Barry Linnert

Nichtsequentielle und verteilte Programmierung, SS2021

Übung 7

Tutor: Florian Alex
Tutorium 3

Rui Zhao, William Djalal, Simeon Vasilev

12. Juni 2021

1 N-Body in C (10 Punkte)

In einem zwei-dimensionalem Universum (Fläche) befinden sich mehrere Objekte. Für diese Objekte wird angenommen, dass sie eine Masse haben, aber keine Fläche einnehmen (Massepunkte). Die Massepunkte ziehen sich gegenseitig an. Die Anziehungskraft wird durch die Masse und die Entfernung voneinander bestimmt. Implementieren Sie in C eine Simulation des Problems als sequentielles Programm. Wählen Sie für alle Eingaben und Bedingungen geeignete Werte. Dokumentieren Sie Ihr jeweiliges Programm und stellen Sie immer die Ausgaben des jeweiligen Programms zur Verfügung.

Wenn in dieser Simulation die Massepunkte als keine-Fläche-Punkte betrachtet wird, dann tendiert die Gravitationskraft mit abnehmendem Abstand gegen unendlich. Was hat die Fläche damit zu tun?
? Um dieses Problem zu vermeiden, nehmen wir an, dass diese virtuelle 'Ball' einen Radius hat.
Und nehmen wir an, dass die gegenseitige Beschleunigung (acceleration) Null wird, wenn sie sich treffen, d.h. wenn der Abstand gleich dem Durchmesser ist.

Die Kommentare zu den Variablen wurden in den Code geschrieben.

Die Grundidee dieser Simulation besteht darin, zuerst einen Massenpunkt auszuwählen und dann die Beschleunigung (a = F / m) zu berechnen, indem die Kräfte($F = G * M * m/r^2$)berechnet werden, die von allen anderen Massenpunkten auf ihn einwirken.
Geschwindigkeit (v' = v + a* $\triangle$t) wird durch Beschleunigung berechnet und
Verschiebung (s' = s + v* $\triangle$t) wird durch Geschwindigkeit berechnet. ✓

?

```c
#include <unistd.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>
#include <sys/time.h>


#define N 20     // Number of N-body

const double r = 0.01;   // Radius of mass point
const double G = 6.67E-1;  // Gravitational constant G
// Here G is used for simulation.
// To speed up the simulation process, the order of magnitude of G is reduced.
```

```
const double delta_t = 0.1;  // Indicates the time elapsed during each update

double Ballvx[N];     // velocity x (The component of velocity in the x-direction)
double Ballvy[N];     // velocity y
double Ballax[N];     // acceleration x
double Ballay[N];     // acceleration y
double Ballpx[N];     // position x
double Ballpy[N];     // position y
double Ballm[N];     // Mass of the ball

void init() {
 for (int i = 0; i < N; i++)
 {
  // This is used for simple initialization and can be changed at will.
  Ballpx[i] = 3 * (i+1) ;
  Ballpy[i] = 4 * i ;
  Ballvx[i] = 0;
  Ballvy[i] = 0;
  Ballax[i] = 0;
  Ballay[i] = 0;
  Ballm[i] = 1 * i;
 }
}


// Calculate the force F, and from this, calculate the acceleration a = F/m.
// Here the components of acceleration in the x and y directions are calculated,
// which makes it easier to calculate the components of velocity afterwards.
void force(int index) {
 Ballax[index] = 0;
 Ballay[index] = 0;

 // Calculate the impact of all other points, for the selected point
 for (int i = 0; i < N; i++) {
  if (i == index) continue;
  double dx = Ballpx[index] - Ballpx[i];
  double dy = Ballpy[index] - Ballpy[i];
  double d = dx * dx + dy * dy;
  // No collision of balls
  if (d >= r * r){
   Ballax[index] += (G * Ballm[i] / d) * (dx / sqrt(d)) *(-1);
   // printf("%f\n", Ballax[index]);
   Ballay[index] += (G * Ballm[i] / d) * (dy / sqrt(d)) *(-1);
  }
  // The balls meet each other
  else{
   Ballax[index] += 0;
   Ballay[index] += 0;
  }
 }
}


// With acceleration, the velocity is calculated. (v' = v + aΔ*t)
// Here the components of the velocity in the x and y directions are calculated,
// which makes it easier to calculate the components of the displacement afterwards.
void velocity(int index) {
 Ballvx[index] += Ballax[index] * delta_t;
 Ballvy[index] += Ballay[index] * delta_t;
}


// With velocity, the displacement is calculated. (s' = s + vΔ*t)
void position(int index) {
 Ballpx[index] += Ballvx[index] * delta_t;
 Ballpy[index] += Ballvy[index] * delta_t;
}
```

