Hochleistungsrechnen

Steffen Börm

Stand 23. Januar 2017

Inhaltsverzeichnis

1 Einleitung

Die ersten Rechenmaschinen, aus denen sich die modernen Computer entwickelt haben, wurden für mathematische Fragestellungen konstruiert: Es ging darum, Logarithmentafeln fehlerfrei aufzustellen oder die Flugbahn von Geschossen vorherzusagen.

Moderne Computer lassen sich frei programmieren und haben sich so viele Anwendungsgebiete jenseits der Mathematik erschlossen, beispielsweise bei der Automatisierung von Verwaltungsvorgängen oder in der Unterhaltungsindustrie.

Derartige Aufgaben stellen keine allzu hohen Ansprüche an die Leistungsfähigkeit des Computers, so dass relativ preiswerte Bauteile zum Einsatz kommen können und man bei der Programmierung die Aufmerksamkeit mehr auf die Zuverlässigkeit und Benutzerfreundlichkeit der Software als auf die Geschwindigkeit legen kann.

Im wissenschaftlichen Bereich dagegen wachsen die Anforderungen an die Rechenleistung ständig, denn beispielsweise bei Simulationen hängt die erreichbare Genauigkeit unmittelbar von der Rechenzeit und häufig auch von der Menge des verfügbaren Speichers ab. Um zunehmend komplexere Phänomene behandeln zu können, werden zunehmend größere Computersysteme erforderlich.

Während sich in der Vergangenheit Programmierer darauf verlassen konnten, dass die nächste Prozessorgeneration wesentlich schneller als alle Vorgänger sein würde, so dass die bestehende Software auf dem nächsten Computersystem ohne weiteres Zutun eine deutlich kürzere Rechenzeit versprach, stagnieren die Taktfrequenzen seit einigen Jahren.

Ursache ist die Physik: Einerseits führen höhere Taktfrequenzen zu höherer Verlustleistung, andererseits müssen Signale eine Chance haben, von einer Seite des Chips zur anderen zu gelangen. Bei einer Vakuumlichtgeschwindigkeit von 3×10^{10} Zentimetern könnte ein Signal beispielsweise einen Chip von einem Zentimeter Durchmesser 3×10^{10}-mal pro Sekunde durchqueren. Das entspricht einer maximalen Taktfrequenz von 30 Gigahertz. Wenn man bedenkt, dass die Signale auch noch Informationen tragen müssen, sind 3 Gigahertz, die heutige Mittelklasse-Prozessoren erreichen, eine beachtliche technische Leistung. Sowohl das Problem der Verlustleistung als auch das der Signalübertragung lassen sich im Prinzip lösen, indem man die Chips verkleinert, allerdings gelangt man bei den heute üblichen Strukturgrößen in Bereiche, in denen die Quantenphysik damit beginnt, Schwierigkeiten zu verursachen.

Eine Lösung bietet die Parallelisierung von Rechenvorgängen: Wenn ich eine Berechnung gleichmäßig auf 2 Prozessoren verteilen kann, wird sich die Rechenzeit dadurch halbieren. Je mehr Prozessoren ich einsetze, desto schneller erhalte ich das Ergebnis der Berechnung.

Leider gilt diese Rechnung nur unter idealen Bedingungen, nämlich wenn alle Prozessoren exakt gleich lange rechnen. Bei den meisten Algorithmen ist es allerdings erforder-

lich, dass sich die Prozessoren untereinander abstimmen, so dass es im ungünstigsten Fall passieren kann, dass ein Prozessor ein Zwischenergebnis berechnet, auf das alle anderen warten müssen. Unter diesen Umständen wird kaum eine Beschleunigung festzustellen sein.

Die Synchronisation vieler Prozessoren, die gemeinsam an einer größeren Berechnung arbeiten, ist eine anspruchsvolle Aufgabe, der man sich auf unterschiedlichen Wegen nähern kann. Besonders einfach ist es, nicht mehrere Prozessoren zu verwenden, sondern lediglich mehrere Rechenwerke. In diesem Fall kann ein Prozessor in einem Taktzyklus beispielsweise vier Additionen statt einer einzigen durchführen und im Idealfall eine Rechnung viermal schneller abschließen als zuvor. Derartige Systeme eignen sich gut für die Verarbeitung von Blöcken gleichartiger Datenobjekte und tragen den Namen *Vektorrechner*, da in der linearen Algebra Vektoren ein wichtiges Beispiel für derartige Blöcke sind. Vektorrechner lassen sich relativ einfach realisieren, da Rechenwerke wesentlich einfacher strukturiert sind als Prozessoren und deshalb nur einen geringen Schaltungsaufwand mit sich bringen. Ihr Nachteil besteht allerdings darin, dass alle Rechenwerke immer dieselbe Operation ausführen müssen, so dass sich nur bestimmte Algorithmen effizient umsetzen lassen. Besonders erfolgreich sind Vektorrechner derzeit im Bereich der Computergrafik, bei der für alle Bildpunkte ähnliche Operationen durchzuführen sind, und bei der Simulation partieller Differentialgleichungen, die ähnliche Eigenschaften aufweisen.

Wesentlich flexibler, aber auch wesentlich schwieriger zu programmieren, sind Shared-Memory-Systeme, die beispielsweise in Gestalt von Mehrkernprozessoren in vielen modernen Computern auftreten. Bei diesen Systemen arbeiten mehrere Prozessoren gleichzeitig, die unabhängig voneinander programmiert werden können. Der Austausch von Daten erfolgt über einen gemeinsam genutzten Hauptspeicher, ebenso wie die Synchronisation der Prozessoren. Shared-Memory-Systeme sind flexibel genug, um eine große Anzahl von Problemen effizient behandeln zu können, allerdings stellt ihre Programmierung eine Herausforderung dar, da durch die asynchrone Ausführung der verschiedenen Programme das Verhalten des Gesamtsystems schwer vorhersagbar wird. Ein Nachteil des Shared-Memory-Ansatzes besteht darin, dass der für den gemeinsamen Zugriff auf den Speicher erforderliche Schaltungsaufwand mit zunehmender Prozessorenzahl sehr schnell wächst.

Eine Mischung aus Shared-Memory-Systemen und Vektorrechnern findet man bei Grafikkarten, in deren Verbesserung dank ihrer Bedeutung für Computerspiele genug Geld investiert wurde, um ihnen zu einer Rechenleistung zu verhelfen, die mit der früher Supercomputer konkurrieren kann. Eine typische Grafikkarte besteht aus einer *graphics processing unit* (GPU) und einem Speicher, der über eine besonders breite Schnittstelle angebunden wird. Die GPU setzt sich aus mehreren Prozessorkernen zusammen, die jeweils wie bei einem Vektor-Rechner mit mehreren Rechenwerken ausgestattet sind. Im Prinzip lassen sich Grafikkarten wie Shared-Memory-Systeme programmieren, allerdings erreichen sie ihre volle Leistung nur, wenn die Vektor-Rechenwerke geeignet angesteuert werden können. Um die Programmierung der Grafikkarten zu erleichtern werden sie einerseits mit Schaltungen für die effiziente Verwaltung von Tausenden von Befehlsfolgen (*threads*) ausgestattet, die es erlauben, die bei ungünstigen Algorithmen auftretenden

Wartezeiten zu verstecken, während andererseits Compiler zum Einsatz kommen, die konventionell geschriebene Programme in eine für die Grafikkarte möglichst gut geeignete Form zu bringen versuchen.

Die Beschränkung der Shared-Memory-Systeme auf (relativ) wenige Prozessoren ist eine Konsequenz des hohen Schaltungsaufwands, der nötig ist, um allen Prozessoren den gemeinsamen Zugriff auf den Hauptspeicher zu ermöglichen. Bei einem verteilten System wird auf diesen gemeinsamen Hauptspeicher verzichtet, die Synchronisation erfolgt über ein (mehr oder weniger) konventionelles Kommunikationsnetzwerk. In der Regel erfolgt die Kommunikation in einem verteilten System wesentlich langsamer als in einem Shared-Memory-System, aber dafür lässt es sich auch wesentlich kostengünstiger und wesentlich größer aufbauen. Die größten aktuellen Großrechner sind verteilte Systeme, allerdings bestehen die einzelnen Knoten des korrespondierenden Kommunikationsnetzwerks durchaus häufig aus Shared-Memory-Systemen, die häufig von Grafikkarten unterstützt werden. Die Programmierung eines verteilten Systems ist auf den ersten Blick sehr einfach: Jeder Knoten führt ein Programm aus, jeder Datenaustausch muss explizit abgewickelt werden, die einzelnen Knoten können sich nicht „versehentlich“ stören. Auf den zweiten Blick stellt sich allerdings heraus, dass einerseits die Synchronisation der Knoten ähnlich kompliziert wie auf Shared-Memory-Systemen werden kann und dass andererseits alle Zwischenergebnisse, die auf einem Knoten berechnet wurden, mit expliziten Funktionsaufrufen auf alle Knoten übertragen werden müssen, die diese Ergebnisse brauchen könnten.

Ziel dieser Vorlesung ist es, die grundlegenden Konzepte zu vermitteln, die man benötigt, um ein Programm „möglichst schnell“ zu machen, indem die besonderen Eigenschaften des zur Verfügung stehenden Computersystems möglichst gut ausgenutzt werden. Dabei gehen wir von einer Anzahl typischer Modellprobleme aus und untersuchen, welche davon sich für bestimmte Rechnerarchitekturen gut eignen und wie man sie implementieren sollte, um die Möglichkeiten des Computers gut auszunutzen.

Danksagung

Ich bedanke mich bei Sabrina Reif für Hinweise auf Fehler in früheren Fassungen dieses Skripts und für Verbesserungsvorschläge.

2 Anwendungsbeispiel: Wellenausbreitung

Hochleistungsrechner werden in der Regel für die Behandlung wissenschaftlicher Fragestellungen, eingesetzt, die typischerweise in der Sprache der Mathematik formuliert sind. In diesem Kapitel werden einige Modellprobleme vorgestellt, an denen sich Vor- und Nachteile der unterschiedlichen Programmiertechniken und Rechnerarchitekturen exemplarisch demonstrieren lassen.

2.1 Wellengleichung

Die Simulation der Ausbreitung von Wellen ist ein typisches Beispiel für eine Anwendung leistungsfähiger Computer im Bereich der Naturwissenschaften.

Die korrespondierende *Wellengleichung* lässt sich gut am Beispiel einer schwingenden Saite motivieren, die an ihren beiden Endpunkten eingespannt ist.

Physikalisches Modell

Um eine Gleichung herleiten zu können, die das mechanische Verhalten der Saite beschreibt, ersetzen wir die kontinuierliche Saite durch viele „hinreichend kleine“ Punktmassen, die durch Federn aneinander gekoppelt sind (siehe Abbildung 2.1). Entsprechend den Regeln der klassischen Mechanik müssen wir die Kräfte berechnen, die auf die einzelnen Punktmassen wirken.

Eine einzelne Punktmasse ist in unserem Modell über Federn an ihre beiden Nachbarn gekoppelt. Sie befindet sich im Gleichgewichtszustand, wenn die von den Federn ausgeübten Kräfte sich gerade gegenseitig aufheben, wenn also die Masse exakt zwischen beiden Nachbarn liegt.

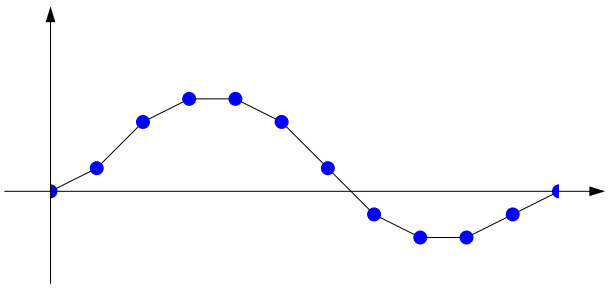

Abbildung 2.1: Approximation einer schwingenden Saite durch „hinreichend kleine“ Punktmassen

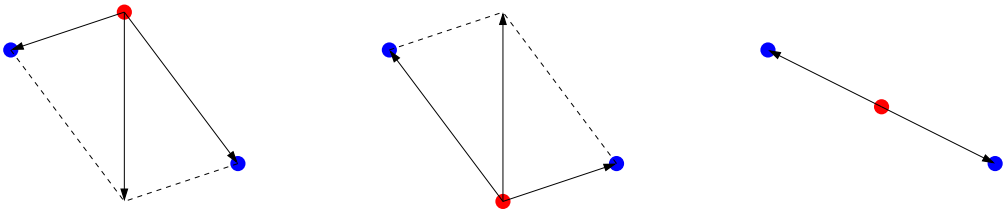

Abbildung 2.2: Auf eine Punktmasse wirkende Kräfte

Falls die Punktmasse von diesem Gleichgewichtspunkt abweicht, üben die Federn eine Rückstellkraft aus, die sie in Richtung des Gleichgewichtspunkts zieht (siehe Abbildung 2.2). Diese Kraft bewirkt nach den Regeln der Mechanik eine Beschleunigung der Punktmasse in dieser Richtung.

Mathematische Formulierung

Um die Saite im Computer simulieren zu können, müssen wir die auftretenden Größen durch Zahlen ausdrücken. Als Vereinfachung gehen wir davon aus, dass die einzelnen Punktmassen in horizontaler Richtung einen festen Abstand $h \in \mathbb{R}_{>0}$ aufweisen. Wir brauchen also nur die vertikale Position jeder Masse abzuspeichern. Dazu bezeichnen wir mit $n \in \mathbb{N}$ die Anzahl der Massen und numerieren sie von links nach rechts von 1 bis n durch. Um die Notation einfach zu halten, fügen wir die Indizes 0 und $n+1$ für den linken und rechten Randpunkt hinzu.

Für ein $i \in \{0, \dots, n+1\}$ und einen Zeitpunkt $t \in \mathbb{R}$ bezeichnet $x_i(t)$ die Auslenkung der i-ten Punktmasse. Neben der Auslenkung besitzt die Masse auch eine Geschwindigkeit, die wir mit $v_i(t)$ bezeichnen. Bei beiden Größen soll ein positives Vorzeichen eine Auslenkung beziehungsweise eine Bewegung nach oben bedeuten, während ein negatives Vorzeichen zu der Richtung nach unten gehört.

Der Tatsache, dass sich die Randpunkte nicht bewegen, entsprechen die Gleichungen

$$x_0(t) = x_{n+1}(t) = 0, \qquad v_0(t) = v_{n+1}(t) = 0 \qquad \text{für alle } t \in \mathbb{R}.$$

Um die Bewegung unserer Saite beschreiben zu können, müssen wir die Kräfte berechnen, die auf die einzelnen Punktmassen wirken. Das können wir mit dem Federgesetz von Hooke tun, das besagt, dass die von der Feder ausgeübte Kraft proportional zu ihrer Auslenkung aus der Ruhelage und umgekehrt proportional zu ihrer Länge ist. Die i-te Punktmasse wird also von ihrem linken Nachbarn mit der Kraft

$$f_{i,-}(t) := \frac{c}{h} \begin{pmatrix} -h \\ x_{i-1}(t) - x_i(t) \end{pmatrix}$$

angezogen und von ihrem rechten mit der Kraft

$$f_{i,+}(t) := \frac{c}{h} \begin{pmatrix} h \\ x_{i+1}(t) - x_i(t) \end{pmatrix}.$$

In horizontaler Richtung (der ersten Komponente der Vektoren) heben sich die beiden Kräfte gerade auf, in vertikaler Richtung wirkt die summierte Kraft

$$f_i(t) := \frac{c}{h}(x_{i-1}(t) + x_{i+1}(t) - 2x_i(t)).$$

Nach dem zweiten Axiom der Festkörpermechanik von Newton bewirkt eine Kraft eine Veränderung der Geschwindigkeit. Mathematisch wird diese Veränderung durch die *Ableitungen* $v_i'(t)$ der Funktionen v_i nach der Zeit t beschrieben, so dass wir

$$v_i'(t) = \frac{1}{m_i} f_i(t)$$

erhalten. Hier gibt $m_i \in \mathbb{R}_{>0}$ die Masse des i-ten Punkts an. Die i-te Punktmasse steht stellvertretend für einen Abschnitt der Saite der Länge h, also liegt es nahe, ihre Masse als proportional zu dieser Länge anzusetzen, also als

$$m_i = \varrho h,$$

so dass wir insgesamt

$$v_i'(t) = \frac{c}{\varrho} \frac{x_{i-1}(t) + x_{i+1}(t) - 2x_i(t)}{h^2} \qquad \text{für alle } i \in \{1, \dots, n\},\ t \in \mathbb{R} \tag{2.1a}$$

erhalten. Nach dem ersten Axiom der Festkörpermechanik sind die Geschwindigkeiten die Ableitungen $x_i'(t)$ der Positionen x_i nach der Zeit t, es ergibt sich

$$x_i'(t) = v_i(t) \qquad \text{für alle } i \in \{1, \dots, n\}. \tag{2.1b}$$

Insgesamt bilden die Gleichungen (2.1a) und (2.1b) ein Differentialgleichungssystem, das die zeitliche Entwicklung der Saite beschreibt. Beide Gleichungen sind aneinander gekoppelt: Die Ableitung der Geschwindigkeit hängt von der Auslenkung ab, die Ableitung der Auslenkung von der Geschwindigkeit.

Approximation der Ableitungen

Um diese Gleichungen als Ausgangspunkt einer numerischen Simulation verwenden zu können, müssen wir uns den Begriff der Ableitung etwas genauer anschauen. Eine Funktion $f : [a, b] \to \mathbb{R}$ heißt *differenzierbar* in einem Punkt $t \in (a, b)$, falls der Grenzwert

$$f'(t) = \lim_{\delta \to 0} \frac{f(t + \delta/2) - f(t - \delta/2)}{\delta}$$

existiert. In diesem Fall nennt man ihn die Ableitung von f in t.

Wir approximieren den Grenzübergang, indem wir einfach ein „hinreichend kleines“ $\delta \in \mathbb{R}_{>0}$ auf der rechten Seite der Definition einsetzen. Dadurch führen wir eine weitere Näherung ein, denn wir haben ja auch schon bei der Herleitung des mathematischen Modells die kontinuierliche Saite durch „hinreichend kleine“ Punktmassen angenähert.

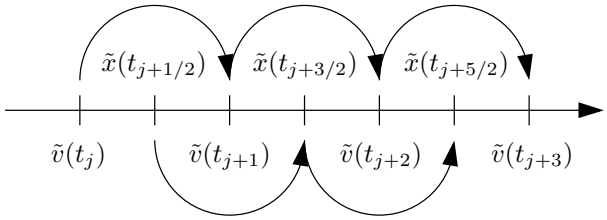

Abbildung 2.3: Vorgehensweise des *leapfrog*-Verfahrens

Aus den Gleichungen (2.1a) und (2.1b) ergibt sich so

$$\frac{v_i(t+\delta/2) - v_i(t-\delta/2)}{\delta} \approx \frac{c}{\varrho} \frac{x_{i-1}(t) + x_{i+1}(t) - 2x_i(t)}{h^2}, \tag{2.2a}$$

$$\frac{x_i(t+\delta/2) - x_i(t-\delta/2)}{\delta} \approx v_i(t) \qquad \text{für alle } i \in \{1,\dots,n\},\ t \in \mathbb{R}. \tag{2.2b}$$

Die Idee des *leapfrog-Verfahrens* besteht darin, die Geschwindigkeiten zu diskreten Zeitpunkten

$$t_k := k\delta \qquad \text{für alle } k \in \mathbb{N}_0$$

zu approximieren, während die Auslenkungen zu den dazwischen liegenden Zeitpunkten

$$t_{k+1/2} := (k+1/2)\delta \qquad \text{für alle } k \in \mathbb{N}_0$$

berechnet werden. Indem wir (2.2a) auf $t = t_{k+1/2}$ anwenden erhalten wir

$$\begin{aligned} \frac{v_i(t_{k+1}) - v_i(t_k)}{\delta} &\approx c\frac{x_{i-1}(t_{k+1/2}) + x_{i+1}(t_{k+1/2}) - 2x_i(t_{k+1/2})}{m_i}, \\ v_i(t_{k+1}) &\approx v_i(t_k) + \delta\frac{c}{\varrho}\frac{x_{i-1}(t_{k+1/2}) + x_{i+1}(t_{k+1/2}) - 2x_i(t_{k+1/2})}{h^2}, \end{aligned} \tag{2.3a}$$

während wir aus (2.2b) für $t = t_{k+1}$ die Beziehung

$$\begin{aligned} \frac{x_i(t_{k+3/2}) - x_i(t_{k+1/2})}{\delta} &\approx v_i(t_{k+1}), \\ x_i(t_{k+3/2}) &\approx x_i(t_{k+1/2}) + \delta v_i(t_{k+1}) \end{aligned} \tag{2.3b}$$

erhalten, bei der wir die soeben für t_{k+1} berechneten Geschwindigkeiten einsetzen können. Mit Hilfe von $x_i(t_{k+3/2})$ lässt sich wiederum $v_i(t_{k+2})$ berechnen und so die Simulation beliebig lange fortsetzen.

Wir verwenden die Gleichungen (2.3a) und (2.3b), um Näherungslösungen $\tilde{x}_i(t_{k+1/2})$ und $\tilde{v}_i(t_k)$ für alle $k \in \mathbb{N}_0$ zu definieren:

$$\tilde{v}_i(t_{k+1}) := \tilde{v}_i(t_k) + \delta\frac{c}{\varrho}\frac{\tilde{x}_{i-1}(t_{k+1/2}) + \tilde{x}_{i+1}(t_{k+1/2}) - 2\tilde{x}_i(t_{k+1/2})}{h^2}, \tag{2.4a}$$

$$\tilde{x}_i(t_{k+3/2}) := \tilde{x}_i(t_{k+1/2}) + \delta\tilde{v}_i(t_{k+1}) \qquad \text{für alle } i \in \{1,\dots,n\},\ k \in \mathbb{N}_0. \tag{2.4b}$$

Implementierung

Eine besonders nützliche Eigenschaft der Gleichungen (2.4) besteht darin, dass wir bei der Berechnung von $\tilde{v}_i(t_{k+1})$ die alten Werte $\tilde{v}_i(t_k)$ unmittelbar mit den neuen Werten überschreiben können, da sie für spätere Rechnungen nicht mehr benötigt werden. Entsprechend können wir mit $\tilde{x}_i(t_{k+3/2})$ die alten Werte $\tilde{x}_i(t_{k+1/2})$ überschreiben, so dass unser Algorithmus lediglich die beiden Vektoren $\tilde{v} = (\tilde{v}_i)_{i=1}^n$ und $\tilde{x} = (\tilde{x}_i)_{i=1}^n$ abzuspeichern braucht und besonders effizient und einfach zu implementieren ist.

Damit erhalten wir den folgenden Algorithmus:

procedure wave_leapfrog(**var** x, v);
for $i \in \{1, \ldots, n\}$ **do begin**
  $d_i \leftarrow (x_{i-1} + x_{i+1} - 2x_i)/h^2$;
  $v_i \leftarrow v_i + d_i \delta c/\rho$
end;
for $i \in \{1, \ldots, n\}$ **do**
  $x_i \leftarrow x_i + \delta v_i$
end

Der Algorithmus überschreibt die zu den Zeitpunkten t_k und $t_{k+1/2}$ gehörenden Geschwindigkeits- und Auslenkungswerte mit denen für die Zeitpunkte t_{k+1} und $t_{k+3/2}$. Durch wiederholte Anwendung des Algorithmus können wir die schwingende Saite über beliebig lange Zeiträume simulieren.

Dabei ist eine Besonderheit zu berücksichtigen: Bei der Berechnung der Geschwindigkeit v_i gehen nur die Auslenkungen in x_{i-1}, x_i und x_{i+1} ein, also in Punkten auf der Saite, die höchstens einen horizontalen Abstand von h aufweisen. Das bedeutet, dass sich eine Welle in unserem Modell in einem Zeitschritt höchstens die Strecke h bewegen kann. Bei einem Zeitschritt δ bedeutet das, dass wir nur Wellen mit einer Höchstgeschwindigkeit von h/δ simulieren können. Man kann nachrechnen, dass die Wellengeschwindigkeit in unserem Modellproblem $\sqrt{c/\varrho}$ beträgt, also von der Elastizität und der Dichte der Saite abhängt. Damit die Simulation funktioniert, muss also mindestens

$$\sqrt{c/\varrho} \leq h/\delta$$

gelten. Diese Ungleichung ist unter dem Namen *Courant-Friedrichs-Lewy-Bedingung* (CFL-Bedingung) bekannt und bedeutet, dass wir bei einer Verbesserung der räumlichen Auflösung, also bei einer Reduktion der räumlichen Schrittweite h, auch eine Verbesserung der zeitlichen Auflösung, also der Zeitschrittweite δ, vornehmen müssen. Dadurch wächst der Rechenaufwand relativ schnell, wenn wir die räumliche Auflösung verbessern.

Mehrdimensionale Wellen

In der Praxis sind sich eindimensional ausbreitende Wellen eher selten interessant, wesentlich häufiger interessiert man sich für Wellen, die sich im dreidimensionalen Raum

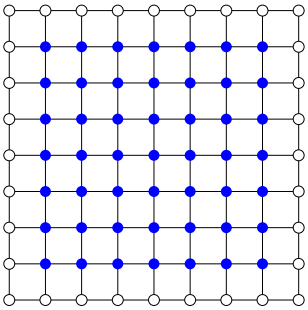

Abbildung 2.4: Punktmassen in einem regelmäßigen zweidimensionalen Gitter

ausbreiten. In bestimmten Situationen, beispielsweise für symmetrische Geometrien, sind auch zweidimensionale Wellen von Interesse, Erfreulicherweise lässt sich das von uns entwickelte Verfahren sehr leicht an diese Situationen anpassen: Statt in einer Linie ordnen wir die Punktmassen in einem zwei- oder dreidimensionalen Gitter an, an die Stelle des Index i treten Paare $(i_x, i_y) \in \{0, \ldots, n+1\}^2$ oder Tripel $(i_x, i_y, i_z) \in \{0, \ldots, n+1\}^3$ von Indizes.

Der Einfachheit halber soll hier nur die Verallgemeinerung auf den zweidimensionalen Fall diskutiert werden. Wir ordnen die Punktmassen in einem regelmäßigen zweidimensionalen Gitter (siehe Abbildung 2.4) an, bei dem die Punktmasse in der Zeile i_x und der Spalte i_y mit ihren oberen, unteren, linken und rechten Nachbarn durch eine Feder verbunden ist. Ihre Auslenkung nach vorne, also aus der Zeichenebene heraus in Richtung des Betrachters, bezeichnen wir mit x_{i_x,i_y}, ihre Geschwindigkeit in dieser Richtung mit v_{i_x,i_y}.

Wie im eindimensionalen Fall können wir die Kräfte berechnen, die vier Federn ausüben, daraus die Beschleunigung ermitteln und schließlich mit dem *leapfrog*-Ansatz die Formeln

$$\begin{aligned}\tilde{v}_{i_x,i_y}(t_{j+1}) := \tilde{v}_{i_x,i_y}(t_{j+1}) + \delta\frac{c}{\rho}\Bigg(&\frac{\tilde{x}_{i_x-1,i_y}(t_{j+1/2}) + \tilde{x}_{i_x+1,i_y}(t_{j+1/2}) - 2\tilde{x}_{i_x,i_y}(t_{j+1/2})}{h^2} \\ &+\frac{\tilde{x}_{i_x,i_y-1}(t_{j+1/2}) + \tilde{x}_{i_x,i_y+1}(t_{j+1/2}) - 2\tilde{x}_{i_x,i_y}(t_{j+1/2})}{h^2}\Bigg), \\ \tilde{x}_{i_x,i_y}(t_{j+3/2}) := \tilde{x}_{i_x,i_y}(t_{j+1/2}) + \delta\tilde{v}_{i_x,i_y}(t_{j+1}) \qquad &\text{für alle } (i_x, i_y) \in \{1, \ldots, n\}^2, j \in \mathbb{N}_0\end{aligned}$$

erhalten, mit denen wir die Ausbreitung von Wellen im Zweidimensionalen simulieren können. Für den dreidimensionalen Fall kommt entsprechend ein weiterer Bruch bei der Berechnung von $\tilde{v}_{i_x,i_y}(t_{j+1})$ hinzu, in dem die Federn vor und hinter der Punktmasse berücksichtigt werden.

3 Speicher

Hochleistungsrechner finden in der Regel bei Aufgaben Anwendung, bei denen große Datenmengen zu verarbeiten sind, beispielsweise um eine Simulation mit einer hohen Genauigkeit durchzuführen. Damit die Daten effizient zu dem Prozessor gelangen, empfiehlt es sich, die Kommunikation zwischen Prozessor und Hauptspeicher etwas genauer zu untersuchen.

3.1 Logischer und physischer Speicher

Prozesse. Wenn wir ein Programm starten, legt das Betriebssystem unseres Rechners eine Datenstruktur an, die den aktuellen Zustand des Programms beschreibt, also beispielsweise die Adresse des aktuell ausgeführten Befehls und den Inhalt des Speichers. Diese Datenstruktur nennt man einen *Prozess*.

Damit Prozesse sich nicht gegenseitig stören können, sorgt das Betriebssystem dafür, dass jeder Prozess in einem eigenen Speicherbereich arbeitet, der sich mit denen der anderen Prozesse (bis auf wenige Ausnahmen) nicht überschneidet.

Intern wird dieser Mechanismus umgesetzt, indem zwischen *logischen* und *physischen* Adressen unterschieden wird: Mit den logischen Adressen arbeitet der Prozess, die physischen Adressen geben Orte im tatsächlich vorhandenen Speicher an. Der Prozessor enthält Tabellen, die die Zuordnung von logischen zu physischen Adressen beschreiben und die durch das Betriebssystem verwaltet werden.

MMU. Da die Umrechnung von logischen zu physischen Adressen im Prozessor durch eine separate Funktionseinheit (die *memory mapping unit*, MMU) erfolgt, entsteht im Normalfall keine spürbare Verzögerung.

Es gibt allerdings zwei wichtige Sonderfälle, die bei ungeschickter Programmierung die Effizienz eines Programms erheblich verschlechtern können:

- In vielen modernen Betriebssystemen erfolgt die Zuordnung von logischen zu physischen Adressen nicht in dem Moment, in dem der Speicher durch das Programm angefordert wird, sondern erst in dem Moment, in dem zum ersten Mal auf den Speicher zugegriffen wird. Ein einfacher Lese- oder Schreibzugriff kann deshalb dazu führen, dass das Betriebssystem allerhand Datenstrukturen aktualisieren muss und so eine unerwartete Verzögerung auftritt.

- Typische Betriebssysteme stellen Prozessen soviel *logischen* Speicher zur Verfügung, wie sie haben wollen, auch falls nur wesentlich weniger physischer Speicher

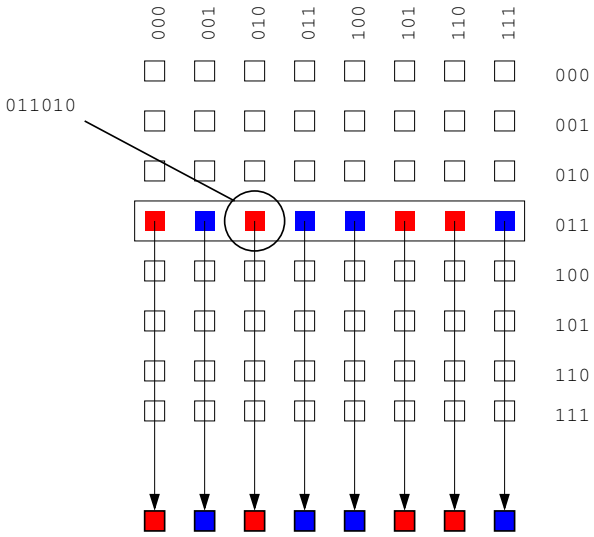

Abbildung 3.1: Zweidimensionale Organisation des Speichers: Die binäre Adresse 011010 entspricht der Zeile 011 und der Spalte 010. Bei einem Lesezugriff wird die gesamte Zeile in einen Hilfsspeicher übertragen.

vorhanden ist. Der zusätzliche Speicher wird dann simuliert, indem Speicherbereiche beispielsweise auf Festplatten ausgelagert werden (engl. *swapping*). Auch in dieser Situation kann ein Lese- oder Schreibzugriff deshalb dazu führen, dass das Betriebssystem Daten von der (langsamen) Festplatte beschaffen muss und wieder eine unerwartete Verzögerung auftritt.

3.2 Zweidimensionaler Aufbau des Speichers

Auch wenn unsere Programme ausschließlich mit physischen Adressen arbeiten würden, würden wir feststellen, dass von der Programmlogik her ununterscheidbare Speicherzugriffe unterschiedlich schnell erfolgen. Eine Ursache dieses Effekts ist der Aufbau des Hauptspeichers: Ein einzelner Speicherbaustein besteht nicht aus einer eindimensionalen Folge von Speicherzellen, die jeweils ein Bit aufnehmen, sondern aus einem zweidimensionalen Gitter. Die Adresse wird in zwei Teile zerlegt, die die Zeile und die Spalte in diesem Gitter angeben, unter der das zu dieser Adresse gehörende Bit gespeichert ist.

Zeilen- und Spaltenadresse. Dabei geben typischerweise die niedrigen Bits der Adresse die Spalte an, die höheren die Zeile, so dass aufeinanderfolgende Adressen meistens in derselben Zeile liegen.

Bei einem Lesezugriff wird immer eine *gesamte Zeile* des Speichers gelesen, also aufgrund der Interpretation der Adresse ein ganzer Satz von oft einigen tausend Bits, die zu aufeinanderfolgenden Adressen gehören. Diese Zeile wird in einem Zwischenspeicher gelagert, aus dem dann dem Prozessor die Bits geschickt werden, die er angefordert hat.

Das Kopieren einer Speicherzeile in den Zwischenspeicher kostet Zeit, deshalb kann der Speicherbaustein die Anforderungen des Prozessors wesentlich schneller erfüllen, wenn aufeinander folgende Adressen angefordert werden. Die typischen Protokolle für den Datenaustausch sehen für diesen Spezialfall den sogenannten *burst mode* vor, bei dem eine Reihe aufeinander folgender Adressen besonders effizient abgearbeitet werden. Bei Programmen, die große Datenmengen verarbeiten, empfiehlt es sich deshalb, die Speicherzugriffe so zu organisieren, dass die Adressen möglichst nahe beieinander liegen.

Beispiel: Wellengleichung. Als Beispiel können wir die zweidimensionale Wellengleichung untersuchen: Jeder Punktmasse haben wir zwei Indizes $i_x, i_y \in \{0, \ldots, n+1\}$ zugeordnet die ihre Position entlang der x- und der y-Achse beschreiben. Die Einträge x_{i_x,i_y} und v_{i_x,i_y} speichern wir in eindimensionalen Arrays `xa` und `va` ab, die wir gemäß

$$x_{i_x,i_y} = \texttt{xa[ix+iy*ld]}, \qquad v_{i_x,i_y} = \texttt{va[ix+iy*ld]}, \qquad \texttt{ld} = n+2$$

numerieren. Dadurch sind alle Punktmassen mit derselben y-Koordinate fortlaufend numeriert. Um zu dem rechten Nachbarn zu gelangen, addieren wir `1` zu dem Index, den oberen Nachbarn erreichen wir, indem wir den Wert `ld` (engl. für *leading dimension*) addieren.

Einen Schritt des *leapfrog*-Verfahrens können wir nun durchführen, indem wir die Arrays zeilen- oder spaltenweise durchlaufen. In einem Experiment mit einem Prozessor des Typs Intel Core i7-920 zeigte sich, dass 1 000 Zeitschritte für $n = 4\,000$ bei spaltenweise Anordung gut 352 Sekunden benötigen, während bei zeilenweiser Anordnung knapp 99 Sekunden ausreichen. Die richtige Anordnung der Speicherzugriffe kann also die Laufzeit drastisch verkürzen.

3.3 Caching

Viele Programme verbringen den größten Teil ihrer Laufzeit damit, relativ häufig auf relativ wenige Variablen zuzugreifen. Da der Zugriff auf den Hauptspeicher relativ langsam verglichen mit der Taktfrequenz eines modernen Prozessors ist, gehört zu modernen Prozessoren mindestens ein kleiner und sehr schneller Hilfsspeicher, der *Cache* (im Englischen bedeutet *cache* ursprünglich soviel wie „verstecktes Lager“), der Kopien besonders häufig benötigter Teile des Hauptspeichers aufnehmen kann.

Cache-Hierarchie. Während der Hauptspeicher typischerweise so konstruiert wird, dass eine möglichst hohe Kapazität bei möglichst geringen Kosten erreicht wird, sind Cache-Speicher so gebaut, dass sie eine besonders hohe Geschwindigkeit erreichen. Um einen möglichst guten Kompromiss zwischen Geschwindigkeit und Kapazität zu erreichen, verwenden moderne Prozessoren häufig mehrere Caches unterschiedlicher Geschwindigkeit und Größe: Der *first level cache* ist sehr schnell, aber auch relativ klein. Der *second level cache* ist etwas langsamer, aber erheblich größer. Bei aufwendigen Mehrkernprozessoren kommt gelegentlich auch ein *third level cache* zum Einsatz, den sich mehrere Kerne teilen.

