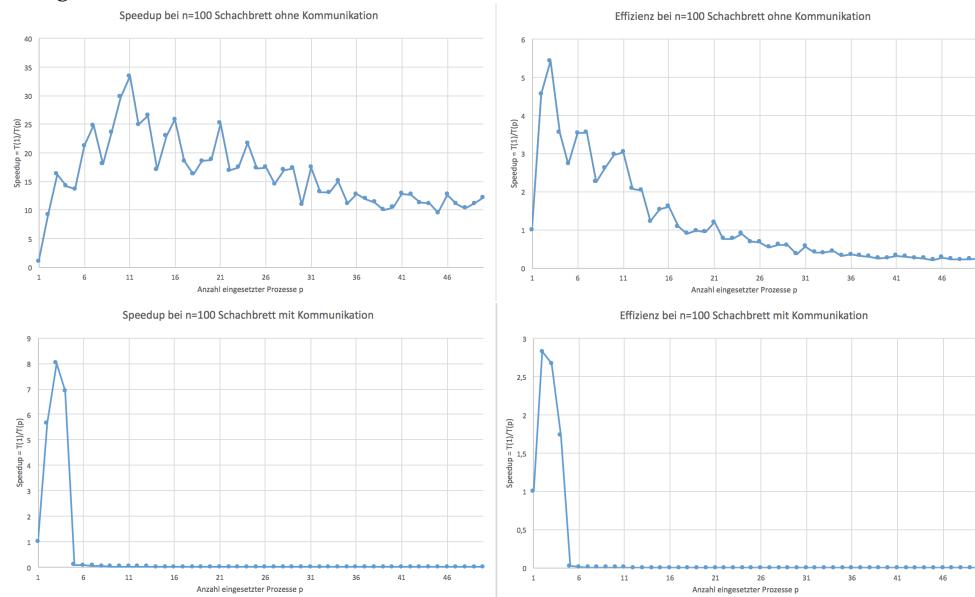
### Feldgröße 100x100 - Alle Felder Schwarz



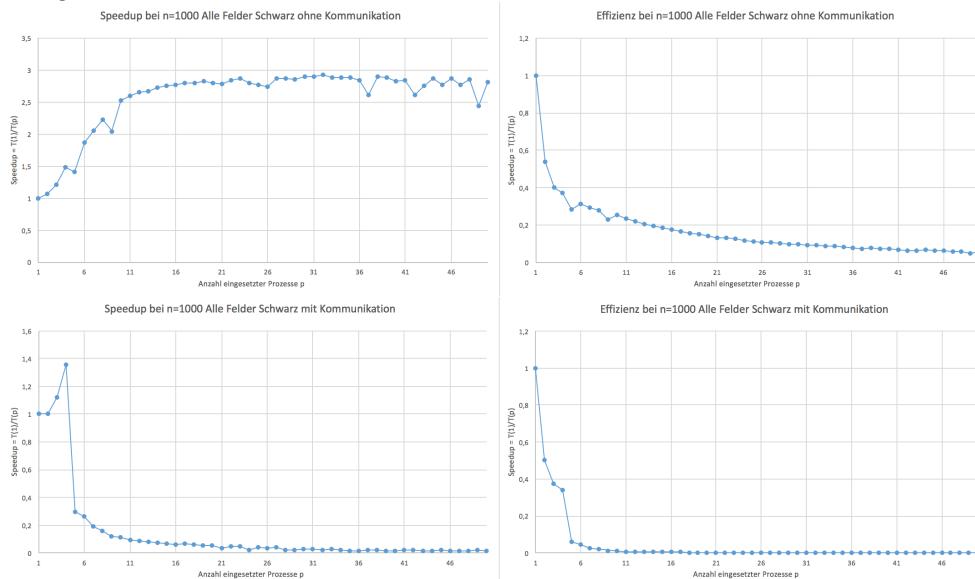
### Feldgröße 100x100 - Alle Felder Weiß



