Dokumentation zu den Projektaufgaben 1 und 2

Message Passing Programmierung

Matthias Zober und Michael Horn

25. Januar 2017

Inhaltsverzeichnis

1 Überblick

In dieser Dokumentation werden die Aufgaben „Rechteckmustererkennung“ sowie „Numerische Integration mittels Parabelformel“ betrachtet und hinsichtlich ihrer Parallelität untersucht. Der entstandene Quellcode kann unter:
`https://github.com/MZober1993/MessagePassingProjects`[1] eingesehen werden.
Ebenfalls befinden sich in diesem Repository alle ermittelten Messwerte als CSV-Dateien sowie die erstellten Skripte zur Ermittlung der Laufzeiten.

2 Projektaufgabe 1: Rechteckmustererkennung

In diesem Abschnitt wird die Umsetzung der Projektaufgabe 1 dargestellt, die Verwendung erläutert sowie Laufzeitverhalten, Speedup und Effizienz untersucht. Außerdem erfolgt eine Auswertung am Ende dieses Abschnittes. Das zu spezifizierende parallele Programm soll eine Rechteckmustererkennung für ein Digitalbild über ein Cluster realisieren.

2.1 Realisierung

Für die Umsetzung eines solchen Programms ist es zunächst notwendig eine sequentielle Erkennung eines $n * m$ Pixel Bildes zu realisieren. Für die Parallelisierung muss jeder Prozessor diese sequentielle Erkennung auf sein Teilbild anwenden. Hierzu ist es notwendig, dass nach dem Einlesen des Bildes durch einen ausgezeichneten Hauptprozess (`Master`), die resultierenden Teilbilder auf alle Prozesse verteilt werden. Diese Aufteilung wird im Programm durch `MPI_Scatter` realisiert, durch diese Verteilung muss eine ungerade Prozessoranzahl ausgeschlossen werden.

Im Wesentlichen wird ein Rechteck durch das Finden eines schwarzen Pixels (schwarz=0, weiß=1) und das Überprüfen von ausschließenden Mustern gefunden. Die Erkennung kann realisiert werden durch Speicherung der Koordinaten des ersten (`startX` und `startY`) und zuletzt gefundenen Pixels (`stopX` und `stopY`) und die Länge der oberen gefunden Rechteckkante (`xSize`). Im folgenden erkennt man die ausschließenden Mustern mit deren Regeln.

[1] Letzter Aufruf: 25. Januar 2017

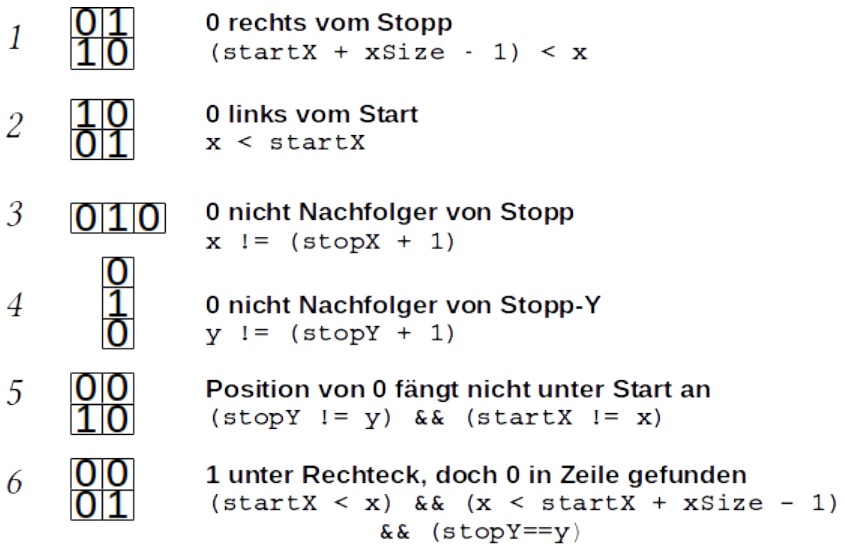

Abbildung 1: Ausschließende Bit-Muster mit Regeln

Werden diese Muster nicht erkannt und schwarze Pixel existieren, dann existiert ein Rechteck (`FoundOne`). Andernfalls werden nicht zuzuordnende schwarze Pixel (`Mismatch`) oder gar keine schwarzen Pixel (`Nothing`) im Bild gefunden.

Nachdem jeder Prozess sein Teilbild bearbeitet hat, muss der `Master` alle Ergebnisse zusammenfassen. Zunächst sammelt der `Master` alle Teilergebnisse mit `MPI_Gather` ein, danach bewertet er die Ergebnisse in folgenden Schritten.

1. Gibt es ein Teilbild mit schwarzen Pixeln, welche aber kein Rechteck bilden, dann kann direkt abgebrochen werden mit: `Mismatch`.
2. Wenn alle Teilbilder `Nothing` finden, dann kann ebenfalls abgebrochen werden mit `Nothing`.
3. Gibt es nur ein `FoundOne` und sonst `Nothing`, kann das Rechteck ausgegeben werden.

Falls mehrere `FoundOne` vorliegen und es wurde nicht abgebrochen, dann muss der `Master` überprüfen ob die gefundenen Rechtecke zusammenhängend sind. Dies kann durch folgende Kriterien überprüft werden.

