Karlsruhe Institute of Technology

Parallele Breitensuche in X10
Second title line

Diplomarbeit
von

Dominic Rausch

An der Fakultät für Informatik
Institute for Program Structures
and Data Organization (IPD)

Erstgutachter:	?
Zweitgutachter:	?
Betreuender Mitarbeiter:	Andreas Zwinkau
Zweiter betreuender Mitarbeiter:	?

Bearbeitungszeit: XX. Monat 20XX – XX. Monat 20XX

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

PLACE, DATE

.....

(**YOUR NAME**)

Inhaltsverzeichnis

1. Einleitung

Die Breitensuche (engl: Breadth first search, kurz BFS) ist einer der Standardalgorithmen zur Graphtraversierung. Ausgehend von einem Knoten, dem Wurzelknoten, werden alle transitiv erreichbaren Knoten gesucht Zu jedem der erreichbaren Knoten kann außerdem die Distanz, gemessen in Kantenanzahl, oder der Vorgängerknoten ausgegeben werden. Die Breitensuche findet Anwendung, wenn der kürzeste Weg von einem Knoten zu allen anderen berechnet werden soll. Der von der Breitensuche erzeugte Schichtengraph wird zum Beispiel in Dinic's Algorithmus zur Lösung des Max-Flow Problems wie in [Din06] verwendet.

Lange Zeit wurden Geschwindigkeitssteigerungen der Computer vor allem durch erhöhte Taktraten erreicht. Der Intel 4004 Chip von 1971 taktete mit 108kHz, der 2002 eingeführte Pentium M schon mit 1.7 GHz. 2005 führte Intel den ersten echten Mehrkernprozessor ein([Int06]), der zwei vollständige Kerne auf einem Chip vereinte. Über 30 Jahre lang optimierte man Prozessoren darauf, einen einzelnen, sequentiellen Befehlsstrang möglichst schnell ausführen zu können. Da aber kein Wechsel des Paradigmas stattfand, skalierten vorhandene Anwendungen sehr gut mit der Geschwindigkeit der Prozessoren. Seit einigen Jahren ist aber klar, dass weitere Geschwindigkeitssteigerungen nur durch (massive) Parallelität geschehen können.

Die Breitensuche ist ein einfacher Algorithmus, dessen sequentielle Version in ein paar Zeilen Code ausgedrückt werden kann. Das Problem bei der Anpassung an parallele Programmierparadigmen ist, dass keine voneinander unabhängigen Aufgaben für einzelne Bearbeitungsfäden definiert werden können. Sowohl Daten- als auch Kontrollfluss müssen teuer synchronisiert werden.

Ziel: Das Paper von [BM11] beschreibt Ansätze zur Parallelisierung der Breitensuche. Dabei werden zur Kommunikation vor allem MPI Operationen eingesetzt. Wie die Autoren des Papers bereits bemerken, bleibt die Frage offen, ob die vorgeschlagenen Konzepte auch mit modernen, implizit parallelen, Programmiersprachen funktionieren und performant sein können, da eine abstraktere Beschreibungssprache meistens weniger flexibel ist und etwas mehr Overhead benötigt. Weiter soll erforscht werden, wie die Breitensuche mit den Besonderheiten des Inasiven Rechnens zurecht kommt. Dabei sei vor allem die Asymmetrie der Rechenleistung erwähnt, das heißt, einzelne Prozesse haben zum Teil bedeutend mehr Rechenleistung als andere.

2. Grundlagen

2.1. Die Programmiersprache X10

Die Programmiersprache X10 wird seit 2004 am IBM T. J. Watson Research Center bei New York in Kooperation mit einigen Universitäten entwickelt und gepflegt. X10 ist eine stark typisierte, objektorientierte Progammiersprache ohne Mehrfachvererbung. Was sequentielles Programmieren angeht, entspricht die Featuresliste weitgehend denen, die von anderen modernen objektorientierten Sprache bekannt sind. Nach Aussage der Entwickler ist X10 im Moment vor allem noch ein Forschungsobjekt und noch nicht für den Produktiveinsatz geeignet. Das Forschungs- und Entwicklungsziel von X10 ist es, eine Abstraktion von parallelen Programmierung zu finden, die es dem Entwickler erlaubt produktiver zu sein, als es mit traditionelleren Sprachen wie Java möglich wäre. Dazu wurden der Programmiersprache Konstrukte hinzugefügy, die einen impliziteren Umgang mit Parallelismus erlauben. Das Programiermodell entspricht am ehesten PGAS. Die 3 wichtigesten Aspekte sollen hier kurz beschrieben werden.[x1012]

