

Dokumentation zu den Projektaufgaben 1 und 2

Message Passing Programmierung

Matthias Zober und Michael Horn

21. Januar 2017

Inhaltsverzeichnis

1	Einleitung	3
2	Projektaufgabe 1: Rechteckmustererkennung	3
2.1	Realisierung	3
2.2	Verwendung	5
2.2.1	Ausführung	5
2.2.2	Konfigurationsfile	5
2.3	Laufzeitverhalten	6
2.3.1	Festes n	6
2.3.2	Festes p	7
2.4	Fazit	8
3	Projektaufgabe 2: Numerische Integration mittels Parabelformel	9
3.1	Verwendung	9
3.2	Realisierung	10

3.3	Ergebnisse	12
3.3.1	Testfunktionen	12
3.3.2	Laufzeitverhalten	12
3.4	Fazit	12

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 das finden eines schwarzen Pixels und das Überprüfen von ausschließenden Mustern. Die Erkennung kann einfach realisiert werden durch die Speicherung der Koordinaten des ersten und zuletzt gefundenen Pixels, die Länge der oberen gefunden Rechteckkante und die Information ob eine Zeile mit schwarze Pixeln von weißen Zeilen umschlossen wurde. Im folgenden erkennt man die ausschließenden Mustern mit deren Regeln.

¹Letzter Aufruf: 21. Januar 2017

<div>01 10</div>	X(0) rechts vom Stopp (startX + xSize - 1) < x	<div>00 01</div>	X(1) unter Rechteck, doch Y(0) in Zeile gefunden (startX < x) && (x < startX + xSize - 1) && (stopY==y)
<div>10 01</div>	X(0) links vom Start x < startX	<div>00 10</div>	Position von 0 fängt nicht unter Start an (stopY != y) && (startX != x)
<div>0 1 0</div>	Y(0) nicht Nachfolger von Stopp-Y y != (stopY + 1)	<div>010</div>	X(0) nicht Nachfolger von Stopp x != (stopX + 1)

Werden diese Muster nicht erkannt, aber schwarze Pixel erkannt, 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 ausgezeichnete Hauptprozess (**Master**) alle Ergebnisse zusammenfassen, dies geschieht 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 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 Umsetzung 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 Überprüfungen stattfinden. Der Worst-Case besteht darin mehrere Rechtecke zu finden, da alle Rechtecke zusammenhängend sein müssen. Für die Untersuchungen zur Laufzeit werden, stets diese drei Fälle berücksichtigt.

2.2 Verwendung

2.2.1 Ausführung

Der Quellcode kann mit Hilfe des folgenden Aufrufs auf dem PC oder dem Cluster kompiliert werden:

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

Mit folgenden Befehl wird das Programm auf 4 Prozessoren ausgeführt:

- `mpirun -np 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.

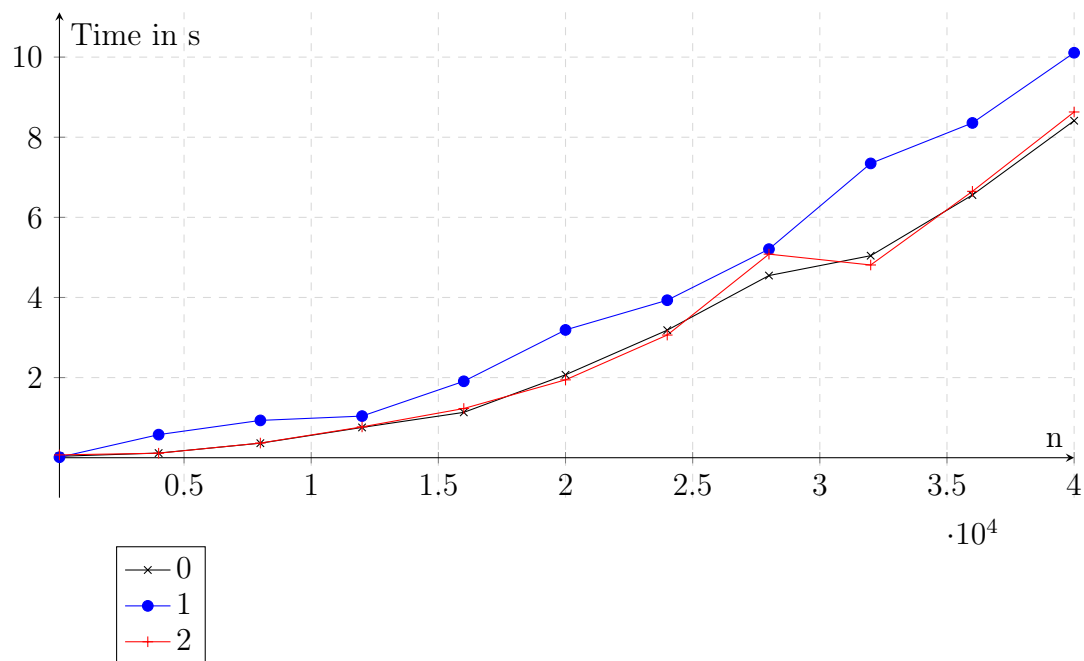
Modi 2 - Custom Mode Der Custom Mode ermöglicht die direkte angabe einer $n \times n$ Matrix aus Nullen und Einzen.

