Universität Hamburg
Fachbereich Informatik
Wissenschaftliches Rechnen
Prof. T. Ludwig, H. Lenhart, Enno Zickler

Übungsblatt 1 zum Praktikum
Paralleles Programmieren für
Geowissenschaftler im SS 2016
Abgabe: 09.05.2016

Aufgabe 2: Erstes strukturiertes FORTRAN Programm und Debugging Übungsaufgaben

Dieses Übungsblatt umfasst die Aufgaben eines ersten FORTRAN Programms und Aufgaben zum Debugging.

Zum Debugging werden auf der Webseite Programme bereitgestellt, die mit **gdb** oder **Valgrind** auf Fehler untersucht werden sollen. Die Korrektur soll so erfolgen, dass fehlerhafte Zeilen/Blöcke auskommentiert werden und die korrigierten Zeilen/Blöcke in unmittelbarer Nähe stehen. Zusätzliche Kommentarzeilen sollten die Korrektur leicht nachvollziehbar machen. (Ausnahme: Game of Life)

Wie gehabt, sollten Probleme auftauchen, wendet Euch bitte an die Mailingliste:

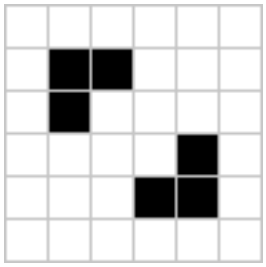
`PPG-16@wr.informatik.uni-hamburg.de`

Aufgabe 1A: Game of Life (120 Punkte)

In der erste Aufgabe soll das Konzept von Conway's "Game of Life" in einem Programm umgesetzt werden. Für nähere Informationen zum Game of Life, siehe

`http://en.wikipedia.org/wiki/Conway's_Game_of_Life`

Hierbei sollen die "Lebenszyklen" der verschiedene Muster mit 10 Iterationen durchlaufen werden.

 Blinker  Toad  Beacon

Dazu soll eine Matrix vom Typ Logic erstellt werden, die als Initialisierung mit 'false' belegt wird. Danach werden für die Muster der Figuren Blinker, Toad und Beacon (s.o.) ensprechend der schwarzen Kästchen Werte in der Matrix mit dem Wert 'true' belegt. Die Berechnung soll auf einem Feld der Größe 30 X 20 erfolgen.

Zur Entwicklung des Lebenszyklus in der jeweils nächsten Iteration werden folgende Abfragen gestellt:

1. Hat eine lebende Zelle (schwarzes Kästen, in der Matrix mit 'true' belegt) weniger als 2 Nachbarn, so stirbt sie an Unterbevölkerung. D.h. sie wird in der nächsten Iteration auf 'false' gesetzt.
2. Hat eine lebende Zelle genau 2 oder 3 Nachbarn, so überlebt sie in der Iteration. D.h. der Wert 'true' bleibt erhalten.
3. Hat eine lebende Zelle mehr als 3 Nachbarn, so stirbt sie an Überbevölkerung.
4. Hat eine tote Zelle (weißes Kästen, in der Matrix mit 'false' belegt) genau 3 Nachbarn, so wird diese zu einer lebenden Zelle. D.h. sie wird in der nächsten Iteration auf 'true' gesetzt. Ansonsten bleibt die Zelle tot.

Es sollen die "Lebenszyklen" der verschiedene Muster mit 10 Iterationen durchlaufen und die Muster ensprechend dargestellt werden. Das Programm soll mit einem Makefile gesteuert werden und im Kopf des Programmes hätten wir gerne eine kurze Erklärung der wichtigsten Elemente des Programmes.

Definiert dazu zunächst die Subroutinen mit den Namen *createField* und *createFigures*, die für die Initialisierung des Spielfelds mitsamt der drei Figuren zuständig sind. Diese sollen im Modul *mod_initializeField* zusammengefasst werden.

Für den Durchlauf der Iterationen erstellt das Modul *mod_lifecycle* mit den beiden Subroutinen *developLife* und *countNeighbors*. Die Subroutine *developLife* soll dabei *countNeighbors* aufrufen.

Für die Ausgabe stellen wir das Programm Glider Movie auf unserer Webseite zur Verfügung, dessen Ausgangsroutine Unicode-Zeichen verwendet um eine gute lesbare Darstellung zu erreichen.

Ihr dürft beleibige Teile von Glider Movie wiederverwenden,
die **Subroutine printTwoDLogical** müsst Ihr verwenden.

Die Subroutine *printTwoDLogical* soll vom Hauptprogramm aus aufgerufen werden. Das Ziel hierbei ist ein übersichtliches Hauptprogramm, in dem nur vier Aufrufe erfolgen: *createField*, *createFigures* , *developLife* und *printTwoDLogical* .

Aufgabe 2B: Debuggen von Sortieralgorithmen (120 Punkte + 120 Bonuspunkte)

Das Testprogramm sorter (runterzuladen auf der Website) implementiert einige unterschiedliche Sortieralgorithmen, enthält aber noch mehr Bugs als Sortieralgorithmen. Diese gilt es zu finden. Da der Aufwand zum Debuggen extrem variiert, müsst ihr für die volle Punktzahl nicht alle Bugs dingfest machen, welche, das ist euch überlassen. Ihr könnt das Programm natürlich auch vollständig debuggen und zusätzliche Punkte einstreichen.

