Parallele Algorithmen

Michael Ikkert
Tim Kieritz
Prof. Dr. Peter Sanders

Stand: 16. Oktober 2009

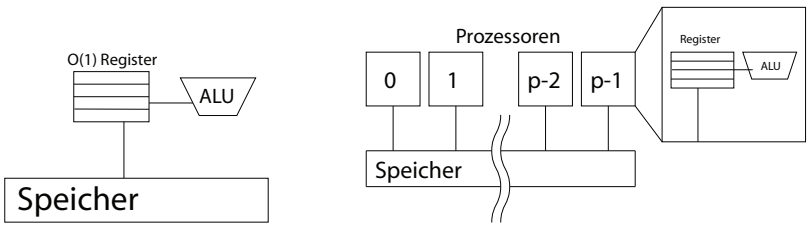
Inhaltsverzeichnis

1 Einführung

1.1 Maschinenmodelle

1.1.1 PRAM

Das PRAM-Modell bietet sich für die Untersuchung von parallelen Algorithmen in erster Linie wegen der Einfachheit der Darstellung an. Es basiert auf der Random Access Machine, die nur aus einem auf einem großen Speicher operierenden Prozessor mit konstanter Anzahl Registern besteht. Das PRAM-Modell parallelisiert dieses einfache Modell durch weitere Prozessoren, mit eigenen Registern, die synchron auf dem Speicher operieren. Häufig wird dann angenommen, es wäre zunächst nur ein Prozessor aktiv, es könnten jedoch beliebig viele Prozessoren nach Bedarf "aufgeweckt" werden. Im Folgenden wollen wir uns aber darauf beschränken, dass die Anzahl der benötigten Prozessoren im Voraus genannt wird und auch nicht mehr zur Verfügung stehen, diese dafür von Anfang an aktiv sind. Entsprechend wird auch festgelegt, dass jedes dieser PE (Prozessorelemente) sowohl des-



(a) RAM (b) PRAM

Abbildung 1.1: Modelle im Vergleich

sen Index kennt, als auch die Anzahl der verfügbaren Prozessoren. So einfach das Modell auch ist, wirft doch die gemeinsame Speicherverwaltung sofort die Frage nach Zugriffskonflikten auf einzelnen Speicherstellen auf. Abhängig davon, wie mit diesen Konflikten umgegangen wird, werden die PRAM-Modelle folgendermaßen unterteilt:

- **EREW** (Exclusive Read Exclusive Write) PRAM: Gleichzeitige Zugriffe sind verboten.
- **CREW** (Concurrent Read Exclusive Write) PRAM: Nur gleichzeitiges Lesen ist erlaubt, gleichzeitiges Schreiben nicht.
- **CRCW** (Concurrent Read Concurrent Write) PRAM: Hier wird nun beides erlaubt, was Maßnahmen bei Schreibkonflikten erfordert. Dafür gibt eine weitere Unterteilung wie folgt:
  - **common**: Eine naheliegende Erweiterung des CREW PRAM, die mehreren Prozessoren erlaubt das **gleiche Datum** auf einmal zu schreiben. Sind sie sich jedoch nicht einig, ist das weiterhin eine unzulässige Aktion.
  - **arbitary**: Hier sind auch Fälle abgedeckt, in denen unterschiedliche Werte geschrieben werden sollen, die Auswahl des schreibenden Prozessors erfolgt jedoch **zufällig**
  - **priority**: Oft ist es von Vorteil zu wissen welcher der Prozessoren sich durchsetzt, wenn sich die schreibenden Prozessoren nicht einig sind. Daher setzt sich bei diesem Modell der Prozessor mit dem **kleinsten Index** durch. Dieses Verfahren ist nicht nur deterministisch, sondern liefert auch eine Möglichkeit zusätzliche Entscheidungen beim Schreibzugriff zu treffen.
  - **combine**: Diese zusätzliche Operation beim Speicherzugriff lässt sich natürlich auch erweitern, so dass man Modelle annehmen könnte, die bei einem Schreibzugriff eine Operation auf die zu schreibenden Daten anwendet um sie zu kombinieren. So könnten beispielsweise die an einer Speicherstelle zu schreibenden Daten aufsummiert werden.

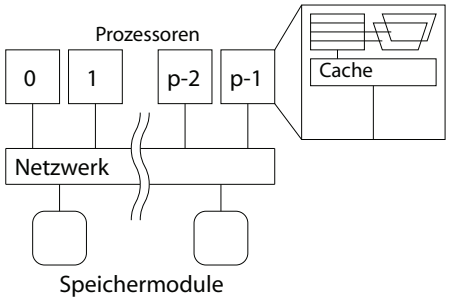
Bei einer derart reichhaltigen Vielfalt an Unterkategorien drängt sich natürlich die Frage auf, ob diese Unterteilung denn auch wirklich notwendig ist. Hier zeigt sich schnell, dass verschiedene Probleme sich nur schnell lösen lassen, wenn viele Prozessoren Zugriff auf eine gemeinsame Speicherstelle haben. Als Beispiel dafür ließe sich eine parallele AND-Operation angeben,

die für n Elemente auf einer CRCW PRAM mit n Prozessoren in einem Schritt das Ergebnis bestimmen kann (1.1). Dafür muss sich jeder Prozessor nur über seinen Index ein Element aussuchen und bei Fehlschlagen einer lokalen AND-Operation eine 0 als Ergebnis schreiben. Ist es nun allerdings nicht mehr erlaubt, dass mehrere Prozessoren zur gleichen Zeit ihr negatives Ergebnis eintragen, so kommt mindestens ein Faktor von $\log n$ hinzu, der benötigt wird um Speicherkonflikte zu vermeiden (siehe auch 1.7.2).

1.1.2 Realistischere Modelle

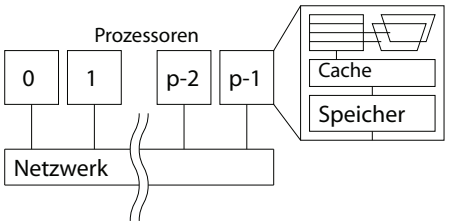
Die einfache Repräsentation der Prozessorinteraktion im PRAM-Modell ignoriert wichtige Eigenschaften reeller Systeme. Hier will man sich von der starken Annahme entfernen, dass alle Prozessoren schnell auf einen großen gemeinsamen Speicher zugreifen könnten. Daher zwei Modelle, die der Realität bereits näherkommen:

- **(Symmetric) Shared Memory (gemeinsamer Speicher):** Dieses Modell hat tatsächlich gewisse Ähnlichkeit mit dem PRAM-Modell. Zusätzlich berücksichtigt es allerdings Caches der einzelnen Prozessoren, sowie die Tatsache, dass in der Realität ein Netzwerk nötig ist, um die Prozessoren mit Speichermodulen zu verbinden. Neben der tatsächlichen Verwaltung der Speicherzugriffe kommt hier noch "false sharing" hinzu. Sobald ein Prozessor auf eine Speicheradresse zugreift, die ein anderer Prozessor im Cache hat, wird eine Synchronisation erforderlich. Selbst wenn der andere Prozessor diese Speicherstelle nie benötigt hat kann es passieren, dass sie zu einer Cachezeile gehört, auf die über eine andere Adresse häufig zugegriffen wird.



- **Distributed Memory (verteilter Speicher):** In vielen parallelen Systemen ist es nicht der Fall, dass die Prozessoren auf einem gemeinsamen Speicher arbeiten. Stattdessen besitzt jeder Prozessor hier einen

(vergleichsweise) kleinen lokalen Speicher. Zugriff auf die Daten anderer Prozessoren ist nur über expliziten Nachrichtenaustausch möglich. Eingaben erfolgen in diesem Fall üblicherweise verteilt über alle Prozessoren und auch die Ausgabe muss zum Schluss geeignet verteilt werden.



1.1.3 DAG

Ganz kurz soll hier auch auf das Modell der Schaltkreisfamilien eingegangen werden. Repräsentiert werden Schaltkreise hier als **gerichtete azyklische Graphen (directed acyclic graph, DAG)**. Bei Berechnungen in Graphenform wird die Eingabe durch Knoten mit Eingangsgrad 0 zur Verfügung gestellt. Das Ergebnis der Berechnung findet sich in Knoten mit Ausgangsgrad 0. Die weiteren Knoten werden "**innere Knoten**" genannt und sind für die eigentlichen Berechnungen verantwortlich. Sinnvollerweise wird für innere Knoten der Eingangsgrad durch eine kleine Konstante beschränkt, ihr Ausgangsgad darf dagegen beliebig sein.

Der Begriff, der der Laufzeit eines DAG am nächsten kommt, ist der Begriff der **Tiefe**. Für einen Knoten wird der längste Pfad von einem Eingangsknoten so bezeichnet. Alle Knoten, deren Höhe h ist, werden **Schicht** der Höhe h genannt. Die Höhe des DAG ist die maximale Höhe eines Ausgangsknotens, also der längste Pfad von einem Eingang zu einem Ausgang. Da die Anzahl Zwischenschritte, die benötigt werden können alle nötigen Informationen an die entsprechenden inneren Knoten zu bringen direkt mit dieser Größe zusammenhängt, gibt es bei einem "hinreichend kompakten" DAG stets mindestens eine Eingabe für die die Ausgabe die Auswertung aller Knoten auf dem längsten Weg erfordert. Damit lässt sich die Tiefe eines DAG direkt als Schranke für die Laufzeit benutzen. Mit Schaltkreisen ist hier ein Spezialfall der DAG gemeint, bei dem nicht beliebige Worte Ein- und Ausgabe sein können, sondern die Berechnung sich auf Bitfolgen konstanter Länge beschränkt.

Ein Schaltkreis hat stets eine feste Anzahl von Eingabeknoten. Um so etwas wie einen Algorithmus schreiben zu können, ist es jedoch nötig mit

unterschiedlichen Eingabegrößen umzugehen. Diese Möglichkeit wird durch die **Schaltkreisfamilien** gegeben. Als Schaltkreisfamilie wird eine Vorschrift bezeichnet, nach der für eine vorgegebene Eingabegröße ein Schaltkreis algorithmisch generiert werden kann. Sinnvollerweise möchte man sich dann auf Schaltkreisfamilien beschränken, bei denen der Schaltkreisgenerator nicht zu mächtig ist. Eine solche Einschränkung sind die **uniformen Schaltkreisfamilien**, die hier nicht weiter erläutert werden.

Uniforme Schaltkreisfamilien lassen sich aber durch ein PRAM simulieren, indem alle Knoten einer Schicht parallel mit einem Prozessor pro Knoten abgearbeitet werden. Naheliegenderweise ist der Bedarf der Prozessoren durch die maximale Anzahl der Berechnungsknoten einer Schicht beschränkt. Der Zeitbedarf ist bei schichtweiser Abarbeitung die Anzahl der Schichten, also gerade die Tiefe des Schaltkreises für die jeweilige Eingabelänge. Versucht man nun umgekehrt PRAM durch ein DAG zu simulieren muss vor allem das Programm von Zyklen befreit werden, die in einem DAG nicht repräsentiert werden können. Zuletzt lässt sich ein DAG auch als Verbindungsnetzwerk repräsentieren, wo Prozessoren die Aufgabe der Berechnungsknoten übernehmen und für jede Schicht aus Knotenlasten, Kantenlasten und Pfadlängen die Ausführungszeit bestimmt werden kann.

1.2 Netzwerk- und Kommunikationsmodelle

Will man Algorithmen entwerfen, die auch praktisch von Nutzen sein sollen, ist das ohne Berücksichtigung der Kommunikation unmöglich. Dieser Abschnitt beschäftigt sich daher damit, wie man die Verbindungsnetzwerke zwischen den Prozessoren repräsentieren kann und wie sie später bei der Analyse der Algorithmen eingehen.

1.2.1 Explizites "Store-and-Forward"

Bei dem ersten Modell das hier betrachtet wird, wird ein Netzwerk explizit als Graph modelliert. In diesem Fall repräsentieren Knoten des Graphen Prozessoren; Kanten repräsentieren die Verbindungen. Busse, an denen mehrere Prozessoren angeschlossen sind, lassen sich in so einem Fall mit Hyperkanten repräsentieren. Ebenfalls möglich sind zusätzliche Routerknoten, die auch über eigenen Pufferspeicher verfügen können, aber nur die Kommunikation unterstützen ohne an den Berechnungen beteiligt zu sein. Es ist auch möglich (global) Kantenkapazitäten festzulegen, die die Anzahl von Paketen konstanter Länge festlegen, die in einer Zeiteinheit über die zugehörige Verbindung transportiert werden können. Meist wird dieser Wert jedoch auf 1 festgesetzt. Ebenso lässt sich die Anzahl der Nachrichten beschränken, die ein Knoten parallel versenden oder empfangen kann. Ist dieser Wert k, so ist von k-**Port-Maschinen** die Rede, im Spezialfall $k = 1$ spricht man von einer **single-ported** Maschine.

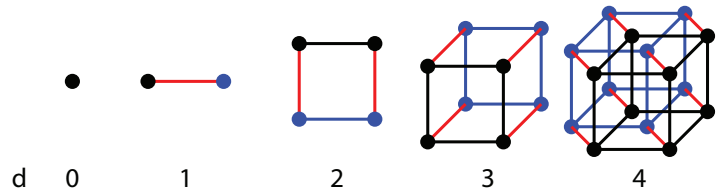


Abbildung 1.2: Hyperwürfel für einige d

Das Hauptproblem bei dieser Modellierung der Netzwerke ist der nicht gerechtfertigte Aufwand für den Algorithmenentwurf. Da das Routing damit praktisch in den Algorithmus integriert werden muss, leidet die Übersichtlichkeit. Nachdem Hardwarerouter diese Aufgabe zufriedenstellend übernehmen, ist es nicht sinnvoll diese Aufgabe in den Entwurf von Algorithmen einzugliedern. Es gibt allerdings durchaus auch Algorithmen, die spezielle Netzwerkstrukturen voraussetzen und dann ausnutzen, um eine geringe Laufzeit zu erreichen.

Eine Spezielle Netzwerkstruktur bei 2^d Prozessoren für ein geeignetes d, die oft elegante Algorithmen erlaubt, ist der (d-dimensionale) **Hyperwürfel**. Dabei sind die Knoten durch Tupel aus $(\mathbb{Z}/2\mathbb{Z})^d$ durchnummeriert. Diese Tupel werden für Algorithmen auch gerne als Wort über dem Alphabet $\{0, 1\}$ interpretiert. Kanten gibt es zwischen zwei Knoten genau dann, wenn sich ihre Koordinaten nur in einer Stelle unterscheiden.

1.2.2 Vollständige Verknüpfung

Das nun folgende Modell abstrahiert das explizite Store-and-Forward, so dass es sich bei der Analyse von Algorithmen beliebiger Netzwerke besser verwenden lässt. Dazu wird für das Netzwerk zunächst vollständige Verknüpfung angenommen. Nun wird das Versenden von m Bytes durch $T_{\text{comm}}(m) = T_{\text{start}} + mT_{\text{byte}}$ geschätzt. Dabei bezeichnet T_{start} die Zeit, die für den Verbindungssaufbau nötig ist, bevor in jeweils T_{byte} Zeit ein Byte übertragen wird.

Das sind zwar Annahmen, die auf reale Netzwerke kaum zutreffen, jedoch

kommt dem die Realität bereits relativ nahe. Hardwarerouter übernehmen das Auflösen von Konflikten auf Kanten und bei Strukturen mit schwacher Verbindung lassen sich noch immer T_{start} und T_{byte} künstlich erhöhen um häufige Konflikte zu modellieren. Ein besonderer Vorteil an diesem Modell ist dass es nicht notwendigerweise synchrone Kommunikation beschreiben muss.

Zuletzt soll noch geklärt werden, was "eine Kommunikation" mit einer m-Byte-Nachricht eigentlich ist. Insbesondere zwei der folgenden Begriffe tauchen später immer wieder auf um die "Mächtigkeit" der Kommunikation zu beschreiben:

- **halbduplex:** Bei dieser Variante wird nicht zwischen Senden und Empfangen unterschieden. Das bedeutet, dass das Versenden von m Bytes sowohl bei Sender, als auch bei Empfänger die Zeit für m Byte Kommunikation kostet.
- **telefon:** In diesem Fall dürfen zwei Prozessoren die Zeit nutzen, um m-Bytes parallel in beide Richtungen auszutauschen. Jedoch darf nur von einem Prozessor parallel empfangen werden, an den auch gesendet wird.
- **(voll)duplex:** Hier fällt zusätzlich zum Telefonmodell auch die Festlegung der Partner weg, und eine beliebige m-Byte-Empfangsoperation darf sich stets mit einer m-Byte-Sendeoperation überlappen.

Es folgt eine Hierarchie: $T^{\text{duplex}} \leq T^{\text{telefon}} \leq T^{\text{halbduplex}} \leq 2T^{\text{duplex}}$

1.2.3 BSP

Eine weitere Möglichkeit die Kosten einer Kommunikation darzustellen ist das Bulk Synchronous Parallel (BSP) Modell. Die Idee bei diesem Modell ist, dass nach jeweils einer Rechenphase alle Prozessoren Nachrichten in einer kollektiven Kommunikationsoperation austauschen und anschließend vor der nächsten Rechenphase durch eine Barriere synchronisiert werden. Rechenphase, Kommunikation und Synchronisation werden unter dem Begriff "Superschritt" zusammengefasst. In der Origialfassung wird der Startupoverhead für die Kommunikation nicht modelliert mit der Begründung dass er bei der angestrebten Größe der Kommunikationsoperation vernachlässigbar wäre, hier soll er jedoch in einem Parameter mit in die Zeitberechnungen einfließen.

In diesem Modell wird der Zeitaufwand einer Kommunikationsoperation durch $l + hg$ dargestellt. Die Parameter haben dabei folgende Bedeutung:

- **l:** Repräsentiert den **Startupoverhead** für den kollektiven Nachrichtenaustausch und die Kosten für die **Barrierensynchronisation**. Es sollte beachtet werden, dass die Barrierensynchronisation stets mit $\log p$ Zeit zu Buche schlägt.
- **h: Maximale Anzahl der versendeten Pakete** pro PE.
- **g:** Steht für "gap" und beschreibt die Zeit, die ein Netzwerk braucht, um ein einzelnes Paket in einem **kontinuierlichen Strom** an **ein gleichverteilt-zufälliges Ziel** auszuliefern. Gemessen wird g in Rechenschritten und ist wie schon l üblicherweise eine Funktion von p. Wie schon l kann auch g eine Funktion von p sein.

In diesem Modell ist damit also ein Verbindungsnetzwerk bereits durch l, g und p hinreichend beschrieben, die Topologie wird vernachlässigt. Ein klarer Vorteil dieses Modells ist die einfache aber präzise Beschreibung, sowie die gute Anpassungsfähigkeit. Die Festlegung auf Superschritte, die jeweils mit globaler Synchronisation enden, ist andererseits häufig eine zu radikale Einschränkung. Die Vernachlässigung der Netzwerktopologie hat den zusätzlichen Nachteil, dass bei Kommunikation nicht zwischen verschieden schwierigen Kommunikationsmustern unterschieden werden kann. So hat es im BSP-Modell keinen Einfluss auf die Laufzeitabschätzung sich auf Nachbarschaftskommunikation zu beschränken. Es existieren allerdings durchaus erweiterte BSP-Modelle, die zusätzliche Kommunikationsmuster berücksichtigen oder auch unterschiedliche Nachrichtenlängen.

Der für BSP typische kollektive Nachrichtenaustausch, bei dem jedes PE bis zu h Pakete sendet oder empfängt (wobei die Adressaten keine Rolle spielen) wird h-Relation genannt. Man beachte, dass nicht nur die Anzahl der gesendeten Pakete durch h begrenzt ist, sondern auch die der empfangenen Pakete. Diese Kommunikationsoperation wird in einem späteren Kapitel (3.6.4) auch unabhängig vom BSP-Modell betrachtet.

1.3 MPI und OpenMP

Bisher haben wir uns damit beschäftigt mit welchen Modellen parallele Algorithmen analysiert werden. Dieses Kapitel soll die Grundlagen der tatsächlichen Implementierung vermitteln. Es werden die Standards des Open Multi Processing (OpenMP) und des Message Passing Interface(MPI) vorgestellt.

1.3.1 OpenMP

OpenMP ist ein API, das eine Entwicklung von portierbaren, skalierbaren shared Memory Programmen erleichtert. Die OpenMP Befehle werden mit Hilfe von Pragmas in C/C++ Code eingearbeitet. Dadurch kann der Code auch Kompiliert werden, wenn der Compiler kein OpenMP beherscht. Am Besten kann das gesehen werden an einem Beispiel:

```
#pragma omp parallel for schedule(static, n)
for( int i = 0; i < 100; i++){
  do_something(i);
}
```

Wenn dieser Code mit einem Compiler kompiliert wird, der kein OpenMP unterstützt, dann wird er sequenziell ausgeführt. Mit Hilfe von OpenMP werden die Schleifendurchläufe parallelisiert. Dabei werden jedoch nicht 100 Threads erzeugt, sondern die 100 Schleifendurchläufe werden in Chunks aufgeteilt. Der Parameter **schedule(...)** gibt an, wie groß die Chunks sind und wie sie an die Threads verteilt werden. Man kann ihn weglassen und damit die Entscheidung dem Compiler überlassen. Gültige Werte sind **schedule(static, n)**, **schedule(dynamic, n)** oder **schedule(guided, n)** benutzen. Bei **schedule(static, n)** werden Blöcke der Größe n zyklisch an die Threads verteilt. **schedule(dynamic, n)** sorgt dafür dass jeder Thread einen Block der Größe n bekommt und immer wenn er damit fertig ist, dann fordert er n neue Iterationen an. **schedule(guided, n)** ist so ähnlich wie **schedule(dynamic, n)**, bloss dass jeder Prozessor max(n, (Anzahl an noch nicht zugewiesenen Iterationen)/p) bekommt.

```
int square_shared = 0, square_private = 0;
#pragma omp parallel for shared(square_shared) private(square_private)
for( int i = 0; i < 100; i++){
  square_shared = i*i;
  square_private = i*i;
  printf("Iteration %d: square_shared = %d, square_private = %d\n",
    i, square_shared, square_private);
}
printf("Outside of loop: square_shared = %d, square_private = %d\n",
  square_shared, square_private);
```

Dieser Code gibt Quadratzahlen bis 99^2 aus. Jeder Thread hat seine eigene *square_private* Variable aber es gibt insgesammt nur eine *square_shared* Variable für alle Threads. Dadurch überschreiben sie sich den Wert von *square_shared* gegenseitig und so sind die Werte für *square_shared* und *square_private* in der Ausgabe nicht immer gleich. Ausserhalb der Schleife ist

der *square_shared* auf den letzten Wert gesetzt den er in der Schleife bekommen hat; das muss aber nicht 99^2 sein. Der Wert von *square_private* ist nach der Schleife nicht definiert.

```
omp_set_num_threads(5);
#pragma omp parallel for num_threads(7)
for( int i = 0; i < 100; i++){
  printf("Iteration %d wird von Thread %d/%d bearbeitet\n",
    i, omp_get_thread_num(), omp_get_num_threads());
}
```

Bei vielen parallelen Programmen ist es wichtig zu Bestimmen wie viele Threads erzeugt werden oder zumindestens zu wissen wie viele Threads erzeugt wurden. Mit Hilfe des Parameter **num_threads** des Pragma kann man festlegen wie viele Threads erzeugt werden. Falls es keinen **num_threads** Parameter gibt, dann wird eine Variable benutzt die man mit Hilfe von **omp_set_num_threads** setzen kann. Mit Hilfe von **omp_get_num_threads** kann festgelegt werden wie viele Threads in dieser parallelen Region erzeugt wurden. Das bedeutet außerhalb von parallelen Konstrukten gibt es Eins zurück. Um die Threads unterscheiden zu können, kann mit Hilfe **omp_get_thread_num** die Thread ID herausfinden.

```
#pragma omp parallel sections
{
  #pragma omp section
  {
    printf("Section 1\n");
  }
  #pragma omp section
  {
    printf("Section 2\n");
  }
  #pragma omp section
  {
    printf("Section 3\n");
  }
}
```

Eine Andere Möglichkeit Parallelität zu erzeugen ist es Sections zu benutzen. Jede dieser Sections wird von genau einem Thread ausgeführt. Falls es mehr Threads gibt als Section dann warten die anderen Threads. Es kann passieren, dass mehrere Sections von dem gleichen Thread ausgeführt werden. Auch bei diesem Pragma kann man **shared**, **private** und **num_threads** benutzen.

```
#pragma omp parallel num_threads(3) shared(i) private(j)
{
  #pragma omp sections
  {
    #pragma omp section
    {
      printf("Section 1 wird von Thread %d/%d bearbeitet\n",
        omp_get_thread_num(), omp_get_num_threads());
    }
    #pragma omp section
    {
      printf("Section 2 wird von Thread %d/%d bearbeitet\n",
        omp_get_thread_num(), omp_get_num_threads());
    }
  }
  #pragma omp single
  {
    printf("work only one thread does\n");
  }
  printf("work everybody does\n");
  #pragma omp barrier
  #pragma omp for schedule(static, 2)
  for( int i = 0; i < 10; i++){
    printf("Iteration %d wird von Thread %d/%d bearbeitet\n",
      i, omp_get_thread_num(), omp_get_num_threads());
  }
}
```

Es ist möglich mehrere parallele Abschnitte hintereinander zu hängen ohne, dass die Threads zwischendurch zerstört werden und ihre privaten Daten verlieren. Dazu erzeugt man verschiedene Threads und gibt dann die parallelen Befehle an die abzuarbeiten sind. Normale Befehle werden dabei von jedem abgearbeitet. Damit Befehle nur von einem Prozessor abgearbeitet werden, muss man ein **#pragma omp single** davor setzen. Falls es irgendwann wichtig ist, dass ein Schritt von allen Threads abgearbeitet ist bevor ein Thread zu dem nächsten übergeht, dann kann man das mit Hilfe von **#pragma omp barrier** erreichen.

1.3.2 MPI

Das Message Passing Interface(MPI) ist für Systeme mit verteiltem Speicher gedacht. Dadurch funktioniert es auch auf gemeinsamen Speicher, nutzt aber dessen Vorzüge nicht. Jeder Thread führt das gleiche Programm aus. Wenn jeder Thread jedoch auch die identischen Befehle ausführen würde, dann würde es zu einem Deadlock kommen sobald alle Threads senden oder alle Threads empfangen. Unterschiedliche Programmverläufe werden durch die Thread ID erzeugt. Die Thread ID wird durch die Funktion **MPI_Comm_rank** ausgelesen. Wie der Name sagt basiert MPI auf dem versenden von Nachrichten. Daher sind **MPI_Send** und **MPI_Recv** die wichtigsten Funktionen von MPI. **MPI_Send** wird die Nachricht übergeben, die Anzahl an Elementen, der Typ der Elemente, der Zielthread, ein beliebiges Tag und ein MPI_Comm. Ein MPI_Comm verwaltet eine Gruppe von Threads. Jeder Thread kann mehrerer dieser Gruppen angehören und kann in jeder dieser Gruppen eine unterschiedliche Thread ID haben. MPI_COMM_WORLD ist ein MPI_Comm dem alle Threads angehören. Die Parameter von **MPI_Recv** sind ein Empfangsbuffer, die Länge des Buffers, der Typ des Buffers, die Quelle und der Tag der beim Absenden gesand wurde, das MPI_Comm und ein Pointer auf ein Status-Object. Wenn man die Quelle und den Tag der Nachricht nicht kennt, oder es egal ist woher bzw. mit welchem Tag die Nachricht gesendet wurde, dann kann man auch die Wildcards MPI_ANY_SOURCE bzw. MPI_ANY_TAG verwenden. Das Status-Object wird von dem Recv gefüllt und enthält dann den Tag und den Ursprungsort der Nachricht. Außerdem kann man die Datenmenge mit Hilfe der Funktion **MPI_Get_count** auslesen.

```
int main(int argc, char* argv[]){
  MPI_Init(&argc, &argv);
  int size, rank.
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Hello, I am process %d of %d.\n", rank, size);
  if((rank == 0)&&(size>=2)){
    char* message = "Hallo von 0"
    MPI_Send(message, strlen(message)+1, MPI_CHAR,
      1, 0, MPI_COMM_WORLD);
  }else if(rank == 1){
    MPI_Status status;
    char message[100];
    int count;
    MPI_Recv(message, 100, MPI_CHAR, MPI_ANY_SOURCE,
```

```
      MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    MPI_Get_count(&status, MPI_CHAR, &count);
    printf("Nachricht von %d ist: \"%s\", hat eine Laenge "
      "von %d und hat den Tag %d\n",
      status.MPI_SOURCE, message, count, status.MPI_TAG);
  }
  MPI_Finalize();
}
```

In vielen Fällen will man aber eine Nachricht von einem Thread an alle anderen senden (3.2 oder 3.6) oder andere Operationen machen an denen alle Threads teilnehmen soll. Die wichtigsten dieser Funktionen sind in MPI definiert. Es gibt die Funktionen MPI_Bcast, MPI_Scatter, MPI_Gather, MPI_Reduce, MPI_Alltoall und andere, die die Funktionen Broadcast(3.2), Scatter(ein Thread sendet an alle anderen), Gather(ein Thread empfängt von jedem anderen), Reduktion(3.1) und All-to-All(3.6) definieren. Genaue Parameterdefinitionen und die anderen kollektiven Operationen können dem Internet entnommen werden. Hier sei noch gesagt, dass die Operationen blockieren bis alle Threads des MPI_Comm die Funktion aufgerufen haben. Falls man nicht will, dass alle Threads an der kollektiven Operation teilnehmen, dann muss man vorher einen neuen MPI_Comm erzeugen, dem nicht alle angehören. Das geht jedoch über die Grundlagen, die hier beschrieben sind hinaus. Hier noch ein abschließendes Beispiel, das kollektive Operationen verwendet. Jeder Thread sendet eine Quadratzahl an den 0ten Thread. Dieser addiert 20 drauf und sendet es zurück.

```
int main(int argc, char* argv[]){
  MPI_Init(&argc, &argv);
  int size, rank.
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int square = rank*rank;
  int result;
  int buffer[size];
  MPI_Gather(&square, 1, MPI_INT,
      buffer, 1, MPI_INT, 0, MPI_WORLD);
  if(rank == 0){
    for(int i=0; i<size; i++)
      buffer[i]+=20;
  }
  MPI_Scatter(buffer, 1, MPI_INT,
      &result, 1, MPI_INT, 0, MPI_WORLD);
  printf("Thread %d has reult %d\n", rank, result);
```



```
  MPI_Finalize();
}
```

1.4 MCSTL

Die **M**ulti **C**ore **S**tandard **T**emplate **L**ibrary [7] stellt eine Erweiterung der C++ STL dar. Hinter ihr steht die Idee die Algorithmen, die von der STL zur Vefügung gestellt werden effektiv zu parallelisieren, so dass der Aufwand für die Parallelisierung von Anwendungen sinkt. Im besten Fall würde ein neues Kompilieren bereits reichen, damit ein Programm, das Algorithmen aus der STL verwendet, von Parallelisierung profitieren kann.

Die MCSTL basiert auf OpenMP und daher sind ihr Einsatzgebiet parallele Maschinen mit gemeinsamem Speicher. Ziel ist es für alle parallelisierbaren Algorithmen parallele Implementierungen zu integrieren, die der STL-Semantik folgen. Einige Algorithmen – wie Binärsuche – lassen sich offensichtlich nicht effektiv parallelisieren und für alle parallelen Algorithmen werden Iteratoren mit wahlfreiem Zugriff benötigt. Da ein breites Einsatzgebiet angestrebt wird, soll die Parallelisierung bereits bei kleinen Eingaben und wenigen Prozessoren eine Beschleunigung bewirken. Da damit auch nicht mehr zu erwarten ist, dass ein solches Programm vorrangig ausgeführt wird, ist auch mit starken Schwankungen in der Leistung der Prozessoren zu rechnen. Das macht es nötig dass die Arbeitsaufteilung im Laufe der Abarbeitung dynamisch angepasst wird. Im umgekehrten Fall sollen die Algorithmen das System nicht unnötig auslasten, um die Laufzeit geringfügig zu verbessern. In anderen Worten: man möchte hohe Effizienz erreichen. Die Funktionsweise einiger Algorithmen aus der MCSTL wird im Zusammenang mit Sortieren im entsprechenden Kapitel betrachtet.

1.5 Analyse von Algorithmen

Für die Analyse der Qualität von Algorithmen sollen im Folgenden verschiedene Kenngrößen eingeführt werden. Begonnen werden soll ganz analog zur Analyse sequenzieller Algorithmen mit der Ausführungszeit:

- $T(I)$: Ausführungszeit, Anzahl von Zyklen für geg. Probleminstanz I
- $T(n) := max_{|I|=n} T(I)$: Worst case Ausführungszeit

- $T_{\text{avg}}(n) := \frac{\sum_{|I|=n} T(I)}{|\{I:|I|=n\}|}$: Average case Ausführungszeit

  Beispiel: Quicksort besitzt average case Ausführungszeit $O(n \log n)$

Eine Warnung an dieser Stelle: Bei probabilistischen (randomisierten) Algorithmen ist $T(n)$ eine Zufallsvariable. Die erwartete worst case Laufzeit $\mathbb{E}[T(n)]$ sollte nicht mit dem average case verwechselt werden.

Betrachtet man nun parallele Algorithmen kommt im Wesentlichen nur p als zusätzlicher Parameter hinzu. Es wird bei den Zeitbetrachtungen also $T(I,p)$ statt $T(I)$ betrachtet und die restlichen Kostenmaße wieder wie gewohnt abgeleitet. So erhält man die parallele Ausführungszeit $T(n,p)$ für Eingablänge n mit p Prozessoren. Jetzt liefert allerdings die Prozessorzahl auch eigene Perspektiven, aus denen die Qualität eines Algorithmus betrachtet werden kann:

- $W := pT(n,p)$: Arbeit, beschreibt die gesamte Zeit, die alle Prozessoren zusammen benötigen.
- $S := T_{seq}/T(p)$: (absoluter) Speedup , beschreibt Beschleunigung gegenüber dem **besten bekannten sequentiellen** Algorithmus. Die Erwartung, dass S durch p nach oben beschränkt ist, ist naheliegend, schließlich würde es bedeuten, dass die Rechenzeit verlustfrei über die Prozessoren verteilt ist. Dennoch gibt es Fälle, in denen $S > p$ gemessen werden kann. In diesem Fall wird von superlinearem Speedup gesprochen. Der häufigste Grund dafür sind Cacheeffekte durch eine größere Anzahl Prozessoren mit eigenem Cache. Suchalgorithmen können bei der Parallelisierung auch davon profitieren dass mehrere Pfade im Suchraum parallel durchsucht werden können und bereits ein Treffer reicht.
- $S_{rel} := T(1)/T(p)$: relativer Speedup. Wie unterschiedlich diese beiden Arten sind zeigt ein Beispiel: Maximum (1.2) hat $S_{rel} = \Theta(n^2)$ aber $S = \Theta(n)$.
- $E := S/p$: Effizienz, beschreibt wie gut das Problem bei einem Algorithmus auf alle Prozessoren verteilt wird und auch Zeitverlust durch die Verwaltung der Verteilten Berechnung. Ziel ist $E \approx 1$ oder zumindest $E = \Theta(1)$.

Für einige randomisierte Algorithmen werden Laufzeitschranken nicht allgemein bewiesen. Oft lässt sich nur beweisen, dass sie mit hoher Wahrscheinlichkeit gelten. Für diesen Fall wird das $\tilde{O}$-Kalkül als Erweiterung des

O-Kalküls eingeführt. Definiert wird es wie folgt:

$$f(n) \in \tilde{O}(g(n)) \Leftrightarrow \forall \beta > 0 : \exists c > 0, n_0 > 0 : \forall n \geq n_0 : \mathbb{P}[f(n) > cg(n)] \leq n^{-\beta}$$

Das bedeutet also, dass $f(n)$ dann mit hoher Wahrscheinlichkeit in $O(g(n))$ liegt. Insbesondere wird die Wahrscheinlichkeit mit wachsendem n beliebig groß.

Um zu zeigen, dass $f(n) \in \tilde{O}(g(n))$ gilt, sind Chernoff-Schranken häufig nützlich. Chernoff-Schranken liefern eine Abschätzung für die Wahrscheinlichkeit mit der $X := \sum_i X_i$ eine bestimmte Abweichung von seinem Erwartungswert erreicht. Dabei sind X_i unabhängige Zufallsvariablen. Im Folgenden sei $0 < \epsilon < 1$ und $\alpha > 1$. Es gelten dann die Ungleichungen:

$$\mathbb{P}[X \leq (1-\epsilon)\mathbb{E}[X]] \leq e^{-\epsilon^2 \mathbb{E}[X]/2}$$
$$\mathbb{P}[X \geq (1+\epsilon)\mathbb{E}[X]] \leq e^{-\epsilon^2 \mathbb{E}[X]/2}$$
$$\mathbb{P}[X \geq \alpha \mathbb{E}[X]] \leq e^{(1-\frac{1}{\alpha}-\ln\alpha)\alpha\mathbb{E}[X]}$$

1.6 Konvention

Dieses Skript bemüht sich die Symbole in den folgenden Algorithmen einheitlich zu verwenden. Im Allgemeinen haben sie die folgende Bedeutung:

p: Anzahl der PE

n: Anzahl der Elemente einer Eingabe, Nachrichtenlänge bei Analyse von Kommunikation

N: Anzahl der Elemente pro Prozessor bei verteilter Eingabe ($n = Np$)

Die Algorithmen werden in an C angelehntem Pseudocode beschrieben, erweitert um parallele Operationen:

- `parallel_for(...)` bezeichnet eine for-Schleife, deren Iterationen über die Prozessoren verteilt werden. Offensichtlich ist dafür notwendig dass die Iterationen unabhängig voneinander sind.
- **v@i** bezeichnet für verteilten Speicher die Variable v auf dem Prozessor mit Index i. Damit wird so implizit eine (nicht näher beschriebene) Kommunikationsoperation dargestellt.
- `concurrently{ ... } with { ... }` wird benutzt, um explizit darauf hinzuweisen, dass die (Kommunikations-)Operationen in beiden Blöcken auf einem Prozessor parallel ausgeführt werden können.

Grundsätzlich wird auf allen Prozessoren das gleiche Programm ausgeführt (single program multiple data, nicht zu verwechseln mit single instruction multiple data). Damit die Symmetrie gebrochen wird steht jedem Prozessor **der eigene Index** als Variable i zur Verfügung. Für einige Algorithmen mit $p = k^d$ Prozessoren wird zugunsten der Anschaulichkeit ein Index-Tupel verwendet. Die Prozessoren werden dann durch Tupel mit Werten in $(\mathbb{Z}/k\mathbb{Z})^d$ bezeichnet. Zur Veranschulichung kann man sich in diesem Fall die Prozessoren in einem d-dimensionalen Gitter angeordnet vorstellen. Der Index wären dann die Koordinaten im Gitter.

1.7 PRAM-Algorithmen

Die in diesem Abschnitt beschriebenen Algorithmen sind weniger von praktischer Bedeutung. Stattdessen deuten sie die Möglichkeiten und Probleme der Parallelausführung an.

1.7.1 Global AND

Problembeschreibung: Gegeben ist ein Array von Bits: $A[1..n], A[i] \in \{0,1\}$. Gesucht ist $\bigwedge_{i=1}^{n} A[i]$ auf einer common CRCW PRAM.

