Technische Universität München
Institut für Informatik
Prof. Dr. Angelika Steger

SS 2003
Merkblatt 2
12.05.2003

Praktikum Diskrete Optimierung

C++ Programmierung unter Linux - Anmerkungen

Das folgende Merkblatt soll (hauptsächlich stichpunktartig) einen kleinen Überblick über verschiedene Tools geben, die das Programmierer-Leben (nicht nur unter Linux) vereinfachen. Zusätzlich soll dieses Merkblatt aber auch Anregung sein, bei Problemen selbst nach geeigneten Tools zu suchen.
Die vorgestellte Liste erhebt weder einen Anspruch auf Vollständigkeit noch einen Anspruch darauf, die besten Tools für den jeweiligen Zweck vorzustellen – oftmals ist das eher eine Geschmacksfrage. Die hier vorgestellten Tools haben wir jedoch zum größten Teil bereits selbst eingesetzt.

1 Der Editor - eine Geschmacksfrage

Die häufigste Verwendung finden hier `vi(m)` und `(x)emacs`. Welche von beiden Varianten man selbst lieber verwendet, ist eigentlich nur eine Geschmacksfrage.

- **vi(m):** Projekthomepage: `http://www.vim.org/`
- **(x)emacs:**
  Projekthomepage emacs: `http://www.gnu.org/software/emacs/`
  Projekthomepage xemacs: `http://www.xemacs.org/`
- **Weitere Editoren** Zusätzlich zu den beiden "Großen" gibt es auch eine Reihe von Editoren, die insbesondere für Windows-Benutzer einen kleineren Einarbeitungsaufwand erfordern, aber bei weitem nicht die Fähigkeiten von `vi(m)` bzw. `(x)emacs` erreichen. Zu nennen wären:
  - **pico**
  - **nedit**

2 Kompilieren: Makefiles und Compiler

2.1 Makefiles

Dokumenation zu `make` finden Sie unter `http://www.gnu.org/manual/make/`.
Ein sehr, sehr einfaches Beispiel-Makefile könnte wie folgt aussehen:

```
INCLUDE  = -I. -I/usr/local/include/
CXXFLAGS = -Wall
LDFLAGS  =
LDLIBS   =
```

```
# MAKE wird von GNU make (und von BSD make) selbst gesetzt
CC = c++
CXX = c++
CPP = c++

all:
    $(CXX) $(CXXFLAGS) -o a<Nummer>.o a<Nummer>.cpp
    $(CXX) $(CXXFLAGS) $(LDFLAGS) -o a<Nummer> a<Nummer>.o $(LDLIBS)
```

2.2 Compiler

Im Rahmen des Praktikums werden wir ausschließlich den g++ aus der GNU Compiler Collection (gcc) verwenden. Die Projekthomepage der gcc ist unter `http://gcc.gnu.org/` zu finden.
Wichtige Compiler-Flags:

- `-g` bzw. `-g<level>`: mit Debug-Informationen compilieren (wichtig, falls die im Folgenden angesprochenen Tools verwendet werden)
- `-Wall`: Auszug aus der man-page: This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros.
- `-O<level>`: Optimierung einschalten (*Vorsicht:* Sollte nur bei bereits getesteten Programmen aktiviert werden, Bugs in der g++-Optimierung sind – insbesondere bei Version 2.95.x – nicht auszuschließen).

Folgendes ist bei der Abgabe zu beachten:

- Es gibt erhebliche **Versionsunterschiede** zwischen den beiden Compiler-Versionen `g++ 2.95.x` und `g++ 3.2.x`, insbesondere was die Einhaltung des C++-Standards betrifft. Programme, die mit einem `g++ 2.95.x` kompiliert werden können, werden sich deshalb meist nicht mit dem (standardkonformeren) `g++ 3.2.x` übersetzen lassen.
- Ebenso gilt: Programme, die sich mit anderen Compilern übersetzen lassen, werden sich in vielen Fällen nicht mit einem `g++ 3.2.x` kompilieren lassen. Grund ist oft die fehlende Standardkonformität der anderen Compiler (dies gilt insbesondere auch für den MS C++-Compiler!).

3 Fehler finden: Debugger, Memory Debugger, Tools und andere Möglichkeiten

3.1 cout, cerr - Standard Debugger

- Zum Debuggen *immer* `cerr` und niemals `cout` verwenden. Gründe:

  - cerr ist im Gegensatz zu cout ungepuffert. Deshalb wurden auch im Falle eines Programmabsturzes alle anstehenden Ausgaben schon durchgeführt (das ist bei cout *nicht* garantiert) und
  - Die Ausgaben des Programms auf stdout können von den Debug-Ausgaben auf stderr leicht getrennt werden.

