Dokumentation zu den Projektaufgaben 1 und 2

Message Passing Programmierung

Matthias Zober und Michael Horn

22. Januar 2017

Inhaltsverzeichnis

1 Einleitung

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.

2 Projektaufgabe 1: Rechteckmustererkennung

Das zu spezifizierende parallele Programm soll eine Rechteckmustererkennung über ein Cluster realisieren.

2.1 Realisierung

Für die Umsetzung eines solchen Programms, ist es zunächst notwendig eine sequentielle Erkennung eines $n * m$ Digitalbild zu realisieren. Die sequentielle Erkennung ist der Teil des Programms der sich nicht parallelisieren lässt. Für die Parallelisierung muss jeder Prozessor diese Erkennung auf sein Teilbild anwenden.

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

[1] Letzter Aufruf: 22. Januar 2017

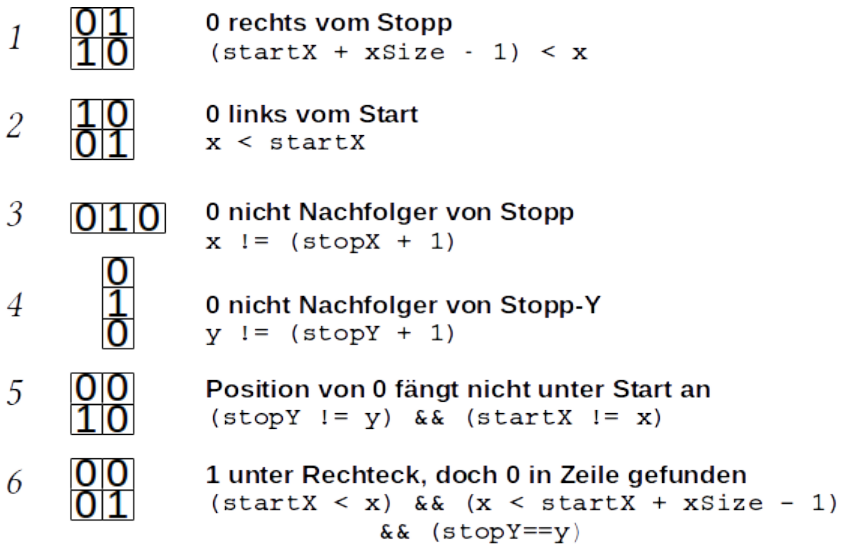


Abbildung 1: Ausschließende Bit-Muster mit Regeln

Werden diese Muster nicht erkannt, aber schwarze Pixel, 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 ein ausgezeichneter Hauptprozess (`Master`) alle Ergebnisse zusammenfassen, dies geschieht in den 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 auch abgebrochen werden mit `Nothing`.
3. Gibt es nur ein `FoundOne` und sonst `Nothing`, kann das Rechteck ausgegeben werden.

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

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

2. Bis auf dem 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 letzten Überprüfung kann der Master-Prozess das Ergebnis ermitteln und die Start- & 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, da direkt abgebrochen werden kann. Der Average-Case stellt ein Digitalbild ohne schwarze Pixel dar, da nach überprüfen aller Pixel abgebrochen werden kann, aber keine weiteren Überprufungen stattfinden. Der Worst-Case besteht darin mehrere Rechtecke zu finden, da alle Rechtecke zusammenhängend sein 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 Einzen.

