Diplomarbeit

Paralleles Sortieren am Beispiel der schnellen Multipolmethode

TECHNISCHE UNIVERSITÄT
CHEMNITZ

Michael Hofmann

Betreuer:

Prof. Dr. Gudula Rünger
Technische Universität Chemnitz
Fakultät für Informatik
Professur für Praktische Informatik

Dr. Holger Dachsel
Forschungszentrum Jülich GmbH
Zentralinstitut für Angewandte Mathematik

20. Dezember 2005

Inhaltsverzeichnis

1 Einleitung

Das Sortieren einer Folge von Elementen gehört zu einem der meist auftretenden Probleme im Bereich der Informatik und der Angewandten Mathematik. Während in der Literatur das Sortieren meist zum reinen Selbstzweck behandelt wird, tritt es in der Praxis vielmehr immer als Bestandteil einer übergeordneten Anwendung auf. Daraus ergeben sich in vielen Fällen zusätzliche Anforderungen, beispielsweise bezüglich der Laufzeit- oder Speicherkomplexität, welche die Auswahl der verfügbaren Sortierverfahren stark einschränkt. Ein umfangreicher Überblick zum Thema Sortieren und den zahlreichen existierenden Verfahren wird in [6] gegeben.

Die schnelle Multipolmethode (FMM, eng. *Fast Multipole Method*) ist eine linear skalierende Methode zur Berechnung langreichweitiger Wechselwirkungen von Teilchensystemen. Die am Zentralinstitut für Angewandte Mathematik (ZAM) des Forschungszentrums Jülich entstandene Implementierung der FMM dient zur effizienten Berechnung von Coulombwechselwirkungen. Damit kann sie beispielsweise als Bestandteil von Molekulardynamiksimulationen in den Bereichen *Computational Chemistry & Physics* eingesetzt werden.

Innerhalb der FMM werden an mehreren verschiedenen Stellen Daten sortiert. Um die Laufzeitkomplexität der gesamten Implementierung nicht zu beeinflussen, müssen dafür linear skalierende Verfahren eingesetzt werden. Darüber hinaus ist es möglich, dass der zur Verfügung stehenden Hauptspeicher zu einem Grossteil von den zu sortierenden Daten selbst belegt ist. Das Sortierverfahren muss deshalb auch mit nur wenig zusätzlichem Speicher auskommen können und die Daten "an Ort und Stelle" (*in-place*) sortieren. In der bisherigen sequentiellen Implementierung der FMM wird dafür ein einfaches Radixsort-Verfahren verwendet.

Die Simulation von immer grösseren Systemen geht dabei einher mit einer steigenden Rechenzeit und einer ständig wachsenden Menge der zu verarbeitenden Daten. Dies macht die Verwendung von Parallelrechnern unumgänglich und eine parallele Implementierung der FMM notwendig. Im gleichen Zug werden dadurch, schon allein auf Grund der verteilt vorliegenden Daten, parallele Sortierverfahren benötigt, welche ebenfalls die Anforderungen der FMM bezüglich Laufzeit- und Speicherkomplexität erfüllen. Als Parallelrechner stehen am ZAM das IBM Regatta p690 System JUMP und das Blue Gene/L System JUBL zur Verfügung.

Paralleles Sortieren wurde in der Vergangenheit ebenfalls ausführlich untersucht und es existieren eine Vielzahl an verschiedenen Verfahren. Ein genereller Überblick wird in [16, 17] gegeben und mit [18] existiert eine Aufstellung über die verfügbare Literatur bis 1986. Da viele parallele Sortierverfahren auch ein sequentielles enthalten, ist es in Verbindung mit einem Radixsort-Verfahren teilweise leicht möglich linear skalierende Verfahren zu

realisieren. Darüber hinaus existieren aber auch spezielle parallele Radixsort-Verfahren [8, 9, 10]. Diese benötigen jedoch alle zusätzlichen Speicher in Abhängigkeit von der Anzahl der zu sortierenden Elemente. In den meisten Fällen wird dieser bereits durch die Verwendung von kollektiven Kommunikationsoperationen nötig da hierfür nur Verfahren bekannt sind, die getrennte Sende- und Empfangsbereiche voraussetzen. Das einzige echte parallele *in-place* Sortierverfahren ist ZZ-sort [11]. Durch die Verwendung einer *Min-Max-Heap*-Datenstruktur [12] besitzt es jedoch keine lineare Laufzeitkomplexität mehr.

Obwohl paralleles Sortieren eine immer wiederkehrende Aufgabe darstellt, beginnt meist jeder Anwender von neuem die immer gleichen Verfahren selbst zu implementieren. Allgemeine Programmbibliotheken, wie sie beispielsweise mit *BLAS* oder *LAPACK* im Bereich des wissenschaftlichen Rechnens bereits existieren, sind für Sortierverfahren nicht bekannt.

Ziel dieser Arbeit war es Sortierverfahren zu finden und zu implementieren, die es der parallelen Implementierung der FMM erlauben ihre Daten *in-place* mit einem linear skalierenden Verfahren zu sortieren. Um eine einfache Wiederverwendbarkeit der entstandenen Verfahren zu gewährleisten, wurden sie flexibel und unabhängig vom konkreten Anwendungsfall der FMM implementiert. Damit wird versucht einen Ansatz zu geben, wie eine Bibliothek von sequentiellen als auch parallelen Sortierverfahren realisiert werden kann.

2 Die schnelle Multipolmethode (FMM)

Die schnelle Multipolmethode von Greengard und Rokhlin [1] ist ein Verfahren zur Berechnung der Coulombwechselwirkungen von Teilchensystemen. Ein derartiges System, bestehend aus n Teilchen, wird durch die Positionen x_j, y_j, z_j und Ladungen q_j der Teilchen ($1 \leq j \leq n$) beschrieben. Die Coulomb-Energie E_C des Systems, die Potentiale Φ_j und der Gradient ∇E lassen sich dann wie folgt berechnen:

$$
\begin{aligned}
E_C &= \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} \frac{q_j q_k}{r_{jk}} \\
\Phi_j &= \sum_{\substack{k=1 \\ k \neq j}}^{n} \frac{q_k}{r_{jk}} \\
\nabla E &= \begin{pmatrix} \frac{\partial E}{\partial x_1} \\ \frac{\partial E}{\partial y_1} \\ \frac{\partial E}{\partial z_1} \\ \vdots \\ \frac{\partial E}{\partial x_n} \\ \frac{\partial E}{\partial y_n} \\ \frac{\partial E}{\partial z_n} \end{pmatrix} \qquad \begin{aligned} \frac{\partial E}{\partial x_j} &= q_j \sum_{\substack{k=1 \\ k \neq j}}^{n} \frac{q_k (x_k - x_j)}{r_{jk}^3} \\ \frac{\partial E}{\partial y_j} &= q_j \sum_{\substack{k=1 \\ k \neq j}}^{n} \frac{q_k (y_k - y_j)}{r_{jk}^3} \\ \frac{\partial E}{\partial z_j} &= q_j \sum_{\substack{k=1 \\ k \neq j}}^{n} \frac{q_k (z_k - z_j)}{r_{jk}^3} \end{aligned} \\
r_{jk} &= \sqrt{(x_j - x_k)^2 + (y_j - y_k)^2 + (z_j - z_k)^2}
\end{aligned}
$$

Diese Vorgehensweisen zur Berechnung der Wechselwirkungen besitzen eine Laufzeitkomplexität von $\mathcal{O}(n^2)$. Darüber hinaus existieren eine Vielzahl von Methoden mit geringer Komplexität [4, 5]. Eine Methode, anhand der sich die Wechselwirkungen mit einer Laufzeitkomplexität von $\mathcal{O}(n)$ berechnen lassen, ist die FMM.

Der Ansatz der FMM basiert auf der Entwicklung des reziproken Abstands $\frac{1}{r_{jk}}$ zwischen zwei Teilchen in Polarkoordinaten (r_j, θ_j, ϕ_j) und (r_k, θ_k, ϕ_k) mit Hilfe von Legendrepolynomen P_{lm}.

$$
\frac{1}{r_{jk}} = \sum_{l=0}^{\infty} \sum_{m=-l}^{l} \frac{(l-m)!}{(l+m)!} \frac{r_j^l}{r_k^{l+1}} P_{lm}(\cos\theta_j) P_{lm}(\cos\theta_k) e^{-im(\phi_j - \phi_k)} \quad , r_j < r_k
$$

Daneben wird, wie in den meisten Verfahren auch, eine Unterteilung der einzelnen Wechselwirkungen in Near- und Farfield vorgenommen. Die Farfield-Wechselwirkungen werden dann mit Hilfe der sog. Multipolentwicklungen und die Nearfield-Wechselwirkungen weiterhin klassisch berechnet.

2.1 Sortieren in der FMM

Die vorliegende Implementierung der FMM basiert auf den Arbeiten von White und Head-Gordon [2, 3]. In ihr müssen Daten an mehreren verschiedenen Stellen sortiert werden. Dies umfasst zum einen das Sortieren der Eingabedaten, da diese in der Regel nicht in der vom Benutzer zur Verfügung gestellten Reihenfolge verarbeitet werden können. Zum anderen müssen die Ausgabedaten so angeordnet werden, dass sie mit der ursprünglichen Reihenfolge der Eingabedaten übereinstimmen.

In beiden Fällen müssen Elemente, bestehend aus mehreren verschiedenen Daten, nach einem als Schlüssel ausgezeichneten Datum, sortiert werden. Die verschiedenen Daten aller Elemente liegen jeweils in einem separaten Feld vor. Die Anzahl der zu sortierenden Elemente hängt direkt von der Anzahl der Teilchen n des zu berechnenden Systems ab. Zusätzlich zu den Ein- und Ausgabedaten werden pro Teilchen noch weitere, implementierungsspezifische Daten erzeugt, welche ebenfalls mit sortiert werden müssen. Die Grösse eines zu berechnenden Systems wird somit, abgesehen von der Rechenzeit, schon allein durch den zur Verfügung stehenden Hauptspeicher begrenzt. Die Verwendung von *out-of-place* Sortierverfahren, welche die sortierte Folge in einem separaten Feld zurückgegeben, würde die maximale Grösse eines zu berechnenden Systems noch weiter einschränken. Es werden daher nur *in-place* Sortierverfahren verwendet, die keinen zusätzlichen Speicher in Abhängigkeit von n benötigen.

Eine weitere wichtige Anforderung betrifft die Laufzeitkomplexität. Da es sich bei der FMM um ein linear skalierendes Verfahren bezüglich n handelt ist es ebenfalls erforderlich, linear skalierende Sortierverfahren zu verwenden, um die Komplexität der gesamten Implementierung nicht zu beeinflussen.

2.1.1 Eingabedaten

Die Eingabedaten der FMM bestehen aus den Daten des zu berechnenden Systems von Teilchen, welche jeweils durch Position und Ladung gegeben sind. Darüber hinaus werden jedem Teilchen ein Adresswert, eine Boxnummer und ein nur teilweise benötigter Zwischenwert zugeordnet.

	Anzahl und Datentyp pro Teilchen
Boxnummer (`box`)	`1 integer`
Position (`xyz`)	`3 doubles`
Ladung (`q`)	`1 double`
Adresse (`addr`)	`1 integer`
Zwischenwert (`scr`)	`1 integer`

Tabelle 2.1: Zu sortierende Eingabedaten der FMM pro Teilchen

Der Adresswert wird gebildet, indem zu Beginn alle Teilchen durchnummeriert werden. Er dient dazu, die im Verlauf der Berechnung umgeordneten Teilchen abschliessend

wieder in ihre ursprüngliche Reihenfolge zu bringen. Während der Sortierung der Eingabedaten stellt der Adresswert nur ein weiteres, mitzuführendes Datum dar.

Der Schlüssel nach dem die Eingabedaten sortiert werden müssen ist die Boxnummer, welche im wesentlichen durch die Position eines Teilchens bestimmt wird. Dazu werden die Koordinaten aller Teilchen gleichmässig in den Bereich $[0,1] \times [0,1] \times [0,1]$ skaliert. Damit wird das gesamte System in einen Würfel mit Kantenlänge 1 eingebettet. Die weitere Vorgehensweise zur Bestimmung der Boxnummern wird in Abbildung 2.1 an einem zweidimensionalen Beispiel gezeigt.

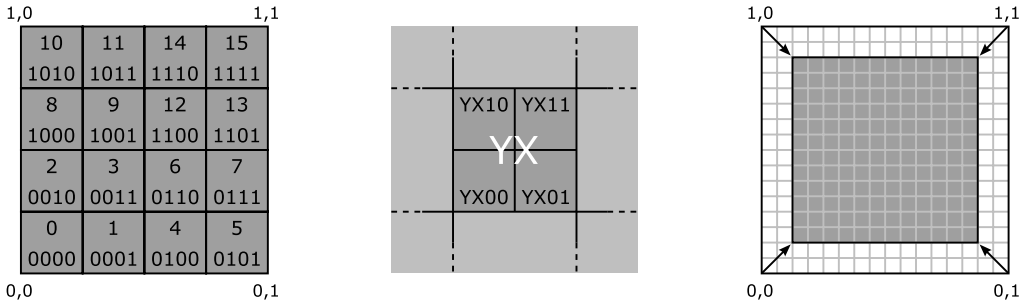

Abbildung 2.1: Zweidimensionales Beispiel zur Nummerierung der Boxen und abschliessender Skalierung des gesamten Systems.

In einem ersten Schritt wird die Ausgangsbox gleichmässig in 4 Kinderboxen unterteilt. In Abhängigkeit vom sog. Separationskriterium ws wird diese Unterteilung mit den Kinderboxen so oft wiederholt, bis zwischen zwei Boxen in mindestens einer Raumrichtung ws Boxen dazwischen liegen. Für $ws = 1$ werden deshalb im angegebenen Beispiel im ersten Schritt $4 \times 4 = 16$ Kinderboxen erzeugt. Diese werden nach einem festen Schema durchnummeriert und allen Teilchen jeweils die Boxnummer zugeordnet, in der sie sich befinden. Im nächsten Schritt werden die bestehenden Boxen wieder in 4 Kinderboxen unterteilt und die Boxnummern der einzelnen Teilchen entsprechend angepasst. Dies erfolgt, in dem die alte Boxnummer in Binärdarstellung um zwei Bitstellen nach links geschoben wird und die dadurch freiwerdenden niederwertigsten 2 Bitstellen für die Einteilung in die 4 neu entstandenen Kinderboxen verwendet werden. Dieser Vorgang wird mehrmals wiederholt, bis ein implementierungsspezifisches Abbruchkriterium erreicht ist. Abschliessend wird das gesamte System noch einmal in Richtung des Mittelpunkts der Ausgangsbox zusammengedrückt. Je nach Ausmass dieser Skalierung kann es durch diese letzte Änderung des Systems dazu kommen, dass einzelne Teilchen ihre ursprüngliche Box noch einmal verlassen. Dementsprechend werden auch die Boxnummern noch einmal angepasst.

Im dreidimensionalen Fall erfolgt mit $ws = 1$ die Unterteilung im ersten Schritt in $8 \times 8 = 64$ Kinderboxen und in jedem weiteren Schritt in jeweils 8 Kinderboxen. Nach jedem dieser Schritte müssen die Eingabedaten aller Teilchen anhand der neu zugeordneten Boxnummern sortiert werden. Es ergeben sich daraus drei verschiedene Szenarien, in denen die Eingabedaten sortiert werden müssen. Diese unterscheiden sich jeweils durch

den Wertebereich der Schlüssel und den Grad der Vorsortierung.

1. Nach der ersten Einteilung in 64 Boxen, sind nur Boxnummern von 0 bis 63 vergeben. Es werden daher nur die niederwertigsten 6 Bit verwendet und alle höherwertigeren Bit sind 0. Bei einem einzelnen Aufruf der FMM sind die Teilchen des Systems beliebig verteilt und liegen daher bezüglich ihrer Boxnummern vollkommen unsortiert vor. Im Rahmen von Molekulardynamiksimulationen ist es aber beispielsweise auch möglich, dass das Zurücksortieren der Ausgabedaten ausgesetzt wird und die FMM jeweils nur mit einem leicht veränderten System aufgerufen wird. In diesen Fällen wäre dann bereits eine Vorsortierung in unbekanntem Ausmass vorhanden.

2. Durch andauerndes verkleinern der Boxen nimmt deren Anzahl zu und der Wertebereich der Boxnummern wird grösser. In dem aber in jedem Schritt die ursprünglichen Boxnummern um 3 Bit nach links verschoben werden, sind die Teilchen anhand der neuen Boxnummern bereits bis auf die niederwertigsten 3 Bit vorsortiert.

3. Mit der abschliessenden Skalierung des gesamten Systems können sich noch einmal die Boxnummern aller Teilchen ändern. Die Gesamtzahl der Boxen hängt von der Anzahl der vorher durchgeführten Unterteilungen ab. Werden 64-Bit Integerwerte für die Boxnummern verwendet können maximal $2^{63} = 2^{3*21}$ verschiedene Boxen erzeugt und damit auch nur höchstens 21 Unterteilungsschritte durchgeführt werden. Je nach verwendetem Skalierungsfaktor zwischen 0.5 und 1 sind die Teilchen danach wieder unsortiert, teilweise vorsortiert oder immer noch sortiert.

Der Hintergrund für diese iterativen Vergabe der Boxnummern ist eine Optimierung bezüglich der Rechenzeit. Innerhalb einzelner Boxen und zwischen den ws nächsten Nachbarboxen werden die Nearfield-Wechselwirkungen direkt berechnet. Alle restlichen Wechselwirkungen gehören zum Farfield und werden mit Mutlipolentwicklungen berechnet. Durch die schrittweise Verkleinerung der Boxen wird der Nearfield-Anteil verringert und der Farfield-Anteil erhöht bis eine, bezüglich der Rechenzeit optimale Unterteilungstiefe erreicht ist. Durch Interpolation wird dann im Anschluss noch eine fraktionale Tiefe bestimmt, die näher an der optimalen Tiefe liegt als ein ganzzahliger Wert. In Abhängigkeit von der fraktionalen Tiefe wird dann das gesamte System zum Mittelpunkt der Ausgangsbox hin zusammengedrückt. Durch diese Veränderung des Systems werden an den Rändern leere Boxen erzeugt die bewirken, dass sich die optimale Tiefe dem ganzzahligen Wert annähert der dann im weiteren Verlauf verwendet wird. Die Veränderung der Koordinaten der Teilchen entsprechen einer einheitlichen Skalierung des gesamten Systems und können an den berechneten Ausgabedaten wieder rückgängig gemacht werden.

2.1.2 Ausgabedaten

Nach dem Abschluss der FMM, liegen sowohl die berechneten Ausgabedaten als auch die Eingabedaten in einer, für den Benutzer der FMM unbekannten Reihenfolge vor. Aus

diesem Grund ist es notwendig, die Daten wieder in die ursprüngliche Reihenfolge zu bringen. Zu diesem Zweck wurde, wie zuvor beschrieben, jedem Teilchen ein Adresswert zugeordnet, nach dem die Ausgabedaten zum Abschluss nocheinmal sortiert werden müssen.

	Anzahl und Datentyp pro Teilchen
Adresse (`addr`)	`1 integer`
Gradient (`g`)	`3 doubles`
Potential (`p`)	`1 double`
Ladung (`q`)	`1 double`

Tabelle 2.2: Zu sortierende Ausgabedaten der FMM pro Teilchen

Zusammen mit den Ausgabedaten, bestehend aus dem Gradienten und den Potentialen, müssen auch die Ladungen der einzelnen Teilchen wieder zurücksortiert werden. Diese sind ein Bestandteil der Eingabedaten und es ist vorgesehen, sie dem Benutzer der FMM in ihrer ursprünglichen Reihenfolge zurückzugeben. Da die Positionen der einzelnen Teilchen, wie in Abschnitt 2.1.1 beschrieben, abgesehen von der Umsortierung noch zusätzliche verändert wurden, werden sie nicht mit zurücksortiert.

2.2 Paralleles Sortieren in der FMM

Die Entwicklung einer parallelen Implementierung der FMM macht es erforderlich, dass auch die im vorherigen Abschnitt vorgestellte Sortierung der Ein- und Ausgabedaten parallelisiert wird. Dabei gilt es besonders die begrenzte bzw. flexible Verwendung von zusätzlichem Speicher zu ermöglichen und nur linear bezüglich der Anzahl Teilchen n skalierende Verfahren zu verwenden. Für die parallele Implementierung der FMM wurde das *Global Arrays Toolkit* [15] verwendet, welches im folgenden kurz vorgestellt wird.

2.2.1 Das Global Arrays Toolkit

Das *Global Arrays Toolkit* bietet eine *shared-memory*-artige Programmierumgebung für Parallelrechner mit physikalisch verteiltem Speicher. Zentraler Bestandteil sind *Global Arrays* genannte Datenstrukturen zum Anlegen von Feldern beliebigen Datentyps. Jedes dieser Felder wird automatisch auf die verschiedenen Prozesse eines parallelen Programms verteilt. Lesender und schreibender Zugriff auf die Daten eines *Global Arrays* wird durch die Funktionen `ga_get` bzw. `ga_put` ermöglicht. Diese verbergen die unterschiedlichen Zugriffsarten, einerseits direkt auf den lokalen Speicher eines Prozesses oder andererseits mittels einseitiger Kommunikation auf den verteilten Speicher. Des Weiteren ermöglicht es die Funktion `ga_access` mittels eines Zeigers direkt auf auf den lokalen Speicher eines Prozesses zuzugreifen. Darüber hinaus werden bestimmte Funktionalitäten des *Global Arrays Toolkit* mit Hilfe einer darunterliegenden MPI-Implementierung [32] realisiert.

