Karlsruhe Institute of Technology

Parallele Breitensuche in X10

Bachelorarbeit
von

Dominic Rausch

An der Fakultät für Informatik
Institut für Programmstrukturen
und Datenorganisation(IPD)

Verantwortlicher Mitarbeiter: Prof. Gregor Snelting
Betreuender Mitarbeiter: Andreas Zwinkau

Abgabe: 3. Oktober 2012

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.

Karlsruhe, 3. Oktober 2012

.....

(**Dominic Rausch**)

Inhaltsverzeichnis

1. Einleitung

Lange Zeit wurden Geschwindigkeitssteigerungen der Computer vor allem durch erhöhte Taktraten erreicht. Der Intel 4004 Chip von 1971 taktete mit 108 kHz, 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 in dieser Zeit kein Wechsel des Paradigmas stattfand, skalierten vorhandene Anwendungen sehr gut mit der Taktfrequenz der Prozessoren. Seit einigen Jahren ist aber klar, dass weitere Geschwindigkeitssteigerungen nur durch (massive) Parallelität geschehen können, da eine weitere Steigerung der Taktraten nur mit sehr hoher Verlustleistung möglich wäre.

In dieser Arbeit geht es im Speziellen um die Breitensuche. 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 und der Vorgängerknoten ausgegeben werden. Die Breitensuche findet Anwendung, wenn der kürzeste Weg von einem Knoten zu allen anderen berechnet werden soll. Eine andere Anwendung ist die Erzeugung des Schichtengraphs, der zum Beispiel in Dinic's Algorithmus zur Lösung des Max-Flow Problems [Din06] verwendet wird.

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

Diese Arbeit baut auf einem Paper von *Aydin Buluç* und *Kamesh Madduri* auf [BM11]. Es beschreibt Ansätze zur Parallelisierung der Breitensuche. Dabei werden zur Kommunikation vor allem MPI Operationen eingesetzt. Wie die Autoren bereits vermerken, 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. Dazu sollen nun verschiedene Datenstrukturen für Graphen getestet und der Algorithmus auf verschiedene Arten an eine PGAS-System angepasst werden. Weiter soll erforscht werden, wie gut die Breitensuche mit den Besonderheiten des invasiven Rechnens harmoniert. Als Problem wurde dabei vor allem die Asymmetrie der Rechenleistung betrachtet, was be-

deutet, dass die Rechenleistung an einigen Speicherstandorten im Sinne des PGAS-Modell zum Teil bedeutend größer ist, als an anderen.

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 und statisch typisierte, objektorientierte Programmiersprache ohne Mehrfachvererbung. Was sequentielles Programmieren betrifft, entspricht die Feature-Liste 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 paralleler 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 hinzugefügt, die einen impliziteren Umgang mit Parallelismus erlauben. Das Programmiermodell folgt dem PGAS (Partitioned Global Address Space) Speichermodel. Die drei für das Verständnis wichtigsten Aspekte von X10 sollen hier kurz beschrieben werden.[x1012]

2.1.1. Activities und das Keyword *async*

Das Konzept der *Activity* in X10 ist dem der Threads sehr ähnlich. Eine Activity hat einen Programmzähler, einen eigenen Stack und ist semantisch nebenläufig zu allen anderen Activities. Jede Activity lebt zu einem Zeitpunkt auf genau einem Place. Beim Programmstart wird automatisch eine Activity gestartet, die bei der main-Methode beginnt. Mehrere Activities rechnen also potentiell parallel. Tatsächlich werden alle Activities aus einem Threadpool versorgt, der für den Programmierer aber vollkommen transparent ist. Die Laufzeitumgebung von X10 weißt den Threads aus dem Pool nach einer eigenen Policy Activities zu, was vom Programmierer nicht direkt beeinflusst werden kann. Eine neue Activity kann ausschließlich mit dem Keyword *async* gestartet werden. *async x = calculateX();* führt nebenläufig *calculateX()* aus und weist x dann das Ergebnis zu. Um auf die Fertigstellung zu warten, gibt es das Keyword *finish*{...}, auf das ein Codeblock folgt. Bevor eine Activity einen finish-Block verlässt, wartet sie, bis alle (auch transitiv) von dieser Activity durch *async*s in diesem Block gestarteten Activities beendet wurden.[VSG12]

2.1.2. Places und das Keyword *at*

Ein Place in X10 ist die Abstraktion eines Prozessors oder eines Prozessorkerns, auf dem gerechnet werden kann. Zwei unterschiedliche Places haben keinen gemeinsamen Speicher.

Um zwischen Places zu wechseln und zu kommunizieren erfolgt kein explizites Message Passing. Stattdessen gibt es das Keyword *at(place)* { */* code */* }. Mit *at* wechselt die aktuelle Activity den Place. Der Code innerhalb des *at*-Blockes wird auf dem entfernten Place ausgeführt. 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. Die Activity, die den Place-Wechsel veranlasst hat, blockiert, bis das Ergebnis des *at*-Blockes zurück kommt. Ein *at* ist aus Programmierersicht also ein 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. Benötigte Daten sind dabei definiert, als alle Objekte und Werte, die von allen Zeigern, die innerhalb des *at*-Blockes verwendet werden, (transitiv) erreicht werden können. Oft wird ein *at* mit eine *async* verbunden. *async at(place) code* startet quasi eine neue Activity auf dem Place place, die initialisierende Activity wird aber nicht blockiert. *async at* und *finish* funktionieren ganz intuitiv miteinander. [VSG12]

2.1.3. Distributions und distributed Arrays

Distributions und DistArrays sind ein Konzept von X10, um Daten auf Places aufzuteilen. Im X10 Jargon nennt man den Index eines Arrays *Point*. Eine Distribution ist ein Objekt, das zu jedem Point eines Arrays einen Place ausrechnen kann. Ein DistArray (DistributedArray) ist nun ein Array, dessen Werte verteilt auf verschiedenen Places liegen und nur 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, usw., wobei nach dem letzten wieder der erste Place kommt (Cyclic Distribution). Auf die Werte, die auf einem Place liegen darf, ausschließlich von diesem Place aus zugegriffen werden. [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 eines kürzesten Pfades von der Wurzel zu i, gemessen in Anzahl der traversierten Kanten. Ist der Knoten nicht erreichbar, ist die BFS-Distanz ∞. Außerdem gibt die Breitensuche zu jedem Knoten i den Vorgängerknoten j aus, sodass j der vorletzte Knoten auf einem kürzesten Pfad von der Wurzel zu i ist. Damit gibt die Breitensuche also für einen gegebenen 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 ∞, so wird die Distanz auf die um eins erhöhte Distanz des Vorgängerknotens gesetzt. Ist die Distanz kleiner als unendlich, wird der Knoten ignoriert, da der kürzeste Weg bereits gefunden wurde. Die Vereinigung aller Knoten, deren BFS Distanz während einer Iteration reduziert wird, ist die Menge der aktiven Knoten für die nächste Iteration. Sobald am Ende einer Iteration kein Knoten für die nächste Iteration als aktiv markiert ist, ist die Berechnung abgeschlossen.

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 einzelnen Queue, sie leer ist. Hier wird eine Variante mit zwei Listen, eine für die aktuelle und eine für die nächste Iteration, bevorzugt, weil die Laufzeit darunter nicht leidet, sie den Vorteil hat, dass sie der parallelen Version im nächsten Kapitel aber ähnlicher ist.

Algorithm 1 Sequentielle Breitensuche

```
1:  current : List<Node>()
2:  nexts : List<Node>()
3:  s : 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 Bedingung 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. Zeile 9 wird also sowohl höchstens n mal, als auch höchstens m mal erreicht. Eine obere Schranke im O-Kalkül ist damit $O(max(n, m))$, was gerade $O(n + m)$ entspricht.

2.3. Invasives Rechnen

Im Rahmen dieser Arbeit wurde die Breitensuche für das invadeX10 Framework [Zwi12] portiert, das im InvasIC Projekt entwickelt wird. Dieses Framework realisiert das Programmierparadigma des invasiven Rechnens. Das invasive Rechnen ist ein Ansatz, der dem Programmierer die Entwicklung von ressourcengewahrem Programmcode ermöglichen soll. Das Ziel dabei ist, die Effizienz eines gesamten Systems zu steigern, das heißt

mehr Rechenjobs pro Zeit zu verrichten, als es mit herkömmlichen Scheduling möglich ist. Das kann nur dadurch geschehen, dass die Verwaltungsstelle für Rechenressourcen mehr über ein Programm weiß, als ob es rechnen will oder nicht. Der Vergleich eines einzelnen Programms, das nicht ressourcengewahr die gesamte Rechenleistung benutzt mit einem einzelnen Programm, dass ressourcengewahr arbeitet, geht im besten Fall unentschieden aus. Das ressourcengewahre Programm muss ebenso alle Berechnungen durchführen, die das herkömmliche Programm ausführt, es muss aber zusätzlich noch auf Ressourcennutzung und -verwaltung achten. Deswegen ist und soll ein invasives Programm nicht schneller, als sein herkömmlich parallelisiertes Gegenstück sein.

Im invadeX10 Framework hat jedes Programm, das zu einem Zeitpunkt läuft, einen Agent, der die Schnittstelle zum System darstellt. Über den Agent regelt das Programm Ressourcenanfragen und -abgaben. Gibt es konkurrierende Ressourcenanfragen verschiedener Programme, handeln die betroffenen Agenten unter sich eine Lösung aus.

2.3.1. Grundoperationen

Im invadeX10 Framework gibt es drei grundlegende Operationen, um mit dem Betriebssystem zu kommunizieren.

Invade Das Programm baut sich zunächst einen sogenannten Constraint zusammen. Ein Constaint ist ein Objekt, dass die Ressourcenforderung eines Programms enthält, etwa die Anzahl an Processing Elements, die eine abstrakte Repräsentation von CPU-Cores sind. Dieser Constraint wird bei dem *invade* Aufruf an den Agenten übergeben. Der Agent versucht nun, die Forderungen möglichst gut mit freien Ressourcen zu erfüllen, wenn nötig auch in Absprache mit den anderen Agenten und dem globalen System. Das Ergebnis der Anforderung wird in ein Claimobjekt verpackt an das Programm zurückgegeben. Die durch den Claim abstrahierten Ressourcen sind nun für dieses Programm reserviert. Da ein Programm bei der Operation ehemals freie Ressourcen quasi besetzt, wird sie *invade* genannt.

Infect Die *infect* Operation „infiziert“ die Ressourcen mit Programmcode und Daten. Das Programm übergibt dazu dem Claim eine Funktion, meist *ilet* genannt. Die referenzierte Funktion wird dann auf jeder Ressource dieses Claims nebenläufig ausgeführt. Zur Kommunikation stehen herkömmliche X10 Primitive und APIs zur Verfügung. Auf einen und denselben Claim kann mehrmals infect aufgerufen werden, die zugewiesenen Ressourcen „verbrauchen“ sich also nicht bei einer Berechnung.

Retreat Mit der *retreat* Operation teilt das Programm dem Agenten mit, dass bestimme Ressourcen, die dem Programm zugewiesen wurden, nicht mehr benötigt werden und damit wieder für alle anderen zur Verfügung stehen. Ein Retreat ist unwiderruflich. Das bedeutet, dass erneut ein invade aufgerufen werden muss, falls später wieder mehr Rechenleistung benötigt wird. Ein temporäres Abgeben mit späterem zurückholen von Ressourcen ist vorgesehen aber noch nicht umgesetzt.

Ein sehr einfaches Programm könnte beispielsweise folgende Struktur aufweisen:

$$invade \rightarrow infect \rightarrow retreat$$

Ein komplexeres, iterierenderes Programm könnte aber auch diese Struktur bei jeder Iteration wiederholen. Der Vorteil davon ist nicht, dass das Programm schneller ausgeführt wird, sondern die Möglichkeit, brachliegende Ressourcen abzugeben oder mehr Rechenleistung zu beantragen. Ein Programm kann jederzeit weitere Ressourcen anfordern, Ressourcen infizieren oder die eigenen Ressourcen zum Teil oder vollständig abgeben.

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 bereits daran, dass allein die Laufzeit der Graphpartitionierung schon mindestens so lang wie die der Breitensuche ist. Um grundsätzlich die BFS-Distanz eines Knotens zu setzen, muss die Distanz des Vorgängers bekannt sein. Wegen der Forderung im folgendem Absatz, liegt diese aber im Allgemeinen nicht im eigenen Speicher vor. Deswegen muss für jede Iteration der BFS in irgendeiner Weise eine Kommunikation stattfinden.

Weiterhin muss zu jedem Knoten die aktuelle BFS-Distanz gespeichert werden, was global O(n) Speicheraufwand bedeutet. Um bei sehr großen Graphen nicht extern arbeiten zu müssen, wird deswegen gefordert, bei p Prozessen mit O(n/p) + O(1) Speicherbedarf je Place auszukommen. Für eine bessere Anschaulichkeit werden in den folgenden theoretischen Überlegungen zunächst Adjazenzmatrizen als Graphrepräsentation verwendet, auch wenn reale Graphen oft so 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 [BM11] vorgestellt. Der Name 1D bzw. 2D Partitionierung bezieht sich dabei auf die Aufteilung der Daten auf Places. Es geht dabei nicht um Parallelität mit gemeinsamen Speicher. Auf jedem Place rechnet immer nur eine Activity gleichzeitig.

3.2. Datenstrukturen für Graphen

Im Laufe dieser Arbeit wurde eine serielle Version der Breitensuche implementiert, um unter anderem verschiedene Datenstrukturen für Graphen zu testen. Implementiert wurde die Breitensuche immer nach dem Schema von Algorithmus 1 auf Seite 5. Die verschiedenen Implementierungen unterscheiden sich nur in der Weise, wie über die adjazenten Kanten iteriert wird, da nur an dieser Stelle die Graphrepräsentation eine Rolle spielt. Die implementierten Datenstrukturen sind:

Adjazenzmatrizen Adjazenzmatrizen haben unabhängig von der Dichte eines Graphen einen Speicherbedarf von $\Theta(n^2)$. Die Iteration über alle Kanten, die adjazent zu einem

Knoten sind, benötigt immer $\Theta(n)$ Schritte, unabhängig davon, wie viele Kanten wirklich adjazent sind. Diese unteren Schranken machen Adjazenzmatrizen für solche Graphen geeignet, deren Ausgangsgrad in $\Theta(n)$ liegt. Pro Eintrag in der Matrix reicht 1 Bit Speicher aus.

Adjazenzlisten Die Implementierung verwendet ein Array, das für jeden Knoten eine Liste aller ausgehenden Kanten enthält. Als Listen werden Arraylisten eingesetzt. Der Aufwand zur Iteration über alle Kanten eines Knoten ist bei Adjazenzlisten linear zu der Anzahl der Kanten. Im Vergleich zu den Adjazenzarrays muss hier für jede Kante ein Integer gespeichert werden. Deswegen brauch diese Datenstruktur bei dichten Graphen mehr Speicherplatz.

Adjazenzarrays Die Implementierung von Adjazanzarrays entspricht der aus [Meh08]. Es werden zwei Arrays benötigt, eines der Länge n+1, genannt V und eines der Länge m, genannt E. Um alle angehenden Kanten eines Knoten i zu finden, muss über das Array E von der Stelle V[i] bis V[i+1] iteriert werden. Es wird V[i+1] = m+1 gesetzt um zu garantiert, dass der Eintrag in V an der Stelle i+1 für alle gültigen i existiert. Im Grunde entspricht das einer Optimierung der Adjazenzlisten, die möglich ist, weil keine neuen Kanten im Laufe des Algorithmus hinzukommen.

Die durchgeführten Tests zeigten die zu erwartenden Ergebnisse. Adjazenzlisten mittels Arrayliste oder mittels Adjazenzarrays erreichen meistens eine sehr viel höhere Performance als Matrizen. Wie zu erwarten, nimmt der Vorsprung der Listen gegenüber Matrizen ab, umso dichter die Graphen sind. Einen Unterschied zwischen der Performance von Adjazenzarrays und Adjazenzlisten konnte nicht festgestellt werden, weswegen für die weitere Arbeit ausschließlich Adjazenzlisten verwendet wurden. Diese haben weiterhin den Vorteil, dass sie für jeden Knoten eine eigene Liste verwenden. Bei der Partitionierung auf mehrere Places vereinfacht das die Implementierung.

3.3. 1D Partitionierung

Die 1D-Partitionierung ist eine Methode, um Graphdaten auf PGAS-Systemen auf die einzelnen Places aufzuteilen um darauf eine Breitensuche zu starten. Hierbei wird die Adjazenzmatrix entlang einer Dimension aufgeteilt. Die Aufteilung erfolgt derart, dass jeder Knoten genau einem Place gehört und jeder Place möglichst gleich viele Knoten besitzt. Es ist wichtig, dass jeder Place sehr schnell (in O(1)) herausfinden kann, welchem Place ein beliebiger Knoten gehört. Um dies zu erreichen, sollte es einen arithmetischen Zusammenhang von Knoten k zu seinem besitzenden Place p geben. Die nötige Arithmetik wird durch eine Distribution abstrahiert. Außerdem wird definiert, dass alle Kanten, die von Knoten k ausgehen, demselben Place gehören, wie Knoten k. Diese Partitionierung entspricht einer horizontalen Zerschneidung der Matrix.

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

Diesem Muster folgend wird auch das BFS-Distanz Array partitioniert. Der Place, dem ein Knoten gehört, ist dem entsprechend der einzige, der die BFS-Distanz dieses Knotens kennt. Daraus folgt, dass nur Activities auf diesem Place wissen, ob der Knoten bereits

erreicht wurde oder nicht. Eine sehr abstrakte Beschreibung liefert Algorithmus 2. Dabei ist zu beachten, dass der Pseudocode an X10 angelehnt ist. Die erste Zeile wird nur auf dem ersten Place ausgeführt. Die anderen Places werden erst ab Zeile 2 aktiv. Entsprechend der X10 Nomenklatur wertet $dist(k)$ zu dem Place aus, 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 active nodes on any place > 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
```

Es gibt auf jedem Place zunächst genau eine Activity. Zur Initialisierung erstellt sie auf ihrem jeweiligen Place 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 in sich lokal auf einem Place ablaufen. Es wird solange über die 3 Phasen iteriert, bis auf allen Places die Liste der aktiven Knoten leer ist.

