Flaschenhalsfreie parallele Priority queues

Peter Sanders
Lehrstuhl Informatik für Ingenieure und Naturwissenschaftler
Universität Karlsruhe, 76128 Karlsruhe
E-mail: `sanders@ira.uka.de`

Zusammenfassung

Priority queues (PQs) – abstrakte Datentypen, die das Einfügen von Elementen sowie das Entnehmen des kleinsten Elements unterstützen, – sind ein zentrales Element in der Ablaufsteuerung vieler Algorithmen (z.B. Branch-and-bound). Sollen sich nun die Prozessoren eines Parallelrechners eine gemeinsame PQ teilen, so ergibt sich das Problem, daß eine zentrale Verwaltung zu einem Flaschenhals wird. Bisher bekannte parallele Algorithmen, die diesen Flaschenhals vermeiden, sind für PRAMs formuliert und dürften sich nur schwer und unter Leistungsverlust auf praktikablere Architekturen übertragen lassen. Hier wird ein randomisierter Algorithmus vorgestellt und analysiert, der auf vielen Architekturen effizient realisierbar ist und für die Ausführung von n PQ-Operationen (n sei die Prozessorzahl) im wesentlichen solange braucht wie für das Sortieren von $O(n)$ Werten. Daneben wird eine Reihe von Variationsmöglichkeiten vorgestellt, die es erlauben, parallele PQs einer verschiedenen Anwendungsszenarien anzupassen.

1 Einführung

Priority queues (PQs) sind abstrakte Datentypen, die das Einfügen von Elementen (`insert`) sowie das Entnehmen des kleinsten Elements (`deleteMin`) unterstützen. Für diese Funktionalität gibt es eine einfache heap-basierte sequentielle Implementierung, die die Grundoperationen in Zeit $\Theta(\log m)$ für eine PQ mit m Elementen realisiert.

PQs lassen sich zur Ablaufsteuerung von Algorithmen verwenden, bei denen Teilprobleme dynamisch entstehen und jeweils die wichtigsten zuerst abgearbeitet werden sollen. Ein wichtiges und typisches Beispiel, das im folgenden des öfteren verwendet wird, sind best-first Branch-and-bound Suchalgorithmen: Teilprobleme sind Knoten eines Suchbaums. Es steht eine heuristische Funktion zur Verfügung, die eine untere Schranke für die Kosten einer aus dem Knoten ableitbaren Lösung berechnet. Die Strategie von best-first Branch-and-bound besteht nun darin, die Knoten mit der jeweils kleinsten Bewertung aus einer PQ zu entnehmen, die Nachfolger zu bestimmen und entsprechend ihrer Bewertung in die PQ einzufügen.

Da die oben genannten Anwendungen gerade auch für parallele Algorithmen interessant sind, stellt sich nun die Frage, wie PQs sich am besten parallel implementieren lassen. Das einfachste Verfahren ist, einen Prozessor (PE – Processing Element) mit der Verwaltung eines zentralen Heaps zu beauftragen. Dies funktioniert jedoch nur für kleine PE-Zahlen oder wenn PQ-Operationen selten benötigt werden, da der Verwaltungsprozessor sonst zu einem Flaschenhals wird.

Frühe Versuche der Parallelisierung haben jeder Ebene des Heaps eine PE einer EREW-PRAM (Exclusive Read Exclusive Write Parallel Random Access Machine) zugeordnet. Aber selbst mit diesem mächtigen Maschinenmodell kann nur eine konstante Zahl von Anforderungen pro Zeiteinheit durchgeführt werden, weil die Spitze des Heaps, auf die für jede Anforderung zugegriffen wird, weiterhin ein Flaschenhals ist. In der Praxis stellen sich diese Algorithmen oft als langsamer heraus als der zentrale Ansatz [11].

Einen Ausweg bietet der *Parallel heap* Algorithmus von Deo und Prasad [2]. Hier werden in jedem Blatt des Heaps r Elemente gespeichert und aus den Vergleichs- und Tauschoperationen der üblichen Heap-Verwaltung werden Sortier- und Mischoperationen, für die es effiziente parallele Implementierungen gibt. Durch geschickte Wahl von r und Aufteilung der n PEs auf die Ebenen des parallelen Heaps ist es möglich, $\Theta(n)$ Operationen in Zeit $\Theta(\log m)$ zu erledigen. Der Algorithmus macht allerdings intensiven Gebrauch von den Fähigkeiten einer PRAM und komplexen, oft nur theoretisch sinnvollen Sortier- und Mischalgorithmen. Er läßt sich deshalb nur unter deutlichem Leistungsverlust in die Praxis umsetzen.

Ein ganz anderer Ansatz besteht darin, jeder PE eine lokale PQ zuzuordnen, zwischen denen nur einzelne Elemente ausgetauscht werden [8, 4, 5, 7, 3]. Dies führt zwar zu einem hohen Durchsatz an Anforderungen, aber es kann nicht mehr garantiert werden, daß die global kleinsten Elemente zuerst entfernt werden.

