Aufgabe Profiling

Für Windows müssen Sie `a.exe` statt `a.out` angeben.

Teilaufgabe Gcov

Laden Sie sich die Quelltextdatei `taylor.cc` herunter. Sie enthält die näherungsweise Berechnung von e^x mit Hilfe einer Taylorreihenentwicklung. Es werden zwei zusätzliche Funktionen dazu benutzt.

Übersetzen Sie das Programm für die Auswertung mit `gcov` (die Warnungen können Sie ignorieren), führen es anschließend aus und erzeugen den annotierten Quelltext:

```
g++ -Wall -pedantic -fprofile-arcs -ftest-coverage taylor.cc
./a.out
gcov taylor.cc
```

Betrachten Sie den annotierten Quelltext in `taylor.cc.gcov` und suchen Sie die Hotspots und identifizieren Sie toten Code.

Teilaufgabe Gprof

Übersetzen Sie das Programm erneut, diesmal für die Analyse mit Gprof, führen das Programm anschließend aus und erzeugen den Analysebericht, den sie am besten in eine Datei umlenken:

```
g++ -Wall -pedantic -pg taylor.cc
./a.out
gprof > taylor.gprof
```

Unter Windows muss ggf. bei gprof zusätzlich das Executable angegeben werden: `gprof ./a.exe > taylor.gprof`.

Falls die Datei taylor.gprof keine Analysedaten am Anfang enthält, dann fügen sie zum Compileraufruf die Option `-no-pie` hinzu (workaround für einen Bug bei manchen GCC-Versionen).

Notieren Sie sich die kumulierte verbrauchte Zeit für `e_hoch_x`.

Sind diese Zeiten akkurat? Stimmen die angegebenen Ausführungszeiten wirklich mit der real vergangenen Zeit nahezu überein?

Teilaufgabe Optimierung

Die offensichtlichen Hotspots lassen sich eliminieren: Beim n-ten Schleifendurchlauf werden `x^n` und `n!` immer wieder vollständig berechnet, obwohl im vorherigen Durchlauf schon x^{n-1} und $(n-1)!$ berechnet wurden.

Optimieren sie die Berechnung, in dem sie den Quotienten $x^n/n!$ in jeder Schleife durch Multiplikation von x/n an den Quotientenwert des vorherigen Schleifendurchlaufs berechnen.

Verwenden Sie wieder gprof, um die kumulierte Zeit mit der notierten zu vergleichen.

Aufgabe Raytracer Profiling

Verwenden Sie gcov, um die Hotspots des Raytracers zu finden, und gprof, um die Programmteile zu identifizieren, die am meisten Rechenzeit beanspruchen.

Quelltext taylor.cc

```
#include <iostream>
#include <cmath>

double power(double x, size_t n) {
  return pow(x,n);
}

double faculty(double n) {
  return tgamma(n + 1);
}

double e_hoch_x(double x, const size_t ITERATIONEN = 1000) {
  double e_hoch_x = 1.0;

  for (size_t n = 1; n < ITERATIONEN; n++) {
    e_hoch_x += power(x, n) / faculty(n);
  }

  return e_hoch_x;
}

// Ihre optimierte Version bei der auf eine lokale Variable in jedem
// Schleifendurchlauf x/n hinzumulitpliziert wird, um
// x^n / n! iterativ zu berechnen, statt power und faculty aufzurufen
// es werden dadurch wiederholte Berechnungen vermieden
//
// zum Aufruf in der main-Methode vertauschen sie die Namen dieser Funktion
// mit der obigen Variante
double e_hoch_x_1(double x, const size_t ITERATIONEN = 1000) {
  double e_hoch_x = 1.0;
  double quotient = 1.0;
  for (size_t n = 1; n < ITERATIONEN; n++) {
     // Ihre Optimierung
  }

  return e_hoch_x;
}

int main(void) {
  // ein Testaufruf
  std::cout << e_hoch_x(1.0) << std::endl;
  // Die Eulersche Zahl e sollte ausgegeben werden

  // ab hier viele Aufrufe durchfuehren
  double e = 0.0;
  for (size_t i = 0; i < 10000; i++) {
```

```
    e += (i % 2 == 0 ? -1 : 1) * e_hoch_x(1.0);
  }

  std::cout << e << std::endl; // es sollte 0 herauskommen
  return 1;
}
```