2.2.2 Gegebenheiten der parallelen FMM

Die Verwendung des *Global Arrays Toolkit* für die parallele Implementierung der FMM hat zur Folge, dass für die Daten, deren Anzahl von der Grösse des zu berechnenden Systems abhängt, jeweils eigene *Global Arrays* angelegt werden. Der Inhalt jedes einzelnen *Global Arrays* wird gleichmässig auf die beteiligten Prozesse verteilt. Für ein System mit n Teilchen sind somit während der parallelen Ausführung des Programms mit p Prozessen jedem Prozess die Daten von $\frac{n}{p}$ Teilchen zugeordnet. Ungleiche Lastverteilungen auf Grund der Datenverteilung sind somit ausgeschlossen und müssen nicht berücksichtigt werden.

Die Verwendung der *Global Arrays* ermöglicht gleichzeitig die Nutzung einer zusätzlichen MPI-Implementierung. Daher werden die parallelen Sortierverfahren unter Verwendung von MPI realisiert. Dies erhöht die Flexibilität und Wiederverwendbarkeit der implementierten Verfahren. Des Weiteren ist darauf zu achten, dass sich die parallelen Sortierverfahren ohne Probleme innerhalb der in Fortran geschriebenen parallelen Implementierung der FMM verwenden lassen.

3 Sequentielles Sortieren

Ein wesentlicher Bestandteil fast aller parallelen Sortierverfahren ist ein sequentielles Sortierenverfahren. Jeder Prozess hat die ihm zugewiesenen Elemente, unabhängig von allen anderen Prozessen zu sortieren. Ein umfangreicher Überblick zum Thema Sortieren und zu existierenden Sortierverfahren wird in [6] gegeben. Im weiteren Verlauf werden allgemeine Folgen von Elementen betrachtet, wobei jedes Element einen Schlüssel enthält, nach dem sortiert wird.

Ein wichtiges Merkmal aller Verfahren ist die Laufzeitkomplexität. Sie gibt Auskunft über die Anzahl durchzuführender Operationen in Abhängigkeit von der Anzahl zu sortierender Elemente n. Sortierverfahren welche auf Vergleichen zwischen Elementen basieren (beispielsweise gegeben durch eine Ordnungsrelation), benötigen mindestens $n \log n$ dieser Vergleichsoperationen ([6] S. 183). Um die von der FMM geforderte lineare Skalierung zu erreichen, wird daher ein nicht auf Vergleichen basierendes sog. Radixsort-Verfahren verwendet.

Neben der Laufzeitkomplexität eines Sortierverfahrens ist auch die Speicherkomplexität von Bedeutung. Ein Verfahren arbeitet *in-place*, wenn es abgesehen von den Eingabedaten, nur noch eine konstante bzw. von der Grösse der Eingabedaten unabhängige Menge an zusätzlichem Speicher benötigt. Ein *in-place* Sortierverfahren gibt die sortierten Elemente in dem gleichen Feld zurück, in dem es auch die unsortierten Elemente übergeben bekommen hat. Im Gegensatz dazu verwendet ein *out-of-place* Sortierverfahren ein zusätzliches Feld für die Ausgabe und besitzt somit eine Speicherkomplexität von $\mathcal{O}(n)$.

Darüber hinaus ist noch der Begriff der Stabilität von Bedeutung. Ein Verfahren wird als stabil bezeichnet, wenn die Reihenfolge von Elementen mit identischen Schlüsselwerten untereinander nicht verändert wird.

3.1 Sortieren durch Einfügen

Sortieren durch Einfügen bezeichnet eine ganze Familie von Sortierverfahren basierend auf Vergleichen. In einem iterativen Prozess wird jeweils ein Element in eine bereits sortierte Folge von Elementen eingefügt. Die verschiedenen Verfahren unterscheiden sich unter anderem in der Art und Weise wie die Position ermittelt wird, an der eine neues Element eingefügt werden muss. Im folgenden soll das *in-place* Verfahren *Straight insertion sort* ([6] S. 80) kurz vorgestellt werden.

Listing 3.1: *Straight insertion sort*

```
1   for (i = 1; i < n; i++)
2   if (keys[i] < keys[i - 1])
3   {
4     j = i - 1;
5     t = keys[i];

7     do {
8       keys[j + 1] = keys[j];
9       j = j - 1;
10      if (j < 0) break;

12    } while (t < keys[j]);

14    keys[j + 1] = t;
15  }
```

In jedem Durchlauf der äusseren Schleife wird ein neues Element zu den bereits sortierten hinzugefügt und durch die innere Schleife bis an seine richtige Position bewegt. Wenn c die maximale Anzahl Stellen bezeichnet, die jedes Element zu Beginn von seiner Position in der sortierten Folge entfernt ist, besitzt *Straight insertion sort* eine Laufzeitkomplexität von $\mathcal{O}(cn)$. Für komplett unsortierte Folgen gilt $c \leq n$ und es ergibt sich eine Komplexität von $\mathcal{O}(n^2)$. Ist c dagegen eine Konstante unabhängig von n hat dies nur noch eine Komplexität von $\mathcal{O}(n)$ zur Folge.

Durch seine Einfachheit lässt es sich mit geringem Aufwand implementieren und eignet sich besonders zum Sortieren von kurzen Folgen von Elementen. Angewendet auf eine bereits vorsortierte Folge von Elementen (c konstant), kann es benutzt werden um effizient eine vollständige Sortierung zu erreichen.

3.2 Radixsort

Radixsort (auch Distributionsort oder Bucketsort) ist ein Sortierverfahren welches nicht auf Vergleichen zwischen einzelnen Elementen basiert. Stattdessen wird eine endliche Menge von Schlüsselwörtern $\mathcal{S} = \Sigma^k$ über einem beliebigen Alphabet Σ vorausgesetzt. Beispielsweise können 64-Bit Integerwerte in Binärdarstellung als Schlüsselwörter über dem Alphabet $\Sigma = \{0, 1\}$ mit $k = 64$ aufgefasst werden. Zeichenketten, als Schlüsselwörter über dem Alphabet $\Sigma = \{a, b, \ldots, z\}$ eignen sich ebenfalls zum Sortieren mit Radixsort.

Die zwei möglichen Varianten von Radixsort sollen im folgenden an einem Beispiel mit dreistelligen Binärzahlen ($\Sigma = \{0, 1\}, k = 3$) kurz vorgestellt werden. Ein Schlüsselwort hat in diesem Fall die Form $a_2a_1a_0$ mit $a_2, a_1, a_0 \in \Sigma$ wobei a_0 als niederwertigste (*least-significant-digit*, LSD) und a_2 als höchstwertigste Stelle (*most-significant-digit*, MSD) bezeichnet wird. Ein Radixsort-Verfahren sortiert die Schlüsselwörter in mehreren Schritten jeweils nach einer einzelnen Stelle. Wird von der niederwertigsten zur höchstwertigsten Stelle vorgegangen, spricht man von LSDF-Radixsort (*least-significant-digit-first*, Abbildung 3.1), bei entgegengesetzter Reihenfolge von MSDF-Radixsort (*most-significant-digit-first*, Abbildung 3.2).

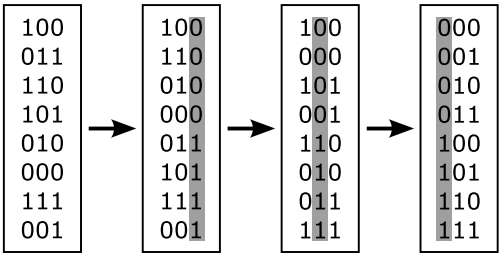

Abbildung 3.1: Beispiel zur Funktionsweise von LSDF-Radixsort.

In jedem Schritt werden bei LSDF-Radixsort alle Schlüsselwörter nach der jeweils betrachteten Stelle stabil sortiert. Ein instabiles linear skalierendes Verfahren wird später in Abschnitt 3.2.2 beschrieben. Stabilität kann jedoch nur erreicht werden, wenn zusätzlicher Speicher für die Ausgabe der sortierten Elemente zur Verfügung steht. LSDF-Radixsort ist daher kein *in-place* Sortierverfahren.

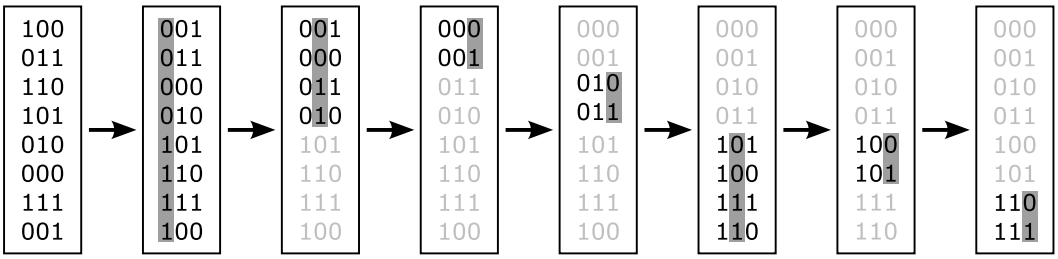

Abbildung 3.2: Beispiel zur Funktionsweise von MSDF-Radixsort.

MSDF-Radixsort beginnt dagegen mit der höchstwertigsten Stelle. Sind alle Schlüsselwörter nach dieser Stelle sortiert, lassen sich daraus $|\Sigma|$ Klassen von Elementen bilden. Für jede Klasse mit mehr als einem Element, wird diese Sortierung rekursiv, anhand der nächst niedrigeren Stelle wiederholt. Die maximale Rekursionstiefe wird durch k begrenzt, da nach dem Sortieren anhand der niederwertigsten Stelle alle verbleibenden Klassen nur noch Elemente mit identischen Schlüsselwörtern enthalten.

Steht zusätzlich $\mathcal{O}(n)$ Speicher zur Verfügung ist LSDF-Radixsort in der Regel leichter zu implementieren. Durch die Rekursivität von MSDF-Radixsort ergibt sich dagegen eine höhere Datenlokalität. In [14] wird daher eine Kombination beider Varianten vorgeschlagen. Diese beginnt mit MSDF-Radixsort und verwendet anschliessend für einzelne Klassen welche in den Datencache passen ein effizienteres LSDF-Radixsort. Für ein *in-place* Sortierverfahren kann diese Vorgehensweise jedoch nicht angewendet werden, da LSDF-Radixsort $\mathcal{O}(n)$ zusätzlichen Speicher für die Ausgabe benötigt. Im folgenden wird daher näher erklärt, wie sich MSDF-Radixsort realisieren lässt.

Auf Grund der Anwendung in der FMM wird im weiteren Verlauf hauptsächlich das Sortieren von k-Bit Integerwerten erläutert. Durch eine flexible Implementierung ist dabei die Angabe der zu verwendenden niederwertigsten und höchstwertigsten Bitstelle zur Laufzeit möglich. Des Weiteren ist eine spezielle Variante vorhanden die es erlaubt, genau bekannte Vorsortierungen gezielt auszunutzen. Damit können beispielsweise die in Abschnitt 2.1.1 beschriebenen, bis auf die niederwertigsten 3 Bitstellen vorsortierten Boxnummern effizient sortiert werden.

3.2.1 Radix-Exchange

Eine der einfachsten Implementierungen von MSDF-Radixsort ist *Radix-Exchange* [7]. Es stellt einen Spezialfall zum Sortieren von Schlüsselwörtern aus einem binären Alphabet dar, der leicht *in-place* zu realisieren ist ([6] S.123). Das in Abbildung 3.2 dargestellte Beispiel verwendet *Radix-Exchange*.

Beginnend mit dem höchstwertigen Bit werden die Elemente, je nachdem ob das betrachtete Bit eine 0 oder eine 1 enthält, in zwei Klassen aufgeteilt. Dann wird die Unterteilung anhand des nächst niedrigeren Bits, jeweils getrennt auf den Elementen der zwei zuvor

entstanden Klassen, rekursiv wiederholt. Die Rekursion bricht ab, wenn eine Klasse nur noch ein Element enthält oder das niederwertigste Bit behandelt wurde. Für das Sortieren von k-Bit Integerwerten ist eine maximale Rekursionstiefe von k gegeben und jedes Elemente wechselt höchstens $k$mal seine Position. Es ergibt sich somit eine Laufzeitkomplexität von $\mathcal{O}(kn)$ und da k unabhängig von n ist, ist *Radix-Exchange* ein linear skalierendes Sortierverfahren.

Die Struktur des Algorithmus besitzt grosse Ähnlichkeiten zum klassischen *Quicksort*-Algorithmus. Bereits vorhandene Implementierungen von *Quicksort* [30] können schon mit geringem Aufwand für die Realisierung von *Radix-Exchange* herangezogen werden. Anstelle der Verwendung von Pivot-Elementen werden für *Radix-Exchange* die einzelnen Bitstellen zur Unterteilung der Schlüsselwörter verwendet.

3.2.2 Count-Split-Radixsort

Während bei *Radix-Exchange* die Schlüsselwörter immer nur anhand einer Bitstelle in zwei Klassen unterteilt werden, ist bei einem allgemeinen Radixsort-Algorithmus auch eine Unterteilung in mehr als zwei Klassen möglich. Werden für einen Aufteilungsschritt r aufeinander folgende Bitstellen betrachtet, hat dies eine Unterteilung in 2^r Klassen zufolge. Für das Sortieren von k-Bit Integerwerten beträgt die maximale Rekursionstiefe damit nur noch $\lceil \frac{k}{r} \rceil$, wodurch auch jedes Elemente höchstens $\lceil \frac{k}{r} \rceil$ mal seine Position wechseln muss.

Die Aufteilung der Elemente in die einzelnen Klassen erfolgt nun in zwei Phasen. In einem ersten Durchgang (*Count*) wird ermittelt wie viele Elemente jede Klasse nach der Aufteilung enthalten wird. Aus diesen Informationen können dann die Bereiche ermittelt werden, in die in einem zweiten Durchgang (*Split*) die Elemente der einzelnen Klassen einzusortieren sind.

Steht für die Ausgabe der Elemente ein zweites, separates Feld zur Verfügung lässt sich die *Split*-Operation leicht realisieren. Eine *in-place* Variante der *Split*-Operation ist in [13] als Bestandteil von *American Flag Sort* zu finden. Ein Nachteil dieser Implementierung ist jedoch, dass unterschiedslos jedes Element, und damit auch solche die sich bereits in ihrer richtigen Klassen befinden, kopiert wird. Im folgenden wird daher ein leicht modifizierte Variante dieser *Split*-Operation vorgestellt. Die zu sortierenden Elemente sind im Feld `keys` und die zuvor ermittelten Grössen der Klassen im Feld `count` gegeben.

Listing 3.2: *in-place Split*-Operation

```
1   p[0] = 0;
2   for (i = 1; i < k; i++) p[i] = p[i - 1] + count[i - 1];

4   end = 0;
5   for (i = 0; i < k; i++)
6   {
7     end += count[i];
8     j = p[i];
9     while (j < end)
10    {
11      c = class_of(keys[j]);
12      while (c != i)
13      {
14        d = class_of(keys[p[c]]);
15        if (d != c) xchange(keys[j], p[c]);
16        p[c]++;
17        c = d;
18      }
19    }
20    j++;
21  }
```

Die Funktionsweise ist ähnlich der von *American Flag Sort*, wobei nun jedoch alle Elemente übersprungen werden, die sich bereits in ihrer richtigen Klassen befinden. Dies kann sich insbesondere bei vorsortierten Folgen als effizienter erweisen.

Für die Ermittlung der Grössen der Klassen wird jedes Element einmal untersucht und mit der vorgestellten *Split*-Operation wird ebenfalls jedes Element direkt in seine Klasse kopiert. Durch die rekursive Vorgehensweise mit maximaler Tiefe $\lceil \frac{k}{r} \rceil$ nimmt jedes Element auch nur $\lceil \frac{k}{r} \rceil$ mal an einer *Split*-Operation teil. Somit ergibt sich auch für CS-Radixsort eine Laufzeitkomplexität von $\mathcal{O}(n)$.

Neben der effizienten Implementierung der *Split*-Operation ist für das Sortieren von binären Schlüsselwörtern die optimale Wahl des Parameters r entscheidend. Für *Cache Conscious Radixsort* [14] wird empfohlen r so zu wählen, dass $r \leq \log_2 S_{TLB} - 1$ gilt, wobei S_{TLB} die Anzahl der Einträge im *Translation Lookaside Buffer* (TLB) bezeichnet. Dies stellt sicher, dass die Anzahl der Klassen für die *Split*-Operation nicht die Anzahl der vorhandenen TLB-Einträge übersteigt und TLB-Caches effizient genutzt werden. Sind zusammen mit den eigentlichen Schlüsselwörtern noch d zusätzliche Daten zu sortieren und werden keine separaten Ausgabefelder verwendet, ergibt sich:

$$r \leq \log_2 \frac{S_{TLB}}{d+1}$$

Der jeweils konkret verwendete Wert für r wurde flexibel implementiert und kann zur Laufzeit angegeben werden. Ein optimaler Wert zum Sortieren von 32-Bit bzw. 64-Bit Integerwerten, ist empirisch bestimmt worden und wird als Standardwert verwendet.

Auf Grund des erhöhten Aufwands, sowohl durch die zusätzliche *Count*-Operation, als auch die aufwändige *Split*-Operation, ist CS-Radixsort sehr ineffizient beim Sortieren von sehr wenigen Elementen. Da die MSDF-Variante von Radixsort aber rekursiv arbeitet, tritt dieser Fall auch beim Sortieren grosser Mengen von Elementen auf und beeinflusst wesentlich die Laufzeit. Für eine effiziente Implementierung ist es daher unerlässlich, im Falle von "zu kleinen" Klassen ein effektiveres Verfahren anzuwenden. Zu diesem Zweck wird die Rekursion ab einer bestimmten Anzahl zu sortierender Elemente t abgebrochen. Somit müssen für eine vollständige Sortierung am Ende alle Elemente nur noch um maximal t Positionen bewegt werden. Dies wird durch einen abschliessenden Aufruf von *Straight insertion sort* erreicht. Der konkrete Wert für t ist ebenfalls empirisch bestimmt worden.

3.2.3 Vergleich

In Abbildung 3.3 sind die Laufzeiten der drei in diesem Abschnitt vorgestellten Varianten von Radixsort gegenübergestellt. Insbesondere wurde dabei auch ihr Verhalten bei vorsortierten bzw. bereits komplett sortierten Folgen von Elementen untersucht.

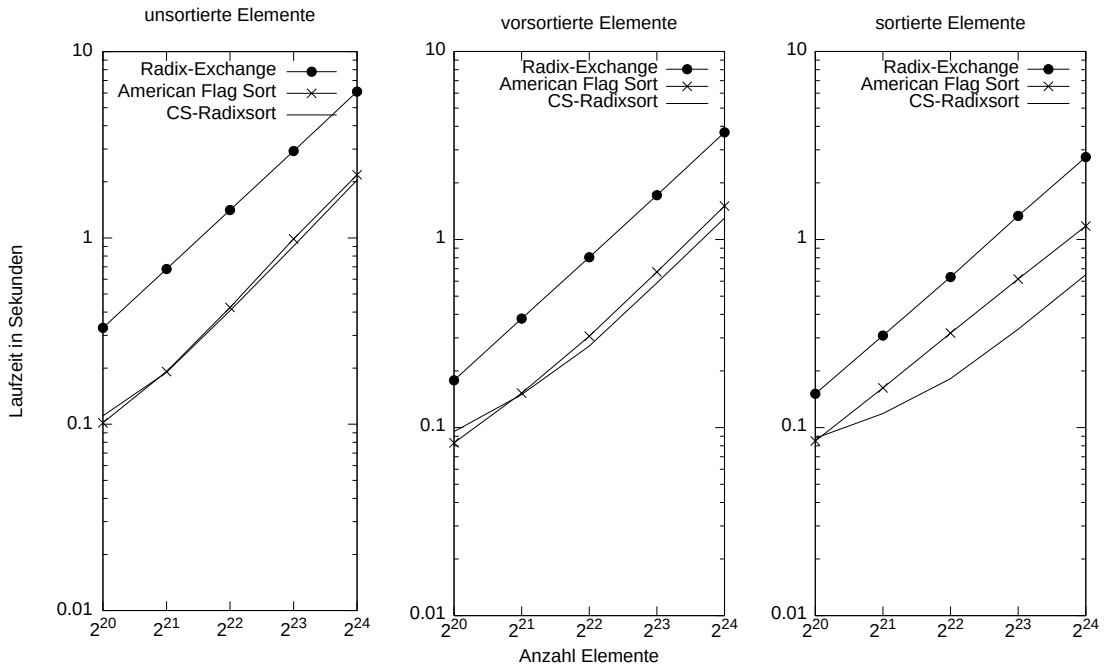

Abbildung 3.3: Laufzeiten der Radixsort-Verfahren mit unsortierten, vorsortierten und sortierten Folgen.

Wie der Vergleich zeigt, ist CS-Radixsort mit der vorgestellten *Split*-Operation das schnellste der drei implementierten Radixsort-Verfahren. Während bei unsortierten Elementen der Unterschied zu *American Flag Sort* nur gering ist, fällt er bei vorsortier-

ten und vollständig sortierten Elementen deutlicher aus. Dies kann auf die neue *Split*-Operation zurückgeführt werden, die bereits richtig positionierte Elemente überspringt anstatt sie, wie *American Flag Sort* unnötig zu kopieren. Die Laufzeiten für das einfache *Radix-Exchange* sind in jedem Fall höher als bei CS-Radixsort.