3.3.1. Phase 1: Adjazente Knoten sortieren

In Phase 1 wird auf jedem Place lokal über die Liste der aktiven Knoten iteriert. Nach Voraussetzung, dass Phase 2 korrekt abläuft und der Startknoten korrekt einsortiert wurde, stehen in der Liste nur Knoten, die dem jeweiligen Place selbst gehören, deswegen kennt der Place auch alle ausgehenden Kanten. Welchem Place der Zielknoten einer dieser Kanten gehört, ist beliebig. Zu jeder Kante muss der Algorithmus nun herausfinden, welchem Place der Zielknoten gehört und den Knoten in einen entsprechenden Sendepuffer einordnen. Jeder Place hält für jeden andern Place einen Sendepuffer bereit. Am Ende dieser Phase ist die Liste der aktiven Knoten leer.

3.3.2. Phase 2: Kommunikation

Phase 2 ist die Kommunikationsphase. Von jedem Place aus werden die Sendepuffer an die jeweiligen Empfänger geschickt. In den Puffern stehen nach Phase 1 die Knoten, die in

dieser Iteration erreichbar sind. Hier ist zu bemerken, dass jeder Place einen Empfangspuffer für jeden anderen Place bereithalten muss. Wenn es p Places gibt, gibt es also global (p^2) Empfangspuffer, je p davon liegen auf einem Place. Wenn es nur einen geteilten Empfangspuffer pro Place gäbe, wäre hier weitere Synchronisation notwendig. Es ist also eine Abwägung zwischen Speicherplatz und Rechenzeit. Das Mehr an Speicheraufwand von p Puffern ist vernachlässigbar klein, da eine leere Liste kaum Platz benötigt. In X10 wird der Empfangspuffer als DistArray implementiert, der pro Place ein Array von Empfangspuffern hält. Der Sendevorgang wurde als asynchrone for-Schleife über alle Places implementiert. Der Place-Wechsel mittels *at*{*place*} wird nur ausgeführt, wenn es tatsächlich Daten zu senden gibt. Nach dieser Phase wird eine globale Barriere benötigt, da im nächsten Schritt die Empfangspuffer ausgewertet werden, die in dieser Phase erst geschrieben werden.

3.3.3. Phase 3: BFS-Distanz aktualisieren

Jeder Prozess hat eine Menge von Empfangspuffern. Alle Knoten, die in den Puffern stehen, gehören nach korrektem Ablauf von Phase 2 dem Prozess selbst. Außerdem gilt für alle Knoten, dass sie von den Knoten der letzten Iteration aus erreichbar sind. Phase 3 entspricht dem Aktualisieren der klassischen Breitensuche. Es wird über alle Knoten in allen Empfangspuffern iteriert. Wenn die BFS-Distanz noch auf ∞ steht, wird sie auf die Nummer der aktuellen Iteration gesetzt und der Knoten der Liste der aktiven Knoten hinzugefügt, wenn die BFS-Distanz kleiner als ∞ ist, wird der Knoten ignoriert und verworfen. Um die BFS-Distanz auf die Nummer der aktuellen Iteration zu setzen, muss die Anzahl der Iterationen mitgezählt werden. Durch die Synchronisation kann das lokal auf jedem Place passieren. Ein einfacher Schleifenzähler reicht aus.

3.3.4. Place-lokale Parallelität

Auch wenn das eigentliche Thema dieser Arbeit nicht die Parallelisierung auf Shared-Memory Architekturen ist, soll hier kurz beschrieben werden, wie die einzelnen Phasen auf einem Place nebenläufig implementiert werden können, falls mehr als ein Kern auf einem Place zur Verfügung steht.

Phase 1 Hier ist einfache Schleifenparallelität möglich. Dabei muss beachtet werden, dass Zugriffe auf die Sendepuffer synchron passieren. In X10 lässt sich das einfach mit einem *finish*, einem *async* und einem *atomic* für den Pufferzugriff bewerkstelligen. Im Allgemeinen kann auch für jeden Ausführungsfaden ein eigenes Set an Sendepuffern erstellt werden, die dann vor dem Versenden zusammengeführt werden. Diese Methode widerstrebt allerdings der Philosophie der impliziten Parallelisierung durch X10. Mehr zu den Problemen dieser Phase ist in Abschnitt 3.3.5 zu finden.

Phase 2 Da Phase 2 nur aus dem Versenden der Puffer besteht, muss nur sichergestellt werden, dass das Versenden asynchron passiert. Da eine Synchronisationsbarriere nach dieser Phase nötig ist, müssen alle Sendevorgänge vollständig abgeschlossen sein, bevor die Barriere betreten werden darf.

Phase 3 In Phase 3 werden die Empfangspuffer ausgewertet. Auch hier ist wieder einfache Schleifenparallelität, wie sie X10 anbietet, möglich. Da die Liste der aktiven Knoten für die nächste Iteration geschrieben wird, ist wiederum ein synchronisierter Zugriff notwendig. Es gibt auch hier, wie in Phase 1, die Möglichkeit, einzelne Puffer pro Ausführungsfaden zu benutzen. Diese Optimierung wurde bei der invasiven Breitensuche angewandt und ist deswegen in Kapitel 4.2.4 genauer erklärt.

3.3.5. Optimierungen

3.3.5.1. Duplikate in Sendepuffern

Es ist sehr wahrscheinlich, dass ein Knoten in einer Iteration gleichzeitig über mehrere Kanten erreicht wird. Um so dichter der betrachtete Graph ist, desto wahrscheinlicher tritt dieser Fall auf. Da in den meisten Fällen die Kommunikation zwischen den Places der Flaschenhals ist, muss versucht werden, die zu versendenden Datenmengen so klein wie möglich zu halten.

Menge frei von Duplikaten machen Zwischen Phase 1 und 2 oder während Phase 1 kann ein Algorithmus angewandt werden, der die Sendepuffer duplikatfrei macht. Das kann mittels Hashing oder sortierten Mengen passieren.

Knoten nur einmal versenden Diese Optimierung lohnt sich vor allem bei sehr dichten Graphen. Die Idee ist, dass jeder Prozess sich merkt, welche Knoten er schon einmal in einen Sendepuffer geschrieben hat. Wenn in Phase 1 der Zielknoten einer Kante ein Knoten ist, für den bekannt ist, dass er schon früher versendet wurde, kann er einfach verworfen werden. So werden die Sendepuffer nebenbei automatisch frei von Duplikaten. Am einfachsten zu realisieren ist es mit #Kanten Bits, die gesetzt werden, sobald die Kante gesehen wurde. Allerdings ist der Speicherbedarf für diese Lösung bei $O(\#Knoten)$. Bei sehr großen Graphen ist das unter Umständen nicht möglich. Die Beschleunigung, die durch diese Taktik erreicht wird, kann aber bei dichten Graphen enorm sein. In dieser Arbeit wurde bei Tests eine Verkleinerung der Sendepuffer um mehr als eine Größenordnung festgestellt.

3.3.5.2. Adjazenzlisten löschen

Eine weitere Optimierung ist nur verfügbar, wenn alle von einem Knoten aus erreichbaren Knoten innerhalb einer getrennten Datenstruktur liegen, wie zum Beispiel bei den Adjazenzlisten. Es ist so, dass die Optimierung „Knoten nur einmal versenden“ nur dafür sorgt, dass ein Place A nicht mehrmals den selben Knoten k an Place B schickt. Place B kann aber weiterhin von jedem anderen Place einmal Knoten k geschickt bekommen, wenn auch höchstens einmal. Damit nicht jedes mal, wenn Place B Knoten k empfängt, über alle adjazenten Knoten iteriert wird, kann Place B die Adjazenzliste von k nach dem ersten Auftreten von Knoten k löschen. Um deswegen keine Sonderfallbehandlung einbauen zu müssen, existiert pro Place eine globale leere Liste für diesen Zweck. Um die asymptotische Laufzeit zu erhalten, ist es wichtig, dass diese Liste wirklich nur einmal existiert. Jedes Mal, wenn Place B Knoten k empfängt, wird über seine Adjazenzliste iteriert und danach, anstelle des Zeigers auf seine Adjazenzliste, der Zeiger auf die leere Dummyliste geschrieben. Auf diese Weise passiert nur beim ersten Auftreten des Knotens eine Iteration über alle Nachfolgerknoten. Bei jedem weiteren Empfangen von Knoten k wird die Dummyliste mit der Dummyliste ersetzt und über die leere Liste iteriert.

Das Ergebnis ist immer korrekt, da die BFS-Distanz eines Knotens k nur einmal während des gesamten Algorithmus gesetzt wird. Das passiert garantiert schon bei der Iteration, in der Knoten k das erste Mal auftrat. Auch sind alle von k aus erreichbaren Knoten in der Liste der aktiven Knoten. Somit kann die Adjazenzliste verworfen werden.

Ein Nachteil dieser Optimierung ist, dass der Algorithmus nur einmal ausgeführt werden kann, da er die Datenstruktur zerstört. Ein Unsicherheitsfaktor ist außerdem die Garbage Collection, die während der Berechnung aktiv werden könnte und den Algorithmus deswegen unvorhersehbar verlangsamen könnte. Man könnte die Adjazenzliste auch in einen Pool von toten Listen hängen, um die GC aufzuschieben.

3.3.5.3. Zusammenlegung der ersten und letzten Phase

Es ist möglich, in der letzten Phase die aktiven Knoten nicht in eine eigene Liste zu schreiben, sondern sofort in die Sendepuffer einzuordnen. Auf diese Weise muss man pro BFS-Iteration nur einmal alle aktiven Knoten anschauen.

3.4. 2D-Partitionierung

Eine andere Möglichkeit der Aufteilung der Daten ist die 2D-Partitionierung. Die zwei Dimensionen der Partitionierung beziehen sich auf die zwei Dimensionen, in denen im Modell mit der Adjazenzmatrix diese unterteilt wird. Die Aufteilung auf Places geschieht dann wie in Abbildung 3.1 illustriert. Jedem Place gehört genau eine Kachel der Matrix. Jeder Place hat dadurch eine Koordinate in der Unterteilung. Diese Koordinate entspricht der Zeile und der Spalte der Unterteilung, in dem die eigene Kachel der Matrix liegt. In Abbildung 3.1 hat Place 0 die Koordinate (0,0) und Place 6 die Koordinate (1,2). Die Matrix wird so unterteilt, dass die Kacheln möglichst quadratisch und gleich groß sind.

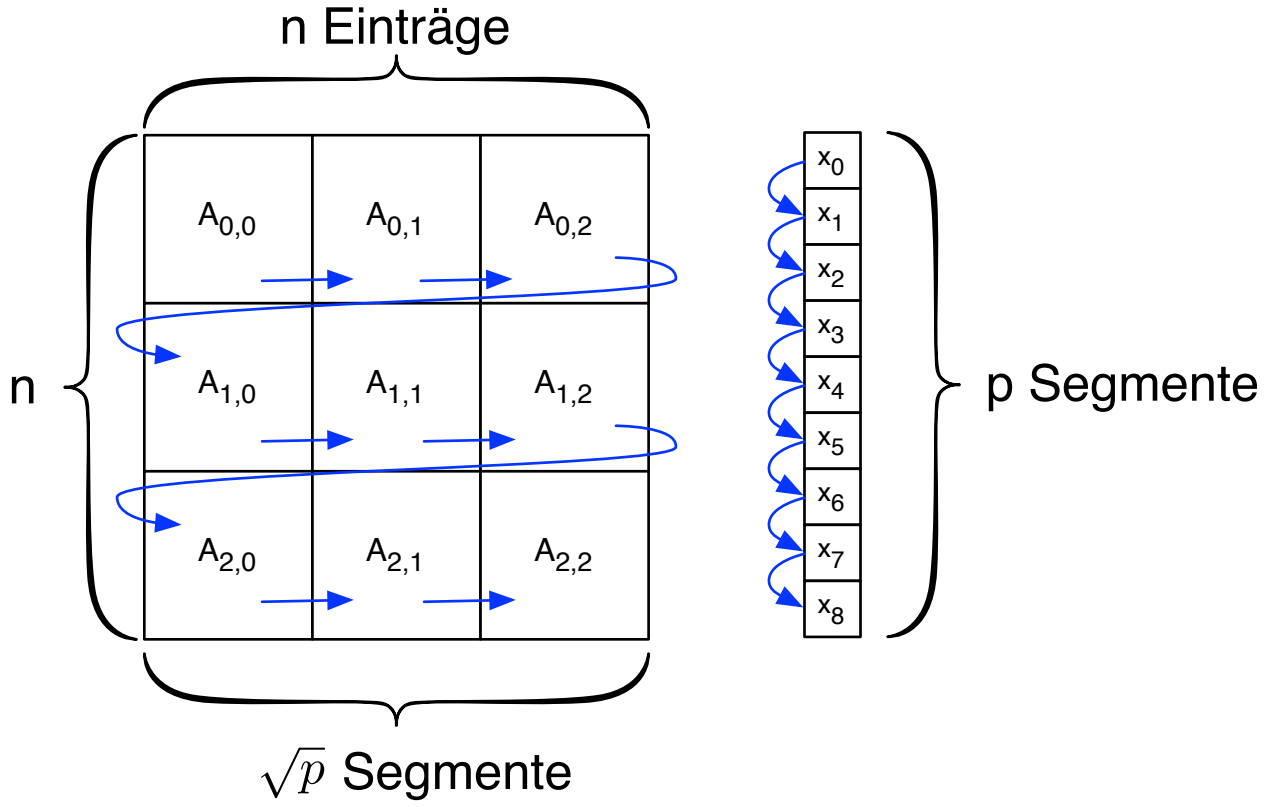


Abbildung 3.1.: Der hier gezeigte quadratische Fall ist ein Spezialfall. n ist die Anzahl der Knoten, p die Anzahl der Places. In diesem Beispiel gibt es 9 Places. Dadurch, dass die Anzahl der Places eine Quadratzahl ist, ergibt sich horizontal und vertikal die selbe Zerlegung. Der Place mit der Nummer 0 speichert $A_{0,0}$ und x_0. Die blauen Pfeile zeigen, in welcher Reihenfolge die Daten auf die nachfolgenden Places aufgeteilt werden. Auf dem zweiten Place liegen demnach $A_{0,1}$ und X_1.

