UNIKASSEL
VERSITÄT

Fachbereich 16 Elektrotechnik/Informatik
Studiengang Informatik

Distributed Priority Queue with Chapel

Bachelorarbeit

Marco Postigo Perez
Matrikelnummer: 272 025 93

Erstprüfer: Prof. Dr. Claudia Fohry
Zweitprüfer: Prof. Dr. Gerd Stumme

Kassel, den 6. September 2013

Selbständigkeitserklärung

Ich versichere hiermit, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Kassel, den 6. September 2013

Marco Postigo Perez

Inhaltsverzeichnis

1 Einleitung

Durch Parallelität erhöht sich die Komplexität der Programmierung. Zum einen müssen die Daten und der Kontrollfluss aufgeteilt, und zum anderen muss auf die Synchronisierung aufgrund von Datenabhängigkeiten geachtet werden. Moderne parallele Programmiersysteme reduzieren diese Komplexität und damit auch die Fehleranfälligkeit der Programme durch Sprachkonstrukte, welche die Kommunikation unter den Prozessen bzw. Threads[1] auf einer abstrakten Ebene unterstützen. Dadurch kann sich der Entwickler auf das eigentlich zu lösende Problem konzentrieren und wird nicht durch plattformabhängige Implementierungsdetails abgelenkt. Orthogonal dazu kann mit Hilfe von Objektorientierung die Wiederverwendbarkeit und Wartbarkeit von Code deutlich erhöht werden.
Der Entwurf von parallelen Programmiersprachen ist schwierig. In dieser Ausarbeitung wird Chapel betrachtet, eine neue Sprache, die sich noch in der Entwicklung befindet. Chapel setzt die bereits erwähnten Eigenschaften zur Reduzierung der Komplexität sowie den objektorientierten Ansatz um. Im Vordergrund steht die Erhöhung der Produktivität der Softwareentwickler.
In dieser Arbeit wurde eine verteilte Vorrangwarteschlange (engl. Distributed Priority Queue - im folgenden DPQ) mit Chapel implementiert, um die derzeitigen Möglichkeiten und die Stabilität der Sprache zu testen. Des Weiteren kann die Datenstruktur auch von anderen Entwicklern wiederverwendet werden, um den Programmieraufwand auf das eigentliche Problem zu reduzieren. Als Basis wurde die von Bernard Mans in [Man98] beschriebene Datenstruktur verwendet, welche im Rahmen der Arbeit an das von Chapel benutzte Rechenmodell angepasst wurde.
Kapitel 2 der Arbeit gibt einen kurzen Überblick über die Programmiersprache Chapel und beschreibt dann ausführlich den verwendeten Algorithmus für die DPQ. Kapitel 3 setzt sich mit der Implementierung der DPQ mit Chapel auseinander. Zu Beginn wird anhand von Klassendiagrammen die Struktur der Implementierung veranschaulicht. Da-

1 Ausführungsstrang, der Teil eines Prozesses ist. Alle Threads eines Prozesses teilen sich denselben Adressraum.

nach werden Implementierungsdetails anhand eines kurzen Codebeispiels erläutert, wobei insbesondere auf die Abarbeitung der eingehenden Einfüge- und Löschoperationen und die damit verbundene Synchronisierung eingegangen wird. Das vierte Kapitel schildert die verwendeten Testszenarien für die ausgeführten Experimente. Abschließend werden die Ergebnisse diskutiert. Das letzte Kapitel enthält eine kurze Zusammenfassung der Arbeit und ein Fazit zu dem verwendeten parallelen Programmiersystem und der implementierten Datenstruktur. Außerdem gibt es einen Ausblick zu offenen Fragen.

2 Grundlagen

2.1 Chapel

Die Cascade High Productivity Language (Chapel) ist eine imperative Programmiersprache, die seit 2003 von dem Unternehmen Cray Inc. entwickelt wird. Chapel unterliegt der BSD-Lizenz[1] und ist demnach ein Open-Source-Projekt. Aktuell befindet sich die Programmiersprache noch im Entwicklungsstadium und weist daher noch einige „Kinderkrankheiten“ wie fehlende Funktionalitäten bspw. in der Vererbung oder im Speichermanagement auf [vgl. CCD+13, Kap.IV]. Dies wird in Abschnitt 3.2 näher erläutert. Im Gegensatz zu vielen anderen parallelen Programmiersystemen ist Chapel eine grundsätzlich neue Sprache und erweitert keine bereits bestehende. Die derzeit aktuellste Version ist 1.7.0, welche auch für die Implementierung in Kapitel 3 verwendet wurde.
Ein primäres Ziel von Chapel ist es die Produktivität im Bereich der parallelen Programmierung zu steigern [vgl. CCD+13, Kap. I]. Zum einen wurde die Sprache dafür nach dem Partitioned Global Address Space (PGAS) Modell entworfen. Dabei handelt es sich um ein Programmiermodell, in dem der globale Adressbereich des Arbeitsspeichers zwischen den Prozessoren aufgeteilt wird. Jeder Prozessor kann auf jede Speicherzelle zugreifen. Der Zugriff auf lokalen Speicher erfolgt natürlich deutlich schneller als auf den von entfernten Prozessoren.
Zum anderen bietet Chapel einen hohen Grad an Abstraktion und Portabilität. Die Sprache ist so aufgebaut, dass man mit ihr unterschiedliche Berechnungen auf verschiedenen Architekturen durchführen kann, ohne dabei verschiedene Programmiermodelle wie OpenMP, MPI, CUDA etc. verwenden zu müssen. Derselbe Code kann auf einem oder mehreren Prozessoren, aber auch auf der GPU ausgeführt werden. Der Grad der Abstraktion, auf dem der Entwickler programmieren möchte, kann selbst bestimmt und die unterschiedlichen Abstraktionsgrade im Programmverlauf gemeinsam verwendet werden.

1 Berkeley Software Distribution License - http://chapel.cray.com/release/LICENSE

Den höchsten Abstraktionsgrad bietet Chapel mit den zur Verfügung gestellten Domain Maps. Sie sind ein mächtiges Werkzeug, mit dem der Entwickler bestimmen kann, wie Arrays auf verschiedene Prozessoren bzw. Rechenknoten (in Chapel als „Locale“ bezeichnet) verteilt und wie genau sie im Speicher abgelegt werden. Darüber hinaus werden Berechnungen die auf dem Array durchgeführt werden implizit auf dem dazugehörigen Locale ausgeführt.
Möchte man mehr Kontrolle über den Programmfluss, so kann man auf einem niedrigeren Abstraktionsniveau programmieren. Unter anderem lässt sich genau bestimmen, welche unterschiedlichen Aufgaben parallel ausgeführt werden können (Taskparallelität). Einen noch niedrigeren Abstraktionsgrad bieten die Konstrukte der Basissprache wie z.B. Iteratoren, die selbst definiert werden können, oder die Möglichkeit bereits existierenden Programmcode anderer Programmiersprachen einzubinden (zur Zeit nur C-Code). Auf der niedrigsten Abstraktionsebene lässt sich auch explizit bestimmen, auf welchen Rechenknoten Berechnungen ausgeführt bzw. Daten abgelegt werden sollen (letzteres ergibt sich bei der Verwendung einer Domain Map implizit, was den Unterschied zwischen den Abstraktionsebenen verdeutlicht). Die Abstraktionsebenen werden noch einmal in Abbildung 2.1 dargestellt. [vgl. CCD+13, Kap. III]

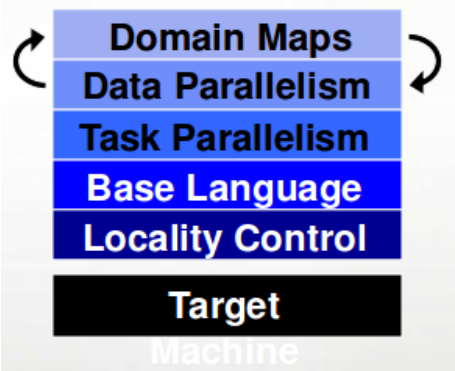


Abbildung 2.1: Chapel Sprachkonzept , entnommen von [Cha12], S. 30

Die parallel ausführbaren Aufgaben werden in Chapel als Tasks bezeichnet und standardmäßig zur Laufzeit den Threads zugeordnet ([vgl. CMSW11, Kap. 3]). Es kann zwischen unterschiedlichen Tasking Layers gewählt werden, welche die Zuordnung der Tasks auf Threads bestimmen. Beispielsweise wird beim Standard Tasking Layer FIFO[2] genau ein

2 Abkürzung für First In - First Out. Dabei handelt es sich im Allgemeinen um ein Verfahren um Daten zu organisieren. Daten die zuerst gespeichert wurden, werden auch wieder als erstes entnommen.

Task einem Thread zugeordnet und dieser bis zur Beendigung durchgeführt. Trifft ein Task auf einen kritischen Abschnitt und muss warten, so führt der zugehörige Thread in der Zwischenzeit keine anderen Tasks aus.
Des Weiteren ist Chapel eine objektorientierte Programmiersprache. Wobei Objektorientierung lediglich angeboten, aber nicht vorausgesetzt wird. Jedoch lässt sich auf diese Weise Code leichter wiederverwenden und komplexere Programme sind einfacher zu warten, da sich einzelne Programmteile leichter austauschen lassen. Chapel bietet zu diesem Zweck zum einen Klassen und zum anderen Records an. Von Klassen abgeleitete Objekte werden durch einen Konstruktor, der in der Klasse definiert ist, instanziiert. Records hingegen können durch einen Konstruktor erzeugt werden, dies ist jedoch nicht notwendig. Sie werden bereits bei der Deklaration mit den in dem Record vordefinierten Standardwerten initialisiert. Der Hauptunterschied zwischen Records und Klassen ist die Art, wie sie gespeichert werden. Die Objektinstanz einer Klasse liegt physikalisch in dem Speicher des Locales, auf dem sie erstellt wurde. Dabei enthält die Variable eines bestimmten Klassentyps eine Referenz auf ein Objekt dieser Klasse. Verwendet man das Objekt nun auf einen beliebigen anderen Locale und weist es einer neuen Variable zu, so wird nicht das Objekt kopiert, sondern nur die Referenz zum entfernten Locale. Im Gegensatz dazu steht bei einer Variablen von einem bestimmten Recordtyp kein Verweis auf ein Objekt sondern direkt die zugehörigen Daten des Records. Außerdem werden sie bei Zuweisungen sowie als Parameter oder Rückgabewert von Methoden kopiert. Records sind Klassen sehr ähnlich. Sie unterstützen wie Klassen Vererbung, Generizität[3], Methodendeklaration und -definition.
Die Synchronisierung unter den Threads bzw. Prozessen wird über sogenannte `sync` und `single` Variablen bewerkstelligt. Die Variablen können zwei verschiedene Zustände annehmen: leer oder voll. Ist die Variable leer und es wird versucht diese zu lesen, so blockiert der Lesezugriff solange bis die Variable voll ist. Ist die Variable voll und es soll ein Schreibzugriff erfolgen, so wird solange blockiert bis diese leer ist. Die `single` Variable unterscheidet sich im Vergleich zu der `sync` dadurch, dass ihr nur einmal ein Wert zugewiesen werden darf. Danach bleibt die Variable trotz weiterer Lesezugriffe im Zustand „voll“ und blockiert nicht mehr. Um mehr Kontrolle über die Synchronisierung zu erhalten, werden zusätzlich Methoden angeboten, welche auf den Variablen aufgerufen werden können. (vgl. [spe13, Kap. 15, 16, 24, 25 und 31], sowie [Cha13])

3 Eigenschaft einer Klasse / eines Records durch einen Typ parametrisiert zu werden. Damit können Klassen bzw. Records allgemein entworfen und für verschiedene Datentypen verwendet werden.

