

# PROJEKTAUFGABE 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	9
3.1	Vorgehen beim Messen . . . . .	9
3.2	Messergebnisse . . . . .	10
3.3	Interpretation der Messergebnisse . . . . .	12
3.4	Unterschied zum einfacheren und Abschluss . . . . .	12
4	Fazit	12
4.1	Einschätzung des Nutzens einer Parallelisierung . . . . .	12
4.2	Mögliche Optimierungen . . . . .	12
A	Anhang	13
A.1	Code: DRT <sub>1</sub> . . . . .	13
A.2	Code: DRT <sub>2</sub> . . . . .	13
A.3	Code: hostfileGenerator.c . . . . .	13
A.4	Code: bsCreateHostfiles.sh . . . . .	13
A.5	Code: bsRunGridTest.sh . . . . .	13
A.6	Alle Messergebnisse . . . . .	13

## ABBILDUNGSVERZEICHNIS

Abbildung 1	Funktionalität von <i>gridGenerator.c</i> . . . . .	3
Abbildung 2	Funktionalität von <i>pixCheckTangle.c</i> sequentiell . . . . .	3
Abbildung 3	Cluster-Prozess: Beispiel-Raster . . . . .	4
Abbildung 4	Cluster-Prozess: Aufteilung Beispiel-Raster mit 4 Prozessen	4
Abbildung 5	Cluster-Prozess: Worker-Prozess: Nicht-Rechteck-Figuren	4
Abbildung 6	Cluster-Prozess: Worker-Prozess: Rechtecke und potentielle Rechtecke . . . . .	5
Abbildung 7	Cluster-Prozess: Beispiel-Raster . . . . .	5
Abbildung 8	Funktionalität von <i>pixCheckTangle.c</i> Cluster-Prozess . . . . .	6
Abbildung 9	Skizze: Zeitbedarf mit Netzwerkkommunikation . . . . .	9
Abbildung 10	Skizze: Zeitbedarf ohne Netzwerkkommunikation . . . . .	10
Abbildung 11	Skizze: Zeitbedarf ohne Netzwerkkommunikation . . . . .	11
Abbildung 12	Skizze: Zeitbedarf ohne Netzwerkkommunikation . . . . .	11

## 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.

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 ist im Anhang ab Seite (DRT oder Verweis angeben)... gelistet.

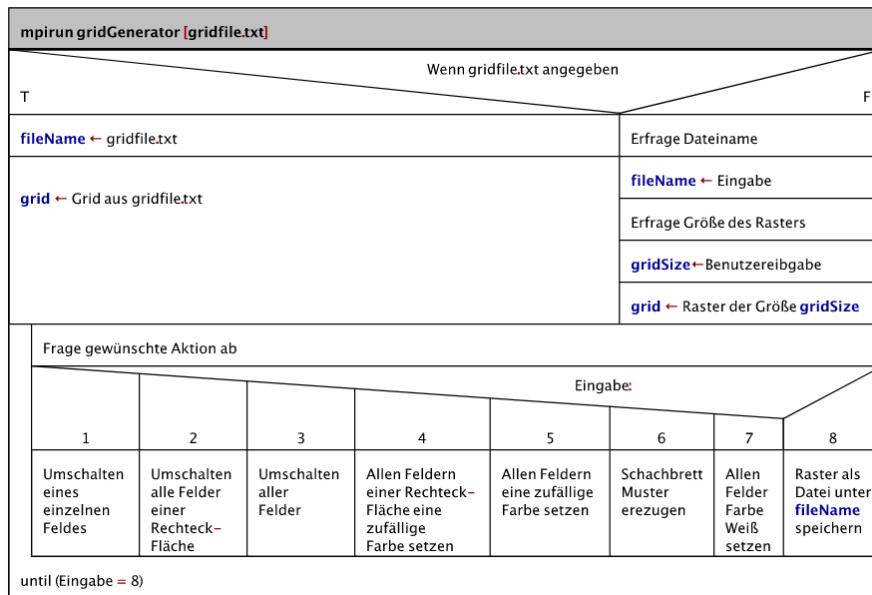


Abbildung 1: Funktionalität von `gridGenerator.c` dargestellt als Nassi-Shneiderman-Diagramm

