Effiziente Programme

Sabrina Herbst | 11807863,
Florian Schabasser | 11810319,
Stefan Öhlwerther | 11809642

Agenda

- Ramasort
  - Anfangsversion
  - *Datenstruktur*
  - *Mergesort*
  - *Zwischenspeicherung von Berechnungen*
  - *Memory Optimierung*
  - *Loop Änderungen*
- Ramanujan
  - Anfangsversion
  - *Kollisionsbehandlung*
  - *Datenstruktur*
  - *Obere / Untere-Schranken*
- Konklusion

Ramasort

Anfangsversion

- Idee:
  - Finde alle möglichen Kandidaten
  - Sortiere die Liste
  - Zähle Kandidaten, die öfter vorkommen

```
Performance counter stats for './ramasort 10000000000000':

    117346015144      cycles
    207312595859      instructions              #    1.77  insn per cycle
       604060603      branch-misses
        66000313      LLC-load-misses
       154214925      LLC-store-misses

    25.303735456 seconds time elapsed

    23.820450000 seconds user
     1.479779000 seconds sys
```

make RAMA=ramasort MEMORY=9000000

Datenstruktur Eliminierung

- Schleifenindizes wurden in struct gespeichert
  - Diese wurde danach nicht mehr verwendet
  - Eliminieren von struct
  - Einzelne long Werte halbiert Memory usage

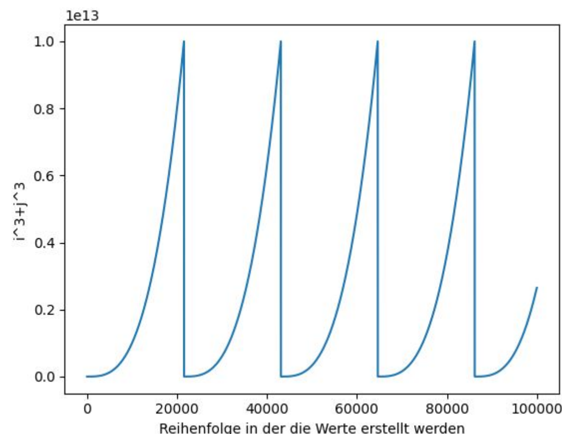
```
Performance counter stats for './ramasort 10000000000000':

     62626910341      cycles
    125774200560      instructions              #    2.01  insn per cycle
       608167102      branch-misses
        24224765      LLC-load-misses
        67840625      LLC-store-misses

    13.458019254 seconds time elapsed

    12.689496000 seconds user
     0.768090000 seconds sys
```

make RAMA=ramasort MEMORY=3500000

Merge Schritt von Mergesort

```
long increment = 1;
for (i = 0; i < (long) log2( x: (double) n_indices); i++) {
    for (j = 0; j <= n_indices - 2 * increment; j += 2 * increment) {
        merge( arr: table, l: indices[j], m: indices[j + increment] - 1, r: indices[j + increment * 2] - 1);
    }
    increment *= 2;
}
```

- In den Schleifen werden sortierte Teilabschnitte erzeugt
  - Diese werden in qsort wieder komplett überarbeitet
  - Wir können dies zu unserem Vorteil ausnutzen, wenn wir die merge Funktion von Mergesort für das Sortieren verwenden




```
Performance counter stats for './ramasort 10000000000000':

     45506875081      cycles
     70670902116      instructions              #    1.55  insn per cycle
       633781192      branch-misses
        45054156      LLC-load-misses
       228110020      LLC-store-misses

     9.947574821 seconds time elapsed

     7.383868000 seconds user
     2.551954000 seconds sys
```

make RAMA=ramasort MEMORY=3500000

Zwischenspeichern von Berechnungen

- Zwischenspeichern damit nur einmal berechnet:
  - cube(i)
  - cube(i) + cube(j)
- Überraschenderweise keine große Performanceverbesserung
  - Wird evtl. schon vom Compiler übernommen?

