FORSCHUNGSZENTRUM JÜLICH GmbH
Jülich Supercomputing Centre
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Beiträge zum Wissenschaftlichen Rechnen Ergebnisse des Gaststudentenprogramms 2007 des John von Neumann-Instituts für Computing

Matthias Bolten (Hrsg.)

FZJ-JSC-IB-2007-12

Dezember 2007

(letzte Änderung: 1. 12. 2007)

Vorwort

Die Ausbildung im Wissenschaftlichen Rechnen ist neben der Bereitstellung von Supercomputer-Leistung und der Durchführung eigener Forschung eine der Hauptaufgaben des John von Neumann-Instituts für Computing (NIC) und hiermit des JSC als wesentlicher Säule des NIC. Um den akademischen Nachwuchs mit verschiedenen Aspekten des Wissenschaftlichen Rechnens vertraut zu machen, führte das JSC in diesem Jahr zum achten Mal während der Sommersemesterferien ein Gaststudentenprogramm durch. Entsprechend dem fächerübergreifenden Charakter des Wissenschaftlichen Rechnens waren Studenten der Natur- und Ingenieurwissenschaften, der Mathematik und Informatik angesprochen. Die Bewerber mussten das Vordiplom abgelegt haben und von einem Professor empfohlen sein. Die zehn vom NIC ausgewählten Teilnehmer kamen für zehn Wochen, vom 6. August bis 12. Oktober 2007, ins Forschungszentrum. Zusätzlich nahm dieses Jahr ein Praktikant der Firma IBM am Gaststudentenprogramm teil. Alle Gaststudenten beteiligten sich hier an den Forschungs- und Entwicklungsarbeiten des JSC und der NIC Forschergruppe Computergestützte Biologie und Biophysik. Sie wurden jeweils einem oder zwei Wissenschaftlern zugeordnet, die mit ihnen zusammen eine Aufgabe festlegten und sie bei der Durchführung anleiteten.

Die Gaststudenten und ihre Betreuer waren:

Eduardo Aguilar Moreno	Bernd Körfgen
Sebastian Birk	Bernhard Steffen
Andreas Brätz	Armin Seyfried
Stephanie Friedhoff	Matthias Bolten
Stefanie Hittmeyer	Thomas Neuhaus, NIC
Thomas Kaczmarek	Paul Gibbon
Michael Knobloch	Michael Hennecke, IBM
Michael Pippig	Matthias Bolten, Godehard Sutmann
Alexander Rüttgers	Holger Dachsel
Andreas Uhe	Thomas Müller
Alexander Weuster	Godehard Sutmann

Zu Beginn ihres Aufenthalts erhielten die Gaststudenten eine viertägige Einführung in die Programmierung und Nutzung der Parallelrechner im JSC. Um den Erfahrungsaustausch untereinander zu fördern, präsentierten die Gaststudenten am Ende ihres Aufenthalts ihre Aufgabenstellung und die erreichten Ergebnisse. Sie verfassten zudem Beiträge mit den Ergebnissen für diesen Internen Bericht des JSC. Wir danken den Teilnehmern für ihre engagierte Mitarbeit - schließlich haben sie geholfen, einige aktuelle Forschungsarbeiten weiterzubringen - und den Betreuern, die tatkräftige Unterstützung dabei geleistet haben, insbesondere Marc-André Hermanns, Bernd Mohr und Boris Orth, die den Einführungskurs gehalten haben. Ebenso danken wir allen, die im JSC und der Verwaltung des Forschungszentrums bei Organisation und Durchführung des diesjährigen Gaststudentenprogramms mitgewirkt haben. Besonders hervorzuheben ist die finanzielle Unterstützung durch den Verein der Freunde und Förderer des FZJ und die Firma IBM. Es ist beabsichtigt, das erfolgreiche Programm künftig fortzusetzen, schließlich ist die Förderung des wissenschaftlichen Nachwuchses dem Forschungszentrum ein besonderes Anliegen. Weitere Informationen über das Gaststudentenprogramm, auch die Ankündigung für das kommende Jahr, findet man unter http://www.fz-juelich.de/jsc/gaststudenten.

Jülich, November 2007 Matthias Bolten

Inhalt

Eduardo Aguilar Moreno:
Entwicklung eines Software-Interfaces zwischen dem FEM-Paket LS-DYNA und dem JSC-eigenen Postprocessing-Tool RAPS 1

Sebastian Birk:
Test paralleler mehrdimensionaler FFT-Software 11

Andreas Brätz:
Zur Druckberechnung in der parallelen Version des NIST Fire Dynamics Simulators v5.0 23

Stephanie Friedhoff:
Algebraische Mehrgitterverfahren für strukturierte Matrizen 33

Stefanie Hittmeyer:
The ISOMAP Algorithm for Non-linear Dimensionality Reduction 51

Thomas Kaczmarek:
Visualization of Star-Disc Evolution in Star Clusters with Xnbody 63

Michael Knobloch:
Performance Optimization on Blue Gene by Optimized Task Placement 75

Michael Pippig:
Parallele schnelle Fouriertransformation nichtäquidistanter Daten 91

Alexander Rüttgers:
Fehlerabschätzungen in der Fast Multipole Method bei Systemen mit periodischen Randbedingungen 107

Andreas Uhe:
Implementation of a Quantum Monte Carlo Code on BGL 123

Alexander Weuster:
Improving Communication in a Force Decomposition Algorithm 139

Entwicklung eines Software-Interfaces zwischen dem FEM-Paket LS-DYNA und dem JSC-eigenen Postprocessing-Tool RAPS

Eduardo Aguilar Moreno

Technische Universität München
Fakultät für Informatik
Lehrstuhl für Wissenschaftliches Rechnen

E-mail: eduardo.aguilarmoreno@mytum.de

Zusammenfassung:
LS-DYNA, ein kommerzielles Finite-Elemente-Analyse Programmpaket, wird vom Jülich Supercomputing Centre (JSC) für die Nutzer im Forschungszentrum Jülich auf dem Supercomputer Jump zur Verfügung gestellt. Im Rahmen des Gaststudentenprogramms des NIC wurde eine neue Schnittstelle zwischen LS-DYNA und dem JSC-eigenen Postprocessing-Tool RAPS entwickelt. Diese Schnittstelle erlaubt die Konvertierung binärer Daten, die auf unterschiedlichen Rechnerarchitekturen erzeugt wurden.

Einleitung

Postprozessoren für Finite-Elemente-Analysen sind im Allgemeinen kommerzielle Pakete, wie z.B. die von LSTC entwickelten LS-POST und LS-PREPOST. RAPS hat im Gegensatz zu kommerziellen Postprozessoren den großen Vorteil, daß der Quelltext modifiziert und RAPS an die Anforderungen der Benutzer angepasst werden kann. In diesem Bericht wird ein Software-Interface vorgestellt, mit dem RAPS auch als ein Postprocessing-Tool zu LS-DYNA benutzt werden kann. Um ein besseres Verständnis des Problems zu ermöglichen, wird im folgenden eine kleine Einführung in die Finite-Elemente-Analyse gegeben.

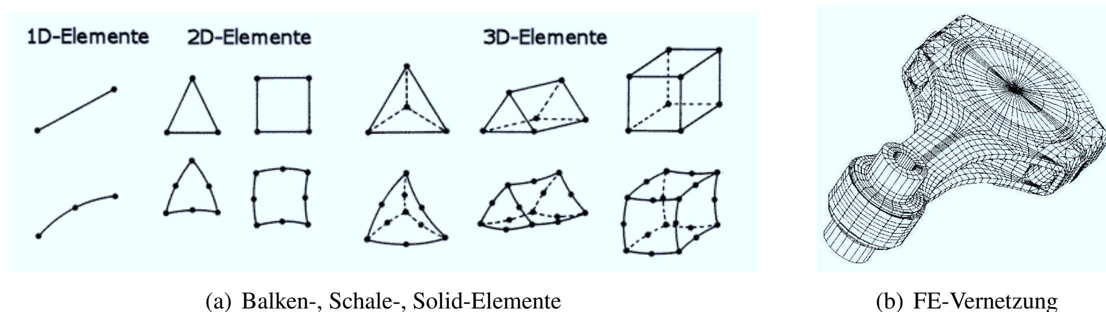
Finite-Elemente-Analyse (FEA)

Die Finite-Elemente-Analyse ist eine auf der *Finite-Elemente-Methode (FEM)* basierende Computersimulationstechnik. Die FEM wurde von R. Courant 1943 entwickelt [1], der die Ritz-Methode [2] der Numerischen Mathematik im Zusammenhang mit einer Variationsformulierung benutzte, um Näherungslösungen von Systemen bei Vibration zu berechnen. Die resultierende FEM ist ein numerisches Verfahren zur näherungsweisen Lösung, insbesondere elliptischer, partieller Differentialgleichungen mit Randbedingungen. Das Berechnungsgebiet wird dabei in eine große Zahl kleiner, aber endlich (finit) vieler Elemente unterteilt. Auf diesen Elementen werden Ansatzfunktionen definiert, aus denen sich über die partielle Differentialgleichung und die Randbedingungen ein großes Gleichungssystem ergibt. Die Lösung dieses Gleichungssystems liefert die gesuchten Ergebnisse.

Ein klassisches Einsatzgebiet der FEM ist die Berechnung der Spannungen, Verschiebungen und Verzerrungen mechanischer Objekte und Systeme. Außerdem wird die FEA häufig benutzt bei der Analyse anderer Problemen, wie z.B. Wärmeleitung, Fluiddynamik und Elektromagnetismus.

Im Allgemeinen besteht eine Finite-Elemente-Analyse aus drei Phasen, nämlich:

- *Preprocessing*
  In dieser Phase muß ein geometrisches Finite-Elemente-Modell von dem Objekt oder System erstellt werden. Die geometrische Darstellung des Modells wird normalerweise mit der Hilfe von CAD-Software in ein, zwei, oder drei Dimensionen durchgeführt. Ein Vernetzungsprozess folgt, um das Modell in kleine Elemente (s. Abbildung 1(a)) zu unterteilen. Die Vernetzung (s. Abbildung 1(b)) besteht aus einer geometrischen Anordnung von Knoten und Elementen. Knoten stellen Punkte dar, an dennen bestimmte Charakteristiken oder Eigenschaften, z.B. Verschiebungen, berechnet werden. Elementen werden durch eine gewisse Anzahl von Knoten definiert und sie beschreiben die lokalisierten Masse-/Steifigkeitseigenschaften des Modells. Für Identifizierungszwecke sind Knoten und Elementen in der Regel nummeriert.



(a) Balken-, Schale-, Solid-Elemente

(b) FE-Vernetzung

Abbildung 1: Diskretisierung eines Modells durch Finite Elemente

- *FEM-Berechnung (Solver)*
  Zusätzliche Eigenschaften, u.a. die auf das Objekt wirkende Kräfte, die Materialeigenschaften sowie die physikalischen Randbedingungen werden in dieser Phase berücksichtigt, um ein Gleichungssystem zu erzeugen, das von einem sogenannten Solver [3] gelöst wird. Die Ergebnisse stellen die Effekte, in der Strukturmechanik z.B. die Verschiebungen, Spannungen und Verzerrungen unter Lasten wie Kräften, Druck oder Gravitation, auf das Modell dar. Abhängig von der Komplexität des Problems können diese Berechnung auf einem normalen Arbeitsplatzrechner (Workstation) durchgeführt werden oder es wird ein Supercomputer, wie z.B. *Jump (Jülich Multi Processor)* [4], benötigt.

- *Postprocessing*
  Die bei der FEM-Berechnung komplexer Probleme erzeugten großen Datenmengen müssen interpretiert und ausgewertet werden. Dafür werden normalerweise Visualisierungstools benutzt, die eine graphische Darstellung der Ergebnisse liefern und damit ein tieferes Verständnis der Prozesse ermöglichen.

LS-DYNA

In den 70er Jahren wurde LS-DYNA ursprünglich als DYNA3D von John Hallquist entwickelt, der damals im Lawrence Livermore National Laboratory (LLNL) arbeitete. 1987 verließ Hallquist LLNL und gründet Livermore Software Technology Corporation (LSTC). Seitdem wird LS-DYNA von LSTC weiterentwickelt und ist eine viel genutzte Software geworden, die zahlreiche Anwendungsgebiete gefunden hat, besonders in Crash-Test-Simulationen, Fluid-Struktur-Wechselwirkung, Simulationen von Fertigungsverfahren und anderen hochdynamischen Problemstellungen.

LS-DYNA gehört zur Familie der FEM-Löser, d.h. LS-DYNA besteht aus einer einzigen, ausführbaren Datei und besitzt keine graphische Benutzeroberfläche. Diese Charakteristik macht LS-DYNA besonders geeignet für eine parallele Ausführung des Programms. Die Ergebnisse der Berechnungen sind in binären Datenbanken gespeichert. Im folgenden wird eine kleine Einführung in die binären Datenbanken von LS-DYNA gegeben.

Binäre LS-DYNA-Ausgabedateien

LS-DYNA erzeugt drei unterschiedliche Klassen von binären Datenbanken, die sogenannten *State Database*, *Time History Database* und *Interface Force Database* [5]. Die Datenbanken werden als adressierbare binäre Dateien mit fixierter Länge geschrieben. Die Größe der Dateien hängt von der Komplexität des FE-Modells ab, sie ist aber immer ein Vielfache von 512 Speicherwörtern. Die Länge der Speicherwörter ist 4 Byte im *Single Precision Mode* und 8 Byte im *Double Precision Mode* von LS-DYNA. Im Allgemeinen ist es nicht sinnvoll, die Datenbank in einer einzigen Datei zu speichern, daher werden die Daten auf einer Familie von Dateien verteilt. Dies ist hilfreich bei der Bearbeitung (z.B. Kopieren, Speichern) der Daten.

Im folgenden wird die Struktur der Zustandsdatenbank (State Database) beschrieben. Für die Implementierung der Schnittstelle wurde momentan nur diese Datenbank berücksichtigt, da sie die wichtigsten Daten enthält, die für die Visualisierung einer FEM-Simulation benötigt werden.

State Database

Die Zustandsdatenbank wird von LS-DYNA als *d3plot* bezeichnet, außerdem gibt es noch zwei andere Datenbanken dieser Art, nämlich *d3drfl* oder *Dynamic Relaxation File* [6], die Information über den Zustand am Ende des DR-Prozess (*Dynamic Relaxation Process*) enthält und *d3part*, die Ausgabedaten über bestimmte Komponenten des Modells beinhaltet. Die Zustandsdatenbank besteht aus drei Abteilungen (s. Abbildung 2). Die erste Abteilung enthält 64 Kontrollwörter, die Informationen über den Inhalt der Datenbank liefern. Die zweite Abteilung schließt die Information der FE-Modellierung, nämlich die Koordinaten und Nummerierung der Knoten, sowie die Konnektivität der Elemente ein. Der dritte Abschnitt umfasst die Ergebnisse für alle Zeitschritte. Die Ausgabe für jeden Zeitschritt enthält globale Variablen wie die totalen Energien und Momente, sowie die Materialien der Komponenten des Modells, Knoten-Daten (Verschiebungen, Geschwindigkeiten, Beschleunigungen und Temperaturen) und Element-Daten (Spannungen und Verzerrungen).

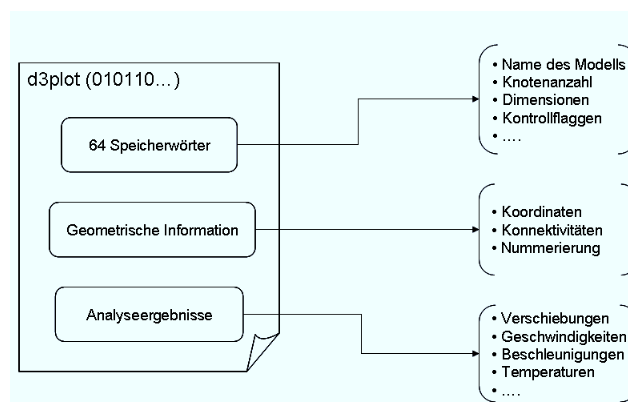


Abbildung 2: Struktur der d3plot-Datei

RAPS

RAPS ist ein interaktiver graphischer Finite-Element-Postprozessor mit Schwerpunkt auf der Darstellung der Rechenergebnisse [7]. RAPS wurde im Forschungszentrum Jülich ursprünglich als ein Postprocessing-Tool zu den FEM-Paketen ASKA und PERMAS entwickelt. Die Architektur der Software wird in der Abbildung 3 gezeigt. RAPS besteht aus den Postprocessing-Routinen sowie aus einer Visualisierungsschnittstelle, die unabhängig sind, d.h. der Quelltext der Postprocessing-Routinen kann modifiziert werden, ohne die Visualisierungsschnittstelle zu beeinflussen. Das Visualisierungsinterface ist nicht nur an ein X-Window-Modul gekoppelt sondern auch an ein VR-Modul *(Virtual Reality Module)*. Die VR-Version von RAPS wird auf einem Stereoprojektionssystem *(Holobench)* [8] des JSC eingesetzt. Ein Interface zwischen RAPS und LS-DYNA bietet dem Benutzer die Möglichkeit, eine mächtige Software zur FEM-Analyse zusammen mit einer dreidimensionalen stereoskopischen Visualisierung zu nutzen.

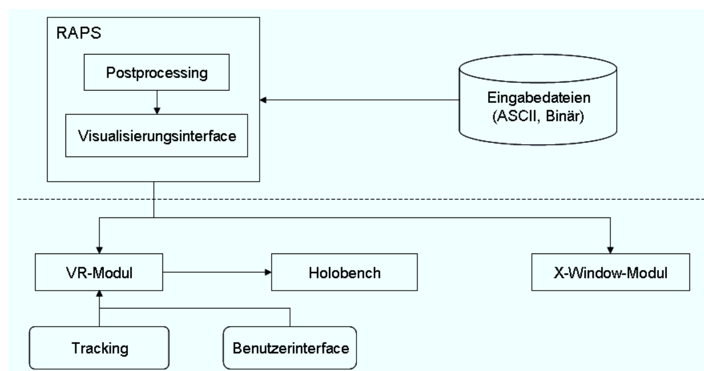


Abbildung 3: RAPS-Architektur

RAPS-Eingabedateien

Als Eingabe benutzt RAPS verschiedene binäre sowie ASCII-Dateien. Für die Entwicklung der Schnittstelle wurden momentan nur jene Dateien verwendet, die die geometrische Information des FE-Modells enthalten, d.h. mit diesen Daten wird eine graphische Darstellung der Simulation ermöglicht.

- *npco*
  Diese Datei wird im binären Format geschrieben und enthält die Nummerierung sowie die Koordinaten aller Knoten. Abbildung 4 zeigt im ASCII-Format die Struktur dieser Datei. Der Dateikopf umfasst u.a. einen Marker (-1111) sowie die Anzahl von Datenspalten und die Dateityp (NPCO). Für einen dreidimensionalen Testfall werden z.B. nur 3 Datenspalten geschrieben. Das Dateiende wird durch einen Marker (-9999) sowie die Marke FIN definiert.

- *elemente*
  Diese Datei wird im ASCII-Format geschrieben und enthält die Konnektivitäten der Elemente (s. Abbildung 5). Der Dateikopf umfasst u.a. einen Marker (-3334), den Name des Modells (AIRBAG AND STRUCTURE), und den Dateityp (LSS). Abhängig von den Eigenschaften der Elemente, wird die Information in Elementgruppen angeordnet. Jede Elementgruppe wird durch einen zugeordneten Dateikopf definiert, der den Elementtyp (QUAD4), die Anzahl der Elemente (400), die Anzahl der Knoten pro Element (4), und die Nummerierung der Elemente in der Gruppe (1) enthält. Die Daten bestehen aus der Nummerierung der Elemente mit der entsprechenden Konnektivität jedes Elements.

- *usr*
  Die Verschiebungen der Knoten werden für jeden Zeitschritt (Lastfall) in binärem Format gespeichert. Abbildung 6 zeigt im ASCII-Format die Struktur dieser Datei. Der Dateikopf enthält einen Marker (-1111), die Anzahl von Datenspalten, den Dateityp (USR) und den Lastfall (1). Das Dateiende wird nach dem letzten Lastfall geschrieben und auch durch einen Marker (-9999) und die Marke FIN bezeichnet.

Dateikopf	-1111	NPCO	1	3	1	1
Daten	3722	2.2500E+00	2.5000E+00	-1.2500E+01	0.0000E+00	0.0000E+00
	3723	2.2068E+00	2.9390E+00	-1.2500E+01	0.0000E+00	0.0000E+00
	3724	2.0787E+00	3.3610E+00	-1.2500E+01	0.0000E+00	0.0000E+00
	3725	1.8708E+00	3.7500E+00	-1.2500E+01	0.0000E+00	0.0000E+00
	3726	1.5910E+00	4.0910E+00	-1.2500E+01	0.0000E+00	0.0000E+00
	3727	1.2500E+00	4.3708E+00	-1.2500E+01	0.0000E+00	0.0000E+00
	3728	8.6104E-01	4.5787E+00	-1.2500E+01	0.0000E+00	0.0000E+00
	3729	4.3895E-01	4.7068E+00	-1.2500E+01	0.0000E+00	0.0000E+00
	3730	0.0000E+00	4.7500E+00	-1.2500E+01	0.0000E+00	0.0000E+00
	3731	-4.3895E-01	4.7068E+00	-1.2500E+01	0.0000E+00	0.0000E+00
	3732	-8.6104E-01	4.5787E+00	-1.2500E+01	0.0000E+00	0.0000E+00
	3733	-1.2500E+00	4.3708E+00	-1.2500E+01	0.0000E+00	0.0000E+00
	3734	-1.5910E+00	4.0910E+00	-1.2500E+01	0.0000E+00	0.0000E+00
	3735	-1.8708E+00	3.7500E+00	-1.2500E+01	0.0000E+00	0.0000E+00
	3736	-2.0787E+00	3.3610E+00	-1.2500E+01	0.0000E+00	0.0000E+00
	3737	-2.2068E+00	2.9390E+00	-1.2500E+01	0.0000E+00	0.0000E+00
	3738	-2.2500E+00	2.5000E+00	-1.2500E+01	0.0000E+00	0.0000E+00
	:	:	:	:	:	:
	7591	-1.2780E+01	0.0000E+00	-1.3780E+01	0.0000E+00	0.0000E+00
	7592	-1.2780E+01	0.0000E+00	1.3780E+01	0.0000E+00	0.0000E+00
Dateiende	-9999	FIN	0	0	0	0

Abbildung 4: npco-Datei

Dateikopf	LSS 1	AIRBAG AND STRUCTURE		-3334	
Elementgruppe	QUAD4	1	400	4	
Daten	6993	7547	7509	7148	7549
	6994	7509	7510	7149	7148
	6995	7510	7511	7150	7149
	6996	7511	7512	7151	7150
	6997	7512	7513	7152	7151
	6998	7513	7514	7153	7152
	6999	7514	7515	7154	7153
	7000	7515	7516	7155	7154
	7001	7516	7517	7156	7155
	7002	7517	7518	7157	7156
	7003	7518	7519	7158	7157
	7004	7519	7520	7159	7158
	7005	7520	7521	7160	7159
	7006	7521	7522	7161	7160
	7007	7522	7523	7162	7161
	7008	7523	7524	7163	7162
	7009	7524	7525	7164	7163
	7010	7525	7526	7165	7164
	7011	7526	7527	7166	7165
	:	:	:	:	:
Elementgruppe	QUAD4	4	1	4	
	7393	7589	7590	7591	7592

Abbildung 5: elemente-Datei

LS-DYNA-RAPS-Konverter

Abbildung 7 zeigt das Flußdiagramm der Konvertierung der binären Ausgabedateien von LS-DYNA in ASCII und binären Eingabedateien für RAPS. Bei der Programmierung der Schnittstelle mussten einige technische Schwierigkeiten überwunden werden, die nachfolgend beschrieben werden:

Dateikopf (Lastfall 1)	-1111	USR	1	3	1	1
	3722	0.0000E+000	0.0000E+000	0.0000E+000	0.0000E+000	0.0000E+000
	3723	0.0000E+000	0.0000E+000	0.0000E+000	0.0000E+000	0.0000E+000
Daten	3724	0.0000E+000	0.0000E+000	0.0000E+000	0.0000E+000	0.0000E+000
	3725	0.0000E+000	0.0000E+000	0.0000E+000	0.0000E+000	0.0000E+000
	3726	0.0000E+000	0.0000E+000	0.0000E+000	0.0000E+000	0.0000E+000
	:	:	:	:	:	:
Dateikopf (Lastfall 62)	-1111	USR	1	3	1	62
	3722	-1.8665E-004	2.3670E-003	-1.1444E-005	0.0000E+000	0.0000E+000
	3723	-2.0228E-004	2.3556E-003	-1.2398E-005	0.0000E+000	0.0000E+000
	3724	-2.1546E-004	2.3403E-003	-1.3351E-005	0.0000E+000	0.0000E+000
	3725	-2.2566E-004	2.3251E-003	-1.4305E-005	0.0000E+000	0.0000E+000
	3726	-2.3252E-004	2.3079E-003	-1.5259E-005	0.0000E+000	0.0000E+000
Daten	:	:	:	:	:	:
	7588	0.0000E+000	0.0000E+000	0.0000E+000	0.0000E+000	0.0000E+000
	7589	0.0000E+000	0.0000E+000	0.0000E+000	0.0000E+000	0.0000E+000
	7590	0.0000E+000	0.0000E+000	0.0000E+000	0.0000E+000	0.0000E+000
	7591	0.0000E+000	0.0000E+000	0.0000E+000	0.0000E+000	0.0000E+000
	7592	0.0000E+000	0.0000E+000	0.0000E+000	0.0000E+000	0.0000E+000
Dateiende	-9999	FIN	0	0	0	0

Abbildung 6: usr-Datei

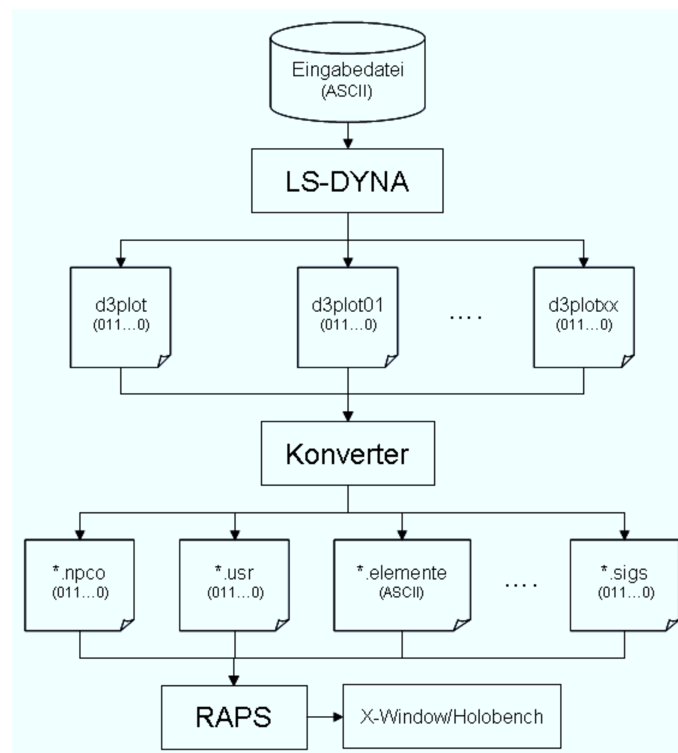


Abbildung 7: Konvertierungsprozess

Endianness

LS-DYNA wird vom JSC für die Nutzer im Forschungszentrum Jülich auf dem Supercomputer Jump zur Verfügung gestellt. Dieser Supercomputer ist ein IBM Regatta p690+ Cluster, der eine Big-Endian-Architektur (s. Abbildung 8(b)) hat. RAPS ist nicht nur auf dem Jump sondern auch auf den Linux-PCs des JSC installiert, die im Gegensatz dazu eine Little-Endian-Architektur (s. Abbildung 8(a)) haben. Dies stellt ein Problem dar, da damit keine Portabilität von binären Dateien zwischen beiden Architekturen gegeben ist. Der Konverter muß in der Lage sein, eine d3plot-Datei unabhängig von ihrer Endianness in RAPS-Datensätze zu konvertieren.

A9FBFFFF **FFFFFBA9**

(a) Little Endian (b) Big Endian

Abbildung 8: Endianness eines 4 Byte-Wortes

Genauigkeit

LS-DYNA erlaubt abhängig von der benötigten Rechengenauigkeit Berechnungen im *Single Precision Mode* und *Double Precision Mode* durchzuführen. Entsprechend werden die Ergebnisse in den d3plot-Dateien in 4 oder 8 Byte-Wörtern abgespeichert. Andererseits wurde RAPS mit einer fixierten Wortlänge von 4 Bytes implementiert. Der Konverter kann beide Arten von Dateien einlesen und eine entsprechende Konvertierung zum 4 Byte-Format durchführen. Der dabei unvermeidliche Verlust in der Genauigkeit ist für die Visualisierung ohne Bedeutung.

Fortran Record Header

Ein weiteres Problem war der sogenannte *Fortran Record Header*. RAPS wurde in Fortran programmiert, während der Konverter in C implementiert wurde. In Fortran werden binäre Daten mit einer *Record*-Struktur inkl. des sogenannten *Record Headers* geschrieben. In C dagegen, werden Binär-Daten ohne Record-Struktur geschrieben. Die Abbildungen 9(a) und 9(b) zeigen 8 Byte in hexadezimalem Format (Little-Endian-Architektur).

A9FBFFFF 4E50434F 80000000 **A9FBFFFF 4E50434F** 80000000

(a) Binäre Ausgabe in C (b) Binäre Ausgabe in Fortran

Abbildung 9: Fortran Record Header

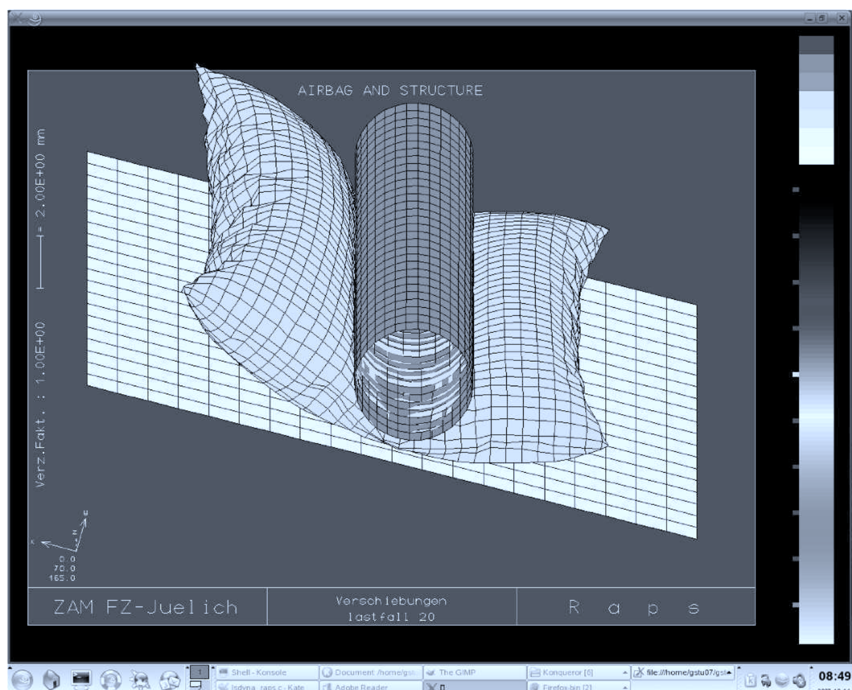
Ergebnisse der Implementierung

Für Testzwecke wurde ein einfaches Problem ausgewählt und zwar die airbag.deploy.k-Datei [9], um die Funktionalität des Konverters zu testen. In diesem Beispiel wird ein Airbag unter einem steifen Zylinder innerhalb von 0.03 Sekunden vollständig aufgeblasen. Der Zylinder wird wegen der Expansion des Airbags in die Luft geworfen. Das FE-Modell besteht aus 3793 Schalen-Elementen. Als Ausgabedaten liefert LS-DYNA 63 d3plot-Dateien die insgesamt mit dem Konverter in die folgende Dateien umgewandelt wurden: *d3plot.npco*, *d3plot.elemente* bzw. *d3plot.usr*.

Die Simulation des Problems wurde erst mit LS-POST visualisiert, so daß sie später mit der Ausgabe von RAPS verglichen werden konnte. Nach der Konvertierung der Dateien konnte die Simulation auch in RAPS erfolgreich visualisiert werden. Abbildung zeigt den Zustand der Simulation nach 0.0094992 Sekunden mit hilfe von RAPS.

Andere Testfälle mit anderen Elementkonfigurationen wurden mit dem Konverter ebenfalls getestet, z.B. die control_energy.bar-impact.k-Datei [9]. In diesem Testfall kollidiert eine Kupfer-Stange bei hoher Geschwindigkeit mit einer Wand. Das FE-Modell besteht aus 972 Hexaeder-Elementen.

Schließlich wurde eine Simulation der Stabilität eines Bauwerks während eines Erdbebens als Testfall benutzt. Das FE-Modell in diesem Fall besteht aus Solid-Elementen sowie Balkenelementen. Der Konverter konnte erfolgreich die Eingabedateien für RAPS mit dieser Konfiguration von Finiten Elementen erzeugen.



Schlußwort und Ausblick

Ein funktionelles Interface zwischen LS-DYNA und RAPS wurde implementiert, das aber wegen der begrenzten Zeit des Gaststudentenprogramms unvollständig bleibt. Das Interface wurde mit einigen Testfällen erfolgreich getestet. Volumen-Elemente mit 8 Knoten sowie Schalen-Elemente mit 4 Knoten und Balkenelemente mit 2 Knoten wurden erfolgreich getestet. Das Verhalten des Konverters beim gleichzeitigen Auftreten aller Elementtypen muss noch untersucht werden.

Diese Arbeit legt die Grundlagen für ein vollständiges Interface zwischen LS-DYNA und RAPS. Obwohl das Interface schon genutzt werden kann, ist weitere Arbeit erforderlich, um alle Möglichkeiten von LS-DYNA und RAPS ausnutzen zu können.

Danksagungen

Ich möchte mich bei Herrn Matthias Bolten herzlich bedanken, für die hervorragende Organisation des Gaststudentenprogramms 2007. Besonderer Dank gilt meinem Betreuer Herrn Dr. Bernd Körfgen sowie Herrn Dietmar Koschmieder, für ihre Beratung und Unterstützung bei der Durchführung meiner Arbeit. Außerdem danke ich sehr allen anderen Gaststudenten, die meinen Aufenthalt in Jülich zu einer denkwürdigen Erfahrung gemacht haben.

Supported by the Programme Alßan, the European Union Programme of High Level Scholarships for Latin America, Scholarship No. E05M056414MX.

Literatur

1. Klein, B., *FEM - Grundlagen und Anwendungen der Finite-Elemente-Methode*, Vieweg, 1990
2. Steinbuch, R., *Finite Elemente - Ein Einstieg*, Springer, 1998
3. Fröhlich, P., *FEM - Leitfaden*, Springer, 1995
4. Jump
   http://jumpdoc.fz-juelich.de/
5. Livermore Software Technology Corporation, *LS-DYNA Database Binary Output Files*, August 2006, pp. 1-30
   http://www.lsdyna-portal.com/fileadmin/files/lsdyna/website/pdf/pdf_lsdyna/ls-dyna_database_manual_2006.pdf
6. Livermore Software Technology Corporation, Hallquist, John O., *LS-DYNA Theory Manual*, März 2006
   http://www.lstc.com/
7. RAPS
   http://www.fz-juelich.de/vislab/software/raps/raps_e.html
8. Holobench
   http://www.fz-juelich.de/jsc/cv/vislab/vr/hardware/
9. Livermore Software Technology Corporation, Reid, John D., *LS-DYNA Examples Manual*, Juni 2001, pp. 1-10
   http://www.lstc.com/
10. Livermore Software Technology Corporation, *Keyword Manual 971 Vol 1&2*, Mai 2007, pp. 605-621
    http://www.lstc.com/
11. Kernighan, Brian W., Ritchie, Dennis M., *Programmieren in C*, Carl Hanser Verlag, München, 1990

Test paralleler mehrdimensionaler FFT-Software

Sebastian Birk

Bergische Universität Wuppertal
Fachbereich C - Mathematik und Naturwissenschaften
Gaußstraße 20
42119 Wuppertal

E-Mail: sbirk@studs.math.uni-wuppertal.de

Zusammenfassung: Im vorliegenden Artikel wird das Laufzeitverhalten dreier FFT-Bibliotheken auf dem Jülicher JUMP-Cluster untersucht. Zunächst wird ein kurzer Überblick über Symmetrien der FFT gegeben, die sich zur Speicher- und Rechenzeitersparnis ausnutzen lassen. Weiterhin werden die drei verwendeten FFT-Bibliotheken vorgestellt und das entwickelte Programm, welches als einheitliches Interface für diese Bibliotheken dient. Abschließend werden Ergebnisse präsentiert, welche das Cacheverhalten in Zusammenhang mit der Laufzeit bringen, und das Transponieren wird näher untersucht.

Einleitung

Fouriertransformationen bilden in vielen Anwendungen einen wesentlichen Bestandteil. So ist z.B. das Lösen partieller Differentialgleichungen über den Weg der Fouriertransformation möglich, da das Differenzieren bei bekannten Fourierkoeffizienten einer einfachen Multiplikation der Koeffizienten entspricht. An dieser Stelle finden FFTs Anwendung, da sie das Berechnen der Fourierkoeffizienten in $\mathcal{O}(N \log N)$ bewerkstelligen.

FFT - Die Fast Fourier Transformation

Seien $x_n, n \in \{0, 1, \ldots, N-1\}$ die Werte einer periodischen Funktion an äquidistanten Stützstellen mit $x_0 = x_N$. Dann bestimmt die FFT die Fourierkoeffizienten

$$\tilde{x}_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i \frac{kn}{N}}$$

Die ursprünglichen Stützstellen lassen sich damit als

$$x_k = \frac{1}{N} \sum_{n=0}^{N-1} \tilde{x}_n e^{2\pi i \frac{kn}{N}}$$

darstellen. Eine d-dimensionale FFT bestimmt die Koeffizienten $\tilde{x}_{k_1,k_2,\ldots,k_d}$ wie folgt:

$$\begin{aligned} \omega_s &:= e^{\frac{2\pi i}{N_s}} \\ \tilde{x}_{k_1,k_2,\ldots,k_d} &= \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \cdots \sum_{n_d=0}^{N_d-1} x_{n_1,n_2,\ldots,n_d} \omega_1^{-k_1 n_1} \omega_2^{-k_2 n_2} \ldots \omega_d^{-k_d n_d} \end{aligned}$$

Wie man durch

$$\tilde{x}_{k_1,k_2,\dots,k_d} = \sum_{n_1=0}^{N_1-1} \cdots \sum_{n_c=0}^{N_c-1} \left(\sum_{n_{c+1}=0}^{N_{c+1}-1} \cdots \sum_{n_d=0}^{N_d-1} x_{n_1,n_2,\dots,n_d} \omega_d^{-k_d n_d} \dots \omega_{c+1}^{-k_{c+1} n_{c+1}} \right) \omega_c^{-k_c n_c} \dots \omega_1^{-k_1 n_1}$$

erkennt, kann man eine d-dimensionale FFT auch durch $(d-c)$-dimensionale FFTs erhalten, deren Ergebnisse als Eingabedaten für eine c-dimensionale FFT verwendet werden. Außerdem kann die Reihenfolge der Summation beliebig variiert werden, da

$$\begin{aligned} \tilde{x}_{k_1,k_2} &= \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x_{n_1,n_2} \omega_1^{-k_1 n_1} \omega_2^{-k_2 n_2} \\ &= \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} x_{n_1,n_2} \omega_1^{-k_1 n_1} \omega_2^{-k_2 n_2} \end{aligned}$$

Symmetrien

Die folgenden wichtigen Symmetrien können ausgenutzt werden, um Rechenzeit und Speicherplatz einzusparen:

- Sind alle x_n reell, so sind die $\tilde{x}_n$ hermitesch und umgekehrt. D.h.

$$(\forall n\ x_n \in \mathbb{R}) \Leftrightarrow (\forall n\ \overline{\tilde{x}_n} = \tilde{x}_{N-n})$$

- Sind alle x_n reell und Stützstellen einer geraden Funktion, so sind die $\tilde{x}_n$ hermitesch mit gerader Symmetrie und umgekehrt. D.h.

$$(\forall n\ x_n \in \mathbb{R} \wedge x_n = x_{N-n}) \Leftrightarrow (\forall n\ \tilde{x}_n \in \mathbb{R} \wedge (\tilde{x}_n = \tilde{x}_{N-n}))$$

- Sind alle x_n reell mit ungerader Symmetrie, so sind die $\tilde{x}_n$ rein imaginär mit ungerader Symmetrie und umgekehrt. D.h.

$$(\forall n\ x_n \in \mathbb{R} \wedge x_n = -x_{N-n}) \Leftrightarrow (\forall n\ \mathrm{Re}(\tilde{x}_n) = 0 \wedge \tilde{x}_n = -\tilde{x}_{N-n})$$

Diese Symmetrien können ebenso in höheren Dimensionen auftreten. So kann z.B. in Richtung der ersten Dimension eine gerade und in Richtung der zweiten Dimension eine ungerade Symmetrie vorliegen. Führt man nun zunächst eine eindimensionale FFT in einer der Dimensionen durch, so kann man deren Symmetrie wie erwähnt ausnutzen. Die ursprüngliche Symmetrie der anderen Dimension geht dabei zwar nicht verloren, aber diese auszunutzen kann unter Umständen recht schwierig werden. Redundanz in den Datensätzen liegt zwar noch immer vor, aber die Anordnung der redundanten Daten im Speicher kann so verteilt sein, dass es sich nicht lohnt, dies auszunutzen. Es kommt also auf den Einzelfall der Anwendung an, ob man nach dem Anwenden der ersten FFT noch weitere Symmetrien betrachtet.

Die verwendeten FFT-Bibliotheken

Für die Tests standen 3 FFT-Bibliotheken zur Verfügung. Diesen ist gemeinsam, dass sie in einer seriellen Version vorlagen. Im Folgenden werden die verwendeten Bibliotheken erläutert. Besonderes Augenmerk gilt dabei den Datenstrukturen, die die jeweilige Bibliothek erwartet, und der Behandlung von Spezialfällen, wie reeller oder symmetrischer Daten.

Die FFTW

Die FFTW (Fastest Fourier Transform in the West) ist eine sehr umfassende C-Bibliothek, welche man unter [1] finden kann. Eine Besonderheit der FFTW ist ihr mächtiges Planning-Interface. Bevor eine FFT durchgeführt werden kann, muss ein Plan z.B. durch `fftw_plan_many_dft(...)` erstellt werden, der unter anderem die Daten

- Eingabe-Array,
- Ausgabe-Array,
- Dimensionen des Arrays,
- Anzahl der Transformationen und
- Intensität der Tests

enthält. Dabei können Eingabe- und Ausgabe-Array übereinstimmen, was die FFTW veranlasst, eine In-Place-Transformation durchzuführen. Mit der Anzahl der Transformationen kann man angeben, dass mehrere gleichartige FFTs auf im Speicher verteilte Daten angewendet werden sollen. So kann man z.B. mit nur einem erstellten Plan auf jede Zeile einer Matrix eine FFT anwenden. Die Intensität der Tests bestimmt, wie intensiv nach einem optimalen Plan für die durchzuführende FFT gesucht werden soll. So kann man mittels des Flags `FFTW_ESTIMATE` angeben, dass ein Plan ohne lange Tests ausgewählt werden soll. Mit dem Flag `FFTW_MEASURE` hingegen, werden umfangreichere Zeitmessungen und Test-Transformationen durchgeführt. Dies resultiert natürlich in einer längeren Planungszeit, die man sich bei der Durchführung der eigentlichen FFT zu gewinnen wünscht. Beim Planen der FFT versucht die FFTW eine Operationsreihenfolge, einen Algorithmus und eine Blockung der Daten so zu wählen, dass die Ausführungszeit des erstellten Plans minimal ist. Das Ausführen eines angelegten Plans erfolgt mit einem Aufruf von `fftw_execute(...)`.

Die FFTW erwartet als Datenformat ein gemischtes Array, in dem jedes Element von der Form

```
typedef struct {
  double re;
  double im;
} fftw_complex;
```

ist. Veranschaulicht wird dies in Abbildung 1. Möchte man hingegen eine Transformation mit reellen Eingangsdaten durchführen, so erwartet die FFTW ein Array aus `double`-Werten. Eine solche FFT kann man unter anderem über das real2complex-Interface `fftw_plan_many_dft_r2c(...)` planen. Wie bereits in der Einleitung gesehen, erhält man bei der FFT einer reellwertigen, geraden oder ungeraden Funktion reelle oder rein imaginäre Ausgabedaten. Diese können also als `double`-Array gespeichert werden. Für diese Aufgaben ist das `fftw_plan_many_dft_r2r(...)`-Interface der FFTW zuständig.

FFT-Routinen von Stefan Goedecker

Bei den FFT-Routinen von Stefan Goedecker handelt es sich um Fortran 77 und Fortran 90 Subroutinen, die man auf der Webseite [3] finden kann. Als einzige der drei Bibliotheken unterstützen diese ausschließlich komplexe Daten. Zur Verfügung standen Subroutinen für 2-, 3- und 4-dimensionale FFTs. Der erste Schritt war also das Ergänzen einer eindimensionalen FFT. Als Grundlage dafür wurde die

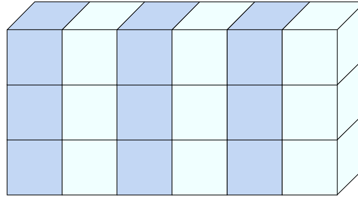


Abbildung 1: Datenformat der FFTW für ein Array der Größe 3×3

2-dimensionale Subroutine verwendet und derartig angepasst, dass man ein eindimensionales Feld transformieren kann. Eine entscheidende Einschränkung der Bibliothek ist die Beschränkung auf Problemgrößen der Form $N = 3^a 4^b 5^c 6^e 8^f$ mit $a, b, c, d, e, f \in \mathbb{N}_0$ so dass $3 \leq N \leq 1024$

Die FFT-Routinen von Stefan Goedecker erwarten ebenso wie die FFTW ein gemischtes Array mit Real- und Imaginärteilen. Eine Besonderheit ist aber ein Workarray, das nach dem Datenarray folgen und die gleiche Größe wie das Datenarray aufweisen muss. Wie in Abbildung 2 veranschaulicht wird, ist es erforderlich die Daten anders zu speichern, wenn man eine 2-dimensionale FFT durchführen will, als wenn mehrere eindimensionale angewendet werden sollen. Mittels eines Übergabeparameters muss man nach der erfolgten FFT prüfen, ob sich die Ausgabedaten im Eingabefeld oder im Workarray befinden.

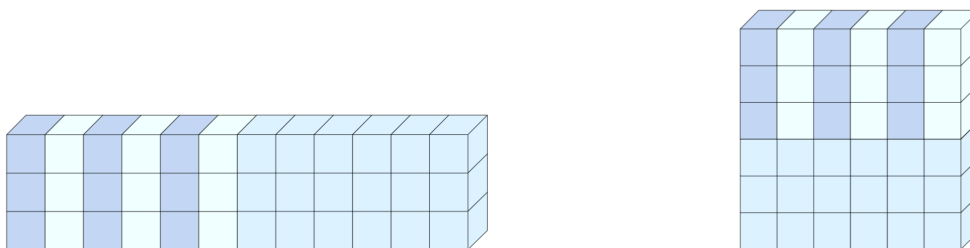


Abbildung 2: Datenformat der FFT-Routinen von Stefan Goedecker für ein Array der Größe 3×3 auf das 3 eindimensionale FFTs angewendet werden sollen (links) bzw. eine dreidimensionale FFT angewendet werden soll (rechts)

FFT-Routinen von Steve Kifowit

Die dritte und letzte FFT-Bibliothek von Steve Kifowit besteht ebenfalls aus einer Ansammlung von Fortran 90 Subroutinen. Zu finden sind diese auf [4]. In dieser Sammlung gibt es separate Subroutinen für die FFT komplexer, reeller und reell-symmetrischer Daten. Allerdings handelt es sich bei diesen Routinen ausschließlich um eindimensionale FFTs, was ein häufiges Transponieren erfordert. Eine weitere Einschränkung stellt die Anforderung an die Problemgröße $N = 2^a$ mit $a \in \mathbb{N}$ dar.

Im Gegensatz zu den beiden anderen Bibliotheken erfordert diese zwei getrennte Arrays - jeweils ein eigenes für den Real- und den Imaginärteil. Im Fall reeller, symmetrischer Daten werden die Eingabedaten mit Padding erwartet. D.h. wenn eine FFT der Größe $N = 2^a$ durchgeführt werden soll, sind zwar nur $n/2$ erforderlich, aber das Array muss trotzdem eine Größe von N haben.

Das erstellte Testprogramm

Allgemeines

Um die im vorherigen Abschnitt genannten Bibliotheken testen zu können, wurde ein C-Programm geschrieben, welches ein einheitliches Interface für alle 3 Bibliotheken bereitstellt. Aufgabe dieses Rah-

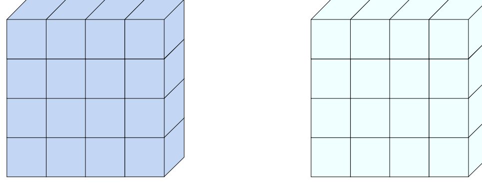


Abbildung 3: Datenformat der FFT-Routinen von Steve Kifowit für ein Array der Größe 4×4

menprogramms ist es zudem, Testfälle zu generieren, die erzeugten Testfälle in das Datenformat der zu testenden Bibliothek zu konvertieren, die Datenverteilung mittels MPI vorzunehmen und das Transponieren der Daten durchzuführen. Es wurde dabei so konzipiert, dass FFTs beliebiger Dimension durchführbar sind. Limitierende Faktoren sind dabei der zur Verfügung stehende Arbeitsspeicher und die unten genauer beschriebene eindimensionale Distribution der Daten.

Im Folgenden sei $A \in \mathbb{C}^{N_0 \times N_1 \times \cdots \times N_{D-1}}$. D ist also die Anzahl der Dimensionen, N_d die Anzahl der Datensätze in Richtung der Dimension $d \in \{0, 1, \ldots, D-1\}$. P sei die Anzahl der Prozessoren.

Datendistribution

Als Datendistribution wurde eine eindimensionale Verteilung bezüglich der letzten Dimension $D-1$ gewählt. Dabei ist Dimension 0 die Dimension, die kontinuierlich im Speicher vorliegt. Außerdem werden die Daten möglichst gleichmäßig auf die Prozessoren verteilt. Um diese gleichmäßige Verteilung zu erreichen erhält jeder Prozessor ein Teilarray $A_p \in \mathbb{C}^{N_0 \times N_1 \times \cdots \times N_{D-2} \times n_p}$ mit $n_p := l(N_{D-1}, p, P)$ für $p \in \{0, 1, \ldots, P-1\}$. Dabei ist l die Abbildung

$$\begin{array}{rcl} l : (\mathbb{N}, \mathbb{N}_0, \mathbb{N}) & \to & \mathbb{N} \\ (n, p, P) & \mapsto & \begin{cases} \left\lfloor \frac{n}{P} \right\rfloor & \text{, falls } (0 = n \mod P) \vee (p \geq n \mod P) \\ \left\lfloor \frac{n}{P} \right\rfloor + 1 & \text{, sonst} \end{cases} \end{array}$$

die die Anzahl der lokalen Datensätze ermittelt.

In Abbildung 4 wird diese Datenverteilung für ein Array der Dimensionen $6 \times 4 \times 7$ und 3 Prozessoren veranschaulicht.

Wie im Beispiel gesehen entsteht eine ungleichmäßige Verteilung der Daten, sofern $0 \neq N_{D-1} \mod P$. Es wäre zwar möglich, ausgewogener zu verteilen, aber das würde bedeuten, dass die $(D-2)$-te Dimension nicht in allen Prozessoren vollständig vorliegt. Damit könnte keine vollständige $(D-1)$-dimensionale FFT ohne zusätzliches Transponieren der Daten durchgeführt werden.

Um eine FFT in Richtung der verteilten Dimension $D-1$ durchführen zu können, werden die Daten transponiert, so dass die vorher verteilte Dimension nach dem Transponieren vollständig lokal vorliegt. Sei dafür S_D die symmetrische Gruppe in D Buchstaben auf der Menge $\{0, 1, \ldots, D-1\}$ und sei $\pi \in S_D$ eine Permutation. Das Transponieren ist dann ein Isomorphismus

$$\begin{array}{rcl} t(\pi) : \mathbb{C}^{N_0, N_1, \ldots, N_{D-1}} & \to & \mathbb{C}^{\pi(N_0), \pi(N_1), \ldots, \pi(N_{D-1})} \\ A = (a_{n_0, n_1, \ldots, n_{D-1}}) & \mapsto & (a_{\pi(n_0), \pi(n_1), \ldots, \pi(n_{D-1})}) = t(\pi)(A) =: \pi(A) \end{array}$$

Die Anzahl der Elemente, die von Prozessor i an Prozessor j gesendet werden müssen beträgt

$$\frac{\prod_{d=0}^{D-1} N_d}{N_{D-1} \cdot N_{\pi(D-1)}} \cdot l(N_{D-1}, i, P) \cdot l(N_{\pi(D-1)}, j, P).$$

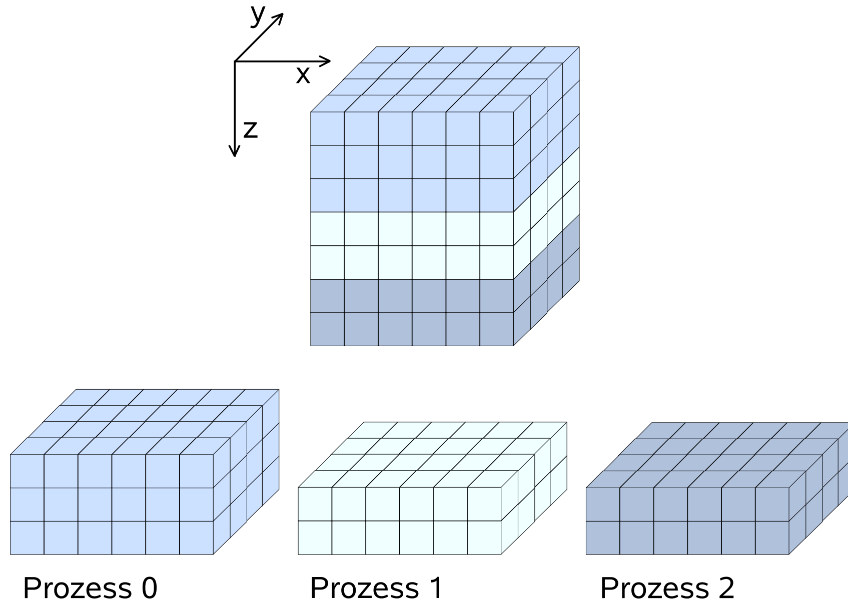


Abbildung 4: Datenverteilung auf 3 Prozessoren

Das Transponieren wird in den Abbildungen 5 und 6 anhand des oben genannten Beispiels mit der Permutation $\pi = (0 \;\; 2)(1)$ verdeutlicht.

