Message Passing Programmierung

Projektaufgabe 1

Rechteckmustererkennung

Andrej Lisnitzki und Max Winkler
Hochschule für Technik, Wirtschaft und Kultur Leipzig
Fakultät Informatik, Mathematik und Naturwissenschaften

Professor: Herr Prof. Dr.-Ing. Axel Schneider

Leipzig, 27. Januar 2017

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Programmierumgebung

Das Projekt wurde mit der Programmiersprache C und C++ geschrieben, wobei für die Realisierung der Parallelisierung C-Prozeduraufrufe der MPI - Bibliothek benutzt wurden. C++ wurde hauptsächlich für komfortablen Umgang mit Zeichenketten (Strings) und Dateioperationen benutzt.

Da im Programm C++ Code benutzt wurde, soll das Programm mit dem **mpicxx** - Compiler kompiliert. Es werden auch Funktionen aus dem **C++11** - Standard benutzt, deswegen beim Kompilieren soll entsprechendes Flag gesetzt werden. Vollständiger Kompilierbefehl sieht wie folgt aus:

```
mpicxx -std=c++11 main.cpp -o project_c
```

2 Aufbau des Programms

Das Programm ist logisch auf zwei Abschnitte unterteilt:

1. Auswerten der Aufrufparameter und
2. Ausführung der gewählten Option

Diese werden im Folgendem detaillierter Beschrieben.

2.1 Aufrufparameter

Dieses Projekt wurde auf einem Rechner mit 4 physikalischen (8 logischen) Rechenkernen entwickelt und größtenteils auch auf diesen Rechner getestet. Um die Anzahl der Recheneinheiten zu steuern wurde ein Parameter **-np n** benutzt, wo **n** die Anzahl der Recheneinheiten bedeutet.

Das Programm hat mehrere Aufrufparameter. Es gibt erforderliche, optionale und miteinander kombinierte Parameter. Diese werden in der folgenden Tabelle aufgelistet.

		Anforderung	
Aufrufparameter	**Parameterargument**	**Parameter**	**Parameterargument**
-f	Dateiname	erforderlich	erforderlich
-g	n	optional	erforderlich
	n_x_y_h_w		erforderlich
-p		optional	
-r		optional	
-d	µSec	optional	optional
-o		optional	
-t	Versuche	optional	optional
-w	Teilstring	optional	optional

Tabelle 1: Alle im Programm existierende Aufrufparameter

In der ersten Spalte in der Tabelle 1 ist zu sehen, dass die Aufrufparameter unterschiedlich nach rechts eingerückt sind. Das bedeutet, dass die Aufrufparameter voneinander abhängen. Zum Beispiel -o hängt von **-r** ab und **-r** hängt von **-f** ab. In der dritten und vierten Spalte wird angegeben ob der Aufrufparameter bzw. deren Parameterargument optional oder erforderlich ist. Im Folgendem werden die Aufrufparameter näher beschrieben.

-f Dateiname

Dieser Parameter gibt an, welche **Datei** zum Testen des Programms oder welcher **Dateiname** zum Generieren der Testdatei verwendet werden soll.

-g n oder **-g n_x_y_h_w**

Mit diesem Parameter wird eine **n*n** Matrix generiert. Es gibt eine Möglichkeit einen einfachen Rechteck in die Matrix einzulegen. Dabei soll man eine obere linke Ecke mit **x_y** – Koordinaten[1] sowie dessen Breite und Höhe mit **h_w** angeben. Alle vier Parameterargumente sollen mit einem Unterstrich getrennt werden. Die generierte Datei kann mit einem beliebigen Texteditor geöffnet und editiert werden. Weiße Felder werden mit dem Punkt(.) und schwarzen Felder mit dem großen **X** dargestellt. Auf diese weise kann man einzelne Matrixfelder editieren.

-p

Damit kann man die generierte Matrix auf der Konsole ausgeben.

-r

Dieser Parameter startet ein Rechteckmustererkennung - Algorithmus.

-o

Mit diesem Parameter werden einige sinnvolle Informationen, welche beim Ablauf des Algorithmus entstehen, auf der Konsole ausgegeben. Diese sind die aufgeteilten Blöcke, welche den einzelnen Prozessoren zugesandt wurden und die Ergebnisse der Rechteckmustererkennung, welche dem Master - Prozess zugesendet werden.

-d oder **-dµSec**

Da die einzelnen Prozessoren nicht in der richtigen Reihenfolge die Information, welche mit dem **-o** - Parameter ausgegeben wird, auf der Konsole ausgegeben wird, soll diese Ausgabe zeitversetzt geschehen. Die einzelnen Prozesse werden einfach mit der bestimmten Zeitlänge schlafengelegt. Die Zeitversetzung wird mit der folgenden Formel berechnet: $delay = rank \cdot \mu Sec$. Der Standardwert der Verzögerung liegt bei 1000µs.

