PARALLELISIERUNG VON PET-REKONSTRUKTIONEN

HOLGER BISCHOF UND FRANK WÜBBELING

ZUSAMMENFASSUNG. In diesem Artikel untersuchen wir die Parallelisierbarkeit eines klassischen Algorithmus zur Rekonstruktion aus PET–Daten, des ListMode–OSEM–Algorithmus. Nach einer kurzen Einordnung und Herleitung geben wir eine konkrete Implementation im Rahmen (zunächst) eines shared–memory–Rechners. Das Laufzeitverhalten unter mehreren Architekturen wird untersucht.

1. Fragestellung im Ganzkörper–PET

In der Positronen–Emissionstomographie (PET) wird ein zu untersuchendes Objekt mit einer radioaktiven Substanz versetzt. Die austretende Positronen–Strahlung wird bei Auftreffen auf ein Elektron vernichtet, es entsteht Photonen–Strahlung, die Photonen werden in (fast) entgegengesetzte Richtungen ausgesandt und außerhalb des Objekts detektiert.

Idealerweise finden die Zerfallsprozesse zeitlich getrennt statt, die einzelnen Koinzidenzen werden entweder ungeordnet in einer Liste (für list mode) oder gesammelt in bins (für rebinning) notiert. Die mathematische Aufgabe ist die Bestimmung einer Schätzung für die radioaktive Verteilung im Objekt (also insbesondere in drei Dimensionen).

In dieser Notiz beschreiben wir Parallelisierungsmöglichkeiten für die Berechnung, wir betrachten daher den klassischen Rekonstruktionsalgorithmus. Korrekturen für auftretende messtechnische (Verlust von Ereignissen, Rauschen durch zufällige Koinzidenzen, Sensorcharakteristik) und quantitative Probleme (Dämpfung, Streuung) sind hier nicht angebracht worden, eine Übertragung der Ergebnisse auf entsprechend korrigierte Algorithmen ist aber möglich.

2. Mathematische Grundlagen

Die korrekte physikalisch–statistische Modellierung des Prozesses erhält man durch die Boltzmanngleichung (siehe [4]).

Sei $u(x, \theta, t)$ die Teilchendichte der Teilchen am Punkt x in Richtung θ zum Zeitpunkt t. Dann gilt für die Teilchendichte zum Zeitpunkt $t+\delta t$ (bei konstanter Geschwindigkeit c und infinitesimal kleinem δt)

$$\begin{aligned} u(x + c\,\delta t\,\theta, \theta, t + \delta t) &= u(x, \theta, t) - \delta t\, a(x) u(x, \theta, t) + \delta t\, q(x, \theta) \\ &\quad + \delta t \int_{S^2} v(x, \theta, \theta') u(x, \theta', t) d\theta' \end{aligned}$$

und damit

$$\frac{\partial u}{\partial t} + c\theta \nabla u + a(x)u = q(x, \theta) + \int_{S^2} v(x, \theta, \theta') u(x, \theta', t) d\theta'$$

innerhalb eines Gebiets Ω.

Hierbei ist q die Quellverteilung, a die Dämpfung, v die Streuung innerhalb des Objekts. Geeignete Randbedingung ist, dass von außen keine Strahlung eintritt. Sei also x ein Punkt aus dem Rand von Ω, ν_x die äußere Normale im Punkt x. Dann gilt für $\theta \cdot \nu_x < 0$

$$u(x, \theta, t) = 0.$$

Durch Übergang zum zeitunabhängigen Fall bekommen wir die Boltzmann–Gleichung

$$c\theta \nabla u(x, \theta) + a(x) u(x, \theta) = q(x, \theta) + \int_{S^2} v(x, \theta, \theta') u(x, \theta') d\theta'$$

mit derselben Randbedingung.

Üblicherweise wird in der Modellierung nun die Streuung vernachlässigt, wodurch sich die Gleichung auf die übliche Transportgleichung reduziert.

Sei nun $P(x, y, z)$ die Messantwort (Point–Spread–Function) auf eine Punktquelle im Punkt x, d.h. bei einer isotropen Strahlungs–Punktquelle im Punkt x werden $P(x, y, z)$ Koinzidenzen in den Sensoren y und z gemessen. Dann gilt für die Messantwort g für eine isotrop streuende Verteilung $q(x)$

$$g(y, z) = \int_{\Omega} P(x, y, z) q(x)\, dx.$$

Bei Vernachlässigung von Streuung und Dämpfung ist P bezüglich x eine Deltafunktion auf der Linie zwischen y und z, d.h. das Integral reduziert sich auf ein Linienintegral zwischen y und z.