1. Die Start- und Stop-X Werte stimmen bei jedem Rechteck überein.

2. Bis auf den ersten Prozess der ein Rechteck in einer Zeile gefunden hat, müssen alle schwarze Pixel in ihrer ersten Zeile finden.

3. Bis auf dem letzten Prozess der ein Rechteck gefunden hat, müssen alle schwarzen Pixel in ihrer letzten Zeile finden.

Nach dieser Überprüfung kann der Master-Prozess das Ergebnis ermitteln und die Start- und Stopp-Koordinaten des gefundenen Rechtecks ausgeben.

Durch die Betrachtung der Umsetztung stellt man fest, dass für den Algorithmus ein Best-Case darin besteht sofort ein `Mismatch` zu finden, damit direkt abgebrochen werden kann. Der Average-Case stellt ein Digitalbild ohne schwarze Pixel dar, denn es kann nach überprüfen aller Pixel abgebrochen werden, aber es findet keine weitere Überprüfung statt. Der Worst-Case besteht darin mehrere Rechtecke zu finden, welche auf Zusammengehörigkeit untersucht werden müssen. Für die Untersuchungen zur Laufzeit werden diese drei Fälle berücksichtigt.

2.2 Verwendung

2.2.1 Ausführung

Der Quellcode kann mit Hilfe des folgenden Aufrufs kompiliert werden:

- `cmake CMakeLists.txt -DCMAKE_BUILD_TYPE=RELEASE && make`

Mif folgenden Befehl wird das Programm auf 4 Prozessoren ausgeführt:

- `mpirun -npernode 4 rectanglePatternDetection` **FILE**

Für das Programm können Kommandozeilenargumente zur Angabe des Konfigurationsfiles zur Erstellung des zu untersuchenden Digitalbildes angegeben werden.

2.2.2 Konfigurationsfile

Die Anwendung unterstützt drei Modi zur Erzeugung des Digitalbildes, welcher dieser Modi verwendet wird, steht immer in der ersten Zeile des Files.

Modus 2: Custom Mode Der `Custom Mode` ermöglicht die direkte Angabe einer $n * n$ Matrix (ab Zeile zwei) aus Nullen und Einsen.

Modus 1: Single Fields und Modus 0: Rectangle Um eine komfortable Eingabe zu realisieren, wird für die Modi 1 und 0 nicht die direkte Angabe der Matrix sondern nur die Angabe einzelne Parameter realisiert. Aus diesen Angaben wird dann das Digitalbild im Programm generiert. Die zweite Zeile beschreibt die Größe n und die Dritte die zu verwendende Hintergrundfarbe (0 oder 1). Aus diesen zwei Parametern kann im Programm leicht eine $n * n$ Matrix der angegebenen Hintergrundfarbe (0 oder 1) generiert werden. Durch die vierte Zeile werden Koordinatenpaare in der Form: `x y` angegeben. Modus 1 interpretiert diese Koordinaten als einzelne Felder, welche im Bild umgedreht $(0 \to 1, 1 \to 0)$ werden müssen. Modus 0 hingegen interpretiert nur die ersten drei Koordinatenpaare als Ortsvektoren der oberen Rechteck-Ecke, die das Rechteck aufspannen. Es ist notwendig, dass dabei die Reihenfolge: obere Ecke links, obere Ecke rechts und untere Ecke links eingehalten wird. Durch die angegebenen Kodierungen (Modus 1 und 0) ist es möglich sehr große Digitalbilder mit der Anwendung zu testen, ohne dass diese tatsächlich als physisch gespeichterte Dateien existieren müssen. Im Gegensatz dazu kann durch Modus 2 direkt die Matrix eingegeben werden, dieser Modus findet vorallem für kleinere n Anwendung. Durch das Konfigurationsfile können somit sehr leicht Digitalbilder mit den vorher betrachteten Fällen `Nothing`, `Mitmatch` und `FoundOne` für kleine und große n erzeugt werden.

2.3 Laufzeitverhalten

Für die Untersuchungen des Laufzeitverhaltens werden im folgenden feste n mit variierenden p bis maximal 44 und feste p mit variierende n bis maximal 40000 betrachtet. Wie man aus dem gegebenen Laufzeitmessungen (Abbildung 2 und Abbildung 3) für feste p entnehmen kann, verschlechtert sich die Laufzeit beider Messungen quadratisch. Dies ist mit dem Aufwand der Überprüfung des quadratischen Bildes zu begründen. Für den sequentiellen Fall liefern alle drei Fälle: `Mismatch`, `FoundOne` und `Nothing` wesentlich kürzere Laufzeiten, als für den parallelen. Dies ist damit begründet, dass im sequentiellen Fall keine Kommunikation für die Auswertung notwendig ist, da der `Master` direkt seine Suche abbrechen kann. Im parallelem Fall muss der `Master` auf alle Teilergebnisse der anderen Prozessoren warten, hierbei entsteht durch das Senden und Empfangen ein Kommunikationsoverhead, welcher bei steigender Anzahl der verwendeten Prozessoren erhöht