-t oder **-tVersuche**

Mit diesem Parameter wird die reine Rechteckfindung auf den Prozessoren mehrmals wiederholt um die Fehlerquote der Zeitmessung, welche durch Beanspruchen der

1 Die Indexierung fängt bei 0 an.

Rechenzeit durch die Anderen verursacht werden kann, zu reduzieren. Der Standartwert beträgt 1.

-w oder **-wTeilstring**

Mit diesem Parameter werden die Ergebnisse der Zeitmessung in eine csv - Datei geschrieben. Ein **Teilstring** wird am Ende der Dateiname angehängt.

Zu Beachten: Alle optionale Parameterargumente sollen **ohne Leerzeichen** hinter dem Aufrufparameter geschrieben werden!

Ausführung der gewählten Option geschieht im Prozess mit dem Rang 0 (im Folgendem - Masterprozess) um die sequentielle Aufgaben nicht parallelisieren zu können. Falls die Aufrufparameter falsch mit einander kombiniert angegeben werden, wird ein entsprechender Fehlercode auf der Konsole ausgegeben und das Programm beendet.

2.2 Programm - Fehlerbehandlung

Beim Aufruf des Programms können einige Fehler auftreten, welche durch falsche Kombination der Programmaufrufparameter oder durch unzulässige interne Programmoperationen verursacht werden. Die entsprechenden Fehlercodes sind im Quellcode als **#define**'s - Liste aufgelistet. Diese sind auch in der folgenden Tabelle dargestellt.

#define	Fehlercode	kurze Beschreibung
ERR_OPT	-1001	unbekannte Aufrufparameter
ERR_OPT_DEFAULT	-1002	unbekannte Aufrufparameter
ERR_GEN_OPTS	-1003	falsche Gen.-Parameterargumente
ERR_FILENAME	-1004	Dateiname fehlt
ERR_GEN_RECT_OVERFLOW	-1005	Gen.-Parameterargumente passen nicht zu einander
ERR_FILE_OPEN	-1006	Fehler beim Öffnen der Datei
ERR_READ_DATA_ARGS	-1007	Fehler in readData()
ERR_DATA_PROC_DIM	-1008	Anzahl der CPUs zu groß
ERR_BAD_ALLOC	-1009	Speicher-Alloc
ERR_TO_FEW_PARAMS	-1010	zu wenig Programmaufrufparameter
ERR_TO_FEW_CPUS	-1011	zu wenig CPUs

Tabelle 2: Fehlercodes aus dem Programm

2.3 Beschreibung des Rechteckmustererkennung - Algorithmus

Als erstes werden die Daten aus der Datei, welche mit dem **-f** - Flag angegeben wurde, gelesen und in ein eindimensionales char - Feld geschrieben. Danach wird die Dimension der Matrix mit der Anzahl der verfügbaren Prozessoren verglichen. Falls die Anzahl der Prozessoren größer als die Dimension der Matrix ist, wird der entsprechender Fehlercode auf der Konsole ausgegeben und das Programm wird beendet. Falls das Dimensionsproblem nicht auftritt, wird die Matrix horizontal auf Datenblöcke aufgeteilt. Die Anzahl der Zeilen sowie die Dimension

wird in einem Datenblock vom Masterprozess an die Slaveprozesse mit der **MPI_Bcast()** - Funktion gesendet. Die Matrix wird so aufgeteilt, dass jeder Prozessor gleich viele Zeilen zu bearbeiten hat. Restliche Zeilen werden möglichst gleichmäßig verteilt. Zum Beispiel: Eine 10*10 Matrix wird auf 3 Prozessoren verteilt ⟶ da die Division (10/3) nicht aufgeht, bekommt jeder Prozessor je 3 Zeilen und einer bekommt zusätzlich die übrige Zeile.

Als Nächstes werden die Datenpakete an die einzelne Prozessoren mit der **MPI_Send()** - Funktion gesendet. Dabei wurde festgelegt (in unserem Fall), dass die maximale Matrix-dimension den Wert $\lfloor\sqrt{2147483647}\rfloor = 46340$ [2] annehmen kann. Es liegt an der **MPI_Send()** - Funktion, welcher ein **int count** - Parameter übergeben werden soll. Da der Parameter vom Typ **int** ist, kann auch dann nur eine Matrix der Dimension 46340 gesendet werden. Mann kann auch eine größere Matrix senden in dem man als Datentyp **long long** nimmt und die Pixelinformation im einzelnen Bit kodiert.

Anschließend wird eine **MPI_Recv()** - Funktion gestartet und auf die einzelne Ergebnisse gewartet.