3.3 Sortieren mit Permutation

Einen Spezialfall stellt das Sortieren bzw. Umordnen anhand einer vorgegebenen Permutation dar. Eine Permutation ist eine bijektive Abbildung $\sigma : X_n \to X_n$ einer n-elementigen Menge auf sich selbst. Im vorliegenden Fall ist $X_n = \{0, \ldots, n-1\}$ die Menge der Positionen, die ein Element in einer zu sortierenden Folge einnehmen kann. Im folgenden ist ein Beispiel für eine Permutation in Matrixdarstellung mit $n = 8$ gegeben.

$$\sigma = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 7 & 2 & 6 & 1 & 0 & 5 & 3 & 4 \end{pmatrix}$$

Die in dieser Permutation enthaltenen Zyklen sind in Abbildung 3.4 dargestellt.

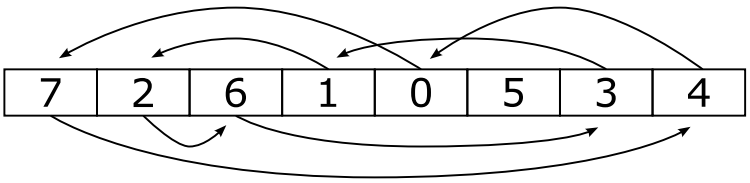

Abbildung 3.4: Darstellung der in einer Permutation enthalten Zyklen.

Eine Permutation σ kann als Vorlage zum Umordnen der Elemente, auf zwei verschiedene Arten interpretiert werden:

1. *vorwärts*: Ein Element an Position i soll sich nach dem Sortieren an Position $\sigma(i)$ befinden.
2. *rückwärts*: Ein Element an Position $\sigma(i)$ soll sich nach dem Sortieren an Position i befinden.

Ein *in-place* Verfahren welches eine gegebene Permutation *rückwärts* anwendet, ist in [6] (S. 595) gegeben. Analog dazu lässt sich, wie in Listing 3.3 dargestellt, auch eine *vorwärts*-Variante entwickeln. Die zu sortierenden Elemente sind im Feld `keys` gegeben und die anzuwendende Permutation in `perm`, wobei `perm[i]` den Wert $\sigma(i)$ enthält.

Listing 3.3: Sortieren mit Permutation: *vorwärts*

```
1   for (i = 0; i < n; i++)
2   {
3     while (perm[i] != i)
4     {
5       j = perm[i];
6
7       xchange(keys[i], keys[j]);
8
9       perm[i] = perm[j];
10      perm[j] = j;
11    }
12  }
```

Die äussere Schleife läuft über alle n Elemente und die innere jeweils über die Elemente eines Zyklus. Da jeder Zyklus nur einmal durchlaufen wird und jedes Element nur an einem Zyklus beteiligt ist, wird insgesamt durch die innere Schleife jedes Element nur einmal betrachtet. Somit besitzt der Algorithmus eine Laufzeitkomplexität von $\mathcal{O}(n)$.

Um bereits abgearbeitete Elemente zu markieren, wird in der vorliegenden Variante die im Feld `perm` gegebene Permutation überschrieben. Für diese Markierung kann aber auch eine ungenutzte Bitstelle in den Einträgen des Feldes `perm` verwendet werden. Mit dem Entfernen dieser Markierung lässt sich dann die ursprüngliche Permutation wieder herstellen.

In der FMM kann Sortieren mit Permutation verwendet werden, um die Ausgabedaten zu sortieren. Wie in Abschnitt 2.1.2 beschrieben wird zu diesem Zweck ein Adressfeld mitgeführt, welches die Position eines Teilchens zu Beginn angibt. Diese Adressen beschreiben eine Permutation anhand derer die Ausgabedaten mit der *vorwärts*-Variante zurücksortiert werden können. Dies hätte den Vorteil, dass jedes Teilchen direkt an seine jeweilige Position kopiert würde. Natürlich kann die Sortierung der Ausgabedaten auch mit einem normalen Sortierverfahren wie CS-Radixsort durchgeführt werden. In Abbildung 3.5 sind daher zum Vergleich die Laufzeiten dieser beiden Verfahren dargestellt.

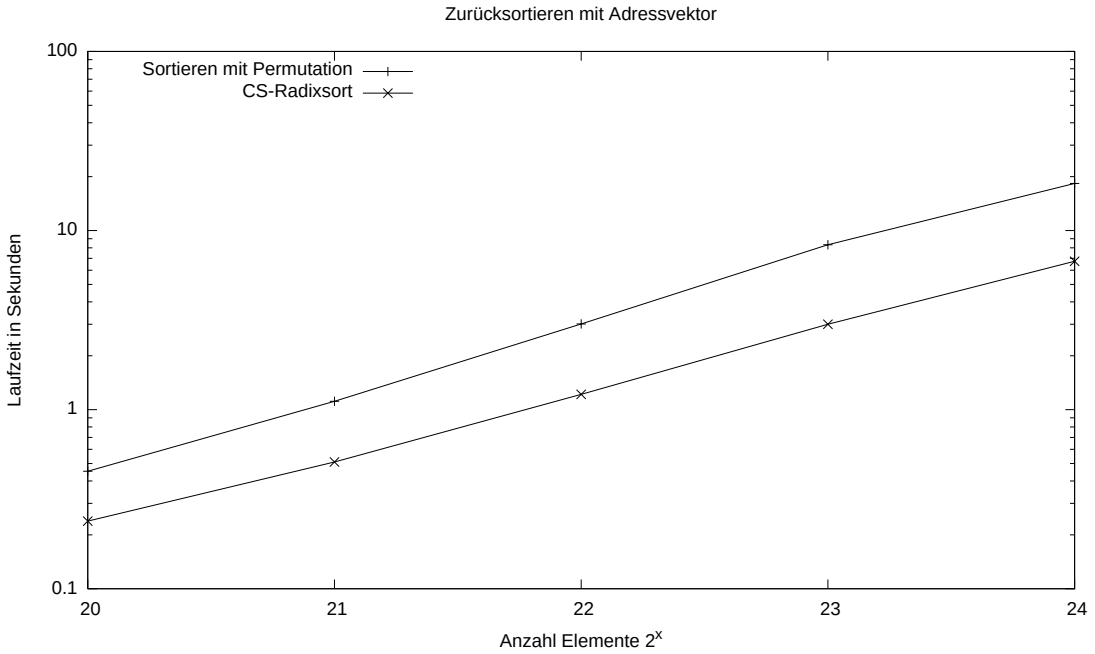

Abbildung 3.5: Laufzeiten für das Zurücksortieren mit Permutation und CS-Radixsort.

Die Vergleich zeigt, dass Sortieren mit Permutation ca. 2-3 mal langsamer ist. Dies steht jedoch im Widerspruch zu der Tatsache, dass mit CS-Radixsort jedes Element öfter kopiert wird.

Die Ursache für dieses Verhalten liegt in der geringen Datenlokalität beim Sortieren mit Permutation. Indem jedes Element direkt an seine richtige Position in einem Feld kopiert wird finden ständig auf dem gesamten Feld verteilte Zugriffe statt. Dies hat eine, im Vergleich zu CS-Radixsort, deutlich schlechtere Ausnutzung von Hardware-Caches zur Folge. Untersuchungen mit einem *Performance Monitoring Tool* [33] haben das bestätigt. Obwohl für CS-Radixsort ca. 60% mehr Instruktionen ausgeführt wurden, sind dennoch ca. 6 mal weniger Daten aus dem Hauptspeicher geladen worden. Darüberhinaus konnten bei CS-Radixsort im Durchschnitt doppelt so viele Speicherzugriffe durchgeführt werden bevor ein Zugriff auf den *Translation Lookaside Buffer* fehlschlug. Somit ist Sortieren mit Permutationen ein Verfahren, welches zwar theoretisch weniger Operationen benötigt, aber praktisch die gegebenene Hardware weniger effizient ausnutzt.

3.4 Umgang mit zusätzlichen Daten

Bisher wurden immer vollständige Elemente sortiert, wobei ein Teil jedes Elements der Schlüssel war, nachdem sortiert wird. In den meisten praktischen Fällen besteht ein Element darüber hinaus noch aus weiteren zusätzlichen Daten. Liegen Schlüssel und Daten eines Elements kontinuierlich im Speicher ist eine Kopieroperation ausreichend, um ein Element zu kopieren. Sind Schlüssel und Daten dagegen auf d verschiedene Felder verteilt, werden für das Kopieren eines Elements auch d Kopieroperationen benötigt.

Für die bisher vorgestellten Sortierverfahren gilt, dass jedes Element nicht direkt an seine endgültige Position kopiert wird, sondern diese erst über mehrere verschiedene Zwischenpositionen erreicht. Für die Schlüsselwerte eines Elements ist dies durchaus sinnvoll, nicht aber für die zusätzlichen Daten. Ein Alternative besteht darin, anstatt der Daten selbst nur eine Referenz auf die Daten zusammen mit den Schlüsseln zu sortieren. Diese Referenz kann beispielsweise durch eine Adresse realisiert werden, wie sie in der Implementierung der FMM bereits für das Zurücksortieren der Ausgabedaten verwendet wird. Nach dem Sortieren der Schlüssel und Adressen können dann die eigentlichen Daten unter Verwendung des Adressfeldes als Permutation mit der *rückwärts*-Variante sortiert werden. In Abbildung 3.6 ist diese Vorgehensweise an einem Beispiel dargestellt.

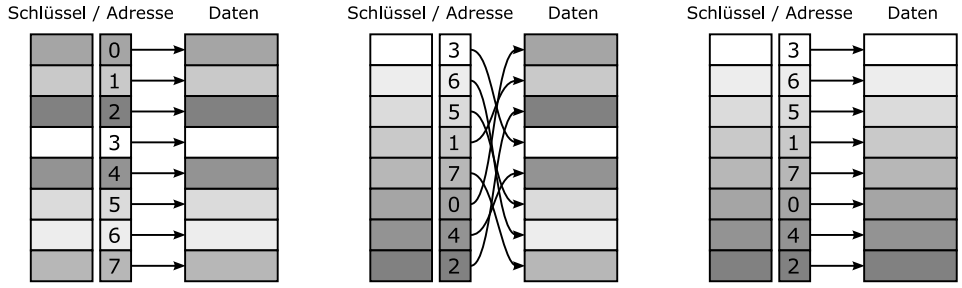

Abbildung 3.6: Vorgehensweise beim getrennten Sortieren von Schlüssel und Daten.

In dem die Daten direkt in die sortierte Reihenfolge gebracht werden und nicht alle Zwischenpositionen zusammen mit den Schlüsseln durchlaufen, werden effektiv weniger Kopieroperationen durchgeführt. Im Fall der FMM ist dieses Adressfeld bereits vorhanden, wodurch kein zusätzlicher Speicher dafür benötigt wird. Um den Inhalt des Adressfeldes nicht zu überschreiben, kann wie in Abschnitt 3.3 beschrieben eine ungenutzte Bitstelle pro Adresswert als Markierung verwendet werden. In Abbildung 3.7 sind die Laufzeiten beider Varianten gegenübergestellt.

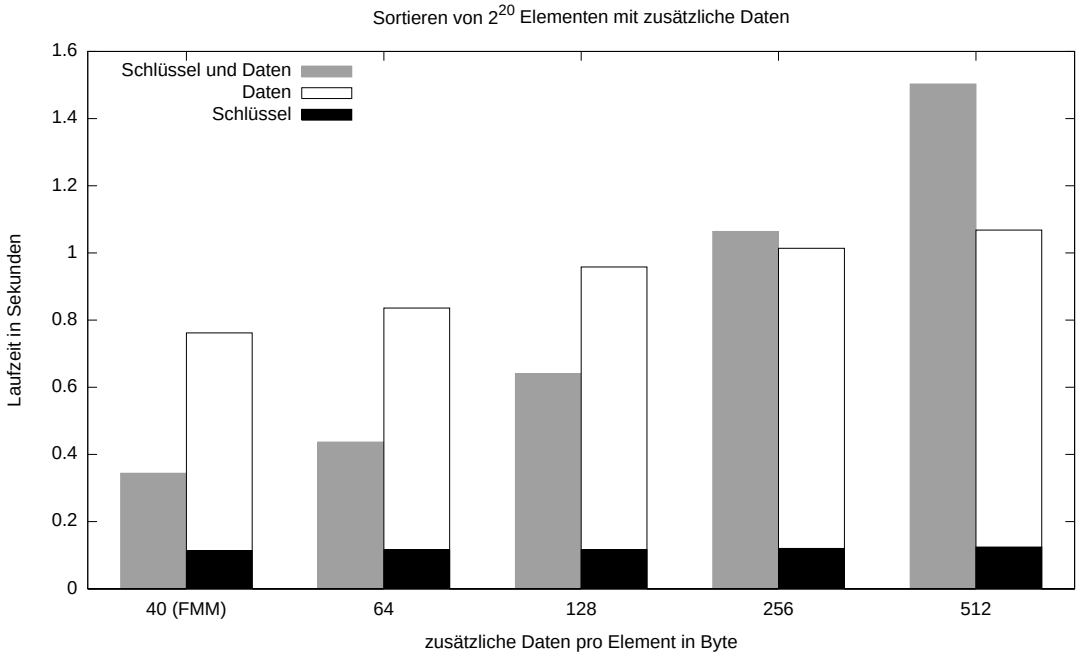

Abbildung 3.7: Laufzeiten für das gemeinsame und getrennte Sortieren von Schlüssel und Daten.

Es zeigt sich deutlich, dass die Anzahl der zusätzlich zu sortierenden Daten der FMM noch zu gering sind, als das sich der vorgestellte Ansatz lohnen würde. Der Grund dafür ist wieder die, wie in Abschnitt 3.3 beschrieben, geringe Effizienz beim Sortieren mit Permutation. Es zeigt sich jedoch auch, dass sich das Verhältnis mit zunehmender Grösse der zusätzlichen Daten ändert. Für die FMM ist es aber effizienter, Schlüssel und Daten als Einheit zu behandeln.

4 Paralleles Sortieren basierend auf Mischen

Parallele Sortierverfahren können bezüglich ihrer Vorgehensweise grob in zwei Kategorien eingeteilt werden. Die erste umfasst Verfahren basierend auf Mischen (*merge-based parallel sorting*). Wie in Abbildung 4.1 dargestellt, wird dabei in zwei Schritten vorgegangen.

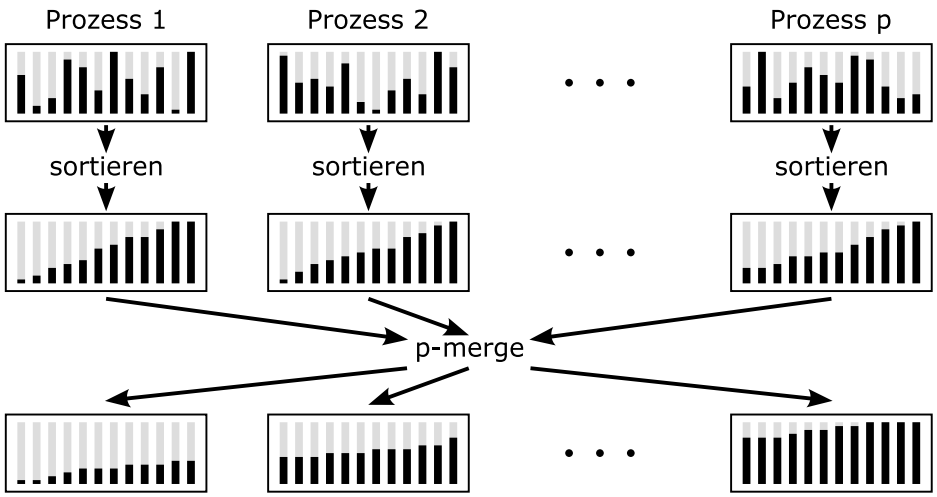

Abbildung 4.1: Paralleles Sortieren basierend auf Mischen.

Zu Beginn sind die unsortierten Elemente auf p Prozessen verteilt. Für das eigentliche Verfahren ist es unerheblich, ob die Elemente gleichmässig verteilt sind. Dennoch ist es für eine möglichst gleichmässige Lastverteilung unter den einzelnen Prozessen und damit auch für die Effizienz entscheidend. Eine eventuelle Umverteilung, falls erwünscht bzw. überhaupt möglich, hat bereits im Vorfeld zu erfolgen. Im Falle der FMM sind die Elemente bereits gleichmässig verteilt und eine Umverteilung durch die begrenzte Verfügbarkeit von zusätzlichem Speicher nahezu unmöglich.

Zuerst sortiert jeder Prozess unabhängig seine Elemente mit einem sequentiellen Sortierverfahren. Je nach Schlüsseltyp und weiteren bekannten Informationen wie zum Beispiel Wertebereich oder Vorsortierung der Schlüssel kann dafür ein entsprechend effizientes Verfahren verwendet werden.

Anschliessend werden die p sortieren Folgen in einem parallelen p-*merge*-Schritt zusammengemischt, wodurch sich eine auf p Prozessen verteilte sortierte Folge von Elementen ergibt. Im folgenden Abschnitt wird beschrieben, wie ein solcher p-*merge*-Schritt realisiert werden kann.

4.1 Das parallele p-merge-Problem

Das allgemeine *p-merge* Problem besteht, darin p sortierte Folgen von insgesamt n Elementen in eine sortierte Folge zu bringen. Für $p = n$ geht dies in das Sortieren von n Elementen über und kann wie in Kapitel 3 beschrieben behandelt werden. Der Fall $p = 2$ wird im Anschluss, als wichtiger Bestandteil des im folgenden vorgestellten Verfahren ausführlich behandelt.

Ein einfacher sequentieller Algorithmus zur Lösung des Problems entsteht aus der Verwendung eines Heaps. In diesen wird zu Beginn jeweils das kleinste Element jeder Folge eingefügt. Durch ständiges Entfernen des kleinsten Elements aus dem Heap und Einfügen des nächst kleineren Elements aus der jeweiligen Folge in den Heap werden alle Elemente in einer sortierten Folge zusammengemischt. Dieses Verfahren hat eine Laufzeitkomplexität von $\mathcal{O}(n \log p)$ und benötigt $\mathcal{O}(n)$ zusätzlichen Speicher für die Ausgabe.

Als Bestandteil eines parallelen Sortierverfahrens liegen die sortierten Folgen verteilt auf p Prozessen vor. Eine naive Parallelisierung des sequentiellen Verfahrens scheitert, da auch die verteilt vorliegenden Elemente durch die Verwendung des Heaps nur sequentiell von einem Prozess verarbeitet werden können. Abgesehen davon, existieren beispielsweise mit [20] parallele *p-merge*-Verfahren, welche aber eher von theoretischer Bedeutung sind. Gemeinsam ist all diesen Verfahren, dass sie zusätzlich $\mathcal{O}(n)$ Speicher benötigen und somit nicht *in-place* arbeiten.

Eine weitere und wie nachfolgend beschrieben auch praktisch zu realisierende Vorgehensweise besteht darin, den allgemeinen Fall mit p Prozessen auf den handhabbaren Fall mit nur 2 beteiligten Prozessen zurückzuführen. Diese u.a. auch als *Merge-Exchange* bezeichnete Operation wird im nächsten Abschnitt genauer vorgestellt. Aus mehreren, teilweise parallel von verschiedenen Prozesspaaren ausgeführten *Merge-Exchange*-Operationen, kann dann der allgemeine Fall mit $p > 2$ beteiligten Prozessen realisiert werden. Als Vorlage zum Bilden der einzelnen Prozesspaare werden, wie im Abschnitt 4.3 beschrieben, Sortiernetzwerke verwendet.

Diese Art der Vorgehensweise für ein paralleles Sortierverfahren wird bereits in [16] beschrieben. Wichtige praktische Aspekte und Implementierungsdetails werden zusätzlich in [21] und [19] gegeben.

4.1.1 Die Merge-Exchange-Operation

An einer *Merge-Exchange*-Operation sind zwei Prozesse P_0 und P_1 beteiligt, welche jeweils eine sortierte Folge von Elementen der Länge n_0 bzw. n_1 enthalten. Nach Abschluss dieser Operation befinden sich die n_0 kleinsten Elemente sortiert in P_0 und alle restlichen, sprich die n_1 grössten Elemente, sortiert in P_1. Zusätzlich soll im weiteren Verlauf $n_0 + n_1 = n$ gelten.

Eine naive Realisierung dieser Operation könnte wie folgt vorgehen. Zuerst überträgt P_1 alle seine Elemente an P_0. Anschliessend mischt P_0 die zwei sortierten Folgen der Länge n_0 und n_1 in eine sortierte Folge der Länge n zusammen. Mögliche Verfahren für

dieses Zusammenmischen werden in Abschnitt 4.2 ausführlich vorgestellt. Zum Abschluss werden dann die n_1 grössten Elemente von P_0 zurück an P_1 übertragen.

Diese Vorgehensweise besitzt einige entscheidende Nachteile. Zum einen werden unterschiedslos alle Elemente von P_1 nach P_0 übertragen und am Ende wird ein Teil dieser Elemente wieder zurück an P_1 geschickt. Die Elemente von P_1, welche am Ende wieder zu P_1 gehören werden somit 2 mal unnötig versendet. Bei bereits vorsortierten Folgen kann dies zur unnötigen Übertragung grosser Datenmengen führen. Zum anderen wird das Zusammenmischen der sortierten Folgen nur lokal und somit sequentiell von Prozess P_0 durchgeführt, während P_1 in dieser Zeit untätig ist. Des Weiteren wird in P_0 zum Empfangen der Elemente von P_1 zusätzlicher Speicher in Abhängigkeit von n benötigt.