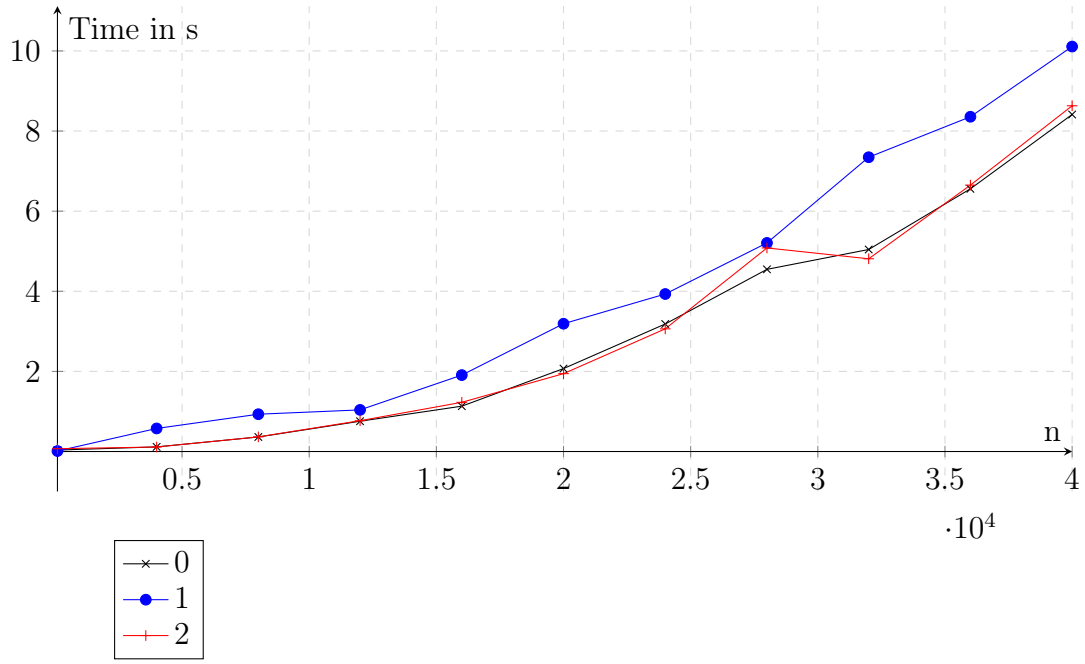
Modus 1: Single Fields & Modus 0: Rectangle Um eine komfortable Eingabe zu realisieren, wird für die Modi 1 und 0 nicht die direkte Angabe der Matrix sondern nur die angabe einzelne Parameter realisiert, aus denen dann das Digitalbild im Programm generiert wird. Die zweite Zeile beschreibt die Größe 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 werden müssen. Modus 0 hingegen interpretiert nur die ersten drei Koordinatenpaare als die Ortsvektoren der oberen Rechteck-Ecke, die das Rechteck aufspannt. Es ist notwendig, dass dabei die Reihenfolge: obere Ecke links, obere Ecke rechts und untere Ecke links eingehalten wird.

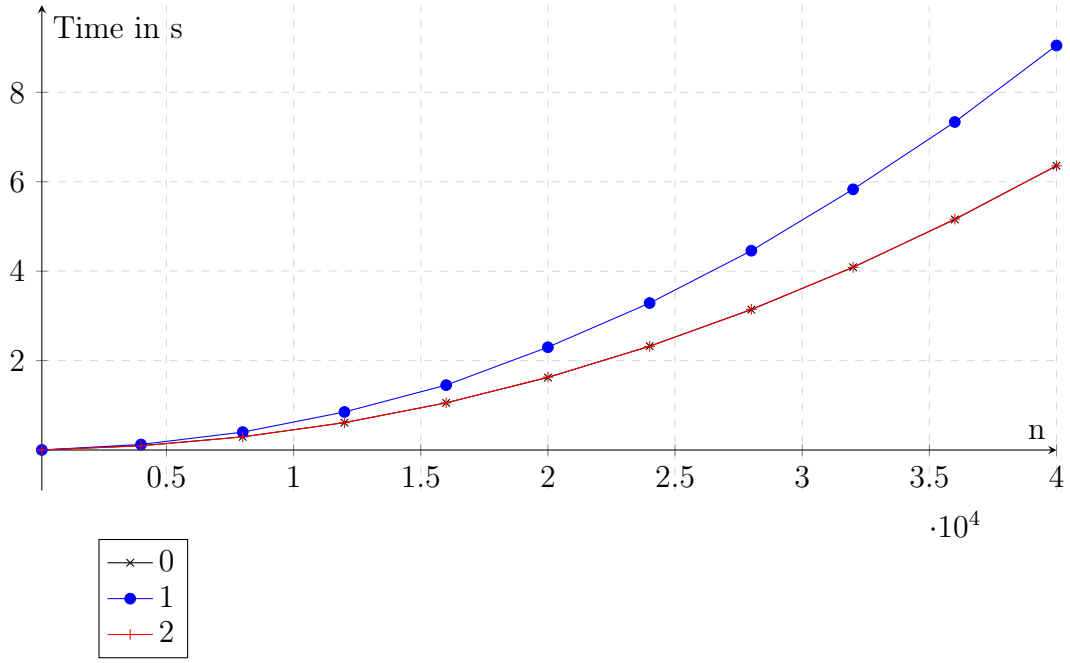
Durch die angegebenen Kodierungen (Modus 1 und 0) ist es möglich sehr große Digitalbilder mit der Anwendung zu testen, ohne das diese tatsächlich als physisch gespeichterte Dateien existieren müssen. Im Gegensatz dazu kann durch Modus 2 direkt die Matrix eingegeben werden, dieser Modus findet vorallem 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