2.3.1 Beschreibung des Rechteckfindung - Algorithmus

Das Rechteckfindung - Algorithmus besteht aus zwei for - Schleifen. Die äußere Schleife adressiert Zeilen und innere Schleife adressiert Zeichen in der Zeile (also Spalten). An der ersten Stelle wird geprüft ob ein Zeichen dem 1 - Wert[3] entspricht. Falls dies der Fall ist, werden alle vier Variablen (links, rechts, oben, unten) mit dem entsprechenden Wert initialisiert und der **inRect** - Flag, welcher das Befinden im Rechteck signalisiert, wird auf 1 gesetzt. Dies bedeutet, dass man sich in einem Rechteck befindet. Am Ende der Rechteckzeile wird die Position der letzten Spalte in dieser Zeile in eine **ir1First** - Variable gespeichert. Die **ir1First** - Variable ermöglicht in den nächsten Zeilen zu prüfen, ob der Rechteck seinen rechten Rand immer in der gleichen Spalte hat. Dies schließt folgende Fälle aus:

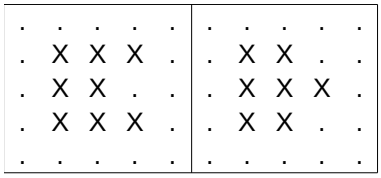

Abbildung 1: Fälle, welche mit der ir1First - Variable ausgeschlossen werden

Wenn man auf einer Zeile im Rechteck ist und danach eine weiße Fläche trifft, dann wird die **inRect** - Variable auf 0 gesetzt. Falls man auf der selben Zeile wieder eine schwarze Fläche Trifft, dann wird ein zweites Rechteck erkannt. Beim Übergang auf die nächste Zeile wird immer geprüft, ob der linke Rand immer in der gleichen Spalte bleibt. So werden folgende Fälle erkannt:

2 2147483647 – maximaler int - Wert
3 1 – Schwarz, 0 – Weiß

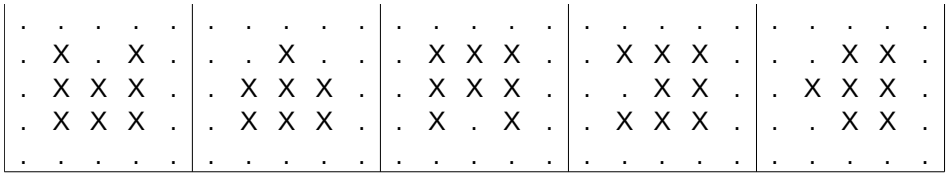

Abbildung 2: Fälle, welche mit der oben beschriebenen Verfahren ausgeschlossen werden

Die Resultate der einzelnen Prozessoren werden in ein int - Feld mit 6 Einträgen (Rang, R, i0, i1, j0, j1)[4] geschrieben und an den Masterprozess gesendet. Die Koordinate der gefundenen Rechtecke werden auf den jeweiligen Datenblock (ein Teil der gesamten Matrix) bezogen.

2.3.2 Beschreibung des Algorithmus über das Vorliegen des Rechtecks

Nachdem der Masterprozess alle Ergebnisse von den Slaveprozessen in ein Ergebnisvektor gesammelt hat, rechnet er die Koordinate der gefundenen Rechtecke der einzelnen Prozessoren auf die Koordinate der ganzen Matrix um. Diese Operation geschieht sehr schnell und die dabei entstehende Resultate können mit dem **-o** - Aufrufparameter auf der Konsole ausgegeben werden.

Nun werden die Ergebnisse ausgewertet. Da der obere und der untere Rand des Rechtecks schon bei den einzelnen Slaveprozessen geprüft wurde, müssen nur noch wenige Restbedingungen geprüft werden. Als Erstes wird die Bedingung geprüft, ob die Rechtecke der einzelnen Prozessoren zusammenhängend[5] sind. Weitere Bedingung[6] ist, dass der linke und der rechte Rand die selbe i - Koordinate haben. Falls im Ergebnisvektor der **R**-Wert den Wert 0 aufweist, wird der Vorgang mit der Meldung „Es gibt kein zusammenhängendes Rechteck!“ abgebrochen. Falls ein zusammenhängendes Rechteck gefunden wurde, werden die Koordinaten des Rechtecks zusammen mit der Meldung „Es gibt ein zusammenhängendes Rechteck!“ auf der Konsole ausgegeben.

3 Laufzeitmessungen

Im Programm werden verschiedene Zeiten gemessen. Eine schnelle Übersicht im Quellcode befindlichen Variablen, in denen die gemessene Zeit gespeichert wird, kann man aus der folgenden Tabelle entnehmen.