Hier werden Algorithmen vorgestellt, die echte parallele PQs implementieren und auf einem weiten Spektrum von Maschinen praktikabel sind. Dazu werden in Abschnitt 2 zunächst grundlegende Begriffe eingeführt. Dann beschreibt Abschnitt 3 einen sehr einfachen Ansatz, der auf parallelem Sortieren beruht. Er kann als Spezialfall von Parallel heap aufgefaßt werden, ist aber flexibler einsetzbar. Da der einfache Ansatz für große PQs zu langsam ist, wird in Abschnitt 4 ein randomisiertes Verfahren entwickelt, das zwar auf einer PRAM etwas langsamer ist als Parallel heap, sich aber auf beliebigen Maschinen effizient realisieren läßt. Außerdem gibt es Varianten des Algorithmus, die einen kontinuierlichen Übergang zwischen einer echten PQ und den approximierten aber effizienteren Implementierungen aus [8, 4, 5, 7, 3] erlauben. Schließlich gibt Abschnitt 5 eine Zusammenfassung der Ergebnisse und einen Ausblick auf zukünftige Fragestellungen.

2 Grundlagen

2.1 Paralleles Maschinenmodell

Unter einem Parallelrechner wird im folgenden ein nachrichtengekoppelter MIMD-Rechner mit n identischen PEs verstanden, die durch ein Verbindungsnetzwerk mit Durchmesser $d(n)$ verbunden sind. Die vorgestellten Algorithmen nutzen die Struktur des Verbindungsnetzwerkes nicht explizit aus, sondern sind zusammengesetzt aus Aufrufen von Standardroutinen wie Broadcast (Senden einer Nachricht an alle PEs), globale Synchronisation, Routen einer Permutation, Reduktion (z.B. globale Summierung eines Wertes oder Minimumbildung), Prefixsummenbildung (PE i berechnet $\sum_{j=1}^{i} x_j$) und Sortieren. Aus den Laufzeiten dieser Standardverfahren kann eine Analyse wie in einem Baukastensystem für beliebige Architekturen zusammengefügt werden.[1] Sei $s(n) \in \Omega(d(n))$ der Zeitbedarf für das Sortieren von n Werten. Zur Vermeidung aufwendiger Notation wird im folgenden angenommen, daß die anderen genannten Operationen in Zeit $O(d(n))$ durchführbar sind. Ferner sei es möglich, für Broadcast und Reduktion Pipelining anzuwenden, k Operationen sollen also in Zeit $O(d(n)+k)$ durchführbar sein.

[1] Mindestens genauso wichtig ist natürlich, daß dieses Baukastenprinzip auch eine portable Implementierung erlaubt.

2.2 Semantik paralleller PQs

Auf einem MIMD-Rechner kann selbst eine zentral verwaltete PQ nicht garantieren, daß niedrig bewertete Elemente vor höher bewerteten abgearbeitet werden. Selbst die Reihenfolge des Beginns der Bearbeitung kann oft nicht garantiert werden, da Unterschiede in der Laufzeit von Nachrichten nicht kontrollierbar sind. Was also bleibt im parallelen Fall von der Semantik einer sequentiellen PQ? Dem Auftraggeber einer `insert`-Operation kann zugesichert werden, daß das abgelieferte Element irgendwann in die PQ aufgenommen wird. Das Ergebnis einer `deleteMin`-Operation ist ein Element[2], das irgendwann durch ein `insert` eingefügt wurde und keiner anderen `deleteMin`-Anforderung zugeteilt wird. Vor allem aber ist garantiert, daß zum Zeitpunkt der Zuordnung des Elements zur `deleteMin`-Anforderung alle Elemente der PQ mit kleinerer Bewertung ebenfalls zugeordnet sind. Steht eine hinreichend synchronisierte Uhr zur Verfügung, so ist es außerdem möglich, früher abgesetzten `deleteMin`-Anforderungen kleinere PQ Elemente zuzuordnen, solange zwischendurch keine neuen Elemente eingefügt werden.

Zur Vereinfachung der Analyse wird im folgenden davon ausgegangen, daß sich sowohl eine Anforderung als auch die Antwort auf eine `deleteMin`-Anforderung als Nachricht konstanter Größe repräsentieren läßt. In der Tat wird es sich oft anbieten, nicht die u.U. recht großen Elemente (z.B. Suchbaumknoten) selbst in der PQ zu verwalten, sondern nur Referenzen darauf.

