HTWK Leipzig
Fachbereich IMN
Wintersemester 2012/2013

Ausarbeitung zum Fach Message-Passing-Programmierung

Beleg im Fach Message-Passing-Programmierung

Kurt Junghanns
Philipp-Rosenthal-Straße 32
04103 Leipzig
kurt.junghanns@stud.htwk-leipzig.de

Marcel Kirbst
Sieglitz 39
06618 Molau
marcel.kirbst@stud.htwk-leipzig.de

1. Februar 2013

Inhaltsverzeichnis

1 Einleitung **3**

2 Message-Passing-Interface (MPI) **3**
2.1 Aufgabenstellung / Problembeschreibung 3
2.2 Programmbeschreibung 3
2.3 Laufzeitumgebung 4
2.4 Ergebnisse und Auswertung 5
2.4.1 Initiale Phase 5
2.4.2 Speedup 6
2.4.3 Effizienz 8
2.4.4 Kommunikationsanteil 9

3 Parallelrechnersystem MC-3 **10**
3.1 Aufgabenstellung / Problembeschreibung 10
3.2 Programmbeschreibung 10
3.3 Laufzeitumgebung 11
3.4 Ergebnisse 12
3.4.1 Speedup 12
3.4.2 Effizienz 13
3.4.3 Kommunikationsanteil 13
3.4.4 Laufzeiten der Funktionen 14
3.4.5 Genauigkeit der Funktionen 14

4 Anhang **16**
4.1 Quellcode-Listings MPI 16
4.2 Quellcode-Listings MC-3 18

1 Einleitung

Diese Ausarbeitung ist das Resultat der Veranstaltung Message-Passing-Programmierung im Wintersemester 2012/2013 und präsentiert die eingereichten Programme als Grundlage der mündlichen Prüfung der Prüflinge Kurt Junghanns und Marcel Kirbst.
Die Aufgabenstellung erfordert die Bearbeitung von zwei Aufgaben, die auf unterschiedlichen Hardware-Plattformen zu implementieren waren.

2 Message-Passing-Interface (MPI)

2.1 Aufgabenstellung / Problembeschreibung

Die empfohlene Aufgabenstellung für die MPI-Teilaufgabe ist die Implementierung eines so genannten Merge-Splitting-Sort-Algorithmus, der eine vorzugebende Anzahl natürlicher Zahlen in zufälliger Reihenfolge auf einer vorzugebenden Anzahl an Prozessoren sortiert. Dabei soll die benötigte Laufzeit ermittelt werden um im Anschluß Aussagen über das Laufzeitverhalten der Implementierung in Abhängigkeit zur verwendeten Element- und Prozessorzahl treffen zu können.
Dieser Algorithmus wurde in einem C-Programm unter Zuhilfenahme der MPI Bibliothek umgesetzt. Nachfolgend werden Aussagen zum Laufzeitverhalten getroffen. Dabei wurden den Laufzeitmessungen die Anzahl zu sortierender Elemente wie in der Aufgabenstellung empfohlen mit 20.000, 40.000 sowie 80.000 Elementen zu Grunde gelegt.

2.2 Programmbeschreibung

Zunächst wird ein Array mit n Zufallszahlen von jedem Prozess erzeugt. Dieses wird sortiert, wobei die lokale Prozessorlaufzeit T(1) ermittelt wird.
Jeder Prozessor erzeugt n/p Zufallszahlen und speichert sie in einem Array ab, wobei p die Anzahl der verwendeten Prozessoren und n die Anzahl der zu sortierenden Zufallszahlen angibt. Die Kommunikation zwischen den Prozessoren wird durch die Funktionen `MPI_Send()` und `MPI_Recv()` realisiert. Die Zeiterfassung erfolgt dabei mit der Funktion `MPI_Wtime()`. Anschließend wird der Merge-Splitting-Sort-Algorithmus ausgeführt. Nach diesem verteilten Sortiervorgang ermittelt jeder Prozessor die benötigten Messwerte:

- Gesamtlaufzeit
- Speedup

- Anteil der initialen Phase
- Kommunikationsanteil

Ein ausgezeichneter Prozessor sammelt die erfassten Werte und die sortierten Arrays ein und ermittelt die Durchschnittswerte. Dies erfolgt mit Hilfe der Funktionen `MPI_Reduce()` und `MPI_Gather()`. Dieser Prozessor ist außerdem für die Ausgabe der Resultate auf der Konsole zuständig.

Bei Aufruf erwartet das Programm folgendene Startparameter:

n: Größe des Gesamtarray an zu sortierenden Elementen, wobei der Wert n ein ganzzahliges Vielfaches der verwendeten Prozessoranzahl sein muss

k: (optional) Die Übergabe dieses Parameters bewirkt zusätzliche Ausgabe des sortierten Arrays

Der vollständige Quellcode des Programms sowie aller Skripte liegt dem Projektordner und ist außerdem aus dem öffentlichen Repository unter [1] abrufbar.