## 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.

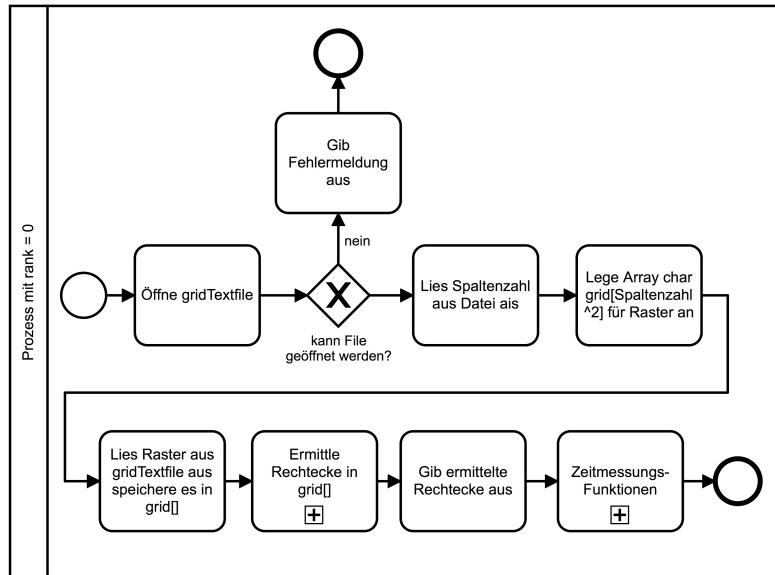


Abbildung 2: Sequentieller Ablauf von `pixCheckTangle.c` dargestellt als BPMN-Diagramm

Der vollständige Code zum Algorithmus ist im Anhang 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 dann in 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 dabei 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 in einem Teil-Raster zu einem Rand der Teil-Rasters reicht und an diesem Rand im gesamten Raster ein weiteres Raster angrenzt, so könnte die Figur im angrenzenden Teil-Raster fortgesetzt werden. In diesem Fall ist also durch das Verarbeiten des Teil-Rasters keine Aussage darüber möglich, ob es sich um ein tatsächlich um ein Rechteck handelt. Entsprechende Rechtecke (Im Weiteren als potentielle Rechtecke) 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 dabei Arbeits-Aufteilung und die Kommunikation zwischen den Prozessen verläuft, sei im Folgenden als BPMN-Diagramm dargestellt.

Der vollständige Code findet sich im Anhang. 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 Prozess gibt. Der Code für die Worker-Prozesse findet sich im Abschnitt, der ausgeführt wird, wenn der Rang des Prozesses nicht 0 ist.

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.