- cerr-Debugging ist generell nur zu empfehlen, wenn man den Fehler bereits einigermaßen gut eingegrenzt hat bzw. eingrenzen kann.

3.2 gdb - GNU Project Debugger

- Projekthomepage: http://www.gnu.org/software/gdb/gdb.html
- Notwendige Änderungen im Makefile:
  - Programm muss mit Debug-Informationen kompiliert sein (CXXFLAGS += -g bzw. CXXFLAGS += -g3)
- Benutzung: als Kommandozeilen-Version …
  - Start mit gdb <Programmname>
  - Ausführen mit run <Programmparameter>
  - weitere Befehle: next (n), print (p), break <Souce:Line>, cont (c), …

…oder einfacher als Modul im (x)emacs:

  - Start mit Meta-X gdb
  - dann Kommandos wie oben oder Befehlsschaltflächen

3.3 DDD - Data Display Debugger

- Projekthomepage: http://www.gnu.org/software/ddd/
- Notwendige Änderungen im Makefile:
  - Programm muss mit Debug-Informationen kompiliert sein (CXXFLAGS += -g bzw. CXXFLAGS += -g3)
- Benutzung: grafische Oberfläche, mehr oder weniger selbsterklärend

3.4 Valgrind - Memory Debugger

- Projekthomepage: http://developer.kde.org/~sewardj/
- Notwendige Änderungen im Makefile:
  - Programm sollte mit Debug-Informationen kompiliert sein (CXXFLAGS += -g bzw. CXXFLAGS += -g3), ansonsten sind die Ausgaben sehr kryptisch

- Benutzung: `valgrind --skin=memcheck <Programmname Programmparameter>`
- Ergebnisse auswerten: Programm schreibt Status/Fehlerreport entweder als logfile oder auf stdout.
- Funktionsweise: Programm läuft auf simulierter CPU, also auch entsprechend langsam.

3.5 MPATROL - Memory Debugger

- Projekthomepage: `http://www.cbmamiga.demon.co.uk/mpatrol/`
- Notwendige Änderungen im Makefile:
  - Programm sollte mit Debug-Informationen kompiliert sein (`CXXFLAGS += -g` bzw. `CXXFLAGS += -g3`), ansonsten sind die Ausgaben sehr kryptisch
  - Zusätzlich linken der MPATROL-Bibliotheken: `LDLIBS += -lmpatrol -lbfd -liberty`
- Notwendige Änderungen im Programm:

  ```
  #include <mpatrol.h>
  ```

  in allen zu bearbeitenden Quellcodedateien.
- Notwendige Änderungen in den Umgebungsvariablen: s. Dokumentation, z.B.

  ```
  export MPATROL_OPTIONS="showall checkall logall usedebug
  oflowsize=8192 pagealloc=upper logfile=/tmp/mpatrol.log LEAKTABLE"
  ```

- Benutzung: bei gesetzten Optionen entsprechend kompiliertes Programm starten
- Ergebnisse auswerten: Programm schreibt Status/Fehlerreport als logfile.
- Funktionsweise: MAPTROL ersetzt die Speicherverwaltungsroutinen durch eigene Funktionen mit Fehlererkennung, das Programm läuft daher sehr langsam. In seltenen (Ausnahme-)Fällen werden Fehler im Programm leider nicht erkannt.

3.6 Weitere nützliche Tools

Hier soll nur eine kleine Liste von Tools angegeben werden, die beim Debuggen helfen können. Näheres zu den Tools findet sich in den `man`-Pages oder im Web.

- `sort`: sortiert Textzeilen in Dateien
- `diff`: zeigt Unterschiede zwischen Dateien an

- `cat`: Ausgabe von Dateien, Ausgabe kann mit Hilfe von `pipes` weitergeleitet werden (gilt auch für alle anderen Tools), z.B.

  ```
  cat sample.in | ./yourSolution
  ```

- ...

3.7 Immer noch "wrong answer" bzw. "runtime error"?

Neben Programmierfehlern gibt es noch eine Reihe weiterer Stolpersteine, die (zumindest bisher :)) nicht mit Hilfe von Tools erkannt werden können. Hierzu gehören:

- **Sonderfälle:** Zunächst sollte man sich fragen, ob der implementierte Algorithmus alle möglichen Sonder- und Randfälle korrekt abarbeitet, oder ob diese evtl. einzeln abgeprüft werden müssen. Bei Graphenalgorithmen sind solche Sonderfälle z.B. der leere Graph (keine Knoten, keine Kanten), Graphen ohne Kanten (aber mit n Knoten) und manchmal auch besondere Graphklassen.