Algorithmus 1.1: Global AND

```
result:=1;
parallel_foreach (i ∈ {1..n}){
  if(¬A[i]) result:=0;
}
```

Erläuterungen: Für die folgende Analyse wird von der für PRAM typischen Annahme ausgegangen, dass eine beliebige Anzahl Prozessoren verwendet werden kann, um ein Problem zu lösen. Da die Schleifeniterationen unabhängig voneinander sind, lassen sich so die n Iterationen über n Prozessoren verteilen, so dass für die Berechnung nur ein Schritt nötig ist. Nun bleibt noch der Schreibzugriff, bei dem allerdings häufig mehrere Prozessoren auf eine Speicherstelle zugreifen müssen. Um die konstante Ausführungszeit zu erhalten, ist daher eine CRCW PRAM notwendig. Da nur eine 0 geschrieben werden soll und die Prozessoren sich damit stets einig sind, reicht hier bereits eine common CRCW PRAM.

Analyse:

$$T(n) \in O(1) \text{ für } p \in O(n)$$

1.7.2 Theoretiker-Maximum

Problembeschreibung: Gegeben ist ein Array von Zahlen $A[1..n]$ für die auf einer common CRCW PRAM das Maximum bestimmt werden soll. Das Ergebnis soll in $M[1..n], M[i] \in \{0,1\}$ geschrieben werden, wobei genau dann eine 1 in $M[i]$ steht, wenn $A[i]$ ein maximales Element in $A[1..n]$ ist, sonst soll eine 0 eingetragen werden.

Algorithmus 1.2: Theoretiker-Maximum

```
parallel_foreach ((i, j) ∈ {1..n}^2){
  B[1, j] := A[i] ≥ A[j];
}
parallel_foreach ((i, j) ∈ {1..n}){
  M[i] := ⋀_{j=1}^{n} B[i, j];
}
```

Erläuterungen: Hier wird wieder die Möglichkeit des PRAM-Modells genutzt eine beliebige Anzahl von Prozessoren zu verwenden, um das Problem zu lösen. In diesem Fall braucht man gerade einmal zwei Schritte, wenn n^2 Prozessoren zur Verfügung stehen. Die Prozessoren werden mit zweidimensionalen Indizes bezeichnet, also mit Werten in $\mathbb{Z}/n\mathbb{Z}^2$. Zu beachten ist, dass in der zweiten Schleife wieder n^2 Prozessoren ausgelastet sind, da in jeder der n Iterationen n Prozessoren für die Bestimmung von AND in einem Schritt (1.1) benötigt werden. Von diesem Algorithmus wird auch die Notwendigkeit einer CRCW PRAM geerbt.

Analyse:

$$T(n) = O(1) \text{ für } p \in O(n^2)$$

2 Lineare Algebra

2.1 Matrixmultiplikation

Problembeschreibung: Gegeben sind zwei Matrizen $A = ((a_{ij})) \in R^{n\times n}$, $B = ((b_{ij})) \in R^{n\times n}$ über einem Ring R. Bestimmt werden soll $C = ((c_{ij})) := A \cdot B$ wobei $c_{ij} := \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$. Das Problem braucht sequentiell $\Theta(n^3)$ Zeit, wobei die Schranke verbessert werden kann, falls R kommutative Multiplikation bietet.

2.1.1 Einfache Parallelisierung

Algorithmus 2.1: Verteilte Matrixmultiplikation I

```
parallel_for(i:=1; i<n; i++){
  parallel_for(j:=1; j<n; j++){
    c_ij := Σ_{k=1}^{n} a_ik · b_kj;
  }
}
```

Erläuterungen: Das ist die naheliegende Parallelisierung der Matrizenmultiplikation. Jedes PE berechnet hier die c_{ij} für einen Teil der i und j. Sind die c_{ij} quadratisch angeordnet, so ist die Breite $n/\sqrt{p}$ und so viele Zeilen von A bzw. Spalten von B werden für die Berechnung benötigt. Dass die quadratische Anordnung auch die Anzahl der benötigten Zeilen/Spalten minimiert, ist naheliegend und damit liegt auch die Untergrenze für die Kommunikation bei $n \cdot (n/\sqrt{p})$ Nachrichten. Bei n^2 Prozessoren sammelt jeder eine Zeile von

A und eine Spalte von B und bestimmt einen Eintrag für die Ergebnismatrix. Hat man mehr Prozessoren, kann auch die Summe für die Berechnung des c_{ij} parallelisiert werden.

Analyse:

$$T(n,p) \in \Omega(T_{\text{byte}}\frac{n^2}{\sqrt{p}})$$

2.1.2 Matritzenmultiplikation nach Cannon

Algorithmus 2.2: Verteilte Matrixmultiplikation nach Cannon

```
//PE(i,j)
k := (i+j) mod ñ;
a := a_ik;
b := b_kj;
c_ij := 0;
for(l := 0; l<ñ-1; l++){
  c_ij := c_ij + a·b;
  concurrently{
    send a to PE(i,(j+ñ-1) mod ñ);
    send b to PE((i+ñ-1) mod ñ,j);
  }with{
    receive a' from PE(i,(j+1) mod ñ);
    receive b' from PE((i+1) mod ñ,j);
  }
  a := a';
  b := b';
}
```

Erläuterungen: Für den Anfang ist es leichter sich vorzustellen p wäre n^2, es gäbe also so viele Prozessoren wie Einträge in den Matrizen. Die Prozessoren denkt man sich nun in einem Torus der Größe $n \times n$ angeordnet. Dementsprechend soll ein Index-Tupel mit Werten in $\mathbb{Z}/n\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z}$ verwendet werden, um Prozessoren zu adressieren.

Was in diesem Algorithmus nun erreicht werden soll, ist das k in jedem Schritt für jedes PE(i,j) so zu wählen, dass keine zwei Prozessoren gemeinsame Daten für die Berechnung von $a_{ik} \cdot b_{kj}$ brauchen. Dazu müssen Prozessoren in der gleichen Reihe/Spalte die Summation nur bei unterschiedlichen Indizes beginnen. Wenn also PE(0,0) im ersten Schritt $a_{00} \cdot b_{00}$ bestimmt, wählt

PE(0,1) $a_{01} \cdot b_{11}$ zuerst. Die Wahl $k := (i + j) \bmod n$ für PE(i,j) erfüllt die gewünschte Bedingung als Wahl für den ersten Schritt. In den Folgenden Schritten kann nun jeder Prozessor ein neues $k' := (k + 1) \bmod n$ wählen. Tun das alle, so bleiben ihre relativen Unterschiede erhalten und sie greifen immer noch alle auf unterschiedliche Daten zu. Die Daten, die sie dabei brauchen, sind dann auch gerade bei ihren Nachbarn. Ein PE(i,j) braucht dann das a von PE$(i,(j + 1) \bmod n)$ und das b von PE$((i + 1) \bmod n,j)$ für den nächsten Schritt. Das bedeutet, dass das a zyklisch nach links weitergereicht werden muss und ebenso das b zyklisch nach oben. Die Ergebnisse der Multiplikationen werden wie gewohnt aufsummiert. Nach n Schritten hat jeder Prozessor alle $a_{ik} \cdot b_{kj}$ einmal berechnet und seine Summe ist damit das gesuchte c_{ij}.

Nun hat man nicht immer n^2 Prozessoren zur Hand und möchte deswegen von dem hohen Bedarf wegkommen. Eine Feststellung, die hier dafür benutzt wird ist, dass eine Matrix auch in Teilmatrizen aufgeteilt werden kann, die wiederrum Elemente einer "groben" Matrix sind, wie es auf Bild 2.2a angedeutet wird. Die Matrixmultiplikation mit zwei solchen groben Matritzen liefert noch immer das gleiche Ergebnis. Mit dieser Einsicht kann man auch bei großen Matrizen so tun als hätte man n^2 Prozessoren zur Verfügung, wenn man statt n ein neues $\tilde{n} := \sqrt{p}$ verwendet. Zur Vereinfachung nehmen wir weiterhin an, dass dass $\sqrt{p}$ eine "schöne" Zahl ist und $\tilde{n}$ n teilt. Dann ist nur noch zu beachten, dass aus der elementaren Multiplikation eine Matrixmultiplikation auf $n/\tilde{n} \times n/\tilde{n}$ Matrizen wird.

Eine schöne Eigenschaft des Algorithmus ist, dass nach der anfänglichen Verteilung jeder Prozessor sich nur noch die Daten für den nächsten Schritt merken muss. Das sind das Zwischenergebnis der bisherigen Summe, ein a_{ik} und ein b_{kj}. Damit müssen alle drei Matrizen nur einmal gleichmäßig über die Prozessoren aufgeteilt im Speicher bleiben. Für die Geschwindigkeit nicht optimal ist die Sequentielle Summe auf jedem Prozessor, wie man am folgenden Algorithmus sieht.

Analyse:

$$T(n,p) = T_{\text{coll}}(n/\tilde{n},p) + \tilde{n}T_{\text{seq}}(n/\tilde{n}) + 2(\tilde{n}-1)(T_{\text{start}} + T_{\text{byte}}(n/\tilde{n})^2)$$

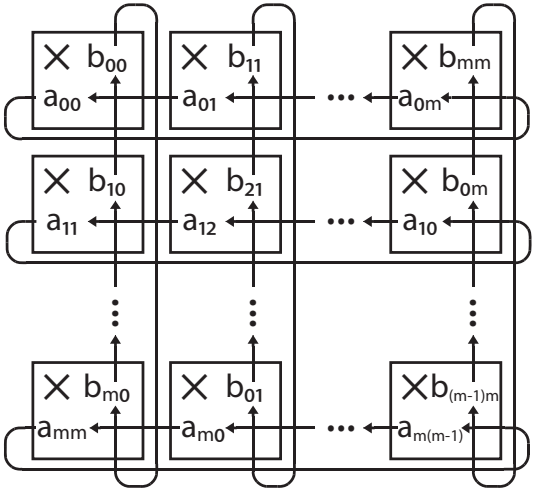


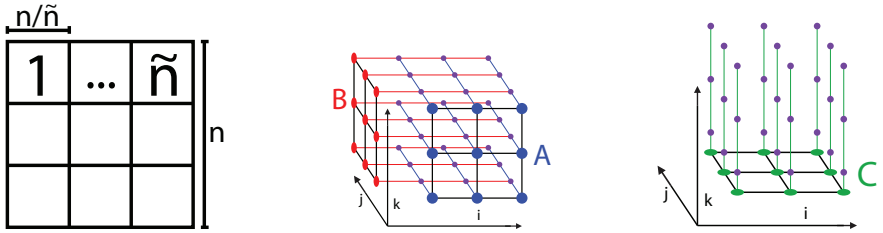
Abbildung 2.1: Beispiel für Matrixmultiplikation nach Cannon

2.1.3 Matrizenmultiplikation nach Dekel Nassimi Sahni

Algorithmus 2.3: Verteilte Matrixmultiplikation nach DNS

store a_{ik} in PE$(i,k,1)$;
store b_{kj} in PE$(1,k,j)$;
PE$(i,k,1)$ broadcasts a_{ik} to PEs(i,k,j) for $j \in \{1..\tilde{n}\}$;
PE$(1,k,j)$ broadcasts b_{kj} to PEs(i,k,j) for $i \in \{1..\tilde{n}\}$;
compute $c_{ikj} := a_{ik} \cdot b_{kj}$ on PE(i,k,j); *//lokale Berechnung*
PEs(i,k,j) **foreach** $(k \in \{1..\tilde{n}\})$ compute $c_{ij} := \sum_{k=1}^{\tilde{n}} c_{ikj}$ to PE$(i,1,j)$;

Erläuterungen: Bei diesem Algorithmus nach Dekel Nassimi Sahni [3] wird nicht mehr Zugriff auf die ganze Zeile/Spalte benötigt. Wie bei der Matrixmultiplikation nach Cannon wird auch hier ausgenutzt, dass sich die Matrix in Teilmatrizen aufspalten lässt. Bei diesem Algorithmus benutzt man aber $\tilde{n} := \sqrt[3]{p}$ und damit $\tilde{n}^3$ in einem Würfel angeordnete Prozessoren. Die Matrixmultiplikation soll dann auf $\tilde{n} \times \tilde{n}$ Matrizen durchgeführt werden, deren Einträge Matrizen der Größe $n/\tilde{n} \times n/\tilde{n}$ sind. Der Einfachheit halber soll $\tilde{n}$ n teilen. Es wird auch hier aus der elementaren Multiplikation eine Matrixmultiplikation auf $n/\tilde{n} \times n/\tilde{n}$ Matrizen.



(a) Aufteilung in Teilmatrizen

(b) DNS-Algorithmus Phase I

(c) DNS-Algorithmus Phase II

Abbildung 2.2: Der DNS-Algorithmus

Der eigentliche Algorithmus besteht aus zwei Phasen. In der ersten Phase werden die $a_{ik} \cdot b_{kj}$ für alle möglichen Indizes i, j und k auf verschiedenen Prozessoren bereitgelegt. Das kann man sich als dreidimensionalen Würfel mit den drei Indizes als Koordinaten vorstellen (vgl. Bild 2.2b). Der Prozessorindex wird dafür auch angepasst, so dass die $\tilde{n}^3$ Prozessoren durch Koordinaten in $\mathbb{Z}/\tilde{n}\mathbb{Z} \times \mathbb{Z}/\tilde{n}\mathbb{Z} \times \mathbb{Z}/\tilde{n}\mathbb{Z}$ bezeichnet werden. Nun müssen auch hier wie zuvor schon $\tilde{n}$ Prozessoren auf das gleiche Datum zugreifen. Diesmal wird diese implizite Broadcast-Operation durch eine explizite ersetzt, bei der die nötigen Daten direkt im Würfel verteilt werden. Sind dann die Teilmatrizen vorberechnet, wird $c_{ij} := \sum_{k=1}^{\tilde{n}} a_{ik} \cdot b_{kj}$ durch eine Reduktion entlang der k-Dimension bestimmt (vgl. Bild 2.2c). Es sollte hier beachtet werden, dass die k als zusätzliche Dimension ihren Speicherbedarf haben. Durch diesen zusätzlichen Faktor von $\Theta(\sqrt[3]{p})$ auf Platzbedarf, ist der Algorithmus weniger geeignet für große Matrizen, aber in diesem Fall kommt es auch nicht mehr auf Kommunikation an. Gegenüber dem Cannon-Algorithmus kann hier Zeit gespart werden, da die nötigen Daten nicht nach und nach zu einem Prozessor tropfen, sondern auf mehrere Prozessoren verteilt werden. Von diesen Prozessoren wird die Summe dann parallel berechnet, was die Kommunikation reduzieren kann.

Analyse:

$$T_{\text{seq}}(n/\tilde{n}) + 2T_{\text{broadcast}}((n/\tilde{n})^2, \tilde{n}) + T_{\text{reduce}}((n/\tilde{n})^2, \tilde{n}) \approx$$
$$3T_{\text{broadcast}}((n/\tilde{n})^2, \tilde{n}) \overset{\tilde{n}=\sqrt[3]{p}}{\rightsquigarrow} T(n,p) \in O(\frac{n^3}{p} + T_{\text{byte}}\frac{n^2}{p^{2/3}} + T_{\text{start}} \log p)$$

Eine wichtige Erkenntnis, die man aus diesen Algorithmen zieht, ist, dass Probleme der Linearen Algebra mit vollbesetzten Matrizen häufig schnell gelöst werden können, indem Matrizen in Teilmatrizen aufgeteilt werden,

für die lokale Berechnungen ausreichen. Die Kommunikation beschränkt sich dann auf wenige Broadcast- und Reduceoperationen, wo sonst ständig auf fremde Daten zugegriffen werden müsste.

3 Kommunikation

Ein entscheidender Teil der Ausführungszeit von parallelen Algorithmen auf distributed Memory wird für kollektive Kommunikation verbraucht. Ist ein Problem nicht auf triviale Art parallelisierbar, so müssen die Prozessoren irgendwann Daten austauschen. Die verschiedenen Arten kollektiven Nachrichtenaustausches sind Probleme, die so grundlegend sind, dass für sie gleich zu Beginn Algorithmen und Zeitschranken besprochen werden sollen, auf die sich spätere Algorithmen immer wieder beziehen werden. Hier nicht besprochen werden die Sammeloperation Gather und die umgekehrte Scatter-Operation. **Gather** ist verwandt mit der in diesem Kapitel beschriebenen Reduce-Operation. Der Operator wäre dann Konkatenation der Nachrichten, was allerdings bedeutet, dass die Nachrichtenlänge nicht konstant bleibt. Durch diese Änderung stellen sich andere Anforderungen an Algorithmen, die rechtfertigen es als eine eigene Operation zu betrachten. Dementsprechend ist **Scatter** die umgekehrte Operation die eine Nachricht aufteilt und die Teile gleichmäßig über die Prozessoren verteilt.

3.1 Reduktion

Problembeschreibung: Gegeben ist ein assoziativer, aber nicht zwangsläufig kommutativer Operator $\oplus$, der in konstanter Zeit berechnet werden kann. Zu berechnen ist $\oplus_{i<n} x_i := (..((x_0 \oplus x_1) \oplus x_2) \oplus ... \oplus x_{n-1})$.

Algorithmus 3.1: PRAM Binomialbaum-Reduktion

```
active := 1;
for(0 ≤ k < ⌈log n⌉){
  if(active){
    if(bit k of i){
      active := 0;
    }else if (i + 2^k < n){
      x_i := x_i ⊕ x_{i+2^k};
    }
  }
}
```

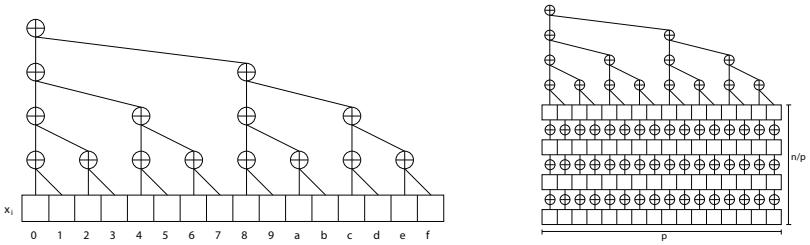
Erläuterungen: Die Grundidee hinter dem Algorithmus ist auf Bild 3.1a durch einen Schaltkreis veranschaulicht, der das gleiche tut. Der Schaltkreis für eine Eingabelänge $n = 2^k$ ist dann gerade ein Binärbaum der Tiefe $k = \log n$. Da der PRAM-Algorithmus gleich funktioniert, ist das auch seine Laufzeit. Die Verallgemeinerung auf beliebige n bedeutet nur einen unvollständigen Teilbaum auf oberster Ebene, macht den gesamten Schaltkreis also nur eine Ebene höher.

Den Namen hat der Algorithmus der Struktur der Beziehungen zwischen den Kommunikationspartnern zu verdanken. Auf Abbildung 3.1c stellen Kanten Kommunikation zwischen zwei Prozesoren im Laufe des Algorithmus dar. Der $\oplus$-Operator markiert die Kommunikation entlang der Kanten im jeweiligen Schritt. Diese Darstellung zeigt, dass die Kommunikationsbeziehungen zwischen Prozessoren die Form eines Binomialbaumes haben.

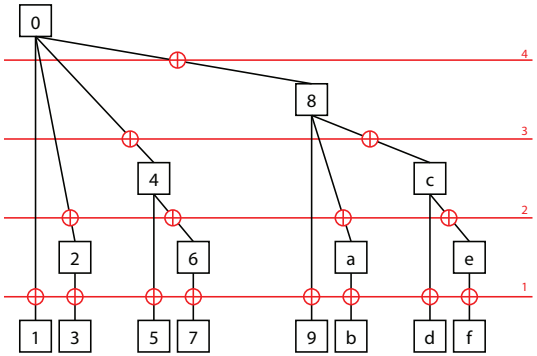
Analyse:

$$\begin{gathered}\text{Für } p = n\text{:} \\ T(n) \in O(\log n) \\ S(n) \in O(n/\log n) \\ E(n) \in O(1/\log n)\end{gathered}$$

Die Effizienz leidet sichtlich darunter, dass nach jedem Schritt die Hälfte der Prozessoren arbeitslos wird und der hohe Bedarf an Prozessoren macht ihn auch nicht praktikabler. Gegen beides kann etwas getan werden. Auch hier gibt es eine Abbildung die die Idee verdeutlichen soll, nämlich Bild 3.1b. In diesem Fall werden die Elemente gleichmäßig über die Prozessoren verteilt, wo jeder Prozessor seinen Teil der Arbeit sequentiell erledigt. Da in diesem sequentiellen Block alle Prozessoren voll ausgelastet sind, hat die Berechnung dort Effizienz 1. Die gesamte Effizienz wird also durch den Anteil



(a) $n = p$ (b) $n > p$



(c) Kommunikationsstruktur

Abbildung 3.1: Binomialbaum-Reduktion

der Binärbaum-Reduktion an der Berechnung bestimmt. Damit ist also für $n \gg \log p$ die Effizienz beinahe 1.

Analyse:

$$\text{Für } p \text{ Prozessoren und } n \geq p\text{:}$$
$$T(n) \in T_{\text{seq}}(n/p) + \Theta(\log p)$$
$$E(n) = \frac{T_{\text{seq}}(n)}{p(T_{\text{seq}}(n/p) + \Theta(\log p))} = \frac{1}{1 + O(\log(p))/n} = 1 - \Theta(\frac{\log p}{n})$$

Bisher wurden nur Fälle betrachtet, in denen alle Prozessoren freien Lesezugriff auf gemeinsamen Speicher hatten. Nun soll untersucht werden, was sich auf einer Distributed Meomory Maschine (DMM) ändert.

Algorithmus 3.2: DMM Binomialbaum-Reduktion

```
active := 1;
s := x_i;
for (k := 0; k < ⌈log n⌉; k++){
  if(active){
    if(bit k of i){
      sync−send s to PE i − 2^k;
      active := 0;
    }else if(i + 2^k < n){
      receive s' from PE i + 2^k;
      s := s ⊕ s';
    }
  }
}
```

Erläuterungen: Hier kommt nun Kommunikation hinzu, eine gute Gelegenheit den Einfluss der Modelle auf die Laufzeit zu untersuchen. Hat man vollständige Verknüpfung, so besteht jeder der $\log p$ Schritts aus Berechnung und einer Kommunikationsoperation, also einem Startup und dem Senden von einem Byte. Zuletzt soll noch das BSP-Modell betrachtet werden. Auf den ersten Blick sieht $\Theta((l+g)\log p)$ nicht anders aus als bei vollständiger Verknüpfung, doch steckt in l ein logarithmischer Faktor, also ist die Laufzeit mindestens $\log^2 p$.

Analyse:

$$\text{Vollständige Verknüpfung: } T(n) \in \Theta((T_{\text{start}} + T_{\text{byte}})\log p)$$
$$\text{Lineares Array: } T(n) \in \Theta(p)$$
$$\text{Lineares Array mit Hardwarerouter: } T(n) \in \Theta((T_{\text{start}} + T_{\text{byte}})\log p)$$
$$\text{BSP: } T(n) \in \Theta((l+g)\log p) = \Omega(\log^2 p)$$

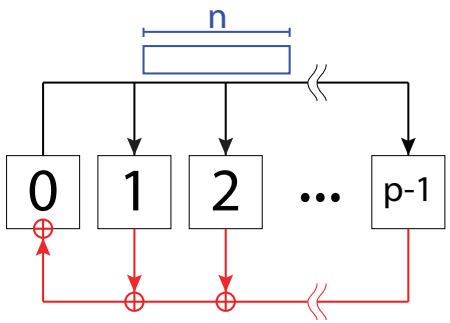


Abbildung 3.2: Broadcast (schwarz) und Reduktion (rot)

3.2 Broadcast

Problembeschreibung: Ein PE (o.B.d.A der erste) sendet eine Nachricht $M[1..n]$ der Länge n an alle anderen.

Es gibt eine enge Beziehung zwischen Broadcast und Reduktion, tatsächlich lassen sich sogar alle Broadcastalgorithmen durch Umkehren der Kommunikationsrichtung als Reduktionsalgorithmen für kommutative Operatoren benutzen. Die meisten Algorithmen, die im Folgenden betrachtet werden, funktionieren dann sogar mit nichtkommutativen Operatoren.

3.2.1 Binomialbaum-Broadcast

Algorithmus 3.3: Binomialbaum-Broadcast

```
if(i > 0) receive M;
for(k := 0; (i >> k) mod 2 = 0; k++){
  if(i + 2^k < p){
    send M to PE i + 2^k;
  }
}
```

Erläuterungen: Den Anfang macht ein Algorithmus, der tatsächlich schon so als Reduktionsalgorithmus betrachtet wurde (3.1). Auch hier soll

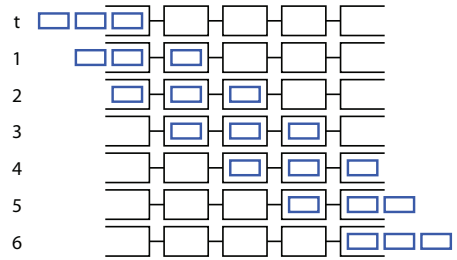


Abbildung 3.3: Broadcast mit Pipeline

die Baumstruktur herhalten um die Vorstellung zu erleichtern. Diesmal beginnt die Broadcastoperation an der Wurzel und die Daten werden entlang der Kanten bis an die Blattknoten propagiert. Bei einer Nachrichtenlänge von 1 ist es auch nicht möglich die Daten schneller an alle Prozessoren zu übermitteln als es hier passiert. Nun ist es allerdings so, dass bei längeren Nachrichten die meisten Prozessoren nichts zu tun haben bis die ganze Nachricht in einem anderen Teil des Baumes übermittelt wurde.

Analyse:

$$T(n,p) = \lceil \log p \rceil \left(T_{\text{start}} + nT_{\text{byte}}\right)$$

3.2.2 Lineare Pipeline

Algorithmus 3.4: Lineare Pipeline

```
piece(j) := M[(j-1)n/k + 1..jn/k];
for(j := 1; j < k+1; j++){
  concurrently{
    if(i = 0 ∨ j = k+1) noop();
    else receive piece(j) from PE i-1;
  }with{
    if(i = p-1 ∨ j = 0) then noop();
    else send piece(j-1) to PE i+1;
  }
}
```

Erläuterungen: Bei diesem Broadcastalgorithmus wird die Nachricht in k Paketen weitergegeben, wobei o.B.d.A. $k|n$ angenommen wird. Wenn das erste Paket angekommen ist, braucht jedes weitere nur noch einen Schritt,

weil die nächste Nachricht schon vor der Tür wartet. Zwar ist die Zeit das erste Paket auszuliefern mit $p-1$ ziemlich hoch, bei (relativ zur Prozessorzahl) sehr langen Nachrichten überwiegt aber, dass die Nachrichtenlänge nicht mehr so stark ins Gewicht fällt. Die Zeit, die die Übermittelung dauert ist $T(n,p,k) = (T_{\text{start}} + \frac{n}{k}T_{\text{byte}})(p+k-2)$ und hängt damit zunächst von der Wahl von k, also der Zahl der Pakete ab. Optimal wird die asymptotische Schranke aber bei

$$k = \sqrt{\frac{n(p-2)T_{\text{byte}}}{T_{\text{start}}}}$$

Bei der Laufzeit, die sich durch diese Wahl ergibt, ist der Wurzelterm gegen die beiden anderen additiven Terme eher klein. Besonders zu beachten ist auch der Term pT_{start}, der sich besonders bei großer Prozessoranzahl bemerkbar macht.

Analyse:

$$T(n,p) \approx nT_{\text{byte}} + pT_{\text{start}} + \sqrt{npT_{\text{start}}T_{\text{byte}}}$$

3.2.3 Binärbaum-Broadcast

Algorithmus 3.5: Pipelined Binary Tree Broadcast

```
piece(j) := M[(j-1)n/k + 1..jn/k];
for(j:=1; j<k; j++){
  if(parent exists)receive piece(j);
  if(left child l exists)send piece(j) to l;
  if(right child r exists)send piece(j) to r;
}
```

Erläuterungen: Die beiden vorherigen Algorithmen zeigen bei extremen Nachrichtenlängen ihre Stärken. Dieser versucht nun beide in einen zu vereinen, der bei allen Nachrichtenlängen vernünftige Ergebnisse liefert. Das Ziel ist es, zu erreichen, dass so viele Knoten wie möglich gleichzeitig ausgelastet sind. Die Nachricht wird wie bei einer Pipeline in k Pakete aufgeteilt (wieder soll o.B.d.A. $k|n$ gelten). Auch in einer Baumstruktur erlaubt eine solche Aufteilung, die Nachricht bereits weiterzupropagieren, bevor sie komplett übermittelt ist. Um möglichst konsistent Daten weiterleiten zu können, wird auch die Baumstruktur angepasst. Fibonacci-Bäume eignen sich gut, da die Pakete nicht synchron an beide Kinder gesendet werden können. Der

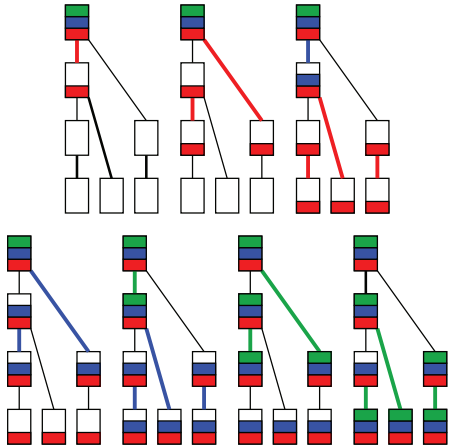


Abbildung 3.4: Pipelined Binary Tree Broadcast

entscheidende Vorteil der Fibonacci-Bäume ist die Invariante, dass von jedem Knoten aus der Höhenunterschied der Teilbäume mit den Kindern als Wurzeln gerade eins beträgt. Der Tiefenunterschied gleicht gerade diese Verzögerung von einem Schritt aus, mit der der rechte Teilbaum das erste Paket bekommt. Anders als beim Binomialbaum-Broadcast werden die Pakete von einer Wurzel auch nur noch an zwei Kinder weitergeleitet, ein Paket abwechselnd an beide. Diese Änderung hat zur Folge dass die Kommunikation nun die Struktur eines Binärbaumes hat.

Ein Kommunikationsschritt, bei dem jeder Prozessor einem der Kinder eine Nachricht sendet, kostet $\frac{n}{k}T_{\text{byte}} + T_{\text{start}}$ Zeit. Eine Schleifeniteration im Algorithmus bei Halbduplex enthält drei dieser Kommunikationsschritte und zwei bei Vollduplex. Es dauert j Schritte bis das erste Paket in Schicht j verteilt ist. Für die Zahl p_j der Prozessoren in einem Baum mit Höhe j gilt:

$$p_j \approx \frac{3\sqrt{5}+5}{5(\sqrt{5}-1)}\Phi^j \approx 1.89\Phi^j \text{ wobei } \Phi = \tfrac{1+\sqrt{5}}{2} \text{ der goldene Schnitt ist.}$$

Damit ist die Tiefe d des Baumes ungefähr $\log_\Phi p$. So lange dauert es also auch, bis das erste Paket alle Prozessoren erreicht hat. Jedes weitere Paket braucht wie zu Beginn festgestellt drei Schritte bei Halbduplex und zwei bei Vollduplex. Für die Gesamtzeit gilt also

$$T(n,p,k) \approx (\tfrac{n}{k}T_{\text{byte}} + T_{\text{start}})(d+3k-3)$$

in Halbduplex bzw.

$$T(n,p,k) \approx (\tfrac{n}{k}T_{\text{byte}} + T_{\text{start}})(d + 2k - 2)$$

in Vollduplex. Dabei wird hier nur der Fall vollständiger Fibonaccibäume betrachtet, für allgemeine p kann aber auch immer der nächstgrößere Baum gewählt und zurechtgeschnitten werden. Die Abhängigkeit von k lässt sich wieder durch Optimieren entfernen:

Halbduplex:

$$k = \sqrt{\frac{n(d-3)T_{\text{byte}}}{3T_{\text{start}}}}$$

Vollduplex:

$$k = \sqrt{\frac{n(d-2)T_{\text{byte}}}{3T_{\text{start}}}}$$

Analyse:

Halbduplex:

$$T(n,p) :\approx 3nT_{\text{byte}} + T_{\text{start}} \log_\Phi p + \sqrt{3n \log_\Phi p T_{\text{start}} T_{\text{byte}}}$$

Vollduplex:

$$T(n,p) :\approx 2nT_{\text{byte}} + T_{\text{start}} \log_\Phi p + \sqrt{2n \log_\Phi p T_{\text{start}} T_{\text{byte}}}$$

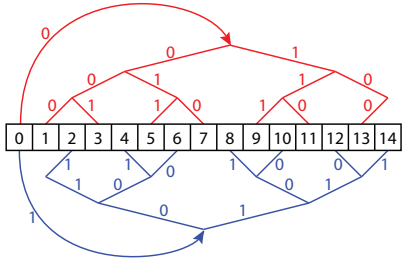
3.2.4 23-Broadcast

Algorithmus 3.6: 23-Broadcast

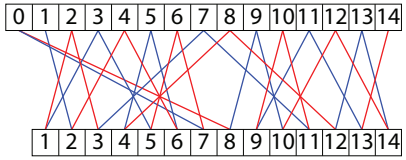
```
if(root process){
  for(j:=1; j<k; j=j+2){
    send piece(j+0) along edge labelled 0;
    send piece(j+1) along edge labelled 1;
  }
}else{
  r := k;
  s := -1;
  while(r>0 ∨ s≥0){
    t := -1;
    concurrently{
      if(receive piece(j) over edge labelled b
      as inner node){
        t := j;
        r := r-1;
      }
      if(receive piece(j') over edge labelled 1-b
      as leaf){
        r := r-1;
      }
    }with{
      if(s≥0){
        send piece(s) along edge labelled b;
        send piece(s) along edge labelled 1-b;
      }
    }
    s := t
  }
}
```


Erläuterungen: Ein Problem, das bisher alle Baumalgorithmen gemeinsam hatten, war, dass auch mit Pipelining die Knoten wenig zu tun hatten. Selbst wenn bereits Pakete im ganzen Baum verteilt waren, konnte ein Knoten nur in jedem zweiten Schritt etwas empfangen und als Blattknoten war das zudem die einzige Tätigkeit. Allerdings ist es so, dass es in einem vollständigen Binärbaum ungefähr genausoviele innere Knoten gibt, wie Blattknoten. Da hat man nun Knoten von denen nur die Hälfte in einem Schritt Pakete empfängt und die Hälfte aller Knoten sind Blattknoten die nichts senden, wieso sollten also die Blattknoten nicht in jedem Schritt Pakete an die Knoten senden, die nichts zu empfangen haben? Konkret wird dies hier realisiert indem man zwei Bäume aufbaut ("two t(h)ree"-Algorithmus). Knoten, die in einem Baum innere Knoten sind, übernehmen im zweiten Baum die Rolle von Blattknoten und umgekehrt. Der Knoten, der die Nachricht zu Beginn hat, ist in den Bäumen nicht enthalten, sondern schickt die Pakete an die beiden Wurzeln. Die Hälfte der k Pakete wird dann über einen Baum gesendet. Die restlichen Pakete werden über den anderen Baum verschickt. Natürlich lässt sich diese Konstruktion auch für Prozessorzahlen verallgemeinern, die keinen vollständigen Binärbaum bilden können. Dafür braucht man nur Knoten so abzuschneiden dass die Bäume abwechselnd kleiner werden, so dass der Größenunterschied stets höchstens 1 ist.

Die Kanten in den Bäumen bekommen jeweils das Label 0 oder 1. Die Labels lassen sich so wählen, dass für jeden Knoten die Eingangskanten in den beiden Bäumen verschiedene Labels haben und auch die Ausgangskanten in dem Baum, in dem der Knoten kein Blatt ist. So kann jeder Prozessor abwechselnd beide 0- und beide 1-Kanten benutzen und so in jedem Schritt eine Nachricht empfangen und eine senden. Da das für alle Prozessoren gilt,



(a) Zwei Bäume



(b) Hilfsgraph zur Bestimmung der Labels

Abbildung 3.5: 23-Broadcast

ist so jeder Prozessor voll ausgelastet, wenn alle Prozessoren in den Bäumen Nachrichten haben und sie über Kanten mit gleichem Label austauschen. Um zu zeigen, dass es eine solche Wahl für die Labels gibt, wollen wir einen Hilfsgraphen konstruieren. In diesem lässt sich diese Wahl gut bestimmen, da sie dort einer Zwei-Färbung der Kanten entspricht. Hierfür werden die Knoten verdoppelt. Jeder, bis auf den Knoten, der die Nachricht verteilt (der ja nur als Sender agiert), ist einmal als Sender und einmal als Empfänger vorhanden. Entsprechend dieser Einteilung der Rollen, gehen die Kanten nun immer zu dem Knoten, dessen Rolle und Nummer mit der des Knotens im Baum übereinstimmt (ein Beispiel findet sich in Abbildung 3.5b). Es gibt zwei Knoten, die jeweils nur ein Kind in ihren Bäumen haben statt zwei Kinder. Dementsprechend haben diese beiden Knoten als einzige Grad 1 als Sender. Da alle anderen Grad 2 haben, existiert ein Pfad zwischen diesen beiden Knoten. Bei diesem Pfad ist es leicht eine Zwei-Färbung zu finden indem man die Kanten abwechselnd mit 0 und 1 kennzeichnet. Entfernt man diesen Pfad, so können nur noch Kreise übrigbleiben. Nun ist der Graph bipartit, also haben sie gerade Länge und können damit ebenfalls in zwei Farben gekennzeichnet werden. Diese Zweifärbung bedeutet, dass an keinem Knoten zwei Kanten mit gleichem Label liegen. Sowohl beim Senden als auch beim Empfangen haben die Kanten also unterschiedliche Labels, was gerade gesucht war.

Nachdem nun klar ist, dass der Algorithmus zumindest immer möglich ist, soll seine Laufzeit betrachtet werden. Es dauert $2j$ Schritte, bis jeder Prozessor in Schicht j mindestens ein Paket erhalten hat. Es gibt insgesamt $d := \lceil \log(p+1) \rceil$ Schichten. Wenn die erste Nachricht im Baum propagiert ist, dauert es 2 Schritte um 2 weitere Pakete an ihr Ziel zu bringen.

$$T(n,p,k) \approx (\tfrac{n}{k} T_{\text{byte}} + T_{\text{start}})(d+k-1)$$

Wieder wollen wir die Abhängigkeit von k durch asymptotisch optimale Wahl beseitigen und wählen für die Analyse

$$k = \sqrt{\frac{n(2d-1)T_{\text{byte}}}{T_{\text{start}}}}$$

Analyse:

$$T(n,p) \approx nT_{\text{byte}} + 2\log p T_{\text{start}} + \sqrt{2n \log p T_{\text{start}} T_{\text{byte}}}$$

3.2.4.1 Implementierungsdetails

Bei der Beschreibung wurde nur die Existenz solcher Bäume und einer geeigneten Färbung gezeigt. Dabei soll es nicht bleiben, denn für eine Implementierung müssen diese Bäume auch schnell bestimmt werden können. Ebenso muss ein Knoten schnell herausfinden können, welche Labels die ein- und ausgehenden Kanten haben.

Von Jochen Speck stammt ein Algorithmus, der es erlaubt für einen Knoten das Label der eingehenden Kante im oberen Baum zu bestimmen. Hat man diese Information für den Knoten selbst und einen Kindknoten, so ist der Rest leicht zu bestimmen.