Im Normalfall ist diese Cache-Hierarchie für den Programmierer unsichtbar, denn der Prozessor prüft bei einem Speicherzugriff automatisch, ob die gewünschten Daten in einem Cache vorliegen, um entweder die gefundenen Daten schnell zurück zu geben oder einen Zugriff auf den Hauptspeicher in die Wege zu leiten. Das Ergebnis ist in beiden Fällen dasselbe.

Cache hit / Cache miss. Allerdings wird sich der Zeitaufwand in der Regel deutlich unterscheiden: Bei einem *cache hit*, falls also die Daten aus dem Cache gelesen werden können, vergeht wesentlich weniger Zeit als bei einem *cache miss*, bei dem sie aus dem Hauptspeicher zu holen sind.

Wir haben bereits gesehen, dass der Zugriff auf aufeinander folgende Speicheradressen besonders effizient ist. Deshalb wird bei einem *cache miss* nicht nur der Inhalt einer einzigen Speicheradresse in den Cache übertragen, sondern ein etwas größerer Block, eine sogenannte *cache line*. Ein typisches Beispiel ist eine Organisation des Caches in *cache lines* von 64 Bytes, die grundsätzlich an durch 64 teilbaren Adressen beginnen. Bei einem Zugriff auf die Adresse 95 werden dann die Inhalte der Adressen 64 bis 127 in eine *cache line* übertragen.

Um eine hohe Effizienz unseres Programms zu erreichen, sollten wir also darauf achten, einmal in den Cache übertragene Daten möglichst häufig zu verwenden, bevor sie durch weitere Speicherzugriffe verdrängt werden.

Cachelines. Wie bereits erwähnt übertragen moderne Prozessoren in der Regel eine vollständige *cache line* aus dem Hauptspeicher in den Cache. Selbst wenn nur ein einziges Byte benötigt wird, werden also beispielsweise 32 oder 64 Bytes übertragen.

Deshalb empfiehlt es sich, die für eine Phase einer Berechnung benötigten Daten möglichst in eng benachbarten Speicheradressen unterzubringen, um sicherzustellen, dass möglichst wenige *cache lines* übertragen werden müssen.

In der Programmiersprache C kann beispielsweise das Schlüsselwort `struct` verwendet werden, um Daten zu größeren Blöcken zusammenzusetzen, die hoffentlich in wenige direkt benachbarte *cache lines* passen.

Bei der PageRank-Berechnung können in dieser Weise die Spaltenindizes und die zu ihnen gehörenden Koeffizienten $1/d_j$ zu einem `struct` zusammengefasst statt in separaten Arrays untergebracht werden. Dadurch sinkt die Laufzeit für 100 Schritte des Verfahrens bei einem zufälligen Graphen mit 500 000 Knoten und gut 25 000 000 Kanten auf einem Intel Core i7-920 von 18 auf 10 Sekunden.

Alignment. Allerdings müssen wir bei der Implementierung beachten, dass auch eine Datenstruktur, die im Prinzip in eine *cache line* passt, das Lesen von zwei *cache lines* erforderlich machen könnte, falls sie an einer ungünstigen Adresse beginnt. So würde etwa eine Datenstruktur von 48 Bytes Größe in eine 64 Bytes umfassende *cache line* passen, falls sie aber beispielsweise an der Adresse 32 beginnt, würden ihre ersten 32 Bytes in der an der Adresse 0 beginnenden *cache line* liegen, während die restlichen 16

Bytes in der *cache line* ab Adresse 64 zu finden wären. Der Prozessor müsste also beide *cache lines* aus dem Hauptspeicher beziehen.

Wir können diesen Effekt vermeiden, indem wir dafür sorgen, dass die Datenstrukturen an Adresse beginnen, die Vielfache der Länge einer *cache line* sind. In der Programmiersprache C gehört zu jedem Datentyp auch eine Angabe darüber, an welchen Speicheradressen er ausgerichtet sein sollte (engl. *alignment*), beispielsweise wird eine `double`-Variable in der Regel an einer durch 8 teilbaren Adresse beginnen.

Diese Ausrichtung wird üblicherweise durch das Einfügen unbenutzter Bytes sichergestellt, kann also den Speicherbedarf eines Programms deutlich wachsen lassen.

Bemerkung 3.1 (Alignment) *Falls wir den C-Compiler der* GNU Compiler Collection *verwenden, können wir der Definition eines Typs oder einer Variable mit Hilfe des Schlüsselworts* `__attribute__` *eine Reihe von Attributen zuweisen. Eines dieser Attribute ist* `aligned`*, mit dem sich die Ausrichtung einer einzelnen Variablen oder eines Typs festlegen lässt:*

```
typedef int __attribute__((aligned(8))) aint;
```

definiert beispielsweise einen neuen Typ `aint`*, der* `int`*-Variablen aufnimmt, die an einer durch 8 teilbaren Adresse stehen müssen.*

Microsoft-Compiler bieten für einen ähnlichen Zweck das Schlüsselwort `__declspec`*, mit dem sich das Attribut* `align` *zuweisen lässt:*

```
typedef __declspec(align(8)) int aint;
```

Es kann zu Fehlermeldungen kommen, falls man diese modifizierten Datentypen in einem Array verwendet, weil bei Arrays davon ausgegangen wird, dass die Daten nach ihrer tatsächlichen Größe ausgerichtet sind, nicht nach dem von uns separat zugewiesenen Attribut.

Da der Cache wesentlich kleiner als der Hauptspeicher ist, muss festgelegt werden, wie Adressen der beiden Speicher einander zugeordnet werden.

Direct-mapped Cache. Die einfachste Form eines Caches ist der *direct-mapped cache*, bei dem aus der Adresse a im Hauptspeicher die Adresse im Cache berechnet wird, indem man durch die Größe c des Caches dividiert und den Rest $b = a \mod c$ als Cache-Adresse verwendet. Da die Cache-Größe in der Regel eine Zweierpotenz ist, lässt sich die Division sehr einfach umsetzen, indem man in a eine Reihe der höchsten Bits auf null setzt. Diese höchsten Bits muss sich der Prozessor natürlich merken, damit er weiß, zu welcher Adresse im Hauptspeicher die einzelnen Adressen im Cache-Speicher gehören.

Trashing. Bei ungeschickter Programmierung kann es zu einem als *trashing* bezeichneten Effekt kommen: Falls ein Algorithmus zwei Variablen x und y verwendet, die an den Adressen 1024 und 2048 liegen, und falls ein *direct-mapped cache* mit einer Größe von

1024 Bytes zum Einsatz kommt, liegen beide Variablen im Cache-Speicher auf derselben Adresse, nämlich 0. Falls der Algorithmus abwechselnd auf x und y zugreift, wird dann jedesmal der falsche Wert im Cache stehen, so dass jedesmal ein *cache miss* eintritt und der langsame Hauptspeicher bemüht werden muss.

Assoziativer Cache. Dieses Problem wird bei modernen Prozessoren umgangen, indem man für jede Adresse im Hauptspeicher *mehrere* mögliche Adressen im Cache-Speicher vorsieht. In unserem Beispiel würden dann x und y an unterschiedlichen Orten im Cache-Speicher liegen, so dass der Algorithmus so schnell wie erwartet arbeiten kann. Nach diesem Prinzip aufgebaute Caches bezeichnet man als *assoziative Caches*, genauer gesagt spricht man von einem *n-fach assoziativen Cache*, falls einer Adresse im Hauptspeicher n Adressen im Cache-Speicher zugeordnet sind.

Um den Einfluss von Speicherzugriffen auf die Geschwindigkeit eines Programms abzuschätzen, sind zwei Parameter von Interesse:

Latenz. Die *Latenz* gibt die Zeit an, die von der Anforderung der Daten aus dem Speicher bis zu deren Verfügbarkeit vergeht. Moderne Prozessoren beherrschen *out-of-order execution*, sind also dazu in der Lage, die auszuführenden Befehle so umzusortieren, dass während der Wartezeit Befehle ausgeführt werden können, die nicht von den Daten abhängen, auf die gerade gewartet wird.

Der Compiler kann den Prozessor unterstützen, indem er beispielsweise Schleifen teilweise expandiert (engl. *loop unrolling*), indem der Rumpf einige Mal kopiert wird: Aus

```
for(i=0; i<n; i++) {
  x = a[i];
  b[i] += x;
}
```

könnte man beispielsweise äquivalent

```
for(i=0; i+1<n; i+=2) {
  x1 = a[i];
  b[i] += x1;
  x2 = a[i+1];
  b[i+1] += x2;
}
for(; i<n; i++) {
  x = a[i];
  b[i] += x;
}
```

machen, und dank *out-of-order execution* könnte der Prozessor die Zeit, in der er auf `a[i]` wartet, dazu nutzen, um schon `a[i+1]` anzufordern, da er dafür nicht das Ergebnis der ersten Operation zu kennen braucht. Während auf `a[i+1]` gewartet wird, könnte wiederum schon `b[i]` aktualisiert werden. Je weiter man die Schleife expandiert, desto

mehr Möglichkeiten zur Vermeidung von Latenzen bieten sich dem Prozessor, allerdings wächst natürlich auch die Länge des Programms.

Latenzen lassen sich also gut „verstecken“, indem die Wartezeit zwischen der Anforderung von Daten und deren Verwendung für andere Rechnungen genutzt wird.

Durchsatz. Die zweite für Speicherzugriffe relevante Größe ist der *Durchsatz*. Er beschreibt die Geschwindigkeit, mit der Daten unter optimalen Bedingungen aus dem Speicher zu dem Prozessor transportiert werden können. Der Durchsatz lässt sich nicht verbessern: Wenn unser Algorithmus 32 GB an Daten verarbeiten muss und der Prozessor einen Durchsatz von 16 GB/s bietet, wird auch eine perfekte Implementierung nicht mit weniger als 2 Sekunden Laufzeit auskommen.

Caches können uns lediglich dabei helfen, die Anzahl der Zugriffe auf den Hauptspeicher, der einen geringen Durchsatz erreicht, zu reduzieren, indem wir sie durch Zugriffe auf die Caches ersetzen, die einen deutlich höheren Durchsatz erreichen.

Prefetching. Falls ein Algorithmus in einem für den Prozessor nicht vorhersagbaren Muster auf den Speicher zugreift, ist die Wahrscheinlichkeit eines *cache miss* relativ hoch. Allerdings bieten moderne Prozessoren uns die Möglichkeit, mit speziellen Befehlen darauf hinzuweisen, dass wir demnächst beabsichtigen, eine bestimmte Speicheradresse zu verwenden. Ein Teil des Prozessors kann sich dann damit beschäftigen, den Inhalt dieser Adresse in den Cache zu übertragen, während der Rest unseren Algorithmus ausführt. Diesen Mechanismus bezeichnet man als *Prefetching*.

Falls der Prozessor mehrere Caches verwendet, können wir teilweise sogar festlegen, in welche Caches die Daten aus dem Hauptspeicher übernommen werden sollen. Das ist sehr praktisch, wenn wir wissen, dass wir die zu lesenden Daten nur genau einmal verwenden werden, denn dann können wir in dieser Weise vermeiden, dass sie andere Daten aus dem Cache verdrängen.

Im Gegensatz zu den vorigen Bemerkungen muss das Prefetching in der Regel durch Maschinenbefehle explizit angefordert werden, entweder durch einen Compiler, der die entsprechenden Anweisungen selbständig in das Maschinenprogramm einfügt, oder durch spezielle Befehle, die wir in unseren Quelltext aufnehmen. Letzten Endes bietet Prefetching uns eine Möglichkeit, die Latenz von Speicherzugriffen zu reduzieren, weil wir „manuell“ den Speicherzugriff in die Wege leiten können, ohne uns darauf verlassen zu müssen, dass der Prozessor durch Umsortieren einen ähnlichen Effekt erreicht.

Bemerkung 3.2 (Prefetching) *Der C-Compiler der* GNU Compiler Collection *bietet den Befehl* `__builtin_prefetch`*, mit dem wir dem Prozessor mitteilen können, welche* cache lines *er demnächst voraussichtlich benötigen wird und wofür wir sie verwenden wollen. Der Befehl kann mit einem, zwei oder drei Parametern aufgerufen werden: Der erste ist ein Zeiger auf die Adresse, die aus dem Hauptspeicher bezogen werden soll, der zweite gibt an, ob wir auf diese Adresse schreibend oder lesend zuzugreifen beabsichtigen, und der dritte, für wie sinnvoll wir es halten, die betreffende* cache line *etwas länger aufzubewahren, weil noch weitere Zugriffe auf sie zu erwarten sind.*

Andere Compiler bieten mit dem Befehl `_mm_prefetch` *eine vergleichbare Funktion.*

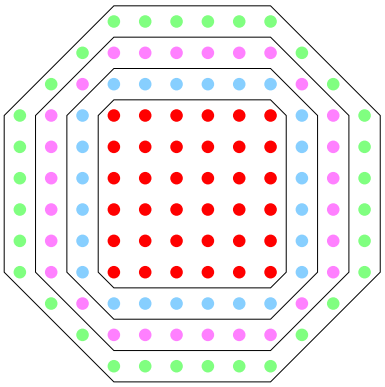

Abbildung 3.2: Teilgebiet mit den für die Durchführung von einem, zwei und drei Zeitschritten erforderlichen Nachbarn.

Bei Prozessoren ohne *out-of-order execution* kann Prefetching erhebliche Geschwindigkeitsgewinne bringen: Auf einem Intel Atom N450 beispielsweise benötigen 100 Schritte des PageRank-Algorithmus für 100 000 Knoten ohne Prefetching mehr als 110 Sekunden, während mit Prefetching knapp 28 Sekunden genügen. Bei Prozessoren mit *out-of-order execution* dagegen zeigt sich kein erwähnenswerter Gewinn, die Verarbeitung der Prefetch-Befehle kann sogar zu einer geringfügig längeren Laufzeit führen.

Beispiel: Wellengleichung. Bei der Behandlung der zweidimensionalen Wellengleichung lässt sich durch geschickte Ausnutzung des Caches die Geschwindigkeit erheblich verbessern: Im Laufe eines Zeitschritts wird jeder Eintrag des Arrays `xa` ungefähr fünfmal verwendet, nämlich für die Berechnung der korrespondierenden Punktmasse und ihrer vier Nachbarn. Falls die Gitterzeilen etwas länger sind, kann es allerdings passieren, dass der Eintrag `xa[ix+iy*ld]` aus dem Cache verdrängt wurde, wenn er für die Berechnung von `va[ix+(iy+1)*ld]` wieder benötigt wird. Im ungünstigsten Fall ist der Cache dann völlig nutzlos.

Um dieses Problem zu vermeiden, können wir das Gitter in kleinere rechteckige Teile zerlegen und unsere Berechnungen so anordnen, dass jeweils alle Einträge innerhalb eines dieser Teilgitter nacheinander berechnet werden. Falls die Daten für ein Teilgitter in den Cache passen, muss fast jeder Eintrag der Arrays `xa` und `va` nur einmal gelesen werden, lediglich die Nachbarn der Randpunkte des Teilgitters werden von mehreren Teilgittern verwendet und müssen eventuell häufiger im Hauptspeicher angefordert werden.

Wenn wir diese — an sich gute — Idee unmittelbar umsetzen, wird der Geschwindigkeitsgewinn auf den meisten Rechnern allerdings relativ gering sein, teilweise kann die Effizienz sogar sinken: Die Caches sind — zumindest bei moderaten Problemdimensionen — groß genug, um die vorige und die nächste Zeile aufzunehmen, so dass wir aus dem zusätzlichen Verwaltungsaufwand kaum Gewinn ziehen.

Patchweises Vorgehen. Die Situation ändert sich, wenn wir auf den Teilgebieten nicht nur einen Zeitschritt durchführen, sondern mehrere: Wenn wir das Ergebnis des k-ten Zeitschritts für ein Indexpaar (i_x, i_y) berechnen wollen, benötigen wir die Auslenkungen und Geschwindigkeiten im $(k-1)$-ten Zeitschritt für dieses Indexpaar *und seine Nachbarn*. Um also k Zeitschritte auf einmal durchzuführen, benötigen wir die Ausgangsdaten für (i_x, i_y) und die „k-te Generation von Nachbarn“. Ein Beispiel für $k = 3$ ist in Abbildung 3.2 dargestellt.

Bei der Implementierung müssen wir darauf achten, dass für jede Punktmasse dieselbe Anzahl von Zeitschritten durchgeführt wird. Es bietet sich an, die zu berechnende Teilmenge und ihre Nachbarn in einen separaten Hilfsspeicher zu kopieren, dort die Berechnungen durchzuführen, und das Ergebnis anschließend aus den Teilergebnissen zusammenzusetzen.

Der Geschwindigkeitsgewinn ist bei dieser Vorgehensweise allerdings gering: Für $n = 4\,000$ und tausend Zeitschritte benötigt die einfache zeilenweise Vorgehensweise ungefähr 98 Sekunden, wenn wir 50 Zeitschritte auf Teilgebieten mit ungefähr 250^2 Punktmassen durchführen, erreichen wir 91 Sekunden.

3.4 Schreibzugriffe

Bisher standen Lesezugriffe im Zentrum unserer Aufmerksamkeit. Sie sind für die Ausführungsgeschwindigkeit eines Programms besonders wichtig, denn solange die nötigen Daten nicht vorliegen, können die eigentlichen Berechnungen nicht in Angriff genommen werden. Ein Schreibzugriff auf den Hauptspeicher ist dagegen häufig das Ende einer Berechnung, so dass eine Verzögerung sich weniger kritisch auf den Ablauf des Programms auswirkt.

Trotzdem können auch Schreibzugriffe störende Nebenwirkungen haben: Da viele Prozessoren grundsätzlich mit vollständigen *cache lines* operieren, kann das Schreiben eines einzelnen Bytes es erforderlich machen, eine *cache line* zu lesen, das Byte einzutragen, und die *cache line* zurück in den Hauptspeicher zu übertragen. Dadurch entstehen bei einem Schreibzugriff ein weiterer Lesezugriff, der wiederum mit anderen Lesezugriffen konkurriert. Außerdem wird der Cache auch bei einem Schreibzugriff mit Daten gefüllt, die möglicherweise für sehr lange Zeit nicht wieder benötigt werden, aber nützlichere Daten verdrängen.

Verzögertes Schreiben. Manche Prozessoren bieten deshalb die Möglichkeit, immerhin diesen letzten Effekt zu vermeiden, indem mit speziellen Befehlen die Cache-Hierarchie weitgehend umgangen und direkter in den Hauptspeicher geschrieben werden kann. In diesem Rahmen ist es auch möglich, dem Prozessor die Freiheit einzuräumen, die Schreibzugriffe so umzusortieren, dass sie besonders effizient durchgeführt werden können. An den Punkten des Programms, an denen der Speicher in einem definierten Zustand sein muss, müssen dann Befehle eingefügt werden, die sicherstellen, dass tatsächlich alle verzögerten Schreibzugriffe ausgeführt worden sind.

Diese Vorgehensweise bietet dem Prozessor die Möglichkeit, mittels *write combining*

mehrere Schreibzugriffe auf eine *cache line* zusammenzufassen und die vollständige *cache line* erst dann in den Hauptspeicher zurückzuschreiben, wenn sie auf dem endgültigen Stand ist.

4 Vektorrechnen

Die Rechenleistung eines Prozessors lässt sich besonders einfach steigern, indem die Anzahl der Rechenwerke erhöht wird. Während bei einem *superskalaren* Prozessor die anfallende Arbeit dynamisch auf die verschiedenen Rechenwerke verteilt wird, werden sie bei einem *Vektorrechner* so zusammengeschaltet, dass sie dieselbe Operation für mehrere Datensätze gleichzeitig ausführen (engl. *single instruction, multiple data*, SIMD).

4.1 Vektoren

Vektorrechner eignen sich gut für Algorithmen, bei denen exakt dieselbe Befehlsfolge für mehrere Datensätze ausgeführt werden muss. Ein Beispiel ist unsere Simulation der Wellenausbreitung, bei der für jeden Index eine Reihe von Multiplikationen, Subtraktionen und Additionen durchzuführen ist. Wichtig ist in diesem Fall, dass jedesmal dieselben Operationen ausgeführt werden müssen, allerdings mit unterschiedlichen Daten. Diese Vorgehensweise wird häufig mit der Abkürzung *SIMD* (engl. *single instruction, multiple data*) bezeichnet.

Damit ein Vektorrechner diese Struktur ausnutzen kann, muss er in die Lage versetzt werden, mit mehreren Datensätzen gleichzeitig zu arbeiten. Ein Prozessor verwendet in der Regel spezielle interne Variablen, die *Register*, um Daten aufzubewahren, auf die Rechenoperationen angewendet werden sollen. Sollen mehrere Datensätze gleichzeitig verarbeitet werden, bietet es sich also an, Register zu verwenden, die mehrere Werte desselben Typs aufnehmen können.

Vektorregister. In Anlehnung an die kartesische Geometrie, in der ein Vektor ein Tupel von Zahlen ist, bezeichnet man Tupel von Variablen desselben Typs als *Vektoren*. Entsprechend verfügt ein Vektorprozessor über *Vektorregister*, die mehrere Werte gleichzeitig annehmen können.

Vektorregister können ähnlich wie konventionelle Register verwendet werden: Sie können mit Daten aus dem Hauptspeicher gefüllt werden, ihre Daten können in den Hauptspeicher geschrieben werden, aber vor allem können wir mit ihren Daten rechnen. Ein moderner Prozessor verfügt dazu über eine Reihe von Befehlen, die die Vektorregister anstelle der konventionellen Register verwenden.

Arithmetische Vektoroperationen. Beispielsweise für die Simulation der Wellenausbreitung wären für uns die Addition und die Multiplikation von Interesse. Bei Verwendung eines Vektorrechners können wir diese Operationen beispielsweise auf Tupel

von vier Zahlen anwenden. Die Ergebnisse der Vektor-Addition $\oplus$ und der Vektor-Multiplikation $\odot$ berechnen sich dann wie folgt:

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} \oplus \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ x_3 + y_3 \\ x_4 + y_4 \end{pmatrix}, \qquad \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} \odot \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} = \begin{pmatrix} x_1 y_1 \\ x_2 y_2 \\ x_3 y_3 \\ x_4 y_4 \end{pmatrix}.$$

Der Reiz dieses Ansatzes besteht darin, dass bei vielen modernen Prozessoren die simultane Addition oder Multiplikation von vier Zahlenpaaren nicht mehr Zeit erfordert als die Addition oder Multiplikation eines einzelnen Zahlenpaars. Die Berechnung würde also bei Einsatz der Vektorregister tatsächlich viermal schneller ablaufen.

Manche Compiler, beispielsweise der C-Compiler der *GNU Compiler Collection*, können unter bestimmten Bedingungen Strukturen in einem Programm erkennen, die es erlauben, Vektorbefehle einzusetzen. In vielen Anwendungsfällen wird es allerdings erforderlich sein, den Algorithmus per Hand zu vektorisieren. Als Beispiel untersuchen wir die Linearkombination zweier Vektoren, die durch die Funktion

```
void axpy(int n, float alpha, const float *x, float *y)
{
  int i;

  for(i=0; i<n; i++)
    y[i] += alpha * x[i];
}
```

einfach realisiert werden kann.

SSE. Moderne Intel- und AMD-Prozessoren bieten eine Erweiterung namens *SSE* (engl. *streaming SIMD extension*), die es uns ermöglicht, mit 4-Tupeln des Datentyps `float` zu arbeiten, die zu einem neuen Datentyp `__m128` zusammengefasst sind und mit neuen Befehlen verarbeitet werden. Eine SSE-Version unserer Funktion könnte die folgende Form annehmen:

```
void axpy(int n, float alpha, const float *x, float y)
{
  __m128 v_alpha;
  int i;

  v_alpha = _mm_load1_ps(&alpha);

  for(i=0; i+3<n; i+=4)
    _mm_store_ps(y+i,
                 _mm_add_ps(_mm_load_ps(y+i),
                            _mm_mul_ps(v_alpha,
```

```
                                        _mm_load_ps(x+i))));

  for(; i<n; i++)
    y[i] += alpha * x[i];
}
```

Die wichtigste Änderung besteht darin, dass die zentrale Schleife jetzt mit einer Schrittweite von 4 arbeitet, entsprechend der Breite des verwendeten Vektor-Datentyps. Damit auch in dem Fall, dass n kein Vielfaches von 4 ist, das korrekte Ergebnis heraus kommt, muss noch eine Schleife ergänzt werden, die die letzten Einträge in der konventionellen Weise verarbeitet.

Innerhalb der Hauptschleife werden SSE-Befehle verwendet, um die nötigen Rechenoperationen auszuführen: `_mm_load_ps` liest vier `float`-Werte aus dem Speicher in ein Vektorregister, `_mm_add_ps` und `_mm_mul_ps` führen Additionen und Multiplikationen mit ihnen aus, und `_mm_store_ps` schreibt den Inhalt eines Vektorregisters zurück in den Speicher.

Eine Besonderheit ist die Funktion `_mm_load1_ps`, die eine `float`-Zahl in alle vier Komponenten eines Vektorregisters überträgt. Das ist in diesem Anwendungsfall sinnvoll, weil alle Komponenten mit demselben Faktor `alpha` multipliziert werden sollen.

Experiment. Auf einem Intel Core i7-920 zeigt sich, dass die ursprüngliche Fassung der Funktion für 10 000 000 Wiederholungen bei $n = 1000$ gut 6,9 Sekunden benötigt, während die SSE-Fassung mit nur 2,0 Sekunden auskommt.

Wenn wir umgekehrt 1 000 Funktionsaufrufe mit $n = 10\,000\,000$ durchführen, machen sich die Zugriffe auf den Hauptspeicher bemerkbar: Die ursprüngliche Version benötigt ungefähr 9,6 Sekunden, die SSE-Version ungefähr 8,4 Sekunden. Da die beiden Vektoren mit jeweils ungefähr 40 MB nicht in den Cache passen, müssen sie für jeden Funktionsaufruf vollständig aus dem Hauptspeicher gelesen werden, so dass für jede Rechenoperation zweimal 4 Bytes gelesen und einmal 4 Bytes geschrieben werden müssen. Insgesamt müssen also ungefähr 111 GBytes übertragen werden, so dass sich ein Durchsatz von ungefähr 13,4 GBytes/s für die SSE-Version ergibt. Das ist etwas mehr als die Hälfte des theoretischen Maximums von 25,6 GBytes/s, das dieser Prozessor unter optimalen Bedingungen bieten kann.

Integrierte Vektor-Datentypen. Der C-Compiler der *GNU Compiler Collection* bietet die Möglichkeit, Vektorregister etwas eleganter einzusetzen: Man kann Vektortypen definieren, im unten stehenden Beispiel `float4`, auf die viele der üblichen C-Operatoren direkt angewendet werden können. Darüber hinaus können einzelne Komponenten der Vektorregister wie Elemente eines konventionellen C-Arrays angesprochen werden und bestimmte binäre Operatoren können auch auf Kombinationen aus einfachen und Vektortypen angewendet werden, wie im unten stehenden Beispiel bei `alpha * xv[i]` geschehen. Diese Vorgehensweise bietet gegenüber den von Intel vorgesehenen Intrinsics den Vorteil, dass sich die so formulierten Programme auch für andere Vektoreinheiten

eignen und sogar „zu lange“ Vektoren automatisch in Teilvektoren zerlegt werden, die zu dem verwendeten Prozessor passen.

Unser Experiment kann dann in der folgenden Form umgesetzt werden:

```
typedef float float4 __attribute__((vector_size (16)));

void axpy_sse(int n, float alpha, const float *x, float *y)
{
  float4 *xv, *yv;
  int i, n4;

  xv = (float4 *) x;
  yv = (float4 *) y;

  n4 = n/4;
  for(i=0; i<n4; i++)
    yv[i] += alpha * xv[i];

  for(i*=4; i<n; i++)
    y[i] += alpha * x[i];
}
```

4.2 Speicherzugriffe

Ein SSE-Vektorregister umfasst 128 Bits, also 16 Bytes, würde also in eine *cache line* passen. Deshalb gilt auch hier das in Kapitel 3 Gesagte: Wir sollten darauf achten, dass die Daten passend im Speicher ausgerichtet sind, damit wir jeweils nur eine *cache line* zu lesen brauchen.

Malloc mit Alignment. Bei Speicherbereichen, die dynamisch mit `malloc` angefordert werde, müssten wir die korrekte Ausrichtung „per Hand“ sicherstellen, indem wir den Zeiger manipulieren, den `malloc` zurückgibt. Glücklicherweise gibt es für SSE-Programme die Funktionen `_mm_malloc` und `_mm_free`, die diese Aufgabe für uns übernehmen und uns passend ausgerichtete Speicherbereiche zur Verfügung stellen: `_mm_malloc(192,16)` beispielsweise legt einen Speicherbereich von mindestens 192 Bytes Größe an, der an einer durch 16 teilbaren Adresse im Speicher beginnt. Dieser Speicherbereich darf nicht durch die übliche `free`-Funktion freigegeben werden, stattdessen muss `_mm_free` verwendet werden.

In unserem Beispiel wurden die Funktionen `_mm_load_ps` und `_mm_store_ps` verwendet, um Daten aus dem Speicher in die Vektorregister und zurück zu übertragen. Diese Funktionen dürfen nur verwendet werden, wenn die Speicheradressen Vielfache von 16 sind, ansonsten sollten sie eine Fehlermeldung verursachen.

Um auch mit mehr oder weniger beliebig ausgerichteten Speicheradressen umgehen zu können, sieht SSE die Funktionen `_mm_loadu_ps` und `_mm_storeu_ps` vor, die auch

für Speicheradressen funktionieren, die keine Vielfachen von 16 sind. Da sich in diesem Fall ein Speicherzugriff über zwei *cache lines* erstrecken kann, sollte man in der Praxis einkalkulieren, dass sie etwas langsamer als `_mm_load_ps` und `_mm_store_ps` arbeiten könnten.

Verzögerte Schreibzugriffe. Für Schreibzugriffe auf den Speicher sieht SSE neben den Funktionen `_mm_store_ps` und `_mm_storeu_ps` auch sogenannten *non-temporal stores* vor, die mit `_mm_stream_ps` veranlasst werden. Während bei normalen Schreibzugriffen die geschriebenen Daten im Cache aufbewahrt werden, weil davon ausgegangen wird, dass sie bald wieder gelesen werden müssen, werden sie bei `_mm_stream_ps` „am Cache vorbei“ in den Hauptspeicher geschrieben, so dass sie keine wichtigeren Daten verdrängen können. Der Prozessor kann dabei *write combining* verwenden, also mehrere Schreibzugriffe in einem Hilfsspeicher, dem *line fill buffer*, sammeln, bevor tatsächlich etwas in den Speicher geschrieben wird. Dadurch können wiederholte Zugriffe auf dieselbe *cache line* vermieden werden, indem sie erst geschrieben wird, wenn sie gefüllt ist. Falls wir sicherstellen wollen, dass der Schreibzugriff tatsächlich ausgeführt wurde, können wir `_mm_sfence` verwenden, um dafür zu sorgen, dass die bisher in Auftrag gegebenen Schreibzugriffe ausgeführt werden, bevor weitere Schreibzugriffe stattfinden. Mit `_mm_mfence` können wir sicherstellen, dass alle Speicherzugriffe ausgeführt werden, bevor weitere Schreib- oder Lesezugriffe erfolgen.

Bei der Verwendung von `_mm_stream_ps` ist allerdings Vorsicht geboten: Da die Daten direkt in den Hauptspeicher geschrieben werden, kann ein Cache auch dann nicht seine Wirkung entfalten, falls alle Daten im Prinzip in ihm untergebracht werden können.

4.3 Fallunterscheidungen

Das Prinzip des Vektorrechnens besteht darin, denselben Befehl für alle Komponenten eines Vektorregisters auszuführen. Dadurch entstehen Probleme, wenn Berechnungen Fallunterscheidungen enthalten, denn dann müssten schließlich abhängig von dem zu behandelnden Fall unterschiedliche Operationen ausgeführt werden.

Sofern sich die Fälle nur durch wenige Befehle unterscheiden, können wir trotzdem von den Vektorregistern profitieren. Als Beispiel verwenden wir die Berechnung des komponentenweisen Maximums zweier Arrays:

```
void vecmax(int n, const float *x, const float *y, float *mx)
{
  int i;

  for(i=0; i<n; i++)
    if(x[i] < y[i])
      mx[i] = y[i]
    else
      mx[i] = x[i];
}
```

Da Vektorbefehle grundsätzlich auf alle Komponenten eines Vektorregisters angewendet werden, können wir die `if`-Anweisung nicht einfach nachbauen.

Stattdessen behelfen wir uns mit bitweisen logischen Operationen: Wir konstruieren eine *Bitmaske* `m`, also ein Vektorregister, in dem diejenigen Komponenten bitweise mit eins gefüllt sind, für die die `if`-Bedingung zutrifft. Bei einer bitweisen Und-Verknüpfung werden dann alle anderen Komponenten auf null gesetzt. Entsprechend können wir mit einer bitweisen Und-Verknüpfung mit dem bitweisen Komplement alle Komponenten eines Vektorregisters auf null setzen, für die die Bedingung *nicht* gilt. Durch Oder-Kombination beider Ergebnisse erhalten wir das gewünschte Resultat. Wenn wir in Standard-C ganzzahlige und Gleitkomma-Typen beliebig mit bitweisen Operatoren kombinieren dürften, würden wir ungefähr das folgende Programmfragment erhalten:

```
void vecmax(int n, const float *x, const float *y, float *mx)
{
  int i, m;

  for(i=0; i<n; i++) {
    m = (x[i] < y[i] ? ~0 : 0);
    mx[i] = (m & y[i]) | (~m & x[i]);
  }
}
```

Formal ist damit die `if`-Anweisung verschwunden, tatsächlich ist sie lediglich in der Berechnung der Bitmaske `m` versteckt.

Vektorisierte Vergleichsoperation. Glücklicherweise ignorieren die Funktionen für bitweise Operationen mit SSE-Vektorregistern die Unterschiede zwischen ganzzahligen und Gleitkomma-Zahlen, so dass wir uns zumindest um diesen Aspekt unseres Ansatzes keine Sorgen zu machen brauchen. Für die Berechnung der Bitmaske `m` können wir die Funktion `_mm_cmplt_ps` verwenden, die ohne Verzweigung genau das leistet, was wir brauchen: Sie erhält zwei Vektorregister und prüft für jedes Paar x, y von Komponenten, ob $x < y$ gilt. Falls ja, werden im Ergebnis alle Bits dieser Komponente auf eins gesetzt, ansonsten auf null.

Die bitweisen SSE-Logikoperationen `_mm_and_ps`, `_mm_andnot_ps` und `__mm_or_ps` führen eine Und-Verknüpfung, eine Und-Verknüpfung mit dem Komplement des ersten Arguments und eine Oder-Verknüpfung durch, so dass wir unseren Plan in die Tat umsetzen können und das folgende Programmfragment erhalten:

```
void
vecmax(int n, const float *x, const float *y, float *mx)
{
  __m128 v_x, v_y, v_mx;
  int i;

  for(i=0; i+3<n; i+=4) {
```

```
    v_x = _mm_loadu_ps(x+i);
    v_y = _mm_loadu_ps(y+i);
    v_mask = _mm_cmplt_ps(v_x, v_y);
    v_mx = _mm_or_ps(_mm_and_ps(v_mask, v_y),
                     _mm_andnot_ps(v_mask, v_x));
    _mm_storeu_ps(mx+i, v_mx);
  }

  for(; i<n; i++)
    if(x[i] < y[i])
      mx[i] = y[i];
    else
      mx[i] = x[i];
}
```

Während die ursprüngliche Version der Funktion 7,4 Sekunden für 10 000 000 Aufrufe mit $n = 1\,000$ benötigt, genügen unserer SSE-Version dafür 4,5 Sekunden.

Logisch-arithmetische Variante. Wir können die Geschwindigkeit noch etwas verbessern, indem wir arithmetische und logische Operationen mischen: Da $x + (y - x) = y$ gilt, können wir den Kern unserer Schleife in der Form

```
mx[i] = x[i] + (x[i] < y[i] ? y[i] - x[i] : 0.0);
```

schreiben. Die Subtraktion können wir der Funktion `_mm_sub_ps` berechnen, anschließend mit der Funktion `_mm_and_ps` eine logische Und-Verknüpfung mit der schon bekannten Bitmaske durchführen, und anschließend das Ergebnis zu `x[i]` addieren.