2.3 Laufzeitumgebung

Um die Entwicklung und die Tests der Implementierung so effektiv wie möglich zu gestalten, wurden mehrere BASH-Skripte erstellt. Das BASH-Skript `run.sh`, dass im Anhang vollständig aufgeführt ist, erfüllt dabei die folgenden Funktionen:

- Ermitteln der Prozessoranzahl **p**, Anzahl der zu sortierenden Elemente **n**, Name der zu kompilierenden C-Datei und Name der kompilierten Binärdatei als Startparameter
- prüfen, welcher der Rechner im Pool per SSH erreichbar sind
- ermitteln der durchschnittlichen Auslastung aller errichbaren Rechner im Pool
- sortieren der erreichbaren Poolrechner aufsteigend nach der durchschnittlichen Auslastung der letzten Minute, der letzten 5 Minuten und der letzten 15 Minuten
- Kompilieren der angegebenen C-Datei
- Ausführung der resultierenden Binärdatei auf den **p** Rechnern mit der geringsten durchschnittlichen Auslastung um das Risiko einer Verfälschung der Messergebnisse durch Fremdeinwirkung zu minimieren

[1] https://github.com/mkirbst/mpp_mpi

Ein weiteres BASH-Skript `bench.sh` ruft die Binärdatei mit den empfohlenen Elementanzahlen 20.000, 40.000 und 80.000 sequentiell für 2, 4, 8, 10, 16 und 20 Prozessoren auf und gibt die jeweils gemessenen Zeitintervalle übersichtlich aus, um eine grafische Auswertung mit gängigen Tabellenkalkulationsprogrammen zu gestatten.

2.4 Ergebnisse und Auswertung

2.4.1 Initiale Phase

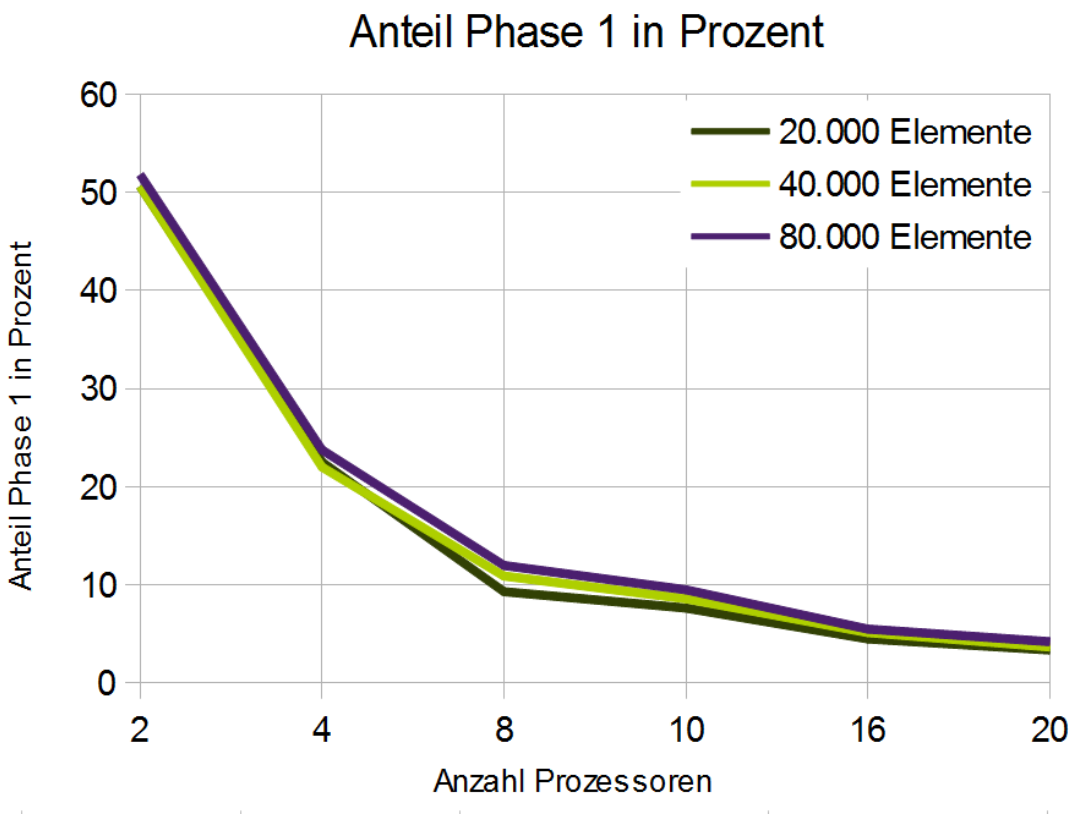

Abbildung 1: Phase-1-Diagramm für die MPI-Implementierung

In der Initialphase des Merge-Splitting-Sort-Algorithmus wird das lokale Array eines jeden Prozessors initial sortiert. Da die Initialphase im Gegensatz zu den nachfolgenden Phasen unabhängig von der Anzahl der genutzen Prozessoren immer nur einmalig durchlaufen wird, sinkt der Anteil der Initialphase an der Gesamtlaufzeit des Algorithmus mit steigender Prozessoranzahl. Während sich bei der Nutzung von nur zwei Prozessoren noch ein Laufzeitanteil der Phase 1 von 50 Prozent ergibt, sinkt dieser Wert bei Nutzung von 10 Prozessoren schon auf unter 10 Prozent, Tendenz weiter fallend.