Auch bei der 2D-Partitionierung müssen Vektoren auf die einzelnen Prozesse aufgeteilt werden. Angenommen die Adjazenzmatrix sei horizontal in h und vertikal in v Abschnitte

unterteilt, außerdem sei n (Anzahl an Kanten) restlos durch p (Anzahl an Places) teilbar. Es gilt $h * v = p$. Die Menge der Places, die in der ersten Zeile der Unterteilung sind, besitzt dann den Abschnitt $[1, n] * \left[1, \frac{n}{h}\right]$ der Adjazenzmatrix. Jedes Teilstück eines Vektors, den ein Place besitzt, ist $l = \frac{1}{v} * \frac{n}{h} = \frac{n}{p}$ groß. Dadurch ist sichergestellt, dass das Teilstück eines Vektors, das der ersten Reihe gemeinsam gehört($\left[1, \frac{n}{p}\right] \cup \left[\frac{n}{p} + 1, \frac{2n}{p}\right] \cup \cdots \cup \left[\frac{(v-1)n}{p} + 1, \frac{vn}{p} = \frac{n}{h}\right]$) dem horizontalen Teilstück $\left[1, \frac{n}{h}\right]$ des Matrixabschnittes entspricht, das genau dieser Reihe gehört.

Bei der zweidimensionalen Unterteilung wird der Breitensuch-Algorithmus als Folge von Matrix-Vektor-Operationen betrachtet, die einzeln parallelisiert werden. Angenommen der Algorithmus läuft auf nur einem Core, dann kann die Kommunikation und die Synchronisation für den Moment weggelassen werden. Der Pseudocode 3 beschreibt für diesen Fall die Essenz des Algorithmus. Der Algorithmus führt Iterationen aus, bis x_k der Nullvektor ist:

$$x_{k+1} = \underbrace{(A^T * x_k)}_{\substack{=1 \text{ an Stelle p} \\ \text{wenn Knoten p} \\ \text{von einem Knoten aus} \\ x_k \text{ erreichbar ist}}} * \underbrace{\neg \sum_{i=1}^{k} x_i}_{\substack{=1 \text{ an Stelle p, wenn} \\ \text{p noch unerreicht}}}$$

Das Produkt aus den beiden Vektoren entspricht der komponentenweisen Konjunktion der booleschen Werte. Die Matrixmultiplikation alleine reicht nicht aus, da Knoten sonst mehrfach als erreicht markiert werden könnten und somit eine falsche, zu große Distanz ausgegeben werden könnte. Es ist zu beachten, dass im Pseudocode die BFS-Distanz nicht gespeichert wird. Die BFS-Distanz kann aber leicht über die Nummer der Iteration herausgefunden und gesetzt werden.

Algorithm 3 BFS auf einem Place

```
1: f(s) ← true
2: t ← 0^n // Nullvektor
3: while f ≠ 0^n do
4:     x ← A^T * f
5:     f ← x * ¬t
6:     t ← t + f
7: end while
```

Um den Code von Algorithmus 3 auf mehrere Places zu parallelisieren, werden die einzelnen Schritte betrachtet und die Daten entsprechend ausgetauscht. Die Zeilen 5 und 6 sind jeweils komponentenweise Operationen und können daher lokal auf jedem Place stattfinden. In Zeile 4 wird die Matrix in der transponierten Form verwendet. Da der Algorithmus nie die nicht transponierte Form der Matrix nutzt, wird in Zukunft mit A die bereits transponierte Matrix bezeichnet. Um Zeile 4 zu parallelisieren ist einiger Aufwand notwendig. Um x[k] zu berechnen, muss sowohl die gesamte k-te Zeile aus A, als auch der gesamte Vektor f bekannt sein. Jeder Place besitzt aber nur einen Teil beider Datenstrukturen.

Der Algorithmus wird dazu wieder in Phasen zerteilt, die in Algorithmus 4 zu sehen sind.

Für die folgenden Erläuterungen wird angenommen, dass es n Knoten gibt und die Adjazenzmatrix $(n \times n)$ auf g^2 Places aufgeteilt ist, also in der Vertikalen und der Horizontalen je g Unterteilungen stattfanden. Der Place p besitze den Ausschnitt der Matrix $[x_{von}, x_{bis}] \times [y_{von}, y_{bis}]$. Dieser Ausschnitt sei in der i-ten Reihe von oben und der j-te Spalte von links.

Algorithm 4 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:     // Place an Position i,j im Grid
5:     while f ≠ 0 do
6:         //Phase 1: Transpose f
7:         for node ∈ local part of f do
8:             j ← node/colsize
9:             put node in sendbuffer for column j
10:        end for
11:        //Phase 2: Kommunikation
12:        fTransposed ← ∑_{places in row j} sendbuffer
13:        barriere
14:        //Phase 3: Lokale Matrixmultiplikation
15:        for i = rowFrom → rowTo do
16:            t[i] ← A|_{i−th row} · fTransposed
17:        end for
18:        //Phase 4: Ergebnisse der Reihe zusammenführen
19:        for node ∈ t do
20:            Send node to owners t-vector
21:        end for
22:        barriere
23:        //Phase 5: Lokale updates durchführen
24:        f ← t ∗ (d == ∞)
25:        Set d[i] = iteration number, where f[i]=true
26:    end while
27: end for
```

3.4.1. Phase 1: Transponieren des Vektors f

Es soll berechnet werden, ob Knoten k im nächsten Iterationsschritt erreicht werden kann. Dazu sei $k \in [y_{von}, y_{bis}]$. Daraus folgt, dass Place p einen Teil der Adjazenzmatix besitzt, der nötig ist, um festzustellen, ob k erreicht werden kann. Place p weiß allerdings nur, ob k von einem der Knoten zwischen x_{von} und x_{bis} erreichbar ist. Um für diese Iteration ein Ergebnis zu berechnen, braucht er also die Information, welche dieser Knoten im Intervall $[x_{von}, x_{bis}]$ aktuell aktive Knoten sind. Diese Information ist aber aufgeteilt auf alle Places in der j-ten Reihe der Unterteilung. Für die Berechnung benötigt tatsächlich jeder der Places in der j-ten Spalte (also unter anderem Place p) genau diese Information. Deswegen wird die erste Phase als Transponieren bezeichnet. Der Algorithmus iteriert auf jedem Place über alle aktiven Knoten und berechnet, welche Spalte der Dekomposition der Matrix wissen muss, dass dieser Knoten aktiv ist. Die Formel kann beispielsweise so aussehen:

$$Spaltennummer = \lfloor \frac{Knotennummer}{Spaltenbreite} \rfloor$$

3.4.2. Phase 2: Kommunikation

Nachdem die Knoten sortiert sind, werden sie jeweils nebenläufig an den berechneten Empfänger geschickt. In der Implementierung zu dieser Arbeit steht dazu auf jedem Empfänger ein Array Empfangspuffer bereit, sodass jeder potentielle Sender einen eigenen Platz für seine Daten hat. Das spart Synchronisation und kostet kaum zusätzlich Speicherplatz.

3.4.3. Phase 3: Lokale Matrixmultiplikation

Bevor dieser Schritt stattfindet, muss sichergestellt werden, dass der vorherige Schritt vollständig abgeschlossen ist. Dazu wird eine globale Barriere über alle Places verwendet. Die Matrixdatenstruktur sieht so aus, dass ein Place zu jedem Knoten in $[x_{von}, x_{bis}]$ eine Liste aller Knoten hält, die von diesem Knoten aus erreichbar sind. Es wird über alle aktiven Knoten iteriert und die Listen der erreichbaren Knoten aller aktiven Knoten aneinander gehängt. Die resultierende Liste enthält Knoten, die in der nächsten Iteration aktiv sind, falls sie nicht schon aktiv waren. Es ist zu beachten, dass jeder Knoten in dieser Liste in dem Intervall $[y_{von}, y_{bis}]$ liegt und dass jeder Place, der ebenfalls in der i-ten Reihe ist, ein Liste von aktiven Knoten hält, die in genau demselben Intervall sind. Nur alle Places einer Reihe gemeinsam wissen, welche Knoten aktiv sein werden und welche nicht.

3.4.4. Phase 4: Ergebnisse der Reihe zusammenführen

Nach dem vorherigen Schritt weiß jeder Place von einigen Knoten, dass diese erreicht wurden. Je eine Reihe gemeinsam hat die Information, welcher von den n/h Knoten, die ihnen gemeinsam gehören, erreicht wurden. Wie oben bei der Vektordekomposition zu sehen, ist jeder Knoten genau einem Place zugeordnet. Die folgende Kommunikation passiert nur zwischen Places einer Reihe, da der Abschnitt des Distanzvektors, der dieser Reihe gemeinsam gehört, genau dem Abschnitt $[y_{von}, y_{bis}]$ entspricht. Das bedeutet, dass alle Knoten, die in einer Reihe der Dekomposition stehen, untereinander austauschen müssen, welche Knoten erreicht wurde. Die Place-Nummer des Besitzers eines Knotens ist $Placenummer = \lfloor \frac{Knotennummer}{AnzahlPlaces} \rfloor$. Jeder sendet an den Besitzer des Knotens, falls ein Knoten erreicht wurde, die Knotennummer. Wenn niemand dem Besitzer eines Knotens sendet, dass dieser erreicht wurde, ist der Knoten nicht erreicht worden. Es muss auf alle Sendevorgänge gewartet werden, bevor weitere Berechnungen stattfinden können. Deswegen ist nach dieser Phase eine globale Barriere von Nöten.

3.4.5. Phase 5: Lokale Aktualisierungen durchführen

Die letzte Phase entspricht der Aktualisierung der BFS-Distanzen der konventionellen Breitensuche. Es liegt eine Liste der in dieser Iteration erreichten Knoten vor. Zu jedem Knoten ist bekannt, ob er bereits erreicht wurde oder nicht. Falls nicht, wird die BFS-Distanz auf die Nummer der aktuellen Iteration gesetzt.

3.5. Allreduce

Ein wichtiges Detail, dass bisher noch keine Erwähnung fand, spielt sowohl in Algorithmus 2 als auch in Algorithmus 4 eine Rolle. Das Allreduce befindet sich beide Male in Zeile 5. Beide Zeilen bedeuten: *solange es mindestens einen aktiven Knoten gibt, iteriere...* Es gibt genau dann noch aktive Knoten, wenn mindestens ein Place mindestens einen aktiven Knoten hat. Es dürfen aber alle Prozesse auf allen Places erst dann aufhören zu iterieren, wenn auf keinem Place mehr ein aktiver Knoten vorhanden ist. In boolescher Logik ausgedrückt, muss jeder Prozess der Allgemeinheit mitteilen, ob er noch aktive Knoten hat. Diese Information muss mittels ODER verknüpft werden. Das Ergebnis der Verknüpfung muss allen Prozessen mitgeteilt werden. Diesen Vorgang nennt man an MPI angelehnt *Allreduce*. Es wurde die Implementierung aus der X10 Standard Library benutzt. Falls X10 zum Beispiel über MPI kommuniziert, ist es unter Umständen wesentlich schneller, die Allreduce Operation aus einer MPI Hardware zu nutzen, als die selbe Funktionalität nachzuprogrammieren.

3.6. Optimierungen

Auch bei der Dekomposition in zwei Dimensionen gilt es zu beachten, dass die Listen, die in Phase 3 erstellt und in Phase 4 versendet werden, Duplikate enthalten können. Um so dichter ein Graph ist, um so wichtiger ist es, diese Listen frei von Duplikaten zu machen. Siehe dazu Abschnitt 3.3.5.1. Ohne weitere Abwandlungen kann auch die Optimierung 3.3.5.3 angewandt werden. Je nach gewählter Datenstruktur können auch die Adjazenzlisten wie in 3.3.5.2 gelöscht werden, was bei der Implementierung zu dieser Arbeit gemacht wurde.

4. Breitensuche im invasiven Kontext

In diesem Kapitel wird beschrieben, wie die Breitensuche unter Verwendung des Frameworks invadeX10 implementiert wurde. Aus Zeitgründen konnte in dieser Arbeit nur ein Ansatz herausgearbeitet werden, der noch nicht alle Möglichkeiten des invasiven Rechnens nutzt. Deswegen ist womöglich an einigen Stellen im Algorithmus nicht sofort ersichtlich, weswegen bestimmte Lösungsansätze gewählt wurden. Das liegt daran, dass die Struktur der Implementierungen so gewählt wurde, dass zukünftige Ideen leicht umsetzbar sind.

4.1. Unterschiede zum nicht invasiven Fall

Zunächst seien in diesem Kapitel einige Unterschiede in den Anforderungen und der Problemstellung erläutert. Die Beschreibung der jeweiligen Lösungen finden sich dann in Kapitel 4.2.

4.1.1. Asymmetrie der Rechenleistung

Im invasiven Rechnen gibt es das Konzept des Processing Elements (abgekürzt PE). Ein PE ist die abstrakte Repräsentation eines Rechenkerns, auf dem ein Thread ausgeführt werden kann. Jedes PE repräsentiert genau eine Recheneinheit in der Hardware, die in einem bestimmten Bereich liegt. Ein Bereich wird durch gemeinsamen Speicher definiert. Alle Rechenkerne, die sich Speicher teilen, liegen in dem selben Bereich. Das invadeX10 Framework benutzt die X10 Funktionalität der Places um im Code diese Bereiche zu repräsentieren. Es kann zum Beispiel ein Prozessor mit 8 Kernen vorliegen. Im invadeX10-System existieren dann 8 PEs, die alle auf dem selben Place liegen, also gemeinsam auf den selben Hauptspeicher zugreifen. Synchronisation und Kommunikation zwischen zwei Activities, die auf dem selben Place laufen, verläuft wesentlich schneller, was sowohl Bandbreite als auch einmalige Startzeit der Kommunikation betrifft, als wenn die Activities auf unterschiedlichen Places liegen.

Es existiert also gewissermaßen eine zweistufige Hierarchie. Das Gesamtsystem ist in Places aufgeteilt, die wiederum in PEs aufgeteilt werden. Die Places entsprechen Computern im Cluster und die PEs entsprechen einzelnen Rechenkernen. Soweit ist die Situation identisch mit der Breitensuche aus Kapitel 3.3, die diese Zweistufigkeit zunächst ignoriert und auf jeden Place gleich viele Daten und damit gleichviel Rechenarbeit legt. Es wird auf jedem Place zunächst genau eine Activity gestartet, sozusagen eine Masteractivity, die dann zum Beispiel auf Schleifenebene Nebenläufigkeit erzeugt. Die beiden Hierarchiestufen

werden explizit getrennt implementiert. Dieser Ansatz geht davon aus, dass auf allen Places gleichviel Rechenleistung zur Verfügung steht, er betrachtet Asymmetrien nicht.

Im invasiven Fall fragt das Programm bei dem Agenten nach Rechenleistung und bekommt daraufhin eine gewisse Menge an PEs als Antwort zurück. Selbst wenn durch Constraints festgelegt würde, dass die PEs gleichmäßig auf Places verteilt sein sollen, kann der Agent das in Abhängigkeit der aktuellen Situation des Gesamtsystems nicht garantieren. Wird der Agent gezwungen, gleichviel Rechenleistung auf jedem Place zu reservieren, liegt womöglich Rechenleistung brach. Deswegen sollte sich das Programm nicht darauf verlassen, dass die PEs gleichmäßig über Places verteilt sind. Das Programm befindet sich also in der Situation, dass es beispielsweise drei PEs, eine auf Place 0 und zwei auf Place 3 hat. Die erste, intuitive Konsequenz muss sein, dass auf Place 3 doppelt so viele Daten wie auf Place 0 liegen. Im Falle der Breitensuche würden Place 3 also doppelt so viele Knoten gehören wie Place 0. Wird nun ein infect auf die drei PEs aufgerufen, werden korrekterweise drei Activities gestartet, die alle denselben Code ausführen. Allerdings stehen die zwei Activities, die auf dem selben Place laufen, in einem anderen Verhältnis zueinander, als zwei Activities, die auf verschiedenen Places laufen. Die Activities haben für die folgenden Erläuterungen die Namen 0, 3.1 und 3.2.

- Will Activity 0 Daten an 3.1 und 3.2 schicken, so ist es effizienter diese in einem Kommunikationsvorgang zusammenzufassen, als das IPC-System zweimal zu starten.
- Ebenso sollten Activity 3.1 und 3.2 gemeinsam ihre Daten an Activity 0 schicken, statt zwei IPC-Vorgänge auszuführen.

Eine Activity muss also den gesamten Kontext kennen und wissen, ob und in welchem Fall sie sich mit anderen Activities auf dem selben Place zusammen tun sollte und wann nicht. Dies wird dadurch erschwert, dass alle Activities gleichberechtigt sind. Es gibt also keine „Masteractivity“ für Synchronisation und Kommunikation pro Place. Die zwei Hierarchiestufen sollen nicht explizit programmiert werden.

4.1.2. Dynamische Ressourcenverwaltung und Verteilung der Daten

Hier ist prinzipiell ein Designentscheidung zu treffen, die schlussendlich nur ein Kompromiss sein kann, da sich zumindest bei der Breitensuche nur schwer folgende Ziele vereinbaren lassen:

- Dynamisches invade und retreat je nach benötigter Rechenleistung, um nicht benötigte Ressourcen für Andere freizugeben
- Daten derart verteilen, dass die Rechenleistung ideal ausgenutzt wird.

Erklärung: Die Situation sei wie in 4.1.1, ein PE auf Place 0, zwei PEs auf Place 3. Es sei bereits eine Iteration der Breitensuche abgeschlossen. Nun ist bekannt, wie viele Knoten auf Place 0 und Place 3 aktiv sind. Die Situation sei so, dass beide ungefähr gleich viele Knoten in der nächsten Iteration zu bearbeiten haben, obwohl Place 3 ja die doppelte Menge an Knoten besitzt, also doppelt so viele Knoten potentiell aktiv sein könnten. Die Hälfte der Rechenleistung auf Place 3 ist damit in der nächsten Iteration nicht verwendbar, da die beiden Activities auf Place 3 nach Beendigung der nächsten Iteration auf Place 0 warten müssten, der für die selbe Iteration ungefähr doppelt so lange benötigen wird. Eine ressourcengewahres System würde also eine der beiden PEs auf Place 3 abgeben. Damit stünde auf den beiden Places gleichviel Rechenleistung zu Verfügung. Durchschnittlich ist die Liste der aktiven Knoten auf Place 3 aber doppelt so lang, wie die Liste auf Place 0. Es ist sehr wahrscheinlich und es wird in der Praxis passieren, dass die Liste der aktiven Knoten auf Place 3 wieder deutlich länger wird, als auf Place 0. Das heißt, es ist zu erwarten, dass genau die abgegebene Rechenleistung später wieder gebraucht wird. Die

Anwendung kann nun versuchen zu reagieren, indem sie probiert, wieder eine weitere PE auf Place 3 zu bekommen. Allerdings könnten bereits alle anderen PEs auf Place 3 von anderen Anwendung besetzt sein. Zudem entspricht es nicht dem Paradigma des invasiven Rechnens, dem Agenten mitteilen zu müssen, welche PE genau gewünscht ist.

Die naheliegende Antwort auf dieses Problem ist die dynamische Umverteilung der Graphdaten, so dass gleich viele Knoten auf Place 0 und 3 liegen, zumindest bis die gewünschte PE wieder verfügbar ist. Dieser Ansatz wurde im Rahmen dieser Arbeit nicht betrachtet, da er zum einen sehr aufwendig zu implementieren ist und zum anderen zu erwarten ist, dass die Performance so schlecht ist, dass die Umverteilung sich nicht lohnt. Es müssten große Datenmengen (Adjazenzlisten, Distanzarrays) verschickt werden, größere Arrays alloziert werden, usw.

Zusammenfassend kann man sagen, dass ein Algorithmus, der auf partitionierten Daten (ein Graph bei BFS) arbeitet, nur sehr schwierig gleichzeitig temporär ungenutzte Ressourcen abgeben und trotzdem noch maximal effizient arbeiten kann. In dieser Arbeit entspricht zwar jede BFS-Iteration einem *invade*, allerdings werden zwischen den Iterationen keine Ressourcen abgegeben oder angefordert.

4.1.3. Nicht fortlaufende Indizes

Trotz Verwendung des invadeX10 Frameworks, soll die X10 API weiter genutzt werden können. Die X10-Funktionalitäten rund um Distributions und DistArrays arbeitet auf der Basis von Places und vor allem deren Nummerierung. Aus diesem Grund kann man im invasiven Fall nicht komplett von der Vorstellung der Places weggehen, sondern muss im Auge behalten, auf welchem Place welche Daten liegen. Im reinen X10 sind die Places immer von 0 bis p - 1 (bei p Places) durchnummeriert. Das macht die Nummer eines Places zu einem sehr praktischen und auch nötigen Ausgangspunkt, um mit Indizes zu rechnen. Es kann zum Beispiel ein DistArray der Größe p erstellt werden, wenn auf jedem Place genau ein Datum liegen soll und jeder Place kann mit seinem eigenen Index auf seine eigenen Daten zugreifen.

Im invasiven Fall werden vom Agent zunächst nur ProcessingElements bereitgestellt. Auf welchem Place diese liegen, ist für den Klienten nur bedingt beeinflussbar, sofern er nicht bereit ist, auf Rechenleistung zu verzichten. Dadurch ist keine natürliche Nummerierung der Places mehr vorhanden. Es lässt sich nicht vermeiden, etwas Speicherplatz und Rechenleistung zu benutzen, um dieses Problem zu lösen. Der unten stehende Vergleich des Codes zum Feststellen, welcher der besitzende Place von Knoten k ist, verdeutlicht das Problem. *placesList* ist eine Liste aller Places und *mapNodeToPlaceIndex* ein Funktion. Damit diese funktioniert, muss vorher schon eine Datenstruktur mit dem vollständigen Wissen über alle PEs initialisiert worden sein. Wie genau das abläuft ist weiter unten beschrieben.

Algorithm 5 Durchnummerierter Fall, wie in Kapitel 3.3