Eine bessere Vorgehensweise wird in [16] als *Bitonic Merge-Exchange* vorgestellt. In Abbildung 4.2 ist die Funktionsweise an einem Beispiel mit $n_0 = n_1 = 8$ dargestellt.

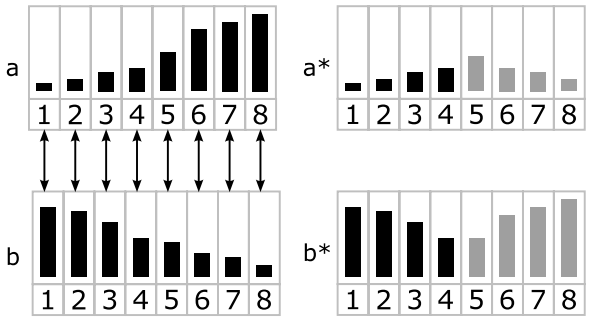

Abbildung 4.2: *Bitonic Merge-Exchange* zwischen einer aufsteigend und einer absteigend sortierten Folge gleicher Länge.

Die Pfeile stellen Vergleiche zwischen zwei Elementen dar, wobei jeweils das kleinere P_0 und das grössere P_1 zugeordnet wird. Die grau markierten Elemente sind die tatsächlich zwischen den Prozessen ausgetauschten Elemente. Durch elementweises Vergleichen der aufsteigend sortierten Folge $[a_i]_{i=1}^{n_0}$ mit der absteigend sortierten Folge $[b_j]_{j=1}^{n_1}$ werden für $n_0 = n_1$ mit:

$$\begin{aligned} a_i^* &= \min(a_i, b_i) \quad (1 \leq i \leq n_0) \\ b_j^* &= \max(a_j, b_j) \quad (1 \leq j \leq n_1) \end{aligned}$$

zwei Folgen $[a_i^*]_{i=1}^{n_0}$ und $[b_j^*]_{j=1}^{n_1}$ erzeugt für die gilt:

$$\max_{1 \leq i \leq n_0} (a_i^*) \leq \min_{1 \leq j \leq n_1} \left(b_j^*\right)$$

Somit befinden sich die n_0 kleinsten Elemente in $[a_i^*]_{i=1}^{n_0}$ und die n_1 grössten in $[b_j^*]_{j=1}^{n_1}$. Sind $[a_i]_{i=1}^{n_0}$ bzw. $[a_i^*]_{i=1}^{n_0}$ dem Prozess P_0 und $[b_j]_{j=1}^{n_1}$ bzw. $[b_j^*]_{j=1}^{n_1}$ dem Prozess P_1 zugeordnet, so kann die *Merge-Exchange*-Operation nun abgeschlossen werden, in dem $[a_i^*]_{i=1}^{n_0}$ und $[b_j^*]_{j=1}^{n_1}$ von P_0 bzw. P_1 sortiert werden.

Die Folgen $[a_i^*]_{i=1}^{n_0}$ und $[b_j^*]_{j=1}^{n_1}$ lassen sich jeweils wieder in zwei sortierte Folgen, eine aufsteigende und eine absteigende, unterteilen. Eine derartige Folge wird u.a auch als bitonische Folge bezeichnet. Durch wiederholtes anwenden dieses Prinzips entsteht das auf Vergleichen basierende bitonische Sortieren.

Im Falle der *Merge-Exchange*-Operation sind sowohl $[a_i^*]_{i=1}^{n_0}$ als auch $[b_j^*]_{j=1}^{n_1}$ zu Beginn aufsteigend sortiert und können darüber hinaus auch von ungleicher Länge sein sein. In Abbildung 4.3 ist daher ein Beispiel für $n_0 = 8$ und $n_1 = 6$ dargestellt.

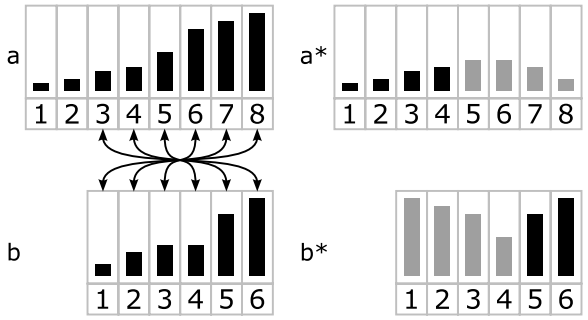

Abbildung 4.3: *Bitonic Merge-Exchange* zwischen zwei aufsteigend sortierten Folgen ungleicher Länge.

Die beiden Folgen $[a_i^*]_{i=1}^{n_0}$ und $[b_j^*]_{j=1}^{n_1}$ werden nun wie folgt erzeugt:

$$a_i^* = \begin{cases} \min(a_i, b_{n_0-i+1}) & i > n_0 - \min(n_0, n_1) \\ a_i & \textit{sonst.} \end{cases} \quad (1 \leq i \leq n_0)$$

$$b_j^* = \begin{cases} \max(a_{n_0-j+1}, b_j) & j \leq \min(n_0, n_1) \\ b_j & \textit{sonst.} \end{cases} \quad (1 \leq j \leq n_1)$$

Insgesamt werden mit dieser Vorgehensweise die x grössten Elemente von P_0 mit den x kleinsten Elementen von P_1 ausgetauscht. Die Anzahl auszutauschender Elemente x wird dabei mit einer Art sequentiellen Suche ermittelt. Gleichzeitig findet ein elementweiser Datenaustausch statt, nach dem die ausgetauschten Elemente in umgekehrter Reihenfolge vorliegen.

In [19] wird mit der Funktion *Find-Exact* ein ähnliches Verfahren vorgestellt. Anstatt der sequentiellen Suche wird jedoch eine binäre Suche verwendet um die Anzahl zu übertragender Elemente x zu bestimmen. Diese Vorgehensweise ist in Abbildung 4.4 an einem Beispiel dargestellt.

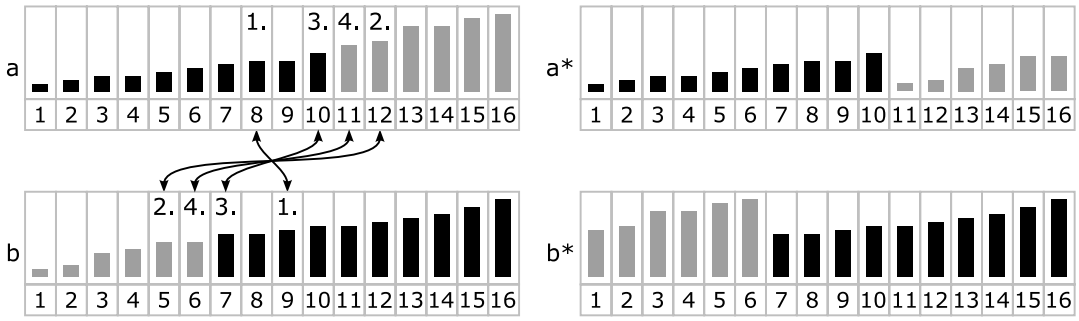

Abbildung 4.4: Funktionsweise von *Find-Exact* mit Datenaustausch.

Zu Beginn werden jeweils die kompletten Folgen als zu durchsuchende Intervalle festgelegt. Dann wird das mittlere Element a_i aus dem Intervall von Prozess P_0 mit dem mittleren Element b_{n_0-i+1} aus dem Intervall von Prozess P_1 verglichen. Abhängig davon welches Element kleiner ist, wird bei einem Prozess die obere und bei dem anderen die unter Hälfte als neues Intervall festgelegt. Dies wird wiederholt bis das zu durchsuchende Intervall leer ist. Im angegebenen Beispiel wird so nacheinander in 4 Schritten zuerst a_8 mit b_9, dann a_{12} mit b_5, usw. verglichen. Dabei wird das zu betrachtende Intervall immer weiter eingeschränkt. Anhand des zu letzt durchsuchten Intervalls können dann die auszutauschenden Elemente, im Beispiel grau markiert, bestimmt werden.

Damit lässt sich die *Merge-Exchange*-Operation nun in 3 Schritten durchführen:

1. Mittels binärer Suche wird die Anzahl auszutauschender Elemente bestimmt. Dieser Schritt besitzt eine Laufzeitkomplexität von $\mathcal{O}(\log n)$. Es wird nur $\mathcal{O}(1)$ zusätzlicher Speicher benötigt, da immer nur einer Element ausgetauscht und verglichen wird.

2. Die zuvor ermittelten Elemente werden mit der *in-place* Kommunikationsoperation `MPI_Sendrecv_replace` ausgetauscht. Diese verwendet den gleichen Speicherbereich zum Senden und Empfangen und benötigt somit keinen zusätzlichen Speicher. Da die maximal zu übertragende Anzahl Elemente durch n begrenzt wird, besitzt dieser Schritt eine Laufzeitkomplexität von $\mathcal{O}(n)$.

3. Abschliessend bringt jeder Prozess seine Elemente wieder in eine aufsteigend sortierte Reihenfolge. Dies kann beispielsweise mit einem sequentiellen Sortierverfahren realisiert werden. Effizienter ist es jedoch auszunutzen, dass die Elemente jedes Prozesses aus zwei bereits aufsteigend sortierten Folgen bestehen (siehe Abbildung 4.4). In Abschnitt 4.2 werden die dafür benötigen *2-merge*-Verfahren vorgestellt.

4.2 Das sequentielle 2-merge-Problem

Das 2-*merge* Problem stellt einen ausführlich untersuchten Spezialfall des *p-merge* Problems dar. In seiner sequentiellen Form sind beide Ausgangsfolgen einem einzelnen Prozess zugeordnet. Damit tritt es beispielsweise zum Abschluss der *Merge-Exchange*-Operation aus Abschnitt 4.1.1 auf.

Im weiteren Verlauf wird eine Folge $[a_i]_{i=1}^{n}$ der Länge n betrachtet. Diese lässt sich zu Beginn mit $[a_i]_{i=1}^{n_0}$ und $[a_i]_{i=n_0+1}^{n}$ in zwei aufsteigend sortierte Teilfolgen der Länge n_0 bzw. n_1 aufteilen. Ziel ist es diese derart zusammenzumischen, dass die n Elemente eine aufsteigend sortierte Folge bilden. Zur Vereinfachung soll für die Vorstellung der einzelnen Verfahren $n_0 \leq n_1$ gelten. Für konkrete Implementierungen einzelner Verfahren kann es u.a. nötigt sein, die beiden Fälle $n_0 \leq n_1$ und $n_0 > n_1$ getrennt zu behandeln.

Im folgenden werden zuerst Verfahren vorgestellt, welche einfach zu realisieren sind und $\mathcal{O}(n_0)$ zusätzlichen Speicher benötigen. Anschliessend werden Verfahren vorgestellt, welche die Funktionsweisen der einfachen Verfahren nutzen, dabei jedoch weniger zusätzlichen Speicher verwenden.

4.2.1 Einfache 2-merge-Verfahren

Das rein intuitiv naheliegenste Verfahren wird in [6] als *Two-way merge* vorgestellt und kann wie in Listing 4.1 beschrieben realisiert werden. Die zwei sortierten Folgen sind zu Beginn in einem Feld an den Positionen `A[0..n0-1]` bzw. `A[n0..n0+n1-1]` abgelegt. Das Feld `T` dient als temporärer Speicher und muss mindestens n_0 Elemente aufnehmen können.

Listing 4.1: *Two-way merge*

```
1   i = j = 0;
2   k = n0;

4   ncopy(A, T, n0);

6   while (j < n0 && k < n1)
7   {
8     if (T[j] <= A[k]) { A[i] = T[j]; j++; }
9     else { A[i] = A[k]; k++; }

11    i++;
12  }

14  ncopy(&T[j], &A[i], n0 - j);
```

Zuerst werden die n_0 Elemente der ersten Folge in das Hilfsfeld `T` kopiert um am Anfang von `A` Platz für die Ausgabe der sortierten Folge zu schaffen. Dann werden in einer Schleife die jeweils kleinsten Elemente beider Folgen miteinander verglichen und das kleinere von beiden zur Ausgabe nach `A` kopiert. Dies wird so lang wiederholt bis eine

der beiden Folgen leer ist. In Abbildung 4.5 wird die Position der Indizes und der Zustand der Folgen in einem Zwischenschritt an einem Beispiel dargestellt.

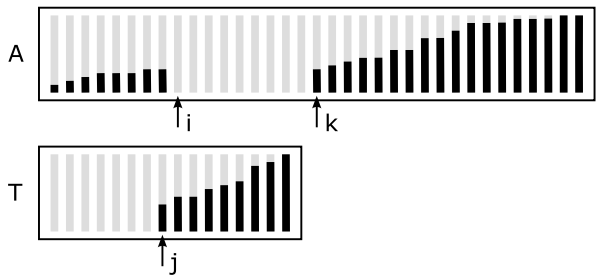

Abbildung 4.5: Zustand der Indizes in einem Zwischenschritt von *Two-way merge*.

Sind zum Abschluss noch Elemente der ersten Folge in `T` vorhanden, werden diese ebenfalls zur Ausgabe nach `A` kopiert. Die Schleife wird höchstens $n_0 + n_1$ mal durchlaufen, wobei in jedem Durchlauf eine Vergleichs- und eine Kopieroperation durchgeführt wird. Vor und nach der Schleife werden jeweils höchstens n_0 Elemente kopiert. Somit besitzt *Two-way merge* eine Laufzeitkomplexität von $\mathcal{O}(n_0 + n_1)$.

Ist die erste Folge im Vergleich zur zweiten relativ klein ($n_0 << n_1$) bietet es sich an, die wenigen Elemente der ersten Folge einzeln in die zweite einzufügen. Die Position zum Einfügen in der sortierten zweiten Folge kann mittels binärer Suche bestimmt werden. Diese Vorgehensweise kann wie in Listing 4.2 beschrieben realisiert werden.

Listing 4.2: Mischen mit binärer Suche

```
1   i = j = 0;
2   k = n0;

4   ncopy(A, T, n0);

6   while (j < n0 && k < n1)
7   {
8     x = binary_search_lt(T[j], &A[k], n1 - k);

10    ncopy(&A[k], &A[i], x);
11    i += x;  k += x;

13    A[i] = T[k];
14    i++; j++;
15  }

17  ncopy(&T[j], &A[i], n0 - j);
```

Die Unterschiede zu *Two-way merge* befinden sich lediglich innerhalb der Schleife. Mittels binärer Suche wird nun jeweils die Anzahl x der noch verbleibenden Elemente der zweiten

Folge bestimmt, die kleiner sind als das aktuell kleinste Element der ersten Folge. Diese x Elemente der zweiten Folge werden dann gefolgt vom kleinsten Element der ersten Folge, zur Ausgabe nach `A` kopiert. Die Schleife wird nun höchstens n_0 mal durchlaufen, wobei in jedem Durchlauf eine binäre Suche in maximal n_1 Elementen erfolgt. Die maximale Anzahl durchzuführender Vergleichsoperationen ist somit durch $n_0 \log n_1$ beschränkt. Darüber hinaus sind auch weiterhin $n_0 + n_1$ Kopieroperationen nötig. Es ergibt sich daher eine Laufzeitkomplexität von $\mathcal{O}(n_0 \log n_1 + n_0 + n_1)$. Wie im folgenden gezeigt, kann $n_0 \log n_1$ im Falle $n_0 < \sqrt{n_0 + n_1}$ durch $n_0 + n_1$ abgeschätzt werden.

$$\begin{aligned} n_0 < \sqrt{n_0 + n_1} &\Rightarrow n_0 < \tfrac{1}{2} + \sqrt{\tfrac{1}{4} + n_1} = z \\ n_0 \log_2 n_1 < n_0 + n_1 &\Rightarrow \left(\tfrac{n_1}{2}\right)^{n_0} < 2^{n_1} \\ \left(\tfrac{n_1}{2}\right)^{z} < 2^{n_1} &\Rightarrow 2 > n_1^{\frac{z}{n_1+z}} \\ &\Rightarrow 2 > \sup n_1^{\frac{z}{n_1+z}} \approx 1.86 \quad \text{w.A.} \end{aligned}$$

Unter der Bedingung $n_0 < \sqrt{n_0 + n_1}$ führt die Verwendung der binären Suche also ebenfalls zu einem linear skalierenden Verfahren, welches jedoch weniger Vergleichsoperationen benötigt als *Two-way merge*.

Wie in [6] beschrieben, ist *Two-way merge* für $n_0 = n_1$ und die binäre Suche für $n_0 = 1$ optimal bezüglich der Anzahl durchzuführender Vergleichsoperationen. Mit *Binary merge* von Hwang und Lin [25] existiert darüber hinaus ein Verfahren, welches im allgemeinen weniger Vergleichsoperationen benötigt. Es führt zuerst eine grobe sequentielle Suche, gefolgt von einer binären Suche aus. Die Anzahl der durchzuführenden Vergleichsoperationen beträgt höchstens $n_0(t+1) + \lfloor n_1/2^t \rfloor$ mit $t = \lfloor \log_2(n_1/n_0) \rfloor$. Damit verhält es sich für $n_0 = n_1$ wie *Two-way merge* und für $n_0 = 1$ wie die binären Suche. Die Vergleichsoperationen haben jedoch nur geringen Einfluss auf die Laufzeit der Verfahren, da diese hauptsächlich von den Kopieroperationen bestimmt werden. Darüber hinaus hat sich eine Kombination aus *Two-way merge* und binärer Suche als effizienter erwiesen.

Die bisher vorgestellten Verfahren benötigen alle zusätzlichen Speicher in Abhängigkeit von n_0. In den Listings ist dieser durch das Feld `T` gegeben, wobei dessen ursprünglicher Inhalt überschrieben wird. Führt man statt der Kopieroperationen jeweils einen Austausch zwischen den Elementen von `A` und `T` durch, so wird der ursprüngliche Inhalt von `T` nicht überschrieben sondern nur umgeordnet. Da ein solcher Austausche mit 3 Kopieroperationen realisiert werden kann, hat dies keine Auswirkungen auf die Laufzeitkomplexität. Derartige Verfahren, die den Inhalt eines Hilfsfeldes nicht überschreiben werden später in Abschnitt 4.2.2 für die Realisierung eines *in-place*-Verfahrens verwendet.

4.2.2 Zusammengesetzte 2-merge-Verfahren

Die bisher vorgestellten Verfahren benötigen in den meisten Fällen zusätzlichen Speicher in Abhängigkeit von n_0. Für das parallele Sortierverfahren der FMM soll es jedoch auch möglich sein, nur konstanten zusätzlichen Speicher zu verwenden. Zu diesem Zweck werden im folgenden zwei weitere Verfahren mit Funktionsweise und Eigenschaften kurz

vorgestellt. Eine detailliertere Beschreibung ist jeweils in den angegebenen Arbeiten zu finden.

Von Tridgell und Brent wird in [19] ein Verfahren vorgestellt welches nur $\mathcal{O}(\sqrt{n_0 + n_1})$ zusätzlichen Speicher benötigt. Dazu werden die beiden Folgen jeweils in Blöcke der Grösse $s = \sqrt{n_0 + n_1}$ eingeteilt. Zusätzlich werden noch 3 weitere Blöcke der Grösse s verwendet, welche zu Beginn leer sind. Dann wird ähnliche wie bei *Two-way merge* begonnen, die beiden Folgen zusammenzumischen. Die sortierten Elemente werden dabei in den leeren Blöcken gespeichert. Sobald zwei der anfangs leeren Blöcke mit sortierten Elementen gefüllt sind, besitzt mindestens eine der beiden Folgen an ihrem Anfang einen neuen leeren Block. Dieser wird dann im weiteren Verlauf ebenfalls für die Aufnahme von sortierten Elementen verwendet. Sind alle Elemente beider Folgen zusammengemischt, liegt die sortierte Folge blockweise vor. Abschliessenden kann dann durch Umordnung der Blöcke die sortierte Folge erzeugt werden. Mit einer Blockgrösse von $s = \sqrt{n_0 + n_1}$ existieren insgesamt nur maximal $s + c$ (c konstant) Blöcke. Somit ergibt sich mit den zusätzlichen leeren Blöcken und einer Liste zur Verwaltung der Blöcke eine Speicherkomplexität von $\mathcal{O}(\sqrt{n_0 + n_1})$. Da jedes Element nur einmal beim Zusammenmischen und einmal bei der Umordnung der Blöcke kopiert wird, besitzt dieses Verfahren eine Laufzeitkomplexität von $\mathcal{O}(n_0 + n_1)$.

Das erste linear skalierende Verfahren, welches nur eine konstante Menge zusätzlichen Speicher benötigt stammt von Kronrod [26]. Es enthält die, besonders für die lineare Laufzeitkomplexität entscheidenden Konzepte des *internen Puffers* und der *Blocksortierung*. Daraufhin sind viele, meist nach einem ähnlichen Prinzip funktionierende Varianten entwickelt worden. Im Rahmen dieser Arbeit wurden einige von ihnen näher betrachtet [27, 28, 29]. Die Funktionsweise der einzelnen Verfahren ist dabei meist leicht nachzuvollziehen. Eine konkrete Implementierung ist jedoch, zum Teil durch eine Vielzahl zu beachtender Sonderfälle, immer mit sehr hohem Aufwand verbunden. Im Zuge dieser Arbeit wurde das Verfahren von Huang und Langston [28] implementiert, da es einerseits keine Unterscheidung zwischen $n_0 \leq n_1$ und $n_0 > n_1$ macht und zum anderen als einziges untersuchtes Verfahren auch Aussagen über sein praktisches Laufzeitverhalten gibt.