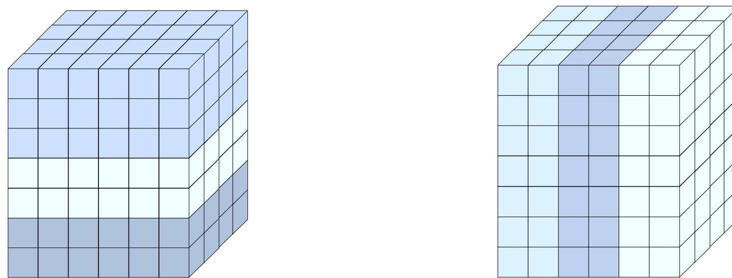


Abbildung 5: Das Array A der Dimension $6 \times 4 \times 7$ (links) soll so transponiert werden, dass die x- und z-Koordinate tauschen und ein Array der Größe $7 \times 4 \times 6$ entsteht, dessen Datenverteilung rechts dargestellt ist.

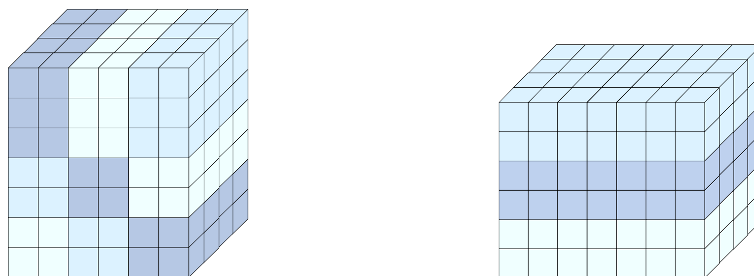


Abbildung 6: Im linken Bild ist die nötige Kommunikation zwischen den Prozessoren zu erkennen. Vertikal ist die alte Datendistribution und horizontal die neue ablesbar. Rechts ist das Ergebnis $t(\pi)(A) = \pi(A)$ des Transponierens dargestellt.

Das geschriebene Testprogramm kann Permutationen beliebiger Dimension durchführen, wenn sich die Permutation π auf folgende Art darstellen lässt:

$$\pi = (0 \quad d)(1 \quad d+1)\ldots(k \quad d+k)(k+1)\ldots(d-1)$$

Dabei muss natürlich $d + k < D$ und $k < d$ sein. D.h. es kann ein beliebiger zusammenhängender Block von Dimensionen in die Dimension 0 transponiert werden, wenn sich dabei keine Dimensionen überschneiden.

Das Transponieren selbst erfolgt durch `MPI`. Dazu ermittelt zunächst jeder Prozessor, welcher Teil der lokalen Daten an welchen anderen Prozessor gesendet werden muss, wie es beispielhaft in den Abbildung 5 und 6 zu erkennen ist.
Anschließend werden D Datenstrukturen vom Type `MPI_Type_hvector` geschachtelt, um den entsprechenden Block an Daten versenden zu können. Die Syntax von `MPI_Type_hvector` findet sich im MPI1-Standard [5]. Das Empfangen läuft identisch ab. Der einzige Unterschied besteht bei der Berechnung der Strides, mit denen die Daten eingetragen müssen. So haben zwei aufeinanderfolgende Elemente der Dimension d vor dem Transponieren einen Stride von $\prod_{i=0}^{d-1} N_d$. Beim Transponieren müssen diese aber mit einem Stride von $\prod_{i=0}^{\pi(d-1)} N_{\pi(d)}$ in das neue Array eingetragen werden.

Der eigentliche Datenaustausch ist ein Aufruf von `MPI_Alltoallw` mit den vorbereiteten Datentypen und entsprechend berechneten Displacements. Die genaue Syntax von `MPI_Alltoallw` kann im MPI2-Standard [6] nachgelesen werden.

In den Abbildungen 7 bis 10 wird das Schachteln der Datenstrukturen `MPI_Type_hvector` mit dem Beispiel von oben und einem Senden von Prozessor 0 zu sich selbst verdeutlicht.

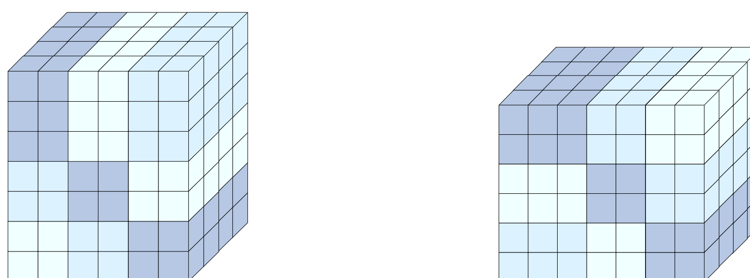


Abbildung 7: Das zu sendende $6 \times 4 \times 7$-Array A (links). Das $7 \times 4 \times 6$-Array $\pi(A)$ (rechts)

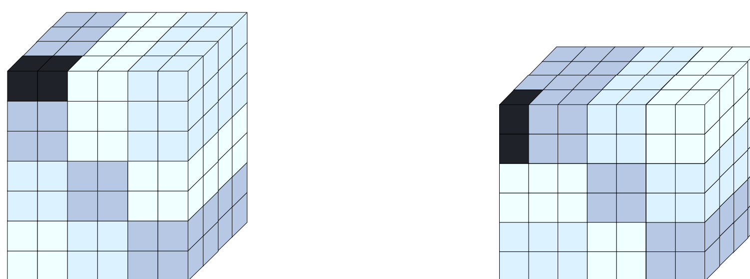


Abbildung 8: Erstellen eines neuen Datentyps bestehend aus 2 komplexe Zahlen mit einem Stride von 1 für das Versenden (links). Erstellen eines neuen Datentyps bestehend aus 2 komplexen Zahlen mit einem Stride von $7 \cdot 4 = 28$ für das Empfangen (rechts).

Im Programm müssen die Strides in bytes angegeben werden. Außerdem muss für die FFT-Routinen von Stefan Goedecker berücksichtigt werden, welche Dimension die zuletzt ausgeführte FFT hatte und welche Dimension die nächste durchzuführende FFT haben wird. Entsprechend müssen die Workarrays bei der Berechnung der Strides berücksichtigt werden.

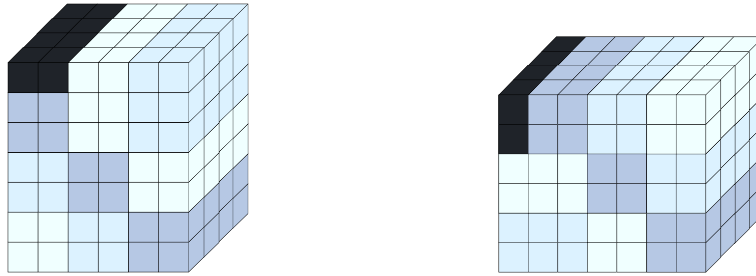


Abbildung 9: Erstellen eines neuen Datentyps bestehend aus 4 Elementen des vorherigen Typs mit einem Stride von 6 für das Versenden (links). Erstellen eines neuen Datentyps bestehend aus 4 Elementen des vorherigen Typs mit einem Stride von 7 für das Empfangen (rechts).

Schedule

Das im vorigen Abschnitt beschriebene Transponieren sowie das Durchführen von FFTs und das Erzeugen von Testfällen lässt sich im Testprogramm mittels eines Input-Files steuern. Das folgende Beispiel-Input-File erzeugt einen Testfall wie er in den bisherigen Beispielen verwendet wurde.

```
3       % 3 Dimensionen
0       % 0=komplexe Testdaten, 1=reelle Testdaten
7 0     % 7 Elemente in x-Richtung
4 0     % 4 Elemente in y-Richtung
6 0     % 6 Elemente in z-Richtung
fgk     % wende alle 3 Bibliotheken an
4       % 4 Aufgaben in der Event-Schedule
f 1 1   % 1.: eindimensionale FFT durchf\"uhren
s 1 2   % 2.: Transponiere eine Dimension beginnend mit Nr. 2 nach 0
f 2 1   % 3.: zweidimensionale FFT durchf\"uhren
s 1 2   % 4.: Transponiere eine Dimension beginnend mit Nr. 2 nach 0
```

Das Transponieren und Durchführen von FFTs erfolgt in einer Event-Schedule, so dass man z.B. eine dreidimensionale FFT durch dreimalige Anwendung einer eindimensionalen FFT oder durch Anwendung einer eindimensionalen und einer zweidimensionalen FFT erreichen kann.

Resultate

Die folgenden Testläufe fanden auf JUMP statt. Dazu wurden jeweils 10 Testläufe durchgeführt und derjenige mit minimaler Laufzeit angegeben. Bei den erzeugten Testfällen handelt es sich um relle Daten, welche in Richtung der ersten Dimension eine gerade Symmetrie aufweisen. Untersucht wurden zunächst Reihenfolgen von FFTs verschiedener Dimension und zwischenzeitlicher Transponierung der Daten. “1,1,2” bedeutet damit also:

- führe eine eindimensionale FFT auf allen lokalen Daten durch
- transponiere mittels $t(\pi)$ mit $\pi = (01)(2)(3)$
- führe eine eindimensionale FFT auf allen lokalen Daten durch
- transponiere mittels $t(\pi)$ mit $\pi = (02)(13)$

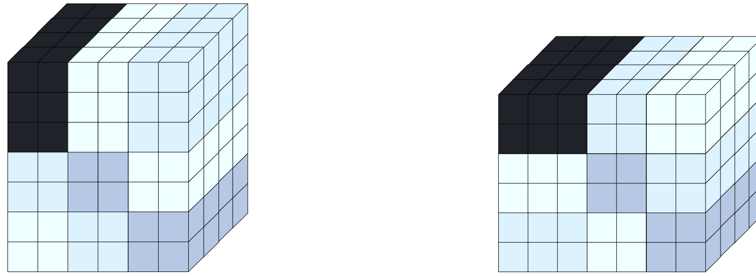


Abbildung 10: Erstellen eines neuen Datentyps bestehend aus 3 Elementen des vorherigen Typs mit einem Stride von $6 \cdot 4 = 24$ für das Versenden (links). Erstellen eines neuen Datentyps bestehend aus 3 Elementen des vorherigen Typs mit einem Stride von 1 für das Empfangen (rechts).

- führe eine zweidimensionale FFT auf allen lokalen Daten durch
- transponiere mittels $t(\pi)$ mit $\pi = (02)(13)$
- transponiere mittels $t(\pi)$ mit $\pi = (01)(2)(3)$

Dabei sind die letzten beiden Punkte notwendig, um die ursprüngliche Anordnung der Daten wiederherzustellen. Als weiteres Beispiel "2,2":

- führe eine zweidimensionale FFT auf allen lokalen Daten durch
- transponiere mittels $t(\pi)$ mit $\pi = (02)(13)$
- führe eine zweidimensionale FFT auf allen lokalen Daten durch
- transponiere mittels $t(\pi)$ mit $\pi = (02)(13)$

Für die Planerstellung der FFTW wurde ausschließlich `FFTW_MEASURE` verwendet. Die Zeiten für die Anwendung der FFTs und für das Transponieren sind jeweils akkumuliert über sämtliche durchgeführten FFTs bzw. Transponierungen. Die "L2 load miss rate" berechnet sich nach [7] mittels

L2 Load miss rate = (loads from memory + L3 loads) / (L2 loads + L3 loads + loads from memory)

"L2 load miss rate" und "L2 load traffic" wurden nur für die FFTs untersucht.

Wie man anhand der Daten aus den Abbildungen 11 bis 14 erkennt, beansprucht das Erstellen eines Planes für die FFTW wesentlich mehr Zeit, als für die Durchführung notwendig ist. Benötigt man eine Vielzahl von gleichen FFTs auf verschiedenen Daten, kann man einen erstellten Plan wiederverwenden. Damit kann sich die anfangs benötigte Planungszeit rentieren.

Weiterhin fällt auf, dass das Transponieren bei Kifowit bis zu doppelt so lang dauern kann wie bei den anderen Bibliotheken. Die Ursache dafür ist vermutlich die Aufteilung in zwei separate Felder für den Real- und den Imaginärteil der Daten. Das Datenvolumen ist zwar gleich, aber das Erstellen der MPI-Strukturen zum Versenden und Empfangen erfordert eine zusätzliche Schachtelung von `MPI_Type_hvector`.

Interessant ist, dass die Bibliotheken von Stefan Goedecker nur wenige Prozent langsamer als die FFTW sind und die Routinen von Steve Kifowit so weit abgeschlagen sind. Das notwendigen Transponieren der Daten vor jeder eindimensionalen FFT bei Kifowit sorgt dabei für den hohen "L2 load traffic".

	Laufzeit	FFT-Zeit	Transponieren	L2 load miss rate	L2 load traffic
FFTW	6.768 s	0.210 s	0.659 s	0.044	239.133 MB
Goedecker	0.867 s	0.239 s	0.627 s	0.084	446.696 MB
Kifowit	3.524 s	2.727 s	0.797 s	0.074	1962.653 MB

Abbildung 11: 64^4, reell, gerade, 16 Prozessoren, Reihenfolge der FFTs: “1,1,2”

	Laufzeit	FFT-Zeit	Transponieren	L2 load miss rate	L2 load traffic
FFTW	6.750 s	0.232 s	0.495 s	0.084	304.388 MB
Goedecker	0.714 s	0.261 s	0.453 s	0.064	440.137 MB
Kifowit	5.071 s	4.289 s	0.782 s	0.062	3040.055 MB

Abbildung 12: 64^4, reell, gerade, 16 Prozessoren, Reihenfolge der FFTs: “1,3”

	Laufzeit	FFT-Zeit	Transponieren	L2 load miss rate	L2 load traffic
FFTW	4.603 s	0.222 s	0.260 s	0.042	362.749 MB
Goedecker	0.370 s	0.122 s	0.247 s	0.059	381.047 MB
Kifowit	3.513 s	3.004 s	0.509 s	0.064	2321.990 MB

Abbildung 13: 64^4, reell, gerade, 16 Prozessoren, Reihenfolge der FFTs: “2,2”

	Laufzeit	FFT-Zeit	Transponieren	L2 load miss rate	L2 load traffic
FFTW	7.497 s	0.235 s	0.549 s	0.086	336.932 MB
Goedecker	0.734 s	0.244 s	0.489 s	0.118	472.638 MB
Kifowit	2.948 s	2.146 s	0.802 s	0.062	1292.404 MB

Abbildung 14: 64^4, reell, gerade, 16 Prozessoren, Reihenfolge der FFTs: “3,1”

Die nächsten Testfälle (Abbildungen 15 und 16) untersuchen das Skalierungsverhalten. Dazu wurde eine vierdimensionale FFT der Größe 64^4 auf komplexen Daten ohne Symmetrien durchgeführt. Die Reihenfolge der FFTs ist dabei “2,2”, wie oben erläutert.

Prozessoranzahl	FFTW	Goedecker	Kifowit
1	3.515 s	2.368 s	13.396 s
2	1.764 s	0.924 s	7.870 s
4	0.842 s	0.503 s	5.044 s
8	0.439 s	0.222 s	3.676 s
16	0.213 s	0.103 s	2.966 s
32	0.110 s	0.074 s	2.705 s
64	0.054 s	0.030 s	2.508 s

Abbildung 15: FFT-Zeit für 64^4, komplexe Daten, keine Symmetrie, Reihenfolge der FFTs: “2,2”

Anhand der Daten ist ablesbar, dass sowohl die FFT als auch das Transponieren recht gut skalieren.

Prozessoranzahl	FFTW	Goedecker	Kifowit
1	5.036 s	5.112 s	8.492 s
2	2.606 s	2.606 s	4.343 s
4	1.217 s	1.190 s	2.210 s
8	0.527 s	0.673 s	1.080 s
16	0.266 s	0.321 s	0.461 s
32	0.172 s	0.114 s	0.154 s
64	0.122 s	0.077 s	0.077 s

Abbildung 16: Transponier-Zeit für 64^4, komplexe Daten, keine Symmetrie, Reihenfolge der FFTs: “2,2”

Die letzten dargestellten Testfälle (Abbildungen 17, 18 und 19) untersuchen die Abhängigkeit von der Größe einer Dimension. Zu erwarten ist, dass bei einer größeren Anzahl von Elementen in Richtung einer Dimension, auf der eine FFT durchgeführt wird, vermehrt cache misses auftreten und sich das in der Rechenzeit widerspiegelt. Für die folgenden Testfälle wurden komplexe Daten ohne Symmetrien verwendet. Außerdem kamen 8 Prozessoren zur Anwendung und die Reihenfolge der FFTs ist “2,2”.

Problemgröße	FFTW	Goedecker	Kifowit
$64 \times 64 \times 64 \times 64$	0.415 s	0.200 s	3.851 s
$128 \times 128 \times 32 \times 32$	0.540 s	0.240 s	3.877 s
$256 \times 256 \times 16 \times 16$	0.592 s	0.287 s	3.908 s
$512 \times 512 \times 8 \times 8$	0.686 s	0.431 s	3.975 s

Abbildung 17: FFT-Zeit, komplexe Daten, keine Symmetrie, Reihenfolge der FFTs: “2,2”

Problemgröße	FFTW	Goedecker	Kifowit
$64 \times 64 \times 64 \times 64$	0.508 s	0.819 s	1.348 s
$128 \times 128 \times 32 \times 32$	0.735 s	0.642 s	1.024 s
$256 \times 256 \times 16 \times 16$	0.633 s	0.604 s	0.777 s
$512 \times 512 \times 8 \times 8$	0.526 s	0.596 s	0.657 s

Abbildung 18: Transpose-Zeit, komplexe Daten, keine Symmetrie, Reihenfolge der FFTs: “2,2”

Problemgröße	FFTW	Goedecker	Kifowit
$64 \times 64 \times 64 \times 64$	0.047	0.041	0.071
$128 \times 128 \times 32 \times 32$	0.067	0.072	0.067
$256 \times 256 \times 16 \times 16$	0.107	0.087	0.061
$512 \times 512 \times 8 \times 8$	0.121	0.090	0.064

Abbildung 19: L2 load miss rate, komplexe Daten, keine Symmetrie, Reihenfolge der FFTs: “2,2”

Wie in Abbildung 17 zu erkennen ist, nimmt die Dauer der FFT bei wachsender Größe der ersten beiden Dimensionen zu. Dies könnte durch die in Abbildung 19 zu erkennende erhöhte “L2 load miss rate” verursacht werden. Bei Kifowit überwiegt wiederum das Transponieren der Daten, weshalb kaum ein Ansteigen der Rechenzeit zu beobachten ist.

Problemgröße	FFTW	Goedecker	Kifowit
$64 \times 64 \times 64 \times 64$	724.167 MB	799.974 MB	2357.742 MB
$128 \times 128 \times 32 \times 32$	558.476 MB	684.312 MB	2317.993 MB
$256 \times 256 \times 16 \times 16$	429.611 MB	628.056 MB	2086.460 MB
$512 \times 512 \times 8 \times 8$	635.984 MB	1463.432 MB	1955.363 MB

Abbildung 20: L2 load traffic, komplexe Daten, keine Symmetrie, Reihenfolge der FFTs: “2,2”

Danksagung

An dieser Stelle bedanke ich mich bei Dr. Bernhard Steffen für die Betreuung während des Gaststudentenprogramms und bei Matthias Bolten für die ausgezeichnete Organisation. Außerdem möchte ich mich bei Prof. Dr. Bruno Lang für das Befürwortungsschreiben bedanken. Auch an die anderen Gaststudenten seien Worte des Dankes gerichtet für die nette Zeit in Jülich. Schließlich danke ich noch dem Forschungszentrum Jülich, dem ZAM/JSC und dem Förderverein, die dieses Gaststudentenprogramm ermöglicht haben.

Literatur

1. FFTW-Bibliothek, Sourcecode und Handbuch
   http://www.fftw.org/
2. S. Goedecker: Fast radix 2,3,4 and 5 kernels for Fast Fourier
   Transformations on computers with overlapping multiply-add instructions,
   SIAM Journal on Scientific Computing 18, 1605 (1997)
3. Software-Homepage von Stefan Goedecker
   http://pages.unibas.ch/comphys/comphys/SOFTWARE/
4. Software-Homepage von Steve Kifowit
   http://faculty.prairiestate.edu/skifowit/fft/
5. Message Passing Interface Forum,
   MPI: A Message-Passing Interface Standard (2003).
6. Message Passing Interface Forum,
   MPI-2: Extensions to the Message-Passing Interface (2003).
7. Dokumentation des Hardware Performance Monitor
   http://domino.research.ibm.com/comm/research_projects.nsf/pages/actc.hardwareperf2.html

Zur Druckberechnung in der parallelen Version des NIST Fire Dynamics Simulators v5.0

Andreas Brätz

Bergische Universität Wuppertal
Fachbereich D - Abt. Sicherheitstechnik
Gaußstraße 20, 42119 Wuppertal

E-mail: andreas.braetz@insa4.de

Zusammenfassung: Die Verwendung rechnerischer Nachweise im Bereich des Brandschutzingenieurwesens hat in den letzten Jahren in Deutschland immer mehr an Gewicht gewonnen. Für komplexe Problemstellungen, beispielsweise der Projektierung von Entrauchungsanlagen zur Rauchfreihaltung von Rettungswegen, ist zunehmend auch der Einsatz von CFD-Berechnungen (**C**omputational **F**luid **D**ynamics) die Regel. Ein oft eingesetztes Programm stellt hier der vom National Institute for Standards and Technology in Gaithersburg entwickelte Fire Dynamics Simulator [2] dar. Mit der Komplexität der betrachteten Szenarien steigt auch der Bedarf an Rechenleistung, so dass auch die Nutzung leistungsfähiger Parallelrechner, wie sie bspw. am JSC, dem **J**uelich **S**upercomputing **C**entre, zur Verfügung stehen, verstärkt ins Blickfeld der Ingenieure rückt. Der Beitrag beschreibt die Probleme bei der Nutzung der aktuellen FDS-Version 5.0 auf massiv parallelen Systemen und versucht Lösungsansätze vorzuzeichnen.

Ingenieurmethoden im Brandschutz

Überblick

In den letzten Jahren werden im baulichen Brandschutz immer häufiger Verfahren verwendet, die sich unter den Sammelbegriff der Ingenieurmethoden im Brandschutz zusammenfassen lassen. Dabei werden im Rahmen der Erstellung von Brandschutzkonzepten vermehrt rechnerische Nachweise genutzt, die an die Stelle explizit formulierter Anforderungen beispielsweise aus den Bauordnungen der Länder oder aus Sonderbauvorschriften treten. Teilweise sind für komplexe Bauvorhaben auch keine konkreten, die Sicherheit im Brandfall behandelnden Regelwerke vorhanden bzw. geben diese keine dezidierten Gestaltungshinweise, etwa im Hinblick auf die Rauchfreihaltung von Rettungswegen, so dass hier nur eine gewissenhafte, gesamtheitliche Risikobetrachtung unter Einbeziehung von Ingenieurverfahren ein hohes Sicherheitsniveau garantiert. Zum Einsatz kommen dabei:

- Makroskopische Verfahren
- Zonenmodellberechnungen
- Finite-Elemente-Programme für thermische Bauteilanalysen
- Programmpakete zur dynamischen Evakuierungssimulation
- CFD-Programme

Die zur Verfügung stehenden Methoden unterscheiden sich hinsichtlich ihrer Komplexität und ihrer Anwendungsmöglichkeiten. Sind etwa die Zonenmodellberechnungen zur Bestimmung von Rauchgastemperaturen in einem Raum auf die Abbildung einfacher Räume beschränkt, können CFD-Anwendungen hier auch die Dynamik der Rauch- und Brandausbreitung in deutlich komplizierteren Geometrien erfassen. Ihnen kommt daher unter den Ingenieurverfahren im Brandschutz eine zentrale Rolle zu, da sie sich so für zahlreiche Fragestellungen eignen. Darunter beispielsweise:

- Sichtweitenabschätzungen
- Beurteilung der Rauchausbreitung
- Ermittlung von Temperatur-Zeit-Verläufen
- Simulation des Brandverhaltens von Baustoffen
- Untersuchungen zum Auslöseverhalten von Systemen zur Branddetektion und -bekämpfung
- Vorbereitung thermischer Bauteilanalysen

Entsprechend ihrer Vielseitigkeit stehen derzeit auf dem Markt zahlreiche, zumeist kommerzielle Programme zur Verfügung. Eine open source-Alternative bietet sich im NIST Fire Dynamics Simulator an.

NIST Fire Dynamics Simulator v5.0

Der Fire Dynamics Simulator, kurz FDS, ist ein CFD-Programm zur Berechnung brandinduzierter Strömungen, wobei der Transport von Wärme und Rauch von besonderem Interesse ist. In FDS wird mittels numerischer Verfahren, eine spezielle Form der Navier-Stokes-Gleichungen gelöst, die für auftriebsbehaftete Strömungen geringer Geschwindigkeiten (Low Mach Number assumptions) geeignet ist. Die Formulierung dieser Gleichungen und die zur Anwendung kommenden numerischen Algorithmen sind ausführlich im FDS Technical Reference Guide [1] dokumentiert. Neben den Algorithmen zu strömungstechnischen Berechnungen sind in FDS weitere Submodelle implementiert, welche die Aspekte der Brandsimulation besonders berücksichtigen. Bestes Beispiel hierfür ist das sogenannte Mixture-Fraction-Modell für die Abbildung der Verbrennungsreaktionen. Diese Submodelle grenzen FDS von vielen kommerziellen CFD-Programmen ab, deren originärer Zweck eher in der Strömungsberechnung als in der Brandsimulation liegt. Konzipiert wurde FDS vom National Institute for Standards and Technology in Gaithersburg, MD, wo es auch ständig weiterentwickelt wird. Die erste Version des Programms wurde im Jahr 2000 als public domain-Software zur Verfügung gestellt. Die Quellcodes sind auch in der aktuellen Version weiter frei verfügbar. An der Verbesserung des Programmpakets beteiligen sich daher auch zahlreiche internationale Forschungseinrichtungen, Ingenieurbüros und Universitäten, darunter die Bergische Universität in Wuppertal. Aktuell besteht Forschungsbedarf vor allem auf den folgenden Feldern:

- Turbulenzmodellierung mittels Large Eddy Simulation
- Abhängigkeit der Resultate von der Gitterauflösung
- Verwendung strukturierter oder hybrider Netze
- Parallelität

Um mit den Simulationen realistische Ergebnisse erzielen zu können, sind gewisse Ansprüche an die Feinheit der Diskretisierung gestellt. Die Kantenlängen der Gitterzellen liegen gewöhnlich in einem Bereich zwischen 5 und 20 cm. Bei den gewöhnlichen Dimensionen von Gebäuden müssen so abhängig

von der Problemstellung oft mehrere Millionen Zellen berechnet werden. Üblicherweise stehen hierfür in den Ingenieurbüros Cluster aus Arbeitsplatzrechnern zur Verfügung. Damit sind Rechendauern von mehreren Tagen für die Simulation von beispielsweise 20 Minuten Brandverlauf bei bis zu 10 Millionen Zellen die Regel. Der Einsatz massiv paralleler Systeme, wie sie am JSC zur Verfügung stehen, kann hier im Hinblick auf eine Minimierung der Rechenzeit deutliche Vorteile bringen. Insbesondere, wenn die Anzahl der Gitterzellen weiter erhöht werden muss. Als Beispiel[1] sei hier eine Simulation der Rauchausbreitung in einem U-Bahnhof angeführt. Durch die großen Abmaße der Verkehrsanlage und einer notwendigerweise sehr feinen Diskretisierung im Inneren des Zuges und im Nahbereich um das Schienenfahrzeug lag die Zellenanzahl bei ca. 10^8. Die auf JUMP[2] auf 32 CPUs durchgeführte Berechnung benötigte dafür ca. 48 Stunden. Welche Grenzen dem Einsatz auf solchen massiv parallelen Systemen eventuell gesetzt sind, bzw. worauf geachtet werden muss, soll im weiteren näher beleuchtet werden. Dazu gilt es zunächst, sich mit dem derzeitigen Konzept der Parallelisierung von FDS vertraut zu machen.

Parallelisierung in FDS v5.0

Um Berechnungen auf mehreren Prozessoren durchzuführen, nutzt FDS den sogenannten Multi-Mesh-Ansatz. Dabei wird die gesamte Domain in kleinere Meshes zerteilt. Von Vorteil ist hier, dass sich die Gitterauflösungen von Mesh zu Mesh unterscheiden können, so dass es ermöglicht wird, an entscheidenden Stellen, zum Beispiel im Brandnahbereich, feiner zu diskretisieren. Die Abbildungen 1 und 2 veranschaulichen dies anhand einer Aufteilung, die jedem der beiden Räume ein eigenes Mesh zuordnet.

Multi Mesh

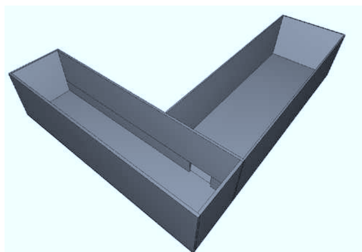


Abbildung 1: Einfache Geometrie aus zwei angrenzenden Räumen

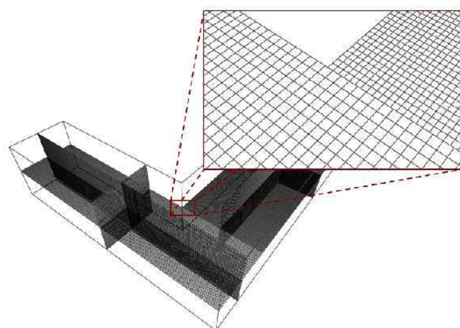


Abbildung 2: Diskretisierung der Geometrie, Ausschnitt zeigt die unterschiedlich feine Diskretisierung der einzelnen Meshes

[1] siehe hierzu auch: http://www.fz-juelich.de/jsc/appliedmath/fire-simulations

[2] **ju**elich **m**ulti **p**rocessor, Supercomputer IBM p690-Cluster

Entscheidend ist nun, dass die Größen, wie die Druckverteilung und die Geschwindigkeitsfelder, nicht mehr für die gesamte Domain gelöst werden, sondern für jedes einzelne Mesh unabhängig. An den Gittergrenzen findet dann eine Übergabe der Werte statt, die für die Berechnungen im Nachbargitter benötigt werden. Wichtig ist weiterhin, dass diese Werte zunächst approximiert werden. Das Prozedere sei am Beispiel der Druckberechnung näher erläutert. Es können sowohl alle Meshes auf einem Prozessor berechnet werden oder aber der Rechenaufwand meshweise auf mehrere Prozessoren verteilt werden. Sollen mehrere Prozessoren mit einbezogen werden, wird die Kommunikation zwischen den Prozessen mit MPI-Aufrufen[3] geregelt.

Poisson-Gleichung zur Druckberechnung

Der Druck wird in FDS über die Hilfsgröße H berechnet. Aus der Erhaltungsgleichung für den Impuls

$$\frac{\partial u}{\partial t} + F + \nabla H = 0\,; \quad H = \frac{|u|}{2} + \frac{\tilde{p}}{\rho} \tag{1}$$

mit dem Geschwindigkeitsvektor u, dem Konvektions-/Diffusions-Term F, der Dichte ρ und dem als perturbation pressure bezeichneten $\tilde{p}$, folgt nach Umformungen eine einzelne, elliptische, partielle Differentialgleichung für den modifizierten Druck H, die als Poisson-Gleichung bekannt ist:

$$\nabla H = -\nabla \cdot F - \frac{\partial}{\partial t}\left(\nabla \cdot u\right) \tag{2}$$

Bzw. in der diskretisierten Form:

$$\begin{aligned}\frac{H_{i+1,jk} - 2H_{ijk} + H_{i-1,jk}}{\partial x} &+ \frac{H_{i,j+1,k} - 2H_{ijk} + H_{i,j-1,k}}{\partial y} + \frac{H_{ij,k+1} - 2H_{ijk} + H_{ij,k-1}}{\partial z} \\ &= -\frac{F_{x,ijk} - F_{x,i-1,jk}}{\partial x} - \frac{F_{y,ijk} - F_{y,i,j-1,k}}{\partial y} - \frac{F_{z,ijk} - F_{z,ij,k-1}}{\partial z} - \frac{\partial}{\partial t}\left(\nabla \cdot u\right)_{ijk}\end{aligned} \tag{3}$$

Alle in der Formel vorkommenden Größen sind für denselben Zeitschritt zu ermitteln. Zur Lösung der Gleichung kommt ein direkter (nicht iterativer), auf der FFT[4] basierender Löser zum Einsatz, der Teil der CRAYFISHPAK-Bibliothek [4] ist. Diese Sammlung von Routinen wurde ursprünglich vom National Center for Atmospheric Research (NCAR) in Boulder, Colorado entwickelt und ist speziell auf die Lösung elliptischer, partieller Differentialgleichungen ausgerichtet.

Die Definition von Randbedingungen erfolgt in Abhängigkeit von den Eigenschaften der Randzellen. So werden beispielsweise für Gitterzellen an Wänden andere Randbedingungen formuliert als an sog. open boundaries. Für die Frage der Parallelisierung der Anwendung sind diese jedoch von untergeordneter Bedeutung. Von wesentlichem Interesse ist die Formulierung der Randbedingungen an den Mesh Interfaces.

Druckberechnung in der Multi-Mesh-Umgebung

Zur Bestimmung der Randbedingungen findet zunächst eine Aufteilung des Druckterms statt:
$H = H^0 + H'$. Die Gleichung

[3] **M**essage **P**assing **I**nterface
[4] **F**ast **F**ourier **T**ransformation

$$\nabla H^0 = -\nabla \cdot F - \frac{\partial}{\partial t}(\nabla \cdot u) \tag{4}$$

wird anschließend für jedes Mesh einzeln gelöst. So hat jedes Mesh zu diesem Zeitpunkt sein eigenes Druckfeld, wenn allerdings die Geschwindigkeiten in jedem Mesh neu berechnet werden, passen die Normalkomponenten der Geschwindigkeit an den Gittergrenzen nicht mehr zueinander. Weiterhin muss sichergestellt werden, dass der Volumenstrom $\dot{V} = \int u \cdot dS$ an den Gittergrenzen von Mesh zu Mesh übereinstimmt. Dieser Volumenstrom lässt sich nicht lokal bestimmen, sodass zur Berechnung eine alternative Methode gewählt wurde. Die ursprüngliche Poisson-Gleichung wird dabei auf einem groben Gitter neu diskretisiert.

$$\int_{\partial\Omega_m} \nabla H \cdot dS = -\int_{\partial\Omega_m} F \cdot dS - \int_{\Omega_m} \frac{\partial(\nabla \cdot u)}{\partial t} dV \tag{5}$$

Der Index m bezeichnet die Zellen des groben Meshes. Nun wird eine skalare Größe $\bar{H}_m$ für jede einzelne Gitterzelle des groben Gitters eingeführt, die der folgenden Bedingung genügt:

$$\frac{\bar{H}_{m^+} - \bar{H}_m}{\delta x} A_{x^+} - \frac{\bar{H}_m - \bar{H}_{m^-}}{\delta x} A_{x^-} + ... = -\int_{x^+} F_x dydz + \int_{x^-} F_x dydz + ... - \int_{\Omega_m} \frac{\partial(\nabla \cdot u)}{\partial t} dV \tag{6}$$

Die Indizes m^+ und m^- bezeichnen zwei benachbarte Zellen im groben Gitter. Mit A_x^+ und A_x^- sind die Grenzflächen dieser Gitterzellen bezeichnet. An den Grenzflächen zweier benachbarter Meshes, sagen wir hier in x-Richtung, existieren nun zwei Werte für F_x. Der Mittelwert aus beiden Werten $\bar{F}_x$ wird in die Poisson-Gleichung für das grobe Gitter eingesetzt und die Gleichung anschließend gelöst. Damit erhält man einen Schätzwert für die zeitliche Veränderung des Volumenstroms. Beispielhaft für die x-Richtung:

$$\left.\frac{d\dot{V}}{dt}\right|_{x^+} = -\int_{x^+} \bar{F}_x dydz - \frac{\bar{H}_{m^+} - \bar{H}_m}{\delta x} A_{x^+} \tag{7}$$

Diese Bedingung soll nun an den Grenzflächen zweier benachbarter Grids auch für die Lösung des feinen Grids erfüllt sein. Dazu wird die Gleichung

$$\nabla H' = 0 \tag{8}$$

mit Neumann-Randbedingungen gelöst, so dass an jeder Schnittstelle zweier Meshes der Gradient von H' konstant ist. So muss der Gradient von H' an der x^+-Schnittstelle der folgenden Bedingung genügen:

$$\left.\frac{d\dot{V}}{dt}\right|_{x^+} = \int_{x^+} \left(F_{x,Ijk} + \frac{H^0_{I+1,jk} - H^0_{Ijk}}{\delta x} \right) dydz + \int_{x^+} \frac{dH'}{\partial x} dydz \tag{9}$$

Die gleiche Bedingung muss beim rechten Nachbarmesh erfüllt sein. Auf diesem Wege stimmen die resultierenden Volumenströme ins Mesh und aus dem Mesh überein. Was allerdings nicht garantiert, dass

die Normalkomponenten der Geschwindigkeiten in jeder einzelnen Zelle dem Wert ihres Gegenparts im Nachbarmesh entsprechen.

Die Frage ist nun, welche Werte für $\nabla H'$ in jeder Zelle an den Gittergrenzen zu wählen sind. Hierzu wurden vom NIST diverse Möglichkeiten durchexerziert, die jedoch bis dato noch zu keiner allgemeingültigen und robusten Lösung führten. Die Auswirkungen dieser Unstimmigkeiten in der Parallelisierung sind für zwei unterschiedliche Gitterkonfigurationen im nächsten Abschnitt beschrieben.

Probleme mit dem aktuellen Parallelisierungsansatz

Im ersten Beispiel soll eine Domain mit einer Grundfläche von 40 x 2 m, einer Höhe von 5 m und einer Diskretisierung in 400.000 Zellen mit einem Mesh und mit fünf Meshes gerechnet werden. Den qualitativen Vergleich der berechneten Druckverteilung in beiden Simulationen zeigt Abbildung 3.

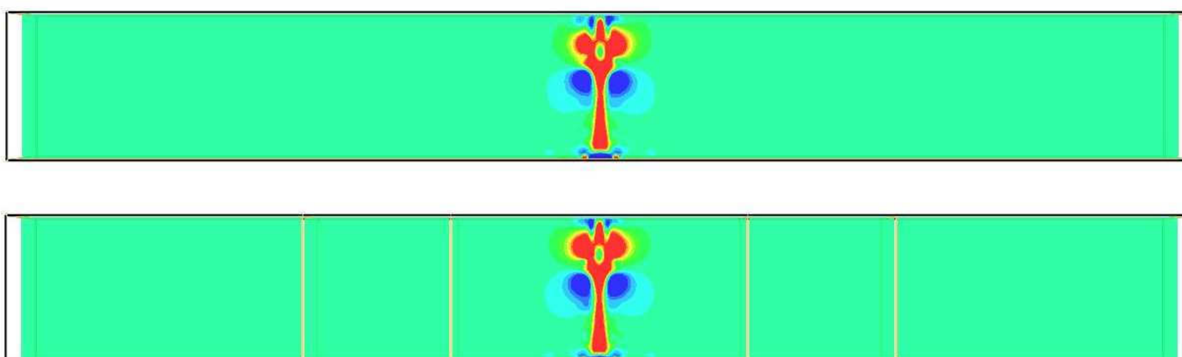


Abbildung 3: Unkritische Mesh-Anordnung für die Parallelisierung, 1 Mesh oben, 5 Meshes unten

Zwar lassen sich Abweichungen erkennen, diese sind aber für die Bewertung der Ergebnisse durch die Ingenieure vernachlässigbar gering.

Eine gänzlich andere Situation stellt sich für das zweite Beispiel dar, bei dem eine Domain mit quadratischer Grundfläche (3 x 3 m) und einer Höhe von 10 m in 20 Meshes aufgeteilt werden soll. Die Zellenanzahl beträgt jeweils 90.000 Zellen. Überdeutlich ist aus Abbildung 4 zu erkennen, dass die Abweichungen hier definitiv nicht mehr toleriert werden können. Es kommt zu deutlichen, qualitativen Änderungen im Strömungsbild, so dass bei sonst gleichbleibenden äßeren Bedingungen ein Übergang von laminarer zu turbulenter Strömung allein durch Einführung von Multimeshes beobachtet werden kann.

Alle Multi-Mesh-Rechnungen wurden auf einem Prozessor durchgeführt, sodass Fehler aus einer inkorrekten MPI-Implementierung ausgeschlossen werden können.

Mögliche Ursachen und Lösungsansätze

Als mögliche Ursachen wurden die folgenden Probleme formuliert:

- Güte der Berechnung durch die Struktur des Parallelisierungsansatzes lässt mit steigender Gitteranzahl nach

- Behandlung der Gittergrenzen durch Randbedingungen ist fehlerhaft und nicht für alle Anwendungsfälle ausreichend genau

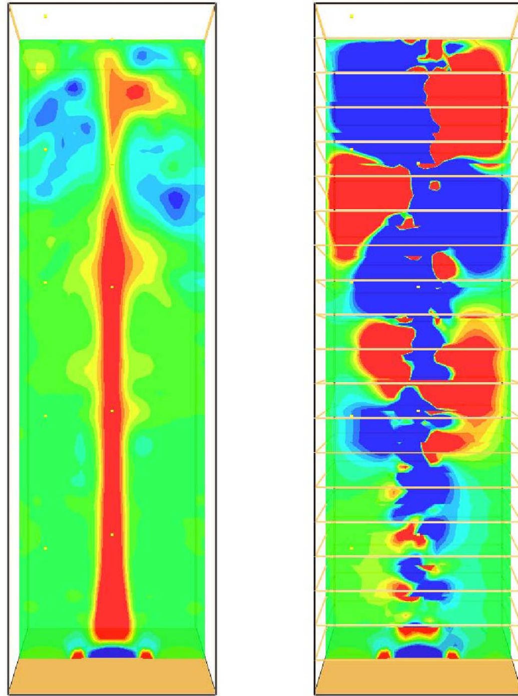


Abbildung 4: Kritische Mesh-Anordnung für die Parallelisierung

Der Vergleich der Norm des Residuums für die Druckgleichung im Single Mesh und im Multi Mesh zeigt, dass die erste Vermutung richtig zu sein scheint.

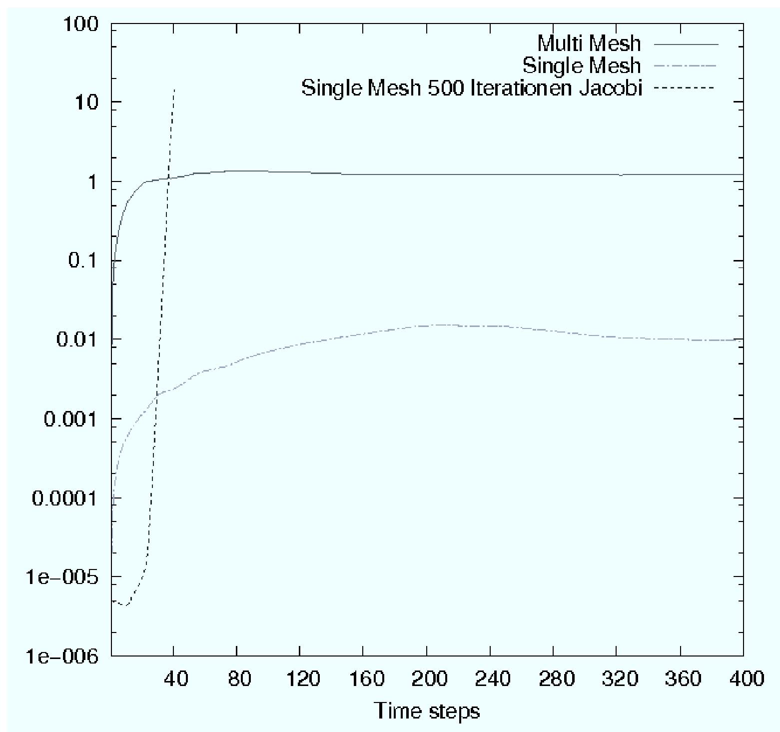


Abbildung 5: Vergleich der Norm des Residuums für die Berechnung von Gleichung 4 mit der Originalversion des Programms auf einem bzw. zwanzig Meshes und der Version mit dem implementierten Jacobi-Löser mit 500 Iterationen

Um dem zu begegnen wurde daher zunächst mit dem Jacobi-Algorithmus ein iterativer Löser in FDS integriert, der eine Erhöhung der Berechnungsgenauigkeit erreichen sollte. Als Startwert wurde das Ergebnis für H aus dem Crayfishpak-Solver genutzt. Weiterhin wurde der originäre Austausch interpolierter Randwerte unterbunden und durch die zuvor direkt berechneten Werte aus den Nachbarzellen ersetzt. Wie aus Abbildung 5 ersichtlich ist, konnte das Residuum anfänglich deutlich vermindert werden. Jedoch liefen alle Simulationen mit dem eingebauten Jacobi-Solver früher oder später in eine numerische Instabilität, sodass befürchtet werden muss, dass in der Ungenauigkeit des Parallelisierungsansatzes nicht das einzige Problem besteht. Die Einbindung des iterativen Lösers an unterschiedlichen Stellen im Quellcode, teilweise innerhalb, teilweise außerhalb diverser Subroutinen, zeigte hier zwar Erfolge dergestalt, dass sich die Anzahl durchgeführter Iterationsschritte bis zum Eintreten der numerischen Instabilität von Simulationsdauern von 0.5 s auf stolze 10.8 s nach hinten verschieben ließ, der Programmabbruch blieb jedoch unabwendbar. Kurioserweise war hier zwar mit mehr Iterationsschritten im Jacobi-Algorithmus eine längere Laufzeit des Programms zu erreichen, die Implementierung des wesentlich schneller konvergierenden CG-Verfahrens[5] an gleicher Stelle im Quellcode führte dahingegen bereits nach 0.6 s Simulationsdauer zu einer numerischen Instabilität. Kurzum ist der Parallelisierungsansatz von Grund auf neu zu überdenken, statt am bisherigen Konzept mit interpolierten Randwerten an den Mesh-Interfaces festzuhalten.

Schlussfolgerung

Grundsätzlich zeigte sich, dass sich FDS durchaus auf parallelen Systemen einsetzen lässt und dabei realistische Ergebnisse liefern kann. Dem Anwender obliegt aber hier die Pflicht, die Gittergrenzen bewusst zu wählen. Eine Sensitivitätsanalyse hinsichtlich der Auswirkungen der gewählten Mesh-Anordnung ist dringend zu empfehlen. Im Zweifel muss auf die Reduzierung der Rechenzeit zugunsten höherer Genauigkeit verzichtet werden. Ebenso muss unter Umständen ein gewisses Maß an load imbalance, also eine ungleichen Auslastung der involvierten Prozessoren, akzeptiert werden, wenn weitere Unterteilungen zu sehr zu Ungunsten der Genauigkeit wären. Die Misserfolge der geprüften Lösungsansätze machen deutlich, dass es einer grundlegenden Neustrukturierung der Parallelisierung dringend bedarf. Die klare Formulierung im User's Guide, dass auf diesem Gebiet derzeit intensiv geforscht wird, lässt hoffen, dass in naher Zukunft die Parallelisierung von FDS so implementiert sein wird, dass einer Nutzung auf massiv parallelen Systemen nichts mehr im Wege steht. Aufgrund der Komplexität des Quellcodes mit einer Länge von rund 15.000 Zeilen, der exzessiven Nutzung von Pointern und Strukturen, sowie einer Vielzahl ineinander verschachtelter Subroutinen war es in der begrenzten Zeit, die während des Gaststudentenprogramms zur Verfügung stand, nicht möglich, den Problemen durch den aktuellen Parallelisierungsansatz zu entgegnen. Vielmehr wurden Wege skizziert, wie sie umschifft werden können, indem Bereiche mit starken Gradienten in Geschwindigkeitsfeldern und großen Druckgefällen nicht durch Gittergrenzen geschnitten werden. Dafür, dass ich mich mit diesen Anwendungsgrenzen, deren Kenntnis ja für die Arbeit des Ingenieurs eminent wichtig ist, im Rahmen des Gaststudenprogramms eingehend beschäftigen konnte, möchte ich mich ganz herzlich bedanken. Mein Dank gilt hier in erster Linie meinem Betreuer, Herrn Dr. Armin Seyfried, der mir bei Problemen stets sein Ohr lieh und natürlich Matthias Bolten für die vorbildliche Organisation. Weiterhin vielen Dank an die gesamte Mannschaft des JSC und alle die mir die Teilnahme am Gaststudentenprogramm ermöglichten, sowie selbstverständlich an die anderen Gaststudenten für fachliche und freizeittechnische Unterstützung.

Literatur

1. K.B. McGrattan, S. Hostikka, J.E. Floyd, H.R. Baum, and R.G. Rehm. Fire Dynamics Simulator (Version 5), Technical Reference Guide. NIST Special Publication 1018-5, National Institute for Standards and Technology, Gaithersburg, Maryland, October 2007.

[5]engl. **C**onjugate **G**radients oder auch Verfahren der konjugierten Gradienten, siehe auch [3]

2. K.B. McGrattan, B.W. Klein, S. Hostikka, and J.E. Floyd. Fire Dynamics Simulator (Version 5), User's Guide, NIST Special Publication 1019-5, National Institute of Standards and Technology, Gaithersburg, Maryland, October 2007.
3. R. Schaback, H. Wendland. Numerische Mathematik. Springer Verlag Berlin Heidelberg.2005
4. The University Corporation for Atmospheric Research, CRAYFISH, http://www.cisl.ucar.edu/softlib/CRAYFISH.html

Algebraische Mehrgitterverfahren für strukturierte Matrizen

Stephanie Friedhoff

Bergische Universität Wuppertal
Fachbereich C - Mathematik und Naturwissenschaften
Gaußstraße 20, 42119 Wuppertal

E-Mail: sfriedho@studs.math.uni-wuppertal.de

Zusammenfassung:
Zunächst werden die Grundlagen von algebraischen Mehrgitterverfahren (AMG) für den allgemeinen Fall beschrieben. Dabei wird auf das Problem der wachsenden Komplexität des verwendeten Galerkin-Operators aufmerksam gemacht. Anschließend wird untersucht, wie für die spezielle Klasse der Toeplitz- und zirkulanten Bandmatrizen ein algebraisches Mehrgitterverfahren optimiert werden kann. Dabei wird der Begriff der Spektraläquivalenz erklärt und diskutiert, wie diese zur Optimierung verwendet werden kann. Auf Veränderungen des Verfahrens, die dadurch zusätzlich nötig sind, wird im Anschluss eingegangen. Desweiteren wird die Verwendung algebraischer Mehrgitterverfahren als Präkonditionierer für ein CG-Verfahren beschrieben und untersucht. Zum Schluss werden Ergebnisse einer parallelen Implementierung vorgestellt und diskutiert. Dabei werden für den vorliegenden Fall AMG, optimiertes AMG, mit AMG präkonditioniertes CG und mit optimiertem AMG präkonditioniertes CG insbesondere im Hinblick auf Konvergenz und Skalierbarkeit verglichen.

Einleitung

Algebraische Mehrgitterverfahren haben zahlreiche Anwendungsgebiete. Dazu zählen z. B. das Lösen von partiellen Differentialgleichungen, Integralgleichungen sowie Gleichungssystemen der Kontrolltheorie. In diesem Bericht wird das Ziel verfolgt, eine Blackbox für die spezielle Klasse der Toeplitz- und zirkulanten Bandmatrizen zu entwickeln.

Algebraische Mehrgitterverfahren für strukturierte Matrizen wurden im Laufe der letzten Jahre entwickelt ([2], [3]). Können Matrizen durch ein trigonometrisches Polynom beschrieben werden, d. h. unabhängig von der Matrixdimension ist eine Beschreibung durch einen Differenzenstern fester Größe möglich, so sind Mehrgitterverfahren optimale ($\mathcal{O}(n)$) Löser. Sie stellen somit eine Alternative zu FFT-Verfahren, die eine Komplexität von $\mathcal{O}(n \log n)$ haben.

Grundlage dieses Berichts ist die Theorie algebraischer Mehrgitterverfahren von Stüben [1], die im folgenden Kapitel kurz beschrieben wird. Nachdem der Galerkin-Operator und Coarse-Grid-Correction-Operator eingeführt wurden und die Rolle der Glättung für algebraische Mehrgitterverfahren geklärt wurde, wird auf das Problem der wachsenden Komplexität des verwendeten Galerkin-Operators eingegangen.

Im nächsten Abschnitt werden Toeplitz- und zirkulante Bandmatrizen zunächst definiert. Entscheidend ist dabei, dass zirkulante Matrizen durch ihr generierendes Symbol beschrieben werden können. Diese Eigenschaft hat zur Folge, dass die komplexe Gittervergröberug zu einer einfachen Dimensionsveränderung und Definition eines geeigneten Interpolationsoperators reduziert werden kann. Als Grobgitter-

operator wird dabei der Galerkin-Operator verwendet. Dieser führt allerdings zu dem Problem, dass die Komplexität eines dünnbesetzten Differenzensterns mit jedem Level des Mehrgitterverfahrens zunimmt. Um dieses Problem zu lösen, wird basierend auf dem Artikel von Bolten und Frommer [4] an Stelle des vollbesetzten Sterns ein dünnbesetzter Stern verwendet, dessen zugehörige Matrix spektraläquivalent zur ursprünglichen Matrix ist.

Bevor auf die Verwendung von algebraischen Mehrgitterverfahren als Präkonditionierer für das CG-Verfahren eingegangen wird, werden mögliche Optimierungsparameter eines algebraischen Mehrgitterverfahrens beschrieben. In diesem Kapitel wird ein optimaler Relaxationsparameter des relaxierten Jacobi-Verfahrens bestimmt. Außerdem wird der für Mehrgitterverfahren, die spektraläquivalente Sterne verwenden, zusätzlich nötige Verschiebungsparameter ermittelt.