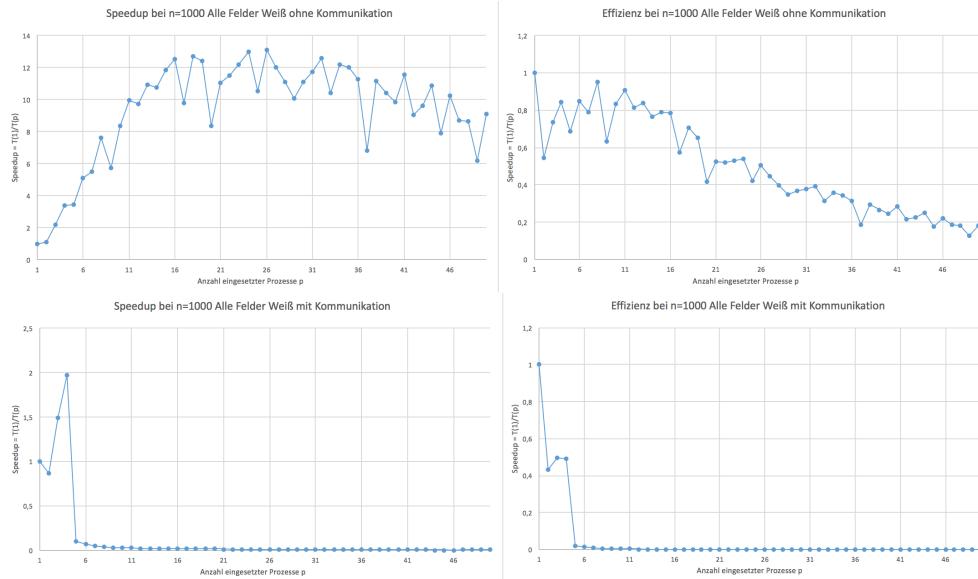
### Feldgröße 100x100 - Schachbrett



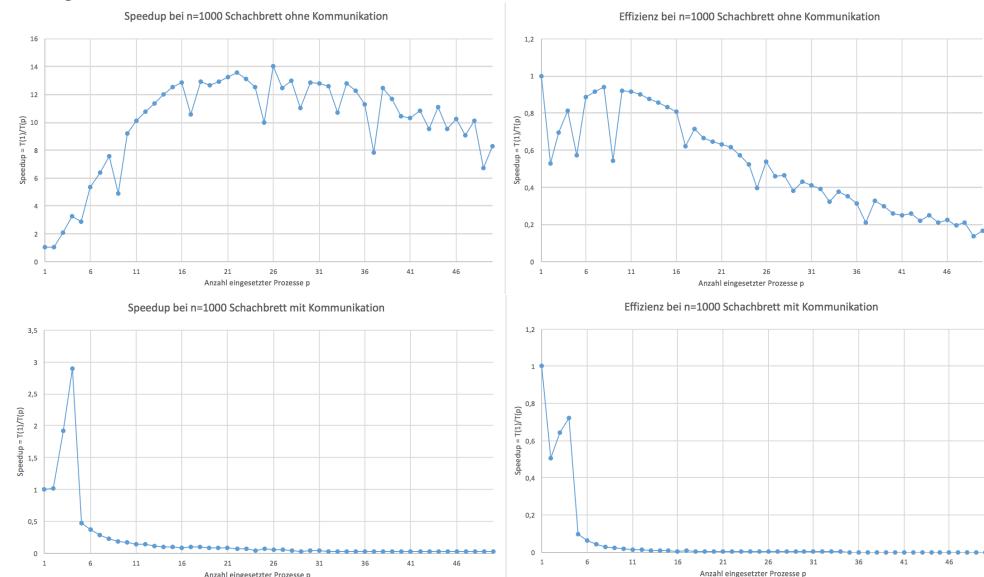
### Feldgröße 1000x1000 - Alle Felder Schwarz