Algorithmus 3.7: 23 Incoming Edge Color

```
inEdgeColor(p, i, h){
  if(i is root of T_1) return 1;
  while((i bitand 2^h) = 0) h++;
  if((2^{h+1} bitand i = 1) ∨ i + 2^h > p) i' := i − 2^h;
  else i' := i + 2^h;
  return inEdgeColor(p, i', h) xor (p/2 mod 2) xor (i' > i);
}
```

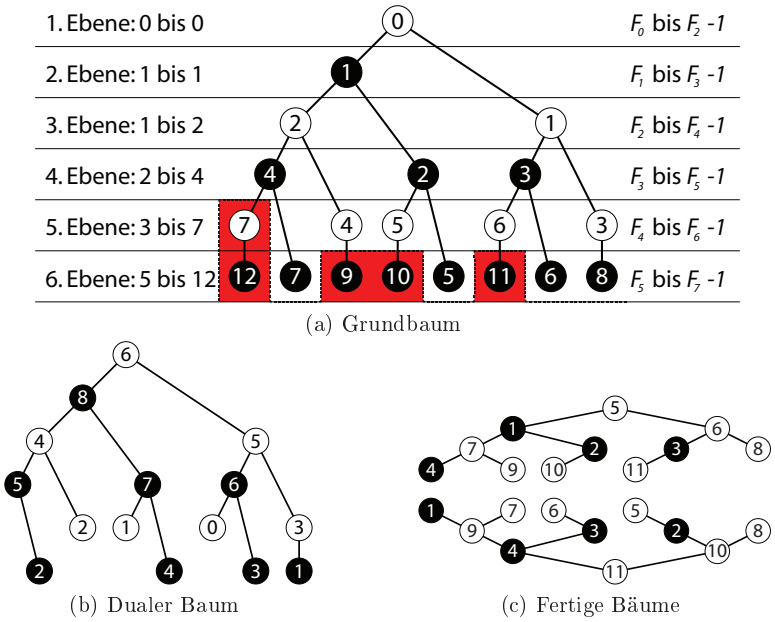
Dieser Algorithmus bestimmt für gegebene Prozessorzahl p das Label der eingehenden Kante von Prozessor i im oberen Baum. Grob nutzt er die Struktur der Färbung aus und steigt rekrusiv im Baum auf um über die eingehende Kante der Elternknoten die eigene Färbung zu bestimmen.

Bisher wurde stets das Duplex-Modell angenommen, was daran liegt, dass es nicht klar ist, ob es überhaupt sinnvoll möglich ist, den Algorithmus direkt zu portieren. Alle Kanten müssten dafür in vier statt zwei Farben gefärbt werden. Ob es hier überhaupt stets möglich ist, ist noch offen. Eine Implementierung mit der doppelten Anzahl der Prozessoren, die dann in vier Schritten zwei Schritte des Duplexmodells simulieren, ist aber möglich.

3.2.4.2 Fibonaccibäume

Der zuvor beschriebene Binärbaum-Algorithmus hat – anders als der 23-Algorithmus – nicht vollständige Binärbäume als Anordnung benutzt, sondern Fibonaccibäume. Von der Struktur würden auch die Bäume im 23-Algorithmus profitieren. Da vollständige Fibonaccibäume aber ein anderes Verhältnis von inneren Knoten zu Blättern haben, müssen sie zunächst geeignet angepasst werden, bevor sie für den 23-Algorithmus benutzt werden können. Aufgebaut werden diese abgeschnittenen Fibonaccibäume in mehreren Schritten. Zunächst soll geklärt werden, wie der Baum aussieht. Im Folgenden soll der größere Teilbaum immer der linke sein. Um die nächsten Schritte zu verdeutlichen, soll auch der Begriff einer Ebene eingeführt werden. Die Wurzel soll in Ebene 1 liegen, von jedem Knoten in Ebene i aus hat der linke Kindknoten Ebene $i+1$ und der rechte Kindknoten Ebene $i+2$. Bei einem vollständigen Baum wären dann also alle Blätter durch diesen Unterschied zwischen den Teilbaumwurzeln in der gleichen Ebene. Desweiteren folgt aus der Struktur des Fibonaccibaumes, dass in Ebene i genau F_i Knoten sind, wobei F_i die i-te Fibonaccizahl ist.

Im ersten Schritt soll dann eine Nummerierung und eine Färbung generiert werden. Die Färbung hat dabei im Wesentlichen den Sinn zu unterscheiden, ob ein Knoten über den oberen Baum Pakete in einem geraden, oder in einem ungeraden Takt bekommt. Es gibt Knoten, die in geraden Takten Pakete über den oberen Baum erhalten und in ungeraden Takten über den unteren Baum, sowie Knoten, bei denen es gerade umgekehrt ist. Diese Aufteilung wird durch die Färbung repräsentiert. Die verschiedenfarbigen Knoten werden hier separat nummeriert, weil aus naheliegenden Gründen kein Knoten in beiden Bäumen verschiedene Farben haben kann. Gefärbt werden die Knoten jeder ungeraden Ebene weiß und die jeder geraden Ebene schwarz. Die Nummerierung erfolgt zeilenweise, die Knoten in Ebene i



(a) Grundbaum

(b) Dualer Baum

(c) Fertige Bäume

Abbildung 3.6: Abgeschnittene 23-Fibonacci-Bäume

werden mit den Zahlen F_{i-1} bis $F_{i+1}-1$ durchnummeriert. Das sind genau $F_i = F_i + F_{i-1} - F_{i-1} = F_{i+1} - F_{i-1}$ Zahlen. Die Zuordnung erfolgt zunächst nach der Ebene des Elternknotens und bei gleicher Ebene nach der Nummer, die der Elternknoten trägt. Es bekommt also ein Knoten, dessen Elternknoten die kleinere Ebenennummer hat, als der eines anderen, eine kleinere Nummer zugewiesen. Ebenso wird bei Elternknoten in der gleichen Ebene der Knoten die kleinere Nummer erhalten, dessen Elternknoten bereits die kleinere Nummer hat. Ein Beispiel für einen auf diese Weise aufgebauten Baum ist in Abbildung 3.6a zu sehen. Invariant ist, dass die Kinder von jedem Knoten gleiche Nummer, aber verschiedene Farbe haben. Auch zu beachten ist, dass in jeder Farbe alle Zahlen, bis zu der größten im Baum vorhandenen, auch vorkommen. Die Indizes lassen sich auch in Abhängigkeit von der Ebene und davon, ob der jeweilige Knoten ein linkes oder ein rechtes Kind ist, bestimmen.

Im zweiten Schritt soll der Baum nun so gekürzt werden, dass es genausoviele Blätter gibt, wie innere Knoten. Dazu werden eine Zahl b und eine Zahl w berechnet. Es bleiben im fertigen Baum nur Knoten erhalten, deren Index kleiner oder gleich b ist, wenn die Knoten schwarz sind und kleiner als

w, wenn sie weiß sind. Die Formel für b und w in Abhängigkeit der Gesamtknotenzahl p soll nur angegeben, aber nicht bewiesen werden:

$$b := \begin{cases} p - 2F_{2i} - 1 & \text{falls } F_{2_i+2} + F_{2i} \leq p < F_{2i+2} + 2F_{2i} \\ \lfloor \frac{1}{2}(p + F_{2i-1} - 1) \rfloor & \text{falls } F_{2_i+2} + 2F_{2i} \leq p < F_{2_i+3} + F_{2i+1} \\ 2F_{2i+1} - 1 & \text{falls } F_{2_i+3} + F_{2i+1} \leq p < F_{2_i+3} + 2F_{2i+1} \\ \lfloor \frac{1}{2}(p - F_{2i} - 1) \rfloor & \text{falls } F_{2_i+3} + 2F_{2i+1} \leq p < F_{2i+4} + F_{2i+2} \end{cases}$$

$$w := \begin{cases} 2F_{2i} + 1 & \text{falls } F_{2_i+2} + F_{2i} \leq p < F_{2i+2} + 2F_{2i} \\ \lfloor \frac{1}{2}(p - F_{2i-1} + 2) \rfloor & \text{falls } F_{2_i+2} + 2F_{2i} \leq p < F_{2_i+3} + F_{2i+1} \\ p - 2F_{2i+1} + 1 & \text{falls } F_{2_i+3} + F_{2i+1} \leq p < F_{2_i+3} + 2F_{2i+1} \\ \lfloor \frac{1}{2}(p + F_{2i} + 2) \rfloor & \text{falls } F_{2_i+3} + 2F_{2i+1} \leq p < F_{2i+4} + F_{2i+2} \end{cases}$$

Wichtig dabei ist, dass durch die zuvor bestimmte Wahl der Farben und Nummern für die Knoten immer eine passende Anzahl von Blattern und inneren Knoten übrigbleibt. Ebenfalls sichergestellt wird, dass für keinen Knoten, der noch im Baum bleibt, der Elternknoten entfernt wird (was ja wegen der unterschiedlichen Farben zu befürchten wäre). Dabei beginnt die Nummerierung für weiß bei 0 und die für schwarz bei 1, daher sind dann b schwarze und w weiße Knoten übrig. Ein Beispiel ist in Abbildung 3.6a zu sehen, wo die Knoten durch eine gestrichelte Line markiert sind, die abgeschnitten werden würden. In diesem Beispiel ist $p = 15$ also $b = 8$ und $w = 7$. Es werden dort somit alle schwarzen Knoten mit Nummer größer als 8 und die weißen mit Nummer größer oder gleich 7 abgeschnitten. Es bleibt ein Baum mit 15 Knoten übrig. Der schwarze Knoten mit Nummer 8 bleibt als achter Knoten im Baum, der weiße Knoten mit der Nummer 7 wird entfernt, weil er wegen der 0 der achte weiße Knoten ist.

Diese Konstruktion ist so gewählt, dass sich im dritten Schritt der duale Baum leicht berechnen lässt. Die Abbildung, die als $f(x) := w - 1 - x$ für weiße Knoten und $f(x) := b + 1 - x$ für schwarze Knoten definiert ist, liefert eine Permutation der Nummern im Baum. Anwendung dieser Abbildung auf alle Knoten des beschriebenen Baumes liefert einen Baum, in dem Blätter und innere Knoten vertauscht sind. Das liegt daran, dass der zuvor definierte Baum die Knoten mit den größten Nummern für jede Farbe in den Blättern hat. Nun da die Bäume aufgebaut sind, muss nur noch die fertige Nummerierung erfolgen, die nur noch daraus besteht, dass auf die Nummern aller weißen Knoten die Anzahl b der schwarzen Knoten addiert wird. Ein nach diesem Prinzip aufgebauter Baum ist in Abbildung 3.6c dargestellt. Bei der tatsächlichen Umsetzung wird der Prozessor, der zu Beginn die Daten hat, herausgenommen und die Bäume werden nur mit den anderen Prozessoren

aufgebaut. Die Pakete werden dann abwechselnd an die Wurzeln der beiden Bäume gesendet.

3.2.4.3 23-Reduktion

Wie jeder Broadcastalgorithmus lässt sich der 23-Broadcast auch für Reduktion verwenden. Will man jedoch vom Pipelining profitieren, so muss die bei der Reduktion vewendete Operation auf einzelnen Paketen definiert sein. Das ist beispielsweise der Fall, wenn in jedem Paket Elemente eines d-dimensionalen Vektors übertragen werden wobei $k|d$ gelten sollte. Dann können tatsächlich beide Teilbäume parallel eine leicht abgeänderte Form der Binomialbaum-Reduktion durchführen. Abgeändert insofern als dass jeder Prozessor nun das eigene Element mit denen der zwei Kinder verknüpft und dann nach oben weiterreicht. Um die Kommutativität zu erhalten müssen die Bäume dann auch so gewählt werden, dass die Prozessor-Indizes gerade einer In-Order-Nummerierung in beiden Bäumen entsprechen (das ist beispielsweise in Abbildung 3.5a der Fall).

3.2.5 ESBT-Broadcast

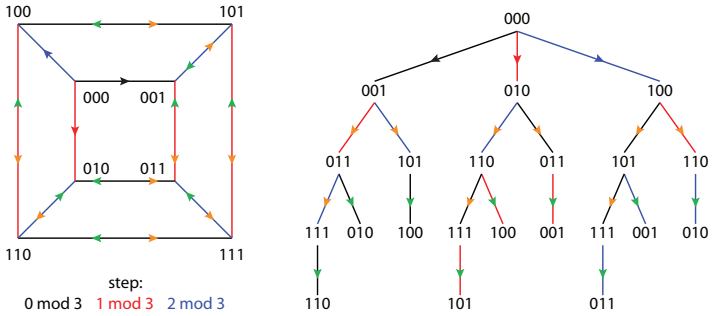


Abbildung 3.7: ESBT-Broadcast

Der hier nur in Grundzügen beschriebene Algorithmus ist – wie der vorherige – optimal, verlangt aber Hypercube als Netzwerk. Statt Vollduplex funktioniert er auch im Telefonmodell, Halbduplex bringt aber weiterhin einen Faktor 2 ein.

Er funktioniert, indem d Binomialbäume in den d-dimensionalen Hyperwürfel (minus dem 0-Knoten) eingebettet werden. In jedem Schritt wird dazu

ein Paket von jedem Knoten aus in eine andere der d verschiedenen Dimensionen verschickt, wenn es dort noch nicht angelangt ist. Sieht man die Hin- und Rückrichtung zwischen zwei Knoten als verschiedene Kanten an, so zeigt sich, dass so Binomialbäume entstehen, deren Wurzeln die d Nachbarknoten des 0-Knotens des Hyperwürfels sind. Sie haben in diesem Fall auch keine gemeinsamen Kanten, weswegen sie als **E**dge-disjoint **S**panning **B**inomial **T**rees bezeichnet werden. Als Beispiel ist in Abbildung 3.7 der ESBT-Graph für Dimension 3 dargestellt. Da die Nachrichten in jedem Schritt nur über Kanten in eine Richtung gesendet werden, gibt es höchstens eine eingehende und eine ausgehende Nachricht, wenn zwei in dieser Richtung benachbarte Prozessoren ihre unterschiedlichen Nachrichten austauschen. Insgesamt können also Pakete die Bäume unabhängig voneinander ungehindert traversieren.

Wenn man also k Pakete hat (wie immer soll $k|n$ gelten), so wird in jedem Schritt ein Paket an eine der Wurzeln nach dem Round-Robin-Prinzip geschickt. Wenn eine Wurzel eines der Pakete hat, so braucht der Baum d Schritte bis das Paket alle Blätter erreicht hat. Da das auch für den Baum gilt, der nach k Schritten als letzter ein Paket erhält und die Bäume einander nicht in die Quere kommen, ist nach $k+d$ Schritten die Nachricht im Würfel verteilt. Die Zeit ist also

$$T(n,p,k) = (k+d)\left(T_{\text{start}} + \tfrac{n}{k}T_{\text{byte}}\right)$$

Wie für alle paketbasierten Algorithmen lässt sich auch hier k so wählen, dass es für die Analyse optimal ist:

$$k = \sqrt{\frac{ndT_{\text{byte}}}{T_{\text{start}}}}$$

Analyse:

$$T(n,p) = nT_{\text{byte}} + dT_{\text{start}} + \sqrt{ndT_{\text{start}}T_{\text{byte}}}$$

Da schon einige Algorithmen für Broadcast eingeführt wurden, lohnt sich an dieser Stelle ein Überblick, um herauszufinden, wann welcher Algorithmus sinnvoll ist und weshalb man selbst einen eigenen Algorithmus implementieren sollte. Beim letzten Punkt sollte man sich bewusst sein, dass viele Bibliotheken kein Pipelining in ihren Implementierungen kollektiver Operationen nutzen. Spielen kollektive Operationen eine wichtige Rolle für die Laufzeit des Algortihmus, lohnt sich eine eigene Implementierung oft.

In der Realität sind Parameter, beispielsweise die Anzahl der Pakete k auch nicht so leicht zu wählen, wie es bei der Abschätzung noch getan wird.

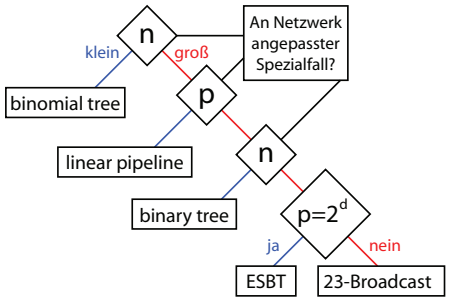


Abbildung 3.8: Übersicht über die Broadcastalgorithmen

Ganz abgesehen von der Rundung, wurde zuvor k in Abhängigkeit von T_{byte} gewählt, doch dieses muss für eine Maschine nicht konstant sein. Oft wird sogar abhängig von der Nachrichtenlänge zwischen Protokollen umgeschaltet, weshalb lineare Abhängigkeit eine Vereinfachung ist. Eine weitere Vereinfachende Annahme, die zuvor getroffen wurde war, dass Senden und Empfangen beide Partner gleich viel Zeit kostet. Wenn Empfangen länger dauert kann in einem Fibonaccibaum der Verzweigungsgrad erhöht werden. Damit soll sichergestellt werden, dass alle Pakete noch immer ungefähr gleichzeitig die Blätter erreichen.

Nun bleibt noch die Frage welcher Algorithmus sich in welcher Situation lohnt. Wie bereits bei der Beschreibung der beiden Algorithmen erwähnt, ist für kurze Nachrichtenlängen der Binomialbaum besonders schnell; ebenso lineare Pipeline für wenige Prozessoren aber sehr lange Nachrichten. In diesen Extremfällen profitieren die Algorithmen davon, dass sie relativ geringen Overhead haben und vergleichsweise einfach zu implementieren sind. Kennt man die Anforderungen im Voraus nicht, beispielsweise weil der Algorithmus für eine Bibliothek bestimmt ist, so bietet sich der 23-Broadcast an, da er in allen Fällen solide Ergebnisse liefert. Der Binärbaum-Broadcast ist bei langen Nachrichten schwächer als 23-Broadcast, bei sehr kurzen ist Binomialbaum schneller. Zwar ist das Intervall in dem der Binärbaum-Algorithmen gegenüber beiden im Vorteil ist eher klein, doch nimmt der Zeitaufwand für den Binomialbaum-Algorithmus mit der Nachrichtenlänge sehr schnell zu, so dass sich im Zwifelsfall der Binärbaum-Broadcast eher lohnt. Eine Übersicht über alle betrachteten Algorithmen findet sich in Abbildung 3.8.

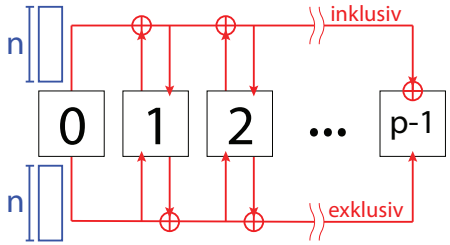


Abbildung 3.9: Präfixsumme

3.3 Präfixsumme

Problembeschreibung: Gesucht ist für jeden Prozessor i die Präfixsumme $x@i := \oplus_{i' \leq i} m@i$, das heißt jeder Prozessor erhält die Summe über die Elemente m der Prozessoren mit kleinerem Index. Die aufsummierten Elemente m können auch Vektoren von n Bytes Länge sein. Es ist von exklusiver Präfixsumme die Rede wenn das eigene Element nicht berücksichtigt wird, die Präfixsumme heißt inklusiv, wenn auch das eigene Element in die Summe mit aufgenommen wird. Der Unterschied ist für Algorithmen jedoch unerheblich.

Wie bereits bei Broadcastalgorithmen bietet sich für große n und kleine p wieder eine **Pipeline** (3.4) an. Die Prozessoren initiieren dann einen Ergebnisvektor mit ihren Daten und addieren die ankommenden Pakete bevor sie den entsprechenden Abschnitt weiterleiten. Algorithmen für Präfixsummen - wie auch Reduktionsalgorithmen - profitieren sogar stärker von Pipelining als Broadcastalgorithmen. Wie bei Broadcast erhält man aber einen pT_{start}-Term in der Laufzeit.

3.3.0.1 Hyperwürfel

Algorithmus 3.8: Hypercube Prefix Sum

```
x := m;
σ := m;
for(k:=0; k < d-1; k++){
  y := σ@(i ⊕ 2^k);
  σ := σ ⊕ y;
  if(bit k of i)x := x ⊕ y;
}
```

Erläuterungen: Der Algorithmus lässt sich motivieren indem man sich

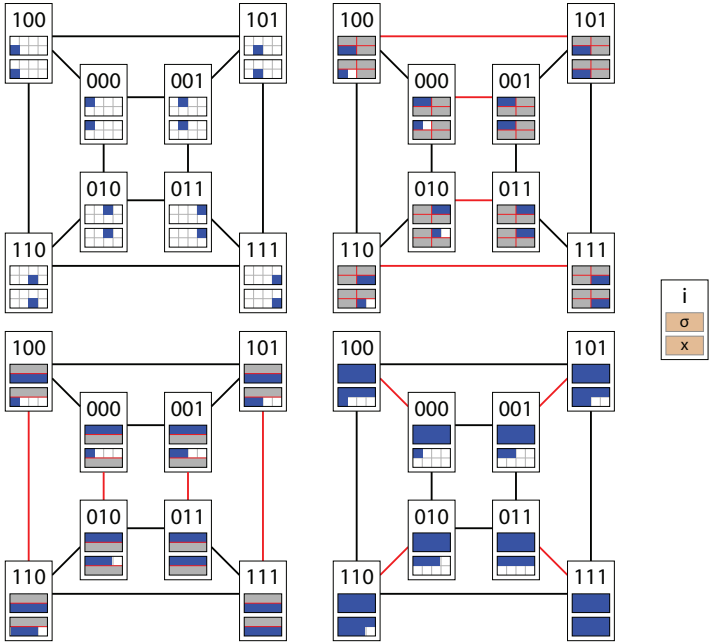


Abbildung 3.10: Hypercube-Präfixsumme

eine rekursive Bestimmung der Präfixsumme in Hyperwürfeln wie folgt überlegt. Ein Hyperwürfel der Dimension d kann in zwei Hyperwürfel der Dimension $d-1$ zerlegt werden, wobei ein Teilgraph dann aus allen Knoten besteht deren Index mit 0 beginnt und der andere aus allen mit einer 1 am Anfang. Diese beiden sollen hier 0-Teilwürfel bzw. 1-Teilwürfel heißen. Ist die Präfixsumme in beiden Teilwürfeln als unabhängigen Teilgraphen für alle Knoten bekannt, so fehlt dem 1-Würfel noch die Summe über alle Elemente im 0-Teilwürfel. Da alle Knoten im 0-Würfel im vollständigen Hyperwürfel kleineren Rang haben als jeder Knoten im 1-Würfel, reicht es die Gesamtsumme im 0-Würfel zu kennen und es wird nicht einmal Kommutativität von $\oplus$ verlangt. Es reicht also, wenn im 0-Würfel für alle Knoten die eigene Präfixsumme und die Gesamtsumme im Teilwürfel bekannt ist, damit alle Knoten im 1-Würfel jeweils vom Knoten mit entsprechendem Index die Gesamtsumme beziehen können. Kennen die Knoten im 1-Würfel auch die Gesamtsumme in ihrem Teilwürfel können sie auch die Teilsummen mit den 0-Knoten austauschen und so kennen alle Knoten die Gesamtsumme im Würfel. Daraus wird klar, dass der Algorithmus sich rekursiv ausführen lässt bis die Teilwürfel jeweils aus zwei Knoten bestehen, für die sich die Präfixsummen und die Teilwürfelsummen leicht bestimmen lassen.

Der oben aufgelistete Algorithmus funktioniert nach genau diesem Prinzip, indem er bei eindimensionalen Würfeln beginnt und induktiv die Summe für den d-dimensionalen Würfel nach dem beschriebenen Verfahren konstruiert. Ein Beispiel findet sich in Abbildung 3.10 Bei der Laufzeit fällt positiv die Verbesserung von p auf $\log p$ als Faktor für T_{start} auf. Auf der anderen Seite steht ein Faktor von $n \log p$ auf T_{byte}. Dieser Term sich auch nicht durch Pipelining drücken, da alle Prozessoren stets beschäftigt sind.

Analyse:

$$T(n,p) = (T_{\text{start}} + nT_{\text{byte}}) \log p$$

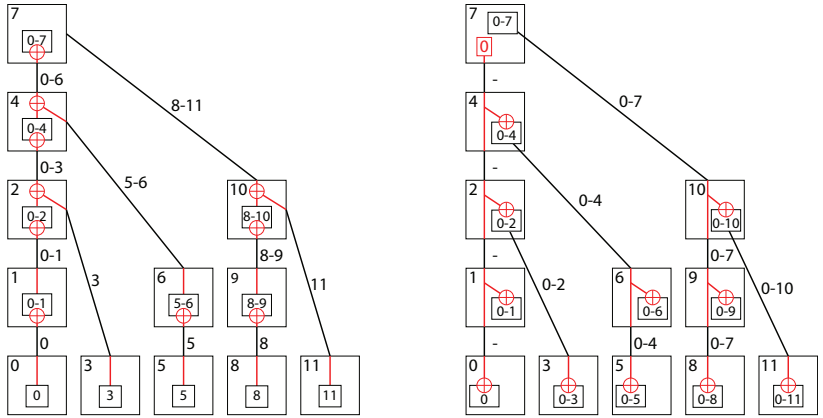
3.3.0.2 Pipeline-Binärbaum-Präfixsumme

Algorithmus 3.9: Pipelined Binary Tree Prefix Sum

```
k := 0;
for(k := 0 to n-1){
  receive m_k from left child;
  x_k := m_k ⊕ x_k;
  receive m_k from right child;
  send m_k ⊕ x_k to parent;
}
for(k := 0; k < n-1; k++){
  m_k := 0;
  if(parent exists){
    receive m_k from parent;
  }
  send m_k to left child;
  send m_k ⊕ x_k to right child;
}
```


Erläuterungen: Der Algorithmus setzt zunächst voraus, dass die Prozessoren in einem Fibonacci-Baum angeordnet sind und zusätzlich infix-Nummerierung vorliegt. Das bedeutet insbesondere, dass von jedem Knoten aus im linken Teilbaum nur Prozessoren mit kleinerem Index sind und im rechten nur Prozessoren mit größerem Index. Für die Präfixsumme bedeutet das zunächst dass ein Prozessor alle Elemente aus dem linken Teilbaum benötigt. Dazu muss jeder Knoten die Summe beider Teilbäume mit dem eigenen Element nach oben weiterreichen. Selbst behalten muss er nur die Summe von linkem Teilbaum und dem eigenen Element. Diese Berechnung entspricht der ersten Phase der Algorithmus, der "Aufwärtsphase" (siehe auch Abbildung 3.11a). Jedem rechten Teilbaum fehlt zu diesem Zeitpunkt das Wissen über die Elemente im linken Teilbaum. Da der Index aller Prozessoren dort kleiner ist, reicht bereits die Summe, die in der Wurzel beider Teilbäume als Ergebnis der Aufwärtsphase gespeichert ist. Diese Phase, in der jeder Prozessor in beiden Teibäumen die noch fehlende Teilsumme weiterpropagert, die er vom Elternknoten erhält und für seinen rechten Teilknoten noch die gespeicherte Summe im linken Teilbaum aufaddiert, wird hier nach der Richtung des Informationsflusses als "Abwärtsphase" bezeichnet (siehe auch Abbildung 3.11b).

Teilweise lassen sich die Phasen auch überlappen. In der Aufwärtsphase werden zwei Pakete von den Kindern empfangen und eines gesendet, was zwei Schritte im Duplex-Modell kostet. In der Abwärtsphase wird ein Paket empfangen und dann werden zwei an die beiden Kinder gesendet, auch wieder in zwei Schritten. Die beiden Phasen hintereinander dauern damit vier Schritte. Sind die benötigten Pakete an den Nachbarknoten vorhanden, reichen für die zwei Phasen drei Kommunikationsschritte und es wird sogar nur das Telefonmodell verlangt. In einem Schritt kann ein Paket nach oben gesendet werden und ein Paket aus einem früheren Teil der Abwärtsphase empfangen. Das empfangene Paket kann in den folgenden Schritten an die beiden Kinder gesendet und parallel werden die Pakete aus der aktuellen Aufwärtsphase empfangen.



(a) Aufwärtsphase (b) Abwärtsphase

Abbildung 3.11: Pipeline-Binärbaum-Präfixsumme

Die von anderen Baumalgorithmen bekannte geringe Auslastung der Blätter bleibt bei der Überlappung allerdings erhalten. Bei genauem Hinsehen stellt man schnell fest, dass die Aufwärtsphase einer modifizierten Reduktionsoperation entspricht und ebenso die Abwärtsphase einem modifizierten Broadcast. Bei beiden kann wie bereits festgestellt durch den 23-Broadcastalgorithmus eine Verbesserung der Effizienz erreicht werden. So kann auch dieser Algorithmus leicht angepasst in den beiden Bäumen parallel auf Teildaten ausgeführt werden. Auch hier soll der Aufbau der Bäume Infix-Nummerierung sicherstellen, da sie im zuvor beschriebenen Algorithmus eine wichtige Rolle spielt.

Analyse:

$$T(n,p) \approx T_{\text{reduce}} + T_{\text{broadcast}} \approx 2T_{\text{broadcast}} = 2nT_{\text{byte}} + 4\log pT_{\text{start}} + \sqrt{8n\log pT_{\text{start}}T_{\text{byte}}}$$

3.4 MCSTL: Präfixsumme

Ein Algorithmus für Shared Memory ist in der MCSTL als `partial_sum` vorhanden. Die n Elemente werden für die Präfixsummenberechnung zunächst in $p+1$ Blöcke aufgeteilt. Auf den ersten p Blöcken berechnet jeweils ein

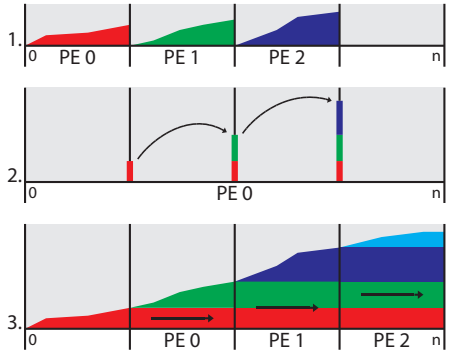


Abbildung 3.12: Beispiel für MCSTL-Algorithmus: `partial_sum` mit 3 Prozessoren

Prozessor die lokale Präfixsumme. Im folgenden Schritt berechnet der erste Prozessor sequentiell die Präfixsumme über die letzten Elemente dieser Blöcke. So ist für jeden Block auch die Summe über die vorhergehenden Blöcke bekannt. Es wird sequentiell gerechnet, da für wenige Prozessoren der Synchronisationsaufwand für eine baumförmige Berechnung nicht gerechtfertigt ist. Das Haupteinstzgebiet von MCSTL sind aber Rechner mit relativ wenigen Prozessoren. Bei einer baumförmigen Präfixsumme über die Blöcke, würde man zusätzlich zu der Synchronisation am Anfang und am Ende noch $\log p$ weitere im Laufe der Berechnung brauchen.

Nach dem zweiten Schritt ist für jeden Block der Offset bekannt und es können für die letzten p Blöcke die Ergebniswerte aus den lokalen Werten und dem Offset berechnet werden. Eine Aufteilung in $p+1$ Blöcke spart überflüssge Berechnungen. Im ersten Schritt wird die Summe zunächst für die Offsetberechnung benötigt. Da der letzte Prozessor keinen Nachfolger hat, ist es nicht nötig auch für ihn in diesem Schritt Berechnungen durchzuführen. Umgekehrt addieren die Prozessoren im dritten Schritt den Blockoffset auf die einzelnen Werte. Das ist für den ersten Block nicht nötig, da die finalen Ergebnisse bereits im ersten Schritt bestimmt wurden.

Insgesamt sind für die drei Schritte auch genauso viele Synchronisationsoperationen nach den Schritten nötig. In den Schritten müssen zwei sequentielle Summen über $n/(p+1)$ und eine über p Elemente berechnet werden. Insgesamt ergibt sich also eine Laufzeit, die in $O(n/p+p)$ liegt.

Analyse:

$$T(n,p) \in O(\frac{n}{p} + p)$$

3.5 Gossip/All Reduce

Problembeschreibung: Der Prozessor i hat zu beginn nur eine Nachricht m_i der Länge n. Nach der **Gossip-Operation** soll jeder Prozessor alle Nachrichten $x := m_0 \cdot \ldots \cdot m_{p-1}$ besitzen.

Algorithmus 3.10: Hypercube-Gossip

```
x := m_i;
for(j := 0; j < d; j++){
  x' := x@(i ⊕ 2^j);
  x := x · x';
}
```

Erläuterungen: Die Idee hinter dem Algorithmus ist an dieser Stelle bereits aus dem Hypercube-Präfixsummenalgorithmus (3.8) bekannt. Es tauschen zunächst benachbarte Prozessoren in den eindimensionalen Teilwürfeln (benachbart bezüglich Abstand 1 in der niedrigstwertigsten Dimension) ihre Nachrichten aus, so dass die Nachrichtenpaare in diesen Würfeln beiden Prozessoren als x bekannt sind. Paare von eindimensionalen Würfeln tauschen dann Nachrichten aus, so dass die vier beteiligten Prozessoren die Nachricht in ihrem Teilwürfel kennen. Analog wird das Verfahren fortgesetzt, bis alle Nachrichten im ganzen d-dimensionalen Würfel bekannt sind. Zu beachten ist hier, dass die Nachrichten konkateniert werden, sich also die Länge der Nachrichten, die zu übertragen sind, in jedem Schritt verdoppelt.

Analyse:

$$T(n,p) \approx \sum_{j=0}^{d-1} (T_{\text{start}} + n \cdot 2^j T_{\text{byte}}) = \log p T_{\text{start}} + (p-1) n T_{\text{byte}}$$

All-Reduce unterscheidet sich von Gossip nur darin, dass die Nachrichten nicht konkateniert werden sollen, sondern wie bei Reduce ein Operator auf zwei Nachrichten angewandt wird. Es ist damit eine Reduce-Operation deren

Ergebnis jedem Prozessor zur Verfügung steht. Umsetzen lässt sich diese Operation durch eine Reduktion mit anschließendem Broadcast des Ergebnisses. In einem Hyperwürfel lässt sich der für Gossip beschriebene Algorithmus anpassen und profitiert von der konstanten Nachrichtenlänge. Das halbiert die Anzahl der Kommunikationsschritte gegenüber Reduce und Broadcast. Der Nachteil ist dann, dass die Prozessoren auch die Zwischenergebnisse mehrfach berechnen. Das führt zu mehr Nachrichten im Netzwerk, da die Daten auch mehreren Prozessoren vorliegen müssen. Stauanfällige Netzwerke können also durch eine Implementierung mit Reduce/Broadcast entlastet werden, da beim Hypercube-All-Reduce Leitungen viel häufiger benutzt werden.

3.6 All-to-All

Problembeschreibung: Jeder Prozessor hat $p-1$ Nachrichten, eine für jeden Anderen Prozessor. Als $m[j]$ wird dabei die lokale Nachricht bezeichnet, die für den Prozessor mit Index j bestimmt ist.

3.6.1 All-to-All im Hyperwürfel

Algorithmus 3.11: Hypercube All-to-All

```
for(j := d-2; j ≥ 0; j--){
  receive from PE i ⊕ 2^j
    where target in my j–dim subcube;
  send messages to PE i ⊕ 2^j
    where target in its j–dim subcube;
}
```

Erläuterungen: Dass dieser Algorithmus aus [4] funktioniert, ist offensichtlich, da in jedem Schritt Nachrichten eine Dimension näher zu ihrem Ziel kommen, wenn sie nicht bereits angekommen sind. Daraus folgt auch dass höchstens d also $\log p$ Schritte benötigt werden. Bleibt dann nur noch die Frage, wie viele Nachrichten in jedem Schritt übermittelt werden müssen. In jedem Schritt des Algorithmus werden gerade $p/2$ Nachrichten verschickt. Für den ersten Schritt folgt das direkt aus der Tatsache, dass für jeden Prozessor die Hälfte der Ziele von Nachrichten nicht im eigenen Teilwürfel liegt. Im in jedem folgenden Schritt hat ein Prozessor nur noch Daten für einen Würfel zu verschicken, der halb so groß ist wie der vorherige, also auch nur noch halb so viele Nachrichten bekommen soll. Dafür aber hat ihm im vor-

herigen Schritt ein Prozessor genausoviele Nachrichten für diesen Teilwürfel geschickt, wie er bereits selbst hielt. Für den Algorithmus sprechen die logarithmisch vielen Startups. Weniger gut ist ein Faktor von $\log p/2$ auf die Untergrenze von $(p-1)n$ Bytes die gesendet werden müssen. Bei vollständiger Verknüpfung und großem n sollten deswegen die Nachrichten lieber einzeln verschickt werden, um den $\log p$-Faktor zu sparen, auch wenn dann jede Nachricht einen zusätzlichen Startup bedeutet.

Analyse:

$$T(n,p) \approx \log p(T_{\text{start}} + \frac{p}{2} n T_{\text{byte}})$$

3.6.2 1-Faktor-Algorithmus

Algorithmus 3.12: 1-Factor

```
for(j := 0; j < p-1; j++){
  idle := (p/2)j mod (p-1);
  if ((i = p-1) ∧ (2|p)){
    exchange data with PE idle;
  }else{
    if(i = idle){
      if(2|p){
        exchange data with PE (p-1);
      }
    }else{
      if(2|p){
        exchange data with PE ((j-i) mod (p-1));
      }else{
        exchange data with PE ((j-i) mod p);
      }
    }
  }
}
```

Erläuterungen: Man stelle sich für diesen Algorithmus die Prozessoren als $\mathbb{Z}/p\mathbb{Z}$ vor, also im Kreis angeordnet. Zunächst soll der Partner durch eine Spiegelungsabbildung bestimmt werden, es wird also Prozessor 0 sich selbst als Partner zugeordnet, Prozessor 1 kommuniziert mit $p-1$, 2 mit $p-2$, usw. Damit auch jede Paarung einmal vorkommt, wird in Schritt j nicht nur die Spiegelung alleine durchgeführt, sondern zusätzlich j Rotatio-

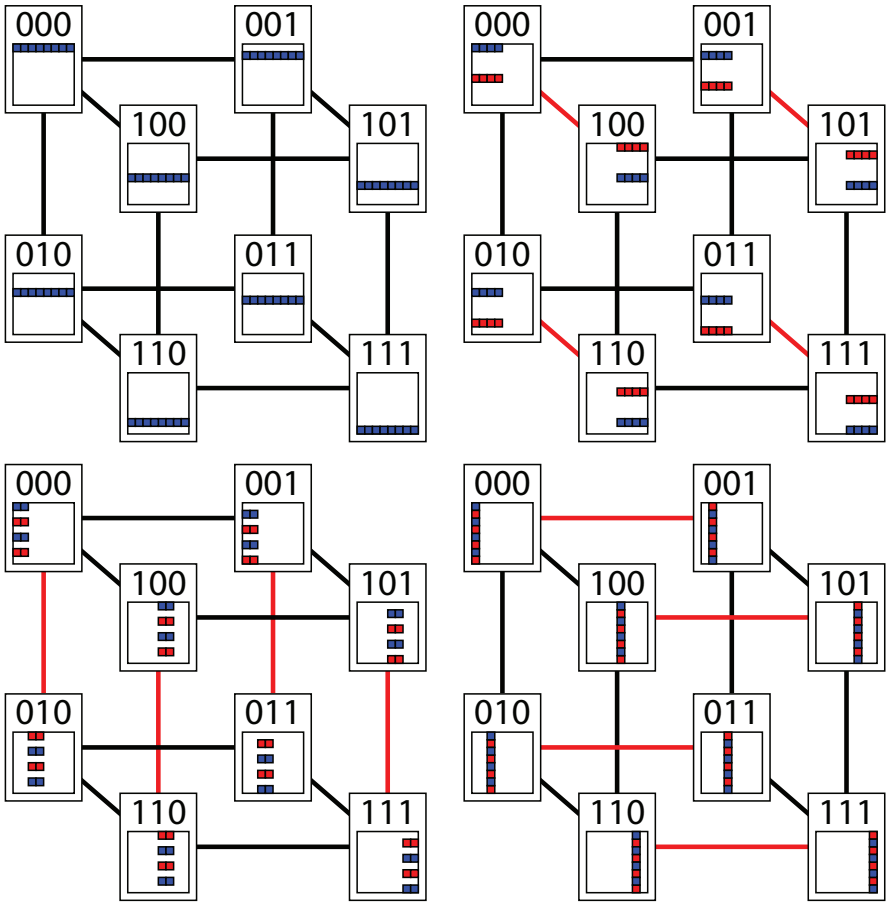


Abbildung 3.13: All-To-All im Hyperwürfel

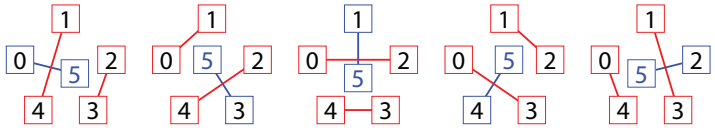


Abbildung 3.14: 1-Faktor-Algorithmus

nen um den Partner zu bestimmen. Das bedeutet dass für einen Prozessor mit dem durch die Spiegelung bestimmten Partner begonnen wird und in den folgenden Schritten auch alle anderen der Reihe nach abgearbeitet werden. Dass die Partnerschaft beidseitig ist, wird dadurch garantiert dass in der Diedergruppe eine Spiegelung mit j-Facher anschließender Drehung zu sich selbst invers ist. Nicht ganz elegant ist hier noch, dass bei gerader Prozessorzahl in jedem zweiten Schritt jeweils zwei Prozessoren sich selbst zugeordnet werden würden. Statt diesen Fall abzufangen, indem die beiden Prozessoren dann einander als Partner zugeordnet werden, wird der letzte Prozessor ausgeschlossen. Auf den übrigen Prozessoren wird das Verfahren für eine ungerade Anzahl von Prozessoren angewandt. Da nun in jedem Schritt genau ein Prozessor sich selbst als Partner zugewiesen bekommt, kann er mit dem ausgeschlossenen Prozessor Daten austauschen.