```
1: val owner = k / p
```

Algorithm 6 Nicht durchnummerierter Fall, wie in diesem Kapitel

```
1: val ownerId = mapNodeToPlaceIndex(k) // Position des owners in der placesList
2: val owner = placesList[ownerId]
```

Das Problem wurde gelöst, indem mit eine Art virtuelle Placesverwaltung eingeführt wurde. Direkt nachdem bekannt ist, welche PEs zur Verfügung stehen, werden alle Places in einer Liste sortiert. Ab diesem Zeitpunkt wird ausschließlich mit dem Rang eines Place in

der sortierten Liste gerechnet. Die Ränge sind durchnummeriert von 0 bis p - 1, womit die gewünscht Situation hergestellt ist. Die einzige Stelle, an der wieder das ursprüngliche Placeobjekt verwendet werden muss, ist beim Argument eines at-Blocks. Der Zugriff geht über den Index in konstanter Zeit.

4.2. Ablauf des Algorithmus

Aus Zeitmangel wurde im Rahmen dieser Arbeit die Breitensuche im invasiven Fall nur mittels der 1D-Dekomposition implementiert. Grundsätzlich ist der Algorithmus implementiert, wie er bereits in Kapitel 3.3 beschrieben wurde. In diesem Kapitel werden deswegen in erster Linie die gewählten Lösungsansätze zu den Problem aus Kapitel 4.1 beschrieben.

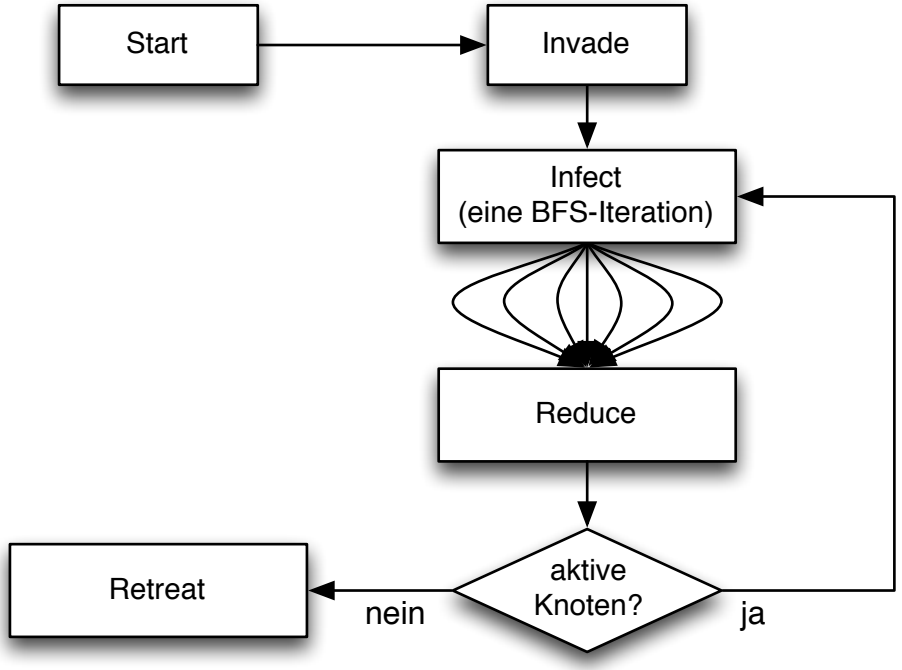


Abbildung 4.1.: Grundsätzlicher Ablauf der Breitensuche. Zu beachten ist, dass zwischen den einzelnen Iterationen kein *retreat* und kein invade stattfindet.

4.2.1. Dekomposition und Datenhaltung

Sobald der Algorithmus nach dem Start den Graph vollständig eingelesen hat und die Antwort des *invade* bekommen hat (eine Liste von PEs), beginnt er damit, sich die benötigten Datenstrukturen aufzubauen.

- ProcessingElements nach Placenummer sortieren. Liste aller involvierten Places in eben dieser Reihenfolge aufstellen und zu jedem Place die Anzahl an verfügbaren PEs in ein Array schreiben. An der Stelle 0 in diesem Array steht also, wie viele PEs auf dem Place zur Verfügung stehen, der in der Placeliste an erster Stelle (Index 0) steht.
- Die Menge an Knoten wird so unterteilt, dass jedes PE einen gleichgroßen Teil bekommt. Wichtig ist, dass alle PEs, die auf dem selben Place liegen, benachbarte

Intervalle der Knotenliste bekommen. Es wird entsprechend Platz für die Adjazenzlisten auf den Places reserviert. Alle Knoten, die zu einem Place gehören, gehören allen dortigen PEs gemeinsam. Um aufzulösen, welchem Place ein Knoten gehört, wird ein Array berechnet, dass für jede PE (in der Reihenfolge der Liste aus dem ersten Punkt) den Rang des Places enthält, auf welchem es liegt. Um den Besitzer von Knoten k zu finden, muss so zur Laufzeit nur noch k durch die Anzahl an PEs geteilt werden und das Ergebnis als Index für den Arrayzugriff verwendet werden. Das Ergebnis ist wiederum die Stelle der Placeliste, an der der besitzende Place steht.

- Die selbe Initialisierung muss für das DistArray passieren, dass die BFS-Distanzen halten soll.
- Es wird pro Place für jeden anderen Place ein Empfangspuffer erstellt. Diese Empfangspuffer befinden sich in einem DistArray mit einer UniqueDist, das bedeutet, dass auf jedem Place genau ein Datum(=Array aus Puffern) liegt.
- Es werden pro Place genau soviele Listen für aktive Knoten erstellt, wie es aktive PEs auf diesem Place gibt. Diese liegen gemeinsam in einem Array.

4.2.2. Zweistufiges Infect und Indizierung

Der Algorithmus verwendet die *Clock*-Funktionalität von X10. Eine Clock entspricht einer Barriere, die alle Activities bei Erreichen so lange blockiert, bis alle registrierten Activities die Barriere erreicht haben. Es wird sowohl für alle Activities (es gibt eine Activity pro PE) global eine Clock benötigt, als auch lokal zwischen je allen Activities, die auf dem selben Place liegen. Um diese Clocks zu erstellen, ist es nötig den Infect-Aufruf manuell zweistufig zu implementieren. Zunächst wird die globale Clock erstellt, dann wird eine Activity pro involviertem Place gestartet, die sich aber alle nicht in der Clock registrieren. Anschließend wird von der je einen Activity auf jedem Place eine weitere Clock erstellt und pro PE, die auf diesem Place zur Verfügung steht eine Activity gestartet. Diese Activity registriert sich jetzt in beiden oben erstellten Clocks und beginnt mit der BFS.

Ein weiterer Vorteil des zweistufigen Infects ist es, dass sehr einfach jeder Activity ein globaler Index und ein lokaler Index zugeteilt werden kann. Der lokale Index ist die Nummer einer Activity auf einem Place. Vor allem der lokale Index wird im Weiteren sehr nützlich sein.

4.2.3. Lokale Parallelität

Wie die Kollaboration zwischen den einzelnen Places aussieht wurde bereits in Kapitel 3.3 beschrieben. Ein Place als ganzes Verhält sich im invasiven Fall exakt gleich. Allerdings gibt es keine „Masteractivity“ pro Place, die Jobs verteilen kann und für die Synchronisation sorgt. Stattdessen werden mehrere gleichberechtigte Activities gestartet, die trotzdem auf den selben Daten arbeiten sollen. In diesem Abschnitt wird beschrieben, wie diese Zusammenarbeit gelöst wurde. Dazu werden die Phasen aus Kapitel 3.3 aufgegriffen. Eine genaue Beschreibung der einzelnen Phasen findet sich auch dort und wird hier nicht wiederholt. Wichtig ist, dass jede Activity die Information hat, die wievielte von wie vielen sie auf diesem Place ist.

4.2.4. Getrennte Puffer für aktive Knoten

In der hier gewählten Implementierung hält jede PE eine eigene Liste für aktive Knoten. Dadurch hat jede PE exklusives Schreibrecht in die eigene Liste, was Synchronisation in Phase 3 einspart. Aus dem exklusiven Schreibrecht folgt sofort, dass in Phase 3 mindestens ein PE nichts zu tun hat, falls es mehr PEs auf seinem Place gibt, als es insgesamt Places gibt. In diesem Fall ist den PEs mit einer zu hohen Id kein Empfangspuffer zum lesen zugeordnet.

4.2.5. Phase 1: Adjazente Knoten sortieren

Die Liste der aktiven Knoten ist für jede PE einmal vorhanden. Es handelt sich dabei um ein Array von Arraylisten, so dass der Zugriff auf ein beliebiges Element einer beliebigen Liste in konstanter Zeit geschehen kann. Zunächst zählt jedes PE zusammen, wie viele Elemente insgesamt in allen Listen liegen, also wie viele aktive Knoten es auf diesem Place gibt. Jedes PE rechnet sich nun aus, welcher Teil der aktiven Knoten, ihr zusteht. Zum Beispiel berechnet PE 0: 5 Elemente ab Index 0 und PE1 3 Elemente ab Index 5. Aus welchen Listen diese Elemente stammen ist zu diesem Zeitpunkt nicht bekannt. Es könnten zum Beispiel alle Elemente in der zweiten Liste liegen. Der Pseudocode, um über alle Elemente zu iterieren, ist in Algorithmus 7 beschrieben.

Algorithm 7 Über mehrere Listen iterieren