Diese Beziehung wird für die numerische Berechnung nun geeignet diskretisiert. Üblicherweise wird $q(x)$ als Linearkombination überlappender Ansatzfunktionen dargestellt, der Einfachheit halber wählen wir

hier Voxel als Ansatzfunktionen. Das numerische Modell führt dann auf eine Gleichung

$$Aq = g.$$

Hierbei ist q der Vektor der Koeffizienten der Ansatzfunktionen für die Quellverteilung $q(x)$, g der Vektor der Messwerte für die Koinzidenzen. Die Systemmatrix A enthält in Zeile k die Messantwort auf eine Quellverteilung wie in Ansatzfunktion k, bei Unterdrückung von Dämpfung und Streuung ist A_{kj} das Linienintegral der Ansatzfunktion k über die Linie (p_j, q_j), falls g_j die Anzahl der Koinzidenzen in den Messpunkten p_j und q_j angibt. Die Gleichung ist nun entweder diskret zu lösen oder über ein analytisches Verfahren. Bei Identifizierung der Messdaten mit Linienintegralen und Rebinning kann eine gefilterte Rückprojektion durchgeführt werden, was sich in der Praxis jedoch nicht bewährt, da der statistische Charakter des Experiments nicht berücksichtigt wird. Es werden deshalb im allgemeinen statistisch begründete Verfahren gewählt, wir betrachten exemplarisch den ListMode–OSEM–Algorithmus.

3. Algorithmus: ListMode–OSEM

Gegeben sei ein PET-Datenvektor g, g_k sei die gemessene Anzahl der Zerfälle auf der Strecke $L = (p_k, q_k)$. Dann bestimmt sich die Schätzung $\tilde{q}(x)$ für die gesuchte radioaktive Dichte $q(x)$ aus der linearen Gleichung $Af = g$ mit $\tilde{q}(x) = \sum f_k \chi_k$ mit geeigneten Ansatzfunktionen χ_k. Im einfachsten Fall ist χ_k die charakteristische Funktion eines Voxels und damit f_k der Mittelwert der Dichte in diesem Voxel, dies nehmen wir hier beispielhaft an.

Im klassischen EM-Verfahren nach [7] wird f als Grenzwert der Iteration

$$f_{l+1} = f_l \left(\frac{1}{A^t \mathbf{1}} A^t \left(\frac{g}{A f_l} \right) \right)$$

berechnet mit $A_{kl} = \int_{p_k}^{q_k} \chi_l(s) ds$. Hierbei ist wie üblich bei der Multiplikation und Division von Vektoren die punktweise Operation gemeint.

Die (p_k, q_k) entstehen dabei durch eine Diskretisierung des gesamten Raums der Linien, die den Träger der Dichte berühren. Die eintreffenden Koinzidenzmessungen (a_s, b_s) werden also in einem Rebinning-Schritt den diskretisierten Linien zugeordnet und gezählt. Dies ist unerwünscht, da notwendigerweise die sehr genau gemessenen Linien verwischen. Zur Behebung dieses Problems wird das Listmode-Verfahren eingeführt, hierbei werden die gemessenen Linienpaare (a_s, b_s) direkt

zur Berechnung der Matrix A eingesetzt. Wir erhalten also $p_k = a_k$, $q_k = b_k$ sowie $g_k = 1$.

Nachteile bei diesem Vorgehen sind die extreme Größe der Matrix A und die Tatsache, dass bei der Konstruktion des Verfahrens noch nicht absehbar ist, welche Linienpaare in der Berechnung auftauchen. Es ist also nicht möglich, in einem Preprocessing–Schritt die Elemente der Matrix A ganz oder teilweise vorzuberechnen, sie müssen im Laufe der Iteration berechnet werden. Tatsächlich nimmt diese Berechnung den weitaus größten Teil der Rechenzeit in Anspruch, die Zeit für die Durchführung des EM-Algorithmus ist vernachlässigbar. Diese Aussage ist insbesondere auf die Dämpfungskorrektur übertragbar: Auch hier liegt beim Aufwand zum weitaus größten Teil das Problem in der Berechnung der Matrixkoeffizienten.