2.4.2 Speedup

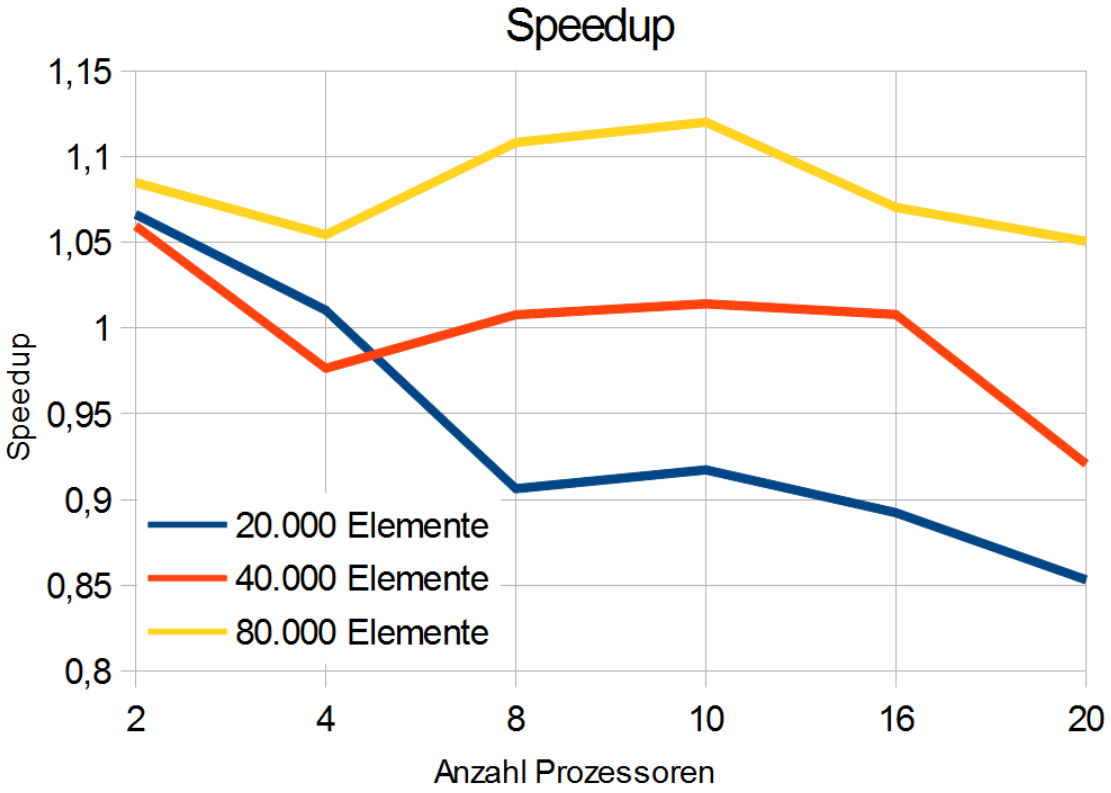

Abbildung 2: Speedup-Diagramm für die MPI-Implementierung

Der Speedup ist der Quotient aus der Laufzeit des Algorithmus bei der Nutzung eines Prozessors und der Laufzeit bei Nutzung mehrerer Prozessoren. Ein Speedup-Wert von 1 sagt aus, dass der Algorithmus auf einem Prozessor der Laufzeit des Algorithmus auf mehreren Prozessoren entspricht. Im Idealfall steigt der Speedup proportional mit der Anzahl der Prozessoren.

Nach Ahmdal setzt sich die Gesamtlaufzeit des parallelisierten Algorithmus zusammen aus einem Anteil mit nichtparallelisierbaren Code (sog.: sequenzieller Anteil) und einem Anteil an parallelisierbaren Code, dessen Laufzeit sich umgekehrt proportional zur Anzahl der benutzten Prozessoren verhält. Ahmdals Gesetz berücksichtigt hierbei jedoch nicht den mit steigender Prozessoranzahl ebenfalls wachsenden Kommunikationsaufwand.

Das zu erwartende Laufzeitverhalten für reale Implementierungen legt daher nahe, dass der Speedup nicht linear mit der Anzahl der eingesetzten Prozessoren ansteigt, sondern auf Grund des ebenfalls ansteigenden Kommunikationsaufwandes ab einer

bestimmten Prozessoranzahl wieder abnimmt.

Die durchgeführten Laufzeitmessungen mit der Implementierung des Algorithmus zeigen jedoch, dass bereits bei Nutzung von mehr als 10 Prozessoren der Speedup mit steigender Prozessoranzahl abnimmt. Der im Test beste erreichte Speedup stellte sich bei Nutzung von 10 Prozessoren und hinreichend vieler Elemente ein (>= 80.000). Bereits beim Einsatz von von 16 Prozessoren war die Laufzeitverringerung gegenüber der vollständig sequenziellen Implementierung nur noch marginal, Tendenz abnehmend. Dieses von den theoretisch erwarteten Messwerten abweichende Laufzeitverhalten ist das Ergebnis weiterer Einflussfaktoren wie beispielsweise:

- Eingesetzte Hardware (Netzwerkstruktur, nicht exklusiv genutzte Hardware)
- Eingesetzte Software (Betriebssystem, genutze Implementierung des Message-Passing-Interface)
- Implementierung des Algorithmus (eingesetzter Sortieralgorithmus, Kommunikationsablauf)

Im Laufe der Implementierung wurde ein direkter Einfluss des verwendeten Sortieralgorithmus auf die Gesamtlaufzeit deutlich. Es wurden verschiedene Quicksort-Implementierungen getestet, wobei durch die in der Standardbibliothek von C enthaltene Funktion `qsort()` die besten Ergebnisse liefert.