Das Verfahren von Huang und Langston funktioniert wie folgt. Zuerst werden die beiden Folgen in Blöcke der Grösse s eingeteilt. Dabei wird sichergestellt, dass die insgesamt s grössten Elemente in einem einzelnen Block vorhanden sind. Dieser Block ist der sog. interne Puffer und wird zu Beginn an den Anfang verschoben. Dann werden alle restlichen Blöcke nach ihren jeweils letzten Elementen aufsteigend sortiert. Anschliessend erfolgt das Zusammenmischen jeweils zweier hintereinander liegender Blöcke mit einem einfachen Verfahren wie *Two-way merge*. Als Hilfsfeld wird der interne Puffer verwendet, wobei dessen Elemente nicht überschrieben, sondern nur wie in Abschnitt 4.2.1 erklärt, vertauscht werden. Infolgedessen wandert der interne Puffer bis ans Ende des Feldes, wobei die sortierte Reihenfolge seiner Elemente verloren geht. Zum Abschluss werden daher die s Elemente des internen Puffers noch einmal sortiert. Wird als Blockgrösse $s = \sqrt{n_0 + n_1}$ gewählt, kann sowohl das Sortieren der s einzelnen Blöcke zu Beginn als auch das abschliessende Sortieren des internen Puffers mit einem quadratisch skalieren-

den Verfahren durchgeführt werden. Insgesamt ergibt sich dadurch eine Laufzeitkomplexität von $\mathcal{O}(n_0 + n_1)$.

4.2.3 Vergleich

In Tabelle 4.2.3 sind die im vorherigen Abschnitt vorgestellten *2-merge*-Verfahren noch einmal zusammen mit ihren jeweiligen Eigenschaften aufgeführt.

	Laufzeitkomplexität	Speicherkomplexität
Two-way merge	$\mathcal{O}(n_0 + n_1)$	$\mathcal{O}(n_0)$
binäre Suche	$\mathcal{O}(n_0 \log n_1 + n_0 + n_1)$	$\mathcal{O}(n_0)$
Tridgell & Brent [19]	$\mathcal{O}(n_0 + n_1)$	$\mathcal{O}(\sqrt{n_0 + n_1})$
Huang & Langston [28]	$\mathcal{O}(n_0 + n_1)$	$\mathcal{O}(1)$

Tabelle 4.1: Zusammenfassung der Eigenschaften der *2-merge*-Verfahren.

In Abbildung 4.6 sind Laufzeiten für die Verfahren aus Tabelle 4.2.3 und CS-Radixsort abgebildet. Diese wurden in Abhängigkeit vom Verhältnis zwischen n_0 und n_1 an einem Beispiel mit jeweils insgesamt 2^{20} Elementen gemessen.

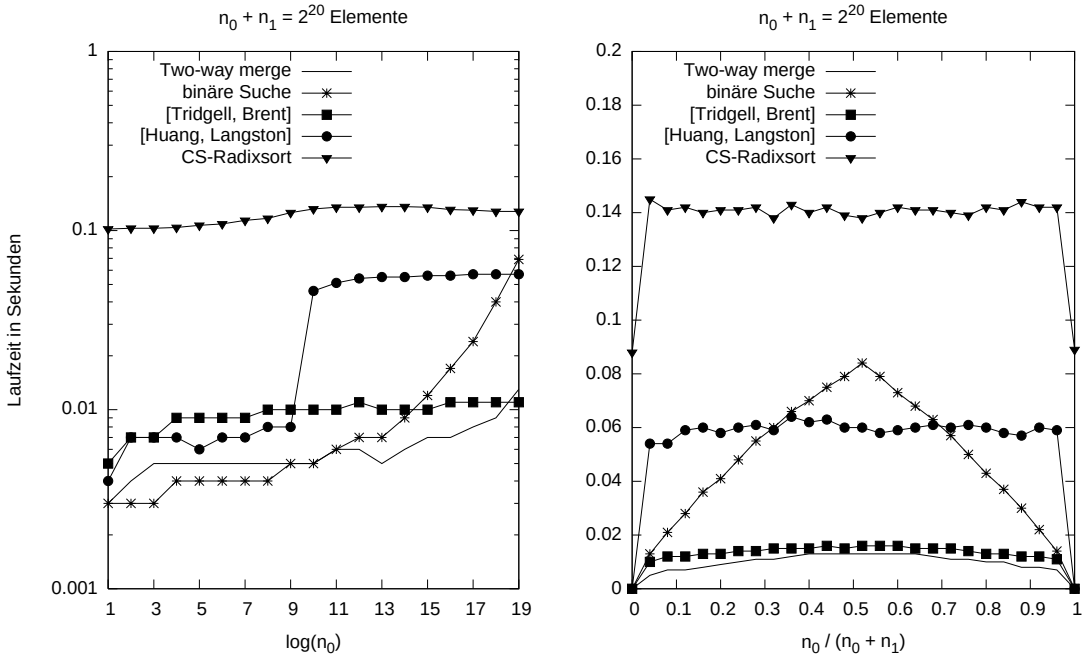

Abbildung 4.6: Laufzeiten der *2-merge*-Verfahren.

Anhand der Laufzeiten ist klar zu erkennen, dass von den Verfahren mit einer Speicherkomplexität von $\mathcal{O}(n)$ *Two-way merge* in den meisten Fällen am schnellsten ist. Lediglich

wenn eine der Folgen sehr kurz ist, verhält sich die binäre Suche etwas schneller. Nur leicht langsamer als *Two-way merge* ist das Verfahren von Tridgell und Brent, welches dafür jedoch schon deutlich weniger zusätzlichen Speicher benötigt. Wiederum mindestens 3 mal langsamer ist dagegen das *in-place* Verfahren von Huang und Langston. Die Verwendung von CS-Radixsort stellt sich als deutlich langsamer als alle implementierten *2-merge*-Verfahren dar und ist daher keine Alternative. Damit ist gezeigt, dass sich geringere Speicheranforderungen deutlich auf die Laufzeit auswirken können.

Zwar besitzen *Two-way merge* und das Verfahren mit binärer Suche eine Speicherkomplexität von $\mathcal{O}(n)$, aber sie benötigt dennoch nur zusätzlichen Speicher für jeweils $\min(n_0, n_1)$ Elemente. Für parallele Sortierverfahren können sie beispielsweise in Kombination mit den anderen Verfahren automatisch immer dann eingesetzt werden, wenn ein gegebener zusätzlicher Speicher noch ausreicht. Dadurch würde sich ein Verfahren ergeben, welches den jeweils verfügbaren Speicher effizient nutzt.

4.3 Sortiernetzwerke

Wie bereits erwähnt werden Sortiernetzwerke [6] als Vorlage verwendet um mit mehreren einzelnen *Merge-Exchange*-Operationen die $p > 2$ sortierten Folgen von Elementen zu einer sortierten Folge zusammenzumischen.

Für eine beliebige Folge von p Elementen beschreibt ein Sortiernetzwerk ein Schema anhand dessen in mehreren Schritten jeweils zwei Elemente miteinander verglichen und gegebenenfalls vertauscht werden. Unabhängig von den konkreten Werten der einzelnen Elemente ist nach der Ausführung dieser Vergleichsschritte sichergestellt, das die Folge sortiert ist. In Abbildung 4.7 ist ein einfaches Sortiernetzwerk für $p = 5$ Elemente als Knuth-Diagramm dargestellt.

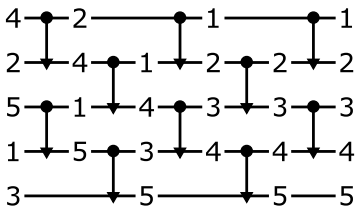

Abbildung 4.7: Beispiel eines Sortiernetzwerks für $p = 5$ Elemente.

Von links nach rechts werden jeweils zwei mit einem Pfeil verbundene Elemente miteinander verglichen. Die Richtung eines Pfeils gibt die Position vor, an die das grössere der beiden Elemente zu verschieben ist. Abhängig von den konkreten Werten zweier Elemente müssen diese gegebenenfalls miteinander vertauscht werden. Meist können in einem Schritt parallel, mehrere Elemente miteinander verglichen werden.

Um nun auf p Prozessen verteilt vorliegende sortierte Folgen von Elementen zusammenzumischen, wird ein Sortiernetzwerk für p Elemente verwendet. Jedes Element steht dabei für einen Prozess und anstatt paarweiser Vergleiche von einzelnen Elementen führen nun jeweils 2 Prozesse eine *Merge-Exchange*-Operation miteinander aus.

Es existieren mehrere verschiedene Sortiernetzwerke mit teils unterschiedlichen Eigenschaften. Sie besitzen eine regelmässige Struktur und sind meist als Konstruktionsvorschrift in Abhängigkeit von p gegeben. In Abbildung 4.8 werden für $p = 8$ vier verschiedene Sortiernetzwerke vorgestellt, welche in verschiedenen parallelen Sortierverfahren verwendet werden.

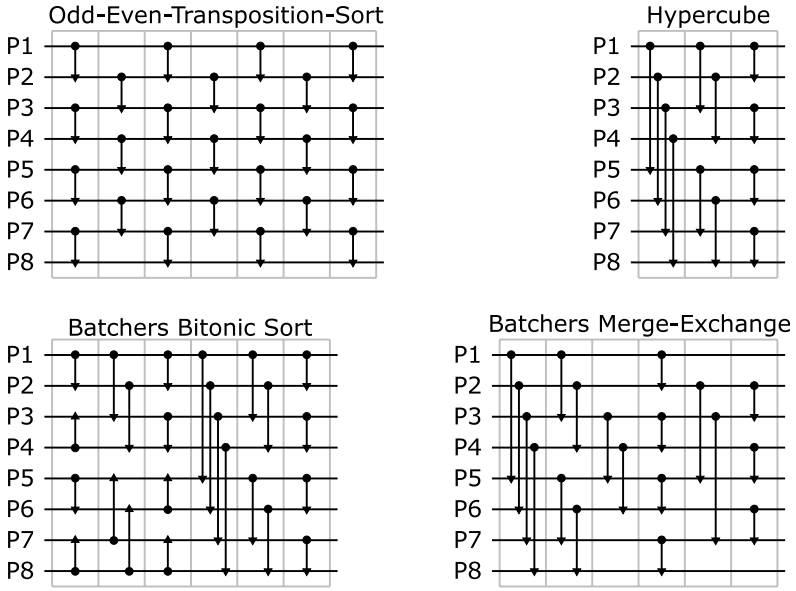

Abbildung 4.8: Verschiedene Sortiernetzwerke für $p = 8$ Prozesse.

Odd-Even-Transposition-Sort benötigt $\mathcal{O}(p)$ Schritte und wird in [17] für das parallel Sortierverfahren *Merge-Splitting Sort* verwendet. *Batchers Bitonic Sort* mit $\mathcal{O}((\log p)^2)$ Schritten kommt in [16] für *Block Bitonic Sort* zur Anwendung. *Hypercube* benötigt nur $\mathcal{O}(\log p)$ Schritte, wobei jedoch nicht sichergestellt ist, dass damit immer vollständig sortiert wird. In [21] und [19] wird es daher in einer ersten Phase verwendet um vorsortierte Folgen zu erzeugen. In einer abschliessenden *Cleanup*-Phase wird dann mit *Batchers Merge-Exchange* in $\mathcal{O}((\log p)^2)$ Schritten eine vollständige Sortierung erreicht.

Batchers Merge-Exchange eignet sich dabei besonders gut für bereits vorsortierte Folgen. Zum einen benötigt es gegenüber *Odd-Even-Transposition-Sort* weniger Schritte. Zum anderen werden die *Merge-Exchange*-Operationen immer nur in absteigender Richtung ($\downarrow$) durchgeführt. Bei vorsortierten Folgen müssen in diesem Fall nur wenige Elemente ausgetauscht werden, wodurch auch die *Merge-Exchange*-Operationen entsprechend effizienter werden. Im Gegensatz dazu wird mit *Batchers Bitonic Sort* eine bereits vorhandene Vorsortierung im Verlauf der einzelnen Schritte teilweise wieder zerstört.

Zusammen mit einem linear skalierenden sequentiellen Sortierverfahren ergibt sich damit beispielsweise für die Kombination aus *Hypercube* und *Batchers Merge-Exchange* ein paralleles Sortierverfahren mit einer Laufzeitkomplexität von $\mathcal{O}(\frac{n}{p}(\log p)^2)$. Die Speicherkomplexität kann dabei je nach verwendetem *2-merge*-Verfahren unterschiedlich ausfallen.

4.4 Laufzeitverhalten

Die in diesem Kapitel vorgestellten Verfahren wurden implementiert und zu einem parallelen Sortierverfahren zusammengesetzt. In Abbildung 4.9 sind Speedups für das Sortieren von 2^{24} Elementen mit Schlüsselwerten im Bereich $0 \ldots 2^{30} - 1$ angegeben. Als sequentielles Sortierverfahren wurde CS-Radixsort und als Sortiernetzwerk eine Kombination aus *Hypercube* und *Batchers Merge-Exchange* verwendet. Durch die Nutzung der verschiedenen *2-merge*-Verfahren ergibt sich die jeweils entsprechende Speicherkomplexität.

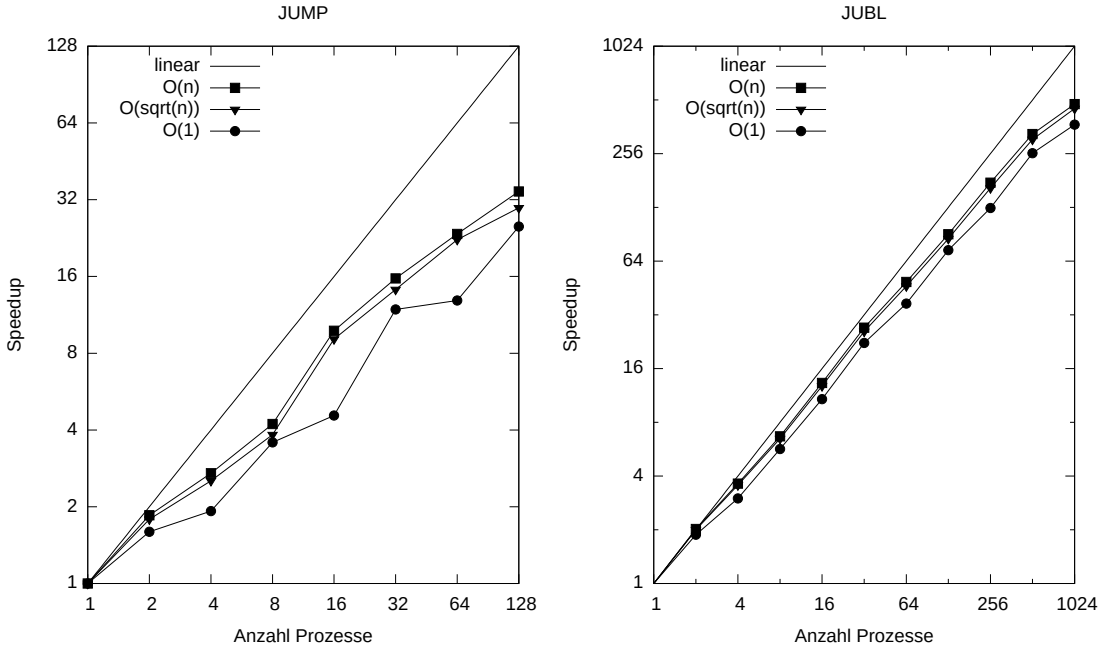

Abbildung 4.9: Speedups für das Sortieren von 2^{20} Elementen mit unterschiedlicher Speicherkomplexität mit den Parallelrechnern JUMP und JUBL.

Wie bereits in Abschnitt 4.2.3 vermutet zeigt sich, dass durch die unterschiedlichen Speicheranforderungen der *2-merge*-Verfahren die Speedups der parallelen Sortierverfahren teilweise stark beeinflusst werden. Die Unterschiede zwischen der Verwendung von $\mathcal{O}(n)$ und $\mathcal{O}(\sqrt{n})$ zusätzlichem Speicher sind dabei aber nur gering. Mit einem Knoten des Parallelrechners JUMP ergibt bei 32 Prozessen eine Effizienz von ca. 37-50%. Werden mehrere Knoten mit insgesamt bis zu 128 Prozessen verwendet, sinkt die Effizienz auf ca. 20-26%. Deutliche bessere Werte ergeben sich mit dem Blue Gene/L System. Zwar sinkt auch hier die Effizienz mit zunehmender Anzahl Prozesse, doch auch mit 512 Prozessen ergibt sich immer noch eine Effizienz von ca. 51-64%. Die exklusive Nutzung der Ressourcen des Blue Gene/L Systems führt dabei auch zu deutlich regelmässigeren Ergebnissen.

In Abbildung 4.10 sind Laufzeiten für das Sortieren in Abhängigkeit von der Anzahl der Elemente für das Blue Gene/L System dargestellt. Als *2-merge*-Verfahren wurde nun die *in-place* Methode von Huang und Langston verwendet. Alle restlichen Einstellungen sind wie im vorherigen Beispiel gewählt wurden.

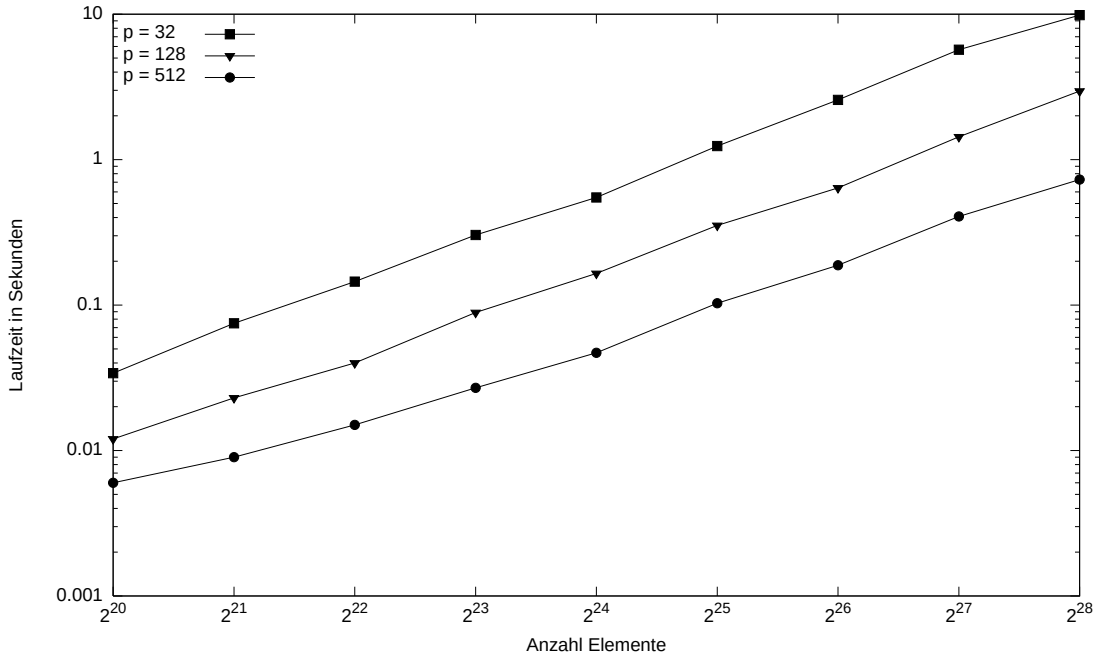

Abbildung 4.10: Laufzeiten für das parallel Sortieren mit einem *in-place* Verfahren basierend auf Mischen mit den Parallelrechner JUBL.

Es lässt sich dabei gut erkennen, dass das implementierte parallele Sortierverfahren linear bezüglich der Anzahl der zu sortierenden Elemente skaliert.

4.5 Anwendung in der FMM

Das in diesem Kapitel vorgestellte parallel Sortierverfahren wird verwendet, um in der parallelen Implementierung der FMM die Eingabedaten, wie in Abschnitt 2.1.1 beschrieben, zu sortieren. Dies schliesst auch mehrere Sortierschritte ein, in denen bereits eine Vorsortierung vorhanden ist.

Für die parallele FMM wird dabei CS-Radixsort als sequentielles Sortierverfahren verwendet. Darüber hinaus hat sich als Sortiernetzwerk *Batchers Merge-Exchange* als optimal herausgestellt. Für das *2-merge*-Verfahren bleibt es weiterhin möglich, je nach verfügbarem zusätzlichen Speicher, ein entsprechendes Verfahren auszuwählen. In Abbildung 4.11 sind Speedups für das parallele Sortierverfahren abgebildet, wie sie sich bei einer Beispielrechnung der FMM mit einem System bestehend aus 2^{24} Teilchen ergeben. Durch den begrenzten Hauptspeicher von 512 MB pro Rechenknoten des Blue Gene/L Systems konnten die Beispielrechnung mit den Daten der FMM erst ab 2 Prozessen durchgeführt werden, wobei deren Ergebnisse für die Berechnung der Speedups verwendet wurden.

