Fehlertolerante Algorithmen

Sebastian Schefczyk
Albert-Ludwigs Universität Freiburg
Chair of Algorithms and Complexity

3. August 2008

1 Einleitung

1.1 Motivation

In der Beschreibung von Algorithmen liegt für gewöhnlich die Annahme zugrunde, dass Speicherinhalte nur dann verändert werden, wenn dies vom Algorithmus so beschrieben wurde. Diese Annahme trifft nicht vollständig zu, da heute oft große Computer mit viel günstigem Speicher ausgerüstet werden, die aus Kosten- und Performanzgründen auf fehlererkennende und fehlerkorrigierende Methoden verzichten.

Hier können aus verschiedenen Gründen (z.B. Hardware-Fehler, Stromunterbrechungen, kosmische Strahlung, Alphastrahlung) Speicherfehler auftreten, bei denen die Speicherwerte (Schlüssel) verfälscht werden. Die Menge der auftretenden Fehler ist zwar gering, jedoch können bereits wenige Speicherfehler in der Eingabe oder während der Ausführung das Ergebnis vollständig unbrauchbar machen. Diesem Problem wirken *fehlertolerante Algorithmen* entgegen.

Am Beispiel Mergesort lässt sich leicht sehen, dass ein Speicherfehler im Laufe der Merge-Schritte dazu führt, dass keine sortierte Menge berechnet werden kann. In Abbildung 1 wird dies an einem Beispiel verdeutlicht.

1.2 Fehlertolerante Algorithmen

Ein Algorithmus ist *fehlertolerant*, wenn er trotz einzelner verfälschter Speicherinhalte (vor oder während der Ausführung) eine fehlerfreie Ausgabe berechnet, auf Basis unverfälschter Speicherinhalte. Für fehlertolerante Algorithmen wird immer reliabler Speicher in der Größe $\mathcal{O}(1)$ benötigt, dessen Inhalt nicht verfälscht werden kann.

Die Anzahl der möglichen Speicherfehler wird mit einer Obergrenze δ angegeben. Diese ist in der Realität unbekannt und muss vorher geschätzt werden. Die Anzahl tatsächlich vorhandener Fehler in einer speziellen Ausführung wird mit α bezeichnet. Es gilt $\alpha \leq \delta$.

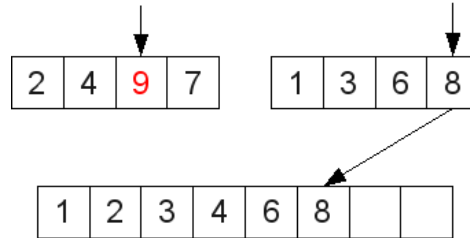

Abb. 1: Beispiel eines Speicherfehlers bei Mergesort. Abgebildet ist ein Rekursionsschritt, in dem aus zwei bereits sortierten Mengen die nächstgrößere sortierte Menge gebildet werden soll, indem links-nach-rechts ein Größenvergleich stattfindet. Der Schlüssel „9“ wurde im Laufe des Merge-Verfahrens, verfälscht und steht nun an einer falschen Position. In allen weiteren Merge-Schritten werden auf die „9“ stets auch kleinere Schlüssel folgen und somit letztlich eine nicht korrekt sortierte Menge ausgegeben.

Ist ein Speicherfehler aufgetreten, so wurde dabei ein Schlüssel *verfälscht*. Anderenfalls redet man von *unverfälschten* Schlüsseln.

2 Fehlertolerantes Mergesort

Bei fehlertoleranten Sortier-Algorithmen können - vor oder während der Ausführung - Verfälschungen der Schlüssel auftreten und diese dürfen die Ausgabe nicht weiter verfälschen. Die Ausgabe muss *0-ungeordnet sein*, wobei gilt: Eine Sequenz ist k-ungeordnet, für $k \geq 0$, wenn das Entfernen von höchstens k unverfälschten Schlüsseln eine Teilsequenz liefert, deren unverfälschte Schlüssel sortiert sind. Der folgende, naive Algorithmus berechnet eine solche 0-ungeordnete Ausgabe.

