Georg-Simon-Ohm Fachhochschule
Parallelrechnen
Frau Prof. Dr. Stry, Herr Prof. Dr. Eck

Dieter Käppel

Modell und Analyse von parallelen Algorithmen für Matrizenoperationen

Abstrakt

Dieses Dokument ist angelehnt an [Akl89] und beschreibt das Modell und die Analyse von parallelen Algorithmen für Matrizenoperationen. Abgehandelt werden die Probleme, welche bei der Parallelisierung allgemein und speziell bei Matrizenoperationen auftreten. Der Autor beschreibt parallele Algorithmen für die gängigsten Matrizenoperationen. Unter der Annahme angepasster Netzwerktopologien wird die Komplexität der vorgestellten Algorithmen in Bezug auf Laufzeit und Aufwand betrachtet.

Motivation

Zur Vorlesung Parallelrechnen von Frau Prof. Dr. Stry und Herrn Prof. Dr. Eck soll im Rahmen des Informatikstudiums des Autors ein ausgewähltes Thema näher untersucht werden. Ergebnis soll ein 45minütiger Vortrag sowie ein mindestens zehn Seiten umfassendes wissenschaftliches Papier sein. Dabei soll zum einen dem Leser die Methoden der Parallelisierung nähergebracht werden, indem Algorithmen für Matrizen beschrieben werden. Zum anderen sollen Anwendungszwecke aufgezeigt werden, welche durch den Einsatz von Parallelrechnern Beschleunigung erfahren können.

Inhalt

Abbildungen

1 Netztopologie

Die meisten Matrix-Operationen sind von polynomialer Laufzeit und trivial parallelisierbar. Jede der Operationen für ein Element der Matrix ist von gleicher Art und kann unabhängig von den anderen Operationen ausgeführt werden. Worin sich schließlich die Parallelisierungen unterscheiden ist zumeist die Netzwerktopologie, welche für die Kommunikationswege und damit dem Gesamtaufwand verantwortlich ist. Eine sehr naheliegende Topologie ist das Gitter, bei der jedes Matrix-Element einem Prozessor zugeordnet wird jedes Element (jeder Prozessor) mit jedem Nachbarelement der Matrix verbunden wird. Die Dimensionalität des Netzwerks ist in diesem Fall vier. Im Kapitel 2 wird die Auswirkung der Topologie auf das Laufzeitverhalten anhand einfacher praktischer Beispiele erklärt.

Offensichtlich kann diese Topologie für Matrix-Operationen, bei denen – im Sinn der Gitter-Topologie – weit entfernten Elemente verknüpft werden müssen, nicht optimal sein. Es werden deshalb in den entsprechenden Kapiteln weitere Topologien vorgestellt, welche spezielle Probleme optimieren, wohingegen in diesem Kapitel allgemeingültige Aussagen über die Netzwerktopologien bei Matrixoperationen getroffen werden sollen. Diese topologischen Eigenschaften treffen auch auf andere Algorithmen zu, welche allgemein mit Arrays fester Größe arbeiten.

Eine sehr allgemeine Netzwerktopologie ist der sogenannte Hyperwürfel, häufig auch als Hypercube bezeichnet. Fordert man, dass die Verbindungslänge, also die Anzahl der Prozessoren zwischen jeweils zwei Kommunikationspunkten, minimal sein soll, sowie die Anzahl der Verbindungen zwischen den Prozessoren minimal sein soll, so erhält man als Kompromiss zwischen diesen widersprüchlichen Forderungen den Graycode. Eine Lösung des Problems besteht darin, z.B. 8 Prozessoren an den Ecken eines Würfels in einen binären Koordinatensystem anzuordnen. Jede Ecke des Würfels kann durch Entlangbewegen an maximal drei Kanten erreicht werden.

