Handout - Fast Fourier Transform

Markus Pawellek

03.Februar 2016

1 Mathematische Grundlagen

Definition: (periodische Funktionen)

Sei $T \in (0, \infty)$. Eine Abbildung $f : \mathbb{R} \longrightarrow \mathbb{C}$ heißt dann T-periodisch, wenn für alle $x \in \mathbb{R}$ gilt

$$f(x) = f(x + T)$$

In diesem Falle nennt man T auch die Periode von f und $1/T$ auch die Frequenz von f.

Für die folgenden Aussagen soll $T \in (0, \infty)$ immer die Periode einer Funktion beschreiben.

1.1 Fouriertransformation periodischer Funktionen

Definition: (Fouriertransformation)

Für eine T-periodische Funktion $f : \mathbb{R} \longrightarrow \mathbb{C}$, welche stückweise stetig differenzierbar ist, kann man die Fouriertransformation $\mathcal{F}$ und ihre Inverse definieren durch

$$\forall k \in \mathbb{Z}: \quad \mathcal{F}f(k) := \frac{1}{T} \int_0^T f(x) \exp\left(-\frac{2\pi i}{T} kx\right) \, \mathrm{d}x$$

$$\forall x \in \mathbb{R}: \quad f(x) := \sum_{k \in \mathbb{Z}} \mathcal{F}f(k) \exp\left(\frac{2\pi i}{T} kx\right)$$

Die Fouriertransformation spaltet damit gerade eine periodische Funktion f in sogenannte Frequenzanteile auf. Ein solcher Frequenzanteil beschreibt dann, zu welchem Anteil die harmonische Schwingung $\exp(i\omega\cdot)$ der angegebenen Frequenz $\omega/2\pi$ in der Funktion vorkommt.

Diese Transformation lässt sich auch noch für andere Funktionen einführen. Für eine genauere Betrachtung der dahinter stehenden Mathematik sei hier auf die Quellen [5] und [6] verwiesen.

1.2 Diskrete Fouriertransformation

Bekannterweise lassen sich in einem Computer Reihen fast immer nur approximieren, da die Addition unendlich vieler Glieder ungleich Null in endlicher Zeit auf endlichem Speicherplatz nicht durchführbar ist. Aus diesem Grund führt man die diskrete Fouriertransformation ein, welche in gewisser Weise eine Näherung der Vorhergehenden ist.

Es sei nun immer $n \in \mathbb{N}$ und $\mathrm{N}_n := \{0, \dots, n-1\}$. Misst man in der Realität nun einen Zusammenhang f, so lässt sich diese nur näherungsweise durch Stützstellen $x_0, \dots, x_{n-1} \in [0, T)$ mit den zugehörigen Funktionswerten $f(x_0), \dots, f(x_{n-1})$ beschreiben. In diesen und folgenden Betrachtungen sollen alle Stützstellen als äquidistant angenommen werden. Die Funktion wurde diskretisiert. Man definiert nun

$$g : \mathrm{N}_n \longrightarrow \mathbb{C}, \qquad g(k) := f(x_k)$$

g stellt damit ein Element des $\mathbb{C}^n$ dar. Es lässt sich also jede diskretisierte Funktion als komplexer n-dimensionaler Vektor darstellen. Weiterhin sei für alle $x, y \in \mathbb{C}^n$

$$\langle \cdot, \cdot \rangle : \mathbb{C}^n \times \mathbb{C}^n \longrightarrow \mathbb{C}, \qquad \langle x, y \rangle := \frac{1}{n} \sum_{j=0}^{n-1} \overline{x(i)} y(i)$$

Die Abbildung $\langle \cdot, \cdot \rangle$ definiert gerade das Standardskalarprodukt des $\mathbb{C}^n$. Das Tupel $(\mathbb{C}^n, \langle \cdot, \cdot \rangle)$ muss dann ein Hilbertraum sein. Dies ermöglicht es eine Orthonormalbasis zu finden.