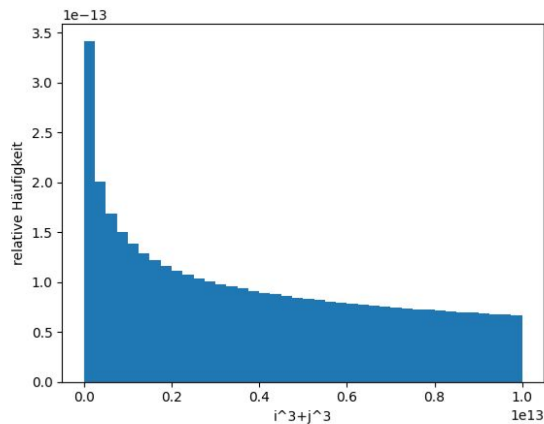
```
Performance counter stats for './ramasort 10000000000000':

     45312366490      cycles
     70669685938      instructions              #    1.56  insn per cycle
       633371049      branch-misses
        45096853      LLC-load-misses
       225109648      LLC-store-misses

     9.894613293 seconds time elapsed

     7.333716000 seconds user
     2.548596000 seconds sys
```

make RAMA=ramasort MEMORY=3500000

Memory Optimierung

- Unterteilen in mehrere Listen
  - Anfangs: bis 20000 Werte pro Liste (inc = 20000)
- Aufgrund von log-ähnlicher Verteilung
  - Dynamische Anpassung von Ober- und Untergrenze

```
if (m <= table_size / 3) {
    inc *= 2;
}

lower = upper;
upper = upper + inc;
```




```
Performance counter stats for './ramasort 10000000000000':

      66330960489      cycles
     194585595296      instructions              #    2.93  insn per cycle
        630940357      branch-misses
         36046545      LLC-load-misses
         51050772      LLC-store-misses

     14.270033435 seconds time elapsed

     14.221782000 seconds user
      0.048006000 seconds sys
```

make RAMA=ramasort

Loop Änderungen

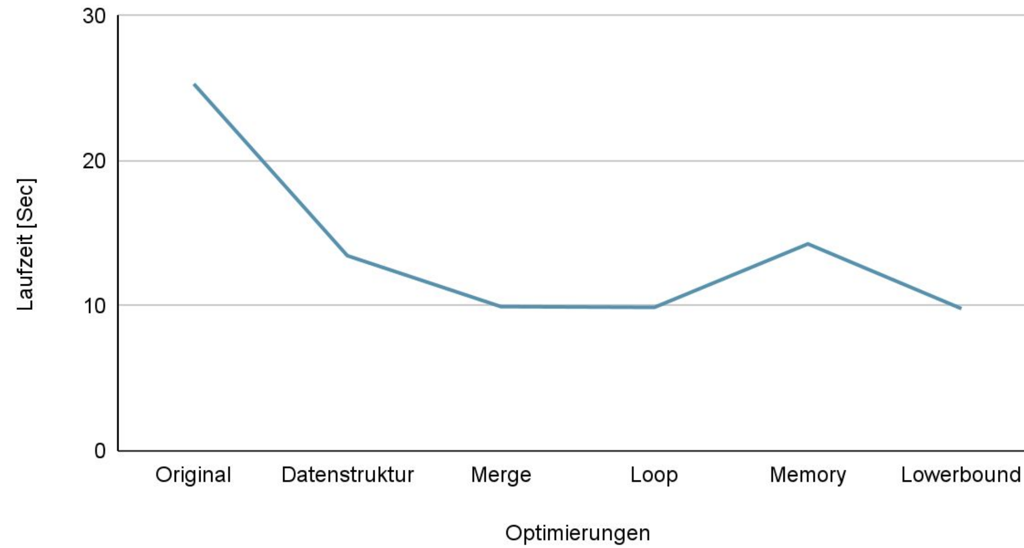
- Dynamische Anpassung von j
  - Wir können uns den ersten relevanten j-Wert vorberechnen, um uns Iterationen zu sparen

```
if (cube_i + cube( n: i + 1) < lower) {
    j = (long) cbrt((double) lower - (double) cube_i) + 1;
} else {
    j = i + 1;
}
```

```
Performance counter stats for './ramasort 10000000000000':

     45359314002      cycles
    110498939557      instructions              #    2.44  insn per cycle
       653142638      branch-misses
        37210107      LLC-load-misses
        50117935      LLC-store-misses

     9.808501714 seconds time elapsed

     9.748262000 seconds user
     0.060001000 seconds sys
```