Zur Beschleunigung der notorisch langsamen EM-Verfahren wird üblicherweise zum OSEM-Verfahren übergegangen [3]. Hier werden für einen Schritt des EM-Verfahrens nicht alle Linien, sondern nur ein kleiner Teil davon eingesetzt. Im Effekt wird im l-ten Schritt des OSEM-Verfahrens ein Schritt des EM-Verfahrens durchgeführt mit einer Matrix A_l, die aus einigen Zeilen der Matrix A besteht. Im klassischen (nicht–Listmode) EM–Algorithmus werden hier üblicherweise die Zeilen zu parallelen Linien zusammengefasst. In Analogie zur Konvergenzanalyse des ART-Verfahrens ist die Konvergenz von OSEM schnell, wenn die zu aufeinander folgenden Matrizen gehörenden Richtungen möglichst orthogonal stehen. Eine Analyse in (Natterer, 1986) zeigt aber, dass eine Alternative mit fast identischen Konvergenzeigenschaften in der Wahl zufälliger Richtungen besteht. In der Übertragung auf den Listmode–Algorithmus folgt, dass (durch die tatsächlich zufällige Anordnung der Linien, die durch den Zerfallsprozess bestimmt ist) der OSEM–Algorithmus in diesem Fall hervorragende Konvergenzeigenschaften zeigt.

4. Listmode OSEM Implementierung

Zur parallelen Implementierung des Listmode OSEM Algorithmus verwenden wir den skelettbasierten Programmieransatz [6], wodurch ein systematisches Vorgehen ermöglicht wird. Einerseits wird durch eine schnelle Anpassung der parallelen Algorithmen an die mathematischen und algorithmischen Modifikationen im OSEM Algorithmus durch die Dämpfungskorrektur erlaubt. Es ist vorgesehen, neuartige numerische Methoden für die Dämpfungskorrektur zu entwickeln. Dabei wird es wichtig sein, neue Lösungsvorschläge schnell in laufende

Programme umzusetzen. Andererseits müssen entwickelte Parallelprogramme möglichst portabel zwischen verschiedenen Rechnerarchitekturen sein. Der Einsatz von Skeletten erlaubt einen schnellen Wechsel des eingesetzten Rechnerarchitekturtyps bzw. eine Anpassung an zukünftige Architekturen für Hochleistungsrechner.

Intuitiv sind Skelette generische Programmstrukturen, die als Bausteine für ein paralleles Programm dienen. Dabei existiert für jedes Skelett eine effiziente Implementierung, die für verschiedene parallele Systeme optimiert vorliegt. Die Parallelität ist vollständig im Skelett gekapselt und bleibt somit transparent für den Anwendungsprogrammierer.

Formal betrachtet werden Skelette als Funktionen höherer Ordnung aufgefasst, d. h. sie sind mittels Parameterfunktionen an eine Applikation anpassbar. Als Datenstrukturen werden Listen bzw. eindimensionale Felder verwendet. Eine Liste, bestehend aus den Elementen $a_1, \ldots, a_n$, wird im Folgenden als $[a_1, \ldots, a_n]$ dargestellt. Typische algorithmische Skelette sind das Map-Skelett *map*, das eine Funktion auf alle Elemente der Parameterliste anwendet und das Reduktion-Skelett *red*, das alle Elemente der Parameterliste mittels eines assoziativen Operators $\oplus$ kombiniert, zum Beispiel $red(min)\,[a, b, c] = min(a, b, c)$. Für beide Skelette, Map und Reduktion, existieren effiziente parallele Implementierungen. Die asymptotische Laufzeit mit einer Parameterliste der Länge n auf p Prozessoren beträgt $O(n/p)$ bei *map* und $O(n/p + \log p)$ bei *red*, für $n >> p$ wird somit eine lineare Beschleunigung gegenüber dem sequentiellen Fall erreicht.