PROPOSITION: (Orthonormalbasis)

Die folgende Menge D bildet eine Orthonormalbasis des $(\mathbb{C}^n, \langle \cdot, \cdot \rangle)$.

$$D := \left\{ \omega_k : \mathrm{N}_n \longrightarrow \mathbb{C} \;\middle|\; k \in \mathrm{N}_n,\ \forall x \in \mathrm{N}_n :\ \omega_k(x) = \exp\left(\frac{2\pi i}{n} kx \right) \right\}$$

Nach der Parsevalschen Gleichung lässt sich nun das Element g durch eine Linearkombination der Basisvektoren ausdrücken.

$$\forall x \in \mathrm{N}_n :\ g(x) = \sum_{k=0}^{n-1} \langle \omega_k, g \rangle \, \omega_k(x)$$

Die $\langle \omega_k, g \rangle$ stellen dabei die Koordinaten von g bezüglich D dar. Diese Koordinaten nennt man nun die diskrete Fouriertransformation oder auch DFT von g. Die Parsevalsche Gleichung liefert auch gleich die Inverse.

DEFINITION: (diskrete Fouriertransformation)

Sei $g : \mathrm{N}_n \longrightarrow \mathbb{C}$. Dann ist die diskrete Fouriertransformation $\hat{g}$ von g und deren Inverse definiert durch

$$\forall k \in \mathrm{N}_n :\ \hat{g}(k) := \frac{1}{n} \sum_{x=0}^{n-1} g(x) \exp\left(-\frac{2\pi i}{n} kx \right)$$

$$\forall x \in \mathrm{N}_n :\ g(x) := \sum_{k=0}^{n-1} \hat{g}(k) \exp\left(\frac{2\pi i}{n} xk \right)$$

Beispiel:
Es sei nun $n = 5$ und

$$\forall x \in \mathrm{N}_5 : \; g(x) := x$$

Tabelle 1 zeigt die Näherungen der berechneten diskreten Fourier-Koeffizienten von g.

x	$\sim \hat{g}(x)$
0	10
1	$-0.5 + 0.688191\,i$
2	$-0.5 + 0.162460\,i$
3	$-0.5 - 0.162460\,i$
4	$-0.5 - 0.688191\,i$

Tabelle 1: Fourierkoeffizienten der Beispielfunktion g

Die Gleichung der inversen DFT motiviert dazu, die Funktion g nicht nur für $x \in \mathrm{N}_n$ auszuwerten, sondern auch für $x \in \mathbb{R}$. Dafür definiert man das folgende trigonometrische Polynom

$$p_n : \mathbb{R} \longrightarrow \mathbb{C}, \qquad p_n(x) = \sum_{k=-\lfloor n/2 \rfloor}^{\lceil n/2-1 \rceil} \hat{g}(k) \exp\left(\frac{2\pi i}{n} k x\right)$$

Dieses trigonometrische Polynom interpoliert die Stützpunkte der Funktion g. Abbildung 1 stellt p_5 und die Stützpunkte dar.