2.2 Vorrangwarteschlange

Eine Vorrangwarteschlange (oder auch Prioritätswarteschlange) ist ein abstrakter Datentyp. Den eingefügten Elementen sind Prioritäten zugeordnet, wobei gilt: je kleiner der Wert, desto wichtiger, eiliger o.ä. ist der Eintrag. Die Priorität bestimmt also die Reihenfolge der Abarbeitung. Dabei sind Duplikate erlaubt. Zum Einfügen und Herauslesen stehen die folgenden Methoden zur Verfügung (vgl. [SS06, Kap. 16.4] und [Man98, Kap. 1]):

a) `insert(i, h)`: neues Element i mit vordefinierter Priorität in h einfügen. Die Priorität ist implizit in i enthalten.

b) `deleteMin(h)`: Kleinstes Elemenent aus h löschen und zurückgeben.

2.3 Distributed Priority Queue - Datenstruktur

2.3.1 Aufbau

Die von Mans in [Man98] vorgeschlagene und als Basis dieser Arbeit verwendete DPQ baut auf dem verteilten Heap von David Peleg [Pel89] auf. Ein Heap ist ein binärer Baum, der wie die Vorrangwarteschlange Elemente mit ihrer zugehörigen Priorität enthält. Zusätzlich zu den Eigenschaften binärer Bäume muss für einen Heap die sog. Heap-Eigenschaft gelten ([vgl. SS06, Kap. 14.6]):

- Der Baum ist vollständig, d.h. die Blätter der Knoten werden von links nach rechts gefüllt. (2.3.1.1)
- Der Schlüssel eines jeden Knotens (in unserem Fall die Priorität des Elements) muss kleiner oder gleich der Schlüssel seiner Kinder sein. (2.3.1.2)

Bei dem von Peleg beschriebenen verteilten Heap handelt es sich um einen d-ary Heap, der eine Generalisierung des Standard Heaps darstellt. In einem d-ary Heap können alle Knoten bis zu d Kinder haben, also entspricht der 2-ary Heap genau unserem Standard Heap. Jeder Prozessor bzw. Rechenknoten stellt einen Knoten des Heaps dar und alle Knoten können mehrere Elemente enthalten, wobei jedes Element eines Knotens kleiner

oder gleich den Elementen der Kindknoten sein muss (vgl. 2.3.1.2). Der Vorteil einer Strukturierung zu einem verteilten Heap ist, dass sehr schnell auf das Element mit der höchsten Priorität zugegriffen werden kann.
In der von Mans beschriebenen DPQ werden die zur Verfügung stehenden Prozessoren von 0 bis $n-1$ durchnummeriert und in einer vordefinierten Baumstruktur angeordnet (siehe Abb. 2.2 und Abb. 2.3). Bei dieser Baumstruktur kann es sich entweder um den zuvor beschriebenen d-ary Heap oder um einen Binomialbaum handeln.
Bei einem Binomialbaum des Grades k gilt für die Gesamtzahl der Knoten a_i auf der i-ten Ebene, dass $a_i \leq \binom{k}{i}$. Wobei: $a_i = \binom{k}{i} \Leftrightarrow n = 2^x$, mit $x \in \mathbb{N}_0$. Die Nummerierung eines Binomialbaums beginnt beim Wurzelknoten. Die Wurzel wird mit 0 und ihre Kinder mit 2^j ($\forall j \in [0,\ \lceil \mathrm{ld}(n) \rceil]$) gekennzeichnet. Die Kinder eines Prozessors p können dabei mit $p + 2^i$ ($\forall i \in [0,\ l-1]$) berechnet werden, wobei l der Exponent von 2 bei der Primfaktorzerlegung von p ist. Darüber hinaus muss $p + 2^i < n$ sein. Ein Beispiel, bei der die Zerlegung von $p = 12$ betrachtet wird, soll zeigen wie l bestimmt wird: $2^2 * 3^1$, woraus $l = 2$ folgt (siehe. Abb. 2.2).

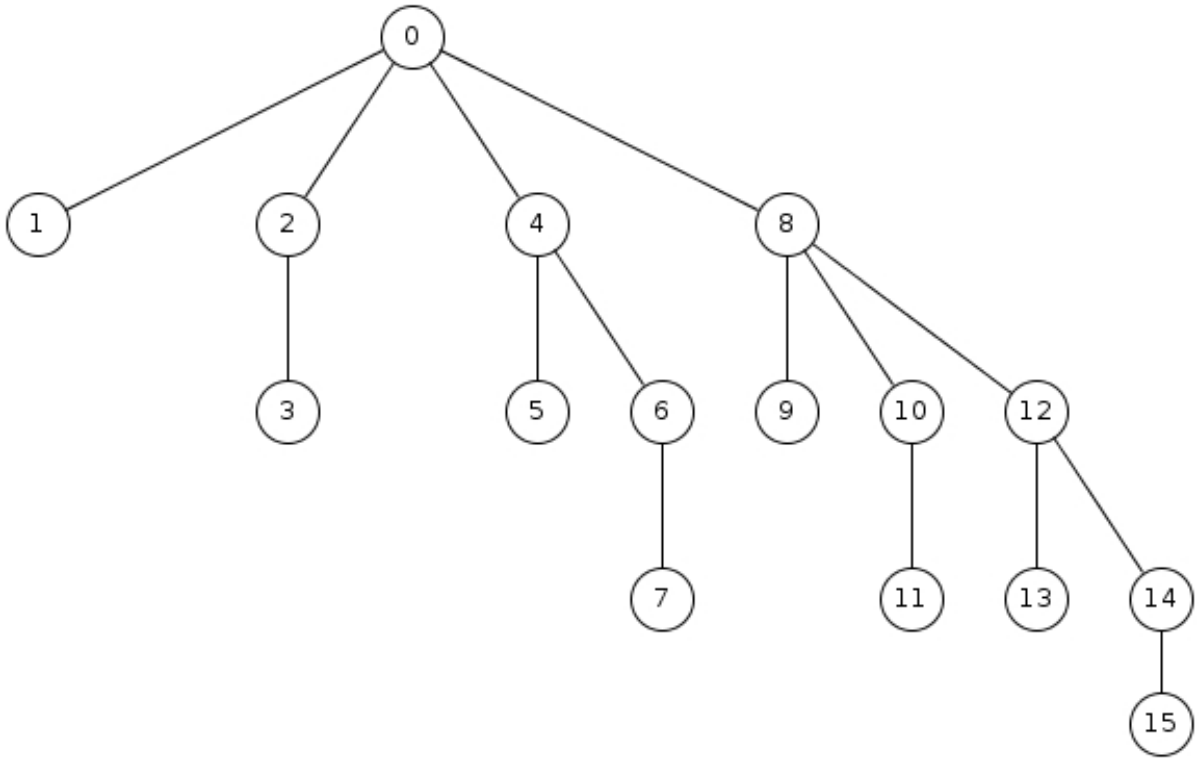


Abbildung 2.2: Binomial-Baum

Bei dem d-ary Heap beginnt die Nummerierung ebenfalls beim Wurzelknoten, der die 0 erhält.
Für die Berechnung eines Kindknotens k_i von p gilt (mit $0 < i \leq d$):

$$k_i = f(d * p + i, n), \text{ mit}$$

$$f(x, y) = \begin{cases} x & \text{falls } x \leq y \\ \perp & \text{sonst} \end{cases}.$$

Außerdem lässt sich der Elternknoten eines Prozessors p mit $\lfloor \frac{(p-1)}{d} \rfloor$ berechnen (vgl. Abb. 2.3).

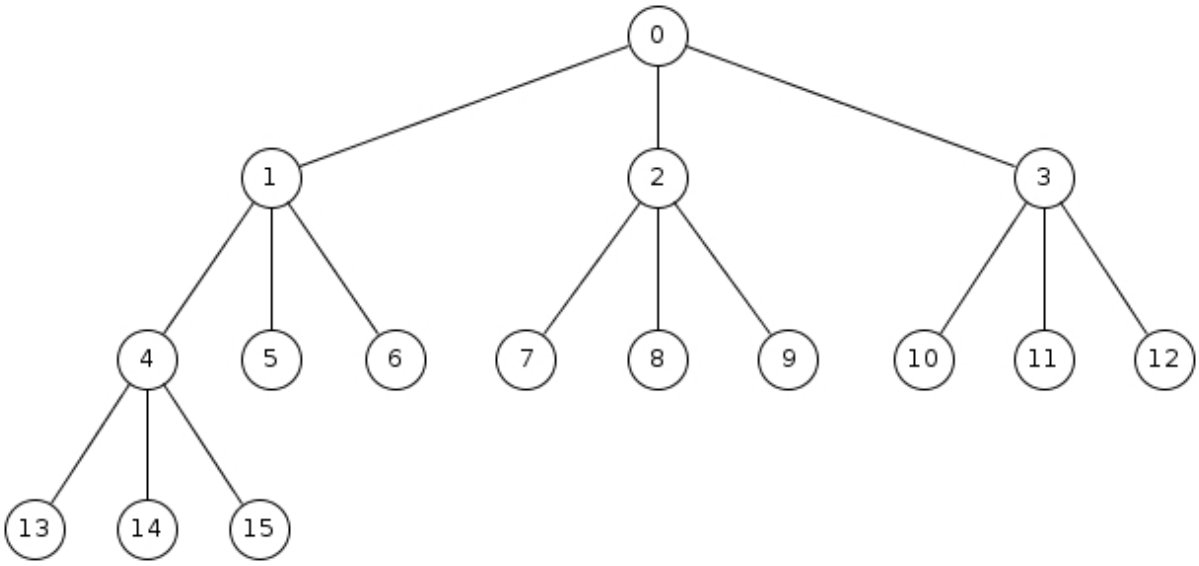


Abbildung 2.3: 3-ary Heap mit 16 Prozessoren

Im Vergleich zu [Man98, Kap. 3.1] wurde in dieser Ausarbeitung bei beiden Varianten die Nummerierung vereinheitlicht. Anstatt beim Binomialbaum mit 0 und beim d-ary Heap mit 1 zu beginnen, starten nun beide im Wurzelknoten mit 0. Zusätzlich wurden die angegebenen Formeln anders dargestellt und aufgrund der Anpassung mit der Nummerierung leicht modifiziert.

2.3.2 Konzept

Daten werden nach dem Round-Robin Verfahren[4] hinzugefügt bzw. entfernt. Alle Zugriffe auf die DPQ werden über den Prozessor in der Wurzel des Heaps initiiert und die Anfrage nach einem Top-Down Schema weitergeleitet. Dafür berechnet der Wurzelknoten die Zielposition bzw. den Zielknoten, auf dem ein Element hinzugefügt bzw. gelöscht werden soll (näheres in Abschnitt 2.3.4) und leitet die Anfrage den Pfad entlang zum Zielknoten weiter. Die hinzugefügten Elemente, werden mit diesem Verfahren gleichmäßig auf die zur Verfügung stehenden Rechenknoten verteilt. Lokal speichert jeder Rechenknoten die Daten in einem Splay-Baum (siehe Abschnitt 2.3.3).