Gegenüber dem Hypercube-Algorithmus erhöht sich die Anzahl der Startups, da die Nachrichten einzeln verschickt werden. Ist aber das n groß, also insbesondere die Zeit für den Kommunikationsaufbau vernachlässigbar gegen die Zeit für das Versenden, profitiert der Algorithmus davon, dass er die Nachrichten direkt an den Zielprozessor sendet. Dann ist er auch optimal, weil mehr direkte Kommunikation nicht möglich ist und indirekte Kommunikation zwar Startups spart, aber für jedes T_{start} zusätzliche nT_{byte} Zeitaufwand bedeutet.

Analyse:

$$T(n,p) = p(T_{\text{start}} + nT_{\text{byte}})$$

3.6.3 Hierarchial Factor-Algorithm

Der folgende Abschnitt beschäftigt sich mit einer modifizierten Form des Problems. Motiviert wird das Problem durch Computercluster, die aus mehreren vernetzten Rechnern (Knoten/Nodes) bestehen, von denen jeder über mehrere Prozessoren verfügen kann. Betrachtet man single-ported-Kommunikation, so können nicht mehrere Prozessoren in einem Knoten gleichzeitig Daten mit

Prozessoren in anderen Knoten austauschen. Zusätzlich ist es durchaus üblich dass Cluster aus verschiedenen Rechnern aufgebaut sind, die insbesondere verschieden viele Prozessoren haben können. Die Anzahl der Knoten im Cluster wird im folgenden mit P bezeichnet, mit $|U|$ die Anzahl Prozessoren in Knoten U. Der Algorithmus stammt von Sanders und Träff [13].

Algorithmus 3.13: Hierarchial Factor-Algorithm

```
A := {Node_0, ..., Node_{P-1}};
done := 0;
while(A ≠ ∅){
  //phase
  current := min{|U| | U ∈ A};
  for j = 0 to |A| − 1 do;
    //round
    parallel_foreach(U ⪯ V ∈ G_A^j){
      foreach(u ∈ U, done ≤ l(u) < current){
        foreach(v ∈ V){
          //step
          exchange data between u and v;
        }
      }
    }
  done := current;
  A := A \ {U | |U| = current};
}
```

Erläuterungen: Im Algorithmus werden noch Terme verwendet, die zunächst definiert werden sollen. Zunächst ist da die Vorgängerrelation $\preceq$ die Definiert wird durch

$$U \preceq V := |U| \leq |V| \vee (|U| = |V| \wedge \text{index}(U) \leq \text{index}(V))$$

Es ist also eine Ordnung, die Knoten nach Anzahl der Prozessoren sortiert und bei gleicher Anzahl die durch den Index vorgegebene Ordnung beibehält. Für den 1-Faktor-Algorithmus wurde ein Verfahren definiert Paarungen von Prozessoren für den Datenaustausch zu wählen. Solche Paarungen verallgemeinert für eine Menge A werden hier durch G_A bezeichnet. Die Menge der Paarungen im j-ten der $|A|$ Schritte des 1-Faktor-Algorithmus ist dann G_A^j. Zuletzt wird mit $l(u)$ der Lokale Index eines Prozessors u in seinem Knoten bezeichnet.

Zugunsten der Übersichtlichkeit wird der Algorithmus hier in Phasen, Runden und Schritte aufgeteilt. Die Operation in jeder Phase wird dadurch

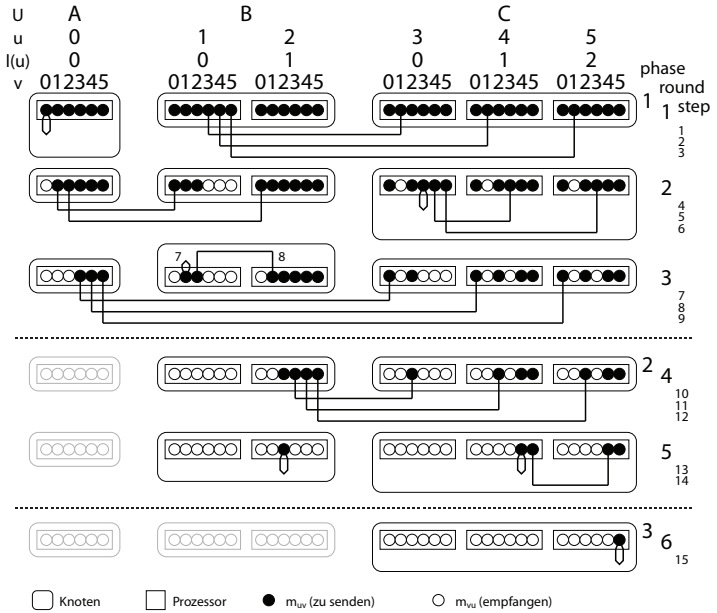


Abbildung 3.15: Hierarchial Factor-Algorithm

bestimmt, dass die Knoten mit der kleinsten Anzahl m Prozessoren zum Schluss der Phase den Datenaustausch abgeschlossen haben sollen. Die gleiche Anzahl der Prozessoren in jedem Knoten soll ebenfalls ihre Daten mit allen Prozessoren von bezüglich der Ordnung $\preceq$ größeren Knoten ausgetauscht haben. Die Daten, die dann für diese Prozessoren noch auszutauschen sind, werden übertragen, wenn der jeweilige Prozessor in einem bezüglich $\preceq$ kleineren Knoten an der Reihe ist. Welche Knoten miteinander kommunizieren legt die Runde fest. In der j-ten Runde wird dazu die Menge der Paarungen G_A^j betrachtet. Aus dem Knoten einer Paarung, der bezüglich $\preceq$ kleiner ist werden die m Prozessoren mit kleinstem lokalen Rang gewählt und kommunizieren mit allen Prozessoren des Partnerknotens. Jede dieser Kommunikationen wird als Schritt bezeichnet. Dadurch ist sichegestellt, dass nach allen $|A|$ Runden einer Paarung die ersten m Prozessoren in jedem Knoten mit allen bezüglich $\preceq$ größeren Knoten ihren Datenaustausch abgeschlossen haben. Die Knoten die bezüglich $\preceq$ kleiner sind werden dann noch ihre eigene Phase haben, in der die noch fehlenden Daten ausgetauscht werden. Das garantiert zumindest die Korrektheit.

Optimale Laufzeit kann leider nicht garantiert werden. Der Algorithmus

braucht immer $(p - M + (M+1)/2)M$ Schritte, wobei

$$M := \max\{|\mathrm{Node}_0|, \ldots, |\mathrm{Node}_{P-1}|\}$$

die maximale Anzahl Prozessoren pro Knoten ist. Der Grund dafür findet sich in der Faktorisierung. Nach einer Phase haben alle Prozessoren im kleinsten Knoten und genausoviele in allen anderen mit dem bezüglich $\preceq$ größten Knoten Daten ausgetauscht. Die Faktorisierung garantiert dabei dass der größte Knoten an jeder Runde beteiligt ist (in einer davon tauschen immer seine kleinsten Prozessoren lokal Daten mit den anderen aus). Bis zur letzten Phase ist der letzte Knoten durchgehend aktiv und hat mit jedem Prozessor in anderen Knoten M Nachrichten ausgetauscht. Zusätzlich braucht er $M(M+1)/2$ Schritte für lokale Kommunikation, die über die Phasen verteilt sind. Da kein anderer Knoten seinen Datenaustausch später beendet ist das auch die Zahl der Schritte die der ganze Algorithmus macht. Das Problem des Algorithmus ist dass jeder Knoten in jeder Phase einmal mit sich selbst kommuniziert. Das ist nötig, da auch Nachrichten zwischen den Prozessoren eines Knotens auszutauschen sind. Diese interne Kommunikation ist aber schneller als Kommunikation zwischen mehreren Knoten. In einigen Fällen kann es ein Schedule geben, das lokale Kommunikation nicht mit welcher zwischen verschiedenen Knoten parallelschaltet.

Analyse:

$$T(n,p,M) = (p-M)M(T_{\text{start}} + nT_{\text{byte}}) + \frac{M(M+1)}{2}(T_{\text{local-start}} + nT_{\text{local-byte}})$$

3.6.4 h-Relation

Im Folgenden entfernt man sich davon, dass jeder Prozessor für jeden anderen gleich viele Daten hat und auch gleich viele von jedem anderen erhält. Jeder Prozessor hat hier eine Reihe von Paketen, die an andere Prozessoren gerichtet sind. Der Parameter h stellt bei dieser Kommunikationsoperation lediglich eine obere Schranke für die Anzahl Schritte dar, die ein Prozessor mit Kommunikation beschäftigt sein kann. Was das konkret bedeutet hängt davon ab, ob man Halbduplex oder Vollduplex für das Modell annimmt. Bei einem Schritt im **Vollduplexmodell** kann eine ausgehende und eine eingehende Nachricht parallelisiert werden. Hier wird also h durch das Maximum der Anzahl h_{in} von eingehenden Pakete und der Anzahl h_{out} von ausgehenden Pakete über alle Prozessoren bestimmt. Formal ausgedrückt bedeutet das

$$h := \max_{i=1}^{p} \max\{h_{\text{in}}(i), h_{\text{out}}(i)\}$$

Damit ist eine h-Relation in Vollduplex ein kollektiver Nachrichtenaustausch bei dem kein Prozessor mehr als h Pakete sendet oder empfängt. Bei **Halbduplex** kann Senden und Empfangen nicht parallelisiert werden. Folglich wird die Zeit, die ein Prozessor mit Kommunikation verbringen muss durch die Summe der Sende- und Empfangsoperationen festgelegt. Es gilt für h also

$$h := \max_{i=1}^{p}\{h_{\text{in}}(i) + h_{\text{out}}(i)\}$$

Da zumindest ein Prozessor alle h Schritte ausführen muss, wenn er ohne Unterbrechung kommunizieren kann, ist $h(T_{\text{start}} + |\text{Paket}|\, T_{\text{byte}})$ eine untere Schranke für die Dauer.

3.6.4.1 h-Relation mit Vollduplex

Im Duplex-Modell kommt man schnell auf eine Lösung, die optimale Zeit für die Kommunikation liefert. Ein Satz von Kőnig [6] über bipartite Graphen ist hier hilfreich. Dieser Satz besagt, dass der chromatische Index eines bipartiten Graphen gerade dem maximalen Grad entspricht. Um diesen Satz zu benutzen, beginnt man mit einem Hilfsgraphen, der zu einem ähnlich ist, der für die Labelwahl des 23-Baumalgorithmus benutzt wurde (3.2.4). Dieser **bipartite Multigraph** $G := (V, E)$ wird im Folgenden definiert.

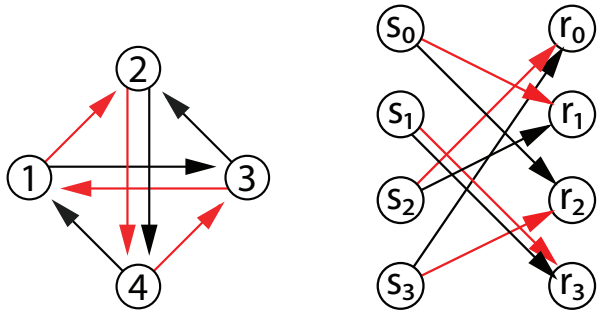
Die Knotenmenge im Hilfsgraphen hat pro Prozessor zwei Knoten, nämlich für ein PE i die Knoten s_i und r_i. Die Knotenmenge ist also:

$$V := \{s_0, \ldots, s_{p-1}\} \cup \{r_0, \ldots, r_{p-1}\}$$

Die Rollen der beiden Knoten werden klar wenn auch die Knotenmenge definiert wird:

$$|\{(s_i, r_j) \in E\}| = \text{ Anzahl Pakete von PE } i \text{ für PE } j$$

Es gibt also so viele Kanten von s_i nach r_j, wie Pakete von PE i nach PE j geschickt werden sollen. Es entspricht also eine Kante (s_i, r_j) einem Paket, das von PE i zu PE j gesendet werden muss. Damit ist der Grad des Knotens s_i gerade $h_{\text{out}}(i)$ und der von r_i ist dann $h_{\text{in}}(i)$. Der Satz von Kőnig sichert die Existenz einer h-Färbung im Hilfsgraphen G. Hat man eine solche Färbung, so lassen sich in einem Schritt immer die zu Kanten einer Farbe gehörenden Pakete konfliktfrei versenden. Da es nur h Farben gibt, benötigt man auch nur h Schritte, also optimale Zeit bei paketweiser Auslieferung. Die Freude über den optimalen Algorithmus wird etwas getrübt wenn man feststellt, dass es leider nicht so leicht ist, die Informationen über die Daten



(a) Färbung für 2-Relation (b) Hilfsgraph zur Bestimmung der Färbung

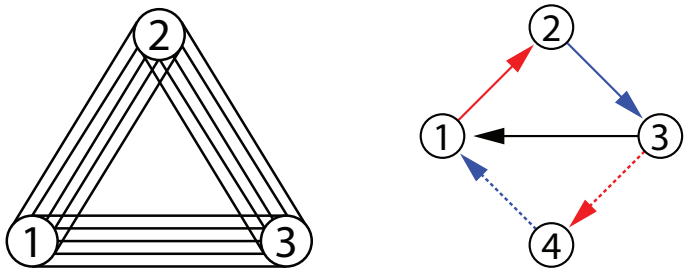
Abbildung 3.16: Beispiel für 2-Relation im Vollduplexmodell

zu sammeln, die versendet werden sollen und dann auch noch eine h-Färbung für den Hilfsgraphen zu bestimmen. Alleine der Zeitaufwand zur Bestimmung einer optimalen Färbung ist hoch, weswegen in der Praxis oft auf schnellere Algorithmen zurückgegriffen wird, die dafür nicht immer optimale Ergebnisse liefern.

3.6.4.2 h-Relation mit Halbduplex

Da eine Kommuikation in Halbduplex sowohl Sender, als auch Empfänger ganz in Anspruch nimmt, lässt sich der Satz von Kőnig nicht anwenden. Genauer ist nur eine Färbung mit $\lfloor 3h/2 \rfloor$ Farben zu beweisen. Tatsächlich gibt es Fälle, in denen auch so viele Schritte benötigt werden, ein Beispiel findet sich in Abbildung 3.17a. Ebenso gibt es aber auch welche, in denen man mit h Schritten auskommt, wo wieder Abbildung 3.16a als Beispiel dienen kann. Eine viel wichtigere Beobachtung ist, dass für ein optimales Schedule direktes Versenden von Paketen nicht immer ausreicht. Das kann man am besten an einem einfachen Beispiel zeigen, das in Abbildung 3.17b dargestellt ist. Wenn hier die Prozessoren 1 bis 3 paarweise Nachrichten für einander haben und der vierte nicht beschäftigt ist, so kann durch Einbindung des vierten Prozessors in die Kommunikation die Anzahl der nötigen Schritte von drei auf zwei Reduziert werden. Allgemein lässt sich immer noch eine untere Schranke von $\frac{6}{5}h$ für gerade p zeigen. Für ungerade p gibt es eine Schranke von $(\frac{6}{5} + \frac{18}{25p})h$. Auch mit indirekter Paketzustellung gibt es also Fälle, in denen $\frac{6}{5}h + O(1)$ Schritte nötig sind.

Die Idee des Algorithmus von Sanders und Solis-Oba [12] der nun be-



(a) worst-case-Beispiel für h-Relation in Halbduplex

(b) gesparter Schritt durch Helfer

Abbildung 3.17: h-Relation im Halbduplexmodell

schrieben wird, besteht darin, die h-Relation in $\lceil h/2 \rceil$ 2-Relationen aufzubrechen, die sich dann auch gut untersuchen lassen. Bei der Beschreibung wird die ursprüngliche Kommunikationsrichtung meist ignoriert. Das hat den einfachen Grund, dass im Halbduplexmodell die Richtung keinen Unterschied macht, weil sowieso beide Kommunikationspartner beschäftigt sind. Im ersten Schritt werden alle Knoten mit ungeradem Grad verbunden, so dass es nur noch Knoten mit geradem Grad gibt. Es kommen so nicht zu viele Kanten hinzu, denn es gibt höchstens eine für zwei Knoten, vor allem steigt der maximale Grad h höchstens um 1, genau dann wenn er ungerade war. Nachdem das getan ist, kann der Graph in kantendisjunkte Kreise zerlegt werden. In diesen Kreisen benutzt man die Möglichkeit die Kantenorientierung beliebig zu wählen und wählt sie so, dass die Kreise alle im Uhrzeigersinn ausgerichtet sind. So ist der Eingangsgrad und der Ausgangsgrad in jedem Knoten gleich groß, insbesondere also ist er immer kleiner oder gleich $\lceil h/2 \rceil$. In diesem Graphen kann wie beim Vollduplex-Algorithmus eine Färbung mit $\lceil h/2 \rceil$ Farben gefunden werden. Betrachtet man nur die Kanten einer Farbe, so hat jeder Knoten höchstens eine eingehende und eine ausgehende Kante. Die Knoten mit allen Kanten einer Farbe bilden also eine 2-Relation und es gibt genau $\lceil h/2 \rceil$ solcher 2-Relationen. Die Kanten – sofern im ursprünglichen Graphen vorhanden – können nun wieder ihre eigentliche Orientierung erhalten, denn die Aufspaltung in 2-Relationen ist fertig.

2-Relationen haben maximalen Grad zwei, können also nur aus knotendisjunkten Pfaden und Kreisen bestehen. In einem Pfad lassen sich die Kanten leicht in zwei Farben färben und damit ergibt sich auch ein Schedule in zwei Schritten. Analog sind auch für einen Kreis gerader Länge nur zwei Schritte nötig. Bei einem Kreis ungerader Länge hofft man zunächst darauf dass es noch einen Knoten gibt, der zu keinem Pfad oder Kreis der 2-Relation gehört.

So ein Knoten kann an beliebiger Stelle in den ungeraden Kreis integriert werden, so dass die Länge gerade wird. Das ist also eine Verallgemeinerung des Beispiels in Abbildung 3.17b für beliebige ungerade Kreise. Schwieriger ist es, wenn es mehr ungerade Kreise gibt, als Helferknoten. Hat man zwei Kreise ungerader Länge, so können Knoten aus einem Kreis für den anderen Kreis als Helferknoten fungieren. Jedes Paket wird dafür in 5 Teilpakete aufgeteilt. In Abbildung 3.18 ist die Vermittlung solcher aufgeteilter Pakete für zwei Kreise A und B der Länge $|A|$ bzw. $|B|$ dargestellt. Diese Vermittlung braucht immer 12 "kleine" Schritte, in denen nur Teilpakete von einem Fünftel der ursprünglichen Länge versendet werden. Eine Paarung von zwei ungeraden Kreisen braucht also ungefähr 12/5 Schritte.

Ist p gerade, so kann es auch nur eine gerade Anzahl an Pfaden und Kreisen ungerader Länge geben, die nicht mit einem unbeteiligten Knoten gepaart werden können. Paart man auch diese Kreise untereinander, so braucht keine 2-Relation mehr also 12/5 Schritte. Es gibt $\lceil h/2 \rceil$ solcher 2-Relationen und $\lceil h/2 \rceil$ ist maximal $(h+1)/2$, also braucht der Algorithmus bei **geradem** p maximal $\frac{12}{5 \cdot 2}(h+1) = \frac{6}{5}(h+1)$ Schritte. Bei ungeradem p kann nicht mehr garantiert werden, dass alle Kreise ungerader Länge untereinander oder mit unbeteiligten Knoten gepaart werden können. Um nicht zu viele Schritte deswegen zu verlieren, wird bei ungerader Anzahl ungerader Kreise immer eine Kante entfernt. Die entfernten Kanten werden für eine spätere Ausführung gesammelt. Dafür ist es wichtig, dass die Kanten keine gemeinsamen Knoten haben, damit die zu den Kanten gehörenden Pakete parallel verschickt werden können. Es lässt sich zeigen, dass sich immer mindestens $\lceil p/4 \rceil$ Kanten geeignet wählen lassen. Es ist also auch nur frühestens alle $\lceil p/4 \rceil$ Schritte nötig einen zusätzlichen Schritt einzufügen. Insgesamt erhöht sich die Schranke bei **ungeradem** p durch diese zusätzlichen Schritte auf insgesamt $(\frac{6}{5} + \frac{2}{p})(h+1)$ Schritte.

Der Algorihmus lohnt sich vor allem bei kleinen Prozessorzahlen und langen Nachrichten. Für große Prozessorzahlen kann neben Engpässen im Netzwerk auch die Bestimmung der Färbung zu viel Zeit kosten. Bei der Länge der Nachrichten ist wichtig, dass auch bei der Aufteilung in 5 Teilpakete der Startupoverhead gegen die Zeit für die eigentliche Datenübermittlung gering ist.

3.6.5 All-to-All mit unregelmäßigen Nachrichtenlängen

Nun verzichtet man auch auf die Einteilung in Pakete. Es geht noch einmal zurück zu einem Algorithmus für All-to-All, diesmal aber einen Fall bei dem

die Nachrichtenlängen nicht gleich sind. Es ist klar, dass dieser Fall noch etwas allgemeiner ist, als der der h-Relation, da dort die Nachrichtenlängen alle Vielfache einer Paketlänge waren. Der Einfachheit halber soll angenommen werden, dass jeder Prozessor gleich viele Bytes sendet und empfängt. Diese Anzahl soll ebenfalls h genannt werden. Die Nachrichten aus dem allgemeinen Fall können aber geeignet verlängert werden, um diesen Spezialfall zu erhalten. Es soll weiterhin angenommen werden, dass sich alle Nachrichtenlängen durch p teilen lassen, was bei langen Nachrichten keine zu starke Einschränkung ist. Zuletzt soll der Einfachheit halber ein Prozessor auch immer für sich selbst eine Nachricht haben, damit lästige Sonderfallbehandlungen vermieden werden können.

Algorithmus 3.14: Zweiphasenalgorithmus für All-to-All

```
alltoall2phase(m[1..p], p){
  for(j := 1; j < p; j++){
    a[j] = ⟨⟩;
    for(k := 1; k < p; k++){
      a[j] := a[j] ⊙ m[k][(i−1)n[k]/p + 1..j n[k]/p];
    }
  }
  b := regularalltoall(a, p);
  δ := ⟨1,...,1⟩;
  for (j := 1; j < p; j++){
    c[j] := ⟨⟩;
    for(k := 1; k < p; k++){
      //All−Gather to implement @:
      c[j] := c[j] ⊙ b[k][δ[k]..δ[k] + n[j]@k/p − 1];
      δ[k] := δ[k] + n[j]@k/p;
  }
  d := regularalltoall(c, p);
   permute d to obtain desired output format ;
}
```

Erläuterungen: Hinter dem Zweiphasenalgorithmus steckt die Idee den Fall mit unregelmäßiger Nachrichtenlänge durch zwei All-to-All-Operationen mit regelmäßiger Nachrichtenlänge zu ersetzen. Im ersten Schritt werden die Daten durch ein reguläres All-to-All gleichmäßig verteilt, so dass jeder Prozessor für jeden anderen gleich lange Nachrichten hat. Zunächst wird dazu jede Nachricht lokal in p Teile aufgespalten, von denen jeder andere Prozessor eines bekommt. Die Ausgehenden Nachrichten haben zusammen die Länge

h, also erhält in diesem Schritt jeder Prozessor von jedem anderen h/p Bytes. Da von einem Prozessor jeder andere gleich lange Teile einer Nachricht bekommt, sind nach der ersten Phase alle Nachrichten gleichmäßig über die Prozessoren verteilt. Weil die Summe der Längen von an einen Prozessor gerichteten Nachrichten h ist, so muss ein Prozessor h/p Bytes für jeden anderen halten, da die Nachrichten gleichmäßig verteilt wurden. Ein weiteres All-to-All liefert also die Nachrichtenteile an ihren Bestimmungsort, wo sie noch geordnet werden müssen, da Teile im ersten Schritt durcheinandergemischt wurden. Am anschaulichsten wird der Algorithmus in Abbildung 3.19 dargestellt. Bei kurzen Nachrichten und vielen Prozessoren werden die Nachrichten so womöglich zu kurz so dass sich der erste Schritt nicht mehr lohnt. Stattdessen bietet sich dort eine Aufteilung jeder Nachricht in $\log p$ Teile. Die Nachrichten werden dann im ersten Schritt zufällig verteilt um eine möglichst gleichmäßige Lastverteilung zu ermöglichen.

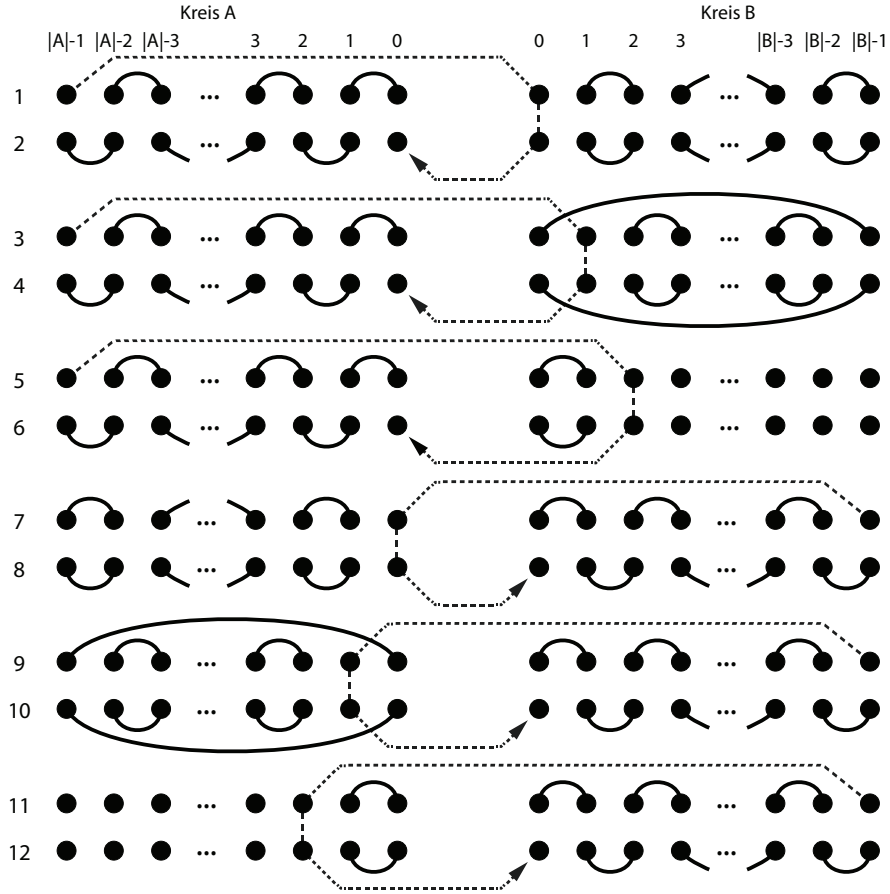


Abbildung 3.18: Paarung ungerader Kreise in 12 Schritten

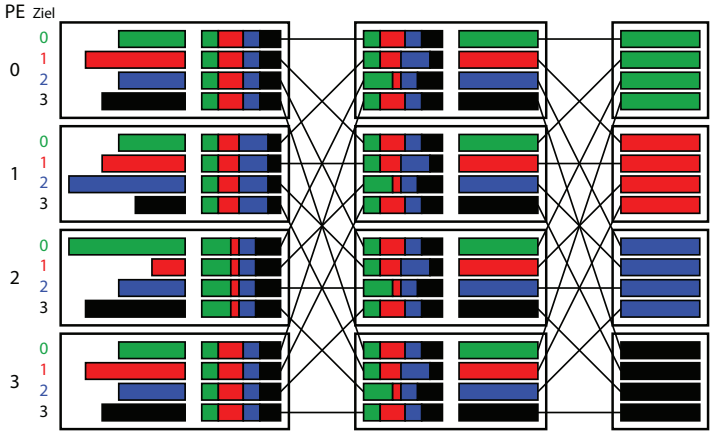


Abbildung 3.19: Zweiphasenalgorithmus für All-to-All mit unregelmäßigen Nachrichtenlängen

4 Sortieren und verwandte Probleme

Hier soll Paralleler Umgang mit einigen Datenstrukturen besprochen werden bzw. ihre parallele Implementierung. Es werden exemplarisch Techniken vorgeführt die sich auf viele weitere ähnliche Probleme anwenden lassen und auch die Nützlichkeit der zuvor besprochenen kollektiven Kommunikationsoperationen verdeutlichen.

4.1 Sortieren

Das erste was bei parallelem Sortieren überlegt werden muss, ist die Form der Eingabe. Bei Shared Memory ist es naheliegenderweise wie gewohnt, doch bei verteiltem Speicher ist es oft sinnvoll, die Eingabe mehr oder weniger gleichmäßig über die beteiligten Prozesoren zu verteilen. Nicht nur weil es im Laufe des Algorithmus sowieso passieren müsste, sondern weil es den Anforderungen an einen DMM-Sortieralgorithmus näherkommt. Dementsprechend wird das Problem für verteilten Speicher definiert:

Problembeschreibung: Gegeben ist eine Distributed Memory Machine mit p Prozessoren. Zu sortieren sind n Elemente $d_{i,j}$ wobei der Prozessor i bei der Eingabe die Elemente $d_{i,0}$ bis $d_{i,N-1}$ erhält mit $n = Np$. Für einige Algorithmen wird jedoch eine davon abweichende Eingabe vorgegeben. Gesucht ist eine Permutation $s_{i,j}$ mit $s_{i,0}$ bis $s_{i,N-1}$ auf Prozessor i. Für alle $s_{i,j}$ soll gelten $s_{i,j} \leq s_{i,k}$ für alle $j \leq k$ und $s_{i,j} \leq s_{i',k}$ für alle j, k und $i \leq i'$. Diese Aufgabe lässt sich sequentiell mit $n \log n + O(n)$ Vergleichen lösen, also in $O(n \log n)$ Zeit.

Häufig begnügen wir uns auch damit, den Rang in einer sortierten Folge zu bestimmen, womit hier auch begonnen wird. Für den ersten Algorithmus

wollen wir zusätzlich annehmen, dass $N = 1$ gilt und dass wir für jeden Prozessor, der ein Element der Eingabe hat noch $n-1$ weitere haben, die bei der Bestimmung der Ausgabe hinzugezogen werden können, um den folgenden Algorithmus zu realisieren:

4.1.1 Schnelles (ineffizientes) Ranking

Algorithmus 4.1: Schnelles Ranking

parallel_foreach $((i,j) \in \{1..n\}^2)$ {
  $B[i,j] := A[j] \leq A[i]$;
}
parallel_foreach $(i \in \{1..n\})$ {
  $M[i] := \sum_{j=1}^{n} B[i,j]$;
}

Erläuterungen: Zunächst sollte geklärt werden, dass die for-Schleifen komplett zu parallelisieren $p = n^2$ Prozessoren benötigt. Will man diesen Algorithmus, der stark nach PRAM aussieht, nun in das DMM-Modell übertragen, so müssen die globalen Speicherzugriffe durch Kommunikation ersetzt werden. Die erste Schleife enthält eine implizite Broadcast-Operation, jedes Element muss von einem Prozessor an $n-1$ weitere gesendet werden, damit sie es mit ihrem vergleichen können. In der Summe der zweiten Schleife steckt ebenso eine Reduce-Operation über n Elemente. Damit wird in jeder Schleife von n Prozessoren eine kollektive Kommunikationsoperation über n Prozessoren ausgeführt.

Ein Prozessorbedarf quadratisch in der Elementanzahl zeigt bereits zwei Eigenschaften des Algorithmus: Zum einen ist der Algorithmus nur für sehr kleine Eingaben zu gebrauchen, zum anderen können die Prozessoren gar nicht effizient arbeiten, nachdem n Prozessoren auf jedes Element der Eingabe kommen.

Analyse:

$$T(n,p) = T(n,n^2) \in T_{\text{broadcast}}(1) + T_{\text{reduce}}(1) \in O(T_{\text{start}} \log(n^2))$$

4.1.2 Ranking für große Eingaben

Nun soll Ranking so weiterentwickelt werden, dass auch große Eingaben verarbeitet werden können. Ganz ohne Einschränkungen an die Prozessoren wollen wir hier auch nicht auskommen, sondern gehen von $p = P \times P$ Prozessoren aus, jeder der p Prozessoren lässt sich dann über ein Index-Tupel (i, j) bezeichnen.

Algorithmus 4.2: Ranking für große Eingaben

$a := \odot_{i'=1}^{P} d@(i', j)$;
$a' := a@(i, i)$;
sort(a);
for$(k := 1;\ k < NP;\ k{+}{+})\{$
  $b_k :=$ rank of a'_k in a;
}
$b' := \sum_{j=1}^{P} b@(i, j)$;

Erläuterungen: Im Folgenden sollen die Prozessoren als in einem Quadrat angeordnet angenommen werden. Eine Zeile bilden dann alle Prozessoren mit dem gleichen Eintrag in der ersten Komponente des Index, entsprechend sind die Prozessoren in der gleichen Spalte, wenn ihre zweite Komponente übereinstimmt. Am Anfang des Algorithmus sammelt jeder Prozessor in a die Eingaben d aller Prozessoren mit der gleichen Spalte. Das entspricht einer Gossip-Operation, da jeder der Prozessoren einer Spalte die Daten aller anderen Prozessoren braucht. a' wird für jeden Prozessor durch eine Broadcast-Operation entlang der Zeile bestimmt. Dabei werden in der j-ten Zeile die Ergebnisse der Reduktion der j-ten Spalte aus dem vorherigen Schritt propagiert. In der j-ten Zeile kann nun jeder Prozessor für die per Broadcast verteilte Teilmenge ein Teilranking bestimmen. Damit ist für jedes Element und jede Teilmenge der Eingabe (da in jeder Spalte mit einer anderen verglichen wird) bekannt, wie viele kleiner oder gleich sind. Eine Summe über alle Teilrankings liefert also den globalen Rang. Diese Summe lässt sich dann im letzten Schritt über eine Reduktionsoperation bestimmen. Ein Beispiel für den Algorithmus ist in Abbildung 4.1 dargestellt.

Analyse:

$$T(n,p) = T(N(P \times P), P \times P) = T_{\text{gossip}}(N, P) + T_{\text{broadcast}}(NP, P) + T_{\text{reduce}}(NP, P) + O(NP \log(NP)) \in O(T_{\text{start}} \log p + N\sqrt{p}(T_{\text{byte}} + \log n))$$

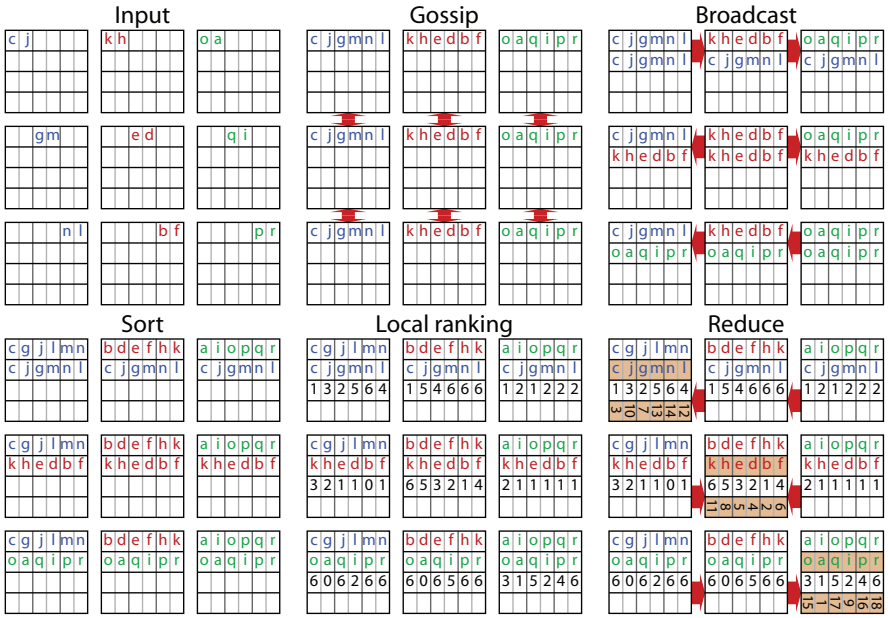


Abbildung 4.1: Ranking für große Eingaben

4.1.3 Quicksort

Zunächst der Algorithmus, um den sich in diesem Abschnitt alles dreht – das (sequentielle) Quicksort:

Algorithmus 4.3: Sequentielles Quicksort

```
qSort(d[], p'){
  if(p' = 1) return;
  select pivot v;
  reorder elements in d such that
    d_0 ≤ ... ≤ d_k = v ≤ d_{k+1} ≤ ... ≤ d_{p'-1};
  qSort([d_0..d_{k-1}], k);
  qSort([d_{k+1}..d_{p'-1}], m-k-1);
}
```

Die erste Idee, die man oft sieht, um den Algorithmus zu parallelisieren, ist die Parallelisierung der rekursiven Aufrufe. In den ersten Rekursionen des Algorithmus ist in diesem Fall die Parallelisierung gering und nimmt erst mit der Rekursionstiefe zu. Gleichzeitig wird in diesen Rekursionen noch auf großen Mengen von Daten operiert, da die Teilfolgen mit steigender Rekursionstiefe kürzer werden. So ist bis zur ersten Aufspaltung nur ein Prozessor auf den gesamten Daten aktiv. Später kann die Lastbalancierung ein Problem werden, da nicht mehr sichergestellt werden kann, dass die verschiedenen Prozessoren ähnlich große Teile bekommen. Soll der Algorithmus dann noch auf eine Distributed Memory Machine übertragen werden, so kommt hinzu, dass bei der Aufspaltung die Daten an die jeweiligen Prozessoren gesendet werden müssen, was großen Kommunikationsaufwand bei jeder Aufspaltung bedeutet.