2.1 Naive-Mergesort

Ausgehend von Mergesort wird der Merge-Schritt verändert [1, Abschnitt 2.1]. In diesem werden aus den beiden zu vereinigenden Mengen statt jeweils nur einem Schlüssel $\delta + 1$ viele Schlüssel je Menge verglichen und das Minimum bestimmt. Somit wird gewährleistet, dass in jeder Menge mindestens ein unverfälschter Schlüssel betrachtet wird. Im Gegensatz zum nicht-fehlertoleranten Mergesort kann somit ein verfälschter Schlüssel nicht mehr die Sortierung der nachfolgenden Schlüssel stören. Die Laufzeit eines Merge-Schrittes, in dem bei n Schlüsseln jeweils 2δ Schlüssel verglichen werden, liegt in $\mathcal{O}(\delta n)$. Ein vollständiges Sortieren einer Sequenz der Größe n mit *Naive-Mergesort* wird in $\mathcal{O}(\delta n \log n)$ berechnet.

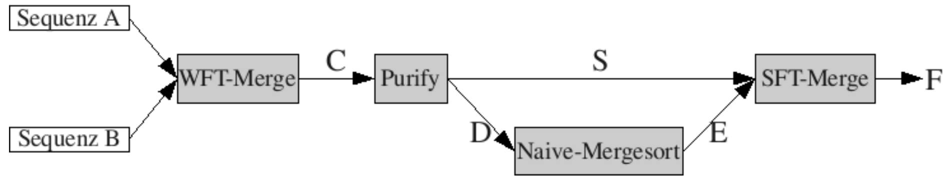

Abb. 2: Ablaufschema des FAST-Algorithmus

2.2 FAST

2004 wurde ein ausgereifterer Algorithmus für fehlertolerantes Mergesort vorgeschlagen [1]. Dieser besteht aus vier Teilen, die in Abbildung 2 veranschaulicht werden. Das oben vorgestellte naive Verfahren Naive-Mergesort wird dabei zusammen mit den Subroutinen Weakly fault-tolerant Merge (WFT-Merge), Purify und Strongly fault-tolerant Merge (SFT-Merge) im Laufe der Merge-Schritte jeweils sequentiell ausgeführt.

2.2.1 Weakly fault-tolerant Merge

WFT-Merge ist ein Merging-Algorithmus, der in linearer Zeit nur *garantiert wenige* Sortierfehler erzeugt [1, Abschnitt 3.1].

Seien i und j die beiden Indizes der beiden zu vereinigenden Mengen A und B. Jede Menge erhält eine Variable, $wait_A$ bzw. $wait_B$. OBdA sei $A[i] \leq B[j]$ und es wird aufsteigend sortiert: Dann wird $wait_A$ auf 0 zurückgesetzt, $wait_B$ um 1 erhöht, i inkrementiert und $A[i]$ mit $B[j]$ wieder verglichen. Wenn aber $wait_B = 2\delta + 1$, dann wird $wait_B$ auf Null gesetzt und das Fenster $W = A[i + 1; i + 2\delta + 1]$ betrachtet. Mit t werden alle Werte in W gezählt, die kleiner als $A[i]$ sind. Wenn $t \geq \delta + 1$, also A[i] offensichtlich falsch einsortiert oder verfälscht, dann schreibe A[i] in die Ausgabe und inkrementiere i. Der Fall $t < \delta + 1$, also $A[i]$ nicht sicher verfälscht oder wenn das Ende von A erreicht wurde, wird wie oben weiter sortiert. $wait_B$ wird analog behandelt.

Zu jedem Zeitpunkt ist eine der beiden *wait* Variablen Null. Wenn $wait_A$ Null ist, also eine A-Häufung stattfindet, dann kann diese Häufung nur die Länge $2\delta{+}1$ haben ($wait_B$ analog). Somit ist die Ausgabe höchstens $\mathcal{O}(\alpha{\cdot}\delta)$-ungeordnet. Die Ausgabe hat die gleiche Länge wie die Eingabe (in Abbildung 2 mit C bezeichnet) und wird in $\mathcal{O}(n)$ berechnet. Die Indizes i, j, die *wait* Variablen und t müssen im reliablen Speicher liegen.

2.2.2 Purify

Purify ist ein fehlertoleranter Algorithmus, der aus einer k-ungeordneten Sequenz C der Länge n eine 0-ungeordnete Subsequenz S, sowie eine Subsequenz D gewinnt, mithilfe eines Stapels [1, Abschnitt 2.2].