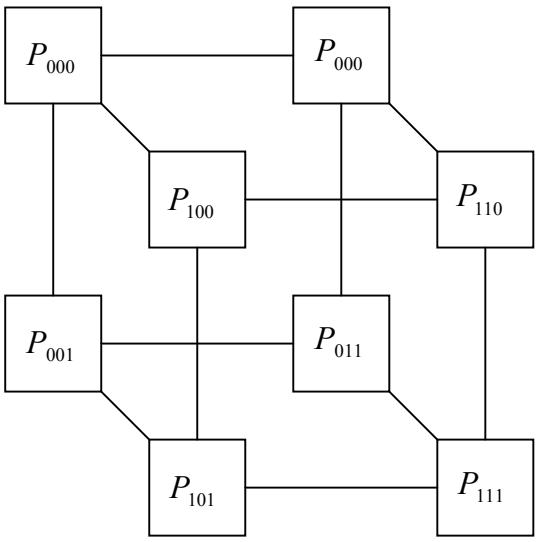

Abbildung 1 - Hyperwürfel

Der Graycode entsteht nun durch ein Verbindungsschema, indem man die n verfügbaren Prozessoren mit den binären Zahlen nummeriert und jeweils die Prozessoren, welche sich um genau ein Bit in ihrer Nummer unterscheiden, miteinander verbindet. Das entspricht dem gleichen Schema wie beim Würfel, nur die Dimensionszahl wurde weiter erhöht und man spricht von Hyperwürfeln, im Englischen von Hypercubes. Da n Prozessoren mit $d = \mathrm{ld}\, n$

Bits nummeriert werden können, gibt es auch d Verbindungen zu jedem Prozessor, denn die Nummern sollen sich genau in einem Bit unterscheiden. Die Weglänge zwischen Prozessor P_1 und P_2 entspricht dann $l \leq \mathrm{ld}\, n$, denn die d-stelligen Nummern können sich höchstens in d Bits unterscheiden. Ordnet man die Nummer des Prozessors zufällig dem Matrixelement zu, so erhält man im Mittel einen Aufwand von $O\left(\frac{1}{2}\mathrm{ld}\, n\right)$, denn zufällig ausgewählte Nummern unterscheiden sich im Mittel in der Hälfte ihrer Bits.

Man kann den Graycode insofern optimieren, dass man je nach Algorithmus die Nummern der Prozessoren so auswählt, dass der Aufwand noch weiter sinkt.

1.1 Speicherzugriff

Abgesehen vom Zugriff auf den Speicher andere Prozessoren im System tritt noch ein elementares Problem, insbesondere beim Rechnen mit Matrizen auf. Sobald die Matrizen der einzelnen Prozessoren größer als deren Cache werden, können Probleme mit der Effizienz auftreten. Vergleicht man folgende Anweisungsblöcke, so wird man zu dem Ergebnis kommen, dass der erste Block auf einem Pentium Pro und einer Größe von size=1000 etwa 7 mal weniger Ausführungszeit benötigt:

```
for (int i=0;i<size;++i) {
        for (int j=0;j<size;++j) {
                data[i][j]=x;
        }
}

for (int i=0;i<size;++i) {
        for (int j=0;j<size;++j) {
                data[j][i]=x;
        }
}
```

Der Unterschied resultiert aus dem ineffizienten Speicherzugriff. Dieses Problem ist natürlich insbesondere bei der Matrix-Multiplikation lästig. Während eine Matrix bei der Multiplikation zeilenweise – also effizient – gelesen werden kann, so muss die andere Matrix dabei spaltenweise gelesen werden. Eine Lösung dafür ist die Verkleinerung der Matrizen unter die Größe des Cache des Prozessors.

1.2 Hierarchische Multiplikation

Der global arbeitende Multiplikations-Algorithmus für Matrizen ist wegen der Überschreitung der Cache-Grenzen nicht optimal, weder auf seriellen noch auf parallelen Rechnern. Eine effizientere Alternative ist deswegen die Hierarchische Multiplikation.