Damit erhalten wir die folgende Variante der SSE-Maximumberechnung:

```
void
vecmax(int n, const float *x, const float *y, float *mx)
{
  __m128 v_x, v_y, v_mx;
  int i;

  for(i=0; i+3<n; i+=4) {
    v_x = _mm_loadu_ps(x+i);
    v_y = _mm_loadu_ps(y+i);
    v_mx = _mm_add_ps(v_x,
                      _mm_and_ps(_mm_cmplt_ps(v_x, v_y),
                                 _mm_sub_ps(v_y, v_x)));
    _mm_storeu_ps(mx+i, v_mx);
  }

  for(; i<n; i++)
```

```
    if(x[i] < y[i])
      mx[i] = y[i];
    else
      mx[i] = x[i];
}
```

Die Ausführungszeit verringert sich auf 3,5 Sekunden, wir benötigen also weniger als die Hälfte der Zeit, die für die konventionelle Variante gemessen wurde. Eine denkbare Erklärung besteht darin, dass logische und arithmetische Operationen bei vielen Prozessoren von unterschiedlichen Rechenwerken ausgeführt werden, die unterschiedlich schnell arbeiten können, so dass ein Verhältnis von drei arithmetischen Operationen zu einer logischen Operation effizienter sein kann als ein Verhältnis von einer arithmetischen zu drei logischen.

Übrigens gibt es für die Berechnung des Maximums eine spezielle Funktion namens `_mm_max_ps`, mit der sich die Rechenzeit auf 1,7 Sekunden reduziert. Für dieses Teilkapitel wurde aus pädagogischen Gründen der ausführlichere Zugang über `_mm_cmplt_ps` und `_mm_and_ps`/`_mm_andnot_ps`/`_mm_or_ps` gewählt, da er sich auch auf allgemeinere Probleme übertragen lässt: Bei jeder beliebigen Fallunterscheidung können wir beide Zweige ausführen und anschließend mit den logischen Verknüpfungen die korrekten Komponenten aus beiden Ergebnissen zusammensetzen.

Allgemeineres Beispiel. Die hier beschriebene Vorgehensweise lässt sich auch auf sehr viel allgemeinere Fallunterscheidungen übertragen. Als Beispiel untersuchen wir das Programmfragment

```
for(i=0; i<n; i++)
  if(x[i] < y[i])
    z[i] = 2.0f * x[i] - 3.5f * y[i] + 1.0f;
  else
    z[i] = 3.5f * x[i];
```

Zur Vereinfachung gehen wir davon aus, dass `n` durch vier teilbar ist, so dass wir keine Sonderbehandlung für „überstehende“ Komponenten des Arrays benötigen. Indem wir wieder `v_mask` als Bitmaske verwenden und die Konstanten `2.0f`, `3.5f` und `1.0f` in Vektorvariablen `v_20`, `v_35` und `v_10` speichern, erhalten wir das folgende SSE-Programmstück:

```
const float c_20 = 2.0f, c_35 = 3.5f, c_10 = 1.0f;
v_20 = _mm_load1_ps(&c_20);
v_35 = _mm_load1_ps(&c_35);
v_10 = _mm_load1_ps(&c_10);
for(i=0; i<n; i+=4)
{
  v_x = _mm_loadu_ps(x+i);
  v_y = _mm_loadu_ps(y+i);
```

```
    v_mask = _mm_cmplt_ps(v_x, v_y);
    v_z1 = _mm_add_ps(_mm_sub_ps(_mm_mul_ps(v_20, v_x),
                                 _mm_mul_ps(v_35, v_y)),
                      v_10);
    v_z2 = _mm_mul_ps(v_35, v_x);
    v_z = _mm_or_ps(_mm_and_ps(v_mask, v_case1),
                    _mm_andnot_ps(v_mask, v_case2));
    _mm_storeu_ps(z+i, v_z);
  }
```

In diesem Beispiel speichern wir die Ergebnisse, die `z` in beiden Zweigen der Fallunterscheidung annimmt, in `v_z1` und `v_z2`, so dass wir mit Hilfe der Bitmaske `v_mask` und der logischen Verknüpfungen aus beiden Variablen das Ergebnis `v_z` zusammensetzen können.

4.4 Umsortieren von Einträgen

Für manche Algorithmen ist es nützlich, die Daten innerhalb eines Vektorregisters umordnen zu können. Dafür stehen uns eine Reihe von Funktionen zur Verfügung, unter denen `_mm_shuffle_ps` vermutlich die flexibelste ist. Sie erwartet zwei Vektorregister a und b sowie eine 8-Bit-Zahl s, die beschreibt, wie sich das Ergebnis aus den Komponenten der Register a und b zusammensetzen soll. Dabei wir s in vier 2-Bit-Paare s_0, s_1, s_2, s_3 zerlegt, von denen s_0 die beiden niedrigsten Bits aufnimmt und s_3 die beiden höchsten. Jedes dieser Paare wird als Zahl zwischen 0 und 3 interpretiert, die angibt, welche Komponente aus den Vektorregistern in die korrespondierende Komponente des Ergebnisses übernommen werden soll. s_0 und s_1 geben dabei die Komponenten von a an, die in die erste und die zweite Komponente des Ergebnisses übernommen werden sollen. s_2 und s_3 geben die Komponenten von b an, die in die dritte und vierte Komponente des Ergebnisses geschrieben werden sollen.

Indem wir beispielsweise für a und b dasselbe Vektorregister verwenden, können wir die Komponenten beliebig umsortieren oder auch kopieren. In dieser Weise lässt sich beispielsweise die bereits erwähnte Funktion `_mm_load1_ps` umsetzen.

Matrix-Vektor-Multiplikation. Als Beispiel für die Verwendung dieser Funktion wählen wir die Multiplikation einer 4×4-Matrix mit einem vierdimensionalen Vektor. Aus gutem Grund wird diese Operation häufig so formuliert, dass das Ergebnis der Multiplikation zu dem Ergebnisvektor hinzuaddiert wird, statt ihn zu überschreiben:

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} \leftarrow \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} + \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}.$$

Diese Operation spielt beispielsweise eine zentrale Rolle bei der Berechnung der für dreidimensionale Computergrafiken wichtiger Koordinatentransformationen.

Wir gehen davon aus, dass x und y in Vektorregistern vorliegen. Die Multiplikation können wir als Folge von Vektoradditionen darstellen:

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} \leftarrow \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} + \begin{pmatrix} a_{11} \\ a_{21} \\ a_{31} \\ a_{41} \end{pmatrix} x_1 + \begin{pmatrix} a_{12} \\ a_{22} \\ a_{32} \\ a_{42} \end{pmatrix} x_2 + \begin{pmatrix} a_{13} \\ a_{23} \\ a_{33} \\ a_{43} \end{pmatrix} x_3 + \begin{pmatrix} a_{14} \\ a_{24} \\ a_{34} \\ a_{44} \end{pmatrix} x_4.$$

Damit ist klar, wie wir vorgehen sollten: Wir holen die vier Spalten der Matrix in Vektorregister und verwenden `_mm_shuffle_ps`, um die einzelnen Komponenten des Vektorregisters x in alle Komponenten eines weiteren Vektorregisters x' zu kopieren, das dann mit der entsprechenden Spalte multipliziert und zu y addiert werden kann:

```
void
vecmatmul(int n, const float *x, float *y)
{
  __m128 v_x, v_xj, v_y, v_a1, v_a2, v_a3, v_a4;
  int i;

  v_a1 = _mm_loadu_ps(a1);
  v_a2 = _mm_loadu_ps(a2);
  v_a3 = _mm_loadu_ps(a3);
  v_a4 = _mm_loadu_ps(a4);

  for(i=0; i<n; i++) {
    v_x = _mm_loadu_ps(x+4*i);
    v_y = _mm_loadu_ps(y+4*i);

    v_xj = _mm_shuffle_ps(v_x, v_x, 0x00);
    v_y = _mm_add_ps(v_y, _mm_mul_ps(v_a1, v_xj));
    v_xj = _mm_shuffle_ps(v_x, v_x, 0x55);
    v_y = _mm_add_ps(v_y, _mm_mul_ps(v_a2, v_xj));
    v_xj = _mm_shuffle_ps(v_x, v_x, 0xaa);
    v_y = _mm_add_ps(v_y, _mm_mul_ps(v_a3, v_xj));
    v_xj = _mm_shuffle_ps(v_x, v_x, 0xff);
    v_y = _mm_add_ps(v_y, _mm_mul_ps(v_a4, v_xj));

    _mm_storeu_ps(y+4*i, v_y);
  }
}
```

Auf einem Intel Core i7-920 benötigen 1 000 000 Wiederholungen der Matrix-Vektor-Multiplikation für 1 000 vierdimensionale Vektoren ungefähr 7,9 Sekunden, unserer vektorisierten Variante genügen ungefähr 2,8 Sekunden.

Wenn wir zu 1 000 Wiederholungen für 1 000 000 Vektoren übergehen, benötigt die einfache Version des Algorithmus knapp 11,7 Sekunden, während die vektorisierte Fassung

mit gut 3,9 Sekunden auskommt, wir erreichen also fast die vierfache Geschwindigkeit, obwohl die Vektoren nicht mehr in den Cache passen. Das dürfte darauf zurückzuführen sein, dass bei diesem Beispiel relativ viele Rechenoperationen (16 Multiplikationen und 16 Additionen) pro Vektor anfallen, so dass der Durchsatz des Hauptspeichers ausreicht, um die Rechenwerke auch bei Verwendung von Vektorbefehlen noch schnell genug zu versorgen.

4.5 Konvertierung und ganzzahlige Arithmetik

Gelegentlich genügen die von dem Befehlssatz des Prozessors gebotenen Rechenoperationen nicht, um eine Aufgabe zu lösen. Beispielsweise haben wir gesehen, dass wir mit SSE-Befehlen im Wesentlichen auf die Grundrechenarten und einige Zusatzoperationen beschränkt sind. Als Beispiel für eine Funktion, die SSE nicht unmittelbar unterstützt, untersuchen wir die Exponentialfunktion

$$x \mapsto \exp(x) = e^x,$$

die beispielsweise für die Wahrscheinlichkeitsrechnung von großer Bedeutung ist.

Exponentialreihe. Die Exponentialfunktion ist *analytisch*, lässt sich also durch eine Taylor-Reihe darstellen, konkret durch die Reihe

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

Falls $|x|$ nicht allzu groß ist, können wir alle Potenzen jenseits eines gewissen $m \in \mathbb{N}$ vernachlässigen und erhalten so

$$e^x \approx 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^m}{m!}.$$

Für die Auswertung dieses Ausdrucks genügen die vier Grundrechenarten, die uns SSE zur Verfügung stellt. Die Berechnung wird besonders elegant, wenn wir ausnutzen, dass

$$\frac{x^{\ell+1}}{(\ell+1)!} = \frac{x \cdot x^\ell}{(\ell+1)\cdot \ell!} = \frac{x}{\ell+1}\frac{x^\ell}{\ell!}$$

gilt, so dass wir durch wiederholtes Ausklammern

$$\begin{aligned} e^x &\approx 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^m}{m!} \\ &= 1 + x\left(1 + \frac{x}{2} + \frac{x^2}{2\cdot 3} + \dots + \frac{x^{m-1}}{2\cdot 3\cdots m}\right) \\ &= 1 + x\left(1 + \frac{x}{2}\left(1 + \frac{x}{3} + \dots + \frac{x^{m-2}}{3\cdot 4\cdots m}\right)\right) \end{aligned}$$

erhalten. Indem wir von innen nach außen die einzelnen Klammern der Reihe nach ausrechnen, erhalten wir das folgende Programmfragment:

```
y = 1.0 + x / m;
y = 1.0 + x / (m-1) * y;
y = 1.0 + x / (m-2) * y;
  ...
y = 1.0 + x * y;
```

Da eine Division wesentlich mehr Zeit als eine Multiplikation erfordert, bietet es sich an, die Werte $1, 1/2, \ldots, 1/m$ als Konstanten in das Programm aufzunehmen, so dass für die Approximation der Exponentialfunktion ausschließlich Multiplikationen und Additionen benötigt werden.

Reduktion des Definitionsbereichs. Dieser Zugang ist einfach und elegant, hat allerdings den Nachteil, dass die Approximation nur genau genug ist, falls der Betrag $|x|$ des Arguments x nicht zu groß wird. Damit wir eine allgemein verwendbare Funktion erhalten, sollten wir also eine Sonderbehandlung für große Werte von $|x|$ vorsehen.

Eine mögliche Vorgehensweise besteht darin, die Exponentialrechengesetze auszunutzen: Es gilt

$$\exp(x) = e^x = e^{2x/2} = (e^{x/2})^2 = \exp(x/2)^2.$$

Falls $|x|$ zu groß sein sollte, können wir also die Exponentialfunktion für $x/2$ berechnen und das Ergebnis anschließend quadrieren, um das gewünschte Ergebnis zu erhalten. Natürlich können wir diesen Ansatz auch rekursiv anwenden, also x durch 2^k dividieren, die Exponentialfunktion von $x2^{-k}$ approximieren und das Ergebnis k-mal quadrieren.

Insbesondere für die Umsetzung auf einem Vektorrechner eignet sich dieser Zugang allerdings nicht gut: Da es von x abhängt, wie oft wir quadrieren müssen, hätten wir wieder das Problem, dass die auszuführenden Operationen von dem Inhalt der einzelnen Komponenten eines Vektorregisters abhängen.

Binärdarstellung von Gleitkommazahlen. Diese Schwierigkeiten lassen sich umgehen, indem man die Binärdarstellung von Gleitkommazahlen ausnutzt: Gleitkommazahlen werden in der Form $\pm\mu 2^p$ dargestellt, wobei $\mu \in [1/2, 1)$ die *Mantisse* und $p \in \mathbb{Z}$ der *Exponent* genannt werden. Eine Gleitkommazahl mit 2 zu multiplizieren ist also besonders einfach, weil wir einfach den Exponenten um eins erhöhen können. Entsprechend können wir mit 2^k multiplizieren, indem wir k zu dem Exponenten hinzuaddieren. Für jedes $k \in \mathbb{Z}$ gilt

$$\exp(x) = e^x = e^{x-k\ln 2+k\ln 2} = e^{x-k\ln 2}e^{k\ln 2} = \exp(x - k\ln 2)2^k.$$

Wir dürfen also beliebige Vielfache von $\ln 2$ von x abziehen, sofern wir das Ergebnis anschließend wieder mit 2^k multiplizieren.

Wir wählen k nun so, dass $x - k\ln 2$ in $[0, \ln 2)$ liegt, nämlich als

$$k = \lfloor x/\ln 2 \rfloor = \lfloor x \operatorname{ld} e \rfloor,$$

wobei $\operatorname{ld} e$ den dyadischen Logarithmus (also den Logarithmus zur Basis 2) von e bezeichnet. Damit ist sichergestellt, dass

$$x' = x - k\ln 2$$

nicht mehr zu groß sein kann, so dass wir unsere Approximation verwenden können, um $y' := \exp(x')$ zu berechnen und schließlich

$$y := \exp(x) = \exp(x')2^k = y'e^k$$

zu erhalten. Der IEEE-Standard für Gleitkommazahlen einfacher Genauigkeit sieht vor, dass der Exponent in den Bits 23 bis 30 der Binärdarstellung gespeichert ist, so dass wir die Multiplikation mit 2^k erreichen können, indem wir k zu diesen Bits addieren, etwa durch das folgende Programmfragment:

```
int32_t *yi = (int32_t *) &y;
*yi += k << 23;
```

Die erste Zeile gibt uns einen Zeiger auf die Adresse, an der die `float`-Variable `y` steht, und dieser Zeiger wird uminterpretiert als Zeiger `yi` auf eine `int32_t`-Variable, so dass wir die einzelnen Bits manipulieren können (derartige Ganzzahlvariablen fester Breite werden laut C99-Standard in der Header-Datei `inttypes.h` definiert). In der zweiten Zeile wird `k` an die korrekte Position verschoben, beginnend bei Bit 23, und zu der als `int32_t` interpretierten Variable `y` addiert.

Im Interesse der Einfachheit verzichten wir auf die Behandlung wichtiger Sonderfälle: Der IEEE-Standard sieht für Gleitkommazahlen einfacher Genauigkeit vor, dass der Exponent durch 8 Bit dargestellt wird. Eigentlich müssten wir also darauf achten, dass k nicht zu groß oder zu klein wird, um gegebenenfalls die im Standard vorgesehenen speziellen Codes für ∞ oder 0 einzutragen. Bei Bedarf (etwa falls eine Bibliotheksfunktion für den allgemeinen Einsatz entwickelt werden soll) können diese Sonderfälle mit den im Abschnitt 4.3 behandelten Techniken abgefangen werden.

Vektor-Typkonvertierung. Um unseren Algorithmus mit SSE-Befehlen durchzuführen, benötigen wir zusätzliche Funktionen, um Gleitkommazahlen und ganze Zahlen ineinander umzuwandeln, um ganze Zahlen bitweise zu verschieben und um Gleitkommazahlen als ganze Zahlen zu interpretieren, damit wir den Exponenten direkt manipulieren können.

Die zweite Generation von SSE, naheliegenderweise SSE2 genannt, bietet uns besonders elegante Befehle, um diese Aufgaben zu erfüllen: Neben dem bereits bekannten Datentyp `__m128` für vier Gleitkommazahlen gibt es den Datentyp `__m128i` für vier ganze Zahlen von jeweils 32 Bit. Für diese Ganzzahl-Vektorregister gibt es eine Reihe von arithmetischen Operationen, von denen wir für unsere Zwecke lediglich die komponentenweise Addition `_mm_add_epi32` benötigen, um k zu dem Exponenten zu addieren.

Für die Umrechnung von Ganzzahl-Vektorregistern in Gleitkomma-Vektorregister stehen uns Funktionen zur Verfügung: `_mm_cvttps_epi32` rundet die vier Komponenten eines Gleitkomma-Vektorregisters ab (es gibt auch eine Funktion `_mm_cvtps_epi32`, die auf-, ab- oder kaufmännisch runden kann) und gibt einen Ganzzahl-Vektorregister zurück, `_mm_cvtepi32_ps` macht aus einem Ganzzahl-Vektorregister wieder ein Gleitkomma-Vektorregister. Beide Funktionen rechnen dabei tatsächlich die unterschiedlichen Zahldarstellungen ineinander um.

Vektor-Cast. Die Funktionen `_mm_castps_si128` und `_mm_castsi128_ps` dagegen interpretieren lediglich ein Gleitkomma-Vektorregister als Ganzzahl-Vektorregister oder umgekehrt, ohne den Inhalt des Registers zu verändern. Sie ermöglichen uns dadurch den Zugriff auf die Bit-Darstellung der Gleitkommazahl.

Die Funktion `_mm_slli_epi32` schließlich verschiebt die Bits in den vier Komponenten eines Ganzzahl-Vektorregisters um eine Anzahl von Stellen. Mit ihr können wir den Korrekturfaktor k für den Exponenten an die richtige Stelle bringen.

Insgesamt können wir die Approximation der Exponentialfunktion damit in der folgenden Form angehen:

```
v_pow2 = _mm_cvttps_epi32(_mm_mul_ps(v_lde, v_x));
v_x1 = _mm_sub_ps(v_x, _mm_mul_ps(v_ln2, _mm_cvtepi32_ps(v_pow2)));

v_y1 = _mm_add_ps(v_one, _mm_mul_ps(v_exp2, v_x1));
v_y1 = _mm_add_ps(v_one, _mm_mul_ps(v_exp1, _mm_mul_ps(v_x1, v_y1)));
v_y1 = _mm_add_ps(v_one, _mm_mul_ps(v_exp0, _mm_mul_ps(v_x1, v_y1)));

v_y1 = _mm_castsi128_ps(_mm_add_epi32(_mm_castps_si128(v_y1),
                                      _mm_slli_epi32(v_pow2, 23)));
```

Die ersten beiden Zeilen berechnen $k = \lfloor x \operatorname{ld} e \rfloor$ in den Komponenten des Ganzzahl-Vektorregisters `v_pow2` und subtrahieren das Produkt $k \ln 2$ von x, um x' zu erhalten, gespeichert in `v_x1`.

Die nächsten drei Zeilen sind eine (sehr ungenaue) Approximation der Exponentialreihe y' mit $m = 2$. Die Koeffizienten 1, 1/2, 1/3 sind dabei in `v_exp0`, `v_exp1` und `v_exp2` gespeichert. Für praktische Anwendungen sollten hier natürlich mehr Terme verwendet werden, um eine brauchbare Genauigkeit zu erreichen. Wahrscheinlich ließe sich die Genauigkeit auch deutlich verbessern, indem man Polynom-Interpolation anstelle der der Exponentialreihe zugrundeliegenden Taylor-Entwicklung einsetzt.

Die letzten beiden Zeilen verschieben die Komponenten von k um 23 Stellen und führen eine Ganzzahl-Addition zu der Ganzzahl-Interpretation der Gleitkommazahl y' durch, um den Exponenten zu korrigieren.

Die in der C-Standardbibliothek enthaltene Funktion `expf` benötigt gut 17,3 Sekunden, um 1 000 000mal für einen Vektor der Länge $n = 1\,000$ die Exponentialfunktion auszuwerten. Unserer Approximation mit Exponentialreihe und Sonderbehandlung des Exponenten genügen dafür gut 11,3 Sekunden, wobei ein maximaler relativer Fehler von weniger als $1,2 \times 10^{-6}$ erreicht wird. Die SSE2-Variante dieses Algorithmus erreicht dieselbe Genauigkeit, benötigt aber nur knapp 3,3 Sekunden.

1

5 Multi-Threading

Bei einem Vektorrechner existieren zwar mehrere Rechenwerke, allerdings führen sie in jedem Schritt dieselbe Operation aus. Da diese Struktur für viele wichtige Anwendungen nicht flexibel genug ist, bietet es sich an, nach Verallgemeinerungen zu suchen.

Ein Ansatz besteht darin, einfach mehrere Prozessoren zu verwenden, die relativ unabhängig voneinander programmiert werden können. In der Regel müssen die Prozessoren dazu in der Lage sein, Daten untereinander auszutauschen. Für den Programmierer sind dabei *Shared-Memory-Systeme* besonders attraktiv, bei denen alle Prozessoren auf denselben Hauptspeicher zugreifen.

Heute am weitestens verbreitet sind derartige Systeme in der Gestalt von *Mehrkernprozessoren*, bei denen mehrere weitgehend unabhängige Prozessorkerne auf einem Chip zusammengefasst sind, der zusätzlich über die nötigen Schaltungen für die Kommunikation der Kerne miteinander und mit dem Rest des Computersystems enthält. Häufig enthält der Chip zusätzlich einen Cache, den sich die Kerne teilen, um schneller Daten austauschen zu können.

5.1 Geteilter Speicher

Bisher haben wir lediglich Rechnersysteme behandelt, bei denen unser Programm auf einem Prozessor läuft und auf den mit ihm verbundenen Hauptspeicher zugreift. Der Name legt bereits nahe, dass bei einem Shared-Memory-System mehrere Prozessoren sich denselben Speicher teilen.

Shared Memory und Caches. Die Folge ist, dass ein Prozessor auf Ergebnisse zurückgreifen kann, die ein anderer Prozessor berechnet und in den Speicher geschrieben hat. Dieses auf den ersten Blick einfache Konzept zieht einen erheblichen Schaltungsaufwand nach sich: Im Interesse der Geschwindigkeit sind in der Regel beide Prozessoren mit Caches ausgestattet und führen Lese- und Schreibzugriffe zunächst im Cache aus.

Dadurch könnte es passieren, dass ein Prozessor zwar Daten schreibt, diese Daten aber zunächst in seinem Cache bleiben und nicht im Hauptspeicher eintreffen. Damit würde ein anderer Prozessor sie nicht finden.

Selbst wenn ein Prozessor sichergestellt hat, dass seine Daten tatsächlich im Hauptspeicher angekommen sind, könnte es passieren, dass ein anderer Prozessor noch einen alten Stand des entsprechenden Speicherbereichs in seinem Cache hält und bei Lesezugriffen diese veraltete Kopie benutzt.

Cache-Kohärenz. Damit der Programmierer wie gewohnt arbeiten kann, als gebe es keine Caches, müssen die Prozessoren die *Cache-Kohärenz* mit Hilfe spezieller Kommunikationsprotokolle sicherstellen, also dafür sorgen, dass allen Programmen jederzeit eine einheitliche Sicht auf den Hauptspeicher geboten wird.

Die Cache-Kohärenz wird in modernen Systemen durch die Hardware garantiert, so dass unsere Programme tatsächlich weiterhin geschrieben werden können, als würden sie direkt mit dem Hauptspeicher arbeiten, weil die Cache-Hierarchie transparent ist.

False Sharing. Allerdings können die Kohärenzprotokolle die Geschwindigkeit unserer Programme erheblich beeinflussen: Bekanntlich arbeitet der Cache grundsätzlich mit vollständigen Cachelines. Angenommen, wir haben zwei Programme, die auf zwei Prozessoren eines Shared-Memory-Systems arbeiten. Das eine Programm liest und schreibt die Variable x an Adresse 5, das andere Programm die Variable y an Adresse 20. Wenn das System mit Cachelines von 32 oder 64 Bytes arbeitet, liegen x und y in derselben Cacheline. Sobald nun der erste Prozessor etwas in die Adresse 5 schreibt, ändert sich die Cacheline. Spätestens sobald der zweite Prozessor auf Adresse 20 zugreift, stellt das Cache-Kohärenzprotokoll fest, dass der erste Prozessor eine Änderung an der Cacheline durchgeführt hat, so dass der aktuelle Stand dieser Cacheline von dem ersten Prozessor zu dem zweiten übertragen werden muss, vermutlich sogar mit einem Umweg über den Hauptspeicher. Sobald der zweite Prozessor die Adresse 20 verändert hat, muss der neue Zustand der Cacheline wieder an den ersten Prozessor gesendet werden. Obwohl also keinerlei Datenaustausch zwischen beiden Prozessoren erforderlich ist und beide im Idealfall unabhängig in ihren jeweiligen Caches arbeiten könnten, wird durch die geforderte Cache-Kohärenz die Geschwindigkeit des Programms deutlich reduziert. Da das Problem durch Daten entsteht, die sich eine Cacheline teilen, bezeichnet man diesen Effekt als *false sharing*. Er kann innerhalb gewisser Grenzen vermieden werden, indem wir darauf achten, dass unsere Datenstrukturen so ausgerichtet sind, dass sie sich keine Cachelines teilen.

Mehrkernprozessoren. Ein Sonderfall eines Shared-Memory-Systems ist ein Rechner mit einem Mehrkernprozessor. Bei derartigen Prozessoren sind mehrere Prozessorkerne (bestehend aus Rechenwerken, Caches und einer Steuereinheit) auf einem Chip zusammengefasst. Häufig enthält der Chip zusätzlich einen Cache, den sich alle Kerne teilen und der beispielsweise den Datenaustausch zwischen den Kernen beschleunigt, indem er aufwendige Zugriffe auf den Hauptspeicher einspart.

Teilweise muss bei Mehrkernprozessoren zwischen logischen und physischen Kernen unterschieden werden: Beispielsweise beruht die *Hyperthreading-Technik* darauf, einem physischen Kern mehrere logische Kerne zuzuordnen, die sich bestimmte Funktionseinheiten teilen, beispielsweise Rechenwerke oder die Steuereinheiten für Speicherzugriffe. Der Durchsatz eines Prozessors wächst durch Hyperthreading nicht, allerdings lassen sich Latenzen besser verstecken, weil in der Zeit, in der ein logischer Kern auf Daten wartet, andere logische Kerne die Rechenwerke benutzen können.

NUMA-Architekturen. Falls sich ein Shared-Memory-Rechner aus mehreren Prozessoren zusammensetzt, kommt häufig eine NUMA-Architektur (*non-uniform memory access*) zum Einsatz, bei der jeder Prozessor einen eigenen Hauptspeicher besitzt, aber durch Kommunikationsprotokolle allen anderen Prozessoren den Zugriff auf diesen Speicher ermöglicht. Für ein Programm erfolgt dieser Datenaustausch transparent, allerdings hängt die Geschwindigkeit der Speicherzugriffe davon ab, ob ein Prozessor auf seinen eigenen Speicher oder auf den eines anderen Prozessors zugreift. Bei ungeschickter Programmierung, wenn nämlich sämtliche Daten in dem Speicher eines einzigen Prozessors liegen, begrenzt dessen Speicherdurchsatz die Leistung des Gesamtsystems.

Die Zuordnung von logischen Adressen zu physischen Adressen, und damit auch die Zuordnung von logischen Adressen zu den einzelnen Prozessoren, erfolgt zu dem Zeitpunkt, zu dem zum ersten Mal auf einen Speicherbereich zugegriffen wird. Typisch ist eine Strategie, bei der versucht wird, demjenigen Prozessor einen Speicherbereich zuzuordnen, der ihn als erster angesprochen hat. Falls also Prozessor 1 einen großen Speicherbereich anfordert und in dessen erster Hälfte arbeitet, während Prozessor 2 sich um die zweite Hälfte kümmert, wird die erste Hälfte in dem Speicher von Prozessor 1 untergebracht werden und die zweite Hälfte in dem von Prozessor 2.

5.2 Threads

Ein modernes Betriebssystem wird in der Regel dafür sorgen, dass jeder von ihm ausgeführte Prozess einen eigenen Adressraum erhält, der sich, abgesehen von Spezialfällen, nicht mit den Adressräumen der anderen laufenden Prozesse überschneidet. Diese Vorgehensweise ist sinnvoll, um zu vermeiden, dass sich die Prozesse gegenseitig stören. Wenn wir auf einem Shared-Memory-System mit mehreren Prozessoren an einem Problem arbeiten wollen, ist aber gerade ein Austausch von Daten erwünscht, so dass der Einsatz mehrerer Prozesse nicht sinnvoll erscheint.

Threads. Deshalb bieten moderne Betriebssysteme unterhalb der Ebene der Prozesse sogenannten *Threads* (auch als *light-weight processes* bezeichnet). Ein Thread ist ein Ausführungsstrang, also eine Folge von Befehlen, die ein Prozessor ausführt. Falls ein Prozess mehrere Threads enthält, teilen sich diese Threads denselben Adressraum, allerdings verfügt jeder Thread über einen eigenen Kellerspeicher (engl. *stack*), auf dem er Rücksprungadressen für Funktionsaufrufe und lokale Variablen ablegen kann. Da mehrere Threads desselben Prozesses auf verschiedenen Prozessoren gleichzeitig ausgeführt werden können, sind sie gut dafür geeignet, die von einem Prozess zu leistende Arbeit auf mehrere Prozessoren oder Prozessorkerne zu verteilen.

POSIX Threads. In der Regel beginnt ein Prozess mit einem einzelnen Thread. Um weitere Threads anlegen und verwalten zu können, sind sowohl die Unterstützung des Betriebssystems als auch geeignete Bibliotheken oder Compiler erforderlich. Eine frühe Implementierung sind *POSIX Threads*, bei denen Threads mit Hilfe einer separaten Programmbibliothek angelegt und beendet werden können. POSIX Threads sind sehr

flexibel, allerdings ist ihre Handhabung etwas umständlich, weil die Thread-Verwaltung ausschließlich über Funktionsaufrufe erfolgt und nicht eng an den Compiler gekoppelt ist.

OpenMP. Ein neuerer Standard für das Arbeiten mit Threads ist *OpenMP*. OpenMP wird über eine Erweiterung des Compilers und eine Bibliothek mit einigen Hilfsfunktionen realisiert. Im Fall der Programmiersprache C wird die Compiler-Erweiterung mit Hilfe der `#pragma`-Direktive umgesetzt, die es uns erlaubt, dem Compiler Hinweise darauf zu geben, an welchen Stellen wir es für sinnvoll halten, Threads einzusetzen. Der C-Standard sieht vor, dass ein Compiler ihm unbekannte `#pragma`-Direktiven ignoriert, also sollten zumindest einfache OpenMP-C-Programme sich auch von C-Compilern übersetzen lassen, die OpenMP nicht unterstützen.

Parallele Abschnitte. Mit der Direktive `#pragma omp parallel` können wir einen *parallelen Abschnitt* markieren, der von mehreren Threads ausgeführt werden kann:

```
int
main()
{
#pragma omp parallel
  {
    printf("Hello, world\n");
  }

  return 0;
}
```

Der parallele Abschnitt erstreckt sich von der öffnenden geschweiften Klammer, die unmittelbar auf `#pragma omp parallel` folgt, bis zu der zugehörigen schließenden Klammer. Er kann von mehreren Threads gleichzeitig ausgeführt werden, im OpenMP-Standard als *Team* bezeichnet, und endet, wenn alle Threads die schließende Klammer erreicht haben. Ein paralleler Abschnitt ist dabei als Zeitspanne zu interpretieren, in der mehrere Threads parallel laufen, nicht als eine Reihe von Programmzeilen. Beispielsweise können in einem parallelen Abschnitt Funktionen aufgerufen werden, die außerhalb der zu der OpenMP-Direktive gehörenden geschweiften Klammern stehen.

Wir können die Direktive um das Schlüsselwort `num_threads` ergänzen, um OpenMP einen Hinweis darauf zu geben, wieviele Threads den Abschnitt unserer Ansicht nach ausführen sollten:

```
#pragma omp parallel num_threads(5)
  {
    printf("Hello, world\n");
  }
```

OpenMP kann sich über unseren Hinweis hinwegsetzen und weniger oder mehr Threads anlegen, es kann sogar auch ganz auf zusätzliche Threads verzichten.

Zugehörigkeit zu parallelen Abschnitten. Da ein paralleler Abschnitt eine Zeitspanne ist, nicht eine Folge von Anweisungen, kann der Fall auftreten, dass eine Funktion manchmal in einem parallelen Abschnitt aufgerufen wird und manchmal in einem sequentiellen Abschnitt des Programms. Falls die Funktion global verfügbare Daten manipuliert, sollte sie im ersten Fall dafür sorgen, dass Schreib- und Lesezugriffe geeignet abgesichert sind, während sie im zweiten Fall keine Seiteneffekte von anderen Threads zu fürchten hat. Die Entscheidung kann die Funktion mit Hilfe der Funktion `omp_in_parallel` treffen, die in der Header-Datei `omp.h` definiert ist und die genau dann einen von null verschiedenen Wert zurückgibt, wenn sie in einem parallelen Abschnitt aufgerufen wird. Das folgende Programmfragment würde einmal `0` und mindestens einmal (abhängig von der Anzahl der verwendeten Threads) einen anderen Wert ausgeben:

```
void
report()
{
  printf("%d\n", omp_in_parallel());
}

int
main()
{
  report();
#pragma omp parallel
  {
    report();
  }
  return 0;
}
```

Nummer des Threads und Anzahl der Threads im Team. Um die zu leistende Arbeit sinnvoll auf die vorhandenen Threads verteilen zu können, wäre es sehr hilfreich, zu wissen, wieviele Threads im aktuellen Team vorhanden sind und wie wir sie identifizieren können. Dazu werden die ebenfalls in `omp.h` definierten Funktionen `omp_get_num_threads` und `omp_get_thread_num` verwendet. Die erste Funktion gibt die Anzahl der Threads im aktuellen Team zurück, die zweite die Nummer desjenigen Threads, der sie aufgerufen hat. Der folgende parallele Abschnitt wird beispielsweise die Nummern von `0` bis `omp_get_num_threads()-1` ausgeben:

```
#pragma omp parallel
  {
    printf("Thread %d of %d\n",
           omp_get_thread_num(),
           omp_num_threads());
  }
```

5.3 Umgang mit nicht-deterministischem Verhalten

Zwar wird jeder Thread seine Befehle in der Reihenfolge ausführen, die unser Programm vorgibt, allerdings stimmen sich mehrere Threads nicht untereinander ab, so dass es zu überraschenden Ergebnissen kommen kann. Im oberen Fall lässt sich in der Regel nicht vorhersagen, in welcher Reihenfolge die einzelnen Threads sich zu Wort melden werden. Theoretisch könnten sogar die Ausgaben der einzelnen Threads bunt gemischt auf unserem Bildschirm ankommen, allerdings verhindert das häufig eine entsprechend abgesicherte Version der `printf`-Funktion.

Geteilte Variablen. Wesentlich schwerwiegendere Konsequenzen können auftreten, wenn mehrere Threads auf dieselben Variablen zugreifen. Zur Illustration untersuchen wir das folgende Beispiel:

```
  i = 0;
#pragma omp parallel
  {
    i++;
  }
  printf("%d\n", i);
```

Falls es mit zwei Threads ausgeführt wird, könnte beispielsweise die folgende Befehlsfolge auftreten:

1. Thread 0 liest den Wert `0` der Variablen `i`.
2. Thread 1 liest den Wert `0` der Variablen `i`.
3. Thread 0 schreibt `1` in die Variable `i`.
4. Thread 1 schreibt `1` in die Variable `i`.

Das Programm würde also den Wert `1` zurückgeben. Es kann aber auch die folgende Befehlsfolge auftreten:

1. Thread 0 liest den Wert `0` der Variablen `i`.
2. Thread 0 schreibt den Wert `1` in die Variable `i`.
3. Thread 1 liest den neuen Wert `1` der Variablen `i`.
4. Thread 1 schreibt den Wert `2` in die Variable `i`.