Zur Vereinfachung soll hier wieder mit einem Algorithmus begonnen werden, dem für jedes zu sortierende Element ein Prozessor zur Verfügung steht.

Algorithmus 4.4: Theoretiker-Parallelisierung von Quicksort

```
procedure theoQSort(d,i,p)
  if(p = 1) return;
  j := common random element from 0..p-1 for partition;
  v := d@j; //Pivot
  f := d ≤ v;
  j := Σ_{k=0}^{i} f@k; //Präfixsumme
  p' := j@(p-1);
  if(f) send d to PE j-1;
  else send d to PE p'+i-j;
  receive d;
  if(i < p'){
    join partition "left";
    theoQSort(d,i,p');
  }else{
    join partition "right";
    theoQSort(d,i-s,p-p');
  }
```


Erläuterungen: Die Prozessoren werden hier in Partitionen aufgeteilt, auf die sich die Kommunikation beschränkt. In einer Partition müssen sie sich dann auf einen Prozessor j einigen, dessen Element das Pivotelement darstellen soll. Dieses Pivotelement v wird dann über einen Broadcast jedem Prozessor in der Partition für einen lokalen Vergleich zur Verfügung gestellt. Über eine Präfixsumme lassen sich nun die Elemente, die kleiner/größer sind als das Pivotelement, neu durchnummerieren. Die Nummerierung wird anschließend verwendet um den Index des Prozessors in der Partition zu finden, an den das eigene Element gesendet wird. Danach halten die ersten $p'-1$ Prozesoren alle $p'-1$ Elemente die kleiner sind als das Pivotelement, folglich sind die restlichen Elemente über die restlichen Prozessoren verteilt. Analog zum sequentiellen Quicksort wird die Partition bei p' in zwei neue Partitionen aufgeteilt, auf denen der Algorithmus rekursiv ausgeführt wird.

Die meiste Zeit in jeder Rekursion kosten die kollektiven Operationen. Das Pivotelement zu verteilen erfordert einen Broadcast. Durchnummerieren (und damit auch zählen) der Elemente, die kleiner sind als das Pivot lässt sich mit einer Präfixsumme bewerkstelligen. Anschließend weiß aber nur ein Prozessor mit Sicherheit, wie viele kleine Elemente es gibt, also ist noch ein weiterer Broadcast nötig, damit alle Prozessoren p' kennen. Das macht zwei Broadcasts und eine Reduktion, aber alles mit kurzen Nachrichten, also noch in Zeit $O(T_{\text{start}} \log p)$. Quicksort hat erwartete Rekursionstiefe $O(\log p)$, die noch als Faktor in die Laufzeit mit einfließt. Damit ist die erwartete Laufzeit $O(T_{\text{start}} \log^2 p)$.

Analyse:

$$T(n,p) = T(n,n) \in O(T_{\text{start}} \log^2 n)$$

Mit diesem Ansatz bewaffnet könnte man nun das Problem für große n auf gewohnte Art angehen. Jeder Prozessor ist nicht mehr für nur eines, sondern für mehrere Elemente verantwortlich. Er überprüft nach dem Broadcast des Pivotelementes auch für alle seine Elemente die Zuordnung. Dabei zählt er

auch wie viele größer sind und wie viele kleiner als das Pivot. Präfixsummen lassen sich wie gewohnt berechnen und liefern einem Prozessor die Information wie viel Platz die vorherigen für ihre kleinen/großen Elemente brauchen. Bei Distributed Memory muss noch geklärt werden, wie mit dem Prozessor umgegangen werden soll, der nach einer Tauschphase möglicherweise sowohl große als auch kleine Elemente hat und damit zu beiden Partitionen gehört. Auf PRAM muss er sich in diesem Fall nur für eine Seite entscheiden und dann ist das Problem in

$$O\left(\frac{n \log n}{p} + \log^2 p\right)$$

Zeit lösbar. Eine Änderung gegenüber dem sequentiellen Quicksort besteht aber darin, dass Elemente auch bewegt werden, wenn sie eigentlich "auf der richtigen Seite" wären. Das bedeutet für jedes Element $\Omega(\log p)$ Bewegungen und eine Laufzeit von

$$O\left(\frac{n}{p}(\log N + T_{\text{byte}} \log p) + T_{\text{start}} \log^2 p\right)$$

auf Distributed Memory. Die zufällige Pivotwahl macht Vorhersagen über die Lastverteilung während der Abarbeitung schwierig. Eine ungleichmäßige Aufteilung kann deutliche Unterschiede in der Rekursionstiefe bedeuten. Mit jeder Aufspaltung verringert sich die Anzahl der für eine Teilsequenz verantwortlichen Prozessoren. Wenn die Aufteilung ungleichmäßig geschieht so gelangt ein Teil der Prozessoren schneller in eine Phase, in der nur noch jeder eine eigene Sequenz sortieren muss. Bei einer Shared Memory Machine können sich Prozessoren, die ihre Arbeit erledigt haben, wieder an späteren Schritten beteiligen. Etwas ähnliches ist bei einer Distributed Memory Machine leider nicht möglich. Allgemein ist Quicksort praktisch nicht effizient auf einer Distributed Memory Machine umzusetzen, da Elemente häufig unnötig verschoben werden. Dadurch entstehen hohe Kommunikationskosten, die den Quicksort-Ansatz für solche Maschinen unpraktikabel machen.

4.1.4 MCSTL: Quicksort

MCSTL enthält eine parallele Implementierung von Quicksort, deren Ziel es ist auch in der Praxis schnell zu sein. Wie die ganze MCSTL ist auch die Quicksort-Implementierung auf Shared Memory ausgelegt, daher ist das Tauschen von Elementen nicht so kritisch wie im Fall des verteilten Speichers.

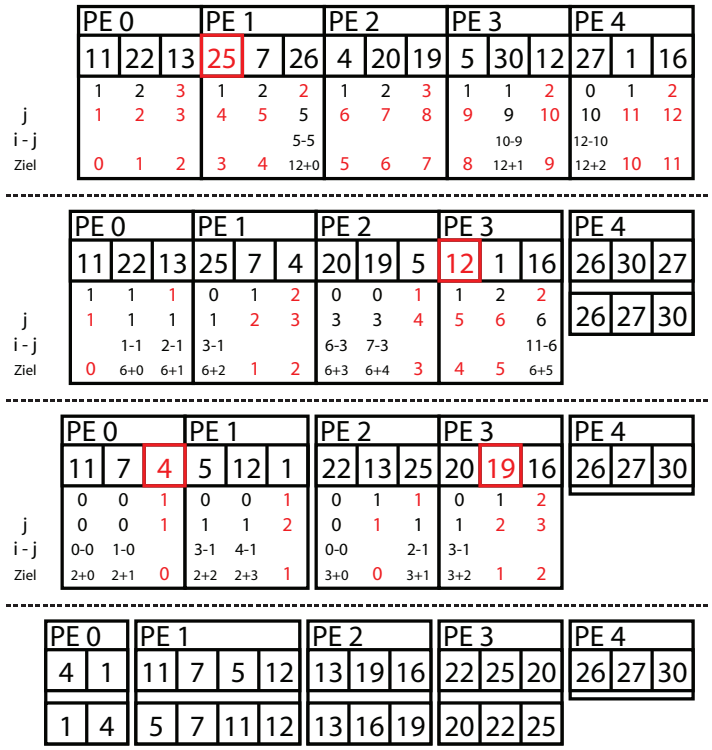


Abbildung 4.2: verallgemeinerung von "Theoretiker-Quicksort"

4.1.4.1 partition

Die Hauptoperation einer Quicksort-Iteration ist bei der MCSTL auch als eigenständiger Algorithmus verfügbar. Als zentraler Bestandteil des Quicksortalgorithmus soll das Partition Problem und der MCSTL-Algorithmus dazu zunächst separat betrachtet werden.

Problembeschreibung: Gegeben sind n Elemente $d_0, .., d_{n-1}$ und ein Pivotelement v. Gesucht ist eine Permutation $s_0, .., s_{n-1}$ von $d_0, .., d_{n-1}$ für die gilt $(s_i \geq v \wedge s_j < v) \Rightarrow i < j$ bzw. $\exists k : (\forall i < k : s_i < v \wedge \forall i \geq k : s_i \geq v)$.

In anderen Worten: Alle Elemente kleiner als das Pivot kommen auf die linke Seite, alle größeren auf die rechte. Zwei Elemente, die beide kleiner oder beide größer sind als das Pivot, müssen aber nicht in der richtigen Reihenfolge sein, noch nicht einmal in der, die sie vor der Partitionsoperation hatten.

Der Algorithmus aus der MCSTL richtet sich nach der Quicksort-Parallelisierung aus [15]. Jeder Prozessor beansprucht einen Block konstanter Länge B vom Anfang und einen vom Ende der Eingabe für sich. Das Beanspruchen kann durch zwei Variablen kontrolliert werden, auf die Prozessoren über `fetch-and-add` zugreifen. Die `fetch-and-add`-Zugriffmethode garantiert dass zwischen dem Auslesen und der Addition keine Zugriffe durch andere Prozessoren stattfinden. Man Speichert in den beiden Variablen also die jeweils erste Position vom Anfang bzw. vom Ende aus gezählt, ab der die Elemente keinem Prozessor zugeordnet sind. So kann ein Prozessor in einer atomaren Operation auslesen, ab welchem Index er B Elemente beanspruchen kann und den Eintrag um B erhöhen, damit kein anderer Prozessor ebenfalls an dieser Stelle beginnt. Dadurch hat sich ein Prozessor den gesamten Block reserviert.

Sobald ein Prozessor zwei Blöcke reserviert hat, beginnt er mit der Abarbeitung. Wie im sequentiellen Fall sucht er das erste Element im linken Block das größer ist, als das Pivotelement und eines im rechten Block, das kleiner ist. Diese Elemente werden getauscht. Sobald das Ende von einem Block erreicht ist, versucht der Prozessor einen neuen anzufordern und seine Arbeit dort fortzusetzen. Dies schlägt fehl, sobald kein ganzer Block mehr beansprucht werden kann, da er sich mit dem letzten auf der gegenüberliegenden Seite überlappen würde. Es kann also höchstens noch ein unbearbeiteter Block vorhanden sein, dessen Größe kleiner ist als B. An dieser Stelle werden alle Elemente, die noch immer ein Partnerelement auf der gegenüberliegenden Seite benötigen, mit dem sie getauscht werden können, an den unvollständigen, nicht zugeordneten Block getauscht. Da kein Prozessor mehr als B solche Elemente haben kann, kommt man auf insgesamt noch $B(p+1)$ Elemente, die

noch betrachtet werden müssen. Diese in der Mitte gesammelten Elemente müssen nur noch rekursiv – oder bei weniger als B Elementen sequentiell – auf die richtige Seite getauscht werden.

Analyse:

$$T(n,p) \in O(\frac{n}{p} + Bp)$$

4.1.4.2 Quicksort

Die MCSTL-Implementierung von Quicksort beginnt mit wiederholter Anwendung des parallelen `partition`. Dabei werden bei jeder Partitionierung auch die Prozessoren aufgeteilt, so dass beide Teile parallel weiter partitioniert werden. Sobald nach erwarteten $O(\log p)$ Aufspaltungen nur noch ein Prozessor für eine Partition verantwortlich ist, führt er auf seinen Daten sequentielles Quicksort aus. Es ist unwahrscheinlich, dass die bis dahin entstandenen Partitionen gleich lang sind. Deswegen wird die Last dynamisch verwaltet. Bei erfolgreicher Lastverteilung bleibt für jeden Prozessor nur $O(n \log n/p)$ erwartete Arbeit. Für die Lastverteiltug besitzt jeder Prozessor einen eigenen Stack. Bei jedem sequentiellen Schritt des Quicksortalgorithmus werden die Daten in zwei Partitionen aufgeteilt. Der Prozessor bearbeitet eine und legt die zweite auf den Stack zur späteren Bearbeitung.

Hat ein Prozessor auch den Stack abgearbeitet, greift er auf den Stack eines anderen Prozessors zu. Dort beansprucht er aber das untere Element für sich. Das hat zwei Vorteile. Der erste ist, dass er so ein großes Stück Arbeit erhält, da die Partitionen im unteren Teil des Stacks zu früheren Rekursionsschritten gehören. Somit bleiben Zugriffe auf Stacks anderer Prozessoren vergleichsweise selten. Damit kleine Probleme überhaupt nicht zwischen Prozessoren wandern, könnte man hinreichend kleine Partitionen ohne Stackzugriffe verwalten. Der zweite Vorteil ist die Ersparnis beim Aufwand für den Stackzugriff. Der konkurrierende Zugriff mehrerer Prozessoren auf gemeinsame Daten muss kontrolliert werden. Da auf das obere Element nur der Besitzer des Stacks zugreift, ist hier keine Kontrolle nötig. Für das untere Element kann über atomares `fetch-and-add` der Zugriff auch ohne locks verwaltet werden. Es kann trotz allem passieren, dass ein Prozessor, der keine eigene Arbeit hat die Ausführung bremst. Das ist besonders dann der Fall, wenn es mehr Threads als Prozessoren gibt und ein Thread noch auf einen anderen wartet, in dem nur erfolglose Anfragen nach Arbeit stattfin-

den. Für diesen Fall gibt ein Thread bei der MCSTL-Implementierung nach einer erfolglosen Anfrage mit `yield` den Prozessor frei.

Analyse:

$$\mathbb{E}T(n,p) \in O(\frac{n \log n}{p} + Bp \log p)$$

4.1.4.3 Andere Algorithmen

Es gibt aber noch weitere Operationen, die von `partition` profitieren. Eine Operation, die später für Distributed Memory unter dem Namen **Select** für einen Spezialfall betrachtet wird (siehe 4.2.2), heißt in der (MC)STL `nth_element`. Für diese Operation, die in einem Array das Element mit Rang r bestimmt, kann der bekannte Quickselectalgorithmus leicht parallelisiert werden. Es wird wie im sequentiellen Fall die Menge um ein Pivotelement partitioniert. Sind auf der linken seite mindestens r Elemente, wird dort rekursiv weitergesucht. Hat sie weniger Elemente, so kann r um die Größe der linken Partition gemindert werden und die Suche wird mit dem neuen r in der rechten Partition fortgesetzt. Unterschreitet die zu durchsuchende Partitionsgröße $2B$, so lohnt sich Parallelisierung nicht mehr und es wird zum sequentiellen Algorithmus gewechselt.

Analyse:

$$\mathbb{E}T(n,p) \in O(\tfrac{n}{p} + Bp \log p)$$

In der MCSTL wird `nth_element` wiederrum verwendet, um `partial_sort` zu realisieren. Diese Operation sortiert die ersten r Elemente. Bei der Bestimmung der r-ten durch `nth_element` ist das Array bereits so partitioniert, dass die ersten r bekannt sind und noch sortiert werden müssen.

4.1.5 Sample Sort

Die Aufteilung in zwei Teilmengen könnte erzwungen vorkommen, wenn man doch ganze p Prozessoren zur Verfügung hat, von denen sich jeder einem Teil der Eingabe widmen könnte. Der nun folgende Algorithmus aus [14] macht sogar etwas mehr. Um Kommunikation zu sparen wird direkt festgestellt welcher Prozessor welche Daten erhält. So muss jedes Datum nur einmal verschickt werden und liegt danach auf dem Prozessor, der dafür verantwortlich

ist. Danach muss jeder Prozessor nur noch die Daten sortieren die er erhalten hat. Die Ersparnis in der Kommunikation ist es, was den Ansatz auch für Distributed Memory interessant macht.

Algorithmus 4.5: Parallel Sample Sort

```
v_0 := −∞;
v_p := ∞;
choose S·p random elements s_k, 0 ≤ k < S·p;
PE i selects s_{S·i}..s_{S·(i+1)−1};
sort([s_0..s_{S·p−1}]);
for(k:=1; j < p−1; j++){
  v_k := s_{S·k};
}
initialize p empty messages N_k, 0 ≤ k < p;
for(j := 0; j < N−1; j++){
  choose k with v_k ≤ d_j < v_{k+1};
  write d_j to message N_k;
}
for(j := 0; j < p−1; j++){
  send N_j to PE j;
}
receive p messages;
sort(received elements);
```

Erläuterungen: Der Algorithmus funktioniert, indem global einige Splitter v_k bestimmt werden, die für die Prozessoren als Orientierung dienen, welcher Prozessor für welchen Teil der Eingabe verantwortlich ist. Lokal wird für die eigenen Elemente bestimmt, in welches Intervall zwischen Splittern sie jeweils fallen. Die Elemente werden dann dem für das Intervall verantwortlichen Prozessor geschickt, der sie sortiert. Naheliegenderweise funktioniert der Algorithmus am besten, wenn die Splitter die Elemente gleichmäßig aufteilen, also die Intervalle gleich viele Elemente enthalten.

Wenn die Splitter bereits als Elemente mit Rang $k \cdot m/p$ in der sortierten Menge gegeben wären, so wäre der Zeitbedarf

$$T(n,p) = \overbrace{O(N \log p)}^{\text{lokale Zuordnung}} + \overbrace{T_{\text{all-to-all}}(N,p)}^{\text{Datenaustausch}} + \overbrace{T_{\text{seq}}(N)}^{\text{lokal sortieren}} \approx \frac{T_{\text{seq}}(Np)}{p} + 2NT_{\text{byte}} + pT_{\text{start}}$$

Solche Splitter könnte man durch $(p-1)$-fache Ausführung von `select` erhalten, das die $k \cdot n/p$-ten Elemente liefert (vgl. Definition in 4.2.2). Vernünftiger

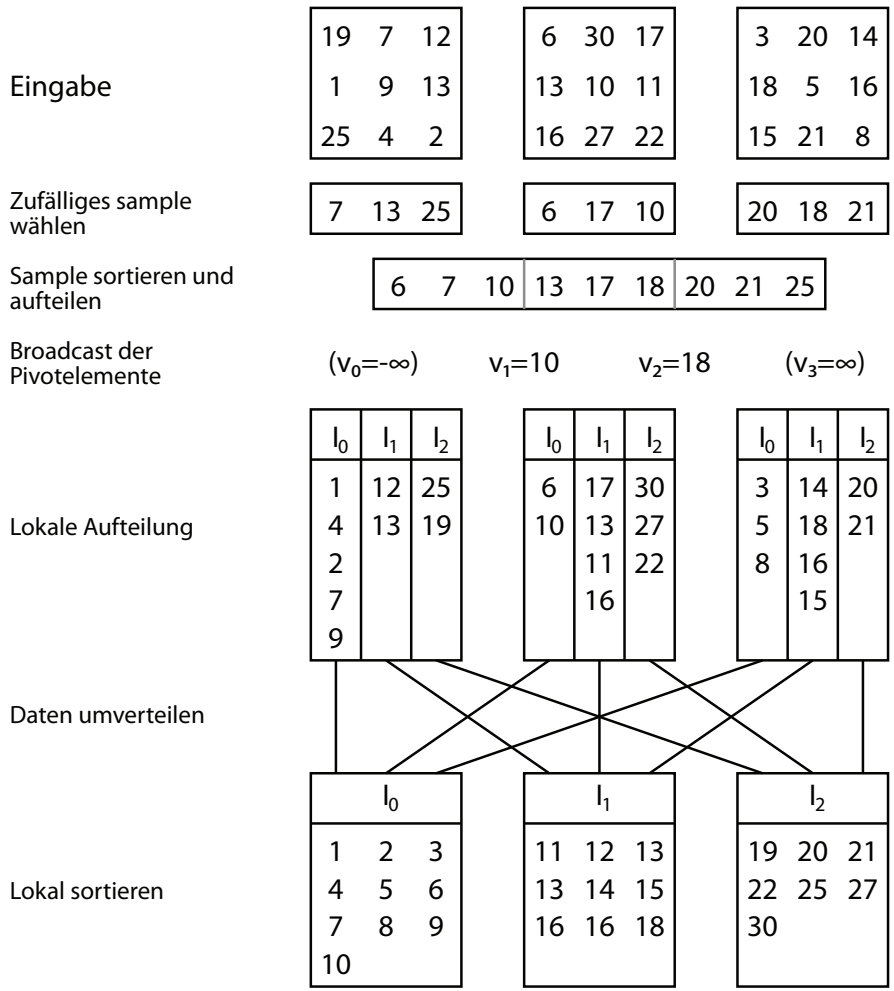


Abbildung 4.3: Sample Sort

ist es dann auch zunächst lokal zu sortieren und dann Multisequence Selection zu verwenden, da dieses Verfahren schneller ist und man auch später noch von der Sortierung der Teilfolgen profitiert. Das würde dann aber zusätzlichen Zeitbedarf von mindestens $\Omega(p \log \frac{n}{p})$ bedeuten (vorausgesetzt die Bestimmung soll vergleichsbasiert erfolgen), da es bereits so viel Zeit kostet ein Element mit einem bestimmten globalen Rang zu bestimmen.

Hier werden die Splitter allerdings nur so gewählt, dass sie das erheblich kleinere zufällige Sample von pS Elementen gleichmäßig aufteilen, bei der gesamten Eingabe muss das nicht notwendigerweise auch der Fall sein. Es lässt sich aber sehr wohl zeigen, dass für ein $S \in \Theta\left(\frac{\log n}{\epsilon^2}\right)$ die Wahrscheinlichkeit, dass kein PE mehr als $(1+\epsilon)N$ Elemente bekommt, größer ist als $1 - \frac{1}{n}$.

Um das zu zeigen betrachtet man die Eingabe in sortierter Reihenfolge, bezeichnet als $\langle e_1, ..., e_n \rangle$. Es interessiert die Wahrscheinlichkeit $\mathbb{P}[\text{fail}]$ dass ein Prozessor mehr als $(1+\epsilon)N$ Elemente bekommt. Dann muss ein $(1+\epsilon)N$ Elemente langer Abschnitt der Eingabe existieren, aus dem höchstens S Elemente als Samples gezogen werden. Das muss gelten, da jede Teilmenge, die ein Prozessor bekommt, S Samples enthält. Der Einfachheit halber soll die Annahme gelten, dass die Samples global und mit Zurücklegen gezogen werden. Das Ereignis $\mathcal{E}_j$ soll also dadurch definiert werden dass in $\langle e_j, ..., e_{j+(1+\epsilon)N} \rangle$ höchstens S Samples liegen. Damit ein Prozessor mehr als $(1+\epsilon)N$ Elemente bekommt, muss $\mathcal{E}_j$ – wie bereits festgestellt – zumindest für ein j eintreten. Da die Wahrscheinlichkeit für $\mathcal{E}_j$ bei zufällig gezogenen Samples nicht von j abhängt, kann man sagen, dass für ein beliebiges, aber festes j, die gesuchte Wahrscheinlichkeit $\mathbb{P}[\text{fail}]$ höchstens so groß ist wie $n\mathbb{P}[\mathcal{E}_j]$. Es muss also $\mathcal{E}_j$ nur für ein j bestimmt werden, also soll j im Folgenden fest sein. Für die Bestimmung von $\mathcal{E}_j$ soll jetzt eine Zufallsvariable eingeführt werden:

$$X_i := \begin{cases} 1 & \text{falls } s_i \in \langle e_j, ..., e_{j+(1+\epsilon)N} \rangle \\ 0 & \textit{sonst} \end{cases}, \; X := \sum_{i=0}^{S \cdot p - 1} X_i$$

Für den Erwartungswert von X_i gilt

$$\mathbb{E}[\mathcal{E}_j] = \mathbb{P}[X_i = 1] = \frac{1+\epsilon}{p}$$

das soll für eine Abschätzung von $\mathbb{P}[\mathcal{E}_j]$ verwendet werden. Nach der Definition von $\mathcal{E}_j$ und der von X ergibt sich

$$\mathbb{P}[\mathcal{E}_j] = \mathbb{P}[X < S] \approx \mathbb{P}[X < (1-\epsilon^2)S] = \mathbb{P}[X < (1-\epsilon)\mathbb{E}[X]]$$

Die ganze Mühe nur damit man die Chernoff-Ungleichung anwenden kann die besagt:

$$\text{Für } X := \sum_i X_i \text{ mit unabhängigen Zufallsvariablen } X_i \text{ gilt}$$
$$\mathbb{P}\left[X < (1-\epsilon)\mathbb{E}\left[X\right]\right] \leq \exp\left(\frac{\epsilon^2 \mathbb{E}\left[X\right]}{2}\right)$$

Da globales Ziehen der Samples mit Zurücklegen angenommen wurde, ist die Forderung nach unabhängigen Zufallsvariablen erfüllt. Dann wird es schon Zeit zu zeigen, dass die Fehlerwahrscheinlichkeit tatsächlich so klein wird wie angekündigt, wenn S nur richtig gewählt wurde.

$$n\mathbb{P}\left[X < S\right] \leq n \exp\left(\frac{\epsilon^2 S}{2}\right) \leq n\frac{1}{n^2} \text{ für } S \geq \frac{4}{\epsilon^2}\ln n$$

Für die Laufzeit bedeutet das zunächst, dass bei Splitterbestimmung durch Samples der Länge S mit bis zu $(1+\epsilon)N$ Elementen auf einem Prozessor zu rechnen ist, die dementsprechend auch verschickt und sortiert werden müssen. Leider muss das Sample gemäß der bewiesenen Schranke sehr groß gewählt werden um ϵ klein zu halten, da ϵ dort quadratich eingeht. Die Zeit für die eigentliche Bestimmung der Splitter hängt auch davon ab, welches Verfahren für das Sortieren der Samples verwendet wird. Dabei bieten sich alle bisher betrachteten Verfahren an. Bei kleinen Samples kann es sich als sinnvoll erweisen die Samples über eine Gossip-Operation zu verteilen und jeden Prozessor für sich alle Splitter bestimmen zu lassen oder sie über eine Gather-Operation bei einem Prozessor zu sammeln der sie sortiert und über einen Broadcast verteilt. Ist das Sample größer, so kann in einem Zwischenschritt für das Sample wiederrum Sample Sort verwendet werden. Das kleinere dort entstehende Sample kann dann mit einem Algorithmus für kleine Eingaben sortiert werden.

Analyse:

$$T(n,p) = \overbrace{\overbrace{T_{\text{sort}}(O\left(\frac{\log n}{\epsilon^2}\right), p)}^{\text{sample sortieren}} + \overbrace{T_{\text{all gather}}(p,p)}^{\text{splitter sammeln/verteilen}}}^{\text{klein wenn } N \gg p\log p} +$$
$$\underbrace{O(N\log p)}_{\text{verteilen}} + \underbrace{T_{\text{all-to-all}}((1+\epsilon)N, p)}_{\text{Datenaustausch}} + \underbrace{T_{\text{seq}}((1+\epsilon)N)}_{\text{lokal sortieren}}$$

Sample Sort wurde bislang als randomisierter Algorithmus betrachtet, es lässt sich aber auch Deterministisch entwerfen. Um alle Phasen zu beschleunigen werden hierbei die Daten zunächst lokal sortiert. Jeder Prozessor wählt

nun für sich Elemente, die die lokalen Daten gleichmäßig in S Teile aufspalten, das ist das deterministische Sample. Ein Gossip der lokalen Samples gibt allen Prozessoren die Möglichkeit für alle Samples lokale Ränge zu bestimmen. Über ein All-Reduce werden daraus die globalen Ränge bestimmt. Aus diesen Rängen lassen sich dann Splitter und auch entsprechende Intervalle berechnen die wie gewohnt verwendet werden, um den für die Daten zuständigen Prozessor zu finden. Dort müssen auch nicht mehr die empfangenden Daten vollständig sortiert werden, es reicht bereits p-Wege-Mischen (Multiway Merge), da sortierte Teilfolgen vorliegen.

4.1.6 Multiway Mergesort

Algorithmus 4.6: Multiway Mergesort

```
sort([d_{i,0}..d_{i,N-1}]);
v_0 := −∞;
v_p := ∞;
for(k := 1; j < p−1; j++){
  v_k := element with global rank k·n/p;
}
initialize p empty messages M_k, 0 ≤ k < p;
l := 0;
for(k := 0; j < p; j++){
  r := max({a|d_{i,a} < v_k} ∪ {−1});
  write [d_{i,l}..d_{i,r}] to message M_k;
  send M_k to PE k;
}
receive p messages;
merge(received lists);
```

Erläuterungen: Bereits in der Beschreibung von Sample Sort wurde die Möglichkeit erwähnt Splitter durch Selection zu bestimmen. Etwas ähnliches wird in Multiway Mergesort getan. Das Sortieren zu Beginn des Algorithmus soll es erleichtern die Splitter zu finden. Verwendet wird für die Suche nach Splittern ein von Quickselect abgeleiteter Multiselect-Algorithmus. Dass die Teilfolgen bereits sortiert gegeben sind erlaubt die Anwendung von Binärsuche. Dieser Algorithmus ist allerdings noch immer um einen Faktor $\log p$ langsamer als die untere Schranke von $\Omega(p\log\frac{n}{p})$ für vergleichsbasierte Algorithmen. Sind die Splitter bestimmt, kann wie bei Sample Sort jeder Prozessor seine Elemente Intervallen zuordnen, wobei hier die Suche auch dadurch vereinfacht wird, dass die Elemente lokal sortiert sind. Der Datenaustausch

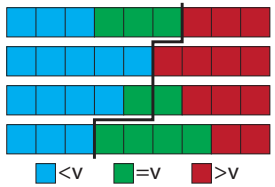


Abbildung 4.4: `multisequence_partition`; gesucht ist eine Aufteilung nach dem 19. Rang, die als grün gekennzeichneten Elemente haben den gleichen Schlüssel, wie das Element mit diesem Rang (v)

wird dadurch begünstigt, dass die Splitter gerade so gewählt wurden, dass alle Prozessoren gleich viele Elemente erhalten. Das sorgt auch für optimale Lastverteilung beim anschließenden Verschmelzen der empfangenen Teilfolgen. An dieser Stelle profitiert man auch wieder vom anfänglichen Sortieren der Elemente, da nun nur noch sortierte Teilfolgen und nicht Mengen von Elementen vorliegen. Das alles macht Multiway Mergesort vor allem für Shared Memory sinnvoll. Bei Distributed Memory muss darauf geachtet werden Datenaustausch gering zu halten, indem sich beispielsweise alle Prozessoren an jedem Select beteiligen.

Analyse:

$$T(n,p) = T_{\text{seq}}(\tfrac{n}{p}) + T_{\text{m-select}}(\tfrac{n}{p},p) + T_{\text{all-to-all}}(\tfrac{n}{p},p) + T_{\text{m-way-merge}}(\tfrac{n}{p},p) \approx \tfrac{n}{p}\log\tfrac{n}{p} + O(p\log p\log\tfrac{n}{p}) + pT_{\text{start}} + \tfrac{n}{p}T_{\text{byte}} + O(\tfrac{n}{p}\log p)$$

4.1.7 MCSTL: Multiway Mergesort

Auch bei Multiway Mergesort bietet die MCSTL einen Algorithmus für Shared Memory.

4.1.7.1 Multisequence Selection/Partition

Partition/Select für mehrere Sequenzen ist nicht Teil der STL. Trotzdem ist der Algorithmus auch jenseits des hier beschriebenen Einsatzgebietes nützlich, daher steht dieser sequentielle Algorithmus in der MCSTL auch separat zur Verfügung. Abweichend vom Fall mit nur einer Sequenz werden die k Sequenzen $S_0,..,S_{k-1}$ als **sortiert** vorausgesetzt. Alle Sequenzen zusammen enthalten $\sum_j |S_j| = n$ Elemente. Es ist für jede Sequenz ein Splitter so zu wählen, dass die r **global** kleinsten Elemente von den $n-r$ größ-

ten getrennt sind. Die MCSTL-Implementierung garantiert sogar, dass bei mehreren Elementen mit gleichem Schlüssel sowohl die Ordnung bezüglich Sequenzindex, als auch bezüglich Reihenfolge in der Sequenz berücksichtigt wird. Der asymptotisch optimale Algorithmus aus [16] dient als Grundlage für den MCSTL-Algorithmus.

Der Algorithmus ist an Binärsuche angelehnt. Zu Beginn werden von allen Sequenzen nur die mittleren Elemente ausgewählt. Auf diesen Elementen wird im Kleinen gemacht, was der Algorithmus im Großen erreichen soll. Es werden unter den k gewählten Elementen die $r \cdot k/n$ kleinsten Elemente in die "linke" Partition übernommen und die restlichen in die "rechte". So ist in jeder Sequenz bereits eine obere oder untere Grenze festgelegt (wobei eine Grenze immer am mittleren Element liegt und die andere an einem Ende der Sequenz). Diese Grenzen wachsen in den Folgenden Schritten aufeinander zu, während neue Elemente zur Auswahl hinzugenommen werden. Der Algorithmus ist fertig, wenn sich die Grenzen treffen.

Jeder Schritt besteht aus zwei Phasen. In der ersten Phase werden k neue Elemente betrachtet. Dazu wählt man sich in jeder Sequenz das Element aus, das in der Mitte zwischen den bisherigen Grenzen liegt. Diese Elemente werden zunächst mit dem (global) größten Element der linken und dem (global) kleinsten in der rechten Partition verglichen. Abhängig vom Ergebnis legt dieses Element dann die neue linke oder rechte Grenze fest. Ist ein neues Element kleiner als das Größte der linken Partition, ist es das neue Grenzelement der linken Partition in der eigenen Sequenz. Ist es größer als das kleinste Element der rechten Partition, begrenzt es die rechte Partition. Bei Elementen, die nicht durch die Elemente in einer der Partitionen klassifiziert werden können, muss man sich zu Beginn festlegen, ob immer die linke oder die rechte Partition sie erhält. Wenn möglich kann man auch den Mittelwert zwischen den Größen der Randelemente für die Zuordnung benutzen, um dieses Problem ganz zu vermeiden. Nach der ersten Phase muss das Verhältnis zwischen den beiden Partitionen nicht erhalten bleiben. Es sind aber höchstens k Elemente auf der falschen Seite. Es müssen also $O(k)$ Elemente aus einer Partition extrahiert werden und in die andere eingefügt. Benutzt man geeignete Datenstrukturen, so braucht man $O(\log k)$ Zeit um ein globales Randelement in einer Partition zu finden und die gleiche Zeit, um es in die andere Partition einzufügen. In der ersten Phase werden also k Elemente betrachtet und in der zweiten Phase benötigt man $O(k \cdot 2\log k)$ Zeit um das Größenverhältnis wiederherzustellen. Insgesamt ergibt sich also eine Laufzeit von $O(k\log k)$ pro Schritt. In jedem Schritt ist das mittlere Element zwischen den Grenzen die neue Grenze und der Abstand zwischen den Grenzen halbiert sich. Nimmt man vereinfachend an, dass Unterschiede in

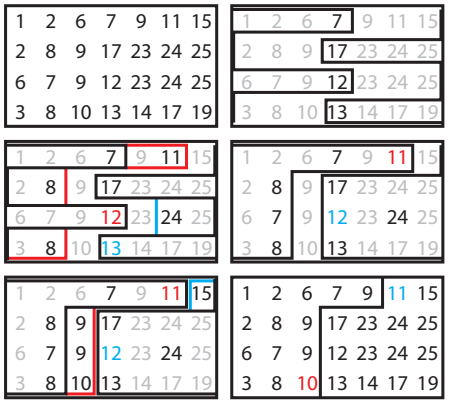


Abbildung 4.5: Beispiel für MCSTL-Algorithmus `multisequence_partition`; gesucht ist eine Aufteilung nach 14 Elementen; In grau sind Elemente dargestellt, die noch nicht betrachtet werden. Die Randelemente sind farblich hervorgehoben. Die 12 wird in der zweiten Phase von Schritt 2 von der linken in die rechte Partition übernommen, ebenso die 11 in der zweiten Phase von Schritt 3

der Sequenzlänge durch Padding gelöst werden, so benötigt man $\log \max_i S_i$ Schritte.

Analyse:

$$T(n,p) \in O(k \log k \cdot \log \max_j |S_j|)$$

4.1.7.2 Multiway Merge

Die `merge`(Mischen)-Operation erhält zwei Sortierte Folgen und mischt die Elemente zu einer einzelnen sortierten Folge. Auch hier verallgemeinert die MCSTL den Algorithmus von zwei auf k sortierte Folgen $S_0, .., S_{k-1}$ mit akkumulierter Länge n. Im ersten Schritt des Algorithmus werden die Folgen durch $(p-1)$-faches Abspalten der n/p kleinsten Elemente aufgeteilt. Hier kann jeder Prozessor eine Abspaltung übernehmen, indem er den sequentiellen `multisequence_partition`-Algorithmus aufruft. Nun hat man für jedes i zwischen 0 und $p-1$ (womöglich leere) Teilfolgen $S_{0,i}, .., S_{(k-1),i}$, über die alle Elemente mit Rang $i \cdot n/p$ bis $(i+1) \cdot n/p$ verteilt sind.

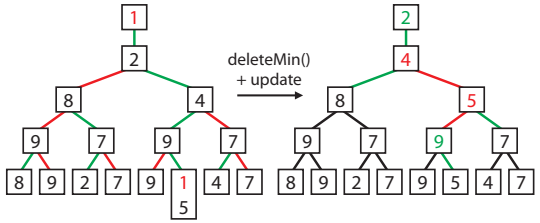


Abbildung 4.6: Loser Tree

Ab diesem Punkt muss jeder Prozessor für sein i ein sequentielles Merge auf den Listen $S_{0,i}$ bis $S_{(k-1),i}$ durchführen. Ein Verfahren, das diese Aufgabe in $O(\log k \cdot n/p)$ Zeit erledigt basiert auf Tournament Trees, genauer einem Loser Tree. Diese Bäume haben die Struktur einer Visualisierung von Turnierergebnissen im K.-o.-System. Die Blätter sind die zu vergleichenden Elemente, in den inneren Knoten stehen die "Verlierer" von Paarungen, in diesem Fall ist es das Kind mit dem größeren Schlüssel. Damit stehen in den inneren Knoten Duplikate von Blattknoten. Entfernt man im einmal gefüllten Baum den kleinsten Knoten, der als Sieger noch über der Wurzel steht, muss der Baum aktualisiert werden. Es ist für den Knoten bekannt, welches Blatt zu ihm gehört. Von dort aus reicht ein Aufstieg zur Wurzel, um den neuen kleinsten Knoten zu bestimmen. Die Laufzeit ein minimales Element zu bestimmen liegt also in $O(\log k)$ sobald der Baum einmal aufgebaut ist. Da ein Prozessor hier bei Elementen mit gleichem Schlüssel ganz leicht die Ordnung der ursprünglichen Sequenzen beibehalten kann, erbt der Algorithmus die Stabilität von `multisequence_partition`.