Gegeben sei die Matrix-Matrix-Multiplikation $C = AB$ mit $A, B, C \in R^{n \times n}$, weiter die Hilfsmatrizen $A_1, A_2, A_3, A_4 \in R^{\frac{n}{2} \times \frac{n}{2}}$ und $B_1, B_2, B_3, B_4 \in R^{\frac{n}{2} \times \frac{n}{2}}$, sodass $A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}$ und $B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$. Die Matrizen A und B sind Matrizen aus Matrizen. Die Multiplikation kann nun geschrieben werden durch $C = \begin{pmatrix} A_1B_1 + A_2B_3 & A_1B_2 + A_2B_4 \\ A_3B_1 + A_4B_3 & A_3B_2 + A_4B_4 \end{pmatrix}$. Entsprechend erhält

man z.B. für $c_{1,1} = \sum_{i=1}^{\frac{n}{2}} a_{1,i}b_{i,1} + \sum_{i=\frac{n}{2}+1}^{n} a_{1,i}b_{i,1} = \sum_{i=1}^{n} a_{1,i}b_{i,1}$ und kann so allgemein beweisen, dass dies für jedes Element $c_{i,j}$ des Produkts gilt.

Falls eine Unterteilung nicht ausreichend ist, um die Cache-Größe der Prozessoren zu unterschreiten, so kann man dieses Verfahren hierarchisch anwenden auf die Teilprodukte.

1.2.1 Verfahren von Strassen

Zerlegt man die Multiplikation in Teilmultiplikationen, so kann man dabei gleich noch eine Einsparung vornehmen. *Strassen* hat entdeckt, dass zur Bildung des Produkts $C = AB$ aus den Teilprodukten A_1, A_2, A_3, A_4 und B_1, B_2, B_3, B_4 keine acht Multiplikationen notwendig sind, sondern dass sieben ebenfalls ausreichend sind. Strassen definierte dazu die temporären Produkte $P_1 = (A_1 + A_4)(B_1 + B_4)$, $P_2 = (A_3 + A_4)B_1$, $P_3 = A_1(B_2 + B_4)$, $P_4 = A_4(B_3 + B_1)$, $P_5 = (A_1 + A_2)B_4$, $P_6 = (A_3 - A_1)(B_1 + B_2)$, $P_7 = (A_2 - A_4)(B_3 + B_4)$ und daraus das entgültige Produkt $C = \begin{pmatrix} P_1 + P_4 - P_5 + P_7 & P_3 + P_5 \\ P_2 + P_4 & P_1 + P_3 - P_2 + P_6 \end{pmatrix}$, wie man durch ausmultiplizieren und vereinfachen bestätigt.

2 Matrix-Operationen

2.1 Transposition

Bei einer gegebenen Matrix A der Größe n deren Transponierte A^T zu finden, ist eine der einfachsten Matrix-Operationen. Da aber bereits diese Operation bei sequentiellen Algorithmen $O(n^2)$ Zeitaufwand benötigt, kann eine Parallelisierung Vorteile bringen. Prinzipiell ist es bei der Aufteilung der Elemente einer Matrix in einem parallelen System naheliegend, jedem Prozessor ein Element der Matrix zuzuordnen. Verbindet man jeden Prozessor des Matrixelements $a_{i,j}$ mit den Elementen $a_{i-1,j}$, $a_{i+1,j}$, $a_{i,j-1}$ und $a_{i,j+1}$, so erhält man folgende Topologie:

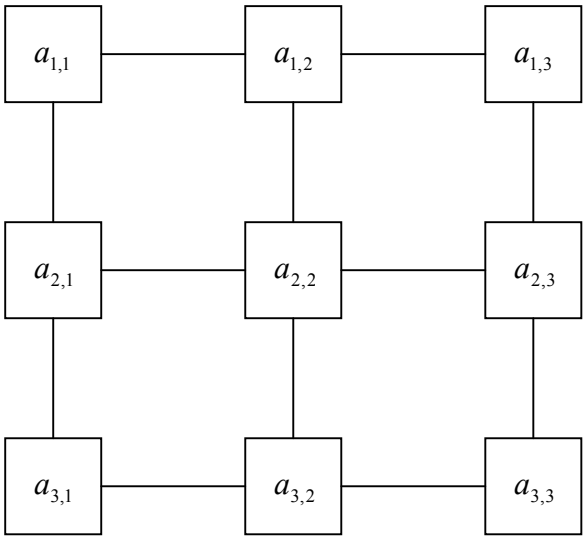

Abbildung 2 - Gittertopologie

Man erkennt sehr gut, dass die Elemente auf der Hauptdiagonalen gar nicht bewegt werden zu brauchen, während die Elemente $a_{1,n}$ und $a_{3,1}$ im äußersten Fall $2n-2$ mal weitergereicht werden müssen. Folgende Illustration verdeutlicht das Vorgehen:

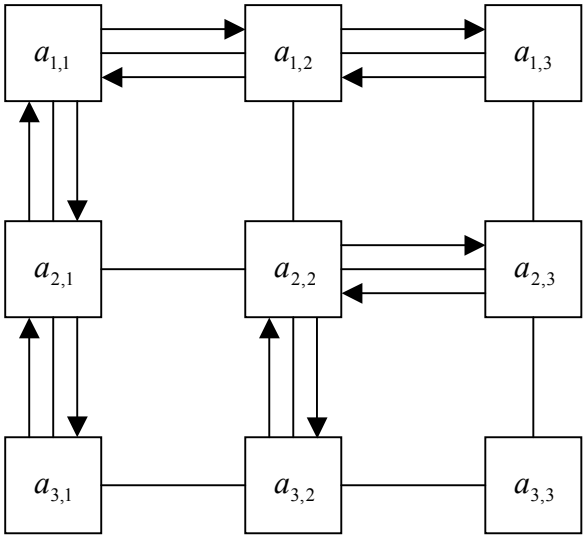

Abbildung 3 - Transposition im Gitter

Weil alle diese Weitergaben parallel ausgeführt werden können, erreicht man damit die Zeitkomplexität $O(n)$. Durch die Verwendung von n^2 Prozessoren, benötigt man mit dieser Topologie einen Gesamtaufwand von $O(n^3)$, was nicht optimal ist, wie man an den sequentiellen Algorithmen sehen kann.

Eine bessere Anordnung der Verbindung zwischen den Prozessoren, bzw. eine andere Verteilung der Matrixelemente im Gitter kann das Problem verbessern. Man nummeriere die Prozessoren von P_0 bis P_{n-1}. Dann ordne man jedem Prozessor P_k das Matrixelement $k = 2^q(i-1)+(j-1)$ zu, wobei die Größe der Matrix $n = 2^q$ entsprechen muss. Die Prozessoren verbinde man durch Darstellung in Binärschreibweise. Für $q = 2$ sind z.B. Zyklen (P_{0000}), (P_{0001}, P_{0010}, P_{0100}, P_{1000}), (P_{0101}, P_{1010}). Jeder Prozessor P_k sendet sein Element, welches er gerade speichert an den Prozessor $P_{2k \bmod (2^{2q}-1)}$. Weil q konstant iteriert wird, erreicht jedes Element in logarithmischer Zeit das Ziel. Die Komplexität des Algorithmus ist also $O(\log n)$. Da weiterhin n^2 Prozessoren Verwendung finden, ist die Gesamtkomplexität bestimmt durch $O(n^2 \log n)$. Dies ist zwar schneller als bei einfachen Gitterverbindungen, jedoch immer noch nicht optimal.

Ein optimaler Algorithmus ist z.B. konstruierbar, wenn sich die Matrix in einem gemeinsamen Speicher befindet und die Anzahl der verwendeten Prozessoren $\sum_{i=1}^{n-1} i = \frac{n^2-n}{2}$ beträgt. Die Zahl errechnet sich aus der Summe der Vertauschungen in jeder Zeile. Die Elemente auf der Hauptdiagonale müssen dabei nicht angerührt werden. Jeder Prozessor vertauscht dabei $a_{i,j}$ mit $a_{j,i}$ für $i \neq j$. Die Komplexität beträgt $O(1)$ und der Gesamtaufwand $O(n^2)$ wegen der quadratisch wachsenden Zahl von Prozessoren. Dies ist optimal.

2.2 Multiplikation

Die Multiplikation erweist sich bereits komplizierter als die Transposition, weil es sich nicht mehr um 1-stellige Operationen, sondern um n-stellige Operationen der Elemente handelt.