make RAMA=ramasort

Ergebnis



Ramanujan

Anfangsversion

- Idee:
  - Ermittle Kandidat
  - Halte alle Kandidaten in einer Hash Tabelle
    - LinkedList hinter jedem Eintrag (Kollisionen)
  - Lookup für den neuen Kandidaten:
    - Neuer Wert => Kandidat wird gespeichert
    - Vorhandener Werte => Ramanujan Zahl

```
Performance counter stats for './ramanujan 10000000000000':

     106268218679      cycles
      58344209669      instructions              #    0.55  insn per cycle
        131366562      branch-misses
        403828417      LLC-load-misses
        127010217      LLC-store-misses

     32.377986809 seconds time elapsed

     30.157414000 seconds user
      2.207810000 seconds sys
```

make RAMA=ramanujan MEMORY=9000000

Behandlung von Kollisionen

- Kollisionen ohne Listen behandeln
- Nächste freie Position der Hash Tabelle nutzen

```c
struct node {
    long value;
    long count;
};
```

```c
struct node **lookup(long key, struct node **table, size_t table_s
/* comment */
{
    long pos = key;
    struct node **pp = table + hash(key, table_size);
    for (; *pp != NULL; pp = table + hash(++pos, table_size))
        if ((*pp)->value == key)
            return pp;
    return pp;
```

```
Performance counter stats for './ramanujan 10000000000000':

     111663894102      cycles
      62114278831      instructions              #    0.56  insn per cycle
        164147917      branch-misses
       1051836403      LLC-load-misses
        126466368      LLC-store-misses

     40.124572328 seconds time elapsed

     37.658556000 seconds user
      2.451906000 seconds sys
```

make RAMA=ramanujan MEMORY=5500000

Datenstruktur Eliminierung

- Ramanujan Zahl direkt abspeichern
- Ergebnis Tabelle anstatt der count Variable

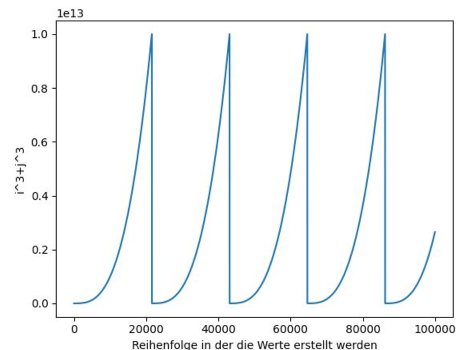
```
for (i = 0; cube(i) <= n; i++) {
    for (j = i + 1; cube(i) + cube(j) <= n; j++) {
        long sum = cube(i) + cube(j);
        long *pos = lookup(sum, candidate_table, candidate_table_size);
        if (*pos == sum) {
            long *pos_res = lookup(sum, res_table, res_table_size);
            if (*pos_res != sum) {
                count++;
                checksum += sum;
                *pos_res = sum;
            }
        } else {
            *pos = sum;
        }
    }
}
```

```
Performance counter stats for './ramanujan 10000000000000':

      30495312171      cycles
      11936295103      instructions              #    0.39  insn per cycle
        165392111      branch-misses
        336633255      LLC-load-misses
         37109256      LLC-store-misses

      8.699761645 seconds time elapsed

      8.355553000 seconds user
      0.344146000 seconds sys
```

make RAMA=ramanujan MEMORY=2500000

Obere / Untere Schranken

- Fixierte Tabellengröße: 2^23
  - Speicherplatz von ~70MB
- Aufteilung des Wertebereichs mittels Schranken (71)
  - 0-30.000.000.000
  - 30.000.000.000-84.498.890.434
  - ...