Im Quellcode bezeichnet Variable	**Benötigte Zeit für:**
tAll	Gesamte Laufzeit
tRD	Auslesen der Daten aus einer Datei in eine Matrix
tSendInit	Versenden an die Slaves der Initinformation
tSendData	Versenden an die Slaves der Datenblöcke
tPAll	Gesamte Zeit der Berechnungsversuche
tCR	Auswertung der Teilergebnisse

Tabelle 3: Beschreibung im Quellcode befindlichen Variablen mit der Zeitinformation

4 R, i0, i1, j0, j1 – siehe Aufgabestellung
5 Siehe 2. Bedingung in der Aufgabestellung.
6 Siehe 1. Bedingung in der Aufgabestellung.

Die **tAll** - Variable enthält die Zeit, welche das Programm nach dem Betreten des Blocks, welches mit dem **-r** - Aufrufparameter freigeschaltet wird, verstricht. Dieser Block enthält alle weitere Operationen, welche in der Tabelle 3 nach der **tAll** - Variable aufgelistet sing.

Wichtig ist zu erwähnen, wie die parallele Arbeit der Slaveprozessoren gemessen wird. Es wird unmittelbar vor der **findRecktInBlock()** - Funktion[7] eine **MPI_Barrier()** - Funktion gestartet um das synchrone Starten der Slavejobs zu starten. **MPI_Barrier()** - Funktion wird auch im Masterprozess unmittelbar vor dem Start der Zeitmessung gestartet. Die Zeitmessung der Berechnung wird erst dann beendet, wenn alle Slaveprozesse seine Ergebnisse dem Master zusenden. Das Ergebnis eines Prozesses ist nur $6 \cdot \text{sizeof(int)} = 24\,Bytes$ groß, deswegen dient diese Nachricht auch als Etikett für das Ende der Berechnung der Slaveprozesse. Diese Zeitmessung wird **t**-Mal[8] wiederholt und in ein Vektor geschrieben. Bei der Ausgabe der Zeitmessungen auf der Konsole, was mit dem **-t** - Programmaufrufparameter aktiviert werden soll, wird ein Mittelwert aus den wiederholten[9] Zeitmessungen berechnet. Dabei wird auch die Varianz und die quadratische Standartabweichung berechnet und falls es große Abweichung gibt, werde die Zeitmessung wiederholt.

Der Master kümmert sich nur für den sequentiellen Programmstück (Versenden der Datenpakete, Auswertung der Ergebnisse), deswegen die Anzahl der Prozessoren in Diagrammen immer ungerade (Master wird abgezogen).

3.1 Laufzeitmessungen mit dem festen n[10]

Die Laufzeitmessungen wurden mit dem festen n=30000 durchgeführt. Dabei wurden nur 2 CPUs pro Rechner benutzt. Bei den Messungen wurden immer drei Fälle durchgespielt:

- Die Matrix ist komplett weiß.
- Die Matrix ist komplett schwarz.
- Ein schwarzes Rechteck 10000 x 10000 ist in der Mitte der Matrix.

Das Beispielkommando zum Starten des Programms mit der Ausgabe der Resultate in eine csv - Datei sie wie folgt aus:

```
mpirun -hostfile myhosts2 -npernode 2 project_c -f Dateiname -r -t11 -w
```

wobei **myhosts2** – eine txt - Datei mit zwei Rechnernamen, **-f Dateiname** – die txt - Datei mit der Testmatrix, **-r** – ruft den Rechteckmustererkennung – Algorithmus auf, **-t11** – Anzahl der Messversuche (11-1=10 Mal) und **-w** – speichert Ergebnisse in eine Dateiname.csv - Datei ab. Die Messergebnisse sind in den folgenden Abbildungen zu sehen.

7 Diese Funktion ist für das Finden des Rechtecks im Datenblock verantwortlich.
8 Siehe **-t** - Programmaufrufparameter
9 Außer der ersten Messung
10 n – Problemgröße (Matrixdimension)