Die Sequenz C wird links-nach-rechts durchlaufen und überprüft, ob das Element an der Stelle i, mit $C[i]$ bezeichnet, größer ist als das oberste Element des Stapels *Stack.top*. Wenn $C[i] \geq Stack.top$ wird $C[i]$ auf dem Stapel abgelegt und i inkrementiert. Sonst verschiebe $C[i]$ und *Stack.top* in der *Liste verworfener Schlüssel* und berechne ein neues *Stack.top* aus dem Maximum der $\delta + 1$ obersten Schlüssel des Stapels.

Im Verlauf des Algorithmus gilt die Stackinvariante, dass der oberste Schlüssel größer oder gleich den darunter liegenden unverfälschten Schlüsseln ist. Somit sind nur Werte, die während der Ausführung von Purify verfälscht wurden, in S falsch sortiert. *Stack.top* sowie der Index i müssen im reliablen Speicher liegen.

Die Subsequenz S ist mindestens $n - 2(k + \alpha)$ lang (da jeweils zwei Elemente, unverfälschte oder verfälschte, aussortiert werden), also werden $\mathcal{O}(k + \alpha)$ Schlüssel in die Subsequenz D eingefügt. Purify durchläuft die Sequenz der Länge n und berechnet bei jedem der falsch einsortierten unverfälschten, sowie bei jedem verfälschten Schlüssel, das oberste Element des Stapels aus den $\delta(+1)$ obersten Elementen. Somit liegt die Worst-case Laufzeit in $\mathcal{O}(n + \delta \cdot (k + \alpha))$.

2.2.3 Strongly fault-tolerant Merge

SFT-Merge ist ein fehlertoleranter Merge-Algorithmus, der zwei unbalancierte Subsequenzen zu einer 0-ungeordneten Menge vereinigt [1, Abschnitt 3.2].

Seien A und B zwei Sequenzen, oBdA sei $|A| = n_1 \geq n_2 = |B|$ und beide Mengen sind *garantiert gut geordnet*[1]. SFT-Merge ordnet nun die Schlüssel aus B in A ein.

Die Indizes i, j zeigen die aktuellen Positionen in A, B. In jedem Schritt wird das Minimum aus $B[j, j + \delta]$ gesucht, fortan mit $B[h] = b$ bezeichnet ($j \leq h \leq j + \delta$), und nach $B[j]$ verschoben (die Schlüssel $B[j]$ bis $B[h - 1]$ werden entsprechend verschoben). Anschließend wird j inkrementiert. In A werden alle Schlüssel zur Ausgabe hinzugefügt und i inkrementiert, bis $A[i] > b$. In diesem Fall wird das Fenster $W = A[i + 1; i + 2\delta + 1]$ betrachtet und mit t alle Schlüssel gezählt, die kleiner als A[i] sind. Wenn $t \geq \delta + 1$, also $A[i]$ sicher verfälscht, dann wird $A[i]$ in die Ausgabe geschrieben, i inkrementiert und A weiter durchlaufen. Anderenfalls wird W partitioniert in „$\leq b$“ und „$> b$“, wobei die relative Ordnung bestehend bleibt. Die Ausgabe wird um „$\leq b$“ sowie „b“ ergänzt. Anschließend wird wieder ein neues b gesucht.

Um zu sehen, dass die Ausgabe nun 0-ungeordnet ist, müssen die Auswahl- und Einfüge-Abläufe in B und A betrachtet werden. Innerhalb der aufsteigend sortierten Sequenz B werden fehlerhaft sortierte bzw. verfälschte

[1] Beide Eingangsmengen sind von WFT-Merge und Purify bzw. Naive-Mergesort sortiert.

Schlüssel bei der Auswahl von b unter $\delta + 1$ Schlüsseln erkannt und an die entsprechende Stelle gebracht. Bei der Einordnung von b in A werden ebenso in den $\delta + 1$ Folgeschlüsseln etwaige verfälschte Schlüssel entdeckt und beim Schreiben in die Ausgabe berücksichtigt. Somit ist in der Ausgabe garantiert, dass alle nachfolgenden Schlüssel größer oder gleich allen vorhergehenden Schlüsseln sind. Aus diesem Grund durchläuft SFT-Merge einmal die größere Sequenz und betrachtet in der Größe der kleineren Sequenz und der Anzahl vorhandener Fehler jeweils ein Fenster der Größe δ. Die Laufzeit ist somit $\mathcal{O}(n_1 + (n_2 + \alpha)\delta);\ \forall n_2 \leq n_1$.

