Universität Hamburg
Department Informatik
Wissenschaftliches Rechnen
Prof. T. Ludwig, J. Kunkel, N. Hübbe

Übungsblatt 1 zum Praktikum
Paralleles Programmieren
im SoSe 2013

---

In diesem Übungsblatt wird die Benutzung des Clusters bzw. Linux nochmal erwähnt, sofern Sie sich damit noch nicht auseinander gesetzt haben. Das Debugging von Fehlerhaften Programmen wird angesprochen. Schließlich sollen kleine Programme mit MPI erstellt werden und somit erste parallele Anwendungen.
Als kleiner Hinweis: ein stures Abarbeiten der Aufgaben ist durchaus möglich, gerne könnt ihr aber etwas mehr lesen und nicht nur den wenigsten Aufwand betreiben. Die Lösungen werden zwar kurz im Praktikum besprochen, aber ihr solltet durchaus eigene Erfahrungen mit der Programmierung von C bzw. Fortran und MPI sammeln.
Sollten Probleme auftauchen, wenden Sie sich bitte an die Mailingliste der Veranstaltung:

`papo-13@wr.dkrz.de`

# 1 Cluster-Kennung

Im Folgenden sollen Sie sich auf dem Cluster einloggen und das Navigieren in einer Shell üben.
Bitte melden Sie sich auf dem Cluster an und machen Sie sich ein wenig mit den grundlegenden Linux-Befehlen vertraut. Informationen dazu finden Sie in unserem Wiki in der Sektion Cluster:

`http://wr.informatik.uni-hamburg.de/teaching/ressourcen`

Beachten Sie insbesondere den Link „Beginner Guide“.
Die folgenden konkrete Aufgaben haben Sie zu bewältigen:

1. *Einloggen*

   Loggen Sie sich auf dem Cluster mit ihrem Benutzer und Passwort ein.

2. *Bewegen im CLI (Command Line Interface)*

   a) Machen Sie sich mit der Verwendung von Manual-Pages vertraut: `$> man man`
   b) Lassen Sie sich den Namen des aktuellen Arbeitsverzeichnisses anzeigen: `$> man pwd`
   c) Lassen Sie sich den Inhalt Ihres Homeverzeichnisses anzeigen: `$> man ls`
   d) Erzeugen Sie ein neues Verzeichnis mit dem Namen `testdir`: `$> man mkdir`
   e) Ändern Sie das Arbeitsverzeichnis in das neue Verzeichnis: `$> cd testdir`
   f) Lassen Sie sich noch einmal das aktuelle Arbeitsverzeichnis anzeigen.
   g) Erzeugen Sie eine leere Datei mit dem Namen `testfile`: `$> man touch`
   h) Benennen Sie die neue Datei um in `testfile2`: `$> man mv`
   i) Kopieren Sie die umbenannte Datei in `testfile3`: `$> man cp`
   j) Löschen Sie die Datei `testfile2`: `$> man rm`

   **Frage:** Warum gibt es keine Manual-Page zum Kommando `cd`? (Tipp: `$> man bash`)

3. *Packen eines Archiv*

   a) Erstellen Sie ein Verzeichnis mit dem Namen `testarchiv`.
   b) Erzeugen Sie darin eine Datei mit zufälligem Inhalt:
   `$> dd if=/dev/urandom of=testarchiv/zufallsdatei bs=1k count=256`

c) Lassen Sie sich die Größe der Datei anzeigen:
`$> ls -lh testarchiv/zufallsdatei`

d) Lassen Sie sich die Größe des Verzeichnisses anzeigen:
`$> ls -ldh testarchiv`

e) Erzeugen Sie ein `tar`-Archiv, welches das Verzeichnis enthält:
`$> tar -cf testarchiv.tar testarchiv`

f) Lassen Sie sich die Größe des Archives `testarchiv.tar` ausgeben.
Was fällt Ihnen auf?

g) Komprimieren Sie das Archiv:
`$> gzip testarchiv.tar`

Das Archiv ist nun erstellt. `gzip` hat das Archiv automatisch in `testarchiv.tar.gz` umbenannt.

h) Lassen Sie sich die Größe des gepackten Archives `testarchiv.tar.gz` ausgeben.