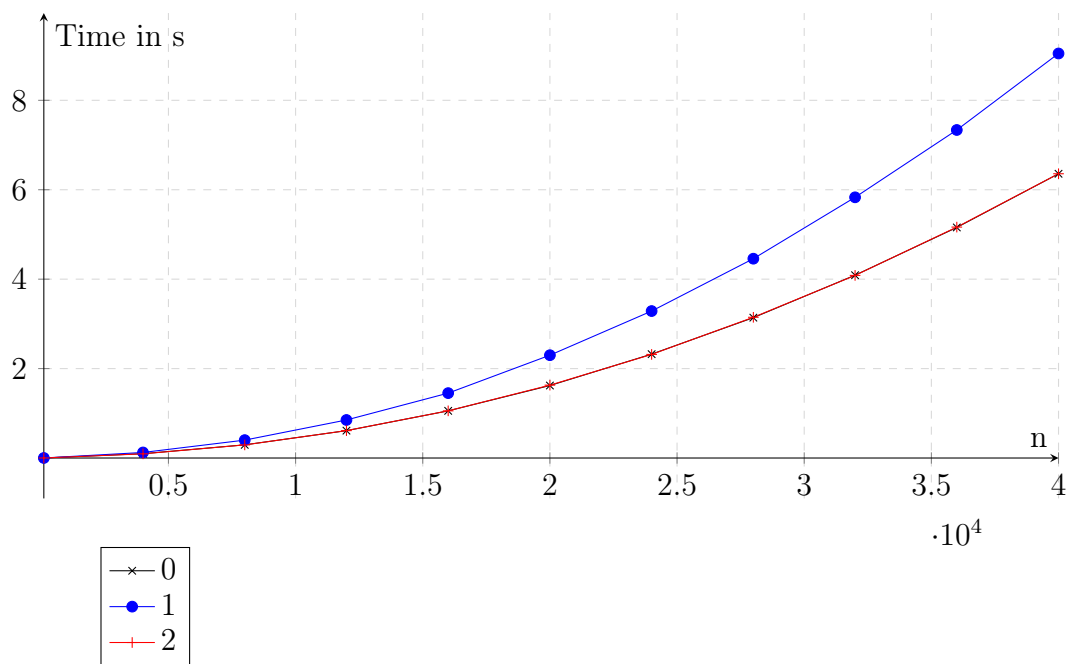
TODO: Beschreibe dass Modi 0 und 1 die Eingabe mit der Angabe von n und der Hintergrundfarbe gemeinsam verwenden

TODO: beschreibe die Erzeugung von Rechtecken mittels Mode 0 und die Erzeugung von einzelnen Punkten durch Mode 1