Eine gegebene Matrix-Vektor-Multiplikation $y = Ax$ lässt sich in die elementaren Operationen $y_j = \sum_{i=1}^{n} a_{ji} x_i$ zerlegen. Die temporären Produkte $t_{ij} = a_{ji} x_i$ können vollständig parallel berechnet werden, weil keine Datenabhängigkeiten existieren. Der Aufwand ist dabei $O(1)$. Offen ist jedoch noch die Kommunikation der Prozessoren bzw. die Topologie der Verbindungen. Bei gitterförmiger Anordnung der Prozessoren, wenn jedem Prozessor $p_{i,j}$ das Element $a_{i,j}$ der Matrix A und das Element x_i zugeordnet ist (jedes Vektor-Element x_i ist n-fach gespeichert), können in einem Schritt alle Multiplikationen ausgeführt werden und die Summation kann in n Schritten durchlaufen.

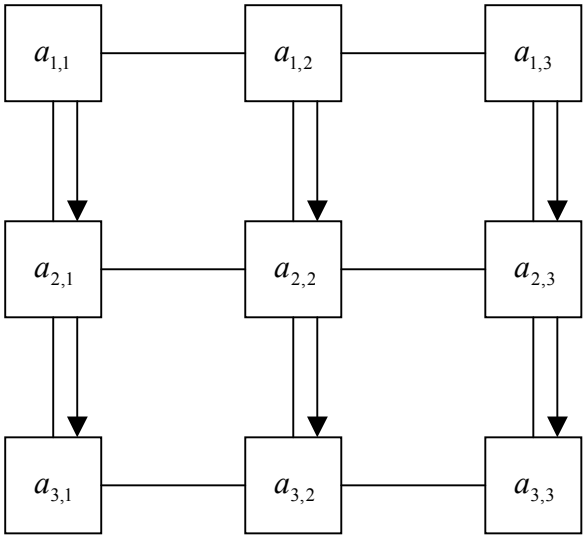

Abbildung 4 - Summierung im Gitter

Als problematisch erweisen sich die Summen $y_j = \sum_{i=1}^{n} t_{ij}$. Eine Lösung bei assoziativen Operationen wie der Addition ist, die Operationen rekursiv zu definieren mit $\sum_{i=1}^{m} t_{ij} := \sum_{i=1}^{m/2} t_{ij} + \sum_{\lfloor m/2 \rfloor + 1}^{m} t_{ij}$.