**Frage:** Es ist möglich, ein gepacktes Archiv (`.tar.gz`) mit einem Aufruf von `tar` zu erzeugen? Wie hätte dieser Aufruf lauten müssen?

i) Lassen Sie sich den Inhalt des gepackten Archives ausgeben.

4. *Kompilieren*

Machen Sie sich schlau, was ein Makefile ist. (Tipp: The GNU Make Manual)
Erstellen Sie ein Verzeichnis und schreiben sie in diesem Verzeichnis ein kleines C-Programm (z. B. `helloworld.c`). Erstellen Sie ebenfalls in dem Verzeichnis ein Makefile, so dass durch den Aufruf von `$> make` in diesem Verzeichnis Ihr C-Programm kompiliert wird (Tipp: `make`, `gcc`). Durch den Aufruf von `$> make clean` soll das Verzeichnis wieder in den Zustand von vor dem Kompilieren versetzt werden.

### 1.1 Slurm

Slurm ist ein Batchsystem (Portable Batch System - PBS), welches die verfügbaren (Rechen)-Ressourcen verwaltet und Jobs auf diese verteilt. Ein Benutzer übergibt Slurm ein Jobskript (PBS-Skript), welches Information enthält was gestartet soll und welche Ressourcen benötigt sind. Die verfügbaren Ressourcen werden unter allen Benutzern aufgeteilt und somit der einzelne Job warten, bis er Ressourcen zugeteilt bekommt. Sobald der Job ausgeführt wird, werden die Ausgaben in entsprechenden Dateien dokumentiert. Siehe auch: `http://wr.informatik.uni-hamburg.de/teaching/ressourcen/beginners_guide#job_managing`

1. Erstellen Sie ein Jobskript für 4 Prozesse.
2. Nutzen Sie den Befehl `mpiexec` um den Befehl `hostname` auf jedem der Knoten auszuführen.
3. Starten Sie das Jobskript mehrfach.
**Frage:** Was fällt Ihnen auf? Versuchen Sie Ihre Beobachtung(en) zu erklären!

## 2 Versionsverwaltung mit Git

Ein Versionsverwaltungssystem ist für die Entwicklung größerer Projekte unbedingt erforderlich. Für die Programmieraufgaben im Team sind sie ein Werkzeug, welches die parallele Entwicklung am Quellcode ermöglicht.

- Setzen Sie mit `git config --global` ihre Benutzerinformation.
- Schauen Sie mit `git help` die Hilfe an. Mit `git help <Kommando>` können Sie mehr über ein Kommando erfahren.
- Erstellen Sie ein Verzeichnis und initialisieren Sie ein Repository darin.
- Erstellen Sie eine Datei und fügen Sie diese zum Index hinzu.
- Committen Sie die Daten ins Repository.

- Ändern Sie den Dateiinhalt und committen Sie erneut.
- Betrachten Sie die Protokolle der Veränderungen, welchen Hash haben die Commits.

# 3 Debugging

Um später effizient programmieren zu können, wollen wir uns ein wenig mit der Fehlersuche in (parallelen) Programmen beschäftigen. Hierzu schauen wir uns den GNU Debugger `gdb` und den Speicherprüfer `memcheck` von der `valgrind`-Tool-Suite an. Diese können sowohl für sequentielle als auch für parallele Programme genutzt werden. In C ist die Speicherallokation sehr fehlerträchtig, `valgrind` hilft hierbei typische Fehler aufzuspüren.

## 3.1 GNU Debugger

Experimentieren Sie etwas mit GDB herum um den Debugger kennen zu lernen. Im folgenden Vorschläge um GDB kennen zu lernen.
Erstellen Sie ein einfaches Programm mit einer rekursiven Funktion z. B. Fakultät.
Kompilieren Sie das Programm mit Debug-Symbolen und starten Sie es mit GDB.