Zum Schluss dieses Berichtes werden Ergebnisse eines Testbeispiels präsentiert, das auf dem Supercomputer IBM Blue Gene/L JUBL getestet wurde. Dabei werden Mehrgitterverfahren mit Galerkin-Operator und spektraläquivalentem Operator sowie ein mit diesen beiden Varianten präkonditioniertes CG-Verfahren miteinander verglichen. Der Vergleich erfolgt im Hinblick auf Konvergenz, Skalierbarkeit, Speedup und Effizienz.

Algebraische Mehrgitterverfahren

Es soll das Gleichungssystem

$$Ax = b \qquad A \in \mathbb{C}^{n\times n} \quad x, b \in \mathbb{C}^n$$

gelöst werden. Dazu wird das LGS modifiziert zu

$$Ae = r \qquad A \in \mathbb{C}^{n\times n} \quad e, r \in \mathbb{C}^n \tag{1}$$

wobei x^0 ein beliebiger Startvektor, $e = x^* - x^0$ der Fehler und $r = b - Ax^0$ das Residuum ist. Dieses Gleichungssystem wird im Laufe des Algorithmus in eine Grobgitterstruktur übertragen, die mit

$$\tilde{A}\tilde{e} = \tilde{r} \qquad \tilde{A} \in \mathbb{R}^{m\times m} \quad \tilde{e}, \tilde{r} \in \mathbb{R}^m$$

bezeichnet wird. Ist ein Vektor v in der feinen Struktur gegeben, so sei $\tilde{v}$ der korrespondierende Vektor in der Grobstruktur. Damit überhaupt von einem gröberen Gitter gesprochen werden kann, ist natürlich $m < n$.

Fiktives Gitter

Bei dem algebraischen Mehrgitterverfahren liegt kein Gitter im eigentlichen Sinne vor. Der Operator A in (1) wird allerdings als ein Gitter mit n Knoten interpretiert, wobei Einträge $a_{ij} \neq 0$ mit $i \neq j$ die (gewichteten) Kanten zwischen den Knoten i und j darstellen.

Sei $\Omega = \{1, \ldots, n\}$ die Menge aller Variablen bzw. Knoten. Eine Vergröberung des Gitters ist nun gleichbedeutend damit, dass bestimmte Variablen weggelassen werden. Zum Transport eines Vektors vom feinen ins grobe Gitter wird die Anzahl der Variablen bzw. die Dimension verkleinert, der Vorgang wird als *Restriktion* bezeichnet. Der lineare Operator, der hierfür zuständig ist, sei R. Die Werte, die beim Transport in das grobe Gitter verloren gegangen sind, müssen beim Rücktransport interpoliert werden. Diesen Vorgang nennt man auch *Prolongation*, der Operator hierfür sei P. Somit ergibt sich

$$x, y \in \mathbb{C}^n, \tilde{x}, \tilde{y} \in \mathbb{C}^m \quad P \in \mathbb{C}^{n\times m}, R \in \mathbb{C}^{m\times n} \qquad Rx = \tilde{x}, P\tilde{y} = y.$$

Grobgitterkorrektur

Definition 1. *Die Matrix $\tilde{A}$ der Grobgitterstruktur wird unter Benutzung von A als der* ***Galerkin-Operator*** *definiert:*

$$\tilde{A} := R \cdot A \cdot P.$$

Definition 2. *Die Matrix K, die zur Bestimmung des neuen Fehlers verwendet wird, nennt man* ***Coarse-Grid-Correction-Operator****:*

$$e^{k+1} = \underbrace{(I - P\tilde{A}^{-1}RA)}_{=:K} e^k$$

Glättung

Die Grobgitterkorrektur alleine reicht nicht aus, um eine Konvergenz des Verfahrens zu ermöglichen, denn es werden nicht alle Fehleranteile effektiv reduziert. Hochfrequente Fehleranteile können bekanntermaßen durch einen Glätter herausgefiltert werden. Dieser soll vor- und/oder nachgeschaltet werden. Typische Glätter für algebraische Mehrgitterverfahren sind Gauß-Seidel, relaxiertes Jacobi, Richardson und ILU.

Das Zusammenspiel von Glättern und Grobgitterkorrekturen kann in vielen Varianten realisiert werden. So kann man sowohl zur Glättung des aktuellen Fehlers einen Glätter einsetzen (S^{ν_1}, *pre-smoothing*), als auch nach der Berechnung den neuen Fehler glätten (S^{ν_2}, *post-smoothing*). Die Potenzen ν_1 und ν_2 geben dabei die Anzahl der Vor- bzw. Nachglättungsschritte an.

Die beiden Möglichkeiten fasst man mit dem *two-grid iteration operator* zusammen:

$$e_{k+1} = Me_k \qquad M := S^{\nu_2} K S^{\nu_1}.$$

Algorithmus

Mit den obigen Bezeichnungen und $A_0 := A$ hat der Algorithmus (*V-Zyklus*) eines algebraischen Mehrgitterverfahrens die folgende Gestalt:

$$x^k = AMG(x^k, b^k, k)$$

1. Pre-smoothing: $x^k = S^{\nu_1} x^k$
2. Bestimme aktuelles Residuum: $r^k = b^k - A^k x^k$
3. Transformiere die Gleichung in das grobe Gitter: $b^{k+1} = R^k r^k$
4. Bestimme den Galerkin-Operator: $A_{k+1} = R_k A_k P_k$
5. Löse die Gleichung rekursiv: $x^{k+1} = AMG(0, b^{k+1}, k+1)$
6. Berechne die neue Iterierte: $x^k = x^k + P^k x^{k+1}$
7. Post-smooting: $x^k = S^{\nu_2} x^k$

Abbildung 1: AMG V-Zyklus

Algebraische Mehrgitterverfahren für Toeplitz- und zirkulante Bandmatrizen

In diesem Kapitel wird das algebraische Mehrgitterverfahren für die spezielle Klasse der Toeplitz- und zirkulanten Bandmatrizen näher untersucht. Dabei wird der Begriff der Spektraläquivalenz erklärt und diskutiert, wie dieser zur Optimierung des Verfahrens verwendet werden kann. Theoretische Ergebnisse, die die Glättungseigenschaft und Konvergenz des optimierten Verfahrens betreffen können in dem Artikel von Bolten und Frommer [4] nachgelesen werden.

Toeplitz- und zirkulante Bandmatrizen

In [5] findet man folgende Definition:

Definition 3. *Eine Folge von* ***Toeplitz-Matrizen*** T_n *ist gegeben durch das auf* $[-\pi, \pi]$ *definierte generierende Symbol*

$$f(\theta) = \sum_{k=-\infty}^{\infty} t_k e^{-ik\theta} \quad \textit{mit } t_k = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(\theta)e^{ik\theta}\, d\theta \quad k \in \mathbb{Z}.$$

Die Koeffizienten einer Toeplitz-Matrix sind also gerade die Fourierkoeffizienten des generierenden Symbols. Ist f ein trigonometrisches Polynom, d. h. es gilt

$$f(\theta) = \sum_{k=-p}^{q} t_k e^{-ik\theta},$$

so sind die zugehörigen Toeplitz-Matrizen Bandmatrizen mit Bandbreite $p+q+1$. Es ergibt sich folgende Struktur:

$$\begin{pmatrix} t_0 & t_{-1} & t_{-2} & \cdots & t_{-n+1} \\ t_1 & t_0 & t_{-1} & \cdots & t_{-n+2} \\ t_2 & t_1 & t_0 & \cdots & t_{-n+3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ t_{n-1} & t_{n-2} & t_{n-3} & \cdots & t_0 \end{pmatrix}$$

Zirkulante Matrizen sind spezielle Toeplitz-Matrizen. Sie sind folgendermaßen definiert:

Definition 4. *Eine* ***zirkulante Matrix*** C *mit dem auf* $[-\pi, \pi)$ *definierten generierenden Symbol* f *ist definiert durch*

$$C = C(f) = F_n \mathit{Diag}\left(f\left(w^{[n]}\right)\right) F_n^H,$$

wobei $n \in \mathbb{N} \setminus \{0\}$ *die Dimension von* C *ist,* F_n *die Fouriermatrix dieser Dimension, gegeben durch*

$$F_n = \frac{1}{\sqrt{n}} \left[e^{-ikw_j^{[n]}} \right]_{j,k \in \{0,\ldots,n-1\}} \qquad \textit{und } w_j^{[n]} = \frac{2\pi i j}{n}.$$

Anschaulich ist jeder Zeilenvektor zum darüberliegenden Zeilenvektor um einen Eintrag nach rechts verschoben:

$$\begin{pmatrix} c_1 & c_2 & c_3 & \cdots & c_n \\ c_n & c_1 & c_2 & \cdots & c_{n-1} \\ c_{n-1} & c_n & c_1 & \cdots & c_{n-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_2 & c_3 & c_4 & \cdots & c_1 \end{pmatrix}$$

Die Eigenwerte einer zirkulanten Matrix sind durch das generierende Symbol f, ausgewertet an den Stellen $w_j^{[n]}, j = 0, \ldots, n-1$, gegeben. Diese Eigenschaft wird im Folgenden zur Optimierung der Parameter verwendet.

Definition 5. *Seien $C_1, C_2, \ldots, C_d$ zirkulante Matrizen. Eine* ***d-level zirkulante Matrix*** *C der Dimension $n = (n_1, \ldots, n_d) \in (\mathbb{N} \setminus \{0\})^d$ ist dann definiert durch*

$$C = C_1 \otimes C_2 \otimes \ldots \otimes C_d.$$

Zum Lösen von linearen Gleichungssystemen mit einer zirkulanten Koeffizientenmatrix, bieten sich FFT-Verfahren an, da zirkulante Matrizen durch die Fouriermatrix der passenden Dimension diagonalisierbar sind. Für Matrix-Vektor-Produkte ergibt sich bei diesen Verfahren eine Komplexität von $\mathcal{O}(n_1 \cdots n_d \log(\max_j n_j))$. Ist das generierende Symbol f ein trigonometrisches Polynom festen Grades, so werden nur $\mathcal{O}(n_1 \cdots n_d)$ Matrix-Vektor-Operationen benötigt. Somit ist die Verwendung eines Mehrgitterverfahrens günstiger als die Verwendung eines FFT-Verfahrens.

Komplexität des Galerkin-Operators

Algebraische Mehrgitterverfahren, die auf gröberen Gittern den Galerkin-Operator verwenden haben das Problem, dass die Komplexität dieses Operators auf jedem Gitter zunimmt.

Das Problem lässt sich sehr schön an dem folgenden Beispiel erläutern: Betrachtet wird die Diskretisierung der Poisssongleichung in 2D mit periodischen Randbedingungen. Die Diskretisierung der zirkulante Koeffizientenmatrix erfolgt dabei durch den bekannten 5-Punkte-Differenzenstern

$$\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}.$$

Der Galerkin-Operator des Grobgitters lautet

$$\begin{bmatrix} -\frac{1}{64} & -\frac{1}{32} & -\frac{1}{64} \\ -\frac{1}{32} & \frac{6}{32} & -\frac{1}{32} \\ -\frac{1}{64} & -\frac{1}{32} & -\frac{1}{64} \end{bmatrix},$$

ist also ein vollbesetzter 9-Punkte-Stern.

Um dieses Problem zu umgehen, werden im folgenden Abschnitt spektraläquivalente Differenzensterne eingeführt.

Spektraläquivalenz

Die spektraläquivalente Differenzensterne dieses Abschnitts basieren auf folgender Definition von Spektraläquivalenz:

Definition 6. *Seien $\{A_j\}_j$ und $\{B_j\}_j$ zwei Folgen von $N(j) \times N(j)$-Matrizen, wobei $N(j+1) > N(j)$. $\{B_j\}_j$ ist genau dann* ***spektraläquivalent*** *zu $\{A_j\}_j$, wenn alle Eigenwerte von $\{B_j^{-1} A_j\}_j$ im Intervall $[\lambda, \Lambda]$ mit $0 < \lambda \leq \Lambda < \infty$ unabhängig von j liegen.*

Für die spezielle Klasse der Toeplitz- und strukturierten Bandmatrizen soll an Stelle des Galerkin-Operators eine spektraläquivalente Matrix verwendet werden. Ein Beweis der Konvergenz eines algebraischen Mehrgitterverfahrens, das diese Idee umsetzt, ist in dem Artikel von Bolten und Frommer [4] zu finden.

Die folgenden spektraläquivalenten Sterne in 2D und 3D definieren eine Folge von Matrizen, die spektraläquivalent zur ursprünglichen Folge von Matrizen ist:

Definition 7. *Sei*

$$\begin{bmatrix} c & b & c \\ a & -2(a+b)-4c & a \\ c & b & c \end{bmatrix}$$

ein beliebiger 9-Punkte-Stern in 2D, dessen generierendes Symbol eine Nullstelle am Ursprung hat und dessen Gradient dort verschwindet.

Der ***spektraläquivalente 5-Punkte-Stern*** *ist definiert als*

$$\begin{bmatrix} & b+2c & \\ a+2c & -2(a+b)-8c & a+2c \\ & b+2c & \end{bmatrix}$$

Definition 8. *Sei*

$$\begin{bmatrix} g & f & g \\ e & c & e \\ g & f & g \end{bmatrix}$$

$$\begin{bmatrix} d & b & d \\ a & -2(a+b+c)-4(d+e+f)-8g & a \\ d & b & d \end{bmatrix}$$

$$\begin{bmatrix} g & f & g \\ e & c & e \\ g & f & g \end{bmatrix}$$

ein beliebiger 27-Punkte-Stern in 3D, dessen generierendes Symbol eine Nullstelle am Ursprung hat und dessen Gradient dort verschwindet.

Der ***spektraläquivalente 7-Punkte-Stern*** *ist definiert als*

$$\begin{bmatrix} c+2(e+f)+4g \end{bmatrix}$$

$$\begin{bmatrix} & b+2(d+f)+4g & \\ a+2(d+e)+4g & -2(a+b+c)-8(d+e+f)-24g & a+2(d+e)+4g \\ & b+2(d+f)+4g & \end{bmatrix}$$

$$\begin{bmatrix} c+2(e+f)+4g \end{bmatrix}$$

Optimierungsparameter

Optimaler Relaxationsparameter

Als Glätter für ein Mehrgitterverfahren soll das relaxierte Jacobi-Verfahren verwendet werden. Für eine optimale Glättung wird in diesem Abschnitt ein optimaler Relaxationsparameter bestimmt. Sei dazu

$$\begin{bmatrix} g & f & g \\ e & c & e \\ g & f & g \end{bmatrix}$$

$$\begin{bmatrix} d & b & d \\ a & -2(a+b+c)-4(d+e+f)-8g & a \\ d & b & d \end{bmatrix}$$

$$\begin{bmatrix} g & f & g \\ e & c & e \\ g & f & g \end{bmatrix}$$

ein beliebiger 27-Punkte-Stern mit $a, b, c, d, e, f, g \geq 0$.

Das generierende Symbol des relaxierten Jacobi-Verfahren lautet dann

$$\begin{aligned} F(x,y,z) = (1-\omega) + \omega \underbrace{\frac{1}{-2(a+b+c)-4(d+e+f)-8g}}_{=:-\alpha} \cdot (&-2a\cos(x) - 2b\cos(y) - 2c\cos(z) \\ &-4d\cos(x)\cos(y) \\ &-4e\cos(x)\cos(z) \\ &-4f\cos(y)\cos(z) \\ &-8g\cos(x)\cos(y)\cos(z)). \end{aligned}$$

Dies ist offensichtlich äquivalent zu

$$\begin{aligned} F(x,y,z) = 1 - \omega \cdot (1 + \alpha(&-2a\cos(x) - 2b\cos(y) - 2c\cos(z) \\ &-4d\cos(x)\cos(y) \\ &-4e\cos(x)\cos(z) \\ &-4f\cos(y)\cos(z) \\ &-8g\cos(x)\cos(y)\cos(z))). \end{aligned}$$

Um den optimalen Relaxationsparameter ω zu bestimmen werden Maximum und Minimum des Faktors von α für $x, y, z \in [-\pi, \pi]^3 \setminus \left(-\frac{\pi}{2}, \frac{\pi}{2}\right)^3$ gesucht.

Maximum und Minimum dieses Faktors zu bestimmen ist nicht besonders schwierig, denn die Variablen x, y und z sind jeweils Argumente einer Cosinus-Funktion. Diese nimmt bekanntermaßen ihre Extremwerte im zu untersuchenden Definitionsbereich an den Stellen $0, \pm\frac{\pi}{2}$ und $\pm\pi$ an. Zur Bestimmung des Maximums und Minimums des generierenden Symbols reicht es deshalb $3^3 - 1 = 26$ Fälle zu untersuchen (der Fall $x = 0, y = 0$ und $z = 0$ ist nicht erlaubt). Der folgenden Tabelle sind diese 26 Fälle zu entnehmen:

Tabelle 1: Die 26 Fälle für das generierende Symbol

	x	y	z	$F(x,y,z)$
1	0	$\pm\frac{\pi}{2}$	$\pm\frac{\pi}{2}$	$1-\omega(1+\alpha\cdot(-2a))$
2	$\pm\pi$	$\pm\frac{\pi}{2}$	$\pm\frac{\pi}{2}$	$1-\omega(1+\alpha\cdot(2a))$
3	$\pm\frac{\pi}{2}$	0	$\pm\frac{\pi}{2}$	$1-\omega(1+\alpha\cdot(-2b))$
4	$\pm\frac{\pi}{2}$	$\pm\pi$	$\pm\frac{\pi}{2}$	$1-\omega(1+\alpha\cdot(2b))$
5	$\pm\frac{\pi}{2}$	$\pm\frac{\pi}{2}$	0	$1-\omega(1+\alpha\cdot(-2c))$
6	$\pm\frac{\pi}{2}$	$\pm\frac{\pi}{2}$	$\pm\pi$	$1-\omega(1+\alpha\cdot(2c))$
7	$\pm\pi$	0	$\pm\frac{\pi}{2}$	$1-\omega(1+\alpha\cdot(2a-2b+4d))$
8	$\pm\pi$	$\pm\pi$	$\pm\frac{\pi}{2}$	$1-\omega(1+\alpha\cdot(2a+2b-4d))$
9	0	$\pm\pi$	$\pm\frac{\pi}{2}$	$1-\omega(1+\alpha\cdot(-2a+2b+4d))$
10	0	0	$\pm\frac{\pi}{2}$	$1-\omega(1+\alpha\cdot(-2a-2b-4d))$
11	$\pm\pi$	$\pm\frac{\pi}{2}$	0	$1-\omega(1+\alpha\cdot(2a-2c+4e))$
12	$\pm\pi$	$\pm\frac{\pi}{2}$	$\pm\pi$	$1-\omega(1+\alpha\cdot(2a+2c-4e))$
13	0	$\pm\frac{\pi}{2}$	$\pm\pi$	$1-\omega(1+\alpha\cdot(-2a+2c+4e))$
14	0	$\pm\frac{\pi}{2}$	0	$1-\omega(1+\alpha\cdot(-2a-2c-4e))$
15	$\pm\frac{\pi}{2}$	$\pm\pi$	$\pm\pi$	$1-\omega(1+\alpha\cdot(2b+2c-4f))$
16	$\pm\frac{\pi}{2}$	$\pm\pi$	0	$1-\omega(1+\alpha\cdot(2b-2c+4f))$
17	$\pm\frac{\pi}{2}$	0	$\pm\pi$	$1-\omega(1+\alpha\cdot(-2b+2c+4f))$
18	$\pm\frac{\pi}{2}$	0	0	$1-\omega(1+\alpha\cdot(-2b-2c-4f))$
19	$\pm\pi$	0	0	$1-\omega(1+\alpha\cdot(2a-2b-2c+4d+4e-4f+8g))$
20	0	$\pm\pi$	0	$1-\omega(1+\alpha\cdot(-2a+2b-2c+4d-4e+4f+8g))$
21	0	0	$\pm\pi$	$1-\omega(1+\alpha\cdot(-2a-2b+2c-4d+4e+4f+8g))$
22	$\pm\pi$	$\pm\pi$	$\pm\pi$	$1-\omega(1+\alpha\cdot(2a+2b+2c-4d-4e-4f+8g))$
23	$\pm\pi$	0	$\pm\pi$	$1-\omega(1+\alpha\cdot(2a-2b+2c+4d-4e+4f-8g))$
24	0	$\pm\pi$	$\pm\pi$	$1-\omega(1+\alpha\cdot(-2a+2b+2c+4d+4e-4f-8g))$
25	$\pm\pi$	$\pm\pi$	0	$1-\omega(1+\alpha\cdot(2a+2b-2c-4d+4e+4f-8g))$
26	$\pm\frac{\pi}{2}$	$\pm\frac{\pi}{2}$	$\pm\frac{\pi}{2}$	$1-\omega$

Nun können einige dieser 26 Fälle ausgeschlossen werden:

1. Fall 1 kann kein Maximum sein, denn a ist nichtnegativ und somit ist Fall 2 immer größer. Fall 1 kann aber auch kein Minimum sein, denn z. B. Fall 10 ist mindestens genauso klein.

2. Fall 2 kann kein Minimum sein, denn z. B. Fall 10 ist immer mindestens genauso klein. Fall 2 kann aber auch kein Maximum sein, denn wäre Fall 2 ein Maximum, so müsste dieser Fall insbesondere immer größer oder gleich Fall 7 und Fall 8 sein. Aus Fall 7 ergibt sich dann die Bedingung $b > 2d$, aus Fall 8 folgt $b < 2d$. Dies ist ein Widerspruch, also kann Fall 2 kein Maximum sein.

3. Die Fälle 3-6 können ähnlich wie die Fälle 1 und 2 ausgeschlossen werden.

4. Aufgrund der Fälle 10, 14 und 18 können die Fälle 7-9, 11-13 und 15-17 keine Minima sein.

5. Fall 7 kann kein Maximum sein, denn: Wäre Fall 7 ein Maximum, so folgt aus Fall 19 die Bedingung $c + 2f > 2e + 4g$ und aus Fall 23 die Bedingung $c + 2f < 2e + 4g$, ein Widerspruch.

6. Analog zu Fall 7 können auch die Fälle 8,9, 11-13 und 15-17 als Minima ausgeschlossen werden.
7. Fall 26 kann als Minimum ausgeschlossen werden, da einer der Fälle 10, 14 oder 18 mindestens genauso klein ist. Fall 26 kann aber auch kein Maximum sein, denn dieser Fall ist nur ein Maximum, falls alle Koeffizienten Null sind. Dann ist dieser Fall aber in den übrig gebliebenen Fällen enthalten.

Als potentielle Maxima und Minima bleiben insgesamt 10 Fälle übrig:

Tabelle 2: Potentielle Maxima und Minima

	x	y	z	$F(x,y,z)$
10	0	0	$\pm\frac{\pi}{2}$	$1-\omega(1+\alpha\cdot(-2a-2b-4d))$
14	0	$\pm\frac{\pi}{2}$	0	$1-\omega(1+\alpha\cdot(-2a-2c-4e))$
18	$\pm\frac{\pi}{2}$	0	0	$1-\omega(1+\alpha\cdot(-2b-2c-4f))$
19	$\pm\pi$	0	0	$1-\omega(1+\alpha\cdot(2a-2b-2c+4d+4e-4f+8g))$
20	0	$\pm\pi$	0	$1-\omega(1+\alpha\cdot(-2a+2b-2c+4d-4e+4f+8g))$
21	0	0	$\pm\pi$	$1-\omega(1+\alpha\cdot(-2a-2b+2c-4d+4e+4f+8g))$
22	$\pm\pi$	$\pm\pi$	$\pm\pi$	$1-\omega(1+\alpha\cdot(2a+2b+2c-4d-4e-4f+8g))$
23	$\pm\pi$	0	$\pm\pi$	$1-\omega(1+\alpha\cdot(2a-2b+2c+4d-4e+4f-8g))$
24	0	$\pm\pi$	$\pm\pi$	$1-\omega(1+\alpha\cdot(-2a+2b+2c+4d+4e-4f-8g))$
25	$\pm\pi$	$\pm\pi$	0	$1-\omega(1+\alpha\cdot(2a+2b-2c-4d+4e+4f-8g))$

Für Eingabedaten a, b, c, d, e, f und g können Maximum und Minimum bestimmt werden. Zur Berechnung des optimalen Relaxationsparameter ω wird dann das Maximum gleich dem Negativen des Minimum gesetzt:

$$1-\omega(1+\alpha\cdot\max) = -1+\omega(1+\alpha\cdot\min).$$

Die folgenden Umformungen führen schließlich zum optimalen Parameter ω:

$$1-\omega(1+\alpha\cdot\max) = -1+\omega(1+\alpha\cdot\min)$$

$$\begin{aligned}
&\Leftrightarrow 1-\omega\left(\frac{2(a+b+c)+4(d+e+f)+8g+\max}{2(a+b+c)+4(d+e+f)+8g}\right)\\
&= -1+\omega\left(\frac{2(a+b+c)+4(d+e+f)+8g+\min}{2(a+b+c)+4(d+e+f)+8g}\right)\\
&\Leftrightarrow 2 = \omega\left(\frac{2(2(a+b+c)+4(d+e+f)+8g)+\min+\max}{2(a+b+c)+4(d+e+f)+8g}\right)\\
&\Leftrightarrow \omega = 2\cdot\frac{2(a+b+c)+4(d+e+f)+8g}{2(2(a+b+c)+4(d+e+f)+8g)+\min+\max}
\end{aligned}$$

Optimaler Verschiebungsparameter

Wird eine zum spektraläquivalenten Stern zugehörige Matrix an Stelle des Galerkin-Operators auf den gröberen Gittern verwendet, so ist ein zusätzlicher Verschiebungsparameter sinnvoll. Anschaulich sorgt dieser Parameter dafür, dass die Funktionsgraphen der generierenden Symbole des Galerkin-Operators F und des spektraläquivalenten Operators G möglichst übereinander liegen. D. h. um den optimalen

Verschiebungsparameter zu bestimmen, muss das Minimum des generierenden Symbols F/G gesucht werden. Sei dazu

$$\begin{bmatrix} g & f & g \\ e & c & e \\ g & f & g \end{bmatrix}$$
$$\begin{bmatrix} d & b & d \\ a & -2(a+b+c)-4(d+e+f)-8g & a \\ d & b & d \end{bmatrix}$$
$$\begin{bmatrix} g & f & g \\ e & c & e \\ g & f & g \end{bmatrix}$$

wieder ein beliebiger 27-Punkte-Stern mit $a, b, c, d, e, f, g \geq 0$. Das generierende Symbol F lautet

$$\begin{aligned} F(x,y,z) = &-2(a+b+c)-4(d+e+f)-8g+2a\cos(x)+2b\cos(y)+2c\cos(z) \\ &+4d\cos(x)\cos(y)+4e\cos(x)\cos(z)+4f\cos(y)\cos(z)+8g\cos(x)\cos(y)\cos(z). \end{aligned}$$

Nach obiger Definition lautet der spektraläquivalente 7-Punkte-Stern

$$\begin{bmatrix} c+2(e+f)+4g \end{bmatrix}$$
$$\begin{bmatrix} & b+2(d+f)+4g & \\ a+2(d+e)+4g & -2(a+b+c)-8(d+e+f)-24g & a+2(d+e)+4g \\ & b+2(d+f)+4g & \end{bmatrix}$$
$$\begin{bmatrix} c+2(e+f)+4g \end{bmatrix}$$

mit dem generierenden Symbol

$$\begin{aligned} G(x,y,z) = &-2(a+b+c)-8(d+e+f)-24g+2(a+2(d+e)+4g)\cos(x) \\ &+2(b+2(d+f)+4g)\cos(y)+2(c+2(e+f)+4g)\cos(z). \end{aligned}$$

Ziel ist die Minimierung des Quotienten F/G

$$\begin{aligned} F(x,y,z)/G(x,y,z) = (&-2(a+b+c)-4(d+e+f)-8g+2a\cos(x)+2b\cos(y)+2c\cos(z) \\ &+4d\cos(x)\cos(y)+4e\cos(x)\cos(z)+4f\cos(y)\cos(z) \\ &+8g\cos(x)\cos(y)\cos(z))\ / \\ (&-2(a+b+c)-8(d+e+f)-24g+2(a+2(d+e)+4g)\cos(x) \\ &+2(b+2(d+f)+4g)\cos(y)+2(c+2(e+f)+4g)\cos(z)). \end{aligned}$$

Offensichtlich ist dieser Ausdruck äquivalent zu

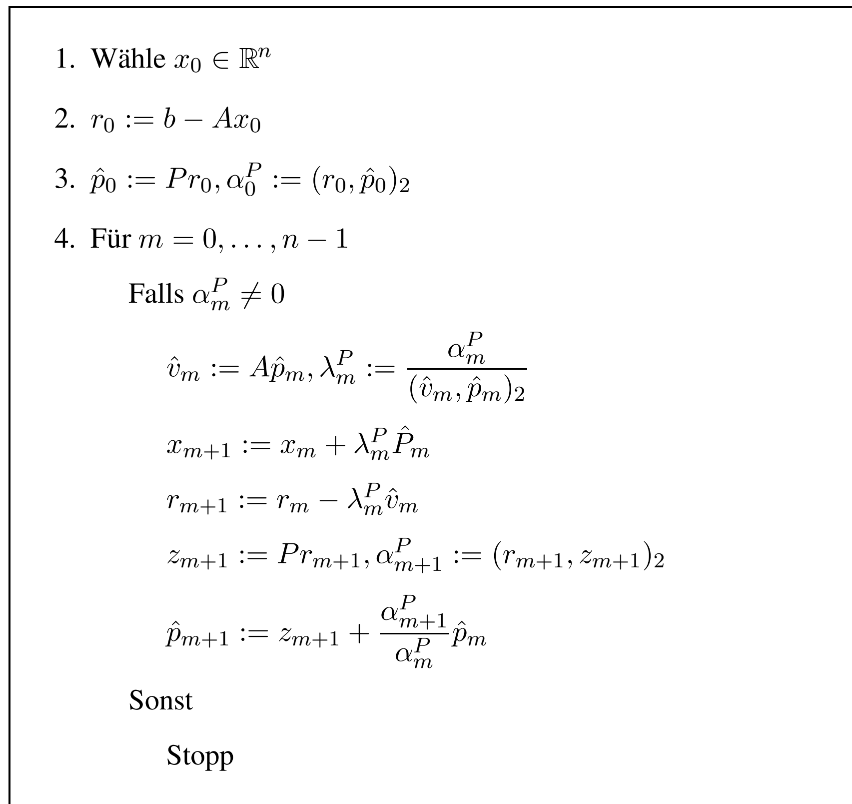
$$\begin{aligned} F(x,y,z)/G(x,y,z) = 1 - (&2(d+e+f+4g-e\cos(z)-f\cos(z)-2g\cos(z)) \\ &-cos(y)(d+f+2g-f\cos(z)) \\ &+\cos(x)(-d-e-2g+e\cos(z)+\cos(y)(d+2g\cos(z))))\ / \\ (&a+b+c+2d+2e+2f+4g-c\cos(z)-\cos(y)(b+2f\cos(z)) \\ &-\cos(x)(a+2e\cos(z)+2\cos(y)(d+2g\cos(z))). \end{aligned}$$

Von diesem Ausdruck können die Extremwerte bestimmt werden: Entlang der Koordinatenachsen ist der Ausdruck gleich Null, somit wird das Maximum in den Ecken von $[-\pi, \pi]^3$ angenommen. Der optimale Verschiebungsparameter lautet somit

$$\frac{4(d+e+f+2g)}{a+b+c+4d+4e+4f+12g}.$$

Präkonditionierung

Algebraische Mehrgitterverfahren können als Präkonditionierer für Konjugierte-Gradienten Verfahren (CG-Verfahren) verwendet werden [6]. Der folgende Algorithmus aus [7] diente als Grundlage für die Implementierung:



1. Wähle $x_0 \in \mathbb{R}^n$
2. $r_0 := b - Ax_0$
3. $\hat{p}_0 := Pr_0, \alpha_0^P := (r_0, \hat{p}_0)_2$
4. Für $m = 0, \dots, n-1$

   Falls $\alpha_m^P \neq 0$

   $\hat{v}_m := A\hat{p}_m, \lambda_m^P := \frac{\alpha_m^P}{(\hat{v}_m, \hat{p}_m)_2}$

   $x_{m+1} := x_m + \lambda_m^P \hat{P}_m$

   $r_{m+1} := r_m - \lambda_m^P \hat{v}_m$

   $z_{m+1} := Pr_{m+1}, \alpha_{m+1}^P := (r_{m+1}, z_{m+1})_2$

   $\hat{p}_{m+1} := z_{m+1} + \frac{\alpha_{m+1}^P}{\alpha_m^P} \hat{p}_m$

   Sonst

   Stopp

Abbildung 2: PCG-Verfahren

Die Präkonditionierungsmatrix P entspricht dabei τ Schritten eines algebraischen Mehrgitterverfahrens mit ν_1 pre-smoothing-Schritten und ν_2 post-smoothing-Schritten. Wird mit einem algebraischen Mehrgitterverfahren präkonditioniert, das einen spektraläquivalenten Operator verwendet, so wird dieser nicht für das CG-Verfahren, sondern ausschließlich für das Mehrgitterverfahren und schon auf dem feinsten Gitter verwendet.

Ergebnisse

Als Vorlage diente eine Implementierung eines algebraischen Mehrgitterverfahrens in C von meinem Betreuer Matthias Bolten. Parallelisierungen erfolgen darin mit MPI. Folgende größere Änderungen habe ich an den Quellcodes vorgenommen:

1. Gewünschte Eingabeparameter können von einer Textdatei eingelesen werden. Diese enthält
   - die maximale Anzahl von Iterationsschritten des Mehrgitterverfahrens,
   - die Problemgröße (dreidimensional, in jeder Dimension für zirkulante Matrizen $2^k, k \in \mathbb{N}$, für Toeplitz-Matrizen $2^k - 1, k \in \mathbb{N} \setminus \{0\}$,
   - die Periodizität (für zirkulante Matrizen eine Eins, für Toeplitz-Matrizen eine Null),
   - die Anzahl der Vor- und Nachglättungsschritte,

- die Größe des Differenzensterns,
- die Koeffizienten des Differenzensterns sowie
- die Verschiebung des Differenzensterns in x-, y- und z-Richtung.

2. Die vorliegende Datenstruktur zur Speicherung der Daten des Mehrgitterverfahrens wurde entsprechend erweitert.

3. Ein optimaler Relaxationsparameter wird für jedes Gitter bestimmt.

4. Spektraläquivalente Sterne können bestimmt und verwendet werden.

5. Falls spektraläquivalenten Sterne verwendet werden, so wird der zugehörige optimale Verschiebungskoeffizient berechnet.

Verglichen wurden

MG: Algebraisches Mehrgitter mit Galerkin-Operator,

MGse: Algebraisches Mehrgitter mit spektraläquivalentem Operator,

PCG: präkonditioniertes CG mit algebraischem Mehrgitter mit Galerkin-Operator und

PCGse: präkonditioniertes CG mit algebraischem Mehrgitter mit spektraläquivalentem Operator.

Testumgebung

Mit den folgenden Eingabedaten wurden Testläufe durchgeführt:

- Differenzenstern (Approximation 4. Ordnung an eine partielle Differentialgleichung):

$$\begin{bmatrix} & 1 & \\ 1 & 2 & 1 \\ & 1 & \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & -24 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} & 1 & \\ 1 & 2 & 1 \\ & 1 & \end{bmatrix}$$

- Anzahl Vor- und Nachglättungsschritte
  - PCG: 3
  - PCGse: 1 Vorglättungsschritt, kein Nachglättungsschritt
- Maximale Anzahl von Iterationsschritten im Mehrgitterverfahren: 10
- Anzahl von Unbekannten: $127^3/128^3 - 1023^3/1024^3$
- Anzahl Prozessoren: $32 - 8192$

Konvergenz

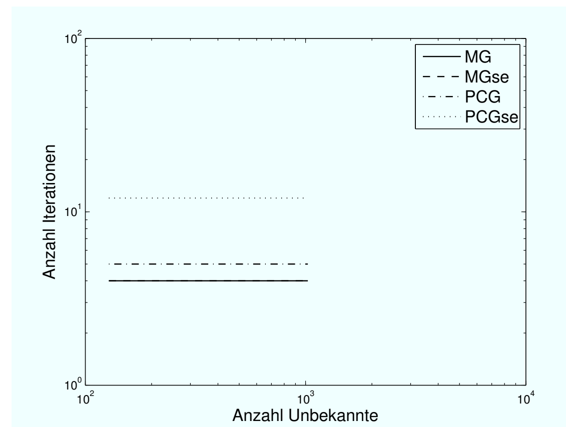
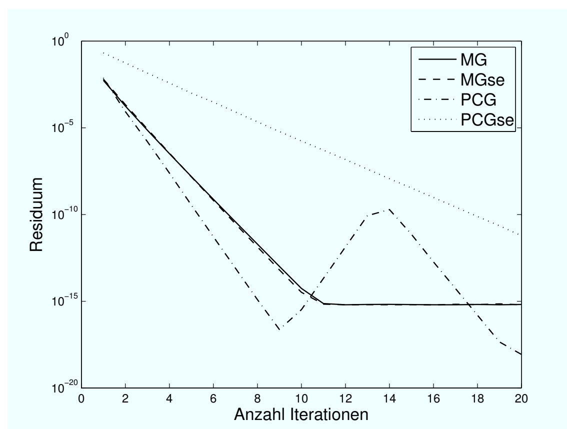
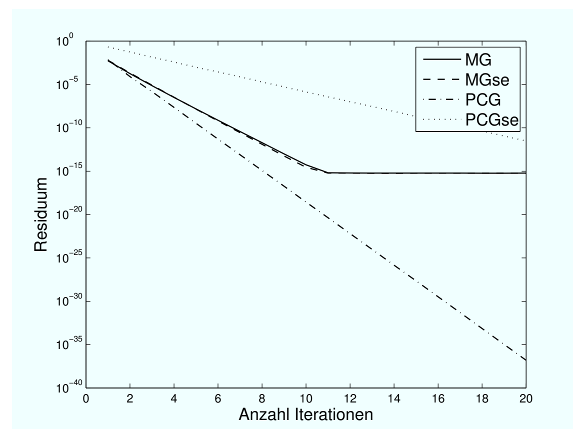


Abbildung 3: Konvergenz in Abhängigkeit der Anzahl der Iterationen



(a) 128^3 Unbekannte

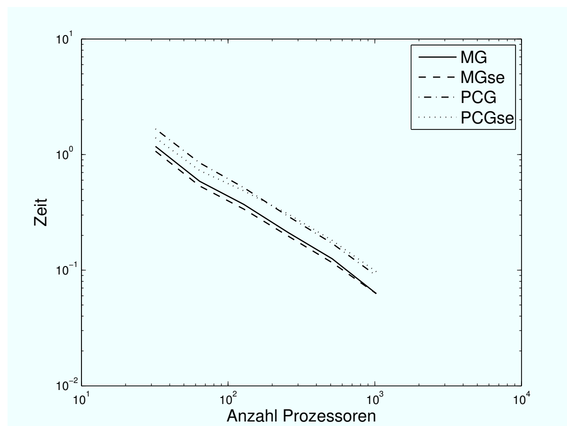


(b) 127^3 Unbekannte

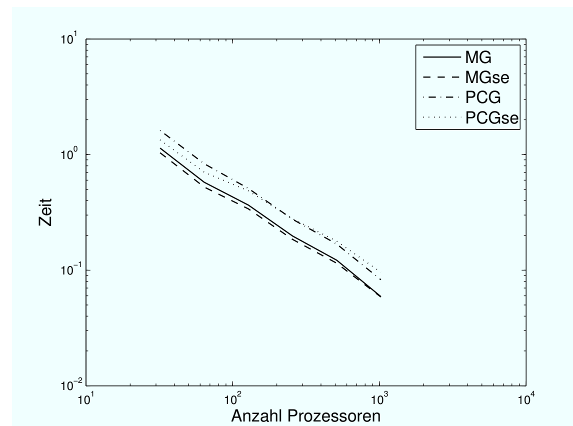
Abbildung 4: Konvergenz

Das Oszillationsverhalten des Residuums des präkonditionierten CG-Verfahrens der zirkulanten Matrix mit 128^3 ist nicht verwunderlich, da das generierende Symbol des Testbeispiels eine Nullstelle am Ursprung hat. D. h. die Matrix hat einen Eigenwert Null und ist somit singulär.

Skalierbarkeit

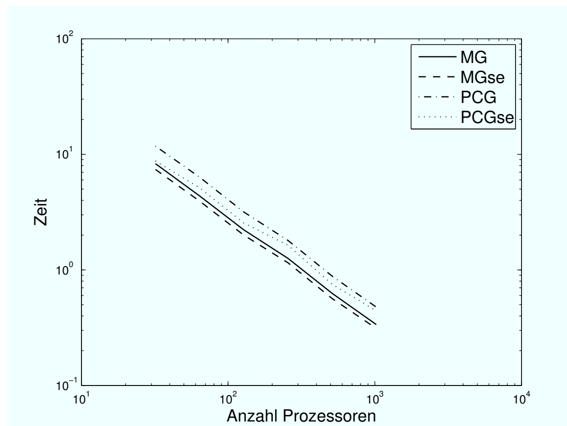


(a) 128^3 Unbekannte

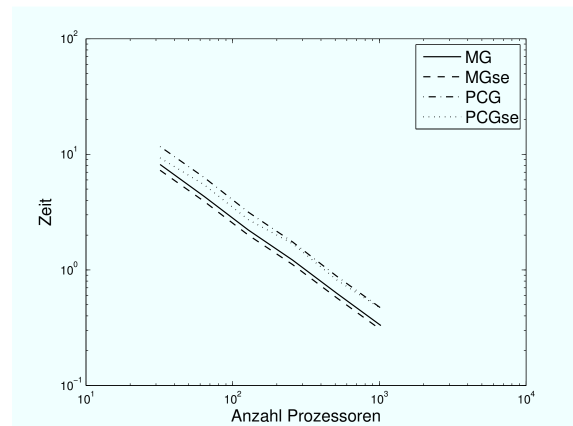


(b) 127^3 Unbekannte

Abbildung 5: Starke Skalierbarkeit (1)



(a) 256^3 Unbekannte

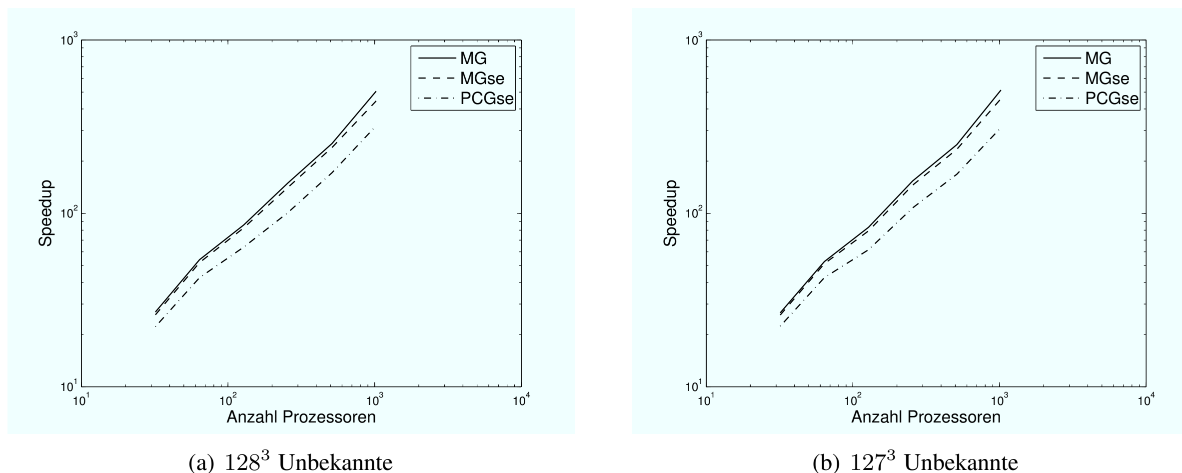


(b) 255^3 Unbekannte

Abbildung 6: Starke Skalierbarkeit (2)

Speedup und Effizienz

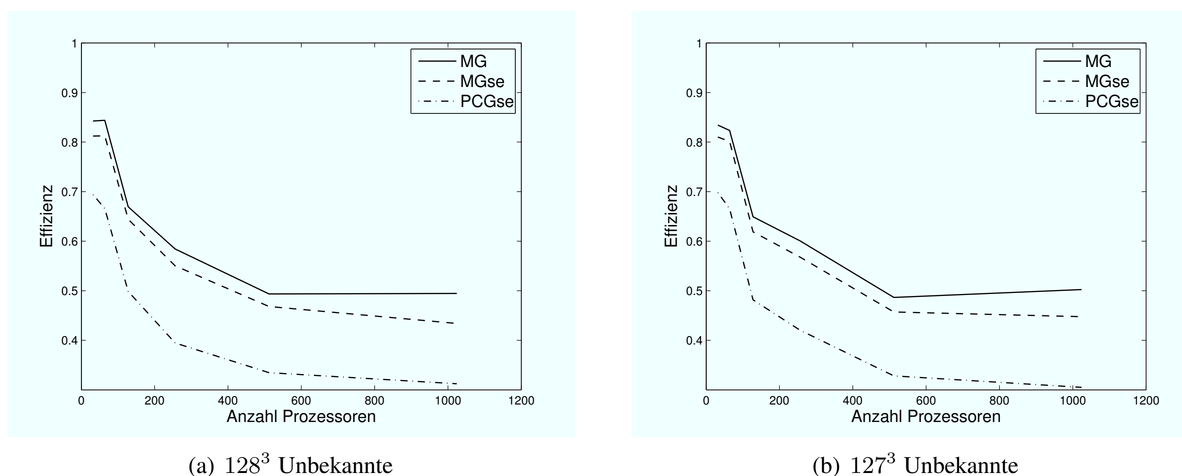
In diesem Abschnitt werden lediglich die beiden Mehrgittervarianten und das mit dem Mehrgitterverfahren, das einen spektraläquivalenten Operator verwendet, präkonditionierte CG-Verfahren miteinander verglichen. Eine Zeitmessung des präkonditionierten CG-Verfahrens mit dem Mehrgitterverfahren, das den Galerkin-Operator verwendet, ist aufgrund hohen Speicherbedarfs mit einem Prozessor für die Problemgrößen mit 127^3 bzw. 128^3 Unbekannten auf JUBL schon nicht mehr möglich.



(a) 128^3 Unbekannte

(b) 127^3 Unbekannte

Abbildung 7: Speedup



(a) 128^3 Unbekannte

(b) 127^3 Unbekannte

Abbildung 8: Effizienz

Weitere Ergebnisse sind den folgenden Tabellen zu entnehmen:

Tabelle 3: Weitere Testergebnisse für den Toeplitz-Fall

Anzahl Unbekannte	Anzahl Prozessoren	Typ	rel. Residuum	Zeit pro Iteration	Gesamtzeit
511	512	MG	$3.2474e-07$	1.02 s	4.19 s
511	512	MGse	$3.4268e-07$	0.91 s	3.76 s
511	512	PCG	$2.3506e-08$	1.19 s	5.99 s
511	512	PCGse	$6.6597e-07$	0.38 s	4.57 s
511	1024	MG	$3.2474e-07$	0.54 s	2.23 s

Tabelle 3: Weitere Testergebnisse für den Toeplitz-Fall

Anzahl Unbekannte	Anzahl Prozessoren	Typ	rel. Residuum	Zeit pro Iteration	Gesamtzeit
511	1024	MGse	$3.4268e-07$	0.49 s	2.01 s
511	1024	PCG	$3.3243e-08$	0.63 s	3.18 s
511	1024	PCGse	$9.4182e-07$	0.21 s	2.56 s
1023	1024	MG	$3.2578e-07$	4.15 s	17.12 s
1023	1024	MGse	$3.4721e-07$	3.54 s	14.67 s
1023	1024	PCG	$2.3494e-08$	4.66 s	23.34 s
1023	1024	PCGse	$3.4634e-07$	2.29 s	29.54 s
1023	2048	MG	$3.2578e-07$	2.17 s	9.03 s
1023	2048	MGse	$3.4721e-07$	1.86 s	7.76 s
1023	2048	PCG	$2.3494e-08$	2.47 s	12.48 s
1023	2048	PCGse	$3.2901e-07$	0.79 s	10.31 s

Tabelle 4: Weitere Testergebnisse für den zirkulanten Fall

Anzahl Unbekannte	Anzahl Prozessoren	Typ	rel. Residuum	Zeit pro Iteration	Gesamtzeit
512	512	MG	$3.2595e-07$	1.01 s	4.19 s
512	512	MGse	$3.5008e-07$	0.91 s	3.75 s
512	512	PCG	$2.3773e-08$	1.19 s	5.99 s
512	512	PCGse	$6.9645e-07$	0.38 s	4.55 s
512	1024	MG	$3.2595e-07$	0.54 s	2.23 s
512	1024	MGse	$3.5008e-07$	0.49 s	2.02 s
512	1024	PCG	$9.8493e-08$	0.22 s	2.57 s
512	1024	PCGse	$3.3619e-07$	0.64 s	3.19 s
1024	1024	MG	$3.2595e-07$	4.15 s	17.12 s
1024	1024	MGse	$3.5010e-07$	3.54 s	14.68 s
1024	1024	PCG	$2.3625e-08$	4.66 s	23.36 s
1024	1024	PCGse	$3.2766e-07$	2.30 s	29.68 s
1024	2048	MG	$3.2595e-07$	2.17 s	9.05 s
1024	2048	MGse	$3.5010e-07$	1.82 s	7.60 s
1024	2048	PCG	$2.3625e-08$	2.43 s	12.29 s
1024	2048	PCGse	$2.4331e-07$	0.79 s	10.30 s
2048	8192	MGse	$3.5010e-07$	3.53 s	14.62 s

Danksagung

Mein größter Dank gilt meinem Betreuer Matthias Bolten, der immer Zeit für mich hatte, wenn ich seine Unterstützung brauchte. Ich danke auch Herrn Prof. Dr. Andreas Frommer von der Bergischen Universität Wuppertal für die Befürwortung, die meine Teilnahme an diesem tollen Gaststudentenprogramm ermöglicht hat. Außerdem danke ich allen anderen Gaststudenten für ihre Anregungen und Hilfe und nicht zuletzt für eine schöne Zeit in Jülich.

Literatur

1. K. Stüben, Algebraic Multigrid (AMG): An Introduction with Applications. (1999).
2. A. Aricò, M. Donatelli, A V-cycle Multigigrid for multilevel matrix algebras: proof of optimality, Numer. Math., Vol. 105. (2007).
3. S. Serra Capizzano, C. Tablino-Possio, Multigrid methods for multilevel circulant matrices, SIAM J. Sci. Comput., Vol. 26. (2004).
4. M. Bolten, A. Frommer, Eingereicht bei Num. Linear Algebra Appl.
5. M. K. Ng, Iterative Methods for Toeplitz Systems, Oxford University Press. (2004).
6. O. Tatebe, The multigrid preconditioned conjugate gradient method, Sixth Copper Mountain Conference on Multigrid Methods, NASA, Vol. CP 3224. (1993).
7. A. Meister, Numerik linearer Gleichungssystem, Vieweg. (1999).

The ISOMAP Algorithm for Non-linear Dimensionality Reduction

Stefanie Hittmeyer

Faculty of Mathematics
University of Bielefeld
POBox 100 131
D-33501 Bielefeld
Germany

E-mail: stefanie.hittmeyer@web.de

Abstract: In many areas such as physics, engineering, astronomy and biology scientists have to analyze and visualize high-dimensional data. Thus it is important for them to understand the intrinsic degrees of freedom and the underlying geometry of such a data set. It is often convenient to find a mapping from the input space to a lower dimensional subspace such that certain aspects of the geometry of the data points are preserved. The results can be used for analysis and interpretation of the original high-dimensional data. There are many different methods for nonlinear dimensionality reduction. I will introduce the method of Isometric Feature Mapping (ISOMAP).

Introduction

We have a given set of m-dimensional data. We want to find meaningful low-dimensional structures in this data set. Is it possible for some $d < m$ to find a d-dimensional manifold which has the same local properties as the original data set?

The ISOMAP algorithm is based on a paper of J. B. Tenenbaum et al [1]. For a given $d < m$ it provides d-dimensional projections of the original data points which approximately preserve the geometric properties of the input data. These are the basic steps of the ISOMAP algorithm:

Basic Steps

1. For each data point X_i determine a set of neighboring points $N(i)$.
2. Approximate the geodesic distances between all pairs of points by using the distances between neighboring points.
3. Estimate the projections $Y_1, ..., Y_N$ of the data points $X_1, ..., X_N$ on the manifold M by minimizing the cost function

$$E(Y) = \|\tau(D_G) - \tau(D_Y)\|.$$

The Algorithm

First step: For each data point X_i determine a set of neighboring points $N(i)$.

Let d_X be a metric on the input space $\tilde{X} \subset \mathbb{R}^m$, e.g. the Euclidean metric. We compute the distances $d_X(i,j)$ between all pairs of data points X_i, X_j.
Since we want to use the local geometric properties of the data set we define a set of neighbors $N(i)$ for every X_i. There are two ways to do this:

1. Take all points within a sphere of fixed radius ϵ: $\quad N(i) := \{X_j | d(i,j) \leq \epsilon\}$
2. Take the k nearest points for a fixed k.

Define an undirected weighted graph G on the set X of all data points. $X_1, ..., X_N$ are the vertices and the edges connect the neighboring points. The edges are weighted with the distances $d_X(i,j)$ between the neighboring vertices X_i and X_j.

Second step: Approximate the geodesic distances between all pairs of points by using the distances between neighboring points.

On the graph G we compute the shortest path distances $d_G(i,j)$ between X_i, X_j. Therefore we use the Floyd-Warshall Algorithm:

Set $d_G(i,j) = \begin{cases} d(i,j) & \text{if } X_i \text{ and } X_j \text{ are connected by an edge} \\ \infty & \text{else} \end{cases}$

For $k = 1, .., N;\ j = 1, .., N;\ i = 1, .., N$
replace $d_G(i,j)$ by $\min(d_G(i,j), d_G(i,k) + d_G(k,j))$

Then $d_G(i,j)$ approximates the geodesic distance between X_i and X_j. Define the matrix

$$D_G := \{d_G(i,j)\}$$

of the shortest path distances of the original data and the matrix

$$D_Y := \{\|Y_i - Y_j\|\}$$

of the Euclidean distances of the projections on the manifold.