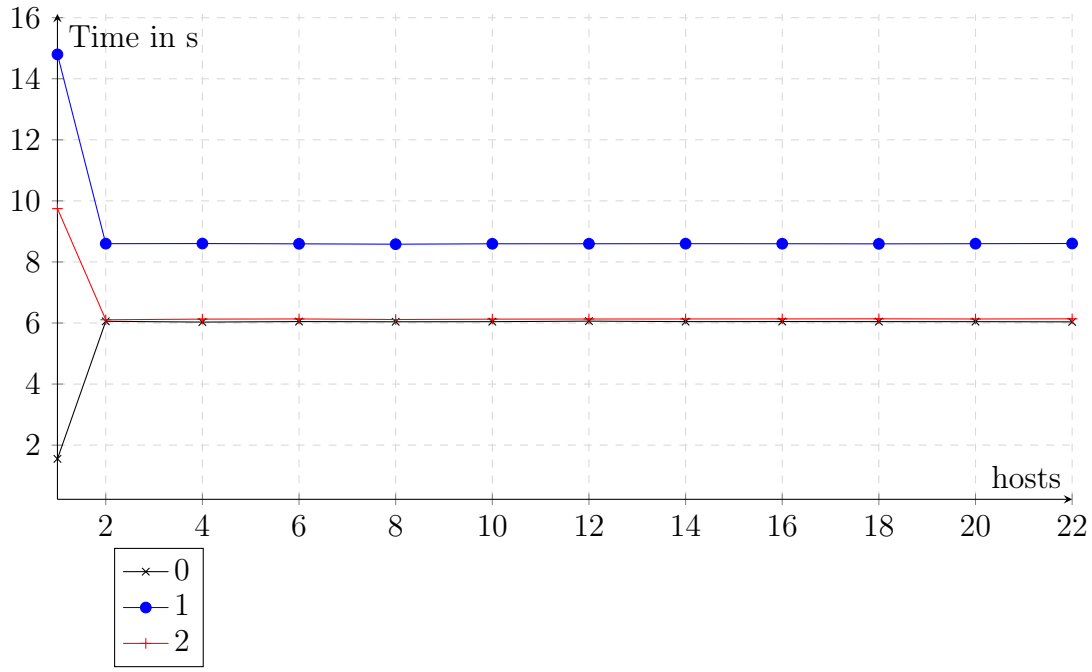
Wie man aus dem gegebenen Laufzeitmessungen entnehmen kann, liefert der Best-Case `Mismatch` wesentlich kürzere Laufzeiten für den sequentiellen Fall, als für den parallelen. Dies ist damit begründet, dass kein Kommunikationsoverhead für die Auswertung existiert, da der `Master` direkt seine Suche abbrechen kann, während im parallele Programm der `Master` auf alle Teilergebnisse warten muss.