Im skelettbasierten Programmieransatz startet man mit der mathematischen Spezifikation des ListMode OSEM Algorithmus:

$$f_{l+1} = f_l c_l, \quad c_l = \frac{1}{A_l^t \mathbf{1}} A_l^t \left(\frac{1}{A_l f_l} \right)$$

bei der die Zeilen der Matrix A_l durch die Events des Subsets S_l gebildet werden. Die Berechnungen der c_l innerhalb eines Subsets S_l können als Summe über alle Events i dieses Subsets umformuliert werden:

$$c_l = \frac{1}{A_l^t \mathbf{1}} \sum_{i \in S_l} \left(A_l^1\right)_i \frac{1}{(A_l)_i \, f_l} \tag{1}$$

wobei $(A_l)_i$ die Zeile i der Matrix A_l ist, die sich wiederum aus der Messung i ergibt.

Gleichung (1) lässt sich als Komposition von Skeletten auf Listen darstellen. Als Datenstruktur wird eine Liste von Events des Subsets

S_l verwendet. Dann gilt:

$$(2)\quad c_l = g_l \circ \mathit{red}(+) \circ (\mathit{map}(h_l \circ \mathit{line})),\ g_l\, x \stackrel{\text{def}}{=} \frac{1}{A_l^t \mathbf{1}} x,\ h_l\, x \stackrel{\text{def}}{=} x^t \left(\frac{1}{x f_l}\right)$$

wobei die Relation $\circ$ zwei Funktionen komponiert, d. h. $(f \circ g)\, x = f(g(x))$, und die Funktion *line* eine Zeile der Matrix aus den Messwerten eines Events erzeugt. Untersuchungen der Laufzeit mit einem Profiler haben gezeigt, dass die Funktionen h_l und *line* ca. 80 % der Rechenzeit in der vorliegenden sequentiellen Implementierung benötigen. Somit ist eine gute Beschleunigung durch den Einsatz der beiden Skelette zu erwarten.

Ein wichtiger Aspekt wurde in (2) noch nicht berücksichtigt: nach der Berechnung wird das erzeugte dreidimensionale Bild noch entrauscht. Beim Entrauschen wird eine dreidimensionale Faltung durchgeführt, die als Komposition von drei eindimensionalen Faltungen in alle drei Dimensionsrichtungen berechnet wird. Messungen der vorliegenden Implementierung haben ergeben, dass dieser Programmteil einen wesentlichen Teil der gesamten Laufzeit benötigt (ca. 10 %), wodurch seine Parallelisierung ebenfalls notwendig ist.

Da die Berechnungen innerhalb einer Dimension unabhängig voneinander sind, kann hier eine modifizierte Version des *map* Skelettes verwendet werden, die in [2] eingeführt wurde: $\mathit{map}_d\, f$ bezeichnet die Anwendung der Funktion f auf alle Vektoren entlang der Dimension $d \in \{1, 2, 3\}$ einer dreidimensionalen Datenstruktur. Somit ergibt sich für dir Faltung:

$$(3)\qquad \mathit{blur} = (\mathit{map}_3\ \mathit{conv}) \circ (\mathit{map}_2\ \mathit{conv}) \circ (\mathit{map}_1\ \mathit{conv})$$

wobei *conv* die Entrauschung eines Vektors mit Hilfe eines Filters berechnet. Wenn man nun (3) der Implementierung (2) hinzufügt, ergibt sich insgesamt:

$$(4)\quad c_l = (\mathit{map}_3\ \mathit{conv}) \circ (\mathit{map}_2\ \mathit{conv}) \circ (\mathit{map}_1\ \mathit{conv}) \circ g_l \circ \mathit{red}(+) \circ (\mathit{map}(h_l \circ \mathit{line}))$$

Die rechte Seite von (2) stellt die parallele Implementierung für die Berechnung der c_l in skelettbasierter, funktionaler Notation dar. Die Implementierung in einer imperative Programmiersprache des gesamten Listmode OSEM Algorithmus besteht dann aus einer sequentiellen Schleife, in der die c_l sukzessive parallel berechnet werden. Abb. 1 zeigt die Implementierung in einer vereinfachten pseudo-C Notation.