Third step: Estimate the projections $Y_1, ..., Y_N$ of the data points $X_1, ..., X_N$ on the manifold M by minimizing the cost function $E(Y) = \|\tau(D_G) - \tau(D_Y)\|$.

Let

$$S_G := \{d_G^2(i,j)\}$$

denote the squared shortest path distance matrix of the data points and

$$S_Y := \{\|Y_i - Y_j\|^2\}$$

the squared Euclidean distance matrix of the projections.

The problem is invariant to rotation, rescaling and translation. Thus we can modify the data points such that they are centered in the origin, i. e. the mean of every coordinate is zero. We do this by multiplying the centering matrix H with S_G from both sides. H is defined by

$$H_{ij} = \delta_{ij} - \frac{1}{N}\mathbf{1}_{ij}$$

with

$$\mathbf{1} = \begin{pmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{pmatrix}.$$

For any matrix $A \in \mathbb{R}^{N\times N}$

$$AH = \{A_{ij} - \frac{1}{N}\sum_k A_{ik}\}.$$

That means multiplication with H translates the columns of A to the geometric center. Thus

$$H^T S_G H = \{d_G^2(i,j) - \frac{1}{N}\sum_k d_G^2(k,j) - \frac{1}{N}\sum_k d_G^2(i,k) + \frac{1}{N^2}\sum_{l,k} d_G^2(l,k)\}.$$

Demand that $Y_1, .., Y_N$ are concentrated in the origin as well. Let Y denote the matrix $Y = (Y_1 \cdots Y_N)$. So we have

$$S_Y = \{\left\|(Y_i - Y_j)^T(Y_i - Y_j)\right\|\} = \{\|Y_i\|^2 + \|Y_j\|^2 - 2Y_i^T Y_j\}$$

$$= \begin{pmatrix} \|Y_1\|^2 & \dots & \|Y_1\|^2 \\ \vdots & & \vdots \\ \|Y_N\|^2 & \dots & \|Y_N\|^2 \end{pmatrix} + \begin{pmatrix} \|Y_1\|^2 & \dots & \|Y_N\|^2 \\ \vdots & & \vdots \\ \|Y_1\|^2 & \dots & \|Y_N\|^2 \end{pmatrix} - 2Y^TY$$

We want the projections to have the same local geometry as the original data, i.e. they should have the same squared distances.
$Y_1, .., Y_N$ should minimize $\left\|-\frac{1}{2}H(S_G - S_Y)H\right\|$. Define the cost function

$$E(Y) = \|\tau(D_G) - \tau(D_Y)\|$$

where

$$\tau(D_G) = -\frac{1}{2}HS_GH \text{ and } \tau(D_Y) = -\frac{1}{2}HS_YH.$$

Since $(a \ \cdots \ a)H = 0$ and $H\begin{pmatrix} a \\ \vdots \\ a \end{pmatrix} = 0$ for all $a \in \mathbb{R}$

$$HS_YH = H(-2Y^TY)H.$$

And since the Y_i are centered in the origin, i.e. $YH = Y$, we obtain

$$\tau(D_Y) = -\frac{1}{2}HS_YH = HY^TYH = Y^TY.$$

We want to calculate the minimum of the cost function

$$E(Y) = \|\tau(D_G) - \tau(D_Y)\| = \left\|\tau(D_G) - Y^TY\right\|.$$

Since $\tau(D_G)$ is symmetric, we can diagonalize it. We compute its eigenvalues $\lambda_1 > .. > \lambda_N$ and the corresponding eigenvectors $v_1, .., v_N$ with Householder transformation and the QR algorithm:

$$\tau(D_G) = \begin{pmatrix} v_1 & \dots & v_N \end{pmatrix} \begin{pmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_N \end{pmatrix} \begin{pmatrix} v_1 & \dots & v_N \end{pmatrix}^T$$

We obtain

$$E(Y) = \left\| V^T V - Y^T Y \right\|$$

with

$$V = \begin{pmatrix} \sqrt{\lambda_1} & & \\ & \ddots & \\ & & \sqrt{\lambda_N} \end{pmatrix} \begin{pmatrix} v_1^T \\ \vdots \\ v_N^T \end{pmatrix} \in \mathbb{R}^{N \times N}.$$

We get the minimum $Y \in \mathbb{R}^{d \times N}$ of $E(Y)$ by taking the largest d eigenvalues $\lambda_1, ..., \lambda_d$ and their eigenvectors $v_1, ..., v_d$:

$$Y = \begin{pmatrix} \sqrt{\lambda_1} & & \\ & \ddots & \\ & & \sqrt{\lambda_d} \end{pmatrix} \begin{pmatrix} v_1^T \\ \vdots \\ v_d^T \end{pmatrix}$$

So for $i = 1, .., N$ the projection of X_i on the d-dimensional manifold M has the form

$$Y_i = \begin{pmatrix} \sqrt{\lambda_1}(v_i)_1 \\ \vdots \\ \sqrt{\lambda_d}(v_i)_d \end{pmatrix}.$$

Since this is the minimum of a quadratic form this is an example of convex optimization. The advantage of this method is that there is an unique minimum.

Euclidean Distance Matrix

Existence of a Solution with $E(Y) = 0$

When does the problem have a solution with

$$E(Y) = \left\| \tau(D_G) - Y^T Y \right\| = 0 \ ?$$

$E(Y) = 0$ iff the matrix $\tau(D_G)$ is a squared Euclidean distance matrix for some points $Y_1, ..., Y_N \in \mathbb{R}^d$.

If D_G is the exact geodesic distance matrix this is equivalent to

$$\lambda_{d+1} = ... = \lambda_N = 0.$$

When is a matrix $T = \{t_{ij}\}_{ij}$ a squared Euclidean distance matrix?

The roots of its entries have to satisfy the four metric properties:

1. Nonnegativity: $\sqrt{t_{ij}} \geq 0$
2. Identity of elements with distance 0: $\sqrt{t_{ij}} = 0 \Leftrightarrow i = j$
3. Symmetry: $\sqrt{t_{ij}} = \sqrt{t_{ji}}$
4. Triangle inequality: $\sqrt{t_{ij}} \leq \sqrt{t_{ik}} + \sqrt{t_{kj}}$

The Euclidean metric has a fifth property [2]:

5. $\cos(\theta_{ikl} + \theta_{lkj}) \leq \cos\theta_{ikj} \leq \cos(\theta_{ikl} - \theta_{lkj})$
   and $0 \leq \theta_{ikl}, \theta_{lkj}, \theta_{ikj} \leq \pi$

where $\theta_{ikj} = \theta_{jki}$ is the angle between the vectors from point k to vertices i and j.

Obviously the first four properties are satisfied by the centered squared shortest path distance matrix $\tau(D_G)$ in the ISOMAP algorithm. If the fifth property is fulfilled too there is a lower-dimensional representation which has the same geometry as the original data.

All five properties are satisfied by the squared geodesic distance matrices of a plane, a cylinder section and a Swiss roll in the three-dimensional space. That means you find two-dimensional projections which have exactly the same geometry as the three-dimensional input points.

A sphere section satisfies the first four properties but not the fifth:
Define the angles of a triangle on the surface of a hemisphere as the angles between the tangents in the three vertices. You see that the sum of the interior angles of such a triangle is not $180°$. For the same reason the constraint to the cosines of these angles is not satisfied. Thus there is no projection from a sphere section to a two-dimensional plane which preserves all distances.

Examples

Swiss Roll

There are $N = 2000$ data points randomly put on the surface of a Swiss roll. We use the sphere neighborhood definition with radius $\epsilon = 0.23$.

The Application of the ISOMAP algorithm gives us the following first four eigenvalues of $\tau(D_G)$:

$$\begin{aligned} \lambda_1 &= 4102.07 \\ \lambda_2 &= 164.42 \\ \lambda_3 &= 4.09 \\ \lambda_4 &= 3.89 \end{aligned}$$

The root of the ratio of λ_2 to λ_1 is

$$\sqrt{\frac{\lambda_2}{\lambda_1}} = \sqrt{\frac{164.42}{4102.07}} \approx 0.2$$

It is an approximation of the proportion of the dimensions of the Swiss roll:

$$\frac{\text{height}}{\text{arc length}} \approx \frac{1}{4.89} \approx 0.2$$

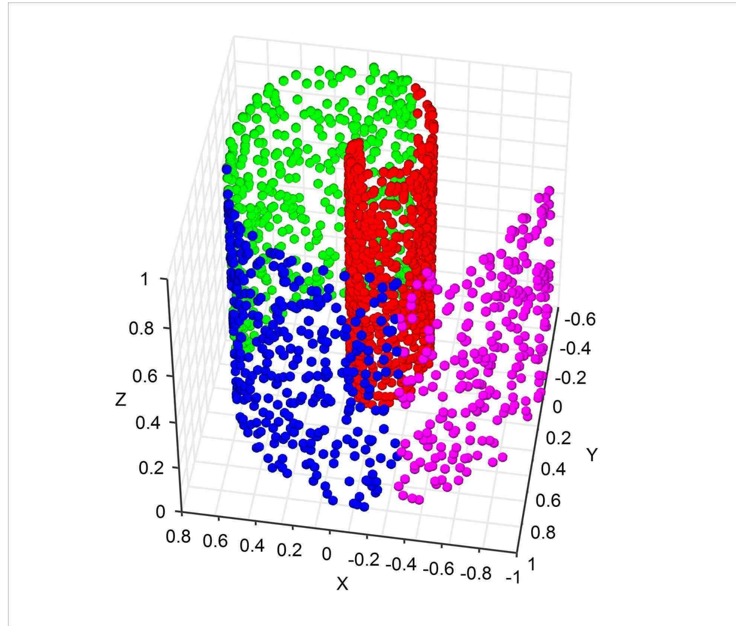


Figure 1: Swiss Roll with 2000 points

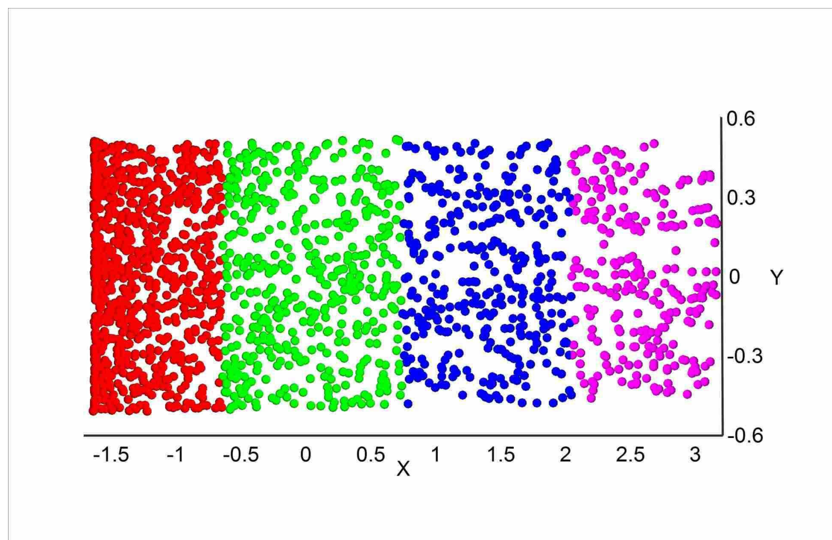


Figure 2: Projection of the Swiss roll on the two-dimensional manifold ($d = 2$)

Since the shortest path distances are only an approximation of the geodesic distances the eigenvalues λ_i with $i > 2$ are not exactly zero. Looking at the projections with $d = 3$ (figure 3) we see the fluctuation in the third dimension.

Convergence of the third eigenvalue

We chose the radius ϵ of the neighborhood spheres to decrease with higher density such that the average number of neighbors of a point is about 30. For $N \to \infty$ the shortest path distances converge to the

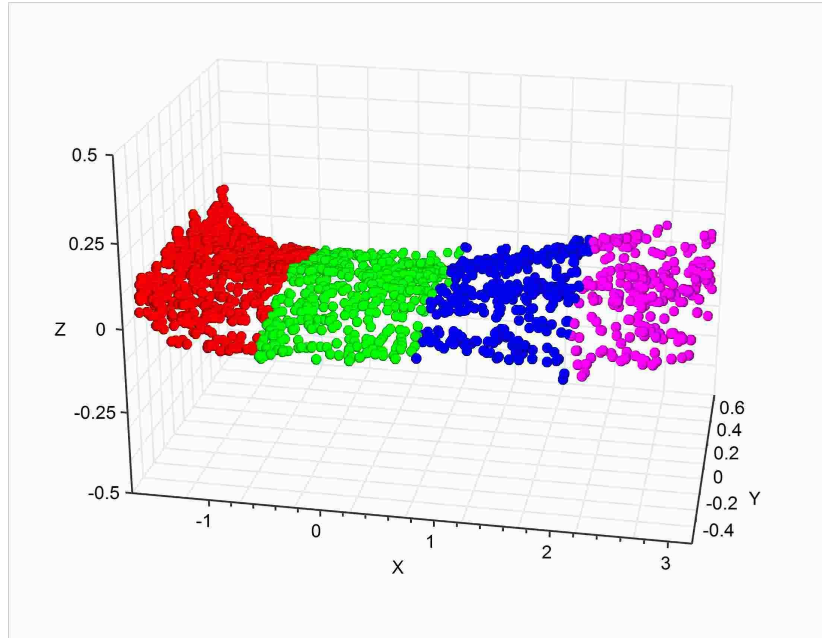


Figure 3: Projection of the Swiss roll on the three-dimensional manifold ($d = 3$)

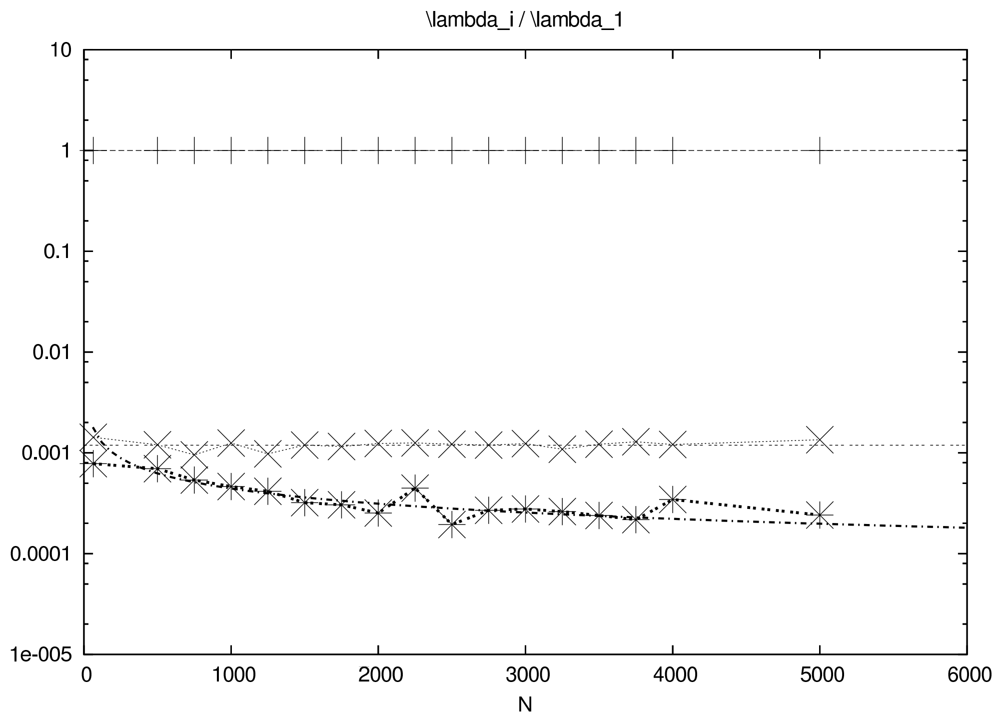


Figure 4: Swiss roll: λ_i/λ_1 for $i = 1, 2, 3$

geodesic distances. That means

$$\lim_{N\to\infty} \frac{\lambda_3}{\lambda_1} = 0.$$

In figure 4 we see that this convergence is quite slow.

Convergence of the average fluctuation

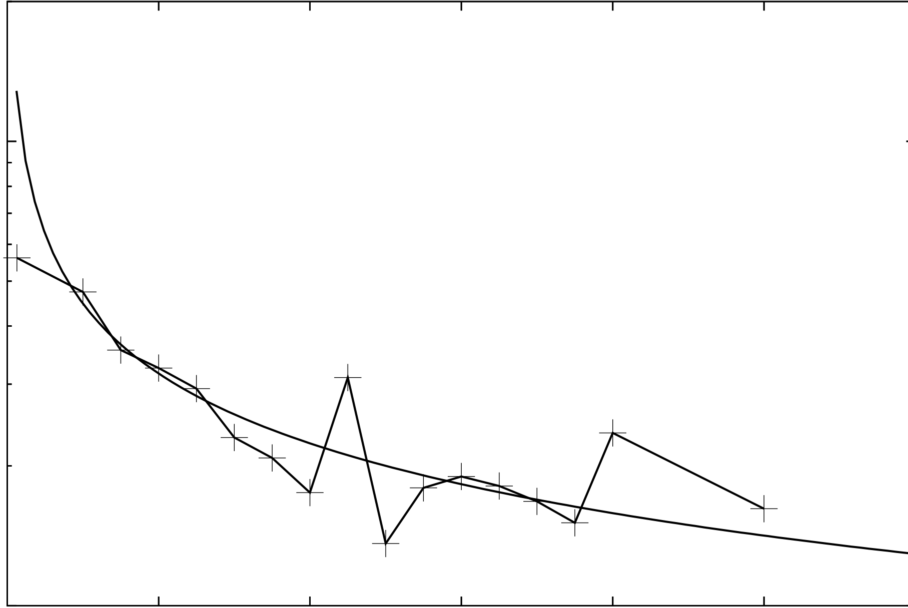


Figure 5: Swiss roll: average fluctuation of the third coordinate $\overline{Y}_3$

For increasing density the average fluctuation is converging to zero (figure 5):

$$\lim_{N\to\infty} \overline{Y}_3 = \lim_{N\to\infty} \frac{1}{N}\sum_{i=1}^{N}(Y_i)_3 = 0$$

The Memory Wall in Floyd-Warshall Algorithm

In the Floyd-Warshall algorithm we updated the whole $N \times N$-matrix D_G in every step. For $N > 4000$ the cache is to small for this matrix. Since this results in more processor-memory traffic the run time increases dramatically (figure 6).

There are different approaches in optimization of cache performance concerning the Floyd-Warshall algorithm [3]. One method is to divide the matrix D_G into $b \times b$ tiles such that b^2 fits in the cache. By reducing the working set size you divide the problem into many smaller problems.

The Droplet Model

We have two-dimensional droplets with the shape of a circle with fixed radius moving within a square (figure 7(a)). What happens if we choose another metric than the Euclidean on the original data set? Consider the metric

$$1 - A(i,j)/\pi r^2$$

where $A(i,j)$ is the overlap of the droplets i and j (figure 7(b)).

We use the sphere neighborhood definition where the average number of neighbors of a point is 30. The droplet radius is $r = 0.2$ and the circles are distributed randomly in a square with edge length 1. In figure 8 we see that the ratio of λ_3 to λ_1 is decreasing quite slow whereas the ratio $\frac{\lambda_2}{\lambda_1}$ is constantly 1

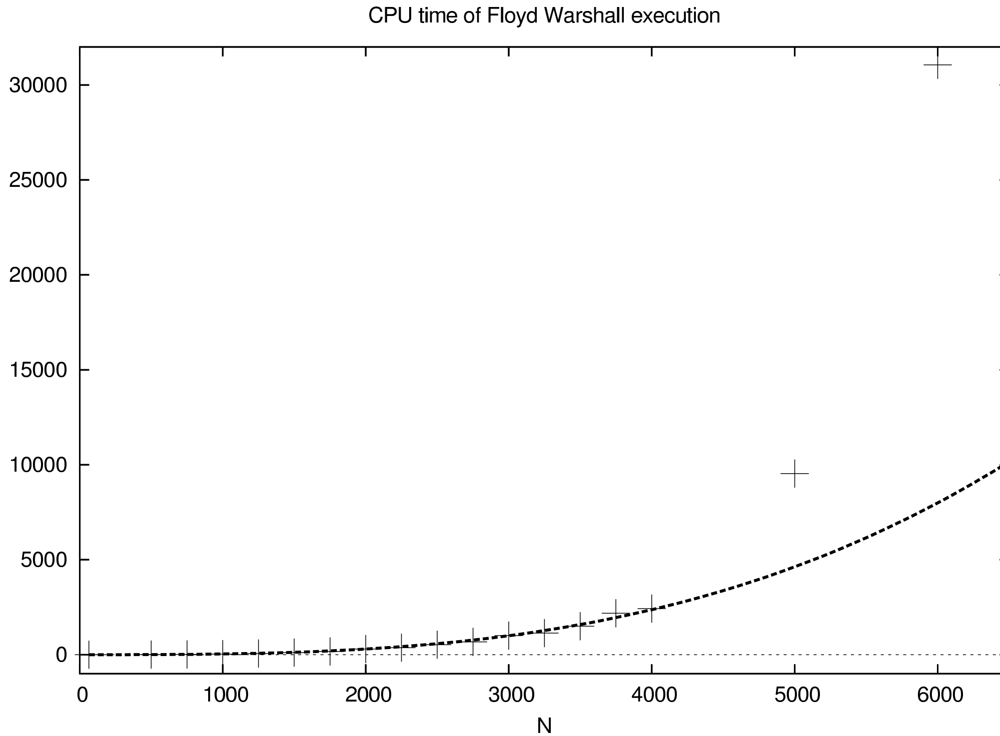
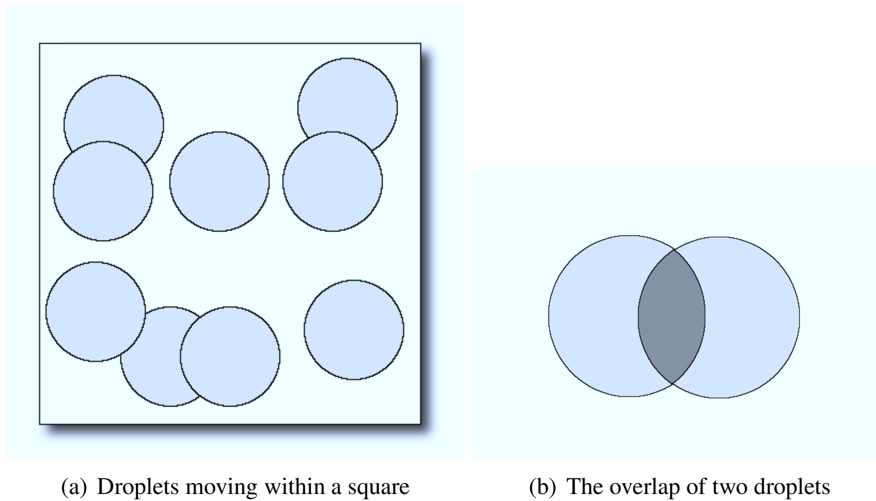


Figure 6: Swiss roll: run time of the Floyd-Warshall algorithm on an Opteron 250 in CPU seconds



(a) Droplets moving within a square (b) The overlap of two droplets

Figure 7: Droplet model

because the proportion of the edge lengths is 1.

The droplet model is a very simple model of the Ising model. The Ising model is a system of spins, which can only have the states “spin up” or “spin down” (figure 9). On this system you can also define a metric using the overlap. You get the overlap of two configurations by counting all spin positions who have the same state.

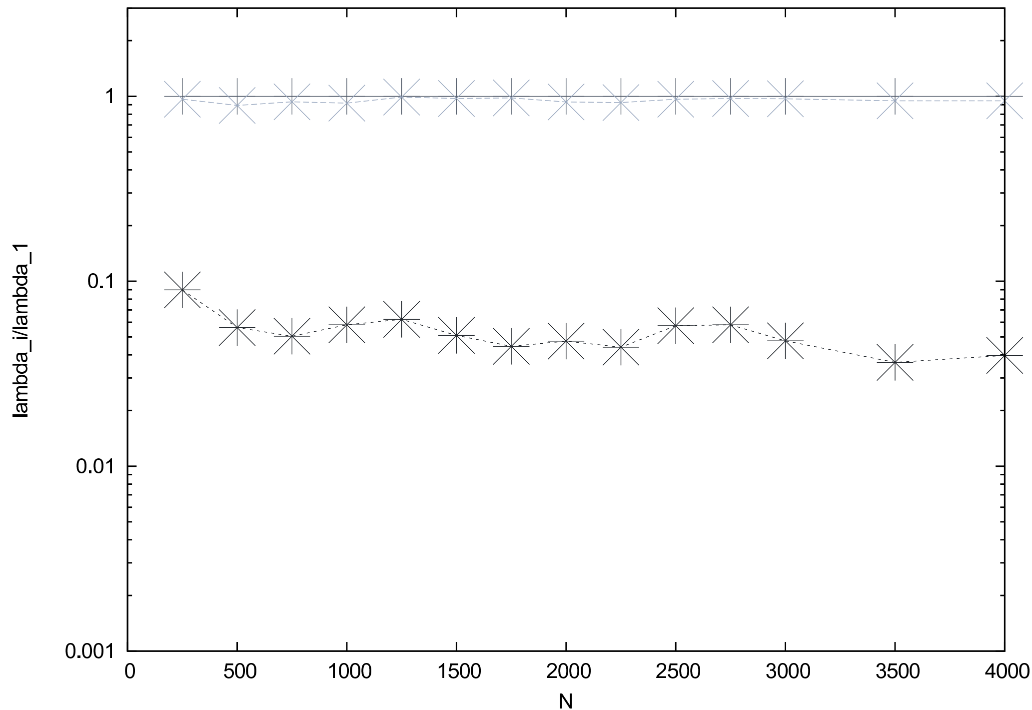


Figure 8: Droplet model: λ_i/λ_1 for $i = 1, 2, 3$

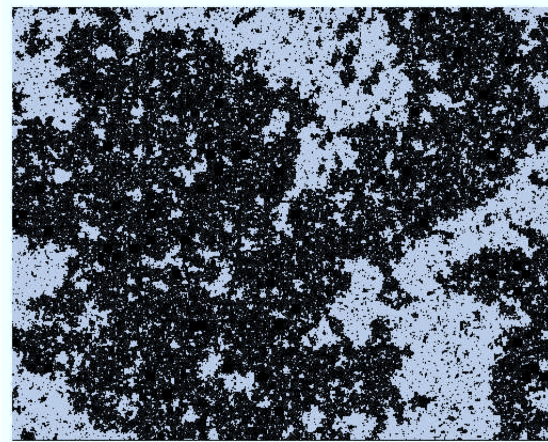


Figure 9: The Ising model

Application in Biophysics

Protein folding

A protein consists of a chain of amino acids. Each protein assumes a well-defined three-dimensional shape which is characteristic for its type (figure 10). Its function depends on this three-dimensional structure. Several diseases e.g. Alzheimer are believed to result from misfolded proteins. You can gain high-dimensional protein folding trajectory data using molecular dynamics and Monte Carlo simulations.

Biophysicists want to find meaningful low dimensional structures in the original high-dimensional data sets which can act as some kind of reaction coordinates to describe the process of protein folding. Using

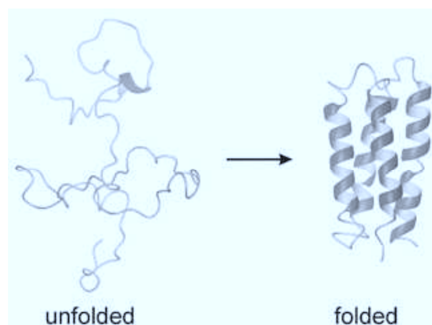


Figure 10: Protein folding

ISOMAP free-energy surfaces can be computed as a function of these low-dimensional coordinates [4].

Conclusion

During my work I observed that in spite of its slow convergence the ISOMAP algorithm provides quite good results for geometrical objects like the Swiss roll. Using it for systems with more than 4000 data points on an Opteron 250 you hit a memory wall mainly caused by the cache problem in the Floyd-Warshall algorithm.
It was interesting for me to explore the theory about Euclidean distance matrices on the one hand and the practical method of ISOMAP on the other hand. I saw that this algorithm is quite useful in technical applications such as image or handwriting recognition but there are many applications whose detailed properties are not known. Particularly in application to problems in physics there is the question of the dimension d. This cannot be answered in general but depends on the detailed properties, i.e. the physical interactions, of the problem under study.
The question arises to me if this method is able to provide new results about a system or if it is only capable of presenting properties we already know. It appears to me that you need good experience to be able to interpret the results of ISOMAP in statistical analysis.

Acknowledgments

I would like to thank Prof. Dr. Edwin Laermann who advised me about this programme and wrote the recommendation letter. My special thanks go to my supervisor Dr. Thomas Neuhaus for giving me a topic near to his current research and showing me an insight in his approach to problems and in his research methods. Further I thank Prof. Dr. Ulrich Hansmann and his group who introduced the physics of proteins and the protein folding problem to me, especially to Dr. Sandipan Mohanty and Dr. Jan Meinke for helping me with this article. I also thank my fellow guest students for their support and their humor. I had a great time!

References

1. J.B. Tenenbaum, V. de Silva, J.C. Langford, *A Global Geometric Framework for Nonlinear Dimensionality Reduction*, Science 290, 2000
2. J. Dattorro, *Convex Optimization & Euclidean Distance Geometry*, Meboo, 2005
3. M. Penner, M.K. Prasanna, *Cache-Friendly implementations of transitive closure*, Journal of Experimental Algorithmics 11, 2006
4. P. Das, M. Moll, H. Stamati, L.E. Kavraki, C. Clementi, *Low-dimensional, free-energy landscapes of protein folding reactions by nonlinear dimensionality reduction*, Proceedings of the National Academy of Sciences 103, 2006

Visualization of Star-Disc Evolution in Star Clusters with Xnbody

Thomas Kaczmarek

I. Physikalisches Institut
Universität zu Köln
Zülpicher Str. 77
50937 Köln

E-mail: kaczmarek@ph1.uni-koeln.de

Abstract:
Visualization of scientific data helps researchers understand, where the important regions are in the dataset. This is especially important in computational science, where all physical processes are represented by data and no direct feeling can be established like in a normal experiment. This article describes how one can use the xnbody visualization software to visualize nbody simulations. The focus lies on visualizing the star disc evolution in star clusters like the Orion Nebula Cluster. Star discs, also protoplanetary discs, are the remnants of the gas and dust cloud the star formed from. Studying the evolution of these star discs in star cluster can give us a idea, why astronomers do not see many star discs and how gravitation instabilities (which are potentially important for planet formation) can form.

Motivation

The physics of star discs

Formation

To understand how star discs form one has to understand how the primary star forms. Stars mainly form in so called star formation regions, which are usally embedded into nebulae consisting of interstellar gas (up to 99% of the mass) and dust. This material is the remnant of stars which have exhausted their lives in one of the biggest events in the universe, a supernova. In such a supernova, a very big star ($M_{Star} > 1.38 M_{Sun}$) explodes and throws many of its material into open space. This procedure is also important for the people on Earth, because in this "circle of life" the heavy elements (all after iron) in the chemical periodic system are produced via fusion of lighter elements.

Star forming starts now if certain parts of the cloud begin to collapse. This can also be induced by supernovae explosion. While collapsing the cloud releases its now free energy in form of radiation. This slows down the collapsing process. This happens until the point the pressure in the center of this cloud is so dense and hot that the hydrogen burning can start. In this process very energetic radiation is produced and this may slow down the collapse too.

From this point on one can say the gas cloud is now a very young star surrounded by a gas-shell. But for the observer, this star is not visible because it hides itself in its gas shell. In its further evolution this shell will begin to break up and the radiation of the star and the rotation of the shell will flatten the shell,

so that a star disc will form (s. fig. 1). Such a disc can be detected in infrared observations of the star, because they emit light in the infrared and so augment the infrared part of the star's spectrum.

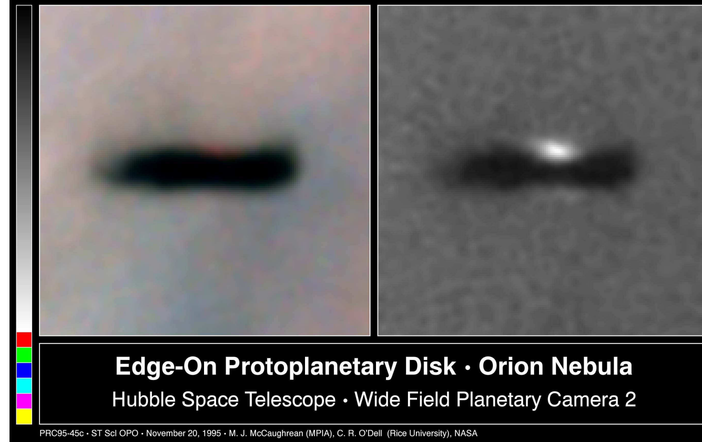


Figure 1: Image of two star discs in the ONC

Following this model, each star should have its own disc. But the observers do not see star discs around old stars. So the star discs must be destroyed in the evolution of the stars. Principally there are two different ways this may occur: first the destruction of discs via photoevaporation. Here the discs are blown away by the star via its own radiation. The second way, this one is treated in this report, is the destruction of star discs by dynamical interaction in star clusters.

Star Clusters and the ONC

Open star clusters are groups of several hundred to many thousand stars. They are gravitationally bound and the distances between the stars can be much smaller than in the sun's neighborhood. This is important for the dynamical evolution of the discs because in denser regions in space, star encounters are much more probable. So if star encounters are important for the star disc evolution, they should be examined in such star clusters.

For our simulations we have chosen the Orion Nebula Cluster (from now on ONC). This has several reasons. On the one hand it is relatively near to sun and so it can be easily studied by observers, which they have already in the past (for example see [1] or [2]). So there is much data available about the ONC and this makes it easier to model our simulations. In our simulations we model the ONC with 4000 stars with a mass range from 0.08 M_{Sun} up to 50 M_{Sun}. On the other hand the ONC is a very dense and young cluster. So one expects that in its lifetime, there will have been several star-disc encounters but not all discs should be destroyed.

Dynamical star disc evolution

To understand the dynamics of a star-disc encounter, Pfalzner et al. [3] investigated the interaction of a star and a star-disc system in an encounter with a parameter study. Therefore the following parameter ranges were varied (following [4]) mass of star 1 & star 2, perturbers curve eccentricity, inclination-angle and the disc-mass and width. After analysing this data, Pfalzner et al. were able to devise the following fit formula, which describes the disc-mass loss in dependency to the star masses and periastra:

$$\frac{\Delta M_d}{M_d} = \left(\frac{M_2^\star}{M_2^\star + 0.5 M_1^\star}\right)^{1.2} \ln\left[2.8\left(\frac{r_p}{r_d}\right)^{0.1}\right] \times \exp\left\{-\sqrt{\frac{M_1^\star}{M_2^\star + 0.5 M_1^\star}}\left[\left(\frac{r_p}{r_d}\right)^{3/2} - 0.5\right]\right\} \quad (1)$$

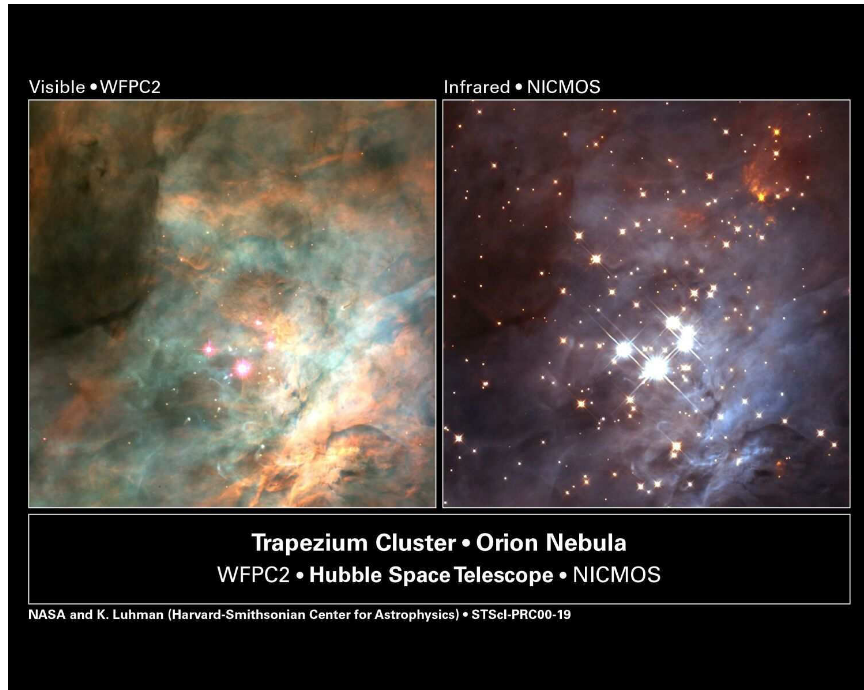


Figure 2: Hubble Space Telescope image of the Orion Nebula Cluster

Here it is important to mention that in this study the gas part of the disc was not included, so that the results are an approximation. It is also important to know that this study has been done for light discs ($M_d = 0.01 M_{star}$). This can be a good approximation because there is no evidence (yet), that there are heavy discs in the sky.

Used codes

nbody

A brief history

The nbody code has a very long history. It has been developed (and is still under development) by mainly Sverre Aarseth from the Institute of Astronomy in Cambridge, England. The development started in 1961, when Aarseth coded a tool for primitive nbody simulations. At this point the code included a force polynomial, individual time steps and force softening to avoid numerical problems. After some years, it became clear that the close encounters of stars are very important for the field and have to be resolved better. So first a two-body regularization was introduced (Kustaanheimo and Stiefel regularization, see [5]) and after some time three-body and finally chain regularization, which depend on the KS-regularization.

In these regularizations, the two body problem (in the three-body and chain regularization, one uses pairs of KS-pairs) is transformed into a four dimensional space. Here the code does not calculate the stars motion directly, instead of that the center of mass motion and the relative motion are computed so that the equations of motion have no singularity. One of these integration cycles takes more single steps but it has the advantage that fewer integration cycle have to be used with higher accuracy.

Instead of using divided differences, in newer version of nbody the Hermite Scheme is used, which got its name because it uses the Hermite interpolation formula. The basic idea is very simple and relies on employing a Taylor series of third order for both the force, $\vec{F}$, as well as its time derivative $\vec{F}^{(1)}$. The

advantage of this fourth-order scheme is that it is selfstarting. To boost the performance one can introduce hierarchical timesteps, with which it is possible to predict the coordinates and velocities for all particles needed at a given time (see Ref. [6]).

In the 1990's R. Spurzem parallized the nbody-code so that it is now useable by modern super-computers like the JUMP. Therefore several routines have been parallelized such as the integrator or energy calculation.

For studying the star-disc evolution in star clusters with the nbody code, S. Pfalzner and C. Olczak have included several routines in the nbody code which write out the important parameters of an encounter when it happens. This makes it possible to do post-processing analysis of the data and to compute the star-disc evolution over the whole star cluster.

xnbody

Major features and basic structure

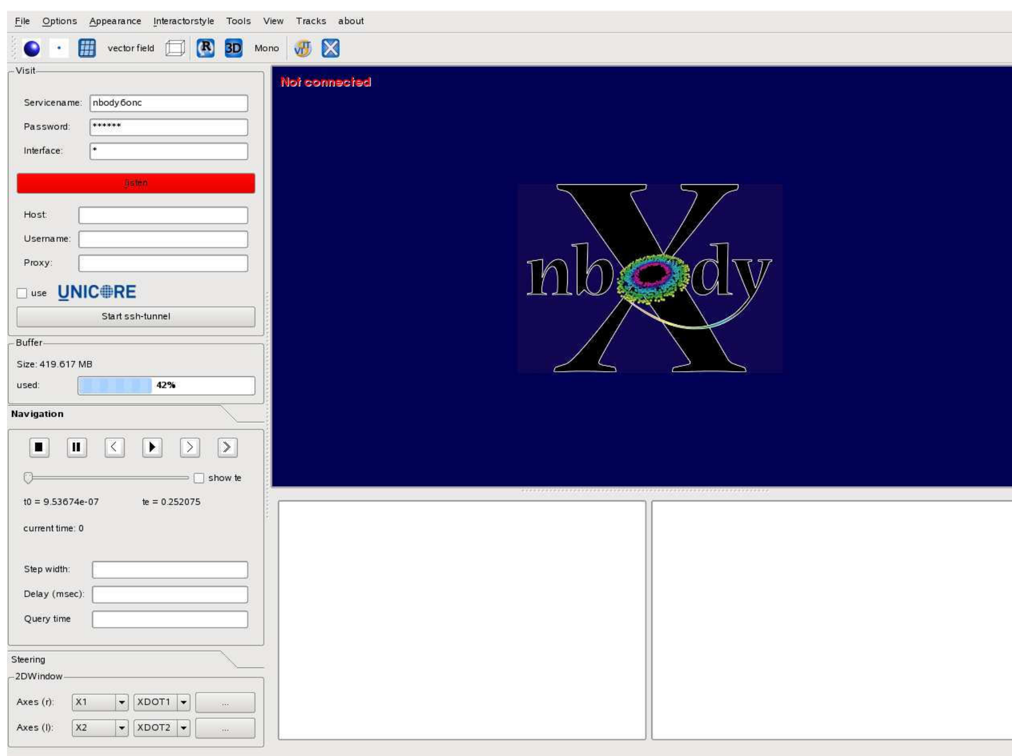


Figure 3: Xnbodys interface

The xnbody code has been developed by S. Habbinga at the FZ-Juelich and is a online visualization software for nbody simulations , this means the simulation can be visualized while running. Then it is also possible to change the simulations parameters. This is of course not so important in astrophysical simulations, but in other fields, this can be a helpful tool. In this report xnbody is only used as an post-visualizer. Xnbody runs on an workstation and requires a relatively good graphics card.

As you can see in fig. 3, xnbody uses the QT library to set up the human input interface. This has the additional advantage that in xnbody itself the QT signal-slot mechanism can be used. It enables the program to wait for human input while doing other things like playing the movie. This has the advantage that no time is spent on needless waiting for human input.

For rendering the simulation data in 3d and in 2d, xnbody uses the Visualization Tool Kit (VTK), which is freely available. It includes routines for visualizing many geometrical objects and can do several operation on datasets like interpolation or data reduction.

For communication between the simulation and the visualization the VISIT library is used, which has also been developed at the FZ-Juelich.

Communication

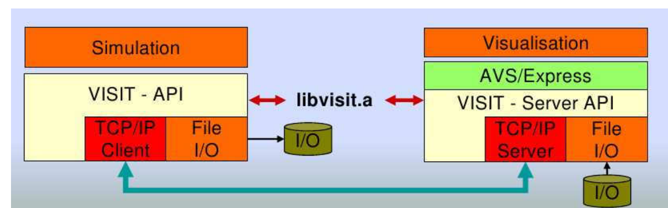


Figure 4: The VISIT communication library

As mentioned above, xnbody uses the VISIT library to communicate with the simulation. Therefore some changes have to be made in the simulation itself. A buffer (here nbodybuffer) has to be included, in which data points are stored. Once the buffer is full, the data is send to xnbody. This is essential because nbody uses individual time steps. So it is very possible that only very few particles would be sent to xnbody. This would slow down the simulation, because for the nobdy-xnbody communication the TCP/IP protocoll is used which is much slower then the internal supercomputer communication.

The data which is sent to xnbody is also stored in a buffer so that xnbody can look up the data at a later time.

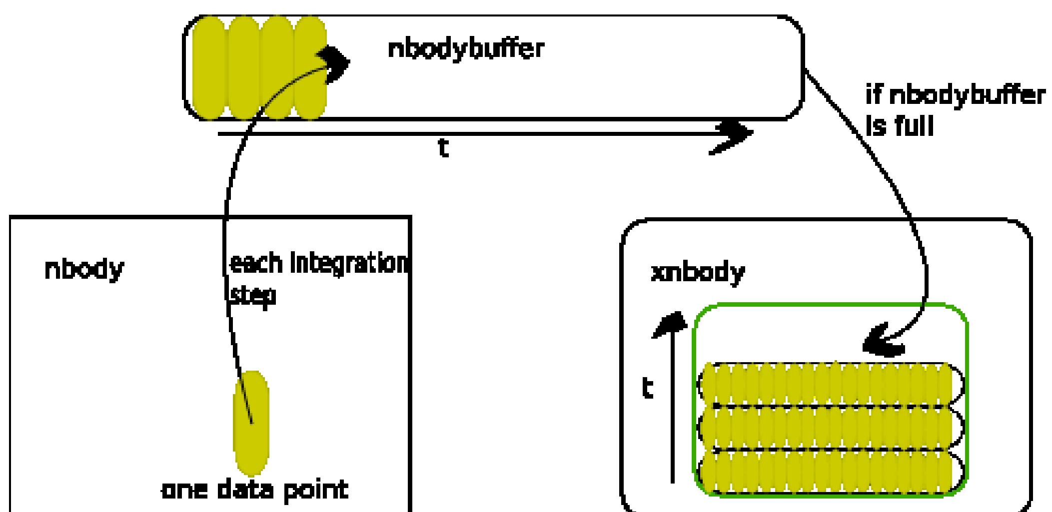


Figure 5: Buffer scheme used for nbdoy-xnbody communication

Visualization of stardiscevolution in starclusters with xnbody

Preparing nbody

Enabling the communication

Before one can start to visualize arbitrary nbody simulations with xnbody, the nbody code has to be extended by visit calls. Therefore, xnbody delivers some example nbody code, which the user can compare with his own code. I used the kompare program for this. The following files have been changed:

file	change
cmbody.F	remove one of the merged particles
co_newout.F	setup communication
commonv.h	common block for xnbody variables
input.F	save coreamount parameter
intgrt.F	send new particle data to the buffer
ksreg.F	send new KS-particle to the buffer
ksterm.F	send termination-instructions of KS-particle
nbody6.F	initialize communication and send all particles once
output.F	same as co_newout.F
remove.F	remove a particle
viscon.F	enholds communication routines
ncnbody.c	procedure for netcdf-file storage

Table 1: Additions in the nbody code

The files commonv.h, viscon.F and ncnbody.c have to be added to the nbody code completely and also in the Makefile. After this is done and the code is recompiled, nbody should be able to communicate with xnbody. A snapshot of a visualization can look like fig. 6. As you can see in the lower windows there are still problems to be solved (there should be no particles outside the cluster).

Choosing Cluster visualization cutoff radius

In a star cluster, most of the dynamics happens in the center of the cluster, so it is worthwhile to visualize this part of the cluster. The problem is that the stars have to be rendered so big to be able to see them (particularly the small ones), that in the center they overlap. This is very extreme with the biggest star ($M_{star} = 50M_{sun}$) because within the sphere of this star, there can be several stars hidden. Another problem is that xnbody and VTK computes the near and far planes of the render area with help of the stars positions. This means if we look at the whole cluster, the near plane is so far away that we cannot zoom into the center of the cluster.

To get around this a cutoff radius was defined, where particles outside are not rendered and thus do not need to be are not sent to xnbody (this could also speed up the simulation, because there is less communication needed). Therefore a additional switch was added in the inSIM file which holds the startup parameters for nbody (input.F). This parameter, from now on called coreamount, regulates the cutoff radius with respect to the Lagrange radii. The Lagrange radius with 100% mass holds all particles. The radii are calculated in lagr.F and are globally available. The relation between coreamount and the Lagrange radii is written down in table . So if one wants to render the center of the cluster, he can choose coreamount=4 or smaller.

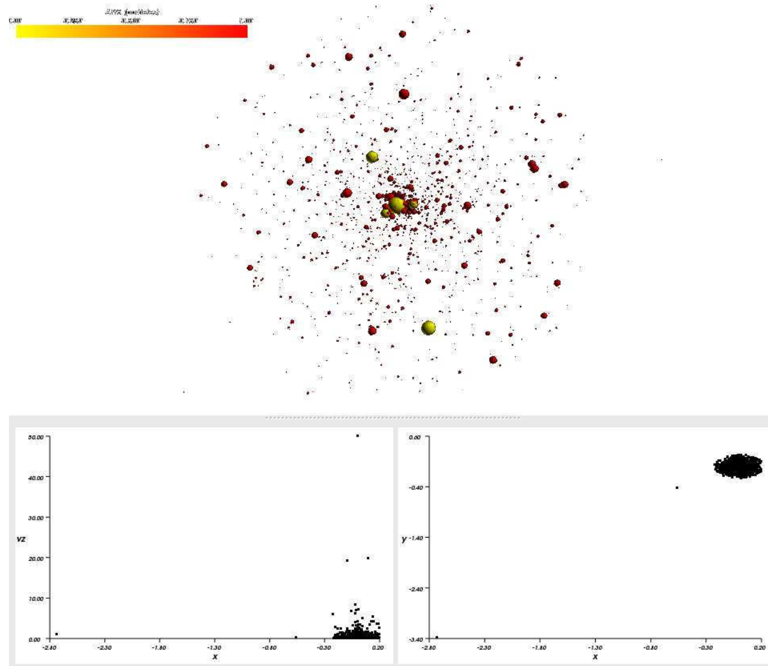


Figure 6: First snapshot, stars radii scaled by their masses ($\log(1 + M_{star})$

% use packages: array

coreamount	1	2	3	4	5	6	7	8	9	10	11
lagrange radius	1%	2%	5%	10%	20%	30%	40%	50%	70%	90%	100%

Table 2: Relation between coreamount and Lagrange radii

The determination if a particle is inside or outside the cutoff sphere is done in the `visconpart` subroutine. See figure 7 for more details.

Including star disc evolution

The previous mentioned changes in nbody are very general ones and are applicable to other problems too. The following changes are special for the problem of star-disc evolution considered here.

The first step for including the star-disc evolution is to add a unit star-disc to each star. This is done within startup by simply setting up an array with length `N` (number of stars in the simulaton at current timestep) and entries 1. Then the star-disc evolution formula (1) has to be added to the code. Therefore the subroutine `DISKEVOLUTION(I,MASSPER,MASS,ECC,PERI)` has been implemented.

Before calculating the amount of disc-mass loss in one encounter, it is necessary to identify the indices of the two stars. For this the `DEHIARCH` routine from C. Olczak is used, which delivers the right indices. This has to be done, because nbody often changes the indices during runtime. For example in KS-pairs, one particle becomes the center of mass and gets an new index. But nbody has an second index `NAME(I)` which only changes if a particle gets within a chain and becomes the c.o.m. or if it merges or if it is part of an complex hierarchy. Getting the right indices with `DEHIARCH` is very time consuming, but has to be done to get the right results.

The differences in the star disc evolution can be seen in figures 8 and 9. In figure 8 the star-disc mass is correlated with the used index and is also increasing. This can be interpreted as reading and writing

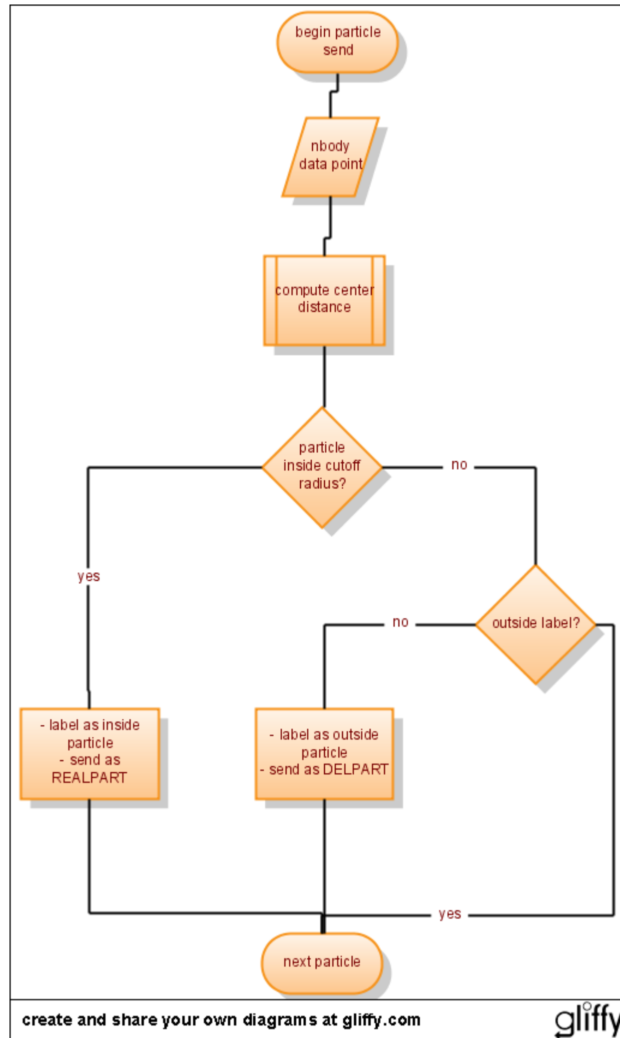


Figure 7: Inside - outside determination algorithm

in the wrong array parts. After including the upper described algorithm, figure 9 shows the anticipated behaviour of the disc mass with time: it decreases. To illustrate this, look at figure . This figure shows the disc mass evolution of particle 10. Every time the disc mass decreases after it has been constant for a while, it has been included into a strong encounter.

After this costly procedure, the star-disc mass loss is calculated with formula (1) and the disc mass loss is subtracted from the disc mass of the pertubed mass. Also here statistical failures have to be mentioned and so the subtraction is only done if the disc mass loss is minimum 3% of the momentary disc mass.

Debugging in xnbody

The star hopping problem

At this point of the work, physcially meaningful simulations were possible. But there was still a bothersome problem with the $M = 50M_{sun}$ star. It hopped all around the screen in a very unphysical way (see figures 11 to 13). The reason for this strange behaviour was the method used by xnbody to render the scene:

When xnbody renders a new timestep, it first has to get all the positions of all stars at the current timestep.