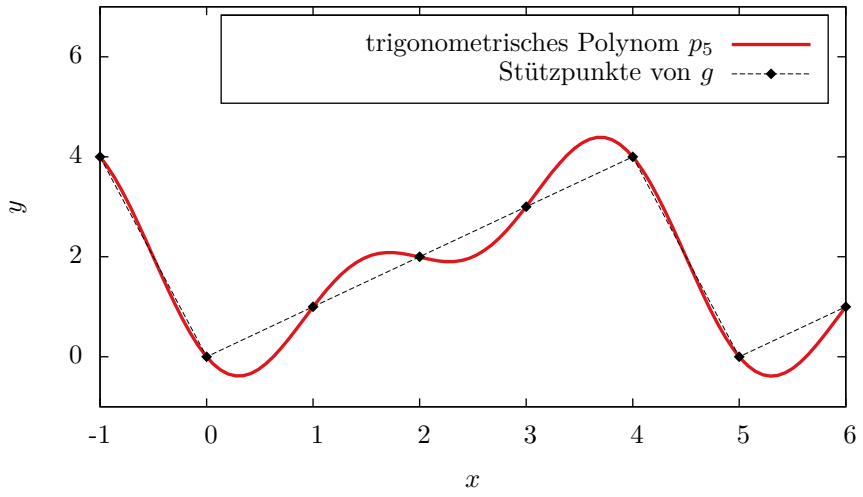

Abbildung 1: trigonometrisches Polynom und Stützpunkte der Beispielfunktion g

2 Serieller Algorithmus

2.1 Idee der Fast Fourier Transform

Im Folgenden bezeichne $W(n)$ die Anzahl der Operationen (hier $+, \cdot$), die ein gegebener Algorithmus in Abhängigkeit der Eingabegröße n ausführt.

Für einen naiven Algorithmus, der nach obiger Formel die DFT von g berechnet, gilt nun

$$W(n) = \underbrace{n(n+1)}_{\text{Multiplikation}} + \underbrace{n(n-1)}_{\text{Addition}} = 2n^2 \in \Omega\left(n^2\right)$$

Dabei wurde die Berechnung der Faktoren $e^{-i\xi}$, die auch Twiddle-Faktoren genannt werden, nicht beachtet. Diese können durch geeignete Tabellenwerte, die dem Programm vorab zur Verfügung gestellt werden, direkt abgelesen werden. Die Berechnung muss nun für alle n Koeffizienten durchgeführt werden.

Um diese Laufzeitkomplexität zu verbessern, teilt man die Berechnung der DFT in gerade und ungerade Summanden ein. Es sei nun $n = 2m$ für ein $m \in \mathbb{N}$ und $\mathcal{F}_n$ bezeichne die DFT auf dem Raum $\mathbb{C}^n$. Für $k \in \mathrm{N}_n$ folgt

$$\begin{aligned}
\mathcal{F}_n g(k) &= \frac{1}{n} \sum_{x=0}^{n-1} g(x) \exp\left(-\frac{2\pi i}{n} kx\right) \\
&= \frac{1}{n} \left[\sum_{x=0}^{m-1} f(2x) e^{-\frac{2\pi i}{n} 2kx} + \sum_{x=0}^{m-1} f(2x+1) e^{-\frac{2\pi i}{n} k(2x+1)} \right] \\
&= \frac{1}{2} \left[\frac{1}{m} \sum_{x=0}^{m-1} g(2x) e^{-\frac{2\pi i}{m} kx} + e^{-\frac{\pi i}{m} k} \frac{1}{m} \sum_{x=0}^{m-1} g(2x+1) e^{-\frac{2\pi i}{m} kj} \right]
\end{aligned}$$

Für $k < m$ können diese beiden Summen durch diskrete Fouriertransformationen für m Punkte ersetzt werden.

$$\begin{aligned}
g_0 : \mathrm{N}_m \longrightarrow \mathbb{C}, \qquad g_0(x) &:= g(2x) \\
g_1 : \mathrm{N}_m \longrightarrow \mathbb{C}, \qquad g_1(x) &:= g(2x+1)
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}_n g(k) \qquad &= \frac{1}{2} \left(\mathcal{F}_m g_0(k) + e^{-\frac{\pi i}{m} k} \ \mathcal{F}_m g_1(k) \right) \\
\mathcal{F}_n g(k+m) &= \frac{1}{2} \left(\mathcal{F}_m g_0(k) - e^{-\frac{\pi i}{m} k} \ \mathcal{F}_m g_1(k) \right)
\end{aligned}$$

Das Ausführen der Fouriertransformation für n Punkte wird also auf die Ausführung zweier Fouriertransformationen für $n/2$ Punkte zurückgeführt. Man möchte nun durch ein rekursives Wiederholen dieser Prozedur die Laufzeitkomplexität verbessern. Aus diesem Grund wird nun hier und im Folgenden $n = 2^p$ für ein $p \in \mathbb{N}$ gesetzt. Dadurch sind exakt p rekursive Schritte mögliche.