2.4.3 Effizienz

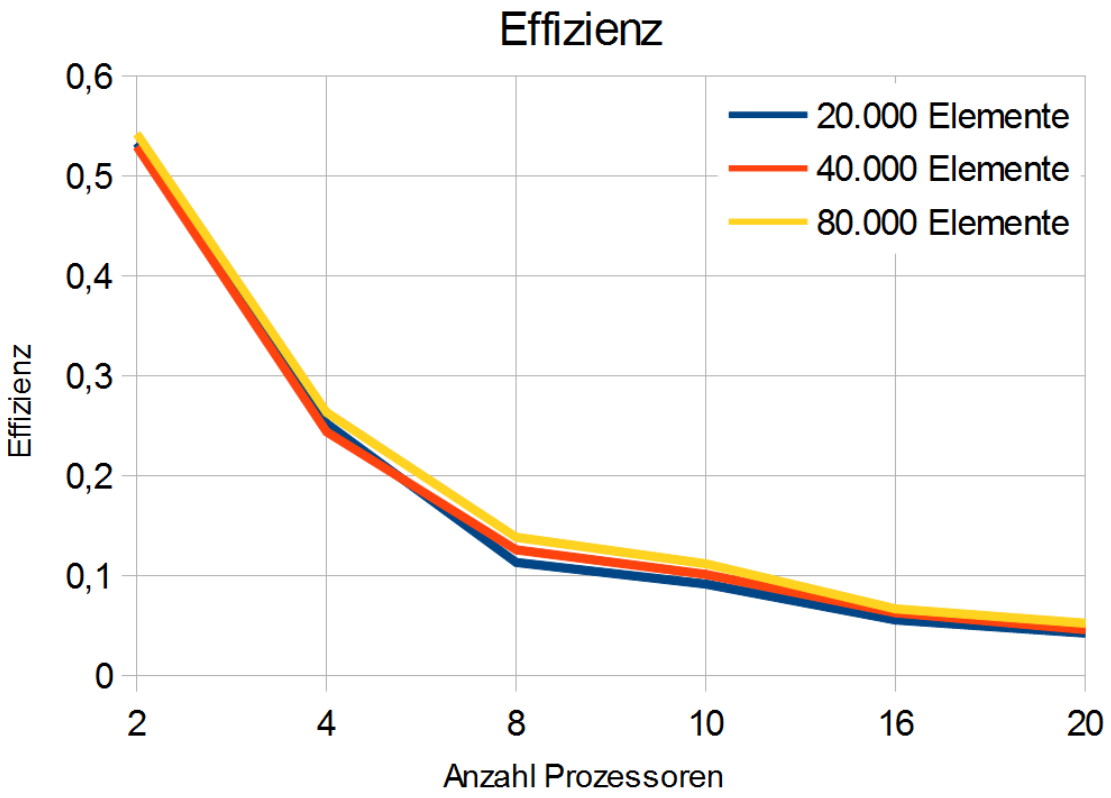

Abbildung 3: Effizienz-Diagramm für die MPI-Implementierung

Eine weiterer aussagekräftiger Wert ist die Effizienz. Die Effizienz gibt die relative Verbesserung in der Verarbeitungsleistung an und ergibt sich aus dem Quotient von Speedup und Prozessoranzahl. Wie aus dem betreffenden Diagramm ersichtlich wird, nimmt die Effizienz umgekehrt proportional zur Anzahl der eingesetzten Prozessoren ab. Dabei hat die Anzahl der zu sortierenden Elemente nur marginalen Einfluss auf die jeweiligen Effizienzwerte.