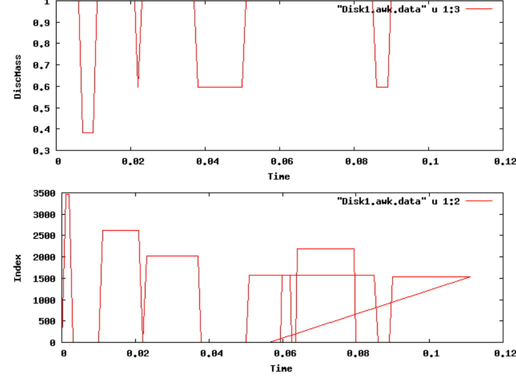


Figure 8: Disk evolution without index-correction for particle with $M_{star} = 50 M_{sun}$

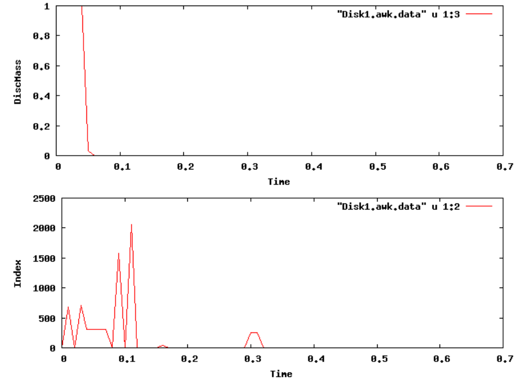


Figure 9: Disk evolution with index-correction for particle with $M_{star} = 50 M_{sun}$

Therefore, xnbody searches backwards in its own buffer for the last entry which is newer than the wanted time. From this point on, it reads in all particle data available for each star, but only if it hasn't read in data before. So the data xnbody about each particle is older than actually desired. What xnbody does then is that it predicts (like an integrator!) the positions and velocities of the particle with respect to the known data. This is done by the following formula:

$$\Delta\vec{v} = \dot{\vec{a}} \cdot \Delta t^2 + \vec{a} \cdot \Delta t \tag{2}$$

where Δv is the velocity increase, a and $\dot{a}$ the acceleration and its derivative, and Δt the time-step.

What can happen now is that a star may find itself in an strong interaction and the forces between the two stars are very strong. Then expression (2) will deliver a very big velocity increase and the star will be plotted at the wrong point according formula

$$\vec{r} = (\Delta\vec{v} + \dot{\vec{r}}) \cdot \Delta t \tag{3}$$

In reality, the strong forces will not only change the velocity amount. It will also change the velocity's direction and the gain of speed will be rather small. Formula 2 can also calculate this in the right way, but the time-step would have to be smaller. Because this is not practical (more communication slows down the simulation), a new prediction formula was introduced:

$$\Delta v = min(\dot{a} \cdot \Delta t^2 + a \cdot \Delta t, 0.1 \cdot v) \tag{4}$$

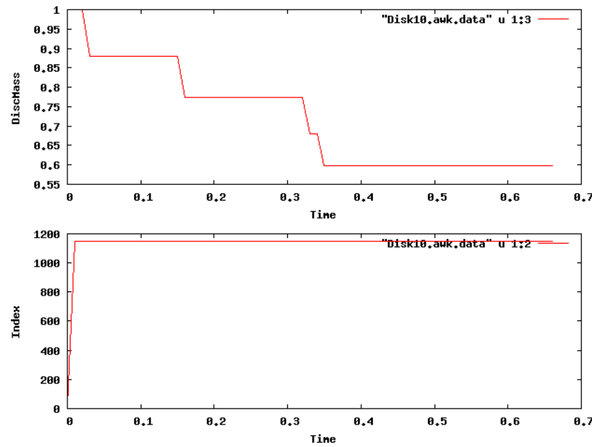


Figure 10: Disk evolution with index-correction for particle with $NAME(I) = 10$

This formula clamps the velocity increase to ten percent of the actual velocity. Of course this is only a approximation, but ensures that the visualisation is much smoother.

Conclusions and Outlook

In this report it was shown how to visualize nbody simulation with xnbody generally and how to implement a star-disc evolution model. This can be in some places very tricky (e.g. indices).

There are still several areas which should be improved in the future. First it would be a good idea to enable cutoff radius steering during runtime. This would make it possible to search for interesting regions in space during the simulation and then to concentrate the simulation on this region. Second, it would be a good idea to implement a 'pickup' routine, with which the user is able to get information about a particular star. Here it would be worthwhile to implement a plotting window especially for the disc mass evolution (e.g. like upper part in figure 9) or any other data over time.

To speed up the simulation the identification of the stars indices in each communication call should be moved to an other place in the code. One way could be to identify the indices not in each communication call but in each, let's say 20th, step. This would have the disadvantage that the time intensive `DEHIRACH` routine is still used. The other way is to change the array each time the index of a particle changes, but this would have the disadvantage that this requires many changes in the code and one has to spend a lot of time finding places where the index changes. Finally it would also be interesting to do some profiling and tracing on the code.

I want to thank the Juelich Supercomputer Center (aka ZAM) for the ability to get to know the scientific computing in general and the work in a research center. I also want to thank S. Habbinga for the help with xnbody and finally I want to thank my tutor P. Gibbon for his help in many situations.

References

1. Hillenbrand, L.A. 1997, AJ, 113, 1733
2. Rebull, L. M.; Hillenbrand, L. A.; Strom, S. E.; Duncan, D. K.; Patten, Brian M.; Pavlovsky, C. M.; Makidon, R.; Adams, Mark T., The Astronomical Journal, Volume 119, Issue 6, pp. 3026-3043
3. Pfalzner, S. et al. 2005, A&A. 437,967P
4. Olczak, C. et al. 2006, ApJ 642.1140O
5. Aarseth, S. 1971, Astrophysics and Space Science, Volume 14, Issue 1, pp.118-132
6. Makino J, Aarsteh S. 1992, PASJ: Publications of the Astronomical Society of Japan (ISSN 0004-6264), vol. 44, no. 2, 1992, p. 141-151

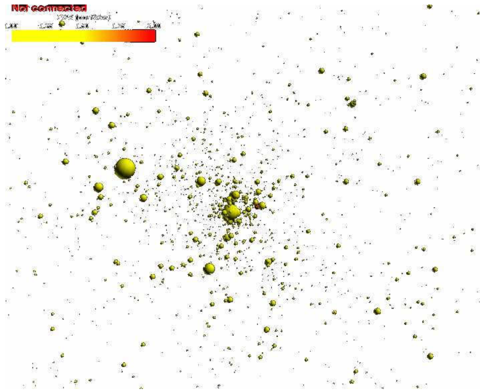


Figure 11: Star hopping frame1

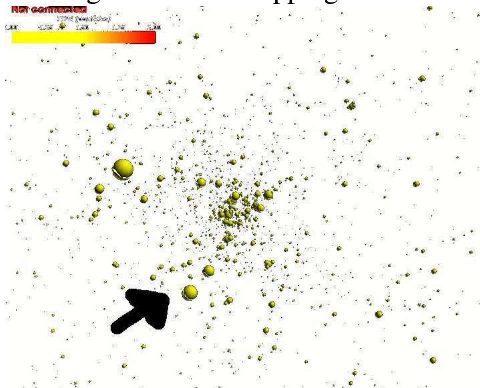


Figure 12: Star hopping frame2

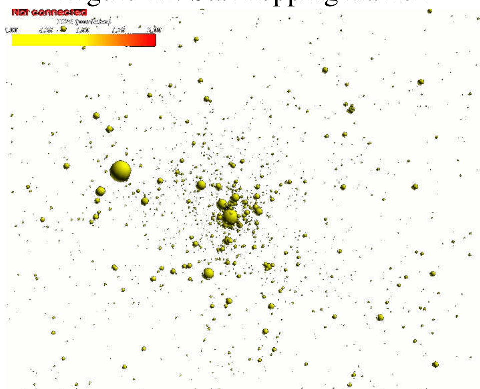


Figure 13: Star hopping frame3

Performance Optimization on Blue Gene by Optimized Task Placement

Michael Knobloch

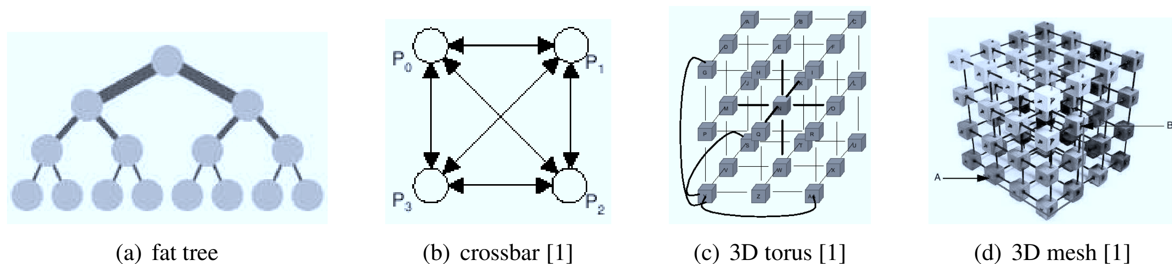
Technische Universität Dresden
Fachrichtung Mathematik
Institut für wissenschaftliches Rechnen
D-01062 Dresden

E-mail: michael.knobloch@mailbox.tu-dresden.de

Abstract: In this paper we present a new approach to task placement on Blue Gene to minimize communication time. Extending previous work ([2]), we use much more detailed communication patterns to exploit the network properties better. To obtain these communication patterns we present an MPI wrapper library.

Problem description

To understand the need of task placement, it is important to know about the communication networks of a Blue Gene supercomputer. Unlike many other supercomputers, which have fat-tree networks (Fig. 1(a)) or crossbar-switches (Fig. 1(b)), the communication network for point-to-point (p2p) communication of a Blue Gene is a 3D torus network (Fig. 1(c)), that is a 3D mesh (Fig. 1(d)) in which the opposing corners are connected. That means every node has links to six neighbouring nodes. Furthermore there is a tree network for global communication, this is not studied here.



(a) fat tree (b) crossbar [1] (c) 3D torus [1] (d) 3D mesh [1]

Figure 1: Network Types

Since the p2p communication network is the torus network, we will focus on that one. It is obvious that the distance of two nodes depends on their positions in the torus, the maximum distance is $\frac{x}{2} + \frac{y}{2} + \frac{z}{2}$. So the placement of the tasks on the torus has influence on the communication time.

There are several possibilities for task placement (refer to [1], p. 217ff). The first way is to use the standard mapping XYZT, where X,Y,Z donates the x, y and z coordinates on the torus and T denotes the processor id (only significant in virtual node (VN) mode, i.e. both processors on a node are used for computation, but share the available memory). The mapping is of the following form:

1. task 0 is mapped to (0,0,0,0), task 1 to (1,0,0,0) and so on until the maximum x value is reached
2. increment the y value

3. perform steps 1 and 2 until the maximum y value is reached
4. increment the z value
5. perform steps 1, 2 and 3 until the maximum z value is reached
6. increment the t value (a value of 1 means that the 2nd processor on the node is used), perform steps 1 to 5 until all ranks are mapped

In applications with mostly nearest neighbour communication (and using VN mode), it might be a good start to change the default mapping from XYZT to TXYZ, so that neighbours are mapped on the 2 CPUs of the same node. That could have considerable influence on the performance of the application.
It is also possible to specify an explicit mapfile. In this file the torus coordinates of each rank have to be given in the form XYZT. This method should be used if the best mapping of the application on the torus is known or the mapfile can be generated by external programs. We will present such a tool in this paper.
The last possibility is an application internal method. It is to use MPI Topologies, a very mighty feature in the MPI Standard ([4], p. 177ff). The main idea is to create communicators corresponding to the topology of the user application and which are able to reorder the ranks of the tasks (with functions like `MPI_Cart_create`). On Blue Gene (and maybe other machines with special network topologies) the MPI library will try to map (or reorder) the tasks on the torus so that they match the given network topology. For applications with a communication scheme that is mappable on the torus, it is probably the best way to start with that method and see how the communication improves. That version also has the advantage of portability. The disadvantage of this method is that for existing applications without Cartesian communicators a code change is required.

BG/L performance characteristics

To determine how to map the tasks on the torus, we will have to understand the properties of the 3D torus network.

First, we will have a look on the influence of the latency.

Latency is the time taken for a packet of minimal size to be sent by a node, travel and be received by another node.

If we consider the latency as a function of the Manhattan metric ("hop count"), i.e. the minimal distance of two positions on the torus, we see that the result is a nearly linear function, with every step it increases by about a 100 nanoseconds (Fig. 2).

The second factor we have to take into account is the bandwidth.

Bandwidth denotes the number of bytes that are conveyed per unit of time.

To understand the bandwidth behaviour of the 3D torus network we analyse the messages that are sent. Each link delivers up to 2 bit = 0.25 byte per CPU cycle. The package sizes are multiples of 32 bytes, up to 256 bytes. The first 16 bytes of each package contain destination, routing information and a header. That means that there is a maximum of 240 bytes of user data in each packet. If more data has to be sent, multiple packages have to be created. Furthermore there are 14 byte crc (cyclic redundancy check) data transmitted for every 256 byte. That results in a network efficiency of $\eta = 240/270 = 89\%$ or $\eta * 0.25$ byte/cycle $\hat{=}154$ MB/s at 700 MHz clock frequency ([3]). As mentioned above, each node has 6 links to neighbouring nodes, so that for bi-directional communication, i.e. incoming and outgoing links are used, a maximum bandwidth of 1800 MB/s is possible. So what can we get on BG/L? As we see in Fig. 3, a maximum bi-directional bandwidth of about 900 MB/s can be achieved. this is due to processor

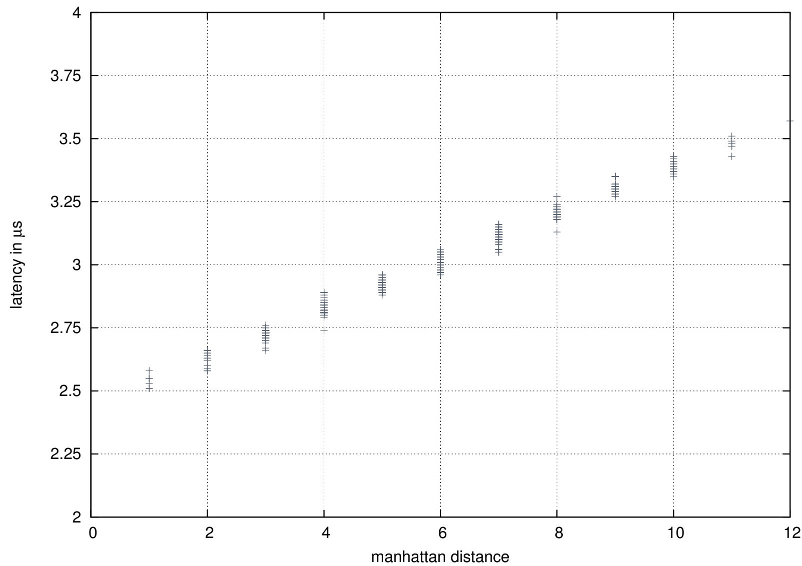


Figure 2: Latency – 8x8x8 torus

limitations. So even if you can use all six links at a time, only the maximal bandwidth of three links can be achieved. More on this can be found in [7].
The bandwidth that can be achieved depends also on the sending protocol. There are two possibilities.

When using the **eager** protocol, sending starts directly after the function call. In **eager** mode there is static routing, i.e. each package of a message goes the same way on the torus

The **rendezvous** protocol requires a "handshake" between sender and receiver after the function call, i.e. communication only starts when both communication partners are ready. In **rendezvous** mode there is adaptive routing, i.e. each package of a message can take a different way on the torus.

Both protocols always use a shortest path for routing, that means that if you send a message in rendezvous mode in one dimension, each package will go the same way, since there is only one shortest path. In Fig. 4 we see that the bandwidth curves are independent from the position of the receiver on the torus. This is not what we expected since in former work it was documented that adaptive routing starts with the first link, so you can get two times the bandwidth if you send in 2D and three times the bandwidth if you send in 3D (refer to [2], p. 491). This is not true with the current version of the BG/L software. There still is adaptive routing, but every package of a message leaves the sender over the same first link and then takes some shortest path on the torus.

On Blue Gene, the eager limit can be set by the user by setting the variable `BGLMPI_EAGER=xxx`, where `xxx` is the largest message size for which the eager protocol is used. The default eager limit on JUBL, the 8 rack Blue Gene/L at the Jülich Supercomputing Centre, is 1000. But it is important to know that, independently from the eager limit, messages that fit in one package, i.e. with a maximal size of 240 bytes, are always sent directly. Fig. 5 shows that using the default eager limit is usually not the best choice. For each application at least three different runs should be made, one with the default eager limit, one with eager limit nearly infinity (which has the highest bandwidth for medium size), so only eager is used and one with a eager limit of zero, i.e. only rendezvous is used. Then the one which gives the best performance to the application should be used. In Fig. 5 we see "bends" in some graphs at eager limit zero and 1000. The minor increase of the slope was expected since these are the limits where the protocol is switched from eager to rendezvous, but we can not explain the bends. But it is obvious that we get a very high bandwidth only for the large messages of about 16 kByte and larger, whereas we have very small bandwidth for very small messages. In this case the impact of the latency is dramatically higher,

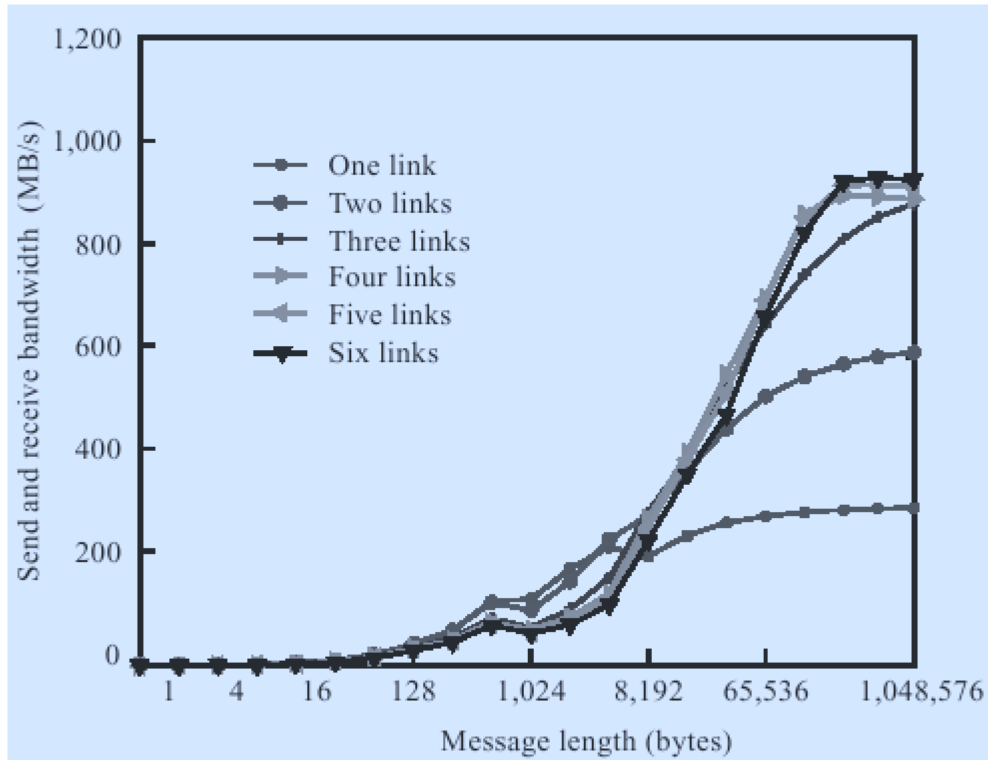


Figure 3: Bi-directional communication (from [3])

so that tasks which exchange a lot of small messages should be mapped very close together.

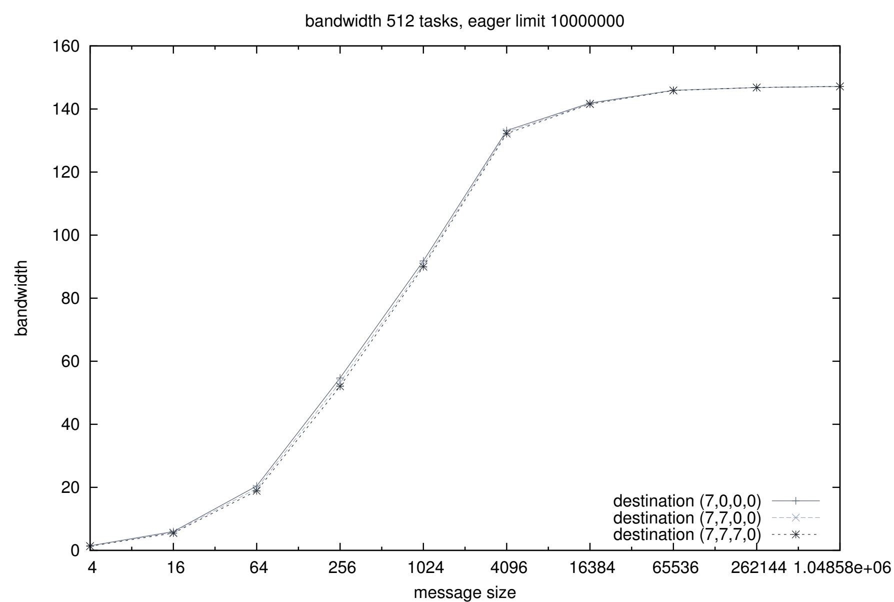


Figure 4: Bandwidth on 8x8x8 torus in 3 dimensions

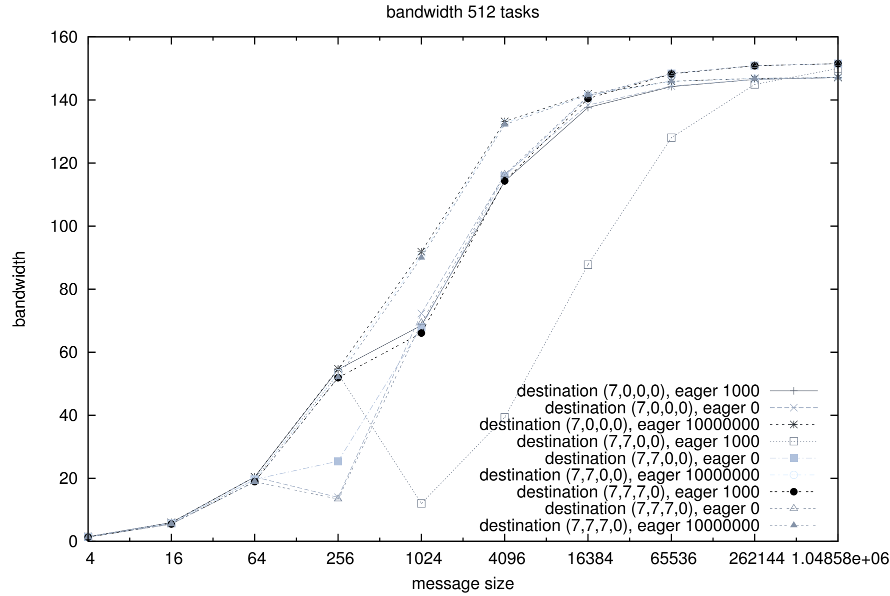


Figure 5: Bandwidth on 8x8x8 torus in 3 dimensions with different eager limits

MPI profiling library

In this section we will describe how we obtain the application communication patterns which we need to optimize the task layout.

The MPI profiling mechanism

According to [4], each MPI library has to have a profiling interface, so that every MPI function `MPI_xxx` is also callable via `PMPI_xxx`. This means that everybody is able to write wrappers for MPI functions without any knowledge of the internals of the library. Inside the wrapper the required measurements can be done and then the `PMPI_xxx` function can be called.

We provide C and Fortran wrappers for `MPI_Init`, `MPI_Finalize` and the complete set of the point-to-point (p2p) communication routines, i.e. `MPI_Send`, `MPI_Bsend`, `MPI_Rsend`, `MPI_Ssend`, `MPI_Sendrecv`, `MPI_Sendrecv_replace` and the corresponding nonblocking versions `MPI_Ixxx`.

General overview

The library in the current version can perform two tasks. It can print the actually used mapping in a file called Mapfile and it can generate a complex communication histogram, i.e. it counts the number of messages of different, user specifiable sizes and the average size of each of these “bins”. Like the IBM High Performance Computing Toolkit (HPCT) it is controlled by environment variables whose names are closely related to the ones of the HPCT (refer to [6]). Due to the limited memory on BG/L and for performance reasons, all IO-operations are performed with MPI-IO (for more information on that refer to [5]). As with every other profiling library, the library has to be linked before the MPI library.

Mapfile generation

If the environment variable `TRACE_MAPFILE` is set, then the mapping at the time of the call of `MPI_Init` is written to disk as an ASCII-file. This is demonstrated in listing 1. Therefore each task

writes its position at the appropriate position. To obtain the position of a task, Blue Gene provides a `personality`-library, which includes functions to get the coordinates on the torus as well as the processor-id. More on this can be found in [1], p. 331ff.

```
/* write the mapfile if TRACE_MAPFILE is set */
  if((env = getenv("TRACE_MAPFILE"))!=NULL) {

    /* get personality of the task */
    rts_get_personality(&personality, sizeof(personality));

    position[0] = BGLPersonality_xCoord(&personality),
    position[1] = BGLPersonality_yCoord(&personality),
    position[2] = BGLPersonality_zCoord(&personality),
    position[3] = rts_get_processor_id(),

    /* open the file */
    MPI_File_open(MPI_COMM_WORLD, "mapfile",
    MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &file);
    /* calculate the offsets */
    offset = 25*myrank;
    offset_nl = 25*myrank+24;
    /* generate the output */
    sprintf(data[0],"%6d",position[0]);
    sprintf(data[1],"%6d",position[1]);
    sprintf(data[2],"%6d",position[2]);
    sprintf(data[3],"%6d",position[3]);
    data_nl = '\n';
    /* write the data to file */
    MPI_File_write_at(file, offset, data, 24, MPI_CHAR, &status);
    MPI_File_write_at(file, offset_nl, &data_nl, 1, MPI_CHAR, &status);
    /* close the file */
    MPI_File_close(&file);
  }
```

Listing 1: Generating the mapfile with MPI-IO

Communication pattern

The most important function of the library is the communication profiling. It can be enabled by setting the environment variable `TRACE_COMM_MATRIX`. The value of this variable must be of the form $bin_0, \ldots, bin_{\#bins-1}$ where $bin_i \in \mathbb{N}$, $i = 0, \ldots, \#bins - 1$ and bin_i is the upper bound of the bin (in bytes), so if you want to collect the message sizes 480 Byte, 1kByte, 16 kByte and 512 kByte you should set (in a bash) `export TRACE_COMM_MATRIX=480,1024,16384,524288`. The overhead of each communication call is quite low since most of the work is done during `MPI_Finalize`. There a binary file is written with the following layout:
[Header]
$size\ \#bins\ bin_0, \ldots, bin_{\#bins-1}$
[Communication histograms]

# Racks	Mode	# Processors	Memory consumption	File size
1/2	CO	512	#bins∗4 kB	#bins∗2+1 MB
1	CO	1024	#bins∗8 kB	#bins∗8+4 MB
1	VN	2048	#bins∗16 kB	#bins∗32+16 MB
8	CO	8192	#bins∗64 kB	#bins∗512+256 MB
8	VN	16384	#bins∗128 kB	#bins∗2+1 GB
16	VN	32768	#bins∗256 kB	#bins∗8+4 GB
64	CO	65536	#bins∗512 kB	#bins∗32+16 GB
64	VN	131072	#bins∗1 MB	#bins∗128+64 GB

Figure 6: Memory consumption and file size

$$\begin{array}{cccccc}
mc_{0,0,0} & \cdots & mc_{0,0,size-1} & as_{0,0,0} & \cdots & as_{0,0,size-1} \\
\vdots & & \vdots & \vdots & & \vdots \\
mc_{0,\#bins-1,0} & \cdots & mc_{0,\#bins-1,size-1} & as_{0,\#bins-1,0} & \cdots & as_{0,\#bins-1,size-1} \\
\vdots & & & & & \\
\vdots & & & & & \\
mc_{size-1,0,0} & \cdots & mc_{size-1,0,size-1} & as_{size-1,0,0} & \cdots & as_{size-1,0,size-1} \\
\vdots & & \vdots & \vdots & & \vdots \\
mc_{size-1,\#bins-1,0} & \cdots & mc_{size-1,\#bins-1,size-1} & as_{size-1,\#bins-1,0} & \cdots & as_{size-1,\#bins-1,size-1}
\end{array}$$

[HPCT like communication matrix]

$$\begin{array}{ccc}
comm_{0,0} & \cdots & comm_{0,size-1} \\
\vdots & & \vdots \\
comm_{size-1,0} & \cdots & comm_{size-1,size-1}
\end{array}$$

Here $size$ is an integer and denotes the size of `MPI_COMM_WORLD`, $\#bins$ and $bin_i,\ i = 0, \ldots, \#bins-1$ are integers and defined as above. $mc_{x,y,z}$ denotes the number of messages sent from task x to task z in bin y as integer, $as_{x,y,z}$ the average size (in bytes) of the messages sent from task x to task z in bin y as float and $comm_{x,y}$ the sum of all bytes sent from task x to task y as float.

The memory consumption per task is about $\#bins * (size * \text{sizeof(int)} + size * \text{sizeof(float)})$, since each task saves the number of messages sent and the average size of them for each bin and each other task. Fig. 6 gives the memory consumption and the file size for some commonly used partitions.

Further work

To be able to use our new functions together with the other libmpitrace measurements (within one run), a first step would be to include the wrappers in the IBM HPCT, so far more information can be collected in a single run without re-linking.
A limitation is, that only communication via the communicator `MPI_COMM_WORLD` is traced. It is possible to improve the library so it can handle multiple communicators.
Another possibility of improvement is to monitor changes in the current mapping, for example if a communicator is created which can reorder the ranks of the MPI tasks (like `MPI_Cart_create`). That way we could understand the way the underlying MPI library works.

Example Application – sweep3d

Sweep3d (http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/) is a 3D Discrete Ordinates Neutron Transport application. Figure 7 shows the communication scheme of sweep3d. As we see it has

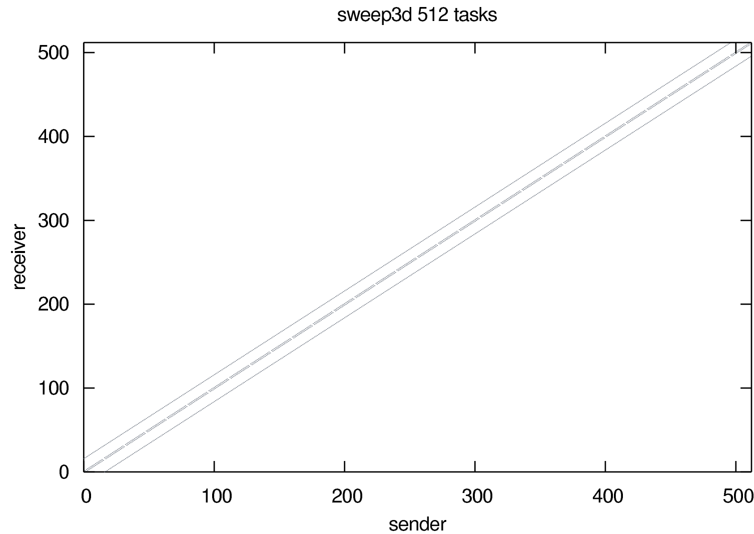


Figure 7: sweep3d – 512 tasks

mostly nearest neighbour communication, so it can be optimally mapped to the torus. We have also made traces with the HPCT. If `TRACE_ALL_EVENTS` is set, it generates an overview of which MPI functions were called and how much time was spent in them (listing 2).

```
MPI Routine                  #calls     avg. bytes      time(sec)
-----------------------------------------------------------------
MPI_Comm_size                     1            0.0          0.000
MPI_Comm_rank                     1            0.0          0.000
MPI_Send                       9984         7680.0          3.768
MPI_Recv                       9984         7680.0         16.928
MPI_Bcast                         4          537.0          0.000
MPI_Barrier                       3            0.0          0.000
MPI_Allreduce                    32            6.5          6.097
-----------------------------------------------------------------
total communication time = 26.792 seconds.
total elapsed time       = 134.195 seconds.
```

Listing 2: sweep3d hpct trace – MPI function overview for rank 0

In sweep3d there is a lot of blocking communication and very few collective communications. Communication time is about 20 % of the total time.
The HPCT also generates message size distribution statistics (listing 3).

```
Message size distributions:

MPI_Send                 #calls    avg. bytes      time(sec)
                           9984        7680.0          3.768

MPI_Recv                 #calls    avg. bytes      time(sec)
                           9984        7680.0         16.928
```

```
MPI_Bcast                    #calls    avg. bytes      time(sec)
                                  1           4.0          0.000
                                  1          32.0          0.000
                                  1          64.0          0.000
                                  1        2048.0          0.000

MPI_Allreduce                #calls    avg. bytes      time(sec)
                                 12           4.0          6.096
                                 20           8.0          0.000

-----------------------------------------------------------------------------
```

Listing 3: sweep3d hpct trace – message size distribution for rank 0

In sweep3d all p2p messages have the same size of 7.5 kByte, so sweep3 in not a good candidate for the improved heuristic, see heuristic improvements on page 87
If `TRACE_ALL_TASKS` is set, a communication summary where the communication time and the position of all task is printed. If `TRACE_SEND_PATTERN` is set, the average number of hops to the communication partners is printed additionally. They are calculated as following:

$$AverageHops = \frac{\sum_i Hops_i * Bytes_i}{\sum_i Bytes_i}$$

where $Hops_i$ is the distance between the processors for ith MPI communication and $Bytes_i$ is the size of the data transferred in this communication. If the communication processor pair is close to each other in the coordinate, the $AverageHops$ value will tend to be small ([6], p. 7). So it is the aim of our heuristic to decrease this value for applications with mainly small messages.

```
Communication summary for all tasks:

  minimum communication time = 26.520 sec for task 5
  median  communication time = 29.634 sec for task 410
  maximum communication time = 31.098 sec for task 356

taskid  xcoord  ycoord  zcoord  procid  total_comm(sec)  avg_hops
     0       0       0       0       0           26.792      1.50
     1       1       0       0       0           27.737      1.33
     .
     .
     .
   509       5       7       7       0           27.361      1.33
   510       6       7       7       0           27.727      1.33
   511       7       7       7       0           26.825      1.50
```

Listing 4: sweep3d hpct trace – communication summary

For more information on the HPCT refer to [6].

With all the information we got through the traces and the bandwidth measurements we can determine the minimal time each task needs for sending:

$$t_{min} = \sum_{i=0}^{\#bins-1} \sum_{j=0}^{size-1} \frac{m_{i,j} * as_{i,j}}{bw_i}$$

In this case the maximum over all ranks is 1.03 s, i.e. under perfect conditions you could save up to 2.70 s or 72 %.

Example Application – smg2000

Smg2000 (http://www.llnl.gov/asci/platforms/purple/rfp/benchmarks/limited/smg/) is a parallel semicoarsening multigrid solver for linear systems.

As we see in figure 8, smg2000 has a more complicated communication scheme than sweep3d. It is a very regular multidimensional pattern which should map well on the torus.

In listing 5 we see that this time non blocking communication is used and nearly all the communication time is spent in the `MPI_Waitall`. This is due to the implementation of the MPI library on Blue Gene/L, that all the work of non blocking communication is done in the `MPI_Wait`. In smg2000 a lot

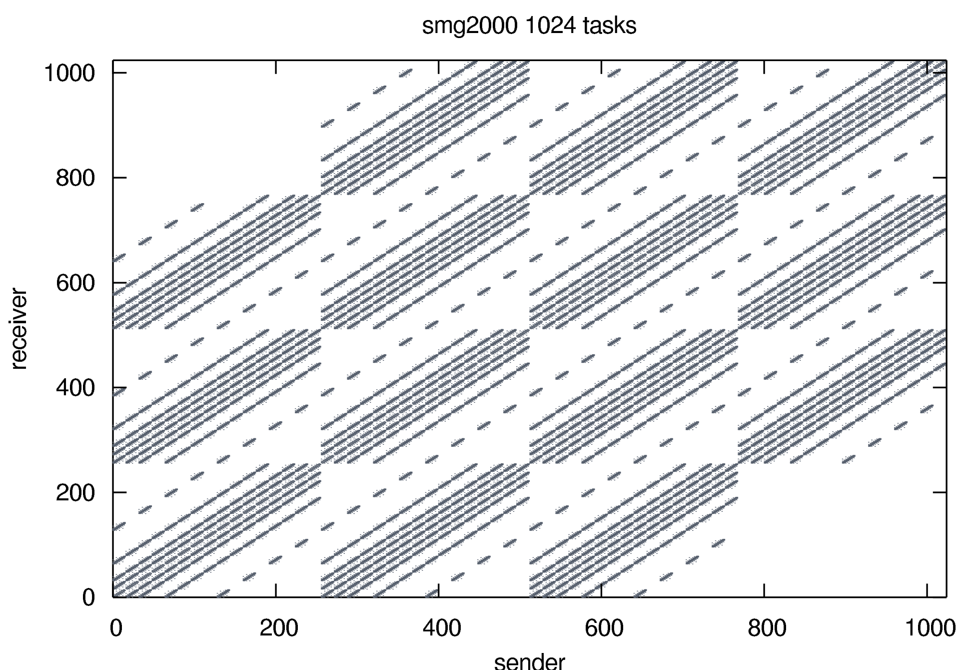


Figure 8: smg2000 – 1024 tasks

of different sized messages are transmitted, most of them smaller than 480 bytes. That means they would fit in one or two packets. That makes smg2000 an ideal candidate for the improved heuristic (see page 87).

```
Message size distributions:

MPI_Isend                    #calls     avg. bytes      time(sec)
                              20594           4.0          0.045
                               6179           8.0          0.016
                               6227          16.0          0.029
```

```
MPI Routine                  #calls     avg. bytes      time(sec)
--------------------------------------------------------------------
MPI_Comm_size                 23646            0.0          0.003
MPI_Comm_rank                 27114            0.0          0.004
MPI_Isend                    105029          626.0          0.723
MPI_Irecv                    104829          698.0          0.125
MPI_Waitall                  104509            0.0         13.237
MPI_Barrier                       1            0.0          0.000
MPI_Allgather                     1            4.0          0.000
MPI_Allgatherv                    1           28.0          0.000
MPI_Allreduce                    12            8.0          0.630
--------------------------------------------------------------------
total communication time = 14.724 seconds.
total elapsed time       = 25.537 seconds.

--------------------------------------------------------------------
```

Listing 5: smg2000 hpct trace – MPI function overview for rank 0

```
 7038        32.0      0.035
 8145        64.0      0.043
 9646       127.8      0.061
12818       253.7      0.079
15929       438.5      0.085
 9915       823.5      0.077
 3797      2039.8      0.067
 2688      4073.3      0.067
 1579      8140.1      0.055
  209     15678.5      0.019
  228     31582.3      0.026
   29     48587.0      0.010
    3    131072.0      0.006
    5    262144.0      0.004
```

Listing 6: smg2000 hpct trace – message size distribution for rank 0

Heuristic task placement

In this section we describe some ideas for heuristic task placement and the implementation we have done.

A simple heuristic

The main idea for a simple heuristic is the following [2]:

1. Map domain $i = 0$ to arbitrary location (x, y, z, t)
2. Map all domains j with $comm_{i,j} > 0$ or $comm_{j,i} > 0$ to a neighbouring location
3. Repeat previous steps for $i = 1, \ldots, size - 1$ (if not yet mapped)

Implementation details

In step one we choose the position on the torus randomly. That can be quite expensive if many domains are already mapped, so that most locations are already occupied. So we only try ten times randomly and then, if no free position was found, we are looking for the first free node on the torus in XYZT order. We will have to check whether the ten tries are enough for a good mapping or if that value should be increased. A big challange was the way to find the neighbouring nodes in step two. The method we use is the so called iterative deepening search (*ids*), that is a mixture of breadth first search and depth first search. With *ids* you start at iteration limit 0 and do a depth first search until you find a free node on the torus or the depth reaches the iteration limit. In the latter case the iteration limit increases by one and you start again. Listing 7 gives a pseudo-code implementation. This method has the advantage of a low memory consumption compared to breadth first search, since you only need to save the current path. It also guarantees that the nearest neighbours are found. And, unlike a real depth first search, it can not get lost in circles (of which we have a lot, considering the torus as a graph).

```
/* iterative deepening search
 - a recursive function to find vertices in graphs */
ids(torus_pos source, torus_pos position, int iteration) {
  if(iteration==0) {
    /* max. depth reached -> pos free? */
    if(occupied(position)) return source;
    else return position;
  }
  forall(neighbour of position) {
    /* max depth not yet reached -> go on searching */
    new = ids(source, neighbour, iteration -1);
    /* if free pos found -> return it */
    if(new!=source) return new;
  }
  /* no free pos found -> return source */
  return source;
}

findNeighbour(torus_pos source) {
  iteration = 0;
  /* max. (x+y+z)/2 iterations needed,
  since then all positions are visited */
  while(iteration < 0.5*(x+y+z)) {
    forall(neighbour of node) {
      /* call the iterative deepening search */
      new = ids(source, neighbour, iteration);
      /* if free pos found -> return it */
      if(!occupied(new)) return new;
    }
    /* no free pos found -> increase depth */
    iteration++;
  }
  //error
}
```

Listing 7: iterative deepening search

As in our MPI wrapper library, most IO operations are performed with MPI-IO, especially for getting the communication data out of our generated file. Because of the layout of the file, some IO operations, especially reading columns of the matrix are currently quite slow. It is planned to change this by the use of MPI-IO file views.

Example

We have made an example run with smg2000 on 1024 processors and the simple heuristic.

```
Without heuristic:
------------------------------------------------------------
total communication time = 14.724 seconds.
total elapsed time       = 25.537 seconds.

------------------------------------------------------------

With heuristic:
------------------------------------------------------------
total communication time = 14.258 seconds.
total elapsed time       = 25.178 seconds.

------------------------------------------------------------
```

Listing 8: smg2000 heuristic performance

We save about 4 % in the communication time. With sweep3d we saved about 10 %.

Improving the simple heuristic

There are, of course, some ideas to improve this heuristic, which we will present in the this section.

As mentioned above, it would be good to map tasks which exchange lots of small messages close together, since that way we can benefit from low latency at low bandwidth. For this we could change the heuristic to:

1. Map the domain i which sends the most small messages to arbitrary location (x, y, z, t)
2. Map all domains j with $comm_{i,j} > 0$ or $comm_{j,i} > 0$ to a neighbouring location, but map the tasks with most small messages nearest.
3. Repeat previous steps for all unmapped domains

This is good for applications which send primary small messages, since these applications are latency dominated. This heuristic has already been implemented, but currently due to MPI-IO problems it is not very efficient.

Another alternative is to map the domain which sends most messages in total first, so that the receivers are guaranteed to be mapped near.

You could also alter step 3 and map the rank which communicates most with the already mapped domains next, although the effect on communication is not yet clear.

Another early idea was to use a “3D heuristic”, i.e. to map tasks which exchange a lot of big messages in a space diagonal to benefit from higher bandwidth, but this approach is not very useful as long as

adaptive routing starts at the second link, so that no higher bandwidth can be achieved, as we discovered in Fig. 4.

But even if we do not get more bandwidth, a “3D” mapping could be used to make sure that the shortest paths of two senders/receivers do not cross each other, in order to avoid collisions on the nodes between them. This is especially interesting for applications using nonblocking communication, since many messages may be sent during an `MPI_Wait`.

Optimization through evolution strategies

The next step after generating a heuristic task placement would be to optimize this mapping. The method of our choice was an evolution strategy, which are a part of evolutionary or genetic algorithms (refer to [8], p. 353ff). This type of algorithm is mainly based on mutation, in our case the swapping of two tasks on the torus. In minor cases recombination are used, but we have not implemented them yet.

Our implementation is still very simple and more or less a proof of concept. There is much further work needed to get the optimization in a productive state.

Acknowledgment

I would like to thank IBM Germany GmbH for giving me the opportunity to attend this guest student program as an IBM trainee. Special thank goes to Michael Hennecke from IBM for his support and advice during this period. Without him, this work could never have been done. I’d also like to thank Matthias Bolten from the JSC, he was a great organiser and did everything to support us. My thanks to the rest of the JSC staff, especially Bernd Mohr, Brian Wylie, Markus Geimer and Michael Stephan for very interesting and enlightening conversations and their always friendly, helpful and open way. Finaly I have to thank all the other guest students for an always warm and helpful atmosphere.

This work was supported by IBM University Relations
(http://www-05.ibm.com/de/ibm/unternehmen/university_relations/index.html).

References

1. N. Allsopp, J. Follows, M. Hennecke, F. Ishibashi, M. Paolini, D. Quintero, A. Tabary, P. Vezolle, H. Reddy, C. Sosa, S. Prakash, O. Lascu,
   Unfolding the IBM e-server Blue Gene Solution (2005),
   http://www.redbooks.ibm.com/redbooks/pdfs/sg246686.pdf
2. G. Bhanot et al,
   Optimizing task layout on the Blue Gene/L supercomputer,
   IBM Journal of Research and Development 49 (2005) p. 489ff,
   http://www.research.ibm.com/journal/rd/492/bhanot.pdf
3. G. Almási et al,
   Design and implementation of message-passing services for the Blue Gene/L supercomputer,
   IBM Journal of Research and Development 49 (2005) p. 393ff,
   http://www.research.ibm.com/journal/rd/492/almasi.pdf
4. Message Passing Interface Forum,
   MPI: A Message-Passing Interface Standard (2003),
   http://www.mpi-forum.org/docs/mpi-11.ps
5. Message Passing Interface Forum,
   MPI-2: Extensions to the Message-Passing Interface (2003),
   http://www.mpi-forum.org/docs/mpi-20.ps
6. I-Hsin Chung,
   IBM High Performance Toolkit - MPI Tracing/Profiling User Manual (2007).

7. C. Archer,
   Blue Gene/L Coprocessor Use and Progress Engine Update,
   http://www.spscicomp.org/ScicomP12/Presentations/User/ArcherBGCoproc0706.pdf (2006).
8. W. Lippe,
   Soft-Computing,
   Springer (2006).

Parallele schnelle Fouriertransformation nichtäquidistanter Daten

Michael Pippig

Technische Universität Chemnitz, Fakultät für Mathematik
Reichenhainer Straße 39, 09107 Chemnitz

E-mail: michael.pippig@s2004.tu-chemnitz.de

Zusammenfassung:
Ausgehend von einer Bibliothek für die multivariate schnelle Fouriertransformation nichtäquidistanter Daten wird eine angepasste Implementation für den dreidimensionalen Fall vorgestellt, welche im Folgenden mit MPI parallelisiert wird. Die Theorie der NFFT wird kurz umrissen, die serielle Version analysiert und mögliche Parallelisierungsmethoden vorgestellt. Eine Implementierung einer parallelen Version der NFFT wird daraufhin vorgestellt und ebenfalls analysiert. Außerdem wird ein Algorithmus zur besseren Skalierung von NFFTs mit schlechter Stützstellenverteilung entwickelt und ein Ausblick auf weitere Verbesserungmöglichkeiten gegeben.

Notation, die NDFT und die NFFT

Dieser Abschnitt fast die theoretischen Grundlagen der NFFT zusammen. Sei der Torus

$$\mathbb{T}^d := \left\{ \boldsymbol{x} = (x_t)_{t=0,\ldots,d-1} \in \mathbb{R}^d : -\frac{1}{2} \le x_t < \frac{1}{2},\ t = 0,\ldots,d-1 \right\}$$

der Dimension $d \in \mathbb{N}$ gegeben. Dieser dient im folgenden als Wertebereich der nichtäquidistanten Stützstellen $\boldsymbol{x}$. Somit ist die Abtastmenge gegeben durch $\mathcal{X} := \{\boldsymbol{x}_j \in \mathbb{T}^d : j = 0,\ldots,M-1\}$.

Zugelassene Frequenzen $\boldsymbol{k} \in \mathbb{Z}^d$ werden in der Multi-Indexmenge

$$I_{\boldsymbol{N}} := \left\{ \boldsymbol{k} = (k_t)_{t=0,\ldots,d-1} \in \mathbb{Z}^d : -\frac{N_t}{2} \le k_t < \frac{N_t}{2},\ t = 0,\ldots,d-1 \right\},$$

zusammengefasst, wobei $\boldsymbol{N} = (N_t)_{t=0,\ldots,d-1}$ das **gerade** Multi-Frequenzbereichs-Limit ist, d.h. $N_t \in 2\mathbb{N}$.

Das innere Produkt zwischen dem Frequenzindex $\boldsymbol{k}$ und der Stützstelle $\boldsymbol{x}$ ist in der gewohnten Weise als $\boldsymbol{kx} := k_0x_0 + k_1x_1 + \ldots + k_{d-1}x_{d-1}$ definiert. Des weiteren können zwei Vektoren mit einem komponentenweisen Produkt $\boldsymbol{\sigma} \odot \boldsymbol{N} := (\sigma_0 N_0, \sigma_1 N_1, \ldots, \sigma_{d-1} N_{d-1})^\top$ und dessen Inverse $\boldsymbol{N}^{-1} := \left(\frac{1}{N_0}, \frac{1}{N_1}, \ldots, \frac{1}{N_{d-1}}\right)^\top$ verknüpft werden.

Der Raum aller d-variaten, 1-periodischen Funktionen $f : \mathbb{T}^d \to \mathbb{C}$ wird auf den Raum der d-variaten trigonometrischen Polynome

$$T_{\boldsymbol{N}} := \operatorname{span}\left(\mathrm{e}^{-2\pi\mathrm{i}\boldsymbol{k}\cdot} : \boldsymbol{k} \in I_{\boldsymbol{N}}\right)$$

vom Grad N_t $(t = 0,\ldots,d-1)$ in der t-ten Dimension eingeschränkt. Die Dimension $\dim T_N$ des Raumes der d-variaten trigonometrischen Polynome T_N ist durch $\dim T_N = |I_{\boldsymbol{N}}| = \prod_{t=0}^{d-1} N_t$ gegeben.

NDFT - nichtäquidistante diskrete Fouriertransformation

Für eine endliche Anzahl gegebener Fourierkoeffizienten $\hat{f}_{\boldsymbol{k}} \in \mathbb{C}$, $\boldsymbol{k} \in I_{\boldsymbol{N}}$, sind wir an der Berechnung der trigonometrischen Polynome

$$f(\boldsymbol{x}) := \sum_{\boldsymbol{k} \in I_{\boldsymbol{N}}} \hat{f}_{\boldsymbol{k}} \mathrm{e}^{-2\pi \mathrm{i} \boldsymbol{k}\boldsymbol{x}} \tag{1}$$

an den ebenfalls gegebenen Stützstellen $\boldsymbol{x}_j \in \mathbb{T}^d$ interessiert. Somit besteht das Problem in der Berechnung von

$$f_j = f(\boldsymbol{x}_j) := \sum_{\boldsymbol{k} \in I_{\boldsymbol{N}}} \hat{f}_{\boldsymbol{k}} \mathrm{e}^{-2\pi \mathrm{i} \boldsymbol{k}\boldsymbol{x}_j}, \tag{2}$$

$j = 0, \ldots, M-1$. In Matrix-Vektor-Schreibweise stellt man dies dar als

$$\boldsymbol{f} = \boldsymbol{A}\hat{\boldsymbol{f}} \tag{3}$$

mit

$$\boldsymbol{f} := (f_j)_{j=0,\ldots,M-1}, \quad \boldsymbol{A} := \left(\mathrm{e}^{-2\pi \mathrm{i} \boldsymbol{k}\boldsymbol{x}_j}\right)_{j=0,\ldots,M-1;\, \boldsymbol{k} \in I_{\boldsymbol{N}}}, \quad \hat{\boldsymbol{f}} := \left(\hat{f}_{\boldsymbol{k}}\right)_{\boldsymbol{k} \in I_{\boldsymbol{N}}}.$$

Der direkte Algorithmus dieses Matrix-Vektor-Produkt zu berechnen, auch NDFT genannt, benötigt $\mathcal{O}(M|I_{\boldsymbol{N}}|)$ arithmetische Operationen.

Äquidistante Stützstellen

Mit $d \in \mathbb{N}$, $N_t = N \in 2\mathbb{N}$, $t = 0, \ldots, d-1$ und $M = N^d$ äquidistanten Stützstellen $\boldsymbol{x}_{\boldsymbol{j}} = \frac{1}{N}\boldsymbol{j}$, $\boldsymbol{j} \in I_{\boldsymbol{N}}$ ist die Berechnung von (3) auch als multivariate diskrete Fouriertransformation (DFT) bekannt. Für diesen Spezialfall bezeichnet man $\hat{f}_{\boldsymbol{k}}$ als diskrete Fourierkoeffizienten und die Abtaststellen $f_{\boldsymbol{j}}$ können mit der wohlbekannten schnellen Fouriertransformation (FFT) mit lediglich $\mathcal{O}(|I_{\boldsymbol{N}}| \log |I_{\boldsymbol{N}}|)$ arithmetischen Operationen berechnet werden. Außerdem erhält man die Inversionsformel

$$\boldsymbol{A}\boldsymbol{A}^{\mathsf{H}} = \boldsymbol{A}^{\mathsf{H}}\boldsymbol{A} = |I_{\boldsymbol{N}}|\boldsymbol{I},$$

welche im nichtäquidistanten Fall im allgemeinen **nicht** gilt.

NFFT - nichtäquidistante schnelle Fouriertransformation

Um die Erklärungen einfach zu halten, beschränken sich die folgenden Ausführungen auf den Fall $d = 1$. Die Verallgemeinerung der FFT ist ein approximativer Algorithmus und hat die Komplexität $\mathcal{O}(N \log N + \log(1/\varepsilon) M)$, wobei ε die gewünschte Genauigkeit darstellt. Der Grundidee ist die Verwendung von herkömmlichen FFTs und Fensterfunktionen φ, welche im Orts-/Zeit-Bereich $\mathbb{R}$ und im Frequenzbereich $\mathbb{R}$ gut lokalisiert sind. Im weiteren wird sich auf die Gaußfunktion als eine mögliche Fensterfunktion beschränkt.

Das Problem ist die Berechnung von

$$f(x) = \sum_{k \in I_N} \hat{f}_k \mathrm{e}^{-2\pi \mathrm{i} k x} \tag{4}$$

an beliebigen Stützstellen $x_j \in \mathbb{T}$, $j = 0, \ldots, M-1$.

Der Ansatz

Wir wollen das trigonometrische Polynom f von (4) durch eine Linearkombination von verschobenen 1-periodischen Fensterfunktionen $\tilde{\varphi}$ wie folgt darstellen:

$$s_1(x) := \sum_{l \in I_n} g_l \, \tilde{\varphi}\left(x - \frac{l}{n}\right) . \tag{5}$$

Mit Hilfe der Überabtastrate $\sigma > 1$ wird die Länge der FFT durch $n := \sigma N$ gegeben.