2.3 Analyse randomisierter Algorithmen

Die hier verwendete Konvention zur Quantifizierung des Verhaltens randomisierter paralleler Algorithmen ist das Verhalten *mit hoher Wahrscheinlichkeit*:

Definition 1 ([10]) *Eine von einem Parameter n abhängige Zufallsvariable $X(n)$ ist in* $O(f(n))$ mit hoher Wahrscheinlichkeit *(kurz $X(n) \in \tilde{O}\left(f(n)\right)$) gdw.*

$$\exists c > 0, n_0 > 0 : \forall \beta \geq 1, n \geq n_0 : \mathbf{P}\left[X(n) > c\beta f(n)\right] \leq n^{-\beta}.$$

Aussagen über das Verhalten mit hoher Wahrscheinlichkeit sind nicht nur i.allg. stärker als Aussagen über das Verhalten im Mittel, sondern sondern es gibt auch eine Reihe nützlicher Aussagen, die Analysen erleichtern. Hier werden die folgenden beiden Sätze benötigt:

Satz 1 (Chernoff Schranken [1, 10, 6]) *Die Zufallsvariable X repräsentiere die Anzahl der Erfolge bei l unabhängigen Bernoulli-Versuchen mit Erfolgswahrscheinlichkeit p. Dann gilt*

$$\mathbf{P}\left[X \leq (1-\epsilon)lp\right] \leq e^{-\epsilon^2 lp/3} \text{ für } 0 < \epsilon < 1 \tag{1}$$

$$\mathbf{P}\left[X \geq \alpha lp\right] \leq e^{\left(1-\frac{1}{\alpha}-\ln\alpha\right)\alpha lp} \text{ für } \alpha > 1 \tag{2}$$

.

Satz 2 ([12]) *Seien $X_1(n) \in \tilde{O}\left(f_1(n)\right), \ldots, X_m(n) \in \tilde{O}\left(f_m(n)\right)$ Zufallsvariablen, wobei m höchstens polynomiell in n ist. Dann gilt*

$$\max_{i=1}^{m} X_i(n) \in \tilde{O}\left(\max_{i=1}^{m} f_i(n)\right).$$

Ein wichtiger Spezialfall ergibt sich, wenn alle f_i gleich f sind. Dann gilt einfach $\max_{i=1}^{m} X_i(n) \in \tilde{O}\left(f(n)\right)$, d.h. Maximumbildung von polynomiell vielen identisch verteilten Zufallsvariablen kann man „unter den Tisch fallen“ lassen.

[2] Der Sonderfall einer leergelaufenen PQ ist auch interessant, wird hier aber der Einfachheit halber nicht näher betrachtet.

3 „Kleine“ asynchrone PQs

Erste Voraussetzung zur Vermeidung von Flaschenhälsen bei der Verwaltung von parallelen PQs ist eine verteilte Speicherung der Elemente. Dies allein reicht aber noch nicht. Zusätzlich müssen die Anforderungen so koordiniert werden, daß es zu keinen Stauungen kommt. Die Schlüsselidee dazu ist, Anforderungen nicht unmittelbar durchzuführen, sondern für eine Weile lokal zu lagern. Gelegentlich werden alle PEs synchronisiert und arbeiten die aufgelaufenen Anforderungen dann kollektiv ab. Im folgenden wird zunächst ein vereinfachter Algorithmus beschrieben und analysiert. Insbesondere wird davon ausgegangen, daß die PQ nie mehr als n Elemente enthält.

1. Es werden Anforderungen gesammelt bis $O(n)$ `deleteMin`-Anforderungen aufgelaufen sind oder ein maximaler Zeitraum $O(s(n))$ verstrichen ist. Durch wiederholtes Zählen der aufgelaufenen Anforderungen verstreicht maximal eine Zeit in $O(d(n))$, bis diese Bedingung erkannt wird. Da das Zählen nur den $O\left(\frac{1}{d(n)}\right)$-ten Teil der während dieser Zeit ausgeführten Operationen in Anspruch nimmt, hält es die eigentlichen Berechnungen nicht nennenswert auf.[3]

2. Erkläre alle neu eingefügten Elemente zu PQ Elementen. Da jede PE in Phase 1 maximal $O(s(n))$ Anforderungen abgesetzt hat, geht dies in Zeit $O(s(n))$.

3. Nun wird das i-t kleinste PQ-Element PE Nummer i zugeordnet. Die neuen PQ-Elemente seien so gleichmäßig verteilt, daß diese Sortierung in Zeit $O(s(n))$ möglich ist.[4]

4. Die k `deleteMin`-Anforderungen werden nun den PEs $1, \ldots, k$ zugeordnet. Wenn die Anforderungen hinreichend gleichmäßig verteilt sind, kann diese „Sammeloperation“ in Zeit $O(d(n) + s(n)) \subseteq O(s(n))$ durchgeführt werden. (Der Bestimmungsort der Anforderungen kann durch Prefixsummenbildung über die lokale Anzahl der Anforderungen bestimmt werden.)

5. Jede PE, die sowohl ein PQ-Element als auch eine `deleteMin`-Anforderung enthält, schickt das PQ-Element an den Absender der `deleteMin`-Anforderung und entfernt beides.