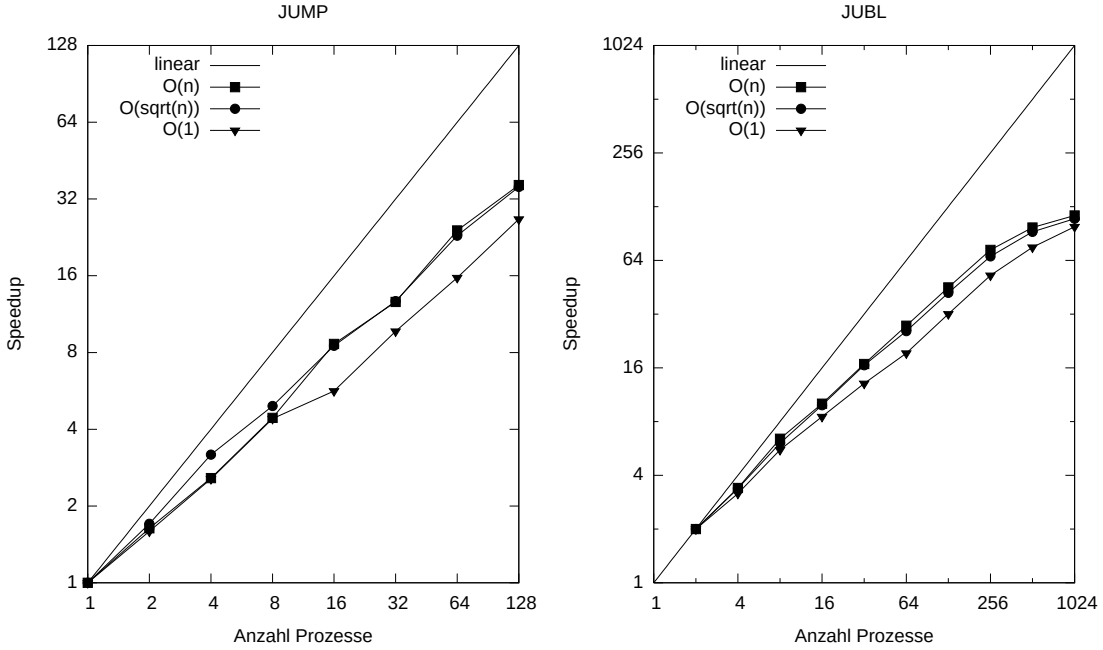

Abbildung 4.11: Speedups für das Sortieren der Eingabedaten der FMM mit jeweils unterschiedlicher Speicherkomplexität mit den Parallelrechnern JUMP und JUBL.

Mit den Beispieldaten der FMM ergeben sich für den Parallelrechner JUMP ähnliche Werte wie in Abschnitt 4.4. Unter Verwendung von 128 Prozessen liegt die Effizienz bei ca. 21-28%. Weiterhin zeigt sich deutlich, dass das Blue Gene/L System nun mit

steigender Anzahl Prozesse schlechter abschneidet als in Abschnitt 4.4. Die Effizienz sinkt bereits bei 512 Prozessen auf deutlich unter 20%.

5 Paralleles Sortieren basierend auf Partitionierung

Die zweite Kategorie der parallelen Sortierverfahren basiert auf Partitionierung. Wie in Abbildung 5.1 dargestellt kann die Vorgehensweise grob in zwei Schritte eingeteilt werden.

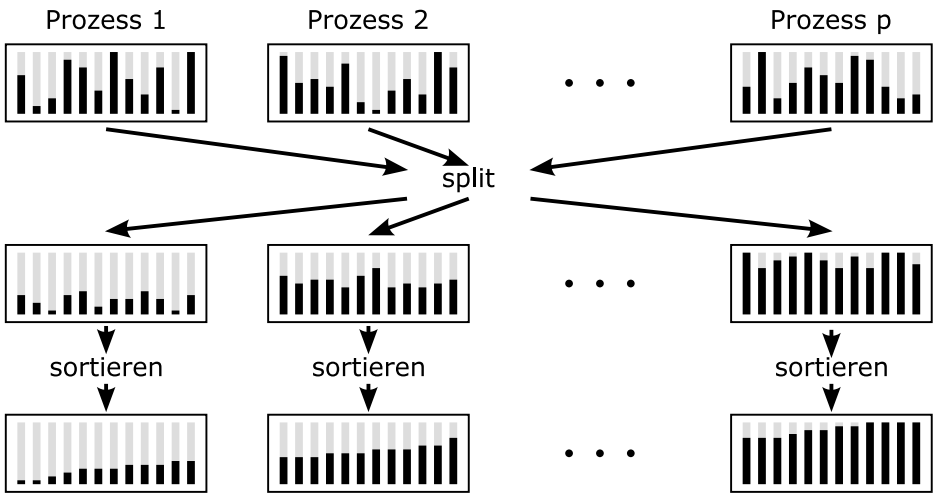

Abbildung 5.1: Paralleles Sortieren basierend auf Partitionierung.

Zu Beginn besitzt jeder Prozess seine unsortierten Elemente. Eine ungleichmässige Verteilung der Elemente ist im allgemeinen möglich. Sie kann jedoch eine ungleichmässige Lastverteilung zur Folge haben bzw. spezielle Behandlung durch einzelne Verfahren erfordern. In einem ersten Schritt werden mit einer parallelen sog. *Split*-Operation alle Elemente unter den p Prozessen umverteilt. Mit dieser Aufteilung wird erreicht, dass alle Elemente eines Prozesses j grösser als die Elemente von Prozess $j-1$ und kleiner als die Elemente von Prozess $j+1$ sind. Anschliessend sortiert jeder Prozess unabhängig die ihm zugeteilten Elemente mit einem sequentiellen Sortierverfahren.

In den folgenden Abschnitten wird die Realisierung der *Split*-Operation näher erläutert. Zu diesem Zweck muss zuerst eine Partitionierungsfunktion konstruiert werden, die jedes Element einem neuen Prozess zuordnet. Anhand dieser kann dann die eigentliche Umverteilung der Elemente durchgeführt werden. Für ein allgemeines paralleles Sortier-

verfahren ist die Konstruktion der Partitionierungsfunktion mit zusätzlichem Aufwand verbunden.

Das im folgenden vorgestellte Verfahren kann für die Sortierung der Ausgabedaten der FMM verwendet werden, da sich in diesem Fall die Partitionierungsfunktion mit sehr geringem Aufwand konstruieren lässt. Die anschliessende Umverteilung der Elemente muss dann wieder *in-place* erfolgen. Zu diesem Zwecke werden in Abschnitt 5.2 zwei Vorgehensweisen beschrieben, die sich auch mit Standard-MPI-Funktionen realisieren lassen.

5.1 Die Partitionierungsfunktion

Die Aufteilung der Elemente in p Partitionen wird durch $p-1$ aufsteigend sortierte sog. *Splitter*-Elemente $[s_i]_{i=1}^{p-1}$ vorgegeben. Durch die nachfolgende Funktion kann damit jedes Element x einer Partition $f(x)$ zugeordnet werden.

$$f(x) = \begin{cases} 1 & x < s_0 \\ i & s_{i-1} \leq x < s_i \quad (1 < i < p) \\ p & x \geq s_{p-1} \end{cases}$$

Je nach Vorgehensweise bei der Auswahl der *Splitter*-Elemente können zwei verschiedene Partitionierungsstrategien unterschieden werden.

Eine probabilistische Partitionierung verwendet eine Auswahl von Elementen, sog. *Samples*, als repräsentative Stellvertreter aller Elemente. Anhand der ausgewählten *Samples* werden dann die *Splitter*-Elemente bestimmt. Derartige Sortierverfahren werden zumeist unter dem Begriff *Samplesort* zusammengefasst, wobei die Auswahl der Samples sowohl zufällig [22] als auch deterministisch [23] erfolgen kann.

In den meisten Fällen wird mit diesen Verfahren eine annähernd gleichmässige Partitionierung erreicht. Eine exakte bzw. bis auf einen konstanten Teil gleichmässige Partitionierung kann jedoch nicht garantiert werden. Für ein *in-place* Sortierverfahren ist diese Vorgehensweise daher eher ungeeignet bzw. erfordert zusätzlichen Aufwand zur Behandlung von ungleichmässigen Partitionierungen.

Mit einer exakten Partitionierung ist garantiert, dass jeder Partition eine vorher genau festgelegte Anzahl Elemente zugeordnet wird. Da hierfür, im Gegensatz zu *Samplesort*-Verfahren, immer alle Elemente berücksichtigt werden müssen, ist dies entsprechend aufwändiger. So müssen beispielsweise für *Parallel Sorting by Exact Splitting* [24] zuerst alle Elemente lokal sortiert werden. Diese Sortierung bleibt dann bei der Umverteilung der Elemente erhalten, wodurch die abschliessende Sortierung durch ein sequentielles *p-merge*-Verfahren erfolgen kann. Eine weitere Möglichkeit zur Ermittlung von exakten *Splitter*-Elementen, ist die Verwendung von parallelen Implementierungen von *Order-Statistics*-Verfahren [30].

Besitzen mehrere Elemente die gleichen Schlüsselwerte so ist es im allgemeinen für eine exakte Partitionierung notwendig, Elemente mit identischen Schlüsselwerten auf unterschiedliche Partitionen verteilen zu können. Sind beispielsweise nur k verschiedene Schlüsselwerte vorhanden, so können auch nur maximal k unterschiedliche *Splitter*-Elemente ausgewählt werden. Auch in diesem Fall muss jedoch eine Partitionierung in $p > k$ Partitionen möglich sein.

Das parallele Sortieren der Ausgabedaten der FMM stellt einen Sonderfall dar, in dem die *Splitter*-Elemente für eine exakte Partitionierung, ohne vorherige Auswahl von Samples oder ähnlichem, direkt bestimmt werden können. Wie in Abschnitt 2.1.2 vorgestellt, müssen die Ausgabedaten anhand der Adresswerte sortiert werden. Diese wurden zu Beginn mittels Durchnummerierung aller n Teilchen bestimmt. Die zu sortierenden Elemente besitzen daher Schlüsselwerte zwischen 1 und n, wobei jeder Wert genau einmal

vergeben ist. Sollen diese nun in p Partitionen mit den Grössen n_i $(1 \leq i \leq p)$ aufgeteilt werden, können die $p-1$ *Splitter*-Elemente direkt mit

$$s_i = 1 + \sum_{j=1}^{i} n_j \quad (1 \leq i < p)$$

bestimmt werden. Die Grössen der Partitionen werden dabei durch die Anzahl der Elemente vorgegeben, die jeder Prozess besitzt. Mit einer kollektiven Kommunikationsoperation (`MPI_Allgather`) werden daher zuerst die Grössen unter allen Prozessen verteilt und anschliessend die entsprechenden *Splitter*-Elemente von jedem Prozess selbständig bestimmt.

5.2 Umverteilung der Elemente

Vor der Umverteilung besitzt jeder Prozess P_i $(1 \leq i \leq p)$ genau n_i unsortierte Elemente, die in einem Feld abgelegt sind. Des Weiteren ist eine Partitionierungsfunktion f gegeben, die jedes Element x_j $(1 \leq j \leq n)$ einem neuen Prozess $f(x_j)$ zuordnet. Es wird vorausgesetzt, dass die Partitionierungsfunktion f eine exakte Aufteilung der Elemente liefert, so dass gilt:

$$n_i = \sum_{j=1}^{n} g(i,j) \quad \text{mit} \quad g(i,j) = \begin{cases} 1 & f(x_j) = i \\ 0 & f(x_j) \neq i \end{cases}$$

Die Ausgangssituation ist damit ähnlich wie vor der *Split*-Operation aus Abschnitt 3.2.2. Die Elemente liegen nun jedoch in verteiltem Speicher vor und die Durchführung hat parallel zu erfolgen.

Für parallele *out-of-place* Sortierverfahren ist diese Umverteilung der Elemente meist ein nicht näher beschriebenes Implementierungsdetail. Vorgehensweisen zur Realisierung von *in-place* Verfahren werden nicht gegeben. Um ein derartiges paralleles Sortierverfahren dennoch in der FMM verwenden zu können, werden im folgenden zwei Implementierungsvorschläge gegeben. Die erste verwendet kollektive und die zweite Punkt-zu-Punkt Kommunikationsoperationen.

5.2.1 Kollektive Kommunikationsoperationen und Sendepuffer

Da die Elemente auf jedem Prozess unsortiert vorliegen, können mit einzelnen Sendeoperationen meist auch nur jeweils einzelne Elemente übertragen werden. Da das Versenden vieler einzelner Elemente, bedingt durch die jeweilige Latenzzeit des Kommunikationsnetzwerks, meist sehr ineffizient ist, werden daher lokal in jedem Prozess $p - 1$ sog. Sendepuffer angelegt. In diese können mehrere Elemente, die an den gleichen Zielprozess gesendet werden müssen, aufgesammelt werden. Mit der Verteilung der Elemente auf die Sendepuffer wird am Anfang des Feldes begonnen, wodurch an dieser Stelle ein freier Bereich entsteht. Sind einzelne Sendepuffer voll, werden die darin enthaltenen Elemente an den jeweiligen Zielprozess gesendet und der Sendepuffer damit geleert. Zu empfangende Elemente werden in dem freien Bereich abgelegt. Elemente welche sich bereits auf ihrem Zielprozess befinden, werden direkt in den freien Bereich kopiert. Jeder Prozess kann immer nur so viele Elemente empfangen wie in seinem freien Bereich abgelegt werden können. Mit fortschreitendem Verlauf bewegt sich der freie Bereich in einem Stück, jedoch mit teils unterschiedlicher Grösse durch das gesamte Feld bis am Ende die letzten Elemente empfangen worden sind. In Abbildung 5.2 ist diese Vorgehensweise schematisch dargestellt.

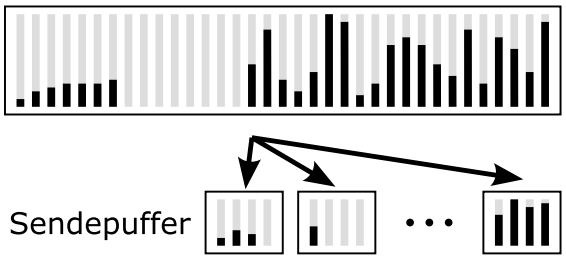

Abbildung 5.2: Vorgehensweise bei der Verwendung von Sendepuffern.

Findet das Senden und Empfangen der Elemente ungeordnet statt können, wie am einem Beispiel in Abbildung 5.3 beschrieben, Deadlocksituationen auftreten.

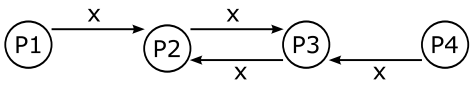

Abbildung 5.3: Beispiel eines Sendegraphen für das Auftreten von Deadlocksituationen.

Prozess P_1 hat x Elemente im Sendepuffer für Prozess P_2 abgelegt und versucht diese an P_2 zu versenden. Ähnliches gilt für P_2, P_3 und P_4. Darüber hinaus kann jeder Prozess in seinem freien Bereich x Elemente aufnehmen. Daraufhin entschliesst sich P_2 die x Elemente von P_1 und P_3 die von P_4 zu empfangen. Damit sind die freien Bereiche von P_2 und P_3 belegt und es können vorerst keine weiteren Elemente mehr empfangen werden. Da die Sendepuffer von P_2 und P_3 aber immer noch voll sind, können auch keine Elemente an die Sendepuffer verteilt werden, um einen neuen freien Bereich zu schaffen. Die beiden Prozesse befinden sich in einem Deadlock. Dies wurde möglich weil P_2 und P_3 von mehreren Prozessen gleichzeitig Elemente empfangen konnten und sich zufällig für die "falschen" entschieden haben.

Damit die einzelnen Prozesse die richtigen Entscheidungen treffen können, benötigen sie Informationen über die Inhalte der Sendepuffer aller anderen Prozesse. Daher werden diese in einem ersten Schritt, mit Hilfe einer kollektiven Kommunikationsoperation (`MPI_Allgather`) verteilt. Mit diesen Informationen kann dann jeder Prozess einen gerichteten gewichteten Graphen aufbauen, wie in Abbildung 5.4 an einem Beispiel dargestellt.

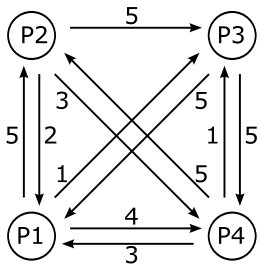

Abbildung 5.4: Beispiel eines Sendegraphen für 4 Prozesse.

Die Knoten repräsentieren die einzelnen Prozesse. Eine gerichtete Kante von P_i nach P_j mit dem Gewicht x_{ij} bedeutet, dass sich bei P_i genau x_{ij} Elemente im Sendepuffer für P_j befinden. Kanten mit dem Gewicht 0 werden nicht dargestellt. Anhand dieses Sendegraphen ermittelt jeder Prozess an wen er Elemente versenden darf bzw. von wem er welche empfangen muss. Dabei wird darauf geachtet, dass ein Prozess in jedem Kommunikationsschritt gleich viele Elemente versendet wie empfängt. Die Ermittlung der genauen Werte erfolgt in zwei Schritten.

1. Zuerst wird der direkte Austausch zwischen jeweils zwei Prozessen berücksichtigt. Sind zwei Prozesse P_i und P_j durch zwei Kanten mit den Gewichten x_{ij} und x_{ji} verbunden, dann können diese Prozesse sich gegenseitig $\min(x_{ij}, x_{ji})$ Elemente senden. Die Gewichte der Kanten werden anschliessend um $\min(x_{ij}, x_{ji})$ verringert und Kanten mit dem Gewicht 0 entfernt. Dies wird für alle möglichen Prozesspaare durchgeführt.

2. Mit Hilfe einer Tiefensuche werden in dem noch verbleibenden Graphen Zyklen ermittelt. Stellt x das kleinste Gewicht aller Kanten in einem ermittelten Zyklus dar, so versenden alle beteiligten Prozesse genau x Elemente an ihre jeweiligen Nachfolger in dem Zyklus. Anschliessend werden die Gewichte der Kanten um x verringert, Kanten mit einem Gewicht von 0 entfernt und der Zyklus somit zerstört. Dies wird solange durchgeführt bis keine Zyklen mehr vorhanden sind.

In Abbildung 5.5 ist dies anhand des Sendegraphen aus Abbildung 5.4 dargestellt.

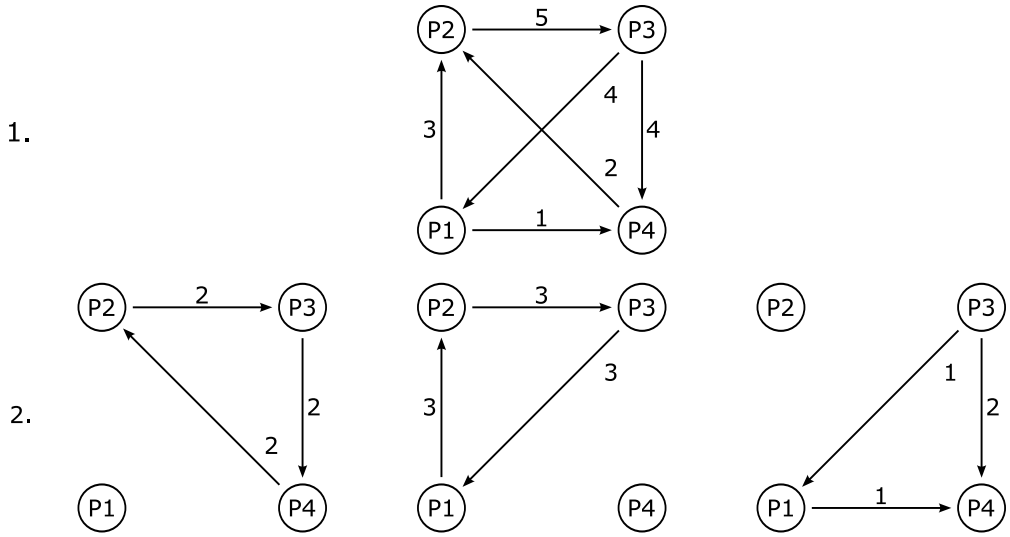

Abbildung 5.5: 1. Sendegraph nach dem Entfernen der direkt zwischen zwei Prozessen auszutauschenden Elemente und 2. Aufteilung in seine Zyklen sowie einen zyklenfreien verbleibenden Sendegraphen.

Nachdem dann jeder Prozess ermittelt hat wie viele Elemente er zu versenden und empfangen hat, wird die eigentliche Übertragung mit einem `MPI_Alltoallv`-Aufruf durchgeführt. Wie bereits erwähnt, wird durch diese Vorgehensweise sichergestellt, das in jedem Kommunikationsschritt von einem Prozess gleich viele Elemente versendet wie empfangen werden. Dies hat zur Folge, dass die Grösse des freien Bereichs immer mit der Anzahl der Elemente in den Sendepuffern übereinstimmt.

Damit die zuvor beschriebene Deadlocksituation auftreten konnte, mussten in einem Prozess gleichzeitig zwei Bedingungen erfüllt sein:

1. Der Prozess hat keinen freien Bereich zum Empfangen von Elementen.
2. Es kann auch kein neuer freier Bereich erzeugt werden, da der Prozess einen vollen Sendepuffer hat welcher verhindert, dass weitere Elemente auf die Sendepuffer verteilt werden können.

Indem die Grösse des freien Bereichs immer gleich der Anzahl der Elemente in den Sendepuffern ist, wird diese Situation ausgeschlossen.