Wieso das?

```
84 // After a short time, the acceleration, velocity and displacement of all masses are
       updated.
85 // The order here cannot vary and must be executed sequentially.
86 // This is because the acceleration needs to be known to calculate the velocity change
87 // and the velocity needs to be known to calculate the position change.
88 void update() {
89  for (int i = 0; i < N; i++)
90   force(i);
91
92  for (int i = 0; i < N; i++)
93   velocity(i);
94
95  for (int i = 0; i < N; i++)
96   position(i);
97 }
98
99
100 int main() {
101  int j = 0;
102  struct timeval start;
103     struct timeval end;
104     unsigned long timer;
105
106  init();
107
108  gettimeofday(&start, NULL);
109
110  while (j <= 1000)
111  {
112
113
114   for (int i = 0; i < N; i++)
115   {
116    printf("round %d : Nr. %d, position_x %f, position_y %f \n", j, i, Ballpx[i], Ballpy[i])
        ;
117   }
118   update();
119
120   printf("\n\n\n");
121   // nanosleep((const struct timespec[]){{0, 100000L}}, NULL);
122   j++;
123  }
124
125  gettimeofday(&end, NULL);
126  timer = 1000000 * (end.tv_sec - start.tv_sec) + end.tv_usec - start.tv_usec;
127     printf("timer = %ld us\n", timer);
128
129     printf("Number of physical cores: %d\n",omp_get_num_procs());
130
131  return 0;
132 }
```

✓

✓

Ausgaben?

8/10

2 N-Body mit OpenMP (10 Punkte)

Erweitern Sie Ihre Lösung der Aufgabe 1 mit Hilfe von OpenMP so, dass geeignete Bereiche der Simulation durch mehrere Prozessoren parallel bearbeitet werden.

Die Kommentare zu den Variablen wurden in den Code geschrieben.

Die Idee ist, dass der Code im Update-Teil parallel berechnet werden kann, um die Geschwindigkeit zu erhöhen. Denn in jeder Update-Runde wird der neue Zustand (Beschleunigung, Geschwindigkeit, Verschiebung) jedes 'Balls' berechnet. Die Reihenfolge der ausgewählten 'Ball' hat keinen Einfluss auf die Ergebnisse.


```c
#include <unistd.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>
#include <sys/time.h>


#define N 20     // Number of N-body

const double r = 0.01;   // Radius of mass point
const double G = 6.67E-1;  // Gravitational constant G
// Here G is used for simulation.
// To speed up the simulation process, the order of magnitude of G is reduced.
const double delta_t = 0.1;  // Indicates the time elapsed during each update

double Ballvx[N];    // velocity x (The component of velocity in the x-direction)
double Ballvy[N];    // velocity y
double Ballax[N];    // acceleration x
double Ballay[N];    // acceleration y
double Ballpx[N];    // position x
double Ballpy[N];    // position y
double Ballm[N];    // Mass of the ball

void init() {
 for (int i = 0; i < N; i++)
 {
  // This is used for simple initialization and can be changed at will.
  Ballpx[i] = 3 * (i+1) ;
  Ballpy[i] = 4 * i ;
  Ballvx[i] = 0;
  Ballvy[i] = 0;
  Ballax[i] = 0;
  Ballay[i] = 0;
  Ballm[i] = 1 * i;
 }
}


// Calculate the force F, and from this, calculate the acceleration a = F/m.
// Here the components of acceleration in the x and y directions are calculated,
// which makes it easier to calculate the components of velocity afterwards.
void force(int index) {
 Ballax[index] = 0;
 Ballay[index] = 0;

 // Calculate the impact of all other points, for the selected point
 for (int i = 0; i < N; i++) {
  if (i == index) continue;
  double dx = Ballpx[index] - Ballpx[i];
  double dy = Ballpy[index] - Ballpy[i];
```