2.4.4 Kommunikationsanteil

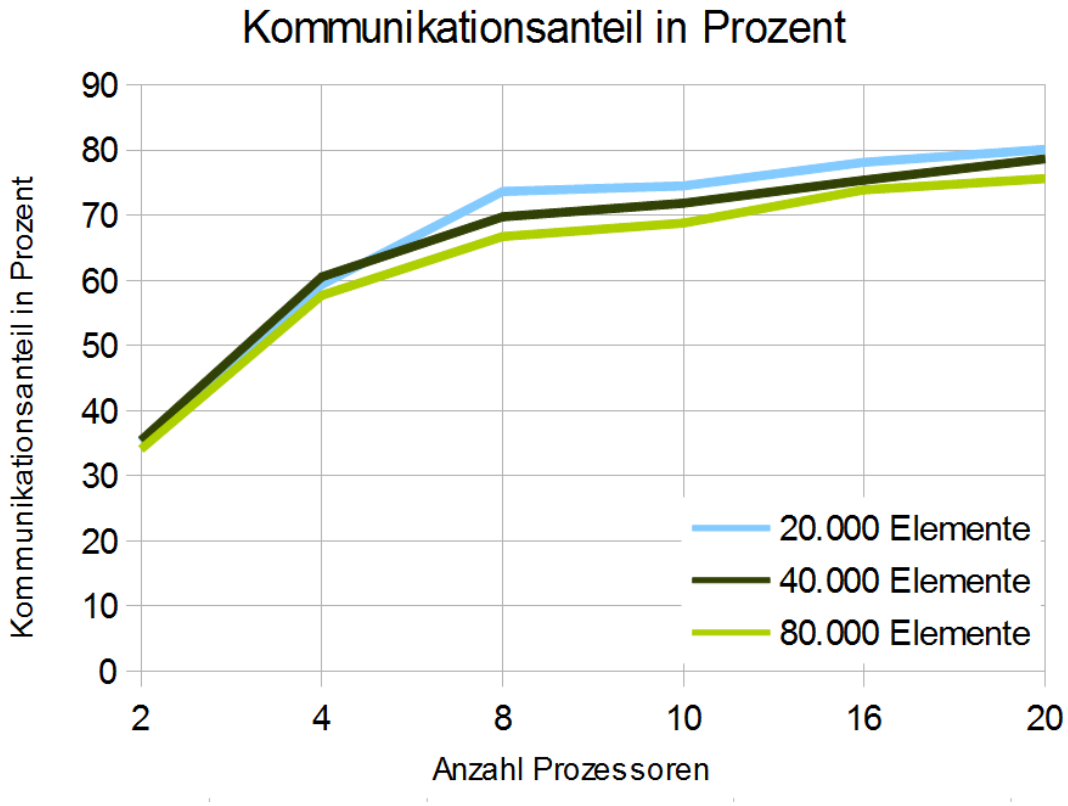

Abbildung 4: Diagramm des Kommunikationsanteils für die MPI-Implementierung

Die Auswertung der Messwerte zeigt, dass mit steigender Anzahl zu sortierender Elemente der Kommunikationsanteil an der Gesamtlaufzeit exponentiell ansteigt. Bereits bei Verwendung von 8 Prozessoren beträgt der Kommunikationsanteil an der Gesamtlaufzeit etwa 70 Prozent und steigt weiter an, sodass schon für 20 Prozessoren der Kommunikationsanteil über 80 Prozent mit weiterhin steigender Tendenz beträgt. Dieser hohe Anteil der Kommunikation an der Gesamtlaufzeit ist durch die in 2.4.2 aufgezählten Einflussfaktoren, besonders aber auf die Netzwerkstruktur und und MPI Implementierung, zurückzuführen.

Zusammenfassend kann die Aussage getroffen werden, dass die Implementierung des Merge-Splittung-Sort Algorithmus unter der MPI-Umgebung nicht geeignet ist.
Die unter bestimmten Voraussetzungen zu erreichende maginale Verkürzung der Gesamtlaufzeit ist gering, sodass vor der konkreten Nutzung dieser Implementierung im vornherein betrachtet werden muss, ob sich der Aufwand für die Nutzung der MPI Umgebung lohnt.

3 Parallelrechnersystem MC-3

3.1 Aufgabenstellung / Problembeschreibung

Für das Parallelrechnersystem MC-3 wurde die Aufgabe der nummerischen Integration mittels Parabelformel gewählt. Hierbei sollen Funktionen mit Hilfe der Parabelformel über einem gegebenen Intervall parallel von mehreren Rechenkernen nummerisch integriert werden.
Vorgegeben waren zu diesem Zweck folgende Funktionen:

- $f(x) = x * sin(x)$ in dem Intervall von 0 bis π
- $f(x) = \frac{4}{x^2+1}$ in dem Intervall von 0 bis 1

Das Ergebnis der Integration soll π sein. Dabei soll das Maschinen-π mit doppelter Genauigkeit als Referenz für die Präzision der Berechnung dienen. Dies wurde in einem C-Programm umgesetzt. Die Anzahl der Prozessoren, die zu integrierende Funktion und die Anzahl der Zerlegungen des Intervalls sollen dabei Eingabewerte sein. Als Resultat soll das Programm das Laufzeitverhalten und die Genauigkeit der Funktionen messen.

3.2 Programmbeschreibung

Das Programm wird auf dem Rechner Abel im Rechnerpool gestartet, der mit dem Multiprozessorsystem MC-3 verbunden ist. Als Startparameter muss `n` und die Nummer der zu nutzenden Funktion übergeben werden. `n` wird intern mit 8 multipliziert, um die Teilbarkeit mit der Prozessoranzahl zu gewährleisten. Der Funktionsnummer 1 ist $f(x) = \frac{4}{x^2+1}$ zugeordnet und 2 der Funktion $f(x) = x * sin(x)$.

Die Zeiterfassung findet mit der Funktion `TimeNowHigh()` statt. Die Kommunikation hingegen mit `Send()` und `Recv()`.
Für die Anordnung der Prozessoren wurde ein Stern gewählt. Dabei werden Links von allen Prozessoren mit der ID ungleich 0 zum Prozessor mit der ID 0 angelegt und verwendet.

Nach Auswertung der Argumente, ermittelt jeder Prozessor die Laufzeit für die sequentielle Berechnung des Integrals T(1).
Anschließend berechnet jeder Prozessor das Integral seines Teilintervalls. Schlussendlich empfängt der Prozessor mit der ID 0 alle ermittelten Teilintegrale sowie die

Gesamtlaufzeit, T(1) und die zur Kommunikation benötigte Zeit. Dieser Prozessor bildet aus den Zeitwerten Durchschnittswerte und gibt sie in der Konsole aus.
Der vollständige Quellcode dieses Programms sowie aller Skripte liegt dem Projektordner und ist außerdem aus dem öffentlichen Repository unter [2] abrufbar.

3.3 Laufzeitumgebung

Hier ist nur ein Skript zur Erfassung der Messwerte für verschiedene Prozessoranzahlen, Intervallteilungen pro Prozessor und Funktionen zu nennen.
Dieses Skript ist im Anhang Quellcode-Listings MC-3 zu finden.