```
1:  var min : Int // Start Index of Region
2:  var listIndex : Int = 0
3:  for counter ∈ length-1..0 do
        // Skip to the list containing the first element
4:      while min ≥ activeNodes(listIndex).size do
5:          min -= activeNodes(listIndex).size
6:          listIndex++;
7:      end while
        // calculate how many elements to take from current list
8:      val upperBoundForThisList = min(current(listIndex).size(), min + counter);
9:      val elementsTakenFromThisList = upperBoundForThisList - min;
10:     counter -= elementsTakenFromThisList; // if counter < 0, the outer loop ends
11:     for idx=min; idx < upperBoundForThisList; idx++ do
12:         element = activeNodes(listIndex)(idx)
13:         // Do Stuff
14:     end for
15:     min = activeNodes(listIndex).size() // The next element is from the next list
16: end for
```

Es wird zu jeder Zeit durch den *listIndex* die aktuell akive Liste gespeichert. Die erste while-Schleife springt solange zur nächsten Liste, bis der Index *min* innerhalb der aktuellen Liste ist. Dazu wird in jeder Iteration die Größe der aktuellen Liste vom Startindex abgezogen. Wenn der Startindex innerhalb der aktuellen Liste ist, wird die while-Schleife verlassen. Danach wird ausgerechnet, wie lange Elemente aus der aktuellen Liste genommen werden. Wenn noch mehr Elemente gebraucht werden, als die Liste lang ist, dann ist die Obergrenze die Listengröße, ansonsten der Startindex + Anzahl an Elementen.

Diese Implementierung gönnt sich pro PE ein Set von Sendepuffern. Ein Set von Sendepuffern besteht aus einem Sendepuffer pro beteiligtem Place. Es gibt also auf Place k $p * c_k$ Sendepuffer, wobei p die Anzahl der Places und c_k die Anzahl an PEs auf Place k ist. Nach dieser Phase ist lediglich eine lokale Barriere nötig um sicherzustellen, dass alle Sendepuffer korrekt und vollständig geschrieben wurden.

4.2.6. Phase 2: Kommunikation

In Phase 2 werden die Sendepuffer gesendet. Eine Activity ist für das Senden zu einem anderen Place genau dann zuständig, wenn die ID des Ziels modulo der Anzahl an Activities auf diesem Place der lokalen ID dieser Activity entspricht. Ist eine Activity für das Senden zu einem Place verantworlich, kopiert sie den Inhalt der c_k Sendepuffer aller PEs auf diesem Place in einen Puffer, den sie dann auf den Zielplace schiebt. Ansonsten ist

in dieser Phase nichts anzupassen. Nach dem Senden wird eine globale Barriere benötigt, damit alle Empfangspuffer korrekt geschrieben wurden, bevor diese ausgewertet werden.

4.2.7. Phase 3: BFS-Distanz aktualisieren

Wie in Phase 2 liest eine Activity genau dann einen Empfangspuffer, wenn die ID des Senders modulo Anzahl an Activities auf diesem Place die lokale ID der Activity ergibt. Diese Aufteilung der Arbeit ist nicht unbedingt gleichmäßig, dafür geht es aber ohne Synchronisation. An dieser Stelle wäre die selbe Technik wie in Phase 1 denkbar.

4.2.8. Allreduce

Das Allreduce ist im invasiven Fall insofern nicht mehr nötig, als dass es keine All-to-All Operation mehr ist. Es wird ohnehin nach jeder BFS-Iteration zunächst die Berechnung gestoppt und nur falls eine weitere Iteration nötig ist erneut ein *infect* aufgerufen. Das Ergebnis der aktuellen Iteration schreiben alle Activities in ein Array, das Anfangs erstellt wird und das eine Zelle für jede Activity bereithält. Um das Array zu adressieren, wird jeder Activity eine GlobalRef übergeben. Die Masteractivity reduziert dieses Array dann lokal, um festzustellen, ob der Algorithmus fertig ist. Falls nicht, wird eine weitere Iteration gestartet.

5. Methoden

5.1. Graphen

Da die Breitensuche ein asymptotisch relativ schneller Algorithmus ist, sind relativ große Testgraphen nötig, um einigermaßen aussagekräftige Ergebnisse zu erhalten. Für diese Arbeit wurden Graphen in der Größenordnung von 100 000 Knoten gewählt. Diese wiederum relativ kleine Ausdehnung wurde gewählt, da der Testmodus unter anderem beinhaltet, dass bei gleichbleibender Knotenanzahl die Dichte des Graphen variiert. Sehr dicht besiedelte Graphen mit 100 000 Knoten passen gerade noch in den Arbeitsspeicher. Würden noch größere Graphen verwendet, müsste entweder die Dichte des Graphen beschränkt werden oder das Betriebssysteme müsste Speicherseiten auslagern, was die Messergebnisse unbrauchbar machen würde. Da die asymptotische Laufzeit der Breitensuche aber O(n + m) ist [Meh08], wäre eigentlich zu erwarten gewesen, dass bei gleicher (absoluter) Kantenzahl und erhöhter Knotenzahl, die Laufzeit nur leicht erhöht ist. Ein Beispiel: BFS auf einem Graph mit 1 000 000 Knoten und durchschnittlichem Knotengrad von 100 (ergibt 100 Mio Kanten) sollte kaum länger dauern, als die BFS auf einem Graph mit 100 000 Knoten und durchschnittlichem Knotengrad von 1000 (ergibt ebenso 100 Mio Kanten). Diese Annahme ist für den parallelen Fall falsch, da die zu versendenden Datenmengen erheblich größer werden, wenn der Graph mehr Knoten hat.

Es wurde ein Tool namens graph-generator [gra09] eingesetzt, um zufällige Graphen zu generieren. Um einen Graph zu erstellen, müssen folgende Parameter gewählt werden:

1. Die Anzahl an Knoten.
2. Der minimale Ausgangsgrad jedes Knotens.
3. Der maximale Ausgangsgrad jedes Knotens.
4. Der Exponent der Exponentialverteiltung. Es wurde immer 5 gewählt.
5. Der mittlerer Knotengrad z.

Der Ausgangsgrad der Knoten ist folgendermaßen verteilt:

$$P(X = k) \propto (k + offset)^{-exp}$$

Der Offset wird dabei automatisch von dem Tool derart gewählt, dass sich ein durchschnittlicher Ausgangsgrad von z ergibt.

5.2. Testplattform

Als Testplattform kann ein Apple Notebook von 2011 zum Einsatz. Es hat einen Intel Core i7-2720QM „Sandy Bridge“ Prozessor, der mit 2.2Ghz getaktet wird. Es stehen 4 physikalische Kerne zur Verfügung, die jeweils Intels Hyper Threading Technologie unterstützen. Dadurch sind physikalisch 8 parallel laufende Threads möglich. Beim Vergleich von sequentiellen Algorithmen mit parallelen ist zu beachten, dass der Prozessor einen Kern auf bis zu 3.3 GHz übertakten kann, falls die anderen Kerne momentan nicht verwendet werden. Der optimal erreichbare Speedup ist demnach nicht 8.0, sondern deutlich darunter. Das Testsystem ist außerdem mit 8GB Hauptspeicher ausgestattet, der bei einem Takt von 1333Mhz arbeitet. Auf dem Testsystem wird als Betriebssystem Mac OS X 10.6.8 „Snow Leopard“ und der x10 Compiler in der Version 2.2.3 verwendet.

5.3. Modus

Um Ergebnisse aus je einer Algorithmus - Graph - Kombination zu erhalten, wird der gewählte Algorithmus drei mal auf dem gewählten Graph ausgeführt und die Zeit gemessen, die die reine Berechnung benötigt. Die Zeit, um den Graph in den Speicher einzulesen und die Daten auf die Places aufzuteilen, wurde nicht gemessen, da sie wenig mit dem Algorithmus oder X10 zu tun hat. Die Zeit, die benötigt wird, um das Ergebnis von den beteiligten Places zurück zum Ursprungsplace zu kopieren, wird allerdings mitgemessen.

Die X10 Laufzeitumgebung liest beim Programmstart die Umgebungsvariablen X10_NPLACES, in der steht, wie viele Places lokal auf diesem Rechner simuliert werden sollen. Die einzelnen Places werden durch Prozesse (nicht Threads) repräsentiert, wodurch sie tatsächlich getrennte Speicherbereiche haben. Der Aufbau entspricht zwar nicht dem Optimalsetup, in dem jeder X10 Prozess auf einen physikalisch getrennten Computer operiert, doch auch der Kontextwechsel, der bei der Kommunikation zwischen Places auftritt, ist relativ langsam und somit eine Annäherung an realen Kommunikationsoverhead. Trotzdem sind diese Ergebnisse nicht eins zu eins auf einen Rechnerverbund zu übertragen. Das liegt zum einen daran, dass, wie gesagt, die Kommunikation nochmal erheblich teurer wird, zum anderen können mit einem Rechnerverbund wesentlich größere Graphen bearbeitet werden, die offensichtlich ein viel höheres Potential zur Parallelisierung bieten. Andererseits dürfen die Ergebnisse dieser Arbeit auch nicht mit der lokalen Parallelisierung der Breitensuche auf einem einzelnen Rechner verwechselt werden. Kommunikation mittels geteiltem Speicher ist deutlich schneller, als die hier verwendete Inter-Prozess-Kommunikation. Es sei hier auch nochmal darauf hingewiesen, dass pro Prozess, also pro Place, immer nur ein Thread aktiv ist. Um die Möglichkeit der Parallelität zu messen, wurde der Algorithmus in der 1D und der 2D Zerlegung jeweils in einer Konfiguration mit 1, 2, 4, 8 und 9 Places ausgeführt. Die Konfiguration mit 9 Places wurden hinzugenommen, da so bei der 2D Zerlegung eine symmetrische 3 mal 3 Zerlegung stattfinden kann. Es wurde vermutet, dass eine quadratische Anzahl an Places besonders günstig für diesen Algorithmus sind, obwohl 9 Places mehr sind, als Rechenkerne zur Verfügung stehen. Der Vollständigkeit halber wurde auch der 1D Algorithmus mit 9 Places durchgeführt. Um wirklich vollständige Ergebnisse zu bekommen, sollte eigentlich pro Graph die Breitensuche einmal von jedem Knoten aus gestartet werden. Allein diese Unterfangen würde den Zeitrahmen der kompletten Arbeit sprengen.

Der Modus, mit nur einem Thread pro Place und ohne die Nutztung von geteilten Speicher zwischen den einzelnen Ausführungsfäden lässt neben der Übertragbarkein auf die invasive Hardware auch Rückschlusse auf Manycore - Systeme zu. Bei diesen CPUs der Zukunft sollen viele kleine Rechenkerne auf einer CPU platziert sein. Cachekohärenz mit geteiltem Speicher ist dann kaum noch möglich. Auch wird jeder Kern nur einen Ausführungsfaden

gleichzeitig bearbeiten können. Die hier vorgeschlagenen Implementierungen sind also für solch eine Hardware interessant.

6. Ergebnisse und Diskussion

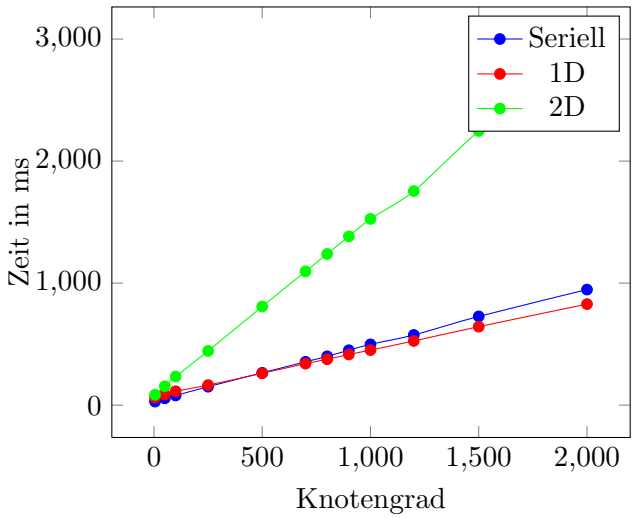
Die vollständigen Messergebnisse finden sich in den Anhängen A, B und C. Es wurden die Auswirkungen der Variation der Dichte, der Knotenzahl und der Verteilung des Graphen auf die Laufzeit gemessen, wobei jeweils die anderen Parameter fest gewählt waren. Sehr dünn besetzte und kleine Graphen waren deutlich schneller mit einer seriellen Version der Breitensuche zu lösen, als es mit jedwedem parallelen Algorithmus möglich war. Der Vergleich der seriellen Breitensuche, mit der 1D-Breitensuche mit einem Place ist durchaus auch interessant und wird in Kapitel 6.1 vertieft. Der 2D-Algorithmus schnitt in allen Testfällen dermaßen schlecht ab, dass er gesondert in Kapitel 6.3 behandelt wird.

6.1. Serieller Fall, 1D mit einem Place und 2D mit einem Place

Jeder Algorithmus muss pro Iteration jeden aktiven Knoten mindestens einmal anfassen. Außerdem muss jeder Algorithmus pro Iteration jeden der Knoten mindestens einmal anfassen, der von einem der aktiven Knoten aus erreichbar ist. Der serielle Algorithmus tut genau das und nicht mehr. In Tests wurde herausgefunden, dass eine Iteration mittels einer herkömmlichen for-Schleife mit anschließendem direkten Elementzugriff über den Index deutlich schneller ist, als eine foreach-Schleife. Der 1D-Algorithmus muss pro Iteration die Knoten in Sendepuffer einsortieren (jeden aktiven Knoten einmal anfassen) und verschicken, was im Fall mit nur einem Place aber eine einfache Zeigerzuweisung ist. Anschließend muss dann nochmals jeder aktive Knoten angefasst werden, um alle erreichbaren Knoten zu erhalten. Der 1D-Algorithmus muss also zweimal über alle aktiven Knoten iterieren, zumindest in einer naiven Implementierung. Die verwendete optimierte Version legt diese beide Phasen aber zusammen. Zusätzliche Arbeit, im Gegensatz zu der seriellen Version, hat der 1D Algorithmus also nur beim Zurückkopieren des gesamten BFS-Distanz-Arrays. Die Messergebnisse zeigen, dass der serielle Algorithmus für recht dichte Graphen langsamer ist, als der 1D Algorithmus, wenn er mit nur einem Place gestartet wird. Nur bei sehr dünn besetzten Graphen ist die Laufzeiten des seriellen Algorithmus ein Wenig schneller. Die Messergebniss zur Variation der Größe zeigen ebenfalls, dass auch bei großer Knotenzahl der serielle Algorithmus der schnellste ist, solange der durchschnittliche Knotengrad nicht zu groß wird.