Zu beachten ist, dass aufgrund ihrer Größe nicht alle Daten gleichzeig in den Arbeitsspeicher eingelesen werden können. Deshalb wird immer nur ein Subset von der Funktion `Read()` sequentiell eingelesen

```
for(l=0, l<subsets, l++)
    Read();     // Daten eines Subsets einlesen
    Map(line);  // line und h auf alle Elemente
    Map(h);     // des Subsets anwenden
    Red();      // Ergebnisse aller Events aufsummieren
    g();
    for(d=1, d<=3, d++)   // entrauschen
      Map(d,conv);
```

Abbildung 1. Pseudo C Code des OSEM Algorithmus

und nach den Berechnungen auf diesem Subset wieder freigegeben. Um die Funktion `Read` zu parallelisieren, müssen die Daten auf verteilten Festplatten abgelegt sein. Das Einlesen der Daten nimmt jedoch nur einen geringen Teil der Gesamtlaufzeit ein (ca. 3 %), sodass dieser Teil in der momentanen Version nicht parallelisiert wurde.

Die beiden Skelette Map und Reduce werden im folgenden Abschnitt parallel mit Hilfe von Posix Threads implementiert.

5. Implementierung der Skelette mit Posix Threads

POSIX threads [5] (oder kurz pthreads) ist eine von IEEE standardisierte Bibliothek, die C Funktionen zur Programmierung mit mehreren Threads bereitstellt. Insbesondere wird die Funktion `pthread_create` zum Erzeugen eines neuen Threads und die Funktion `pthread_join` zum Warten auf die Beendigung eines Threads benutzt. Zur Synchronisierung mehrerer Threads werden so genannte Mutexe (von "Mutual Exclusion") verwendet, um kritische Programmabschnitte unter gegenseitigem Ausschluss auszuführen.

Die Funktion `pthread_mutex_lock` zeigt an, dass ein Thread einen geschützten Bereich betritt, indem er den Mutex sperrt. Fordert nun ein anderer Thread den selben Mutex an, wird dieser so lange blockiert, bis der Mutex durch den ersten Thread mit `pthread_mutex_unlock` freigegeben wird.

Mit Hilfe dieser Pthread Funktionen kann sowohl das Map als das Reduktions-Skelett implementiert werden.

Für das Map werden so viele Threads gestartet wie Prozessoren zur Verfügung stehen. Jeder der erzeugten Threads führt nun die Berechnungen auf einem Teil der Daten aus. Danach wartet der Haupt-Thread mit der Funktion `pthread_join` auf die Beendigung der Berechnung aller Threads.

Für die Implementierung der Reduktion wird als erster Schritt das parallel Map aufgerufen, d. h. es werden p Threads erzeugt, wobei jeder Thread auf einem gleich großen Teil der Eingabedaten die Ergebnisse der Funktion h aufsummiert. Am Ende der Berechnungen addiert jeder Thread sein lokales Ergebnis auf das globale Ergebnis. Da hier alle Threads gleichzeitig auf das globale Ergebnis zugreifen, muss der Zugriff durch einen Mutex geschützt werden.

Die parallele Implementierung der beiden Skelette, Map und Reduktion, wird nun im Programm aus Abb. 1 verwendet. Auf die Darstellung der Implementierungen der Funktionen g, h und *conv* wird hier aus Platzgründen verzichtet. Die Implementierung der Funktion *line* wird im folgenden Abschnitt dargestellt.

5.1. **Erzeugung der Matrix.** Die Messungen, die das Programm als Eingabe bekommt, sind die Positionen der beiden Punkte, an denen ein Zerfall im PET-Scanner gemessen wurde. Die Zeile der Matrix gibt den Schnitt jedes Voxels mit der Linie an, die durch die beiden Messpunkte bestimmt ist. Abb. 2 zeigt die Schnitte einer Linie mit den Voxels in der Ebene (grau unterlegte Voxel).

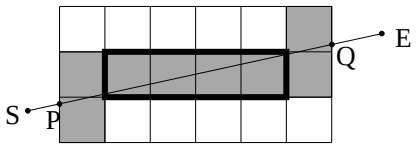

Abbildung 2. Schnitte einer Linie mit Voxels in der Ebene

Für die Berechnung der Schnitte wird die Methode von Siddon [8] verwendet. Hierbei wird der jeweils nächste Schnittpunkt mit dem Rand eines Voxels ermittelt, beginnend mit dem Startpunkt S. Im Beispiel aus Abb. 2 ist dies der Punkt P. Das wird solange wiederholt, bis der Rand der Voxelebene erreicht ist (im Beispiel der Punkt Q). Für jedes Voxel wird die Länge der Linie innerhalb des Voxels berechnet. Da eine Linie nur wenige Voxels schneidet, können die errechneten Längen als Liste von Index/Längen-Paare gespeichert werden (im Beispiel acht Paare).

