

HTWK Leipzig
Fachbereich IMN
Wintersemester 2012/2013

**Ausarbeitung zum Fach
Message-Passing-Programmierung
–VORABVERSION–**

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

30. Januar 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	4
2.4.1	Initiale Phase	4
2.4.2	Speedup	5
2.4.3	Effizienz	7
2.4.4	Kommunikationsanteil	8
3	Parallelrechnersystem MC3	9
3.1	Aufgabenstellung / Problembeschreibung	9
3.2	Programmbeschreibung	9
3.3	Laufzeitumgebung	9
3.4	Ergebnisse	9
3.5	Auswertung	9
4	Anhang	9
4.1	Quellcode-Listings MPI	9
5	Glossar	12
6	Literaturverzeichnis	13

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

```
1 #include "mpi.h"
2 #include <stdlib.h>
3 #include <stdio.h>
4 #include <time.h>
5
6 //ungerade Prozessornummer
7
8 //erhalte Array
9 MPI_Recv(&local[nLocal], nLocal, MPI_INT, rank_world+1, 1, MPI_COMM_WORLD, &
    status);
10
11 //sortiere Array
12 wtimesinnersort[j*2+0] = MPI_Wtime();
13 quicksort(local, 0, nLocal*2-1);
14 wtimesinnersort[j*2+1] = MPI_Wtime();
15
16 //obere Teil des Arrays wird an Prozessor rank_world+1 gesendet
```

```

17 MPI_Send(&local[nLocal], nLocal, MPI_INT, rank_world+1, 1, MPI_COMM_WORLD);
18 }

```

Listing 1: MPI C-Programm: cluster.c

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**, das 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, Name der kompilierten Binärdatei als Startparameter
- prüfen, welcher der Rechner im Pool per SSH erreichbar sind
- ermitteln der durchschnittlichen Auslastung aller erreichbaren Rechner im Pool
- sortieren der erreichbaren Poolrechner aufsteigend nach der durchschnittlichen Auslastung der letzten Minute, der letzten 5 Minuten, 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

Ein weiteres 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 gestalten.

2.4 Ergebnisse und Auswertung

2.4.1 Initiale Phase

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 genutzten 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