Wieso der serielle Algorithmus auf Listenbasis nicht der schnellste ist, wie eigentlich erwartet, ist nicht ohne weiteres zu erklären. Zufall in Verbindung mit der Garbage Collection

Abbildung 6.1.: Vergleich der Laufzeiten bei serieller Ausführung, also nur ein Place bei 1D und 2D Algorithmus, jeweils schnellste gemessene Laufzeit.



sind in Anbetracht der Deutlichkeit der Ergebnisse auszuschließen. Eine mögliche Erklärung ist, dass der Compiler den einen Code besser optimieren konnte, als den anderen, ohne dass sofort offensichtlich ist, woran das liegt.

Wie aus Abbildung 6.1 hervorgeht, ist der 2D Algorithmus deutlich langsamer als die beiden anderen. Der 2D Algorithmus hat 2 Kommunikationsphasen pro Iteration. In den beiden Phasen wir aber jeweils nur mit $\sqrt(p)$ anderen Places kommuniziert (bei p Places)[BM11], während im Fall des 1D Algorithmus jeder Place potentiell mit allen anderen kommuniziert. Im seriellen Fall ist diese zusätzliche Komplexität nicht nötig, verlangsamt aber den Ablauf.

6.2. Ergebnisse der Parallelisierung

Da die Breitensuche mit dem 2D Dekomposition keine vergleichbaren Ergebnisse lieferte, wird hier nur der serielle Algorithmus mit dem 1D Algorithmus verglichen. Mehr zu der 2D Breitensuche steht in Kapitel 6.3. Es wurden drei Testreihen durchgeführt.

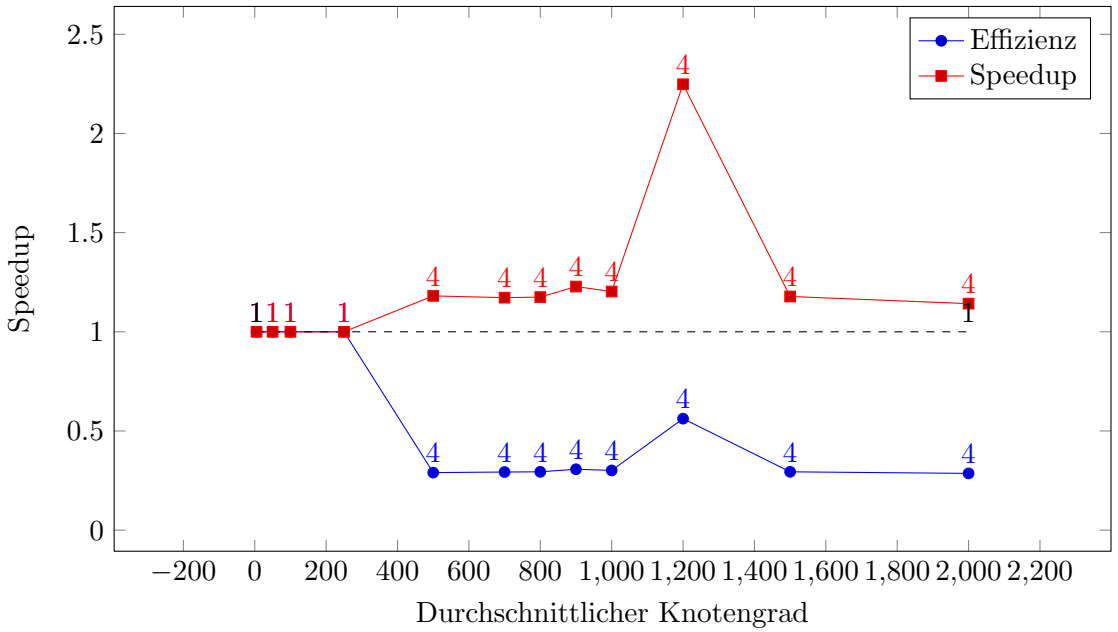
Dichte Die Knotenzahl und die Verteiung sind konstant, die Kantenanzahl variiert. Die Knotenzahl ist 100 000, der Knotengrad ist zwischen 1 und ∞

Verteilung Die Knotenzahl und Kantenanzahl ist konstant, der minimale und maximale Knotengrad variiert. Die Knotenzahl ist 100 000, der Knotengrad 750

Größe Der durchschnittliche Knotengrad und die Verteilung sind konstant, die Knotenzahl variiert. Der durchschnittliche Knotengrad ist 100, der minimale Knotengrad 1, der maximale 500.

6.2.1. Dichte

Die hier behandelten Graphen haben 100 000 Knoten bei einem minimalen Knotengrad von 1. Nach oben ist der Knotengrad nicht beschränkt. Als Dichte wird hier das Verhältnis von Kanten zu Knoten bezeichnet. Dieses Verhältnis ist gleichzeitig der durchschnittliche Knotengrad. Der Knotengrad ist auf nur 100 000 beschränkt, damit sehr dichte Graphen trotzdem noch in den Hauptspeicher passen. Die vollständigen Ergebnisse der Testreihe sind unter A angehängt.



Die Messergebnisse liefern kein eindeutiges Bild. Aus Abbildung 6.2.1 ist leicht ersichtlich, dass eine mindeste Graphgröße vorhanden sein muss, damit es sich überhaupt lohnt, parallel an einer Instanz des BFS zu arbeiten. Dabei scheint nicht nur die Anzahl der Knoten relevant zu sein, sondern auch die Anzahl der Kanten. Auffällig ist, dass der Speedup mit steigender Dichte nicht wächst. Der maximal erreichte Speedup liegt im Bereich zwischen 1.2 und 1.3, ausgenommen den einen Ausreißer. Der Ausreißer ist ohne weiter Analyse des Graphen nicht zu erklären. Es ist unwahrscheinlich, dass gerade bei einer bestimmten Graphdichte die Parallelisierung dermaßen viel besser funktioniert, als bei allen anderen Graphdichten. Wahrscheinlicher ist es, dass der Zufallsgenerator einen Graph generiert hat, der wenig Kanten zwischen Knoten hat, die auf unterschiedlichen Places liegen. Es kann auch sein, dass durch Zufall die Last auf den Places bei diesem Graph eher gleichmäßig verteilt ist.

In Abbildung 6.2 ist der Speedup von 4 beispielhaft gewählten Graphen aus der Testreihe über dem Grad der Parallelität aufgetragen. Es ist deutlich zu sehen, wie schlecht der 1D-Algorithmus mit dem durchschnittlichen Knotengrad von 5 performt. Auch ist zu sehen, wie ähnlich die Speedup bei 500 und 2000 verläuft, dass also scheinbar ab einer bestimmten Grenze, die bei den Tests bei ca. 500 lag, kein verbesserter Speedup mehr allein durch dichtere Graphen erreichbar ist. Im Kapitel 6.2.2 ist nachzulesen, dass die hier gewählten Grenzen des Knotengrads nicht ideal für die Parallelsierung sind. Ob der durchschnittliche Knotengrad einen stärkeren Einfluss hat, wenn der Graph gleichmäßiger ist, müssen weitere Tests zeigen.

6.2.2. Verteilung

Die Ergebnisse der hier besprochen Testreihe, finden sich allesamt in Anhang B. Bei der Testreihe wurde die Knotenzahl fest auf 100 000 festgelegt und der Kantengrad im Durchschnitt auf 750 eingestellt. Jeder Graph hat also auch die selbe absolute Kantenzahl. Diese Zahl wurde relativ hoch angesetzt, um genug Spielraum für die Variation der Verteilung zu haben. Ein Graph, dessen minimaler und maximaler Knotengrad weit vom durchschnittlichen Knotengrad entfernt ist, wird *ungleichmäßig* genannt. Das Gegenteil dazu wird als *gleichmäßig* bezeichnet.

Zunächst fällt auf, dass die seriellen Laufzeiten in der Praxis konstant sind, wie es die Theorie uns vorhersagt. Dadurch ist in dieser Testreihe der Speedup besonders gut vergleichbar.

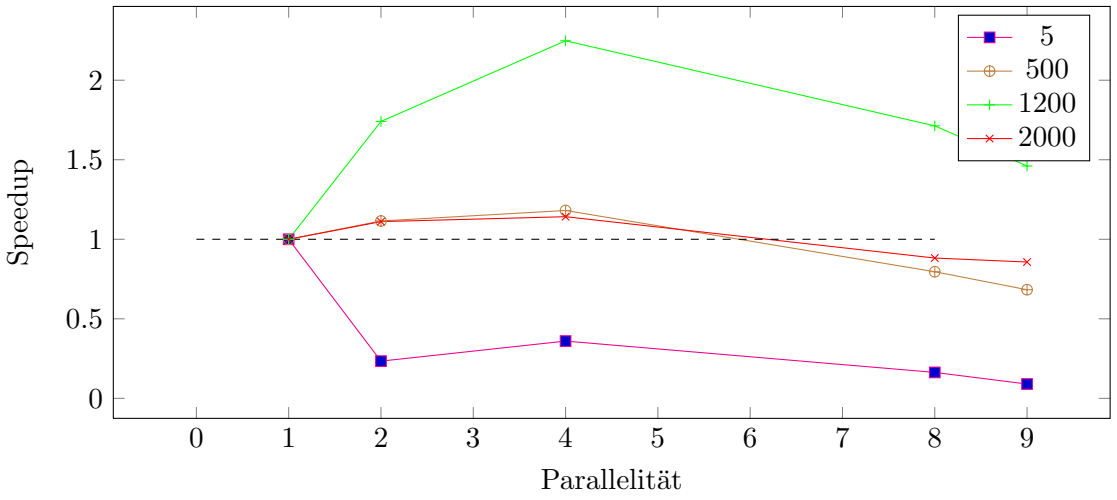


Abbildung 6.2.: Speedup des 1D-Algorithmus über Anzahl der Places. Die Dichte variiert. Als Referenzwert wurde jeweils der schnellste serielle Algorithmus genommen. Es wurden immer die schnellsten, gemessenen Werte verwendet.

In Abbildung 6.3 ist die oberste Kurve der Speedup des gleichmäßigsten Graphen und die untere Kurve die, des ungleichmäßigsten Graphen. Es ist deutlich ersichtlich, dass das Parallelisierungspotential eines Graphen stark von der Verteilung der Knotengrade abhängt. Um so gleichmäßiger diese verteilt sind, um so besser funktioniert die Parallelisierung. Ein mögliche Erklärung ist, dass bei einem gleichmäßig verteilten Graph die Wahrscheinlichkeit größer ist, dass die Rechenlast gleichmäßig auf alle Places verteilt wird. Bei ungleichmäßigen Graphen hingegen, ist es eher zufällig, auf welchem Place wie viele aktive Knoten liegen. Auf die Größe der Datenmengen, die zwischen den Places transportiert werden muss, dürfte die Verteilung keinen Einfluss haben.

6.2.3. Größe

Die Testreihe, um die Auswirkung der Größe eines Graphen auf die Algorithmen herauszufinden, orientiert sich an einem Socialgraph. Es wurde geschätz, dass jede Person durchschnittlich 100 Freunde hat, dass jeder mehr als einen Freund hat und keiner mehr als 500 Freunde hat. Das ergibt einen durchschnittlichen Knotengrad von 100, wobei der Knotengrad zwischen 1 und 500 liegt. Variiert wurde nun die Größe der Graphen, also die Anzahl an Knoten. Es handelt sich hier um einen relativ dünn besetzte Graphen, so dass selbst der 500 000 Knoten Graph noch unter 500 MB liegt. Es wurde versucht, die Graphgröße weiter auf bis zu 2 Millionen Knoten zu steigern, was allerdings nicht möglich war. Graphen mit vielen Knoten resultieren in großen Sendepuffern und wie sich herausstellte, ist X10 dem nicht gewachsen. Der Sendevorgang (der *at* Block) blockiert das Programm in den meisten Fällen für immer. Deswegen wurde hier die Knotenzahl auf 500 000 beschränkt. Es ist zu vermuten, dass diese Größe nicht ausreicht, um gute Ergebnisse bei der Parallelisierung zu erreichen.

Des weiteren wurde in dieser Testreihe der Modus mit 9 Places und der invasive Algorithmus weggelassen. Dass mit 9 Places auf 8 Kernen die Performance nicht gut sein würde, war bereits zu erwarten und bestätigte sich. Dass der invasive Algorithmus weggelassen wurde, liegt daran, dass er ebenso Probleme mit der Graphgröße hatte, was zwar durch Implementierungstricks umgangen werden könnte, aber allerdings den Zeitrahmen gesprengt hätte. Wieso der invasive Algorithmus nicht funktioniert, ist in der Bemerkung unten erläutert.

Bei jedem Graph dieser Testreihe war die serielle Version am schnellsten, was nach den vorangegangenen Ergebnissen auf keinen Fall zu erwarten war. Die serielle Version war

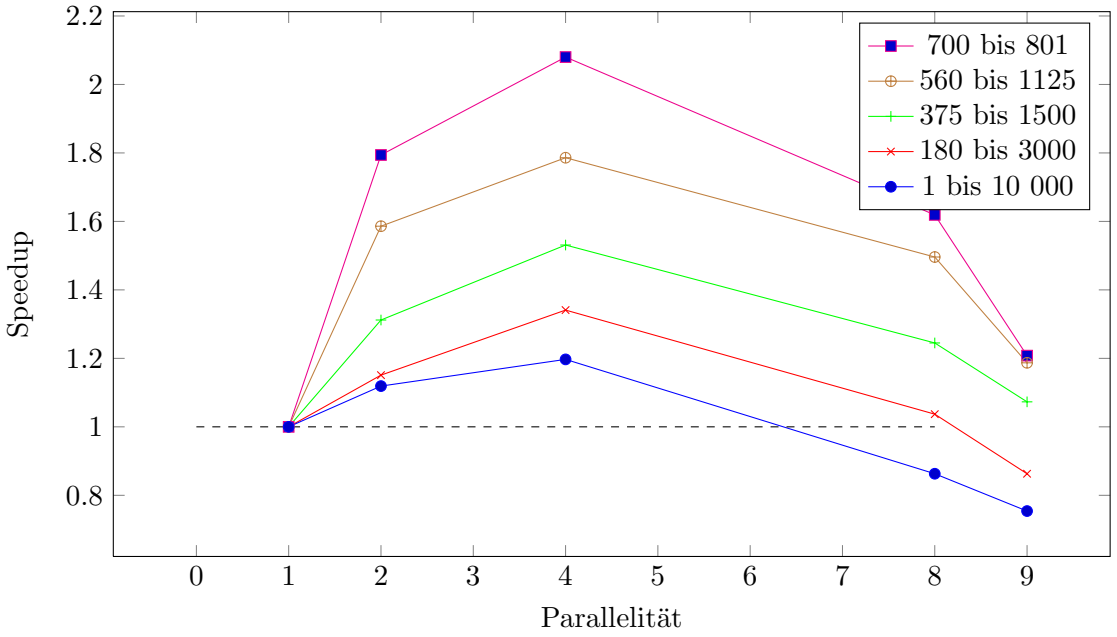


Abbildung 6.3.: Speedup des 1D-Algorithmus über Anzahl der Places. Die Verteilung variiert. Als Referenzwert wurde jeweils der schnellste serielle Algorithmus genommen. Pro Graph ist eine Kurve eingezeichnet. Es wurden immer die schnellsten, gemessenen Werte verwendet.

schneller als jeder andere Algorithmus, egal mit wie vielen Places. Im Kapitel 6.2.1, in dem die Dichte variierte, war der 1D-Algorithmus schon im Fall mit nur einem Place schneller, als der serielle Algorithmus. Da das hier nicht so ist, ist zu vermuten, dass der 1D Algorithmus für dichte Graphen ein wenig schneller zu sein scheint, während der serielle Algorithmus für dünn besetzte Graphen schneller ist.

Vergleicht man nur den 1D-Algorithmus mit sich selbst bei steigender Anzahl an Places, zeichnet sich das Bild, dass 2 Places das Maximum an Parallelität zu sein scheint, dass sich noch lohnt. In Abbildung 6.4 ist sehr schön zu sehen, dass der Algorithmus mit 2 Places immer schneller war, als der Algorithmus mit 4 Places. Bei 8 Places war der Speedup immer kleiner als 1, was langsamere Laufzeit als die Referenzzeit bedeutet. Wie zu erwarten steigt tendenziell mit der Anzahl der Knoten das Parallelisierungspotential, wobei es durch den dünn besetzte Graph einigermaßen beschränkt zu sein scheint. Wäre dem nicht so müsste der Algorithmus auf 4 Places gegenüber den 2 Places aufholen, bei steigender Knotenzahl.

Für den 2D Algorithmus gilt, dass nie ein Speedup größer als 1 erreicht wurde. Die hier verwendeten Graphen sind dafür zu klein.

Bemerkung: Beim Parsen der Graphdatei gehen alle Implementierungen so vor, dass sie Zeile für Zeile in die Datenstruktur eingepflegt werden. Dabei steht schon vor dem Parsen fest, welches Datum später auf welchem Place liegen muss. Die Informationshäppchen, die von Place zu Place übertragen werden müssen, sind relativ klein. Im invasiven Fall ergibt es wenig Sinn, schon ein *invade* aufzurufen, bevor überhaupt der Graph eingelesen wurde, bevor also die Größe des Problems bekannt ist. Aus diesem Grund wird der Graph erst lokal auf den ersten Place eingelesen, dann der Constraint zusammengebaut und erst wenn der Claim bekannt ist, auf die involvierten Places verteilt. Die hier zu übertragenen Daten sind relativ groß, womit X10 Probleme zu haben scheint. Das Problem könnte umgangen werden, indem die Datenstruktur in kleinere Häppchen aufgeteilt wird. Das ist aber höchstens ein Workaround um ein Problem, das bei X10 liegt.

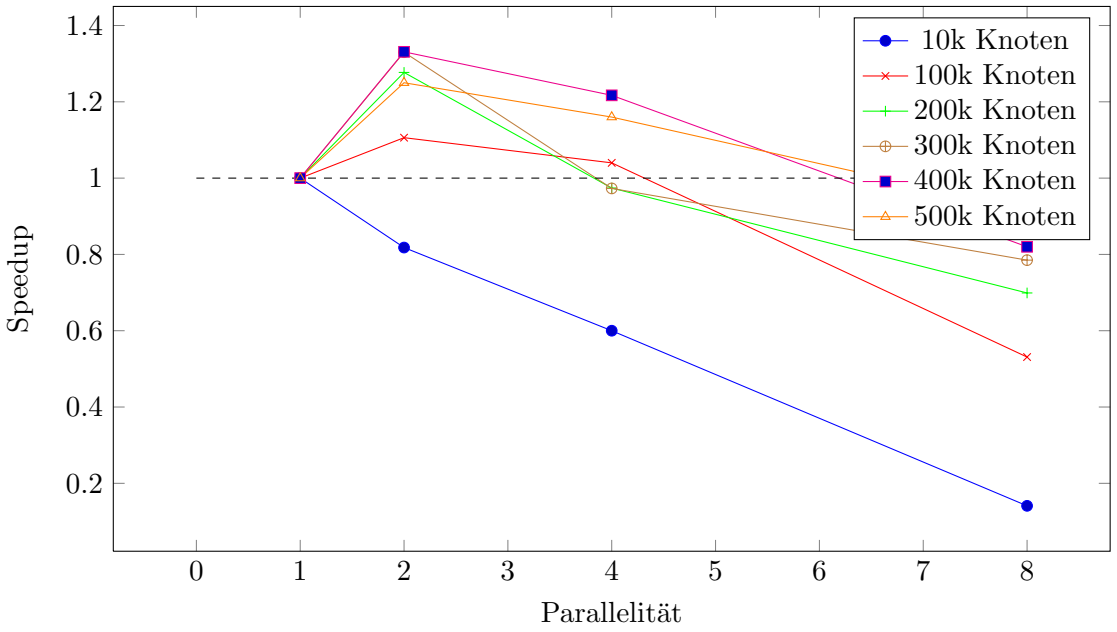


Abbildung 6.4.: Speedup des 1D-Algorithmus über Anzahl der Places. Variiert wurde die Größe des Graphen. Als Referenzwert wurde jeweils der 1D-Algorithmus mit einem Place genommen. Eine Kurve pro Graph.

6.3. Die 2D Breitensuche

Die 2D Breitensuche ist in jedem Testfall der mit Abstand langsamste Algorithmus gewesen, was so nicht zu erwarten war. Der implementierte Algorithmus ist zwar zunächst deutlich komplexer, als die anderen beiden, doch gibt es zwei Vorteile, die diesen Algorithmus gewiss studierenswert machen. Erstens sind die Gruppen von Places, die untereinander kommunizieren deutlich kleiner. In der ersten Kommunikationsphase sendet jeder Place seine Daten an eine ganze Spalte, und empfängt dafür von einer ganzen Zeile die Daten für den nächsten Schritt. Das sind $2 * \sqrt(p)$ Places, insgesamt, mit denen jeder Place kommunizieren muss, also $O(p * 2 + \sqrt(p) * \frac{1}{2}) = O(p * \sqrt(p)) = O(p^{\frac{3}{2}})$ Kommunikationsvorgänge im System. In der zweiten Kommunikationsphase muss jeweils nur eine Zeile miteinander kommunizieren. Das sind zusätzlich $O(\sqrt(p) * p * \frac{1}{2})$ Kommunikationsvorgänge. Es sind also im O-Kalkül pro Iteration $O(p^{\frac{3}{2}})$. Im Vergleich dazu kommuniziert jeder Place im 1D Algorithmus mit allen anderen p-1 Places, also $O(p^2)$ Kommunikationsvorgänge. Der zweite Vorteil der Implementierung wie in [BM11] ist die Möglichkeit, alle Kommunikation und Synchronisation auf drei MPI Operationen abzubilden. Die auf entsprechender Hardware vergleichsweise schnell sind.