Die erste Approximation - Abschneiden im Frequenzbereich

Wechselt man in Definition (5) in den Frequenzbereich, erhält man

$$s_1(x) = \sum_{k \in I_n} \hat{g}_k \, c_k(\tilde{\varphi}) \, \mathrm{e}^{-2\pi \mathrm{i} kx} + \sum_{r \in \mathbb{Z} \setminus \{0\}} \sum_{k \in I_n} \hat{g}_k \, c_{k+nr}(\tilde{\varphi}) \, \mathrm{e}^{-2\pi \mathrm{i}(k+nr)x}$$

mit den diskreten Fourierkoeffizienten

$$\hat{g}_k := \sum_{l \in I_n} g_l \, \mathrm{e}^{2\pi \mathrm{i} \frac{kl}{n}} . \tag{6}$$

Vergleich von (4) mit (5) und die Annahme, dass $c_k(\tilde{\varphi})$ für $|k| \geq n - \frac{N}{2}$ hinreichend klein wird, suggeriert

$$\hat{g}_k := \begin{cases} \frac{\hat{f}_k}{c_k(\tilde{\varphi})} & \text{for } k \in I_N, \\ 0 & \text{for } k \in I_n \setminus I_N . \end{cases} \tag{7}$$

So erhält man die Werte g_l aus (6) durch

$$g_l = \frac{1}{n} \sum_{k \in I_N} \hat{g}_k \, \mathrm{e}^{-2\pi \mathrm{i} \frac{kl}{n}} \qquad (l \in I_n),$$

einer FFT der Länge n.

Diese Approximation liefert einen Aliasing-Fehler.

Die zweite Approximation - Abschneiden im Zeit-/Ortsbereich

Wenn die Funktion φ im Zeit-/Ortsbereich $\mathbb{R}$ gut lokalisiert ist, kann sie durch eine Funktion

$$\psi(x) = \varphi(x) \, \chi_{[-\frac{m}{n}, \frac{m}{n}]}(x)$$

mit $\operatorname{supp} \psi = \left[-\frac{m}{n}, \frac{m}{n}\right]$, $m \ll n$, $m \in \mathbb{N}$ approximiert werden. Man definiert ihre periodische Version $\tilde{\psi}$ mit kompaktem Träger in $\mathbb{T}$ als

$$\tilde{\psi}(x) = \sum_{r \in \mathbb{Z}} \psi(x + r) .$$

Mit Hilfe der Indexmenge

$$I_{n,m}(x_j) := \{ l \in I_n : n x_j - m \leq l \leq n x_j + m \}$$

wird eine Approximation für s_1 wie folgt definiert:

$$s(x_j) := \sum_{l \in I_{n,m}(x_j)} g_l \, \tilde{\psi}\left(x_j - \frac{l}{n}\right) . \tag{8}$$

Diese Approximation führt zu einem Abschneidefehler.

Der Fall $d > 1$

Beginnend mit dem Problem der Berechnung der multivariaten trigonometrischen Polynome aus (1) müssen einige Verallgemeinerungen gemacht werden. Die Fensterfunktion wird durch

$$\varphi(\boldsymbol{x}) := \varphi_0(x_0)\,\varphi_1(x_1)\ldots\varphi_{d-1}(x_{d-1})$$

gegeben, wobei φ_t eine univariate Fensterfunktion ist. Somit folgt für die Fourierkoeffizienten

$$c_{\boldsymbol{k}}(\tilde{\varphi}) = c_{k_0}(\tilde{\varphi}_0)\,c_{k_1}(\tilde{\varphi}_1)\ldots c_{k_{d-1}}(\tilde{\varphi}_{d-1})\,.$$

Der Ansatz wird verallgemeinert zu

$$s_1(\boldsymbol{x}) := \sum_{\boldsymbol{l}\in I_{\boldsymbol{n}}} g_{\boldsymbol{l}}\,\tilde{\varphi}\left(\boldsymbol{x} - \boldsymbol{n}^{-1}\odot\boldsymbol{l}\right),$$

wobei die Länge der FFT durch $\boldsymbol{n} := \boldsymbol{\sigma}\odot\boldsymbol{N}$ und die Überabtastraten durch $\boldsymbol{\sigma} = (\sigma_0,\ldots,\sigma_{d-1})^\top$ gegeben sind. Anstelle von (7) definiert man

$$\hat{g}_{\boldsymbol{k}} := \begin{cases} \frac{\hat{f}_{\boldsymbol{k}}}{c_{\boldsymbol{k}}(\tilde{\varphi})} & \text{for } \boldsymbol{k}\in I_{\boldsymbol{N}},\\ 0 & \text{for } \boldsymbol{k}\in I_{\boldsymbol{n}}\backslash I_{\boldsymbol{N}}.\end{cases}$$

Die Werte $g_{\boldsymbol{l}}$ können mit einer (multivariaten) FFT der Länge $n_0\times n_1\times\ldots\times n_{d-1}$ wie folgt erhalten werden:

$$g_{\boldsymbol{l}} = \frac{1}{|I_{\boldsymbol{n}}|}\sum_{\boldsymbol{k}\in I_{\boldsymbol{N}}}\hat{g}_{\boldsymbol{k}}\,\mathrm{e}^{-2\pi\mathrm{i}\boldsymbol{k}\left(\boldsymbol{n}^{-1}\odot\boldsymbol{l}\right)},\quad \boldsymbol{l}\in I_{\boldsymbol{n}}.$$

Unter Verwendung der Funktion $\psi(\boldsymbol{x}) = \varphi(\boldsymbol{x})\,\chi_{[-\frac{m}{n},\frac{m}{n}]^d}(\boldsymbol{x})$ mit kompaktem Träger erhält man

$$s(\boldsymbol{x}_j) := \sum_{\boldsymbol{l}\in I_{\boldsymbol{n},m}(\boldsymbol{x}_j)} g_{\boldsymbol{l}}\,\tilde{\psi}\left(\boldsymbol{x}_j - \boldsymbol{n}^{-1}\odot\boldsymbol{l}\right),$$

wobei $\tilde{\psi}$ wiederum die 1-periodische Version von ψ ist und die Multiindexmenge gegeben wird durch

$$I_{\boldsymbol{n},m}(\boldsymbol{x}_j) := \left\{\boldsymbol{l}\in I_{\boldsymbol{n}} : \boldsymbol{n}\odot\boldsymbol{x}_j - m\boldsymbol{1} \le \boldsymbol{l} \le \boldsymbol{n}\odot\boldsymbol{x}_j + m\boldsymbol{1}\right\}.$$

Der Algorithmus

Zusammenfassend erhält man Algorithmus 1 für die schnelle Berechnung von (3) mit $\mathcal{O}\left(|I_{\boldsymbol{n}}|\log|I_{\boldsymbol{n}}| + mM\right)$ arithmetischen Operationen.

Die serielle NFFT

Motivation

Die NFFT Bibliothek beinhaltet bereits einen Algorithmus zur Berechnung der d-variaten nichtäquidistanten Fouriertransformation für beliebige $d\in\mathbb{N}$. Dennoch war das erste Anliegen dieser Arbeit eine Version des Programms zu schreiben, welche speziell auf den Fall $d = 3$ zugeschnitten ist. Man erwartete eine nennenswerte Laufzeitverringerung, da der Overhead des Programmes deutlich verringert werden kann. Des weiteren war ein grundlegendes Verständnis des Algoithmus notwendig um später Möglichkeiten der Parallelisierung zu erarbeiten. Auch der Umgang mit den verwendeten Programmpaketen konnte so gelernt werden.

Eingabe: $d, M \in \mathbb{N}$, $\boldsymbol{N} \in 2\mathbb{N}^d$
Input: $\boldsymbol{x}_j \in [-\frac{1}{2}, \frac{1}{2}]^d,\ j = 0, \dots, M-1$, and $\hat{f}_{\boldsymbol{k}} \in \mathbb{C},\ \boldsymbol{k} \in I_{\boldsymbol{N}}$,

1: Für $\boldsymbol{k} \in I_{\boldsymbol{N}}$ berechne

$$\hat{g}_{\boldsymbol{k}} := \frac{\hat{f}_{\boldsymbol{k}}}{|I_{\boldsymbol{n}}|\, c_{\boldsymbol{k}}(\tilde{\varphi})}.$$

2: Für $\boldsymbol{l} \in I_{\boldsymbol{n}}$ berechne mit d-variater FFT

$$g_{\boldsymbol{l}} := \sum_{\boldsymbol{k} \in I_{\boldsymbol{N}}} \hat{g}_{\boldsymbol{k}}\, \mathrm{e}^{-2\pi \mathrm{i} \boldsymbol{k} \left(\boldsymbol{n}^{-1} \odot \boldsymbol{l}\right)}.$$

3: Für $j = 0, \dots, M-1$ berechne

$$f_j := \sum_{\boldsymbol{l} \in I_{\boldsymbol{n},m}(\boldsymbol{x}_j)} g_{\boldsymbol{l}}\, \tilde{\psi}\left(\boldsymbol{x}_j - \boldsymbol{n}^{-1} \odot \boldsymbol{l}\right).$$

Ausgabe: approximierte Werte $f_j,\ j = 0, \dots, M-1$.
Komplexität: $\mathcal{O}(|\boldsymbol{N}| \log |\boldsymbol{N}| + M)$.

Algorithm 1: `NFFT`

Die FFTW

Die FFTW ist ein open source Programmpaket zur Berechnung der schnellen diskreten Fouriertransformation, welche in Schritt 2 von Algorithmus 1 benötigt wird. Es wurde von Matteo Frigo und Steven G. Johnson am Massachusetts Institute of Technology entwickelt. Verwendet wurde die aktuelle Version 3.2alpha2, da diese die erste Version aus der dritten Entwicklungsreihe der FFTW mit einer Unterstützung für parallele FFTs mit MPI ist.

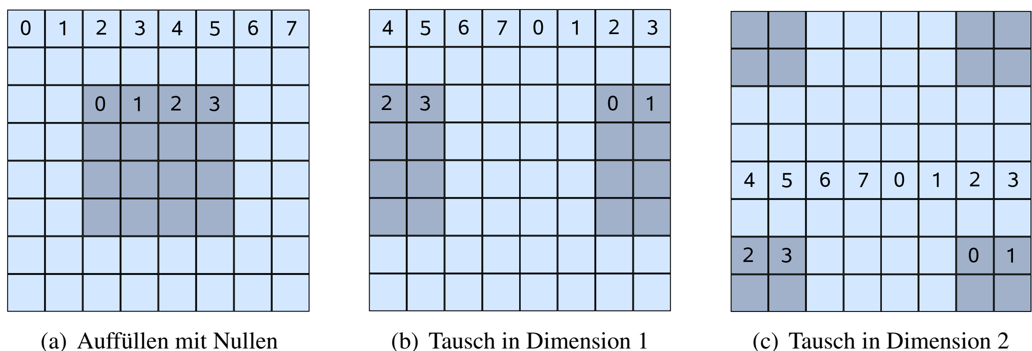
Programmablauf

Die Funktionsweise des Programmes, welches im folgenden mit nfft_ser3d bezeichnet wird, wird nun mit einem Ablaufdiagramm dargestellt, wobei besonders die wesentlichen drei Schritte von Algorithmus 1 erläutert werden. Hierzu betrachten wir wegen der besseren Übersicht den zweidimensionalen Fall. Der Algorithmus des Programms lässt sich hieraus leicht analog mit einer zusätzlichen Dimension ableiten.

Schritt 1

Die gegebenen Fourierkoeffizienten $\hat{f}_k$ werden eingelesen und gleichzeitig mit den inversen Fourierkoeffizienten der Fensterfunktion multipliziert, d.h. da die Daten in einem Feld an die FFTW übergeben werden, kann man das Speichern der Koeffizienten mit der Multiplikation verbinden. Da man ein Feld der Größe $n_1 \cdot n_2$ mit $n_t \geq N_t$ für $t \in \{1, 2\}$ anlegen muss, werden die Einträge an den Rändern mit Nullen aufgefüllt. Dieses Auffüllen wird durch eine einmalige Initialisierung des gesamten Feldes mit Nullen ersetzt. Zu beachten ist, dass der Summationsindex in Schritt 2 über den 2-dimensionalen Würfel $\left[\frac{-1}{2}, \frac{1}{2}\right)^2$ läuft, während die FFTW ihre Eingabedaten als Koeffizienten in $[0, 1)^2$ erwartet. Aufgrund der 1-Periodizität der Funktion f lässt sich dieses Eingabeformat mit einem Tausch der vorderen Hälfte des Feldes an das Ende ermöglichen. Dieser Tausch muss bezüglich aller Dimensionen vorgenommen werden und wird bereits beim Abspeichern des Feldes berücksichtigt. Schritt 1 ist in Abb. 1 für $N_1 = N_2 = 4$, $n_1 = n_2 = 8$, $m = 1$ bildlich dargestellt. Die hellen Kästchen symbolisieren die

Nullleinträge und die dunkleren die $\frac{\hat{f}_k}{c_{\boldsymbol{k}}(\tilde{\varphi})}$



(a) Auffüllen mit Nullen (b) Tausch in Dimension 1 (c) Tausch in Dimension 2

Abbildung 1: Speichern und Tauschen der Fourierkoeffizienten für die FFTW.

Schritt 2

Die bereits richtig angeordneten Daten werden mit der FFTW transformiert.

Schritt 3

Für jede Stützstelle x_j wird der Index $\lceil nx_j \rceil - m$ und gemäß (8) die Approximation durch Summation über $(2m+1)^2$ Terme bestimmt. Wiederum ist darauf zu achten, dass die Indizes vertauscht sind, da sich die Ergebnisse der FFT auf eine Indexmenge in $[0,1)^2$ beziehen. Abb. 2 verdeutlicht dies wieder für $N_1 = N_2 = 4$, $n_1 = n_2 = 8$, $m = 1$. Das schwarze Kästchen stellt den Index $\lceil nx_j \rceil$ dar und die umliegenden Indizes werden in die Summation einbezogen.

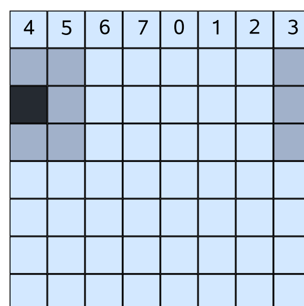


Abbildung 2: Berechnung des Abtastwertes an einer Stützstelle.

Laufzeitvergleiche

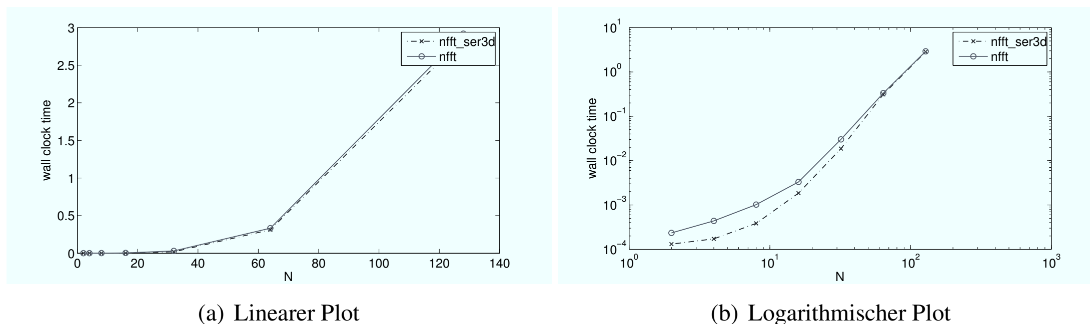
Die Messungen der Laufzeiten wurden auf einem Rechner mit Intel Core 2 Duo T7200 mit 2 GHz unter Linux durchgeführt, das Programm wurde dazu mit dem gcc mit O3 Optimierung übersetzt. Die drei Dimensionen N_t wurden jeweils gleichgroß gewählt, weshalb im Folgenden nur noch N mit $N = N_t, t \in \{1,2,3\}$, verwendet wird. Verglichen wurden zwei verschiedene Verhältnisse zwischen der Anzahl der Stützstellen und der Dimensionen N_t. Des weiteren wurde darauf geachtet, dass die Optionen der Bibliotheksfunktion identisch zu den in der nfft_ser3d implementierten gewählt wurden. Dies betrifft vor allem die Verwendung der Gauß-Fensterfunktion, Berechnung der FFT inplace und keine Vorberechnung der Werte der Fensterfunktion an den Stützstellen $\boldsymbol{x}$.

Der Fall $M = N$

Im ersten Fall, vgl. Abb. 4, wächst die Anzahl der Stützstellen M mit der gleichen Ordnung wie N, d.h. die Stützstellenanzahl steigt linear und die Größe der 3-dimensionalen FFT kubisch.
Besonders für kleine Problemgrößen ist die nfft_ser3d der Bibliotheksfunktion überlegen. Das asymptotische Verhalten deutet eine gleiche Laufzeit für große N an. Dies lässt sich durch den wachsenden Anteil der FFTW am Gesamtprogramm begründen. Sowohl die Initialisierung der Daten in Schritt 1 von Algorithmus 1 als auch die Berechnung der Abtastwerte an den Stützstellen in Schritt 3 nehmen für diese Fälle deutlich weniger Zeit in Anspruch als die FFTW in Schritt 2, welche in beiden Programmen identisch ausgeführt wird.

N	2	4	8	16	32	64	128
nfft_ser3d	1.31e2	1.72e2	3.84e2	1.85e3	1.87e4	3.12e5	2.86e6
nfft	2.34e2	4.38e2	1.02e3	3.31e3	3.02e4	3.33e5	2.92e6

Abbildung 3: Laufzeitvergleich der nfft_ser3d mit der Bibliotheksfunktion nfft. M wächst mit N



(a) Linearer Plot (b) Logarithmischer Plot

Abbildung 4: Laufzeitvergleich der nfft_ser3d mit der Bibliotheksfunktion nfft. M wächst mit N

Der Fall $M = N^3$

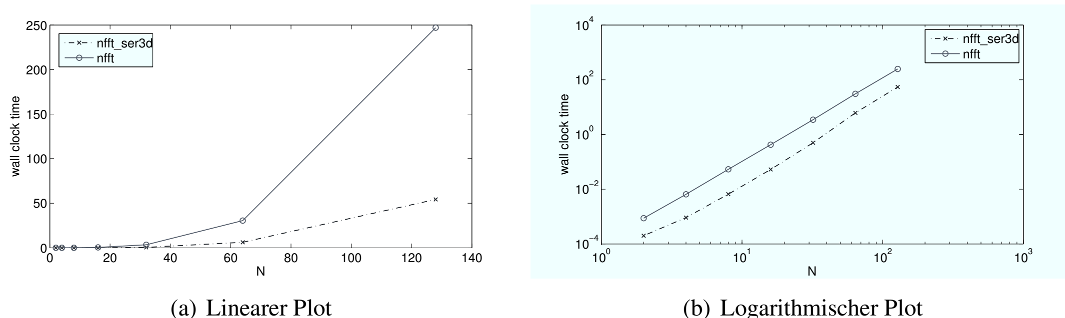
Als zweiter Test, vgl. Abb. 6, wurde M gleich mit N^3 gewählt. Dies erscheint sinnvoll, da auch in der FFT oftmals die Anzahl der Eingangs- und Ausgangsdaten gleichgroß ist. Wiederum ist die nfft_ser3d schneller, was sich besonders für große N_t bemerkbar macht.

N	2	4	8	16	32	64	128
nfft_ser3d	2e2	9.15e2	6.57e3	5.25e4	4.96e5	6.15e6	5.43e7
nfft	8.71e2	6.5e3	5.32e4	4.22e5	3.45e6	3.05e7	2.47e8

Abbildung 5: Laufzeitvergleich der nfft_ser3d mit der Bibliotheksfunktion nfft. M wächst mit N^3

Zusammenfassung

Die Laufzeitmessungen bestätigten die Vermutung, dass sich der Overhead der Bibliotheksfunktion negativ auf die Geschwindigkeit auswirkt. Besonders für große dreidimensionale Probleme ist die Funktion nfft_ser3d vorzuziehen. Analoge Ergebnisse spiegelten sich in der Programmentwicklung auch schon für ein- und zweidimensionale NFFTs wider. Eine Implementierung dieser Fälle kann schnell nachgeholt werden. Somit erscheint es sinnvoll die allgemeinere Funktion der NFFT-Bibliothek für diese Spezialfälle zu ersetzen.



(a) Linearer Plot

(b) Logarithmischer Plot

Abbildung 6: Laufzeitvergleich der nfft_ser3d mit der Bibliotheksfunktion nfft. M wächst mit N^3

Die parallele NFFT

Motivation

Wie auch die FFT benötigt die NFFT mit wachsenden N_t und M zu viel Speicher um sie noch mit einem Rechner zu verwenden. Eine NFFT mit den Parametern $N_t = 256, n_t = 512$ für $t \in \{1, 2, 3\}$, benötigt zum Abspeichern der Fourierkoeffizienten aus Schritt 2 bereits 2GB. Auch die Rechenzeit sollte sich mit dem Einsatz mehrerer Prozessoren verringern. Anliegen dieser Arbeit war es zu untersuchen wie man die NFFT parallelisieren kann und eine erste Implementation zu erstellen.

Möglichkeiten der Parallelisierung

Die Unterteilung des Algorithmus 1 in 3 Schritte kann auch in der parallelen Version verwendet werden um die Möglichkeiten des Vorgehens einzuschätzen. Schritt 1 beinhaltet nur die Initialisierung des Feldes für die FFT. Dennoch müssen besonders die Einschränkungen beachtet werden, welche die Dateneingabe in Schritt 2 betrachten.
Da die FFTW bereits eine parallele Version der FFT beinhaltet, musste in Schritt 2 lediglich das Eingabeformat der FFTW berücksichtigt werden. Dies bedeutet z.B. dass die FFTW in der aktuellen Version das Feld nur bezüglich der ersten Dimension zerlegt. Dies manifestiert gewisse Grenzen an die Parallelisierbarkeit von Problemen, bei denen die erste Dimension kleiner ist als die Anzahl der verwendeten Prozessoren. Es existieren auch andere Bibliotheken, welche das Feld bezüglich mehrerer Dimensionen zerlegen, welche allerdings nicht getestet wurden.
Im letzten Schritt muss beachtet werden, dass jeder Prozessor nur einen Teil des Feldes besitzt. D.h. es lässt sich nicht umgehen, die Stützstellen entsprechend der lokalen Feldgrenzen zuzuordnen. Die erste Möglichkeit, *alle* Stützstellen an *alle* Prozessoren zu übergeben, kostete jeden Prozess viel Speicher, welcher auch für wachsende Prozessorenanzahl nicht abnimmt, und setzt voraus, dass jeder Prozessor *alle* Stützstellen durchsuchen muss. Das Ordnen der Stützstelle wurde also dem Benutzer überlassen, sodass er dafür Sorge tragen muss jedem Prozess die richtigen Stützstellen zu übergeben. Dabei wird er durch mehrere Initializierungsroutinen unterstützt, wie später näher erläutert wird.
Zur Berechnung der Summen in Schritt 3 kann es erforderlich sein, Daten eines anderen Prozesses anzufordern. Um die Kommunikation gering zu halten wurde sich dafür entschieden die erforderlichen Randgebiete einmal im Ring an den direkten Vorgänger zu senden. Die Abtastwerte an den Stützstellen werden deshalb nicht auf jenem Prozessor berechnet, welcher den Feldindex $\lceil nx_j \rceil$ der ersten Dimension besitzt, sondern auf jenem, welcher den Feldindex $\lceil nx_j \rceil - m$ besitzt. Man kann so die Anzahl der Kommunikationsaufrufe um eins reduzieren, da nur die Daten des Nachfolgers benötigt werden. Man vermeidet also das Aufteilen des Datenpaketes in zwei kleinere. Das Senden im Ring muss eventuell wiederholt werden, wenn mehr Streifen benötigt werden, als ein Prozessor besitzt.

Programmablauf

Der Ablauf wird schematisch für das bereits verwendete Beispiel in zwei Dimensionen dargestellt. Dabei sind $N := N_1 = N_2 = 4, n := n_1 = n_2 = 8, m = 1, M = 1$ wie im seriellen Ablaufplan.

Schritt 1

Die Fourierkoeffizienten $\hat{f}_k$ werden in der richtigen Reihenfolge an jene Prozessoren übergeben, welche sie später erhalten sollen. Vergleiche hierzu Abb. 1(b) und Abb. 7. Dabei ist besonders der Tausch bezüglich der ersten Dimension bereits durch das geeignete Einlesen der Daten vermieden worden. Nun

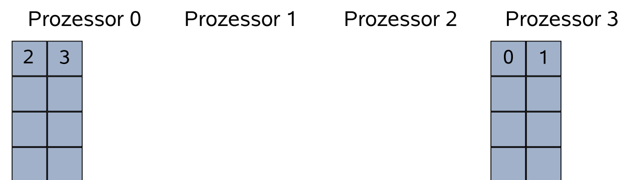


Abbildung 7: Übergabe der Fourierkoeffizienten.

wird wie in der seriellen Version initialisiert, d.h. man multipliziert die Fourierkoeffizienten $\hat{f}_k$ mit den Inversen der Fourierkoeffizienten $c_{\boldsymbol{k}}(\tilde{\varphi})$ der Fensterfunktion und die Felder werden mit Nullen aufgefüllt, wie in Abb. 8 zu sehen ist. Der Tausch bezüglich aller Dimensionen mit Außnahme der ersten wird wiederum beim Speichern berücksichtigt.

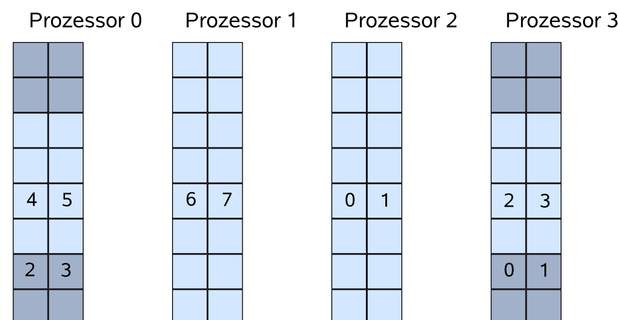


Abbildung 8: Initialisierung der Daten.

Schritt 2

Nun liegen die Daten bereits in der richtigen Anordnung vor, um problemlos die parallele FFTW aufzurufen.

Schritt 3

Der Abschneideparameter war als $m = 1$ gesetzt. Somit benötigt jeder Prozessor 3 Streifen um einen Abtastwert berechnen zu können. Im schlechtesten Fall liegt eine Stützstelle so, dass 2 der 3 Streifen nicht auf dem Prozessor liegen, der den zugehörigen Abtastwert berechnen soll. Deshalb werden zwei Streifen von jedem Prozessor im Ring an den Vorgänger geschickt. Nun kann jeder Prozessor die Approximation der Abtastwerte an den ihm zugeordneten Stützstellen auswerten. Vgl. Abb. 9.

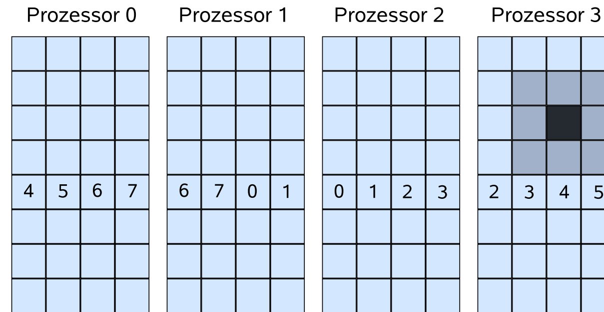


Abbildung 9: Senden im Ring und Berechnung der Abtastwerte.

Benutzung

Für erfahrene Benutzer bietet die Variable `size_fftw` eine Möglichkeit, die Anzahl der an der FFTW beteiligten Prozessoren von oben zu beschränken. Es folgen später die Erklärungen, wozu diese Möglichkeit implementiert wurde. Innerhalb dieses Abschnittes setzten wir `size_fftw` gleich der Anzahl aller an der NFFT beteiligten Prozessoren und übergeben die Variable wie gefordert an die Initialisierungsroutinen.
Die erste Routine dient der Bekanntgabe der Grenzen, in denen sich die Stützstellen für jeden Prozessor befinden. Man übergibt den Abschneideparameter m, das Feld der Dimensionen der FFT n, den *Zeiger* auf `size_fftw` sowie zwei Zeiger auf double Variablen, welche nicht initialisiert sein müssen. Letzere beide enthalten dann die Intervallgrenzen, zwischen denen sich die Stützstellen dieses Prozesses befinden sollen, d.h. $\boldsymbol{x}_{\boldsymbol{j}}^{(1)} \in [lower_border, upper_border)$. Zwei Sonderfälle sind zu betrachten, wenn nämlich $lower_border = upper_border$ gilt, erhält der Prozess keine Stützstellen. Gilt hingegen $lower_border \geq upper_border$, so müssen die Stützstellen mit $\boldsymbol{x}_{\boldsymbol{j}}^{(1)} \in \left[-\frac{1}{2}, upper_border\right)$ oder $\boldsymbol{x}_{\boldsymbol{j}}^{(1)} \in \left[lower_border, \frac{1}{2}\right)$ gewählt werden.

```
int nfft_mpi_init_borders(int m, int* n, int* ptr_size_fftw,
    double* lower_border, double* upper_border);
```

Sinn dieser Routine ist noch nicht das Anlegen der Stützstellen, sondern der Benutzer ist nun verpflichtet die Menge der Stützstellen zu durchsuchen und anzugeben, wie groß die Anzahl der Stützstellen im jeweiligen Intervall ist. Diese Anzahl speichert er in einer Variablen `local_x_num`.
Damit der Benutzer die Fourierkoeffizienten richtig bereit stellen kann, muss er wissen, wie die Streifen auf die Prozessoren verteilt werden. Hierzu ruft er analog zur FFTW eine Initialisierungsroutine auf:

```
int nfft_mpi_local_size_3d(int N0, int N1, int N2, int n0, int n1,
    int n2, int size_fftw, int *local_N0, int *local_N0_start);
```

Er bekommt mit `local_N0_start` analog zur parallelen Version der FFTW den Startindex der ersten Dimension, ab welchem `local_N0` Streifen erwartet werden. Dabei kann `local_N0` durchaus Null betragen, da manche Prozessoren keine Daten bekommen und ihr komplettes lokales Feld mit Nullen füllen.
Eine weitere Funktion dient der entgültigen Verteilung der Stützstellen und legt einen Kommunikator an, welcher der `nfft_par3d` später übergeben wird.

```
int get_local_border(int size_fftw, int* ptr_local_x_num,
    int* ptr_local_x_start, double* ptr_lower_border,
    double* ptr_upper_border, MPI_Comm* ptr_comm_local);
```

Die Anzahl der Stützstellen kann über mehrere Prozessoren verteilt werden, deshalb wird `local_x_num` als Zeiger übergeben und die neue Variable `local_x_start` wird analog zu `local_N0_start` verwendet. Sie gibt an, dass ab der `local_x_start` -ten Stützstelle in diesem Intervall genau `local_x_num` Stützstellen für den jeweiligen Prozess angelegt werden sollen. Nun legt man ein Feld für die Stützstellen an.

```
double* local_x;
local_x = (double*) malloc( sizeof(double)*local_x_num*3 );
for(i = 0; i < local_x_num; i++)
  for(j = 0; j < 3; j++)
    local_x[3*i+j] = users_data_x;
```

Analog verfährt man mit den Fourierkoeffizienten, allerdings speichert man die Koeffizienten im row major Format in einen eindimensionalen Eingabevektor. Unter der Annahme, der Benutzer habe ein dreidimensionales Feld `users_data_koeff` von Fourierkoeffizienten, ist die Initialisierung folgendermaßen zu implementieren:

```
double complex* in;
in = (double complex*)
      malloc(sizeof(double complex)*local_N0*N[1]*N[2]);
for(i = local_N0_start; i < local_N0_start+local_N0; i++)
  for(j = 0; j < N[1]; j++)
        for(k = 0; k < N[1]; k++)
          local_x[ k + N[2]*(j + N[1]*(i-local_N0_start))] ...
          = users_data_koeff[i,j,k];
```

Es folgt die Standardinitialisierung der NFFT, welche um einige Parameter erweitert wurde.

```
void nfft_mpi_init_guru (nfft_plan *myplan, int d, int *N, int M,
      int *n, int m, int    local_N0, double* local_x,
      double complex* in, unsigned nfft_flags, unsigned fftw_flags);
```

Nun kann man die Transformation mit folgendem Aufruf durchführen:

```
void nfft_par3d(nfft_plan* myplan, int local_N0_start, int size_fftw,
      int local_x_num, int local_x_start, MPI_Comm comm_local);
```

Laufzeitmessungen

Die Messungen der Laufzeiten wurden auf dem Supercomputer JUMP am JSC durchgeführt. Die drei Dimensionen N_t wurden jeweils gleich groß gewählt, weshalb im folgenden nur noch N mit $N = N_t, t \in \{1, 2, 3\}$, verwendet wird. Die Stützstellen wurden zufällig, aber gleichmäßig verteilt, sodass jeder Prozess nahezu die gleiche Anzahl an Abtastwerten zu berechnen hatte. Die Problemgröße N beträgt 128.
Die Abbildungen 11 und 12 zeigen die Auswertungen der Laufzeiten für 6 verschiedene Prozessorenanzahlen. Die Laufzeit wird bei Verdoppelung der Prozessoren bis zu 16 mehr als halbiert. Dieses Verhalten ohne weitere Analysen zu begründen ist schwer, es ist aber naheliegend, dass die kleineren Felder, welche von jedem Prozessor verwaltet werden eine deutlich besseres Cache-Verwaltung ermöglichen. Außerdem sind die Stützstellen in der seriellen Variante nicht sortiert, sodass die Feldzugriffe randomisiert und damit Cache-unfreundlich ablaufen. Das parallele Programm kann dies kompensieren, da jeder Prozessor nur innerhalb eines deutlich kleineren Feldes agieren muss. Der Speedup des Programmes ist deshalb auch besser als linear und die Effizienz liegt teilweise deutlich über eins. Dennoch müssten noch weitere Laufzeitmessungen durchgeführt werden um das Programm exakt bewerten zu können.
Abbildung 14 zeigt den Anstieg der Laufzeit für eine wachsende Problemgröße und 64 Prozessoren.

Anz. Prozesse	1	2	4	8	16	32
nfft_par3d	1.84e2	7.27e1	3.48e1	1.67e1	7.89e0	4.93e0

Abbildung 10: Laufzeit der nfft_par3d in Abhängigkeit von der Prozessorenanzahl für $N = 128$

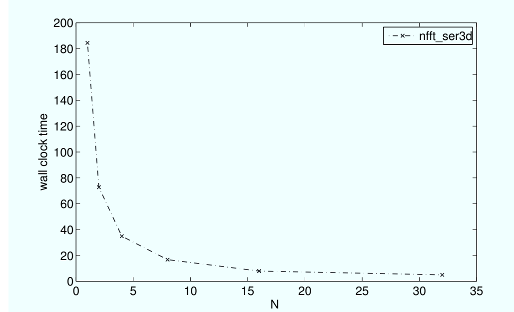


Abbildung 11: Laufzeit der nfft_par3d in Abhängigkeit von der Prozessorenanzahl

Erweiterte Funktionen

Es wurde bereits erwähnt, dass die Anzahl der an der FFTW beteiligten Prozessoren herabgesetzt werden kann. Des weiteren erscheint es noch nicht sinnvoll die Anzahl der Stützstellen auf einem Prozessor, sowie die Grenzen des Intervalls in welchem diese liegen nochmals einer Initialisierung zu übergeben. In der Tat könnte man das Verfahren zur Verwendung der parallelen NFFT deutlich vereinfachen und abkürzen. Um dennoch die Verwendung der zusätzlichen Parameter zu motivieren, folgen zwei Probleme, die eine einfachere Variante nicht zu beheben fähig wäre.

Grenzen der Parallelisierung

Wird die Anzahl der Prozessoren über die Größe der ersten Dimension erhöht, kann die FFTW nicht mehr alle Prozessoren nutzen. Man müsste sich auf N_1 Prozessoren beschränken, was allerdings gerade bei vielen Stützstellen zu langen Laufzeiten führt.

Probleme in der Stützstellenaufteilung

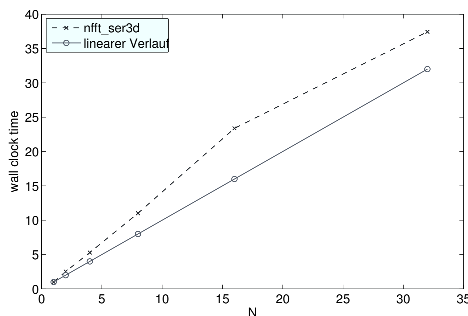
Da die Stützstellen $\boldsymbol{x_j}$ beliebig im Intervall $\left[-\frac{1}{2}, \frac{1}{2}\right)$ liegen dürfen, kann es im extremen Fall vorkommen, dass alle Abtastwerte nur von einem Prozessor zu berechnen sind. Aber selbst bei weniger extremen Beispielen wirkt sich eine unausgewogene Stützstellenverteilung sehr negativ auf die Laufzeit des Programms aus.

Analyse der Laufzeitverteilung

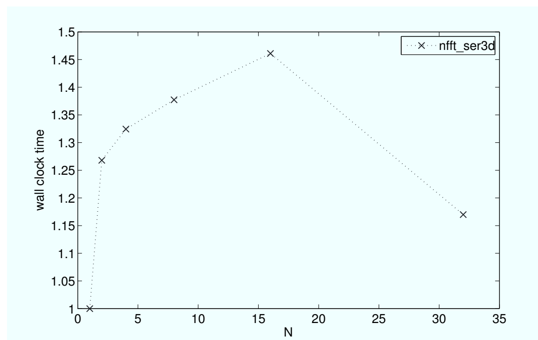
Durch Messungen der Laufzeiten einzelner Unterroutinen der `nfft_par3d` stellte sich heraus, dass der Anteil der FFTW an der Gesamtlaufzeit nicht so groß war wie ursprünglich erwartet. Hingegen beanspruchte Schritt 3 von Algorithmus 1 den größten Teil der Rechenzeit für sich. Dies ermöglicht aber eine andere Herangehensweise an die Parallelisierung.

Verwendung aller Prozessoren

Selbst wenn die Anzahl der Prozessoren die erste Dimension übersteigt, kann man noch eine Laufzeitverringerung erreichen, indem die übrigen Prozessoren in Schritt 3 mit einbezogen werden. Dazu muss



(a) Speedup



(b) Effizienz

Abbildung 12: Effizienz und Speedup der Funktion nfft_par3d

das komplette lokale Feld der Prozessoren, die an der FFTW teilnahmen, an die übrigen Prozessoren versand werden. Es bleibt zu untersuchen, ab welchen Problemgrößen sich diese Kommunikation lohnt.

Verbesserte Stützstellenverteilung

Sind die Stützstellen sehr ungleichmäßig verteilt, kann man die Anzahl der an der FFTW beteiligten Prozessoren herabsetzen. Dadurch verlangsamt sich die FFTW, allerdings können stark geballte Stützstellen an mehrere Prozessoren verteilt werden. Dies illustrieren die Abbildungen 15 und 16 .

Zusammenfassung

Die Laufzeitmessung der seriellen und parallelen Variante zeigen, dass die gestellten Probleme sehr zufriedenstellend gelöst wurden. Das dreidimensionale serielle Programm ist deutlich schnelle als die Bibliotheksfunktion der NFFT, wobei man noch Messungen mit anderen Compilern berücksichtigen sollte.
Es wurde gezeigt, dass sich die NFFT sehr gut parallelisieren lässt.

Mögliche Verbesserungen

Trotz der guten Ergebnisse gibt es noch einige Verbesserungen, welche in der zur Verfügung stehenden Zeit nicht mehr implentiert werden konnten. So lässt die Laufzeitbetrachtung des seriellen Programms im Vergleich mit dem parallelen den Schluss zu, dass man mit einem optimierten Speicherzugriff die Cache-Verwaltung unterstützen könnte und so eine deutliche Verbesserung der seriellen Laufzeit erreichen könnte. Dies könnte man z.B. mit einer Sortierung der Stützstellen erreichen.
Ein weiterer Punkt ist der begrenzte Speicher jedes einzelnen Prozessors. Hier gibt es noch Verbesserungen, wenn man die Eingabedaten sofort inplace in das Feld einspeichert, welches der FFTW übergeben wird. Die Einteilung des Feldes in Streifen limitiert das Aufteilen der Daten auf eine große Anzahl von Prozessoren. Man könnte noch andere FFT-Bibliotheken testen, welche die Daten räumlich aufteilen. Eventuell wird diese Option auch irgendwann in der FFTW implementiert.
Das Senden der Streifen geschieht im Ring, allerdings ist dies nur implentiert, indem jeder Prozess an den Rang vor ihm sendet. Man könnte eine Topologie anlegen, welche dem Ring optimal entspricht.
Die Sortierung der Stützstellen ist momentan dem Benutzer überlassen. Da dies sehr kompliziert werden kann, kann man z.B. einen parallelen Sortieralgorithmus anbieten um den Benutzer zu unterstützen.
Die Initialisierung kann noch optimiert werden. Man kann die ersten beiden Initialisierungsroutinen zusammenfassen, da sie völlig unabhängig voneinander sind. Damit der Benutzer nicht selbst alle Daten

N	32	64	128	256	512
nfft_par3d	1.51e-1	3.86e-1	2.97e0	1.74e1	1.51e2

Abbildung 13: Laufzeit der nfft_par3d in Abhängigkeit von der Problemgröße

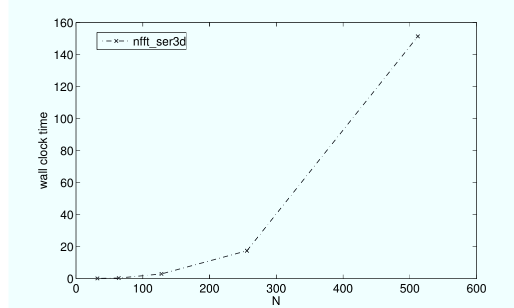


Abbildung 14: Laufzeit der nfft_par3d in Abhängigkeit von der Problemgröße für 64 Prozessoren

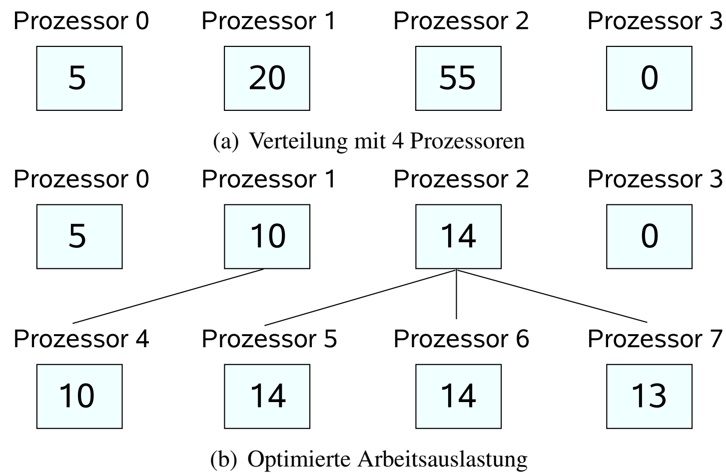
übergeben muss, kann man die nötigen Parameter auch in den NFFT-Plan aufnehmen.

Danksagungen

An dieser Stelle möchte ich mich für die Unterstützung bedanken, die ich während der 10 Wochen im Gaststudentenprogramm erhalten habe. Ich danke meinen Betreuern Matthias Bolten und Godehard Sutmann, außerdem Stefan Kunis und Daniel Potts, welche den Kontakt zum JSC hergestellt haben, sowie den anderen Gaststudenten.

Literatur

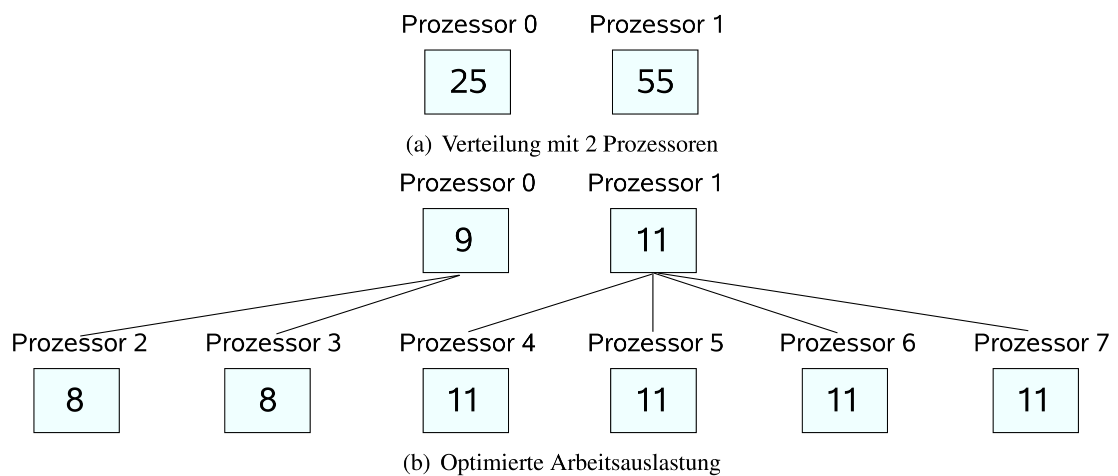
1. J. Keiner, S. Kunis, D. Potts, NFFT 3.0 Tutorial (2006), `http://www-user.tu-chemnitz.de/~potts/nfft/`
2. M. Frigo, S. G. Johnson, FFTW 3.2alpha2 Tutorial (2006), `http://www.fftw.org/fftw-3.2alpha2-doc/`



(a) Verteilung mit 4 Prozessoren

(b) Optimierte Arbeitsauslastung

Abbildung 15: Verbesserte Stützstellenverteilung, Bsp. 1



(a) Verteilung mit 2 Prozessoren

(b) Optimierte Arbeitsauslastung

Abbildung 16: Verbesserte Stützstellenverteilung, Bsp. 2

Fehlerabschätzungen in der Fast Multipole Method bei Systemen mit periodischen Randbedingungen

Alexander Rüttgers

Institut für Numerische Simulation
Universität Bonn

E-mail: ruettger@ins.uni-bonn.de

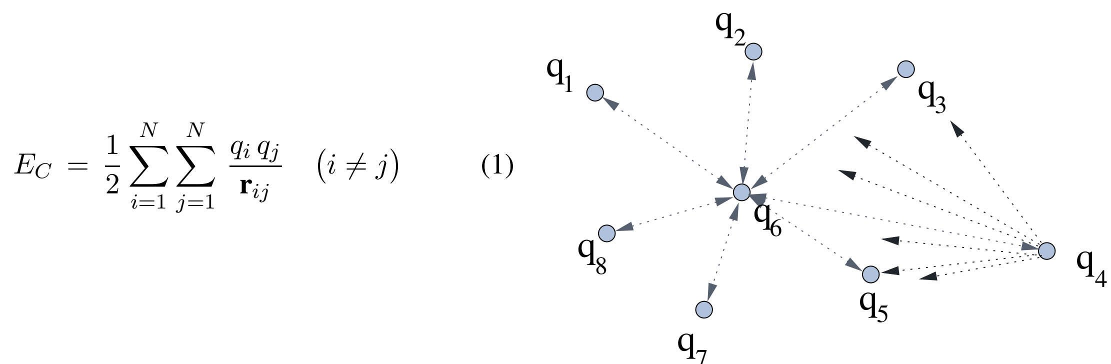
Zusammenfassung:
Die Fast Multipole Method (FMM) erlaubt eine Reduktion der Komplexität des Coulomb-Problems von $\mathcal{O}(N^2)$ auf $\mathcal{O}(N)$. Nach einer Vorstellung des theoretischen Hintergrundes der FMM, wird der bei der Berechnung auftretende Fehler betrachtet und in einen numerisch berechenbaren Ausdruck umgeformt. Die Zielsetzung der Arbeit bestand in der Berechnung des Fehlerterms, um die Genauigkeit der bereits bestehenden FMM-Implementierung abschätzen zu können. Das zur Fehlerberechnung entwickelte Programm wird im Anschluss daran vorgestellt und in der Nutzung beschrieben.

1. Einführung in die Fast Multipole Method

Motivation

In wissenschaftlichen Anwendungen wie der Moleküldynamik tritt das Problem der Berechnung der Coulomb-Energie eines N-Teilchensystems auf. Die direkte Lösung des Coulomb-Problems erfordert die Behandlung aller auftretenden Teilchenwechselwirkungen und skaliert in der Berechnung mit $\mathcal{O}(N^2)$.

$$E_C = \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \frac{q_i \, q_j}{\mathbf{r}_{ij}} \quad (i \neq j) \qquad (1)$$



Aufgrund dieser Komplexität ist die Größe der zu simulierenden Teilchensysteme stark begrenzt. Da das Coulomb-Potenzial aufgrund seiner Skalierung mit $\frac{1}{\mathbf{r}}$ zu den langreichweitigen Potenzialen zählt, ist eine Vernachlässigung weit entfernter Ladungsanteile nicht möglich. Eine Problemlösung muss also bei der Reduktion der Komplexität ansetzen.

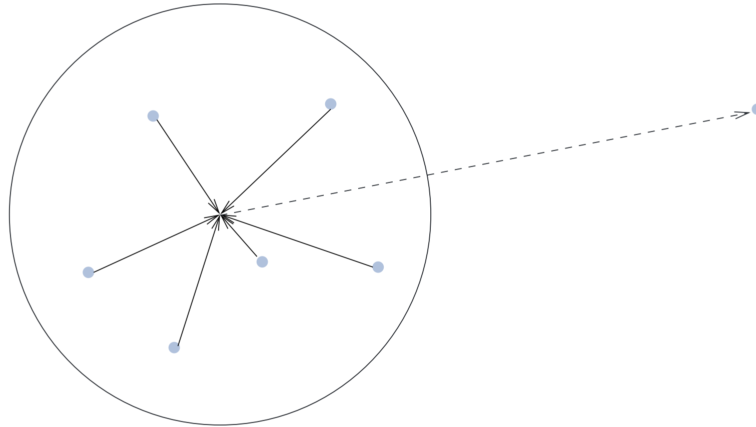


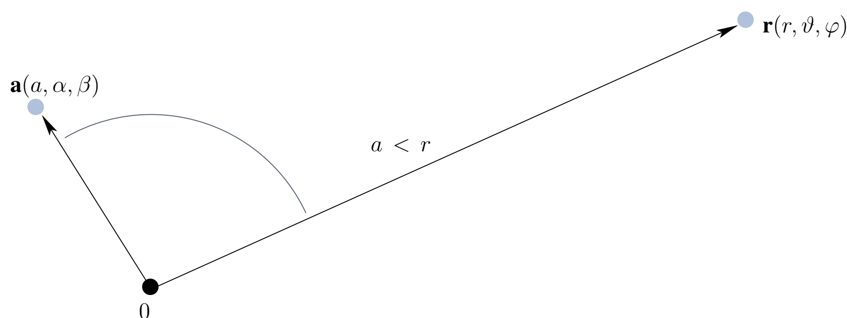
Abbildung 1: Zusammenfassung eines fernen Ladungsclusters

Grundüberlegungen zur FMM

Die Fast Multipole Method (FMM) geht auf Arbeiten von Leslie Greengard [1] aus dem Jahr 1988 zurück. Weitere Entwicklungen erfolgten durch White and Head-Gordon [2, 3]. Die FMM erlaubt eine Reduktion des Aufwandes in der Fernfeldberechnung, so dass die Komplexität des Problems in der Größenordnung von $\mathcal{O}(N)$ liegt. Dadurch wird es möglich, Teilchensysteme mit einer Größe von einigen Milliarden Ladungen numerisch zu behandeln.

Die Vereinfachung durch die FMM basiert auf dem Ansatz, dass ein Cluster beieinander liegender Ladungen aus der Distanz als eine gemeinsame Ladung angesehen wird. Diese Ladungszusammenfassung umfasst umso mehr Partikel, je weiter diese von der Wechselwirkungsladung entfernt liegt. Darauf basiert die (für das Problem optimale) lineare Skalierung. Die Berechnung des Nahfeldes erfolgt weiterhin direkt und skaliert quadratisch. Im Hinblick auf die Rechengeschwindigkeit ist es nahe liegend, nur das unmittelbare Nachbarfeld direkt auszuwerten. Dies hat als Nachteil zur Folge, dass das Konvergenzverhalten der Multipolnäherung verschlechtert wird. Die spätere Implementierung erfordert aus diesem Grund ein sorgsames Abwägen der auftretenden Effekte in der Behandlung von Nah- und Fernfeld.

Entwicklung der Multipolnäherung



Wir vereinfachen nachfolgend die Problemstellung auf die Interaktion zweier Ladungen an den Positionen $\mathbf{a}(a, \alpha, \beta)$ und $\mathbf{r}(r, \vartheta, \varphi)$ mit $a < r$. Wir verwenden zur Beschreibung der Ladungen Kugelkoordinaten mit dem Radius a bzw. r, dem Polarwinkel α bzw. ϑ und dem Azimutwinkel β bzw. φ. Wir sind an einer Entwicklung des inversen Abstandes $\frac{1}{|\mathbf{r}-\mathbf{a}|}$ in einen Fernfeld- und einen Nahfeldanteil interessiert. Diese Entwicklung besitzt die Form

$$\frac{1}{|\mathbf{r}-\mathbf{a}|} = \sum_{l=0}^{\infty}\sum_{m=-l}^{l}\frac{(l-m)!}{(l+m)!}\frac{a^l}{r^{l+1}}P_{lm}(cos\alpha)P_{lm}(cos\vartheta)e^{-im(\beta-\varphi)} \qquad \text{für} \quad a < r \tag{2}$$

Mit P_{lm} bezeichnen wir die assoziierten Legendre-Polynome, die sich als Lösung der allgemeinen Legendre-Gleichung ergeben (vgl. [4]). Diese Differentialgleichung besitzt nur für ganzzahlige l und m mit $0 \leqslant m \leqslant l$ nichtsinguläre Lösungen.

Die Entwicklung des inversen Abstandes setzt sich zusammen aus:

Multipolentwicklung des Nahfeldanteils. Die auftretenden Momente bezeichnen wir mit $\omega_{lm} = qO_{lm}$.

Taylorentwicklung des Fernfeldanteils. Die Taylorkoeffizienten werden mit $\mu_{lm} = qM_{lm}$ bezeichnet.