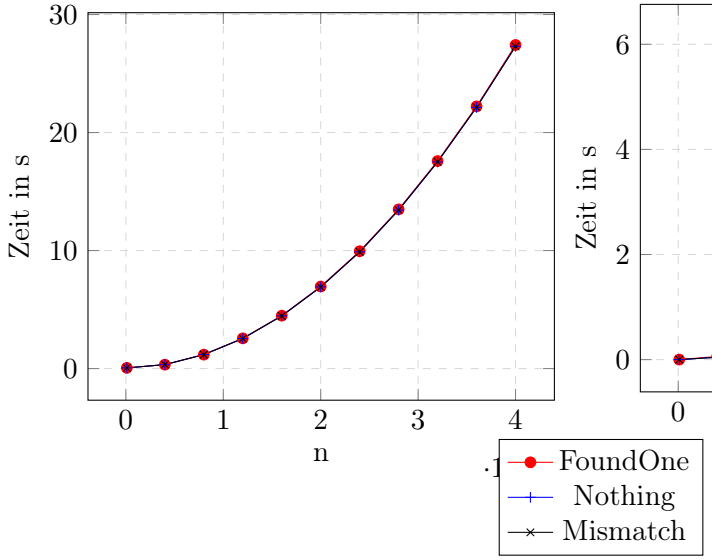

Abbildung 2: p = 44

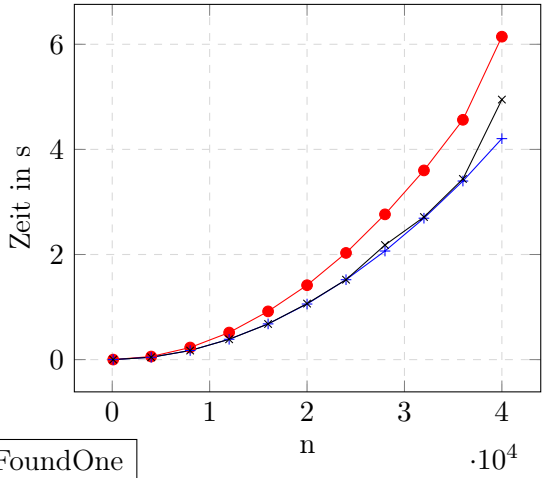

Abbildung 3: p = 1

wird. Außerdem ist zu beobachten, dass in Abbildung 3 `Nothing` die Untergrenze darstellt und nicht `Mismatch`. Dies ist damit begründet, dass der `Mismatch` in den letzten Pixeln liegt[2] und nicht eher abgebrochen werden kann, sondern immer alle $n * n$ Felder überprüft werden müssen. Neben dem bloßen Erkennen, ob es sich um weiße oder schwarze Pixel handelt, muss bei einem `Mismatch` wenigstens ein ausschließendes Muster von Abbildung 1 überprüft worden sein. Dieser Mehraufwand ist der Grund weshalb `Mismatch` ein schlechteres Laufzeitverhalten als `Nothing` in Abbildung 3 aufweist. Ferner liefert `FoundOne` den Worst-Case für beide Messungen, da in beiden Fällen die ausschließenden Muster geprüft werden müssen. Durch die folgendenden Darstellungen (Abbildung 4 und Abbildung 5) wird das Laufzeitverhalten für ein festes n ersichtlich.

Aus den Darstellungen kann man entnehmen, dass sich das Laufzeitverhalten für steigende p logarithmisch verschlechtert. Das heißt für kleine p ist der Anstieg der Laufzeit stets hoch, für große p nimmt dieser Anstieg wieder ab bis die Laufzeit in Abhängigkeit von p einer konstanten Funktion gleicht. Außerdem kann man erkennen, dass der sequentielle Fall wieder die besten Laufzeiten besitzt. Betrachtet man die unterschiedlichen Ergebnisse bildet `FoundOne` wieder den Worst-Case. Der Best-Case für $p = 1$ kann diesmal durch `Mismatch` gezeigt werden, da ein vorzeitigen Abbruch durch das Vorkommen eines `Mismatches` in Zeile 4000 erreicht wird. Im parallelen Fall kann nicht eher abgebrochen werden, da erst alle Teilbilder durchsucht werden müssen, so dass ab einer Prozessorzahl

[2]Für die Messungen wird ein `Mismatch` am Ende einer Zeile durch 010 erzeugt.

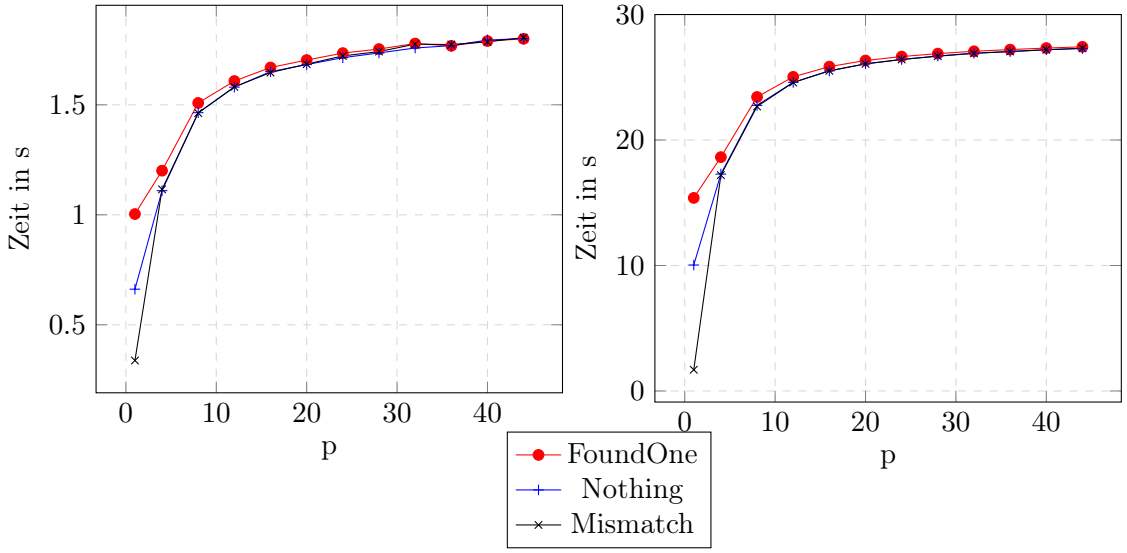

Abbildung 4: n = 10000

Abbildung 5: n = 40000

von 2 `Nothing` und `Mismatch` das gleiche Laufzeitverhalten besitzen.