2.1.1. Aktivitäten und das Keyword *async*

Das Konzept der Aktiviäten in X10 ist dem der Threads sehr ähnlich. Beim Programmstart wird automatisch eine Aktivität gestartet, die bei der main-Methode einsteigt. Eine Aktivität hat einen Programmzähler und einen eigenen Stack und läuft potentiell nebenläufig zu allen anderen Aktivitäten ab. Tatsächlich werden alle Aktivitäten aus einem Threadpool versorgt, der aber für den Programmierer vollkommen transparent ist. Eine neue Aktivität wird mit dem Keyword *async* gestartet. *x = async calculateX();* führt nebenläufig *calculateX()* aus und weist x dann das Ergebnis zu. Um auf die Fertigstellung zu warten, gibt es das Keyword *finish*{...}. Bevor ein finish-Block verlassen wird, wird gewartet, bis alle (auch transitiv) gestarteten Aktivitäten beendet wurden.[VSG12]

2.1.2. Places und das Keyword *at*

Ein Place ist in X10 die Abstraktion eines Prozessors oder eines Prozessorkerns, auf dem gerechnet werden kann. Zwei unterschiedliche Places haben keinen gemeinsamen Speicher. Um zwischen Places zu wechseln erfolgt kein explizites Message Passing. Stattdessen verwendet man das Keyword *at(place)* { /* *code* */ }. Mit *at* wechselt die aktuelle Aktivität den Place. Dazu werden bei jedem *at* alle Objekte, die innerhalb des Blockes verwendet werden, auf den Ziel-Place kopiert. Der Kontrollfluss kehrt erst nach vollständiger Ausführung des Blockes wieder zu dem initialen Place zurück. Ein *at* ist also ein vollkommen

synchrones Konstrukt. Änderungen an Daten innerhalb eines *at*-Blockes werden nicht auf den initialen Place übernommen. Bei jedem *at*, das von Place p nach k wechselt, werden alle benötigten Daten erneut von p nach k kopiert. Um Daten dauerhaft an einem Place zu halten, gibt es das Konzept der DistArrays.[VSG12]

2.1.3. Distributions und distributed Arrays

Distributions und DistArrays sind das Konzept von X10 um Daten auf Places aufzuteilen. Im X10 Jargon nenn man den Index eins Arrays *Point*. Eine Distribution ist ein Objekt, dass zu jedem Point eines Arrays einen Place ausrechnet. Ein DistArray (DistributedArray) ist nun ein Array, dessen Werte verteilt auf verschiedenen Places liegen und von dem jeweiligen Place aus erreichbar sind. Wo ein Wert liegt, definiert die Wahl der Distribution, die schon bei Arrayerstellung bekannt sein muss. Es kann beispielsweise auf jedem Place ein möglichst gleichgroßer, zusammenhängender Block liegen (Block Distribution) oder jeweils eine Sequenz von k Werten auf einem Place, die nächsten k auf dem nächsten Place liegen, wobei nach dem letzten wieder der erste Place kommen (Cyclic Distribution).[VSG12]

2.2. Breitensuche

Breitensuche (oft BFS vom englischen Breadth-First-Search) ist einer der fundamentalen Graphtraversierungsalgorithmen der Informatik. Die Breitensuche startet bei einem Knoten, dem Wurzelknoten. Die Ausgabe der Breitensuche ist zu jedem Knoten eine Zahl, die BFS Distanz. Die BFS Distanz des Knoten i ist die Länge des kürzesten Pfades von der Wurzel zu i, gemessen in Anzahl der traversierten Kanten. Außerdem gibt die Breitensuche zu jedem Knoten i den Vorgängerknoten j aus, sodass der vorletzte Knoten auf einem kürzesten Pfad von der Wurzel zu i der Knoten j ist. Damit gibt die Breitensuche also für einen gegebneen Startknoten den kürzesten Weg zu jedem erreichbaren Knoten sowie dessen Länge aus.

2.2.1. Funktionsweise

Es wird jedem Knoten vor Beginn die BFS Distanz ∞ zugewiesen, einzig der Startknoten bekommt die Distanz 0, außerdem wird er einer Liste von aktiven Knoten hinzugefügt. In einer Breitensuchiteration wird nun jeder Knoten, der von einem der aktiven Knoten aus erreichbar ist angeschaut. Ist seine BFS Distanz ∞, wird die Distanz auf die um eins erhöhte Distanz des Vorgängerknotens gesetzt. Ist die Distanz kleiner als unendlich, wird der Knoten ignoriert. Die Vereinigung aller Knoten, deren BFS Distanz wärend einer Iteration reduziert wird, ist die Menge der aktiven Knoten für die nächste Iteration. Sobald am Ende einer Ieration kein Knoten füt die nächste Iteration als aktiv markiert ist, ist der Algorithmus fertig.