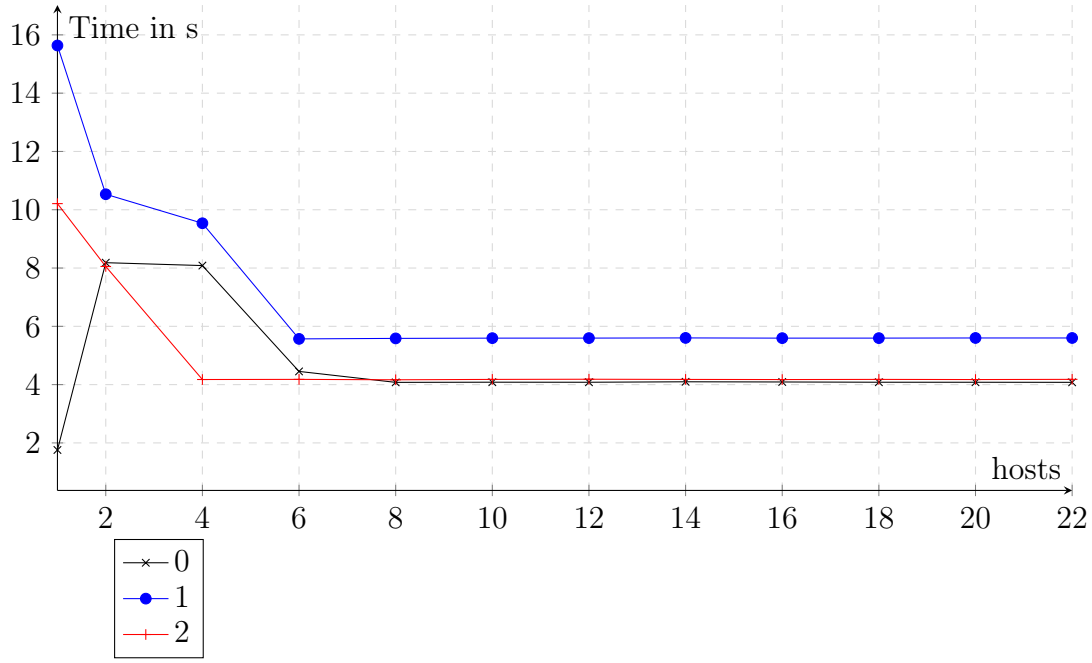
2.3.1 Festes n




Time in s
8
6
4
2
n
0.5
1
1.5
2
2.5
3
3.5
4
$\cdot 10^4$
0
1
2

2.3.2 Festes p


16
14
12
10
8
6
4
2
Time in s
hosts
2
4
6
8
10
12
14
16
18
20
22
0
1
2



2.4 Fazit

3 Projektaufgabe 2: Numerische Integration mittels Parabelformel

In diesem Abschnitt wird auf die Projektaufgabe 2 eingegangen. Zunächst wird im Unterabschnitt 3.1 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 im Unterabschnitt 3.2 betrachtet. Anschließend erfolgt im Unterabschnitt 3.3 die Auswertung der Realisierung. Im letzten Unterabschnitt 3.4 wird ein Fazit zu der Aufgabe gezogen.

3.1 Verwendung

Der Quellcode kann mit Hilfe folgendes Aufrufes auf dem MC-3-System kompiliert werden:

- f77.px -o A2star.px A2star.f

Mit folgendem Befehl wird das Programm ausgeführt:

- run -f0 4 2 A2star.px **funktion n**

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

funktion: Das Programmargument **funktion** 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 wird eine ungerade Prozessorzahl ausgeschlossen, damit stets gewährleistet ist, dass die zu berechnende Anzahl an Teilstücke pro Prozessor eine ganzzahlige Zahl 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. Wie die Topologie erstellt wird zeigt Listing 1.