Beide dieser Vorteile sind bei der X10 Implementierung, die auf nur einem CPU läuft, hinfällig. Zunächst wurden die MPI Operationen in X10 „nachprogrammiert“. Um das zu tun, muss *at* verwendet werden, dass einen für diesen Zweck unnützen Rückkanal bereithält, der zusätzliche Latenz in jede Kommunikation bringt. Weiterhin gibt es bei der Verwendung einer hochperformanten MPI Hardware den Vorteil, dass ein Datum nur einmal gesendet werden muss, wenn es an mehrere Empfänger gehen soll. Das ist in X10-Syntax nicht möglich. Die zusätzliche Kommunikation ist also teurer, als sie sein müsste. Der zweite Vorteil, eben dass weniger Kommunikationsvorgänge im System sind, ist aus zwei Gründen hier nicht ausschlaggebend. Zum einen sind dermaßen wenig Places im Spiel, dass es kaum einen Unterschied zwischen $O(p^{\frac{3}{2}})$ und $O(p^2)$ gibt, zum anderen haben erste Tests gezeigt, dass die Kommunikationsphasen etwa 4 bis 5 Mal so schnell sind wie die Rechenphasen, also die Kommunikation nicht so ausschlaggebend ist, wie das zu erwarten war. Das wiederum

ist auf die Testumgebung ohne wirklich getrennte Places zurückzuführen.

Wie in [BM11] ist also auch in X10 die 2D Implementierung eher die schlechtere, wobei in den Dimensionen, in denen in dieser Arbeit gedacht wird, die Unterschiede gravierender sind. Ob in größeren Testumgebungen die Ergebnisse anders ausfallen, muss in einer weiteren Arbeit untersucht werden.

6.4. Invasive Breitensuche

Im Rahmen dieser Arbeit wurden Ansätze herausgearbeitet, um die Breitensuche an das invasive Rechnen im Allgemeinen und an das Framework invadeX10 im Speziellen, anzupassen. Die Ergebnisse fielen dabei zwar besser aus, als erwartet, doch sind die Laufzeiten immernoch erheblich länger, als die der 1D Breitensuche. Die benötigte Zeit zur Lösung einer BFS-Instanz zu vergleichen wäre unfair und nicht zielführend, weswegen darauf gänzlich verzichtet wird. Die Intention des invasiven Rechnens war es nie, eine einzelne Instanz möglichst schnell zu lösen. Um einen fairen Vergleich durchzuführen, müsste eine Menge von verschiedenen Jobs und eine Zielhardware definiert werden. Daraufhin muss verglichen werden, wie lange es mit verschiedenen Strategien braucht, bis alle Jobs erledigt wurden. Die Strategien sind zum Beispiel, alle gleichzeitig starten, ein Job nach dem anderen starten oder eben invasives Vorgehen. In Mangel der invasiven Hardware und ausgereiften Softwareinstrumenten um diese zu simulieren, musste darauf in dieser Arbeit verzichtet werden. Die Ergebnisse sind aber insofern nicht schlecht, als dass sie nur 3 - 5 mal so langsam sind, wie die schnellsten Laufzeiten. Außerdem profiert auch die invasive Breitensuche, die sehr viel overhead hat, von den Parallelisierung auf mehrere Places.

6.5. Wikipedia und die Philosophie

Der bekannte Webcomic xkcd stellt die Behauptung auf, wenn man in der Wikipedia wiederholt auf den ersten Link klickt (der nicht in Klammern steht), gelangt man irgendwann zu dem Arikel über Philosphie [xkc]. Im Rahmen dieser Arbeit wurde die deutsche Wikipedia heruntergeladen, zu einem Graph verarbeitet und mittels Breitensuche herausgefunden,

1. dass diese Behauptung für ca. 42% der Artikel wahr ist,
2. dass man durchschnittlich 5,52 mal klicken muss, um zum Artikel „Philosphie“ zu gelangen, falls das denn möglich ist,
3. dass es einen Zykel gibt: Philosphie → Wissenschaft → Wissen → Erkenntnistheorie → Philosphie. Punkt 1 gilt also für alle vier Wikipediaartikel
4. und dass die Wissenschaft noch „zentraler“ in der Wikipedia ist. Die durchschnittliche Klickanzahl ist hier bei nur 4,49.

Den eigentlichen Zweck des Graphs, nämlich als Testdatum zu fungieren, konnte er leider nicht erfüllen, da er einfach zu klein ist, um sich für die Parallelisierung zu eignen.

7. Fazit und zukünftige Arbeit

Im Rahmen dieser Arbeit konnten einige Ideen und Ansätze nicht umgesetzt werden, die womöglich in zukünftiger Arbeit anzugehen sind.

Um die bestehende Arbeit besser zu testen und um die Vor- und Nachteile besser zu verstehen, sollte der existierende Code auf einem echten Rechencluster getestet werden. Die Bedingungen unterscheiden sich stark von der parallelen Ausführung auf nur einem Prozessor. Zum einen können mit einem Rechencluster, das echten verteilten Speicher hat, wesentlich größere Graphen getestet werden. Größere Graphen bedeuten immer auch größeres Parallelisierungspotential. Zum anderen ist X10 laut den Entwicklern dafür gemacht, effiziente Programme für große verteilte Rechensysteme zu programmieren. Aus diesen zwei Gründen ist zu erwarten, dass der in dieser Arbeit erreichte Speedup weit unter dem Potential der Breitensuche geblieben ist.

Zudem muss die Breitensuche natürlich auf der Invasiven Hardware getestet werden, die im Rahmen des invasIC Projekts entsteht.

Auch implementierungstechnisch gibt weitere Variationsmöglichkeiten, deren Auswirkung auf Geschwindigkeit und Beschleunigung getestet werden kann. Die einzelnen Processing Elements können auch autonomer Implementiert werden, als sie es im Moment sind. Wenn jede PE ein eigenes Intervall des Graphen bekommt und darauf autonom rechnet, würde die Synchronisation zwischen den einzelnen PEs komplett wegfallen. In der aktuellen Codebasis wäre das eine Barriere weniger. Außerdem sparte man sich die Funktionsaufrufe und die Arithmetik, um die Liste der aktiven Knoten aufzuteilen. Der Nachteil dieser Implementierung wäre, dass von einem Place zu einem anderen pro Iteration mehrere Kommunikationen stattfinden, falls mehrere PEs zum gleichen Place senden müssen.

In der invasiven Implementierung werden im Moment keine Processing Elements abgegeben oder neu beantragt, sobald der Algorithmus einmal gestartet ist. Die Masteractivity weiß aber nach jeder Iteration, wie lang die Liste der aktiven Knoten auf jedem Place ist. Mit dieser Information könnte sie Rechenleistung freigeben oder neu beantragen. Eine entsprechende Implementierung würde die Möglichkeit des Framework nutzen, eine einmal abgegebene PE wieder zurück zu verlangen. Diese Funktion würde das in Kapitel 4.1.2 beschriebenen Problem lösen, dass anzunehmender Weise zu einem späteren Zeitpunkt genau die abgegebene Rechenleistung wieder gebraucht wird. Diese Funktionalität ist aber noch nicht vorhanden und konnte somit nicht getestet werden.

Auch nicht getestet wurde, wie es sich ein Programm verhält, wenn die Datenhaltung auf eine veränderte Situation der Rechenleitung reagiert. Dieser Ansatz geht im Gegensatz

zu dem gerade genannten davon aus, dass Rechenleistung genutzt werden soll, egal auf welchem Place sie liegt. Wenn also Rechenleistung abgegeben wurde oder neue hinzukam, müssen die Graphdaten dynamisch entsprechend an den richtigen Ort kopiert werden. Womöglich können die Daten so verteilt werden, dass ab einem gewissen Zeitpunkt für jeden Knoten mindestens zwei Places verantwortlich sind und dadurch auf weitere Änderungen sehr schnell reagiert werden kann.

Fazit: In dieser Arbeit wurden zwei Strategien, die Breitensuche zu Parallelisieren, auf die moderne Programmiersprache X10 übertragen und getestet. Dazu wurde in mehreren Iterationen durch Tests und Veränderungen am Code, der Algorithmus so weit wie möglich optimiert und an die Parallelitätskonstrukte von X10 angepasst. Eine der beiden Strategien, die 1D-Breitensuche wurde zusätzliche für das invasive Framework invadeX10 angepasst. Dabei wurde besonders auf die dabei neu auftretenden Probleme und deren Lösungen eingegangen, sowie erwähnt, an welchen Stellen noch zusätzliche Arbeit notwendig ist.

Die Messungen ergaben, dass sich die 1D-Dekomposition im kleinen Maßstab wesentlich besser eignet, als die 2D-Dekomposition, deren zusätzliche Komplexität sich nicht auszahlt. Ob überhaupt eine Parallelisierung sinnvoll ist, hing dabei stark von dem gewählten Graph ab. Es scheint, als bräuchte man eine mindeste Anzahl von Knoten und einen möglichst gleichverteilten Knotengrad, um gute Ergebnisse zu erhalten. Die verwendete Hardware in Verbindung mit den verwendete Messmethoden lassen eine Übertragbarkeit der Ergebnisse auf Rechencluster aber nicht zu. Ebenso sind die Ergebnisse für Shared-Memory Systeme kaum aussagekräftig. Die Algorithmen sind nur ein Zwischenschritt auf dem Weg zum invasiven Rechnen oder den Manycore-Systemen der Zukunft.

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 evens 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.

[gra09] *graph-generator*, 2009. https://code.google.com/p/graph-generator/.

[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.

[Meh08] Peter Mehlhorn, Kurt ; Sanders (Herausgeber): *Algorithms and Data Structures : The Basic Toolbox*. Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 2008, ISBN 978-3-540-77978-0.

[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*, Juli 2012. http://x10-lang.org/home/faq-list.html.

[xkc] xkcd: *Extended Mind*. https://xkcd.com/903/.

[Zwi12] Andreas Zwinkau: *Resource Awareness for Efficiency in High-Level Programming Languages*, 2012. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000028712.

Anhang

A. Messwerte - Dichte

Alle Messwerte sind in Millisekunden (ms) angegeben. Alle Graphen haben 100 000 Knoten. Der Name avX steht für einen Graphen mit 100 000 Knoten und einem durchschnittlichen Ausgangsgrad von X. Alle Algorithmen verwenden Arraylisten als Datenstruktur. Der invasive Algorithmus wurde mit einem PE pro Place ausgeführt. Zwischen den Iterationen wurde nichts an dem Claim geändert. Der Knotengrad liegt bei allen Algorithmen zwischen 1 und ∞.

av5					
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	33				
1D	62	124	78	204	324
2D	85	289	570	302	560
Invasive	83	87	110	164	269
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	29				
1D	63	247	75	178	345
2D	88	509	524	266	639
Invasive	80	88	80	163	247
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	30				
1D	63	211	79	179	446
2D	86	98	590	318	446
Invasive	82	87	85	157	243
Fastest					
Seriell	29				
1D	62	124	75	178	324
2D	85	98	524	266	446
Invasive	80	87	80	157	243
Average					
Seriell	30.67				
1D	62.67	194	77.33	187	371.67
2D	86.33	298.67	561.33	295.33	548.33
Invasive	81.67	87.33	91.67	161.33	253

av50					
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	57				
1D	88	172	81	148	223
2D	155	294	342	424	395
Invasive	160	172	129	201	251
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	56				
1D	88	77	92	156	291
2D	155	175	181	427	571
Invasive	160	172	140	200	256
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	57				
1D	88	77	80	148	315
2D	156	173	181	279	381
Invasive	154	172	143	211	253
Fastest					
Seriell	56				
1D	88	77	80	148	223
2D	155	173	181	279	381
Invasive	154	172	129	200	251
Average					
Seriell	56.67				
1D	88	108.67	84.33	150.67	276.33
2D	155.33	214	234.67	376.67	449
Invasive	158	172	137.33	204	253.33

av100					
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	79				
1D	118	95	108	186	220
2D	234	233	327	343	541
Invasive	225	252	208	266	315
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	82				
1D	117	124	98	184	271
2D	234	231	385	333	486
Invasive	232	244	207	276	295
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	79				
1D	114	107	96	180	228
2D	235	240	662	381	456
Invasive	231	207	199	279	290
Fastest					
Seriell	79				
1D	114	95	96	180	220
2D	234	231	327	333	456
Invasive	225	207	199	266	290
Average					
Seriell	80				
1D	116.33	108.67	100.67	183.33	239.67
2D	234.33	234.67	458	352.33	494.33
Invasive	229.33	234.33	204.67	273.67	300

av250					
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	151				
1D	166	170	170	234	314
2D	444	417	543	640	722
Invasive	465	471	389	480	483
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	157				
1D	168	171	158	253	304
2D	444	554	503	613	710
Invasive	466	498	403	480	514
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	158				
1D	164	152	163	238	318
2D	444	434	486	622	790
Invasive	467	501	392	492	509
Fastest					
Seriell	151				
1D	164	152	158	234	304
2D	444	417	486	613	710
Invasive	465	471	389	480	483
Average					
Seriell	155.33				
1D	166	164.33	163.67	241.67	312
2D	444	468.33	510.67	625	740.67
Invasive	466	490	394.67	484	502

av500					
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	275				
1D	263	235	226	351	382
2D	808	871	810	995	1089
Invasive	845	909	683	840	863
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	265				
1D	261	235	224	337	396
2D	809	773	912	1064	1113
Invasive	844	874	709	808	818