6. Verbleibende `deleteMin`-Anforderungen werden in die nächste Phase übernommen.

Dieser Zyklus wird bis zur Terminierung des Programms fortgesetzt.

Jede der beschriebenen Phasen kommt mit einer Zeit in $O(s(n))$ aus. (Setzt keine PE mehr als $O(d(n))$ Operationen ab, so kommen alle Phasen außer der Sortierung in Phase 3 mit Zeit $T(n) = O(d(n))$ aus.) Insbesondere gilt in einem u-dimensionalen Gitter $T(n) \in O(n^{1/u})$. Zum Vergleich: Ein zentral verwalteter Heap braucht im günstigsten Fall sehr seltener Anforderungen ebenfalls Zeit $O(n^{1/u})$, bei hoher Last aber Zeit $\Omega(n \log n)$. Auf Netzwerken mit logarithmischem Durchmesser ist der dominierende Term der Aufwand für die Sortierung, da dafür in der Praxis Zeit $\Omega((\log n)^2)$ zu veranschlagen ist.

---

[3] Wenn Reduktionsoperationen synchron arbeiten oder Aufsetzzeiten eine große Rolle spielen, muß allerdings nach jeder Zähloperation eine Pause eingelegt werden, damit die eigentlichen Berechnungen zum Zuge kommen.

[4] Liegen viele PEs mit einer besonders großen Anzahl Anforderungen dicht beieinander, so kann dadurch ein Flaschenhals entstehen, der aber von der PQ Implementierung nicht verhindert werden kann und den wir deshalb der Anwendung anlasten.

Der hier vorgestellte Algorithmus ab Phase 2 ist im Prinzip ein datenparalleler Algorithmus. In der Tat wurden ähnliche Verfahren schon zur dynamischen Lastverteilung für parallele Tiefensuche auf SIMD-Rechnern wie der TMC-CM2 und der MasPar MP-1 eingesetzt [9, 13]. Dort gibt es allerdings keine explizite PQ und nur wenige mögliche Elementbewertungen.

3.1 Verfeinerungen

Es gibt eine ganze Anzahl Varianten des Algorithmus aus Abschnitt 3, die ihn effizienter oder flexibler machen.

- In sequentiellen PQs werden den zuerst ankommenden `deleteMin`-Anforderungen die kleinsten Elemente zugeordnet. Im verteilten Fall kann dieses first-come-first-serve Prinzip angenähert werden, indem `deleteMin`-Anforderungen mit einem (lokalen) Zeitstempel versehen und nach diesem sortiert werden.

- Auf MIMD-Rechnern können Berechnungen sowie die einzelnen Phasen überlappt werden. Zum Beispiel können `deleteMin`-Anforderungen eines neuen Zyklus bereits sortiert werden, während der vorhergehende Zyklus noch Elemente zuordnet. Dies führt zu einer gleichmäßigeren Auslastung des Kommunikationsnetzes und trägt der Tatsache Rechnung, daß viele moderne Parallelrechner autonom arbeitende Kommunikationsprozessoren enthalten.

- Indem jede PE mehrere virtuelle PEs emuliert, können auch PQs mit mehr als n Elementen verwaltet werden. Allerdings sinkt dadurch die Leistung deutlich ab, weshalb in Abschnitt 4 ein Algorithmus entwickelt wird, der auch für große PQ ähnlich effizient ist wie im hier beschriebenen einfachen Fall.

4 „Große“ PQs

In diesem Abschnitt wird ein Algorithmus betrachtet, der auch für PQ-Größen $m \gg n$ sinnvoll einsetzbar ist. Um die Darstellung und Analyse nicht unnötig zu komplizieren, wird von nun an von einem einfachen synchronen Modell der Anwendung ausgegangen: Jede PE holt genau ein Element aus der PQ, bearbeitet es und fügt dann eine Anzahl neuer Elemente ein. Ist nicht für jede PE ein Element vorhanden, so erhalten einige PEs ein „dummy“-Element mit Bewertung ∞.

Die Grundidee besteht nun darin, auf jeder PE eine lokale PQ zu verwalten, die parallele `deleteMin`-Operation aber so zu realisieren, daß die global besten Elemente identifiziert werden. Offenbar ist dies im schlimmsten Fall ein sehr aufwendiges Unterfangen, da nicht auszuschließen ist, daß die gesuchten Elemente auf einer einzigen PE konzentriert sind. Diese ungünstigen Fälle lassen sich aber – unabhängig von der Anwendung – sehr unwahrscheinlich machen, indem neu einzufügende Elemente einer zufällig ausgewählten lokalen PQ zugeordnet werden. Für eine PQ der Größe m ist dies in Zeit $O(d(n) + \log \frac{m}{n})$ möglich. (Elemente versenden und in lokalen Heap einfügen.)