Heap-Bedingung überprüfen und wiederherstellen

Die Überprüfung und ggf. notwendige Wiederherstellung der Heap-Bedingung (2.3.1.2) wird im Folgenden auch als `heapify` bzw. `heapify`-Operation bezeichnet. Der Prozessor, der diese Operation ausführt, muss sein größtes Element mit dem kleinsten der Kindknoten ist vergleichen. Ist das lokale Maximum größer als das Minimum der Kindknoten, so werden die beiden Elemente ausgetauscht.
Die `heapify`-Operation muss nach jeder Löschoperation, die auf einem Knoten stattfindet, ausgeführt werden. Im Gegensatz dazu ist dies beim Einfügen nur in Spezialfällen nötig:

(a) Der Knoten, auf dem das Element einfügt wurde, ist der Zielknoten.

(b) Der Knoten hat das Element durch ein vorangegangenes `heapify` seines Elternknotens erhalten.

Im schlechtesten Fall kann bei (b) das Element durch den ganzen Teilbaum „durchgereicht“ werden.
Das folgende Beispiel soll die `heapify`-Operation noch einmal demonstrieren (siehe Abb. 2.4). Der Ausgangspunkt ist eine zuvor ausgeführte Einfügeoperation. Dabei wurde ein Element mit Priorität 7 auf Knoten (1), welcher der Zielknoten war, hinzugefügt. Infolgedessen überprüft Knoten (1) die Heap-Bedingung (siehe Punkt (a)) und stellt fest, dass beide Kindknoten kleinere Elemente besitzen als sein Maximum. Um die Heap-Bedinung

4 Ein Verfahren bei dem mehrere gleichartige Ressourcen gleichmäßig beansprucht werden

wiederherzustellen muss Knoten (2), der das kleinste Element ($\rightarrow$ 4) hat, sein Minimum mit dem Maximum von Knoten (1) tauschen. Im nächsten Schritt muss Knoten (2) die `heapify`-Operation ausführen (siehe Punkt (b)). Da Knoten (4) ein kleineres Element ($\rightarrow$ 6) als das eben Erhaltene hat, wird das Maximum von Knoten (2) mit dem Minimum von Knoten (4) getauscht.

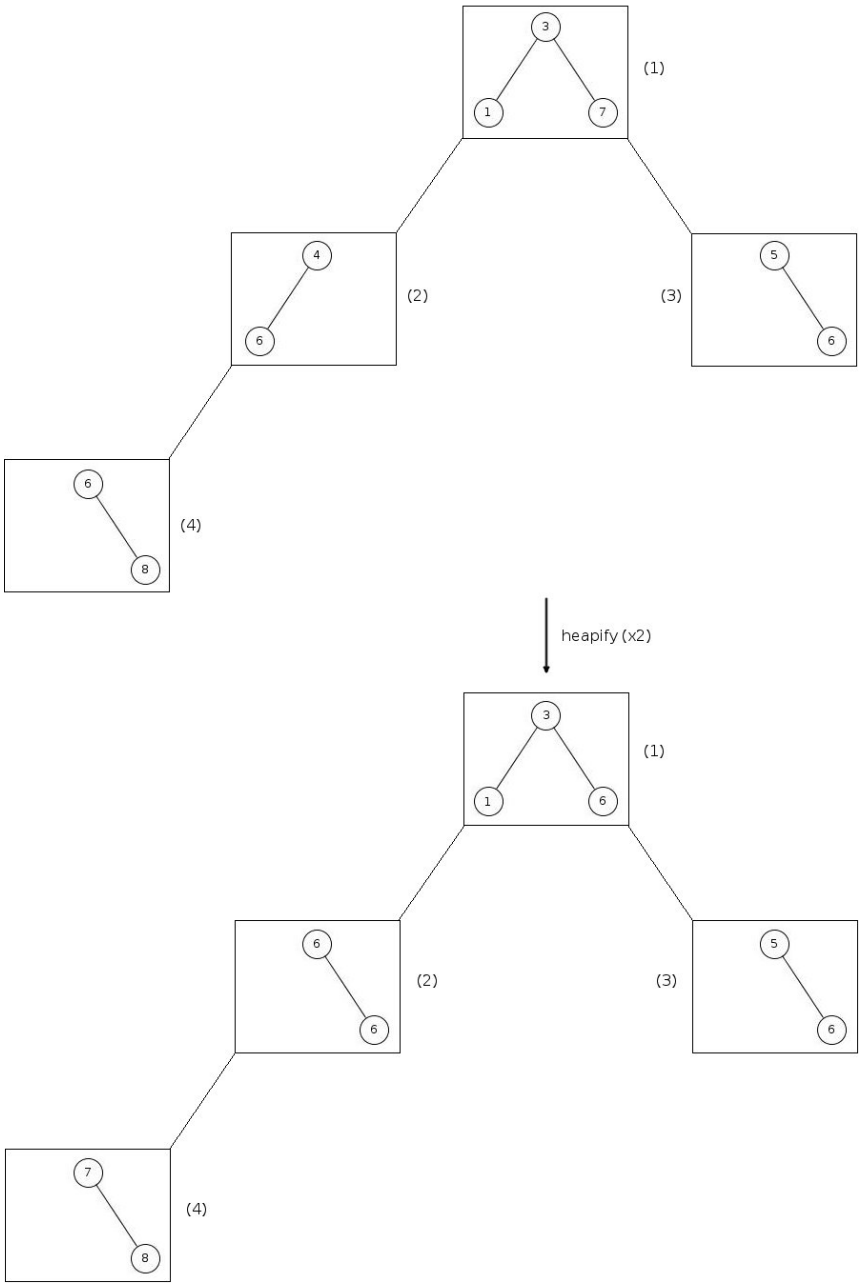


Abbildung 2.4: Beispiel: Heap-Bedingung überprüfen und wiederherstellen

Nun müsste auch Knoten (4) `heapify` durchführen, jedoch sind keine Kindknoten vorhanden, so dass dies überflüssig ist. Im Beispiel Die beiden durchgeführten `heapify`-Operationen wurden aufgrund der Übersichtlichkeit zusammengefasst („heapify (x2)).

Einfügen neuer Elemente

Das Einfügen von neuen Elementen ist nicht blockierend, d.h. ein Prozessor, der ein Element in die DPQ einfügen möchte, kann seine Arbeit direkt fortsetzen und muss nicht auf das Beenden der Operation warten. Nachdem berechnet wurde, auf welchem Knoten das Element eingefügt werden soll (siehe Abschnitt 2.3.4) wird die Anfrage den Pfad entlang zu dem Zielknoten weitergeleitet. Dabei überprüft jeder Knoten, der auf diesem Pfad liegt (inkl. dem Wurzelknoten), ob sein größtes Element größer ist als das Element, dass eingefügt werden soll. Sollte dies der Fall sein, so speichert der jeweilige Knoten das Element lokal ab und leitet das lokale Maximum den Pfad entlang weiter. Sobald der Zielknoten erreicht wurde, wird das Element dem entsprechenden Splay-Baum hinzugefügt und anschließend die `heapify`-Operation durchgeführt. Sollte der Zielknoten keine Kindknoten besitzen, so entfällt `heapify`.
Der Einfügevorgang soll an einem Beispiel (siehe Abb. 2.5) veranschaulicht werden. Dabei wird ein Element mit der Priorität 2 in einen binären (2-ary) Heap mit drei Knoten eingefügt. Der Zielknoten, dem ein Element hinzugefügt werden soll ist die (3), da dort zu Beginn nur zwei Elemente vorhanden sind und in den anderen bereits drei. Der Wurzelknoten überprüft nun, ob das einzufügende Element kleiner ist als sein größtes Element ($\rightarrow$ 5). Da dies der Fall ist, speichert der Wurzelknoten das neue Element lokal ab und leitet das Element mit der Priorität 5 den Pfad entlang weiter. Der nächste Knoten auf dem Pfad ist bereits der Zielknoten, also wird das Element nun direkt auf dem Knoten (3) eingefügt. Abschließend müsste nun noch für den Zielknoten die `heapify`-Operation ausgeführt werden, da Knoten (3) jedoch keine Kindknoten hat ist dies nicht nötig.

Entfernen von Elementen

Das Entfernen von Elementen läuft sehr ähnlich ab, jedoch wird dieser Aufruf im Gegensatz zum Einfügen solange blockiert, bis das Element zum aufrufenden Prozessor gesendet

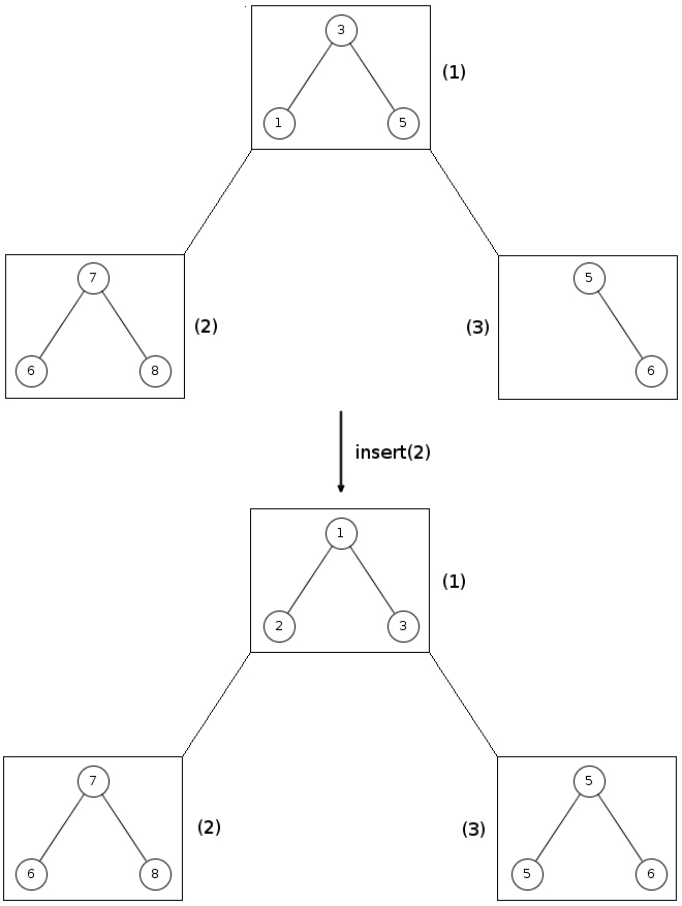


Abbildung 2.5: Beispiel: Einfügen eines neuen Elements

wurde. Sollte die DPQ leer sein, so muss der aufrufende Prozessor solange warten, bis ein neues Element hinzugefügt wurde.
Nach dem Erhalt der Anfrage im Wurzelknoten sendet dieser sein lokales Minimum direkt zum aufrufenden Prozessor (lt. Heap-Bedingung muss das kleinste Element im Wurzelknoten sein). Nun wird mit der berechneten Zielposition (siehe Abschnitt 2.3.4) die Anfrage wieder den Pfad entlang zum Zielknoten weitergeleitet. Alle Knoten auf dem Pfad senden ihr lokales Minimum zu dem Elternknoten und leiten (sofern sie nicht der Zielknoten sind) die Anfrage weiter. Die Elternknoten speichern das von ihrem Kind erhaltene Element lokal ab. Sie müssen solange mit der Ausführung anderer Operationen warten, bis der Kindknoten, der auf dem Pfad zum Zielknoten liegt, sein Minimum gesendet hat und sie selbst anschließend die Heap-Bedingung (2.3.1.2) überprüft haben.
Auch das Entfernen eines Elements soll nun noch einmal an einem Beispiel verdeutlicht werden (siehe Abb. 2.6). Dabei wird wieder ein binärer Heap mit drei Knoten verwendet.