Die ladungsfreien Momente $O_{lm}(\mathbf{a})$ bzw. Koeffizienten $M_{lm}(\mathbf{r})$ können wir wie folgt schreiben:

$$\begin{aligned} O_{lm}(\mathbf{a}) &= a^l \frac{1}{(l+m)!} P_{lm}(\cos\alpha)\, e^{-im\beta} \\ M_{lm}(\mathbf{r}) &= \frac{1}{r^{l+1}} (l-m)! P_{lm}(\cos\vartheta)\, e^{+im\varphi} \end{aligned} \tag{3}$$

Dies ermöglicht die Darstellung von Gleichung (2) in der Form

$$\frac{1}{|\mathbf{r}-\mathbf{a}|} = \sum_{l=0}^{\infty}\sum_{m=-l}^{l} O_{lm}(\mathbf{a})\; M_{lm}(\mathbf{r}) \qquad \text{für} \quad a < r. \tag{4}$$

Vorbereitungschritt

In der FMM wird eine hierarchische Unterteilung des Rechengebietes vorgenommen. Ausgehend von der Konstruktion einer Mutterbox, erfolgt eine fortlaufende Untergliederung des behandelten Raumes. Zunächst werden alle Teilchen von einer Box der Dimension $[0,1] \times [0,1] \times [0,1]$ umschlossen. Man befindet sich zu diesem Zeitpunkt auf dem ersten Level des Verfahrens. Die Skalierung der Teilchenkoordinaten erfolgt im Hinblick auf die numerische Stabilität des späteren Verfahrens. Davon ausgehend wird die Umgebungsbox ("**Mutterbox**") in Richtung jeder Koordinatenachse halbiert. Es ergeben sich auf dem zweiten Level 8 Kindboxen, die im nächsten Verfeinerungsschritt weiter unterteilt werden. Bei Abbruch des Verfahrens auf Level l ergeben sich 8^{l-1} Kindboxen. Die Darstellung der Unterteilung der Boxen wird in Abbildung 2 für den $\mathbb{R}^2$ dargestellt.

Die Halbierung wird so lange fortgesetzt bis

- ein geforderter Energiefehler nicht überschritten und
- das Minimum der Rechenzeit für diesen Fehler

erreicht wird.

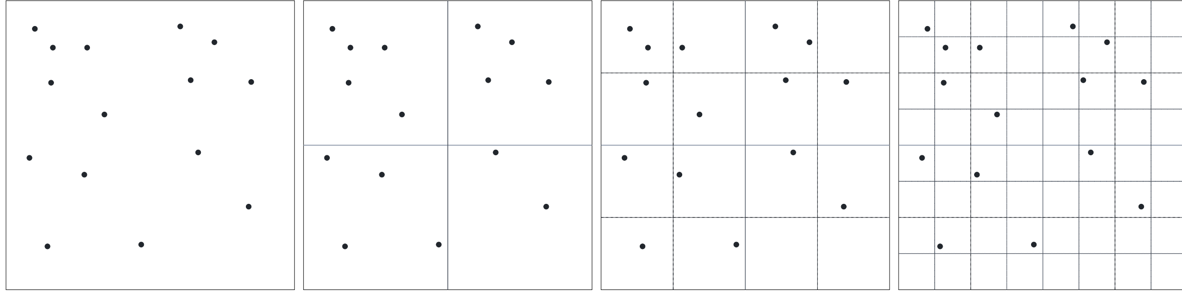


Abbildung 2: Die Abbildung verdeutlicht die Verfeinerung der Boxen bei fortschreitendem Level l

Ablauf der FMM

Das Verfahren kann in seinem Ablauf in fünf Teilschritte untergliedert werden, die nachfolgend beschrieben werden. Für den Ablauf werden drei Operatoren (A,B,C) vorgestellt:

Operator A wird zur Translation der Multipolentwicklung in einen anderen Entwicklungspunkt benötigt.

Operator B transformiert eine Multipolentwicklung um Punkt 1 in eine Taylorentwicklung um Punkt 2.

Operator C ermöglicht die Verschiebung der Taylorentwicklung in einen anderen Entwicklungspunkt.

Schritt 1: *Operator A: Translation der Multipolmomente*

Zunächst erfolgt eine Berechnung der Multipolmomente ω_{lm} der Kinderboxen bzgl. der Boxzentren (auf dem „lowest level"). Die Momente einer jeden Box werden von den anderen Boxen zur Bestimmung des Fernfeldes benötigt. Um die Komplexität des Problems zu reduzieren werden die Multipolmomente als Träger der Fernfeldinformationen zunächst ("nach oben") in die Zentren der Mutterboxen verschoben. Diese können die von den Unterboxen erhaltenen Momente direkt aufsummieren, da alle Momente den gleichen Entwicklungspunkt (das Zentrum der Mutterbox) besitzen.

Zur Durchführung der beschriebenen Translation wird der Operator **A** eingeführt. Dieser ist gekennzeichnet durch die Operation

$$\omega_{lm}(\mathbf{a}+\mathbf{b}) = \sum_{j=0}^{l} \sum_{k=-j}^{j} A_{jk}^{lm}(\mathbf{b})\,\omega_{jk}(\mathbf{a}), \tag{5}$$

wobei **a** das Zentrum der Kindbox und **a+b** das Zentrum der Mutterbox bezeichnet und definiert wird durch

$$A_{jk}^{lm}(\mathbf{b}) = O_{l-j,m-k}(\mathbf{b}) \tag{6}$$

mit den ladungsfreien Momenten $O_{l-j,m-k}$.

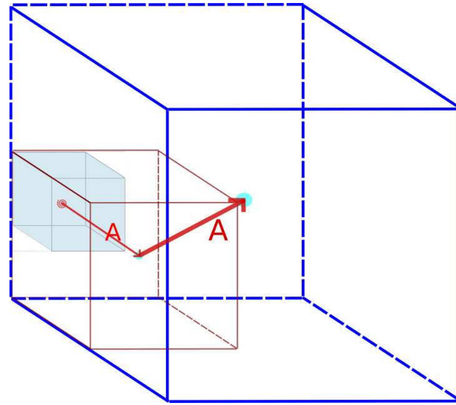


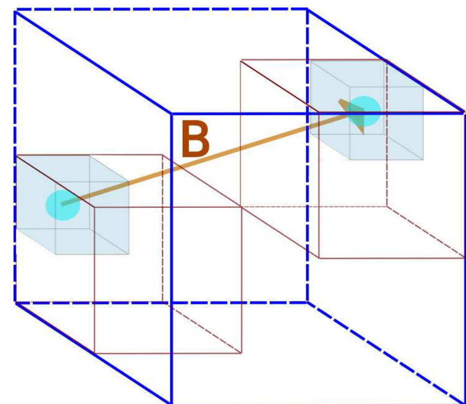
Abbildung 3: Schritt 1: Die verschiedenen Multipolmomente in den Kindboxen werden auf der „höheren“ Ebene der Mutterbox zusammengefasst

Schritt 2: *Transformationsoperator B*

Im nachfolgenden Schritt wird der Operator **B** genutzt, um die zuvor durchgeführten Multipolentwicklungen in Taylorentwicklungen umzuformen. Die Multipolmomente von Box 1 werden dafür in Taylorkoeffizienten von Box 2 umgeformt und können von der Empfängerbox aufaddiert werden (gleicher Entwicklungspunkt). Die Transformation durch Operator **B** erfolgt nur für nicht beieinander liegende Boxen (da dieses Verfahren nur für Fernfeldberechnung angewendet werden kann), deren Eltern Nachbarn sind (keine weitere Ersparnis durch Translation „nach oben“ mit Operator **A** mehr möglich).

Der Operator **B** ermöglicht die Umformung

$$\mu_{lm}\left(\mathbf{a}-\mathbf{b}\right) = (-1)^l \sum_{j=0}^{\infty} \sum_{k=-j}^{j} B_{jk}^{lm}\left(\mathbf{b}\right) \omega_{jk}\left(\mathbf{a}\right) \qquad (7)$$



zwischen den verschieden Zentren der Blöcke. Man kann zeigen, dass Operator **B** die Form

$$B_{jk}^{lm} = M_{l+j,m+k} \quad \text{besitzt} \qquad (8)$$

Schritt 3: *Operator C: Translation der Taylorkoeffizienten*

In diesem Schritt wird der Operator **C** eingesetzt, um die Taylorkoeffizienten ("nach unten") auf die Zentren der Kindboxen zu verschieben. Dadurch lässt sich Schritt 3 als Umkehrschritt zu Schritt 1 auffassen. Nach Beendigung von Schritt 3 enthält jede Box auf dem „lowest level“ alle Informationen des Fernfeldes und man kann die Coulomb-Energie des Fernfeldes bestimmen.

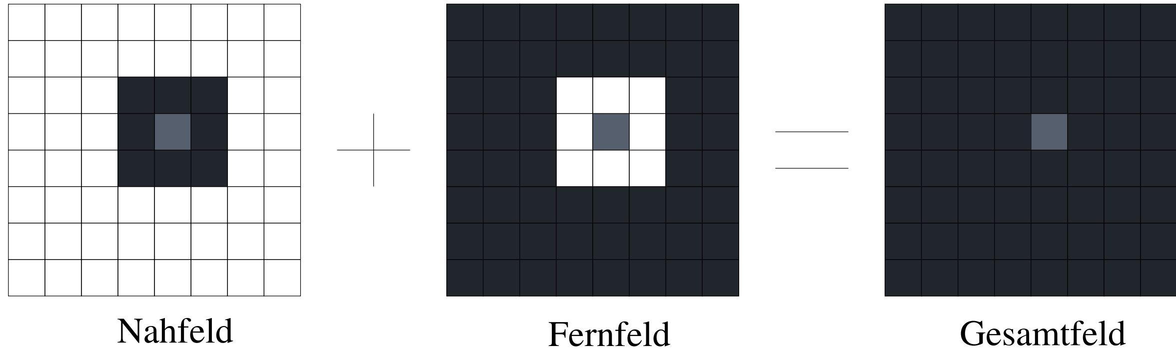
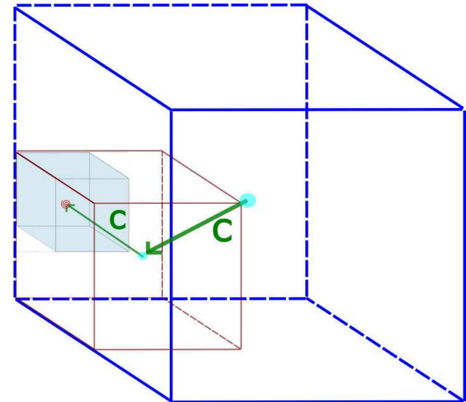


Abbildung 4: Schritt $4 + 5$: Getrennte Berechnung: Das Fernfeld wird durch die beschriebene Multipolnäherung berechnet, während die Auswertung des Nahfelds weiterhin direkt erfolgt. In der Abbildung besteht das Nahfeld einer Box nur aus den unmittelbaren Nachbarboxen. Dies hat einen geringen Berechnungsaufwand zur Folge, jedoch wird das Konvergenzverhalten der Multipolentwicklung verschlechtert. Ein äußerer Parameter der FMM besteht deshalb in der Festlegung der Weite des Nahfeldes auf die umgebenden ws („well-seperated") Nachbargebiete.

Der für die Durchführung nötige Operator **C** gestattet die Translation

$$\mu_{lm}(\mathbf{r} - \mathbf{b}) = \sum_{j=l}^{\infty} \sum_{k=-j}^{j} C_{jk}^{lm}(\mathbf{b})\, \mu_{jk}(\mathbf{r}). \tag{9}$$



Der Operator **C** kann durch die Beziehung

$$C_{jk} = O_{l-j,k-m} \tag{10}$$

bestimmt werden.

Schritt $4 + 5$: $E_{C,Fern}$ bzw. $E_{C,Ges}$ berechnen

In Schritt 4 wird die Coulomb-Energie des Fernfeldes bestimmt. Aufgrund der vorhergehenden Vereinfachungen skaliert diese Berechnung mit der Komplexität $\mathcal{O}(N)$. Die Coulomb-Energie für das Nahfeld muss weiterhin durch direkte Summation (Auswertung der Doppelsumme) ausgerechnet werden, was für das Nahfeld einen Berechnungsaufwand in der Größenordnung von $\mathcal{O}(N^2)$ bedeutet. Das Nahfeld einer Box besteht einerseits aus der aktuellen Box und andererseits aus den benachbarten Boxen. Der Parameter ws (i.A. beträgt $ws = 1$) wird vor der Berechnung festgelegt.

Betrachtung von periodischen Randbedingungen

In den Anwendungen ist man an Systemen mit einer großen Teilchenanzahl interessiert. Trotz einer linearen Skalierung sind die zu behandelnden Probleme in ihrer Ausdehnung beschränkt. Um das Verhalten eines in seiner Größe unbeschränkten Teilchensystems zu simulieren, verwenden wir periodische Randbedingungen.

Der gesamte Raum wird mit Boxen ausgefüllt, die die folgenden Eigenschaften erfüllen:

1. In allen betrachteten Boxen befindet sich eine identische Ladungsverteilung.

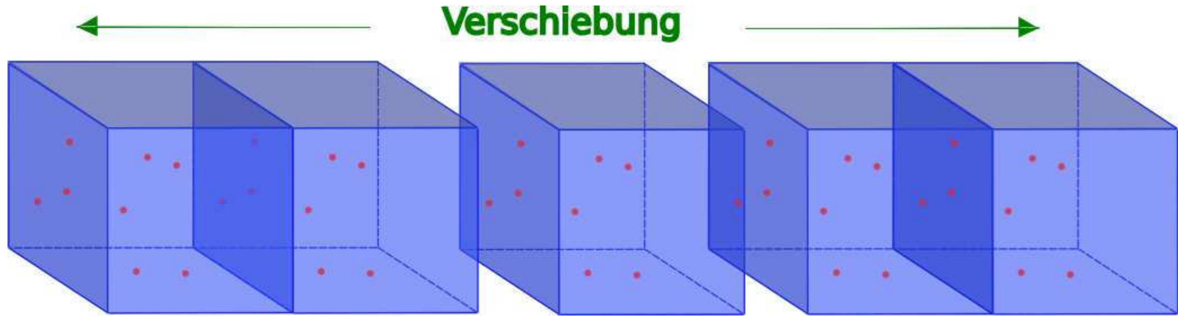


Abbildung 5: Darstellung der räumlichen Verschiebung einer Ausgangsbox in einer Dimension

2. Eine Unterscheidung zwischen den einzelnen Boxen kann nur im Hinblick auf den räumlichen Aufenthaltsort erfolgen.

3. Die unterschiedlichen Boxpositionen lassen sich durch den Verschiebungsvektor **n** angeben, der stets eine Verschiebung um ganzzahlige Vielfache einer Boxlänge bewirkt.

Die Coulomb-Energie für dieses System wird bestimmt durch Gleichung (11):

$$E = \frac{1}{2}\sum_{i=1}^{N}\sum_{\substack{j=1\\i\neq j}}^{N}\frac{q_i\,q_j}{\mathbf{r}_{ij}} + \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\underbrace{\sum_{n_x}^{\infty}\sum_{n_y}^{\infty}\sum_{n_z}^{\infty}}_{\mathbf{n}\neq\vec{0}}\frac{q_i\,q_j}{\mathbf{r}_i-(\mathbf{r}_j+\mathbf{n})} \quad (\text{Verschiebungsvektor } \mathbf{n}) \tag{11}$$

2. Fehlerabschätzungen

Dieser Abschnitt behandelt die Bestimmung eines Fehlerterms für die Fernfeldenergie zwischen zwei Boxen. Die Bestimmung dieser Energie führt auf die Auswertung von Gleichung (12). Trotz der Tatsache, dass in dem Ausdruck eine unendliche Reihe über j auftaucht, kann man die Summation von m und j miteinander vertauschen.

$$\begin{aligned} E_{Lattice} &= \frac{1}{2}\sum_{l=0}^{\infty}\sum_{m=-l}^{l}\sum_{j=0}^{\infty}\sum_{k=-j}^{j}(-1)^l\omega_{lm}M_{l+j,m+k}\omega_{jk} \\ &= \frac{1}{2}\sum_{l=0}^{\infty}\sum_{j=0}^{\infty}\sum_{m=-l}^{l}\sum_{k=-j}^{j}(-1)^l\omega_{lm}M_{l+j,m+k}\omega_{jk} \end{aligned} \tag{12}$$

Zur Vereinfachung betrachten wir zunächst nur die beiden äußeren Summen. Diese lassen sich wie folgt aufspalten:

$$\sum_{l=0}^{\infty}\sum_{j=0}^{\infty}\cdots = (\sum_{l=0}^{p}\cdots + \sum_{l=p+1}^{\infty}\cdots)(\sum_{j=0}^{p}\cdots + \sum_{j=p+1}^{\infty}\cdots)$$

Durch Auswerten der vier Summen ergibt sich Gleichung (13). In der FMM bestimmten wir nur die erste Summe von Gleichung (13). Die übrigen Terme drücken den bei der Berechnung gemachten Fehler aus und werden nachfolgend abgeschätzt.

$$E_{Lattice} = \underbrace{\frac{1}{2}\sum_{l=0}^{p}\sum_{j=0}^{p}(-1)^l \ldots}_{Berechneter\,Wert} + \underbrace{\frac{1}{2}\sum_{l=0}^{p}\sum_{j=p+1}^{\infty}(-1)^l \ldots}_{Fehler}$$
$$+ \underbrace{\frac{1}{2}\sum_{l=p+1}^{\infty}\sum_{j=0}^{p}(-1)^l \ldots + \frac{1}{2}\sum_{l=p+1}^{\infty}\sum_{j=p+1}^{\infty}(-1)^l \ldots}_{Fehler} \quad (13)$$

Wir sind nur an dem Fehlerausdruck interessiert, so dass sich nachfolgende Umformungen nur auf $\Delta E_{Lattice}$ beziehen.

$$\begin{aligned}\Delta E_{Lattice} &= \frac{1}{2}\sum_{l=0}^{p}\sum_{j=p+1}^{\infty}(-1)^l \ldots + \frac{1}{2}\sum_{l=p+1}^{\infty}\sum_{j=0}^{p}(-1)^l \ldots + \frac{1}{2}\sum_{l=p+1}^{\infty}\sum_{j=p+1}^{\infty}(-1)^l \ldots \\ &= \frac{1}{2}\sum_{l=0}^{p}\sum_{j=p+1}^{\infty}(-1)^l \ldots + \frac{1}{2}\sum_{l=0}^{p}\sum_{j=p+1}^{\infty}(-1)^j \ldots + \frac{1}{2}\sum_{l=p+1}^{\infty}\sum_{j=p+1}^{\infty}(-1)^l \ldots\end{aligned} \quad (14)$$

Schreibt man (14) wieder vollständig aus, so ergibt sich in Gleichung (15) der Ausdruck

$$\begin{aligned}\Delta E_{Lattice} &= \frac{1}{2}\sum_{l=0}^{p}(-1)^l \sum_{m=-l}^{l}\sum_{j=p+1}^{\infty}\sum_{k=-j}^{j}\omega_{lm}M_{l+j,m+k}\omega_{jk} \\ &+ \frac{1}{2}\sum_{l=0}^{p}\sum_{m=-l}^{l}\sum_{j=p+1}^{\infty}\sum_{k=-j}^{j}(-1)^j\omega_{lm}M_{l+j,m+k}\omega_{jk} \\ &+ \frac{1}{2}\sum_{l=p+1}^{\infty}(-1)^l \sum_{m=-l}^{l}\sum_{j=p+1}^{\infty}\sum_{k=-j}^{j}\omega_{lm}M_{l+j,m+k}\omega_{jk}\end{aligned} \quad (15)$$

Wir schätzen nun die hinteren Doppelsummen in (15) durch
$\sum_{j=p+1}^{\infty}\sum_{k=-j}^{j}(\pm 1)^j\omega_{lm}M_{l+j,m+k}\omega_{jk}$
nach oben ab. Das Einsetzen der Definition der Momente der Multipolentwicklung an der Stelle $\mathbf{r}(r,\vartheta,\varphi)$

$$\omega_{lm}(\mathbf{r},q) = qr^l\frac{1}{(l+m)!}P_{lm}\cos(\vartheta)e^{-im\varphi}$$

liefert für den Fehler den Ausdruck

$$\begin{aligned}\Delta E_{Lattice} &\approx \frac{1}{2}q\sum_{l=0}^{p}(-1)^l r^l \sum_{m=-l}^{l}\frac{1}{(l+m)!}P_{lm}\cos(\vartheta)e^{-im\varphi}\sum_{j=p+1}^{\infty}\sum_{k=-j}^{j}(\pm 1)^j M_{l+j,m+k}\omega_{jk} \\ &+ \frac{1}{2}q\sum_{l=0}^{p}r^l \sum_{m=-l}^{l}\frac{1}{(l+m)!}P_{lm}\cos(\vartheta)e^{-im\varphi}\sum_{j=p+1}^{\infty}\sum_{k=-j}^{j}(\pm 1)^j M_{l+j,m+k}\omega_{jk} \\ &+ \frac{1}{2}q\sum_{l=p+1}^{\infty}(-1)^l r^l \sum_{m=-l}^{l}\frac{1}{(l+m)!}P_{lm}\cos(\vartheta)e^{-im\varphi}\sum_{j=p+1}^{\infty}\sum_{k=-j}^{j}(\pm 1)^j M_{l+j,m+k}\omega_{jk}\end{aligned} \quad (16)$$

Man bemerkt in (16), dass in allen drei Summanden der Ausdruck
$\sum_{m=-l}^{l} \frac{1}{(l+m)!} P_{lm} \cos(\vartheta) e^{-im\varphi} \sum_{j=p+1}^{\infty} \sum_{k=-j}^{j} (\pm 1)^j M_{l+j,m+k} \omega_{jk}$
auftritt. Unabhängig von weiteren Möglichkeiten der Vereinfachung von (16) ist stets diese Teilsumme zu berechnen.

Berechneter Fehlerterm

Setzt man in die Teilsumme von (16) noch die Definition der Momente der Multipolentwicklung an der Position $\mathbf{a}(a, \alpha, \beta)$

$$\omega_{jk}(\mathbf{a}, q) = q a^l \frac{1}{(j+k)!} P_{jk} \cos(\alpha) e^{-ik\beta}$$

ein und zieht die Ladung q aus der Summe heraus, so lautet der im späteren Programm numerisch auszuwertende ladungsunabhängige Ausdruck:

$$\sum_{m=-l}^{l} \frac{1}{(l+m)!} P_{lm}(\cos\vartheta) e^{-im\varphi} \sum_{j=p+1}^{\infty} \sum_{k=-j}^{j} (\pm 1)^j M_{l+j,m+k}\, a^j \frac{1}{(j+k)!} P_{jk}(\cos\alpha) e^{-ik\beta} \tag{17}$$

Äußere Parameter sind:

l Summationsindex

p Länge der Multipolentwicklung

Für die spätere Fehlerabschätzung soll Gleichung (17) für unterschiedliche Werte des Summationsindexes l und der Länge der Multipolentwicklung p berechnet werden. Im nachfolgend beschriebenen Programm wurde die Berechnung für

- Entwicklungsgrad $p = 0, \ldots, 50$
- Summationsindex $l = 0, 1, 2, 3, \ldots, 9, p, p+1$ durchgeführt.

Ein problematischer Aspekt der Gleichung (17) ist die numerische Stabilität. Numerische Instabilität kann an zwei Stellen auftreten.

1. Die Fakultät $\frac{1}{(j+k)!}$ nimmt abhängig von den Größen j und k numerisch kleine Werte an. Wertet man die von j abhängige Summe für die ersten 200 Summanden aus, so nimmt der Index k Werte im Bereich von $-200 \ldots 200$ an. Die Auswertung $\frac{1}{400!} \approx 1.6 \cdot 10^{-869}$ ergibt ein nicht mehr numerisch darstellbares Ergebnis (der numerische Zahlenbereich für den Datentyp *double* wird nach unten durch 10^{-308} begrenzt).
2. Die Gittersumme $M_{l+j,m+k}$ enthält Werte, deren Betrag größer als 10^{308} ($\rightarrow$ Numerischer Zahlenbereich) ist.

Aufgrund der Tatsache, dass sowohl ein *Overflow* als auch ein *Underflow* des Bereichs der Maschinenzahlen auftritt, lässt sich mit Hilfe einer geeigneten Skalierung ein darstellbarer Wert erreichen. Die Skalierung wird mit dem Ziel gewählt, die Gittersumme $M_{l+j,m+k}$ in der Größe zu reduzieren und die zu

bestimmenden Fakultäten $\frac{1}{(j+k)!}$ entsprechend zu vergrößern. Ein numerisch stabilerer Wert für (17) ist durch Ausdruck (18) gegeben.

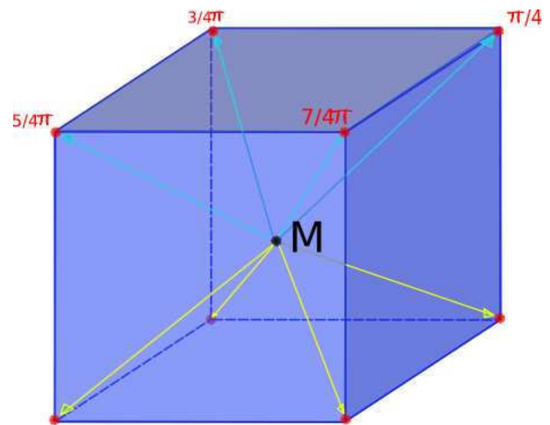
$$\sum_{m=-l}^{l} \frac{l!}{(l+m)!} P_{lm}(\cos\vartheta) e^{-im\varphi} \sum_{j=p+1}^{\infty} \sum_{k=-j}^{j} (\pm 1)^j \frac{M_{l+j,m+k}}{(l+j)!} \frac{(l+j)!}{l! \quad j!} a^j \frac{j!}{(j+k)!} P_{jk}(\cos\alpha) e^{-ik\beta} \tag{18}$$

Anmerkungen zur Berechnung

Prinzipiell sind die Aufenthaltsorte für die Ladung an der Stelle $\mathbf{r}(r, \vartheta, \varphi)$ nicht festgelegt, solange diese innerhalb der betrachteten Box bleiben. Da wir nur eine Abschätzung für den auftretenden Fehler benötigen, untersuchen wir das Maximum über die acht Eckpositionen der Betrachtungsbox des Teilchens.

Für den Polarwinkel ϑ und den Azimutwinkel φ treten folgende Werte auf:

- $\cos(\vartheta) = \pm\frac{1}{\sqrt{3}}$
- $\varphi = \frac{\pi}{4}, \frac{3}{4}\pi, \frac{5}{4}\pi, \frac{7}{4}\pi$



Die Ladung an der Position $\mathbf{a}(a, \alpha, \beta)$ wird auf eine endliche Anzahl von Aufenthaltsmöglichkeiten eingeschränkt. In jeder Raumdimension werden 33 mögliche Aufenthaltspunkte angenommen. Aus diesem Grund ergeben sich für den $\mathbb{R}^3$ eine Anzahl von $33^3 = 35937$ Positionen. In der Implementierung laufen die äußeren drei Schleifen über die 33 Punkte in jeder Raumrichtung. Über jeden dieser Punkte wird das Maximum von Gleichung (18) ausgerechnet.

Es verbleibt in (18) die Auswertung der Summe $\sum_{j=p+1}^{\infty} \ldots$. Diese Summe wurde so weit ausgewertet, wie dies die Genauigkeit des Datentyps **long double** (8 byte bzw. 16 byte) erfordert. In der Praxis ergaben sich sinnvolle Auswertungsgrenzen bei „$\infty \leq 200$“.

In Gleichung (19) wird ein kurzer Überblick über den auszuwertenden Ausdruck für alle 33^3 Positionen, die Berechnungen für die Länge der Multipolentwicklung $p = 0, \ldots, 50$ und den Summationsindex $l = 0, 1, 2, 3, \ldots, 9, p, p+1$ gegeben.

$$\underbrace{\sum_{m=-l}^{l} \frac{l!}{(l+m)!}}_{l=0,\ldots,9,p,p+1} \underbrace{P_{lm}(\cos\vartheta) e^{-im\varphi}}_{\cos(\vartheta)=\pm\frac{1}{\sqrt{3}}, \varphi=\frac{\pi}{4},\frac{3}{4}\pi,\frac{5}{4}\pi,\frac{7}{4}\pi} \underbrace{\sum_{j=p+1}^{\not\infty\, 200}}_{p=0,\ldots,50} \sum_{k=-j}^{j} (\pm 1)^j \frac{M_{l+j,m+k}}{(l+j)!} \frac{(l+j)!}{l!\, j!} \underbrace{a^j \frac{j!}{(j+k)!} P_{jk}(\cos\alpha) e^{-ik\beta}}_{33^3 \text{ Werte}} \tag{19}$$

3. Programmbeschreibung

Aufbau des Programms

Das Programm zur Auswertung von (19) wurde in der Programmiersprache C verfasst. Das Programm setzt das Prinzip der modularen Programmierung um. Es besteht neben dem Hauptprogramm aus vier Modulen. Die wichtigsten Funktionen in diesen Modulen werden nachfolgend beschrieben.

1. **parse.c** Einlesen der Koeffizienten $M_{l+j,m+k}$, Initialisierungen
2. **calc.c** Berechnung der Fakultäten, Legendre-Polynome, Mehrfachsummen, ...
3. **parallel.c** Parallelisierung der Summenberechnung
4. **visual.c** Ausgabe der berechneten Werte für jedes p
   $\longrightarrow$ Ausgabe in 51 Dateien

In dem Modul **parse.c** wird die Funktion „**void** Parse_M_lm (**REAL** $**M$, **int** $*size$)“ zum Einlesen der Gittersumme $M_{l+j,m+k}$ verwendet. Die Funktion erwartet, dass sich im Verzeichnis der ausführbaren Datei eine Datei „m_daten“ mit den einzulesenden Ausdrücken befindet. Die $M_{l+j,m+k}$ werden in einem dynamisch allozierten Feld des Datentyps long double gespeichert. Bei dem Argument REAL **M handelt es sich um einen Zeiger auf dieses Feld.

Die Funktion „**int** Parse_Args (**char** $*Inputfile, \ldots$)“ wird für das (optionale) Einlesen eines Parameterfiles benötigt. Das Paramterfile muss im Arbeitsverzeichnis mit der Bezeichnung „Inputfile“ vorliegen und erlaubt die Variation folgender Größen:

1. **GRID** Anzahl der Gitterpunkte in jeder Raumdimension
2. **SIZE** Anzahl der Punkte in der Liste m_daten
3. **PMIN** Kürzester Entwicklungsterm p der Multipolentwicklung
4. **PMAX** Längster Entwicklungsterm p der Multipolentwicklung

Das Modul **calc.c** beinhaltet mathematische Funktionen zur Bestimmung des Terms (19). Die Funktion „**void** RightTerm (**REAL** $**M, \ldots$)“ bestimmt die Teilsumme $\sum_{j=p+1}^{\infty}\sum_{k=-j}^{j}(\pm1)^j\frac{M_{l+j,m+k}}{(l+j)!}\frac{(l+j)!}{l!\quad j!}a^j\frac{j!}{(j+k)!}P_{jk}(\cos\alpha)e^{-ik\beta}$. Die vollständige Summenbestimmung erfolgt durch die Funktion „**REAL** MainSumMax (**REAL** $**M$, **int** $tag, \ldots$)“.

Die Parallelisierung des Programms erfolgt mit den Funktionen des Moduls **parallel.c**. Wichtig ist hier die Funktion „**void** ExchangeData (**VALUE** $*result, \ldots$)“. Diese organisiert das Zusammenfügen der von jedem Prozessor berechneten Teildaten. Die Daten werden getrennt für jeden Wert von p an den Root-Prozess geschickt, der diese dann in einer vorgegebenen Reihenfolge ausgibt.

Zuletzt erfolgt mit den Funktionen des Moduls **visual.c** die Ausgabe der Daten. Das Zusammenfügen der im Speicher angelegten Daten ist in der Funktion „**REAL** ResultSearch (**VALUE** $result[]$, **int** $anzahl$, ...)“ implementiert. Diese durchläuft das Feld mit den gespeicherten Ergebnissen und liefert zu einem festen Gitterpunkt $(i,\ j,\ k)$ und Werten von l und p den entsprechenden Eintrag zurück. Das Schreiben der Daten in Dateien mit dem Namen „result_p*.dat“ (das Ersetzungszeichen bezeichnet einen Wert für p von 0 bis 50) erfolgt in der Funktion „**void** Print_Result (**VALUE** $result[]$, **int** $anzahl$, **int** p)“.

Allgemeine Konstanten, Makros und Variablen, die im Verlauf der Berechnung in jedem Modul verwendet werden, sind in der Headerdatei **typen.h** deklariert. Das Makro INFINTY gibt die obere Grenze der

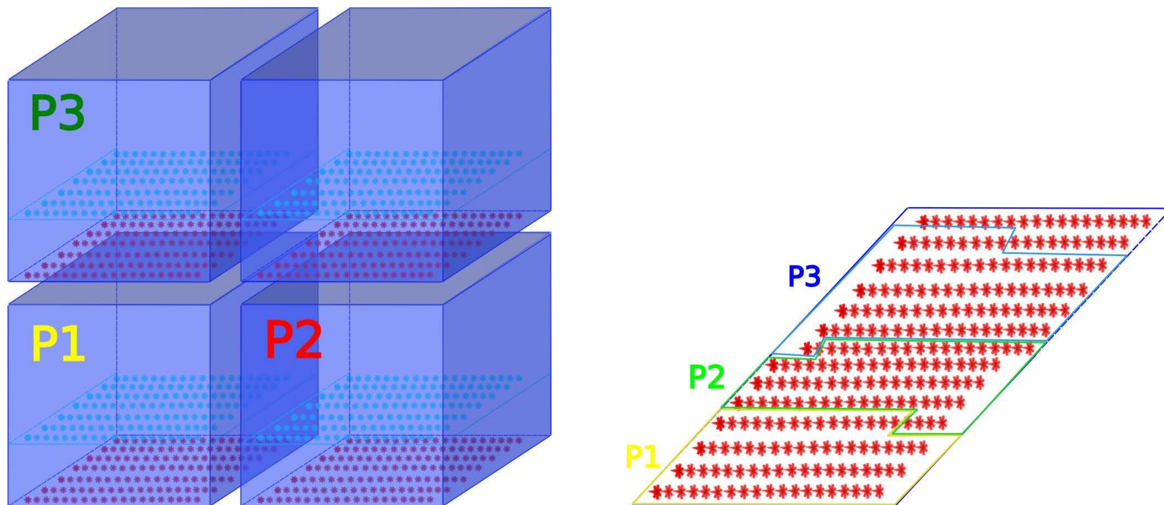


Abbildung 6: Gegenüberstellung der beiden Parallelisierungsansätze

Summe über j an. Prinzipiell handelt es sich um eine unendliche Reihe, die bei einer Auswertung bis 200 in den ersten zwanzig gültigen Stellen des Ergebnisses keine Veränderung mehr zeigt. Die Rechenzeit hängt stark von der gewählten Grenze ab, so dass man auf Kosten der Genauigkeit der Rechnung diese Grenze herabsetzen kann.

Parallelisierungsstrategien

Die Berechnung der in Gleichung (19) beschriebenen Ausdrücke ist mit einem sequentiellen Programm nicht mehr in vertretbarer Rechenzeit zu realisieren. Während des Gaststudentenprogramms konnten die Rechenkapazitäten des JSC genutzt werden. Dazu gehören der Supercomputer IBM p690-Cluster Jump mit 1312 Prozessoren und der massiv-parallele Supercomputer JUBL mit 16.384 Prozessoren. Da die Berechnung von (19) für die verschiedenen Gitterpunkte unabhängig voneinander erfolgen kann, bietet sich in der Implementierung eine möglichst homogene Zerlegung der Gitterpunkte an. Im Allgemeinen (sofern die Anzahl der verwendeten Prozessoren keine Zweierpotenz ist) wird diese Zerlegung nicht vollkommen gleichmäßig sein, da die 33^3 Gitterpunkte keine Zweierpotenz darstellen.

Es wurde zwei verschiedene Gebietszerlegungansätze programmiert. Die Darstellung dieser Ansätze ist in Abbildung 6 dargestellt.

- Zerlegung des Rechengebietes in Teilwürfel
- Abzählen der Gebietspunkte und Aufteilung auf verschiedenen Prozessoren

Bei der Zerlegung in Teilwürfel wird in jeder Raumdimension das Gebiet auf die in dieser Dimension vorhandene Prozessoranzahl aufgeteilt. Aufgrund der Anzahl von 33 Punkten kann sich die Gebietsgrenze und damit die Anzahl der Berechnungspunkte in jeder Dimension um ein Element unterscheiden. Dieser Effekt kann, da er in jeder Dimension auftritt, zu unerwartet asymmetrischen Gebietsaufteilungen führen. Im Extremfall kann Prozessor 1 in jeder Dimension einen Punkt und Prozessor 2 in jeder Dimension zwei Punkte zu berechnen haben. Im Ergebnis hat Prozessor 2 die achtfache Punkteanzahl von Prozessor 1 zu berechnen. Diese Parallelisierungsstrategie würde Vorteile bieten, wenn ein hoher Kommunikationsbedarf über die Randgebiete bestehen würde. Das günstige Verhältnis von Gebietsoberfläche zum Volumen reduziert die Anzahl an Randpunkten, die gegebenenfalls von einem Würfel an die Nachbarn weitergegeben werden müssten. Da im betrachteten Fall ein Datenaustausch nur für die Ergebniszusammenfassung erfolgen muss, bietet dies im vorliegenden Fall keinen Vorteil.

Verwendet man ein einfaches Abzählen aller Punkte, so lässt sich eine gleichmäßige Punkteaufteilung (in der Homogenität bis auf einen Punkt genau) erzielen. Das Rechengebiet wird in diesem Fall beliebig aufgeteilt, was aufgrund der Problemstellung keinerlei Nachteile zur Folge hat. Man lässt jeden Prozessor in den Schleifen über das zu berechnende Gebiet laufen und führt die Rechenoperation nur für den Fall durch, dass dieser Punkt in den vorher bestimmten Gebietsgrenzen i_left und i_right liegt. Eine Implementierung besitzt dann die folgende Form:

```
int division = 0;

/* Schleife ueber alle Gitterzellen */
for( i = -16; i <= 16; i++)
for( j = -16; j <= 16; j++)
for( k = -16; k <= 16; k++)
{
/* Soll Berechnung durchgefuehrt werden */
     if( division >= i_left && division <= i_right )
     {

          /* Main-Loop */

     }
        division++;
}
```

Da im Nachhinein schwer rekonstruiert werden kann, welche Datenpunkte jeder Prozessor bestimmt hat, muss diese Information direkt nach der Berechnung dem Ergebnis hinzugefügt werden. Das Ergebnis ist durch die Angabe der Raumkoordinaten i, j, k, des Summationsindexes l und der Länge der Multipolentwicklung p eindeutig bestimmt. In der Programmierung mit C bietet sich das Speichern in einer Struktur an. Die Struktur kann die nachfolgende Form annehmen und besitzt für jeden Datenpunkt eine Größe von 26 byte, sofern $REAL$ einen Datentyp *long double* mit einer Größe von 16 byte bezeichnet.

```
/*Fuer das Speichern des Ergebnisses */
typedef struct value
{
short i,j,k,p,l;

REAL result_lp;
} VALUE;
```

Schwierigkeiten in der Umsetzung

Die Anforderungen an das zu entwickelnde Programm haben neben der Korrektheit der Berechnung im wesentlichen die folgenden Aspekte umfasst:

1. Die Genauigkeit der Ausgabe sollte bei mindestens 18 gültigen Stellen liegen.
2. Effizienz in der Programmierung, um die notwendige Rechenzeit zu reduzieren.

Hürden in der Umsetzung dieser Anforderung traten insbesondere bei dem ersten Punkt auf. Der im Allgemeinen verwendete Datentyp double besitzt bei einer nativen Größe von 8 byte eine Genauigkeit

von 15 Stellen. Diese Genauigkeit war für die spätere Auswertung nicht ausreichend. Der auf JUMP verwendete IBM XL C/C++ - Compiler bietet diesbezüglich die folgende Möglichkeit. Setzt man bei der Übersetzung das Flag „-qldbl128“ und bindet die zugehörigen Bibliotheken hinzu, so wird die Größe für den Datentyp long double auf 16 byte erhöht (die native Größe dieses Datentyps liegt für diesen Compiler bei 8 byte). Dies ermöglicht eine Genauigkeit der Umsetzung in der Größenordnung von 30 Stellen.

Bei dem Versuch diese Genauigkeit in der Programmierung zu erreichen, mussten einige Aspekte berücksichtigt werden.

1. Bei der Parallelisierung mit MPI kann nur noch ein eigener Datentyp für das Ergebnis verwendet werden. Der Datentyp MPI_LONG_DOUBLE kann nicht mehr als Pendant zu dem Typ long double der Größe 16 byte angesehen werden. Ein Versuch der (fehlerhaften) Umsetzung führt zu einem Abbruch des Programms.

2. Es werden nun Gleitpunktoperationen zwischen Gleitpunktzahlen der doppelten Größe durchgeführt. Dies führt zu einer Verlangsamung der Rechenoperationen und damit des ausführbaren Programms um den Faktor 4-5. Dadurch wird die Bedeutung der zweiten Anforderung an das zu schreibende Programm herausgestellt.

3. Scheinbar enthält die Implementierung der Potenzfunktion „**long double** powl(**long double** x, **long double** y)“ einen Fehler, der sich bei einer Basis mit negativer Mantisse zeigt. Potenzen dieser Basis ergeben den Ausdruck "Not a Number Quiet"(NaNQ), falls das Flag „-qldbl128“ gesetzt ist. Ohne das entsprechende Flag zeigt die Funktion dieses Verhalten nicht. In der Umsetzung wurden Ausdrücke der Form a^j in $|a|^j$ umgeformt und auf das entsprechende Vorzeichen untersucht.

Rechenzeit des Programms

Es wurden vor der eigentlichen Gesamtrechnung verschiedene Teilausdrücke (zumeist nur für einen festen Wert des Indexes p) durchgeführt. Dies sollte einerseits die Genauigkeit der späteren Lösung im Vorfeld kontrollieren und andererseits eine Abschätzung für die spätere Gesamtrechenzeit liefern. Das Programm wurde mit dem Skript „mpcc_r“ (zur Einbindung der MPI-Bibliotheken) und den Optionen „-O4 -qldbl128 -qlanglvl=stdc99“ übersetzt. Ohne die Option „-qldbl128“ lässt sich das Programm mit „-O5“ übersetzen. Die Rechenzeit wird dadurch wesentlich reduziert, jedoch das Ergebnis der Rechnung nur auf etwa 13 Stellen genau bestimmt.

Die Messung der Rechenzeit des Programms auf JUMP bei Verwendung von 16 Prozessoren für

- die obere Summationsgrenze „$\infty = 200$“,
- dem Entwicklungsgrad $p = 9$ und
- dem Datentyp long double mit einer Größe von 16 byte

ergab eine Rechenzeit von **80 Minuten**. Die Gesamtrechnung umfasst alle 51 Entwicklungsgrade p. Man muss für 128 Prozessoren eine Rechenzeit von **12 Stunden** für die Gesamtrechnung einplanen. Das Ergebnis dieser Rechnung erzielt dann die erforderliche Genauigkeit.

Danksagung

Mein besonderer Dank gilt Dr. Holger Dachsel für die vorzügliche Betreuung während des zehnwöchigen Gaststudentenprogramms. Seinen vielfältigen Anregungen und Impulsen ist es zu verdanken, dass viele Ungenauigkeiten, die in der Rechnung aufgetreten sind, beseitigt werden konnten. Ansonsten hätte die Genauigkeit der Rechnung nur in der Größenordnung von etwa 13 Stellen gelegen. Weiterhin möchte ich mich bei Matthias Bolten für die perfekte Organisation bedanken. Ich kann mir nicht vorstellen, dass die Organisation noch in irgendeiner Hinsicht verbessert werden könnte. Zuletzt danke ich den anderen Teilnehmern des Gaststudentenprogramms für den vielfältigen Gedankenaustausch und die angenehme Zeit.

Literatur

1. R. Beatson and L. Greengard. A short course on fast multipole methods.
   http://math.nyu.edu/faculty/greengar/shortcourse_fmm.pdf
2. C. A. White and M. Head-Gordon. Derivation and efficient implementation of the fast multipole method. J. Chem. Phys., 101:6593-6605, October 1994
3. C. A. White and M. Head-Gordon. Rotating around the quartic angular momentum barrier in fast multipole method calculations. J. Chem. Phys., 105:5061-5067, September 1996
4. E. W. Weisstein, Legendre Polynomial, From Mathworld – A Wolfram Web Resource.
   http://www.mathworld.wolfram.com/LegendrePolynomial.html

Implementation of a Quantum Monte Carlo Code on BGL

Andreas Uhe

Rheinisch-Westfälische Technische Hochschule Aachen,
Fachgruppe Chemie,
Landoltweg 1, 52074 Aachen

E-mail: andreas.uhe@rwth-aachen.de

Abstract: The parallelisation of the Quantum Monte Carlo program Quantum Magic[1] is described. Speed-up measurements on the parallel computer systems JUMP and JUBL are presented. Diffusion Monte Carlo was applied to test the effect of the usage of STO-6G molecular orbitals combined with a STO basis as trial function with the first row atoms as examples. Also multireference wave functions are applied to Be, B and C. Furthermore the development of the maximum time step with growing atomic number is examined using He, Ne and Ar as test cases. Application of the program to the small molecules N_2 and ethylene is conducted. Variational Monte Carlo calculations were used for preprocessing in order to get acceptable distributions of walkers.

Introduction

Quantum Chemistry deals with the quantum mechanical treatment of electronic manybody systems to evaluate molecular properties. Properties of particular interest are the ground state and excitation energies. Energy differences between different molecular geometries yield activation or reaction energies, which are of fundamental importance in Chemistry. Unfortunately, the exact mathematical treatment is possible for two-particle systems, only. For more complicated systems approximative techniques such as the Hartree Fock method yield about 99 % of the energy of a system. The high percentage of the total energy recovered is somewhat misleading. Energies of chemical accuracy require an error of no more than $1\,\mathrm{kcal\,mol}^{-1}$, which corresponds to $\approx 10^{-3}$ % of the total energy of a single carbon atom. The energy difference between the exact, non-relativistic energy and the Hartree-Fock energy is termed electron correlation energy. Electron correlation methods such as Coupled Cluster Theory recover the electron correlation energy almost quantitatively. Unfortunately, their computational cost scales at best $O(N^7)$ where N is measure for the size of the molecule which can easily lead to insurmountable computational work even for the most advanced computer systems existing today. For energy differences like reaction energies there is some amount of error cancelation making it possible to use results of lower accuracy.

A group of relatively new procedures apply Monte Carlo methods to quantum chemical problems and promise a more favourable accuracy to effort ratio. There is still a lot of work to be done to investigate these methods and keep them up to date with the constantly increasing potentials of available computer systems. In this report porting of the serial Monte Carlo program Quantum Magic[1] to the massively parallel machines JUMP and JUBL situated in the Jülich Supercomputing Centre (JSC) is described as well as the application of the code to various simple test cases for which literature presents accurate reference data.

Theoretical Background

Fundamentals of Quantum Chemistry

In Quantum Chemistry, the motions of electrons and nuclei is usually separated (Born-Oppenheimer approxmation) and solely the electrons are treated quantum mechanically. The electronic state of a molecule or atom is completely described by its electronic wave function which depends on one spin and three space coordinates for each electron. All possible states Ψ_i of a system are given by the eigenfunctions of the hamiltonian $\mathcal{H}$ of the system as expressed by the stationary Schrödinger equation (1).

$$\mathcal{H}\Psi_i = E_i\Psi_i \tag{1}$$
$$\text{with } \mathcal{H} = \frac{1}{2}\sum_i \nabla_i^2 - \sum_{iA} \frac{Z_A}{|r_i - R_A|} + \sum_{i>j} \frac{1}{r_i - r_j}$$

As there is no analytical solution for the eigenfunctions of a hamiltonian containing more than one electron, one has to use approximative methods in order to find these eigenfunctions and eigenvalues. The so far most successful way was to express the many electron wave function as combination of one-electron functions ϕ_i (orbital approximation). The many-electron wave function is expanded in an antisymmetrised product of one-electron functions denoted a Slater Determinant Δ (2). The Hartree-Fock method expands the wave function in a single Slater Determinant, whereas electron correlation methods employ linear combinations of determinants.

$$\Psi(\tau_1, \ldots, \tau_n) = \frac{1}{\sqrt{n!}}\hat{\mathcal{A}}\pi\phi_1(\tau_1) \cdot \ldots \cdot \phi_n(\tau_n) = \Delta \tag{2}$$

The one-electron functions consist of a spatial part and a spin part where the spatial part is constructed as linear combination of usually (but not necessarily) atom centered basis functions χ_μ.

$$\phi_i(\tau) = \varphi_i(r)\sigma_i = \left(\sum_{\mu=1}^{n} c_{i\mu}\chi_\mu(r)\right)\sigma_i \tag{3}$$

The Hamiltonian contains no higher then two particle interactions, so that no more than n^4 integrals contribute to the expectation value of the total energy where n denotes the number of basis functions (Slater Condon rules). The wave function is computed by minimizing the total energy with respect to the n^2 variational parameters $c_{i\mu}$.

Monte Carlo Methods[2]

Originally Monte Carlo is a method for estimating a definite integral with boundaries a and b as indicated in (4) by taking $(b-a)$-times the average value of the function f.

$$F = \int_a^b f(x)dx = \lim_{N\to\infty} F_N \quad \text{, with} \quad F_N = \frac{b-a}{N}\sum_{j=1}^{N} f(X_j) \tag{4}$$

There are several variants of Monte Carlo techniques applicable to quantum chemistry. In this work Variational Monte Carlo (VMC) was primarily used for preprocessing while Diffusion Monte Carlo (DMC) was used to evaluate energy expectation values.

Variational Monte Carlo

In VMC the energy expectation value of the wave function is evaluated by the sum (5) over the local energy E_L without reference to analytical integrals over basis functions. For the exact wave function the local energy equals the exact energy and is a constant, as the kinetic contribution cancels the divergent potential energy contributions exactly. For approximate Ψ this cancelation is not exact and E_L has nonzero variance.

$$\langle E\rangle = \frac{\langle\Psi\,|\mathcal{H}|\,\Psi\rangle}{\langle\Psi|\Psi\rangle} = \frac{\int \Psi^2 \cdot \frac{\mathcal{H}\Psi}{\Psi}}{\int \Psi^2} = \lim_{N\to\infty} \frac{\sum_{j=1}^{N} \frac{\mathcal{H}\Psi}{\Psi}}{\sum_{j=1}^{N} 1} = \frac{1}{N}\sum_{j=1}^{N} E_L(j) \tag{5}$$

An advantage of VMC is that the (input) wave function Ψ can have an arbitrary form provided analytical first and second derivatives exist. A very compact form is economic to reduce the overhead for the frequent numerical evaluation of the wave function and its derivatives at given points in space. To ensure proper antisymmetry of the wave function commonly a Slater determinant is complemented by a symmetric correlation factor Ψ_{CF} (6). Specifically, Ψ_{CF} takes the form of a product of two Padé-Jastrow type functions (7) with the parameters a,b,λ_A and ν_A.

$$\Psi = \left(\sum_i \Delta_i\right)\Psi_{CF} \tag{6}$$

$$\Psi_{CF} = e^{\sum_{i=1}^{n}\sum_{j<i}^{n}\frac{ar_{ij}}{1+br_{ij}}}\, e^{\sum_{i=1}^{n}\sum_{A=1}^{N}\frac{\lambda_A r_{iA}}{1+\nu_A r_{iA}}} \tag{7}$$

Random Walk

A Random Walk is an effective technique to sample the high-dimensional wave function Ψ. A random walk consists of a sequence of sets of values - one for each variable - sampling the phase space. The walker, following such a random walk, takes a step at random. After each step the local energy E_L is computed according to equation (5). Taking the average over all walkers and all steps gives an estimate of the exact expectation value of the input wave function. Typically a Random Walk is divided in several blocks that contain a fixed number of steps where each block has its own estimate for the local energy. Then the overall result for the energy is the average over the results of each block. With a sufficiently large number of steps per block one can assume the results of each block as independent of each other and the standard deviation is given by (8).

$$\sigma^2 = \frac{\left|\langle E_L^2\rangle - \langle E_L\rangle^2\right|}{M-1} \tag{8}$$

Importance sampling

A simple Random Walk would take too many steps to produce an acceptable result and it would yield an unfavorable standard deviation. The technique of importance sampling in form of the Metropolis algorithm circumvents this difficulty. In this algorithm proposed steps are accepted with a probability that depends on the electron density for initial and final positions which ensures that the walker distribution is biased by the electron density of the input wave function.

Diffusion Monte Carlo

For DMC the starting point is the time dependent Schrödinger equation (9) with substituted imaginary time $\tau = it$. Initially, E_T is an arbitrary energy shift that approaches the ground state energy with evolving imaginary time.

$$
\begin{aligned}
-\frac{\partial \phi\left(x,\tau\right)}{\partial \tau} &= \left(\mathcal{H} - E_T\right)\phi\left(x,\tau\right) \\
&= \frac{1}{2}\nabla^2 \phi\left(x,\tau\right) + \left(E_T - V\left(x\right)\right)\phi\left(x,\tau\right)
\end{aligned}
\tag{9}
$$

DMC belongs to the Green's function methods which in principle do not need an input wave function, but solve the Schrödinger equation itself iteratively (10).

$$
\begin{aligned}
\phi\left(\mathbf{y},\tau+\delta\tau\right) &= \int G\left(\mathbf{y},\mathbf{x},\delta\tau\right)\phi\left(\mathbf{x},\tau\right) d\mathbf{x} \\
\text{with } G\left(\mathbf{y},\mathbf{x},\delta\tau\right) &= \left\langle \mathbf{y} \left| e^{-(\mathcal{H}-E_T)\delta\tau} \right| \mathbf{x} \right\rangle
\end{aligned}
\tag{10}
$$

This series converges to the ground state wave function for $\tau \to \infty$ with exponential decay of the excited states as can be shown by formally expanding the wave function as linear combination of the eigenfunctions of $\mathcal{H}$. Thus this method is capable of finding the exact solution to the Schrödinger equation. $G\left(\mathbf{y},\mathbf{x},\delta\tau\right)$ can be interpreted as probability that a set of particles moves from position **x** to position **y** in an imaginary time step $\delta\tau$ and the problem lies in finding an analytical expression for G.

Short time approximation

In DMC the problem is solved by separating the kinetic and potential energy part of the Hamilton operator in two different processes and factorising $G \approx G_{diff}G_B$. The correction term

$$
G - G_{diff}G_B = \frac{1}{2}\left[\mathcal{V},\mathcal{T}\right]\left(\delta\tau\right)^2 + O\left(\left(\delta\tau\right)^3\right) \tag{11}
$$

shows that this is only a good approximation for small time steps $\delta\tau$ because the operators for potential and kinetic energy do not commute. Hence, the time step decreases with increasing kinetic energy of the electrons so that the core electrons of the atom with the highest atomic number determines the time step. Yet, the short time approximation is advantageous, because it yields the two solvable equations (12) and (13) to determine G. The exact result can be estimated by executing several simulations with different time steps and extrapolation to $\delta\tau \to 0$.