Analyse:

$$T(n,p) \in O(k \log k \cdot \log \max_j |S_j| + \frac{n}{p} \log k)$$

4.1.7.3 (Stable) Multiway Mergesort

Nun ist alles vorbereitet, was man für Multiway Mergesort in der MCSTL braucht. Im ersten Schritt des Verfahrens sortiert jeder Prozessor bis zu $\lceil n/p \rceil$ Elemente sequentiell. Die so entstandenen p sortierten Folgen werden durch `multiway_merge` zu einer gemischt. Das Ergebnis von `multiway_merge` ist eine neue, sortierte Folge, die noch an den Platz der ursprünglichen kopiert

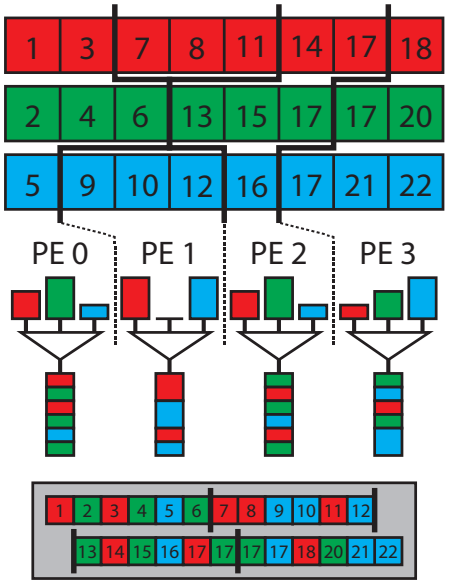


Abbildung 4.7: MCSTL-Algorithmus: `multiway_merge`

werden muss. Gegenüber Quicksort hat man also den Nachteil, dass das Sortieren nicht in-place ist, sondern vorübergehend doppelten Platz braucht. Zum Ausgleich benutzt man in der MCSTL für den ersten Schritt ein stabiles, sequentielles Sortierverfahren, damit für den ganzen Algorithmus die Stabilität erhalten bleibt. Daher wird dieser Algorithmus verwendet, wenn `stable_sort` aufgerufen wird.

Analyse:

$$T(n,p) \in O(\frac{n \log n}{p} + p \log p \cdot \log \frac{n}{p})$$

4.2 Parallele Prioritätslisten

Problembeschreibung: Eine Prioritätsliste ist eine abstrakte Datenstruktur. Ihr zugrunde liegt eine Menge M, deren Elemente einen Schlüssel besitzen müssen, der ihre Ordnung festlegt. Zu Beginn gilt $M := \emptyset$. Prioritätslisten

müssen die Operationen `insert` und `deleteMin` unterstützen. `insert` fügt ein Element e zu M hinzu. `deleteMin` liefert das Element mit dem kleinsten Schlüssel zurück und entfernt es aus M. Die Dauer jeder dieser Operationen kann von $n := |M|$ zum Zeitpunkt der Ausführung abhängen. Bei der hier betrachteten parallelen Prioritätsliste werden `insert` und `deleteMin` geändert. Bei der parallelen `insert*`-Operation fügt jeder Prozessor konstant viele Elemente ein. Bei dem parallelen `deleteMin*` werden die p kleinsten Elemente entfernt und jeder Prozessor erhält eines.

Benutzt man für die sequentielle Prioritätsliste Binary Heaps, so liegt der Zeitbedarf sowohl bei `insert`, als auch bei `deleteMin` in $O(\log n)$. Ein Ähnliches Ergebnis wünscht man sich auch für parallele Prioritätslisten. Es soll allerdings auch berücksichtigt werden dass insgesamt $\Theta(p)$ Elemente eingefügt oder entfernt werden. Die angestrebte Zeitschranke ist daher $O(\log n + \log p)$. Es gibt eine andere Definition des Problems für verteilte Algorithmen, bei der die Prozessoren asynchron einzelne `insert`- oder `deleteMin`-Operationen ausführen. Bei diesem Problem, das hier nicht betrachtet wird, muss nicht nur hohe Parallelisierung und damit Geschwindigkeit erreicht werden. Es ist auch wichtig, dass trotz Zugriffen zu verschiedenen Zeiten auf verschiedenen Prozessoren die Korrektheit erhalten bleibt.

Viele Algorithmen, die Prioritätslisten verwenden, können mit der hier betrachteten Parallelisierung nicht auf die gleiche Art verwendet werden. Algorithmen, wie Dijkstras Algorithmus für die Bestimmung kürzester Pfade, funktionieren nur durch die sequentielle Struktur normaler Prioritätslisten. Bei einer sequentiellen Implementierung von Dijkstras Algorithmus hat der gefundene Pfad zu einem Knoten stets minimale Kosten, da ein Weg nur erweitert wird, wenn es die günstigste Erweiterung unter den möglichen ist. Werden bei der Parallelisierung auch suboptimale Wege neben dem optimalen verfolgt, so kann später der Knoten auf einem besseren Weg erreicht werden. Ein typisches Problem, das von diesen Prioritätslisten profitiert, ist die Simulation von diskreten Ereignissen. Wird ein Ereignis simuliert, so können mehrere neue Ereignisse als Folge entstehen. Alle Prozessoren beziehen also Ereignisse aus der Prioritätsliste und fügen bei der Simulation anfallende wieder in die Liste ein, bevor sie ein neues Ereignis beziehen.

Eine einfache Umsetzung ist es die Prioritätsliste einem Prozessor zu überlassen. Er verwaltet dann eine sequentielle Prioritätsliste und beantwortet Anfragen aller anderen Prozessoren. Nicht nur dass diese Implementierung dank ihrer Überschaubarkeit gut zu debuggen ist, sie schlägt bei speziellen Anwendungen sogar viele verteilte Implementierungen. Durch eine zentralisierte Verwaltung beschränkt sich Kommunikation auf Anfragen und Antworten. Asynchrone Zugriffe werden auf natürliche Art und Weise geregelt.

Bei wenigen Zugriffen wird also Zeit für die zur Verwaltung nötige Kommunikation gespart. Bei der hier betrachteten Anwendung werden aber immer p Anfragen gleichzeitig gestellt, also wäre sowohl für `insert`, als auch für `deleteMin` $\Omega(p(T_{\text{start}} + \log n))$ Zeit für einen Prioritätslistenzugriff nötig.

4.2.1 Eine Anwendung: Branch-and-Bound

Ein weiteres typisches Einsatzgebiet, das von nun an durchgehend als motivierendes Beispiel dienen wird, ist das Branch-and-Bound Konzept. Für ein gegebenes Problem wird die optimale Lösung gesucht. Hier ist eine Lösung dann optimal, wenn sie minimale Kosten hat. Das Problem soll Teillösungen besitzen, durch die sich die Kosten nach unten abschätzen lassen. Ein solches Problem wäre die Suche nach einem Pfad in einem Graph mit positiven Kantengewichten, der vorgegebene Eigenschaften erfüllt und minimale Kosten unter allen solchen Pfaden hat. Eine Teillösung sind dann Pfade, die die gegebene Eigenschaft nicht erfüllen, sich aber geeignet erweitern lassen. Die Kosten für einen solchen Pfad sind eine untere Schranke für die Kosten einer daraus resultierenden Lösung. Ein Schritt in einem auf Branch-and-Bound basierenden Algorithmus besteht aus zwei Phasen. In der ersten Phase, der Branchphase werden aus einer Teillösung neue Teillösungen bzw. Lösungen generiert. Die Zeit für das Erzeugen neuer Teillösungen wird im Folgenden T_{X} genannt. Die Schranke für abgeleitete Teillösungen kann sich verbessern, also nur gegen die tatsächlich nötigen Kosten wachsen. Ein Absinken der Schranke ist nicht möglich, da sonst die Schranke der vorhergehenden Teillösung verletzt wäre. In der Boundphase werden aus allen generierten Teillösungen die entfernt, deren untere Schranke größer ist als die Kosten für die Beste gefundene Lösung. Ziel ist es, Lösungsansätze nicht weiterzuverfolgen, sobald sie die beste gefundene Lösung nicht mehr verbessern können.

Hier soll Branch-and-Bound als Suche nach einem Knoten mit minimalen Kosten in einem Baum $H(V, E)$ interpretiert werden. Die Knotenmenge V besteht dann aus Teillösungen und Lösungen. Die Kinder eines Knotens sind aus einer Teillösung abgeleitete Teillösungen und Lösungen für das Problem. Ein Knoten ist genau dann ein Blatt, wenn er eine Lösung des Problems repräsentiert. Es wird weiterhin eine Kostenfunktion $c(v)$ definiert. Repräsentiert v eine Lösung, so entspricht $c(v)$ den Kosten dieser Lösung. Ist v eine Teillösung, so ist $c(v)$ die zu der Teillösung gehörende untere Schranke für abgeleitete Lösungen. Die Kostenfunktion steigt als untere Schranke für die Kosten der Lösung auf einem Abwärtspfad monoton an. Das gesuchte Blatt, das minimale Kosten hat, wird als v^* bezeichnet. In $\tilde{V} \subseteq V$ sollen alle Knoten liegen, deren Kosten höchstens so groß sind, wie die von v^*. Es gilt

also $\tilde{V} := \{v \in V : c(v) \leq c(v^*)\}$ und es soll $m := \left|\tilde{V}\right|$ sein. Diese Menge enthält also alle Teillösungen, die in jedem Fall untersucht werden, da ihre untere Schranke noch unter den Kosten der optimalen Lösung liegt. Die Einschränkung des Baumes auf diese Knoten wird mit $\tilde{H}$ bezeichnet, die Höhe von $\tilde{H}$ als h.

4.2.1.1 Sequentielles Branch-and-Bound

Algorithmus 4.7: Sequentielles Branch-and-Bound

```
Q := (sequential) Priority Queue;
Q.insert(root node);
while(v := Q.deleteMin() is not a leaf){
  foreach(v' ∈ successors of v){
    Q.insert(v');
  }
}
process(solution v);
```

Erläuterungen: Im ersten Beispiel soll vorgeführt werden, wie sich eine Prioritätsliste benutzen lassen kann, um einen sequentiellen Branch-and-Bound-Algorithmus zu realisieren. Es wird in jedem Schritt der Knoten v mit dem niedrigsten $c(v)$ gewählt. Ist es kein Blatt, so werden in T_{X} Zeit die Nachfolger bestimmt und in die Prioritätsliste eingefügt. Da `deleteMin()` stets den Knoten mit den niedrigsten Kosten liefert, steckt die Bound-Phase in der Sortierung der Prioritätsliste. Das erste Blatt, das geliefert wird, hat minimale Kosten unter allen Knoten, die noch in der Prioritätsliste enthalten sind, also könnte kein Knoten mehr gezogen werden, der die Lösung verbessern würde. Es werden also alle Knoten, die nicht in $\tilde{V}$ liegen, sofort so eingefügt, dass sie erst nach dem besten Blatt gezogen werden würden. Alle m Knoten von $\tilde{V}$ müssen dagegen durchsucht werden und bis auf v^* liegen nur innere Knoten in $\tilde{V}$. Somit muss für jeden T_{X} Zeit aufgebracht werden, um Nachfolger zu generieren. Bei den Prioritätlistenoperationen wird zunächst m-faches `deleteMin()` benötigt, da jeder Knoten in $\tilde{V}$ extrahiert werden muss. Weiterhin gehören zu jedem Knoten in $\tilde{V}$ beschränkt viele Einfügeoperationen. Da Elemente außerhalb von $\tilde{V}$ nicht mehr betrachtet werden, ist das auch alles was an Einfügeoperationen notwendig ist. Es ist also für jedes der m Elemente eine durch eine Konstante beschränkte Anzahl an Prioritätslistenoperationen nötig, deren Zeitbedarf in $O(\log m)$ liegt.

Analyse:

$$T_{\text{seq}} \in m(T_{\text{X}} + O(\log m))$$

4.2.1.2 Branch-and-Bound nach Karp und Zhang

Algorithmus 4.8: Branch-and-Bound nach Karp und Zhang

```
Q := (sequential) Priority Queue;
if (i = 0){
  Q.insert(root node);
}
c* := ∞;
while(∃j : Q@j ≠ ∅){
  v := Q.deleteMin();
  if(c(v) ≥ c*){
    Q := ∅;
  }else{
    if(v is a leaf){
      process(new solution v);
      c* := c(v);
      c* := min_j c*@j;
    }else{
      foreach(v' ∈ successors of v){
        select random j ∈ {0,...,p−1};
        (Q@j).insert(v');
      }
    }
  }
}
```

Erläuterungen: Dieser Algorithmus aus [5] stellt eine Parallelisierung von Branch-and-Bound dar. Es besitzt jeder Prozessor eine eigene sequentielle Prioritätsliste. Im Wesentlichen funktioniert er wie der sequentielle Algorithmus. Eine Änderung ist, dass mehrere Knoten parallel durch verschiedene Prozessoren untersucht werden. Um gute Lastverteilung zu erreichen, werden die generierten Nachfolger von Knoten nicht nur in die eigene Prioritätliste eingefügt. Stattdessen verteilt man die neuen Knoten zufällig über die Prioritätslisten aller Prozessoren. Wenn ein Prozessor bei diesem Algorithmus einen Blattknoten findet, hat er nur das minimale Element von allen, die er besitzt. Auf globaler Ebene kann es noch immer bessere Lösungen geben.

Damit andere Prozessoren ihre Suche ebenfalls rechtzeitig abbrechen können, teilt jeder Prozessor anderen die Kosten eines gefundenen Blattes mit. Wenn in einem Schritt mehrere Blätter gefunden werden, so wird über Reduktion die niedrigste Kostenschranke bestimmt. So können Prozessoren, die nur teure Knoten besitzen ihre Prioritätsiste bereinigen, bis sie neue Knoten mit niedrigen Kosten erhalten.

Da $\tilde{H}$ noch immer durchsucht werden muss, ist m/p eine Untere Schranke für die Laufzeit. Eine zweite Laufzeitschranke ergibt sich aus der Tatsache, dass in $\tilde{H}$ auf dem längsten Pfad keine Parallelisierung möglich ist. Es ist also auch h eine untere Schranke. Damit beträgt auch die parallele Laufzeit mindestens $\max\{m/p, h\} \in \Omega(m/p + h)$. In [9] wird gezeigt, dass mit einer Wahrscheinlichkeit von $1 - m^{-k_1}$ höchstens $k_2(\frac{m}{p} + h + \log m + \log p)$ Schritte für die Ausführung benötigt werden, wobei k_1 und k_2 Konstanten sind. Diese Zeit ist asymptotisch optimal.

Analyse:

$$T(n, m, p) \in \tilde{O}((\tfrac{m}{p} + h)(T_{\text{X}} + T_{\text{coll}} + \log m))$$

4.2.1.3 Branch-and-Bound und parallele Prioritätslisten

Algorithmus 4.9: Branch-and-Bound mit Parallelen Prioritätslisten

```
Q := (sequential) Priority Queue;
Q.insert(root node);
c* := ∞;
while(Q ≠ ∅){
  v := Q.deleteMin*();
  if(c(v) < c*){
    if(v is a leaf){
      process(new solution v);
      c* := c(v);
      c* := min_j c*@j;
    }else{
      foreach(v' ∈ successors of v){
        Q.insert*(v');
      }
    }
  }
}
```

Erläuterungen: Jetzt möchte man bei der parallelen Implementierung so viel wie möglich auf die Prioritätslisten abwälzen. Hier erhält jeder Prozessor am Anfang des Schrittes eines der p kleinsten Elemente. Das reicht auch hier noch nicht, um sicher sein zu können, dass das erste gefundene Blatt auch das gesuchte ist. Ein anderer Prozessor könnte im gleichen Schritt einen inneren Knoten erhalten haben, der zu einem günstigeren Blatt expandiert. Es ist daher wieder nötig die Kosten der besten Lösung zu speichern, um festzustellen, wann eine Expansion nicht mehr nötig ist.

Die Abschätzung ist in diesem Fall etwas leichter. Es gilt immer noch die untere Schranke von $\max\{m/p, h\}$. Wenn in der Prioritätsliste in einem Schritt mindestens p Knoten aus $\tilde{H}$ liegen, dann kann ein p-tel der Knoten verarbeitet werden. Gäbe es nur solche Schritte, so wäre der Algorithmus nach m/p Schritten fertig. Es kann aber passieren, dass weniger als p Knoten aus $\tilde{H}$ in der Prioritätsliste liegen, dann werden auch Knoten betrachtet, die im sequentiellen Fall nicht betrachtet werden würden. Die in diesem Schritt betrachteten Knoten aus $\tilde{H}$ werden alle expandiert. Die Kindknoten dieser Bäume haben eine um 1 größere Höhe, also kann dieser Fall nur h Male eintreten. Insgesamt kann der Algorithmus durch Alternieren zwischen beiden Fällen höchstens $m/p + h$ Schritte benötigen. Wie gewohnt kann in jedem Schritt T_{X} Zeit für die Expansion der Knoten und $O(T_{\mathrm{PQueue}})$ Zeit für die Prioritätslistenoperation verbraucht werden.

Analyse:

$$T(n, m, p) \in (\tfrac{m}{p} + h)(T_{\mathrm{X}} + O(T_{\mathrm{PQueue}}))$$

Bisher muss bei Knoten, deren Nachfolger schnell generiert werden können, gewartet werden, bis auch die anderen Knoten bearbeitet wurden. Um das zu vermeiden, kann jeder Prozessor separate Prozesse für Prioritätslistenverwaltung und für Branch-and-Bound anlegen. Es wird nicht mehr gewartet, bis alle Prozessoren neue Knoten brauchen, die expandiert werden sollen. Sobald eine hinreichend große Anzahl an Prozessoren mit der Expansion fertig ist, wird ein `deleteMin*` ausgeführt und sie erhalten bereits neue Daten.

4.2.2 Paralleles Select

Bevor es an die Realisierung paralleler Prioritätslisten geht, wird hier eine Hilfsfunktion eingeführt, die für das parallele `deleteMin*` wichtig sein wird. Deswegen wird auch die Parallelisierung nicht allgemein, sondern nur für den später benötigten Spezialfall betrachtet.

Problembeschreibung: Gegeben ist eine Menge Q mit $|Q| = n$ und ein $k \leq n$. Suche die k kleinsten Elemente in Q.

Die hier beschriebene Lösung des Select-Problems angelehnt an einen Algorithmus aus [2]:

Algorithmus 4.10: Select

```
Select(Q,k){
  if(|Q| ≤ s){
    sort(Q);
    return first k elements of Q;
  }
  S := s random elements from Q;
  sort(S);
  u := element ranked ⌈(k/n)s + Δ⌉ in S;
  l := element ranked ⌊(k/n)s − Δ⌋ in S;
  Q< := {q ∈ Q : q < l};
  Q> := {q ∈ Q : q > l};
  Q' := Q \ (Q< ∪ Q>);
  if(|Q<| < k){
    return (Q< ∪ Select(Q', k − |Q<|));
  }else{
    if(|Q<| + |Q'| < k){
      return Q< ∪ Q' ∪ Select(Q>, k − (|Q<| + |Q'|));
    }else{
      return Select(Q<, k);
    }
  }
}
```

Erläuterungen: Die Idee hinter dem Algorithmus ist zunächst durch ein zufälliges Sample S abzuschätzen, welche Elemente mit hoher Wahrscheinlichkeit unter den k kleinsten liegen. Dazu wird aus dem Sample das Element gewählt, das das Sample in einem ähnlichen Verhältnis aufteilt, wie k das vollständige Q. Es müssen noch Abweichungen abgefangen werden, die unvermeidlich durch unzureichend repräsentatives Sample, sowie die grobe Aufteilung entstehen. Dazu wird ein um $-\Delta$ im Rang abweichendes Element l als untere und ein um Δ abweichendes Element u als obere Grenze ausgewählt. Diese zwei Grenzen legen eine Aufteilung von Q in drei Mengen fest. Die Menge Q' enthält alle Elemente mit Größe zwischen l und u, zu kleine Elemente kommen in $Q_<$ zu große in $Q_>$. Ist das k-te Element bei sortiertem Q zwischen den so gewählten Grenzelementen, so sind alle Elemente in $Q_<$

unter den kleinsten k. Weiterhin können alle Elemente in $Q_>$ ausgeschlossen werden. Es muss also noch überprüft werden, welche Elemente in Q' unter den ersten k sind. Rekursiv wird die Intervall so weit eingegrenzt, bis so wenige Elemente übrig sind, dass sie schnell sortiert werden können.

Die Wahrscheinlichkeit, dass in einem Rekursionsschritt das Problem auf Q' eingegrenzt werden kann hängt von s und Δ ab. Beide Parameter beeinflussen ebenfalls die Zeit, die ein einzelner Rekursionsschritt kostet. Ein größeres s erhöht den Sortieraufwand, macht das Sample aber repräsentativer. Den Parameter Δ zu vergrößern erhöht die Wahrscheinlichkeit, dass das k-te Element zwischen l und u liegt, vergrößert aber auch das Q' auf das eingeschränkt wird.

Für die Parallelisierung wird hier nur ein Sonderfall betrachtet, bei dem $n = p \log p$ und $k = p$ gilt. Dieser Fall wird später auftreten. Für die Samplegröße s bietet sich im parallelen Fall $\sqrt{p}$ an, da so viele Elemente durch schnelles Ranking (4.1) mit wenigen kollektiven Kommunikationsoperationen sortiert werden können. Bei $p \log p$ Elementen ist die Samplegröße dabei noch ausreichend, um in jedem Iterationsschritt bei $\Delta \in \Theta(p^{1/4+\epsilon})$ mit hoher Wahrscheinlichkeit wie gewünscht eine Einschränkung auf Q' zu erreichen. Beweisen lässt sich das mit Chernoff-Schranken. Ähnlich wie bei SampleSort (4.5) zeigt man, dass bei Ziehen der Samples mit Zurücklegen die Wahrscheinlichkeit, dass zu viele zwischen u und l liegen sehr niedrig ist. Damit sind von anfänglich $p \log p$ Elementen bereits nach $O(1)$ Rekursionsschritten höchstens noch $\sqrt{p}$ Elemente übrig, die wieder schnell sortiert werden können. Ist T_{coll} die Zeit für eine kollektive Kommunikation, so braucht der Algorithmus in jeder Rekursionsebene $O(T_{\text{coll}})$ Zeit zum Sortieren des Samples mit schnellem Ranking und ebenfalls $O(T_{\text{coll}})$ für eine Präfixsumme, mit der die Elemente den drei Mengen zugeordnet werden. Auf eine Umverteilung wird nach der Präfixsumme verzichtet, da zufällige Verteilung bereits eine gute Lastverteilung wahrscheinlich macht. Nach $O(1)$ Ebenen sind noch $\sqrt{p}$ Elemente vorhanden, die wieder in $O(T_{\text{coll}})$ sortiert werden.

Analyse:

$$T(p \log p, p) \in \tilde{O}(T_{\text{coll}})$$

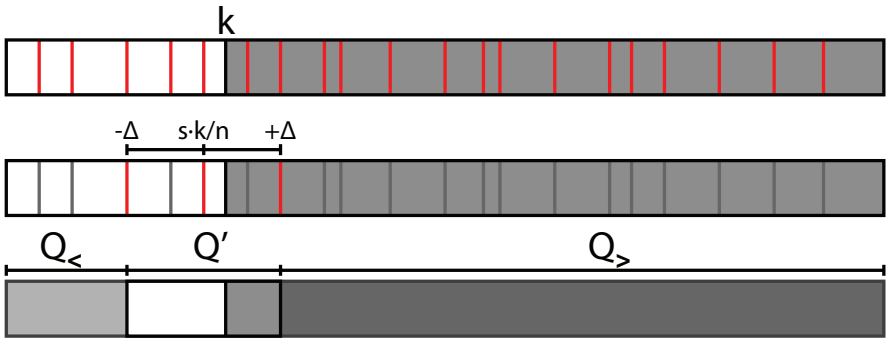


Abbildung 4.8: Eine Select-Rekursion

4.2.3 Parallele Prioritätslisten

Die nun beschriebene Prioritätsliste nach [11] setzt wieder auf eine sequentielle Prioritätsliste für jeden Prozessor. Für `insert*` greift man die Idee aus dem Algorithmus von Karp und Zhang auf. Randomisierung hilft auch hier dabei die Knoten einigermaßen gleichmäßig über die lokalen Prioritätslisten zu verteilen. Chernoff-Schranken lassen sich anwenden, um zu zeigen, dass bei einer `insert*`-Operation mit hoher Wahrscheinlichkeit höchstens $O(\frac{\log p}{\log \log p})$ neue Elemente in eine lokale Prioritätsliste eingefügt werden. Da die Elemente gleichmäßig verteilt sein sollten, kostet das Einfügen eines Elementes in eine lokale Prioritätsliste $O(\log \frac{n}{p})$ Zeit. Zusammen mit dem kollektiven Nachrichtenaustausch, der stattfinden muss, um die Elemente auf die Prozessoren zu bringen, in deren Prioritätsliste sie gehören, ergibt sich für ein `insert*` ein Zeitbedarf von

$$T_{\text{coll}} + \tilde{O}(\frac{\log p}{\log \log p} \cdot \log \frac{n}{p})$$

Bessere lokale Prioritätslisten können diese Schranke senken, es sollte aber darauf geachtet werden, dass bessere amortisierte Schranken möglicherweise nicht reichen. Es ist keine Verbesserung zu erwarten, wenn jeder Prozessor für sich nur in seltenen Fällen lange braucht, aber bei jedem `insert*` für einen der Prozessoren dieser seltene Fall eintritt.

Eine komplexere Funktion stellt bei dieser Realisierung `deleteMin*` dar. Anders als bei Karp und Zhang sollen die Prozessoren nicht mehr nur das kleinste Element ihrer Prioritätsliste erhalten. Nach `deleteMin*` soll jeder Prozessor eines der p kleinsten Elemente aus der Vereinigung über alle lokalen Prioritätslisten besitzen. Eine erste Version könnte dann wie folgt aussehen:

Algorithmus 4.11: deleteMin*

```
deleteMin*(Q_1, n, p){
  Q_0 := the O(log p) smallest elements of Q_1;
  M := select(∪_j Q_0@j, p);
  enumerate M = {e_0, ..., e_{p-1}};
  assign e_j to PE j;
  if(max_j e_j > min_k Q_1@k){
    expensive special case treatment;
  }
  empty Q_0 back into Q_1;
}
```

Im ersten Schritt werden aus jeder der p lokalen Listen Q_1 die kleinsten $\log p$ Elemente herausgenommen und in eine sortierte Liste Q_0 abgelegt. Bei $O(\log \frac{n}{p})$ Zeit für ein lokales `deleteMin` ist die Gesamtzeit zum erzeugen der Q_0 in $O(\log p \log \frac{n}{p})$. Es ist sehr wahrscheinlich, dass in der Vereinigung über alle lokalen Q_0 die p kleinsten Elemente liegen. Paralleles Select (4.2.2) liefert die p kleinsten Elemente aus allen lokalen Q_0. Wie bereits gezeigt, ist $\tilde{O}(T_{\text{coll}})$ Zeit nötig, um die kleinsten p Elemente aus insgesamt $p \log p$ zu bestimmen. Die so bestimmte Menge der p kleinsten Elemente wird mit M bezeichnet. Über eine Präfixsumme lassen sich die Elemente in M durchnummerieren. Diese Elemente werden in einer weiteren kollektiven Operation an die entsprechenden Prozessoren ausgeliefert. Präfixsumme und Verteilen der Elemente kosten jeweils $O(T_{\text{coll}})$ Zeit. Es ist wieder $O(T_{\text{coll}})$ Zeit nötig, um zu prüfen, ob in einem Lokalen Q_1 noch immer mindestens ein Element ist, das kleiner ist, als das größte in M. Wenn das passiert, dann hat es nicht gereicht nur die $\log p$ kleinsten Elemente aus jeder lokalen Prioritätsliste zu berücksichtigen. Dieser Sonderfall kostet dann polynomielle Zeit. Anschließend sind die p kleinsten Elemente bekannt und was noch von den lokalen Q_0 übrig ist, muss wieder in Q_1 einsortiert werden. Da auf jedes Q_0 ein Element in M kommt, muss zumindest ein Prozessor $O(\log p)$ Elemente in Q_1 mischen. Damit kostet es $O(\log p \log \frac{n}{p})$ Zeit Q_0 wieder in Q_1 zu mischen.

Eine Verbesserung erreicht man dadurch, dass man Q_0 nicht mehr für jede Operation komplett neu bestimmt. Da sich an einem lokalen Q_1 bei jedem Einfügen relativ wenig ändert, ändert sich auch auch an Q_0 nicht viel. Für die Elemente von Q_0 bedeutet es vor allem, dass viele immer wieder neu aus Q_1 gezogen und einsortiert werden müssen. Um diesen Aufwand zu senken macht man die Aufteilung in Q_0 und Q_1 permanent. Dabei bleibt Q_1 eine sequentielle Prioritätsliste und Q_0 ein sortiertes Array. Neue Elemente werden durch `insert*` nun in Q_0 eingefügt. Dazu werden die einzufügenden

Elemente sortiert und durch ein Merge mit dem sortierten Q_0 kombiniert. Um Q_0 zu beschränken, wird es nach $\log p$ Einfügeoperationen wieder in Q_1 geleert. Mit Chernoff-Schranken lässt sich wieder zeigen, dass bis dahin mit hoher Wahrscheinlichkeit $O(\log p)$ Elemente in Q_0 sind. Das Verschmelzen mit Q_1 hat dann Kosten $O(\log p \cdot \log \frac{n}{p})$. Da alle Prozessoren das zur gleichen Zeit tun, lassen sich hier amortisierte Schranken anwenden und über die $\log p$ Schritte gemittelt bleibt noch $O(\log \frac{n}{p})$ Zeitbedarf übrig. Da man von $O(\log p)$ Elementen zu jedem Zeitpunkt ausgeht, kann auch die Mergeoperation beim Einfügen abgeschätzt werden. Es werden bis zu $O(\frac{\log p}{\log \log p})$ neue Elemente mit $O(\log p)$ Elementen in Q_0 gemischt, das hat Zeitbedarf in $O(\log p)$. Dieser logarithmische Term wird durch die Zeit für die kollektive Nachrichtenoperation dominiert, in der zu Beginn die Elemente zufällig über die Prozessoren verteilt werden. Ein `insert*` dauert also

$$O(T_{\text{coll}} + \log \frac{n}{p})$$

Zeit, zusammengesetzt aus dem Verteilen und dem amortisierten Leeren von Q_0, das alle $\log p$ Schritte stattfindet.

Bei jeder `deleteMin*`-Operation wird zunächst geprüft, ob die p kleinsten Elemente in Q_0 liegen. Dafür muss nur in einer kollektiven Operation bestimmt werden, ob mindestens p Elemente in Q_0 existieren, die kleiner sind, als das minimale in Q_1. Gilt das nicht, so nehmen alle Prozessoren das kleinste Element aus ihrem Q_1 zu Q_0 hinzu. Das zufällige Verteilen beim Einfügen macht es sehr wahrscheinlich, dass es reicht $O(1)$ Male das kleinte Element aus jedem Q_1 in das zugehörige Q_0 zu verschieben, bis keines der p kleinsten noch in einem Q_1 liegt. Wenn also die Anzahl der `insert*`-Aufrufe der für `deleteMin*` entspricht, so werden auch hier mit hoher Wahrscheinlichkeit nur $O(\log p)$ Elemente in Q_0 aufgenommen. Insgesamt ändert sich also nichts daran, dass Q_0 eine erwartete Größe von $O(\log p)$ hat. Es ergibt sich die folgende, verbesserte Realisierung:

Algorithmus 4.12: deleteMin*

```
deleteMin*(Q_0, Q_1, n, p){
  while(|{e ∈ ∪_j Q_0@j : e < min(∪_j Q_1@j)}| < p){
    Q_0 := Q_0 ∪ {Q_1.deleteMin()};
  }
  M := select(Q_0, p);
  enumerate M = {e_0, ..., e_{p-1}};
  assign e_j to PE j;
}
```

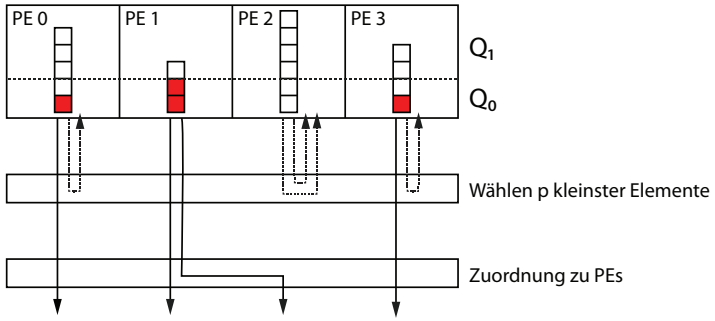


Abbildung 4.9: `deleteMin*`

Damit lässt sich auch für `deleteMin*` eine Laufzeit erreichen, die mit hoher Wahrscheinlichkeit in $O(T_{\text{coll}} + \log \frac{n}{p})$ liegt. Die Zeit für Select, Durchnummerieren und Verteilen der kleinsten p Elemente ändert sich nicht. Es muss aber in jedem Schritt nur $O(1)$ Male auf Q_1 zugegriffen werden. Ein Zugriff hat die Kosten $\log \frac{n}{p}$ und damit liegt die Zeit für diese Variante von `deleteMin*` in

$$\tilde{O}(T_{\text{coll}} + \log \frac{n}{p})$$

Analyse:

$$T_{\text{insert}*}(n,p) \in \tilde{O}(T_{\text{coll}} + \log \tfrac{n}{p})$$
$$T_{\text{deleteMin}*}(n,p) \in \tilde{O}(T_{\text{coll}} + \log \tfrac{n}{p})$$

Weiter optimieren lassen sich noch Operationen auf lokalen Prioritätslisten. An den Schranken ändern diese Optimierungen aber nichts. Da alle $\log p$ Schritte Q_0 geleert wird, kann es sich beispielsweise lohnen Listenstrukturen für Q_1 zu wählen, die schnelle Mergeoperationen mit einer sortierten Liste ermöglichen. Umgekehrt kann Q_0 nach dem Leeren auch wieder mit Elementen gefüllt werden. Dafür kann Q_1 ebenfalls so angepasst werden, dass auch `deleteMin` für Mengen unterstützt wird.

4.3 Weiteres zur MCSTL

4.3.1 Find

Problembeschreibung: Gegeben ist ein Array $a\,[0..n-1]$ mit n Elementen

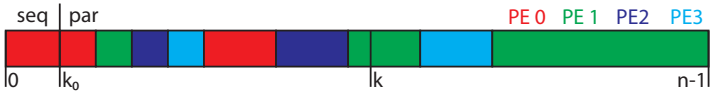


Abbildung 4.10: Beispiel für den `find` MCSTL-Algorithmus

und ein Prädikat P. Gesucht ist das minimale k für das das $P(a\,[k])$ wahr wird.

Das allgemeine Problem wird in der STL von `find_if` gelöst wo ein übergebenes Prädikat zu erfüllt werden soll. Aber auch die Spezialfälle `find` und `mismatch` lassen sich über Gleichheit als Prädikat realisieren.

Sequentiell braucht das Problem $O(k)$ Zeit, man geht einfach durch das Array und überprüft, ob das gesuchte Element gefunden ist. Bei der naiven Parallelisierung wird das Array gleichmäßig unter den Prozessoren aufgeteilt, jeder sucht in seinem Abschnitt nach einem geeigneten Element. Gibt es im ganzen Array nur eines, so liegt der Aufwand in $\Omega(n/p)$, da nur ein Prozessor vorzeitig abbrechen kann. Statt aber für einen frühen Abbruch zusätzliche Kosten zur Überprüfung zu investieren, überlässt man diese Aufgabe der Arbeitsaufteilung.

Sollte das gesuchte Element bereits sehr früh vorkommen, so lohnt es sich womöglich gar nicht mit der Parallelisierung zu beginnen, da dort die Kosten für die Initialisierung möglicherweise höher sind, als für sequentielle Bestimmung. Deswegen werden die ersten k_0 Elemente, für ein festes k_0, sequentiell durchsucht. Von da an nimmt sich jeder Prozessor einen immer größer werdenden Block aus dem Array. Dazu wird eine Variable für den Index des ersten Elementes angelegt, das keinem Prozessor zugewiesen ist. Über `fetch-and-add` kann der Zugriff auf diese Variable kontrolliert werden. So wird sichergestellt, dass nur ein Prozessor sich einen neuen Block sichert. Wenn ein Prozessor seinen Block durchsucht hat, beginnt er einen neuen, wobei die beanspruchte Blocklänge jedes Mal um einen konstanten Faktor erhöht wird, bis sie eine maximallänge erreicht hat. Hat er das gesuchte Element gefunden, beansprucht er alle verbleibenden Elemente für sich. So kann eine Laufzeit von $O(k/p)$ garantiert werden. Im schlimmsten Fall haben alle Prozessoren einen neuen Block mit einer um Faktor c vergrößerten Länge begonnen und ein Prozessor trifft gleich beim ersten Element auf die Lösung. Damit muss jeder Prozessor insgesamt $c(k-1)/p + B$ Elemente bearbeiten, wobei B die anfängliche Blocklänge ist. Nur einer ist nach $(k-1)/p+1$ fertig, was die anderen Prozessoren aber erst bemerken, wenn sie ihre neuen Blöcke abgearbeitet haben.

Analyse:

$$T(n,p) \in O(\frac{k}{p})$$

4.3.2 Random Shuffle

Aufgabe der `random_shuffle`-Operation ist es, eine vorgegebene Folge zufällig zu mischen. Mit dem Sortierproblem gemeinsam hat dieses Problem, dass viele Daten schnell umverteilt werden müssen. Auch wenn das Tauschen der Elemente zufällig stattfindet und nicht strukturiert, bleiben viele Eigenschaften erhalten. So hat `random_shuffle` als paralleler Algorithmus sowohl bei verteiltem Speicher und dem Kommunikationsaufwand, als auch bei gemeinsamem Speicher und konkurrierenden Cachezugriffen vergleichbare Anforderungen.

Der STL-Ansatz besteht hier daraus, einmal durch die Elemente zu iterieren und jedes dabei mit einem zufällig gewählten anderen zu tauschen. Ein solches Vorgehen ist alles andere als cacheeffizient, da immer wieder auf verschiedene Positionen in der Sequenz geschrieben wird. Stattdessen wird für die MCSTL in drei Schritten auf Basis des Mersenne Twister zufällig gemischt. Zunächst teilt jeder Prozessor für sich seine Elemente zufällig k lokalen Buckets zu. Hier wird nur sequentiell gelesen und in k Arrays sequentiell geschrieben. Die kp so entstandenen Buckets werden im zweiten Schritt zu insgesamt k zusammengefasst, indem alle p lokalen Buckets mit dem gleichen Index zu einem globalen vereinigt werden. Im dritten Schritt wird innerhalb dieser k Buckets zufällig permutiert. Zum Schluss können alle Buckets wieder zu einer Folge zusammengesetzt werden.

Analyse:

$$T(n,p) \in O(\frac{n}{p} + p)$$

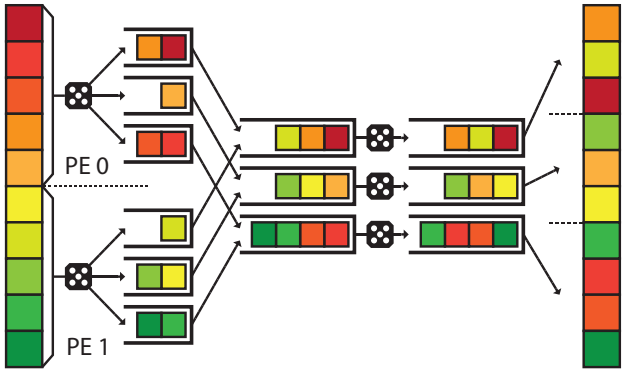


Abbildung 4.11: Beispiel für den `random_shuffle` MCSTL-Algorithmus

5 Graphenalgorithmen

5.1 List Ranking

Den Anfang macht ein Problem, das an der Grenze zu den Graphenalgorithmen steht. Ziel ist es eine verkettete Liste in ein Array zu konvertieren. Eine verkettete Liste kann durchaus als spezieller Graph betrachtet werden. Im Zusammenhang mit parallelen Algorithmen ist das Problem auch besonders interessant, da lange Pfade sich schlecht parallelisieren lassen. In Arrays lassen sich dagegen viele Probleme gut unter den Prozessoren aufteilen, selbst wenn die Reihenfolge der Elemente eine wichtige Rolle spielt. Um überhaupt Parallelisierung zu ermöglichen wird hier das Problem in einer Form angenommen, in der zwar bereits über einen Index auf alle Elemente zugegriffen werden kann, dieser Index aber nichts über die Reihenfolge in der Liste aussagt.