3.1.1 Nur parallele Rechenzeit

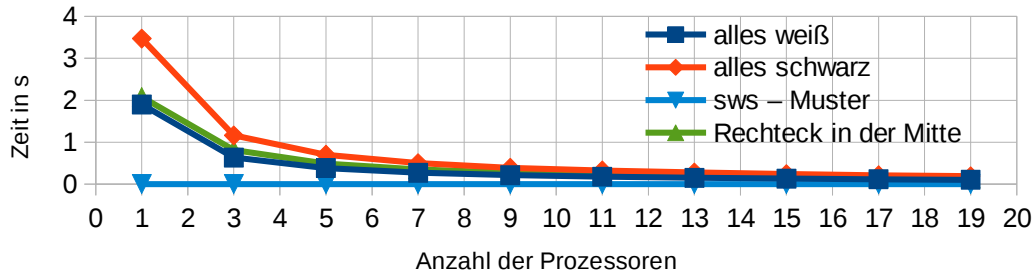

Abbildung 3: Laufzeitverhalten (nur parallele Rechenzeit) mit n=30000

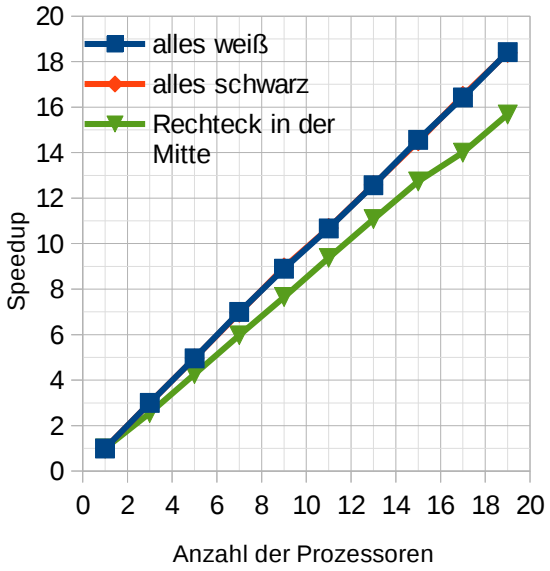

Abbildung 4: Speedup (nur parallele Rechenzeit) mit n=30000

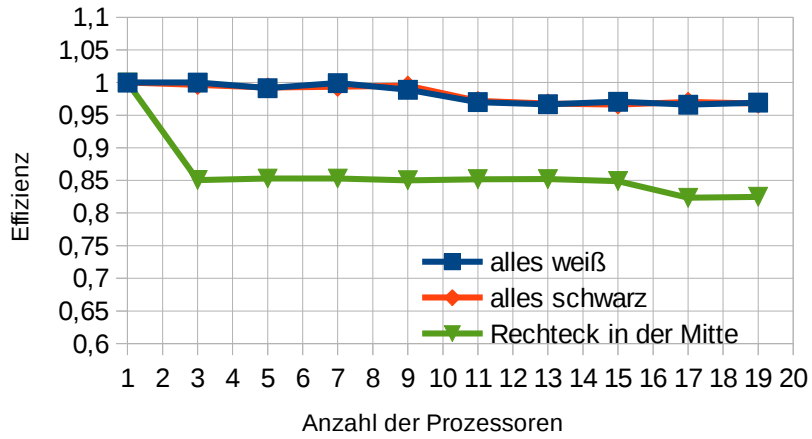

Abbildung 5: Effizienz (nur parallele Rechenzeit) mit n=30000

Bei der Abbildungen 3, 4 und 5 wird nur die Zeit gemessen, welche für rein parallele Berechnungen gebraucht wurde. In der Abbildung 3 werden Laufzeitverhalten der einzelnen Fällen dargestellt. Die schnellste Variante ist die, in welcher die erste und die dritte Spalte der Matrix ist schwarz ist. Dies ist einfach zu erklären, weil der Rechteckmustererkennung – Algorithmus erkennt schon beim dritten Pixel, dass es um mindestens zwei nicht zusammenhängende Rechtecke handelt und bricht sofort ab.

Die nächst schnellste Variante ist die, in welcher die Matrix komplett weiß ist. Der Algorithmus schaut in diesem Fall einzelne Pixel an und falls diese weiß sind geht er zu den nächsten. So muss er die gesamte Matrix durchsuchen in der Hoffnung, das er ein Rechteck findet.

Die langsamste Variante ist die, wo die Matrix komplett schwarz ist. In diesem Fall schaut der Algorithmus in die einzelne Zelle der Matrix rein, stellt fest das diese schwarz ist und setzt die Koordinaten des Rechtecks immer neu. Dabei werden noch einige Bedingungen zum Zusammenhang des Rechtecks beim Zeilenwechsel geprüft. Diese Operationen verbrauchen zusätzliche Rechenleistung und deswegen ist das der Worst Case für dieses Algorithmus.

Die Variante mit dem schwarzen Rechteck in der Mitte positioniert sich in der Mitte, was sich auch aus den oben genannten Gründen herauskristallisiert.

Auf der Abbildung 4 ist der Speedup, welcher mit der zunehmender Anzahl der CPUs fast linear nach oben wächst, zu sehen. Die Effizienz, welche auf der Abbildung 5 zu sehen ist, bleibt fast immer auf dem gleichen Niveau. Dies kann man damit begründen, dass in diesem Fall nur die Zeit für rein parallele Berechnungen und das Zurücksenden der Resultate an Master gemessen wird. Die leicht abweichende grüne – Kurve[11] kann man damit erklären, dass je nach Datenblock die Slaveprozessoren unterschiedlich schnell mit der Lösung der Aufgabe werden. Da der Fall mit dem sws - Muster zu sehr streut (wegen sehr kurzen Ausführungszeit), wurde dafür kein Speedup- und Effizienz - Diagramm erstellt.