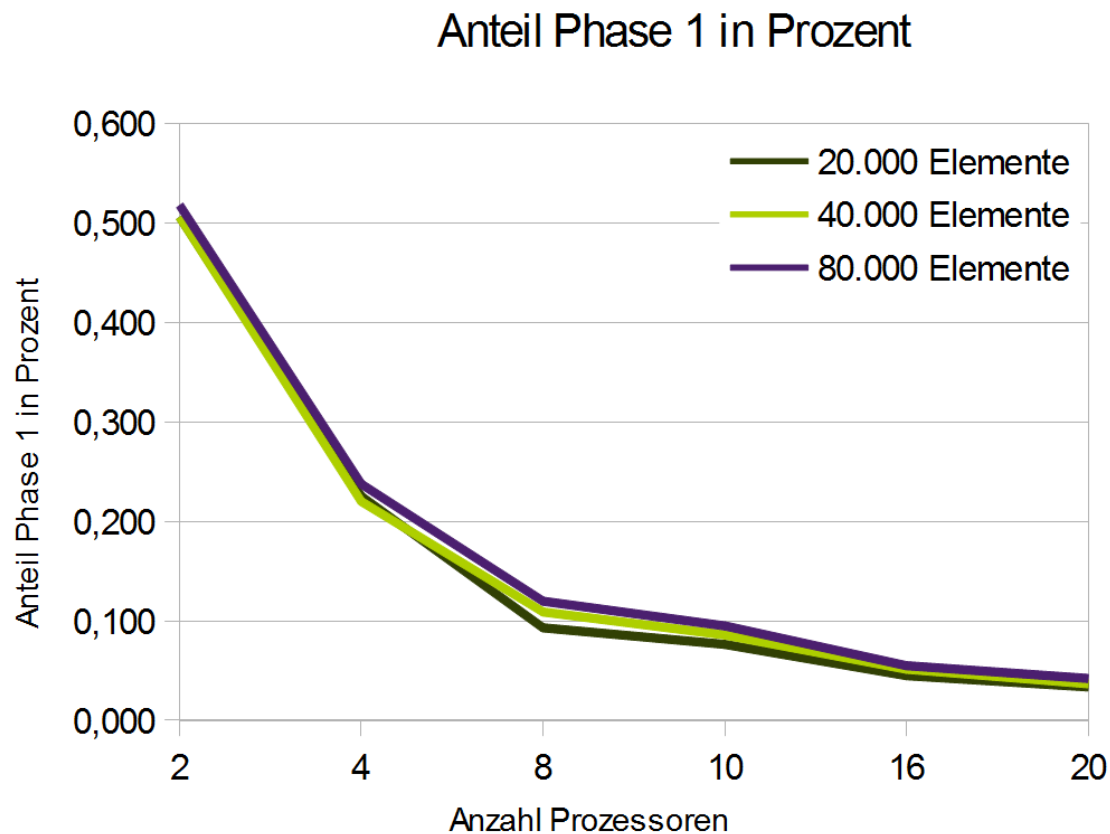


Abbildung 1: Phase-1-Diagramm für die MPI-Implementierung

dieser Wert bei Nutzung von 10 Prozessoren schon auf unter 10 Prozent, Tendenz weiter fallend.

2.4.2 Speedup

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

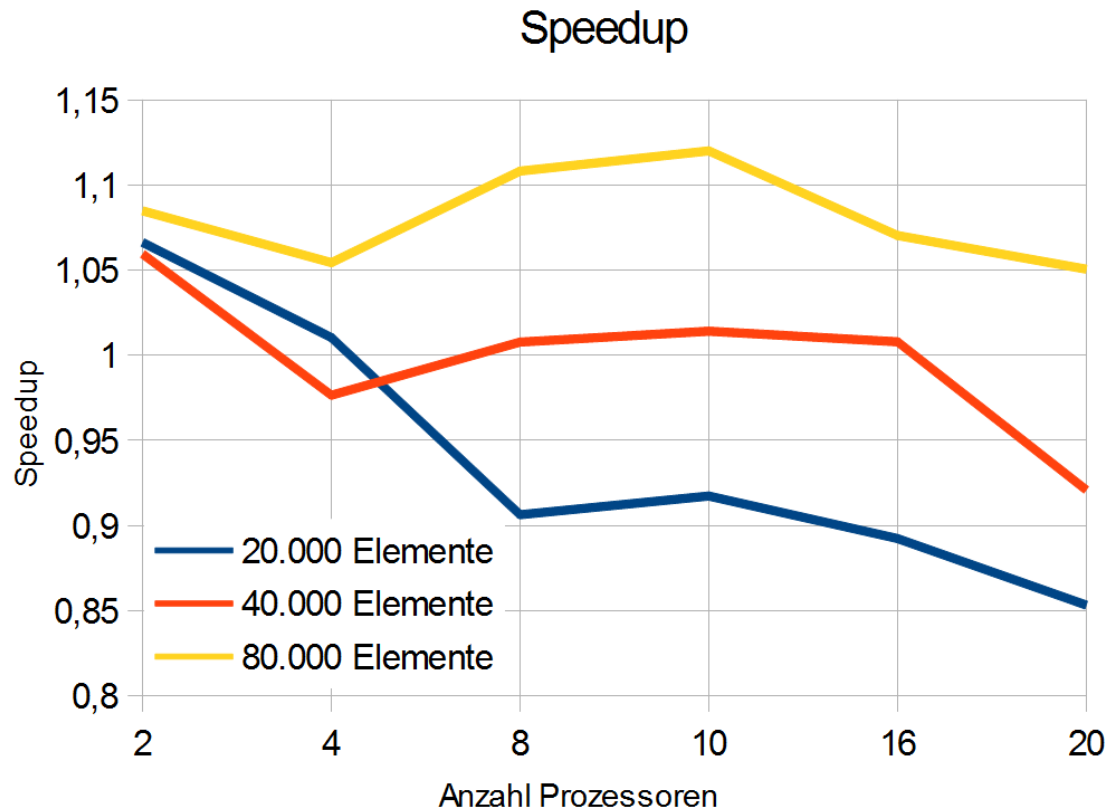


Abbildung 2: Speedup-Diagramm für die MPI-Implementierung

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 vielen Elementen ein (≥ 80.000). Bereits beim Einsatz 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, genutzte Implementierung des Message-Passing-Interface)
- Implementierung des Algorithmus (eingesetzter Sortieralgorithmus, Kommunikationsablauf)