2.3 Eigenschaften von FAST

Für den FAST-Algorithmus, der auf dem $\mathcal{O}(n \log n)$ Mergesort-Algorithmus aufbaut, ergibt sich aus den vorgestellten Verfahren eine Gesamtlaufzeit von $\mathcal{O}(\delta n \log n + \alpha\delta^2)$. Diese resultiert aus den einzelnen Merge-Schritten, die in $\mathcal{O}(n + \alpha\delta^2)$ berechnet werden.

Eine weitere Eigenschaft des FAST-Algorithmus ist, dass er maximal $[(n \log n)^{\frac{1}{3}}]$ Speicherfehler tolerieren kann. Dazu sei erwähnt, dass in [1, Abschnitt 4] bewiesen wird, dass ein fehlertoleranter Sortieralgorithmus höchstens gegenüber $[(n \log n)^{\frac{1}{2}}]$ Speicherfehlern robust sein kann.

2.4 Weiterentwicklungen von FAST

Im weiteren Verlauf sollen zwei auf FAST aufbauende Algorithmen kurz skizziert werden, die schneller sind als FAST und auch beide auch die obere Grenze der möglichen Speicherfehler erreichen.

2.4.1 OPT

OPT wurde 2006 vorgestellt und ersetzt in FAST die beiden Subroutinen WFT-Merge und Purify mit einer namens *Purifying-Merge*. In dieser Subroutine wird mit Hilfe eines Puffers der Größe $\Theta(\delta)$ eine 0-ungeordnete Sequenz sowie eine kleinere ungeordnete Sequenz berechnet, analog zu S und D in Abbildung 2. Der Puffer dient hier zur Zwischenspeicherung der Schlüssel, bevor sie in die Ausgabe geschrieben werden. Die Größe der ungeordneten Menge wird dabei auf $\mathcal{O}(\alpha)$ reduziert und somit die Eingabe für den aufwendigen Naive-Mergesort-Algorithmus verkleinert. Die beiden weiteren Subroutinen bleiben unverändert. Die Laufzeit des Merge-Schrittes verbessert sich dadurch zu $\mathcal{O}(n + \alpha\delta)$.

2.4.2 OPT-NB

OPT-NB ist eine Abwandlung von OPT, bei der auf den Puffer verzichtet wird („no buffering“). Grund hierfür sind Beobachtungen, dass bei der Ausführung von OPT bei der Speicherverwaltung ein großer Mehrauwand

NAIVE	FAST	OPT	OPT-NB
$\mathcal{O}(\delta n \log n)$	$\mathcal{O}(\delta n \log n + \alpha\delta^2)$	$\mathcal{O}(\delta n \log n + \alpha\delta)$	$\mathcal{O}(\delta n \log n + \alpha\delta)$

Tabelle 1: Die Laufzeiten der vorgestellten Mergesort-Algorithmen

durch den Puffer entsteht. Wie zuvor schon in FAST werden geeignete Indizes verwendet und mit diesen die Berechnungen direkt in der Eingabe bzw. Ausgabe vorgenommen, statt die Schlüssel zuerst in Hilfspuffern zwischen zu speichern. Die Laufzeit eines Merge-Schrittes ist gleich der von OPT.

3 Experimente

3.1 Setup

Alle nachfolgenden Experimente wurden auf einem PC mit zwei Opteron-Prozessoren (2Ghz, 64Bit) und 2GB RAM durchgeführt [2, Abschnitt 3,4].

In einem Fehler-Injektions-Verfahren wurden bei den 5 - 20 Millionen Schlüsseln (n) immer so viele Fehler injiziert, bis $\alpha = \delta$. Zur Fehlerinjektion wurde jeweils während der Ausführung der Wert eines Schlüssels erhöht.

3.2 Ergebnisse

Zum Vergleich wurde eine (nicht-fehlertolerante) iterative Variante von Mergesort namens VANILLA hinzugezogen. Der Anspruch an die fehlertoleranten Algorithmen muss hier sein, dass sie nicht nur 0-ungeordnete Sequenzen ausgeben, sondern auch nicht wesentlich langsamer sind beziehungsweise nicht abhängig von der Größe der Eingabe langsamer werden im Vergleich zur nicht-fehlertoleranten Variante.