Knoten (1) erhält die Anfrage und sendet sein lokales Minimum ($\rightarrow$ 1) direkt an den aufrufenden Prozessor, so dass dieser mit seinen Berechnungen fortfahren kann. Anschließend wird die Anfrage wieder den Pfad entlang zum Zielknoten (2) weitergeleitet. Dieser sendet ebenfalls sein Minimum ($\rightarrow$ 6) und ist daraufhin fertig. Knoten (1) muss nun nach Erhalt des Minimums von Knoten (2) überprüfen, ob eines seiner Kinder ein kleineres Element besitzt als er (Heap-Bedingung) und dieses ggf. austauschen. In diesem Fall besitzt Knoten (3) ein Element mit Priorität 5, welches nun mit der zuvor erhaltenen 6 ausgetauscht wird. Nun müsste auch Knoten (3) überprüfen, ob die Heap-Bedingung noch gilt, jedoch besitzt er in diesem Beispiel ebenfalls keine Kindknoten, so dass sich Operation erspart werden kann..

2.3.3 Splay-Baum

Auf jedem Prozessor wird ein Splay-Baum als lokale Vorrangwarteschlange genutzt. Ein Splay-Baum ist ein spezieller binärer Suchbaum, der bei jeglichen Anfragen und Modifikationen die Anordnung der Elemente neu ausrichtet. Dabei werden die angefragten Elemente näher zur Wurzel des Baumes gebracht, so dass der nächste Zugriff schneller erfolgen kann. Mans hat in [Man98] dem Splay-Baum zusätzliche Funktionen zum Suchen und Löschen des kleinsten und größten Elements hinzugefügt (`findMin`, `findMax`, `deleteMin`, `deleteMax`), um den Anforderungen seiner Datenstruktur gerecht zu werden. Die amortisierte Laufzeit[5] von Einfüge-, Such- und Löschoperationen liegt bei $\mathcal{O}(log(n))$ bei einem Splay-Baum mit n Elementen.
Im Vergleich zu anderen Bäumen haben Splay-Bäume eine spezielle Splay-Operation. Wird diese Operation auf ein Element x angewendet, so wird dieses in die Wurzel des Baums hinaufrotiert. Dabei wird x mit seinem Eltern- bzw. Großelternknoten verglichen, wodurch sich 6 Fallunterscheidungen ergeben von denen die Hälfte symmetrisch ist.

(1) Ist x das linke Kind (bzw. rechte Kind), so wird eine zig-Rotation (bzw. zag-Rotation) durchgeführt (siehe Abb. 2.7).

(2) Ist x das linke Kind (bzw. rechte Kind) seines Elternknotens, welcher wiederum das linke Kind (bzw. rechte Kind) des Großelternknotens von x ist, so wird eine zig-zig-Rotation (bzw. zag-zag-Rotation) durchgeführt (siehe Abb. 2.8).

5 Beschreibt die durchschnittlichen Kosten pro Operation in einer Sequenz von Worst Case Szenarien

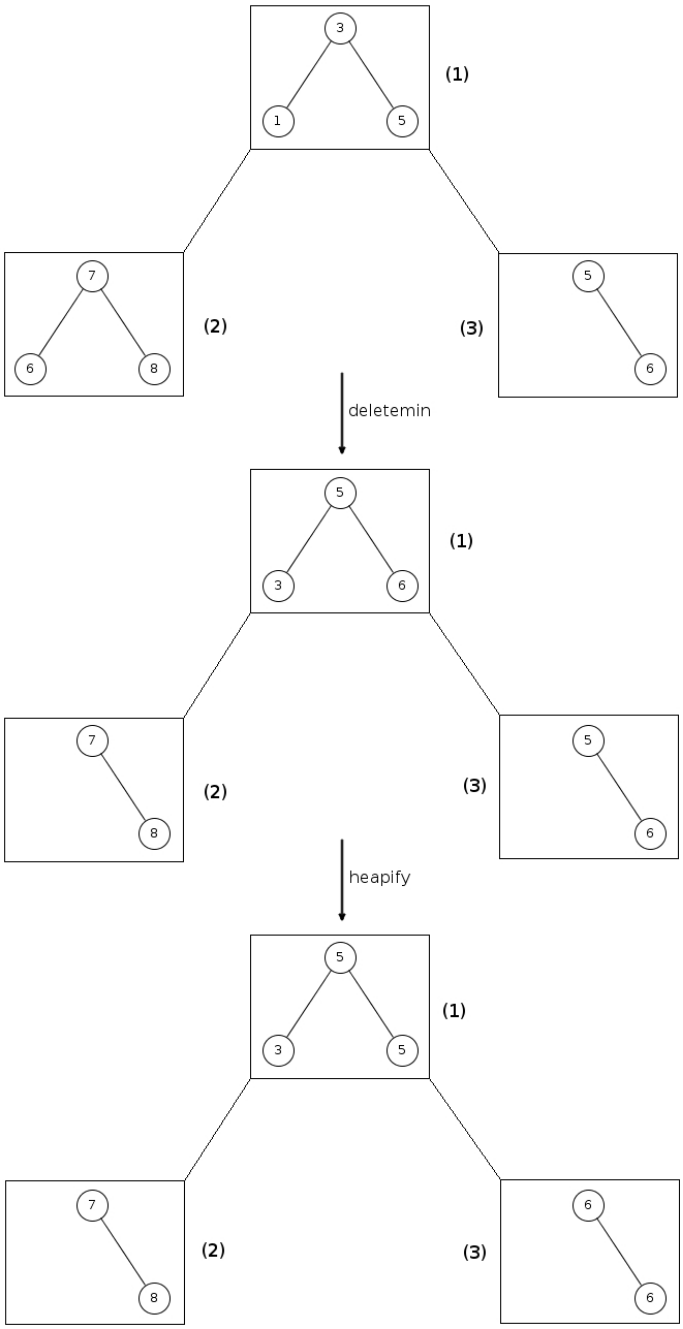


Abbildung 2.6: Beispiel: Entfernen eines Elements

(3) Ist x das rechte Kind (bzw. linke Kind) seines Elternknotens, welcher das linke Kind (bzw. rechte Kind) des Großelternknotens von x ist, so wird eine zig-zag-Rotation (bzw. zag-zig-Rotation) durchgeführt. (siehe Abb. 2.9)

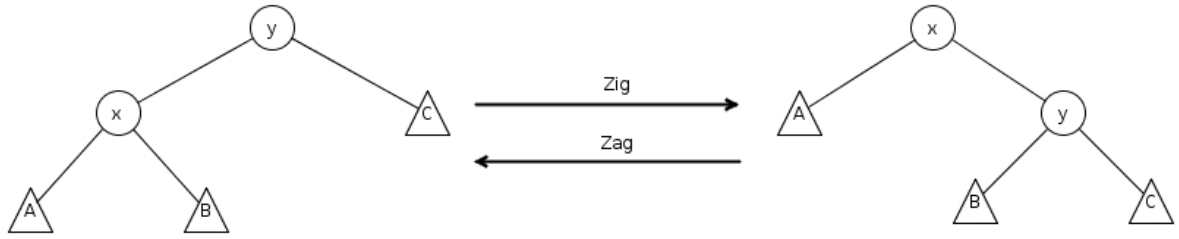


Abbildung 2.7: Splay Tree Operation zig / zag

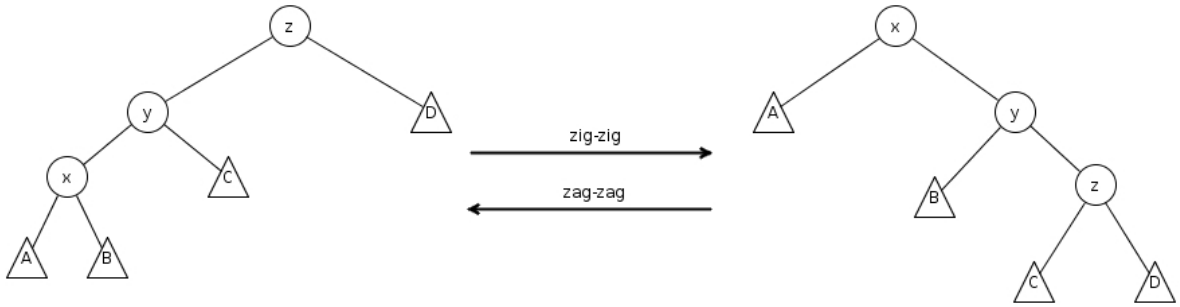


Abbildung 2.8: Splay Tree Operation zig-zig / zag-zag

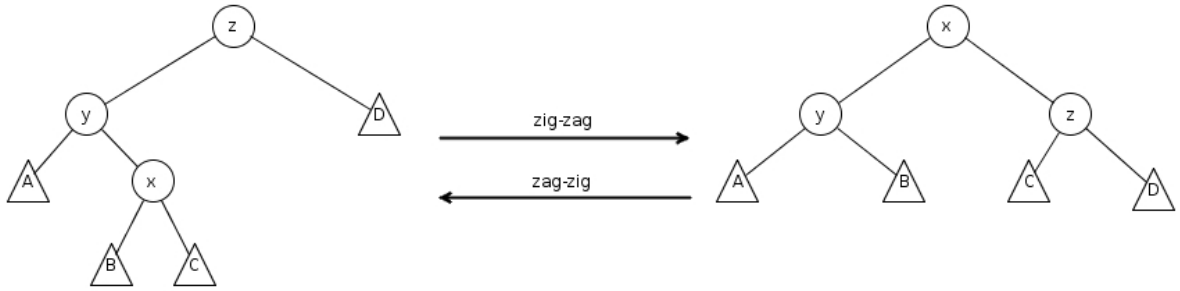


Abbildung 2.9: Splay Tree Operation zig-zag / zag-zig

Diese Operationen werden solange ausgeführt, bis x in der Wurzel des Splay Baumes steht. Sollte x nicht im Splay-Baum vorhanden sein, so befindet sich der Knoten in der Wurzel, an dem x eingefügt werden könnte. [ST85]