  Solche Sonderfälle sind *keine fehlerhaften Eingaben*, sondern müssen von einer korrekten Routine abgearbeitet werden können. Im Praktikum haben wir bisher darauf verzichtet, solche Randfälle abzuprüfen, aber eine Ausgabe "Fehlerhafte Eingabe" bei einem leeren Graphen oder bei einem Graphen ohne Kanten ist im allgemeinen *nicht korrekt*.

- **Implizite Annahmen:** Werden die Randfälle korrekt abgearbeitet, sollte man sich Gedanken über mögliche implizit getroffene Annahmen machen, die so nicht in der Aufgabenspezifikation gegeben sind. Solche (fehlerhaften) Annahmen können im Bereich von Graphenalgorithmen z.B. sein: Wird von einer bestimmten Nummerierung der Knoten ausgegangen? Wird von einer bestimmten Eingabereihenfolge der Kanten ausgegangen? Wird von bestimmten Grapheigenschaften ausgegangen, die nicht vorhanden sein müssen (Zusammenhangskomponenten, Planarität, Zugehörigkeit zu bestimmten Graphklassen)?

- **Korrekter Algorithmus?** Kann man die Fehler trotz allem nicht beheben, sollte man sich nochmals Gedanken darüber machen, ob man für das gestellte Problem wirklich den korrekten Algorithmus verwendet hat. Gute Literatur zum Thema hilft hier oft weiter.

4 Korrekter Algorithmus, aber zu langsam? – Profiling

4.1 TAU - Tuning and Analysis Utilities

- Projekthomepage (Dokumentation, Download):
  `http://www.cs.uoregon.edu/research/paracomp/tau/tautools/`

- Installation: s. Dokumentation
- Notwendige Änderungen im Makefile:

```
TAUROOTDIR = <your_tau_installation_directory>
include $(TAUROOTDIR)/i386_linux/lib/Makefile.tau-pthread
CXX = $(TAU_CXX)
CXXFLAGS += $(TAU_INCLUDE)
CXXFLAGS += $(TAU_DEFS)
LDLIBS += -lm $(TAU_LIBS)
LDFLAGS += $(USER_OPT)
```

- Notwendige Änderungen im Programm:
  - am Anfang jeder zu profilenden Routine:

```
TAU_PROFILE(function_name, type, group);

Example:
int main(int argc, char **argv)
{
  TAU_PROFILE(main(),int (int, char **),TAU_DEFAULT);
  ...
```

  - einzelne Timer (um Statements, Teile einer Funktion etc. zu profilen):

```
TAU_PROFILE_TIMER(timer, name, type, group);
TAU_PROFILE_START(timer);
TAU_PROFILE_STOP(timer);


Example:
template< class T, unsigned Dim >
void BareField<T,Dim>::fillGuardCells(bool reallyFill)
{
  // profiling macros
  TAU_TYPE_STRING(taustr, CT(*this) + ^\ void (bool)^] );
  TAU_PROFILE(^\BareField::fillGuardCells()^], taustr,
  TAU_FIELD);
  TAU_PROFILE_TIMER(sendtimer, ^\fillGuardCells-send^],
  taustr, TAU_FIELD);
  TAU_PROFILE_TIMER(localstimer, ^\fillGuardCellslocals^],
  taustr, TAU_FIELD);
  // ...
  TAU_PROFILE_START(sendtimer);
  // set up messages to be sent
  Message** mess = new Message*[nprocs];
```

```
int iproc;
for (iproc=0; iproc<nprocs; ++iproc) {
  mess[iproc] = NULL;
  recvmsg[iproc] = false; }//... other code
TAU_PROFILE_STOP(sendtimer);
...
```

- Ergebnisse auswerten: Das Programm muss sich ordnungsgemäß beenden (*kein Segmentation Fault!*). Dann findet man im Ausführungsverzeichnis eine Datei mit Profiling Informationen einerseits zu den profilten Routinen, andererseits zu den angegebenen Timern. Diese Datei kann dann mit Hilfe von tau-racy (Tcl/Tk-GUI) oder mit jracy (Java-GUI) visualisiert werden.

4.2 gprof - GNU Project Profiler

- Dokumentation:
`http://www.gnu.org/manual/gprof-2.9.1/html_mono/gprof.html`

- Notwendige Änderungen im Makefile:

  - Programm muss mit Debug-Informationen kompiliert sein (`CXXFLAGS += -g` bzw. `CXXFLAGS += -g3`)
  - `CXXFLAGS += -pg`
  - `LDFLAGS += pg`

- Benutzung: `gprof option [executable-file [profile-data-files]] [> outfile]`

- Ergebnisse auswerten: Das Programm muss sich ordnungsgemäß beenden (*kein Segmentation Fault!*). Dann findet man im Ausführungsverzeichnis eine Datei `gmon.out` mit Profiling Informationen.