- Schauen sie sich den Befehl (help) an.
- Erstellen Sie einen Breakpoint auf die rekursive Funktion (break).
- Starten Sie das Programm (run).
- Geben Sie sich den Stack aus.
- Wechseln Sie den Stackframe auf die Main-Routine.
- Zeigen Sie sich den Code an.
- Wechseln Sie zur Funktion zurück.
- Schauen Sie sich einzelne Variablenwerte an (print).
- Welchen Typ haben die Variablen?

## 3.2 Valgrind

Schreiben Sie ein einfaches Programm welches ein Array aus Integern verwendet (z. B. 5 Elemente). Lassen Sie jeweils einen Wert des Arrays ausgeben und spielen Sie mit folgenden Konfigurationen:

- Der Wert wurde nicht initalisiert.
- Greifen Sie auf ein ungültiges Element zu z.b. das 6 Element (obwohl der Array nur aus 5 Elementen besteht).
- Schreiben Sie eine Funktion welche den Array (statisch) deklariert, initalisiert und dann den Array zurück gibt. Geben Sie dann in der Main-Funktion einen Wert des zurückgebenen Arrays aus.
- Passen Sie die Funktion an: Allokieren Sie Speicher mit `malloc` (in C) bzw. `allocate` (in Fortran). Starten Sie hier Valgrind mit "–leak-check".

Starten Sie die Programme mit Valgrind, was gibt ihr Programm aus, welchen Fehler meldet Valgrind?

# 4 Erste Schritte mit MPI

Hier wollen wir erste MPI Anwendungen in C oder Fortran entwickeln. Diese kleinen Testprogramme dienen dafür erste Erfahrungen zu sammeln und kleinere Aufgaben zu lösen. Für jede Aufgabe: erstellen Sie das dazu gehörige MPI Programm (in einer oder mehrerer Source-Code Dateien), ein Makefile welches eine ausführbare Datei erstellt und ein dazu passendes Jobskript um das Programm ausführen zu können.
Als Namensgebung für die Teilaufgaben erstellen Sie am besten einen Ordner: *Zettelnummer/Section_Subsection*. Somit ist uns auch immer klar wo ihr Code auf dem Cluster liegt, sofern es Rückfragen gibt.
*Hinweise (siehe auch Wiki):* Um mit MPI Anwendungen zu entwickeln müssen Sie zunächst mit `modules` das Modul für MPICH2 laden. Dafür verwenden Sie: `module load mpich2`.

## 4.1 Hello World

Alle Prozesse sollen "Hello World *Prozess* von *Prozesszahl*". Wobei die konkrete Prozessnummer (Rang) bzw. die gesamte Prozesszahl verwendet werden sollte.

## 4.2 Pre/Post-Processing

Oftmals wird zu Beginn einer Anwendung ein Preprocessing durchgeführt, diese Information wird von einem Master-Prozess (Rang 0) an alle Prozesse weitergeben. Am Ende des Programmes werden die Informationen wieder von dem Master eingesammelt und ausgegeben.
Schreiben Sie ein Programm, welches von der Kommandozeile beliebig viele Parameter im Zahlenformat annimmt. Diese Zahlen sollen fair auf die vorhanden Prozesse aufgeteilt werden (kein Prozess muss 2 Zahlen mehr berechnen als ein anderer, maximal 1 Zahl). Die Prozesse addieren dann ihre zugeteilten Zahlen in einer Schleife. Der Master-Prozess sammelt dann die Zahlen ein, gibt die einzelnen Summen aus, summiert alle Zahlen und gibt diese Gesamtsumme aus. Verwenden Sie zunächst nur die MPI Funktionen "Send" und "Recv" (und natürlich "Init" bzw. "Finalize").

## 4.3 Kollektive Operationen

Ersetzen Sie im vorangegangen Beispiel (Pre/Post-Processing) alle Sende und Empfangsoperationen durch die MPI Funktionen "Scatterv" und "Gather". Nun sollten Sie keine "Send" und "Recv" Operationen mehr benötigen!

## 4.4 Aggregationen

Ersetzen Sie im vorangegangen Beispiel (Pre/Post-Processing) das Aufsammeln aller Zahlen im Master-Prozess durch die "Reduce" Funktion. Welche Vereinfachung könnte man durch das Nutzen der "Reduce" Funktion noch vornehmen?