2.4 Speedup und Effizienz

Die folgenden Abbildungen beziehen sich auf die Auswertungen zum Speedup und zur Effizienz.

Wie schon bei der Untersuchung des Laufzeitverhaltens bemerkt wurde, verschlechtert sich die Laufzeit für den parallelen Fall enorm. Dieses Verhalten wird durch Abbildung 7 und Abbildung 6 bestätigt. Anstatt einen linearen Anstieg anzustreben, ähnelt die Funktion einer Hyperbel mit indirekter Proportionalität für alle Messungen. Mit steigenden n müssen größere Teilbilder an alle Prozessoren kommuniziert werden, deshalb ist der Anstieg für $n = 40000$ flacher als für $n = 10000$, da der Kommunikationsoverhead stärker ins Gewicht fällt als die eigentliche Überprüfung. Außerdem erkennt man aus den Speedup-Darstellungen, die Abhängigkeit zu der sequentiellen Laufzeit. So dass der eigentliche Best-Case nun den schlechtesten Speedup aufweist, da der Referenzwert im sequentiellen Fall wesentlich geringer ist als die anderen Referenzwerte. Entsprechend verhält sich der Average-Case und Worst-Case. Folglich konnte gezeigt werden, dass die Problemstellung nicht geeignet ist für die Parallelisierung durch die anfallende Kommunikation.

Im Anschluss wird näher auf die Effizienz eingegangen.