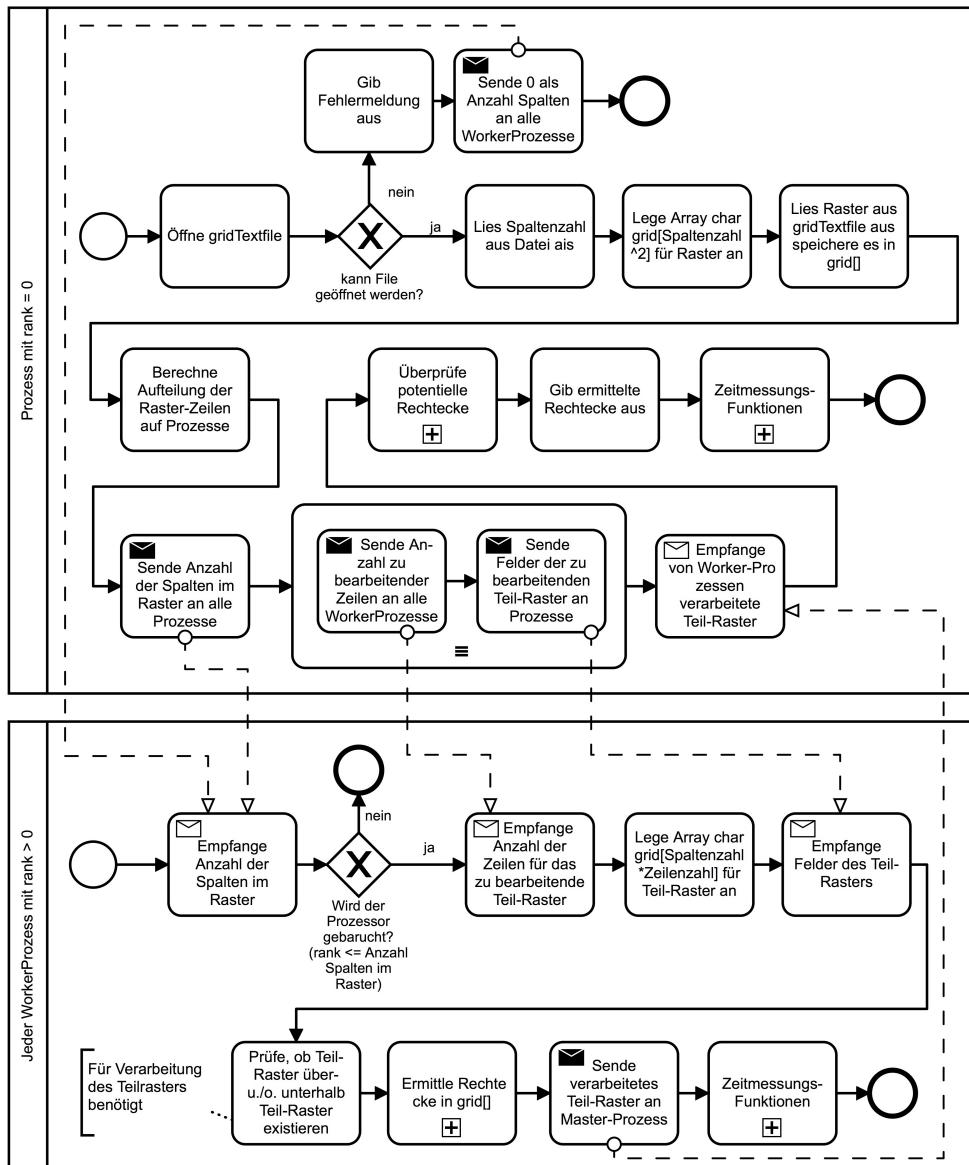


Abbildung 8: Ablauf des Cluster-Prozesses von *pixCheckTangle.c* dargestellt als BPMN-Diagramm

## 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 Anhang im Programm *pixCheckTangle.c* in der Funktion *processSubgrid*.

## 2.5 pixCheckTangle: Funktion für das Festlegen potentieller Rechtecke

Für die Funktion zum Überprüfen der potentiellen Rechtecke, sobald die Worker-Prozesse die Teil-Raster abgearbeitet haben, wird die im Folgenden per Pseudo-

Code beschriebene Vorgehensweise angewandt. Die Kernidee ist dabei nur noch Stichprobenartige Prüfungen zu machen, anstatt die Schritte aus dem oben aufgezeigte Prozess zu wiederholen.