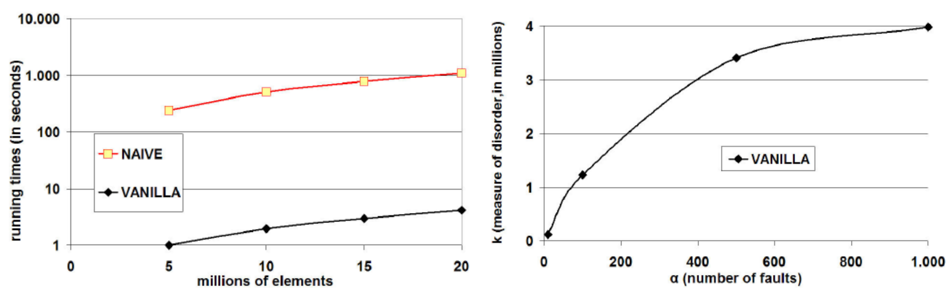

Abb. 3: Links: Speicherfehler unter VANILLA. Rechts: Vergleich der Ausführungszeiten von VANILLA und Naive-Mergesort.

3.2.1 VANILLA und Naive-Mergesort

In den einleitenden Experimenten wird klar, dass die naive fehlertolerante Variante im Vergleich zur nicht-fehlertoleranten wesentlich langsamer ist (Abb. 3, links). VANILLA gibt nach weniger als 8s (Sekunden) sein Ergebnis zurück, Naive-Mergesort erst nach über 1000s. Die Ergebnisse sind jedoch von sehr unterschiedlicher Qualität: Während Naive-Mergesort immer eine 0-ungeordnete Menge zurückliefert, gibt VANILLA eine äußerst ungeordnete Menge zurück (siehe Abb. 3, rechts). Die Diskrepanzen zwischen den Ordnungen der Mengen und den Ausführungszeiten der Algorithmen unterstreichen den Bedarf nach ausgereifteren, fehlertoleranten Mergesort-Algorithmen.

3.2.2 Vergleich der ausgereifteren Algorithmen

Die oben vorgestellten ausgereifteren, fehlertoleranten Algorithmen FAST, OPT und OPT-NB können sich durchaus an VANILLA messen: Alle berechnen bei 20 Millionen Elementen und 500 gleichmäßig verteilten Speicherfehlern ihre Ausgabe in weniger als 15s (Abb. 4, links). Approximativ sind die fehlertoleranten Algorithmen 2,5 bis 3,0 mal langsamer als VANILLA bei gleichbleibendem α.

Belässt man nun die Anzahl der Schlüssel und variiert über die Anzahl der Fehler α (und deren Maximum δ), so ist zu beobachten, dass FAST als einziger Algorithmus langsamer wird, während die anderen gleich schnell sortieren (Abb. 4, rechts). Bei näherer Untersuchung zeigt sich, dass FAST sehr viel Zeit mit Naive-Mergesort verbringt. Dessen Eingabe ist die Ausgabe D von Purify, die in der Größe $\mathcal{O}(k+\alpha)$ liegt und in $\mathcal{O}(n+\delta(k+\alpha))$ berechnet wird (siehe 2.2.2).

Wie unter 2.4 erwähnt, berechnen OPT und OPT-NB mit der Subroutine „Purifying-merge“ die ungeordnete Sequenz D in nur $\mathcal{O}(n+\alpha\delta)$. Weiter liegt diese nur in der Größe $\mathcal{O}(\alpha)$, die anschließend noch von Naive-Mergesort