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

2.4.3 Effizienz

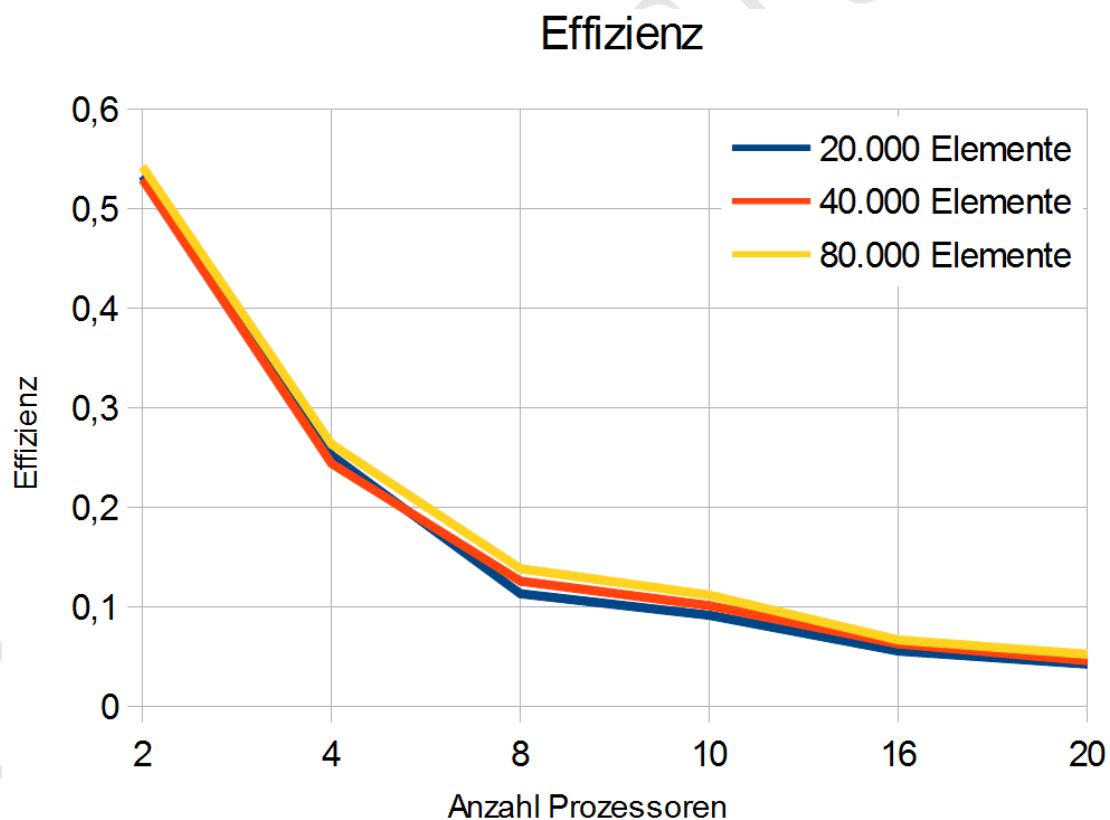


Abbildung 3: Effizienz-Diagramm für die MPI-Implementierung

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

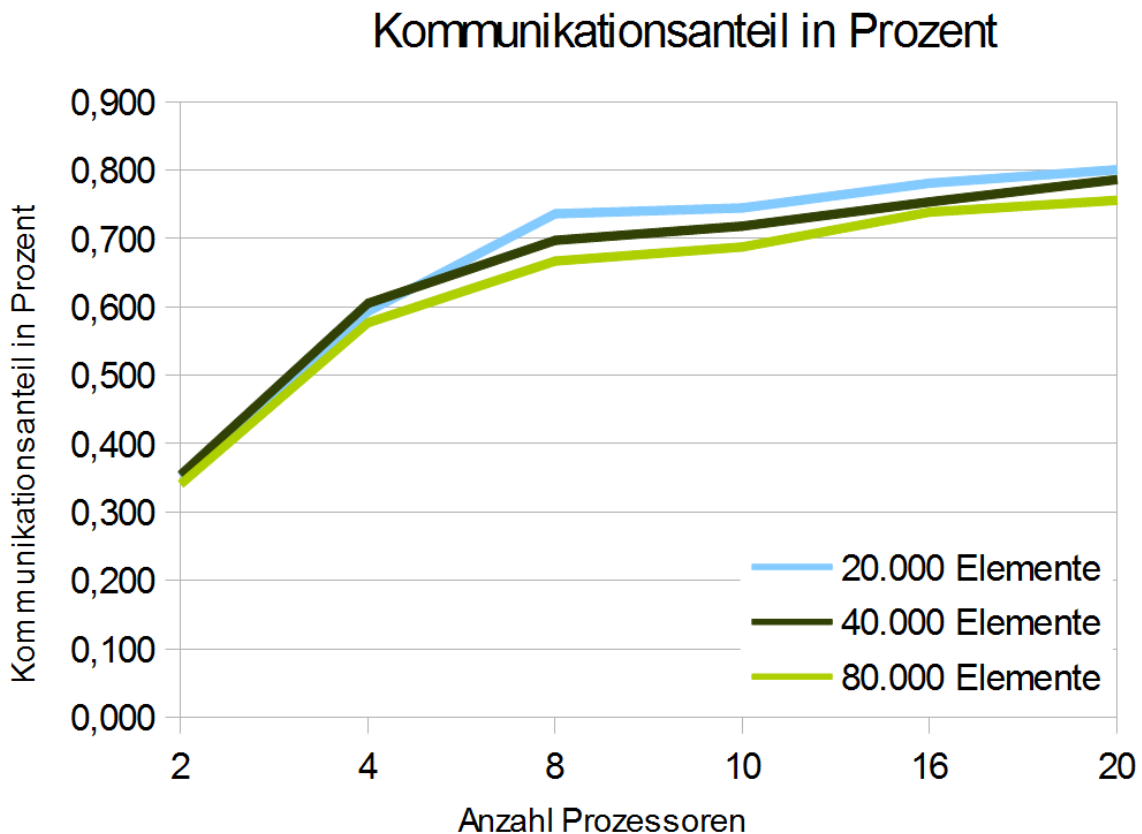


Abbildung 4: Diagramm des Kommunikationsanteils für die MPI-Implementierung

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 marginale Verkürzung der Gesamtlaufzeit ist gering, sodass vor der konkreten Nutzung dieser Implementierung im vornherrein betrachtet werden muss, ob die Anzahl der Prozessoren und zu sortierenden Elemente zu einer Verbesserung der Gesamtlaufzeit führt.

3 Parallelrechnersystem MC3

3.1 Aufgabenstellung / Problembeschreibung

3.2 Programmbeschreibung

3.3 Laufzeitumgebung

3.4 Ergebnisse

3.5 Auswertung

4 Anhang

4.1 Quellcode-Listings MPI

```
1 #!/ bin / bash
2
3 ##0## process script args:
4 CPUCOUNT=0
5 VRANGE=0
6 INPUTFILENAME="cluster.c"
7 OUTPUTFILENAME="cluster"
8
9
10 function usage {
11     echo "Usage: $0 -c CPUCOUNT -v VALUERANGE -i INPUTFILE -o OUTPUTFILE"
12     exit 1;
13 }
14
15 ##8 params required
16 if [ $# -ne 8 ] ; then ## erzwinge die Angabe aller Startparameter
17     usage;
18 fi
19
20 ##process args
21 while getopts c:hi:o:v: opt
22 do
```

```

23  case "$opt" in
24      c) CPUCOUNT=$OPTARG;;
25      h) usage;;
26      i) INPUTFILENAME=$OPTARG;;
27      o) OUTPUTFILENAME=$OPTARG;;
28      v) VRANGE=$OPTARG;;
29      \?) usage;;
30  esac
31  done
32  echo "CPUCOUNT: $CPUCOUNT"
33  echo "VRANGE:    $VRANGE"
34  echo "INPUT:      $INPUTFILENAME"
35  echo "OUT:        $OUTPUTFILENAME"
36
37
38  ##1## compile
39  echo "STAGE 1 - compiling $INPUTFILENAME ..."