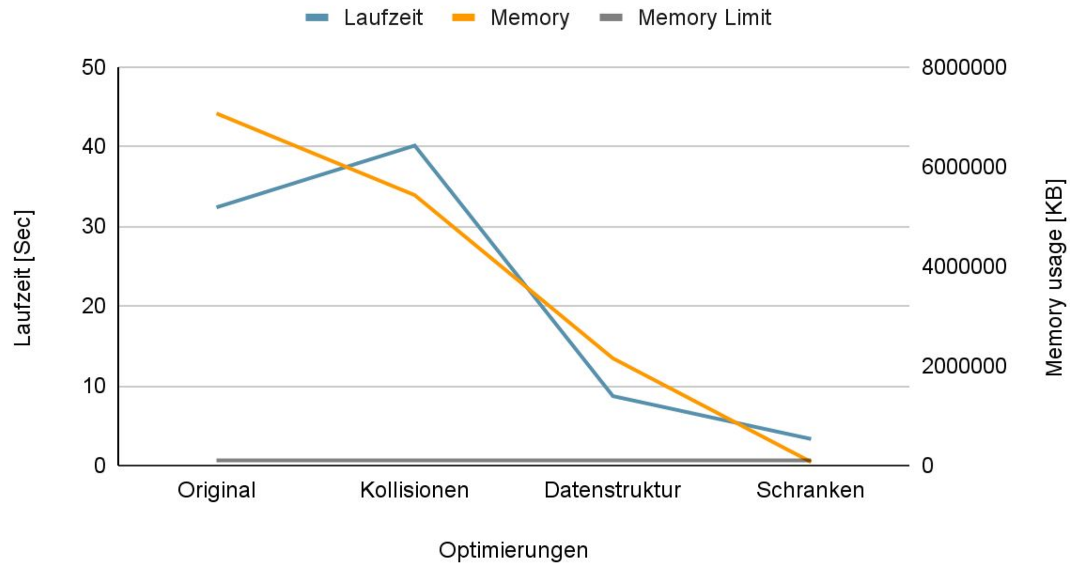



```
for (int k = 0; upperBound < n; ++k) {
    lowerBound = upperBound;
    upperBound = min( x: n,  y: calcUpperBound(lowerBound,  window: k));

    for (i = 0; cube( n: i) <= upperBound; i++) {
        for (j = calcMinJ(lowerBound, i); cube( n: i) + cube( n: j) <= upperBound; j++) {
            long sum = cube( n: i) + cube( n: j);

            if (sum >= lowerBound) {
```

```
Performance counter stats for './ramanujan 100000000000000':

     12680889029      cycles
     11364229535      instructions              #    0.90  insn per cycle
        82420606      branch-misses
       156552286      LLC-load-misses
        66478838      LLC-store-misses

     3.272736540 seconds time elapsed

     3.264458000 seconds user
     0.008001000 seconds sys
```

make RAMA=ramanujan MEMORY=102400

Ergebnis



Konklusion

Konklusion

- Laufzeit- vs. Speicheroptimierung
  - Stehen nicht zwangsweise im Konflikt
- Allokierung
  - Art der Allokierung macht einen großen Unterschied
  - Wenn nacheinander auf Elemente zugegriffen wird, sollten diese in einem Speicherbereich stehen
- Schwierig abzuschätzen, ob Änderungen zu Verbesserungen führen
  - Compiler macht vieles selbst
  - Blackbox