Sequenzielles debugging

Paralleles Programmieren für Geowissenschaftler

Michael Blesel, Hermann Lenhart

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

2. Mai 2019

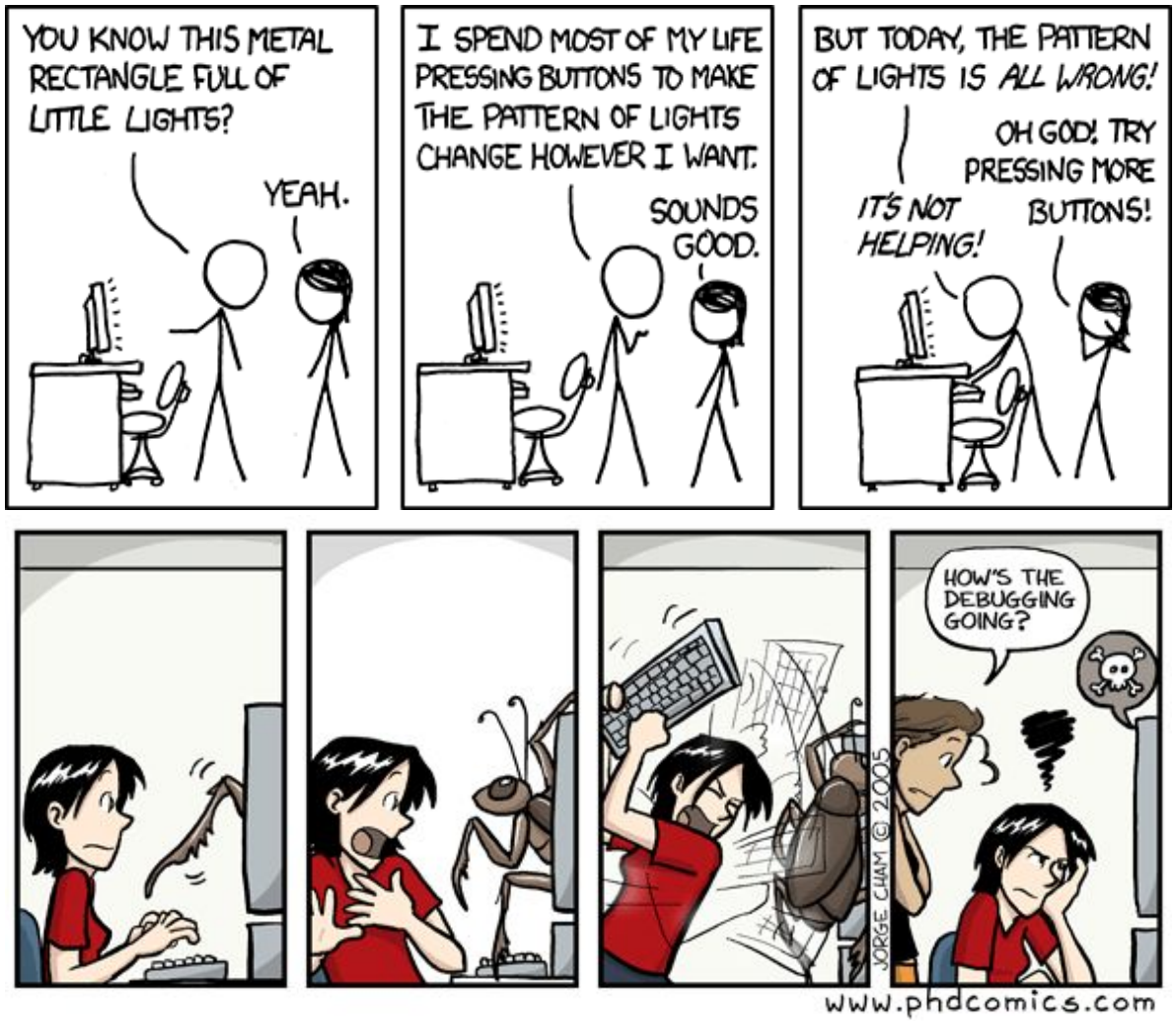

Abbildung: XKCD, xkcd.com / PHD Comics (2006), phdcomics.com

Warum ist debugging wichtig?

> *[...] the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs.*
>
> - *Maurice Wilkes*

- Das Finden und Beheben von Fehlern ist eine der wichtigsten Kompetenzen für Programmierer.
- Oft arbeitet man an fremdem Code den man nicht komplett kennt oder versteht.
- Effizientes debugging erspart viel Zeit und Nerven.

Guidelines für einfacher debugbaren Code

- Sprechende Bezeichner verwenden
- Probleme aufteilen
  - Mache eine simple Sache zur Zeit
    $\Rightarrow$ Funktionen/Subroutinen sind kurz
    $\Rightarrow$ Module/Dateien sind kurz
- Code nicht komplizierter machen als notwendig
  - Code wird öfter gelesen als geschrieben
  - Aussagekräftige Kommentare über die Semantik und den Effekt von Code Segmenten

Guidelines für einfacher debugbaren Code

- Keine globalen Variablen
  - Der Wert von Variablen hängt maximal von einer Datei ab
  - Alle Daten werden explizit übergeben
  - Zusammengehörige Daten zu Typen zusammenfassen
- Keine ‘magic numbers’
  - Keine hartcodierten Array-/Bufferlängen
  - Man definiere entsprechend benannte Parameter für Konstante Werte
- Kurz gesagt: Man folge einem vernünftigen Styleguide :)

Debugging Methoden

- Code anschauen und im Kopf durchgehen
  - Fehleranfällig und nur begrenzt machbar
- Write/printf debugging
  - Problematisch bei parallelen Anwendungen
- Man benutzt debugging Tools (gdb, valgrind, etc.)