[2] https://github.com/TBoonX/mpp_mc3

3.4 Ergebnisse

3.4.1 Speedup

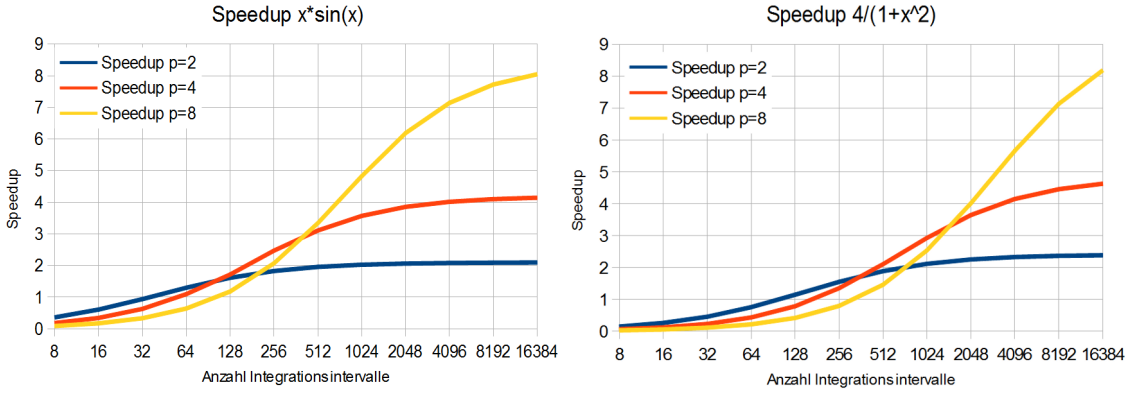

Abbildung 5: Diagramm des Speedup für die MC-3-Implementierung abhängig von n

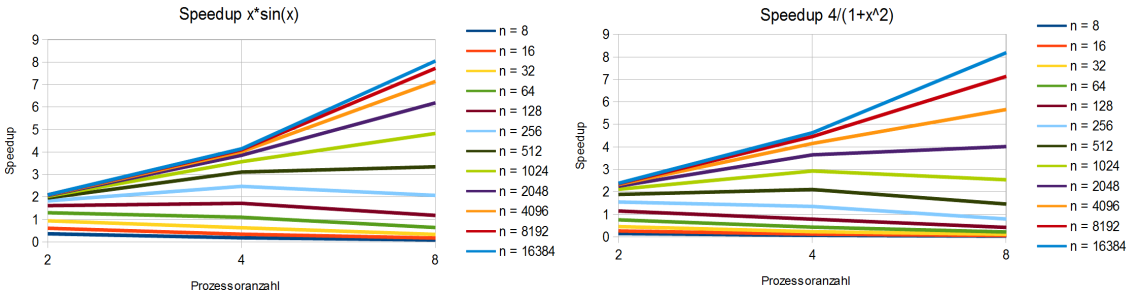

Abbildung 6: Diagramm des Speedup für die MC-3-Implementierung abhängig von p

Auf Grund der Bregrenzung auf maximal acht Prozessoren können nur besschränkte Aussagen darüber getroffen werden, ob sich das Laufzeitverhalten mit weiter ansteigender Prozessoranzahl an das von Ahmdals Gesetz beschriebene Verhalten annährt.

Die Messwerte zeigen, dass bei 128 Integrationsintervallen und mehr ein Speedup größer 1 vorherrscht. Werden alle 8 Prozessoren und 16384 Integrationsintervalle genutzt, ergibt sich sogar ein Speedup von 8.

3.4.2 Effizienz

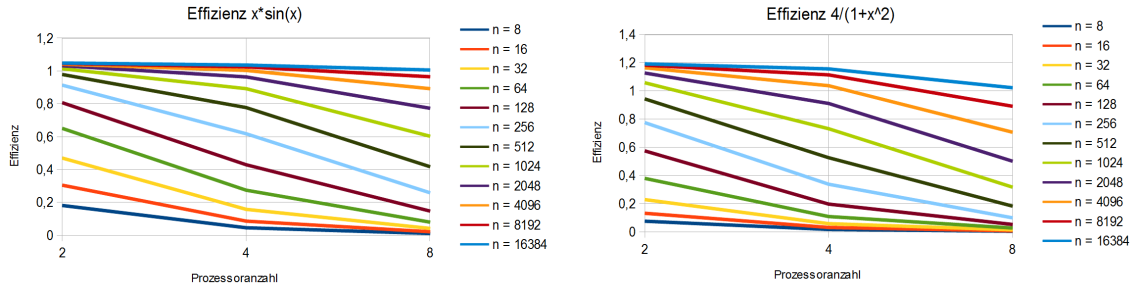

Abbildung 7: Diagramm der Effizienz für die MC-3-Implementierung

Bei einem hohem Integrationsintervall ergibt sich ein Effizienzwert von bis zu 1,2. Das legt den Schluss nahe, dass diese Implementierung gut skaliert, was bedeutet, dass sich die Laufzeit bei Verdoppelung der Prozessoranzahl mehr als halbiert.