2.3.4 Lastbalancierung

Um eine möglichst hohe Lastbalancierung zu gewährleisten, wird zum Einfügen bzw. Löschen der Elemente das vorher skizzierte Round-Robin Verfahren gewählt. Nach [Pel89] bestimmt dieses die Position zum Einfügen oder Löschen mit Hilfe der Anzahl der Elemente in der Datenstruktur (i) und der Anzahl der zur Verfügung stehenden Prozessoren (n). Sei p_i (mit $i \in \mathbb{N}_0$) die nächste Position, an der ein Element eingefügt bzw. entfernt werden soll, wenn bereits i Elemente in der Datenstruktur vorhanden sind. Dann ergibt sich:

$$p_i = i \bmod n$$

Leider gibt es bei diesem Ansatz das Problem, dass die aufeinanderfolgenden berechneten Positionen mit sehr hoher Wahrscheinlichkeit im selben Teilbaum liegen, was zu einer hohen Auslastung entlang der Pfade führt. Um dies zu verhindern wurde in [Man98] eine Anpassung vorgenommen, welche mittels einer Primzahl p_z zwischen n und $\lceil \frac{n}{4} \rceil$ die Position bestimmt. Die Position p_i lässt sich wie folgt berechnen:

$$p_0 = 0$$

$$p_{i+1} = \begin{cases} p_i + p_z & \text{falls ein Element hinzugefügt werden soll} \\ p_i & \text{falls ein Element entfernt werden soll} \end{cases}$$

Dabei muss der Sonderfall beachtet werden, dass $n < m$ gilt. Sollte dies der Fall sein, so muss $p_z = 1$ gewählt werden, um die Elemente gleichmäßig auf die Knoten zu verteilen.

$$p_z = \begin{cases} 1 & \text{falls } n < i \\ minprim(\lceil \frac{n}{4} \rceil, n) & \text{sonst} \end{cases}$$

mit

$$minprim(x, y) = \begin{cases} z & \text{mit } z = \text{kleinste Primzahl im Intervall } [x, y] \text{ und } z \nmid y \\ 1 & \text{sonst} \end{cases}.$$

Auch diese Formeln wurden im Vergleich zu [Man98, Kap. 3.2] anders dargestellt und an den Beginn der Nummerierung der Rechenknoten bei 0 angepasst.
Wichtig ist, dass die gewählte Primzahl kein Teiler von n sein darf, da ansonsten nur $\frac{n}{p_z}$

Knoten verwendet werden. Würde man eine solche Primzahl wählen, so wäre beispielsweise für das wiederholte Einfügen von Elementen mit $n = 15$ und $p_z = 5$ folgende Reihenfolge festgelegt:

$$0, 5, 10, 0, 5, 10, 0, 5, \ldots$$

2.3.5 Nebenläufigkeit und Konsistenz

Wie bereits erwähnt ist das Grundkonzept der DPQ ein Top-Down Schema, welches stets im Wurzelknoten beginnt, d.h., alle Anfragen zum Hinzufügen bzw. Löschen werden zu dem Prozessor in der Wurzel gesendet und von dort aus auf dem Pfad zu den entsprechenden Zielknoten weitergeleitet. Wichtig dabei ist, dass immer nur eine Anfrage pro Prozessor bzw. Knoten aktiv sein darf, so dass Modifikationen an den Daten atomar sind. Es sind immer nur maximal zwei Knoten mit derselben Anfrage beschäftigt, wodurch der Wurzelknoten neue Anfragen abarbeiten kann, sobald er die vorherige Anfrage an den nächsten Knoten weitergeleitet hat. So können in den verschiedenen Teilbäumen mehrere Anfragen gleichzeitig bearbeitet werden, ohne dass die Konsistenz der Daten beeinträchtigt wird. Die Anfragen werden auf den Knoten in FIFO Reihenfolge bearbeitet. Das Top-Down Schema verbunden mit dem FIFO Konzept garantiert Atomarität und Konsistenz ohne komplizierte Lock-Mechanismen oder kollektiver Kommunikation. ([vgl. Man98, Kap. 3.4])

3 Implementierung

Die in Abschnitt 2.3 beschriebene DPQ nach B. Mans wurde mit der derzeit aktuellsten Version von Chapel umgesetzt.

3.1 Struktur

Die Locales wurden wie in Abschnitt 2.3 beschrieben in einer Baumstruktur angeordnet. In Chapel beginnt die Nummerierung der Locales bei 0. Auf die jeweilige Nummer bzw. ID der Locales kann jederzeit im Programmverlauf zugegriffen werden, so dass das in Abschnitt 2.3.1 beschriebene Verfahren zum Nummerieren der Knoten direkt übernommen werden konnte. Umgesetzt wurde aus Zeitgründen lediglich die Strukturierung zu einem d-ary Heap.
Auf jedem Locale wird zu einem Zeitpunkt maximal eine Anfrage bearbeitet (siehe Abschnitt 2.3.5). Dadurch erscheint die Verwendung mehrerer Threads auf einem Locale wenig sinnvoll, zumal sich die Splay-Baum Operationen auch schwer parallelisieren lassen.
Der Wurzelknoten erstellt einen neuen Task nach dem Berechnen des Zielknotens und der Modifikation der Gesamtgröße für das Einfügen und Löschen von Elementen. Dieser Task führt dann die Einfüge- bzw. Löschoperation nach dem Top-Down Schema aus (siehe Abschnitt Abschnitt 2.3.2). Dadurch kann der Wurzelknoten schon mit der Vorbereitung des nächsten Tasks beginnen. Die Erstellung dieses Tasks muss allerdings solange warten, bis der vorherige beendet wurde. Würde man nicht auf den vorherigen Task warten, so könnte der Tasking Layer die wartenden Tasks in einer beliebigen Reihenfolge den zur Verfügung stehenden Threads zuordnen, wodurch die FIFO Reihenfolge nicht eingehalten werden würde. Analog gilt dasselbe für das Entfernen des Minimums.
Das Klassendiagramm in Abb. 3.1 gibt einen Überblick über die Struktur der DPQ. Da es in Chapel noch keine Interfaces oder abstrakten Klassen gibt, musste leider auf ein

Design mit einfach austauschbaren Komponenten verzichtet werden. Bspw. würde die Klasse `DistributedPriorityQueue` in diesem Fall ein Interface implementieren oder eine abstrakte Klasse erweitern. Ferner müsste die Klasse `SplayTree` nicht direkt verwendet werden, sondern ein entsprechendes Interface und darüber hinaus wäre es möglich gewesen, dass für die Methoden zum Einfügen neuer Elemente (von `SplayTree` und `DistributedPriorityQueue`) nur ein Argument notwendig gewesen wäre. Dafür hätte das einzufügende Objekt ein bestimmtes Interface implementieren müssen bzw. von einer bestimmten abstrakten Klasse erben. Dieses Interface bzw. diese abstrakte Klasse würde nun vorgeben eine Methode zu implementieren, mit der es möglich ist die Objekte untereinander zu vergleichen und damit einzusortieren. Außerdem kann derzeit noch nicht von generischen Klassen geerbt werden, ansonsten hätte anstatt eines Interfaces oder einer abstrakten Klasse auch vorerst eine normale Klasse verwendet werden können.
Eine weitere Möglichkeit wären die von Chapel zur Verfügung gestellten Tupel gewesen. Bei Tupeln handelt es sich um endliche Listen, die es erlauben unterschiedliche Datentypen zusammenzufassen. Dieses Sprachkonstrukt vereinfacht das Zurückgeben mehrerer Werte in einer Methode und damit auch die direkte Umsetzung mathematischer Konzepte (wie bspw. die Bildung des kartesischen Produkts[1]). Außerdem lassen sich so die Parameter von Methoden gruppieren und übersichtlicher gestalten ohne auf Objektorientierung angewiesen zu sein. Mit Hilfe von Tupeln hätte vermieden werden können, dass die Klasse `DistributedPriorityQueue` die Klasse `SplayItem` verwendet. Jedoch gab es dabei Schwierigkeiten, wenn der lokale Splay-Baum keine Elemente enthielt und man das Tupel als leer (bspw. als) kennzeichnen wollte. Dieser Fall kann zum Beispiel bei der Überprüfung der Heap-Bedingung auftreten (siehe Abschnitt 2.3.2). Die Möglichkeit Klassen bzw. Records, deren Typ man erst zur Laufzeit kennt (aufgrund der Generizität), in Chapel mit einem Default-Konstruktor zu erzeugen oder einem Tupel einen Nullzeiger[2] zuzuweisen, hätte dieses Problem behoben.
Alle in Abb. 3.1 aufgeführten Klassen sind generisch (Typparameter T, U), um beliebige Daten aufnehmen zu können. Die Klasse `DistributedPriorityQueue` implementiert die in Abschnitt 2.3 beschriebenen Methoden `insert` und `deleteMin` und verfügt über 1 bis n (Anzahl der Prozessoren bzw. Rechenknoten) Splay-Bäume, die zur Datenhaltung auf dem jeweiligen Locale genutzt werden. Die Splay-Bäume sind in einem verteiltem Array ge-

1 Das kartesische Produkt bzw. Kreuzprodukt konstruiert aus zwei oder mehr gegebenen Mengen eine neue Menge. Wobei z.B. für zwei Mengen gilt: $A \times B := \{(a, b) \mid a \in A, b \in B\}$

2 Ein Zeiger ist wie eine normale Variable, die anstelle von Datenobjekten Adressen eines bestimmten Speicherbereichs enthalten. Der Nullzeiger gibt an, dass auf nichts verwiesen wird.

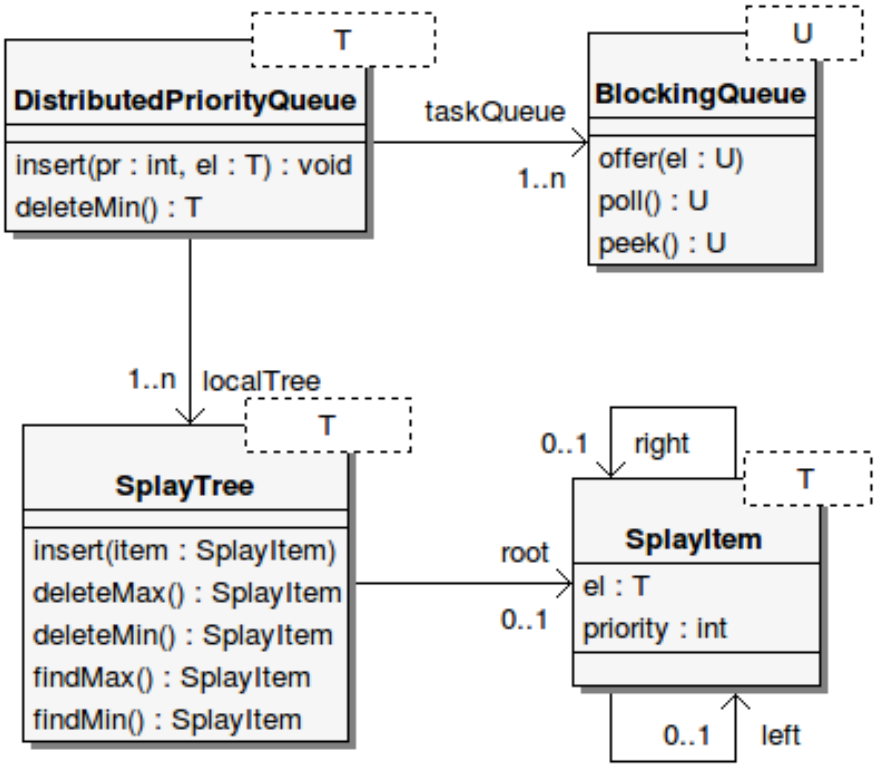


Abbildung 3.1: Klassendiagram DPQ

speichert. Dabei kann jeder Locale mit seiner ID als Index, auf seinen lokalen Splay-Baum zugreifen.
Der Splay-Baum wird von der Klasse `SplayTree` repräsentiert und stellt Methoden zum Finden und Löschen des Elements mit der höchsten bzw. niedrigsten Priorität zur Verfügung (`findMin` / `findMax` und `deleteMin` / `deleteMax`). Jede dieser Methoden gibt ein `SplayItem` zurück, welches die Priorität und das zugehörige Element enthält. Zudem besitzt der `SplayTree` ebenfalls eine `insert` Methode, um die in die DPQ eingefügten Daten entgegenzunehmen. Darüber hinaus hat der `SplayTree` (entweder keine oder) eine Referenz zu einem `SplayItem` Objekt, das die Wurzel (`root`) des Splay-Baums darstellt. Die Klasse `SplayItem`, beinhaltet den eingefügten Datensatz und dessen Priorität. Außerdem kann es ein linkes (`left`) und ein rechtes (`right`) Kind haben, das ebenfalls vom Typ `SplayItem` ist.
Des Weiteren gibt es noch die Klasse `BlockingQueue`, die auf jedem Locale die Aufgaben enthält, die noch zu erledigen sind (Element einfügen / löschen).