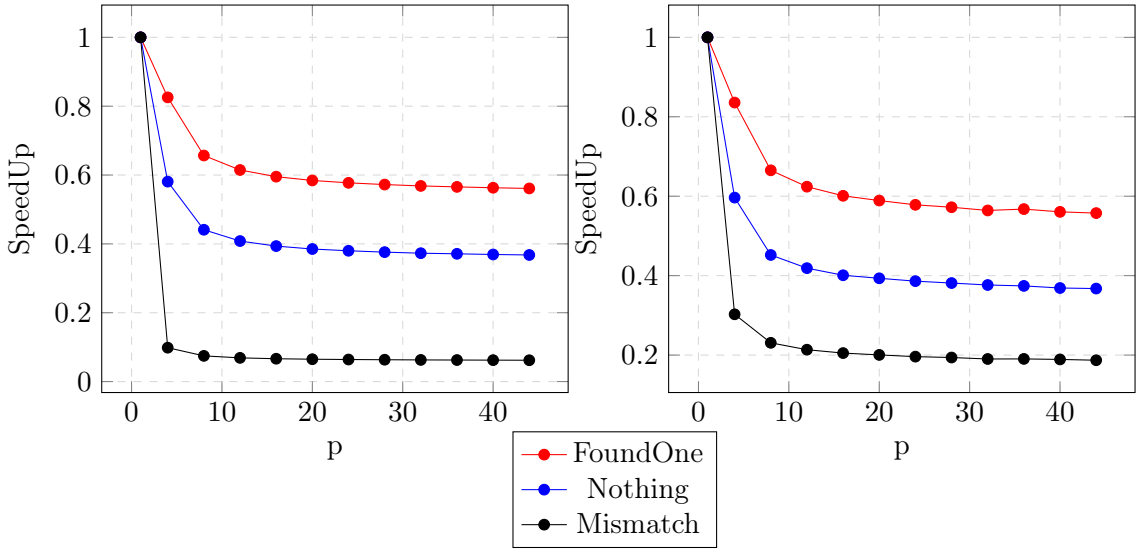

Abbildung 6: SpeedUp: n = 40000

Abbildung 7: SpeedUp: n = 10000

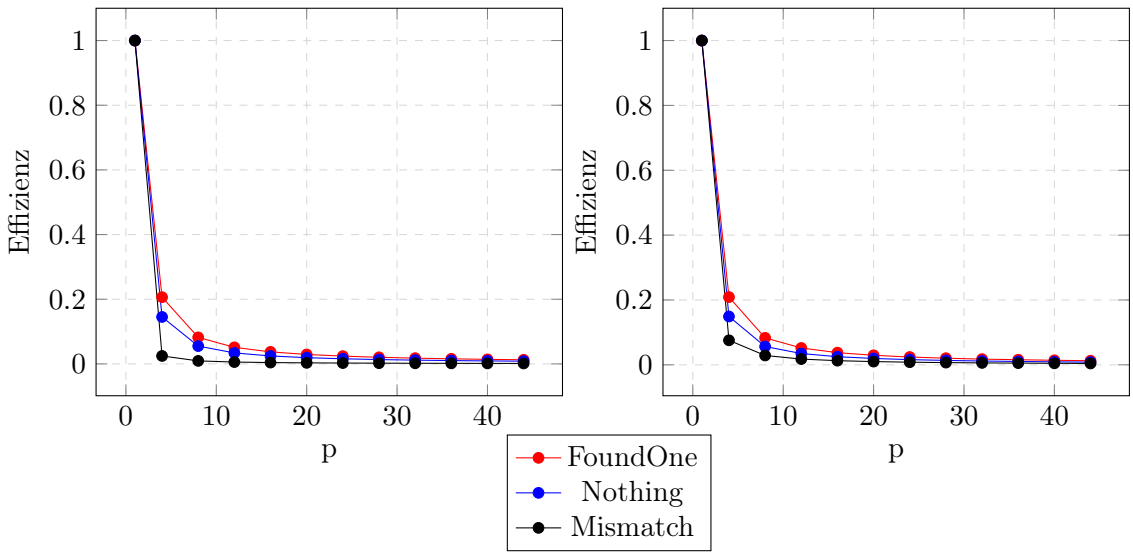

Abbildung 8: Effizienz: n = 40000

Abbildung 9: Effizienz: n = 10000

Ähnlich zum Speedup weichen die Werte der Effizienz enorm zu einem normal parallelisierten Programm ab. Somit kann nur für den sequentiellen Fall gezeigt werden, dass die Effizienz bei den Optimalwert 1 liegt, wodurch niemals eine Verbesserung durch eine Parallelisierung für diese Problemstellung stattfindet. Außerdem erkennt man wie beim Speedup für größere n (Abbildung 8) einen stärkeren Abfall als für kleinere n. Dies ist wieder durch den stärker ins Gewicht fallenden Kommunikationsoverhead zu begrün-

den.

2.5 Fazit

Die Problemstellung eine Rechteckmustererkennung umzusetzen und zu parallelisieren konnte erfolgreich, wie in Unterabschnitt 2.1 beschrieben, umgesetzt werden. Laufzeitmessungen ergaben, dass eine sequentielle Erkennung einer parallelen vorzuziehen ist. Begründet ist diese starke Verschlechterung durch den anfallenden Kommunikationsoverhead und der zusätzlich notwendigen Auswertung der Teilergebnisse. Außerdem gibt es für die sequentielle Suche die Möglichkeit vorzeitig abzubrechen. Anders als im parallelen Fall, da jeder Prozess sein Teilbild entsprechend durchsuchen muss. Durch die Untersuchungen von variierenden n konnte gezeigt werden, dass sich die Laufzeit quadratisch mit steigenden n verschlechtert für alle p. Für die parallele Umsetzung verschlechtert sich die ohnehin schon schlechte Laufzeit noch mehr durch den zusätzlich steigenden Kommunikationsaufwand. Da sehr geringe Werte für Speedup und Effizienz ermittelt worden, wurde bewiesen dass die Problemstellung sich nicht für die Parallelisierung eignet. Folglich wird für die Erkennung von Rechteckmustern die sequentielle Variante empfohlen.

3 Projektaufgabe 2: Numerische Integration mittels Parabelformel

In diesem Abschnitt wird auf die Projektaufgabe 2 eingegangen. Zunächst wird kurz erklärt wie das Programm kompiliert und gestartet wird sowie auf die Kommandozeilenargumente eingegangen. Im Anschluss wird die Realisierung und damit die Lösung des Problems betrachtet. Anschließend erfolgt die Auswertung und im letzten Abschnitt wird ein Fazit gezogen.

3.1 Verwendung

Der Quellcode kann mit dem Aufruf von 1 auf dem MC-3-System kompiliert und mit 2 ausgeführt werden:

1. f77.px -o A2star.px A2star.f
2. run -f0 4 2 A2star.px **FUNCTION N**

Für das Programm können Kommandozeilenargumente zur Auswahl der Testfunktion und die Größe der zu berechnenden Teilstücke des Integrals angegeben werden.

FUNCTION: Das Programmargument **FUNCTION** entscheidet darüber, welche der beiden gegebenen Testfunktionen verwendet wird. Als gültige Eingabe wird 1 für Funktion 1 und 2 für Funktion 2 erwartet.

N: Das zweite Programmargument **N** gibt die 2er Potenz der zu berechnenden Teilstücke des Integrals an. Je größer das **N**, umso genauer wird das berechnete PI. De höchstmöglichste zulässige Eingabe ist 20, das sind 1048576 zu berechnende Teilstücke. Diese Vorgehensweise soll zum Einen sicher stellen, dass **N** stets durch 2 Teilbar ist und zum Anderen das Laufzeitmessen vereinfachen, da ein exponentielles Wachstum vorliegt. Sollte kein Programmargument angeben werden, so wird intern die Funktion 1 mit $n = 2^{15} = 32768$ verwendet.

Weiterhin muss die Divison von n durch Prozessorzahl ganzahlig sein, damit stets gewährleistet ist, dass die zu berechnende Anzahl an Teilstücke pro Prozessor eine ganzzahlige ist.

3.2 Realisierung

Grundlegend wurde sich bei der Realisierung für eine Stern-Topologie entschieden. Gewählt wurde diese Topologie, da bei der Parallelisierung dieser Aufgabe ein Master existert, welche alle Teilergebnisse empfängt und summiert. Weiterhin wurde diese Topologie bereits in einem Seminar implementiert.

Bedingt durch die Realisierung als Kommandozeilenparameter, entfällt der Kommunikationsoverhead für das Verteilen von **N** und **FUNCTION**. Daher kennen alle Prozessoren **N** sowie die Testfunktion beim Programmstart.