11 Der Fall mit dem schwarzen Rechteck in der Mitte der Matrix.

3.1.2 Gesamte Rechenzeit

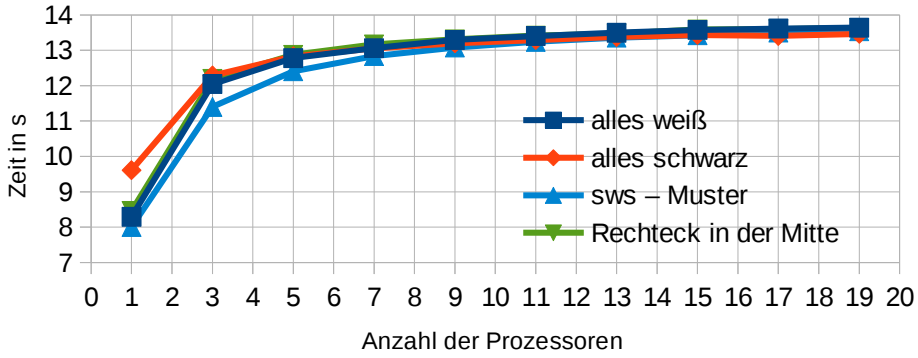

Abbildung 6: Laufzeitverhalten (gesamte Rechenzeit) mit n=30000

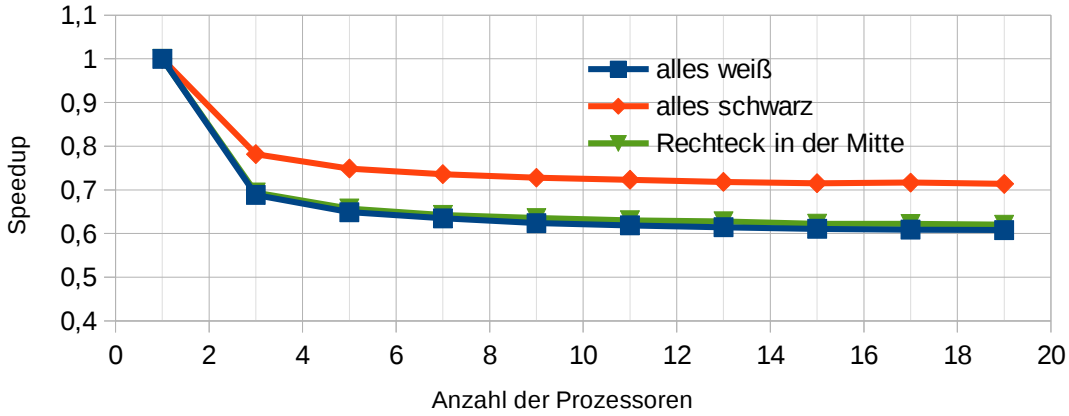

Abbildung 7: Speedup (gesamte Rechenzeit) mit n=30000

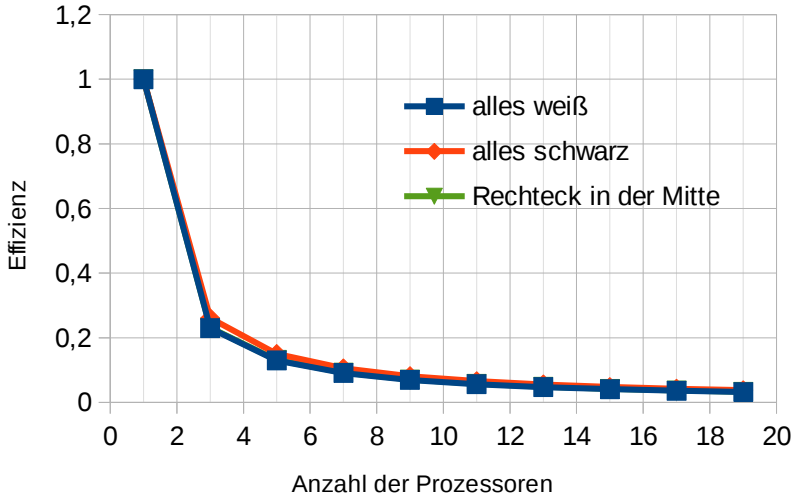

Abbildung 8: Effizienz (gesamte Rechenzeit) mit n=30000

Die Abbildungen 6, 7 und 8 wurden jetzt mit der Berücksichtigung der gesamten Rechenzeit, welche für das Rechteckmustererkennung – Algorithmus benötigt wird, erstellt. Abbildung 6 zeigt deutlich, dass mit der Zunahme der CPUs (Rechnerknoten) die Laufzeit nach oben wächst. Der Speedup[12] wird schon ab drei CPUs (zwei Rechner) fast konstant, was man auch über die Effizienz sagen kann. Die Ursache für schlechte Speedup und Effizient stellen die sequentielle Programmstücke