Abbildung 1 zeigt eine Pseudocode Beschreibung der parallelen `deleteMin`-Operation. Aus jeder der (als Heap implementierten) lokalen PQs werden $\alpha \log n$ Elemente[5] entfernt und in einem Puffer aufbewahrt. Dies wird solange wiederholt, bis mindestens n der entnommenen Elemente kleiner sind als das Minimum aller in den Heaps verbliebenen Elemente. Dies ist

[5] α ist eine geeignet zu wählende Konstante, die für die asymptotische Betrachtung nicht weiter von Bedeutung ist.

```
(* Assign the n smallest of m PQ Elements to PEs 1...n *)
DOPAR
    h: Heap OF Element
    b: Set OF Element (* Buffer *)
    e, e̲: Element
    REPEAT
        remove α log n elements from h
        and put them into b
        e̲ := reduceMin(readMin(h))
    UNTIL reduceAdd(|{e ∈ b : e < e̲}|) ≥ n
    Assign the n smallest elements from all bs to PEs 1 through n
    reinsert the remaining elements of b into h
```

Abbildung 1: Parallele `deleteMin`-Operation.

eine notwendige und hinreichende Bedingung dafür, daß die gesuchten n kleinsten Elemente sich im Puffer befinden. In Abschnitt 4.1 wird gezeigt, daß mit hoher Wahrscheinlichkeit eine konstante Zahl von Iterationen durch die **REPEAT**-Schleife ausreicht. Es ist also möglich, in Zeit $\tilde{O}\left(d(n) + (\log \frac{m}{n})(\log n)\right)$ die Anzahl der zu betrachtenden Elemente auf $\tilde{O}(n \log n)$ zu reduzieren. Im Prinzip könnte nun analog zu Abschnitt 3 ein Sortierverfahren eingesetzt werden, um die n größten Elemente ausfindig zu machen. In Abschnitt 4.2 wird sich jedoch herausstellen, daß es eine noch schnellere Möglichkeit gibt: Eine Zeit in $\tilde{O}(s(n) + (\log n)^2)$ reicht zum Identifizieren und Zuordnen der n besten Elemente. Das Wiedereinfügen der nichtbenötigten Elemente schließlich kann nicht länger dauern als das vorherige Herausnehmen. Insgesamt ergibt sich eine Zeit von $\tilde{O}\left(s(n) + \log n(\log \frac{m}{n} + \log n)\right)$ für eine parallele `deleteMin`-Operation. Für die meisten praktisch relevanten Fälle (m polynomiell in n und $s(n) \in \Omega((\log n)^2)$) ist dies asymptotisch genauso schnell wie der Fall kleiner PQs aus Abschnitt 3.

4.1 Anzahl der zu betrachtenden Elemente

Über das Verhalten der **REPEAT**-Schleife aus Abbildung 1 läßt sich folgende Aussage beweisen:

Satz 3 *Die Zufallsvariable $X(n)$ bezeichne die Anzahl Elemente, die aus jedem lokalen Heap entnommen werden müssen, um die n kleinsten Elemente zu erfassen. Es gilt*

$$X(n) \in \tilde{O}(\log n).$$

Beweis: OBdA seien alle Elemente der PQ verschieden; dann sind die n kleinsten Elemente der PQ eindeutig bestimmt. Durch die Randomisierung werden diese n Elemente unabhängig voneinander und zufällig auf die n Heaps verteilt. Vom Standpunkt einer festen PE i werden n unabhängige Bernoulliexperimente mit Erfolgswahrscheinlichkeit $1/n$ durchgeführt (Die Zuteilung eines der minimalen Elemente zähle als „Erfolg“). Sei $X_i(n)$ die Anzahl der kleinsten Elemente, die in PE i landen. Nach Satz 1 (Gleichung (2)) gilt

$$\mathbf{P}\left[X_i(n) \geq \beta \ln n\right] \leq e^{\left(1 - \frac{1}{\beta \ln n} - \ln(\beta \ln n)\right)\beta(\ln n)\left(n\frac{1}{n}\right)} \leq e^{(1 - \ln\ln n)\beta \ln n} \leq e^{-\beta \ln n} = n^{-\beta}$$

für $\beta \geq 1$ und $\ln\ln n \geq 2$. Also $X_i(n) \in \tilde{O}(\log n)$. Der gesuchte Wert von $X(n)$ ist das Maximum der X_i, und Satz 2 liefert

$$X(n) \in \tilde{O}(\log n).$$

■

4.2 Schnelle Bestimmung der n kleinsten Elemente

Sei B die Menge aller durch Algorithmus 1 aus den lokalen PQs entnommenen Elemente. In der letzten Phase dieses Algorithmus gilt es, die n kleinsten aus den lokalen PQs entnommenen Elemente aus B zu bestimmen. Nach Abschnitt 4.1 sind dies mit hoher Wahrscheinlichkeit $O(n \log n)$ viele. Nur von diesen ist im folgenden noch die Rede. Offensichtlich ist es verschwenderisch, alle Elemente zu sortieren, wenn nur ein Bruchteil tatsächlich benötigt wird. Dies ist auch nicht nötig, denn der größte Teil der nichtbenötigten Elemente läßt sich schnell aussondern:

1. Sei $|B| = bn \ln n$. Bestimme $b(\ln n)^2$ zufällig gewählte Elemente[6] und sortiere sie. Seien $a_1, \ldots, a_{b(\ln n)^2}$ die soeben sortierten Elemente. Da die Elemente unabhängig voneinander zufällig plaziert wurden, reicht es, daß $O(\log n)$ benachbarte PEs die lokal vorhandenen Elemente durch Mischoperationen zusammenfügen. Der Zeitbedarf ist dann in $O((\log n)^2)$.

2. Bestimme *Partitionierungselemente*[7] $e_0 = -\infty, e_1 = a_{\ln n}, e_2 = a_{2\ln n}, \ldots, e_{b\ln n-1} = a_{b(\ln n)^2-\ln n}, e_{b\ln n} = \infty$. Zeitbedarf $O(\log n)$.

3. Übermittle die Partitionierungselemente an alle PEs. Unter Verwendung von Broadcast mit Pipelining geht das in Zeit $O(d(n) + \log n)$.

4. Partioniere B in Partitionen $B_1, \ldots, B_{b\ln n}$, so daß $B_j = \{m \in B : e_{j-1} \leq m < e_j\}$. Da sowohl die lokal vorhandenen Elemente als auch die Partitionierungselemente sortiert sind, reicht dazu ein einziger Durchlauf der beiden Felder. (Es werden keine Elemente bewegt, sondern lediglich die Zugehörigkeit zu einer Partition festgehalten. Der Zeitbedarf hierfür ist in $O(\log n)$.

5. Bestimme ein i, so daß $|B_1 \cup \cdots \cup B_{i-1}| \leq n$ und $|B_1 \cup \cdots \cup B_i| > n$. Dies läßt sich effizient realisieren, indem die $|B_j|$ für $j = 1 \ldots b\ln n$ durch $b\ln n$ Reduktionsoperationen bestimmt werden, die durch Pipelining ineinander verschränkt sind. Zeitbedarf: $O(d(n) + \log n)$.

6. Die Elemente in $B_1 \cup \cdots \cup B_{i-1}$ gehören in jedem Fall zu den n kleinsten und können bereits verteilt werden. $O(d(n) + \log n)$.

7. Sortiere die Elemente von B_i und verteile die $n - |B_1 \cup \cdots \cup B_{i-1}|$ kleinsten. Im folgenden wird gezeigt, daß $|B_i| \in \tilde{O}(n)$ und sich folglich ein Zeitbedarf in $O(s(n))$ ergibt.

Insgesamt benötigt das Identifizieren der n kleinsten Elemente eine Zeit in $\tilde{O}(s(n) + (\log n)^2)$.

Abbildung 2 stellt die Situation nach der Bestimmung von B_i dar. In einer gedachten sortierten Anordnung der betrachteten Elemente (die eindeutig ist wenn die Elemente als verschieden angenommen werden) sind die Elemente im dunkel schraffierten Bereich bereits identifiziert und können verteilt werden. Die Elemente im unschraffierten Bereich gehören garantiert nicht zu den gesuchten.

---

[6] Die plötzliche Verwendung des natürlichen Logarithmus dient nicht der Verwirrung des Lesers, sondern der Vereinfachung der nachfolgenden Analyse. Aus dem gleichen Grunde werden auch die im Prinzip nötigen Auf- und Abrundungen weggelassen. Es ist im übrigen nicht zu erwarten, daß das asymptotische Verhalten des Algorithmus sich ändert, wenn in einer realen Implementierung andere konstante Faktoren gewählt werden.

[7] Ähnliche Ideen finden sich in vielen anderen Arbeiten. Zum Beispiel wird in [14] beschrieben, wie durch Sortieren einer Probe (Sample) von n^ϵ aus n Elementen ein ungefährer Median bestimmt werden kann.