---

```

Daten : Raster,zuPruefendeZeile InfoZuRasterumbruecke
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 letzem 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 Teilraster-Bruch folgt tue
13 wenn Figur setzt unter Teilraster-Bruch fort dann
14 wenn In erster Zeile nach Teilraster-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 Anhang im Programm *pixCheckTangle.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

### 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.

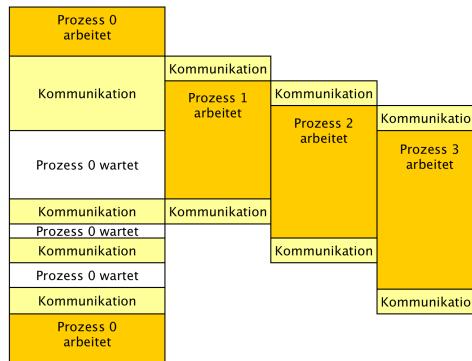


Abbildung 9: Skizze: Zeitbedarf mit Netzwerkkommunikation

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

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

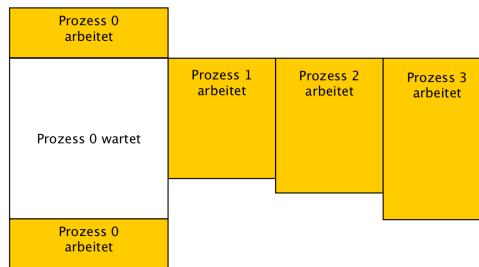


Abbildung 10: Skizze: Zeitbedarf ohne Netzwerkkommunikation

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 Messreihen in einem Lauf werden die folgenden hilfs-Skripte/Programme verwendet. Diese sind im Anhang einzusehen.

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

Gemessen werden:

- Laufzeit mit und ohne Netzwerkkommunikation
- Für Raster mit  $n=100/1000/10000$  Feldern Breite.
- Pro Rastergröße: Messungen mit je  $p=1,2..50$  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 daraus gebildete Mittelwert betrachtet.