Der Algorithmus berechnet für jeden Knoten k also folgendes Minimum[HBP10]:

$$bfsDistanz(k) = \min_{v \in Vorgänger\ von\ k} (level(v)) + 1$$

2.2.2. Sequentieller BFS Pseudocode

Der Pseudocode für eine sequentielle Breitensuche kann wie in Algorithmus 1 aussehen. Statt den zwei Listen *current* und *nexts* wird oft eine einzelne Queue verwendet. Dadurch spart man sich die äußere Schleife und iteriert stattdessen solange über die Elemente der Queue, bis sie leer ist. Hier wird eine Variante mit zwei Listen, eine für die aktuelle und ein für die nächste Iteration, bevorzugt, weil sie der parallelen Version im nächsten Kapitel ähnlicher ist und die Laufzeit dadurch nicht schlechter wird.

Algorithm 1 Sequentielle Breitensuche

```
1:  current : List<Node>()
2:  nexts : List<Node>()
3:  startNode : Node
4:  ∀i : bfsDistance(i) ← ∞
5:  bfsDistance(s) ← 0
6:  current.add(s)
7:  while current.size() > 0 do
8:      for all nodes i in current do
9:          for all successors j of i do
10:             if bfsDistance(j) < ∞ then
11:                 bfsDistance(j) ← bfsDistance(i) + 1
12:                 nexts.add(j)
13:             end if
14:         end for
15:     end for
16:     current ← nexts
17:     nexts.clear()
18: end while
```

2.2.3. Analyse

Eine Breitensuche hat die asymptotische Laufzeit von $O(n + m)$, wenn n die Anzahl der Knoten und m die Anzahl der Kanten ist Informell ist das dadurch begründbar, dass die innerste Schleife (Zeile 9-13) höchstens m mal durchlaufen wird (insgesamt, nicht pro Schleifendurchlauf der äußeren Schleife). Da jeder Knoten höchstens einmal zu *current* hinzugefügt wird und in jeder Iteration *current* geleert wird, kann die Bedignung in Zeile 7 höchstens n mal erfüllt sein. Ebenso kann jeder Knoten höchstens einmal in Zeile 8 aus current genommen werden. Damit wird Zeile 9 $O(n)$ mal erreicht. Eine obere Schranke im O-Kalkül ist also $O(max(n, m))$, was gerade $O(n + m)$ entspricht.

3. Paralleler Algorithmus

3.1. Vorüberlegungen

Eine einzelne Instanz der Breitensuche ist relativ schwer und nur unter recht hohem Synchronisationsaufwand nebenläufig lösbar. Das liegt daran, dass keine unabhängigen Ausführungsstränge definierbar sind. Überlegungen, etwa jedem Prozess eine starke Zusammenhangskomponente des Graphen zur Berechnung zu geben, scheitern daran, dass allein die Laufzeit der Graphpartitionierung schon mindestens so lang wie die der Breitensuche ist.

Weiterhin muss zu jedem Knoten die aktuelle BFS-Distanz gespeichert werden, was O(n) Speicheraufwand bedeutet. Um bei sehr großen Graphen nicht extern arbeiten zu müssen, wird deswegen gefordert, bei p Prozessen mit O(n/p) Speicherbedarf je Prozess auszukommen. Für eine bessere Anschaulichkeit werden in den folgenden theoretischen Überlegungen zunächst Adjazenzmatrizen als Graphrepräsentation verwendet, auch wenn reale Graphen meist dünn besetzt sind und Adjazenzmatrizen deswegen eher ungeeignet sind. Dabei sei $A(i, j) = true$, wenn eine gerichtete Kante von i nach j existiert. Im folgenden werden die zwei grundlegenden Konzepte der Breitensuche wie vorgeschlagen von [BM11] vorgestellt. Der Name 1D bzw. 2D Partitionierung bezieht sich dabei auf die Aufteilung der Daten auf Prozesse.

3.2. 1D Partitionierung

Bei der 1D Paritionierung wird die Adjazenzmatrix entlang einer Achse aufgeteilt. Die Aufteilung erfolgt derart, dass jeder Knoten genau einem Prozess gehört und jeder Prozess möglichst gleichviele Knoten hat. Es ist wichtig, dass sehr schnell herausgefunden werden kann, welchem Prozess ein bestimmter Knoten gehört. Außerdem wird definiert, dass alle Kanten, die von Knoten k ausgehen, demselben Prozess gehören, wie Knoten k. Diese Partitionierung entspricht einer horizontalen Zerschneidung der Matrix.

$$\left(\begin{array}{c} \dots \\ \hline \textit{Daten von Prozess } 1 \\ \hline \dots \\ \hline \textit{Daten von Prozess } 2 \\ \hline \dots \\ \hline \dots \\ \hline \textit{Daten von Prozess } p \end{array}\right)$$

Diesem Muster folgend wird auch das BFS-Distanz Array partitioniert. Der Prozess, dem ein Knoten gehört, ist dem entsprechend der einzige, der die aktuelle BFS-Distanz kennt. Damit weiß auch nur dieser Prozess, ob der Knoten bereits erreicht wurde oder nicht. Sehr abstrakt kann der verwendete Algorithmus wie in 2 beschrieben werden. Dabei ist zu beachten, dass der Pseudocode an X10 angelehnt. Die erste Zeile wird nur auf dem ersten Place ausgeführt. Die anderen Places werden erst ab Zeile 2 aktiviert. Entsprechend der X10 Nomenklatur wird mit $dist(k)$ der Place gemeint, dem der Knoten k gehört.