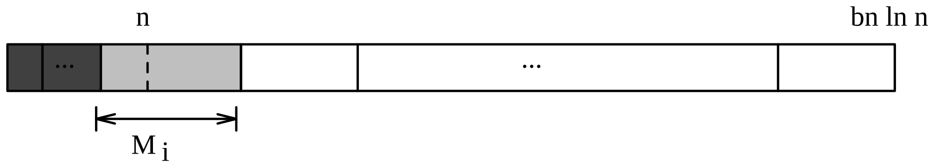

Abbildung 2: Situation nach Bestimmung von B_i

Satz 4 $|B_i| \in \tilde{O}(n)$

Beweis: Es gilt zu zeigen, daß die Länge des Intervalls B_i mit Wahrscheinlichkeit $1 - n^{-\beta}$ nicht größer ist als $c\beta n$ für $\beta \geq 1$, eine geeignet gewählte Konstante c und hinreichend große n. Das Ereignis $|B_i| > c\beta n$ kann nur dann eintreten, wenn in Schritt 1 weniger als $\ln n$ Elemente aus B_i ausgewählt wurden. (Sonst würde in Schritt 2 eine Intervallgrenze aus B_i gewählt.) Die Wahrscheinlichkeit dafür läßt sich durch die Wahrscheinlichkeit abschätzen, daß bei $b(\ln n)^2$ Bernoulliexperimenten mit Erfolgswahrscheinlichkeit $\frac{c\beta n}{bn \ln n} = \frac{c\beta}{b} \ln n$ weniger als $\ln n$ Erfolge erzielt werden. (Gleichheit läge dann vor, wenn die Elemente in Schritt 1 „mit Zurücklegen" ausgewählt würden.) Die Zufallsvariable X bezeichne nun die Anzahl der Erfolge bei den oben beschriebenen Bernoulliexperimenten. Dann gilt

$$P := \mathbf{P}\left[|B_i| > c\beta n\right] \leq \mathbf{P}\left[X < \ln n\right]$$

Durch Anwendung von Satz 1 (Gleichung (1), $\epsilon = \left(1 - \frac{1}{c\beta}\right)$) ergibt sich daraus

$$P \leq \exp\left[-\left(1 - \frac{1}{c\beta}\right)^2 \frac{c\beta \ln n}{3}\right].$$

Unter Ausnutzung von $\beta \geq 1$ kann weiter abgeschätzt werden zu

$$P \leq \exp\left[-\left(1 - \frac{1}{c}\right)^2 \frac{c\beta \ln n}{3}\right] = n^{-\beta\left(1-\frac{1}{c}\right)^2 \frac{c}{3}} \leq n^{-\beta}$$

für $c = 4.8$. ■

4.3 Verfeinerungen

Wieder gibt es eine ganze Reihe von Möglichkeiten, den Grundalgorithmus zu verbessern und wechselnden Anforderungen anzupassen:

- Die Ideen aus Abschnitt 3 lassen sich auf den neuen Algorithmus übertragen. Insbesondere können auch asynchron erzeugte Anforderungen bearbeitet werden.
- Die bei der Minimumbildung in Algorithmus 1 gewonnene Information kann genutzt werden, um Elemente, die größer als das zuletzt berechnete Minimum sind, unmittelbar auszuschließen und umgekehrt Elemente, die kleiner als das zweitletzte berechnete Minimum sind, unmittelbar zuzuordnen.
- Die meisten der aus den lokalen Heaps entnommenen Elemente werden nicht zugeordnet, sondern wieder eingefügt. Dies läßt sich beschleunigen, indem die $O(\log \frac{m}{n})$ lokal kleinsten PQ Elemente in einem sortierten Feld gespeichert werden, das gewissermaßen als Puffer wirkt. Dadurch ist es möglich, die Mehrzahl der Heap-Zugriffe einzusparen.

- In einer realen Implementierung könnte sich herausstellen, daß ein unverhältnismäßig hoher Aufwand getrieben wird, um den letzten Rest der in Intervall B_i gelegenen Elemente zuzuordnen. In einigen Fällen könnte es deshalb sinnvoll sein, diese Elemente überhaupt nicht zuzuordnen, sondern lieber einige `deleteMin`-Anforderungen zurückzustellen und erst in der nächsten Phase zu berücksichtigen.

- Die Idee der Partitionierung läßt sich ausbauen. Zum Beispiel könnten auf einem 2D Gitter $O(\sqrt{n})$ Partitionierungselemente verwendet werden, ohne daß ein neuer Flaschenhals entstünde. $|B_i|$ würde dann deutlich kleiner. Außerdem ist es möglich, den Partitionierungsprozeß zu iterieren, bis auch ein einfacher Sortieralgorithmus zeitlich nicht mehr ins Gewicht fällt.

- Die Bearbeitung kann auf Kosten der PQ Semantik beschleunigt werden, indem die Maximalzahl der aus den lokalen PQs entnommenen Elemente (u.U. durch eine Konstante) beschränkt wird. Dadurch läßt sich ein stufenloser Übergang zwischen echten parallelen PQs und dem Algorithmus von Karp und Zhang [4] erreichen, bei dem `insert`-Operationen zwar zufällig verteilt werden, aber überhaupt nicht sortiert wird.