## 4.5 Datentypen

Erstellen Sie einen abgeleiteten Datentyp für eine Struktur bestehend aus 3 Integern und einem String aus 20 Zeichen. Initalisieren Sie die Struktur mit Werten und Senden Sie die Werte zu einem zweiten Prozess. Lassen Sie sich auf beiden Prozessen die Werte zur Sicherheit ausgeben.

## 4.6 Datenaustausch

Tauschen Sie zwischen zwei Prozessen gegenseitig mindestens 1 MByte an Daten aus (erstellen Sie z. B. einen Array mit 1 Million Integern). Jeder Prozess sollte nach dem `MPI_Finalize` noch eine Nachricht ausgeben, dass er beendet hat.
Der erste Prozess soll die Daten an den zweiten Prozess schicken und umgekehrt. Verwenden Sie nur "Send" und "Recv" (und natürlich "Init" bzw. "Finalize"). Drehen Sie die Reihenfolge der Send und Recv der einzelnen Prozesse um. Welches Problem kann auftreten. "WARNUNG": Verwenden Sie eine niedrige Wallclock Zeit im Jobskript.

## 4.7 Matrix-Multiplikation

In dieser Aufgabe wollen wir eine Matrix Multiplikation mit einem Skalar durchführen.
Schreiben Sie ein Programm, welches von einer Textdatei eine Matrix einliest. Von einer zweiten Textdatei soll ein Skalar eingelesen werden. Die Dateien sollen per Kommandozeile übergeben werden.
Die Matrix wird auf die einzelnen Prozesse aufgeteilt und jeder Prozess führt die Multiplikationen durch.
Die Ausgabe soll eine dritte Datei geschrieben werden.
Vielleicht ist der Speicher eines Knotens nicht groß genug die gesamte Matrix zu halten, daher stellen Sie sicher dass jeder Prozess nur die benötigten Daten im Speicher hält.
Verwenden Sie POSIX-Funktionen ("open()", "read()", "write()") um die Datei zu lesen bzw. zu schreiben.
Prüfen Sie die Korrektheit ihrer Lösung!
Beispiele für die Eingabedateien finden Sie auf der Webseite. Testen Sie verschiedene Prozesszahlen z. B. "1, 2, 4, 8" mit den bereitgestellten Eingabedateien und lassen Sie sich die maximale (und minimale) Laufzeiten über alle Prozesse ausgeben (messen Sie zwischen "Init" und "Finalize"). Verwenden Sie hierfür die Funktion `clock_gettime` (verwenden Sie `man clock_gettime()` bzw. in Fortran (ab 95) `cpu_time()`). Um die maximale (und minimale) Laufzeiten eines Prozesses zu ermitteln bietet sich ein MPI Befehl an, welchen Sie bereits kennen gelernt haben. Wie erreichen Sie dass Sie nur einen MPI Aufruf benötigen um die minimale und maximale Zeit zu erhalten? Wie verhalten sich hier die Laufzeiten in Abhänigkeit zur Prozessanzahl?

## 4.8 Matrix-Matrix-Multiplikation

Erweitern Sie das Programm mit der Skalaren Multiplikation, so dass eine Matrix-Matrix Multiplikation erfolgt. In dem Beispiel sollten beide Matrizen gleichmäßig auf beide Prozesse aufgeteilt werden. Versuchen Sie die Kommunikation zwischen den Prozessen gering zu halten! Achten Sie auch auf den Speicherbedarf (kein Prozess sollte eine Matrix ganz im Speicher halten).
Dokumentieren Sie auch hier die Zeiten die Sie mit den bereitgestellten Eingabedateien erzielen. Wie verhalten sich hier die Laufzeiten in Abhänigkeit zur Prozessanzahl?

## 4.9 Ergebnisse vergleichen

Notieren Sie die erzielten Laufzeiten und Prozesszahlen in Sekunden, mit 3 Nachkommastellen in einer Textdatei, schicken Sie sie in einer E-Mail an: `kunkel@dkrz.de`.