Zu beachten ist, dass die Werte für alle Voxels, die im Beispiel dick eingerahmt sind, gleich sind, d. h. hierfür muss der Schnitt nicht jeweils neu berechnet werden. Diese Optimierung wurde von Reader und Zhao vorgeschlagen [9].

Die Schnitte müssen für jede Messung berechnet werden und verbrauchen den überwiegenden Teil der Laufzeit des gesamten Algorithmus. Aufgrund der zweimaligen Verwendung dieser Werte bei der Berechnung von h, werden sie zwischengespeichert. Man beachte, dass der benötigte Speicher klein ist, da eine Linie nur wenige Voxels schneidet.

6. Experimente

Die ersten Experimente wurden auf folgenden zwei modernen Parallelrechnern durchgeführt: (1) einer SunFire V880 mit acht UltraSparc III+ 1200 MHz Prozessoren und (2) einem Linux System bestehend aus zwei Intel Xeon 2.8 GHz Prozessoren, wobei jeder Prozessor Hyperthreading unterstützt.

Ausgangsbasis der untersuchten parallelen Implementierung war ein von Projektpartnern zur Verfügung gestellte sequentielle Programm in der Programmiersprache C. Dieser Code wurde im Hinblick auf (2) analysiert und die entsprechenden Skelette identifiziert. Dabei mussten kleine Modifikationen vorgenommen werden. Danach konnten die identifizierten Skelette durch ihre parallelen Versionen basierend auf pthreads ersetzt werden. Diese Version ist nur als Testversion zu sehen, um die Skalierbarkeit des ListMode OSEM Algorithmus zu demonstrieren. Das langfristige Ziel ist eine portable Skelett-Bibliothek zu benutzen, damit neue Algorithmen schneller implementiert werden können.

Als Datensatz für unsere Zeitmessungen wurden sechs Millionen Messungen von Koinzidenz-Ereignissen des PET Scanners QuadHidac für ein Maus Phantom verwendet. Der Datensatz ist folglich unter realen Bedingungen (Daten vom PET Scanner) entstanden, sodass davon auszugehen ist, dass Koinzidenz-Messungen an lebenden Objekten zu ähnlichen Laufzeitmessungen führen werden.

Die Ereignis-Daten wurden in 53 Subsets zu je 120 000 Messungen geteilt. Dabei wurde ein Ausschnitt von $150 \times 150 \times 280$ Voxel betrachtet. Abb. 3 zeigt die Beschleunigung abhängig von der Anzahl der verwendeten Threads, links für die Linux-Maschine und rechts für die SunFire.

Die Hyperthreading Technologie ermöglicht eine bessere Auslastung des Prozessors durch das Bereitstellen eines virtuellen Prozessors, wobei im Wesentlichen ein zusätzlicher Registersatz verfügbar ist, jedoch kein zusätzlicher vollständiger Prozessorkern. Während eines Zugriffs auf den langsamen Hauptspeicher bekommt der andere Thread den Zugriff auf die Funktionseinheiten des Prozessorkerns, sodass beide Threads quasi parallel ausgeführt werden. Da diese Prozessoren keine

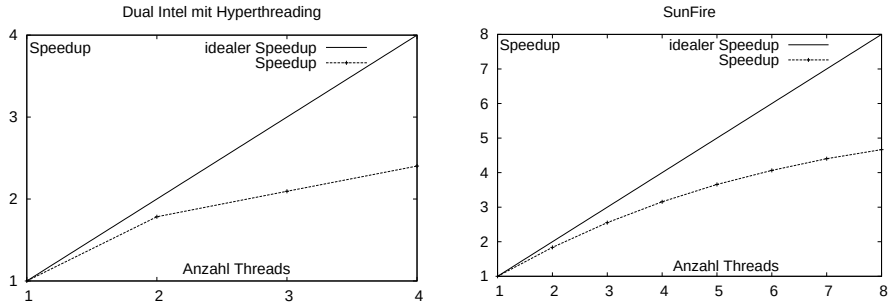

ABBILDUNG 3. Beschleunigung durch Parallelisierung, abhängig von der Anzahl der Threads

zwei vollständigen Kerne besitzen, ist erfahrungsgemäß eine Beschleunigung von ca. 30–40 % zu erwarten. Im Beispiel für die Messungen in Abb. 3 (links) wurde eine Beschleunigung von 35 % erreicht.