PROPOSITION: (Zeitkomplexität)

Gilt $W(1) = 0$ *und*

$$\forall k \in \mathrm{N}_m : \quad W(2k) = 2W(k) + 4k$$

so folgt für die explizite Darstellung der Zeitkomplexität

$$W(n) = 2n \log_2 n \in \Omega(n \log_2 n)$$

Auch bei der angegebenen Proposition werden die Twiddle-Faktoren nicht beachtet. Mithilfe dieses Algorithmus erfährt die Berechnung der DFT eine signifikante Beschleunigung. Algorithmen, welche bei der Berechnung der DFT diese Zeitkomplexität aufweisen, werden im allgemeinen Fast Fourier Transform oder auch FFT genannt.

2.2 Rekursiver Algorithmus

Ein erster einfacher FFT-Algorithmus ergibt sich nun direkt aus den oben beschriebenen rekursiven Gleichungen. Um dies in der gegebenen Sprache zu verwirklichen ist eine zusätzliche Prozedur im Programm nötig.

Listing: rekursiver FFT-Algorithmus

program Recursive-FFT
declare
  procedure fft(out, in : array$[0..2^q - 1 - 1]$ of complex ; q : integer);
  t : integer ;
  ω, b, c : array$[0..2^p - 1]$ of complex
initially
  {*look-up table for twiddle factors*}
  $\langle\ ;\ t\ :\ 0 \leq t < 2^p\ ::\ \omega[t] = \exp(-\frac{2\pi i t}{n})\ \rangle$;
  $\langle\ ;\ t\ :\ 0 \leq t < 2^p\ ::\ b[t] = g(t)\ \rangle$
assign
  fft(c, b, p)
end

{*extra procedure for recursive algorithm*}
procedure fft(out, in : array$[0..2^q - 1 - 1]$ of complex ; q : integer)
begin
  if $q = 0$ **then**
    $out[0] = in[0]$
  else begin
    fft$(out[0..2^{q-1} - 1]\ ,\ in[0..2..2^q - 2]\ ,\ q - 1)$;
    fft$(out[2^{q-1}..2^q - 1]\ ,\ in[1..2..2^q - 1]\ ,\ q - 1)$;
    $\langle \parallel\ t\ :\ 0 \leq t < 2^{q-1}\ ::$
      $\begin{bmatrix} b[t] \\ b[t + 2^{q-1}] \end{bmatrix} := \frac{1}{2} \begin{bmatrix} 1 & \omega[2^{p-q}t] \\ 1 & -\omega[2^{p-q}t] \end{bmatrix} \begin{bmatrix} b[t] \\ b[t + 2^{q-1}] \end{bmatrix}$
    $\rangle$
  end
end

2.3 Von Bit-Reversal zu Butterfly

Um aus dem oben beschriebenen Algorithmus einen nicht-rekursiven Algorithmus zu erstellen, ist eine genau Analyse nötig. Diese soll anhand eines Beispiels veranschaulicht werden. Es sei dafür $p = 3$ und damit $n = 8$. Das Eingabearray b besitzt also 8 Elemente. In jedem rekursiven Aufruf werden die Werte von b mit geraden und ungeraden Indizes in zwei neue Eingabearrays aufgespalten. Dies wird solange wiederholt, bis die Länge des Eingabearrays 1 beträgt. Am Ende wird dann nur noch auf einem permutierten Array $\tilde{b}$ mit $\tilde{b}[t] := b[\sigma_p(t)]$ gearbeitet. Abbildung 2 veranschaulicht dies. Stellt man nun alle Indizes im Binärformat mit p Stellen dar, so ergibt sich zum Beispiel