LISTING 1: Erzeugen der Stern-Topologie

```
if(id.eq.0) then
        do i=1,(np-1)
                links(i)=addnewlink(topid,i,1)
        enddo
else
        link=addnewlink(topid,0,1)
endif
```

Bedingt durch die Realisierung als Kommandozeilenparameter, entfällt der Kommunikationsoverhead für das Verteilen von **n** und **funktion**. 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 Berech-

nungen, 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 h/3 auf das Gesamtergebnis an. Im Anschluss werden die Ergebnisse wie Referenzwert-PI, Berechnetes-PI, Abweichung und Laufzeit (nicht im Listng dargestellt) ausgegeben. Das folgende Listing 2 zeigt dies.

Listing 2: Empfangen und Auswerten

```
c       starte die Integration
        call startIntegration(f,n,integral,h)
        if(id.eq.0) then
                print*,"use f:",f," with n:",n
c       Integralteilstück vom Master
                summe = integral
                do i=1,(np-1)
c       Empfange alle Teilstücke und addiere Sie
                        call recv(topid,links(i),integral,8)
                        summe = summe + integral
                enddo
                summe = h/3 *summe
                derivation =  PI() - summe
                print*,"------------------------------"
                print*,"PI Referece:",PI()
                print*,"PI Integral:",summe
                print*,"PI Derivat.:",derivation
        else
c       Sende Integralteilstücke zum Master
                call send(topid, link,integral,8)
        endif
```

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 dem Listing 3 zu sehen. Zunächst berechnet jeder Prozessor die Anzahl der durchzuführenden Berechnungen, als lokales **n** (**nL**). Im Anschluss werden Startpunkt (**aL**) und Endpunkt (**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** die derzeitigen Schrittwert.

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

LISTING 3: Berechnen der relevanten Bereiche für jeden Prozessor

```
      id = myprocid()
      nL = n / nprocs()
      aL = a + id * nL * h
      bL = aL + nL * h
c     Randwerte, getVal liefert Funktionswert der Testfunktion
      summe = getVal(f,aL) + getVal(f,bL)
      step = aL + h
c     Berechnung zwischenstücke, factor alterniert mit 2 und 4
      do i=1,nL-1
            exponent = mod(i,2) + 1
            factor=2**exponent
            summe = summe + dble(getVal(f,step) * factor)
            step = step + h
      enddo
```

3.3 Ergebnisse

3.3.1 Testfunktionen

Für die Testfunktionen wird folgender Referenzwert verwendet: 3.141592653589793. Die Abbildung 2 zeigt die Abweichung zum Referenzwert von Pi bei einer konstanten Prozessoranzahl von 8 und einem sich verändernden **n**. Die Abbildung 3 zeigt ebenfalls die Abweichung, aber hier wird ein konstantes **n** von 32768 verwendet und die Prozessoranzahl variiert.

Wie in den Tabellen zu sehen ist liefert die Testfunktion 2 schneller bessere Pi-Näherungswerte als Funktion 1. Ein Grund dafür das 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.

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	$-6.33 \cdot 10^{-9}$	$8.88 \cdot 10^{-15}$
64	$-1.01 \cdot 10^{-7}$	$5.77 \cdot 10^{-13}$

Abbildung 2: Mit 8 Prozessoren

p	Abweichung-F1	Abweichung-F2
8	$-6.84 \cdot 10^{-13}$	$-1.78 \cdot 10^{-15}$
6	$1.87 \cdot 10^{-4}$	$3.03 \cdot 10^{-4}$
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 3: Mit n = 32768

3.3.2 Laufzeitverhalten

Festes n

Festes p

3.4 Fazit

Kommunikationsoverhead spielt hier eine untergeordnete Rolle, da Minimum an Kommunikation zur Berechnung notwendig ist. Weiterhin lohnt sich die Parallelisierung,..