Optimierung

Nathanael Hübbe

nathanael.huebbe@informatik.uni-hamburg.de

University of Hamburg

16-05-2013

informatik
die zukunft

Beobachtung:

Praktisch alle Programme
verbringen praktisch alle Zeit
in einem verschwindend kleinen Stück Code.

- Das ist die innere Schleife
- Optimierungen an anderer Stelle sind sinnlos

Der Begriff der inneren Schleife, in der Form wie er hier verwendet wird, ist heute nicht mehr gebräuchlich. Leider gibt es aber keinen geläufigeren, adäquaten Ersatz für diesen Begriff. Da dieser Begriff aber sehr hilfreich ist im Nachdenken über Performanzprobleme und Optimierungen, wird er in dieser Präsentation weiterhin verwendet.

Was ist die innere Schleife?

- Innere Schleife ist über die Zeit definiert
- Meist besonders oft durchlaufener Code
- Kann auch Warten beinhalten

Beispiel 1: Arraysumme

```
double threeDSum(long width, long height, long depth, double (*data)[
    height][width]) {
  double sum = 0;
  for(long i = 0; i < width; i++) {
    for(long j = 0; j < height; j++) {
      for(long k = 0; k < depth; k++) {
        sum += data[k][j][i];
      }
    }
  }
  return sum;
}
```

Innere Schleife:

- `k < depth;`
- `sum += data[k][j][i];`
- `k++;`

Beispiel 2: Fibonaccizahlen

```
long fibonacci(long n) {
  if(n <= 1) return n;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

Innere Schleife:

- Alles
- Aber n <= 1 wird doppelt so oft ausgeführt

Beispiel 3: Bubblesort

```
void sortStrings(char** strings, long count) {
  for(long rounds = count-1; rounds > 0; rounds--) {
    for(long i = 0; i < rounds; i++) {
      if(strcoll(strings[i], strings[i+1]) > 0) {
        char* temp = strings[i];
        strings[i] = strings[i+1];
        strings[i+1] = temp;
      }
    }
  }
}
```

Innere Schleife:

- **inner**halb von `strcoll()`

Drei Strategien:

- Herausziehen
  - Nichts mehrfach berechnen, sondern einmal am Anfang
- Optimieren
  - Keine überflüssigen Berechnungen
  - Teure Operationen durch billige ersetzen
  - Warten vermeiden
    - Cacheeffekte
- Ersetzen
  - Zahl der Durchläufe reduzieren

Algorithmische Ebene

- Welche Ordnung hat der Algorithmus, der die innere Schleife enthält?
- Gibt es Algorithmen besserer Ordnung, die auch wirklich schneller sind?
- Kann ich Anwendungseinschränkungen ausnutzen?

Beispiel 1: Arraysumme

```
double threeDSum(long width, long height, long depth, double (*data)[
    height][width]) {
  double sum = 0;
  for(long i = 0; i < width; i++) {
    for(long j = 0; j < height; j++) {
      for(long k = 0; k < depth; k++) {
        sum += data[k][j][i];
      }
    }
  }
  return sum;
}
```

Beispiel 1: Arraysumme

Cacheeffekte

```
double threeDSum(long width, long height, long depth, double (*data)[
    height][width]) {
  double sum = 0;
  for(long k = 0; k < depth; k++) {
    for(long j = 0; j < height; j++) {
      for(long i = 0; i < width; i++) {
        sum += data[k][j][i];
      }
    }
  }
  return sum;
}
```

Beispiel 1: Arraysumme

Cacheeffekte

Herausziehen von Mehrfachberechnungen

```
double threeDSum(long width, long height, long depth, double (*data)[
    height][width]) {
  double sum = 0;
  for(long k = 0; k < depth; k++) {
    double (*curSlice)[width] = data[k];
    for(long j = 0; j < height; j++) {
      double* curLine = curSlice[j];
      for(long i = 0; i < width; i++) {
        sum += curLine[i];
      }
    }
  }
  return sum;
}
```

Beispiel 2: Fibonaccizahlen

```
long fibonacci(long n) {
  if(n <= 1) return n;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

Beispiel 2: Fibonaccizahlen

Ersetzen der inneren Schleife: $O(2^n)$ -> $O(n)$

```
long fibonacci(unsigned long n) {
  if(n < 1) return 0;
  if(n == 1) return 1;
  long numbers[n+1], i;
  numbers[0] = 0;
  numbers[1] = 1;
  for(i = 2; i <= n; i++) numbers[i] = numbers[i-1] + numbers[i-2];
  return numbers[n];
}
```

Beispiel 2: Fibonaccizahlen

Ersetzen der inneren Schleife: $O(2^n)$ -> $O(n)$

Teure Speicherzugriffe durch Registerzugriffe ersetzen

```
long fibonacci(unsigned long n) {
  if(n < 1) return 0;
  if(n == 1) return 1;
  long number1 = 0;
  long number2 = 1;
  for(long i = 2; i <= n; i += 2) {
    number1 += number2;
    number2 += number1;
  }
  return (n & 1) ? number2 : number1;
}
```