Problembeschreibung: Gegeben ist eine Liste L mit n Elementen. Die Funktion $S(j)$ liefert den Index des Nachfolgers gemäß der Listenordnung für das Element mit Index j. Die Funktion $P(j)$ liefert analog dazu den Vorgängerindex für das Element mit Index j. Für das erste Element h in der Liste ist $P(h) := h$, für das letzte Element t gilt $S(t) := t$. Zu bestimmen ist $R(j) := \min\{k : S^k(j) = t\}$ für jedes Element j. Es wird also eine Rangfunktion gesucht, die für jedes Element den Abstand vom Listenende zurückgibt. Ein Array $A[0..n-1]$ erhält man mit dieser Funktion ganz leicht, indem man die Zuordnung $A[n-1-R(j)] := L[j]$ für jedes j anwendet.

5.1.1 Pointer Chasing

Algorithmus 5.1: Pointer Chasing

```
j := t;
for(k := 0; k < n-1; k++){
  R(j) := k;
  j := P(j);
}
```

Erläuterungen: Die sequentielle Lösung für das Problem bedarf kaum einer Erklärung. Ist das letzte Element der Liste t gefunden, so folgt man nur noch den Vorgängereferenzen und nummeriert die Elemente dabei durch. Ist die suche nach dem Listenende nötig, so lässt sie sich leicht parallelisieren. Das Traversieren der Liste dagegen bietet keine natürlichen Angriffspunkte für Parallelisierung.

Analyse:

$$T(n) \in O(n)$$
$$W(n) \in \Theta(n)$$

5.1.2 Doubling mit PRAM

Ein wichtiger Ansatz lineare Strukturen parallel zu untersuchen nennt sich Doubling. Dabei profitiert man in jedem Schritt von den Abkürzungen, die andere Prozessoren finden, um logarithmische Zeit zu erreichen:

Algorithmus 5.2: Doubling

```
Q(i) := S(i);
if (S(i) = i) {
   R(i) := 0;
} else {
   R(i) := 1;
}
while (S(Q(i)) ≠ Q(i)) {
   R(i) := R(i) + R(Q(i));
   Q(i) := Q(Q(i));
}
```

Erläuterungen: Dieser Algorithmus ist auf eine CREW PRAM mit $p = n$ Prozessoren ausgelegt. Der Prozessor mit dem Index i bestimmt dabei den Rang $R(i)$ für ein Listenelement. Das Verfahren, das dafür benutzt wird, wird Doubling genannt. In jedem Schritt wird durch jeden Prozessor in ein eigenes $Q(i)$ eine Abkürzung gespeichert. $R(i)$ enthält die Länge des Pfades, der über diese Abkürzung übersprungen wird. Zu Beginn wird der direkte Nachfolger dort gespeichert und entsprechend ist $R(i)$ für diesen Fall 1, wenn das Element nicht am Ende der Liste liegt. In jedem Schritt wird der Pfad, der durch $Q(i)$ übersprungen wird, verlängert. Dazu wird beim neuen Pfad das Element mit dem Index $Q(i)$ übersprungen. Stattdessen wird die Abkürzung direkt auf $Q(Q(i))$ umgeleitet. Die Länge des übersprungenen Pfades setzt sich aus der Länge des Pfades bis $Q(i)$ und der Länge $R(Q(i))$ von dort aus bis $Q(Q(i))$ zusammen. Wenn $Q(Q(i))$ nicht auf das letzte Element der Liste zeigt, dann verdoppelt sich so der Pfad, der durch $Q(i)$ übersprungen wird, sonst steht in $R(i)$ anschließend das Ergebnis. Insbesondere sind für das erste Element in der Liste $\Theta(\log p)$ Schritte nötig, bis in $R(i)$ das Ergebnis steht.

Analyse:

$$T(n,n) \in \Theta(\log n)$$
$$W(n,n) \in \Theta(n \log n)$$

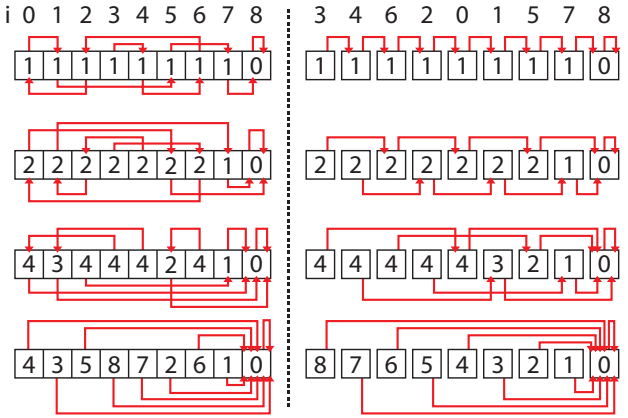


Abbildung 5.1: Doubling mit PRAM; links als Array dargestellt, rechts die Einträge in Listenreihenfolge

5.1.3 Paralleles List Ranking durch Entfernen unabhängiger Teilmengen

Algorithmus 5.3: List Ranking by Independent Set Removal

```
independentSetListRanking(n, S, R){
  if(p > n){
    doubling(n, S, R);
    return;
  }
  I :⊂ {0..n−1}, j ∈ I ⇒ S(j) ∉ I;
  f : {0..n−1} \ I → {0..n−1−|I|}  bijective;
  //remove independent set:
  parallel_foreach(i ∉ I){
    if(S(i) ∈ I){
      S'(f(i)) := S(S(i));
      R'(f(i)) := R(i) + R(S(i));
    }else{
      S'(f(i)) := S(i);
      R'(f(i)) := R(i);
    }
  independentSetListRanking(n − |I|, S', R');
  parallel_foreach(i ∉ I){
    R(i) := R'(f(i));
  }
  parallel_foreach(i ∈ I){
    R(i) := R(i) + R'(f(S(i)));
  }
}
```

Erläuterungen: Man könnte Doubling auf allgemeine p übertragen, indem jeder Prozessor $\frac{n}{p}$ Elemente übernimmt. Dadurch erhält man eine Laufzeit von $O(\frac{n}{p} \log n)$. Es liefert aber eine Verbesserung nicht das gesamte Doublingverfahren für jedes Element durchzuführen. Es wird stattdessen zunächst in jedem Schritt eine Teilmenge I entfernt, wobei für kein Element j von I der Nachfolger $S(j)$ auch in I liegen darf. Ein solches I wird als unabhängige Teilmenge (Independent Set) bezeichnet. Insbesondere liegt das letzte Element nicht in I, da es sein eigener Nachfolger ist. Die Eigenschaft, dass keine benachbarten Elemente in I liegen, sichert zu, dass zunächst für jedes Element, dessen Nachfolger oder Vorgänger in I liegt, in einem Schritt eine "Abkürzung" gelegt werden kann, die das Element in I überspringt. Wie bei Doubling muss sich wieder die Anzahl der so übersprungenen Elemente

als Pfadlänge gemerkt werden. Auch hier addieren sich beim Überspringen die Längen der zusammengefassten Pfade auf. Rekursives Entfernen unabhängiger Teilmengen liefert irgendwann eine Restmenge, die noch p oder weniger Elemente enthält. Für diese Elemente kann wie gewohnt Doubling verwendet werden, um die finalen Ränge der Elemente zu bestimmen, da die Längen der Pfade, die über als unabhängige Teimengen entfernte Elemente verlaufen, bereits bekannt sind. Die unabhängigen Teilmengen können dann in umgekehrter Reihenfolge wieder hinzugenommen werden. Für jedes Element einer unabhängigen Teilmenge kann – wieder dank der Eigenschaft, dass der Nachfolger für ein Element nicht enthalten ist – in einem Schritt aus dem bekannten finalen Rang des Nachfolgers ebenfalls der finale Rang bestimmt werden. Sobald also auf diese Weise alle unabhängigen Teilmengen wieder integriert wurden, ist der Rang für jedes Element bekannt. Es fällt auf, dass man sich hier gegenüber der naiven Parallelisierung des Doubling gespart hat diese Elemente in jedem Schritt zu aktualiseren.

Für die Analyse ist wichtig, wie eine unabhängige Teilmenge bestimmt wird und wie gut sich die ursprüngliche Liste dadurch verkleinern lässt. Beides soll jetzt präzisiert werden. Das Problem eine unabhängige Teilmenge der Größe $\frac{n}{2}$ zu bestimmen lässt sich schlecht parallelisieren. Zu gunsten der Laufzeit wird daher keine optimale Lösung verlangt und dann hilft wieder einmal Randomisierung weiter. Jeder Prozessor ist für $\frac{n}{p}$ Elemente verantwortlich. Er wählt für jedes davon zunächst zufällig 1 oder 0. Die Elemente, bei denen eine 1 gewählt wurde, sind Kandidaten für I. Jetzt geht jeder Prozessor seine Elemente noch einmal durch und setzt das Kandidatenbit bei allen Elementen j auf 0, deren Nachfolger $S(j)$ auch eine 1 zugewiesen wurde. Von allen Ketten aus Elementen mit Kandidatenbit 1 darf also nur das letzte seines behalten. Nach dieser Bereinigung erhält man ein I, das den Anforderungen entspricht. Da die Bits zufällig gesetzt wurden, ist vor der Bereinigung jede der vier möglichen Kombinationen von eigenem und Nachfolgerbit gleich wahrscheinlich. Das ergibt eine Wahrscheinlichkeit von $\frac{1}{4}$ dass ein Element zu I hinzugenommen wird. Es ist also $|I| \approx \frac{n}{4}$ zu erwarten. Tatsächlich kann I sehr klein werden, durch wiederholtes Ausführen des Verfahrens kann aber ein $|I| > \frac{n}{5}$ garantiert werden. Eine Ausführung benötigt $O(\frac{n}{p})$ Zeit, in der jeder Prozessor einen gleichen Teil der Elemente zweimal durchgeht. Da nach konstant vielen Versuchen $|I| > \frac{n}{5}$ zu erwarten ist, bleibt es bei einer erwarteten Laufzeit von $O(\frac{n}{p})$ für die Bestimmung einer unabhängigen Teilmenge I.

Die Laufzeit für einen Rekursionsschritt wird durch Terme in $O(\frac{n}{p})$ und in $O(\log p)$ bestimmt. Bei der letzten Rekursion mit bis zu p Elementen fällt $O(\log p)$ Zeit für Doubling an. Bei den vorhergehenden Rekursionen mit

mehr Elementen muss zunächst I mit $O(\frac{n}{p})$ Zeitbedarf bestimmt werden. Die Elemente, die nicht zu I gehören müssen einen neuen Index erhalten, damit es keine Lücken gibt. Diese Aufgabe wird von einer Präfixsumme erfüllt, wenn die Werte, die durch sie aufsummiert werden sollen, bei allen Elementen aus I 0 sind und 1 für alle, die nicht in I liegen. Das benötigt wieder $O(\frac{n}{p})$ Zeit für lokale Operationen auf einzelnen Elementen und $O(\log p)$ für Austausch zwischen den Abschnitten. Alle weiteren Operationen in einem Schritt haben wieder einen Aufwand von $O(\frac{n}{p})$, da auch hier alle Elemente aktualisiert werden müssen und die Prozessoren diese Arbeit gleichmäßig untereinander aufteilen können. Bei mehr als p Elementen in einem Rekursionsschritt wird der Algorithmus mit $\frac{4}{5}n$ Elementen rekursiv aufgerufen. Insgesamt ergibt sich damit folgende Rekurrenzgleichung:

$$T(n,p) \in O(\frac{n}{p} + \log p) + T(\frac{4}{5}n, p)$$

Die Rekursionstiefe liegt in $O(\log \frac{n}{p})$, also gilt:

$$\begin{aligned} T(n,p) &\in O(\sum_{k=0}^{\log \frac{n}{p}} (\frac{n(\frac{4}{5})^k}{p} + \log p)) \\ &= O(\frac{n}{p} \underbrace{\sum_{k=0}^{\log \frac{n}{p}} (\frac{4}{5})^k}_{\leq \sum_{k=0}^{\infty} (\frac{4}{5})^k = \frac{1}{1-\frac{4}{5}}} + \log \frac{n}{p} \log p) \\ &= O(\frac{n}{p} + \log \frac{n}{p} \log p) \end{aligned}$$

Bei $p = \frac{n}{\log n \log\log n}$ ergibt sich ein besonders eleganter Fall. Dann gilt nämlich:

$$\begin{aligned} T(n,p) &= T(n, \frac{n}{\log n \log\log n}) \\ &\in O(n\frac{\log n \log\log n}{n} + \log(n\frac{\log n \log\log n}{n}) \log \frac{n}{\log n \log\log n}) \\ &= O(\log n \log\log n + \log(\log n \log\log n) \log \frac{n}{\log n \log\log n}) \\ &= O(\log n \log\log n + \underbrace{(\log\log n + \log\log\log n)}_{\in O(\log\log n)} \underbrace{\log \frac{n}{\log n \log\log n}}_{\in O(\log n)}) \\ &= O(\log n \log\log n) \end{aligned}$$

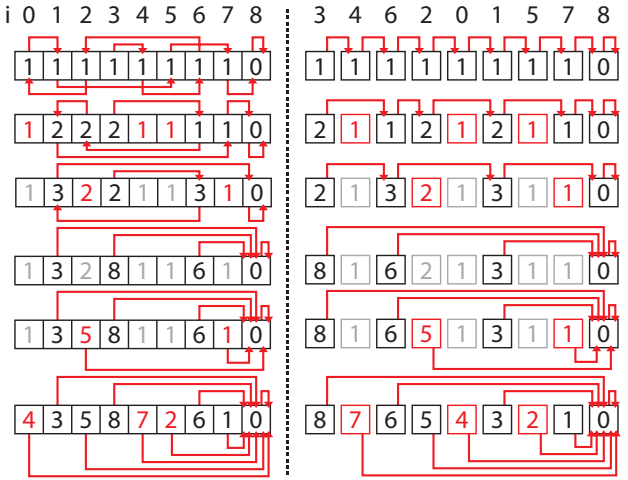


Abbildung 5.2: Paralleles List Ranking durch Entfernen unabhängiger Teilmengen; links als Array dargestellt, rechts die Einträge in Listenreihenfolge

Bei der Arbeit ergibt sich durch die Nutzung dieses Algorithmus und von $\frac{n}{\log n \log\log n}$ Prozessoren sogar eine Verbesserung gegenüber dem PRAM-Doubling mit n Prozessoren:

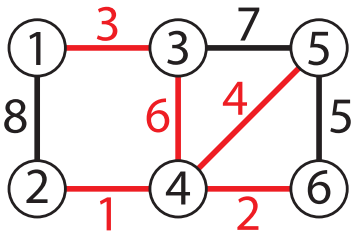
$$\begin{aligned} W(n,p) &= W(n, \frac{n}{\log n \log\log n}) \\ &= \frac{n}{\log n \log\log n} T(n, \frac{n}{\log n \log\log n}) \\ &\in \frac{n}{\log n \log\log n} O(\log n \log\log n) \\ &= O(n) \end{aligned}$$

Analyse:

$$\begin{aligned} &T(n,p) \in O(\frac{n}{p} + \log \frac{n}{p} \log p) \\ &T(n, \frac{n}{\log n \log\log n}) \in O(\log n \log\log n) \\ &W(n, \frac{n}{\log n \log\log n}) \in O(n) \end{aligned}$$

5.2 Minimum Spanning Tree

Das Problem des Minimalen Spannenden Baums (MST) ist weit verbreitet, als ein einfaches, nicht triviales Graphenproblem. Deshalb wird es hier stellvertretend für alle Graphenprobleme behandelt.

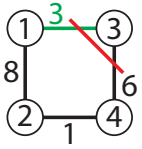


Ein Graph mit seinem MST

Beim MST-Problem ist ein zusammenhängender ungerichteter Graph $G = (V, E)$ durch seine Knoten V und Kanten $E \subseteq V \times V$ gegeben. Jede Kante $e \in E$ hat ein Gewicht $c(e) \in \mathbb{R}_+$. Die Knotenanzahl $|V|$ wird mit n bezeichnet und Kantenanzahl $|E|$ mit m. Gesucht ist ein zusammenhängender Subgraph $G' = (V, E')$ mit $E' \subseteq E$, der alle Knoten beinhaltet und das minimale Gewicht $\sum_{e \in E'} c(e)$ hat.

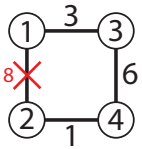
Die meisten Algorithmen arbeiten mit folgenden beiden Eigenschaften um festzustellen, dass eine Kante auf jeden Fall in der Lösung sein kann, bzw. auf keinem Fall in der Lösung sein muss.

5.2.0.1 Cut Property



Für einen beliebigen Schnitt $S \subset V$ werden die geschnittenen Kanten $\{\{u, v\} \in E : u \in S, v \in V \setminus S\}$ betrachtet. Eine der leichtesten dieser Kanten muss in jedem MST vorkommen.

5.2.0.2 Cycle Property



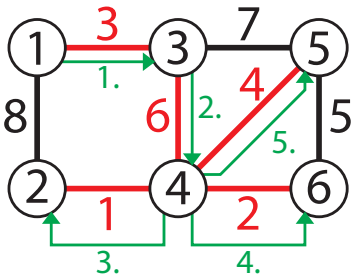
Bei einem beliebigen Kreis $v_0, v_1, ..., v_{x-1}, v_x = v_0$ kann man eine der schwersten Kanten $\{v_j, v_{j+1}\}$ aus dem Ursprungsgrad löschen, ohne dass das Gewicht der Lösung erhöht wird.

5.2.1 Der Jarník-Prim Algorithmus

Der Algorithmus wurde 1930 vom tschechischen Mathematiker Vojtěch Jarník beschrieben. Er bekam in der Informatik jedoch erst Aufmerksamkeit als er vom amerikanischen Informatiker Robert Prim 1957 neu veröffentlicht wurde. Deshalb wird er hier Jarník-Prim Algorithmus genannt.

Algorithmus 5.4: Jarník-Prim Algorithmus

```
T = ∅
S = {s} für einen beliebigen Knoten s ∈ V
while(S ≠ V){
   {u,v} = min{{u,v} : u ∈ S, v ∈ V \ S}
   S = S ∪ {v}
   T = T ∪ {u,v}
}
return T
```



Erläuterungen: Die zu S adjazenten Knoten werden in einer adressierbaren Prioritätsliste gehalten. Ihre Priorität ist die leichteste Kante von S zu dem Knoten. Dadurch kann man die leichteste Kante durch eine Prioritätslistenoperation finden. Beim Erweitern von S muss jedoch jede Kante vom neuen Knoten betrachtet werden. Falls die Kante leichter ist als die Kanten, die bisher zu dem Zielknoten führen, dann muss die Prioritätsliste aktualisiert werden. Jede Kante die zu einem Knoten führt, der noch nicht in der Prioritätsliste war, muss hinzugefügt werden.

Dieser Algorithmus arbeitet auf offensichtliche Weise mit Hilfe der Cut Property. Die Schleife wird für jeden Knoten einmal durchlaufen. Abgesehen von den Prioritätslistenoperationen wird jede Kante dabei genau 2 mal betrachtet (nämlich wenn die Schleife für die beiden Endknoten durchlaufen wird). Damit hat man einen Aufwand von $O(m+n)$ außerhalb der Prioritätsliste. Hinzu kommt bei jedem Schleifendurchlauf ein deleteMin ($O(\log n)$) und maximal für jede Kante ein decreaseKey (amortisiert $O(1)$). Insgesammt hat der Algorithmus von Jarník-Prim also eine Laufzeit von $O(m + n \log n)$ wenn man Fibonacci Heaps verwendet. Große Teile dieses Algorithmus können nicht parallelisiert werden. Man kann aber z.B. $log(n)$ Prozessoren verwenden um die Prioritätslistenzugriffe zu parallelisieren.

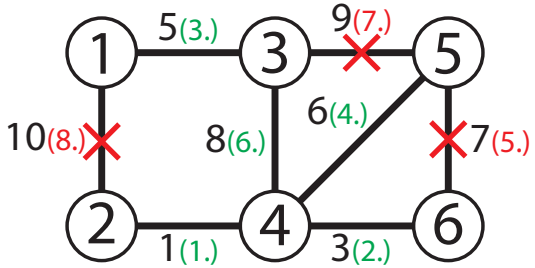
5.2.2 Kruskal's Algorithmus

Algorithmus 5.5: Kruskal's Algorithmus

```
T = ∅
sort(E)
for({u,v} ∈ E in aufsteigender Reihenfolge){
  if(u und v sind in verschiedenen Teilbäumen){
    T = T ∪ {u,v}
  }
}
return T
```

Erläuterungen: Die Teilbäume werden mit Hilfe einer UNION-FIND-Datenstruktur gespeichert. Dadurch hat die Überprüfung ob u und v im selben Teilbaum sind einen amortisierten Aufwand von $O(\alpha(m,n))$ wobei α die Umkehrung der Ackermannfunktion ist. Details können im Algorithm Engineering Skript nachgeschlagen werden. Kruskal's Algorithmus hat einen Aufwand von $O(sort(m)+n\alpha(n))$. Auch dieser Algorithmus hat Teile die sequenziell ausgeführt werden müssen. Das Sortieren kann jedoch parallelisiert werden.

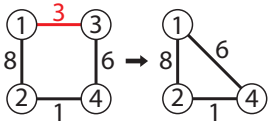
Schritt	1	2	3	4	5	6	7	8
Gewicht	1	3	5	6	7	8	9	10
Cycle P.	j	j	j	j	n	j	n	n



Kruskals Algorithmus

5.2.3 Borůvka's Algorithmus

Der Algorithmus von Borůvka basiert auf Kantenkontraktion. Für eine Kantenkontraktion in dem Graphen (V,E) wählt man eine beliebige Kante $\{u,v\} \in E$. Über diese Kante wird kontrahiert, indem v entfernt wird und jede Kante, die

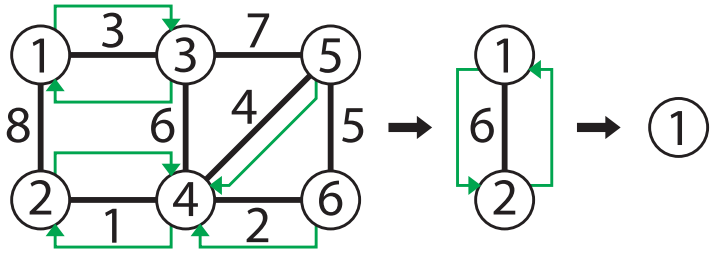


zu v geführt hat, anschließend zu u führt. Die Kantengewichte werden dabei beibehalten und falls man nicht nur das Gewicht des MST, sondern auch die Kanten des MST ausgeben will, dann muss gespeichert werden, zwischen welchen Knoten die Kante ursprünglich verlief.

Algorithmus 5.6: Borůvka's Algorithmus

```
T := ∅
while(|V| > 0) do
  S := ∅
  for v ∈ V do
    S := S ∪ min{{u,v} ∈ E}
  for {u,v} ∈ S
    kontrahiere über {u,v}
  T = T ∪ S
return T
```

Erläuterungen: Durch das Kontrahieren kann es passieren, dass 2 Knoten mehrere Kanten zwischen sich haben. Die intuitive Lösung zum Entfernen einer dieser Kanten kann nicht in $O(m)$ Zeit durchgeführt werden. Wenn man alle Kontraktionen, die gemeinsame Knoten haben, gleichzeitig ausführt ist das jedoch möglich. Jede Iteration braucht $O(m)$ Zeit und halbiert die Knotenanzahl. Da abgebrochen wird, sobald man nur noch einen Knoten hat, gibt es maximal $\log(n)$ Iterationen. Die gesammte Laufzeit liegt daher in $O(m \cdot log(n))$



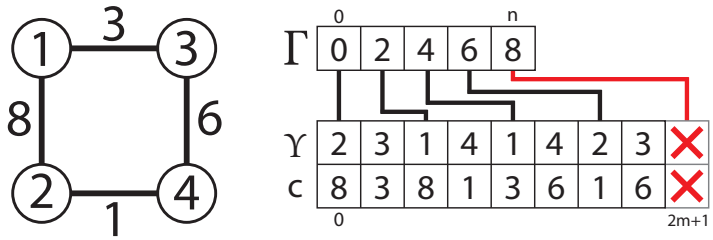
5.2.4 Paralleler Borůvka

Das Ziel beim Parallelisieren dieses Algorithmus ist es das Problem bei gleicher Arbeit in polylogarithmischer Zeit zu lösen. D.h. für konstantes bzw. geeignet wachsende p gilt: $T(m,n,p)\cdot p \in O(m\cdot\log(n))$ und $\exists c : T(m,n,p) \in O(\log^c(m))$. Der Algorithmus wird folgendermaßen aussehen:

Algorithmus 5.7: Paralleler Borůvka Algorithmus

```
1 while(|V| > 1){
2   finde die leichtesten angrenzenden Kanten //O(m/p + log(n) + log(p))
3   ordne jedem Knoten seinen Subgraphen zu //O(n/p + log(n))
4   kontrahiere die Subgraphen //O(m/p + log(n))
5 }
```

Der MST ist die Vereinigung aller Kanten, die in Zeile 2 gefunden wurden. Diese Parallelisierung benutzt die Adjazenz-Array-Darstellung des Graphen. Bei der Adjazenz-Array-Darstellung gibt es ein Array Γ für die Knoten der Länge n+1 und ein Array Υ der Kanten der Länge 2m. Das Kantenarray speichert den Zielort der Kante zusammen mit den Kosten. Die Kanten von V[i] bis V[i+1] sind adjazent zu Knoten i.

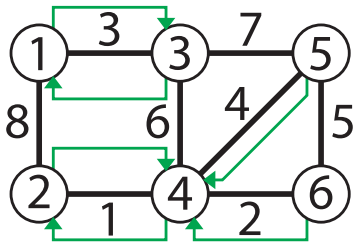


Beispiel für Adjazenz-Array-Darstellung

In dieser Darstellung können die Knoten und Kanten als Integer repräsentiert werden. Dann gilt $V = \{0..n-1\}$ und $E = \{0..2m-1\}$. Eine Kante e geht von v nach w genau dann wenn $\Gamma[v] \leq e < \Gamma[v+1]$ und $\Upsilon[e] = w$ Jetzt ein genauerer Blick auf jeden Schritt

5.2.4.1 leichteste angrenzende Kante finden

Die Kanten werden gleichmäßig aufgeteilt, so dass jeder die Kanten von $i \cdot \frac{2m}{p}$ bis $(i+1) \cdot \frac{2m}{p} - 1$ des Kantenarrays erhält. Jeder Prozessor muss noch wissen zu welchen Knoten die von ihm zu bearbeitenden Kanten gehören. Das kann in $O(\log(n))$ Zeit durch p Binärsuchen oder in $O(\frac{n}{p} + p)$ durch lineares Suchen gemacht werden. Der addi-



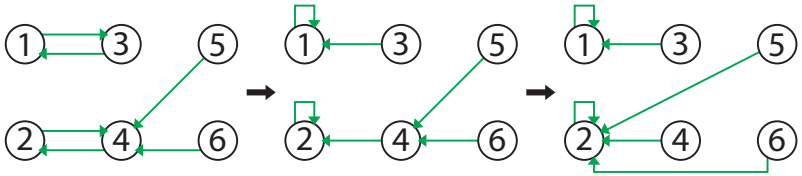
tive Term $+p$ kommt dabei durch das Senden der Ergebnisse an die entsprechenden Knoten. Die Binärsuche liegt zwar in einer besseren O Klasse, kann aber langsamer sein, da jeder Prozessor auf den gesammten Datenbereich zugreift. Dadurch kann es zu Verzögerungen durch den Cache auf einem Shared-Memory-Modell, bzw. vielen Kommunikationen auf einem Distributed-Memory-Modell kommen. Zu den so bestimmten Knoten berechnet jeder Prozessor in $O(\frac{m}{p})$ dann die leichteste adjazente Kante aus seiner Knotenmenge.

```
v = find(i · 2m/p, Γ);
for(e = i · 2m/p; e<(i+1) · 2m/p − 1; e++){
  if(Γ[v+1]==e) v++;
  if(c[e] < c[pred[v]]) pred[v]=e;
}
```

Dieser Code sieht zwar einfach aus, versteckt jedoch ein Problem. Einige Knoten werden von mehreren Prozessoren bearbeitet. Deshalb braucht man entweder recht viel Kommunikation während der Berechnungen, oder jeder Rechner hat ein eigenes prev-Array, das hinterher durch Reduktion mit denen der anderen Prozessoren verschmolzen wird. Hier wird der zweite Fall betrachtet, da er einfacher zu programmieren und analysieren ist. Da jeder Prozessor maximal an 2 solchen Reduktionen teilnimmt und eine Reduktion $\log(p)$ Zeit braucht, braucht dieser Schritt $O(\log(p))$ Zeit. Insgesammt braucht dieser Schritt $O(\frac{m}{p} + \log(n) + \log(p))$ Zeit.

5.2.4.2 Zuordnung der Knoten zu Subraphen

Im Folgenden wird der Graph betrachtet, der ausschließlich die Kanten enthält, die vorher die leichtesten Kanten an einem Knoten im Ursprungsgraph waren. Also die Kanten die im vorherigen Schritt bestimmt wurden. Diese Kanten sind gerichtet und zeigen von dem Ort, an dem sie die leichteste Kante sind, weg. Der daraus entstehende Graph zerfällt in mehrere schwache Zusammenhangskomponenten. Ziel dieses Schrittes ist es jedem Knoten seine Zusammenhangskomponte zuzuordnen. Dabei werden die Zusammenhangskomponten durch einen beliebigen Knoten in ihnen repräsentiert. Da von jedem Knoten eine Kante ausgeht, ist jede Zusammenhangskomponente ein Pseudotree, d. h. ein Baum mit einer zusätzlichen Kante. Diese extra Kante ist entlang der leichtesten Kante des Subgraphen und wird mit Hilfe folgenden Codes zu einer Schlaufe.




```
forAll(v\in V) do parallel
{
  w = pred[v]
  if((pred[w] == v) && (v<w)) pred[v]=v
}
```

Jetzt besteht eine Zusammenhangskomponente aus einem Baum dessen Wurzel auf sich selbst zeigt. Diese Wurzel wählen wir als Representanten der Zusammenhangskomponente. Durch Doubling kann jedem Knoten sein Representant zugeordnet werden.

```
while ∃v ∈ V: pred[v] ≠ pred[pred[v]]
  forall v ∈ V: pred[v] = pred[pred[v]]
```

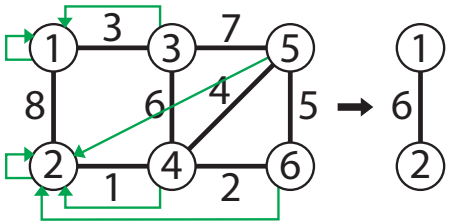
Dieser Teil braucht $O(\frac{n}{p}\log(n))$ Zeit. Wenn man jedoch ähnliche Ideen wie bei List Ranking anwendet kann man das auch in $O(\frac{n}{p}+\log(n))$ durchführen. Jeder Graph hat jetzt Sternenform.

5.2.4.3 Kontrahieren der Subgraphen

In diesem Schritt wird jeder Subgraph zu einem Knoten zusammenkontrahiert.

```
k = Anzahl an Subgraphen.
V'= \{0.. k-1\}
finde eine bijektive Abbildung f:Sternwurzel → 0..k-1
E' = {(f(pred[v]), f(pred[w]), c, e_old) : (v,w) ∈ E ∧ pred[v] ≠ pred[w]}
```

Die bijektive Abbildung kann mit Hilfe einer Prefixsumme in $O(\frac{n}{p}+\log(p))$ bestimmt werden. Neben einer neuen Knoten und Kantenmenge muss auch Γ und Υ der Adjazenzarraydarstellung neu bestimmt werden. Das kann gemacht werden indem man Integersort auf E' anwendet. Das braucht $O(\frac{m}{p}+\log(p))$ Zeit. [8, Rajasekaran and Reif 1989].



5.2.4.4 Analyse

Eine Iteration braucht $O(\frac{m}{p}+\log(n))$ Zeit und wie beim sequentiellen Algorithmus gibt es $\log(n)$ Iterationen. Die Laufzeit ist daher in $O(\log(n)(\frac{m}{p}+\log(n)))$ Zeit. Für $m \in \Omega(p\log^2(p))$ gilt daher $E_{rel}(m,n,p) = \frac{\log(n)(m+\log(n))}{p\cdot\log(n)(\frac{m}{p}+\log(n))} = \frac{m+\log(n)}{m+p\log(n)} \in \Theta(1)$. Es ist also relativ Effizient. Und falls sogar $m \in O(n)$, dann: $E(m,n,p) = \frac{n\log(n)+m}{p\cdot\log(n)(\frac{m}{p}+\log(n))} = \frac{n\log(n)+m}{m\log(n)+p\log^2(n)} \in \Theta(1)$ und es ist dann auch absolut Effizient.

6 Lastverteilung

Mit der Parallelisierung entsteht ein neues, typisches Problem. Die Laufzeit eines Algorithmus richtet sich stets nach dem letzten Prozessor, der die Ausführung seiner Aufgaben beendet. Kann man Aufgaben also in parallelisierbare Teile zerlegen, so möchte man erreichen, dass kein Prozessor besonders viel Arbeit bekommt und damit die Laufzeit in die Höhe treibt. Dieses Problem wird hier auf eine Menge von n Teilaufgaben ("Jobs") verallgemeinert, die unter den Prozessoren aufzuteilen sind. Der Aufteilung werden Kosten zugeordnet die sich aus Kosten für die Aufteilung und dem Aufwand der jeweiligen Jobs bestimmen lassen. Ziel ist es diese Kosten zu minimieren. Die Verwendung eines allgemeinen Kostenbegriffs ist dadurch motiviert, dass unterschiedliche Faktoren angefangen bei Laufzeit, bis hin zum Stromverbrauch zu optimieren sein können. Im Folgenden steht aber die Laufzeit im Vordergrund.

Das Problem der Lastenverteilung ist in mehr oder weniger gut erkennbarer Form bereits bei einigen der bisher betrachteten Algorithmen aufgetreten. So stellte sich bei Sample Sort die Frage, ob zwischen zwei Pivotelementen ungefähr gleich viele Elemente liegen würden. Um einzusehen, dass das ein Problem der Lastenverteilung ist, soll wieder aufgefrischt werden, worum es bei Sample Sort geht. Sample Sort geht das Problem eine Menge zu sortieren an, indem es die Eingabe durch Pivotelemente in Teilfolgen aufspaltet. Eine Teilfolge zu sortieren kann dann als "Job" betrachtet werden. Es ist also auch ein Lastverteilungsproblem, bei dem eine große Aufgabe in p Jobs aufzuteilen ist, deren Größe so wenig variieren soll, wie möglich. Dieser Fall, in dem man bei der Generierung der Jobs wenig Wissen darüber hat, was man bekommt, ist aber vielleicht nicht die erste Vorstellung, die man von einem Problem der Lastverteilung hat. Vielleicht stellt man sich unter Lastverteilung zuerst eher etwas vor wie das, was die parallelen Prioritätslisten als Lastverteilungsalgorithmus geleistet haben. Sie haben im Brach-and-Bound-Beispiel eine Menge von Knoten verwaltet, die zu bearbeiten waren. Jeder Knoten kann auch als Job betrachtet werden, da zu jedem (inneren) Knoten Arbeit für Expansion

gehört. Bei Branch-and-Bound wurde also ein Lastverteilungsproblem betrachtet, bei dem aus einem Pool von Jobs jeder Prozessor in einem Schritt immer einen bekommt und im Laufe der Abarbeitung immer wieder neue Jobs entstehen.

Man sieht also, dass zunächst genauere Eigenschaften festgelegt werden müssen. Das beginnt bereits bei der Frage, welche Eigenschaften die Jobs haben. Wie genau kennt man den Aufwand für einen Job? Kann man ihn weiter in kleinere Jobs aufteilen, oder muss er komplett sequentiell auf einem Prozessor abgearbeitet werden? Noch schwieriger wird es bei der Kostenbestimmung. Kosten setzen sich aus mehreren Komponenten zusammen. Es sollte zunächst nicht vernachlässtigt werden, wie schwierig die Bestimmung einer Zuordnung ist. Auch Kommunikation, die nötig ist Jobs zu verteilen, schlägt sich auf die Gesamtkosten nieder, genauso wie mögliche Kosten zur Umverteilung von Jobs im Laufe der Abarbeitung. Letztlich kommt natürlich auch die maximale Last auf einem Prozessor hinzu, die sich aus den Kosten einzelner Jobs bestimmen lässt. Hier kann es sein, dass Kosten nicht nur von einem Job, sondern auch von dem Prozessor abhängen. Wenn als Kosten die Laufzeit betrachtet wird und die Prozessoren unterschiedlich schnell sind, so ist die Abhängigkeit vom Prozessor offensichtlich. Das ist beispielsweise in inhomogenen Clustern der Fall. Motiviert durch diese Situation kann man sogar den Fall betrachten, dass die Kosten nur geschätzt werden können, aber schwanken können sobald das analysierte Programm nur eines von mehreren ist, die auf den einzelnen Prozessoren ausgeführt werden. Es bestehen vielleicht sogar Abhängigkeiten zwischen Jobs so dass die Kosten von einem Job vom Fortschritt bei einem anderen abhängen.

Die Betrachtungen im Folgenden beschränken sich auf den Fall dass Kosten für einen Job stets **unabhängig vom ausführenden Prozessor** sind. Bei den anderen Eigenschaften des Lastverteilungsproblems werden verschiedene interessante Kombinationen betrachtet für die typische Lösungsansätze exisiteren. Ob sich Jobs aufspalten lassen und wie genau die Kosten bekannt sind wird im Einzelfall definiert. Häufig kann man es sich aber so vorstellen, dass die Jobs aus einer bekannten Anzahl atomarer Aufgaben bestehen, deren Kosten nicht genau bekannt sind. Abhängigkeiten zwischen Jobs werden in den Beispielen hier erst zum Schluss berücksichtigt.

6.1 Statische Lastverteilung

Statische Ansätze für das Problem der Lastenverteilug legen sich bereits zu Beginn darauf fest, welcher Prozessor welche Jobs erhält. Umverteilung der

Jobs ist dadurch nicht nötig, also beschränkt sich der Verwaltungsaufwand auf die Bestimmung einer Zuodnung und einmaliges Verschieben für jeden Job. Auf der anderen Seite lassen sich bei diesem Ansatz auch Fehler bei der anfänglichen Verteilung nicht mehr korrigieren. Besonders wenn die Jobgrößen nicht im Voraus bekannt sind, kann so nicht garantiert werden, dass alle Prozessoren optimal ausgelastet sind.

6.1.1 Ein ganz einfacher Fall

Den Anfang macht ein Lastverteilungsproblem, das einfach zu lösen und zu parallelisieren ist, damit man einen Einstieg erhält. Es wird der Fall betrachtet dass sich die Jobs beliebig unterteilen lassen und **die Größe l_j für jeden Job** stets genau bekannt ist. In diesem Fall reicht ein eifacher Greedy-Algorithmus, um das Problem zu lösen.