3.4.3 Kommunikationsanteil

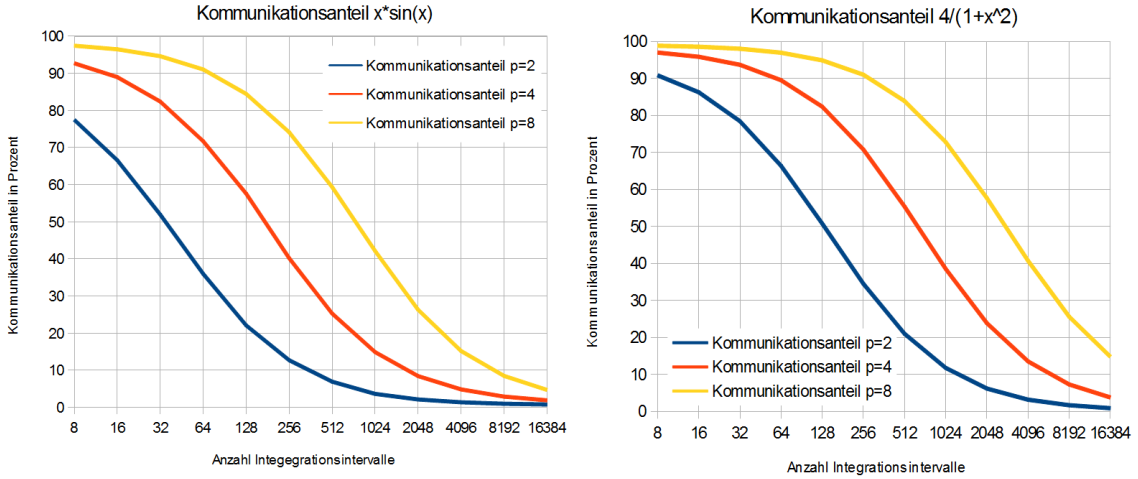

Abbildung 8: Diagramm des Kommunikationsanteils für die MC-3-Implementierung

In dieser Implementierung steigt der Kommunikationsanteil mit der Anzahl der Prozessoren und sinkt mit der Anzahl der Integrationsintervalle. Bei bis zu 32 Integrationsintervallen ist dieser Kommunikationsoverhead sehr groß. was den Schluss nahe legt, dass die Funktionen zum senden und empfangen der Optimierung bedürfen.

Bei ausreichend vielen Integrationsintervallen (=> 8192) ist dieser Anteil mit weniger als 10 Prozent sehr gering.

3.4.4 Laufzeiten der Funktionen

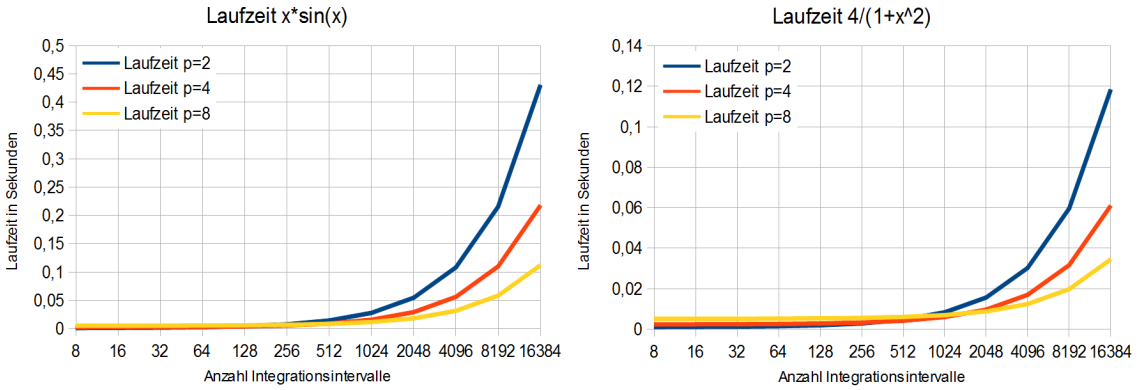

Abbildung 9: Diagramm der Laufzeiten für die MC-3-Implementierung

Es wird ersichtlich, dass Funktion 2 eine deutlich höhere Laufzeit besitzt. Dies liegt darin begründet, dass sie einen eigenen Funktionsaufruf besitzt (`sin()`) wodurch ein Kontextwechsel vollzogen werden muss und zusätzlich mehr Operationen als bei Funktion 1 durchgeführt werden müssen.

3.4.5 Genauigkeit der Funktionen

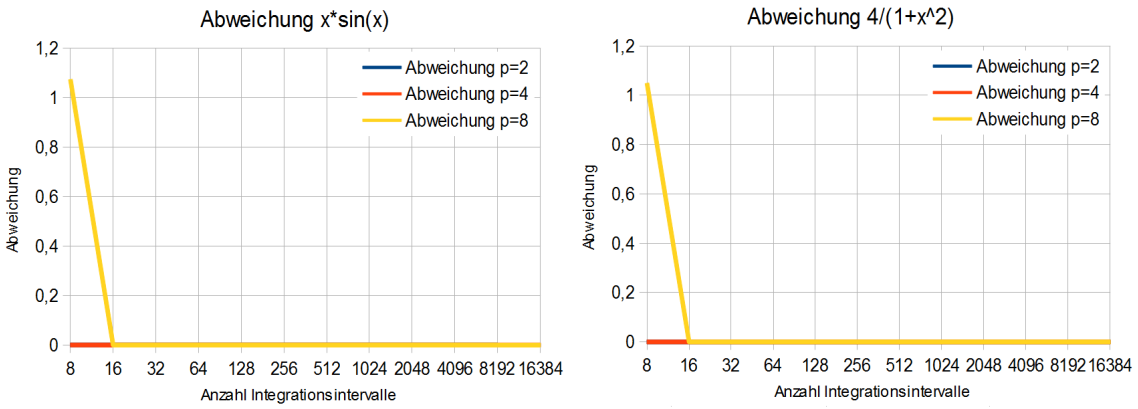

Abbildung 10: Diagramm der Abweichung für die MC-3-Implementierung

Bereits bei 16 Integrationsintervallen ist die Abweichung des ermittelten Wertes von π sehr gering.
Ab 256 Integrationsintervallen weicht der mit der Funktion 1 ermittelte Wert für π nicht mehr vom Maschinen-π ab.
Mit Funktion 2 ist dies erst bei 8192 Integrationsintervallen der Fall.
Dieser Unterschied liegt in der oberen Integrationsgrenze der Funktionen begründet.