### 3.2 Messergebnisse

Im folgenden sollen die gemessenen Daten analysiert und ausgewertet werden. Dabei werden direkt die errechneten Verläufe von Speedup und Effizienz der einzelnen Probleminstanzen bei steigender Anzahl der aktiven Prozesse betrachtet. Die zugrundeliegende Berechnung ist hierbei:

Speedup:

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

Effizienz:

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

Um einschätzen zu können, warum die Kurve wie aussieht im kleinen Anfangen, wo es vorstellbar ist. Für Rastergröße  $n=10$  Alle Felder Weiß. Mit Netzwerkkommunikation:

Festzustellen:

Einbruch nach 4. Prozessor. hostfilessine wiefolgt aufgebaut:

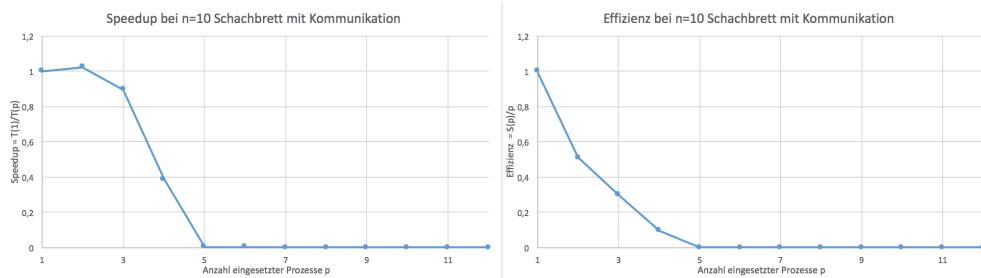


Abbildung 11: Skizze: Zeitbedarf ohne Netzwerkkommunikation

Bsp. *myhostfile15*


---

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

---

Aller 4 Prozessoren also nächster Rechner. Auffallend: Nach 4 Prozessoren, also sobald über 1 Rechner des Clusters arbeitet, aufgrund der Netzwerkkommunikation nicht so gut. Das zieht sich durch alles durch Siehe Anhang.

Bewertung:

Durch viele Daten hohe NWK -> Große Datenmenge, schnelle Rechenoperationen. Andersrum wäre es natürlich besser

Daher jetzt mal ein Blick auf die ONWK:

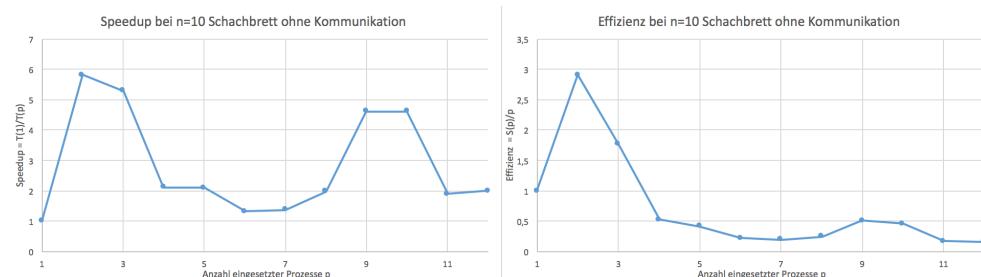


Abbildung 12: Skizze: Zeitbedarf ohne Netzwerkkommunikation

Zwei Dinge, die auffallen:

1) Die Prozessoren scheinbar teilweise untescheidlich schnell 1 und 2 hier gut erkennbar weil datenmenge sehr klein im Folgenden bleibt das sichtbar, relativiert sich aber weil Zeitwerte größer werden.

2) Schwankung / Einbruch von 3 bis 8, wo Speedup vermutet werden könnte.

Kommt durch zwei Sachen:

- Amdahl -> Nicht parallelisierbarer aufwand durch Prüfung potentieller Rechtecke in Master-Prozess nach Arbeit der Worker-Prozesse.

- Aufteilung auf Sub-Prozesse in kleinem Zahlen-Raum nicht optimal.

Im folgenden in Tabelle werden beide Faktoren ersichtlich.

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

Steigender Aufwand für Sequentiellen Prozess je mehr Prozessoren -> Prüfung Zeilen in  $P_0$ .

Parallelisierung nicht gut sieht man an den Zeilen, die  $P_p$  abzuarbeiten hat. Fällt bei zunehmendem n nicht mehr so hart ins Gewicht. Diese Schwankungen sind in folgenden Grafiken auch zu erkennen.

Möglich: Erst an den senden, dann alle anderen, dann wieder einholen -> Würde besser aufteilen ODER: Anders rechnen. Bestes Bsp: Prozesse =7 Worker =6 Pro Prozess nur 1 Zeile Prozess 6 nimmt dann 5 Zeilen -> Besser: Das nochmal verteilen. zweiterer Teil in folgenden Messungen nicht mehr so hart -> Schwankungen werden kleiner.

Genug Vorüberlegung. Jetzt mal im Ernst: Zunächst betrachten von n=100 alle drei Muster.

Es fällt auf:

Da sehen wir was: - Hoher Speedup je kleiner und weniger die Rechtecke werden -> Als Extremfall hier Messwerte für Kein Rechteck == Alles Weiß - Speedup verringert sich je mehr Rechtecke geprüft werden müssen, wobei besonders in's Gewicht fällt, dass Rechtecke sehr groß sind und über viele Teil-Raster-Grenzen reichen. - Je mehr Teil-Raster-Grenzen inkludiert -> desto mehr Arbeit in  $P_0$  zum festlegen von potentiellen Rechtecken - Je größer diese sind -> Aufwand für Powächst auch - Als Extremfall hier alles Schwarz -> Speedup nur mäßig Dazwischen Schachbrett-Muster

So. Dann erhöhen wir mal die Feld-Größe. n= 1000 und n=10.000

Am Beispiel: Alles Schwarz. Bogen im Speedup wird länger Einzelne Prozessoren bringen also nicht mehr so große Verbesserung -> Effizienz wird kleiner. Bereits bei n=1000 nicht mehr sinnvoll zu Parallelisieren, da  $P_0$ -Anteil mit wachsender Feldgröße mit-skaliert und mehr in's Gewicht fällt.

Relationen bleiben.

### 3.3 Interpretation der Messergebnisse

### 3.4 Unterschied zum einfacheren und Abschluss

Vermutlich bei höherer Feldgröße noch sinnvoll, weil Arbeit von  $P_0$  nicht so hart -> kann abbrechen. Also: Im Vor allem muss  $P_0$  die Figur-Grenzen nicht neu suchen. Entweder die Grenzen untereinander passen zusammen oder nicht -> Vergleichsoperationen 2 (Anfang und Ende) \* Anzahl der Sub-Raster im Alles schwarz