Algorithm 2 1D-partitionierte Breitensuche

```
1:  Startknoten: s, Kantenanzahl n, Anzahl Places: p
2:  bfsDistance : DistArray of size n          ▷ Mit ∞ initialisiert, 0 an Stelle s
3:  for each place, do async on place do
4:      current : List<Nodes>(s)                          ▷ Lokale Liste pro Place
5:      while Σ_{i=0}^{p-1} #current_i > 0 do
6:          //Phase 1:
7:          for u ∈ current do
8:              for each neighbor v of u do
9:                  put u in the sendbuffer for place dist(u)
10:             end for
11:         end for
12:         //Phase 2:
13:         Send sendbuffer to corresponding place
14:         barriere
15:         //Phase 3:
16:         for u in receivebuffer do
17:             if bfsDistance(u) == ∞ then
18:                 Update bfsDistance(u)
19:                 Put u in current
20:             end if
21:         end for
22:     end while
23: end for
```

Zur Initialisierung erstellt jeder Prozess lokal eine Liste aus aktiven Knoten (Zeile 4). Die Liste wird auf allen Places leer initialisiert, außer auf dem Place, dem der Startknoten gehört. Dort wird der Startknoten in die Liste eingefügt. Der Algorithmus kann in 3 Phasen aufgeteilt werden, die jeweils lokale in einem Prozess ablaufen. Es wird solange über die 3 Phasen iteriert, bis auf allen Places die Liste der aktiven Knoten leer ist.

3.2.1. Phase 1: Adjazente Knoten sortieren

In Phase 1 iteriert jeder Prozess für sich über seine Liste an aktiven Knoten. Nach Voraussetztung stehen in der Liste nur Knoten, die dem jeweiligen Prozess selbst gehören, deswegen kennt der Prozess auch alle ausgehenden Kanten. Zu jeder Kante muss der Prozess nun herausfinden, welchem Prozess der Zielknoten gehört und den Knoten in einen entpsrechenden Sendepuffer einordnen. Am Ende dieser Phase ist die Liste der aktiven Knoten leer.

3.2.2. Phase 2

3.2.3. Phase 3

3.2.4. Allreduce

3.3. 2D Partitionierung

$$\left(\begin{array}{c|c|c} \textit{Daten von } P_1 & \textit{Daten von } P_2 & \textit{Daten von } P_3 \\ \hline \dots & \dots & \dots \\ \hline \textit{Daten von } P_{p-2} & \textit{Daten von } P_{p-1} & \textit{Daten von } P_p \end{array}\right)$$

4. Breitensuche im invasiven Kontext

5. Evaluation

...

5.1. Section 1

...

5.2. Section 2

...

5.3. Section 3

...

6. Conclusion

...

Literaturverzeichnis

[BM11] Aydin Buluç und Kamesh Madduri: *Parallel breadth-first search on distributed memory systems.* In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, Seiten 65:1–65:12, New York, NY, USA, 2011. ACM, ISBN 978-1-4503-0771-0. http://doi.acm.org/10.1145/2063384.2063471.

[Din06] Yefim Dinitz: *Theoretical Computer Science.* Kapitel Dinitz' algorithm: the original version and even's version, Seiten 218–240. Springer-Verlag, Berlin, Heidelberg, 2006, ISBN 3-540-32880-7, 978-3-540-32880-3. http://dl.acm.org/citation.cfm?id=2168303.2168313.

[HBP10] M. Amber Hassaan, Martin Burtscher und Keshav Pingali: *Ordered and unordered algorithms for parallel breadth first search.* In: *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, Seiten 539–540, New York, NY, USA, 2010. ACM, ISBN 978-1-4503-0178-7. http://doi.acm.org/10.1145/1854273.1854341.

[Int06] Intel: *The Evolution of a Revolution*, 2006. http://download.intel.com/pressroom/kits/IntelProcessorHistory.pdf.

[VSG12] Igor Peshansky Olivier Tardieu Vijay Saraswat, Bard Bloom und David Grove: *X10 Language Specification v2.2*, Januar 2012. http://x10.sourceforge.net/documentation/languagespec/x10-222.pdf.

[x1012] *X10 Website, X10 FAQ*, 2012.

Anhang

A. First Appendix Section

ein Bild

Abbildung A.1.: A figure

...