Abhängig von der Reihenfolge, in der die beiden Threads auf den Speicher, nämlich auf die Variable `i`, zugreifen, können also zwei verschiedene Ergebnisse herauskommen. Dieser Effekt kann dazu führen, dass die Fehlersuche in Programmen sich sehr frustrierend gestaltet, weil beispielsweise ein Fehler nur in einem von tausend Testläufen auftritt.

In der Regel wissen wir nicht, in welcher Reihenfolge die Speicherzugriffe der nebenläufigen Threads erfolgen, deshalb lassen sich die Ergebnisse unseres Beispielprogramms nicht vorhersagen. Man spricht in diesem Kontext von nicht-deterministischem Verhalten.

Atomare Operationen. Dass Speicherzugriffe in einer nicht vorsehbaren Weise erfolgen, muss dem Nutzen eines mit mehreren Threads arbeitenden Programms nicht schaden. Beispielsweise hätten wir kein Problem, wenn jeder Thread nur auf Variablen zugreifen würde, die kein anderer Thread verwendet, denn dann wäre das *Ergebnis* unabhängig von der Ausführung der Threads immer dasselbe.

In unserem Beispiel lässt sich das nicht-deterministische Verhalten des Programms beherrschen, indem wir das Hochzählen der Variable `i` als eine *atomare Operation* deklarieren, bei der das Lesen des alten Werts, die Addition und das Zurückschreiben des neuen Werts „unteilbar“ erfolgen, so dass kein zweiter Thread die Variable lesen kann, während ein anderer sie verändert. Auf vielen Prozessoren werden atomare Operationen umgesetzt, indem der Zugriff auf den betroffenen Speicherbereich für alle Prozessorkerne gesperrt wird (beispielsweise bei der IA32 durch das Präfix `LOCK` vor dem korrespondierenden Maschinenbefehl).

In OpenMP können wir diese Aufgabe mit der Direktive `#pragma omp atomic` lösen:

```
#pragma omp atomic
    i++;
```

Die Direktive wirkt auf die unmittelbar folgende Anweisung, bei der es sich um eine Inkrement-, Dekrement- oder Zuweisungsanweisung handeln muss, und stellt sicher, dass sie atomar ausgeführt wird. In unserem Fall stellt diese Modifikation des Programms sicher, dass am Ende des parallelen Abschnitts immer das korrekte Ergebnis vorliegt, nämlich die Anzahl der laufenden Threads.

5.4 Private Variablen

Kollisionen bei Zugriffen auf den Speicher lassen sich besonders einfach vermeiden, wenn jeder Thread möglichst mit Variablen arbeitet, die für andere Threads unsichtbar sind. Derartige *private Variablen* liegen im Kellerspeicher eines Threads und sind deshalb für andere Threads nicht ohne Weiteres zugänglich (da alle Threads einen Adressraum teilen, kann ein fehlgeleiteter Thread allerdings im Prinzip trotzdem auf die lokalen Variablen eines anderen zugreifen).

Funktionen und Blöcke. Private Variablen entstehen beispielsweise bei Funktionsaufrufen: In C werden lokale Variablen (solange wir nicht mit den Schlüsselwörtern `static` oder `register` intervenieren) der aufgerufenen Funktion im Kellerspeicher angelegt, und jeder Thread besitzt einen eigenen Kellerspeicher. Ein Funktionsaufruf ist allerdings gar nicht zwingend erforderlich, denn auch in einem durch geschweifte Klammern eingeschlossenen *structured block* dürfen wir neue Variablen definieren. Beispielsweise erhält in dem folgenden Programmfragment jeder Thread eigene Versionen der Variablen `first` und `last`:

```
#pragma omp parallel
  {
```

```
    int first = n * omp_get_thread_num() / omp_get_num_threads();
    int last = n * (omp_get_thread_num()+1) / omp_get_num_threads();
    printf("From %d to %d\n", first, last-1);
  }
```

Wenn unser Programm ein Array der Länge n zu verarbeiten hätte, könnten wir in dieser Weise das Array in ungefähr gleich große zusammenhängende Teile zerlegen, um die sich dann jeweils ein Thread kümmert. Die Teile sind nur „ungefähr“ gleich groß, weil wir nicht voraussetzen dürfen, dass n ein Vielfaches der Anzahl der Threads ist.

Paralleler Abschnitt. Wir können auch beim Betreten eines parallelen Abschnitts dafür sorgen, dass *außerhalb* des Abschnitts definierte Variablen durch private Kopien ersetzt werden, indem wir die Direktive um das Schlüsselwort `private` ergänzen:

```
#pragma omp parallel private(i)
  {
    i = 0;
    i++;
    printf("%d\n", i);
  }
```

In diesem Beispiel wird jeder Thread `1` ausgeben, weil er nur eine private Kopie der Variablen `i` verändert hat. Wenn wir statt `private` das Schlüsselwort `firstprivate` verwenden, wird der bei Betreten des parallelen Abschnitts aktuelle Wert der Variablen in die lokalen Kopien übernommen:

```
  i = 1;
#pragma omp parallel firstprivate(i)
  {
    i++;
    printf("%d\n", i);
  }
```

Bei diesem Programmfragment wird jeder Thread die Zahl `2` ausgeben.

5.5 Arbeitsteilung

Wie wir gesehen haben, können wir mit Hilfe der Funktionen `omp_get_num_threads` und `omp_get_thread_num` bereits den Threads in einem Team unterschiedliche Aufgaben zuweisen und so Arbeit verteilen. Wesentlich kürzer und eleganter lässt sich diese Aufgabe lösen, indem wir dafür vorgesehene `#pragma`-Direktiven verwenden. Beispielsweise wird im folgenden Beispiel die in der Schleife anfallende Arbeit auf alle Threads des Teams verteilt:

```
#pragma omp parallel
{
```

```
  int i;
#pragma omp for
  for(i=0; i<n; i++)
    x[i] += alpha * y[i];
}
```

Die Direktive `#pragma omp for` gibt dem Compiler in diesem Beispiel einen Hinweis darauf, dass die folgende `for`-Schleife auf mehrere Threads verteilt werden soll. Es sind lediglich Schleifen erlaubt, bei denen die Schleifenvariable eine vorzeichenbehaftete ganze Zahl ist und bei denen der Compiler bei Erreichen der Schleife berechnen kann, wieviele Iterationen erforderlich sein werden. Die Schleifenvariable `i` wird für die Dauer des Schleifenkonstrukts durch eine private Variable gleichen Namens ersetzt, nach dem Ende der Schleife ist ihr Zustand undefiniert. Alle Threads des aktuellen Teams können die auf die `for`-Schleife folgenden Befehle erst bearbeiten, wenn alle Threads mit ihrem Teil der Schleife fertig sind und sämtliche Ergebnisse der Schleife im Speicher vorliegen. Falls also beispielsweise `n=1` gelten sollte und nur eine Iteration durchzuführen ist, müssen alle Threads darauf warten, dass einer von ihnen diese Iteration ausgeführt hat.

Scheduling. Mit Hilfe des Schlüsselworts `schedule` können wir beeinflussen, wie die in der `for`-Schleife durchzuführenden Iterationen auf Threads verteilt werden. Bei einer *statischen* Strategie werden die Iterationen in zusammenhängende Blöcke einer gegebenen Größe zerlegt und diese Blöcke zyklisch auf die vorhandenen Threads verteilt:

```
#pragma omp for schedule(static, 2)
for(i=0; i<n; i++)
  printf("%d handled by thread %d\n",
         i, omp_get_thread_num());
```

In diesem Fall werden jeweils zwei aufeinander folgende Iterationen demselben Thread zugewiesen. Falls das aktuelle Team aus drei Threads besteht, wird bei dieser Schleife Thread 0 für die Indizes 0 und 1 zuständig sein, Thread 1 für 2 und 3, Thread 2 für 4 und 5. Falls `n` größer als 5 ist, wird für die restlichen Iterationen wieder bei Thread 0 begonnen, der für 6 und 7 zuständig ist, gefolgt von Thread 1, der sich um 8 und 9 zu kümmern hat, und Thread 2, der 10 und 11 versorgt. Und so weiter.

Die statische Strategie ist von Vorteil, falls der Rechenaufwand für alle Iterationen ungefähr derselbe ist. Falls der Rechenaufwand variieren kann, bietet sich die *dynamische* Strategie an, bei der die zusammenhängenden Blöcke von Iterationen jeweils an den nächsten Thread vergeben werden, der unbeschäftigt ist:

```
#pragma omp for schedule(dynamic, 2)
for(i=0; i<n; i++)
  printf("%d handled by thread %d\n",
         i, omp_get_thread_num());
```

Dadurch kann die Schleife Iterationen mit unterschiedlichen Rechenzeiten gleichmäßiger auf mehrere Threads verteilen.

Falls wir keine Blockgröße angeben, arbeitet OpenMP, als wäre sie gleich eins. Das kann beispielsweise bei der Verarbeitung von Arrays die Geschwindigkeit beeinträchtigen, weil mehrere Threads auf Daten derselben Cacheline zugreifen und es so zu dem bereits beschriebenen Effekt des *false sharing* kommen kann.

Sektionen. Falls wir eine Reihe von unterschiedlichen Aufgaben lösen müssen, können wir die Direktive `#pragma omp sections` verwenden, die es uns erlaubt, mehrere Anweisungsblöcke zu definieren, die dann von verschiedenen Threads des aktuellen Teams ausgeführt werden:

```
#pragma omp sections
{
#pragma omp section
  printf("Thread %d in first section\n", omp_get_thread_num());
#pragme omp section
  printf("Thread %d in second section\n", omp_get_thread_num());
}
```

Auch in diesem Fall müssen alle Threads des aktuellen Teams am Ende des Blocks darauf warten, dass alle Ergebnisse vorliegen.

Einzelne Aufgaben. Gelegentlich fallen Aufgaben an, die nur von einem einzigen Thread einmal ausgeführt werden müssen und sich nicht auf mehrere Threads verteilen lassen, beispielsweise die Ausgabe einer Statusmeldung oder das Schreiben einer Datenstruktur in eine Datei. Für diese Situationen gibt es die Direktive `#pragma omp single`, die einen Anweisungsblock markiert, der nur von einem Thread einmal ausgeführt werden soll:

```
#pragma omp single
{
  printf("Hello, world!\n");
}
```

Auch hier warten alle Threads des aktuellen Teams am Ende des Blocks, bis derjenige Thread, der ihn ausführt, alle Ergebnisse in den Speicher geschrieben hat.

Kurzform. Falls beispielsweise nur eine einzige Schleife parallelisiert werden soll, können wir das Anlegen eines parallelen Abschnitts und die Parallelisierung der Schleife zusammenfassen, indem wir die Direktive `#pragma omp parallel for` verwenden, die die Direktiven `#pragma omp parallel` und `#pragma omp for` kombiniert. Die Parallelisierung einer Schleife erfordert dann nur noch eine einzige Zeile:

```
#pragma omp parallel for schedule(static, 16)
for(i=0; i<n; i++)
  x[i] += alpha * y[i];
```

Wie schon bei `#pragma omp for` wird auch hier die Variable `i` für die Ausführung der Schleife in jedem Thread durch eine private Kopie ersetzt, um unerwünschte Wechselwirkungen zwischen den Threads zu vermeiden. Die Blockgröße 16 sorgt dafür, dass jeweils 64 Byte (oder 128 Byte bei Gleitkommazahlen doppelter Genauigkeit) eines Arrays von einem Thread bearbeitet werden, so dass *false sharing* vermieden wird.

5.6 Synchronisation

Wenn ein Thread die von einem anderen Thread berechneten Ergebnisse verwenden soll, muss sichergestellt sein, dass der andere Thread diese Ergebnisse bereits berechnet und in den Speicher geschrieben hat, die Threads müssen sich also synchronisieren.

Lokale und globale Sicht auf Variablen. Damit ein Thread die ihm zugewiesenen Rechenoperationen möglichst effizient durchführen kann, bietet es sich an, die betroffenen Variablen in Register zu lesen und mit diesen Registern zu arbeiten. Dadurch entsteht allerdings das Problem, dass die Register eines Prozessors für andere Prozessoren nicht sichtbar sind, so dass unterschiedliche Threads möglicherweise unterschiedlicher Meinung darüber sind, welche Werte gemeinsam genutzte Variablen gerade haben: Die *lokalen* Werte der Variablen, also die Inhalte der Register, können sich von den *globalen* Werten, also den Inhalten des Hauptspeichers, unterscheiden.

Auf die Verwendung von Registern zu verzichten würde die Geschwindigkeit unserer Programme erheblich beeinträchtigen, deshalb sieht OpenMP vor, dass eine global konsistente Sicht auf die Variablen nur an bestimmten Punkten des Programms zu erwarten ist, so dass sich der Compiler zwischen diesen Punkten große Freiheiten bei der Optimierung des erzeugten Maschinenprogramms erlauben kann.

In unserem Programm können wir mit der Direktive `#pragma omp flush` dafür sorgen, dass die lokale Sicht des aufrufenden Threads mit der globalen Sicht übereinstimmt. Es werden also beispielsweise eventuell in Registern aufbewahrte Variablen in den Hauptspeicher geschrieben oder aus dem Hauptspeicher gelesen.

Barrieren. Falls mehrere Threads sich eine Aufgabe teilen, kann das Problem auftreten, dass ein Thread die Ergebnisse eines anderen Threads benötigt, die dieser eventuell noch nicht berechnet hat. Eine einfache Lösung besteht darin, den Thread an einem Punkt seines Programms warten zu lassen, bis alle anderen Threads denselben Punkt erreicht haben. Dadurch wäre sichergestellt, dass die bis zu diesem Punkt zu leistenden Arbeiten abgeschlossen wurden und alle Teilergebnisse vorliegen.

Genau diesem Zweck dienen *Barrieren*, die ein Thread erst dann passieren kann, wenn alle anderen Threads seines Teams ebenfalls dieselbe Barriere erreicht haben. Wir können OpenMP mit der Direktive `#pragma omp barrier` dazu veranlassen, eine solche Barriere einzurichten.

Es muss sichergestellt werden, dass eine Barriere jeweils entweder von allen Threads eines Teams erreicht wird oder von keinem, denn sonst würden einige Threads ewig

warten und das Programm niemals fertig werden. Eine Barriere beinhaltet einen Abgleich der lokalen und globalen Sicht der Variablen, den wir deshalb nicht explizit mit `#pragma omp flush` erzwingen müssen.

Im folgenden Beispiel beschäftigen sich alle Threads eine Weile mit der Berechnung von Linearkombinationen, bevor sie sich synchronisieren:

```
#pragma omp parallel
{
  int i, j, first, last;

  first = n * omp_get_thread_num() / omp_get_num_threads();
  last = n * (omp_get_thread_num()+1) / omp_get_num_threads();

  for(j=0; j<iterations; j++)
    for(i=first; i<last; i++)
      x[i] += alpha * y[i];

#pragma omp barrier

  printf("Thread %d is done.\n", omp_get_thread_num());
}
```

Durch die Barriere ist sichergestellt, dass die Meldungen erst ausgegeben werden, nachdem *jeder* Thread seinen Teil der Arbeit erfolgreich abgeschlossen hat.

Implizite Sychronisation. Allen vorgestellten Arbeitsteilungs-Direktiven gemeinsam ist, dass sie die Threads des aktuellen Teams implizit synchronisieren: Bevor die Befehle des geteilten Blocks ausgeführt werden, müssen alle Threads des aktuellen Teams sicherstellen, dass ihre Sicht auf den Hauptspeicher konsistent mit der der anderen Threads ist, es wird also implizit `#pragma omp flush` verwendet. Am Ende des Blocks wird nicht nur wieder sichergestellt, dass die Sicht auf den Speicher konsistent ist, es wird sogar implizit eine Barriere verwendet, damit alle Threads erst weiterarbeiten, wenn der geteilte Block vollständig abgearbeitet wurde. Diese Vorgehensweise ist häufig sinnvoll, um sicherzustellen, dass alle Ergebnisse berechnet wurden und für weitere Operationen zur Verfügung stehen, sie kann aber unter Umständen die Geschwindigkeit eines Programms erheblich reduzieren. Wir können die Synchronisation verhindern, indem wir das Schlüsselwort `nowait` hinzufügen:

```
#pragma omp parallel
{
  int i, j;

  for(j=0; j<iterations; j++) {
#pragma omp for nowait
    for(i=0; i<n; i++)
```

```
      x[i] += alpha * y[i];
  }
}
```

Auf einem Intel Core i7-920 benötigt dieses Programmfragment für $n = 1\,000$ und $10\,000\,000$ Iterationen 3,8 Sekunden. Wenn wir auf `nowait` verzichten und eine Synchronisation nach jeder Iteration fordern, erhöht sich die Rechenzeit auf 7,4 Sekunden. Bei $n = 10\,000\,000$ und $1\,000$ Iterationen dagegen benötigt die unsynchronisierte Fassung 13,6 Sekunden, während die synchronisierte 14,3 Sekunden benötigt, der zusätzliche Aufwand fällt also deutlich geringer aus. Wir sollten darauf achten, dass wir die Threads im Team nur dann synchronisieren, wenn es für das korrekte Funktionieren des Programms erforderlich ist. Wie unser Beispiel zeigt, kann die Synchronisation viel Zeit kosten, da der OpenMP-Standard vorsieht, dass im Rahmen der Synchronisation garantiert werden muss, dass alle Ergebnisse, die bis zu diesem Punkt in den Speicher hätten geschrieben werden sollen, auch wirklich dort angekommen sind. Je nach Implementierung und Hardware kann das bedeuten, dass sämtliche Inhalte der Caches in den Hauptspeicher geschrieben werden, worunter natürlich die Geschwindigkeit des Programms leidet.

Kritische Abschnitte. Wie wir bereits gesehen haben, können wir mit der Direktive `#pragma omp atomic` bestimmte Befehle vor Unterbrechungen schützen, um von mehreren Threads gemeinsam genutzte Variablen konsistent zu halten. Allerdings betrifft diese Direktive jeweils nur einen einzelnen Befehl und kann nur für bestimmte Befehle verwendet werden.

Falls wir ganze Blöcke beliebiger Anweisungen vor Seiteneffekten durch andere Threads schützen wollen, können wir *kritische Abschnitte* verwenden: Wenn wir mit `#pragma omp critical` einen Teil des Programms als kritischen Abschnitt markieren, stellt OpenMP sicher, dass sich nie mehr als ein Thread in diesem Abschnitt aufhalten kann. Falls wir also Zugriffe auf von mehreren Threads gemeinsam genutzte Variablen konsequent nur innerhalb eines kritischen Abschnitts durchführen, können Schreib- und Lesezugriffe mehrerer Threads einander nicht mehr stören. Ein einfaches Beispiel ist die folgende Iteration:

```
#pragma omp parallel for
for(i=0; i<iterations; i++) {
#pragma omp critical
  {
    a = exp(-a*a);
  }
}
```

Obwohl für einen Schritt der Iteration Lesezugriffe auf die gemeinsam genutzte Variable `a`, Rechenoperationen mit dem Wert und Schreibzugriffe erforderlich sind, besteht nicht die Gefahr, dass sich mehrere Threads stören.

Benannte kritische Abschnitte. Wenn wir `#pragma omp critical` innerhalb eines Programms mehrfach verwenden, besteht der kritische Abschnitt aus *allen* entsprechend markierten Programmteilen. Im folgenden Beispiel wäre es also ausgeschlossen, dass zwei Threads parallel die Variablen `a` und `b` aktualisieren:

```
#pragma omp parallel for
for(i=0; i<iterations; i++) {
#pragma omp critical
  {
    a = exp(-a*a);
  }
#pragma omp critical
  {
    b = cos(b);
  }
}
```

Das ist wenig sinnvoll, weil Schreib- und Lesezugriffe auf `a` nicht von denen auf `b` gestört werden können. Deshalb bietet uns OpenMP die Möglichkeit, mehrere kritische Abschnitte zu definieren, indem wir ihnen Namen geben:

```
#pragma omp parallel for
for(i=0; i<iterations; i++) {
#pragma omp critical(rw_a)
  {
    a = exp(-a*a);
  }
#pragma omp critical(rw_b)
  {
    b = cos(b);
  }
}
```

Mit dieser Modifikation kann nun ein Thread in dem Abschnitt `rw_a` arbeiten, während ein anderer in dem Abschnitt `rw_b` beschäftigt ist.

Warteschlangen. Ein typisches Beispiel für die Verwendung eines kritischen Abschnitts ist eine Warteschlange, über die mehrere Threads anfallende Aufgaben untereinander aufteilen:

```
void
enqueue(workitem *item)
{
#pragma omp critical(queue)
  {
    if(tail)
```

```
      tail->next = item;
    item->next = 0;
    tail = item;
    if(!head)
      head = tail;
  }
}

workitem *
dequeue()
{
  workitem *item = 0;
#pragma omp critical(queue)
  {
    if(head) {
      item = head;
      head = head->next;
    }
    if(!head)
      tail = head;
  }
  return item;
}
```

Die Warteschlange besteht aus Datenstrukturen des Typs `workitem`, die unter anderem den Zeiger `next` auf das nächste Element in der Warteschlange enthalten. Eine globale Variable `head` zeigt auf das erste (also älteste) Element der Warteschlange, eine Variable `tail` auf das letzte (also jüngste). Die Funktion `enqueue` fügt ein Element am Ende der Warteschlange hinzu, die Funktion `dequeue` entfernt ein Element am Anfang. Beide greifen auf die globalen Variablen `head` und `tail` zu, so dass wir sie mit einem kritischen Abschnitt namens `queue` schützen.

Locks. Ein Nachteil der kritischen Abschnitte besteht darin, dass sie statisch festgelegt werden, während das Programm übersetzt wird. Falls wir beispielsweise den Zugriff auf ein Array steuern wollen, dessen Länge erst zur Laufzeit festgelegt wird, können wir nicht für jedes Element des Arrays einen eigenen kritischen Abschnitt anlegen, sondern lediglich einen für das gesamte Array.

Diesen Nachteil können wir umgehen, indem wir *Locks* (englisch für Schlösser) verwenden. Ein Lock ist eine Datenstruktur, die die beiden Zustände „offen“ und „geschlossen“ annehmen kann. Zwischen den beiden Zuständen wechseln wir, indem wir auf- oder zuschließen.

Der entscheidende Punkt eines Locks besteht darin, dass es nicht abgeschlossen werden kann, wenn es schon geschlossen ist. Falls ein Thread es trotzdem versucht, wird er aufgehalten, bis das Lock wieder geöffnet wurde. Auf diesem Weg lassen sich beispielsweise

kritische Abschnitte sehr einfach umsetzen: Bei Betreten des kritischen Abschnitts wird das Lock abgeschlossen, bei Verlassen wird es wieder aufgeschlossen. Der erste Thread, der den kritischen Abschnitt erreicht, findet ein offenes Lock vor und schließt es, so dass alle anderen Threads warten müssen, bis es wieder geöffnet wird.

In OpenMP wird ein Lock durch die Datenstruktur `omp_lock_t` dargestellt. Diese Datenstruktur muss mit der Funktion `omp_init_lock` initialisiert werden, bevor sie verwendet werden kann. Sobald sie nicht mehr benötigt wird, sollte sie mit `omp_destroy_lock` wieder in einen uninitialisierten Zustand zurückversetzt werden, beispielsweise um eventuell angelegten Hilfsspeicher freizugeben.

Das Ab- und Aufschließen des Locks erfolgt dann mit den Funktionen `omp_set_lock` und `omp_unset_lock`. Falls das Lock bereits geschlossen ist, wartet `omp_set_lock`, bis es wieder geöffnet wurde, so dass der aufrufende Thread nichts tun kann.

Die Funktion `omp_test_lock` dagegen versucht ebenfalls, das Lock zu schließen, wartet allerdings nicht, falls es schon geschlossen sein sollte. Stattdessen gibt die Funktion den Wert null zurück, so dass der Thread sich die Zeit mit anderen Dinge vertreiben kann. Falls die Funktion einen von null verschiedenen Wert zurückgibt, hat sie das Lock erfolgreich geschlossen.

Mit Hilfe von Locks können wir beispielsweise mehrere Warteschlangen anlegen, die vor Störungen während des Einfügens und Löschens von Einträgen geschützt sind:

```
typedef struct {
  workitem *head, *tail;
  omp_lock_t lock;
} queue;

queue *
new_queue()
{
  queue *q;
  q = (queue *) malloc(sizeof(queue));
  q->head = q->tail = 0;
  omp_init_lock(&q->lock);
  return q;
}

void
del_queue(queue *q)
{
  omp_destroy_lock(&q->lock); free(q);
}

void
enqueue_queue(queue *q, workitem *item)
{
```

```
  omp_set_lock(&q->lock);
  if(q->tail) q->tail->next = item;
  item->next = 0;
  q->tail = item;
  if(!q->head) q->head = q->tail;
  omp_unset_lock(&q->lock);
}
```

5.7 Task-basierte Parallelisierung

Bisher haben wir zwei Techniken kennen gelernt, mit denen OpenMP Arbeit auf mehrere Threads verteilen kann: Wir können mit den OpenMP-Bibliotheksfunktionen `omp_get_thread_num` und `omp_get_num_threads` herausfinden, aus wie vielen Threads das aktuelle Team besteht und der wieviele davon gerade einen Teil des Programms ausführt, um dann „per Hand“ diesem Thread Arbeit zuzuteilen. Dieser Ansatz ist flexibel, kann aber auch etwas umständlich sein, beispielsweise falls sich die Threads untereinander abstimmen müssen. OpenMP bietet für die Synchronisation nur relativ beschränkte Möglichkeiten, beispielsweise Barrieren.

Alternativ können wir mit Arbeitsteilungsdirektiven beispielsweise die Iterationen einer Schleife automatisch auf die verfügbaren Threads verteilen lassen. Für einfache Schleifen ist das ein idealer Zugang, für Programme mit komplizierteren Strukturen ist er weniger gut geeignet.

Mit der Version 3.0 wurden *Tasks* in den OpenMP-Standard aufgenommen. Ein Task ist dabei eine zu bewältigende Aufgabe, ausgedrückt durch eine Folge von Anweisungen, die unabhängig von anderen Aufgaben verarbeitet werden kann.

Jeder Task kann weitere Tasks definieren, und er kann darauf warten, dass diese Tasks abgeschlossen werden, um anschließend selbst weitere Berechnungen vorzunehmen. Dadurch ergibt sich eine Baumstruktur, bei der ein Vater-Task einen oder mehrere Sohn-Tasks anlegen, auf ihre Fertigstellung warten, und anschließend die von ihnen berechneten Ergebnisse weiter verarbeiten kann.

Die entscheidende Besonderheit bei Tasks besteht darin, dass sie durch das OpenMP-Laufzeitsystem auf alle Threads des aktuellen Teams verteilt werden. Dadurch kann die Rechenlast gleichmäßig verteilt werden, falls wir dafür sorgen, dass arbeitsintensive Aufgaben in viele kleinere Tasks zerlegt werden.

Als Beispiel untersuchen wir die Berechnung der Anzahl der Knoten eines Binärbaums, der durch die Datenstruktur

```
typedef struct _tree tree;
struct _tree {
  tree *left;
  tree *right;
};
```

beschrieben ist. Hier sind `left` und `right` Zeiger auf die beiden Söhne. Ein Nullzeiger bezeichnet einen leeren Baum.

Die Berechnung der Anzahl der Knoten können wir mit einer einfachen Rekursion durchführen:

```
int
count_descendants(const tree *t)
{
  int desc_left, desc_right;

  if(t == 0) return 0;

  desc_left = count_descendants(t->left);
  desc_right = count_descendants(t->right);

  return desc_left + desc_right + 1;
}
```

Wir stellen fest, dass das Durchzählen des linken und rechten Teilbaums unabhängig voneinander erfolgen kann, so dass wir es parallelisieren können. Dazu erzeugen wir jeweils einen Task für den linken und den rechten Teilbaum, der jeweils den rekursiven Funktionsaufruf enthält.

Das Endergebnis können wir erst berechnen, wenn beide Tasks abgeschlossen wurden, deshalb müssen wir der `taskwait`-Direktive dafür sorgen, dass der Vater-Task auf die Sohn-Tasks wartet. Damit ergibt sich die folgende Variante:

```
int
count_descendants(const tree *t)
{
  int desc_left, desc_right;

  if(t == 0) return 0;

  #pragma omp task shared(desc_left)
    desc_left = count_descendants(t->left);

  #pragma omp task shared(desc_right)
    desc_right = count_descendants(t->right);

  #pragma omp taskwait

  return desc_left + desc_right + 1;
}
```

Wenn wir mit der `task`-Direktive einen Task anlegen, erhält dieser Task seine eigenen Kopien lokaler Variablen. In unserem Fall müssen wir dafür sorgen, dass die Ergebnisse jeweils in die Variablen `desc_left` und `desc_right` des Vater-Tasks geschrieben werden, die wir deshalb mit der `shared`-Klausel als gemeinsam genutzte Variablen deklarieren.

Damit diese Funktion korrekt arbeitet, müssen wir sie innerhalb eines parallelen Abschnitts aufrufen, damit es ein Team gibt, das sich die Arbeit teilen kann, aber sie sollte natürlich nur einmal aufgerufen werden, damit nicht jeder Thread des Teams dasselbe berechnet. Ein einfacher Zugang sieht wie folgt aus:

```
int desc;

#pragma omp parallel
{
  #pragma omp single
  desc = count_descendants(root);
}
```

Die `single`-Direktive sorgt dafür, dass lediglich einer der Threads des Teams die Funktion aufruft. Die in der Funktion erzeugten Tasks werden dann auf alle verfügbaren Threads des Teams verteilt. Da der parallele Abschnitt erst verlassen werden kann, wenn alle Threads ihre Arbeit vollendet haben, ist sichergestellt, dass nach dem Endes des Abschnitts die Variable `desc` den korrekten Wert enthält.

Als zweites Beispiel untersuchen wir die Berechnung der LR-Zerlegung einer Matrix

$$A = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix}$$

in eine linke untere Dreiecksmatrix

$$L = \begin{pmatrix} \ell_{11} & & \\ \vdots & \ddots & \\ \ell_{n1} & \dots & \ell_{nn} \end{pmatrix}$$

und eine rechte obere Dreiecksmatrix

$$R = \begin{pmatrix} r_{11} & \dots & r_{1n} \\ & \ddots & \vdots \\ & & r_{nn} \end{pmatrix}.$$

Um Speicherplatz zu sparen, sollen die Diagonalelemente der Matrix L alle gleich eins sein, so dass wir sie nicht explizit zu speichern brauchen.

Wir setzen voraus, dass eine solche Zerlegung existiert, und möchten sie möglichst schnell berechnen. Dazu verwenden wir eine *Blockzerlegung* der Matrix A: Wir wählen ein $p \in \mathbb{N}$ so, dass $m := n/p \in \mathbb{N}$ gilt und zwei $m \times m$-Matrizen gut in den Cache des verwendeten Prozessors passen. Nun zerlegen wir unsere Matrizen in $m \times m$-Teilmatrizen A_{ij}, L_{ij} und R_{ij} für $i, j \in [1 : p]$, so dass

$$A = \begin{pmatrix} A_{11} & \dots & A_{1p} \\ \vdots & \ddots & \vdots \\ A_{p1} & \dots & A_{pp} \end{pmatrix}, \quad L = \begin{pmatrix} L_{11} & & \\ \vdots & \ddots & \\ L_{p1} & \dots & L_{pp} \end{pmatrix}, \quad R = \begin{pmatrix} R_{11} & \dots & R_{1p} \\ & \ddots & \vdots \\ & & R_{pp} \end{pmatrix}$$

gelten. Aus $A = LR$ folgt

$$\begin{pmatrix} A_{11} & \dots & A_{1p} \\ \vdots & \ddots & \vdots \\ A_{p1} & \dots & A_{pp} \end{pmatrix} = \begin{pmatrix} L_{11} & & \\ \vdots & \ddots & \\ L_{p1} & \dots & L_{pp} \end{pmatrix} \begin{pmatrix} R_{11} & \dots & R_{1p} \\ & \ddots & \vdots \\ & & R_{pp} \end{pmatrix},$$

so dass wir insbesondere

$$A_{11} = L_{11}R_{11}, \qquad A_{1j} = L_{11}R_{1j}, \qquad A_{i1} = L_{i1}R_{11} \qquad \text{für alle } i, j \in [2:p]$$

erhalten. Für die rechten unteren $(p-1) \times (p-1)$ Blockmatrizen ergibt sich

$$\begin{pmatrix} A_{22} & \dots & A_{2p} \\ \vdots & \ddots & \vdots \\ A_{p2} & \dots & A_{pp} \end{pmatrix} = \begin{pmatrix} L_{21} \\ \vdots \\ L_{p1} \end{pmatrix} \begin{pmatrix} R_{12} & \dots & R_{1p} \end{pmatrix} + \begin{pmatrix} L_{22} & & \\ \vdots & \ddots & \\ L_{p2} & \dots & L_{pp} \end{pmatrix} \begin{pmatrix} R_{22} & \dots & R_{2p} \\ & \ddots & \vdots \\ & & R_{pp} \end{pmatrix},$$

so dass wir

$$\begin{pmatrix} A_{22} - L_{21}R_{12} & \dots & A_{2p} - L_{21}R_{1p} \\ \vdots & \ddots & \vdots \\ A_{p2} - L_{p1}R_{12} & \dots & A_{pp} - L_{p1}R_{1p} \end{pmatrix} = \begin{pmatrix} L_{22} & & \\ \vdots & \ddots & \\ L_{p2} & \dots & L_{pp} \end{pmatrix} \begin{pmatrix} R_{22} & \dots & R_{2p} \\ & \ddots & \vdots \\ & & R_{pp} \end{pmatrix}$$

erhalten. Mit den ersten drei Gleichungen können wir also die erste Blockzeile und -spalte berechnen, mit der letzten können wir die Berechnung auf die LR-Zerlegung einer kleineren Matrix mit nur $p-1$ Blockzeilen und -spalten zurückführen.

Wenn wir davon ausgehen, dass uns Funktionen

```
void lrdecomp(matrix *a);
void lsolve(const matrix *l, matrix *a);
void rsolve(const matrix *r, matrix *a);
void submul(const matrix *l, const matrix *r, matrix *a);
```

zur Verfügung stehen, die eine $m \times m$-Matrix mit ihrer LR-Zerlegung überschreiben, ein System mit einer linken unteren oder rechten oberen Dreiecksmatrix lösen oder das Produkt zweier Matrizen von einer dritten subtrahieren, können wir unseren Algorithmus in der folgenden Form darstellen:

```
for(k=0; k<p; k++) {
  lrdecomp(a[k][k]);
  for(j=k+1; j<p; j++)
    lsolve(a[k][k], a[k][j]);
  for(i=k+1; i<p; i++)
```

```
    rsolve(a[k][k], a[i][k]);
  for(i=k+1; i<p; i++)
    for(j=k+1; j<p; j++)
      submul(a[i][k], a[k][j], a[i][j]);
}
```

Im Ergebnis findet sich unterhalb der Diagonalen die Matrix L, oberhalb die Matrix R. Die Diagonalelemente gehören zu der Matrix R, die Diagonalelemente der Matrix L sind nach Voraussetzung gleich eins und brauchen deshalb nicht explizit dargestellt zu werden.

Um den Algorithmus parallelisieren zu können, müssen wir Teilaufgaben identifizieren, die unabhängig voneinander ausgeführt werden können. Die Aufrufe von `lsolve` und `rsolve` bearbeiten unterschiedliche Teilmatrizen und können deshalb parallel ausgeführt werden. Sobald die Matrizen `a[k][j]` und `a[i][k]` berechnet wurden, können wir daran gehen, die Matrizen `a[i][j]` zu aktualisieren. Die dafür erforderlichen Matrix-Multiplikationen sind wieder unabhängig voneinander und können deshalb parallelisiert werden, so dass wir zu der folgenden Implementierung gelangen:

```
for(k=0; k<p; k++) {
  #pragma omp single
  lrdecomp(a[k][k]);

  #pragma omp single
  for(j=k+1; j<p; j++)
  #pragma omp task firstprivate(j)
    lsolve(a[k][k], a[k][j]);

  #pragma omp single
  for(i=k+1; i<p; i++)
  #pragma omp task firstprivate(i)
    rsolve(a[k][k], a[i][k]);

  #pragma omp taskwait

  #pragma omp single
  for(j=k+1; j<p; j++)
    #pragma omp task private(i),firstprivate(j)
    for(i=k+1; i<p; i++)
      submul(a[i][k], a[k][j], a[i][j]);

  #pragma omp taskwait
}
```

Hier ist zu beachten, dass die `single`-Direktive eine Synchronisation beinhaltet, so dass alle Threads warten müssen, bis der Aufruf der Funktion `lrdecomp` abgeschlossen wurde.