Beispiel 3: Bubblesort

```
void sortStrings(char** strings, long count) {
  for(long rounds = count-1; rounds > 0; rounds--) {
    for(long i = 0; i < rounds; i++) {
      if(strcoll(strings[i], strings[i+1]) > 0) {
        char* temp = strings[i];
        strings[i] = strings[i+1];
        strings[i+1] = temp;
      }
    }
  }
}
```

Beispiel 3: Bubblesort

Zahl der Durchläufe reduzieren

```
void sortStrings(char** strings, long count) {
  long lastFirstChange = 1;
  for(long rounds = count-1; rounds > 0; rounds--) {
    long firstChange = -1;
    for(long i = lastFirstChange-1; i < rounds; i++) {
      if(strcoll(strings[i], strings[i+1]) > 0) {
        char* temp = strings[i];
        strings[i] = strings[i+1];
        strings[i+1] = temp;
        if(firstChange == -1) firstChange = i;
      }
    }
    lastFirstChange = firstChange;
    if(!lastFirstChange) lastFirstChange = 1;
    if(lastFirstChange < 0) return;
  }
}
```

Beispiel 3: Bubblesort

Algorithmus ersetzen: Mergesort $O(n^2)$ -> $O(nlogn)$

```
void sortStrings(char** strings, long count) {
  //Split into two parts
  long count1 = count/2, count2 = count - count/2, i;
  char* half1[count1 + 1], *half2[count2 + 1];
  for(i = 0; i < count1; i++) half1[i] = strings[i];
  for(i = 0; i < count2; i++) half2[i] = strings[i + count1];
  half1[count1] = half2[count2] = NULL;
  //Sort the parts
  sortStrings(half1, count1);
  sortStrings(half2, count2);
  //Merge the parts
  char** curA = half1, **curB = half2;
  for(i = 0;; i++) {
    if(!*curA) for(; *curB; i++) strings[i] = *curB++;
    else if(!*curB) for(; *curA; i++) strings[i] = *curA++;
    else strings[i] = (strcoll(*curA, *curB) < 0) ? *curA++ : *curB++;
  }
}
```

Beispiel 3: Bubblesort

Ausnutzung von Anwendungseinschrängung:

fast sortierte Eingabedaten $\Rightarrow$ Insertionsort

```
void sortAlmostSortedStrings(char** strings, long count) {
  for(long i = 1, j; i < count; i++) {
    char* temp = strings[i];
    for(j = i - 1; j >= 0; j--) {
      if(strcoll(strings[j],temp) <= 0) break;
      strings[j + 1] = strings[j];
    }
    strings[j + 1] = temp;
  }
}
```

Kosten von Operationen

- Rechenbefehle O(1 Zyklus)
- Level 1 Cache O(3-4 Zyklen)
- Funktionsaufruf O(10 Zyklen)
- Speicherzugriff O(25 Zyklen)
- malloc O(>150 Zyklen)
- syscall/Kontextwechsel O(1.000-30.000 Zyklen)
- Netzwerknachricht O(500.000 Zyklen)
- Festplattenzugriff O(30.000.000 Zyklen)

Verstecken von Latenzen

- Parallelausführung von CPU-Befehlen
  - Benachbarte Befehle sollten unabhängig sein
- Festplatte: Asynchrone I/O
  - Lange möglich, selten verwendet

Die Kosten der Kommunikation

- syscall/Kontextwechsel O(1.000-30.000 Zyklen)
- Latenz Netzwerknachricht O(500.000 Zyklen)
- + Warten auf Sender/Empfänger!

$\Rightarrow$ Nur Kommunizieren, wenn absolut unvermeidbar

- ... und wenn, dann möglichst viel auf einmal

Synchronisationsverluste enstehen durch

- Unterschiedliche Last
  - Oft schwer vermeidbar
- Unterschiedliche Unterbrechungen
  - Weder eliminier- noch beeinflussbar!

$\Rightarrow$ Nutzung der Wartezeit ist Pflicht

Wartezeitnutzung in MPI

- Nichtblockierendes Senden/Empfangen (isend/irecv)
- Sicherstellen, dass Wartezeit gefüllt werden kann
  - z. B. durch Aufgabenpuffer

Beispiel MPI-Kommunikation

```
void exchangeData(const exchangeDescriptor& descriptor) {
  for(int process = 0; process < procCount; process++) {
    if(isReciever(descriptor, process)) MPI_Isend(...);
  }
  for(int process = 0; process < procCount; process++) {
    if(isSender(descriptor, process)) MPI_Irecv(...);
  }
  MPI_Waitall(...);
}
```

Beispiel MPI-Kommunikation verbessert

```
void dataReady(const exchangeDescriptor& descriptor) {
  MPI_Waitall(oldSendRequests);
  for(int process = 0; process < procCount; process++) {
    if(isReciever(descriptor, process)) MPI_Isend(...);
  }
}
void recieveBufferReady(const exchangeDescriptor& descriptor) {
  for(int process = 0; process < procCount; process++) {
    if(isSender(descriptor, process)) MPI_Irecv(...);
  }
}
void finishRecieve(const exchangeDescriptor& descriptor) {
  MPI_Waitall(recieveRequests);
}
```

Zusammenfassung

- Grundvoraussetzung: Wissen was wieviel Zeit kostet
- Immer auf die innere Schleife konzentrieren
- Auf allen Ebenen optimieren
- Algorithmische Ebene ist die wichtigste
- Nicht warten, rechnen!