Die Prozessoren können explizit ihre eigenen Bereich für die Teilstücke bestimmen und somit die numerische Berechnung durchführen. Nach Abschluss der Berechnungen, werden diese an den Master-Prozessor gesendet. Der Master-Prozessor summiert die Ergebnisse der anderen Prozessoren zu seinem eigenem Ergebnis auf und wendet die Multiplikation $\frac{\mathbf{h}}{3}$ auf das Ergebnis an. Im Anschluss werden die Ergebnisse wie Referenzwert-PI, Berechnetes-PI, Abweichung und Laufzeit (nicht in Abbildung dargestellt) ausgegeben. Die Abbildung 10 zeigt dies.

```
call startIntegration(f,n,integral,h)
if(id.eq.0) then
 summe = integral
 do i=1,(np-1)
  call recv(topid,links(i),integral,8)
  summe = summe + h/3 * integral
 enddo
else
 call send(topid, link,integral,8)
endif
```

Abbildung 10: Empfangen und Auswerten

```
nL = n / nprocs()
aL = a + id * nL * h
bL = aL + nL * h
summe = getVal(f,aL) + getVal(f,bL)
step = aL + h
do i=1,nL-1
 factor=2**(mod(i,2) + 1)
 summe = summe + (dble(getVal(f,step) *
       factor)
 step = step + h
enddo
```

Abbildung 11: Berechnung der Teilstück

Die numerische Berechnung des Integrals wird durch jeden Prozessor ausgeführt, aber jeder Prozessor führt nur einen Teil der kompletten Berechnung aus. Die eigentliche Berechnung ist in der Abbildung 11 zu sehen. Zunächst berechnet jeder Prozessor die Anzahl der durchzuführenden Berechnungen, als lokales **n** (**nL**). Im Anschluss werden Startwert (**aL**) und Endwert (**bL**) ermittelt. Die Funktionswerte von **aL** und **bL** der Testfunktion werden addiert und in einer Summe gespeichert. Im Anschluss werden die Zwischenstücke mit entsprechendem Faktor berechnet und ebenfalls summiert. Die Variable **h** gibt dabei die allgemeine Schrittweite an und **step** den derzeitigen Schrittwert.

Nach erfolgter Berechnung sendet jeder Prozessor sein Ergebnis an den Master (falls der Prozessor nicht selbst der Master ist)

3.3 Ergebnisse der Testfunktionen

In diesem Unterabschnitt wird auf die erzielten Ergebnisse der Testfunktion 1 und 2 hinsichtlich ihrer Näherung zu Pi eingegangen. Als Referenzwert für die Testunfktionen wurde Pi mit **3.141592653589793** verwendet. Angemerkt sei, dass die Prozessoranzahl 6 ausgeschlossen wurde, da dadurch bei Division durch die 2er Potenz von **n**, eine nicht ganzzahliges lokales **n** enstehen würde. Durch das lokale **n**, würde zwar ein Näherungswert zu Pi bestimmt, aber dieses besitzt eine deutlich größere Abweichungen im Vergleich zu einer korrekten Eingabe. Deswegen wird beim Programmstart diese Eingabe überprüft und ggf. bei falscher Eingabe das Programm beendet.

Die Abbildung 12 zeigt die Abweichung zum Referenzwert von Pi bei einer konstanten Prozessoranzahl von 8 und einem sich verändernden **n**. Wie in der Tabelle zu sehen ist liefert die Testfunktion 2 bessere Näherungswerte als Funktion 1. Ein Grund dafür, dass die Funktion 1 schlechte Näherungswerte liefert, liegt unter Anderem an der Oberegrenze. Diese Obergrenze ist Pi selbst. Zusätzlich sind in der Funktion 1 die Schrittweiten bedeutend größer, als bei der Funktion 2, daher benötigt die Funktion ein größeres **n** um mehr Teilstücke berechnen zu können und somit ein genaueres Ergebnis zu liefern. Die Abbildung 13 zeigt ebenfalls die Abweichung, aber hier wird ein konstantes **n** von

n	Abweichung-F1	Abweichung-F2
1.024	$-1{,}57 \cdot 10^{-12}$	0
512	$-2{,}47 \cdot 10^{-11}$	$-4{,}44 \cdot 10^{-16}$
256	$-3{,}96 \cdot 10^{-10}$	$-4{,}44 \cdot 10^{-16}$
128	$-6{,}33 \cdot 10^{-9}$	$8{,}88 \cdot 10^{-15}$

Abbildung 12: Mit 8 Prozessoren

p	Abweichung-F1	Abweichung-F2
8	$-6.84 \cdot 10^{-13}$	$-1.78 \cdot 10^{-15}$
4	$1.34 \cdot 10^{-12}$	$-6.22 \cdot 10^{-15}$
2	$-1.73 \cdot 10^{-12}$	0
1	$-1.43 \cdot 10^{-12}$	$1.07 \cdot 10^{-14}$

Abbildung 13: Mit n = 32768

32768 verwendet und die Prozessoranzahl variiert. Das die Näherungswerte sich in Abhängigkeit von der verwendeten Prozessorzahl unterscheiden ist damit begründet, dass die Parabelformel mittlere Teilstücke mit einem Faktor von 4 beziehungsweise 2 wichtet.