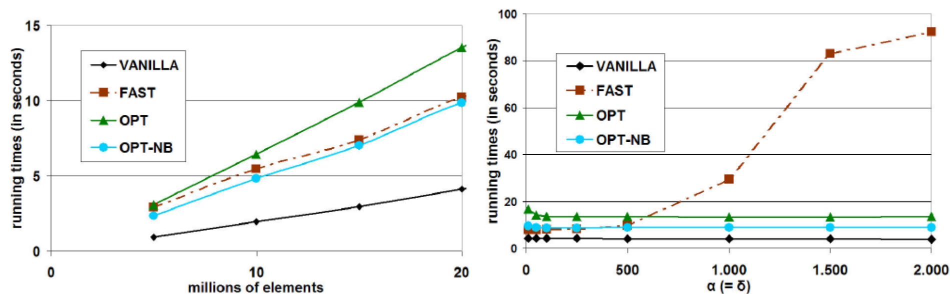

Abb. 4: Links: Vergleich der Ausführungszeiten mit $\alpha = \delta = 500$ bei variierender Anzahl an Schlüsseln. Rechts: Versuche mit unterschiedlicher Anzahl von Fehlern bei 20 Millionen Schlüsseln.

sortiert werden muss, was den Vorteil gegenüber FAST erklärt.

3.2.3 Temporäre Fehlerinjektion

Bei den vorherigen Experimenten wurden die Speicherfehler gleichmäßig über die Ausführungszeit verteilt. In Anbetracht der in der Einleitung erklärten Ursachen für Speicherfehler, ist ein zeitlich konzentriertes Vorkommen von Speicherfehlern nicht unwahrscheinlich. Deshalb wurden in zwei Versuchen die Fehler in den vorderen bzw. hinteren 20% der Ausführungszeit injiziert. In beiden Versuchen werden jeweils 20 Millionen Schlüssel sortiert und über α $(= \delta)$ variiert. Bei der frühen Fehlerinjektion (Abb. 5, links) zeigt sich keine Veränderung gegenüber dem vorigen Versuch. Im Zweiten Versuch jedoch, mit der späteren Fehlerinjektion, verlangsamt sich FAST rapide während die OPT und OPT-NB weiter in gleicher Zeit sortieren(Abb. 5, rechts). Während sonst alle Versuche höchstens 15s benötigen, braucht FAST bei 2000 Speicherfehlern über 250s.

Zu Beginn der Ausführung werden viele kleine Sequenzen sortiert und am Ende nur wenige aber große. Deshalb führen die früh-injizierten Fehler bei der Ausführung von Purify nur zu kleinen Fehlersequenzen (D) und die spät-injizierten entsprechend zu größeren. In Verbindung mit der Begründung aus dem vorigen Versuch erklärt sich so die starke Verlangsamung von FAST bei der Fehlerinjektion in den hinteren 20%.

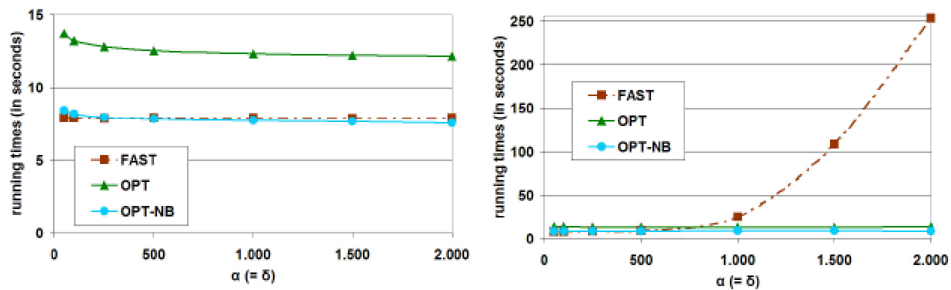

Abb. 5: Konzentration der Fehler in den vorderen 20% der Ausführungszeit des Algorithmus (linkes Bild), bzw. in den hinteren 20% (rechtes Bild).

Literatur

[1] I. Finocchi and G. F. Italiano. Sorting and searching in the presence of memory faults (without redundancy). Proc. 36th STOC, 101–110, 2004.

[2] U. Ferraro-Petrillo, I. Finocchi and G. F. Italiano. The price of resiliency: A case study on sorting with memory faults. In Proceedings of the 14th Conference on Annual European Symposium (ESA'06), 2006.

Abbildungsverzeichnis

1 Beispiel: Speicherfehler bei Mergesort 2
2 Ablaufschema des FAST-Algorithmus 3
3 Bilder der Experimente. Quelle: [2] 6
4 Bilder der Experimente. Quelle: [2] 7
5 Bilder der Experimente. Quelle: [2] 8

Tabellenverzeichnis

1 Die Laufzeiten der vorgestellten Mergesort-Algorithmen . . . 6