40  mpicc -Wall -o $OUTPUTFILENAME $INPUTFILENAME
41
42
43  ##2## create hostlist dynamically
44  echo "STAGE 2 - creating host list ..."
45  HOSTLISTFILENAME="load.txt"
46
47  ##remove already existing file without warning
48  touch $HOSTLISTFILENAME          ## create file if not already there
49  rm $HOSTLISTFILENAME             ## remove file
50
51  ##ssh trough simson clients for every pingable simson
52  for i in 01 02 03 04 05 06 07 08 09 {10..24}
53  do
54      ping -c 1 simson$i > /dev/null
55      if [ $? = 0 ]
56      then
57          ## check per ssh cat /proc/loadavg and check with regex
58          echo "`ping -c 1 simson${i} | grep "64 bytes" | awk ' BEGIN {FS="("} {print ←
59              $2}' | awk ' BEGIN {FS="}")" {print $1}'" `ssh simson${i} cat
60  /proc/loadavg`" | grep -v -E '141.57.9.[0-9]{2} $' >> $HOSTLISTFILENAME
61      fi
62  done
63  HOSTNR=`wc -l $HOSTLISTFILENAME | awk '{print $1}'` ## zaehle Anzahl ←
64  erreichbarer Hosts
65
66  ##remove already existing file without warning
67  touch $HOSTLISTFILENAME.sorted    ## create file if not already there
68  rm $HOSTLISTFILENAME.sorted       ## remove file
69
70  echo "Sortiere ${HOSTNR} Hosts nach Auslastung ..."
71  sort -k 2 $HOSTLISTFILENAME >> $HOSTLISTFILENAME.sorted
72  awk '{print $1}' $HOSTLISTFILENAME.sorted > $HOSTLISTFILENAME
73
74  echo "Zur Ausfuehrung werde folgenden ${CPUCOUNT} Hosts benutzt, da diese derzeit←
75  die geringste Auslastung haben:"
76  head -n ${CPUCOUNT} ${HOSTLISTFILENAME}.sorted > head.list
77  awk '{print "Node: " $1 " - Load on this Node:  " $2 " (avg last min)  " $3 " (←
78  avg last 5 min)  " $4 " (avg last 15 min) }'" head.list