im Algorithmus dar. In unserem Fall sind das das Laden der Datei, was bei n=30000 konstant[13] ca. 6 Sekunden in Anspruch nimmt, und das Versenden der Datenpakete an die Slaves. Das Laufzeitverhalten für das Versenden der Datenpakete kann man in der folgenden Abbildung beobachten.

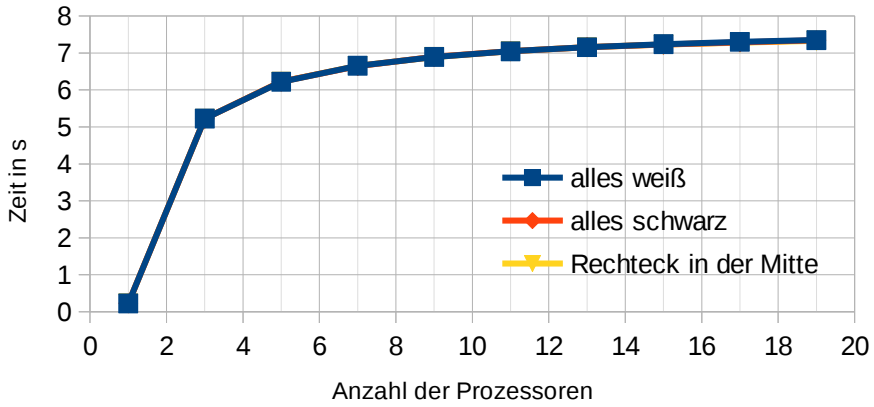

Abbildung 9: Laufzeitverhalten fürs Versenden der Datenpakete an die Slaves mit n=30000

3.2 Laufzeitmessungen mit festen p[14]

Die Laufzeitmessungen wurden mit p=1, 7 und 15 und mit der Matrixdimensionen in n=5000 Schritten durchgeführt. Da das Laufzeitverhalten für verschiedene Matrix – Fälle im vorherigen Kapitel beschrieben wurde, wurden hier die Zeitmessungen nur mit der komplett weißen Matrix durchgeführt.

12 Siehe Abbildung 7
13 Konstant, da die Datei unabhängig von dem Inhalt immer gleich groß ist.
14 p – Anzahl der Prozessoren