Jedes Element wird nur einmal in den Sendepuffer kopiert und von da direkt zu seinem Zielprozess gesendet. Die Tiefensuche in dem Sendegraphen mit p Knoten und maximal p^2 Kanten besitzt eine Laufzeitkomplexität von $\mathcal{O}(p^2)$. Da in jedem Kommunikationsschritt eine Tiefensuche durchgeführt wird und immer mindestens zwei Elemente

ausgetauscht werden können, ergibt sich eine Laufzeitkomplexität von $\mathcal{O}(np^2)$ für die Umverteilung der Elemente.

Die Nachteile dieses Verfahrens sind, dass es sowohl zusätzlichen Speicher in Form der Sendepuffer als auch zusätzliche Kommunikation für die Verteilung der Sendepufferinformationen benötigt. Im folgenden Abschnitt wird daher eine alternative Vorgehensweise vorgestellt, die beides umgeht.

5.2.2 Punkt-zu-Punkt-Kommunikationsoperationen

Die Sendepuffer ermöglichen es, dass Elemente mit gemeinsamem Zielprozess aufgesammelt und dann mit einer Kommunikationsoperation versendet werden können. Wenn dieser zusätzliche Speicher nicht mehr zur Verfügung steht und dennoch nicht jedes Element einzeln versenden werden soll, so müssen die Elemente direkt in ihrem Feld umsortiert werden. In einem ersten Schritt werden daher nun alle Prozesse lokal und unabhängig voneinander ihre Elemente anhand der Partitionierungsfunktion in p Klassen einteilen und umordnen. Dies wird mit einer sequentiellen *Split*-Operation durchgeführt und ist in Abbildung 5.6 an einem Beispiel dargestellt.

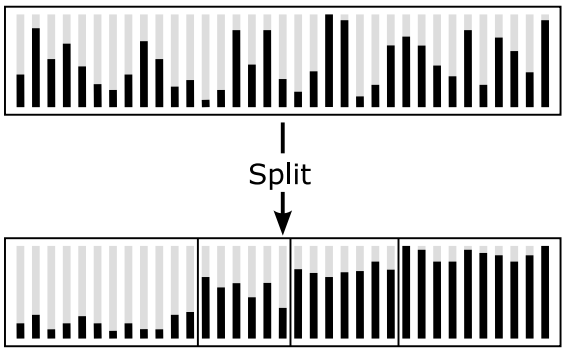

Abbildung 5.6: Umordnung der Elemente eines Prozesses in 4 Klassen.

Diese *Split*-Operation kann mit einem ähnlichen Verfahren realisiert werden, wie es in Abschnitt 3.2.2 für CS-Radixsort verwendet wird. Zuerst werden die Grössen der Klassen ermittelt und dann erfolgt die Umsortierung. Anstatt der Bitoperationen zur Klassifizierung eines Elements wird nun die Partitionierungsfunktion eingesetzt.

Anschliessend erfolgt die Umverteilung der Elemente unter den Prozessen. Dafür werden zuerst die lokalen Grössen der Klassen mit einem `MPI_Allgather`-Aufruf unter den Prozessen verteilt. Dann erfolgt die Verteilung der Elemente mit der *in-place* Kommunikationsoperation `MPI_Sendrecv_replace`. Auch in diesem Fall müssen immer gleich viele Elemente versendet wie empfangen werden. Zur Ermittlung der jeweiligen Kommunikationspartner wird dafür wieder, wie in Abschnitt 5.2.1 beschrieben, mit Hilfe des Sendegraphen vorgegangen.

Mit der kollektiven Kommunikationsoperation konnte jeder Prozess in einem Kommunikationsschritt gleichzeitig Elemente an mehrere Prozesse versenden und von mehreren Prozessen empfangen. Durch die Punkt-zu-Punkt-Kommunikation wird dies eingeschränkt und es können nur noch Elemente an einen Prozess versendet und von einem weiteren empfangen werden. Fand vorher die Kommunikation in mehreren `MPI_Alltoallv`-Aufrufen statt, erfolgt sie nun durch mehrere `MPI_Sendrecv_replace`-Aufrufe.

Die *Split*-Operation aus Abschnitt 3.2.2 ist ein linear skalierendes Verfahren bezüglich der Anzahl Elemente. Des Weiteren wird, wie auch in der Variante mit den Sendepuffern, jedes Element direkt an seinen Zielprozess versendet und pro Kommunikationsschritt eine Tiefensuche durchgeführt. Die Laufzeitkomplexität hat sich daher nicht verändert und die Umverteilung der Elemente bleibt ein linear skalierendes Verfahren bezüglich der Anzahl der Elemente.

Durch diese Vorgehensweise werden die zwei Nachteile der vorherigen Variante beseitigt. Zum einen wird kein zusätzlicher Speicher mehr für die Sendepuffer benötigt und zum anderen ist nur noch ein Aufruf von `MPI_Allgather` zur Verteilung von Informationen unter den Prozessen nötig. Des Weiteren ist die Anzahl der in einem Kommunikationsschritt versendeten Elemente nicht mehr durch die Grösse der Sendepuffer beschränkt. Hat darüber hinaus bei der Ermittlung der *Splitter*-Elemente für die Partitionierungsfunktion jeder Prozess seine Elemente bereits vorsortiert, kann sich dies ebenfalls verkürzend auf die Laufzeit der *Split*-Operation auswirken.

5.2.3 Flexibilität

Werden keine exakten Partitionierungsfunktionen verwendet, so muss ein Prozess mehr Elemente empfangen können als er selbst zu versenden hat. Bei der Ermittlung der zu versendenden Elemente anhand des Sendegraphen kann dies erkannt und entsprechend reagiert werden. Beispielsweise könnte in solchen Fällen ein Hilfsspeicher pro Prozess verwendet werden, der eine bestimmte Anzahl zusätzlicher Elemente aufnehmen kann. Reicht dies nicht aus oder ist kein Hilfsspeicher vorhanden, kann aber in jedem Fall kontrolliert abgebrochen werden. Je nach Fortschritt der Umverteilung wäre das Ergebnis eine entsprechende Vorsortierung der Elemente. Anschliessend könnte dann mit einem weiteren Sortierverfahren, wie in Abschnitt 4 vorgestellt, schnell eine vollständige Sortierung erreicht werden. Durch diese Vorgehensweise wäre es möglich auch effiziente probabilitische Partitionierungen für ein *in-place* Sortierverfahren zu verwenden.

5.3 Anwendung in der FMM

Das in diesem Kapitel vorgestellte Sortierverfahren kann verwendet werden, um in der parallelen Implementierung der FMM die Ausgabedaten, wie in Abschnitt 2.1.2 beschrieben, zurückzusortieren. Da sich in Abschnitt 3.3 herausgestellt hat, das Sortieren mit Permutation sehr ineffizient ist, wird für das Zurücksortieren der Ausgabedaten ebenfalls CS-Radixsort als sequentielles Sortierverfahren verwendet. In Abbildung 5.7 sind Speedups für die Parallelrechner JUMP und JUBL abgebildet, wie sie sich bei einer Beispielrechnung der FMM mit einem System bestehend aus 2^{24} Teilchen ergeben. Mit dem begrenzten Hauptspeicher eines einzelnen Blue Gene/L Rechenknoten konnten keine Messungen durchgeführt werden. Stattdessen wurde dafür die doppelte Laufzeit für das sequentielle Sortieren eines Systems mit 2^{23} Teilchen verwendet.

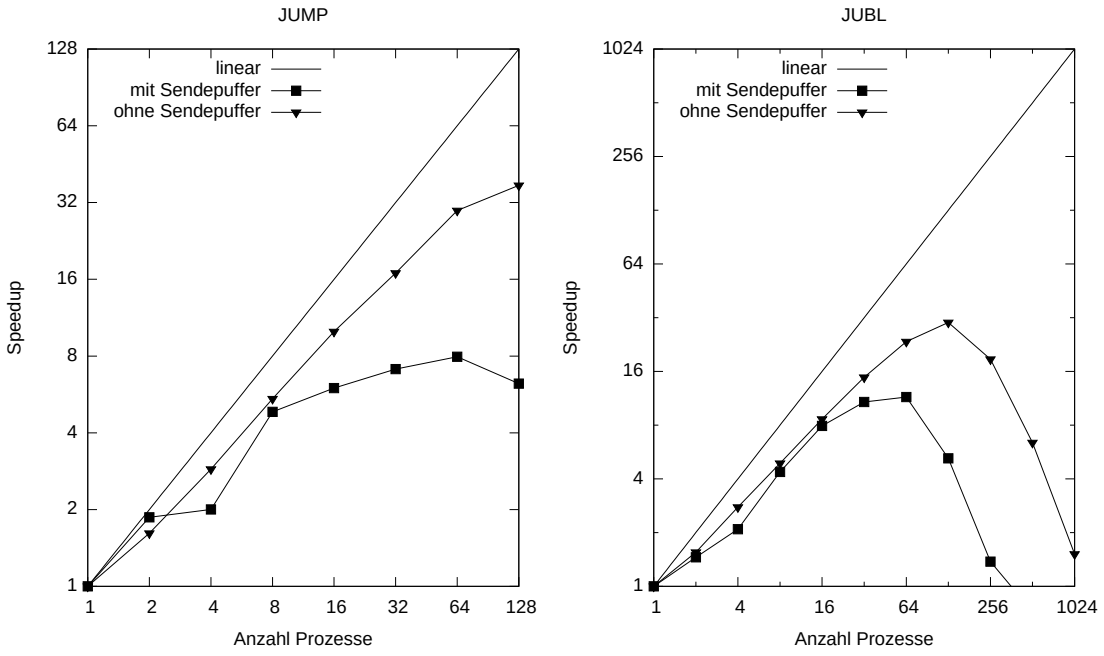

Abbildung 5.7: Laufzeiten für das Zurücksortieren der Ausgabedaten der FMM mit den Parallelrechnern JUMP und JUBL.

Es ist deutlich zu erkennen, dass die Variante ohne Sendepuffer die effizientere von beiden ist. Bis 128 Prozesse ergibt sich noch eine Effizienz von ca. 25%. Während für den Parallelrechner JUMP keine weiteren Messungen durchgeführt wurden, ist auf dem Blue Gene/L System zu erkennen, dass ab 256 Prozessen die Leistung enorm einbricht. Ähnlich verhält sich auch die Variante mit Sendepuffer, wobei diese bereits ab 128 Prozessen sinkte Speedup-Werte und damit steigende Laufzeiten verzeichnet.

Genauere Untersuchungen haben ergeben, dass der Anstieg der Laufzeit in beiden Varianten durch die Tiefensuche in dem Sendegraphen entsteht. Diese besitzt eine Lauf-

zeitkomplexität von $\mathcal{O}(p^2)$ und eignet sich offensichtlich nur bis zu einer bestimmten Anzahl von Prozessen. Um jedoch auch für eine grössere Anzahl von Prozessen das Zurücksortieren der Ausgabedaten anzubieten, kann beispielsweise ein Sortierverfahren aus Kapitel 4 verwendet werden. In Abbildung 5.8 sind die Laufzeiten der beiden Varianten und einem parallelen *in-place* Sortierverfahren basierend auf Mischen für das Blue Gene/L System dargestellt. Zusätzlich sind die Laufzeiten für die Durchführung der Tiefensuche mit angegeben.

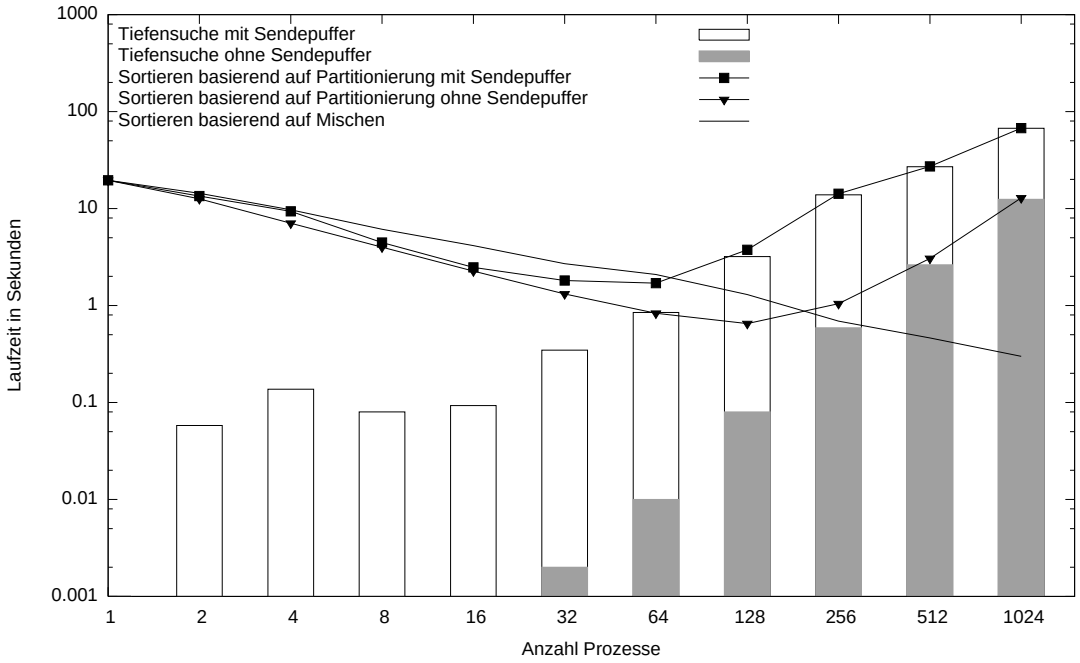

Abbildung 5.8: Laufzeiten für das Zurücksortieren der Ausgabedaten der FMM mit dem Parallelrechner JUBL.

Bis 128 Prozesse ist die Variante ohne Sendepuffer das schneller Verfahren. Mit zunehmender Anzahl von Prozessen steigt die Laufzeit für die Tiefensuche jedoch immer weiter an und bestimmt letztendlich das Verhalten des gesamten Sortierverfahrens. Dagegen sinkt die Laufzeit für das parallele Sortieren basierend auf Mischen auch noch mit mehr als 128 Prozessen und ist in diesen Fällen vorzuziehen.

6 Sorting Library (SL)

Um die Implementierung der in den vorherigen Kapiteln vorgestellten Verfahren möglichst flexibel und wiederverwendbar zu gestalten, wurden sie alle innerhalb einer mit SL (für *Sorting Library*) bezeichneten Bibliothek zusammengefasst.

Für die meisten parallelen Sortierverfahren gilt, dass sie sich aus mehreren einzelnen Komponenten zusammensetzten. Für das in Kapitel 4 vorgestellte parallele Sortierverfahren werden beispielsweise ein sequentielles Sortierverfahren, sequentielle 2-*merge*-Verfahren, Sortiernetzwerke usw. benötigt. Je nach Anforderung können dabei unterschiedliche Verfahren verwendet werden. Im Rahmen der SL sind für die einzelnen Komponenten konkrete Verfahren implementiert worden, welche für die vorgestellten parallelen Sortierverfahren nötig waren.

Um die Wiederverwendbarkeit der SL zu gewährleisten, wurden alle Funktionen so unabhängig wie möglich von der Art der zu sortierenden Elemente implementiert. Je nach Anwendungsfall kann die SL konfiguriert und übersetzt werden, ohne das Änderungen am eigentlichen Quellcode durchzuführen sind.

Dieses Konzept ist auch für die konkrete Anwendung am Beispiel der parallelen Implementierung der FMM von Nutzen, da es wie in Kapitel 2 beschrieben notwendig ist, zum Teil unterschiedliche Elemente mit mehreren verschiedenen Verfahren zu sortieren. Im folgenden wird daher die Verwendung der SL am Beispiel der FMM näher erläutert. Abschliessend wird ein Überblick gegeben über die Implementierung der Funktionen innerhalb der SL.

6.1 Anwendung

Um die SL als paralleles Sortierverfahren innerhalb anderer Anwendungen nutzen zu können, muss sie zuerst konfiguriert werden. Neben einigen plattformspezifischen Einstellungen werden dabei hauptsächlich Angaben zur Beschreibung der zu sortierenden Elemente und der zu verwendenden Verfahren benötigt. Im folgenden wird dies anhand der Sortierung der Eingabedaten der FMM beschrieben.

6.1.1 Konfiguration

Die Konfiguration der SL wird mittels C-Präprozessordirektiven innerhalb einer C-Headerdatei durchgeführt, welche in einem speziellen Verzeichnis abgelegt wird. In dieser Konfigurationsdatei werden einzelne C-Präprozessormakros definiert, welche die konkreten Datentypen der Elemente beschreiben.

Die Art der zu sortierenden Eingabedaten der FMM ist hier noch einmal aufgeführt:

	Anzahl und Datentyp pro Teilchen
Boxnummer (`box`)	`1 integer`
Position (`xyz`)	`3 doubles`
Ladung (`q`)	`1 double`
Adresse (`addr`)	`1 integer`

Tabelle 6.1: Zu sortierende Eingabedaten der FMM pro Teilchen

Insgesamt müssen die Inhalte von vier Feldern sortiert werden. Das erste Feld mit den Boxnummer wird dabei als Schlüssel verwendet. Eine entsprechende Konfigurationsdatei für die SL muss dafür die in Listing 6.1 beschrieben Einträge enthalten.

Listing 6.1: Konfiguration für das Sortieren der Eingabedaten der FMM

```
1  /* key section */
2  #define sl_key_type_c                long
3  #define sl_key_type_mpi              MPI_LONG
4  #define sl_key_size_mpi              1

6  #define sl_key_integer

8  /* data section */
9  #define SL_DATA

11 #define SL_DATA0 /* xyz */
12 #define sl_data0_type_c              double
13 #define sl_data0_size_c              3
14 #define sl_data0_type_mpi            MPI_DOUBLE
15 #define sl_data0_size_mpi            3

17 #define SL_DATA1 /* q */
18 #define sl_data1_type_c              double
19 #define sl_data1_size_c              1
20 #define sl_data1_type_mpi            MPI_DOUBLE
21 #define sl_data1_size_mpi            1

23 #define SL_DATA2 /* addr */
24 #define sl_data2_type_c              long
25 #define sl_data2_size_c              1
26 #define sl_data2_type_mpi            MPI_LONG
27 #define sl_data2_size_mpi            1

29 /* config section */
30 #define SL_USE_MPI
```

Für jeden Bestandteil der zu sortierenden Elemente müssen Angaben über den C-Datentyp und den MPI-Datentyp gemacht werden. Das Makro `sl_key_integer` signalisiert, dass der Schlüssel ein Integerdatentyp ist und somit auch Radixsort-Verfahren verwendet werden können. Mit dem Makro `SL_DATA` wird festgelegt, dass neben dem Schlüssel noch zusätzliche Daten zu einem Element gehören.

Darüber hinaus ist es mit Makros wie `sl_key_cmp_eq_`, `sl_key_cmp_le_`, ... möglich zu bestimmen, wie Schlüssel miteinander verglichen werden. Sind diese nicht angegeben, werden Standardvergleichsoperationen verwendet. Ähnliches gilt für Makros wie `sl_key_copy`, `sl_key_ncopy`, ... mit denen spezielle Funktionen zum Kopieren einzelner bzw. mehrerer Schlüssel angegeben werden können.

Um die SL für ein paralleles Sortierverfahren nutzen zu können, muss noch das Makro `SL_USE_MPI` definiert werden. Damit werden auch alle auf einer MPI-Implementierung basierenden Funktionen verfügbar. Darüber hinaus können noch weitere Einstellungen vorgenommen werden, um beispielsweise automatisch Informationen über Laufzeiten oder die Anzahl durchgeführter Vergleichs- und Kopieroperationen zu protokollieren.

6.1.2 Verwendung

Im Anschluss an die Konfiguration muss die SL mit Hilfe eines `makefiles` übersetzt werden. Dabei werden alle Funktionen der SL mit den entsprechend konfigurierten Datentypen übersetzt und in einer Archivdatei zusammengefasst. Des Weiteren werden automatisch die entsprechenden C-Headerdateien erzeugt um die Bibliotheksfunktionen aufrufen zu können. Alle für die Übersetzung notwendige Einstellungen, wie beispielsweise der zu verwendende Compiler, spezielle Compileroptionen oder die zu benutzende MPI-Implementierung können zusammen mit weiteren plattformspezifischen Einstellung in einer separaten Datei konfiguriert werden.

Da die eigentlichen Bibliotheksfunktionen nicht direkt aus einem in Fortran geschriebenen Programm aufgerufen werden können, muss in diesem Fall noch eine Wrapper-Funktion erzeugt werden. Für das Sortieren der Eingabedaten der FMM ist dies in Listing 6.2 dargestellt.

Listing 6.2: Wrapper-Funktion für das Sortieren der Eingabedaten der FMM

```
1  void pfmm_sort(long *n, long *box, double *xyz, double *q,
2   long *addr)
3  {
4    elements s;
5    int size, rank;
6
7    MPI_Comm comm = MPI_COMM_WORLD;
8
9    s.size = *n;
10   s.keys = box;
11   s.data0 = xyz;
12   s.data1 = q;
13   s.data2 = addr;
14
15   MPI_Comm_size(comm, &size);
16   MPI_Comm_rank(comm, &rank);
17
18   sort_radix(&s, NULL, 0, -1, -1, -1);
19
20   mpi_mergek(&s, sn_batcher, NULL, merge2_compo_hula, NULL,
21    size, rank, comm);
22 }
```

Mit dem abgebildeten C-Quellcode wird die Wrapper-Funktion `pfmm_sort` definiert. Da Argumente in Fortran durch *Call-by-Reference* übergeben werden, sind alle Argumente der Wrapper-Funktion Zeiger. Dabei enthält `n` eine Adresse an der ein Integerwert steht, welcher die Anzahl der zu sortierenden Elemente enthält. Die Zeiger `box`, `xyz`, `q` und `addr` enthalten die Adressen der Felder in denen die zu sortierenden Daten enthalten sind.