```

```

76 sleep 1
77
78 ###3## run program on this hosts
79 echo "STAGE 3 - run $OUTPUTFILENAME on $CPUCOUNT cpus "
80 mpirun -np $CPUCOUNT -hostfile $HOSTLISTFILENAME $OUTPUTFILENAME $VRANGE
81
82 ##cleanup - remove temporary used files
83 #rm $HOSTLISTFILENAME
84 #rm $HOSTLISTFILENAME.sorted

```

Listing 2: MPI BASH-Script: run.sh

```

1 #!/bin/bash
2
3 ## Initial run.sh aufrufen um Auslastung der Pool-Rechner zum jetzigen Zeitpunkt ↔
  zu ermitteln
4 ./run.sh -c 20 -v 20 -i cluster.c -o cluster
5
6 for val in 20000 40000 80000 # Anzahl der zu messenden n Elemente
7 do
8     for cpu in 2 4 8 10 16 20 # fuer p Prozessoren
9     do # jeweils 5 Messungen
10         mpirun -np $cpu -hostfile load.txt cluster $val
11         mpirun -np $cpu -hostfile load.txt cluster $val
12         mpirun -np $cpu -hostfile load.txt cluster $val
13         mpirun -np $cpu -hostfile load.txt cluster $val
14         mpirun -np $cpu -hostfile load.txt cluster $val
15     done
16 done

```

Listing 3: MPI BASH-Script: bench.sh

5 Glossar

DHCP-Server DHCP steht als Abkürzung für "Dynamic Host Configuration Protocol" und beschreibt Techniken um Hosts in Netzwerken dynamisch Netzwerkparameter wie IP-Adressen zuzuweisen¹

Router Ein Rechnersystem mit mindestens zwei Netzwerkschnittstellen, das Netzwerkverkehr zwischen diesen Netzwerkschnittstellen nach einem Regelwerk vermittelt und weiterleitet.

Routerdistribution Eine spezielle Art von Betriebssystem, deren Hauptaugenmerk bei der Konzeption und Entwicklung darauf liegt Router-Funktionen sicher und stabil auszuführen

VLAN Die Abkürzung VLAN steht für Virtual Local Area Network und fasst Techniken zusammen um physikale Netzwerkstrukturen logisch zu Segmentieren, beispielsweise zur Erhöhung der Sicherheit oder um Broadcast-Domänen zu verkleinern.

¹ [6]

6 Literaturverzeichnis

Musterfrau, Renate: Muster. Frankfurt 2003.

Mustermann, Helmut: Noch ein Muster. Mit einer Einleitung hrsg. von Frank Muster. Frankfurt 2003.

Literatur

- [1] <http://www.ipcop.org/1.4.0/en/install/html/> , abrufbar am 16.12.2012
- [2] http://www.ipcopwiki.de/index.php/Samba_Server , abrufbar am 20.12.2012
Anm.: Der Artikel zu diesem Addon ist zwar noch verfügbar, jedoch nicht die eigentlichen Dateien, die für das Addon erforderlich sind.
- [3] <http://www.ipcop.org/2.0.0/en/admin/html/whatsnew.html> , abrufbar am 11.01.2013
- [4] <http://wiki.ipfire.org/de/addons/start> , abrufbar am 10.01.2013
- [5] <http://www.freebsd.org/doc/de/books/faq/hardware.html#which-hardware-to-get> , abrufbar am 12.01.2013
- [6] <http://www.isc.org/software/dhcp> , abrufbar am 11.01.2013