

Abgabe Effiziente Programme WS22/23

Sie können ein beliebiges Beispiel wählen.

Wer nichts anderes vorhat, kann die [unten vorgegebene Aufgabe](#) für dieses Semester wählen.

Natürlich können Sie, wenn Sie wollen, auch die Implementierung eines anderen Problems optimieren; allerdings hat das einige Nachteile: Sie müssen einen Teil der Zeit Ihrer Präsentation für die Erklärung des Problems und des Algorithmus aufwenden, und die Ergebnisse sind nicht direkt vergleichbar. Der Vorteil (vor allem, wenn Sie einen späteren Termin wählen) ist, dass Sie nicht das wiederholen, was andere gemacht haben, oder ein langsamerer Programm präsentieren.

Bereiten Sie eine 18-minütige Präsentation vor (am besten machen Sie einen Probelauf, damit sich die Präsentation auch sicher in der Zeit ausgeht). Da Sie dabei nicht soviel Zeit haben wie ich in der Vorlesung, präsentieren Sie die meisten Schritte nur im Überblick (also eventuell nur, wieviel er gebracht hat), und nur ein paar besonders interessante Schritte mit mehr Details. Besonders interessant sind u.a. die Schritte, die unerwartet viel oder wenig bringen.

Vorgegebene Aufgabe: Ramanujan-Zahlen

Hintergrund

[Ramanujan-Zahlen](#) sind Zahlen, die auf mehrere Arten als Summen zweier Kubikzahlen dargestellt werden können.

Genaue Aufgabe

Die beiden [vorgegebenen Programme](#) (ramasort und ramanujan) zählen die Ramanujan-Zahlen bis zu einer bestimmten Grenze n , indem sie alle Kandidaten-Zahlen $i^3 + j^3 \leq n$ (wobei $i \leq j$) erzeugen und dann durch Sortierung (ramasort) oder eine Hash-Tabelle (ramanujan) herausfinden, ob dieselbe Zahl mehrmals vorkommt.

Dabei wächst der Speicherverbrauch linear mit der Anzahl der Kandidaten, bei $n=10^{13}$ ist er schon $>3.5\text{GB}$ (ramasort) bzw. $>7\text{GB}$ (ramanujan). Wenn man also die Anzahl der Ramanujan-Zahlen für grosse n zählen will, geht leicht der Speicher aus. Daher sollen Sie in dieser Aufgabe einen Algorithmus entwickeln und effizient implementieren, der mit deutlich weniger Speicher auskommt. Damit wir aber nicht die langen Laufzeiten für grosse n brauchen, setzen wir $n=10^{13}=10000000000000$, und das Speicherlimit auf 100MB; beachten Sie, dass das Speicherlimit die Grösse des ausführbaren Files enthält, malloc() noch etwas Speicher für seine Verwaltung braucht, und durch externe Fragmentierung (übriggebliebene Speicherstellen, die zu klein sind, um genutzt zu werden) auch noch Speicher verbraucht werden kann, sodass der nutzbare Speicher bei diesem Limit u.U. deutlich geringer sein kann als 100MB.

Wenn Sie make aufrufen, wird per default ramasort kompiliert und dann mit einem Speicherlimit von 100MB und $N=10^{13}$ gemessen (wobei das Speicherlimit in diesem Fall zu einem Segmentation Fault führt). Man kann bei Aufruf von make die entsprechenden Parameter übergeben:

Parameter	Default
MEMORY	1024000 (100MB)
N	10000000000000 (10^{13})
RAMA	ramasort
CC	gcc
CFLAGS	-O -Wall

Z.B. kann man ramanujan mit knapp 9GB wie folgt laufen lassen:

```
make RAMA=ramanujan MEMORY=9000000
```

Die beiden Programme geben (wenn sie genügend Speicher haben) folgende Zeile (für $n=10^{13}$) aus:

```
114359 Ramanujan numbers up to 10000000000000, checksum=355033384379411459
```

Ihr Programm muss für jedes n diese Zeile genauso ausgeben wie eines der vorgegebenen Programme (wenn die mit genug Speicher gestartet werden).

Zusätzlich geben die vorgegebenen Programme noch Informationen über ihre internen Datenstrukturen und eine Untergrenze für den Speicherverbrauch aus (wobei ramosort mit $n=10^{13}$ und $\text{MEMORY}=4000000$ zwar funktioniert, aber deutlich langsamer ist als bei $\text{MEMORY}=9000000$; offenbar muss die Sortierfunktion `qsort()` mehr arbeiten, wenn der Speicher knapp ist); diese zusätzlichen Ausgaben muss Ihr Programm nicht machen bzw. darf es andere zusätzliche Ausgaben machen.

Hier die Laufzeiten auf der `g0` für die beiden Programme mit $\text{MEMORY}=9000000$:

Performance counter stats for 'ramasort 1000000000000':

```
120074048548    cycles
207077932729    instructions          #    1.72  insn per cycle
   606189276    branch-misses
   58956280     LLC-load-misses
  158296196     LLC-store-misses
```

```
25.854830070 seconds time elapsed
```

```
24.468421000 seconds user
 1.384023000 seconds sys
```

Performance counter stats for 'ramanujan 1000000000000':

```
99013672697     cycles
57773078853     instructions          #    0.58  insn per cycle
  131312766     branch-misses
  415725921     LLC-load-misses
 125359064      LLC-store-misses
```

```
31.283403372 seconds time elapsed
```

```
29.152750000 seconds user
 2.119763000 seconds sys
```

Hier sieht man, dass der große Speicherverbrauch auch zu nennenswerten System-Zeiten (zum Löschen des Speichers) führt. Die LLC-Misses tragen viel bei, bei ramanujan offenbar den größten Teil der Zeit (bei 50ns/LLC-load-miss machen die LLC-load-misses schon 20.79s aus).