Debugger

- Debugger erlauben es die Ausführung eines Programms zu beobachten
- Der Kontrollfluss innerhalb des Programms kann direkt verfolgt werden
- Die Ausführung kann beliebig unterbrochen und fortgesetzt werden
- Werte von Variablen können beobachtet und beeinflusst werden

GDB: The Gnu Project Debugger

- Um ein Programm mit gdb zu starten muss es mit Debugsymbolen gebaut werden
- Hierfür das `-ggdb` flag beim Kompilieren benutzen

```
CC=f95
FFLAGS=-ggdb

main : main.o
    $(CC) $(FFLAGS) $< -o $@
```

```
]$ f95 -ggdb main.f90 -o main
```

Abbildung: Kompilieren mit Debugsymbolen

Grundlegendes

- Programm mit gdb starten: `gdb <programm>`
  - Mit Argumenten: `gdb --args <programm> [argumente]`
- Starten: `run`, unterbrechen: `<CTRL>c`
  - Starten und direkt pausieren: `start`
- Weiterlaufen: **`continue`**
- Step in: `step`, Step over: `next`
- Variablenwert ausgeben: `print <Variable>`
- Code anzeigen: `list`, TUI: `<CTRL>x + a`
- Breakpoint setzen: **`break <Zeilennr. | Funktionsname>`**
- Watchpoint setzen: `watch <Variable>`
- gdb beenden: `quit`

Breakpoints und Watchpoints

- Breakpoints können auf Codezeilen oder Funktionen gesetzt werden und unterbrechen das Programm wenn der Breakpoint erreicht wird
- Watchpoints können auf Variablen gesetzt werden und unterbrechen das Programm wenn mit der Variable etwas passiert

Stackframe (backtrace)

- Mit dem `backtrace` Befehl kann man die momentane Aufrufkette von Funktionen oder Subroutinen sehen
- Es gibt eine Liste an frames aus. Beginnend mit frame 0 (aktuelles frame) gefolgt von den jeweiligen Callern

```
Temporary breakpoint 1, MAIN__ () at main.f90:1
1       program main
(gdb) break mod_lifecycle::countneighbors
Breakpoint 2 at 0x555555556d09: file lifecycle.f90, line 7.
(gdb) c
Continuing.
```

```
Breakpoint 2, mod_lifecycle::countneighbors (field=..., neighbors=...) at lifecycle.f90:7
7           subroutine countNeighbors(field, neighbors)
(gdb) backtrace
#0  mod_lifecycle::countneighbors (field=..., neighbors=...) at lifecycle.f90:7
#1  0x0000555555556901 in mod_lifecycle::developlife (field=...) at lifecycle.f90:32
#2  0x00005555555553fd in MAIN__ () at main.f90:20
```

Was Debugger nicht können

- Freigabe von Speicher überprüfen
- Arrayindizes prüfen
- Überprüfen ob zugegriffener Speicher noch alloziert ist

- Dafür gibt es valgrind

valgrind

- valgrind simuliert einen Prozessor, der diese Checks durchführt und protokolliert
  - Langsam!
- Hat noch viele andere Funktionen (z.B. Cache-Hit-Analyse)

- Aufruf von valgrind: `valgrind <programm> [Argumente]`
- Anzeigen, wo verlorene Blöcke allokiert wurden:
  `--leak-check=full`
- Auswahl anderer Tools als memcheck (z.B. cachegrind):
  `--tool=cachegrind`
- Programm muss wie bei gdb mit `-g` bzw. `-ggdb` kompiliert werden

```
[michael@X280 debug2]$ valgrind --leak-check=full ./main
==5260== Memcheck, a memory error detector
==5260== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5260== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==5260== Command: ./main
==5260==
==5260==
==5260== HEAP SUMMARY:
==5260==     in use at exit: 400 bytes in 1 blocks
==5260==   total heap usage: 31 allocs, 30 frees, 17,520 bytes allocated
==5260==
==5260== 400 bytes in 1 blocks are definitely lost in loss record 1 of 1
==5260==    at 0x483877F: malloc (vg_replace_malloc.c:299)
==5260==    by 0x10932E: __mod_matrix_MOD_do_matrix_stuff (mod_matrix.f90:8)
==5260==    by 0x10920D: MAIN__ (main.f90:10)
==5260==    by 0x109292: main (main.f90:2)
==5260==
==5260== LEAK SUMMARY:
==5260==    definitely lost: 400 bytes in 1 blocks
==5260==    indirectly lost: 0 bytes in 0 blocks
==5260==      possibly lost: 0 bytes in 0 blocks
==5260==    still reachable: 0 bytes in 0 blocks
==5260==         suppressed: 0 bytes in 0 blocks
==5260==
==5260== For counts of detected and suppressed errors, rerun with: -v
==5260== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Fragen?

FRAGEN ?