Insgesamt ergibt sich auf der Linux Maschine mit zwei Prozessoren (bzw. vier virtuellen) eine Beschleunigung um den Faktor 2,4.

Die Messungen auf der Sunfire Multiprozessor-Maschine zeigen, dass unsere Implementierung mit pthreads auf Maschinen mit gemeinsamem Speicher für größere Anzahl von Prozessoren nicht mehr linear skaliert. Der Grund liegt darin, dass einige Programmteile noch nicht parallelisiert wurden, speziell das Einlesen von Messdaten und Schreiben der Ergebnisse (3D Bilder) auf die Festplatte. Die Laufzeit der Ein-/Ausgabe beträgt in der sequentiellen Implementierung ca. 3 % zur gesamten Laufzeit bei, sodass der Anteil bei acht Threads bereits auf 14 % anwächst. Somit müsste für einen höheren Beschleunigung bei acht oder mehr Prozessoren auch die Ein-/Ausgabe parallelisiert werden.

7. AUSBLICK

Die Implementierung im Abschnitt 4 ist eine modifizierte Version der sequentiellen Implementierung unserer Projektpartner, die um Multithreading erweitert wurde. Wünschenswert ist, ein Skelettsystem wie z. B. die generische C++-Bibliothek DatTeL für Parallelrechner oder das Grid-System der Arbeitsgruppe PVS für einen Rechnerverbund [1] zu verwenden. Dadurch können Programme für verschiedene parallele Maschinen portabel und effizient entwickelt und implementiert werden. Hierzu müssen sequentielle Codes für die Operationen $\cdot$ und $+$ sowie für die Funktionen g und h implementiert werden, die dann als Parameter an entsprechende DatTeL- Bibliotheksfunktionen bzw. Grid-Services übergeben werden.

Als weiterer Schritt soll die Implementierung nach MPI (Message Passing Interface) portiert werden, um die Programme auf Hochleistungsrechnern mit verteiltem Speicher ausführen zu können. Dies wird auch eine verteilte Daten Ein-/Ausgabe über verteilte Festplatten ermöglichen und verspricht einen wesentlich größeren Beschleunigungsfaktor für das ListMode OSEM Verfahren.

Literatur

[1] Holger Bischof, Sergei Gorlatch, and Roman Leshchinskiy. DatTeL: A data-parallel C++ template library. *Parallel Processing Letters*, 13(3):461–472, September 2003.

[2] Sergei Gorlatch and Holger Bischof. A generic MPI implementation for a data-parallel skeleton: Formal derivation and application to FFT. *Parallel Processing Letters*, 8(4):447–458, 1998.

[3] HM Hudson and RS Larkin. Accelerated image-reconstruction using ordered subsets of projection data. *IEEE transcations on medical imaging*, 13(4):601–609, 1994.

[4] Frank Natterer and Frank Wübbeling. Scatter correction in pet based on transport models. Technical report, Fachbereich Mathematik, Universität Münster, 2004.

[5] Technical Committee on Operating Systems and Application Environments of the IEEE. POSIX, part 1: System API, ANSI/IEEE Std 1003.1c, amendment 2: Threads extension. IEEE Standards Press, 1996.

[6] Fethi A. Rabhi and Sergei Gorlatch, editors. *Patterns and Skeletons for Parallel an d Distributed Computing*. Springer Verlag, 2003.

[7] LA Shepp and Y Vardi. Maximum likelihood reconstruction for emission tomography. *IEEE Trans Med Imag*, 1:113–122, 1982.

[8] Robert L. Siddon. Fast calculation of the exact radiological path for a three-dimensional CT array. *Medical Physics*, 12(2):252–255, 1985.

[9] Huaxia Zhao and Andrew J. Reader. Fast ray tracing technique to calculate line integral paths in voxel arrays. Technical Report M11-97, University of Manchester, 2003.

Fachbereich Mathematik und Informatik, Universität Münster, Einsteinstrasse 62, D–48149 Münster

E-mail address: {hbischof,frank.wuebbeling}@math.uni-muenster.de