das ist der aufwendige Teil

```
  double d = dx * dx + dy * dy;
  // No collision of balls
  if (d >= r * r){
   Ballax[index] += (G * Ballm[i] / d) * (dx / sqrt(d)) *(-1);
   // printf("%f\n", Ballax[index]);
   Ballay[index] += (G * Ballm[i] / d) * (dy / sqrt(d)) *(-1);
  }
  // The balls meet each other
  else{
   Ballax[index] += 0;
   Ballay[index] += 0;
  }
 }
}


// With acceleration, the velocity is calculated. (v' = v + aΔ*t)
// Here the components of the velocity in the x and y directions are calculated,
// which makes it easier to calculate the components of the displacement afterwards.
void velocity(int index) {
 Ballvx[index] += Ballax[index] * delta_t;
 Ballvy[index] += Ballay[index] * delta_t;
}


// With velocity, the displacement is calculated. (s' = s + vΔ*t)
void position(int index) {
 Ballpx[index] += Ballvx[index] * delta_t;
 Ballpy[index] += Ballvy[index] * delta_t;
}

// After a short time, the acceleration, velocity and displacement of all masses are
    updated.
// The order here cannot vary and must be executed sequentially.
// This is because the acceleration needs to be known to calculate the velocity change
// and the velocity needs to be known to calculate the position change.
void update() {
 // Here the OpenMP technique is used,
 // and the parts within update are computed in parallel using multiple processes.
 // The order of computation of different 'balls' is not affected,
 // so parallel computation can be used to speed up the computation.
 #pragma omp parallel for num_threads(4)
 for (int i = 0; i < N; i++)
  force(i);

 #pragma omp parallel for num_threads(4)
 for (int i = 0; i < N; i++)
  velocity(i);

 #pragma omp parallel for num_threads(4)
 for (int i = 0; i < N; i++)
  position(i);
}


int main() {
 int j = 0;
 struct timeval start;
    struct timeval end;
    unsigned long timer;

 init();

 gettimeofday(&start, NULL);

 while (j <= 1000)
 {

```

Warum nur 4 Threads?

kann zusammengefasst werden

```
  for (int i = 0; i < N; i++)
  {
   printf("round %d : Nr. %d, position_x %f, position_y %f \n", j, i, Ballpx[i], Ballpy[i])
     ;
  }
  update();

  printf("\n\n\n");
  // nanosleep((const struct timespec[]){{0, 100000L}}, NULL);
  j++;
 }

 gettimeofday(&end, NULL);
 timer = 1000000 * (end.tv_sec - start.tv_sec) + end.tv_usec - start.tv_usec;
    printf("timer = %ld us\n", timer);

    printf("Number of physical cores: %d\n",omp_get_num_procs());

 return 0;
}
```

7/10

3 Bewertung der Lösungen (10 Punkte)

Vergleichen Sie Ihre Lösungen aus Aufgabe 1 und 2 bei der Ausführung mit einem Prozessor. Ermitteln Sie den Speed-up Ihrer Lösung der Aufgabe 2 mindestens bis zur Benutzung von 4 Prozessoren. Wie wird sich der Speed-up für Ihre Lösung beim Einsatz weiterer Prozessoren vermutlich entwickeln? Begründen Sie Ihre Antwort.

Mir ist ein Phänomen aufgefallen: die gleiche erste Aufgabe (sequentielle Ausführung) dauert beim ersten Durchlauf deutlich länger (timer = 7594179 us). Wenn es danach erneut ausgeführt, ist die verstrichene Zeit deutlich kürzer(timer = 1620046 us , 1647477 us). Vermutlich ist es die Zwischenspeicherung des Systems, die nachfolgende Läufe schneller macht.

(✓)

N = 5 , ROUND = 3000

Aufgabe 1
timer = 1620046 us

Aufgabe 2
1 core timer = 1611109 us
2 core timer = 1685038 us
3 core timer = 1661551 us
4 core timer = 1544215 us
5 core timer = 1544074 us
6 core timer = 2445293 us
8 core timer = 2037327 us
10 core timer = 2052098 us
12 core timer = 2064984 us

Habt ihr ein noch größeres N getestet?

N= 20 , ROUND = 1000

Aufgabe 1
timer = 1799661 us

Aufgabe 2
1 core timer = 1795500 us
4 core timer = 1700006 us
5 core timer = 1759065 us
6 core timer = 2565672 us
7 core timer = 2125313 us
8 core timer = 2140112 us

Speed-up?

Wenn es nur einen Prozess gibt, gibt es fast keinen Unterschied in der Rechenzeit zwischen den A1 und A2. das ist gut :D

Wenn beispielsweise vier Prozesse verwendet werden, ist ersichtlich, dass die Geschwindigkeit des parallelen Rechnens deutlich verbessert wurde und die Rechenzeit verkürzt wurde. Und es zeigt sich, dass ab zwei Prozessen – drei Prozessen – vier Prozessen die Rechenzeit kürzer wird, da die Vorteile des Parallel Computing allmählich den Overhead des Parallel Computing übersteigen.

Mein Computer hat "nur" sechs physische Kerne.
Eingeschränkt durch die Anzahl der physikalischen Kerne lässt sich bei einer Anzahl von Prozessen über 6 die Rechenleistung kaum noch steigern und auch der Overhead des parallelen Rechnens steigt. Wir vermuten, dass die bestmögliche Performance des Programms erreicht, wenn die Anzahl der parallelen Berechnungen ungefähr der Anzahl der physischen Kerne (der Anzahl der verfügbaren Prozessoren) entspricht. Zu diesem Zeitpunkt gibt es immer einen Kern, der die Rechenaufgaben eines Prozesses übernimmt.

7/10

22/30

Literatur