3.2 Ausgewählte Aspekte der Chapel Umsetzung

Alle eingehenden Anfragen zum Einfügen und Löschen von Elementen werden in einem Objekt der Klasse `BlockingQueue` auf dem jeweiligen Locale gespeichert um das in Abschnitt 2.3.5 erwähnte FIFO Prinzip zu gewährleisten. Für die Bearbeitung der Anfragen wird auf jedem Locale ein Task erstellt (im Folgenden Taskexecutor genannt), der solange läuft, bis die DPQ explizit beendet wird. Für diesen Zweck wurde eine `shutdown`-Methode hinzugefügt, da in Chapel zur Zeit leider noch kein Destruktor zur Verfügung steht, welcher beim Löschen eines Objekts ausgeführt wird. Die Anfragen werden als `Tasks` (nicht zu verwechseln mit den Chapel Tasks in Abschnitt 2.1) in die `BlockingQueue` eingefügt, welche sich dann mit Hilfe der Methode `execute` ausführen lassen (siehe Abb. 3.2). Für das Einfügen neuer Elemente wird ein `InsertTask` und für das Löschen ein `DeleteTask` erstellt. Die Klassen `InsertTask` und `DeleteTask` sind beide generisch. Sie enthalten denselben Objekttyp, den auch die DPQ beinhaltet. Die Oberklasse `Task` repräsentiert lediglich eine gemeinsame Schnittstelle (Ersatz für Interface) und wird nie instanziiert.

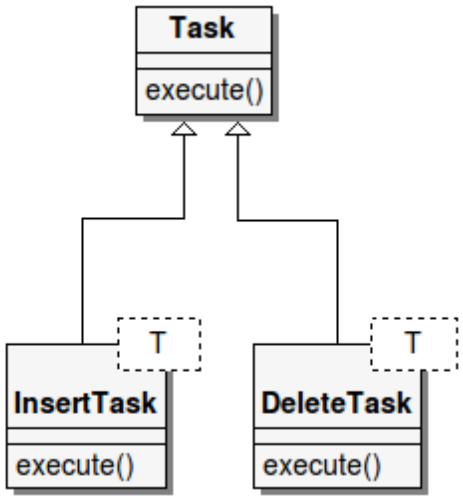


Abbildung 3.2: Klassendiagram Task

Beim Einfügen bzw. Löschen wird das in Abschnitt 2.3.2 beschriebene Verfahren verwendet. Die dort geschilderte `heapify`-Operation ist im `InsertTask` und `DeleteTask` enthalten und wurde daher nicht als zusätzlicher `Task` definiert.
Ein Auszug aus dem Programmcode soll den Löschvorgang noch einmal veranschaulichen (siehe Abb. 3.3). Zu Beginn wird in Zeile 6 eine Variable auf dem Locale erstellt, der die

Löschoperation ausführt. Das Ergebnis wird dann in Zeile 14 in diese Variable geschrieben und in Zeile 16 an den aufrufenden Locale zurückgegeben. Dabei ist zu beachten, dass im Fall von Klasseninstanzen nur eine Referenz auf das Objekt selbst dort gespeichert wird. D.h. das Objekt steht im lokalen Speicher des Rechenknotens, der die Klasse instanziiert hat (siehe Abschnitt 2.1). Abhilfe schafft dort bspw. ein Kopierkonstruktor[3] oder eine vordefinierte Methode zum Klonen bzw. kopieren des Objekts, wofür wiederum abstrakte Klassen oder Interfaces benötigt werden um den Entwickler explizit darauf aufmerksam zu machen, dass diese Methoden implementiert werden müssen.
In Zeile 7 wird sichergestellt, dass jede Anfrage über die Wurzel unseres d-ary Heaps gehen muss, bevor sie den Zielknoten erreicht. Zeile 9 enthält die Definition des auszuführenden `Tasks`. Jeder `Task` benötigt eine ID, damit vor der Ausführung überprüft werden kann um welche Operation es sich handelt. Ein boolescher Wert würde dafür genügen, jedoch ist es aus historischen Gründen noch ein Integer, da auch `heapify` als eigenständiger `Task` existiert hat. Diese ID ist nötig, da derzeit die Typprüfung zur Laufzeit nicht für generische Klassen funktioniert. Der Taskexecutor auf dem Wurzelknoten muss vor der Ausführung des `Tasks` die Gesamtgröße der DPQ anpassen und die Zielposition bestimmen. Deshalb wird diese Information mit Hilfe von einer ID hinterlegt und zur Laufzeit vom Taskexecutor überprüft.
In Zeile 10 wird der in der vorherigen Zeile erstellte `Task` der lokalen `BlockingQueue` hinzugefügt. Dabei blockiert die `BlockingQueue` solange sie leer ist, so das der Taskexecutor automatisch seine Arbeit fortsetzt, sobald ein neuer `Task` hinzugefügt wurde.
Abschließend muss nun der aufrufende Locale in Zeile 13 so lange warten, bis das Minimum von der DPQ entfernt wurde. Nun wird wie bereits erwähnt in Zeile 14 das Ergebnis in der zu Anfang deklarierten Variable gespeichert (bzw. eine Referenz darauf) und in Zeile 16 zurückgegeben.
Das Einfügen neuer Elemente verläuft analog. Jedoch kann dort auf den abschließenden Lock (`task.lock$`) verzichtet werden und der Methodenaufruf kehrt direkt nach dem hinzufügen des `Tasks` zurück.
Der Versuch einen neuen Chapel Task zu erstellen, der auf dem Locale 0 ausgeführt wird, hat leider aufgrund der sehr vielen Tasks die erstellt wurden zu Schwierigkeiten geführt. Ziel war es das beschriebene Verhalten (insert nicht blockierend - der aufrufende Prozessor bzw. Rechenknoten kann seine Arbeit direkt fortsetzen) des Algorithmus in Abschnitt 2.3.2 umzusetzen. Bei den Tests gab es immer wieder Probleme, da es nicht möglich war neue

3 Ein Konstruktor, der mit Hilfe einer Referenz auf ein Objekt desselben Typs eine Kopie erstellt

Threads zu erstellen, weil bereits die maximale Anzahl der Threads erstellt wurde. Dies hat nach einer internen Fehlermeldung von Chapel dazu geführt, dass der Programmablauf blockiert wurde.

```
1  /**
2   * Delete the item with the lowest priority. This method will block
3   * until the min item is retrieved and returned.
4   */
5  proc deleteMin(): eltType {
6      var result: eltType;
7      on Locales(0) {
8              // add task to locale queue
9              var task = new DeleteTask(SplayItem(eltType), DELETE_ID);
10             taskQueue(here.id).offer(task);
11
12             // deleting blocks until item was successfully deleted:
13             task.lock$;
14             result = task.result.item;
15     }
16     return result;
17 }
```

Abbildung 3.3: Code-Ausschnitt - deleteMin-Methode

Die Objekte der Klasse `InsertTask` und `DeleteTask` führen die angefragten Operationen aus. Beispielsweise ruft ein `InsertTask` die Methode `insert` im lokalen `SplayTree`-Objekt auf, wenn es sich bei dem Locale auf den der `Task` ausgeführt wird um den Zielknoten (vgl. Abschnitt 2.3.4) handelt. Sollte es sich bei dem Locale nicht um den Zielknoten handeln, so wird das bereits beschriebene Verfahren angewendet, bei dem überprüft wird ob das Element lokal hinzugefügt und anstelle dessen das Element mit der geringsten Priorität weitergeleitet wird (vgl. Abschnitt 2.3.2). Das Löschen bei einem DeleteTask verläuft analog.
Beim Hinzufügen von Elementen in den Splay-Baum wird überprüft, ob das Attribut `root` bereits gesetzt ist. Sollte dies nicht der Fall sein, so wird dieses auf das neu eingefügte Element gesetzt. Ansonsten wird die Stelle, an der das Element eingefügt werden soll gesucht und für das entsprechende `SplayItem`-Objekt entweder als linkes (einzufügendes Element ist kleiner) oder rechtes Kind (einzufügendes Element ist größer) gesetzt. Dabei ist es wichtig, dass die evtl. vorhandenen Teilbäume, die noch an dem SplayItem-Objekt hängen an dem eingefügt werden soll, an das neue Element gehängt werden.
Zum Einfügen wurden zwei verschiedene Varianten ausprobiert und in Kapitel 4 vergli-

chen:

- Führe die Splay-Operation aus und füge das Element an der entsprechenden Stelle hinzu. (3.2.0.1)
- Führe eine Zig-Operation aus (wenn das einzufügende Element kleiner ist als das Element, das zur Zeit in der Wurzel ist) und suche die Stelle an der das Element eingefügt werden soll. (Analog dazu eine Zag-Operation, falls das Element größer ist) (3.2.0.2)

Bei der ersten Variante wird immer der ganze Baum neu ausgerichtet, so dass das Element direkt an der Wurzel eingefügt werden kann. Die Implementierung dieser Variante ist einfacher, jedoch kann dies im schlimmsten Fall dazu führen, dass die am weitesten außen liegenden Elemente immer in die Wurzel rotiert werden müssen. Die zweite Variante hingegen führt nur eine Rotation aus und sucht anschließend die Stelle, an der das Element eingefügt werden muss. Beide Varianten werden nicht nur beim Einfügen, sondern auch beim Löschen und dem Suchen des Minimums bzw. Maximums verwendet.

4 Benchmarks

Die Benchmark-Tests wurden im ITS-Cluster der Universität Kassel durchgeführt. Es wurden bis zu 4 Rechner mit Linux Kernel Version 2.6.32 verwendet. Dabei standen je Rechner 32 Prozessoren mit 2,3 GHz und jeweils acht Rechenkernen, sowie 64 GB Arbeitsspeicher zur Verfügung. Jeder Test wurde mindestens dreimal wiederholt.
Da die Anzahl der Locales höher war als die zur Verfügung stehenden Rechner mussten sich mehrere Locales einen Rechner teilen. Dabei sind die Locales immer gleichmäßig auf die Rechner verteilt worden, d.h. bei einer Anzahl von vier Locales war jeweils ein Locale auf einem Rechner, bei acht Locales jeweils zwei usw.

4.1 Testverfahren

Für die Tests wurden zwei unterschiedliche Szenarien untersucht, welche beide von [Man98] übernommen wurden ([vgl. Man98, Kap. 4.2]). Jedoch wurde das zweite Verfahren in leicht abgewandelter Form verwendet.