$$\begin{aligned}
0 = (000)_2 &\xrightarrow{\sigma_p} 0 = (000)_2\\
1 = (001)_2 &\xrightarrow{\sigma_p} 4 = (100)_2\\
2 = (010)_2 &\xrightarrow{\sigma_p} 2 = (010)_2\\
3 = (011)_2 &\xrightarrow{\sigma_p} 6 = (110)_2\\
4 = (100)_2 &\xrightarrow{\sigma_p} 1 = (001)_2\\
5 = (101)_2 &\xrightarrow{\sigma_p} 5 = (101)_2\\
6 = (110)_2 &\xrightarrow{\sigma_p} 3 = (011)_2\\
7 = (111)_2 &\xrightarrow{\sigma_p} 7 = (111)_2
\end{aligned}$$

Die Binärdarstellungen der Indizes wird also gerade umgekehrt. Dies gilt nicht nur für den Spezialfall $p = 3$, sonder für alle p.

PROPOSITION: (Bit-Reversal Map)

Sei $t \in \mathrm{N}_{2^p}$ mit der Binärdarstellung $(\alpha_{p-1} \dots \alpha_0)_2$. Dann ergibt sich nach obigen Definitionen für die Binärdarstellung von $\sigma_p(t)$

$$(\alpha_0 \dots \alpha_{p-1})_2$$

σ_p wird dann auch als Bit-Reversal Map bezeichnet.

Der zweite Teil des rekursiven Algorithmus lässt sich durch Abbildung 3 veranschaulichen. Nachdem alle Instanzen der Prozedur aufgerufen wurden, muss jetzt das Matrix-Vektor-Produkt berechnet werden. Die einzelnen + in der Abbildung stehen für die Berechnung der ersten Zeile. Die einzelnen − stehen also für die zweite Zeile. Dieses Schema wird auch Schmetterlingsgraph oder Butterfly Graph genannt.

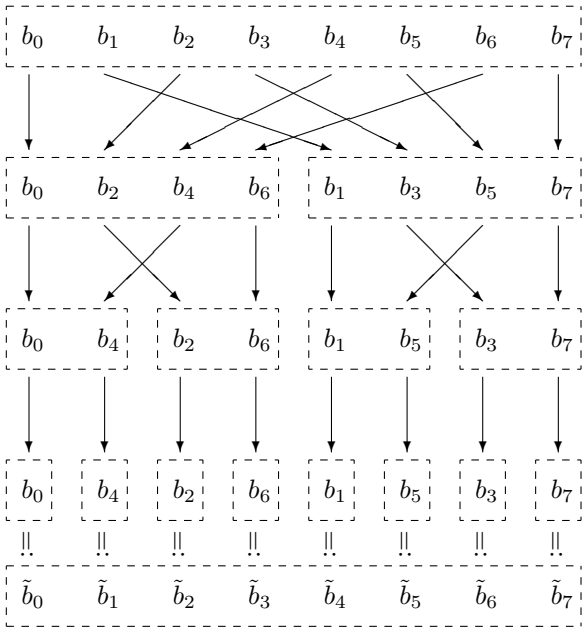

Abbildung 2: Bit-Reversal Map:
Funktionsweise des rekursiven FFT-Algorithmus bis zum Erreichen der Abbruchbedingung für $p = 3$

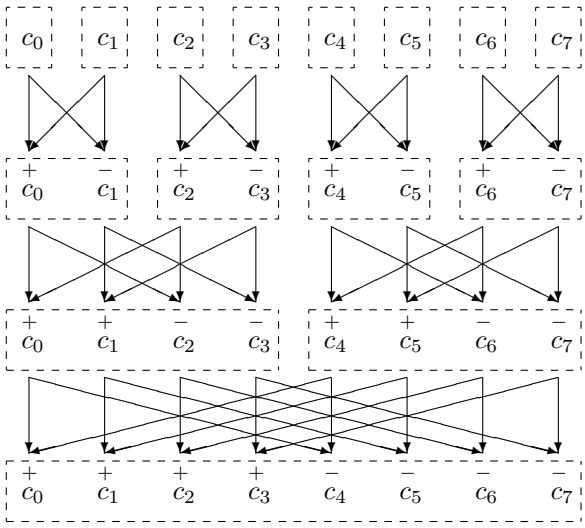

Abbildung 3: Butterfly Graph:
Funktionsweise des rekursiven FFT-Algorithmus nach dem Erreichen der Abbruchbedingung für $p = 3$