In der `lsolve`/`rsolve`-Phase des Algorithmus' werden entsprechend erst alle `lsolve`-Tasks von einem Thread erzeugt, dann alle `rsolve`-Tasks von einem eventuell anderen, bevor veranlasst durch die `taskwait`-Direktive darauf gewartet wird, dass alle Tasks ausgeführt worden sind.

In der `submul`-Phase könnten wir auch jede Multiplikation in einem einzelnen Task abwickeln, Experimente legen aber nahe, dass es günstiger ist, jeweils eine gesamte Blockspalte von einem Task bearbeiten zu lassen. Da der Task von einem Thread auf vermutlich einem Prozessor ausgeführt wird, kann dann `a[k][j]` für die gesamte Berechnung im Cache bleiben, so dass Speicherbandbreite gespart wird.

Abhängigkeitsbeziehungen. Version 3.0 des OpenMP-Standards sieht für die Synchronisation der Tasks im Wesentlichen die `taskwait`-Direktive vor, mit der alle Sohn-Tasks eines Tasks abgeschlossen werden. Mit der Version 4.0 des OpenMP-Standards wurde eine flexiblere Alternative geschaffen: Jedem Task kann eine Liste von Variablen zugeordnet werden, die Eingaben und Ausgaben der entsprechenden Berechnung beschreiben. Wenn eine Ausgabevariable eines Tasks A die Eingabevariable eines Tasks B ist, stellt OpenMP sicher, dass Task A abgeschlossen ist, bevor Task B begonnen wird.

Im Fall unserer LR-Zerlegung können wir so alle Tasks und ihre Abhängigkeiten untereinander definieren und benötigen nur eine einzige `taskwait`-Direktive, um die vollständige Berechnung ausführen zu lassen:

```
#pragma omp single
for(k=0; k<p; k++) {
  #pragma omp task depend(inout:as[k][k]),\
                   firstprivate(k)
  lrdecomp(as[k][k]);

  for(j=k+1; j<p; j++)
    #pragma omp task depend(in:as[k][k]),\
                     depend(inout:as[k][j]),\
                     firstprivate(j,k)
    lsolve(as[k][k], as[k][j]);

  for(i=k+1; i<p; i++)
    #pragma omp task depend(in:as[k][k]),\
                     depend(inout:as[i][k]),\
                     firstprivate(i,k)
    rsolve(as[k][k], as[i][k]);

  for(j=k+1; j<p; j++)
    for(i=k+1; i<p; i++)
      #pragma omp task depend(in:as[i][k],as[k][j]),\
                       depend(inout:as[i][j]),\
                       firstprivate(i,j,k)
```

```
      submul(as[i][k], as[k][j], as[i][j]);
}
#pragma omp taskwait
```

Die gesamte äußere Schleife wird in diesem Beispiel nur von einem einzigen Threads ausgeführt, der alle Tasks definiert. Anders als im vorigen Beispiel erfolgt die Synchronisation über die `depend`-Klausel der Tasks, in der wir jeweils angeben, von welchen Teilmatrizen ein Task abhängt und welche Teilmatrizen aktualisiert werden. Dabei werden mit `in` Eingabevariablen bezeichnet, mit `out` Ausgabevariablen, und mit `inout` Variablen, die sowohl Ein- als auch Ausgabevariablen sind, weil sie im Zuge der Berechnung verändert werden. Falls mehrere Tasks dieselbe Variable als `inout` kennzeichnen, legt die Reihenfolge, in denen die Tasks zur Laufzeit erzeugt werden, fest, welche Abhängigkeiten gelten. Falls mehrere Threads parallel neue Tasks definieren, können dadurch nicht-deterministische Abhängigkeiten entstehen, so dass wir unseren Algorithmus entsprechend konstruieren müssen.

In unserem Beispiel ist durch die `single`-Direktive sichergestellt, dass nur ein Thread alle Tasks der Reihe nach anlegt, so dass immer dieselben Abhängigkeiten erzeugt werden.

6 Anwendungsbeispiel: Gravitationsfeld

Multi-Threading bietet uns die Möglichkeit, relativ komplexe Algorithmen zu parallelisieren, bei denen vielfältige Operationen in komplexer Weise wechselwirken. Als Beispiel untersuchen wir die Berechnung des von einer großen Zahl von Planeten oder Sonnen hervorgerufenen Gravitationsfelds.

6.1 Gravitationsfeld

Unsere Aufgabe besteht darin, das von $n \in \mathbb{N}$ im Raum verteilten Sonnen erzeugte Gravitationsfeld zu berechnen. Für jedes $i \in \{1, \ldots, n\}$ bezeichnen wir mit $x_i \in \mathbb{R}^3$ die Position der i-ten Sonne und mit $m_i \in \mathbb{R}_{\geq 0}$ ihre Masse.

Wir interessieren uns für die Kräfte, mit denen sich diese Sonnen gegenseitig anziehen. Das *Newton'sche Gravitationsgesetz* besagt, dass die j-te Sonne die i-te Sonne mit der Kraft

$$f_{ij} := \gamma m_i m_j \frac{x_j - x_i}{\|x_j - x_i\|_2^3}$$

anzieht. Hier ist $\gamma \in \mathbb{R}_{>0}$ die *Gravitationskonstante*, während

$$\|z\|_2 := \sqrt{z_1^2 + z_2^2 + z_3^2} \qquad \text{für alle } z \in \mathbb{R}^3$$

den *euklidischen Abstand* angibt.

Das *Superpositionsprinzip* besagt, dass die von *allen* Sonnen ausgeübte Kraft sich durch Addition der einzelnen Kräfte ergibt, also durch

$$f_i := \gamma m_i \sum_{\substack{j=1 \\ j \neq i}}^{n} m_j \frac{x_j - x_i}{\|x_j - x_i\|_2^3} \qquad \text{für alle } i \in \{1, \ldots, n\}.$$

Analog zu der im Fall der Wellengleichung verwendeten Vorgehensweise können wir auch diese Gleichung mit dem Leapfrog-Verfahren kombinieren, um die Bewegung der Sonnen zu simulieren.

Allerdings ist der dabei anfallende Rechenaufwand erheblich höher: Bei der Wellengleichung hing die in einem Punkt wirkende Kraft nur von seinen unmittelbaren Nachbarn ab. Im Fall der Gravitation hängt die auf eine Sonne wirkende Kraft dagegen von *allen* anderen Sonnen ab.

Wenn wir die Gleichung direkt programmieren würden, wären $n(n-1)$ einzelne Kräfte zu berechnen und aufzusummieren, und der resultierende Rechenaufwand würde sehr schnell die Möglichkeiten konventioneller Rechner übersteigen.

Deshalb kommen in der Praxis Algorithmen zum Einsatz, die die exakte Berechnung der Kräfte durch eine Näherung ersetzen, die wesentlich schneller bestimmt werden kann.

6.2 Potential

Ein erster Schritt zur Reduktion des Rechenaufwands besteht darin, die Funktion

$$f(x,y) := \frac{y-x}{\|y-x\|_2^3}$$

in eine etwas handlichere Form zu bringen. Wir definieren dazu die Funktion

$$g : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}, \qquad (x,y) \mapsto \begin{cases} \frac{1}{\|y-x\|_2} & \text{falls } x \neq y, \\ 0 & \text{ansonsten,} \end{cases}$$

und stellen fest, dass ihre partiellen Ableitungen für $k \in \{1,2,3\}$ durch

$$\begin{aligned} \frac{\partial g}{\partial x_k}(x,y) &= \frac{\partial}{\partial x_k}((y_1-x_1)^2 + (y_2-x_2)^2 + (y_3-x_3)^2)^{-1/2} \\ &= -\frac{1}{2}((y_1-x_1)^2 + (y_2-x_2)^2 + (y_3-x_3)^2)^{-3/2} 2(y_k-x_k)(-1) \\ &= \frac{y_k-x_k}{\|y-x\|_2^3} \qquad \text{für alle } x,y \in \mathbb{R}^3,\ x \neq y \end{aligned}$$

gegeben sind. Das ist gerade die k-te Komponente der Funktion f.

Zur Abkürzung definiert man den *Gradienten*

$$\nabla_x g(x,y) := \begin{pmatrix} \frac{\partial g}{\partial x_1}(x,y) \\ \frac{\partial g}{\partial x_2}(x,y) \\ \frac{\partial g}{\partial x_3}(x,y) \end{pmatrix} \qquad \text{für alle } x,y \in \mathbb{R}^3,\ x \neq y$$

und kann die von uns soeben gewonnene Gleichung kompakt als

$$f(x,y) = \nabla_x g(x,y) \qquad \text{für alle } x,y \in \mathbb{R}^3,\ x \neq y$$

schreiben. Für die Gravitationskraft ergibt sich

$$\begin{aligned} f_i &= \gamma m_i \sum_{\substack{j=1 \\ j\neq i}}^{n} m_j f(x_i,x_j) = \gamma m_i \sum_{\substack{j=1 \\ j\neq i}}^{n} m_j \nabla_x g(x_i,x_j) \\ &= \gamma m_i \nabla_x \left(\sum_{j=1}^{n} m_j g(x_i,x_j) \right) \qquad \text{für alle } i \in \{1,\ldots,n\}. \end{aligned}$$

Diese Gleichung eröffnet uns die Möglichkeit, statt mit den dreidimensionalen Vektoren f_{ij} mit einfachen skalaren Größen zu rechnen: Wir definieren das *Gravitationspotential*

$$\varphi(x) := \sum_{j=1}^{n} m_j g(x,x_j) \qquad \text{für alle } x \in \mathbb{R}^3 \tag{6.1}$$

und erhalten

$$f_i = \gamma m_i \nabla_x \varphi(x_i) \qquad \text{für alle } i \in \{1,\ldots,n\}.$$

6.3 Approximation

Um den Rechenaufwand weiter zu reduzieren greifen wir auf eine Näherung zurück: Falls die Punkte x_i und x_j hinreichend weit voneinander entfernt sind, können wir x_j an eine Position $\tilde{x}_j$ verschieben, ohne dass sich das Gravitationsfeld stark ändert.

Diese Eigenschaft können wir mathematisch quantifizieren: Es gilt

$$\begin{aligned} g(x_i, x_j) - g(x_i, \tilde{x}_j) &= \frac{1}{\|x_j - x_i\|_2} - \frac{1}{\|\tilde{x}_j - x_i\|_2} = \frac{\|\tilde{x}_j - x_i\|_2 - \|x_j - x_i\|_2}{\|x_j - x_i\|_2 \|\tilde{x}_j - x_i\|_2} \\ &\leq \frac{\|\tilde{x}_j - x_j\|_2 + \|x_j - x_i\|_2 - \|x_j - x_i\|_2}{\|x_j - x_i\|_2 \|\tilde{x}_j - x_i\|_2} = \frac{\|\tilde{x}_j - x_j\|_2}{\|x_j - x_i\|_2 \|\tilde{x}_j - x_i\|_2}, \end{aligned}$$

und analog lässt sich auch

$$g(x_i, \tilde{x}_j) - g(x_i, x_j) \leq \frac{\|\tilde{x}_j - x_j\|_2}{\|x_j - x_i\|_2 \|\tilde{x}_j - x_i\|_2},$$

zeigen, so dass wir insgesamt

$$|g(x_i, x_j) - g(x_i, \tilde{x}_j)| \leq \frac{\|\tilde{x}_j - x_j\|_2}{\|x_j - x_i\|_2 \|\tilde{x}_j - x_i\|_2} \tag{6.2}$$

erhalten. Solange also der Abstand zwischen x_j und $\tilde{x}_j$ in einem günstigen Verhältnis zu dem Abstand beider Punkte zu x_i steht, wird die Verschiebung des Punkts keine allzu große Veränderung des Felds bewirken.

Um mit dieser Beobachtung den Rechenaufwand zu reduzieren, wählen wir eine Menge $s \subseteq \mathbb{R}^3$, die hinreichend weit von x_i entfernt ist, und verschieben *sämtliche* in ihr enthaltenen Sonnen in einen gemeinsamen Punkt $x_s \in s$. In diesem Punkt entsteht so eine einzige „virtuelle Sonne“, die ungefähr dasselbe Gravitationsfeld wie die in s enthaltenen Sonnen erzeugt.

Der Einfachheit halber verwenden wir für s achsenparallele Quader der Form

$$s = [a_1, b_1] \times [a_2, b_2] \times [a_3, b_3]$$

und wählen als Position der virtuellen Sonne den Mittelpunkt

$$x_s := \begin{pmatrix} (b_1 + a_1)/2 \\ (b_2 + a_2)/2 \\ (b_3 + a_3)/2 \end{pmatrix}.$$

Die Menge der Indizes der in s befindlichen Sonnen bezeichnen wir mit

$$\hat{s} := \{j \in \{1, \ldots, n\} \ : \ x_j \in s\},$$

und unsere Näherung nimmt die Form

$$\sum_{j \in \hat{s}} m_j g(x, x_j) \approx \sum_{j \in \hat{s}} m_j g(x, x_s) = m_s g(x, x_s), \qquad m_s := \sum_{j \in \hat{s}} m_j \tag{6.3}$$

an. Auch falls s Millionen von Sonnen enthalten sollte, können wir das Gravitationsfeld in hinreichend großer Entfernung durch eine einzige virtuelle Sonne approximieren, deren Masse gerade der Summe ihrer Massen entspricht.

Diese Technik führt allerdings nur zu guten Ergebnissen, falls s hinreichend weit von x entfernt ist. Um entscheiden zu können, wann dieser Fall vorliegt, definieren wir den Durchmesser

$$\begin{aligned}\operatorname{diam}(s) &:= \max\{\|x-y\|_2 \ : \ x,y \in s\} \\ &= \sqrt{(b_1-a_1)^2+(b_2-a_2)^2+(b_3-a_3)^2}\end{aligned}$$

und den Abstand

$$\begin{aligned}\operatorname{dist}(x,s) &:= \min\{\|x-y\|_2 \ : \ y \in s\} \\ &= \sqrt{\operatorname{dist}(x_1,[a_1,b_1])^2+\operatorname{dist}(x_2,[a_2,b_2])^2+\operatorname{dist}(x_3,[a_3,b_3])^2},\end{aligned}$$

$$\operatorname{dist}(x_k,[a_k,b_k]) := \begin{cases} a_k - x_k & \text{falls } x_k \le a_k, \\ x_k - b_k & \text{falls } b_k \le x_k, \\ 0 & \text{ansonsten,} \end{cases}$$

und verwenden die *Zulässigkeitsbedingung*

$$\operatorname{diam}(s) \le \operatorname{dist}(x,s). \tag{6.4}$$

Indem wir in (6.2) einsetzen und $\|x_s - x_j\|_2 \le \operatorname{diam}(s)/2$ ausnutzen, erhalten wir

$$|g(x,x_j)-g(x,x_s)| \le \frac{\operatorname{diam}(s)/2}{\operatorname{dist}(x,s)^2} \le \frac{1}{2\operatorname{dist}(x,s)} \qquad \text{für alle } j \in \hat{s},$$

dürfen also in hinreichend weiter Entfernung von x auf eine hinreichend gute Näherung hoffen.

Unser Algorithmus arbeitet rekursiv: Für einen gegebenen Quader s prüfen wir, ob die Zulässigkeitsbedingung (6.4) erfüllt ist. In diesem Fall können wir (6.3) verwenden, um das Potential für alle $j \in \hat{s}$ auszuwerten.

Anderenfalls unterteilen wir s in Teilquader, mit denen wir rekursiv fortfahren. Der Einfachheit halber unterteilen wir s immer quer zu einer Koordinatenachse: Wir wählen $k \in \{1,2,3\}$ und definieren den Mittelpunkt des Intervalls $[a_k,b_k]$ durch $c_k := (b_k+a_k)/2$. Dann konstruieren wir die Teilquader

$$\begin{aligned}
s_1 &:= [a_1,c_1]\times[a_2,b_2]\times[a_3,b_3], & s_2 &:= [c_1,b_1]\times[a_2,b_2]\times[a_3,b_3] && \text{falls } k=1,\\
s_1 &:= [a_1,b_1]\times[a_2,c_2]\times[a_3,b_3], & s_2 &:= [a_1,b_1]\times[c_2,b_2]\times[a_3,b_3] && \text{falls } k=2,\\
s_1 &:= [a_1,b_1]\times[a_2,b_2]\times[a_3,c_3], & s_2 &:= [a_1,b_1]\times[a_2,b_2]\times[c_3,b_3] && \text{falls } k=3.
\end{aligned}$$

Die korrespondierenden Indexmengen ergeben sich durch

$$\hat{s}_1 := \{j \in \hat{s} \ : \ x_{j,k} \le c_k\}, \qquad \hat{s}_2 := \{j \in \hat{s} \ : \ c_k < x_{j,k}\}.$$

Die Rekursion wird beendet, sobald wir zulässige Quader finden oder $\hat{s}$ so klein geworden ist, dass wir das Potential direkt auswerten können.

6.4 Clusterbaum

Unsere bisherige Konstruktion hat den Vorteil, dass sich das Gravitationspotential effizient auswerten lässt, falls alle Gebiete s, deren Mittelpunkt x_s und die Massen m_s der virtuellen Sonnen bereits vorliegen. Die Berechnung der Masse m_s über (6.3) hätte allerdings wieder einen relativ hohen Aufwand.

Falls wir das Potential nur in einem einzigen Punkt auswerten wollten, würde uns dieser Ansatz keinen Vorteil bieten. Allerdings müssen wir das Potential in n Punkten auswerten, so dass wir einen „Vorrat“ an Gebieten vorberechnen und dann für alle Punkte verwenden können.

In diesem Kontext nennen wir die Gebiete s *Cluster*, die in einem *Clusterbaum* organisiert werden, der unsere Zerlegungsstrategie widerspiegelt. Ein Cluster kann durch die folgende Datenstruktur dargestellt werden:

```
typedef struct _cluster cluster;
struct _cluster {
  double a[3];
  double b[3];

  double s[3];
  double ms;

  int n;
  double (*x)[3];
  double *m;

  int sons;
  cluster *son;
};
```

Hier geben `a` und `b` die Geometrie des Quaders an, `s` enthält die Koordinaten des Mittelpunkts x_s und `ms` seine Masse m_s. Das Feld `n` enthält die Anzahl der in s enthaltenen Punkte, die Felder `x` und `m` deren Koordinaten und Massen. Falls s unterteilt wurde, gibt `sons` die Anzahl seiner Teilquader an und `son` enthält die Zeiger auf diese Teilquader.

Den Clusterbaum konstruieren wir, indem wir zunächst einen Cluster s konstruieren, der alle Punkte enthält. Das kann beispielsweise geschehen, indem wir die minimalen und maximalen Koordinaten aller Punkte bestimmen.

Sobald ein Cluster s vorliegt, prüfen wir, ob er nur noch wenige Massen enthält. In diesem Fall beenden wir die Rekursion und setzen `s->sons` auf null.

Anderenfalls wählen wir die Koordinatenrichtung k, in der die Ausdehnung $b_k - a_k$ maximal ist, berechnen c_k, und unterteilen den Cluster mit unserer Strategie in Teilcluster s_1 und s_2, setzen `s->sons` auf zwei, und tragen Zeiger auf s_1 und s_2 in `s->son` ein. Dann fahren wir rekursiv mit s_1 und s_2 fort.

Wenn die Baumstruktur vorliegt, können wir daran gehen, die Punkte x_s und die Massen m_s zu berechnen. Für letztere können wir wieder rekursiv vorgehen: Falls die

Massen m_{s_1} und m_{s_2} für die beiden Teilcluster eines Clusters s berechnet sind, gilt

$$m_s = \sum_{j \in \hat{s}} m_j = \sum_{j \in \hat{s}_1} m_j + \sum_{j \in \hat{s}_2} m_j = m_{s_1} + m_{s_2},$$

so dass wir lediglich in den Blättern des Clusterbaums auf m_j zurückzugreifen brauchen und der Rechenaufwand für alle anderen Cluster sehr gering ausfällt.

Sobald der Clusterbaum vollständig aufgestellt wurde, können wir das Potential für ein gegebenes y in der beschriebenen Weise auswerten:

```
double
eval_cluster(const cluster *s, double y[3])
{
  double phi;
  int i;

  if(diam_cluster(s) <= dist_cluster(y, s))
    phi = s->ms * g(y, s->xs);
  else if(s->sons > 0) {
    for(phi=0.0, i=0; i<s->sons; i++)
      phi += eval_cluster(s->son[i], y);
  }
  else {
    for(phi=0.0, i=0; i<s->n; i++)
      phi += s->m[i] * g(y, s->x[i]);
  }
  return phi;
}
```

Unter geeigneten Voraussetzungen lässt sich zeigen, dass diese Auswertung lediglich $\mathcal{O}(\log(n))$ Operationen erfordert, also deutlich schneller als die direkte Auswertung ist, die n Summanden berechnen muss.

Man kann unter denselben Voraussetzungen auch zeigen, dass das Aufstellen des Clusterbaums $\mathcal{O}(n \log(n))$ Operationen erfordert, so dass für n Auswertungen insgesamt auch nur $\mathcal{O}(n \log(n))$ Operationen anfallen.

7 Verteiltes Rechnen

Bei einem Vektorrechner verfügt ein Prozessor über mehrere Rechenwerke, die gleichzeitig dieselbe Operation auf mehreren Daten ausführen. Bei einem Rechner mit geteiltem Speicher können mehrere Prozessoren gleichzeitig unterschiedliche Operationen ausführen, bei denen sie auf die Daten aller anderen Prozessoren zugreifen können, da der Hauptspeicher gemeinsam genutzt wird. Bei einem *verteilten Rechner* sind die Prozessoren wesentlich unabhängiger: Jeder Prozessor hat seinen eigenen Hauptspeicher, der mit keinem anderen Prozessor geteilt wird. Die Kommunikation erfolgt in der Regel über ein Netzwerk, in dem jeder Datenaustausch explizit in Auftrag gegeben werden muss.

Verteilte Rechner bieten den großen Vorteil, dass sie sich sehr einfach konstruieren lassen: Bereits zwei PCs, die an einem Ethernet-Router angeschlossen sind, können als verteilter Rechner behandelt werden. Im Bereich des Hochleistungsrechnens kommen eher Systeme aus Tausenden von Rechnern zum Einsatz, die über besonders schnelle Netzwerke verbunden sind. Die Systeme können sogar auf mehrere Standorte weltweit verteilt und lediglich per Internet verbunden sein.

Da die einzelnen *Knoten* eines Rechnernetzes sehr kostengünstig hergestellt werden können und sich deshalb eine sehr hohe kumulative Rechenleistung mit geringem finanziellen Aufwand erreichen lässt, dominieren verteilte Rechner derzeit in der Welt des Hochleistungsrechnens.

Allerdings stellen derartige Rechner die Programmierer vor erhebliche Herausforderungen: Programme müssen jeden Datenaustausch explizit anfordern, in der Regel sowohl auf Seiten des Senders als auch des Empfängers. Da der Datenaustausch über Kommunikationsnetzwerke in der Regel deutlich langsamer als der innerhalb eines Rechners ist, muss bei der Auswahl und der Implementierung von Algorithmen sorgfältig darauf geachtet werden, die Menge der auszutauschenden Daten möglichst gering zu halten oder zumindest dafür zu sorgen, dass durch das Warten auf Daten entstehende Latenzzeiten sinnvoll genutzt werden.

Der am weitesten verbreitete Standard für die Programmierung von verteilten Rechnern trägt den Namen MPI (*message passing interface*).

7.1 MPI-Programme

Programme, die MPI verwenden, arbeiten anders als die, die wir bisher behandelt haben. Es beginnt schon damit, dass sie nicht mit dem Standard-Compiler eines Systems übersetzt werden, sondern mit dem Befehl `mpicc`, der letzten Endes zwar auch einen Compiler aufruft, allerdings mit zusätzlichen Parametern. Im Bereich der Cluster-Computer ist es nicht unüblich, dass der Compiler auf einer anderen Rechnerarchitektur

läuft als die eigentlichen Knoten des verteilten Rechners, so dass unter Umständen ein Cross-Compiler zum Einsatz kommt, der Maschinencode für eine andere Architektur als diejenige erzeugt, auf der er selbst läuft.

Ausführung. MPI-Programme werden in der Regel auch nicht direkt von der Kommandozeile aus gestartet, sondern entweder über den Befehl `mpirun` oder über ein aufwendigeres *Batch-System*, das die einzelnen zu startenden Programme je nach Priorität in Warteschlangen einsortiert und startet, sobald die nötigen Ressourcen zur Verfügung stehen. Das MPI-Programm läuft anschließend auf einer Reihe von Rechnerknoten, die Kommunikation der einzelnen Prozesse erfolgt über Funktionen einer MPI-Programmbibliothek.

Initialisierung. Manche Implementierungen des MPI-Standards teilen den gestarteten Prozessen über Befehlszeilenparameter mit, in welchem Kontext sie ausgeführt werden. Diese Parameter wertet das MPI-Programm aus, wenn die Funktion `MPI_Init` aufgerufen wird. Diese Funktion muss genau einmal aufgerufen werden, bevor andere MPI-Funktionen zum Einsatz kommen. Sie erhält die Parameter, die der Funktion `main` übergeben wurden, verarbeitet diejenigen davon, die für die Initialisierung der MPI-Bibliothek benötigt werden, und entfernt sie dann aus der Parameterliste. Ein besonders einfaches MPI-Programm hat die folgende Gestalt:

```
#include <mpi.h>

int
main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);
  MPI_Finalize();
  return 0;
}
```

Wie auf Unix-Systemen üblich enthält `argc` die Anzahl der Befehlszeilenparameter, während `argv` ein Array von Zeigern auf die einzelnen Parameter ist. Die Funktion `MPI_Init` erhält mit dem `&`-Operator erzeugte Zeiger auf beide Variablen, damit sie sie anpassen kann.

Programmende. Damit das MPI-Programm Aufräumarbeiten erledigen kann,bevor es beendet wird, beispielsweise um Hilfsspeicher freizugeben oder Netzwerkverbindungen zu schließen, muss vor dem Ende des Programms die Funktion `MPI_Finalize` aufgerufen werden, die sich um diese Aufgaben kümmert. Nachdem diese Funktion aufgerufen wurde, dürfen keine MPI-Funktionen mehr verwendet werden.

7.2 Gruppen von Prozessen

Die auf mehreren Rechnerknoten laufenden Prozesse sollten natürlich in der Lage sein, miteinander zu kommunizieren. Die einfachste Lösung bestünde darin, der Struktur von OpenMP zu folgen und die Prozesse zu einem „Team“ zusammenzufassen, innerhalb dessen jeder Prozess eine eindeutige Nummer besitzt, mit der er angesprochen werden kann.

Communicator und Gruppe. MPI verfolgt einen allgemeineren Ansatz: Kommunikations- und Synchronisationsfunktionen werden jeweils für einen *Communicator* (beschrieben durch ein Object des Typs `MPI_Comm`) ausgeführt. Unter anderem gehört zu einem Communicator eine Gruppe von Prozessen (beschrieben durch den Typ `MPI_Group`), auf die sich diese Funktionen auswirken. Diese Prozesse sind fortlaufend numeriert, so dass wir, ähnlich wie in OpenMP, jeden Prozess mit seiner Nummer ansprechen können.

Anders als in OpenMP können wir Teilmengen der Prozessgruppe definieren und aus ihnen neue Communicators konstruieren. Das ist nützlich, um beispielsweise nur einen Teil der an einer Berechnung beteiligten Prozesse zu synchronisieren oder um einen Teil der zu lösenden Aufgabe nur von einem Teil der Rechnerknoten behandeln zu lassen, während der Rest einen anderen Teil bearbeitet.

Wenn ein MPI-Programm gestartet wird, sind zwei Communicators bereits definiert: `MPI_COMM_WORLD` enthält alle Prozesse, in denen das Programm gestartet wurde, `MPI_COMM_SELF` enthält nur den gerade laufenden Prozess.

Größe und Rang. Die Anzahl der in einer Gruppe zusammengefassten Prozesse können wir mit der Funktion `MPI_Group_size` erfahren, während wir die Nummer des aktuellen Prozesses innerhalb einer Gruppe mit `MPI_Group_rank` abfragen können.

Zu jedem Communicator gehört genau eine Prozessgruppe, die wir mit der Funktion `MPI_Comm_group` erhalten können.

Zur Abkürzung können wir die Größe dieser Gruppe und die Nummer des aktuellen Prozesses innerhalb der Gruppe mit den Funktionen `MPI_Comm_size` und `MPI_Comm_rank` direkt ermitteln:

```
int
main(int argc, char **argv)
{
  int size, rank;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Number %d of %d\n", rank, size);
  MPI_Finalize();
  return 0;
}
```

Beispiel: Linearkombination. Wenn wir als Beispiel wieder eine Linearkombination zweier Vektoren berechnen wollen, müssen wir berücksichtigen, dass wir mit *verteiltem* Speicher arbeiten, dass also jeder Prozess nur den Teil der Vektoren abspeichern sollte, für den er zuständig ist.

Dieses Problem lässt sich besonders einfach lösen, indem wir für jeden Prozess eine Variable `first` berechnen, die den ersten Index enthält, für den er zuständig ist, und eine Variable `last`, die den ersten Index angibt, für den der nächste Prozess zuständig ist. Unser Prozess muss die `last-first` Elemente `first`, ...,`last-1` speichern:

```
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

first = n * rank / size;
last = n * (rank+1) / size;

x = (float *) malloc((size_t) sizeof(float) * (last-first));
y = (float *) malloc((size_t) sizeof(float) * (last-first));

for(i=first; i<last; i++)
  x[i-first] += alpha * y[i-first];
```

7.3 Punkt-zu-Punkt-Kommunikation

Das vorige Beispiel ist relativ praxisfern („*embarrassingly parallel*“), da jeder Prozess für die Bearbeitung seines Teils der Gesamtaufgabe nur die Daten benötigt, über die er selber verfügt. Wesentlich typischer sind Aufgaben, bei denen die Prozesse Daten austauschen müssen, um die Gesamtaufgabe zu lösen. Wir untersuchen zunächst nur einfache Situationen, in denen zwei Prozesse Daten austauschen. Der MPI-Standard bezeichnet diese Kommunikationsoperationen als *Punkt-zu-Punkt-Kommunikation*.

Beispiel: Wellengleichung. Als Beispiel wählen wir die eindimensionale Wellengleichung, bei der für die Berechnung der aktualisierten Geschwindigkeit einer Masse nicht nur die Auslenkung dieser Masse, sondern auch die ihrer beiden Nachbarn benötigt wird. Falls diese Nachbarmassen in die Zuständigkeit eines anderen Prozesses fallen, müssen ihre aktuellen Auslenkungen von diesem Prozess an den aktuellen übermittelt werden, damit wir die korrekten Geschwindigkeiten berechnen können.

Erweitertes Gebiet. Ein eleganter und einfacher Ansatz besteht darin, unsere Arrays um Kopien der unmittelbar benachbarten Massen zu erweitern: Ein Prozess soll die Massen `first` bis `last` berechnen, aber er darf dabei auch auf `pfirst=first-1` und `plast=last+1` zugreifen. Wenn die beweglichen Massen wieder von `1` bis `n` durchnumeriert sind, erhalten wir

```
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

first = n * rank / size + 1;
last = n * (rank+1) / size + 1;
pfirst = first - 1;
plast = last;

x = (float *) malloc((size_t) sizeof(float) * (plast-pfirst+1));
v = (float *) malloc((size_t) sizeof(float) * (plast-pfirst+1));
```

Das Array `x` soll nun die Auslenkungen für die Massen `pfirst` bis `plast` aufnehmen, `v` die Geschwindigkeiten. Anders als im vorigen Beispiel ist auch Speicher für das Element `plast` eingeplant, damit wir Randpunkte elegant behandeln können.