Meiner: Muss bspw. Alles schwarz nochmal rechnen, wo die Figur aufhört um zu ermitteln, in welchen Teil-Rastern Figur liegt dort dann Prüfungen machen.

NWK wäre dann vermutlich auch ein wenig besser. Dennoch besteht das Problem, dass ...

## 4 FAZIT

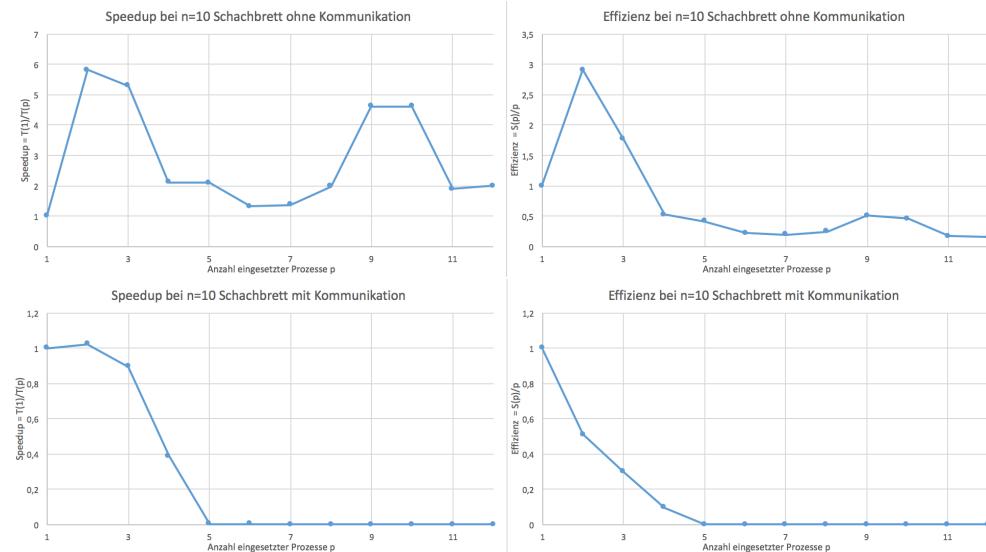
### 4.1 Einschätzung des Nutzens einer Parallelisierung

### 4.2 Mögliche Optimierungen

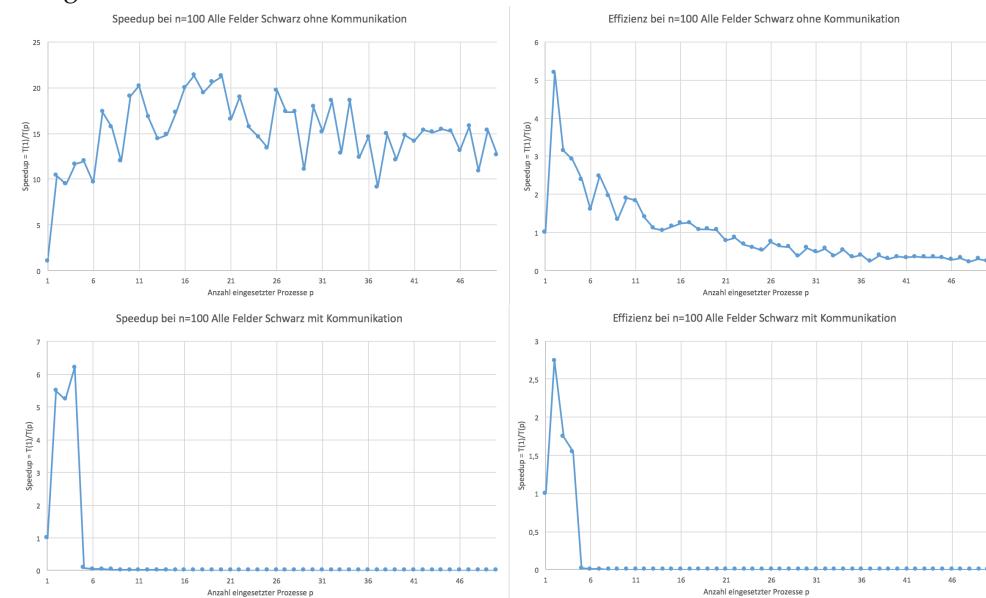
## A ANHANG

- A.1 Code: DRT1**
- A.2 Code: DRT2**
- A.3 Code: hostfileGenerator.c**
- A.4 Code: bsCreateHostfiles.sh**
- A.5 Code: bsRunGridTest.sh**
- A.6 Alle Messergebnisse**

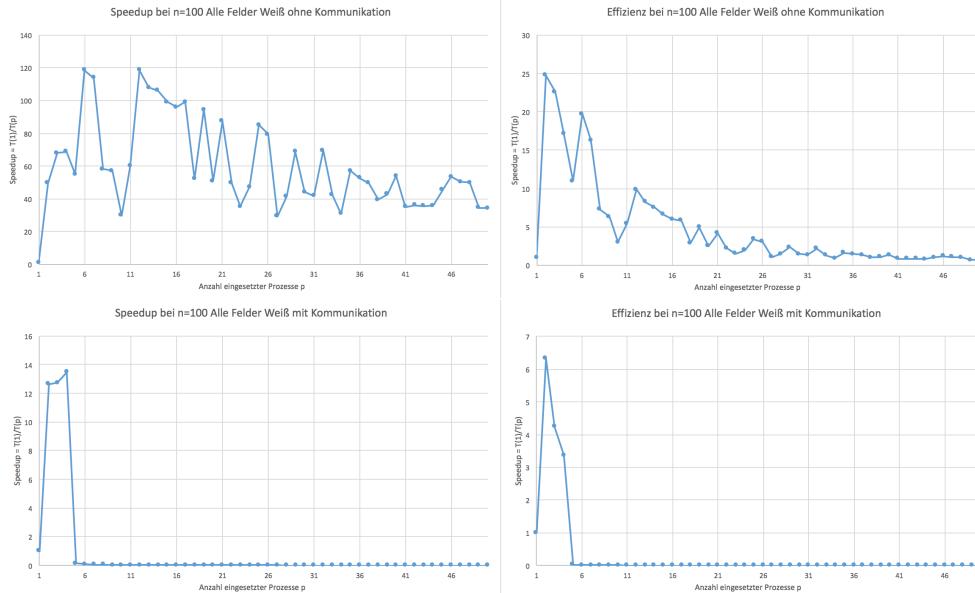
### Feldgröße 10x10 - Schachbrett



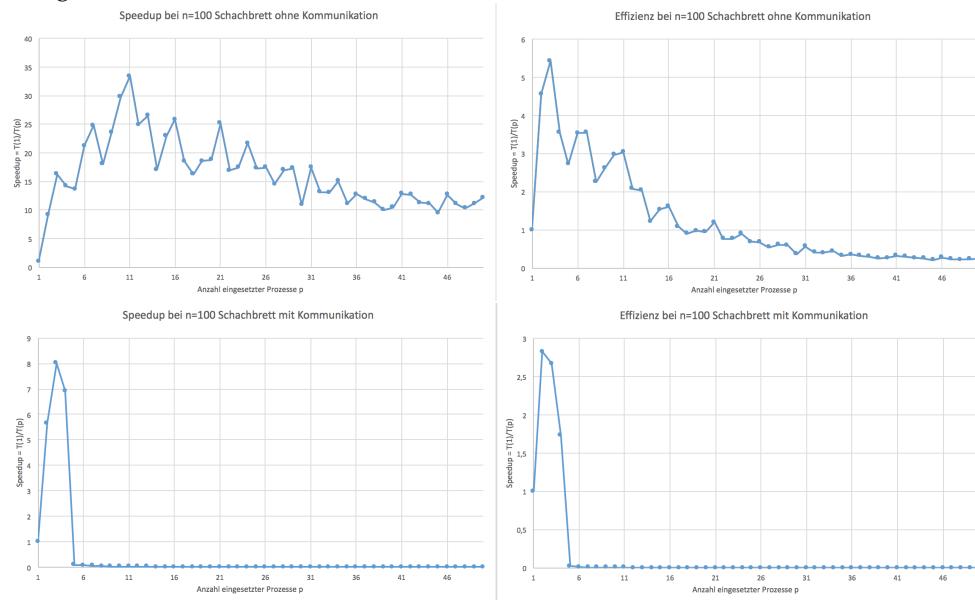
### Feldgröße 100x100 - Alle Felder Schwarz



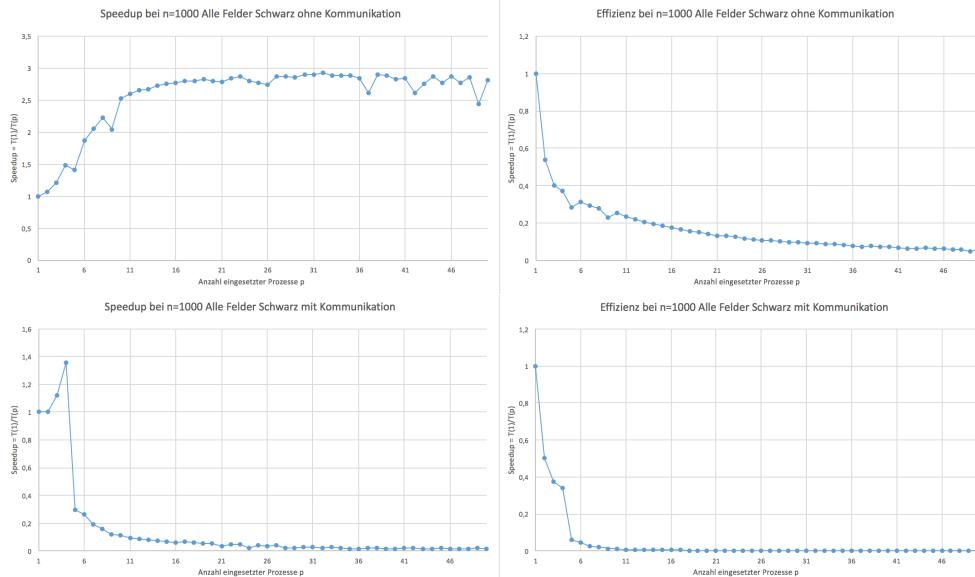
### Feldgröße 100x100 - Alle Felder Weiß



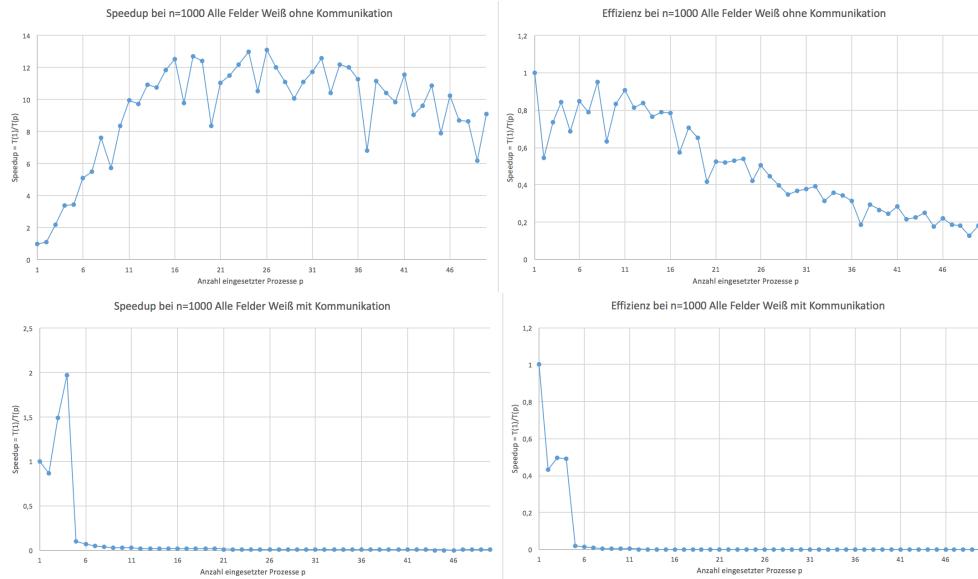
### Feldgröße 100x100 - Schachbrett



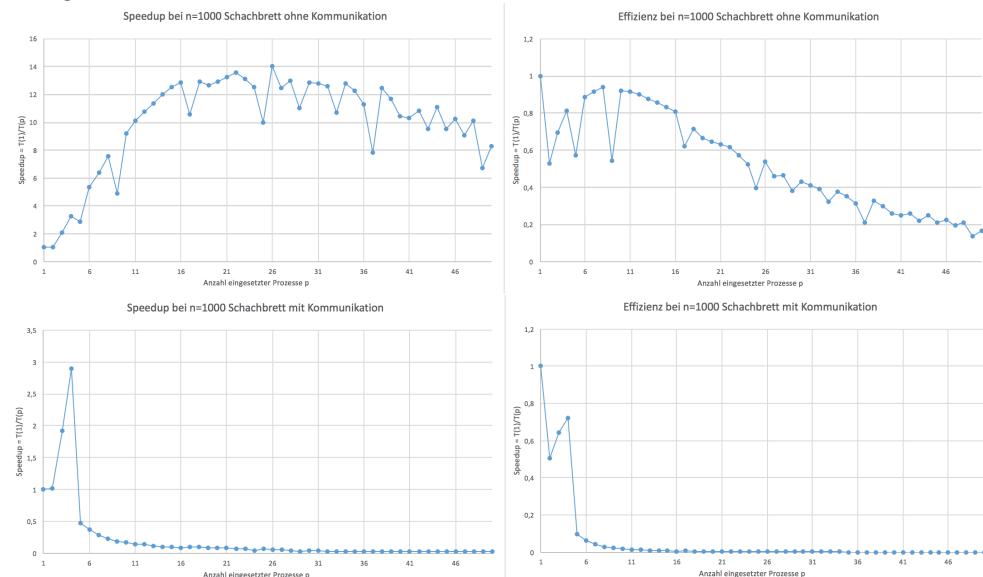
### Feldgröße 1000x1000 - Alle Felder Schwarz



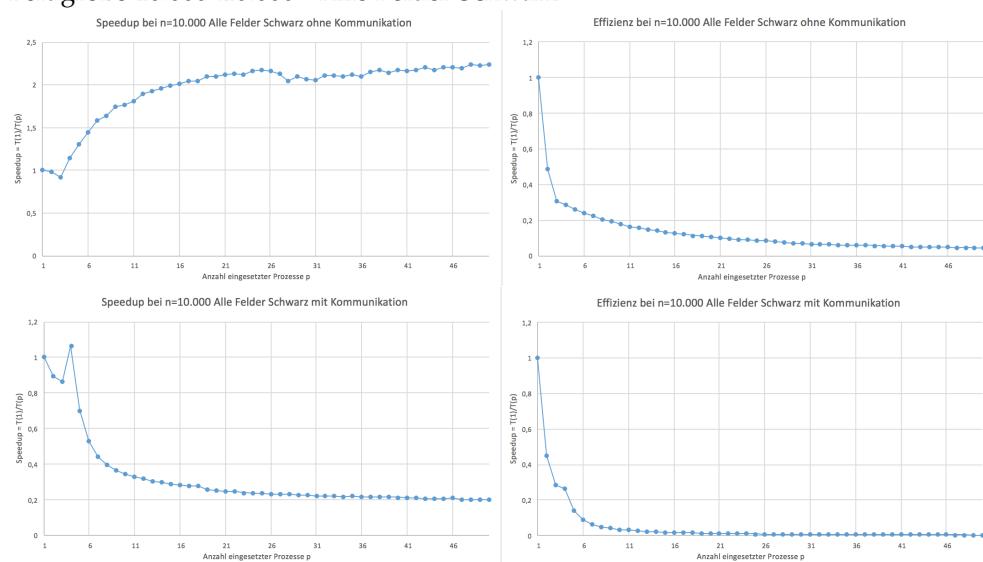
### Feldgröße 1000x1000 - Alle Felder Weiß



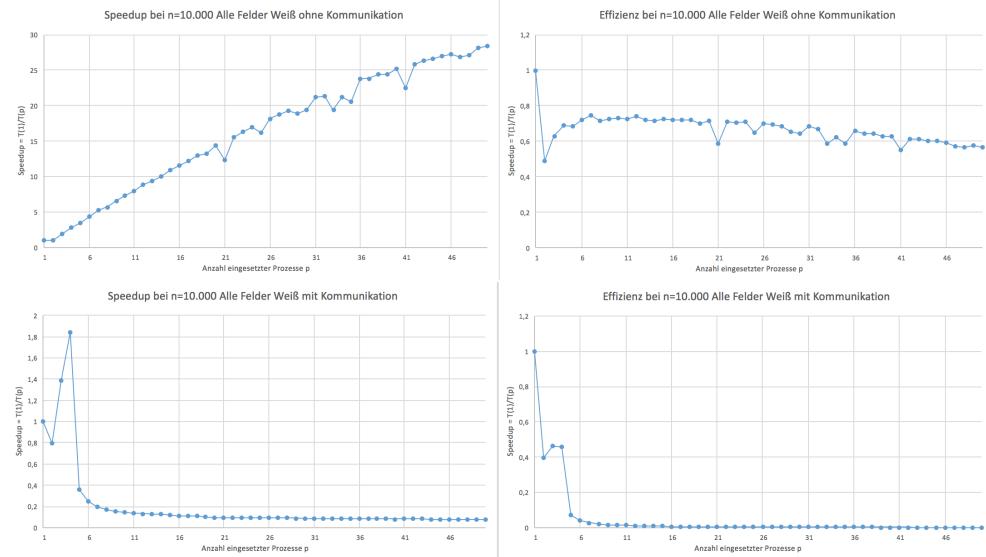
### Feldgröße 1000x1000 - Schachbrett



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



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



### Feldgröße 10.000x10.000 - Schachbrett