Funktion 2 verwendet π mit 15 Nachkommastellen, was bereits eine Ungenauigkeit darstellt. Weiterhin ist die Implementierung der Funktion `sin()` unbekannt, was Einflüsse auf die Genauigkeit und das Laufzeitverhalten hat.

Die Implementierung der nummerischen Integration mittels Parabelform auf MC-3 ist geeignet. Wird eine bestimmte Annäherung an einen gesuchten Wert angestrebt, ist dies in kurzer Zeit möglich.

4 Anhang

4.1 Quellcode-Listings MPI

```
#!/bin/bash

##0##  process script args:
CPUCOUNT=0
VRANGE=0
INPUTFILENAME="cluster.c"
OUTPUTFILENAME="cluster"


function usage {
  echo "Usage: $0 -c CPUCOUNT -v VALUERANGE -i INPUTFILE -o OUTPUTFILE"
  exit 1;
}

##8 params required
if [ $# -ne 8 ] ; then ## erzwinge die Angabe aller Startparameter
  usage;
fi

##process args
while getopts c:hi:o:v: opt
do
  case "$opt" in
    c) CPUCOUNT=$OPTARG;;
    h) usage;;
    i) INPUTFILENAME=$OPTARG;;
    o) OUTPUTFILENAME=$OPTARG;;
    v) VRANGE=$OPTARG;;
    \?) usage;;
  esac
done
echo "CPUCOUNT: $CPUCOUNT"
echo "VRANGE:   $VRANGE"
echo "INPUT:    $INPUTFILENAME"
echo "OUT:      $OUTPUTFILENAME"


##1##  compile
echo "STAGE 1 - compiling $INPUTFILENAME ..."
mpicc -Wall -o $OUTPUTFILENAME $INPUTFILENAME


##2## create hostlist dynamically
echo "STAGE 2 - creating host list ..."
HOSTLISTFILENAME="load.txt"

##remove already existing file without warning
touch $HOSTLISTFILENAME         ## create file if not already there
rm $HOSTLISTFILENAME            ## remove file

```

```
##ssh trough simson clients for every pingable simson
for i in 01 02 03 04 05 06 07 08 09 {10..24}
do
  ping -c 1 simson$i  > /dev/null
  if [ $? = 0 ]
  then
 ## check per ssh cat /proc/loadavg and check with regex
    echo "`ping -c 1 simson${i} | grep "64 bytes" | awk ' BEGIN {FS="("} {print $2}' | awk ' BEGIN {FS=")"} {print $1}'` `ssh simson${i} cat /proc/loadavg`" | grep -v -E '141.57.9.[0-9]{2} $' >> $HOSTLISTFILENAME
  fi
done
HOSTNR=`wc -l $HOSTLISTFILENAME | awk '{print $1}'`  ## zaehle Anzahl erreichbarer Hosts

##remove already existing file without warning
touch $HOSTLISTFILENAME.sorted         ## create file if not already there
rm $HOSTLISTFILENAME.sorted            ## remove file

echo "Sortiere ${HOSTNR} Hosts nach Auslastung ..."
sort -k 2 $HOSTLISTFILENAME >> $HOSTLISTFILENAME.sorted
awk '{print $1}' $HOSTLISTFILENAME.sorted > $HOSTLISTFILENAME

echo "Zur Ausfuehrung werde folgenden ${CPUCOUNT} Hosts benutzt, da diese derzeit die geringste Auslastung haben:"
head -n ${CPUCOUNT} ${HOSTLISTFILENAME}.sorted > head.list
awk '{print "Node: " $1 " - Load on this Node:   " $2 " (avg last min)   " $3 " (avg last 5 min)   " $4 " (avg last 15 min) "}' head.list

sleep 1

##3## run program on this hosts
echo "STAGE 3 - run $OUTPUTFILENAME on $CPUCOUNT cpus "
mpirun -np $CPUCOUNT -hostfile $HOSTLISTFILENAME $OUTPUTFILENAME $VRANGE

##cleanup - remove temporary used files
#rm $HOSTLISTFILENAME
#rm $HOSTLISTFILENAME.sorted
```

Listing 1: MPI BASH-Script: run.sh

```
#!/bin/bash

## Initial run.sh aufrufen um Auslastung der Pool-Rechner zum jetzigen Zeitpunkt zu ermitteln
./run.sh -c 20 -v 20 -i cluster.c -o cluster

for val in 20000 40000 80000 # Anzahl der zu messenden n Elemente
do
  for cpu in 2 4 8 10 16 20  # fuer p Prozessoren
  do       # jeweils 5 Messungen
   mpirun -np $cpu -hostfile load.txt cluster $val
   mpirun -np $cpu -hostfile load.txt cluster $val
   mpirun -np $cpu -hostfile load.txt cluster $val
   mpirun -np $cpu -hostfile load.txt cluster $val
```

```
14    mpirun -np $cpu -hostfile load.txt cluster $val
15   done
16 done
```

Listing 2: MPI BASH-Script: bench.sh

4.2 Quellcode-Listings MC-3

```
1  #!/bin/bash
2  echo 'Prozessoranzahl;Intervalle;Laufzeit;Speedup;Abweichung' > $1
3  for x in 2 ; do
4   for y in 1 2 4; do
5     for z in 1 2 4 8 16 32 64 128 256 512 1024 2048; do
6      echo -n $(( $x*$y ))\;$z\; >> $1
7      run -f1 $y $x mc3.px $z $2 >> $1
8      echo '' >> $1
9     done
10  done
11 done
```

Listing 3: MC-3 BASH-Script: erfassung.sh