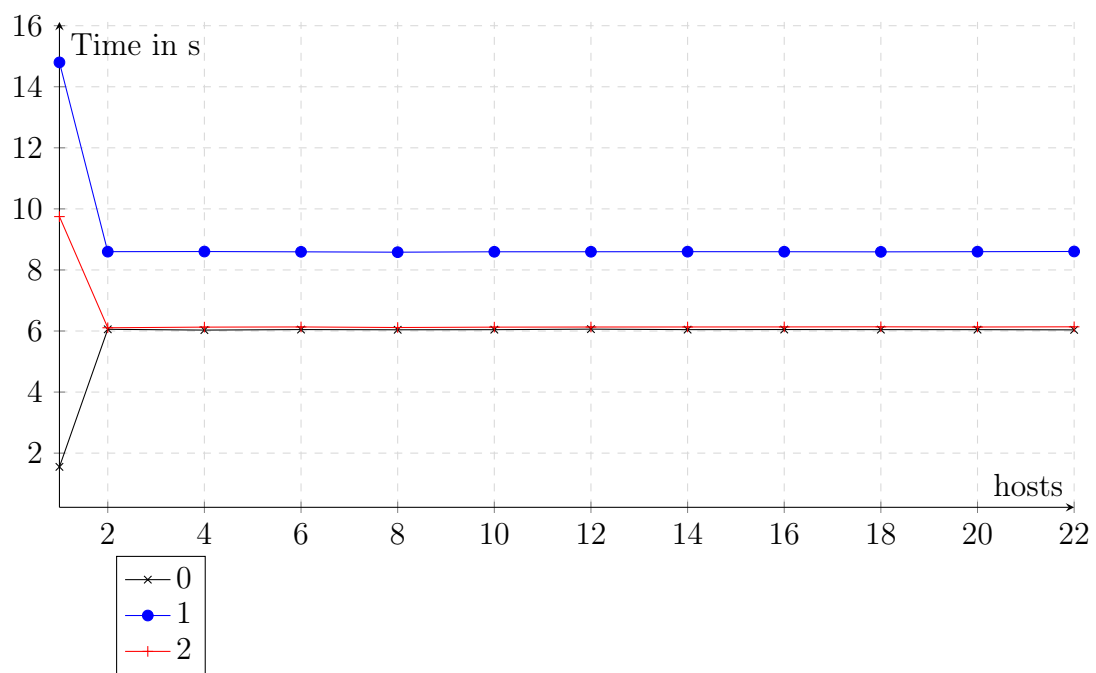
2.3 Laufzeitverhalten

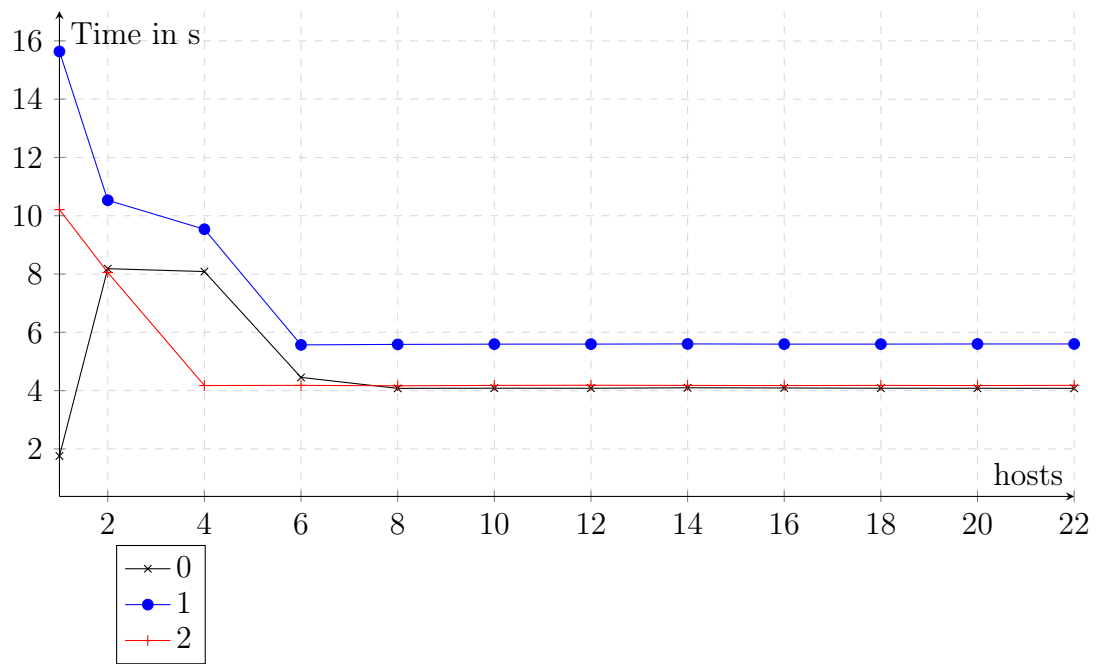
2.3.1 Festes n





2.3.2 Festes p





2.4 Fazit

3 Projektaufgabe 2: Numerische Integration mittels Parabelformel

In diesem Abschnitt wird auf die Projektaufgabe 2 eingegangen. Zunächst wird im ?? 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 ?? betrachtet. Anschließend erfolgt im ?? die Auswertung der Realisierung. Im letzten ?? 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. Die höchstmögliche 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 angegeben 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 existiert, 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 Listing 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

Wie in den Tabellen zu sehen ist liefert die Testfunktion 2 schneller bessere PI-Werte als Funktion 1. Ein Grund dafür das Testfunktion 1 schlechtere Ergebnisse als Funktion 2 liefert, könnte damit zusammenhängen, dass für die Funktion 1 die Oberegrenze PI selbst ist.

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,..