- In einigen Branch-and-bound Anwendungen mit starken Heuristiken wird ein großer Teil der eingefügten Knoten niemals wieder betrachtet, da er durch die Heuristiken beschnitten wird. In diesem Fall kann die Zeit für das globale Verschicken der Knoten die Laufzeit der parallelen PQ dominieren. Dann kann es Sinn machen, die Knoten lokal einzufügen oder zumindest nicht alle Knoten global zu verschicken. Dies wird sich i.allg. aber nur in einer verbesserten Laufzeit niederschlagen, wenn man gleichzeitig bereit ist, wie oben die volle PQ Semantik aufzugeben. Durch diese Maßnahmen entstehen Algorithmen, die zwischen echten parallelen PQs und den Heuristiken aus [8, 5, 7, 3] stehen.

5 Zusammenfassung und Ausblick

Lastverteilungsalgorithmen für viele parallele Algorithmen lassen sich gut durch PQs formulieren. Diese durch eine zentral verwaltete PQ zu implementieren, ist aber kaum skalierbar. Der hier dargestellte vollständig verteilte Algorithmus ermöglicht einen erheblich höheren Durchsatz an Anforderungen. Auf einem System mit n PEs können aus einer PQ mit m Elementen $O(n)$ Elemente in Zeit $\tilde{O}\left(s(n) + \log n \left(\log \frac{m}{n} + \log n\right)\right)$ entnommen werden. Einfügeoperationen benötigen nur $O(d(n))$ Operationen. Der Speicheraufwand pro PE liegt in $\tilde{O}\left(\frac{m}{n} + (\log n)^2\right)$. Da der Algorithmus auf Standardoperationen aufbaut, für die oft bereits effiziente Bibliotheksfunktionen (oder gar Hardwareunterstützung) existieren, ist er effizient und relativ einfach auf einem weiten Spektrum von Maschinen realisierbar. Durch kleine Änderungen ist es möglich, den Grundalgorithmus verschiedenen Anforderungen anzupassen. Beispiele sind asynchrone Betriebsweise oder höherer Durchsatz auf Kosten der vollen PQ Semantik.

Vom methodischen Standpunkt ist es interessant zu beobachten, wie verschiedene Grundtechniken der parallelen Programmierung wie zufälliges Verteilen, Lokalität, Pipelining, Prefixsummenbildung, Proben entnehmen und Sortieren ineinandergreifen. Eine interessante Beobachtung bei der Analyse war, daß der größte Teil der wahrscheinlichkeitstheoretischen Argumente in sehr ähnlicher Form auch in der Analyse des eigentlich anders gearteten Random polling Algorithmus [12] auftritt.

Für die Zukunft ist geplant, den parallelen PQ Algorithmus für paralleles Branch-and-bound einzusetzen und mit dem zentralen Ansatz sowie einigen der Heuristiken aus [8, 4, 5, 7, 3] zu vergleichen.

Literatur

[1] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23:493–507, 1952.

[2] N. Deo und S. Prasad. Parallel heap: An optimal parallel priority queue. *The Journal of Supercomputing*, 6(1):87–98, Mrz. 1992.

[3] D. Henrich. Load balancing for data parallel branch-and-bound. In *International Conference Massively Parallel Processing Applications and Development*. Elsevier, 1994.

[4] R. M. Karp und Y. Zhang. Parallel algorithms for backtrack search and branch-and-bound. *Journal of the ACM*, 40(3):765–789, 1993.

[5] N. Kuck, M. Middendorf und H. Schmeck. Generic branch-and-bound on a network of transputers. In R. Grebe u.a., Herausgeber, *Transputer Applications and Systems*, Seiten 521–535. IOS Press, 1993.

[6] T. Leighton. *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufmann, 1992.

[7] R. Lüling und B. Monien. A dynamic load balancing algorithm with provable good performance. Technical report, Universität Paderborn, 1994.

[8] G. P. McKeown, V. J. Rayward-Smith und S. A. Rush. Parallel branch-and-bound. In *Advances in Parallel Algorithms*, Seiten 349–362. Blackwell, 1992.

[9] C. Powley, C. Ferguson und R. E. Korf. Depth-fist heuristic search on a SIMD machine. *Artificial Intelligence*, 60:199–242, 1993.

[10] S. Rajasekaran. Randomized algorithms for packet routing on the mesh. In L. Kronsjö und D. Shumsheruddin, Herausgeber, *Advances in Parallel Algorithms*, Seiten 277–301. Blackwell, 1992.

[11] V. N. Rao und V. Kumar. Concurrent access of priority queues. *IEEE Transactions on Computers*, 37(12):1657–1665, 1988.

[12] P. Sanders. Analysis of random polling dynamic load balancing. Technical Report IB 12/94, Universität Karlsruhe, Fakultät für Informatik, April 1994.

[13] P. Sanders. Massively parallel search for transition-tables of polyautomata. In *VI. International Workshop on Parallel Proccessing by Cellular Automata and Arrays*, 1994.

[14] S. Sen. Finding an approximate median with high probability in constant parallel time. *Information Processing Letters*, 34:77–80, 1990.