2.4 Nicht-Rekursiver Algorithmus

Durch die oben beschriebene Analyse führt nun auf den folgenden nicht-rekursiven Algorithmus.

Listing: nicht-rekursiver FFT-Algorithmus

program Serial-FFT
declare
m, t : integer ;
ω, b : array$[0..2^p - 1]$ of complex
initially
{*look-up table for twiddle factors*}
$\langle\ ;\ t\ :\ 0 \leq t < 2^p\ ::\ \omega[t] = \exp(-\frac{2\pi i t}{n})\ \rangle$;
{*bit-reverse ordering*}
$\langle\ ;\ t\ :\ 0 \leq t < 2^p\ ::\ b[t] = g(\sigma_p(t))\ \rangle$
assign
{*butterfly calculation*}
$\langle\ ;\ m\ :\ 0 \leq m < p\ ::$
$\langle\ \|\ t\ :\ 0 \leq t < 2^p - 1$ **and** $t \wedge 2^m = 0\ ::$

$$\begin{bmatrix} b[t] \\ b[t+2^m] \end{bmatrix} := \frac{1}{2} \begin{bmatrix} 1 & \omega[2^{p-m-1}t \bmod 2^{p-1}] \\ 1 & -\omega[2^{p-m-1}t \bmod 2^{p-1}] \end{bmatrix} \begin{bmatrix} b[t] \\ b[t+2^m] \end{bmatrix}$$

$\rangle$
$\rangle$
end

3 Paralleler Algorithmus

3.1 Verteilung der Daten

Die Länge des Eingabearrays beträgt $n = 2^p$. Will man nun den nicht-rekursiven Algorithmus parallelisieren, so ist es durchaus sinnvoll für die Anzahl der Prozesse ebenfalls eine Potenz von 2 zu verwenden. Es sei also $P = 2^q$ für ein $q \in \mathbb{N}$ mit $q \leq p$ die Anzahl der Prozesse. Um die Arbeit nun möglichst gleichmäßig zu verteilen, soll nun jeder Prozess gerade 2^{p-q} Elemente speichern. Abbildung 4 stellt diese Verteilung für $p = 3$ und $q = 2$ dar.

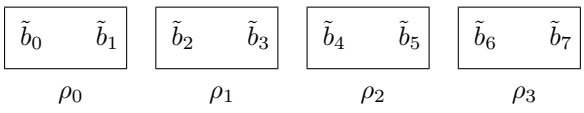

Abbildung 4: Verteilung des Eingabearrays $\tilde{b}$ auf den Prozessen ρ_i für $p = 3, q = 2$

Damit reichen die lokalen Indizes des Arrays $\tilde{b}$ bezüglich eines Prozesses von 0 bis $2^{p-q} - 1$. Ein globaler Index $u \in \mathrm{N}_{2^p}$ bezüglich eines Prozessindex $v \in \mathrm{N}_{2^q}$ und eines lokalen Index $w \in \mathrm{N}_{2^{p-q}}$ ergibt sich dann zu

$$u = 2^{p-q}v + w$$

Bezeichnet man die zugehörigen Binärdarstellungen mit $(u_p \dots u_1)_2, (v_q \dots v_1)_2$ und $(w_{p-q} \dots w_1)_2$ so folgt dann

$$(u_p \dots u_1)_2 = (v_q \dots v_1 w_{p-q} \dots w_1)_2$$

Die ersten q Bits des globalen Index bezeichnen den Prozessindex und die Restlichen den lokalen Index bezüglich dieses Prozesses.