Achtet darauf, wirklich zu verstehen was falsch läuft, bevor Ihr versucht das Problem zu beheben. Denn wenn Ihr nur die Symptome "fixt", werdet Ihr keine volle Punktzahl bekommen. In jedem Fall wollen wir für jeden gefundenen Bug eine angemessene Erklärung sehen, was der Bug eigentlich war.

Einfach (20 Punkte): Sorgt dafür, dass Valgrind keine verlorenen Blöcke mehr auflistet. Die beiden Fehlermeldungen "Invalid read of size 4" und "Process terminating with default action of signal 11 (SIGSEGV)", die Valgrind ausgibt, gehören zu anderen Fehlern und sind für diesen Aufgabenteil unerheblich. Entscheident ist die Ausgabe unter "LEAK SUMMARY" ganz am Ende des Valgrind Outputs.

Einfach (20 Punkte): Bubble sort ist der einfachste aller Sortieralgorithmen. In jedem Schritt geht er einmal durch das gesamte Array und vertauscht benachbarte Werte, wenn der Größere vor dem Kleineren steht. Das macht er so oft, wie das Array lang ist, so dass garantiert am Ende jeder Wert auf seinem Platz steht. Leider ist Valgrind mit unserer Implementierung nicht zufrieden. Findet heraus, woran das liegt und korrigiert den Fehler.

Mittel (40 Punkte): Die Ausgabefunktion ist so aufgebaut, dass sie nicht nur den jeweiligen Zahlenwert ausgibt, sondern zusätzlich noch einen Balken aus Minuszeichen aufbaut, dessen Länge proportional zum Zahlenwert ist. Leider wird bei den großen Zahlen überhaupt kein Balken mehr ausgegeben. Sorgt dafür, dass Balken mit mehr als 21 Zeichen angezeigt werden können.

Mittel (40 Punkte): Slow sort ist, wie der Name schon sagt, ein Sortieralgorithmus, dessen Ziel es ist, möglichst viel Zeit zu vertun. Das mit dem Zeitvergeuden klappt auch schon ganz hervorragend (mit 200 Elementen im Array benötigt Slow sort bereits etliche Sekunden), leider produziert es völligen Datenmüll dabei. Sorgt dafür, dass auch nach dem Slow sort Durchlauf nur Werte im Array stehen, die auch ursprünglich dringestanden haben. Und erklärt, warum der Compiler den Fehler nicht bemerkt hat.

Schwer (60 Punkte): Bucket sort ist ein Sortieralgorithmus, bei dem die Werte in einem Durchgang in eine Anzahl von Eimern mit unterschiedlichen Wertebereichen sortiert werden. Am Ende eines Sortierschrittes können sich also beliebig viele verschiedene Werte in den verschiedenen Eimern befinden. Wenn also alle Werte von Eimer 0 vor den Werten von Eimer 1 stehen, ist bereits eine grobe Sortierung erreicht. Diese Implementation sortiert die Daten vollständig, indem sie alle Eimer, in denen etwas drin ist, anschließend mit dem selben Verfahren durchsortiert. Leider ist sie nicht ganz korrekt, was zum Absturz des Programmes führt. Benutzt gdb, um herauszufinden, warum es zu dem fehlerhaften Speicherzugriff kommt. Tips: Wie wird der Index berechnet, und welche Werte haben die Variablen, die an seiner Berechnung beteiligt sind? Und wie kommt es zu den vielen ähnlichen Zeilen in der Ausgabe von gdb's `bt` Befehl?

Schwer (60 Punkte): Insertion sort sortiert die Zahlen, wie ein Kartenspieler seine Karten sortiert: Er fängt mit einer Karte in der Hand an, und fügt jeweils die nächste Karte an der richtigen Stelle ein, bis er alle Karten auf der Hand hat. Insertion sort geht ähnlich vor: Es geht mit einem Zähler i von links nach rechts durch das zu sortierende Array durch. Bevor i hochgezählt wird, ist der Subarray bis zur Stelle i sortiert. Nach dem Hochzählen ist der Wert an der Stelle i im Normalfall nicht an der richtigen Stelle, er wird aus dem Array herausgenommen. Anschließend werden alle Werte links von i, die größer als der einzusortierende Wert sind, um eine Position nach rechts verschoben. Am Ende wird der einzusortierende Wert an die freigewordene Stelle geschrieben. Leider produziert die Implementation in sorter.f95 am Anfang des Arrays eine mehr oder weniger lange Sequenz von gleichen Werten, die so in den Eingabedaten nicht vorhanden war. Sorgt dafür, dass das nicht passiert.

Abgabe

Die auf dem Cluster lauffähigen FORTRAN Programme sollen bis zum **Montag den 9.5.2016** geschickt werden an:

ppg-abgabe@wr.informatik.uni-hamburg.de

dabei ist zu beachten:

1. **NUR den Quellcode** schicken,
2. für **jede Aufgabe ein separates Verzeichnis anlegen** und
3. alles **als komprimiertes Archiv .tgz oder zip** schicken! D.h. es soll wirklich nur **ein einzelnes Archiv** geschickt werden!

Als Subject im Kopf der Mail bitte die Angabe: PPG-16 Blatt2 und die Liste der Familiennamen der Personen in der Übungsgruppe.