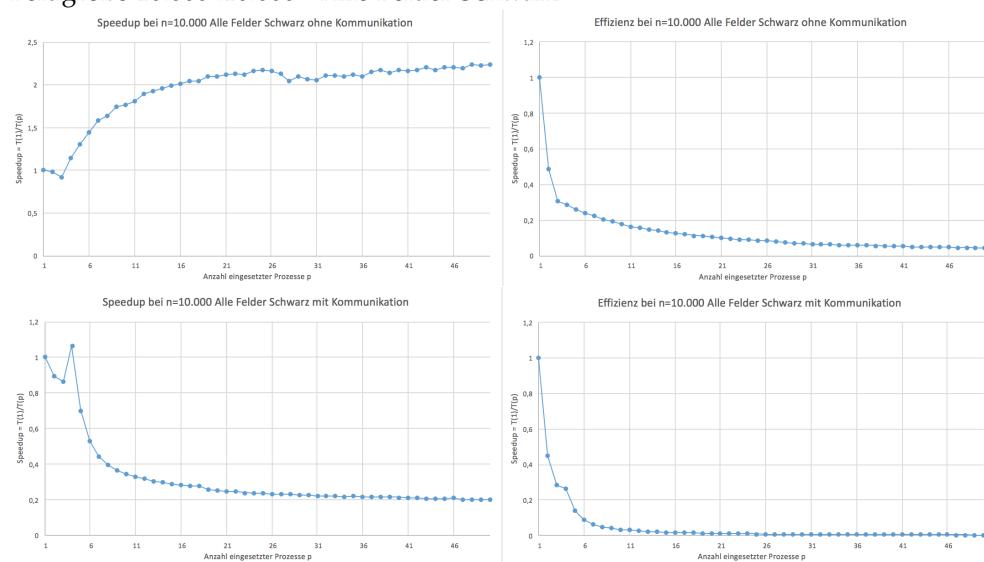
### Feldgröße 1000x1000 - Alle Felder Weiß



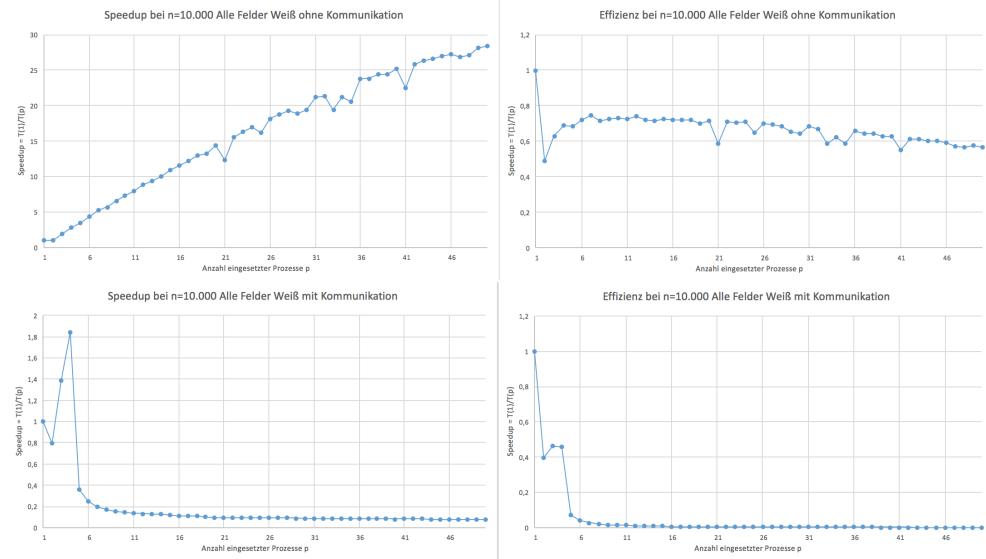
### Feldgröße 1000x1000 - Schachbrett



### Feldgröße 10.000x10.000 - Alle Felder Schwarz



### Feldgröße 10.000x10.000 - Alle Felder Weiß



### Feldgröße 10.000x10.000 - Schachbrett