3.2 Kommunikation und Algorithmus

Man wendet jetzt den Butterfly-Algorithmus auf diese Datenverteilung an (siehe Abbildung 5). Es gibt dann Teilschritte, die keine Kommunikation zwischen den Prozessen benötigen. Für diese Schritte benötigen die Berechnungen nur Werte des eigenen Prozesses. Dies gilt für alle $m \in \mathbb{N}_0$ mit $m < p - q$, sofern die m die im Algorithmus beschriebene Zählvariable bezeichnet.

Für alle anderen Schritte muss ein Prozess immer genau mit einem anderen kommunizieren. Hierbei muss jeder Prozess alle lokal gespeicherten Werte versenden und alle gespeicherten Werte des anderen Prozesses empfangen. Dann kann prinzipiell wieder das Matrix-Vektor-Produkt berechnet werden. Allerdings muss darauf geachtet werden, dass jeweils eine Zeile durch einen Prozess berechnet wird.

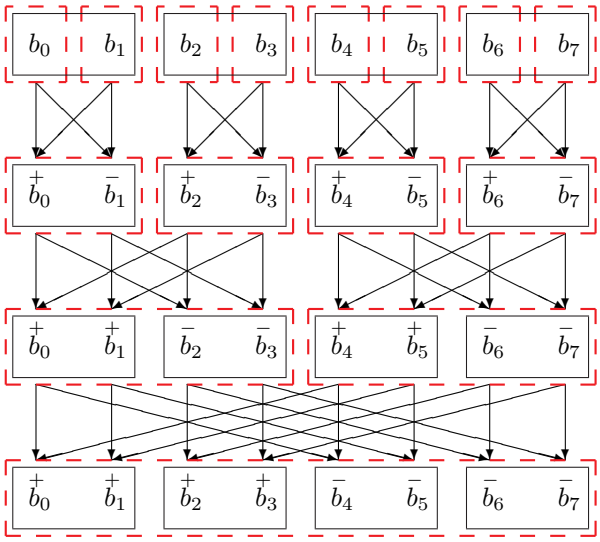

Abbildung 5: Funktionsweise eines Butterfly Graph auf mehreren Prozessen am Beispiel von $p = 3, q = 2$

Im folgenden Algorithmus wurde der Zugriff auf die Twiddle-Faktoren abkürzend durch $\xi(m, p, t)$ beschrieben, da die Verteilung solcher Faktoren im Allgemeinen beliebig sein kann.

Listing: paralleler FFT-Algorithmus

$0..2^q-1 \parallel \rho$ **program** Parallel-FFT
declare
m, t : integer ;
ω : array$[0..2^p-1]$ of complex ;
b, x : array$[0..2^{p-q}-1]$ of complex
initially
{*look-up table for twiddle factors*}
$\langle\ ;\ t\ :\ 0 \leq t < 2^p\ ::\ \omega[t] = \exp(-\frac{2\pi i t}{n})\ \rangle$;
{*bit-reverse ordering*}
$\langle\ ;\ t\ :\ 0 \leq t < 2^{p-q}\ ::\ b[t] = g(\sigma_p(2^{p-q}\rho + t))\ \rangle$
assign
{*butterfly calculation with no communication*}
$\langle\ ;\ m\ :\ 0 \leq m < p-q\ ::$
$\langle\ ;\ t\ :\ 0 \leq t < 2^{p-q}$ **and** $t \wedge 2^m = 0\ ::$
$\begin{bmatrix} b[t] \\ b[t+2^m] \end{bmatrix} := \frac{1}{2} \begin{bmatrix} 1 & \omega[\xi(m,\rho,t)] \\ 1 & -\omega[\xi(m,\rho,t)] \end{bmatrix} \begin{bmatrix} b[t] \\ b[t+2^m] \end{bmatrix}$
$\rangle$
$\rangle$;
{*butterfly calculation with communication*}
$\langle\ ;\ m\ :\ 0 \leq m < q\ ::$
send $\{b[t]\ |\ 0 \leq t < 2^{p-q}\}$ **to** $\rho \bar{\vee} 2^m$;
receive $\{x[t]\ |\ 0 \leq t < 2^{p-q}\}$ **from** $\rho \bar{\vee} 2^m$;
if $\rho \wedge 2^m = 0$ **then**
$\langle\ ;\ t\ :\ 0 \leq t < 2^{p-q}$ **and** $t \wedge 2^m = 0\ ::$
$b[t] := \frac{1}{2}\left(b[t] + \omega[\xi(m,\rho,t)]x[t]\right)\ \rangle$
else
$\langle\ ;\ t\ :\ 0 \leq t < 2^{p-q}$ **and** $t \wedge 2^m = 0\ ::$
$b[t] := \frac{1}{2}\left(x[t] - \omega[\xi(m,\rho,t)]b[t]\right)\ \rangle$
end
$\rangle$
end