Ein eigenartiges Verhalten der `g0` ist, dass sie bei ramosort zwar in der Nähe der 4,7GHz Taktfrequenz landet, die auf dieser Maschine scheinbar das Maximum sind (offiziell kann der Prozessor 5,1GHz, ich habe aber bisher nur 4,7GHz gesehen), bei ramanujan aber nur mit ca. 3,2GHz läuft. Daher reicht es in diesem Fall nicht, für das Ziel Prozessor-Zeit-Effizienz auf die Zyklen zu schauen, sondern man muss auf die Summe von "user" und "sys"-Zeit schauen ("elapsed" ist bei leerer Maschine nahe dran, kann aber steigen, wenn alle Kerne ausgelastet sind). Auf der `g0` ist SMT/Hyperthreading abgeschaltet, um besser reproduzierbare Messwerte zu bekommen.

Hier ein paar Ideen für algorithmische Verbesserungen, die sie verfolgen können, aber nicht müssen (und insbesondere können Sie auch eigene Ideen verfolgen).




- Teilen des Gesamtbereichs 1..n in Abschnitte, in denen jeweils eine genügend kleine Zahl von Kandidaten drinnen sind, um mit dem Speicherlimit auszukommen. Wenn die Abschnitte klein genug für z.B. den LLC-Cache (16MB) sind, sollten die LLC misses deutlich reduziert werden.
- Eine andere Organisation der Hash-Tabelle von ramanujan, die zu weniger Speicherzugriffen und daher weniger misses führt; immerhin werden nur ca. 205M Kandidaten untersucht, da sind 415M LLC-load-misses > 2 misses pro Kandidat im Durchschnitt.
- Man hat nur eine Tabelle mit maximal $\text{cbt}(n/2)$ Elementen (17100 bei $n=10^{13}$), in der für jedes i ein j und $\text{sum}=i^3+j^3$ steht. Und zwar steht das minimale j , sodass sum mindestens so gross ist wie der letzte angeschaute Kandidat. In jedem Schritt wird der Tabelleneintrag mit dem kleinsten sum gefunden, und für diesen Eintrag j erhöht und die zugehörige sum ausgerechnet. Wenn es mehrere Einträge mit der selben sum gibt, hat man wieder eine Ramanujan-Zahl gefunden. Eine möglicherweise geeignete Datenstruktur dafür ist der Heap.
- (Nicht wirklich algorithmisch:) `qsort()` ist zwar eine angenehme Funktion, aber das Interface mit dem Aufruf der Vergleichsfunktion ist nicht sonderlich effizient. Sie können vielleicht durch Schreiben einer spezialisierten Sortierfunktion effizienter werden.

Sie sollten hier aber nicht alle Ideen verfolgen (die ja teilweise in völlig verschiedene Richtungen gehen), sondern sich eine Kombination vielversprechender Ideen überlegen, implementieren, und dann diese Implementierung optimieren; überlassen Sie die anderen Richtungen Ihren Kollegen. Sie können z.B. das Teilen des Gesamtbereichs als einen Optimierungsschritt darstellen (auch wenn's vor allem ein notwendiger funktionaler Schritt ist, wenn man beim Sortieren oder der Hash-Tabelle bleibt), die spezialisierte Sortierfunktion als einen weiteren.

Sie können Ihr Programm (und Doku dazu, z.B. Ihre Präsentation) im Web veröffentlichen (wird nicht beurteilt), und zwar indem Sie auf [nfs/unsafe/http/ftp/pub/anton/lvas/effizienz-abgaben/2022w](https://nfs.unsafe/http/ftp/pub/anton/lvas/effizienz-abgaben/2022w) eine Web-Page (vielleicht mit einem Link zu einem Projekt auf [Sourcehut](https://sourcehut.org) oder Github) oder ein Verzeichnis mit Ihren Dateien anlegen.

Aufgaben vom [[WS02/03](#) | [WS03/04](#) | [WS04/05](#) | [WS05/06](#) | [WS06/07](#) | [WS07/08](#) | [WS08/09](#) | [WS09/10](#) | [WS10/11](#) | [WS11/12](#) | [WS12/13](#) | [WS13/14](#) | [WS14/15](#) | [WS15/16](#) | [WS16/17](#) | [WS17/18](#) | [WS18/19](#) | [WS19/20](#) | [WS20/21](#) | [WS21/22](#)]

[Anton Ertl](#)

Name	Last modified	Size	Description
 Parent Directory		-	
 ramanujan.tar.gz	28-Nov-2022 20:12	1.7K	
 ramanujan/	28-Nov-2022 17:45	-	

*Apache/2.2.22 (Debian) DAV/2 mod_fcgid/2.3.6 PHP/5.4.36-0+deb7u3 mod_python/3.3.1 Python/2.7.3
mod_ssl/2.2.22 OpenSSL/1.0.1e mod_perl/2.0.7 Perl/v5.14.2 Server at www.complang.tuwien.ac.at Port 443*