Test 1

Dieses Szenario verfügt zu Beginn über eine leere DPQ. Der Locale in der Wurzel unseres d-ary Heaps fügt ein Element hinzu, woraufhin alle Locales zusammen anfangen ein Element zu entfernen und dafür wiederum zwei neue Elemente einfügen. Dabei werden insgesamt 2.500 `deleteMin` und somit 5.000 `insert` Operationen ausgeführt. Die Priorität der Elemente liegt im Intervall $[0, 10^6]$.

Test 2

In diesem Szenario beinhaltet die DPQ vor Beginn des Tests bereits 1.000 Elemente. Alle Locales führen eine `insert`-Operation gefolgt von einer `deleteMin`-Operation aus. Dies wird 5.000 mal wiederholt, so dass insgesamt 10.000 Operationen durchgeführt werden. Dabei laufen alle Locales parallel zueinander. Der Wertebereich für die Priorität wurde in diesem Szenario ebenfalls auf $[0, 10^6]$ festgelegt.

4.2 Ergebnisse

Die Abbildungen mit den Ergebnissen der Benchmarks haben im Titel die Bezeichnung der Variante, die getestet wurde. Dabei ist Variante 1 das in (3.2.0.1) und Variante 2 das in (3.2.0.2) beschriebene Verfahren. Bei beiden Testverfahren verhalten sich beide Varianten gleich und haben sehr ähnliche Laufzeiten.

Test 1

Für das erste Testverfahren (siehe Abb. 4.1 und Abb. 4.2) ist auffällig, dass mehr verwendete Locales auch höhere Zugriffszeiten mit sich bringen. Der Grund dafür sind die steigenden Kommunikationskosten, die beim Einfügen bzw. Entfernen von Elementen anfallen. Darüber hinaus sieht man die Auswirkungen bei der Wahl von d des d-ary Heaps. Allgemein scheint zu gelten, dass je größer d ist, desto niedriger ist die benötigte Zeit für den Test. Erst ab zwölf Locales holen die Heap Varianten mit kleineren d langsam auf und sind später sogar schneller. Dieses Verhalten kann ebenfalls auf die Kommunikationskosten zurückgeführt werden. Beispielsweise sind bei acht Locales mit $d = 7$ deutlich weniger `heapify`-Operationen notwendig als für $d = 2$. Der Grund für den Anstieg ab zwölf Locales sind die vermehrt auftretenden Kosten für das `heapify`, welche bei den kleineren Werten für d im Hintergrund auf den tieferen Ebenen des Teilbaumes laufen. Dadurch blockieren sich die Einfüge- und Löschoperationen nicht so häufig gegenseitig, was zu besseren Ergebnissen führt. Zusätzlich sei erwähnt, dass Tests mit d bis zu einem Wert bis 15 durchgeführt worden sind. Dabei wurden die besten Ergebnisse stets erreicht, wenn der Heap eine Tiefe von zwei hatte ($d = n - 1$). Das heißt, dass je mehr `heapify`-Operationen auf den verschiedenen Ebenen des Baumes durchgeführt werden

mussten, desto schlechter hat die DPQ aufgrund der erhöhten Kommunikationskosten abgeschnitten.

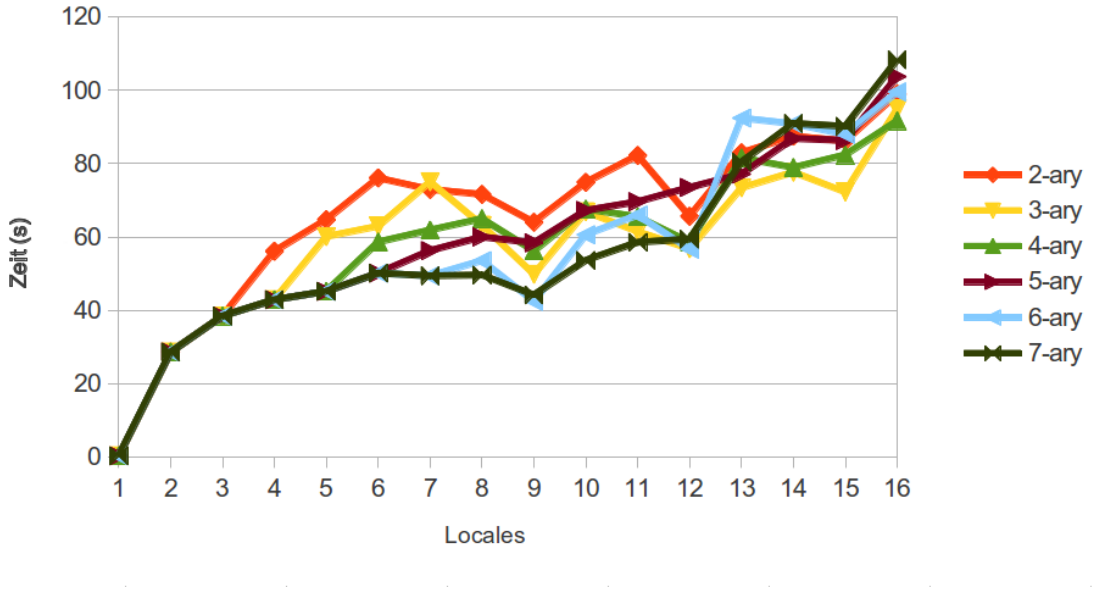


Abbildung 4.1: Testverfahren 1 mit Variante 1

Test 2

Beim zweiten Testverfahren sticht sofort das auffällige „auf und ab“ der Laufzeit bei steigender Anzahl der Locales hervor. Dabei wächst die Laufzeit recht konstant, solange $d \geq n$ ist. Der Grund dafür ist das verwendete Testverfahren. Alle Locales fügen erst ein Element hinzu und entfernen anschließend eins. Da dies alle Locales parallel zueinander tun, kommen die Anfragen immer fast in geordneter Reihenfolge auf dem Locale in der Wurzel des Heaps an. Die abwechselnd aufeinander folgenden hohen und niedrigen Laufzeiten resultieren aus der Verteilung der Elemente, die vor Beginn des Tests hinzugefügt wurden. Ein gutes Beispiel zum Erläutern dieses Phänomens ist der Testlauf mit zehn Locales. Die 1.000 hinzugefügten Elemente liegen gleichverteilt auf den Locales, denn jeder Locale hat genau 100 Elemente in seinen lokalen Splay-Baum. Jetzt wird durch ständiges hinzufügen und löschen von Elementen fast immer nur der Locale 0 (Wurzelknoten) beansprucht, da

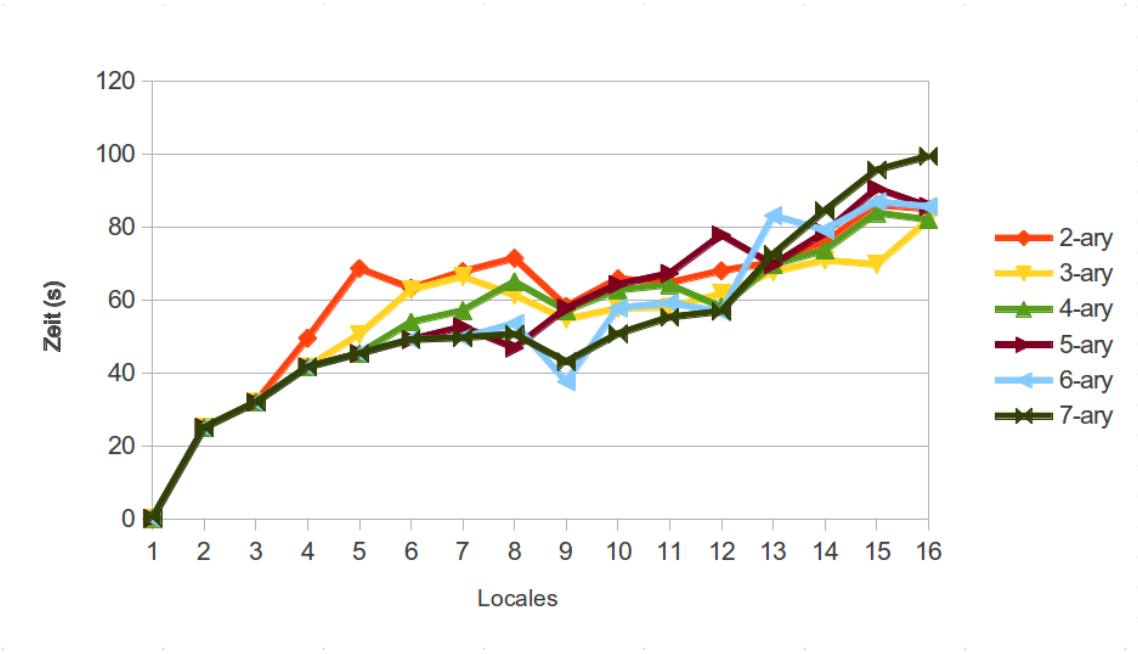


Abbildung 4.2: Testverfahren 1 mit Variante 2

dieser die Zielposition für beide Operationen ist. Dadurch verringern sich die Kommunikationskosten enorm, was dazu führt, dass der Test mit 10 Locales deutlich schneller läuft als mit neun oder elf Locales. Denn diese haben als Zielposition zum Löschen und Einfügen einen Knoten auf einer tieferen Ebene.