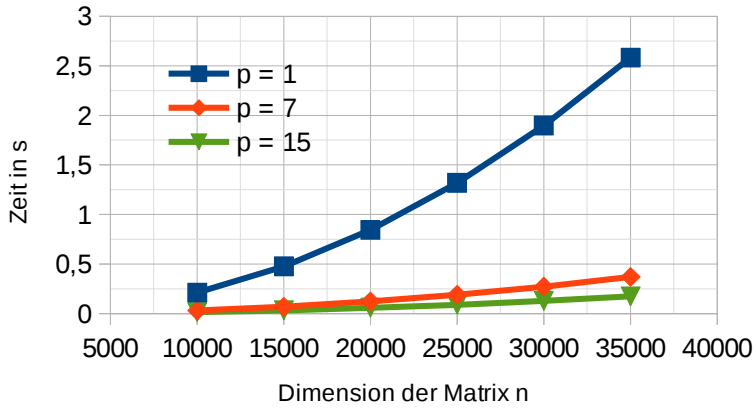

Abbildung 10: Parallele Rechenzeit

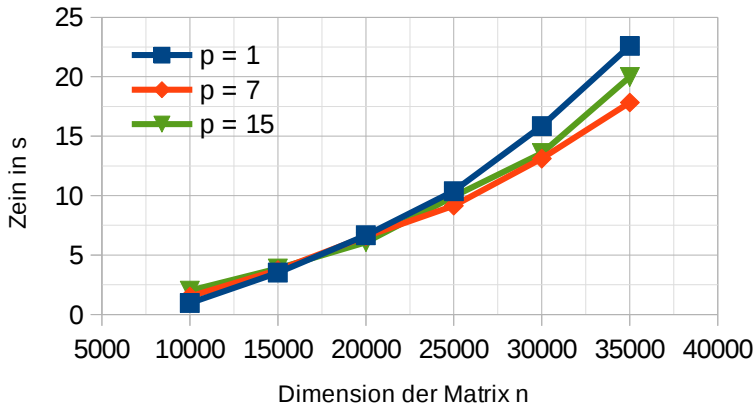

Abbildung 11: Gesamte Rechenzeit

Aus der Abbildung 10 kann man entnehmen, dass die Rechenzeit der parallelen Aufgabe mit der steigenden Anzahl der CPUs immer kleiner wird. In der Abbildung 11 kann man etwas Interessantes beobachten. Bis einer Dimension von ca. n=17000 rechnet der Cluster mit mehreren CPUs schneller als nur eine CPU. Dies liegt wahrscheinlich an dem, dass das Lesen der Datei und das Rechnen der Aufgabe (im Cluster mit mehreren CPUs) bis zu der Matrixdimension von ca. n=17000 schneller als das Versenden der Datenpakete an die einzelne Slaves ist. Dies kann man aus folgenden Abbildungen entnehmen.

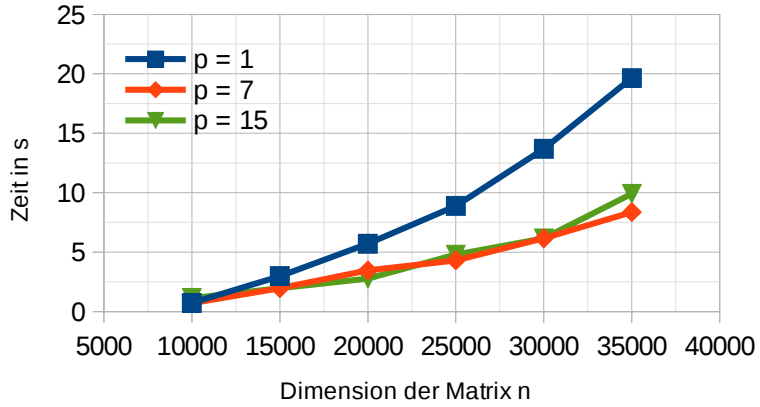

Abbildung 12: Auslesen der Datei mit der Matrixinformation

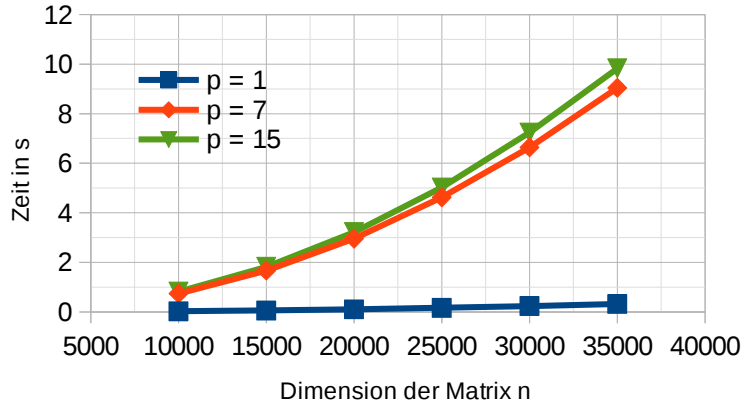

Abbildung 13: Versenden der Pakete aneinzelne Prozessoren p

In der Abbildung 13 ist zu sehen, dass eine CPU wesentlich schneller ein Datenpaket kriegt als die Clustern mit mehreren CPUs. Dies liegt an dem, dass der Master- und der Slave - Prozess (CPU - Kern) auf einem physikalischen Prozessor befinden und das Transferieren der Daten zwischen den Kernen wesentlich weniger Zeit in Anspruch nimmt als der Versand der Datenpakete per Netzwerk an verschiedene Rechnerknoten.

4 Fazit

Nach dem Umsetzen des Projekts konnte festgestellt werden, dass die parallele Realisierung des Algorithmus für n>17000 für das Problem der Rechteckmustererkennung konnte keine Geschwindigkeitsvorteile im Vergleich zu der sequentieller Variante geben. Auch für n<17000 ist der Geschwindigkeitsvorteil nicht signifikant groß. Dies liegt vor Allem, dass das Problem an sich zu einfach ist um diese im Cluster rechnen zu lassen. Die gute Parallelisierung der Aufgabe bringt durch die langsame Netzwerkkommunikation, welche zur Versendung der Datenpakete an einzelne Rechner genutzt wurde, fast keinen Vorteil. Man konnte hohe

Netzwerkkommunikation durch das Abspeicherung der Pixelinformation pro ein Bit um 8 Mal (in unserem Fall) reduzieren.

Für dieses Problem macht ein sequentielles Algorithmus mehr Sinn. Die Parallelisierung schwerer Probleme (Algorithmen) mit kleinem Kommunikationsoverhead kann wesentlich höhere Speedups und Effizienz aufweisen.

5 Quellcode