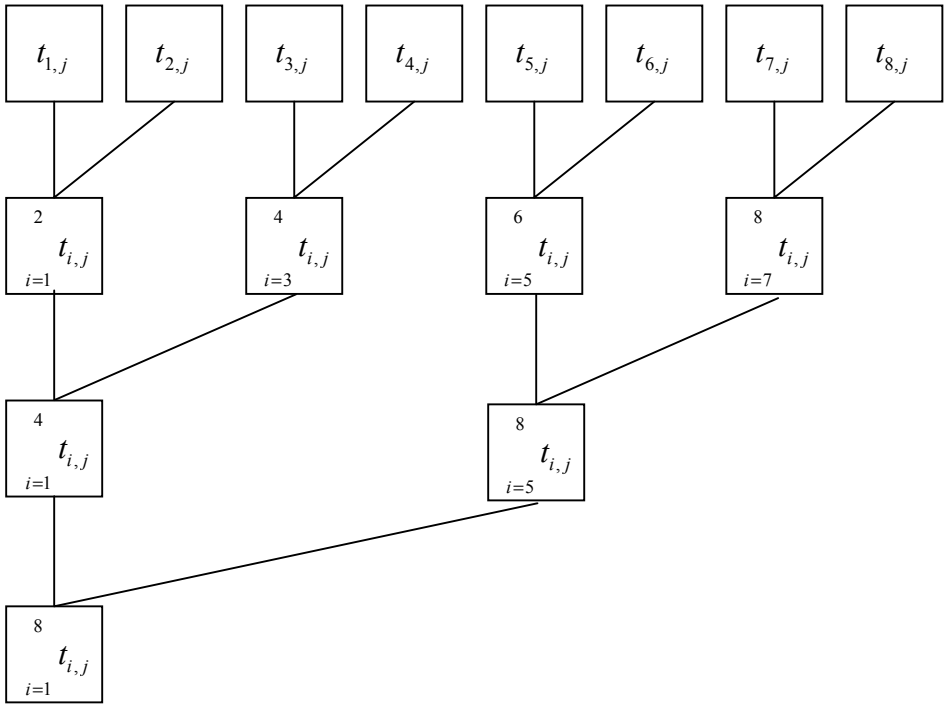

Abbildung 5 - Subsummierung

Wie man erkennt, kann das Ergebnis in $O(\mathrm{ld}\, n)$ Schritten errechnet werden, dabei muss aber die Verbindungszahl der Prozessoren auf bis zu $\mathrm{ld}\, n$ erhöht werden, damit der Geschwindigkeitsgewinn der Subsummation nicht durch die Durchlaufzeit der Prozessoren verloren geht.

Die Matrix-Matrix-Multiplikation wird analog ausgeführt, nur dass nicht nur mir einer 1-spaltigen Matrix, sondern mit einer n-spaltigen Matrix multipliziert wird.

2.3 Lineare Gleichungssysteme

Eine Lösungsmöglichkeit für lineare Gleichungssysteme ist der Gauß-Jordan-Algorithmus. Um den algorithmischen Ablauf in Erinnerung zu rufen, es wird zuerst die erste Zeile der zu lösenden Gleichungs-Matrix A durch das Element $a_{1,1}$ dividiert. Die Division jedes Elements kann parallel erfolgen. Mit dieser Zeile werden alle anderen Zeilen "ausgekämmt", indem das Element $a_{1,j}$ der Zeile $a_{i,1}$ mal von jedem Element $a_{i,j}$ subtrahiert wird. Die Multiplikation und Subtraktion kann jeweils parallel erfolgen. Da der Vorgang insgesamt für jedes Diagonalelement einmal vorgenommen werde muss, ist die Komplexität der parallelisierten Operationen $O(n)$.

Da die Kommunikation des Gauß-Jordan-Algorithmus schlecht parallelisierbar ist und der Algorithmus empfindlich auf schlechte Kondition reagiert, ist es häufig von Vorteil, ein iteratives Verfahren zur Berechnung der Inversen heranzuziehen. Z.B. ist es möglich, mit Hilfe des Newton-Iterationsverfahrens ein solches Verfahren zu konstruieren. Möchte man die Matrix $x_n = M^{-1}$ iterieren, so ist ein gut konvergierender Ansatz $f(x) = \frac{1}{x} - M$. Man erhält da-

durch die Iterationsanweisung $x_{n+1} := x_n - \frac{\frac{1}{x_n} - M}{\frac{-1}{x_n^2}} = 2x_n - x_n M x_n$ eine quadratische Konvergenz zu M^{-1}. Einen brauchbaren Startwert erhält man durch die Betrachtung der Matrixnorm von M, denn es gilt $\left\|M^{-1}\right\| = \left\|M\right\|^{-1}$. Denkt man zusätzlich daran, dass für orthogonale Matrizen gilt $M^{-1} = M^T$, so ergibt sich als Startwert $x_0 = \frac{1}{\left\|M\right\|} M^T$. Als Norm empfiehlt sich die 1-Norm der Matrix.

Die Parallelisierung des Iterationsverfahrens geschieht über die bisher kennen gelernten Methoden, insbesondere die rekursive Subsummierung und die schnelle Multiplikation.