Empfangen von Nachrichten. Bevor unser Prozess die Arbeit aufnehmen kann, muss er die Auslenkung der Masse `pfirst` von dem Prozess `rank-1` erhalten sowie die Auslenkung der Masse `plast` von dem Prozess `rank+1`. Da der erste Eintrag des Arrays `x` zu der Masse `pfirst` gehört, sind also `x[pfirst-pfirst]` und `x[plast-pfirst]` zu empfangen. Für das Empfangen von Nachrichten sieht MPI die Funktion `MPI_Recv` vor:

```
MPI_Recv(x+(pfirst-pfirst), 1, MPI_FLOAT,
         rank-1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Recv(x+(plast-pfirst), 1, MPI_FLOAT,
         rank+1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Das erste Argument ist ein Zeiger auf das Array, in dem die empfangenen Daten abgelegt werden sollen, das zweite gibt die Anzahl der Daten an, das dritte den Typ.

Das vierte Argument ist der Absender, von dem die Daten erwartet werden, das fünfte eine zusätzliche Markierung, die gleich erklärt wird, das sechste der Communicator, auf den sich die Kommunikationsoperation bezieht, und das siebte ein Zeiger auf ein Objekt des Typs `MPI_Status`, mit dem sich das Ergebnis der Operation genauer überprüfen ließe. Wir verwenden die vordefinierte Konstante `MPI_STATUS_IGNORE`, um zu signalisieren, dass wir daran kein Interesse haben.

Umschläge. Die Nachrichten, die per MPI zwischen Prozessen ausgetauscht werden, werden mit zusätzlichen Informationen versehen. In Analogie zu Briefen spricht man von dem *Umschlag* der Nachricht. Auf dem Umschlag sind eine Reihe von Informationen vermerkt, die für die Übertragung und Zuordnung der Nachricht von Bedeutung sind:

- Absender der Nachricht (engl. *source*),
- Empfänger der Nachricht (engl. *destination*),
- Zusätzliche Markierung (engl. *tag*),
- Communicator.

Der Aufruf von `MPI_Recv` empfängt eine Nachricht nur dann, wenn ihr Umschlag zu seinen Parametern passt. Wir können allerdings statt des Absenders auch `MPI_ANY_SOURCE` verwenden, um eine Nachricht von einem beliebigen Absender zu akzeptieren, und statt der Markierung auch `MPI_ANY_TAG`, um Nachrichten mit jeder beliebigen Markierung zu akzeptieren.

Senden von Nachrichten. Die Funktion `MPI_Recv` ist *blockierend*, das Programm kehrt also erst aus ihr zurück, wenn eine Nachricht empfangen wurde. Wir müssen also dafür sorgen, dass unsere Prozesse auch Daten senden, weil sonst niemals etwas berechnet werden würde. Dafür gibt es die Funktion `MPI_Send`:

```
MPI_Send(x+(first-pfirst), 1, MPI_FLOAT,
         rank-1, 0, MPI_COMM_WORLD);
MPI_Send(x+(last-pfirst), 1, MPI_FLOAT,
         rank+1, 0, MPI_COMM_WORLD);
```

Der erste Aufruf überträgt die zu der Masse `first` gehörende Auslenkung an Prozess `rank-1`, der zweite die zu der Masse `last` gehörende an Prozess `rank+1`.

Bei dieser Vorgehensweise stoßen wir allerdings auf ein erhebliches Problem: `MPI_Recv` blockiert die Programmausführung, bis Daten empfangen wurden. `MPI_Send` kann (muss aber nicht) die Programmausführung blockieren, bis die Daten gesendet wurden. Da Prozesse mit diesen Funktionen nicht gleichzeitig senden und empfangen können, ist es nötig, eine geeignete Strategie zu entwickeln, um den erfolgreichen Datenaustausch zu gewährleisten.

Fallunterscheidung. Da wir es mit einem lediglich eindimensionalen Problem zu tun haben, können wir das Problem lösen, indem wir die Prozesse in solche mit geraden und ungeraden Nummern unterteilen: Falls `rank` geradzahlig ist, sind `rank-1` und `rank+1` ungeradzahlig. Wir können festlegen, dass geradzahlige Prozesse zuerst senden und dann empfangen, während ungeradzahlige zuerst empfangen und dann senden:

```
if(rank % 2 == 0) {
  if(rank > 0)      MPI_Send(x+(first-pfirst), 1, MPI_FLOAT,
                             rank-1, 0, MPI_COMM_WORLD);
  if(rank+1 < size) MPI_Send(x+(last-pfirst), 1, MPI_FLOAT,
                             rank+1, 0, MPI_COMM_WORLD);
  if(rank > 0)      MPI_Recv(x+(pfirst-pfirst), 1, MPI_FLOAT, rank-1,
                             0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  if(rank+1 < size) MPI_Recv(x+(plast-pfirst), 1, MPI_FLOAT, rank+1,
                             0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
else {
  if(rank > 0)      MPI_Recv(x+(pfirst-pfirst), 1, MPI_FLOAT, rank-1,
                             0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  if(rank+1 < size) MPI_Recv(x+(plast-pfirst), 1, MPI_FLOAT, rank+1,
```

```
                              0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  if(rank > 0)       MPI_Send(x+(first-pfirst), 1, MPI_FLOAT,
                              rank-1, 0, MPI_COMM_WORLD);
  if(rank+1 < size) MPI_Send(x+(last-pfirst), 1, MPI_FLOAT,
                              rank+1, 0, MPI_COMM_WORLD);
}
```

Nicht-blockierendes Senden. Für allgemeine Anwendungen könnte es schwierig werden, eine einfache Strategie auf Grundlage der Prozessnummer zu formulieren. Unser Problem tritt auf, weil sowohl das Senden der Nachricht als auch das Empfangen blockierende Operationen sind, das Programm also erst aus ihnen zurückkehrt, wenn die Operationen erfolgreich abgeschlossen wurden. Also verwenden wir *nicht-blockierende Operationen*: Mit der Funktion `MPI_Isend` können wir dem System mitteilen, dass eine Nachricht vorliegt, die bei Gelegenheit abgeholt werden kann. Nachdem die MPI-Bibliothek sich eine entsprechende Notiz gemacht hat, kehrt das Programm aus der Funktion zurück, so dass wir uns beispielsweise um das Empfangen einer Nachricht kümmern können. Das eigentliche Versenden der Nachricht erfolgt unabhängig von unserem Programm, beispielsweise in einem separaten Thread oder sogar in separater Hardware.

Wir sollten sicherstellen, dass die zu verschickenden Werte ihren Empfänger erreicht haben, bevor wir sie mit den Werten des nächsten Zeitschritts überschreiben. Aus diesem Grund erhalten wir bei einer nicht-blockierenden Operation ein Objekt des Typs `MPI_Request`, mit dessen Hilfe wir prüfen können, ob die Operation schon abgeschlossen wurde. Wir können sogar mit einer blockierenden Funktion `MPI_Wait` darauf warten, dass sie abgeschlossen wurde:

```
MPI_Request left, right;
if(rank > 0)
  MPI_Isend(x+(first-pfirst), 1, MPI_FLOAT,
            rank-1, 0, MPI_COMM_WORLD, &left);
if(rank+1 < size)
  MPI_Isend(x+(last-pfirst), 1, MPI_FLOAT,
            rank+1, 0, MPI_COMM_WORLD, &right);
if(rank > 0)
  MPI_Recv(x+(pfirst-pfirst), 1, MPI_FLOAT,
           rank-1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
if(rank+1 < size)
  MPI_Recv(x+(plast-pfirst), 1, MPI_FLOAT,
           rank+1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
if(rank > 0)
  MPI_Wait(&left, MPI_STATUS_IGNORE);
if(rank+1 < size)
  MPI_Wait(&right, MPI_STATUS_IGNORE);
```

Dieser Ansatz ist nicht nur flexibel, er bietet auch den Vorteil, dass er sehr effizient umgesetzt werden kann: Eine MPI-Implementierung könnte beispielsweise eine mit `MPI_Isend`

vorbereitete Nachricht solange nicht weiter bearbeiten, bis ein passender Aufruf der Funktion `MPI_Recv` vorliegt. Dieser Aufruf könnte dann veranlassen, dass die Daten direkt aus dem Speicher des einen Prozesses in den des anderen kopiert werden.

Verstecken von Latenzen. Ein weiterer Vorteil der nicht-blockierenden Kommunikation besteht darin, dass sie es uns ermöglicht, die Zeit zu nutzen, in der die Daten übertragen werden. Beispielsweise könnten wir eine nicht-blockierende Leseoperation verwenden, um die Daten `x[pfirst]` und `x[plast]` zu empfangen, während die von diesen Daten nicht abhängenden Größen `v[first+1]`,...,`v[last-1]` berechnet werden. Da für den Datenaustausch in der Regel andere Teile der Hardware zuständig sind als für Gleitkommaoperationen, kann eine gute MPI-Implementierung beide Aufgaben parallel ausführen, so dass sich so bei der Kommunikation auftretende Latenzen „verstecken“ lassen.

Status einer Nachricht. Bisher haben wir als letztes Argument bei den Funktionen `MPI_Recv` und `MPI_Wait` jeweils `MPI_STATUS_IGNORE` verwendet, also auf eine Meldung über das Ergebnis der Empfangsoperation verzichtet. In der Praxis könnte es durchaus sinnvoll sein, zu erfahren, wieviele Daten empfangen wurden und von wem sie stammen (falls beispielsweise `MPI_ANY_SOURCE` oder `MPI_ANY_TAG` benutzt wurden). Für diesen Zweck gibt es Objekte des Typs `MPI_Status`, die als `struct` mit (mindestens) den Feldern `MPI_SOURCE`, `MPI_TAG` und `MPI_ERROR` umgesetzt sind. Außerdem gibt es die Funktion `MPI_Get_count`, mit der wir ermitteln können, wieviele Daten tatsächlich übertragen wurden:

```
MPI_Status status;
int count;

MPI_Recv(buf, 10, MPI_FLOAT,
         MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

MPI_Get_count(&status, MPI_FLOAT, &count);

printf("Received %d floats from %d with tag %d, error code %d\n",
       count, status.MPI_SOURCE, status.MPI_TAG, status.MPI_ERROR);
```

Kommunikationsmodi. Bei einem Aufruf der Funktion `MPI_Recv` ist klar, dass das Programm erst aus dieser Funktion zurückkehrt, wenn die zu empfangenden Daten vollständig angekommen sind.

Bei der Funktion `MPI_Send` dagegen sind verschiedene Interpretationen denkbar: Sicherlich sollte das Programm frühstens dann aus ihr zurückkehren, wenn wir den Speicherbereich, der die zu sendenden Daten enthält, verändern dürfen. Beispielsweise sollten wir bei der Behandlung der Wellengleichung nach dem Abschluss der Funktion `MPI_Send` die Werte des nächsten Zeitschritts in unsere Arrays schreiben dürfen.

Diese Bedingung kann allerdings in unterschiedlicher Weise sichergestellt werden: Die MPI-Bibliothek könnte die zu übertragenden Daten in einen Hilfsspeicher kopieren und später verschicken, sie könnte warten, bis in einem passenden Aufruf der Funktion `MPI_Recv` die Daten empfangen wurden, oder sie könnte fordern, dass bereits ein Aufruf dieser Funktion auf die Daten wartet, so dass sie sofort übertragen werden können.

Bei dem Aufruf `MPI_Send` entscheidet die Implementierung, welche der ersten beiden Vorgehensweisen gewählt wird. Wir können die anderen Methoden durch Varianten der Sendeoperation verwenden: `MPI_Bsend` verwendet einen Hilfsspeicher (engl. *buffer*), den wir mit Hilfe der Funktionen `MPI_Buffer_attach` und `MPI_Buffer_detach` explizit anlegen und anschließend wieder freigeben müssen. Nach der Rückkehr aus der Funktion sind die zu übertragenden Daten in den Hilfsspeicher kopiert und werden bei Gelegenheit im Hintergrund an den Empfänger gesendet, ohne dass wir uns weiter um sie kümmern müssen.

`MPI_Ssend` ist eine *synchrone* Sendeoperation: Das Programm kehrt erst aus der Funktion zurück, wenn auf Seiten des Empfängers die Funktion `MPI_Recv` (oder eine andere Empfangsfunktion) mit passenden Parametern aufgerufen wurde. Nach der Rückkehr aus dieser Funktion wissen wir also nicht nur, dass wir die versendeten Werte überschreiben dürfen, wir wissen auch, dass sie empfangen wurden.

`MPI_Rsend` geht einen Schritt weiter und erwartet, dass auf Seiten des Empfängers bereits die Funktion `MPI_Recv` (oder eine andere Empfangsfunktion) aufgerufen wurde, dass er Empfänger also dazu bereit (engl. *ready*) ist, Daten zu empfangen. Diese Variante der Sendeoperation kann auf manchen Systemen effizienter sein.

7.4 Kollektive Kommunikation

Bei den bisher diskutierten Operationen waren jeweils höchstens zwei Prozesse an einem Datenaustausch beteiligt. Die meisten Algorithmen lassen sich in dieser Weise umsetzen, allerdings gibt es eine ganze Reihe von Situationen, in denen es sinnvoll ist, Daten innerhalb aller zu einem Communicator gehörenden Prozesse auszutauschen. Die entsprechenden Operationen werden im MPI-Standard als *kollektive Kommunikationsoperationen* bezeichnet.

Barrieren. Die einfachste Form der kollektiven Kommunikation ist die uns bereits von OpenMP bekannte *Barriere*. Ein Prozess kehrt erst dann aus einem Aufruf der Funktion `MPI_Barrier` zurück, wenn auch alle anderen Prozesse aus der zu dem als Parameter angegebenen Communicator gehörenden Gruppe die Funktion aufgerufen haben. Es muss sich dabei nicht um *denselben* Aufruf handeln, beispielsweise wird das folgende Programmfragment nicht unbegrenzt lange warten:

```
switch(rank) {
case 0:
  MPI_Barrier(MPI_COMM_WORLD);
  printf("Barrier 0 passed.\n");
  break;
```

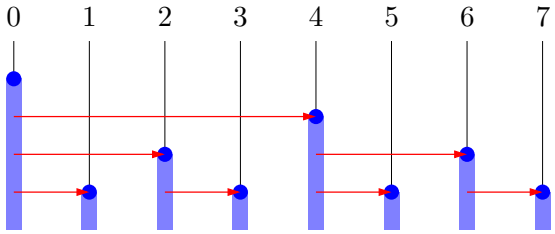

Abbildung 7.1: Umsetzung einer Broadcast-Operation für acht Prozesse $0, \ldots, 7$ mittels einer binären Strategie.

```
default:
  MPI_Barrier(MPI_COMM_WORLD);
  printf("Barrier 1 passed.\n");
}
```

Bei dieser Operationen werden zwar keine Daten im engeren Sinn übertragen, allerdings müssen die Prozesse natürlich trotzdem kommunizieren, um sich gegenseitig mitzuteilen, dass sie die Barriere erreicht haben.

Broadcast. Die einfachste kollektive Kommunikationsoperation, bei der tatsächlich Daten übertragen werden, ist das Versenden eines Datensatzes von einem Prozess an *alle* anderen. Diese Operation wird im MPI-Standard als *Broadcast* bezeichnet und mit Hilfe der Funktion `MPI_Bcast` in Auftrag gegeben. Den sendenden Prozess bezeichnet der Standard als die *Wurzel* (engl. *root*) der Operation.

An der Broadcast-Operation zeigt sich, dass kollektive Kommunikation erhebliche Vorteile gegenüber der Punkt-zu-Punkt-Kommunikation bieten kann: Falls unser MPI-Programm auf einem System mit $n = 2^k$ Prozessen läuft, könnte `MPI_Bcast` als eine einfache Schleife umgesetzt sein, die den Datensatz per `MPI_Send` und `MPI_Recv` an alle Prozesse verschickt. Dann würde der Absender n Kommunikationsoperationen ausführen.

Eine gute MPI-Implementierung kann diese Aufgabe allerdings auch wesentlich geschickter lösen: Falls Prozess 0 der Absender ist, können die Daten in einem ersten Schritt an Prozess 2^{k-1} übertragen werden. Im zweiten Schritt überträgt Prozess 0 die Daten an Prozess 2^{k-2}, aber da Prozess 2^{k-1} die Daten in diesem Schritt auch besitzt, kann er sie *gleichzeitig* an Prozess $2^{k-1}+2^{k-2}$ schicken, so dass nach dem zweiten Schritt die Prozesse 0, 2^{k-2}, $2 \times 2^{k-2} = 2^{k-1}$, $3 \times 2^{k-2} = 2^{k-1} + 2^{k-2}$ im Besitz der Daten sind. Wenn wir so weiter fortfahren, verdoppelt sich in jedem Schritt die Anzahl der Prozesse, die die Daten besitzen und weitergeben können, so dass nach $k = \log_2 n$ Schritten alle Prozesse informiert sind.

Reduction. Den Broadcast können wir besonders effizient umsetzen, indem wir dafür sorgen, dass jeder Prozess, der die zu übermittelnden Daten bereits kennt, eine Kopie

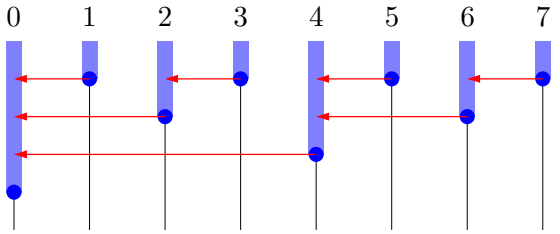

Abbildung 7.2: Umsetzung einer Reduction-Operation für acht Prozesse $0, \ldots, 7$ mittels einer binären Strategie.

an jeweils einen weiteren Prozess übermittelt, der sie noch nicht erhalten hat.

Wenn wir in ähnlicher Weise auch Daten von mehreren Prozessen an einen einzelnen Prozess übertragen wollen, müssten wir diese Kopiervorgänge rückgängig machen, also Daten von mehreren Prozessen zusammenfassen. Genau das geschieht bei der im MPI-Standard als *Reduction* bezeichneten Operation: Aus den Datensätzen mehrerer Prozesse wird ein einzelner Datensatz konstruiert. Ein Beispiel ist die Berechnung einer Summe: Sei $n = 2^k$. Wir gehen davon aus, dass der i-te Prozess eine Zahl a_i berechnet hat und wir an der Summe

$$a_0 + a_1 + \ldots + a_{n-1}$$

interessiert sind. In einem ersten Schritt können wir a_1 an Prozess 0 übertragen und die Summe $a_0^{(1)} = a_0 + a_1$ berechnen. Gleichzeitig können wir auch für alle $\ell \in \{1, \ldots, n/2 - 1\}$ die Zahl $a_{2\ell+1}$ an Prozess 2ℓ senden und $a_{2\ell}^{(1)} = a_{2\ell} + a_{2\ell+1}$ bestimmen. Dank des Assoziativgesetzes gilt

$$a_0 + a_1 + a_2 + a_3 + \ldots + a_{n-2} + a_{n-1} = a_0^{(1)} + a_2^{(1)} + \ldots + a_{n-2}^{(1)},$$

so dass jetzt nur noch $n/2$ Zahlen zu addieren sind. In einem zweiten Schritt können wir $a_2^{(1)}$ an Prozess 0 übertragen und $a_0^{(2)} = a_0^{(1)} + a_2^{(1)}$ berechnen, und wir können für alle $\ell \in \{1, \ldots, n/4 - 1\}$ auch $a_{4\ell+2}^{(1)}$ an Prozess 4ℓ senden und $a_{4\ell}^{(2)} = a_{4\ell}^{(1)} + a_{4\ell+2}^{(1)}$ bestimmen, so dass

$$a_0^{(1)} + a_2^{(1)} + a_4^{(1)} + a_6^{(1)} + \ldots + a_{n-4}^{(1)} + a_{n-2}^{(1)} = a_0^{(2)} + a_4^{(2)} + \ldots + a_{n-4}^{(2)}$$

gilt und wir nur noch $n/4$ Zahlen zu addieren haben. Jeder Schritt halbiert die Anzahl der zu addierenden Zahlen, bis nach k Schritten nur noch eine Zahl übrig ist, nämlich das gesuchte Ergebnis.

Anders als die Broadcast-Operation, bei der lediglich Daten kopiert wurden, muss bei der Reduction-Operation in jedem Schritt auch die Datenmenge reduziert werden, beispielsweise Teilsummen in unserem Beispiel. Die betreffende Verknüpfung wird im MPI-Standard durch ein Objekt des Typs `MPI_Operation` beschrieben. Wir können vordefinierte Verküpfungen wie `MPI_MAX`, `MPI_MIN`, `MPI_SUM` oder `MPI_PROD` verwenden, die das Maximum, das Minimum, die Summe und das Produkt berechnen, oder mit Hilfe

der Funktion `MPI_Op_create` eigene Verknüpfungen definieren. Wichtig ist dabei, dass die Verknüpfungen *assoziativ* sind, so dass die Prozesse Teilergebnisse unabhängig voneinander berechnen können. Das folgende Beispiel verwendet `MPI_MAX`, um die Laufzeit einer Berechnung zu ermitteln:

```
t_start = MPI_Wtime();
/* ... spend some time ... */
t_run = MPI_Wtime() - t_start;
MPI_Reduce(&t_run, &t_max, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
printf("Maximal runtime: %.2f seconds\n", t_max);
```

Die Funktion `MPI_Reduce` erwartet einen Zeiger auf die zu sendenden Daten, einen Zeiger auf die zu empfangenden, eine Zahl, die die Anzahl der Datenobjekte angibt, den Typ der Daten, die Verknüpfung, die Nummer des Prozesses, der das Ergebnis erhalten soll, und den Communicator, in dem die Daten ausgetauscht werden sollen.

Scatter. Bei einer Broadcast-Operation werden dieselben Daten an alle Prozesse versendet. Der MPI-Standard sieht daneben auch die sogenannte *Scatter-Operation* vor, bei der *unterschiedliche* Daten an alle Prozesse übermittelt werden. In dem folgenden Beispiel bittet der Prozess mit der Nummer 0 den Benutzer um die Eingabe einer Zahl n, die beispielsweise die Länge eines zu verarbeitenden Arrays angibt. Dann berechnet er in der bereits bekannten Weise, welche Teile des Arrays von welchem Prozess zu bearbeiten sind, und übermittelt das Ergebnis per `MPI_Scatter` an die anderen Prozesse.

```
int limits[2];
if(rank == 0) {
  printf("Array dimension?\n");
  scanf("%d", &n);
  sendbuf = (int *) malloc((size_t) sizeof(int) * 2 * size);
  for(i=0; i<size; i++) {
    sendbuf[2*i] = i*n/size;
    sendbuf[2*i+1] = (i+1)*n/size;
  }
  MPI_Scatter(sendbuf, 2, MPI_INT, limits, 2, MPI_INT,
              0, MPI_COMM_WORLD);
  free(sendbuf);
}
else
  MPI_Scatter(0, 0, MPI_INT, limits, 2, MPI_INT,
              0, MPI_COMM_WORLD);
```

Prozess 0 verarbeitet die Anfrage und trägt die an die einzelnen Prozesse zu übermittelnden Daten in das Array `sendbuf` ein, in dem für jeden Prozess zwei `int`-Variablen vorgesehen sind. Der Aufruf von `MPI_Scatter` erwartet als erste drei Argumente wieder eine Beschreibung der zu sendenden Daten, also einen Zeiger auf die Daten, die

Anzahl der *pro Prozess* zu übermittelnden Einträge und den Typ, als zweite drei eine Beschreibung der zu empfangenden Daten, die Nummer des sendenden Prozesses und den zu verwendenden Communicator. In allen außer dem sendenden Prozess werden die ersten drei Parameter ignoriert, so dass wir in ihnen auch keinen Speicher für `sendbuf` anzulegen brauchen.

Gather. Selbstverständlich gibt es auch für die Scatter-Operation ein Gegenstück, bei der ein Prozess Daten von allen anderen Prozessen empfängt: Die *Gather-Operation* schreibt die von allen Prozessen empfangenen Daten in einen Empfangsspeicher. Falls wir beispielsweise die Laufzeiten aller Prozesse im Prozess mit der Nummer 0 sammeln und ausgeben wollen, können wir das wie folgt tun:

```
t_start = MPI_Wtime();
/* ... spend some time ... */
t_run = MPI_Wtime() - t_start;
if(rank == 0) {
  t_proc = (double *) malloc((size_t) sizeof(double) * size);
  MPI_Gather(&t_run, 1, MPI_DOUBLE, t_proc, 1, MPI_DOUBLE,
             0, MPI_COMM_WORLD);
  for(i=0; i<size; i++)
    printf("Runtime in process %d: %.2f seconds\n",
           i, t_proc[i]);
}
else
  MPI_Gather(&t_run, 1, MPI_DOUBLE, 0, 0, MPI_DOUBLE,
             0, MPI_COMM_WORLD);
```

Die ersten drei Parameter beschreiben auch hier wieder die zu sendenden Daten, die zweiten drei Parameter die zu empfangenden, der siebte Parameter ist die Nummer des Wurzelprozesses und der achte der Communicator, in dessen Kontext der Datenaustausch stattfindet.

Varianten. Neben den hier beschriebenen grundlegenden Funktionen gibt es die Funktionen `MPI_Scatterv` und `MPI_Gatherv`, die es ermöglichen, unterschiedlich große Datenpakete an die einzelnen Prozesse zu senden oder von ihnen zu empfangen.

Die beiden Funktionen `MPI_Allgather` und `MPI_Allgatherv` arbeiten ähnlich wie `MPI_Gather` und `MPI_Gatherv`, stellen die eingesammelten Daten allerdings allen beteiligten Prozessen zur Verfügung.

Die Funktionen `MPI_Alltoall` und `MPI_Alltoallv` schließlich senden *unterschiedliche* Daten von allen Prozessen an alle anderen Prozesse.

Die Funktion `MPI_Allreduce` arbeitet ähnlich wie `MPI_Reduce`, macht das Ergebnis der Reduktionsoperation allerdings allen beteiligten Prozessen verfügbar, statt es nur dem Wurzelprozess zu überlassen. Weitere Varianten der Reduktion sind die Funktion `MPI_Reduce_scatter`, bei der Teile des Ergebnisses an andere Prozesse verteilt werden,

und die Funktion `MPI_Scan`, bei der Prozess p das Ergebnis der Reduktion der ersten p Datensätze erhält.

8 Heterogene Rechnersysteme und Grafikkarten

Die meisten modernen Computer werden mit Hilfe eines Bildschirms bedient. Für die Ansteuerung dieses Bildschirms ist in der Regel nicht der Prozessor zuständig, sondern ein separater Grafikchip. Frühe Grafikschaltkreise waren lediglich dazu in der Lage, Daten aus dem Speicher zu lesen und in geeignete Signale für den Bildschirm umzusetzen. Es stellte sich allerdings relativ schnell heraus, dass sich bestimmte Aufgaben durch einen spezialisierten Schaltkreis auf dem Grafikchip wesentlich schneller als durch den allgemein einsetzbaren Hauptprozessor durchführen lassen, beispielsweise das Löschen oder Kopieren von Speicherbereichen.

Das hatte zur Folge, dass Grafikchips zunehmend komplexer wurden. Zunächst wurden Funktionen für zweidimensionale Grafik aufgenommen, später folgte die dreidimensionale Grafik, die wesentlich höhere Anforderungen stellt: Es müssen große Mengen vierdimensionaler Vektoren transformiert werden, hinzu kommen mehr oder weniger aufwendige Rechenoperationen, um eine annähernd realistische Beleuchtung der dreidimensionalen Szene zu erreichen. Um die Daten hinreichend schnell dem Grafikchip zur Verfügung stellen zu können, wurde spezialisierter Grafikspeicher hinzugefügt, in der Regel in Form einer Erweiterungskarte, die den Grafikchip und den Grafikspeicher enthält und über ein geeignetes Protokoll mit dem Rest des Computers kommuniziert.

Frühe Grafikkarten sahen spezialisierte Schaltkreise vor, die die Ausführung von in Grafikstandards wie OpenGL definierten Operationen beschleunigten. Durch den großen Erfolg der dreidimensionalen Grafik in Computerspielen entstand ein Massenmarkt, in dem die Realitätsnähe der Grafiken ein wichtiger Faktor für den kommerziellen Erfolg eines Spiels ist. Da realistische Grafiken in der Regel nur mit relativ komplexen Algorithmen zu erreichen sind, wurden die Grafikchips zunehmend flexibler und schließlich frei programmierbar. Beispielsweise wurde in OpenGL inzwischen die ursprüngliche *fixed function pipeline*, die eine Abfolge von Rechenoperationen definierte, die auf jeden Vektor und jeden Bildpunkt angewendet werden, durch die *OpenGL shading language* ersetzt, eine C ähnliche Programmiersprache, mit deren Hilfe sich beliebige Folgen von Rechenoperationen definieren lassen.

Schließlich erkannten die Hersteller der Grafikhardware, dass sie sich mit relativ geringem Aufwand weitere Märkte erschließen können, indem sie eine Möglichkeit schufen, Daten nicht nur an die Grafikkarte zu senden, sondern auch von ihr zu empfangen. Mit der Einführung des CUDA-Standards ermöglichte es die Firma NVIDIA®, die Leistungsfähigkeit moderner Grafikkarten relativ einfach auszunutzen.

Etwas später folgte mit OpenCL ein offener Standard. Während CUDA ausschließlich für Grafikkarten des Herstellers NVIDIA® vorgesehen ist, zielt OpenCL auf allgemeine

heterogene Rechner: Ein Rechner kann aus unterschiedlichen Prozessoren und unterschiedlichen Speichern bestehen, die eine Aufgabe kooperativ lösen. Dabei gibt es einen Hauptprozessor mit einem Hauptspeicher, der die anderen Prozessoren und Speicher steuert. Er veranlasst Kopiervorgänge zwischen den einzelnen Speichern und definiert über eine spezielle Programmiersprache die Rechenoperationen, die die einzelnen Prozessoren übernehmen sollen.

Heute spielen für allgemeine Berechnungen angepasste Grafikkarten eine wichtige Rolle im Bereich des Hochleistungsrechnens: Moderne Grafikkarten verfügen über mehrere hundert Rechenwerke und sind konventionellen Prozessoren in Hinblick auf die reine Rechenleistung deutlich überlegen. Allerdings unterliegen sie gewissen Einschränkungen: Ihre Architektur orientiert sich an Vektorrechnern, so dass Algorithmen mit Fallunterscheidungen zu Schwierigkeiten führen können. Die in der Regel hohe Bandbreite des Grafikspeichers wird dadurch erkauft, dass deutlich weniger Grafikspeicher als Hauptspeicher zur Verfügung steht, so dass bei der Verarbeitung großer Datenmengen der Programmierer dafür sorgen muss, dass die jeweils nötigen Daten rechtzeitig aus dem Hauptspeicher in den Grafikspeicher übertragen werden. Hinzu kommen relative enge Grenzen für die Länge der ausführbaren Programme, die Anzahl ihrer lokalen Variablen und die Schachtelungstiefe für Funktionsaufrufe.

8.1 CUDA-Architektur

Für Grafikprozessoren hat sich die Abkürzung *GPU* (engl. *graphics processing unit*) etabliert, die die Nähe zu den als CPU (engl. *central processing unit*) bezeichneten Hauptprozessoren betont. Einen Grafikprozessor, der auch für allgemeinere als nur grafische Aufgaben geeignet ist, bezeichnet man als *GPGPU* (engl. *general purpose GPU*).

Wir behandeln hier zunächst die von der Firma NVIDIA® eingeführte CUDA-Architektur (engl. *compute unified device architecture*). Ihre hohe Rechenleistung erreichen CUDA-Grafikkarten durch eine Kombination aus Multithreading und Vektorisierung: Die GPGPU enthält eine hohe Anzahl sehr einfacher Prozessorkerne (häufig mehrere hundert), die parallel arbeiten können. Allerdings arbeiten die Kerne nicht unabhängig, sondern sind zu größeren Gruppen, den *Streaming Multiprocessors* zusammengefasst, die ähnlich wie bei einem Vektorrechner jeweils dieselben Befehle für potentiell unterschiedliche Daten ausführen.

SIMT. Die Multiprozessoren verwenden eine Mischung aus Vektorrechnen und Multithreading, die als *SIMT* (engl. *single instruction, multiple threads*) bezeichnet wird: Ähnlich wie ein Vektorrechner führt ein Multiprozessor in jedem Takt nur einen einzelnen Befehl aus, der auf mehrere Variablen gleichzeitig wirkt. Allerdings verfügt jeder Prozessorkern über einen eigenen Programmzähler, so dass sich Fallunterscheidungen anders als bei traditionellen Vektorrechnern behandeln lassen: Ein Befehl wird von allen Kernen ausgeführt, deren Programmzähler gerade bei diesem Befehl steht, während alle anderen Kerne nichts tun. Durch diesen Ansatz lassen sich die Multiprozessoren wie „echte“ Mehrkernprozessoren programmieren, können aber bei geeigneter Program-

mierung die Geschwindigkeit von Vektorrechnern erreichen. Im schlimmsten Fall kann es allerdings auch vorkommen, dass alle Threads jeweils unterschiedliche Befehle ausführen, so dass in jedem Taktzyklus nur jeweils ein Thread aktiv ist und die Geschwindigkeit des Programms erheblich leidet.

Scheduling. CUDA geht davon aus, dass sehr viele Threads ausgeführt werden, beispielsweise einer für jeden Bildpunkt einer zu berechnenden Grafik. Um den Verwaltungsaufwand überschaubar zu halten, werden Threads zu *Warps* von jeweils höchstens 32 Threads zusammengefasst. Mehrere Warps wiederum werden zu einem *Threadblock* zusammengefasst, und ein vollständiger Threadblock wird jeweils von einem Multiprozessor ausgeführt. Dazu wählt der Multiprozessor in jedem Zyklus einen Warp des aktuellen Threadblocks aus und bearbeitet einen Befehl dieses Warps. Wie bereits erwähnt wird dabei im günstigsten Fall ein Befehl in jedem der 32 Threads ausgeführt, im ungünstigsten nur ein Befehl in einem einzigen der Threads. Threads in unterschiedlichen Warps werden nicht parallel ausgeführt und können sich deshalb auch nicht gegenseitig behindern. Allerdings teilen sich alle Threads eines Threadblocks dieselben Ressourcen (beispielsweise Register) des Multiprozessors.

Ein Multiprozessor verwaltet in der Regel mehrere Warps gleichzeitig, von denen in jedem Zyklus einer einen Befehl ausführen kann. Durch diese Vorgehensweise kann der Multiprozessor beispielsweise Latenzen bei Speicherzugriffen verstecken: Während ein Warp auf Daten wartet, kann ein anderer Warp rechnen.

Anders als bei „echtem“ Multithreading gibt es allerdings eine Reihe von Einschränkungen: Alle Warps auf einem Multiprozessor teilen sich die Register dieses Multiprozessors. Dadurch kann einerseits zwischen Warps gewechselt werden, ohne aufwendig Register zu speichern und wiederherzustellen, andererseits kann es bei aufwendigen Programmen, die viele Register benutzen, dazu kommen, dass nur wenige Warps erzeugt werden können: Falls 256 Threads vorliegen, die zu 8 Warps zusammengefasst werden können, und jeder Thread 16 Register benötigt, sind 4096 Register erforderlich. Falls der Multiprozessor nicht genügend viele Register zur Verfügung stellt, können nicht alle 8 Warps gleichzeitig behandelt werden.

Speicher. Neben den Registern verfügt ein Multiprozessor über weitere lokale Speicherbereiche: Der *geteilte Speicher* kann von allen Threads eines Threadblocks angesprochen werden. Dieser Speicher ist allerdings in der Regel sehr klein, um den Schaltungsaufwand gering zu halten. Wie immer bei Systemen mit geteiltem Speicher muss bei Speicherzugriffen sichergestellt werden, dass keine Konflikte auftreten, dass also beispielsweise nicht mehrere Threads gleichzeitig schreiben oder auf Daten zuzugreifen versuchen, die noch nicht geschrieben wurden.

Außerdem gibt es den *Konstantenspeicher*, der durch den Hauptprozessor mit Daten gefüllt werden kann, die die auf der Grafikkarte ablaufenden Programme allerdings nicht ändern können. Da der Konstantenspeicher während der Laufzeit eines Programms unverändert bleibt, lassen sich Lesezugriffe besonders gut optimieren.

Sowohl geteilter als auch Konstantenspeicher sind relativ klein, in der Regel im zwei-

stelligen Kilobyte-Bereich. Für größere Datenmengen teilen sich alle Multiprozessoren, und damit auch alle Threads, den eigentlichen *Grafikspeicher*, der auf modernen Grafikkarten mehrere Gigabytes umfassen kann. Auch auf diesen Speicher können verschiedene Threads auf verschiedenen Multiprozessoren gemeinsam zugreifen, so dass auch hier auf eine korrekte Synchronisation geachtet werden muss.

8.2 CUDA-Programmierung

Ein CUDA-System wird in der Regel in der Sprache *CUDA C* programmiert, einer Variante des Standards C++, die um einige für die Ansteuerung der Grafikkarte nötige Funktionen erweitert wurde. Der wichtigste Unterscheid zu C++ liegt darin, dass manche Passagen einer Quelltextdatei für den Hauptprozessor vorgesehen sind, andere dagegen für den Grafikprozessor. Für die Übersetzung der in CUDA C geschriebenen Programme ist der Befehl `nvcc` zuständig, der die für die Grafikkarte bestimmten Teile von den für den Hauptprozessor bestimmten trennt, beide mit passenden Compilern übersetzt, und die resultierenden Binärdateien wieder zusammenfügt.

Kerne. Funktionen können für den Hauptprozessor oder für die Grafikkarte übersetzt werden. Dabei müssen für die Grafikkarte vorgesehene Funktionen mit den Schlüsselwörtern `__device__` oder `__global__` markiert werden, damit `nvcc` sie dem für sie zuständigen Compiler übergeben kann. Dabei bezeichnet `__device__` Funktionen, die auf der Grafikkarte ausgeführt werden sollen, aber nur von anderen auf der Grafikkarte laufenden Funktionen aufgerufen werden können. Dagegen ist `__global__` für Funktionen vorgesehen, die auf der Grafikkarte laufen, aber von einem auf dem Hauptprozessor laufenden Programm aus gestartet werden können. Diese Funktionen werden häufig als *Kerne* (engl. *kernel*, vermutlich zurückgehend auf die in der Mathematik auftretenden *Faltungskerne*) bezeichnet.

Aufruf eines Kerns. In der Regel führt die CUDA-GPGPU eine Kernfunktion nicht nur einmal aus, sondern in mehreren Threads parallel. Dabei ähnelt die Vorgehensweise der auch bei OpenMP oder MPI verwendeten: Mit einem speziellen Befehl wird die Kernfunktion mehrfach auf der Grafikkarte gestartet und kann dann mit Hilfe geeigneter Mechanismen herausfinden, welchen Teil der anliegenden Arbeit sie bearbeiten soll. Als Beispiel untersuchen wir wieder die einfache Linearkombination zweier Vektoren:

```
__global__ void
kernel_axpy(int n, float alpha, const float *x, float *y)
{
  int i = threadIdx.x + blockIdx.x * blockDim.x;

  y[i] += alpha * x[i];
}
```

```
void
axpy(int n, float alpha, const float *x, float *y)
{
  kernel_axpy<<<n/64,64>>>(n, alpha, x, y);
}
```

Interessant an diesem Beispiel sind drei Dinge: Erstens der Einsatz des Schlüsselworts `__global__`, mit dem die Funktion `kernel_axpy` als für die Grafikkarte vorgesehen markiert wird. Zweitens die Zeile `kernel_axpy<<<n/64,64>>>(...)`, mit der diese Kernfunktion auf der Grafikkarte gestartet wird. In den dreifachen spitzen Klammern wird festgelegt, wieviele Threads diese Funktion ausführen sollen: Die erste Zahl gibt die Anzahl der Threadblöcke an, die zweite die Anzahl der Threads pro Block. Unser Beispielprogramm funktioniert nur, wenn `n` ein Vielfaches von 64 ist, ansonsten müssten wir, wie schon bei Vektorrechnern, eine Sonderbehandlung für überzählige Komponenten implementieren. Drittens werden `threadIdx.x`, `blockIdx.x` sowie `blockDim.x` verwendet. `threadIdx.x` gibt dabei die Nummer des laufenden Threads innerhalb seines Threadblocks an, `blockIdx.x` die Nummer des Threadblocks und `blockDim.x` die Anzahl der Threads in dem Threadblock. Die Anzahl aller Threadblöcke können wir mit `gridDim.x` erfahren. CUDA unterstützt bis zu dreidimensionale Indizes, in diesen Fällen sind die y- und die z-Koordinaten entsprechend mit `threadIdx.y`, `threadIdx.z`, `blockIdx.y` und so weiter zu erreichen.

Einteilung in Threadblocks. Wie die auszuführenden Threads in Threadblocks eingeteilt werden hat entscheidenden Einfluss auf die Geschwindigkeit, mit der ein Programm auf der Grafikkarte ausgeführt wird. Falls ein Threadblock zuwenige Threads enthält, werden nicht alle Kerne des ihn bearbeitenden Multiprozessors ausgelastet. Falls ein Threadblock zuviele Threads enthält, gibt es unter Umständen nicht genügend viele Threadblöcke, um alle auf dem Grafikchip vorhandenen Multiprozessoren auszulasten, oder es könnte der Fall eintreten, dass dem Multiprozessor nicht genügend viele Register zur Verfügung stehen, um effizient alle Threads verarbeiten zu können.

Speicherverwaltung. Das Beispielprogramm ist in der gegebenen Form nicht lauffähig: Die Funktion `axpy` läuft auf dem Hauptprozessor und erhält Zeiger `x` und `y` auf den Hauptspeicher, während der Kern `kernel_axpy` auf der GPU läuft und deshalb mit Zeigern auf den Grafikspeicher arbeiten sollte. Um das Programm praktisch lauffähig zu machen, müssen wir dafür sorgen, dass die Arrays vor der Ausführung des Kerns in den Grafikspeicher übertragen werden und danach das Ergebnis in den Hauptspeicher kopiert wird.

Bereiche im Grafikspeicher können wir mit der Funktion `cudaMalloc` anfordern und mit `cudaFree` freigeben. Die Funktion `cudaMemcpy` dient dazu, Daten zwischen Grafik- und Hauptspeicher hin und her zu kopieren, wobei ein zusätzlicher Parameter festlegt, wie die übergebenen Zeiger zu interpretieren sind: Bei `cudaMemcpyHostToDevice` verweist der erste Zeiger, also das Ziel, auf Grafikspeicher und der zweite, also

die Quelle, auf Hauptspeicher. Bei `cudaMemcpyDeviceToHost` sind die Rollen vertauscht, es werden Daten aus dem Grafikspeicher in den Hauptspeicher kopiert. Bei `cudaMemcpyDeviceToDevice` schließlich wird innerhalb des Grafikspeichers kopiert. Der zusätzliche Parameter ist erforderlich, weil CUDA C für Zeiger auf den Haupt- oder Grafikspeicher keine unterschiedlichen Typen verwendet, so dass es in der Verantwortung der Programmierers liegt, die Zeiger nicht zu verwechseln. Auf der Seite des Hauptspeichers spielt *pinned memory* eine besondere Rolle, also Teile des Hauptspeichers, die nicht durch das Betriebssystem verschoben werden dürfen und es deshalb ermöglichen, Kopieroperationen zwischen Haupt- und Grafikspeicher per *direct memory access* ohne zusätzliche Hilfsspeicher durchzuführen. Derartige Speicherbereiche können mit `cudaMallocHost` angelegt und mit `cudaFreeHost` wieder freigegeben werden.

Mit Hilfe von `cudaMalloc`, `cudaMemcpy` und `cudaFree` können wir jetzt daran gehen, eine tatsächlich funktionsfähige Variante unseres Beispielprogramms zu formulieren. Um auch nicht durch 64 teilbare Arraydimensionen korrekt handhaben zu können, runden wir die Threadzahl auf und sorgen mit einer `if`-Anweisung in der Kernfunktion dafür, dass überzählige Einträge ignoriert werden. Insgesamt erhalten wir das folgende Programmfragment:

```
__global__ void
kernel_axpy(int n, float alpha, const float *d_x, float *d_y)
{
  int i = threadIdx.x + blockIdx.x * blockDim.x;

  if(i < n)
    d_y[i] += alpha * d_x[i];
}

void
axpy(int n, float alpha, const float *x, float *y)
{
  float *d_x;
  float *d_y;

  cudaMalloc((void **) &d_x, sizeof(float) * n);
  cudaMalloc((void **) &d_y, sizeof(float) * n);
  cudaMemcpy(d_x, x, sizeof(float) * n, cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, sizeof(float) * n, cudaMemcpyHostToDevice);
  kernel_axpy<<<(n+63)/64,64>>>(n, alpha, d_x, d_y);
  cudaMemcpy(y, d_y, sizeof(float) * n, cudaMemcpyDeviceToHost);
  cudaFree(d_y);
  cudaFree(d_x);
}
```

Fehlerbehandlung. An dem Beispiel fällt auf, dass bei dem Aufruf von `cudaMalloc` nicht wie bei dem bekannten `malloc` einfach ein Zeiger auf den Speicherbereich zurückgegeben wird, sondern stattdessen ein Zeiger auf den Zeiger übergeben werden muss, der mit dem Ergebnis der Operation überschrieben wird. Diese Besonderheit liegt darin begründet, dass fast alle CUDA-Funktionen einen Fehlercode zurückgeben, mit dem sich überprüfen lässt, ob sie erfolgreich ausgeführt wurden. Dementsprechend gibt auch `cudaMalloc` eine Fehlernummer des Typs `cudaError_t` zurück, die bei erfolgreichem Abschluss der Funktion den Wert `cudaSuccess` aufweist. Bei allen anderen Werten ist ein Fehler aufgetreten, den wir beispielsweise mit der Funktion `cudaGetErrorString` in eine Fehlermeldung konvertieren können. Eine Ausnahme bildet der Aufruf der Kernfunktion `kernel_axpy`, bei dem wir einen Fehlercode mit der Funktion `cudaGetLastError` erfragen können.

Die korrekte Behandlung der Fehlercodes ist bei CUDA-Programmen besonders wichtig, da die Ressourcen der Grafikkarte in der Regel beschränkt sind, so dass auch auf den ersten Blick harmlose Funktionsaufrufe fehlschlagen können. Beispielsweise bietet eine Grafikkarte in der Regel keine virtuelle Speicherverwaltung, so dass nicht die Möglichkeit besteht, bei Bedarf zusätzlichen Speicher zu simulieren. Entsprechend sind auch der Anzahl der gleichzeitig laufenden Threads und Threadblöcke Grenzen gesetzt, an die man bei größeren Aufgabenstellungen durchaus stößt.

Asynchrone Ausführung. Das Hauptprogramm kann (und wird häufig) aus dem Aufruf der Kernfunktion zurückkehren, bevor die Grafikkarte ihre Arbeit abgeschlossen hat. Dadurch kann der Hauptprozessor weitere Arbeit leisten, während die Grafikkarte beschäftigt ist. Der Hauptprozessor kann mit der Funktion `cudaDeviceSynchronize` sicherstellen, dass die Grafikkarte alle Aufgaben abgeschlossen hat, bevor er versucht, die Ergebnisse zu verwenden.

Die Synchronisation verschiedener Operationen auf der Grafikkarte lässt sich über *Streams* bewerkstelligen: Ein Stream beschreibt eine Reihe von Operationen, die in der Reihenfolge ausgeführt werden müssen, in der sie angegeben werden. Wir können mit Hilfe geeigneter Funktionen Kern-Aufrufe und Kopieroperationen unterschiedlichen Streams zuordnen und so zulassen, dass sie unabhängig voneinander bearbeitet werden. Beispielsweise können wir (falls die Hardware es zulässt) in dieser Weise das Kopieren von Daten in den und aus dem Grafikspeicher ausführen lassen, während ein anderer Speicherbereich durch eine in einem anderen Stream laufende Kernfunktion bearbeitet wird.

Anders als bei Prozessen und Threads ist allerdings nicht garantiert, dass Streams nebenläufig ausgeführt werden. Ältere CUDA-Hardware kann beispielsweise nur einen Kern zur Zeit ausführen, frühe Generationen können keine Speichertransfers neben der Kernausführung bearbeiten.

Der Hauptprozessor kann Streams mit der Funktion `cudaStreamCreate` anlegen und mit `cudaStreamDestroy` löschen sowie mit `cudaStreamSynchronize` darauf warten, dass alle einem Stream zugeordneten Operationen ausgeführt wurden. In CUDA C wird ein Stream durch den Datentyp `cudaStream_t` beschrieben. Um die Ausführung einer Kern-

funktion einem bestimmten Stream zuzuordnen, geben wir ihn als viertes Argument in den Klammern `<<<...>>>` an (das dritte Argument setzen wir zunächst gleich null und diskutieren es später). Um die Ausführung einer Kopieroperation einem Stream hinzuzufügen, kann die Funktion `cudaMemcpyAsync` verwendet werden. Falls auf die Angabe des Streams verzichtet wird, verwendet das CUDA-System den Stream `0`.

Unser Beispielprogramm für die Linearkombination arbeitet korrekt, weil `cudaMemcpy` für die hier verwendete Kopieroperation aus dem Grafikspeicher in den Hauptspeicher implizit darauf wartet, dass die Ausführung der Kernfunktion abgeschlossen wurde.

Nebenläufiges Kopieren und Rechnen. Eine wichtige Anwendung von Streams ist die nebenläufige Ausführung von Kopier- und Rechenoperationen. Aktuelle Versionen CUDA-fähiger Grafikkarten können Daten zwischen Haupt- und Grafikspeicher kopieren, während ein Kern ausgeführt wird. Falls der Kern hinreichend lange rechnet, lassen sich so die Kopieroperationen größtenteils „verstecken“. Im folgenden Beispiel werden abwechselnd zwei Streams damit beauftragt, einen Teil des zu verarbeitenden Arrays zu senden, die nötigen Berechnungen durchzuführen und das Ergebnis zu lesen. Falls der Kern pro übertragenem Datenelement genügend lange rechnet, wirken sich nur das Senden des ersten Teils des Arrays und das Lesen des letzten Teils auf die Laufzeit aus, die restlichen Kopieroperationen überschneiden sich mit Rechenoperationen des jeweils anderen Streams.

```
off = 0; current = 0;
while(off < n) {
  chunksize = n - off;
  if(chunksize > blocksize * gridsize)
    chunksize = blocksize * gridsize;

  cudaMemcpyAsync(d_x[current], x+off,
                  (size_t) sizeof(float) * chunksize,
                  cudaMemcpyHostToDevice, stream[current]);
  kernel<<<gridsize,blocksize,0,stream[current]>>>
                  (chunksize, d_x[current]);
  cudaMemcpyAsync(x+off, d_x[current],
                  (size_t) sizeof(float) * chunksize,
                  cudaMemcpyDeviceToHost, stream[current]);
  off += chunksize; current = (current + 1) % 2;
}
cudaDeviceSynchronize();
```

In diesem Beispiel gibt `current` an, in welchen der zwei verwendeten Streams `stream[0]` und `stream[1]` der nächste Abschnitt des Arrays `x` verarbeitet werden soll. In dem Aufruf der Kernfunktion `kernel` gibt der vierte Parameter an, in welchem Stream diese Kernfunktion ausgeführt werden soll, schließlich müssen wir sicherstellen, dass der Kern erst zu rechnen beginnt, wenn die vorangehende Kopieraktion die nötigen Daten in

`d_x[current]` untergebracht hat. Entsprechend muss das Zurückkopieren der Daten warten, bis die Kernfunktion ihre Arbeit abgeschlossen hat.

Abbildung 8.1 illustriert den Ablauf für den Fall, dass das Array `x` in vier Teile zerlegt und auf zwei Streams verteilt wird. Die Zeit läuft dabei „von oben nach unten“, der linke Balken gibt an, welcher der beiden Streams (blau steht für Stream 0, rot für Stream 1) gerade die Recheneinheit beschäftigt, während der rechte Balken die Belegung der Kopiereinheit illustriert. In diesem Beispiel dauert das Rechnen in einem Stream so lange, dass der andere Stream parallel die Ergebnisse seiner letzten Rechenphase abholen und die Eingabedaten für seine nächste Rechenphase senden kann. Lediglich das erste Senden und das letzte Empfangen kann nicht parallel mit einer Rechenphase ablaufen.

Geteilter Speicher. Wie bereits erwähnt verfügt die Grafikkarte neben dem Grafikspeicher auch über geteilten Speicher, den alle Threads eines Threadblocks gemeinsam nutzen können. Da ein Threadblock von jeweils einem Multiprozessor ausgeführt wird, kann der geteilte Speicher direkt in den Multiprozessor integriert werden, um möglichst schnell auf ihn zugreifen zu können. Anders als bei den in Kapitel 5 behandelten Systemen ist der geteilte Speicher der Multiprozessoren aufgrund dieser Umsetzung relativ klein und eher dafür gedacht, schnell Daten zwischen den Threads eines Blocks auszutauschen oder als durch den Programmierer explizit verwalteter Cache zu dienen. Der geteilte Speicher ist nur für jeweils einen Multiprozessor zugänglich, nicht für andere Multiprozessoren und auch nicht für den Hauptprozessor.

Programmvariablen können wir mit dem Schlüsselwort `__shared__` dem geteilten Speicher zuordnen. Als Beispiel untersuchen wir die Berechnung der Summe

$$s := \sum_{i=0}^{n-1} a_i,$$

die wir mit m Threads in einem Threadblock durchführen wollen. Thread j berechnet dabei die Teilsumme

$$s_j := a_j + a_{j+m} + a_{j+2m} + \ldots,$$

so dass sich das Gesamtergebnis als

$$s = \sum_{j=0}^{m-1} s_j$$

ergibt. Um diese Summe auswerten zu können, müssen wir die von den einzelnen Threads berechneten Teilergebnisse zusammenfassen, also Daten zwischen verschiedenen Threads austauschen. Dazu verwenden wir ein Array `partsum`, das im geteilten Speicher untergebracht wird und die Werte s_j für alle Threads aufnimmt. Die Summe aller Einträge berechnen wir ähnlich der in Abbildung 7.4 dargestellten Reduktionsoperation, allerdings in einer leicht abgewandelten Form, in der zunächst die Zwischenergebnisse

$$s'_j := s_j + s_{j+m/2} \qquad \text{für alle } j < m/2$$

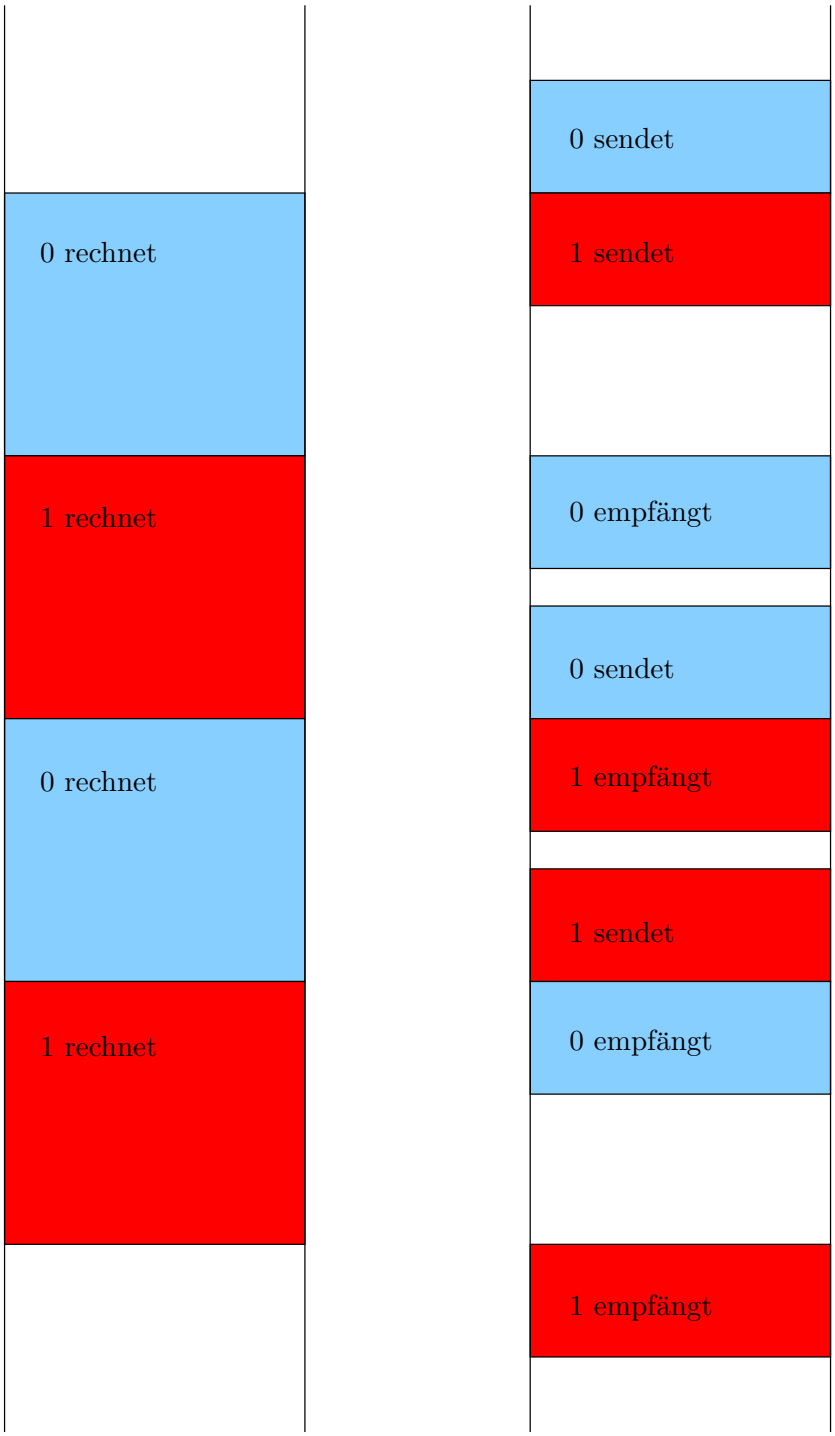

Abbildung 8.1: Paralleles Rechnen und Kopieren: Der linke Balken beschreibt die Auslastung der Recheneinheiten, der rechte die der Kopiereinheiten in Abhängigkeit von der Zeit.

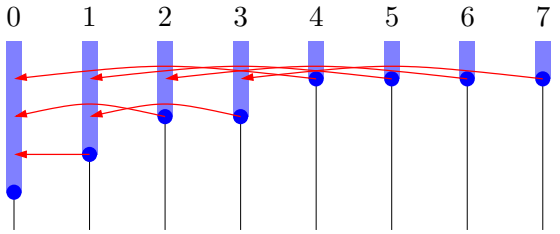

Abbildung 8.2: Reduktionsoperation für die Berechnung einer Summe in CUDA. Im ersten Schritt werden Summanden mit einem Abstand von $m/2$ addiert, im zweiten solche mit einem Abstand von $m/4$, bis nach $\log_2 m$ Schritten das Ergebnis vorliegt.

und dann entsprechend

$$\begin{aligned} s''_j &:= s'_j + s'_{j+m/4} && \text{für alle } j < m/4, \\ s'''_j &:= s''_j + s''_{j+m/8} && \text{für alle } j < m/8 \end{aligned}$$

berechnet werden, bis nach $\log_2 m$ Schritten das Endergebnis vorliegt. Diese modifizierte Reduktionsoperation ist in Abbildung 8.2 dargestellt. Wir erhalten das folgende Programmfragment:

```
j = threadIdx.x; i = j;

while(i < n) {
  sum += a[i];
  i += blockDim.x;
}
partsum[j] = sum;

k = blockDim.x / 2;
while(k > 0) {
  if(j < k)
    partsum[j] += partsum[j+k];
  k /= 2;
  __syncthreads();
}
```

In der ersten `while`-Schleife berechnet jeder Thread seine Teilsumme, die anschließend in `partsum` deponiert wird. Die zweite Schleife führt die Reduktion durch, bei der die Variable `k` angibt, wieviele Teilsummen in einem Durchlauf aktualisiert werden sollen. Der Befehl `__syncthreads()` synchronisiert alle Threads des Blocks, damit der nächste Schleifendurchlauf mit den korrekten Werten erfolgen kann.

Die Größe des geteilten Speichers kann statisch oder dynamisch festgelegt werden. Falls wir beispielsweise die Blockgröße in unserem Beispiel durch eine Konstante `BLKSIZE` definieren, können wir mit

```
__shared__ float partsum[BLKSIZE];
```

den nötigen Speicher statisch anfordern, da während des Übersetzens des Programms bereits bekannt ist, wieviel geteilter Speicher benötigt wird. Falls wir die Blockgröße erst zur Laufzeit definieren wollen, beispielsweise über eine Variable `blksize`, müssen wir den geteilten Speicher dynamisch festlegen. Dazu deklarieren wir das Array als

```
extern __shared__ float partsum[];
```

und geben als *drittes* (optionales) Argument bei dem Aufruf der Kernfunktion an, wieviel Speicher (in Bytes) für dieses Array angelegt werden soll:

```
kernel<<<1,blksize,sizeof(float)*blksize>>>(...);
```

Bei dieser Vorgehensweise ist eine Besonderheit zu beachten: Falls wir mehrere Arrays als `extern __shared__` deklariert haben, teilen sich alle *denselben* Speicherbereich. Beispielsweise wird bei

```
extern __shared__ int x[];
extern __shared__ int y[];
```

ein Schreibzugriff auf `x[2]` auch den Wert von `y[2]` ändern. Es empfiehlt sich also eher, jeweils nur ein dynamisches Array im geteilten Speicher anzulegen und es per Zeigerarithmetik (und gegebenenfalls Typecast) in mehrere Teilarrays zu zerlegen.

8.3 Grundlagen des OpenCL-Standards

Anders als CUDA ist OpenCL ein offener Standard, auf den sich mehrere Firmen geeinigt haben und der deshalb von einer größeren Anzahl von Rechnersystemen unterstützt wird.

Übersetzung. Während in CUDA C für den Haupt- und für den Grafikprozessor bestimmte Programmteile gemischt in einer Datei stehen können, werden sie in OpenCL getrennt: Das Programm für den Hauptprozessor ist ein gewöhnliches C-Programm, das auf in der Header-Datei `CL/cl.h` definierte Deklarationen zurückgreifen kann, um OpenCL-Funktionen zu verwenden. Das Programm für den Grafikprozessor ist in einer Variante der Programmiersprache C beschrieben und wird als String definiert, der mit einer OpenCL-Funktion zur Laufzeit übersetzt werden kann.

Dieser Zugang bietet den großen Vorteil, dass sich OpenCL-Programme mit praktisch jedem C-Compiler übersetzen lassen es leicht möglich ist, OpenCL-Funktionen auch in anderen Programmiersprachen zu verwenden. Allerdings muss das für den Grafikprozessor bestimmte Programm immer in OpenCL C formuliert sein.

Der große Nachteil dieser Vorgehensweise besteht darin, dass bestimmte Operationen, die in CUDA durch eine Erweiterung der Sprache C elegant formuliert werden

können, in OpenCL durch eine Reihe expliziter Funktionsaufrufe umgesetzt werden müssen. Beispielsweise erfordert der Aufruf einer Kernfunktion mehrere Aufrufe von OpenCL-Funktionen, die die an die Funktion zu übergebenden Parameter definieren, bevor schließlich mit einer weiteren OpenCL-Funktion schließlich die Kernfunktion gestartet werden kann. Um die Parameter zu speichern ist außerdem auch noch eine spezielle Datenstruktur erforderlich, die angelegt und später wieder freigegeben werden muss.

Plattformen. OpenCL ist so konzipiert, dass sich vielfältige heterogene Rechnersysteme beschreiben lassen. Es ist sogar vorgesehen, dass mehrere OpenCL-Implementierungen vorliegen, unter denen ein OpenCL-Programm die geeignete auswählen kann. Diesem Zweck dient das Konzept der *Plattform*. Eine Plattform bietet eine Schnittstelle zu einer Anzahl von Rechnersystemen, beispielsweise Grafikkarten oder Koprozessoren.

Auf einem Rechner können mehrere Plattformen installiert sein (beispielsweise eine von NVidia für Grafikkarten und eine von Intel für den Hauptprozessor), und ein OpenCL-Programm kann die vorhandenen Plattformen und ihre Ausstattung erfragen, um die für eine Aufgabe am besten geeignete zu ermitteln.

Jede Plattform wird durch einen Identifikator des Typs `cl_platform_id` beschrieben. Mit der Funktion `clGetPlatformIDs` können wir sowohl in Erfahrung bringen, wieviele Plattformen auf unserem System zur Verfügung stehen, als auch, welche Identifikatoren diese Plattformen besitzen. Die Funktion erwartet drei Parameter. Falls die ersten beiden Parameter gleich null sind, gibt die Funktion über den letzten Parameter die Anzahl der Plattformen zurück. Falls die ersten beiden Parameter ungleich null sind, gibt der erste die maximal Anzahl der Plattformen an, die wir erhalten wollen, und der zweite zeigt auf ein Array des Typs `cl_platform_id`, das groß genug ist, um alle Identifikatoren aufzunehmen.

Wir können also beispielsweise wie folgt vorgehen, um alle vorhandenen Plattformen zu identifizieren:

```
clGetPlatformIDs(0, 0, &nrplat);
platform = (cl_platform_id *) malloc(sizeof(cl_platform_id) * nrplat);
clGetPlatformIDs(nr_platforms, platform, 0);
```

Die Eigenschaften einer konkreten Plattform können wir mit Hilfe der Funktion `clGetPlatformInfo` erfragen, die ähnlich wie `clGetPlatformIDs` zu bedienen ist: Der erste Parameter ist der Identifikator der Plattform, der zweite gibt die abzufragende Eigenschaft an, der dritte die Größe des für das Ergebnis bereitgestellten Speichers, der vierte den Zeiger auf diesen Speicher, während über den fünften wieder abgefragt werden kann, wieviel Speicher erforderlich ist.

Die Namen der Plattformen können wir beispielsweise mit dem folgenden Programmfragment erfahren:

```
for(i=0; i<nrplat; i++) {
  clGetPlatformInfo(platform[i], CL_PLATFORM_NAME,
                    0, 0, &infosize);
  info = (char *) malloc(sizeof(char) * infosize);
```

```
    clGetPlatformInfo(platform[i], CL_PLATFORM_NAME,
                      infosize, info, 0);
    printf("Name: \"%s\"\n", info);
    free(info);
  }
```

Der erste Aufruf der Funktion `clPlatformGetInfo` dient nur dem Zweck, die Größe des nötigen Speichers in Erfahrung zu bringen, der zweite ermittelt dann die gewünschte Auskunft.

Neben `CL_PLATFORM_NAME` sind für den zweiten Parameter des Funktionsaufrufs auch `CL_PLATFORM_VERSION` und `CL_PLATFORM_PROFILE` von Interesse, um die maximal unterstützte OpenCL-Version und das unterstützte OpenCL-Profil (neben dem *full profile* gibt es auch das *embedded profile*, das einen reduzierten Funktionsumfang für einfachere Rechnersysteme vorsieht).

Devices. Jede OpenCL-Plattform läuft auf einem konventionellen Rechner, der als *Host* bezeichnet wird und für die Verwaltung des Hauptspeichers und die Kommunikation mit dem Benutzer zuständig ist. Sie bietet Zugriff auf mehrere Teilsysteme, die Rechenoperationen ausführen können, beispielsweise Grafikkarten, die in der OpenCL-Terminologie als *Devices* bezeichnet werden. Informationen über die Devices erhalten wir ähnlich wie Informationen über Plattformen: Die Funktion `clGetDeviceIDs` lässt uns erfragen, wieviele Devices es in einer Plattform gibt und welche Identifikatoren ihnen zugeordnet sind, die Funktion `clGetDeviceInfo` bietet uns dann Zugriff auf eine Vielzahl von Informationen zu diesen Devices.

Die Funktion `clGetDeviceIDs` erwartet die Angabe des Plattform-Identifikators und eines weiteren Arguments, mit dem wir auswählen können, an welchem Typ von Device wir interessiert sind, beispielsweise `CL_DEVICE_TYPE_ALL` für alle Devices oder `CL_DEVICE_TYPE_GPU` für Grafikkarten. Die restlichen drei Parameter funktionieren wie bei `clGetPlatformIDs`:

```
clGetDeviceIDs(platform[i], CL_DEVICE_TYPE_ALL,
               0, 0, &nrdev);
dev = (cl_device_id *) malloc(sizeof(cl_device_id) * nrdev);
clGetDeviceIDs(platform[i], CL_DEVICE_TYPE_ALL,
               nrdev, dev, 0);
```

Die Funktion `clGetDeviceInfo` entspricht der Funktion `clGetPlatformInfo`, bietet allerdings wesentlich mehr Informationen: Neben `CL_DEVICE_NAME` für den Namen können wir beispielsweise `CL_DEVICE_GLOBAL_MEM_SIZE` verwenden, um die Speichergröße zu erfahren oder `CL_DEVICE_MAX_COMPUTE_UNITS` für die Anzahl der Recheneinheiten. Das folgende Programmfragment listet beispielsweise Informationen zu den Devices der `i`-ten Plattform auf:

```
for(j=0; j<nrdev; j++) {
  cl_ulong memsize;
```

```
    cl_uint compunits;
    cl_device_type devtype;
    clGetDeviceInfo(dev[j], CL_DEVICE_NAME, 0, 0, &namesize);
    name = (char *) malloc(sizeof(char) * namesize);
    clGetDeviceInfo(dev[j], CL_DEVICE_NAME,
                    namesize, name, 0);
    clGetDeviceInfo(dev[j], CL_DEVICE_GLOBAL_MEM_SIZE,
                    sizeof(cl_ulong), &memsize, 0);
    clGetDeviceInfo(dev[j], CL_DEVICE_MAX_COMPUTE_UNITS,
                    sizeof(cl_uint), &compunits, 0);
    clGetDeviceInfo(dev[j], CL_DEVICE_TYPE,
                    sizeof(cl_device_type), &devtype, 0);
    printf("\"%s\": %.1f KB memory, %d compute units",
           name, memsize / 1024.0, compunits);
    switch(devtype) {
    case CL_DEVICE_TYPE_CPU: printf(", type CPU\n"); break;
    case CL_DEVICE_TYPE_GPU: printf(", type GPU\n"); break;
    case CL_DEVICE_TYPE_ACCELERATOR: printf(", type ACC\n"); break;
    case CL_DEVICE_TYPE_DEFAULT: printf(", default type\n");
    }
    free(name);
  }
```

Da beispielsweise `CL_DEVICE_GLOBAL_MEM_SIZE` immer nur ein Element des Typs `cl_ulong` zurück gibt, können wir uns in diesem Fall mit einem einzigen Aufruf der Funktion `clGetDeviceInfo` begnügen, da es nicht erforderlich ist, in einem separaten Schritt die Größe des benötigten Speicherbereichs in Erfahren zu bringen.

Kontext. Sobald wir festgestellt haben, welche Hardware uns das OpenCL-System zur Verfügung stellt, müssen wir entscheiden, welche Devices wir in unserem Programm verwenden wollen. In OpenCL wird diese Information in einem *Kontext* festgehalten, einem Objekt des Typs `cl_context`. Wir können einen Kontext mit der Funktion `clCreateContext` anlegen:

```
cl_context_properties properties[3];
properties[0] = CL_CONTEXT_PLATFORM;
properties[1] = (cl_context_properties) platform[0];
properties[2] = 0;
context = clCreateContext(properties, nrdev, dev, 0, 0, &result);
```

Der erste Parameter der Funktion ist ein mit null endendes Array mit Einträgen des Typs `cl_context_properties`, die zusätzliche Eigenschaften des Kontexts definieren. In unserem Fall legen wir fest, dass der Kontext über der Plattform `platform[0]` verwaltet werden soll. Der zweite und dritte Parameter legen fest, wieviele und welche Devices der gewählten Plattform zu dem neu anzulegenden Kontext gehören sollen. Mit dem

vierten und fünften Parameter können wir eine Callback-Funktion definieren, die aufgerufen wird, falls Fehler auftreten. Im konkreten Beispiel verzichten wir darauf. Über den sechsten Parameter erhalten wir eine Rückmeldung über eventuell bei der Ausführung der Funktion aufgetretene Fehler. Dabei ist `result` eine Variable des Typs `cl_int`, die im Erfolgsfall gleich `CL_SUCCESS` ist.

OpenCL-Programme. Sobald uns ein OpenCL-Kontext zur Verfügung steht, können wir daran gehen, Programme für den Grafikprozessor zu übersetzen. Ein Programm ist dabei eine Folge von Strings, die in einem Array des Typs `const char **` zusammengefasst sind. Im einfachsten Fall genügt ein einziger String:

```
const char kernel_axpy[] =
  "__kernel void\n"
  "kernel_axpy(float alpha, __global const float *x,\n"
  "            __global float *y)\n"
  "{\n"
  "  int i = get_global_id(0);\n"
  "  y[i] += alpha * x[i];\n"
  "}\n";
```

Einen Zeiger auf diesen String schreiben wir in ein Array, das wir dann der Funktion `clCreateProgramWithSource` übergeben, die ein Objekt des Typs `cl_program` anlegt:

```
const char *source_axpy[1];
cl_program program_axpy;
source_axpy[0] = kernel_axpy;
program_axpy = clCreateProgramWithSource(context,
                                         1, source_axpy,
                                         0, &result);
```

Der erste Parameter der Funktion gibt den Kontext an, in dem das Programm definiert werden soll. Der dritte ist ein Array, das den Quelltext enthält, wobei der zweite Parameter die Länge des Arrays angibt. Mit dem vierten Parameter können wir Optionen für die Arbeit des OpenCL-Compilers definieren, der fünfte bietet uns die Möglichkeit, einen Ergebniscode zu erhalten.

Die Datenstruktur `cl_program` verwaltet alle Informationen, die für das Übersetzen von OpenCL-Programmen erforderlich sind: Quelltexte, übersetzter Maschinencode für verschiedene Devices, Compiler-Parameter sowie Fehlermeldungen des Compilers.

Sobald festgelegt ist, welches Programm mit welchen Parametern zu übersetzen ist, können wir das OpenCL-System anweisen, es in eine für Devices geeignete Maschinensprache zu übersetzen:

```
result = clBuildProgram(program_axpy, nrdev, dev, 0, 0, 0);
```

Der erste Parameter beschreibt das Programm, das übersetzt werden soll. Der zweite Parameter gibt an, für wieviele Devices es übersetzt werden soll, der dritte, für welche.

Mit dem vierten und fünften Parameter können wir eine Callback-Funktion definieren, die aufgerufen wird, sobald das Programm übersetzt wurde. Falls der vierte Parameter ungleich null ist, kann der Funktionsaufruf nicht-blockierend arbeiten und seine Fertigstellung über die Callback-Funktion melden, anderenfalls muss das aufrufende Programm warten, bis der OpenCL-Übersetzer seine Arbeit getan hat.

Wie die meisten OpenCL-Funktionen gibt auch `clBuildProgram` einen Ergebniscode des Typs `cl_int` zurück. Falls dieser Code ungleich `CL_SUCCESS` ist, sollten wir mit Hilfe der Funktion `clGetProgramBuildInfo` in Erfahrung bringen, weshalb das Programm nicht erfolgreich übersetzt werden konnte. Die Funktion erwartet als ersten und zweiten Parameter das Programm und das Device, der dritte Parameter gibt an, welche Informationen wir erfahren wollen. Die drei letzten Parameter funktionieren wie die, die wir bereits bei `clGetPlatformInfo` und `clGetDeviceInfo` kennen gelernt haben. Die Liste der Compiler-Fehlermeldungen für das Programm `program` und das Device `device` erhalten wir beispielsweise mit dem folgenden Programmfragment:

```
if(result != CL_SUCCESS) {
  clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG,
                        0, 0, &log_size);
  build_log = (char *) malloc((size_t) sizeof(char) * log_size);
  clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG,
                        log_size, build_log, 0);
  printf("%s", build_log);
  free(build_log);
}
```

Kernfunktionen. Die Daten für die Ausführung einer Kernfunktion werden im Rahmen des OpenCL-Systems durch Objekte des Typs `cl_kernel` beschrieben, die einerseits die aufzurufende Funktion und andererseits ihre Parameter angeben. Die Funktion `clCreateKernel` legt eines dieser Objekte an:

```
cl_kernel kernel_axpy;
kernel_axpy = clCreateKernel(program, "kernel_axpy", &result);
```

Die für den Aufruf der Kernfunktion erforderlichen Parameter können wir mit der Funktion `clSetKernelArg` festlegen:

```
clSetKernelArg(kernel_axpy, 0, sizeof(float), &alpha);
```

Der erste Parameter ist jeweils die Kernfunktion, der zweite die Nummer des zu setzenden Parameter, der dritte die Größe des Parameters und der vierte schließlich dessen Wert. Die hier untersuchte Kernfunktion `kernel_axpy` erwartet als zweiten und dritten Parameter Zeiger auf den Grafikspeicher, und für diese Zeiger müssen wir etwas zusätzlichen Aufwand betreiben.

Grafikspeicher. Speicherbereiche außerhalb des Hauptspeichers werden in OpenCL durch Objekte des Typs `cl_mem` beschrieben, die mit Hilfe der Funktion `clCreateBuffer` angelegt werden können:

```
d_x = clCreateBuffer(context, CL_MEM_READ_ONLY,
                     (size_t) sizeof(float) * n, 0, &result);
d_y = clCreateBuffer(context, CL_MEM_READ_WRITE,
                     (size_t) sizeof(float) * n, 0, &result);
```

Der erste Parameter gibt den Kontext der Operation an, über den zweiten können wir zusätzliche Parameter übergeben, um beispielsweise für `d_x` festzuhalten, dass Kernfunktionen auf den entsprechenden Speicherbereich nur lesend zugreifen dürfen. Der dritte Parameter gibt die Größe des gewünschten Speicherbereichs an, mit dem vierten könnten wir Bereich des Hauptspeichers als Pufferspeicher zur Verfügung stellen. Mit dem fünften Parameter erhalten wir wie üblich einen Ergebniscode, der hoffentlich gerade `CL_SUCCESS` ist.

Wenn die Kernfunktion einen Zeiger als Parameter erwartet, können wir ihr ein Objekt des Typs `cl_mem` übergeben:

```
clSetKernelArg(kernel_axpy, 1, sizeof(cl_mem), &d_x);
clSetKernelArg(kernel_axpy, 2, sizeof(cl_mem), &d_y);
```

Damit sind sowohl unsere Kernfunktion als auch ihre Parameter vollständig definiert, jetzt müssen wir sie „nur noch“ ausführen.

Queue. Sowohl Kopieroperationen als auch die Ausführung von Kernfunktionen können in OpenCL asynchron stattfinden. Die einzelnen Arbeitsaufträge werden über Warteschlangen verwaltet, die im OpenCL-Standard als *Queues* bezeichnet und durch den Datentyp `cl_command_queue` dargestellt werden. Eine Queue für ein Device `device` kann mit Hilfe der Funktion `clCreateCommandQueue` angelegt werden:

```
queue = clCreateCommandQueue(context, device, 0, &result);
```

Die ersten beiden Parameter sind wieder Kontext und Device, über den dritten können zusätzliche Paramter übergeben werden, der vierte ermöglicht uns wie üblich den Zugriff auf einen Ergebniscode.

OpenCL-Queues sind CUDA-Streams sehr ähnlich, und spielen ebenfalls eine wichtige Rolle, wenn wir für eine gute Auslastung der Rechen- und Kopiereinheiten sorgen wollen, indem wir dem System die Möglichkeit geben, mehrere Arbeitsaufträge nebenläufig auszuführen.

Um unsere Kernfunktion ausführen zu können, müssen wir zunächst Daten in den Grafikspeicher übertragen. Diesem Zweck dient die Funktion `clEnqueueWriteBuffer`:

```
clEnqueueWriteBuffer(queue, d_x, CL_TRUE, 0,
                     (size_t) sizeof(float) * n, x, 0, 0, 0);
clEnqueueWriteBuffer(queue, d_y, CL_TRUE, 0,
                     (size_t) sizeof(float) * n, y, 0, 0, 0);
```

Der erste Parameter gibt die Queue an, in der die Kopieroperation ausgeführt werden soll, der zweite das Ziel. Mit dem dritten Parameter können wir festlegen, ob die Operation blockierend (Wert `CL_TRUE`) oder nicht-blockierend ausgeführt werden soll. Der vierte Parameter legt die Menge der zu übertragenden Daten fest und der fünfte die Quelle. Die letzten drei Parameter dienen der Synchronisation unterschiedlicher Operationen und werden später behandelt.

Nachdem `d_x` und `d_y` mit Daten gefüllt sind, können wir (endlich) die Kernfunktion ausführen:

```
clEnqueueNDRangeKernel(queue, kernel_axpy,
                       1, 0, &n, &blksize,
                       0, 0, 0);
```

Dieser Aufruf ergänzt die Queue `queue` um den Aufruf der Kernfunktion `kernel_axpy`. Wie schon in CUDA wird die Kernfunktion nicht nur einmal aufgerufen, sondern es wird für alle Elemente einer potentiell mehrdimensionalen Indexmenge jeweils ein Thread erzeugt. Der dritte Parameter gibt die Dimension dieser Indexmenge an (im Beispiel genügt uns eine Dimension), der vierte und fünfte sind Arrays dieser Dimension, die für jede Dimension den Startindex und die Anzahl der Indizes angeben. Als Sonderfall dürfen wir für den vierten Parameter auch den Nullzeiger angeben, um festzulegen, dass die Indizes in jeder Dimension bei null anfangen sollen. Da wir hier nur eine eindimensionale Indexmenge verwenden, können wir mit `&n` ein Array der Länge 1 erhalten, das nur den Wert `n` enthält.

Wie in CUDA können wir die Threads zu Threadblöcken zusammenfassen, die über geteilten Speicher Daten austauschen und sich untereinander synchronisieren können. Die Größen dieser Blöcke, wieder pro Richtung, werden durch den sechsten Parameter beschrieben. Die letzten drei Parameter sind wieder für die Synchronisation unterschiedlicher Operationen zuständig und werden später diskutiert.

Die Ausführung der Kernfunktion erfolgt wieder asynchron. Wir können entweder mit Hilfe der Funktion `clFinish` darauf warten, dass alle Aufträge in einer Queue ausgeführt worden sind, oder ausnutzen, dass die Aufträge in der Queue in der Reihe ausgeführt werden, in der sie eingetragen wurden, so dass wir mit Hilfe einer blockierenden Kopieroperation implizit auch auf die Ausführung der Kernfunktion warten:

```
clEnqueueReadBuffer(queue, d_y, CL_TRUE, 0,
                    (size_t) sizeof(float) * n, y, 0, 0, 0);
```

Da diese Operation derselben Queue wie der Aufruf der Kernfunktion hinzugefügt wird, wartet sie, bis dieser Aufruf abgeschlossen wurde. Da als dritter Parameter `CL_TRUE` verwendet wird, blockiert der Aufruf das Hauptprogramm, bis er erfolgreich abgeschlossen wurde.

Mit Hilfe der Funktion `clFinish` kann das Hauptprogramm darauf warten, dass alle Aufträge in einer Queue abgearbeitet wurden. Mit `clFinish` können wir darauf warten, dass alle Aufträge an das zugeordnete Device übertragen wurden, auch wenn ihre Bearbeitung noch nicht abgeschlossen ist.

Vollständiges Beispielprogramm. Das vollständige Programm für die Berechnung der Linearkombination nimmt die folgende Form an, wenn wir die erste OpenCL-Plattform und das erste OpenCL-Device dieser Plattform verwenden:

```
const char *src_axpy[] = {
  "__kernel void\n"
  "kernel_axpy(float alpha, __global const float *x,\n"
  "            __global float *y)\n"
  "{ int i = get_global_id(0); y[i] += alpha * x[i]; }\n" };

clGetPlatformIDs(1, &platform, 0);
clGetDeviceIDs(1, CL_DEVICE_TYPE_GPU, 1, &device, 0);
context_properties[0] = CL_CONTEXT_PLATFORM;
context_properties[1] = (cl_context_properties) platform;
context_properties[2] = 0;
context = clCreateContext(context_properties, 1, &device,
                          0, 0, &result);
program = clCreateProgramWithSource(context, 1, src_axpy, 0,
                                    &result);
result = clBuildProgram(program, 1, &device, 0, 0, 0);
kernel_axpy = clCreateKernel(program, "kernel_axpy", &result);
queue = clCreateCommandQueue(context, device, 0, &result);
d_x = clCreateBuffer(context, CL_MEM_READ_ONLY,
                     (size_t) sizeof(float) * n, 0, &result);
d_y = clCreateBuffer(context, CL_MEM_READ_WRITE,
                     (size_t) sizeof(float) * n, 0, &result);
clEnqueueWriteBuffer(queue, d_x, CL_TRUE, 0,
                     (size_t) sizeof(float) * n, x, 0, 0, 0);
clEnqueueWriteBuffer(queue, d_y, CL_TRUE, 0,
                     (size_t) sizeof(float) * n, y, 0, 0, 0);
clSetKernelArg(kernel_axpy, 0, sizeof(float), &alpha);
clSetKernelArg(kernel_axpy, 1, sizeof(cl_mem), &d_x);
clSetKernelArg(kernel_axpy, 2, sizeof(cl_mem), &d_y);
result = clEnqueueNDRangeKernel(queue, kernel_axpy, 1, 0, &n,
                                &blksize, 0, 0, 0);
clEnqueueReadBuffer(queue, d_y, CL_TRUE, 0,
                    (size_t) sizeof(float) * n, y, 0, 0, 0);
```

Bei einer praktischen Implementierung empfiehlt es sich aus den bereits bei der Behandlung von CUDA erwähnten Gründen, die von den verschiedenen Funktionen zurückgegebenen Ergebniscodes zu überprüfen, um eventuell aufgetretene Fehler abzufangen. Das ist insbesondere bei dem Aufruf der Funktion `clBuildProgram` empfehlenswert, um Fehler im Quelltext erkennen zu können.

Nebenläufiges Kopieren und Rechnen. Bei der Verarbeitung von Datenmengen, die nicht vollständig in den Grafikspeicher passen, müssen wir selbstverständlich wieder Daten zwischen Grafik- und Hauptspeicher kopieren. Wie schon bei CUDA empfiehlt es sich, dabei dafür zu sorgen, dass Rechen- und Kopieroperationen möglichst nebenläufig stattfinden können, damit eine hohe Effizienz erreicht wird.

Im CUDA-Fall haben wir diese Aufgabe mit Hilfe von Streams gelöst, in OpenCL spielen Queues dieselbe Rolle: Im einfachsten Fall legen wir zwei Queues an, die wir abwechselnd mit der Verarbeitung je eines Teils unseres Datenarrays beauftragen. Das resultierende Programmfragment sieht dem aus dem CUDA-Beispiel sehr ähnlich:

```
off = 0; current = 0;
while(off < n) {
  chunksize = n - off;
  if(chunksize > blocksize * gridsize)
    chunksize = blocksize * gridsize;
  clEnqueueWriteBuffer(queue[current], d_x[current], CL_FALSE, 0,
                       sizeof(float) * chunksize, x+off, 0, 0, 0);
  clEnqueueWriteBuffer(queue[current], d_y[current], CL_FALSE, 0,
                       sizeof(float) * chunksize, y+off, 0, 0, 0);
  clSetKernelArg(kernel_axpy, 0, sizeof(float), &alpha);
  clSetKernelArg(kernel_axpy, 1, sizeof(cl_mem), d_x+current);
  clSetKernelArg(kernel_axpy, 2, sizeof(cl_mem), d_y+current);
  clEnqueueNDRangeKernel(queue[current], kernel_axpy, 1, 0,
                         &chunksize, &blocksize, 0, 0, 0);
  clEnqueueReadBuffer(queue[current], d_y[current], CL_FALSE, 0,
                      sizeof(float) * chunksize, y+off, 0, 0, 0);
  off += chunksize; current = (current + 1) % 2;
}
clFinish(queue[0]); clFinish(queue[1]);
```

Damit dieses Programm wie gewünscht funktioniert, müssen zwei Queues `queue[0]` und `queue[1]` für das Device angelegt worden sein, und auch die Speicherbereiche `d_x` und `d_y` müssen doppelt vorliegen, damit parallel kopiert und gerechnet werden kann.

Im vorliegenden Beispielfall werden wir allerdings von der Geschwindigkeit des Programms auf handelsüblichen Computern sehr enttäuscht sein: Das Kopieren der Daten in den und aus dem Grafikspeicher geht mit einer Bandbreite vonstatten, die deutlich unter der des Hauptspeichers liegt, so dass die Grafikkarte ihre hohe Rechenleistung nicht ausspielen kann, sondern den größten Teil ihrer Zeit damit verbringt, auf die Kopieroperationen zu warten.

Events. Innerhalb eines OpenCL-Kontexts können wir mehrere Queues für mehrere Devices anlegen und mit Arbeitsaufträgen füllen. Damit stellt sich die Frage, wie wir eine Synchronisation zwischen diesen Aufträgen erreichen können, wie wir also beispielsweise sicherstellen können, dass die Ergebnisse einer Rechenoperation erst in den Hauptspeicher kopiert werden, wenn sie vollständig vorliegen. Unser Beispielprogramm erreicht

dieses Ziel, weil die in einer Queue enthaltenen Befehle im Normalfall in der Reihenfolge ausgeführt werden, in der sie in die Queue eingetragen wurden (*in-order execution*). Manche OpenCL-Implementierungen bieten auch Queues, die Aufträge im Interesse der Effizienz umsortieren können (*out-of-order execution*), dieses Verhalten muss allerdings bei der Einrichtung der Queue explizit angefordert werden.

Im Allgemeinen können wir *Events* verwenden, um eine Synchronisation zwischen einzelnen Aufträgen zu erreichen. Ein Event wird durch den Typ `cl_event` beschrieben und beschreibt das Eintreten eines Ereignisses durch eine Variable, die die Werte `CL_SUBMITTED`, `CL_QUEUED`, `CL_RUNNING`, `CL_COMPLETE` oder einen negativen Wert annehmen kann. Die Werte `CL_SUBMITTED` und `CL_QUEUED` bedeuten, dass das Event noch nicht eingetreten ist. Der Wert `CL_RUNNING` gibt bei Events, die mit der Ausführung einer Operation verbunden sind, an, dass die betreffende Operation gerade ausgeführt wird. Der Wert `CL_COMPLETE` schließlich bedeutet, dass das Event eingetreten ist, dass also beispielsweise die zugehörige Operation erfolgreich abgeschlossen wurde. Ein negativer Wert dagegen signalisiert einen erfolglosen Abschluss und kann als Fehlercode interpretiert werden. Mit geeigneten Funktionen können wir Ereignisse auslösen oder auf das Eintreten von Ereignissen warten.

In dieser Weise können wir sicherstellen, dass Aufträge in einer Queue auf das Eintreten eines Ereignisses oder mehrerer Ereignisse warten, bevor sie ausgeführt werden. Wir können auch dafür sorgen, dass Aufträge ein Ereignis auslösen, sobald sie erfolgreich abgeschlossen wurden, so dass wir die Ausführung eines Auftrags elegant von dem Abschluss anderer Aufträge abhängig machen können.

Diesem Zweck dienen beispielsweise die letzten drei Argumente, die wir bei den Funktionen `clEnqueueNDRangeKernel`, `clEnqueueReadBuffer` und `clEnqueueWriteBuffer` bisher ignoriert haben: Das erste Argument gibt die Anzahl der Events an, auf die gewartet werden soll, bevor eine Aufgabe begonnen werden kann. Das zweite Argument ist ein Array von Events, das diese abzuwartenden Events auflistet. Das dritte Argument ist ein Zeiger auf ein Event, das eintreten soll, sobald der Auftrag ausgeführt wurde.

Im Fall der Linearkombination können wir beispielsweise drei Queues einrichten: Eine ist für das Schreiben in den Grafikspeicher zuständig, eine für das Lesen aus dem Grafikspeicher und eine für die eigentliche Berechnung. Die Berechnung soll erst beginnen, wenn die nötigen Daten im Grafikspeicher angekommen sind, und die Daten sollen erst aus dem Grafikspeicher in den Hauptspeicher transportiert werden, wenn das Ergebnis berechnet wurde. Diese Aufgabe löst das folgende Programmfragment:

```
cl_event xywritten[2], ycomputed;

write_queue = clCreateCommandQueue(context, device, 0, &result);
compute_queue = clCreateCommandQueue(context, device, 0, &result);
read_queue = clCreateCommandQueue(context, device, 0, &result);

clEnqueueWriteBuffer(write_queue, d_x, CL_FALSE, 0,
                     sizeof(float) * n, x, 0, 0, xywritten);
clEnqueueWriteBuffer(write_queue, d_y, CL_FALSE, 0,
```

```
                        sizeof(float) * n, y2, 0, 0, xywritten+1);
clEnqueueNDRangeKernel(compute_queue, kernel_axpy, 1, 0, &n,
                          &blksize, 2, xywritten, &ycomputed);
clEnqueueReadBuffer(read_queue, d_y, CL_TRUE, 0,
                       sizeof(float) * n, y2, 1, &ycomputed, 0);
```

Obwohl die Schreib-, Rechen- und Leseaufträge in unterschiedliche Queues eingetragen werden, ist sichergestellt, dass das Programm das richtige Ergebnis berechnet: Erst wenn `x` im Grafikspeicher steht tritt das Ereignis `xywritten[0]` ein, erst wenn `y` angekommen ist das Ereignis `xywritten[1]`. Die Ausführung der Kernfunktion `kernel_axpy` wartet, bis diese beiden Ereignisse eingetreten sind. Sobald die Ausführung abgeschlossen wurde, wird das Ereignis `ycomputed` ausgelöst, auf das wiederum der Leseauftrag wartet.

Der Hauptprozessor kann mit der Funktion `clCreateUserEvent` ein Event anlegen und mit `clSetUserEventStatus` sein Eintreten signalisieren. Er kann mit der Funktion `clWaitForEvents` darauf warten, dass ein Event oder mehrere eintreten, und er kann mit `clSetEventCallback` eine Callback-Funktion registrieren lassen, die unabhängig vom Hauptprogramm aufgerufen wird, sobald ein Event eintritt. Mit der Funktion `clGetEventInfo` können wir Informationen über ein Event erfragen, beispielsweise seinen Zustand, die korrespondierende Queue oder den korrespondierenden Kontext.

Entsprechend kann ein Device mit der Funktion `clEnqueueMarker` dafür sorgen, dass ein Event ausgelöst wird, sobald alle bisher in eine Queue eingetragenen Aufträge abgearbeitet wurden und mit `clEnqueueWaitForEvents` sicherstellen, dass die in der Queue folgenden Aufträge erst abgearbeitet werden, sobald ein Event oder mehrere eingetreten sind.

Profiling. Mit Hilfe der durch das Betriebssystem des Hosts zur Verfügung gestellten Funktionen für die Zeitmessung erhalten wir in der Regel nur ungenaue Ergebnisse, die sich zwar für die Beurteilung der Qualität eines Gesamtprogramms eignen, aber weniger für die Suche nach Schwachstellen in einzelnen Programmteilen.

Um die Optimierung des Programms zu vereinfachen, können Events auch für die Zeitmessung verwendet werden. Damit die entsprechenden Informationen gesammelt werden, muss die Queue mit dem Parameter `CL_QUEUE_PROFILING_ENABLE` im dritten Argument angelegt werden. Wenn wir in eine solche Queue den Aufruf einer Kernfunktion aufnehmen, können wir über das letzte Argument der Funktion `clEnqueueNDRangeKernel` ein Event anlegen, das für die Zeitmessung verwendet werden kann:

```
clEnqueueNDRangeKernel(queue, kernel_axpy, 1, 0, &n,
                          &blksize, 0, 0, &compute_axpy);
```

Die Funktion `clGetEventProfilingInfo` arbeitet ähnlich wie `clGetPlatformInfo` und `clGetDeviceInfo` und kann etwa mit den Parametern `CL_PROFILING_COMMAND_START` und `CL_PROFILING_COMMAND_END` aufgerufen werden, um die (Device-interne) Zeit in Nanosekunden zu erfahren, zu der die Kernfunktion gestartet und beendet wurde:

```
cl_ulong t_start, t_end;
```

```
clGetEventProfilingInfo(compute_axpy, CL_PROFILING_COMMAND_START,
                        sizeof(cl_ulong), &t_start, 0);
clGetEventProfilingInfo(compute_axpy, CL_PROFILING_COMMAND_END,
                        sizeof(cl_ulong), &t_end, 0);
printf("%lu nanoseconds for kernel_axpy\n", t_end-t_start);
```

8.4 OpenCL C

Auch OpenCL definiert eine Variante der Programmiersprache C, in der für ein Device vorgesehene Programme formuliert werden, in diesem Fall heißt die Sprache naheliegenderweise OpenCL C. Da die in OpenCL C geschriebenen Programme nicht von einem vollwertigen Betriebssystem ausgeführt werden, sondern lediglich in der eingeschränkten Umgebung des Devices, verzichtet die Sprache auf eine Laufzeitbibliothek, also beispielsweise auf Funktionen wie `printf` oder `malloc`. Da die Programme aus einem String und nicht aus einer Datei übersetzt werden, gibt es auch kein `#include`, mit dem sich weitere Dateien einbinden ließen. Um dem Compiler die Optimierung des Maschinenprogramms zu erleichtern, sind sowohl rekursive Funktionsaufrufe als auch Funktionszeiger nicht vorgesehen.

Allerdings bietet OpenCL C auch eine Reihe von Möglichkeiten, die in ANSI C fehlen. Beispielsweise wird die fehlende Laufzeitbibliothek durch eine große Anzahl mathematischer Funktionen kompensiert, die der Compiler direkt umsetzen kann. Da Grafikprozessoren eine enge Verwandtschaft mit Vektorrechnern aufweisen, bietet OpenCL C außerdem eine weitreichende Unterstützung für Vektor-Datentypen unterschiedlicher Dimension, die sich auch auf die erwähnten mathematischen Funktionen erstreckt.

Speicher. Grafikprozessoren und manche Koprozessoren verfügen über unterschiedliche Arten von Speicher: Grafikspeicher ist in der Regel im Umfang einiger Gigabytes vorhanden, lokaler Speicher eher im Umfang einiger Kilobytes, Register stehen pro Thread eher noch deutlich weniger zur Verfügung.

Wir können bei der Deklaration von Variablen festlegen, in welcher Sorte von Speicher sie abgelegt werden sollen: Das Schlüsselwort `__global` weist Variablen einen Platz im Grafikspeicher zu.

Das Schlüsselwort `__local` lässt die Variable in dem für jede Threadgruppe (in OpenCL *work-group* genannt) geteilten Speicher unterbringen, so dass sie für die Kommunikation zwischen den Threads einer Gruppe verwendet werden kann. In CUDA C entspricht `__shared__` diesem Schlüsselwort.

Das Schlüsselwort `__constant` bezeichnet „Variablen“, die das auf dem Device laufende Programm nicht verändern kann, sie müssen ihre Werte entweder bei der Übersetzung des Programms erhalten oder durch eine Zuweisung des Hosts.

Mit `__private` schließlich sind Variablen markiert, die nur innerhalb eines Threads (in OpenCL *work-item* genannt) gültig sind. Bei Grafikkarten ist damit zu rechnen, dass diese Variablen bevorzugt durch Register dargestellt werden, die sich alle bei einem

Kern-Aufruf angelegten Threads teilen müssen. Man sollte darauf achten, nicht zu viele private Variablen zu definieren, um Fehlermeldungen oder — noch schlimmer — eine reduzierte Effizienz zu vermeiden.

Kernfunktionen. OpenCL-Funktionen, die durch den Host aufgerufen werden können, werden Kernfunktionen genannt und müssen mit dem Schlüsselwort `__kernel` gekennzeichnet werden, beispielsweise damit die Funktion `clCreateKernel` sie identifizieren kann. Sie dürfen keine Werte zurückgeben, müssen also als `void` deklariert sein. Neben elementaren Datentypen dürfen sie auch Zeiger auf Daten im Grafikspeicher (Schlüsselwort `__global`) und im lokalen geteilten Speicher (Schlüsselwort `__local`) als Parameter erhalten, allerdings müssen letztere bei einem Aufruf durch den Host immer den Nullzeiger enthalten, sind also nur von Nutzen, wenn die Kernfunktion auch von anderen Funktionen auf dem Device aufgerufen werden soll.

Der Host kann eine Kernfunktion mit zwei Befehlen aufrufen: `clEnqueueTask` trägt einen Auftrag für die Ausführung einer Kernfunktion mit einem einzigen Thread und einer einzigen Threadgruppe in eine Queue ein. Derartige Kernfunktionen sind wahrscheinlich auf Devices mit Vektorrechner-Struktur, beispielsweise auf Grafikkarten, nicht allzu nützlich, sofern sie nicht eine Vektorisierung explizit vornehmen und darin durch einen hinreichend geschickten Compiler unterstützt werden.

Der Normalfall ist der Aufruf einer Kernfunktion mit `clEnqueueNDRangeKernel`, mit der mehrere Threads (bzw. in OpenCL-Nomenklatur *work-items*) in mehreren Threadgruppen (bzw. *work-groups*) gleichzeitig gestartet werden. Damit die Threads unterschieden werden können, wird jedem ein individueller Index aus einer n-dimensionalen Indexmenge zugewiesen. Die Threads werden zu Threadgruppen zusammengefasst, die ebenfalls mit einem n-dimensionalen Index versehen werden, und innerhalb derer ein einzelner Thread durch einen lokalen Index identifiziert werden kann. Threadgruppen sind wichtig, weil alle Threads einer Threadgruppe sich den lokalen Speicher teilen und deshalb beispielsweise besonders effizient untereinander Daten austauschen können.

Die Funktion `clEnqueueNDRangeKernel` erwartet als Parameter die Queue, in die der Auftrag eingereiht werden soll, die Kernfunktion, die ausgeführt werden soll, und eine Beschreibung der Indexmengen. Wie bereits erwähnt können dabei auch Events angegeben werden, auf die gewartet werden soll und die ausgelöst werden sollen, sobald die Ausführung der Kernfunktion abgeschlossen wurde. Die zu bearbeitende Indexmenge wird durch das dritte Argument `work_dim`, das vierte Argument `global_work_offset`, das fünfte Argument `global_work_size` und das sechste Argument `local_work_size` definiert: Die Indexmenge hat die Dimension `work_dim` und die Form

```
{ global_work_offset[0],...
        ..., global_work_offset[0]+global_work_size[0]-1}×
{ global_work_offset[1],...
        ..., global_work_offset[1]+global_work_size[1]-1}×
    ⋮
```

{ `global_work_offset[work_dim-1]`, ...
..., `global_work_offset[work_dim-1]+global_work_size[work_dim-1]-1`}.

In jeder Komponenten `i` zwischen `0` und `work_dim-1` gibt also `global_work_offset[i]` jeweils den ersten und `global_work_size[i]` die Anzahl der aufsteigend numerierten Indizes an. Jeder dieser Indizes gehört zu einem Thread, der in OpenCL-Notation als *Work-Item* bezeichnet wird.

Work-Items werden zu größeren Gruppen zusammengefasst, die in OpenCL *Work-Groups* genannt werden. Die Work-Items derselben Work-Group können sich miteinander synchronisieren und über einen kleinen geteilten Speicher schnell Daten untereinander austauschen. Die Work-Groups korrespondieren mit disjunkten Teilmengen der gesamten Indexmenge, die wieder die Gestalt eines `work_dim`-dimensionalen kartesischen Produkts mit jeweils `local_work_size[i]` vielen Elementen in der `i`-ten Komponente aufweisen. Dabei ist vorausgesetzt, dass `global_work_size[i]` durch `local_work_size[i]` teilbar ist. Ein Beispiel ist in Abbildung 8.3 dargestellt, bei dem eine 8×6-elementige zweidimensionale Indexmenge in 2×3-elementige Teilmengen zerlegt wird.

Innerhalb der Kernfunktion können wir mit der Funktion `get_work_dim` die Dimension der Indexmenge erfahren, mit `get_global_id` den globalen Index, dessen `i`-te Komponente wie bereits erwähnt zwischen `global_work_offset[i]` und `global_work_offset[i]+global_work_size[i]-1` liegt, mit `get_local_id` den Index innerhalb der aktuellen Work-Group, der zwischen `0` und `local_work_size[i]-1` liegt, und mit `get_group_ip` den Index der Work-Group selber, der zwischen `0` und `global_work_size[i]/local_work_size[i]-1` liegt.

Die Abmessungen der Indexmengen können wir bei Bedarf mit den Funktionen `get_global_offset`, `get_global_size`, `get_local_size` sowie `get_num_groups` in Erfahrung bringen.

Die globalen und lokalen Indizes stehen über die Formel

```
get_global_id(i) = get_global_offset(i)
                 + get_local_id(i)
                 + get_group_id(i) * get_local_size(i)
```

miteinander in Beziehung. In typischen Programmen dürfte in der Regel der globale Index für Zugriffe auf den Grafikspeicher verwendet werden, während der lokale Index für Zugriffe auf den innerhalb einer Work-Group geteilten Speicher zum Einsatz kommt.

Datentypen. Neben den üblichen Ganzzahl-Datentypen `char`, `int`, `long`, `short` und dem Gleitkomma-Datentyp `float` kennt OpenCL C auch die vorzeichenlosen Typen `uchar`, `uint`, `ulong` und `ushort`. Zusätzlich kann ein Device auch den Gleitkomma-Datentyp `double` doppelter Genauigkeit und den Typ `half` halber Genauigkeit zur Verfügung stellen. Letzterer darf allerdings nur für Zeiger verwendet werden, wir dürfen keine `half`-Variablen definieren und auch keine Rechenoperationen mit ihnen durchführen. Wir dürfen mit der Funktion `vload_half` eine `half`-Variable aus dem

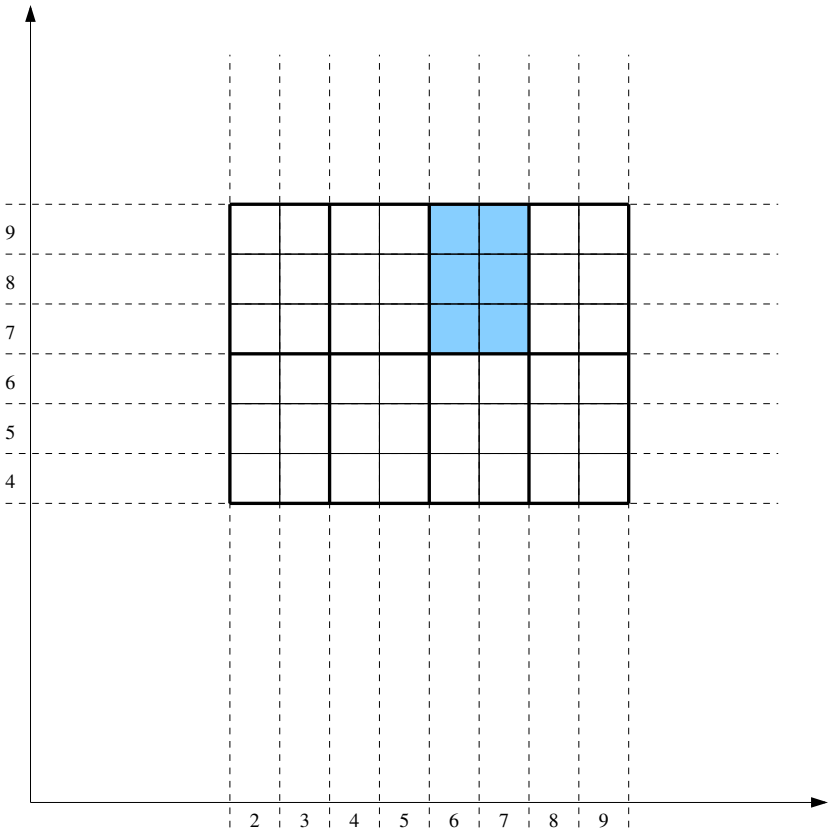

Abbildung 8.3: Indexmenge für die Anfangsindizes `global_work_offset[]`={2,4}, die globalen Mächtigkeiten `global_work_size[]`={8,6} sowie die Work-Group-Mächtigkeiten `local_work_size[]`={2,3}. Markiert ist die Work-Group mit dem Index `(3,1)`.

Speicher lesen und dabei in eine `float`-Variable konvertieren, und wir können mit `vstore_half` eine `float`-Variable auf `half`-Genauigkeit runden und in den Speicher schreiben.

Hinzu kommen `bool` für Wahrheitswerte, `size_t` für Speichergrößen, `ptrdiff_t` für Differenzen zwischen Zeigern, `intptr_t` und `uintptr_t` für die Konvertierung von Zeigern in ganze Zahlen und zurück sowie `void` für Elemente der leeren Menge.

Vektortypen. Da Grafikprozessoren eng verwandt mit Vektorrechnern sind, überrascht es nicht, dass OpenCL C das Arbeiten mit Vektordatentypen besonders elegant unterstützt: Alle Zahlentypen können zu Vektoren mit 2, 3, 4, 8 und 16 Komponenten zusammengefasst werden, die sich simultan verarbeiten lassen. Dazu genügt es, den Datentyp um die entsprechende Zahl zu erweitern, beispielsweise ist `float8` ein Vektor aus 8 `float`-Komponenten. Die Komponenten können wie bei einem `struct` angesprochen werden und tragen dann die Namen `s0`, `s1`, ..., `s9`, `sa`, ..., `sf`. Es ist zu beachten, dass

bei 16-dimensionalen Vektoren der Index hexadezimal angegeben wird, `sa` steht also für die zehnte Komponente und `sf` für die fünfzehnte. Indem wir mehrere Indizes angeben, können wir Teilvektoren direkt zusammensetzen:

```
float8 a;
float4 b = a.s0351;
float2 c = a.s22;
float16 d = a.s0716253470615243;
```

In diesem Beispiel würde `b` die Komponenten 0, 3, 5 und 1 des Vektors `a` aufnehmen, `c` zweimal die Komponente 2 und `d` mehrere permutierte Kopien der Einträge von `a`.

Aufgrund der Nähe zu Grafikkarten bietet OpenCL C für die in diesem Kontext sehr häufig verwendeten vierdimensionalen Vektoren die Kurzschreibweisen `x`, `y`, `z` und `w` für die vier Komponenten an:

```
float4 a;
float4 b = a.wzxy;
float2 c = a.xy;
```

Zusätzlich können mit `lo` und `hi` die erste beziehungsweise letzte Hälfte eines Vektors angesprochen werden und mit `odd` und `even` seine gerad- beziehungsweise ungeradzahligen Komponenten. Die Funktionen `shuffle` und `shuffle2` können eingesetzt werden, um zur Laufzeit ausgewählte Komponenten der Vektoren zu kombinieren.

Besonders elegant wird der Umgang mit Vektortypen dadurch, dass die meisten der in OpenCL enthaltenen mathematischen Funktionen sie direkt unterstützen. Beispielsweise dürfen wir

```
float4 a = { 1.0, 2.0, -5.0, 0.0 };
float4 b = cos(a);
float4 c = exp(a);
```

schreiben und erhalten in `b` und `c` die Vektoren, die sich aus der komponentenweisen Anwendung der Cosinus- und Exponentialfunktion ergeben.

Eingebaute Funktionen. Da OpenCL C keine Bibliotheken verwendet, sind viele Funktionen fest eingebaut, beispielsweise die üblichen mathematischen Funktionen wie `sin`, `cos`, `exp` und so weiter. Wie bereits erwähnt können sie nicht nur auf skalare Datentypen angewendet werden, sondern auch auf Vektoren. Für einige der Funktionen bietet OpenCL C auch eine von der Implementierung abhängige Variante, beispielsweise `native_exp` statt `exp`, die eventuell nicht die im Standard geforderte Genauigkeit erreicht, aber möglicherweise schneller ausgeführt werden kann.

Hinzu kommen nützliche Funktionen für die Umwandlung von Typen ineinander. Beispielsweise können wir mit `convert_int` eine Zahl in einen `int`-Wert umwandeln und dabei in Richtung der Null abrunden. Falls wir auch negative Zahlen abrunden wollen, können wir `convert_int_rtn` („round to negative infinity“) verwenden, für das Aufrunden ist `convert_int_rtp` („round to positive infinity“) zuständig.

Um beispielsweise direkt mit der Binärdarstellung einer Gleitkommazahl arbeiten zu können, wie wir es in Kapitel 4 getan haben, bietet es sich an, mit der Funktion `as_uint` eine `float`-Variable als `uint` zu interpretieren, die gewünschten Manipulationen vorzunehmen und das Ergebnis dann mit `as_float` wieder als Gleitkommazahl zu sehen. Anders als die `convert`-Funktionen verändern die `as`-Funktionen nicht die binäre Darstellung, sondern nur die Interpretation dieser Darstellung.

Kooperation innerhalb einer Work-Group. Die Work-Items innerhalb einer Work-Group können Daten untereinander austauschen und den lokalen geteilten Speicher gemeinsam verwenden. Dafür ist einerseits eine Synchronisation erforderlich, andererseits wären spezialisierte Funktionen für den Zugriff auf den lokalen Speicher hilfreich.

Die Synchronisation innerhalb einer Work-Group (also nur zwischen den Work-Items in der Work-Group, nicht zwischen allen Work-Items) kann am einfachsten über die Funktion `barrier` erfolgen, aus der ein Thread erst dann zurückkehrt, wenn alle Threads seiner Work-Group sie aufgerufen haben. Die Funktion erwartet ein Argument des Typs `cl_mem_fence_flags`, mit dem sich festlegen lässt, ob bei erreichen der Barriere auch sichergestellt werden soll, dass alle Variablen an ihre vorgesehenen Stellen im lokalen oder globalen Speicher geschrieben wurden, statt eventuell noch in temporären Registern oder Caches zu stehen. Die Argumente `CLK_LOCAL_MEM_FENCE` und `CLK_GLOBAL_MEM_FENCE` legen dabei fest, ob lokale oder globale Variablen (oder beide) berücksichtigt werden sollen.

Atomare Speicherzugriffe lassen sich mit Funktionen wie `atomic_add` und `atomic_inc` durchführen. Mit Hilfe diese Funktionen können auch speziellere Synchronisationstechniken wie Spinlocks umgesetzt werden, beispielsweise stellt

```
while(atomic_inc(&lock) > 0)
  atomic_dec(&lock);

/* ... kritisch ... */

atomic_dec(&lock);
```

mit Hilfe einer Variablen `lock` im lokalen Speicher sicher, dass jederzeit nur ein Thread der Work-Group den kritischen Bereich ausführt.

Der lokale Speicher kann auch als durch den Programmierer explizit gesteuerter Cache für Daten aus dem Grafikspeicher dienen, da er in der Regel deutlich schneller arbeitet. Um Daten möglichst schnell zwischen dem lokalen und dem globalen Speicher austauschen zu können, bietet OpenCL C die Funktion `async_work_group_copy`, mit der alle Threads einer Work-Group gemeinsam einen Speicherbereich kopieren. Die Funktion muss von allen Threads mit denselben Parametern aufgerufen werden und gibt eine Variable des Typs `event_t` zurück, mit der auf den Abschluss des Kopiervorgangs gewartet werden kann. Dieser Mechanismus ist nötig, falls das Device den Kopiervorgang nebenläufig ausführen kann und der Funktionsaufruf deshalb zurückkehrt, obwohl der Kopiervorgang noch nicht abgeschlossen wurde. Wir sollten deshalb mit der Funktion

`wait_group_events` darauf warten, dass die Daten ihr Ziel erreicht haben, bevor wir uns daran machen, sie zu verarbeiten.

Index