Mit den durch die Argumente übergebenen Informationen wird dann eine Datenstruktur vom Type `elements` gefüllt. Deren Einträge werden mit der Anzahl und den Adressen

der zu sortierenden Elemente belegt. Diese Datenstruktur stellt eine Schnittstelle dar, mit der allen Bibliotheksfunktionen eine Folge von Elementen übergeben werden können.

Durch die Aufrufe der Funktionen `sort_radix` und `mpi_mergek` werden dann die, für ein paralleles Sortierverfahren basierend auf Mischen notwendigen Komponenten der Bibliothek verwendet. Diese bekommen neben der Datenstruktur mit den zu sortierenden Elementen noch eine Reihe weiterer Argumente übergeben. Die Funktion `mpi_mergek` ist dabei für das Zusammenmischen der einzelnen sortierten Folgen verantwortlich. Sie bekommt mit dem Argument `sn_batcher` das zu verwendende Sortiernetzwerk *Batchers Merge-Exchange* und mit `merge2_compo_hula` das zu verwendende *2-merge*-Verfahren von Huang und Langston übergeben.

Nachdem dieser C-Quellcode ebenfalls übersetzt wurde, kann die Funktion `pfmm_sort` aus einem in Fortran geschriebenen Programm aufgerufen werden. Innerhalb von in C geschriebenen Programmen sind die Bibliotheksfunktionen der SL direkt aufrufbar.

6.1.3 mehrere Konfigurationen

Ist es in einem Programm erforderlich mehrere verschiedene Arten von Elementen zu sortieren, so müssen dafür auch mehrere verschiedene Konfigurationen der SL verwendet werden. Für jede dieser Konfigurationen wird dabei eine eigene Übersetzung der Bibliotheksfunktionen erzeugt. Um zu verhindern, dass es mehrere Funktionen mit dem gleichen Namen, aber erzeugt mit unterschiedlichen Konfigurationen gibt, ist es möglich jeder Konfiguration einen Namenspräfix zuzuweisen. Dieser wird automatisch jedem Funktionsnamen vor der Übersetzung vorangestellt. Darüber hinaus werden für verschiedene Konfigurationen auch automatisch verschiedene C-Headerdateien generiert, welche den Namenspräfix ebenfalls verwenden. Da es in C keine verschiedenen Namensräume gibt, war diese Vorgehensweise notwendig um ein entsprechendes Verhalten nachzubilden.

Am Beispiel der FMM wird dies wie folgt eingesetzt. Für das Sortieren der Eingabedaten wird eine Konfiguration mit dem Namenspräfix "input" erzeugt und für das Zurücksortieren der Ausgabedaten eine mit dem Namenspräfix "output". Die Bibliotheksfunktionen zum Sortieren der Eingabedaten beginnen dann mit dem Zusatz `input_` (`input_sort_radix`, `input_mpi_mergek`) und verwenden eine Datenstruktur vom Typ `input_elements`. Das gleiche gilt mit dem Zusatz `output_` für die Bibliotheksfunktionen zum Sortieren der Ausgabedaten.

6.1.4 maschinenabhängige Optimierung

Für einige innerhalb der SL implementierten Verfahren ist es unerlässlich bestimmte Parameter an die jeweils verwendete Hardware oder das konkrete Problem anzupassen. Dies betrifft beispielsweise das sequentielle Sortierverfahren CS-Radixsort aus Abschnitt 3.2.2. Sowohl die maximale Anzahl der Bitstellen r, die in einem Schritt betrachtet werden, als auch die Anzahl der Elemente t, ab der die Rekursion zugunsten des effizienteren *Straight insertion sort* vorzeitig abgebrochen wird, sind meist direkt von dem verwendeten Rechnersystem abhängig.

Anhand von kleinen Testbespielen können die konkreten Werte für diese Parameter auf einem jeweiligen Rechnersystem empirisch ermittelt werden. Für alle derartigen Parameter existiert innerhalb der SL eine weitere Konfigurationsdatei, in der diese Werte zentral angepasst werden können.

6.2 Implementierung

Die Implementierung der einzelnen Funktionen der SL erfolgt in C. Für die parallele Programmierung wird MPI verwendet. Darüber hinaus werden zur automatischen Übersetzung und Erzeugung der Headerdateien frei verfügbare und weit verbreitete Tools wie beispielsweise *make*, *sed* oder *Perl* verwendet. Die Konfiguration der SL mit konkreten Datentypen für die Elemente erfolgt mit Hilfe von C-Präprozessordirektiven noch vor der Übersetzung. Damit wird es dem Compiler ermöglicht, die Bibliotheksfunktionen gezielt für die verwendete Konfiguration zu optimieren.

Um die Implementierung der verschiedenen Funktionen der SL von der Art der jeweils konkret zu verarbeitenden Elemente unabhängig zu machen, wurde eine allgemeine Datenstruktur `elements` eingeführt. Diese ist wie in Listing 6.3 beschrieben definiert.

Listing 6.3: Definition der Datenstruktur `elements`

```
1   typedef struct _elements
2   {
3     int size;
4     sl_key_type_c *keys;

6   #ifdef SL_DATA
7    #ifdef SL_DATA0
8     sl_data0_type_c *data0;
9    #endif
10   #ifdef SL_DATA1
11    sl_data1_type_c *data1;
12   #endif
13   #ifdef SL_DATA2
14    sl_data2_type_c *data2;
15   #endif
16   #ifdef SL_DATA3
17    sl_data3_type_c *data3;
18   #endif
19  #endif /* SL_DATA */
20 } elements;
```

In jedem Fall enthält diese Datenstruktur einen Eintrag für die Anzahl der Elemente und einen Zeiger auf das Feld indem die Schlüssel abgelegt sind. Dazu kommen, abhängig von der verwendeten Konfiguration, noch Zeiger auf Felder mit zusätzlichen Daten. Dadurch wird es ermöglicht, dass jede Funktion der SL nur noch mit dem allgemeinen Datentyp `elements` arbeitet und losgelöst von konkreten Typen der Schlüssel und Daten realisiert werden kann.

Für die Behandlung der `elements` Datenstruktur innerhalb der einzelnen Funktionen werden ebenfalls C-Präprozessormakros verwendet. In Listing 6.4 wird dies anhand einer Funktion demonstriert, welche eine gegebene Folge von Elementen umdreht.

Listing 6.4: Implementierungsbeispiel einer Funktion der SL

```
1  int sl_elem_reverse(elements *e, elements *t)
2  {
3    elements front, back, end;

5    sl_elem_assign(e, &front);
6    sl_elem_assign_at(e, e->size - 1, &back);
7    sl_elem_assign_at(e, e->size / 2, &end);

9    while (front.keys < end.keys)
10   {
11     sl_elem_xchange(&front, &back, t);
12     sl_elem_inc(&front);
13     sl_elem_dec(&back);
14   }

16   return 0;
17 }
```

Die Datenstruktur `elements` kann dabei allgemein als Zeiger auf eine Folge von Elementen verwendet werden. Mit den Makros `sl_elem_assign` und `sl_elem_assign_at` können Zuweisungen mit diesen Zeigern vorgenommen werden. Die Makros `sl_elem_inc` und `sl_elem_dec` verändern diese Zeiger so dass sie auf das nächste bzw. vorhergehende Element in einer Folge verweisen. Das Makro `sl_elem_xchange` vertauscht zwei Elemente miteinander. Damit kann diese Funktion vollkommen unabhängig davon, ob wirklich zusätzliche Daten vorhanden sind oder wie diese konkret aussehen, realisiert werden. Vor der Übersetzung wird dann durch den C-Präprozessor der jeweils für eine konkrete Konfiguration verwendete C-Quellcode erzeugt, indem die Makros automatisch durch die, auf den Schlüsseln und Daten auszuführenden Operationen, ersetzt werden.

Beim Versenden von Elementen mittels MPI, muss der entsprechende C-Quellcode für jedes zu versendende Feld eine eigene Kommunikationsoperation enthalten. Wie in Listing 6.5 dargestellt werden dabei die Kommunikationsoperationen für die Schlüssel immer übersetzt und für die zusätzlichen Daten nur, wenn auch die entsprechenden Makros `SL_DATA0`, `SL_DATA1` usw. definiert sind.

Listing 6.5: Versenden von Elementen mittels MPI in der SL

```
1  MPI_Send(e->keys, ...);

3  #ifdef SL_DATA0
4   MPI_Send(e->data0, ...);
5  #endif
6  #ifdef SL_DATA1
7   MPI_Send(e->data1, ...);
8  #endif
9  #ifdef SL_DATA2
10  MPI_Send(e->data2, ...);
11 #endif
12 #ifdef SL_DATA3
13  MPI_Send(e->data3, ...);
14 #endif
```

Neben den für die parallelen Sortierverfahren benötigten Funktionen sind im Rahmen der SL noch einige weitere Funktionen implementiert worden. Diese dienen beispielsweise zur Verifikation ob eine Folge von Elementen korrekt sortiert wurde. Des Weiteren existieren Funktionen zur Erzeugung von CRC-Prüfsummen [31] der zu sortierenden Elemente. Diese sind hauptsächlich während der Entwicklung der Sortierverfahren eingesetzt worden, um die Integrität der sortierten Elemente zu überprüfen.

7 Zusammenfassung

Für die parallele Implementierung der FMM werden linear skalierende parallele Verfahren zum Sortieren der Ein- und Ausgabedaten benötigt. Dabei ist es möglich, dass die zu sortierenden Daten selbst den Grossteil des verfügbaren Speichers belegen. Für die parallelen Sortierverfahren muss es deshalb möglich sein, mit einer konstanten bzw. vom Umfang der zu sortierenden Daten unabhängigen Menge an zusätzlichem Speicher aus zukommen. Diese Einschränkung ist mit einer Vielzahl der verfügbaren parallelen Sortierverfahren nicht zu erreichen.

Da für ein paralleles Sortierverfahren auch ein sequentielles benötigt wird, wurde zuerst ein linear skalierende *in-place* Sortierverfahren implementiert. Dabei wurde besonders darauf geachtet, dass es effizient im Falle von bereits vorsortierten Daten arbeitet. Dies betrifft sowohl in bekanntem als auch unbekanntem Ausmass vorsortierte Daten. Das zu diesem Zweck implementierte CS-Radixsort benötigte dabei innerhalb einer Beispielrechnung der FMM nur 60% der Laufzeit des bereits vorhandenen Radixsort-Verfahren.

Darauf aufbauend wurde für das Sortieren der Eingabedaten der FMM ein auf Mischen basierendes paralleles Sortierverfahren implementiert. Als Ausgangspunkt diente ein Verfahren mit einer Speicherkomplexität von $\mathcal{O}(\sqrt{n})$. Dies wurde entsprechend angepasst, so dass es flexibel mit dem jeweils verfügbarem Speicher umgehen kann. Die Verwendung von mehr Speicher wirkt sich dabei positiv auf die Effizienz des Sortierverfahrens aus.

Für das Zurücksortieren der Ausgabedaten der FMM wurde ein auf Partitionierung basierendes paralleles Sortierverfahren implementiert. Da die Schlüssel, nach denen die Ausgabedaten sortiert werden müssen eine Permutation beschreiben, lässt sich in diesem Fall auf einfache Art und Weise die notwendige exakte Partitionierungsfunktion erzeugen. Um die Umverteilung der Elemente *in-place* mit Standard-MPI-Funktionen zu realisieren, wurden zwei Verfahren vorgestellt. Das erste verwendet kollektive Kommunikationsoperationen und benötigt zusätzlichen Speicher in Form von Sendepuffern. Das zweite verwendet *in-place* Punkt-zu-Punkt-Kommunikationsoperationen. Für beide Verfahren war es zur Vermeidung von Deadlocksituationen notwendig die Kommunikation entsprechend zu organisieren, wodurch es bei der Verwendung von mehr als 128 Prozessen zu einem enormen Anstieg der Laufzeiten kommt. In diesen Fällen ist daher ein paralleles Sortierverfahren, wie für das Sortieren der Eingabedaten vorgestellt, zu bevorzugen.

Alle im Rahmen dieser Arbeit implementierten Sortierverfahren wurden innerhalb einer Bibliothek zusammengefasst. Dabei sind die einzelnen Verfahren unabhängig von der genauen Art der jeweils zu sortierenden Daten realisiert worden. Die Bibliothek lässt sich an einer zentralen Stelle konfigurieren und an die jeweilige Aufgabe anpassen. Tief ergehende Eingriffe in den Quellcode einzelner Funktionen sind nur für die Entwicklung

und Wartung der Sortierverfahren nötigt, jedoch nicht für deren Verwendung. Dadurch ist es möglich die Sortierverfahren einfach an veränderte Situationen anzupassen und darüber hinaus auch ausserhalb der FMM zu verwenden.

Insgesamt ist es mit den implementierten Verfahren gelungen, die Anforderungen der FMM bezüglich Laufzeit- und Speicherkomplexität zu erfüllen. Bei der Nutzung innerhalb der parallelen FMM traten keine Probleme auf.

A Parallelrechner

Die in dieser Arbeit genutzten Parallelrechner JUMP und JUBL befinden sich beide am Zentralinstitut für Angewandte Mathematik im Forschungszentrum Jülich. Für die parallele Programmierung steht jeweils eine MPI-Implementierung [32] zur Verfügung.

A.1 IBM Regatta p690 (JUMP)

Der Jülicher Multiprosseror-Rechner (kurz JUMP) [34] ist ein IBM Regatta p690 System bestehend aus 41 Knoten. Jeder davon ist ein symmetrischen Multiprozessorsystem (SMP) mit jeweils 32 Power4+ Prozessoren mit 1.7 GHz Taktfrequenz und 128 GB Hauptspeicher. Verbunden sind die Knoten mittels eines sog. High-Performance-Switch Netzwerks. Mit insgesammt 1312 Prozessoren ergibt sich eine theoretische Gesamtrechenleistung von 8.9 Teraflops.

Als Betriebssystem wird AIX 5L und als C-Compiler *XL C/C++ Version 7.0 for AIX* von IBM verwendet.

Beim JUMP handelt es sich um ein Produktionssystem auf dem im Durchschnitt ca. 80 verschiedene Anwendungen gleichzeitig ausgeführt werden und dabei um gemeinsam genutzte Ressourcen konkurrieren. Dies betrifft beispielsweise die Nutzung des Kommunikationsnetzwerks und Zugriffe auf den Hauptspeicher innerhalb einzelner Knoten.

A.2 Blue Gene/L (JUBL)

Das Jülicher Blue Gene/L System (kurz JUBL) [35] besteht aus einem Rack mit insgesamt 1024 Rechenknoten. Jeder Rechenknoten ist ein Dualprozessor mit 32-Bit PowerPC 440 Kernen mit 700 MHz Taktfrequenz und 512 MB Hauptspeicher. Verbunden sind die Rechenknoten untereinander durch 3 verschiedene Netzwerke. Pro Rack ergibt sich eine theoretische Gesamtrechenleistung von 5.7 Teraflops.

Auf den Rechenknoten selbst läuft nur ein minimales Betriebssystem. Als C-Compiler wird eine spezielle Version des *XL C/C++ Version 7.0 for AIX* von IBM verwendet. Die zur Verfügung stehende MPI-Implementierung basiert auf MPICH2 und ist speziell an die Architektur des Blue Gene/L Systems angepasst.

Das Blue Gene/L System stellt ein Testsystem mit sehr eingeschränktem Nutzerkreis dar. Dies hat zur Folge, dass den darauf ausgeführten Anwendungen die jeweiligen Ressourcen weitestgehend exklusiv zur Verfügung stehen.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] L. Greengard, V. Rokhlin
A fast algorithm for particle simulations
Journal of Computational Physics, Vol. 73, pg. 325-348, 1987

[2] C.A. White, M. Head-Gordon
Derivation and Efficient Implementation of the Fast Multipole Method
The Journal of Chemical Physics, Vol. 101, pg. 6593-6605, 1994

[3] C.A. White, M. Head-Gordon
Rotating around the quartic angular momentum barrier in fast multipole method calculations
The Journal of Chemical Physics, Vol. 105, pg. 5061-5067, 1996

[4] J. Shimada, H. Kaneko, T. Takada
Performance of fast multipole methods for calculating electrostatic interactions in biomacromolecular simulations
Journal of Computational Chemistry, Vol. 15, Issue 1, pg. 28-43, January 1994

[5] W.F. van Gunsteren, H.J.C. Berendsen
Computer Simulation of Molecular Dynamics: Methodology, Applications and Perspectives in Chemistry
Angewandte Chemie International Edition in English, Vol. 29, pg. 992-1023, 1990

[6] Donald E. Knuth
The Art Of Computer Programming - Volume 3 / Sorting and Searching
Addison-Wesley 1973

[7] P. Hildebrandt, H. Isbitz
Radix exchange - an internal sorting method for digital computers
JACM 6, S. 156-163 (1959)

[8] A. Sohn, Y. Kodama
Load Balanced Parallel Radix Sort
Proceedings of the 12th International Conference on Supercomputing, pg. 305-312, 1998

[9] D. Jiménez-González, J. J. Navarro, J.-L. Larriba-Pey
Fast parallel in-memory 64-bit sorting
Proceedings of the 15th International Conference on Supercomputing, pg. 114-122, 2001

[10] S.-J. Lee, M. Jeon, D. Kim, A. Sohn
Partitioned parallel radix sort
Journal of Parallel and Distributed Computing, Vol. 62, pg. 656-668, 2002

[11] S.Q. Zheng, B. Calidas, Y. Zhang
An Efficient General In-Place Parallel Sorting Scheme
The Journal of Supercomputing, Vol. 14, pg. 5-17, 1999

[12] M.D. Atkinson, J.-R. Sack, B. Santoro, T. Strothotte
Min-max heaps and generalized priority queues
Communications of the ACM, Vol. 29, pg. 996-1000, 1986

[13] P.M. McIlroy
Engineering Radix Sort

[14] D. Jimenez-Gonzalez, J. Navarro, J. Larriba-Pey
CC-Radix: A Cache Conscious Sorting Based on Radix Sort
Euromicro Conference on Parallel Distributed and Network based Processing, S. 101-108, Februar 2003

[15] J. Nieplocha, M. Krishnan, B. Palmer, V. Tipparaju, Jialin Ju
The Global Arrays User's Manual
http://www.emsl.pnl.gov/docs/global/

[16] D. Bitton, D.J. DeWitt, D.K. Hsiao, J. Menon
A Taxonomy of Parallel Sorting
Computing Surveys, Vol. 16, No. 3, September 1984

[17] Selim G. Akl
Parallel Sorting Algorithms
Academic Press 1985

[18] Dana Richards
Parallel Sorting - A Bibliography
SIGACT News, Vol. 18, pg. 28-46, 1986

[19] A. Tridgell, R.P. Brent
A general-purpose parallel sorting algorithm
Int. J. High Speed Computing 7, 2 (1995), 285-301

[20] T. Hayashi, K. Nakano, S. Olariu
Work-Time Optimal k-Mege Algorithms on the PRAM
IEEE Transactions On Parallel And Distributed Systems, Vol. 9, No. 3, March 1998

[21] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, D. Walker
Solving Problems on Concurrent Processors - Volume 1 / General Techniques and Regular Problems
Prentice Hall 1988

[22] D.R. Helman, D.A. Bader, J. Jájá
A Randomized Parallel Sorting Algorithm with an Experimental Study
Journal of Parallel and Distributed Computing 52, pg. 1-23, 1998

[23] X. Li, P. Lu, J. Schaeffer, J. Shilligton, P.S. Wong, H. Shi
On the Versatility of Parallel Sorting by Regular Sampling
Parallel Computing, Vol. 19, Issue 10, pg. 1079-1103, 1993

[24] F. Dian, T. Zhizhong
Parallel Sorting by Exact Splitting
2004 International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'04), p. 92

[25] F.K. Hwang, S. Lin
A Simple Algorithm for Merging Two Disjoint Linearly-Ordered Sets
SIAM Journal on Computing, Volume 1, pg. 31-39

[26] M.A. Kronrod
An optimal ordering algorithm without a field of operation
Dokl. Akad. Nauk SSSR 186, pg. 1256-1258, 1969

[27] H. Mannila, E. Ukkonen
A simple linear-time algorithm for in situ merging
Information Processing Letters, Vol. 18, Issue 4, pg. 203-208, May 1984

[28] B.-C. Huang, M.A. Langston
Practical in-place merging
Communications of the ACM, Vol. 31, Issue 3, pg. 348-352, March 1988

[29] V. Geffert, J. Katajainen, T. Pasanen
Asymptotically efficient in-place merging
Theoretical Computer Science, Vol. 237, pg. 159-181, 2000

[30] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein
Introduction to Algorithms (Second Edition)
MIT Press and McGraw-Hill, 2001

[31] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery
Numerical Recipes in C - The Art of Scientific Computing (Second Edition)
Cambridge University Press, 1992

[32] Message Passing Interface Forum
MPI: A Message-Passing Interface Standard
http://www.mpi-forum.org/

[33] IBM Advanced Computing Technology Center
Hardware Performance Monitor
http://www.research.ibm.com/actc/projects/hardwareperf2.shtml

[34] http://jumpdoc.fz-juelich.de/

[35] http://www.fz-juelich.de/zam/ibm-bgl/