3 Anwendungen

3.1 Diskrete Fourier Transformation

Es kann gezeigt werden, dass die DFT (Diskrete Fourier Transformation) durch die Multiplikation einer Matrix (Fourier Koeffizienten Matrix) mit einem Vektor (Zeitreihe) darstellbar ist. Bei einem gegebenen Vektor $x = \begin{pmatrix} x_1 \\ \vdots \\ x_t \end{pmatrix}$ der Zeitreihe, beschreibt die Fourier Transformation $y_\omega = \sum_{\tau=1}^{t} x_\tau e^{2\pi j\omega\tau} = \sum_{\tau=1}^{t} x_\tau \left(\cos(2\pi\omega\tau) + j\sin(2\pi\omega\tau)\right)$ den (diskreten) Spektralvektor $y = \begin{pmatrix} y_1 \\ \vdots \\ y_\Omega \end{pmatrix}$. Dem Informationstheorem von Shannon folgend, ist es nur interessant, die untere Hälfte der Frequenzanteile von y zu betrachten. Praktikabel ist es, die reellen Frequenzanteile in der oberen Hälfte und die Imaginären Frequenzanteile in der unteren Hälfte des Vektors zu abzubilden.

Man die DFT nun durch eine Matrizenoperation $y = Fx$ darstellen, wobei $F = \begin{pmatrix} f_{1,1} & \cdots & f_{1,t} \\ \vdots & \ddots & \vdots \\ f_{t,1} & \cdots & f_{t,t} \end{pmatrix}$ die Fourier Koeffizienten enthält, wegen $y_\omega = \sum_{\tau=1}^{t} x_\tau f_{\omega,\tau}$ kann man die Koeffizienten direkt angeben durch $f_{\omega,\tau} = \cos(2\pi\omega\tau)$ für $1 \le \omega < \frac{\Omega}{2}$ bzw. $f_{\omega,\tau} = \sin\left(2\pi\left(\omega - \frac{\Omega}{2}\right)\tau\right)$ für $\frac{\Omega}{2} \le \omega < \Omega$. Analog dazu entwickelt man die Inverse DFT.

3.2 PBLAS (Parallel Basic Linear Algebra Subroutines)

Eine weit verbreitete Implementierung von Matrizen-Operationen in der Programmiersprache C ist die BLAS (Basic Linear Algebra Subroutines), eine Bibliothek, bei deren Entwicklung man die häufig benötigten Operationen aus der Linearen Algebra effizient implementieren wollte. Eine Variante dieser Bibliothek ist die PBLAS, welche für parallelen Ablauf der Routinen vorbereitet ist.

Schlusswort

Man sieht, dass Matrizenoperationen gut parallelisierbar sind und die Komplexität des Resultats besonders von der Topologie des Netzwerks abhängt. Man erreicht mit den Algorithmen zwar das theoretische Minimum an Laufzeit, häufig muss man diesen Vorteil teuer erkaufen, sodass der Gesamtaufwand zum Teil weit über das Optimum ansteigt.

Quellenverweise

[Akl89] Selim G. Akl *Analysis and Design of Parallel Algorithms*, Prentice-Hall Inc. USA, 1989

[Bla01] http://www.netlib.org/blas/, *BLAS Basic Linear Algebra Subroutines*, Netlib Repository, 2001

[Brä93] Thomas Bräunl, *Parallele Programmierung: eine Einführung*, Vieweg 1993

[Cha96] A. Chalmers, J. Tidmus, *Practical Parallel Processing*, Thomson Computer Press, 1996

[Kum93] Vipin Kumar et al., *Introduction to Parallel Computing*, Benjamin/Cummings 1993

[Lei92] F. T. Leighton, *Parallel Algorithms and Architectures*, Morgan Kaufmann Publishers, 1992

[Les93] Bruce P. Lester, *The Art of Parallel Programming*, Prentice-Hall, 1993

[Meh88] Mehrnoosh Mary Eshaghian, *Parallel Computing With Optical Interconnect*, Dissertation to Doctor of Philosophy, Faculty of the Graduate School University of Southern Calafornie, 1988

[Rus01] http://rustam.uwp.edu/, *Programming in C, C++, PVM for applications in Physics and Mathematics, graphics and simulation*, University of Wisconsin – Parkside Physics, 2001.

[Ung97] Theo Ungerer, *Parallelrechner und parallele Programmierung*, (Spektrum-Hochschultaschenbuch), Spektrum Akademischer Verlag GmbH, 1997