3.4 Laufzeitverhalten

Im vorherigen Abschnitt wurden die Ergebnisse der Testfunktionen hinsichtlich Pi untersucht und nun wird eine Betrachtung hinsichtlich der Laufzeit durchgeführt.

Wie in der Abbildung 14 zu sehen ist, lohnt sich für diese Aufgabenstellung eine Parallelisierung. Mit steigendem **p** verringert sich die Laufzeit im Vergleich zum sequenziellen Fall. Es ist zu vermuten das mit steigendem **p** der Laufzeitgewinn immer schwächer wird und ab einem gewissen Punkt wieder eine Laufzeitverschlechterung eintritt. Begründet ist dies mit steigendem Kommukationsoverhead bei einer steigendem Prozessoranzahl. Bei der hier verwendeten geringen Anzahl an Prozessoren mit einem Maximum von 8, spielt der Kommunikationsoverhead eine eher untergeordnete Rolle. Zusätzlich ist zu sehen, dass die Testfunktion 1 langsamer ist als die Testfunktion 2 ist. Es wird vermutet, dass die Berechnung von Sinus in der Testfunktion 1 äufwändiger ist als die Division in der Testfunktion 2.

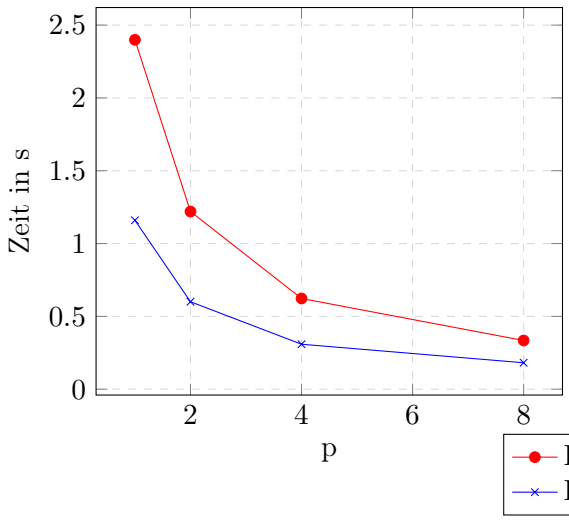

Abbildung 14: Laufzeit mit n = 32768

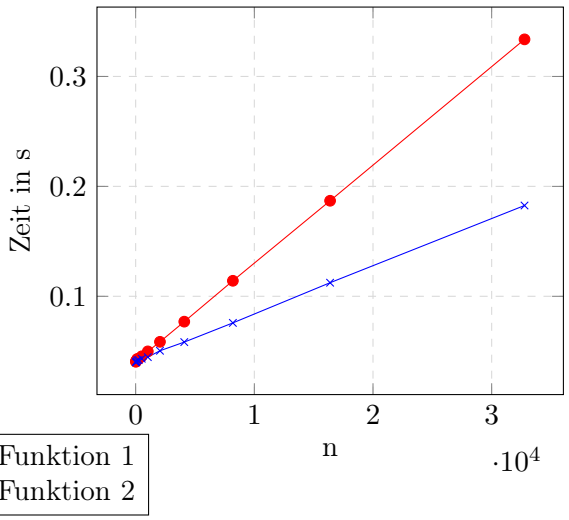

Abbildung 15: Laufzeit mit p = 8

Wie in der Abbildung 15 zu sehen ist steigt mit wachsendem **n** und konstantem **p** die Laufzeit beider Algorithmen an. Es ist festzustellen, dass die Testfunktion 1 dabei einen stärkeren Anstieg aufweißt als Testfunktion 2. Die Ergebnisse entsprechen dem erwarteten Verhalten, denn mit steigendem **n** sollte bei diesem Problem die Laufzeit linear steigen.

3.5 Speedup und Effizienz

In diesem Abschnitt sollen nun Speedup und Effizienz für die Testfunktionen 1 und 2 betrachtet werden.

In der Abbildung 16 ist der ermittelte Speedup für die Tesfunktion 1 mit zwei verschieden Werten für **n** zu sehen. Bei einem großem **n** ist zu sehen, dass ein deutlich beserer Speedup erzielt wird, als bei einem kleineren **n**. Dieses Verhalten wird auch in der Funktion 2 nachgewiesen, wie man in der Abbildung 17 sehen kann. Bei einem kleinen **n** flacht der Speedup bedeutend schneller ab. Bedingt ist dies dadurch, dass bei einem kleineren **n** die Laufzeit für die Kommunikation und Summieren der Ergebnisse markanter in die Berechnung eingeht. Dennoch lässt sich sagen, dass bei dieser Aufgabe bei maximal 8 Prozessoren ein sehr guter Speedup erreicht werden kann. Dies gilt sowohl für Funktion 1 als auch Funktion 2, wobei Funktion 2 einen etwas schlechteren Speedup als Funktion 1 erreicht.