3.3 Leistungsanalyse

Während der Ausführung des Algorithmus gibt es genau $p-q$ Teilschritte, in denen nicht kommuniziert wird. Bezeichnet man die durchschnittliche Durchführungszeit einer Computeroperation mit τ_{A}, dann ergibt sich die Zeit der Teilschritte ohne Kommunikation zu

$$2^{p-q+1}(p-q)\tau_{\mathrm{A}}$$

Für die anderen q Teilschritte ergibt sich die reine Berechnungszeit analog.

$$2^{p-q+1}q\tau_{\mathrm{A}}$$

Es werden $2q$ Kommunikationen durchgeführt, welche alle 2^{p-q} Daten verschicken. Die reine Kommunikationszeit ergibt sich zu

$$q\left(2\tau_{\mathrm{S}} + 2^{p-q+1}\beta\right)$$

Sei nun $T(n,P)$ die Gesamtzeit der Berechnung.

$$\boxed{T(n,P) = 2\frac{n}{P}\left(\tau_{\mathrm{A}}\log_2 n + \beta\log_2 P\right) + 2\tau_{\mathrm{S}}\log_2 P}$$

Der sogenannte Speedup gibt nun eine Aussage darüber, für welches Verhältnis n/P die größte Beschleunigung erlangt werden kann.

$$\boxed{S(n,P) := \frac{T(n,1)}{T(n,P)} = \frac{P}{1+\frac{\log_2 P}{\log_2 n}\left(\frac{\beta}{\tau_{\mathrm{A}}}+\frac{P}{n}\frac{\tau_{\mathrm{S}}}{\tau_{\mathrm{A}}}\right)}}$$

Der Koeffizient $\frac{\tau_{\mathrm{S}}}{\tau_{\mathrm{A}}}$ wird im Allgemeinen vergleichsweise groß sein. Damit sind Berechnungen mit $P \ll n$ sehr effizient und Berechnungen mit $P \approx n$ dagegen nicht.

Literatur

[1] `http://cnx.org/contents/ulXtQbN7@15/Implementing-FFTs-in-Practice`.

[2] `https://en.wikipedia.org/wiki/Cooley-Tukey_FFT_algorithm`.

[3] `https://en.wikipedia.org/wiki/Fast_Fourier_transform`.

[4] Eric F. Van de Velde. *Concurrent Scientific Computing*. Springer-Verlag, 1994.

[5] Rami Shakarchi Elias M. Stein. *Fourier Analysis - An Introduction*, volume I of *Princeton Lectures in Analysis*. Princeton University Press, 2003.

[6] Jürgen Elstrodt. *Maß- und Integrationstheorie*. Springer, 2009.

[7] Martin Hermann. *Numerische Mathematik*. Oldenbourg Verlag München, 2011.