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	269				
1D	262	234	221	328	438
2D	808	844	895	1047	1121
Invasive	845	795	740	818	814
Fastest					
Seriell	265				
1D	261	234	221	328	382
2D	808	773	810	995	1089
Invasive	844	795	683	808	814
Average					
Seriell	269.67				
1D	262	234.67	223.67	338.67	405.33
2D	808.33	829.33	872.33	1035.33	1107.67
Invasive	844.67	859.33	710.67	822	831.67

av700					
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	355				
1D	345	319	301	404	509
2D	1098	1078	1155	1344	1375
Invasive	1139	1193	968	1093	1144
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	361				
1D	340	326	290	404	447
2D	1096	1052	1172	1418	1414
Invasive	1149	1191	955	1077	1131
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	360				
1D	340	345	303	409	468
2D	1098	1124	1126	1286	1518
Invasive	1146	1196	949	1076	1128
Fastest					
Seriell	355				
1D	340	319	290	404	447
2D	1096	1052	1126	1286	1375
Invasive	1139	1191	949	1076	1128
Average					
Seriell	358.67				
1D	341.67	330	298	405.67	474.67
2D	1097.33	1084.67	1151	1349.33	1435.67
Invasive	1144.67	1193.33	957.33	1082	1134.33

av800					
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	410				
1D	390	353	326	460	524
2D	1241	1190	1337	1535	1718
Invasive	1288	1351	1093	1225	1263
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	400				
1D	379	352	348	454	526
2D	1242	1224	1233	1469	1644
Invasive	1333	1350	1190	1254	1214
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	401				
1D	376	352	320	456	529
2D	1239	1272	1313	1554	1548
Invasive	1306	1349	1108	1245	1238
Fastest					
Seriell	400				
1D	376	352	320	454	524
2D	1239	1190	1233	1469	1548
Invasive	1288	1349	1093	1225	1214
Average					
Seriell	403.67				
1D	381.67	352.33	331.33	456.67	526.33
2D	1240.67	1228.67	1294.33	1519.33	1636.67
Invasive	1309	1350	1130.33	1241.33	1238.33

av900					
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	454				
1D	415	390	374	489	524
2D	1384	1297	1397	1697	1718
Invasive	1450	1507	1306	1422	1403
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	452				
1D	422	390	338	499	548
2D	1383	1335	1568	1707	1745
Invasive	1446	1508	1247	1351	1385
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	450				
1D	422	392	357	493	544
2D	1384	1313	1466	1701	1741
Invasive	1448	1506	1250	1385	1370
Fastest					
Seriell	450				
1D	415	390	338	489	524
2D	1383	1297	1397	1697	1718
Invasive	1446	1506	1247	1351	1370
Average					
Seriell	452				
1D	419.67	390.67	356.33	493.67	538.67
2D	1383.67	1315	1477	1701.67	1734.67
Invasive	1448	1507	1267.67	1386	1386

av1000					
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	498				
1D	453	402	375	514	569
2D	1526	1516	1487	1782	1944
Invasive	1611	1663	1348	1476	1504
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	502				
1D	452	472	390	525	589
2D	1533	1476	1581	1787	1814
Invasive	1612	1668	1332	1517	1480
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	498				
1D	451	408	396	507	611
2D	1532	1475	1519	1827	1877
Invasive	1612	1665	1305	1580	1507
Fastest					
Seriell	498				
1D	451	402	375	507	569
2D	1526	1475	1487	1782	1814
Invasive	1611	1663	1305	1476	1480
Average					
Seriell	499.33				
1D	452	427.33	387	515.33	589.67
2D	1530.33	1489	1529	1798.67	1878.33
Invasive	1611.67	1665.33	1328.33	1524.33	1497

av1200					
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	575				
1D	527	302	234	307	377
2D	1756	1186	835	920	954
Invasive	1846	1918	739	820	817
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	576				
1D	526	302	240	314	380
2D	1754	1159	826	929	1009
Invasive	1847	1916	740	812	857
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	578				
1D	527	303	236	321	360
2D	1755	1183	828	946	976
Invasive	1845	1915	915	781	833
Fastest					
Seriell	575				
1D	526	302	234	307	360
2D	1754	1159	826	920	954
Invasive	1845	1915	739	781	817
Average					
Seriell	576.33				
1D	526.67	302.33	236.67	314	372.33
2D	1755	1176	829.67	931.67	979.67
Invasive	1846	1916.33	798	804.33	835.67

av1500					
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	727				
1D	648	722	546	742	782
2D	2253	2056	2462	2631	2694
Invasive	2349	2448	1935	2242	2137
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	728				
1D	643	746	551	711	768
2D	2247	2161	2235	2613	2680
Invasive	2408	2452	1965	2269	2207
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	736				
1D	644	602	562	737	746
2D	2251	2058	2470	2660	2749
Invasive	2358	2443	1958	2212	2162
Fastest					
Seriell	727				
1D	643	602	546	711	746
2D	2247	2056	2235	2613	2680
Invasive	2349	2443	1935	2212	2137
Average					
Seriell	730.33				
1D	645	690	553	730	765.33
2D	2250.33	2091.67	2389	2634.67	2707.67
Invasive	2371.67	2447.67	1952.67	2241	2168.67

av2000					
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	947				
1D	872	828	748	939	966
2D	2969	2741	3309	3521	3477
Invasive	3315	3300	2654	27302	2783
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	1236				
1D	828	746	725	960	992
2D	3042	2726	2981	3717	3560
Invasive	3224	3255	2601	26337	2989
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	1012				
1D	836	745	748	946	1229
2D	2978	2781	3024	3626	3546
Invasive	3222	3250	2738	15083	6068
Fastest					
Seriell	947				
1D	828	745	725	939	966
2D	2969	2726	2981	3521	3477
Invasive	3222	3250	2601	15083	2783
Average					
Seriell	1065				
1D	845.33	773	740.33	948.33	1062.33
2D	2996.33	2749.33	3104.67	3621.33	3527.67
Invasive	3253.67	3268.33	2664.33	22907.33	3946.67

B. Messwerte - Verteilung

Alle Messwerte sind in Millisekunden (ms) angegeben. Jeder Graph hat 100 000 Knoten und einen durchschnittlichen Knotengrad von 750. Der invasive Algorithmus wurde mit einem PE pro Place ausgeführt. Zwischen den Iterationen wurde nichts an dem Claim geändert.

1 bis 10 000					
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	377				
1D	358	322	326	415	475
2D	1212	1111	1118	1522	1499
Invasive	1221	1267	992	1113	1120
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	377				
1D	359	320	313	426	494
2D	1210	1104	1154	1412	1499
Invasive	1214	1274	1016	1164	1132
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	376				
1D	359	325	299	437	495
2D	1212	1098	1362	1455	1558
Invasive	1230	1270	1007	1119	1166
Fastest					
Seriell	376				
1D	358	320	299	415	475
2D	1210	1098	1118	1412	1499
Invasive	1214	1267	992	1113	1120
Average					
Seriell	376.67				
1D	358.67	322.33	312.67	426	488
2D	1211.33	1104.33	1211.33	1463	1518.67
Invasive	1221.67	1270.33	1005	1132	1139.33

180 bis 3000					
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	381				
1D	366	414	426	405	424
2D	1196	1062	999	1126	1240
Invasive	1235	1289	888	918	929
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	387				
1D	370	318	282	385	442
2D	1194	1066	1027	1140	1170
Invasive	1215	1287	932	963	945
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	379				
1D	367	318	273	353	464
2D	1194	1045	1205	1175	1227
Invasive	1223	1287	856	926	977
Fastest					
Seriell	379				
1D	366	318	273	353	424
2D	1194	1045	999	1126	1170
Invasive	1215	1287	856	918	929
Average					
Seriell	382.33				
1D	367.67	350	327	381	443.33
2D	1194.67	1057.67	1077	1147	1212.33
Invasive	1224.33	1287.67	892	935.67	950.33

375 bis 1500					
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	380				
1D	367	282	243	325	341
2D	1180	1000	772	854	934
Invasive	1221	1266	712	718	689
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	379				
1D	366	313	266	306	351
2D	1182	909	778	807	964
Invasive	1229	1270	697	696	676
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	379				
1D	366	279	239	294	348
2D	1182	879	753	936	978
Invasive	1220	1266	681	698	698
Fastest					
Seriell	379				
1D	366	279	239	294	341
2D	1180	879	753	807	934
Invasive	1220	1266	681	696	676
Average					
Seriell	379.33				
1D	366.33	291.33	249.33	308.33	346.67
2D	1181.33	929.33	767.67	865.67	958.67
Invasive	1223.33	1267.33	696.67	704	687.67

560 bis 1125					
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	379				
1D	368	321	368	246	323
2D	1179	806	615	724	743
Invasive	1234	791	556	582	605
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	378				
1D	370	232	206	253	310
2D	1186	806	780	765	778
Invasive	1227	1275	575	609	593
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	381				
1D	368	235	215	304	338
2D	1179	817	1128	725	789
Invasive	1224	1266	582	611	619
Fastest					
Seriell	378				
1D	368	232	206	246	310
2D	1179	806	615	724	743
Invasive	1224	791	556	582	593
Average					
Seriell	379.33				
1D	368.67	262.67	263	267.67	323.67
2D	1181.33	809.67	841	738	770
Invasive	1228.33	1110.67	571	600.67	605.67

700 bis 801					
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	382				
1D	366	204	176	226	320
2D	1225	842	584	563	768
Invasive	1232	1267	526	543	552
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	381				
1D	366	205	346	229	303
2D	1225	831	647	648	728
Invasive	1229	1290	500	528	559
	1 Place	2 Places	4 Places	8 Places	9 Places
Seriell	383				
1D	366	340	184	240	353
2D	1189	732	534	587	653
Invasive	1228	1265	505	516	543
Fastest					
Seriell	381				
1D	366	204	176	226	303
2D	1189	732	534	563	653
Invasive	1228	1265	500	516	543
Average					
Seriell	382				
1D	366	249.67	235.33	231.67	325.33
2D	1213	801.67	588.33	599.33	716.33
Invasive	1229.67	1274	510.33	529	551.33

C. Messwerte - Größe

Alle Messwerte sind in Millisekunden (ms) angegeben. Jeder Graph hat einen durchschnittlichen Knotengrad von 100, wobei der Knotengrad jedes Knotens zwischen 1 und 500 liegt. Es wurde auf den invasiven Algorithmus verzichtet, da die Hardware größeren Graphen mit diesem Algorithmus nicht gewachsen ist. Auf die Testläuft mit 9 Places wurde ebenso verzichtet.

10k				
	1 Place	2 Places	4 Places	8 Places
Seriell	7			
1D	9	12	17	64
2D	26	27	35	85
	1 Place	2 Places	4 Places	8 Places
Seriell	6			
1D	13	11	15	66
2D	26	27	38	109
	1 Place	2 Places	4 Places	8 Places
Seriell	6			
1D	14	12	18	66
2D	27	27	36	87
Fastest				
Seriell	6			
1D	9	11	15	64
2D	26	27	35	85
Average				
Seriell	6.33			
1D	12	11.67	16.67	65.33
2D	26.33	27	36.33	93.67

100k				
	1 Place	2 Places	4 Places	8 Places
Seriell	79			
1D	104	94	100	196
2D	258	311	384	459
	1 Place	2 Places	4 Places	8 Places
Seriell	80			
1D	107	104	106	196
2D	257	276	342	397
	1 Place	2 Places	4 Places	8 Places
Seriell	80			
1D	107	103	110	205
2D	257	264	472	450
Fastest				
Seriell	79			
1D	104	94	100	196
2D	257	264	342	397
Average				
Seriell	79.67			
1D	106	100.33	105.33	199
2D	257.33	283.67	399.33	435.33

200k				
	1 Place	2 Places	4 Places	8 Places
Seriell	178			
1D	256	199	271	366
2D	543	562	680	954
	1 Place	2 Places	4 Places	8 Places
Seriell	172			
1D	249	195	379	356
2D	528	580	683	893
	1 Place	2 Places	4 Places	8 Places
Seriell	176			
1D	252	197	225	371
2D	536	528	691	858
Fastest				
Seriell	172			
1D	249	195	225	356
2D	528	528	680	858
Average				
Seriell	175.33			
1D	252.33	197	291.67	364.33
2D	535.67	556.67	684.67	901.67

300k				
	1 Place	2 Places	4 Places	8 Places
Seriell	278			
1D	391	318	756	498
2D	810	903	972	1194
	1 Place	2 Places	4 Places	8 Places
Seriell	269			
1D	400	294	549	499
2D	811	981	1034	1043
	1 Place	2 Places	4 Places	8 Places
Seriell	276			
1D	399	304	402	502
2D	818	1519	1111	1208
Fastest				
Seriell	269			
1D	391	294	402	498
2D	810	903	972	1043
Average				
Seriell	274.33			
1D	396.67	305.33	569	499.67
2D	813	1134.33	1039	1148.33

400k				
	1 Place	2 Places	4 Places	8 Places
Seriell	393			
1D	557	433	493	646
2D	1112	1133	1232	2053
	1 Place	2 Places	4 Places	8 Places
Seriell	377			
1D	558	476	481	733
2D	1119	1252	1525	1626
	1 Place	2 Places	4 Places	8 Places
Seriell	375			
1D	543	408	446	622
2D	1122	1338	1338	1854
Fastest				
Seriell	375			
1D	543	408	446	622
2D	1112	1133	1232	1626
Average				
Seriell	381.67			
1D	552.67	439	473.33	667
2D	1117.67	1241	1365	1844.33

500k				
	1 Place	2 Places	4 Places	8 Places
Seriell	501			
1D	747	551	594	674
2D	1443	1435	1579	1476
	1 Place	2 Places	4 Places	8 Places
Seriell	501			
1D	715	741	776	643
2D	1461	1437	1524	1496
	1 Place	2 Places	4 Places	8 Places
Seriell	518			
1D	689	600	712	678
2D	1368	1555	1467	1888
Fastest				
Seriell	501			
1D	689	551	594	643
2D	1368	1435	1467	1476
Average				
Seriell	506.67			
1D	717	630.67	694	665
2D	1424	1475.67	1523.33	1620