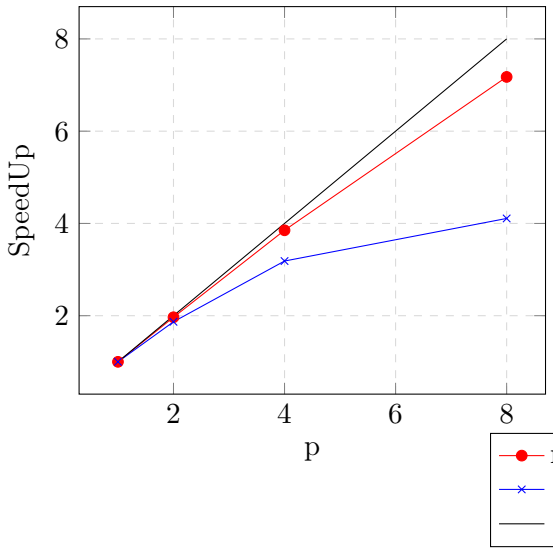

Abbildung 16: SpeedUp Funktion 1

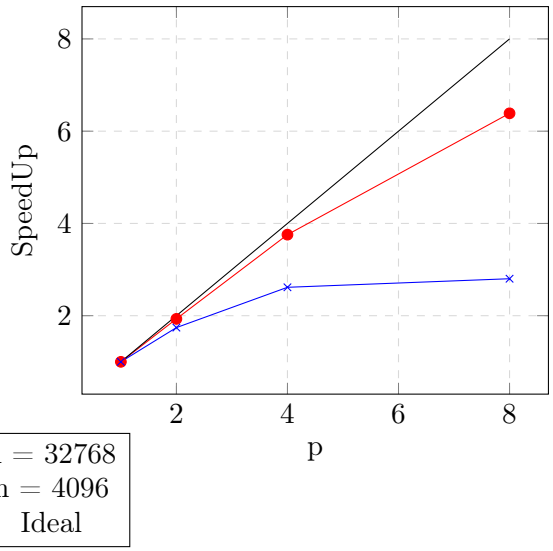

Abbildung 17: SpeedUp Funktion 2

Nun betrachten wir die Funktionen hinsichtlich ihrer Effizienz. In der Abbildung 18 und Abbildung 19 ist die Effizienz für die Funktion 1 sowie 2 dargestellt. Aufgrund das bei einem großen **n** ein besserer Speedup erzielt wird, werden bei Funktion 1 und 2 relativ hohe Effizienzwerte erreicht. Es ist außerdem zu sehen, dass Funktion 1 im Allgemeinen effizienter ist als Funktion 2. Außerdem wird die höchste Effizienz bei der Parallelisierung mit 2 Prozessoren erreicht. Dies gilt für Funktion 1 und Funktion 2 gleichermaßen. Bei

einem kleineren **n** sinkt die Effizienz bedeutend schneller als bei einem großen, wobei auch hier Funktion 1 im Allgemeinen besser ist.

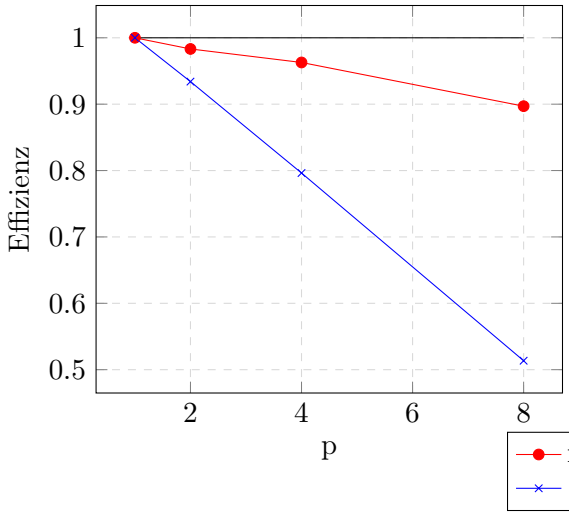

Abbildung 18: Effizienz Funktion 1

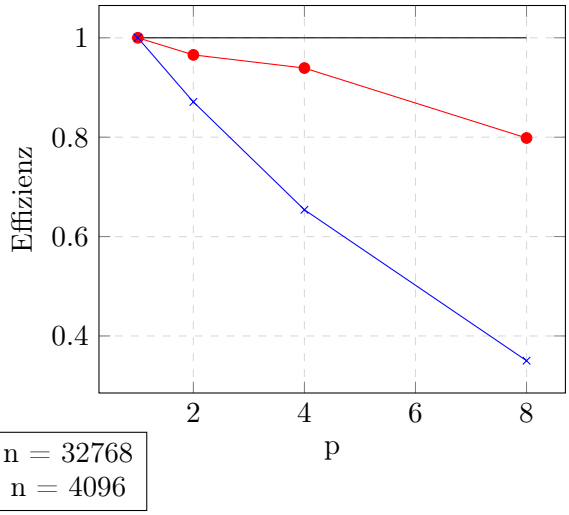

Abbildung 19: Effizienz Funktion 2

3.6 Fazit

Die Problemstellung der numerischen Integration mittels Parabelformel konnte erfolgreich in Fortran umgesetzt werden. Laufzeitmessungen sowie Betrachtung von Speedup und Effizienz ergaben, dass eine Parallelisierung für diese Aufgabenstellung sehr geeignet ist. Aufgrund des geringen Kommunikationsoverhead fällt dieser bei der geringen Prozessoranzahl ($\mathbf{p} \leq 8$) kaum ins Gewicht, falls das **n** ausreichend groß ist. Bei einem kleinen **n** ist dieser Kommunikationsoverhead mit anschließender Summation doch zu spüren wie die Betrachtung des Speedups ergab.

Zusätzlich ist festzuhalen, dass die Funktion 2 bessere Näherungswerte für Pi liefert als die Funktion 1. Obwohl die Funktion 2 ein besseres Laufzeitverhalten als Funktion 1 besitzt, ist Funktion 1 doch etwas effizienter bei der Parallelisierung.