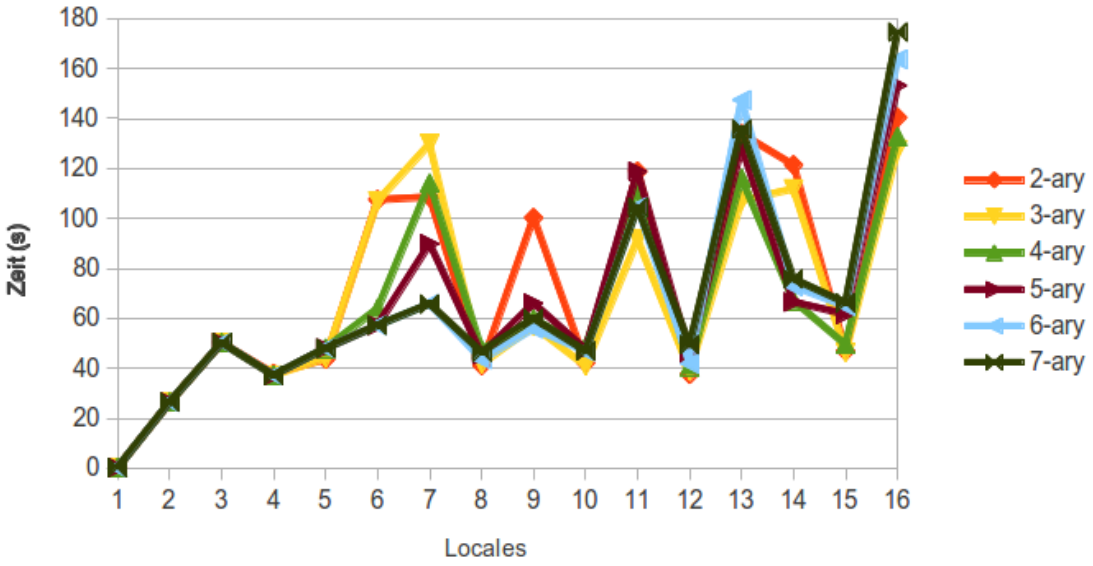


Abbildung 4.3: Testverfahren 2 mit Variante 1

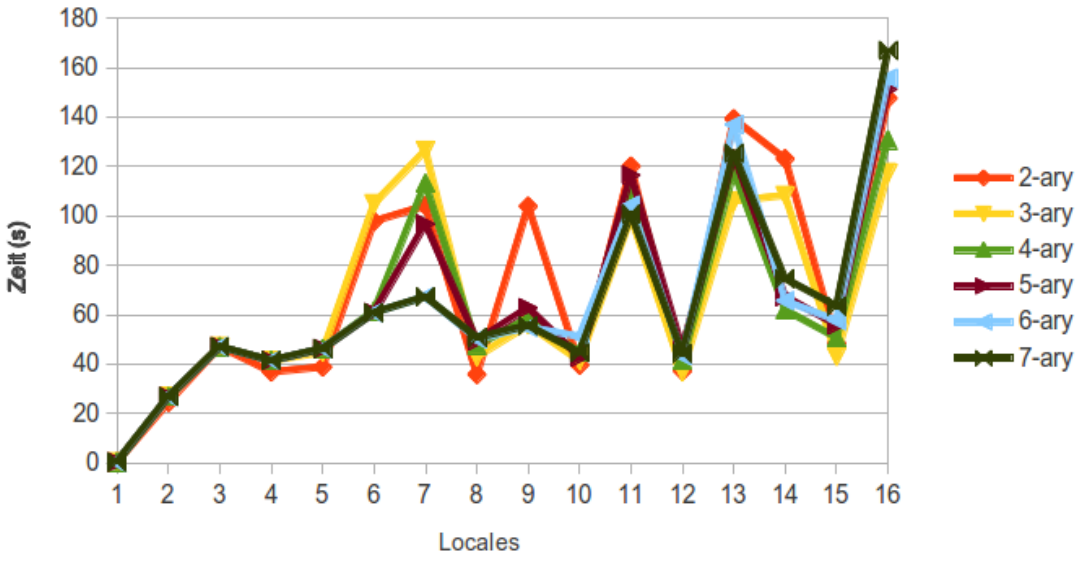
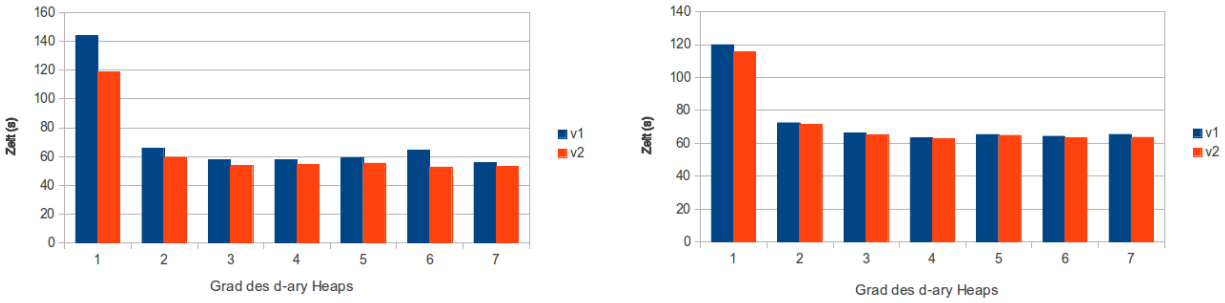


Abbildung 4.4: Testverfahren 2 mit Variante 2

Vergleich der Splay-Baum Varianten

Abb. 4.5 zeigt noch einmal den direkten Vergleich beider Varianten. In beiden Testverfahren kann Variante 2 schnellere Zugriffszeiten vorweisen.



(a) Testverfahren 1 - vgl. von Variante 1 & 2

(b) Testverfahren 2 - vgl. von Variante 1 & 2

Abbildung 4.5: Vergleich beider DPQ-Varianten mit den verwendeten Testverfahren

5 Zusammenfassung, Ausblick und Fazit

Zusammenfassung

Diese Ausarbeitung hat zu Beginn einen Überblick über das parallele Programmiersystem Chapel gegeben. Dabei wurde die Motivation, die zu der Entwicklung einer neuen parallelen Sprache führte und die wichtigsten Grundprinzipien erläutert.
Im Anschluss dazu folgte eine kurze Beschreibung der Vorrangwarteschlange und ein ausführliches Kapitel über die DPQ von Bernard Mans aus [Man98], welche zweistufig aus d-ary Heap bzw. Binomialbaum und Splay-Bäumen aufgebaut ist. Sie gewährleistet Lastbalancierung, Nebenläufigkeit und Konsistenz.
Abschließend wurde die Implementierung mit Hilfe von Klassendiagrammen und einem Codebeispiel erörtert. In diesem Zusammenhang wurde auch auf Komplikationen, die sich bei der Entwicklung mit Chapel ergeben haben, eingegangen. Es gab z.B. Probleme aufgrund fehlender Funktionalitäten (Interfaces, abstrakte Klassen) und interner Fehler im Zusammenhang mit Vererbung bei generischen Klassen. Zudem bereitete die fehlende Typprüfung für generische Klassen und die von Chapel zur Verfügung gestellten Tupel Schwierigkeiten. Darüber hinaus war es nicht möglich ein Tupel als leer zu kennzeichnen.
Die Benchmarks haben gezeigt, dass die Kommunikationskosten einen großen Einfluss auf die Performance der DPQ haben. Je tiefer der Baum des d-ary Heaps ist, desto höher sind die Kommunikationskosten und damit die Laufzeit der durchgeführten Testverfahren.

Ausblick

Einerseits wäre ein Vergleich mit Implementierungen in anderen Sprachen wie beispielsweise X10[1] oder Fortran[2] interessant gewesen. Damit hätte die Performance, aber auch die Komplexität der Programmierung und andere Vor-/Nachteile der Sprachen herausgearbeitet werden können.
Des Weiteren wäre ein etwas komplexeres Beispielprogramm, welches einerseits die DPQ und andererseits eine andere Methode zur Datenverwaltung verwendet, praktisch gewesen um festzustellen wie stark die DPQ die Performance beeinträchtigt.

Fazit

Die Entwicklung mit Chapel war eine neue und interessante Herausforderung. Dabei erwies sich der Einstieg in die Sprache schwieriger als erwartet. Zum einen war das Schreiben von Programmcode ohne Entwicklungsumgebung wie Eclipse oder Netbeans, welche dem Entwickler viel Arbeit abnehmen können, sehr fehleranfällig. Zum anderen war der Umgang mit einigen Sprachkonstrukten wie Arrays nicht immer so ausführlich dokumentiert wie erhofft. Sehr oft musste man die Tutorials auf der Chapel Homepage durchforsten um die gewünschte Antwort zu finden. Allerdings war das Einrichten von Chapel sowie das Erstellen eines ersten lauffähigen Programms sehr einfach und ist trotz der umfangreichen Dokumentation dazu in kürzester Zeit geschafft.
Chapel hat mich trotz der unterschiedlichsten Probleme überzeugt. Im Vergleich zu anderen vergleichbaren Sprachen wie X10 oder OpenMP bietet Chapel leistungsstarke Konstrukte, mit denen man mit deutlich weniger Code dieselben Funktionalitäten implementieren kann. Dabei ist einer der größten Vorteile, dass diese Programme eine hohe Portabilität aufweisen und dass sie in Bezug auf Performance bereits mit anderen parallelen Programmiersystemen mithalten können ([vgl. CCD+13, Kap. IV]).
Die DPQ ist eine effiziente und nützliche Datenstruktur, die sich jedoch nur für große Datenmengen lohnt, bei der der Speicher eines Rechenknotens nicht ausreicht. Dabei ist die benötigte Rechenleistung der DPQ sehr niedrig, wodurch die Performance der Anwendungsprogramme nicht stark beeinflusst wird. Das Hauptproblem liegt bei dem

1 Parallele und Objektorientierte Programmiersprache von IBM. Sie existiert seit 2004 und befindet sich wie Chapel noch immer in der Entwicklung.
2 Fortran steht für FORmula TRANslation und war ursprünglich eine prozedurale Programmiersprache. Seit Fortran 2003 wird aber auch Objektorientierung unterstützt und seit 2008 zusätzlich Parallelisierung.

Wurzelknoten, über den alle Anfragen laufen. Dies stellt Flaschenhals der DPQ dar. Sollte eine große Menge Anfragen in kurzer Zeit eingehen, so verzögern sich die Zugriffszeiten enorm. Außerdem sind die benötigten Kommunikationskosten (zumindest in Chapel) sehr hoch. Daher eignet sich die DPQ insbesondere für Anwendungsprogramme, die nach dem Entnehmen von Elementen rechenintensive Aufträge abzuarbeiten haben und damit zwischen den Zugriffen viel Zeit benötigen. Für solche Programme sollte die DPQ in der Praxis aufgrund der Wiederverwendbarkeit und der daraus resultierenden geringeren Entwicklungszeit (Datenstruktur muss nicht neu entworfen werden) eine große Rolle spielen.

Abbildungsverzeichnis

Literaturverzeichnis

[CCD+13] Chamberlain, Bradford L.; Choi, Sung-Eun; Dumler, Martha; Hildebrandt, Thomas; Iten, David; Litvinov, Vassily; Titus, Greg: The State of the Chapel Union / Cray Inc. Seattle, may 2013. – Forschungsbericht. – `http://chapel.cray.com/papers/ChapelCUG13.pdf`; Stand 08/2013

[Cha12] Chamberlain, Bradford L.: *State of the Chapel Union: HPCS Reflections and Musing about the Future*. Webseite, 2012. – `http://chapel.cray.com/presentations/ChapelForPGAS2012-presented.pdf`; Stand: 08/2013

[Cha13] Chamberlain, Bradford L.: A Brief Overview of Chapel / Cray Inc. Seattle, Januar 2013. – Forschungsbericht

[CMSW11] Chamberlain, Bradford L.; Murphy, Richard C.; Stark, Dylan; Wheeler, Kyle B.: *The Chapel Tasking Layer Over Qthreads*. Webseite, 2011. – `http://chapel.cray.com/publications/CUG11-wheeler.pdf`; Stand: 08/2013

[Man98] Mans, Bernard: Portable distributed priority queues with MPI. In: *Concurrency: Practice and Experience* (1998), März, S. 175–198

[Pel89] Peleg, David: Complexity Considerations for Distributed Data Structures / Weizmann Institute of Science. Israel, 1989. – Forschungsbericht

[spe13] *Chapel Language Specification 0.93*. Webseite, 2013. – `http://chapel.cray.com/spec/spec-0.93.pdf`; Stand: 08/2013

[SS06] Saake, Prof. Dr. G.; Sattler, Prof. Dr. Kai-Uwe: *Algorithmen und Datenstrukturen*. 3. Auflage. dpunkt.verlag GmbH, 2006

[ST85] Sleator, Daniel D.; Tarjan, Robert E.: Self-Adjusting Binary Search Trees / AT&T Bell Laboratories. New Jersey, Januar 1985. – Forschungsbericht. – `http://www.cs.princeton.edu/courses/archive/fall07/cos521/handouts/self-adjusting.pdf`; Stand: 08/2013