Algorithmus 6.1: Next Fit

```
//capacity per PE:
C := sum_{j=0}^{n} l_j / p;
//PE jobs are assigned to currently:
t := 0;
//remaining capacity on PE t:
f := C;
//job to assign next:
j := 0;
//size for not assigned part of job j:
l := l_0;
while(j ≤ n){
  c := min(f,l);
   assign a piece of size c of job j to PE t ;
  f := f-c;
  l := l-c;
  if(f = 0){
    t++;
    f := C;
  }
  if(l = 0){
    j++;
    l := l_j;
  }
}
```


Erläuterungen: Zunächst wird in C gespeichert, wie viel Arbeit bei gleichmäßiger Aufteilung für einen Prozessor anfallen würde. Nun arbeitet der Algorithmus genau so, wie man es von einem Greedy-Algorithmus erwarten würde. Er nimmt sich den ersten nicht zugewiesenen Job und weist ihn dem ersten Prozessor zu, der noch nicht C Arbeit leisten muss. Hat er nach der Zuordnung des nächsten Jobs mehr als C, so wird der neue Job geeignet aufgeteilt, und den überschüssigen Teil erhält der nächste Prozessor.

Next Fit liefert bei aufspaltbaren Problemen und bekannten Jobkosten in linearer Zeit ein optimales Ergebnis. Es muss jeder Job einmal betrachtet werden und nach p Aufspaltungen sind sicher keine mehr nötig, also liegt die Zuordnungsphase in $O(n+p)$. Anschließend ist aber jeder Prozessor nur mit einem Anteil C an den Gesamtkosten beteiligt, was optimal ist.

Analyse:

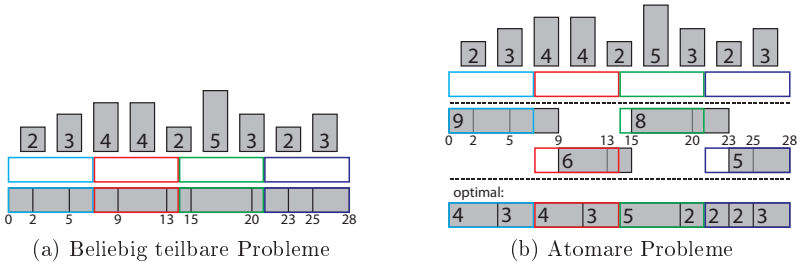
$$T(n,p) \in O(n+p) + \frac{\sum_{j=0}^{n-1} l_j}{p}$$

Die Lineare Zeit für die Bestimmung der Zuordnung möchte man vermeiden und parallelisiert auch die Zuordnung der Jobs zu den Prozessoren. Damit erhält man folgenden Algorithmus:

Algorithmus 6.2: Paralleles Next Fit

```
C := sum_{j=0}^{n} l_j / p;
parallel_for(j = 0; j < n; j++){
  //use prefix sum:
  pos := sum_{k=0}^{j} l_k;
  //use segmented broadcast:
   assign job j to PEs ⌊pos/C⌋ .. ⌊(pos+l_j)/C⌋ ;
   piece size at PE (k := ⌊pos/C⌋): (k+1)C - pos ;
   piece size at PE (k := ⌊(pos+l_j)/C⌋): pos + l_j + kC ;
}
```

Erläuterungen: Bei dieser Parallelisierung zeigt sich, dass bei bekannter Jobgröße Präfixsummen gut geeignet sind die Jobs gut zu verteilen. Mit ihnen kann für jeden Job bestimmt werden, wie viele Kosten alle Jobs mit



(a) Beliebig teilbare Probleme (b) Atomare Probleme

Abbildung 6.1: Next Fit

kleinrem Index zusammen benötigen. Daraus lässt sich dann leicht ablesen, wie viele Prozessoren bereits gesättigt sind, ob das Aufsummieren der Kosten für diesen Job auch den ausführenden Prozessor auslasten würde so dass der Job aufgeteilt werden muss und auch wie die Aufteilung dann aussehen müsste. Jeder Prozessor berechnet die Zuordnung für n/p Jobs. Dazu erhält er aus einer lokalen Summe über die Kosten seiner n/p Jobs und einer Präfixsumme genug Information. Damit verbessert sich die Zeit für die Berechung einer Zuordnung auf $O(\log p)$ Zeit für die Präfixsumme und das Verlegen der Jobs, sowie $O(n/p)$ für Berechnungen auf eigenen Elementen. Diese Abschätzung ist leider etwas optimistisch, da das Versenden der Jobs mehr Aufwand bedeuten kann, als ihm zugestanden wird. Im Schlimmsten Fall sind fast alle Jobs klein und in der Eingabe direkt hintereinander. Statt jedem Prozessor einige kleine und einen Teil von einem großen Job zu geben, müssen dann alle kleinen Jobs auf einen Prozessor. Zwar ist dann die Lastverteilung immer noch gut, aber der Kommunikationsaufwand ist dann höher, als erforderlich.

Analyse:

$$T(n,p) \in O(\frac{n}{p} + \log p) + \frac{\sum_{j=0}^{n-1} l_j}{p}$$

Der Algorithmus lässt sich leicht anpassen, um eine Lösung für ein Lastverteilungsproblem mit atomaren, also nicht weiter aufspaltbaren, Jobs zu bekommen. Dazu verzichtet man nur auf das Aufspalten der Jobs, die auf der Grenze liegen. Man sieht leicht ein, dass die Lösung die man dann bekommt einem Prozessor niemals Jobs mit einem Gesamtaufwand von $C + \max_{0 \le j < n} l_j$ oder mehr zuordnet, da dafür ein mit Kosten von mindestens C ausgelasteter

Prozessor einen weiteren Job erhalten müsste. Da C eine untere Schranke für die optimale Lösung ist und $\max_{0 \le j < n} l_j \le C$ gilt, ist das eine 2-Approximation.

Eine Bessere Approximation erhält man, indem man die Jobs zunächst nach Größe sortiert und stets den größten Job dem Prozessor mit der kleinsten Last zuordnet. So erhält man eine 4/3-Approximation, allerdings ist der Algorithmus sequentiell. Aus [1] stammt ein Algorithmus, der eine 11/9-Approximation liefert, aber zu dem keine Parallelisierung bekannt ist. Besonders bei kleinen $\max_{0 \le j < n} l_j$ ist der zusätzliche Aufwand bei der Berechnung nicht gerechtfertigt, da so nicht genug Zeit bei der tatsächlichen Ausführung eingespart wird.

6.1.2 Statisch und unwissend – Randomisierung

Nun wird auch die zweite hilfreiche Voaussetzung geändert, die Jobgrößen sollen nun auch noch unbekannt sein. Es sollen nun also atomare Jobs unbekannter Größe gleich zu Beginn fest zugeordnet werden. Es überrascht nicht, dass es unmöglich ist das Problem unter diesen Bedingungen so zu lösen, dass die Lastverteilung immer optimal ist. Bevor man sich aber anderen Ansätzen widmet, möchte man doch noch sehen, wie gut blind geratene Lösungen sein können.

Als motivierendes Beispiel für das betrachtete Problem wird hier ein bekanntes Fraktal benutzt. Die Mandelbrotmenge – die ihrer Form auch den Namen "Apfelmännchen" zu verdanken hat – besteht aus allen $c \in \mathbb{C}$ für die die durch $z_0 := 0$, $z_{k+1} := z_k^2 + c$ definierte Folge beschränkt ist. Man kann sich sicher sein, dass ein Punkt nicht zur Mandelbrotmenge gehört, wenn $|z_k| > 2$ für ein k gilt. Leider kann das erste k für das es gilt je nach c beliebig groß sein, so dass praktisch ein kleiner Fehler hingenommen und ein $k_{\max}$ gewählt wird, ab dem man einfach annimmt dass der Punkt zur Menge gehören würde. Natürlich kann das so immer noch nicht für alle Punkte aus $\mathbb{C}$ gemacht werden. Für das Lastverteilungsproblem soll nur für n Punkte bestimmt werden, ob sie zur Mandelbrotmenge gehören oder nicht, was beispielsweise für die Anzeige auf dem Bildschirm vollkommen ausreicht. Das ist schon deswegen ein interessantes Beispielproblem, weil der Aufwand für einen Job, also die Zeit für die Überprüfung eines einzelnen Punktes auch gleichzeitig das Ergebnis der Berechnungen für diesen Punkt ist. Man kennt also den Aufwand nicht, solange der Job noch nicht ausgeführt wurde.

Nun aber zur eigentlichen Verteilung der Jobs. Es werden hier drei Ideen betrachtet, wie die Punkte über die Prozessoren verteilt werden könnten. Ei-

ne Möglichkeit wäre die Daten immer nach n/p Elementen aufzuteilen und immer einen solchen Abschnitt einem Prozessor zu überlassen. Sieht man sich hier die Mandelbrotmenge an, so erhält jeder Prozessor einen Streifen aus dem Gesamtbild. Hier zeigt sich auch die Schwäche dieser Verteilung. In den äußeren Bereichen wird eine Iterationstiefe von $k_{\max}$ kaum erreicht, während in den mittleren Streifen sehr viele Punkte zur Mandelbrotmenge gehören. Offensichtlich würde also eine solche Streifenverteilung dazu führen, dass Prozessoren, die Streifen aus der Mitte bekommen, erheblich mehr arbeiten müssen.

Eine Zweite Art die Jobs zu verteilen wäre nach dem Round Robin Verfahren. Der k-te Job wird also dem ($k \mod p$)-ten Prozessor zugewiesen. Bei der Bestimmung der Mandelbrotmenge bekommt man so eine erheblich bessere Lastverteilung, als mit streifenweiser Zerlegung. Natürlich kann es immer noch Fälle geben, in denen jeder p-te Job besonders schwierig ist, allgemein lässt sich also nicht beweisen, dass die Verteilung gut ist. Aber das ist nicht das einzige Problem dieser Verteilung. Wenn zu den Jobs Elemente in einem Array gehören, so kann die Bearbeitung benachtbarter Jobs Schreibzugriff auf benachbarte Einträge im Array erfordern. Dadurch kann es zu Konflikten kommen, bei denen im schlimmsten Fall alle Prozessoren auf eine Cache-Zeile zugreifen und sich gegenseitig die Caches invalidieren. Die Folge ist hoher Kommunikationsaufwand, obwohl die Schreibzugriffe alle unabhängig sind, ein Paradebeispiel für False Sharing also. Das ist aber kein Grund auf dieses Verfahren zu verzichten, denn geeignete Umverteilung der Daten reicht bereits, um False Sharing zu vermeiden. Dennoch wird bei OpenMP streifenweise Zuordnung verwendet, da sie in vielen Fällen eine gute Lastverteilung liefert ohne Umverteilung zu benötigen.

Das letzte Verfahren benutzt eine gängige Mehtode mit dem "Widersacher" umzugehen, der immer den Fall wählt, mit dem der Algorithmus nicht zurechtkommt. Randomisierung sorgt für eine beweisbar gute Lastverteilung. Wieder einmal beruft man sich auf Chernoff-Schranken und kann dann zeigen, dass unter der Bedingung dass

$$L := \sum_{j=0}^{n-1} l_j \geq 2(\beta+1)pl_{\max}\frac{\ln p}{\epsilon^2} + O(\epsilon^3)$$

gilt mit einer Wahrscheinlichkeit von $(1 - p^{-\beta})$ die Last auf keinem Prozessor $(1+\epsilon)\frac{L}{p}$ übersteigt. Zufällige Verteilung funktioniert am besten, wenn $\frac{L}{l_{\max}}$ sehr groß ist. Sie kann dagegen schlecht damit umgehen, wenn es einen besonders teuren Job gibt, der eigentlich einen eigenen Prozessor bräuchte. Statische Lastverteilungsalgorithmen können alle so einen großen Job nicht

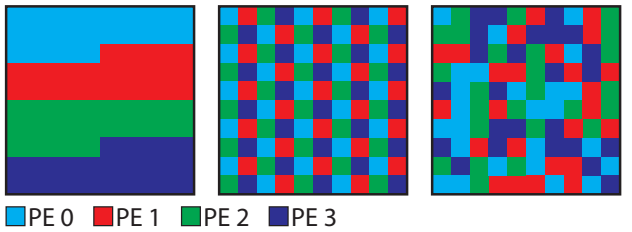


Abbildung 6.2: Aufteilung der Elemente Zeilenweise, nach dem Round Robin Verfahren, sowie zufällige Verteilung

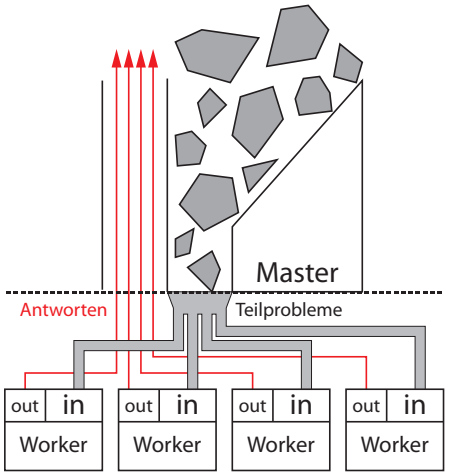


Abbildung 6.3: Master-Worker-Schema

von anderen unterscheiden.

6.2 Master-Worker

Nachdem man nun eine Vorstellung davon bekommen hat, wie weit man mit statischer Lastverteilung kommt, ist hier ein dynamischer Ansatz. Beim Master-Worker-Schema verteilt ein Prozessor die Arbeit unter allen anderen und ist damit auch für die Lastverteilung verantwortlich. Er weiß zu jedem Zeitpunkt welche Prozessoren noch beschäftigt sind und muss entscheiden, welcher Prozessor welchen Teil der verbliebenen Arbeit bekommt. Das Schema bietet sich besonders in Situationen an, in denen sowieso eine hierarchi-

sche Beziehung vorgegeben ist. Ein Knoten in einem Cluster hat vielleicht als einziger Zugriff auf die benötigten Daten und muss sie sowieso unter den Prozessoren verteilen.

Die Zeit für einen speziellen Job muss wieder nicht bekannt sein. Wir nehmen aber an, dass man die Jobgrößen abschätzen kann. Die Mandelbrotmenge kann wieder als Beispiel dienen. Sie ist auch deswegen wieder ein gutes Beispiel, weil womöglich nur ein Prozessor für die Anzeige auf dem Bildschirm verantwortlich ist und sich so wieder als Master anbietet. Ein Job, den der Master vergibt, besteht aus einem Block von Punkten. Ein rechteckiger Bildausschnitt beispielsweise lässt sich kompakt beschreiben. Die Anzahl der Punkte in diesem Ausschnitt liefert eine Schätzung über die Größe des Teilproblems. Der genaue Aufwand ist aber weiterhin nicht bekannt.

Wichtig ist, dass das Master-Worker-Schema nichtpräemptiv ist. Das bedeutet, dass ein weggegebener Job ausgeführt wird und nicht im Nachhinein aufgeteilt werden kann. Eine gute Strategie ist daher zunächst große Jobs zu verteilen und darauf zu warten, dass die ersten Prozessoren fertig werden. Diese erhalten nun kleinere Teile der restlichen Arbeit. Hier muss dann eine Balance zwischen guter Lastverteilung und niedrigem Kommunikationsaufwand gesucht werden. Kleine Jobs gleichen Unterschiede in der Lastverteilung besser aus, einen großen Job statt mehrerer kleiner zu versenden spart Kommunikation. Man sollte sich daher gut überlegen, ob man nicht ungleichmäßige Lastverteilung riskieren möchte, um den Kommunikationsaufwand zu reduzieren.

Der Kommunikationsaufwand ist die wohl wichtigste Schwachstelle. Da der Master an jeder Kommunikation beteiligt ist, stellt seine Bandbreite einen Flaschenhals dar. Das Master-Worker-Schema skaliert also schlecht. Es kommt hinzu, dass auch bei einem Master, der zunächst als einziger auf die Daten zugreifen kann ein Broadcast effektiver sein kann. Häufig benötigt sowieso jeder Prozessor Zugriff auf einen großen Teil der Daten. Für das Problem der schlechten Skalierbarkeit gibt es Ansätze einer komplexeren Hierarchie, mit "Vorarbeitern" unter dem Master, die ihrerseits Master einiger Worker sind. Ob sich jedoch der Aufwand dafür lohnt ist zweifelhaft. Für das Master-Worker-Schema spricht dessen einfacher Aufbau. Der Master hat eine klare Aufgabe, er muss eine gute Aufteilung finden und darauf achten, dass zu allen Jobs anschließend Ergebnisse vorliegen. Da er alle Informationen über die verteilten, fertigen und noch nicht vergebenen Jobs hat, ist das vergleichsweise einfach. Noch einfacher ist die Aufgabe der Worker, die nur die ihnen übertragenen Aufgaben erledigen und das Ergebnis zurücksenden müssen.

6.3 Work Stealing

Das letzte vorgestellte Verfahren könnte man im Vergleich mit dem Master-Worker-Schema als noch dynamischer bezeichnen. Die weggebene Teilproblemgröße wird nicht wie bisher einmalig geschätzt, sondern es findet wiederholte Aufteilung im Laufe der Abarbeitung statt. So wird nicht nur der Engpass beim Master aus dem Vorherigen Ansatz vermieden, sondern es werden in vielen Fällen auch weniger Teilprobleme erzeugt.

6.3.1 Tree Shaped Computations

Der Name Tree Shaped Computations bezeichnet eine ganze Familie von Problemen, die sich mit dem in diesem Abschnitt betrachteten Ansatz gut behandeln lassen. Wie der Name sagt, geht es um Berechnungen, die sich durch Bäume repräsentieren lassen. Allen gemeinsam ist, dass man mit einem Problem P_{root} beginnt und dieses Wurzelproblem bei der Bearbeitung in mindestens zwei unabhängige Teilprobleme zerfällt, deren Lösungen die Lösung des Gesamtproblems bilden. Ebenso sollen auch die Teilprobleme sich weiter in noch kleinere Teilprobleme aufspalten lassen, die Anzahl der durchgeführten Aufspaltungen, bis ein spezielles Problem entstanden ist, wird als dessen **"Generation"** bezeichnet. Die diese Aufspaltung beschreibende Baumstruktur kann sehr unregelmäßig sein und ihre Blätter stellen Arbeit dar, die zwangsläufig sequentiell ist. Kurzgefasst könnte man Tree Shaped Computations beschreiben als Probleme, die bei der Abarbeitung wiederholt in unabhängige Teilprobleme zerfallen. Mit Unabhängigkeit ist hier gemeint, dass für die zur Lösung des Problems I benötigte Zeit $T(I)$ und die die Zeiten für die beiden Teilprobleme I_1 und I_2 gilt $T(I) = T(I_1) + T(I_2)$.

Die Mandelbrotmenge, die zuvor schon als Beispiel für ein Lastverteilungsproblem diente, lässt sich auch auf diese Art angehen. Zu Beginn erhält man ein Intervall von Punkten, für die bestimmt werden soll, ob sie zur Mandelbrotmenge gehören. Dieses Intervall lässt sich in zwei Teilintervalle aufspalten, die unabhängige Teilprobleme darstellen und sich genauso kompakt beschreiben lassen, wie das ganze Intervall. Atomare Teilprobleme wären in dann einzelne Punkte.

Es gibt aber viele Fälle in dieser Problemfamilie, für die – anders als bei der Mandelbrotmenge – im Voraus nicht genau bekannt ist, wie oft sich ein Teilproblem noch aufspalten lässt. Häufig kann zumindest eine obere Schranke h für die Generation nichtatomarer Probleme angegeben werden. Für die Analyse von Algorithmen wichtig sind auch die Zeit T_{split} für die

Aufspaltung eines Problems und die Zeit T_{atomic}, die maximal für ein atomares Teilproblem benötigt wird.

Am Beispiel der Mandelbrotmenge hat man bereits gesehen, dass Teilprobleme in noch folgenden Algorithmen die Jobs darstellen sollen. Um die Kosten für die Verlagerung eines Jobs abschätzen zu können, müssen die maximale Größe einer Teilproblembeschreibung l und die sich daraus ergebende Zeit $T_{\text{rout}}(l) := T_{\text{start}} + lT_{\text{byte}}$ für die Kommunikation bekannt sein. Die Zeit T_{coll} soll auch diesmal eine kollektive Kommunikationsoperation darstellen.

Probleme, die sich mit Tiefensuche oder Backtracking lösen lassen, sind gute Beispiele für Probleme aus dieser Familie, aber selbst das Verteilen von Iterationen einer Schleife über mehrere Prozessoren kann so angegangen werden. Nicht zuletzt kommt man sogar mit Branch-and-Bound zurecht, wo eigentlich die Forderung nach Unabhängigkeit zwischen den Teiproblemen verletzt ist. In einigen der folgenden Beispiele sind nicht alle Forderungen an Tree Shaped Computations erfüllt und dennoch lassen sie sich gut mit dem dafür vorgestellten Lastverteilungsalgorithmus angehen. Typisches Beispiel dafür sind, wie erwähnt, Brach-and-Bound-Probleme, bei denen Ergebnisse von anderen Teilbäumen benutzt werden, andere abzuschneiden. Dort ergibt sich die benötigte Zeit für ein Problem offensichtlich nicht direkt aus den Zeiten für die Teilprobleme. Es können aber auch Abschätzungen benutzt werden, die Teilbäume abschneiden ohne dass die Unabhängigkeit der Probleme verletzt wird. Das klappt genau dann, wenn nur Informationen vom Abstiegspfad zum betreffenden Knoten benutzt werden.

6.3.1.1 Tiefensuche

Ein Problem, bei dem sowieso auf einem Baum gearbeitet wird, bietet sich natürlich besonders als Beispiel an. Wenn man einen Baum auf einem Prozessor bearbeitet kann man immer nur einen Knoten zu einer Zeit untersuchen. Alle anderen landen auf einem Stack, wenn sie anfallen. Ein Problem lässt sich also ganz einfach aufspalten, indem ein Teil des Stack abgegeben wird. Wie viele Knoten ein Teilbaum hat, dessen Wurzel ein Element auf dem Stack darstellt, lässt sich nicht im Voraus sagen. Daher kann man sich fragen, ob es besser ist nur einen Knoten abzugeben, der möglichst nahe an der Wurzel ist, oder die Hälfte aller Knoten auf dem Stack.

6.3.1.2 Loop Scheduling

Der zuvor bereits beschriebene Fall der Mandelbrotmenge war ein Spezialfall des Loop Scheduling. Beim Loop Scheduling Problem geht es um Aufteilung von unahbängiger Schleifeniterationen zwischen Prozessoren. Statische Aufteilung liefert bei starken Schwankungen zwischen der Dauer einzelner Iterationen keine zufriedenstellenden Ergebnisse und auch der Master-Worker Ansatz kann sich gute Lastverteilung nur mit hohem Kommunikationsaufwand am Master erkaufen. Das Iterationsintervall erst bei Bedarf aufzuteilen kann sich als sehr nützlich erweisen.

6.3.1.3 0-1 knapsack

Für das 0-1 knapsack Problem sind n Elemente gegeben, wobei für jedes Element j, Gewicht w_j und Gewinn g_j bekannt ist. Weiterhin ist eine Kapazität N gegeben. Gesucht sind $x_j \in \{0,1\}$, so dass $\sum_{j=1}^{n} p_j x_j$ maximal ist und die Bedingung $\sum_{j=1}^{n} w_j x_j \leq N$ erfüllt ist. Man möchte also Elemente in den Rucksack aufnehmen, die den Gewinn maximieren, ohne dass die Kapazität überschritten wird.

Die besten Lösungsansätze für das Problem basieren auf dem Branch-and-Bound-Prinzip, das wie bereits gezeigt ebenfalls durch Bäume beschrieben werden kann. Genauer wird für das 0-1 knapsack Problem immer zunächst in die Tiefe expandiert. Hier tritt bei unregelmäßigen Probleminstanzen ein Fall auf, für den man mit mit dem im Folgenden betrachteten Ansatz häufig superlinearen Speedup erhält, obwohl – oder gerade weil – das Problem die Unabhängigkeitsbedingung verletzt. Jeder Prozessor, der in einen sehr tiefen Teilbaum gerät, ist lange beschäftigt, auch wenn sich dort keine gute Lösung befindet. Im parallelen Fall können häufig bessere Teilbäume zeitgleich zum großen Baum untersucht werden. Nicht nur, dass andere Prozessoren dort womöglich bessere Fortschritte machen, sie könnten auch Berechnungen im großen Teilbaum durch eine gefundene Schranke abbrechen. Es kann sinnvoll sein gefundene Schranken zunächst am Prozessor mit Index 0 zu sammeln. Oft wird sonst das Netzwerk sonst stark ausgelastet, wenn viele neue Schranken parallel gefunden werden und sowieso eine Reduktion nötig ist, um die minimale darunter zu finden.

6.3.1.4 FSSP

Der Lastverteilungsalgorithmus hat sich auch als nützlich bei der Untersuchung eines Problems für Zellularautomaten erwiesen. Die Frage nach Syn-

chronisation eines eindimensionalen Zellularautomaten ist als “Firing Squad Synchronisation Problem” bekannt. Gegeben ist eine Reihe von n nummerierten Elementen, Zellen genannt. Jede Zelle hat einen Zustand. Zu Beginn befindet sich die erste im Zustand g (“General”) und alle anderen im Zustand s (“Soldat”). Um die Beschreibungen zu vereinfachen wird festgelegt, dass vor der ersten und nach der letzten Zelle jeweils eine mit Zustand $\#$ ist. Dieser spezielle Zustand kann nur von diesen Begrenzungszellen angenommen werden und wird daher nicht bei der Anzahl der möglichen Zustände für Zellen berüksichtigt. Alle Zellen wechseln in einem Schritt synchron ihren Zustand gemäß einer Abbildung, die aus ihrem Zustand und dem der Zellen mit direkt benachbarten Index einen eindeutigen Folgezustand bestimmt. Eine Zuordnung, die einem gegebenen 3-Tupel von Zellen einen Folgezustand zuweist wird Regel genannt. Es sind Regeln so zu wählen, dass nach einer Folge von Schritten alle Zellen zeitgleich in den Zustand f (“feuern”) übergehen und dieser Zustand in keinem vorhergehenden Schritt von einer Zelle angenommen wird. Die Regeln $(s, s, s) \mapsto s$ und $(s, s, \#) \mapsto s$ sind vorgegeben und besagen, dass jede Änderung von der Zelle mit Zustand g ausgeht.

Die optimale Zeit für eine Lösung ist bekannt. Durch Untersuchung aller Fälle konnte in [10] gezeigt werden, dass vier mögliche Zustände für die Zellen (einschließlich g, s und f aber ohne $\#$) nicht reichen, um das Problem in optimaler Anzahl Schritte zu lösen. Dafür wurde ein Algorithmus angewandt, dem Backtracking zugrunde liegt. Man beginnt mit einer Instanz der Länge $n = 2$ und dem Anfangszustand. Immer wenn die Liste der bereits definierten Regeln nicht reicht den Folgezustand einer Zelle zu bestimmen, wird eine Verzweigung erzeugt, die alle möglichen Folgezustände untersucht. Es werden so immer wieder Regeln gebildet, bis sich ein Widerspruch ergibt, oder nach der optimalen Anzahl Schritte die Zellen nicht alle im Feuerzustand sind. Da für jedes Zustands 3-Tupel bei vier Zuständen auch vier Mögliche Nachfolgezustände gewählt werden können, erhält man einen implizit definierten Baum mit einem maximalen Verzweigungsgrad von 4. Da sich bei einigen Regeln schnell ein Widerspruch ergibt, während bei anderen die Regeln für eine Instanz ein korrektes Ergebnis liefern und erst mit einem größeren n ein Problem auftritt, ergibt sich ein stark unregelmäßiger Baum. Dennoch lieferte die hier beschriebene Lastverteilung für dieses Problem fast linearen Speedup.

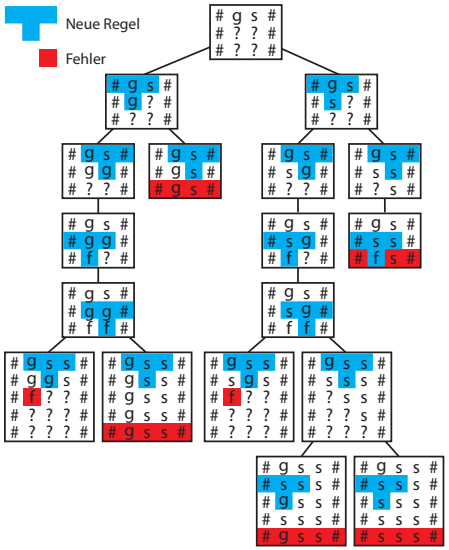


Abbildung 6.4: Beweis, dass keine Lösung des FFSP Problems in Optimalzeit mit 3 Zuständen existiert. Regeln die Zustand f ergeben werden nur für den letzten Schritt überprüft, dort werden nur solche hinzugenommen.

6.3.2 Random Polling

Nach den Beispielen für Anwendungsgebiete nun zum eigentlichen Lastverteilungsalgorithmus. Man möchte eine Laufzeit bekommen, für die

$$T(n,p) \leq (1+\epsilon)\frac{T_{\text{seq}}}{p} + \text{Terme niedriger Ordnung}$$

gilt, wobei diese Terme niedriger Ordnung vor allem den Kommunikationsaufwand darstellen. In diesen Termen sollte p nicht linear und erst recht nicht superlinear vorkommen. Einen ersten Ansatz bietet folgender Algorithmus aus der Familie der "receiver initiated load balancing" Algorithmen:

Algorithmus 6.3: Random Polling

```
L, L' : Subproblem;
if (i = 0) L = L_root;
else L = L_∅;
while(work left){
  L := work( L , Δt);
  m' := |{j : L@j = L_∅}|;
  if(m' ≥ m){
    if(L = L_∅){
      send request to random PE k;
    }
    if(receive request M from PE j){
      (L, L') := split(L);
      send L' to PE j;
      send L_∅ for any other request;
    }
    if(L = L_∅){
      receive new L from k;
    }
  }
}
```

Erläuterungen: Gemäß der Definition hält ein Prozessor zunächst das gesamte Problem L_{root}. Ein Schritt des Algorithmus beginnt damit, dass jeder Prozessor Δt Zeit an seinem Problem arbeitet. Es wird anschließend gezählt, wie viele Prozessoren ihren Teiljob abgearbeitet haben. An dieser Stelle wird auch überprüft, ob überhaupt noch Arbeit vorliegt. Gibt es zwar noch Jobs, aber auch mindestens m Prozessoren die nur noch ein leeres Teilproblem $L_\emptyset$ haben, fragen sie bei zufälligen anderen Prozessoren nach Arbeit. Die angeschriebenen Prozessoren versuchen ihren eigenen Job aufzuteilen und

einen Teil abzugeben. Kann der Job nicht aufgeteilt werden – weil der eigene Job schon leer ist oder mehr als eine Anfrage vorliegt – so wird das leere Problem abgegeben.

Es können auch Suchprobleme betrachtet werden, bei denen es bereits reicht einen Knoten zu finden, der die Vorgaben erfüllt. Dafür muss zusätzlich der Fall behandelt werden, dass ein Prozessor die Lösung gefunden hat. Dann kann für diese Probleme aber superlinearer Speedup erreicht werden. In der Theorie sollte es sich über mehrere Ausführungen statistisch ausgleichen. In der Praxis existiert bei vielen Ausführungen ein großer Teilbaum, der im sequentiellen Fall erst komplett abgearbeitet wird, aber im parallelen Fall nur einen Prozessor auslastet.

Für die Bestimmung der Laufzeit werden die Iterationen separat gezählt, in denen nur gerechnet werden muss und die, in denen auch Jobs umverteilt werden. Ist $m < p$ aber gilt noch immer $m \in \Omega(p)$, so reichen mit hoher Wahrscheinlichkeit $O(h)$ Iterationen, in denen Probleme aufgeteilt werden müssen, bis jeder noch verbleibende Teiljob atomar ist. Es fallen also für $O(h)$ Iterationen Kommunikationskosten an und es müssen dann noch die atomaren Jobs bearbeitet werden. Die Anzahl der übrigen Phasen mit höchstens m arbeitslosen Prozessoren lässt sich durch $\frac{T_{\text{seq}}}{(p-m)\Delta t}$ beschränken. Man erwartet für $m \in \Omega(p)$ also eine Gesamtanzahl von Interationen, die kleiner ist als

$$c\frac{T_{\text{seq}}}{p\Delta t} + dh$$

für geeignete Konstanten c und d. Daraus ergibt sich eine reine Bearbeitungszeit von

$$(c\frac{T_{\text{seq}}}{p\Delta t} + dh)\Delta t = (c\frac{T_{\text{seq}}}{p} + dh\Delta t)$$

Die Konstante c ist durch die Wahl von m bestimmt, und der von h abhängige Term lässt sich über Δt beliebig drücken. Für geeignete m und Δt, erhält man also für jedes ϵ die Abschätzung $(1+\epsilon)\frac{T_{\text{seq}}}{p}$ für die Zeit, in der das Problem bearbeitet ist. In den $\tilde{O}(h)$ Aufspaltungsschritten muss wegen der zufälligen Partnerwahl eine kollektive Kommunikationsoperation für den Austausch angenommen werden. Deswegen kommt zu der Zeit einer Aufspaltung T_{split} und der Zeit für die Übertragung einer Problembeschreibung $T_{\text{rout}}(l)$ noch die Zeit für einen kollektiven Nachrichtenaustausch T_{coll}.

Analyse:

$$\forall \epsilon \exists \Delta t, m : T(n,p) \leq (1+\epsilon)\frac{T_{\text{seq}}}{p} + \tilde{O}(h(T_{\text{rout}}(l) + T_{\text{coll}} + T_{\text{split}}) + T_{\text{atomic}})$$

In dem nun beschriebenen Algorithmus wird der zuletzt erwähnte kollektive Nachrichtenaustausch vermieden:

Algorithmus 6.4: Simplified Random Polling

```
L, L' : Subproblem;
if (i = 0) L = L_root;
else L = L_∅;
while(work left){
  L := work(L, Δt);
  if(L@((i+s) mod p) = L_∅){
    (L, L') := split(L);
    send L' to PE ((i+s) mod p);
  }
}
```

Erläuterungen: Man verzichtet hier auf die Überprüfung, ob m Prozessoren inaktiv sind und führt stattdessen in jedem Schritt eine Balancierungsphase durch. Statt einen zufälligen Prozessor anzuschreiben, wird global ein gemeisamer zufälliger offset s bestimmt. Damit er auf allen Prozessoren gleich ist, reicht es bereits sich zu Beginn auf einen gemeinsamen Seed für die lokalen Zufallsgeneratoren festzulegen. Jeder Prozessor ist in einem Schritt dafür verantwortlich seine Arbeit mit dem zyklisch um s versetzten Prozessor zu teilen, falls dieser keine eigene Arbeit hat. Es ist wichtig, dass s zufällig gewählt wird. Wäre es konstant, so hätte das zur Folge, dass sich die Arbeit mit wachsendem Abstand zu dem Prozessor, der zu Beginn den Gesamten Job hatte, immer wieder halbiert. Die regelmäßig um einen zufälligen Wert versetzte Anfrage sorgt dafür, dass jeder Prozessor seine Arbeit mit genau einem einzigen teilen muss. Davon profitiert die Lastverteilung, da nicht mehrere Anfragen an einen Prozessor gestellt werden können, von denen nur eine beantwortet wird. Dadurch fällt auch der Term T_{coll} in der Analyse weg.

Analyse:

$$\forall \epsilon \exists \Delta t : T(n,p) \leq (1+\epsilon)\frac{T_{\text{seq}}}{p} + \tilde{O}(h(T_{\text{rout}}(l) + T_{\text{split}}) + T_{\text{atomic}})$$

Dass alle Schritte synchron stattfinden müssen ist eine unangenehme Forderung, die in den seltensten Fällen ohne hohen Zusatzaufwand erfüllt werden kann. Praktisch relevant ist also insbesondere der folgende asynchrone Algorithmus:

Algorithmus 6.5: Asynchronous Random Polling

```
L, L' : Subproblem;
if (i = 0) L = L_root;
else L = L_∅;
while(work left){
  if(L = L_∅){
    send request to random PE k;
  }else{
    L := work(L, Δt);
  }
  foreach(M ∈ received Messages){
    if(M is request from PE j){
      (L, L') := split(L);
      send L' to PE j;
    }else{
      L := M;
    }
  }
}
```

Erläuterungen: Ein Vorteil ist, dass ein Prozessor seine Arbeit direkt fortsetzen kann, wenn keine Anfragen vorliegen und nicht erst warten muss, bis auch die anderen Prozessoren ihre Kommunikation abgeschlossen haben. Die Funktionsweise des Algorithmus ist ansonsten sehr ähnlich zum ersten Fall, mit dem Unterschied, dass ein Prozessor sofort neue Arbeit anfordert, wenn er keine eigene hat. Dementsprechend ist auch die Laufzeit wieder vergleichbar.

Analyse:

$$\forall\epsilon\exists\Delta t : \mathbb{E}T(n,p) \leq (1+\epsilon)\frac{T_{\text{seq}}}{p} + O(h(\frac{1}{\epsilon} + T_{\text{rout}}(l) + T_{\text{split}}) + T_{\text{atomic}})$$

Literaturverzeichnis

[1] Richard J. Anderson, Ernst W. Mayr, and Manfred K. Warmuth. Parallel approximation algorithms for bin packing. *Inf. Comput.*, 82(3):262–277, 1989.

[2] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Linear time bounds for median computations. In *STOC '72: Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 119–124, New York, NY, USA, 1972. ACM.

[3] E. Dekel, D. Nassimi, and S. Sahni. Parallel matrix and graph algorithms. *SIAM Journal on Computing*, 10:657–675, 1981.

[4] S. Lennart Johnsson and Ching-Tien Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Comput.*, 38(9):1249–1268, 1989.

[5] Richard Karp and Yanjun Zhang. A randomized parallel branch-and-bound procedure. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 290–300, New York, NY, USA, 1988. ACM.

[6] Dénes Kőnig. Gráfok és alkalmazásuk a determinánsok és a halmazok elméletére. *Matematikai és Természettudományi Értesítő*, 34:104–119, 1916.

[7] Felix Putze, Peter Sanders, and Johannes Singler. Mcstl: the multi-core standard template library. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 144–145, New York, NY, USA, 2007. ACM.

[8] Sanguthevar Rajasekaran and John H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM Journal on Computing*, 18:594–607, 1989.

[9] Abhiram Ranade. A simpler analysis of the karp-zhang parallel branch-and-bound method. Technical Report UCB/CSD-90-586, EECS Department, University of California, Berkeley, Berkeley, CA, USA, Aug 1990.

[10] Peter Sanders. Massively parallel search for transition-tables of poly-automata. In *In Parcella 94, VI. International Workshop on Parallel Proccessing by Cellular Automata and Arrays*, pages 99–108, 1994.

[11] Peter Sanders. Fast priority queues for parallel branch-and-bound. In *IRREGULAR '95: Proceedings of the Second International Workshop on Parallel Algorithms for Irregularly Structured Problems*, pages 379–393, London, UK, 1995. Springer Verlag.

[12] Peter Sanders and Roberto Solis-Oba. How helpers hasten h-relations, 2000.

[13] Peter Sanders and Jesper Larsson Träff. The hierarchial factor algorithm for all-to-all communication.

[14] Peter Sanders and Thomas Worsch. *Parallele Programmierung mit MPI – ein Praktikum*. Logos-Verlag, Berlin, 1997.

[15] P. Tsigas and Yi Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on sun enterprise 10000. *Parallel, Distributed and Network-Based Processing, 2003. Proceedings. Eleventh Euromicro Conference on*, pages 372–381, Feb. 2003.

[16] Peter J. Varman, Scott D. Scheufler, Balakrishna R. Iyer, and Gary R. Ricard. Merging multiple lists on hierarchical-memory multiprocessors. *J. Parallel Distrib. Comput.*, 12(2):171–177, 1991.