The kinetic energy part is analogous to the diffusion equation while the potential energy part finds its analogy in a first order rate equation with branching process.

$$-\frac{\partial G_{diff}(\mathbf{x},\mathbf{y},\delta\tau)}{\partial\tau} = \frac{1}{2}\nabla^2 G_{diff}(\mathbf{x},\mathbf{y},\delta\tau) \tag{12}$$

and

$$-\frac{\partial G_B(\mathbf{x},\mathbf{y},\delta\tau)}{\partial\tau} = (E_T - V(x))\, G_B(\mathbf{x},\mathbf{y},\delta\tau) \tag{13}$$

G_{diff} represents the acceptance criterion for the movement of the walkers very similar to the importance sampling in VMC. G_B may be represented by assigning weights to each Walker or by changing the numbers of walkers with a probability based on G_B. In this case both possibilities are combined so that every walker has a weight and is destroyed or multiplied if the weight reaches a specified minimum or maximum value, respectively.

Importance sampling

In DMC a slightly different form of importance sampling is used to improve the effectiveness of the method. As the exact wave function is unknown one has to use a trial function Ψ to speed up the convergence of the walker distribution to the exact wave function. To achieve this, equation (9) is multiplied with the trial function and the product is substituted with the new function $f(x,\tau) = \phi(x,\tau)\,\Psi(x)$ yielding equation (14).

$$\frac{\partial f(x,\tau)}{\partial\tau} = \frac{1}{2}\nabla^2 f(x,\tau) - \frac{1}{2}\nabla\left(f(x,\tau)\,\mathbf{F}_Q(x)\right) + (E_T - E_L(x))\, f(x,\tau) \tag{14}$$

$$\text{with} \quad \mathbf{F}_Q \equiv \nabla ln\,|\Psi|^2 = \frac{2\nabla\Psi}{\Psi}$$

This adds an external field to the diffusion part of the equation, so the vector field $\mathbf{F}_Q$ can be interpreted as effective velocity of the random walkers away from region where Ψ is small. The result of the simulation for $\tau \to \infty$ is unaffected by the trial function, but a good trial wave function will speed up convergence while reducing the standard deviation.

Fixed-node approximation

Another problem of DMC arises from the fact that the particles of interest in Quantum Chemistry are electrons. As electrons are fermions, their wave function is antisymmetric with respect to exchange of any two electrons, whereas no such restriction has been imposed on our simulation so far. By rejecting all movements for which a walker crosses a node of the trial function it is ensured that $\phi(x,\tau)$ retains the nodes of the trial function. The error associated with the so-called fixed node approximation is of second and higher order as a perturbational analysis shows.

Advantages and disadvantages of Quantum Monte Carlo

DMC is an essentially basis set free method, thereby avoiding the slow convergence of the electron correlation energy with respect to the length of the one-electron basis set expansion. This is especially advantageous e.g. for dispersion interactions, which are solely an electron correlation effect. Formally, QMC techniques are of order $O(N^3)$, albeit with a large pre-factor. Compared to conventional electron

correlation methods, QMC methods may be easily parallelised and have little I/O and memory requirements. A drawback is especially the stochastic nature of this technique causing difficulties in deriving expectation values at high precision. Another problem of QMC is the antisymmetry of the wave function as mentioned in the previous section. Many of these aspects have not yet been explored to the degree required in practical applications.

Results and Discussion

Parallelisation strategy

The parallelisation scheme followed is straightforward: The same rule that was applied in dividing a Random Walk into blocks can be applied to dividing a Random Walk into groups of blocks. So basically every processor can perform its own simulation with different walkers and random numbers to yield one result per processor. The average over these results is then the overall result of the run. Analogously (8) gives an estimate of the error.

The input file specified as argument contains the necessary information on the trial wave function and the parameters of the run. It is read in by process 0 and broadcasted. Afterwards every process calculates block by block incrementing a shared counter before starting a new block until the maximum number of blocks is reached. Every process creates one outputfile and one logfile containing the results of its own run and the overall averages over all processes. At the end process 0 creates two files containing all the walker positions and some other restart information. Additional options affect the generation of restart information and the generation of walker ensembles. These options include some amount of synchronisation otherwise essentially no communication incurs except for the startup phase.

Parallel scaling properties

Measurements were conducted as DMC calculation with atomic fluorine as test case where the product of the number of blocks and the number of walkers was 640000 on JUMP and 544000 on JUBL. Figures 1 and 2 show the speed-up of the calculation with respect to the number of processors used. With increasing numbers of processors the deviation from ideal scaling grows. To analyse the impact of reading and broadcasting the input and collecting the results, the communication overhead was collected separately and subtracted from the total execution time. A minimal loss of parallel scaling is caused by the overhead on JUMP (Fig. 1) while it is slightly larger on JUBL(Fig. 2). Reducing the I/O done ($\triangle,\times$) by not writing out the block result before the end of the calculation resulted in a slight improvement on JUMP (figure 1) and no improvement at all on JUBL (figure 2).

The bigger part of speed-up loss is caused by the varying granularity of the block size due to the branching process. The variation of the computational effort associated with a given block can vary by up to one order of magnitude, depending upon the initial conditions and the evolution of the random walk. Even though using a shared counter to dynamically balance the overall load at block level among the processors, this cannot counter balance the dynamically arising imbalance of the individual block sizes. Reducing the block size would improve the overall load balancing. However there is a limit to this approach, because one has to make sure that the block results are still independent from each other. Also in this way only the maximum possible speed-up loss is decreased and not necessarily the actual speed-up loss, so there is still necessity for improvement of the load balancing.

A possible scheme for improvement could be to redistribute the walkers occasionally. Alternatively, a

pool of walkers shared by all processes may be used to achieve an even walker distribution. The disadvantage of these possibilities is that they include either some synchronisation or an increased communication volume. Hence, it is advantageous to develop a simple model in order to estimate whether the improvement in load-balancing can make up for the loss due to synchronization and communication, especially for a large number of processors. Note, that these scaling problems are specific to DMC and do not occur for VMC.

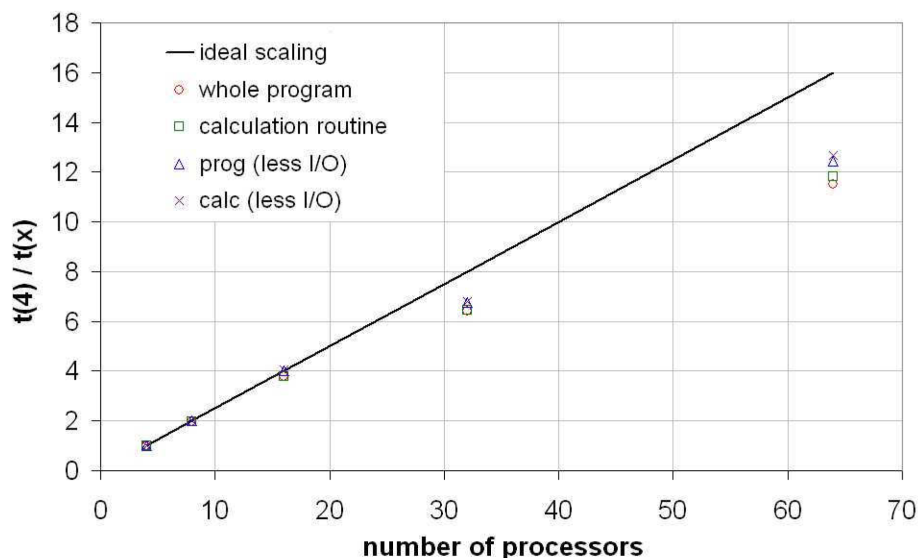


Figure 1: Speed-up on JUMP for complete runtime and for the block calculation only

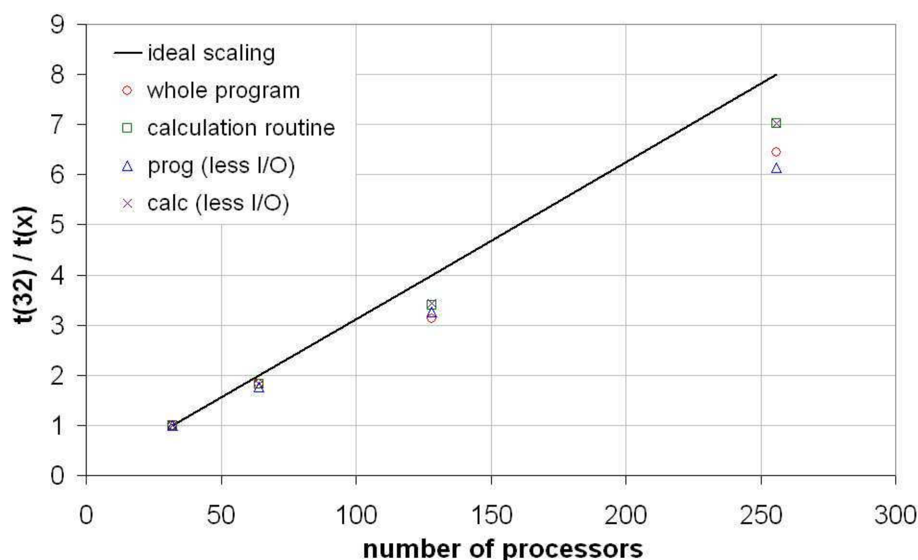


Figure 2: Speed-up on JUBL for complete runtime and for the block calculation only

Applications

A considerable effort has been spent on figuring out how to gain acceptable results with the serial version of Quantum Magic. Even after gaining some experience in performing VMC and DMC calculations with Quantum Magic, some unresolved conspicuities remained. Most apparent were sudden strong variations

of the local energy. To get usable results we excluded blocks that differed more than two times the standard deviation from the average. Another difficulty was the acquisition of walker ensembles, because the random creation often resulted in such bad distributions that no usable results could be obtained with them. We solved this problem by deleting all randomly created walkers not yielding consistent results. Larger ensembles were then created by performing small runs with the remaining walkers adding the walker positions to a list after each block.

The tables given in the appendix collect the details of results and parameter settings for the various calculations. The error bars in the subsequent diagrams refer to the standard deviation of the result.

First-row atoms with STO versus STO-6G trial function

First the total energy of the first row atoms Lithium to Neon were calculated using Bunge's STO basis and MO-Vectors[3] and compared to Davidson's estimates of the exact non-relativistic ground state energies.[4, 5]. Between 85 and $100\,\% \pm 5\,\%$ of the correlation energy could be recovered by DMC calculations. Molecular Quantum Chemistry codes using STO basis sets are fairly uncommon owing to the difficulties to evaluate two-electron-four-center integrals. For DMC calculations, however, the proper description of the electron-nucleus cusp is essential, so that the QMC code offered the possibility to either operate internally in an STO or GTO basis set. Hence, we evaluated the error associated with using the MO vectors from SCF calculations in STO-6G basis to create wave functions in the corresponding STO basis while retaining QMC operating in STO basis. Exponents and coefficients of the corresponding STO-6G expansion to Bunge's STO basis were taken from Stewart's tables[6].

As figure 3 shows, the results are not precisely the same, but within each others standard deviation. Part of the error might occur from the fact that the deviation from the exact electron-nucleus cusp increased from between $2 \cdot 10^{-3}$ and $8 \cdot 10^{-5}$ to between $2 \cdot 10^{-1}$ and $5 \cdot 10^{-2}$ when using the STO-6G MO-Vectors. This might point to an inconsistent normalisation of the wave function, but as we detected this quite late there was no time for further investigation on this. We further note, that the peak at fluorine is presumably due to the rather large time step, as the results have not been extrapolated to zero time step. The details of these calculations are shown in tables 3 and 4.

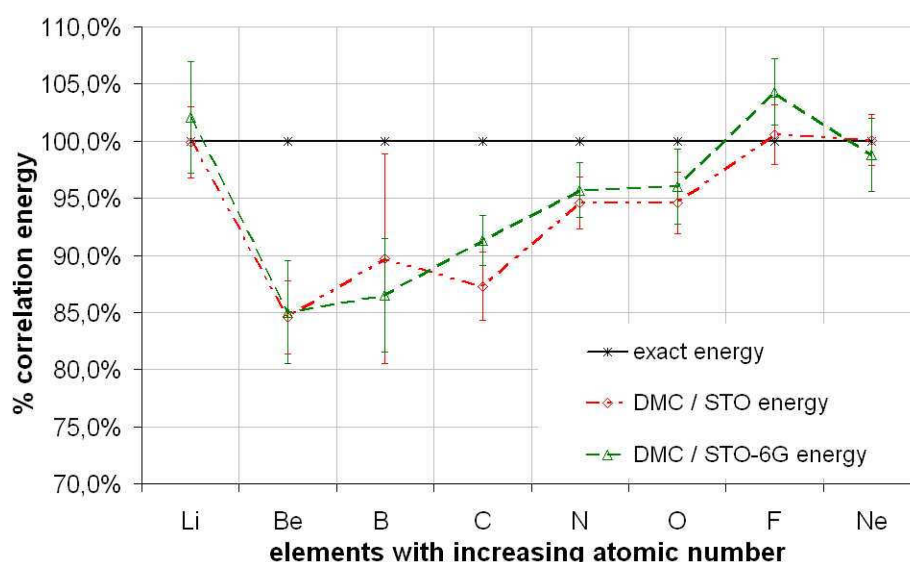


Figure 3: Correlation energy covered with STO and STO-6G wave function

Be, B and C atoms with MCSCF trial function

As a next step we applied MCSCF wave functions to some first row atoms known to have some multireference character, namely beryllium, boron and carbon. To get the MCSCF wave function we added 5 2p-functions to Bunge's basis[3] of Be, 2 3p-functions to Bunge's basis[3] of B and 2 3s-functions to Bunge's basis[3] of C. The exponents for the STO versions of these additional basis functions were taken from Davidson[5] and expanded as STO-6G contraction by Stewart's tables[6]. Then MCSCF calculations in the STO-6G basis were performed and analysed with Columbus[7] to yield MO-Vectors along with a list of configurations with the largest weight in the MCSCF wave function. This multiconfigurational wave function provides the location of the nodes of the wave function for the DMC calculation performed with the STO basis.
As figure 4 shows, the results for these three elements are quite different. For beryllium we obtain nearly the exact energy by including the $1s^22s^2$ and $1s^22p^2$ configurations while including $1s^22s^22p^1$, $1s^22s^23p^1$ and $1s^22p^3$ for boron and $1s^22s^22p^2$, $1s^22p^4$ and $1s^23s^22p^2$ for carbon lead to even higher energies than the single reference wave function. The electron-nucleus cusp of these multireference STO-basis/STO-6G MO-Vector wave functions was in the same range as with the single reference STO-basis/STO-6G MO-Vector wave functions. The details of these calculations are shown in table 5.

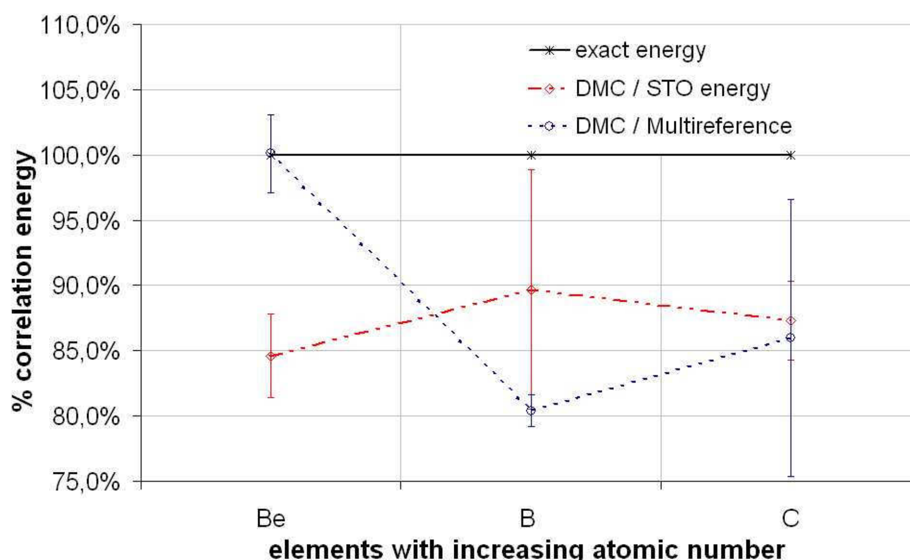


Figure 4: Energies with multireference wave function

Though a multireference wave function cannot worsen the energy result in conventional, variational Quantum Chemical Methods, the effect on the nodes of the wave function is unknown. Investigations regarding the nodes might explain the strange behaviour of the DMC results with MCSCF trial wave functions.

Time step bias in the noble gas group

To execute QMC calculations one has to know which time step to choose in order to get a good estimate in the shortest possible time. Choosing a too large time step leads to the wrong result because of the short time approximation. Choosing a very small time step increases the time until the energy converges to the exact result (within the fixed node approximation).

There are two possibilities how to evaluate the proper time step size for a specific system and it is not obvious on first glance which one is the better. In both cases of course the time step has to be varied, but this affects the number of time steps per block. So one can either decrease the time per block with decreasing time step size to get a constant number of time steps per block or one can leave the block time constant and thus increase the number of time steps per block with decreasing time step size. We investigated both possibilities for the small test system of a helium atom. Figure 5 shows the results for constant block time converging to the exact energy and a similar convergence can be seen in figure 6 for a constant number of steps per block. The difference between the two can be seen in the standard deviation which increases with decreasing block time. So in order to get reliable results one has to keep the block time constant.

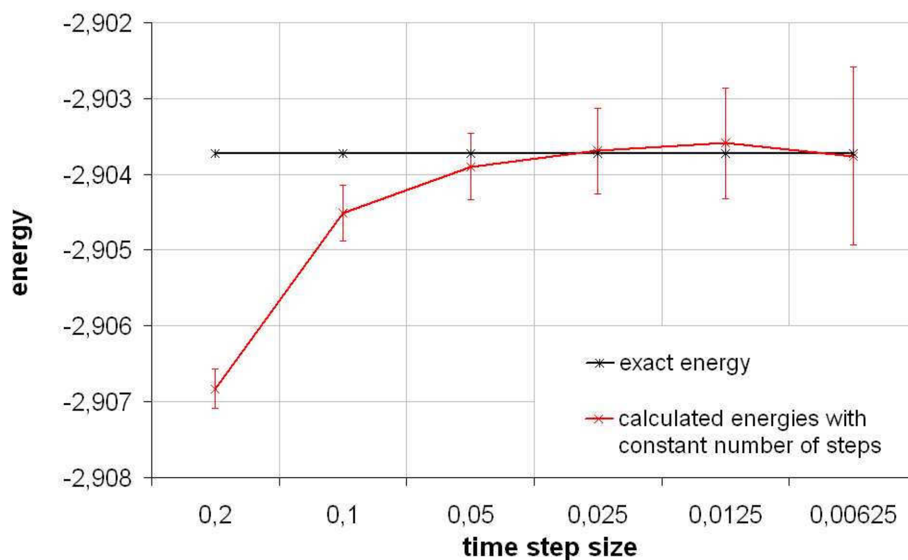


Figure 5: Energy estimate of helium with decreasing time step and constant number of steps per block

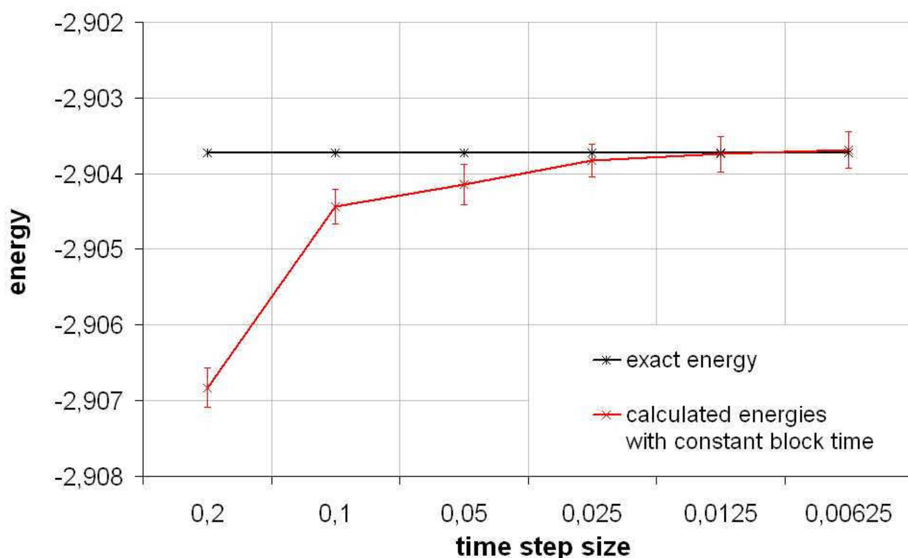


Figure 6: Energy estimate of helium with decreasing time step and constant block time

Regarding the course of this time step bias with increasing atomic numbers we determined the biggest possible time step yielding consistent results also for neon and argon. The results can be seen in table

6 and figures 7 and 8. Table 1 summarizes the estimation of the maximal necessary time step to get consistent results.

He	Ne	Ar
$\approx 10^{-2}$	$\approx 5 \cdot 10^{-4}$	$\approx 10^{-4}$

Table 1: Maximum time step for helium, neon and argon

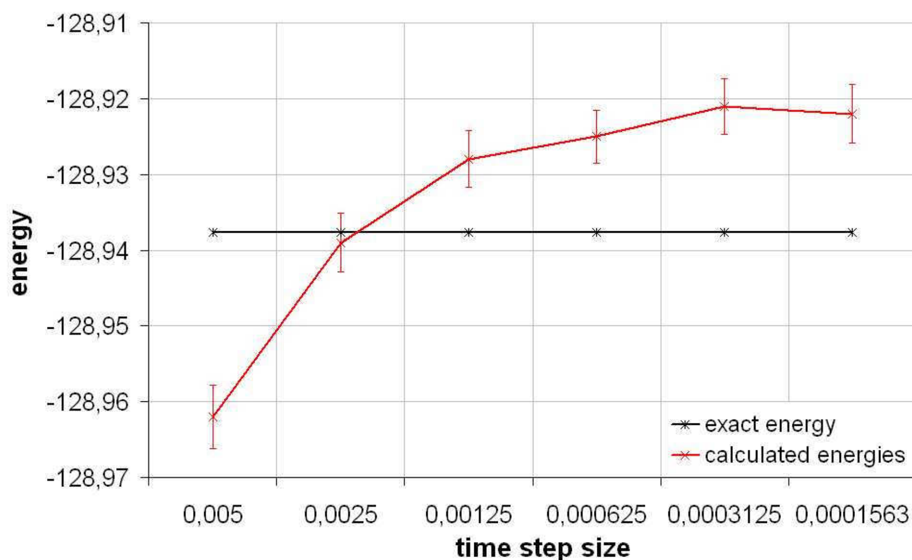


Figure 7: Energy estimate of neon with decreasing time step and constant block time

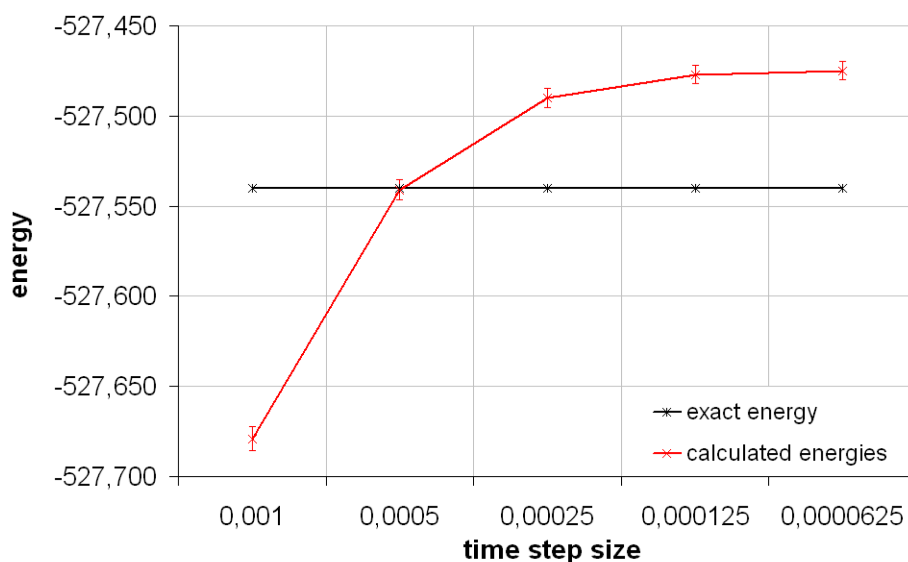


Figure 8: Energy estimate of argon with decreasing time step and constant block time

Simple molecules

As another test of the Monte Carlo program we calculated the energy of ethylene and the nitrogen molecule. We used Bunge's basis[3] and Herzberg's geometry for ethylene[8] and Davidson's geometry

for nitrogen[9] to obtain the STO-6G MO-Vectors with Columbus[7]. With these we performed DMC calculations resulting in energies shown in table 2. Comparison with an estimation of the exact energy of the nitrogen molecule shows our calculated energy to be utterly wrong while the energy of ethylene is at least in the correct range. To improve the energy of the nitrogen molecule we included 3d-functions obtain from Davidson[5] into the basis of the nitrogen atom which did not help at all. We did not find out what went wrong with the energy calculation for the nitrogen molecule. Table 7 shows the details of the calculations.

	ethylene	nitrogen	nitrogen (incl. 3d-functions)
calculated energy	-78.3689	-106.706	-106.455
standard deviation	0.00541	0.00556	0.00701
exact energy[8, 10]	-78,4422	-109.544	-109.544

Table 2: Results of the DMC calculations of ethylene and the nitrogen molecule

Summary

Our parallel version of Quantum Magic has a good speed-up that can further be optimised by better load-balancing. When using STO-6G molecular orbitals combined with a STO basis as trial function the error of the electron nucleus cusp increases. Yet, the deviation from the results using STO molecular orbitals with STO basis is quite small. Using multireference trial functions does not necessarily improve the result of the DMC calculation. The examination of the time step bias with increasing atomic number confirms and quantifies the prediction from theory. The application of the method to small molecules was not succesful and should be examined more closely by future work.

Acknowledgment

I would like to thank Thomas Müller for supervising my work here during which I learned very much. I also want to thank Matthias Bolten for the nice organisation of the guest student program giving us opportunities to look beyond our own noses. I also want to thank the other guest students for giving me such a nice time.

References

1. R.N.Barnett, B.L.Hammond, P.J. Reynolds, L. Terray, W.A. Lester Jr., *Quantum Magic*, Version 7.6.4 (1994), distributed by QCPE.
2. B.L. Hammond, W.A.Lester Jr., P.J. Reynolds *Monte Carlo Methods in Ab Initio Quantum Chemistry*, World Scientific Publishing Co. Pte. Ltd., Singapur 1994.
3. C.F.Bunge, J.A.Barrientos, A.V.Bunge and J.A.Cogordan, *Phys. Rev.*, 1992 **A46**, 3691.
   C.F. Bunge, J.A. Barrientos, A.V. Bunge, *Atomic Data and Nuclear Data Tables*, 1993, **53**, 113.
4. S.J. Chakravorty, S.R. Gwaltney, E.R. Davidson, F.A. Parpia, C.F. Fischer, *Phys. Rev.*, 1991, **A44**, 7071.
5. S.J. Chakravorty, S.R. Gwaltney, E.R. Davidson, F.A. Parpia, C.F. Fischer, *Phys. Rev.*, 1993, **A47**, 3649.
6. R.F. Stewart, *J. Chem. Phys.*, 1970, **52**, 431.
7. H. Lischka, R. Shepard, F.B. Brown and I. Shavitt, *Int. J. Quantum Chem.*, Quantum Chem. Symp., 1981, **15**, 91.
   R. Shepard, I. Shavitt, R.M. Pitzer, D.C. Comeau, M. Pepper, H. Lischka, P.G. Szalay, R. Ahlrichs, F.B. Brown, J. Zhao *Int. J. Quantum Chem.*, Quantum Chem. Symp., 1988, **22**, 149.
   H. Lischka, R. Shepard, R.M. Pitzer, I. Shavitt, M. Dallos, Th. Müller, P.G. Szalay, M. Seth, G.S. Kedziora, S. Yabushita, Z. Zhang, *Phys. Chem. Chem. Phys.*, 2001, **3**, 664.
   H. Lischka, R. Shepard, I. Shavitt, R.M. Pitzer, M. Dallos, Th. Müller, P.G. Szalay, F.B. Brown, R. Ahlrichs, H.J. Böhm, A. Chang, D.C. Comeau, R. Gdanitz, H. Dachsel, C. Ehrhardt, M. Ernzerhof, P. Höchtl, S. Irle, G. Kedziora, T. Kovar, V. Parasuk, M.J.M. Pepper, P. Scharf, H. Schiffer, M. Schindler, M. Schüler, M. Seth, E.A. Stahlberg, J.-G. Zhao, S. Yabushita, Z. Zhang, M. Barbatti, S. Matsika, M. Schuurmann, D.R. Yarkony, S.R. Brozell, E.V. Beck, and J.-P. Blaudeau,

COLUMBUS, an ab initio electronic structure program, release 5.9.1 (2006).
T. Helgaker, H.J.Aa. Jensen, P. Jørgensen, J. Olsen, K. Ruud, H. Ågren, T. Andersen, K.L. Bak, V. Bakken, O. Christiansen, P. Dahle, E.K. Dalskov, T. Enevoldsen, H. Heiberg, H. Hettema, D. Jonsson, S. Kirpekar, R. Kobayashi, H. Koch, K.V. Mikkelsen, P. Norman, M.J. Packer, T. Saue, P.R. Taylor and O. Vahtras, DALTON, an ab initio electronic structure program, Release 1.0, 1997.

8. G. Herzberg, *Electronic Spectra of Polyatomic Molecules, Van Nostrand, Princeton, 1966*
9. D. Feller, C.M. Boyle, E.R. Davidson, *J. Chem. Phys.*, 1987, **86**, 3424.
10. C.C.M. Samson, W. Klopper, *Mol. Phys.*, 2004, **102**, 2499.

Tables

DMC(Bunge/STO)	Li	Be	B	C
HF-limit	-7.4327	-14.5730	-24.5291	-37.6886
correlation energy	-0.0453	-0.0943	-0.1248	-0.1564
exact energy	-7.4781	-14.6674	-24.6539	-37.8450
calculated energy	-7.4780	-14.6528	-24.6411	-37.8252
standard deviation	0.0014	0.0030	0.0115	0.0047
amount of correlation	99.9 %	84.6 %	89.7 %	87.3 %
standard deviation	-3.1 %	-3.2 %	-9.2 %	-3.0 %
time step	0.0100	0.0100	0.0050	0.0050
block time	2.0000	2.0000	1.0000	2.0000
number of processors	16	16	16	16
number of blocks	20	20	20	20
number of walkers	900	500	500	600
quantum force cutoff	-	-	200.0	200.0
	N	O	F	Ne
HF-limit	-54.4009	-74.8094	-99.4093	-128.5471
correlation energy	-0.1883	-0.2579	-0.3246	-0.3905
exact energy	-54.5892	-75.0673	-99.7339	-128.9376
calculated energy	-54.5790	-75.0535	-99.7360	-128.9380
standard deviation	0.0043	0.0070	0.0083	0.0087
amount of correlation	94.6 %	94.6 %	100.6 %	100.1 %
standard deviation	-2.3 %	-2.7 %	-2.6 %	-2.2 %
time step	0.0050	0.0025	0.0050	0.0025
block time	2.0000	1.0000	1.0000	1.0000
number of processors	16	16	16	16
number of blocks	20	20	10	20
number of walkers	750	1000	1000	1000
quantum force cutoff	-	100.0	-	-

Table 3: Details for DMC Calculations with STO-MO Vectors

DMC(Bunge/STO-6G)	Li	Be	B	C
calculated energy	-7.4790	-14.6532	-24.6370	-37.8314
standard deviation	0.0022	0.0042	0.0062	0.0035
amount of correlation	102.1 %	85.0 %	86.5 %	91.3 %
standard deviation	-4.9 %	-4.5 %	-5.0 %	-2.2 %
time step	0.0050	0.0100	0.0025	0.0050
block time	2.0000	2.0000	0.5000	2.0000
number of processors	16	16	16	16
number of blocks	20	20	20	20
number of walkers	500	500	500	600
quantum force cutoff	200.0	200.0	200.0	200.0

	N	O	F	Ne
calcualted energy	-54.5811	-75.0569	-99.7480	-128.9330
standard deviation	0.0046	0.0085	0.0093	0.0125
amount of correlation	95.7 %	96.0 %	104.3 %	98.8 %
standard deviation	-2.4 %	-3.3 %	-2.9 %	-3.2 %
time step	0.0050	0.0025	0.0050	0.0001
block time	2.0000	1.0000	1.0000	0.4000
number of processors	16	16	16	16
number of blocks	20	20	20	20
number of walkers	750	1000	1000	1000
quantum force cutoff	200.0	200.0	200.0	200.0

Table 4: Details for DMC Calculations with STO-6G MO-Vectors

	Be	B	C
calculated energy	-14.6675	-24.6295	-37.8231
standard deviation	0.0028	0.0014	0.0165
amount of correlation	100.1 %	80.4 %	86.0 %
standard deviation	-3.0 %	-1.2 %	-10.6 %
time step	0.0050	0.0050	0.0050
block time	1.0000	1.0000	2.0000
number of processors	16	64	16
number of blocks	20	20	10
number of walkers	500	500	600
quantum force cutoff	200.0	200.0	200.0

Table 5: Details for DMC Calculations with multireference wave function

He (figure 5)						
block time	40	20	10	5	2.5	1.25
time step	0.2	0.1	0.05	0.025	0.0125	0.00625
energy	-2.90683	-2.90451	-2.9039	-2.90369	-2.90359	-2.90376
standard deviation	0.00026	0.00037	0.00044	0.00056	0.00073	0.00117
He (figure 6)						
block time	40	40	40	40	40	40
time step	0.2	0.1	0.05	0.025	0.0125	0.00625
energy	-2.90683	-2.90444	-2.90414	-2.90383	-2.90374	-2.90369
standard deviation	0.00026	0.00023	0.00027	0.00021	0.00024	0.00024
Ne (figure 7) block time	1	1	1	1	1	1
time step	0.005	0.0025	0.00125	0.000625	0.0003125	0.00015625
energy	-128.962	-128.939	-128.928	-128.925	-128.921	-128.922
standard deviation	0.00414	0.00388	0.00375	0.00348	0.00366	0.00390
Ar (figure 8)						
block time	0.1	0.1	0.1	0.1	0.1	
time step	0.001	0.0005	0.00025	0.000125	0.0000625	
energy	-527.679	-527.541	-527.49	-527.477	-527.475	
standard deviation	0.00665	0.00560	0.00533	0.00515	0.00504	

	He	He	Ne	Ar
exact energy	-2.90372	-2.90372	-128.9376	-527.540
number of processors	16	16	32	512
number of blocks	20	20	10	20
number of walkers	1000	1000	1000	700
quantum force cutoff	200.0	200.0	200.0	200.0

Table 6: Details for DMC Calculations to investigate the time step bias

	ethylene	nitrogen	nitrogen (incl. 3d-functions)
time step	0.0001	0.0001	0.0001
block time	0.02	0.04	0.02
number of processors	512	512	512
number of blocks	20	10	10
number of walkers	599	650	700
quantum force cutoff	-	200.0	200.0

Table 7: Details for DMC Calculations of small molecules

Improving Communication in a Force Decomposition Algorithm

Alexander Weuster

Universität Duisburg-Essen
Fachbereich Physik
E-mail: alexander@weuster.eu

Abstract: This report introduces a method of communication in a parallel molecular dynamic simulation for short range interactions. In order to minimize the volume that has to be gathered on every processor, a not particle but cell-based communication list is created in the neighbor list routine. The algorithm is added to a force decomposition algorithm, which uses space filling curves to sort particles and achieve a additional domain decomposition. Coding and decoding algorithms of Hilbert's and Lebesgue's curve are discussed and results are presented.

Introduction

Computer Simulations have become an essential tool in studying multi-body problems, since even for the Newtonian laws of classical Mechanics, it is not possible to calculate the motion of more than two particles analytically. Although statistical mechanics provide powerful methods to describe huge systems with thermodynamic variables, computer simulation allows to calculate the exact trajectories of N particles within numerical precision. However, calculating particle trajectories, especially for long range interaction like the Coulomb potential is considered a $\mathcal{O}(N^2)$ problem and different methods exist to reduce this effort.
The usage of parallel computers enable simulating molecular systems of even larger and larger size. But with an increasing amount of processes working on one simulation, new problems arise to achieve an equal work load and avoid much global communication, or time spend in, e.g. `mpi_barrier` calls, respectively. For further improvement of the existing methods, especially to minimize the data volume, that has to be transfered, a new communication method is introduced. After a short introduction to the theoretical fundamentals, different parallizing strategies are explained. The focus is set on the composition of two strategies by using space filling curves for sorting particles. Therefore, coding and decoding algorithms are presented and advantages and disadvantages are discussed.

Theoretical Fundamentals

The basic task of Molecular Dynamics is to calculate the development in time of an N particle system, e.g. atoms or molecules. The system is completely characterized by it's Hamiltonian $\mathcal{H} = \mathcal{H}_0 + \mathcal{H}_1$, with the internal part:

$$\mathcal{H}_0 = \sum_{i=1}^{N} \frac{\vec{p}_i^2}{2m_i} + \sum_{i<j}^{N} u(\vec{r}_i, \vec{r}_j) + \sum_{i<j}^{N} u^{(3)}(\vec{r}_i, \vec{r}_j, \vec{r}_k) + \dots \tag{1}$$

consisting of a kinetic and potential term. Here only the pair- (u) and three-body$(u^{(3)})$ interaction potential are explicitly mentioned. External forces are described by $\mathcal{H}_1$ and therefore the systems dynamics are given by the equations of motion:

$$\dot{p}_i = -\frac{\partial \mathcal{H}}{\partial q_i} \qquad \dot{q}_i = \frac{\partial \mathcal{H}}{\partial p_i} \tag{2}$$

The interaction potential of $\mathcal{H}_0$ is further classified in a long range and a short range part. If a potential drops down to zero faster than r^{-d} (with r being the distance between interaction partners and d the dimension of space), it is considered a short range, otherwise a long range interaction.
Different methods like the Fast Multipol Method or the Ewald Summation can be used to reduce calculation time for long range potential, e.g. the Coulomb potential. However it is a common method to distinguish between the far and near field of a particle in the simulation. The Algorithm presented in this report is created for short range interaction or the near field. A good example for short range interactions is the Leonard-Jones potential, with a repulsive part r^{-12} and attractive part r^{-6}. To simulate this kind of potentials, a cutoff radius r_c is introduced, so that only particles within a sphere with radius r_c interact. To avoid the potential divergence at small particle separations in random configuration, it is modified in the following way:

$$u_{ij}(r_{ij}) = \begin{cases} 48\epsilon \frac{1}{\pi} \cos\left(\frac{\pi r_{ij}}{2\sigma_{ij}}\right) & r_{ij} < \sigma_{ij} \\ 4\epsilon_{ij}\left(\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right) & r_{ij} \geq \sigma_{ij} \\ 0 & r_{ij} > r_c \end{cases} \tag{3}$$

Parallel Molecular Dynamics

Methods for decomposition

There are several ways to parallelize a molecular dynamic simulation. To give an overview,three different ways are discussed in the following. A focus is set on the force decomposition method, which is implemented in the current algorithm.
As a intuitive method to share the work between P processors, the N particles are distributed homogeneously at the beginning of the simulation. If a particle is assigned a fixed processor for the entire simulation, the method is called *particle decomposition (PD)*. In order to keep the communication low, it is favorable to distribute particles according to their spatial arrangement. This works well for solids and very viscous liquids. However, if the particles mix up during the simulation, communication time increases.
In contrast to PD, one can assign different geometrical domains to a processor in the beginning of a simulation and therefore achieve a *domain decomposition (DD)*. Each processor calculates forces on particles, presently in its domain. Especially for short range interactions, this methods keeps the communication low, since interacting particles are managed by the same processor. Problems arise because a naively implemented DD method does not consider the particle configuration for decomposition. For inhomogeneous systems a strong load-imbalance occurs.
A more sophisticated way to share the work between P processors is the *Force Decomposition (FD)* Method. It is based on the connectivity matrix $C \in \mathbb{N}^{N \times N}$ containing information of the system, which particle pairs (i, j) interact. Because of Newton's third law, the matrix is symmetric and it is also traceless, as no self interaction is taken into account. A few methods can be used to distribute the force calculation and therefore C among the processors, only the stripped row method, which is used in the current algorithm, will be introduced here. Considering the upper triangular matrix, each processors is assigned a fixed number of rows L_k, with $k \in [1, P]$ and the constraint $\sum_{k=1}^{P} = N$. Subsequent rows are combined into blocks A_k.

$$A(L_k) = \left(N - \sum_{j=1}^{k-1} L_j - \frac{L_k + 1}{2} \right) L_k \quad (4)$$

To obtain a equal workload, all areas $A(L_k)$ should be the equal, which is achieved by solving $A(L_k)P = \frac{N(N-1)}{2}$, leading to

$$L_k = \frac{Q_k}{2} \pm \sqrt{\frac{Q_k^2}{4} - \frac{N(N-1)}{P}} \quad (5)$$

with $Q_k = 2N - 1 - 2\sum_{j=1}^{k-1} L_j$.

Neighbor list techniques

To avoid a $\mathcal{O}(N^2)$ computational effort each force routine to check weather particles interact, it is a common method to create neighbor lists, i.e. store interacting particles. This neighbor list has to be updated only every 20 time-steps, depending on the speed of motion. In the current implementation, a combination of two neighbor list techniques is used, which I would like to introduce in the following. For further study, see [3]

Verlet neighbor list

Each particle is surrounded by a sphere with the radius r_c, equal to the potential cutoff radius. Every particle within the sphere of a arbitrary particle has to be considered for interaction. The basic idea is to store each interaction partner for every particle in a list, the *Verlet neighbor list*.
In order not to update this list every time-step, the cutoff-sphere is surrounded by a " skin", resulting in a larger sphere of radius r_l. Hence potentially interacting particles, which might move into the cutoff sphere in following time-steps, are stored in the neighbor list too. The interval between list updates is often fixed about 10-20 time-steps. However, it is possible to vary this interval by storing the total displacement of each particle. If the sum of magnitudes of the two largest displacements exceeds $r_l - r_c$ a list update needs to be done. Increasing the list-radius r_l will decrease the number of neighbor list updates, but will also slow down the force routine, so a good balance needs to be found. The memory amount for such a list is roughly $4\pi r_l^3 \rho N/6$, so if storage is a priority, another method might be taken into account.

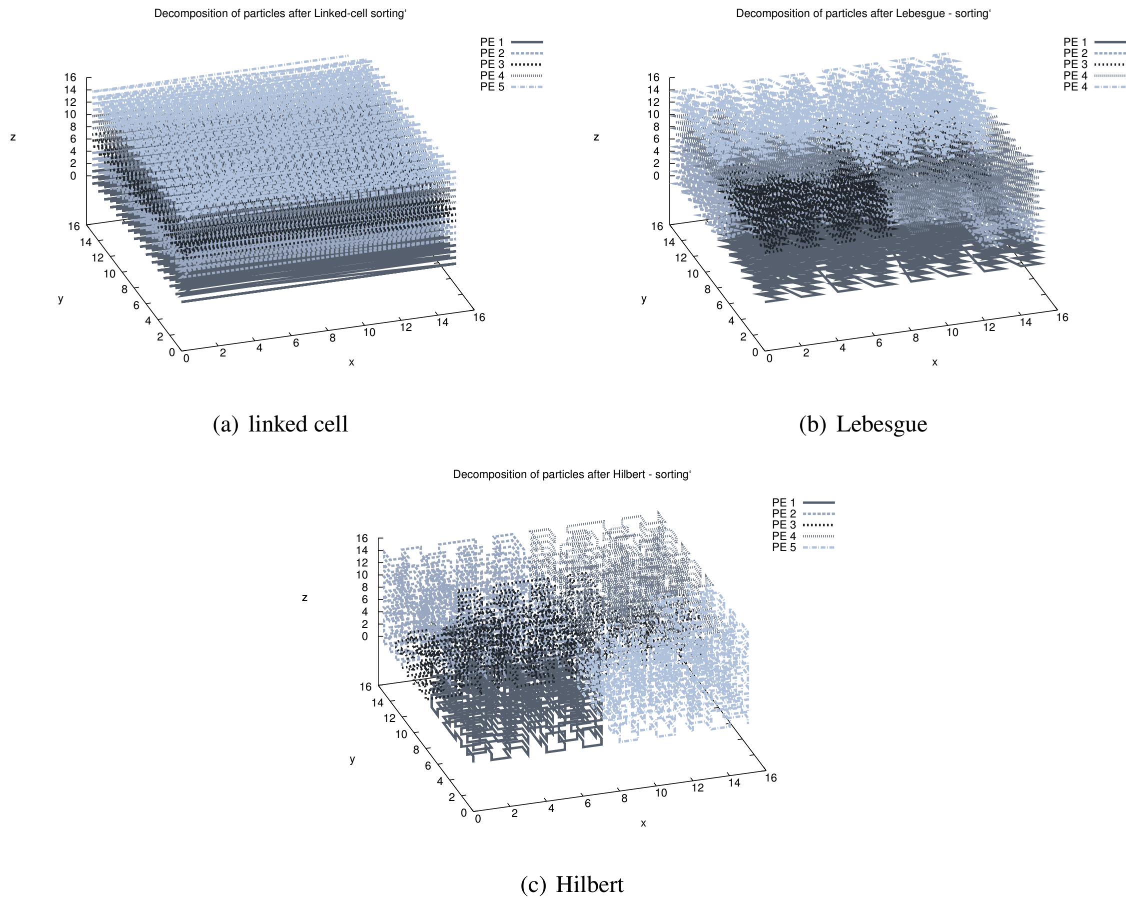
linked cell list

For huge systems the logical testing of every pair in the system is ineffective. Another method to keep track of the neighbors is to make use of a *linked cell list*. Here the simulation box with length B_1, B_2, B_3 in the spatial directions (cubic, non cubic cases also possible), is subdivided into cells, with cell sizes $L_1 \times L_2 \times L_3$ greater than the cutoff sphere. Assigning indices to each cell can be done in several ways, which will be explained in detail later on. For the present, a naive way is just to number the cells successively in x, y and z direction. Two array are created `list_head` with size $M = \frac{B_1}{L_1} \times \frac{B_2}{L_2} \times \frac{B_3}{L_3}$, equals to the total numbers of cells and `list_entry` with size N. Each element j of `list_head` points to the index i of a particle located in cell j. The element i of `list_entry` points to the next index of a particle located in this cell and so on, until a zero entry is reached. Once the linked cell list is created, only neighbor cells need to be checked for interaction partners.

Space filling curves

Although mapping one to two and more dimensions, e.g. the interval $[0, 1]$ to the square $[0, 1]^2$ and the existence of a space filling curve were a challenge for mathematics in the late ninetieth and beginning of twentieth century, it has become a new field of interest for computational science. Due to good locality preserving behavior, curves like Lebesgue's and Hilbert's are often used in data structures for mapping multidimensional data to one dimension

The current algorithm (see also [1]) achieves a combination of a force and domain decomposition by sorting particles according to their spatial position along a space filling curve. This works as follows: Particles are sorted into cells and the cell indices are assigned by the passage order of the curve. The cell index of the particle then determines its new index. Calculating the cell coordinate (x_1, x_2, x_3) as an integer value and vise versa plays a major role and was one of the challenges in creating the new communication method. To Illustrate the achieved domain decomposition see Fig. 1



(a) linked cell

(b) Lebesgue

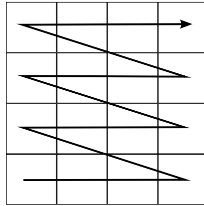
(c) Hilbert

Figure 1: Illustration of the achieved domain decomposition by different sorting methods for 5 Processors

For further insight, the different sorting methods,space filling curves, which are used in the algorithm, and coding and decoding algorithms are discussed. In every case, the simulation box is of length B_1, B_2, B_3, divided into cells of size $L_1 \times L_2 \times L_3$ and hence the coordinate of a cell is given by $\{(x_1, x_2, x_3)|x_i \in \mathbb{N}0 \leq x_i \leq M_i\}$ with $M_i = \frac{B_i}{L_i}$, the number of cells in i-direction.

linked cell sorting

The most naive way of assigning a cell index is the one already explained in the last chapter. Cells are numbered consecutively in x, y and z direction. Hence coding and decoding are very simple operations:(see Fig.2)



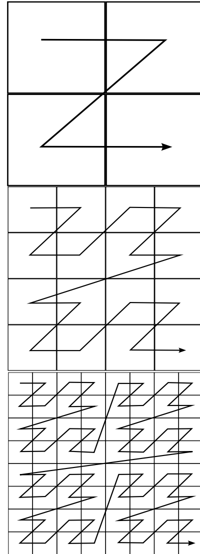
```
cell_index= 1 + x_1 + x_2*M_1 + x_3*M_1*M2

x_1 =  modulo(cell_index-1,M_1)
x_2 =  modulo((cell_index-1)/M_1,M_2)
x_3 =  (cell_index-1)/(M_1*M_2)
```

Figure 2: construction of a linked cell key and a picture of 2d passage order

Although this kind of sorting is not considered a space filling curve in the mathematical sense, is is a pretty effective way to assign cell indices without much computational effort.

Lebesgue's Curve

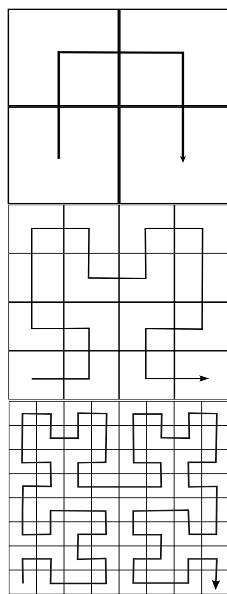


Lebesgue's Curve can be used as a more sophisticated way for numbering the cells. It is based on a 2^n subdivision of the simulation box in each spatial dimension, with $n \in \mathbb{N}$ being the order of the curve. To construct such a curve, you iteratively divide each dimension of the box in 2^n parts with n being the iteration number and assign the indices by a fixed passage order, reminding of a Z. For two dimensions the first three iteration steps are shown on the left. In common $M_1, M_2, M_3 \neq 2^n$ and since M_i is need to kept fixed because of the cutoff radius, a $(2^n)^3$ grid with $2^n \geq \max(M_i)$ and the <u>same</u> cell size is overlaid. The extra cells outside the actual simulation box are not taken into account and act as ghost cells. Therefore not all Lebesgue-indices are actually part of the simulation box and not all consecutive cells are connected in space. Constructing the *Lebesgue key* takes a little more computional effort than the *linked cell key*. It works as follows: The binary representation of the cell coordinate (x_1, x_2, x_3) is used. Beginning from the last bit (2^0), the bits of x_1, x_2, x_3 are transmitted iteratively to three bits of `key` with a well defined offset, also beginning from the last($2^2, 2^1, 2^0$). The offsets depend on the current level i.e. iteration step: $i : 1 * 8^n,\ j : 2 * 8^n,\ k : 4 * 8^n$.

$$\text{key} = \sum_{k=0}^{n-1} 8^k (4\,\text{bit}(x_3, k) + 2\,\text{bit}(x_2, k) + \text{bit}(x_1, k)) \tag{6}$$

For decoding, the procedure is reversed: Transmitting the bits $2^0, 2^3, 2^6 ...$ of `key` to x_1, $2^1, 2^4, 2^7 ...$ to x_2 and $2^2, 2^5, 2^8 ...$ to x_3. If `key` is stored as an integer variable, the order of the Lebesgue curve in three dimensions is limited to 10 or 21 respectively, depending on the bit size of integer (32 or 64 Bit), because every order takes d Bits ($d =$ dimension of space). The Fortran implementation of the Algorithm for coding and decoding are presented in Appendix.

Hilbert's curve



Not only being one of the first mathematics who discovered a space filling curve, David Hilbert was the first who recognized a general geometrical generation procedure that allowed the construction of an entire class of space filling curves. Like Lebesgue's curve, Hilbert's curve is also based on a $(2^n)^d$ grid. The conspicuous property is, that only spatial neighbor cells are connected and therefore it promises a even better locality preserving behavior. However, it has the biggest computational effort in decoding and coding. The illustration of the geometrical generation procedure is shown on the left. In the beginning (first order), the box is divided into 2^d cells, with d being the dimension of space. The passage order reminds of the letter u. In next order, each of those cells is divided into 2^d sub-cells, resulting in a $(2^2)^d$ division of the whole box. In contrast to Lebesgue's curve the passage order varies for each sub-cells resulting in cell coherent curve.
The algorithm I used for coding is introduced in [2]. It is based on the *Lebesgue key* and changes iteratively the passage order of sub-cells. It uses two tables to determine the passage direction `DirTalble` and *Hilbert key* `HilTable` of a certain cell. For two dimensions, four passage direction are classified in `DirTalble` (eight for three dimensions), which assign each sub-cell a new order. Caution must be taken in determing the level of *Hilbert key*, since in [2] it is done dynamically, which is unsuitable for this sorting problem. To avoid defects in passage order, a static level n is defined in the beginning, taken from the $(2^n)^3$ division, with $2^n \geq \max(M_i)$. As mentioned in the calculation of the *Lebesgue key* a $(2^n)^3$ grid is overlaid and the cells outside the simulation box are not taken into account.
The encoding Algorithm was taken from [4] with the improvements suggested in [5]. The computional effort of encoding is quit high and it is left to investigate, if the achieved locality preserving behavior will be worth it. For further interest, the Fortran implementation is given in Appendix B, using three extra functions.

Improvement

Current Algorithm

To get an idea of how the current Algorithm, especially the neighbor routine, might be improved, I would like to describe it a little more detailed.
As mentioned earlier, it combines a force and domain decomposition by sorting the particles according to their spatial position. This ensures, that most interaction particles are stored on the same processor. The connectivity and force matrix becomes dominant around the diagonal and off-diagonal areas are sparse. Sorting is done serial in the present implementation. After sorting, the neighbor list routine is called. Here the force decomposition is done by assigning each processor a fixed area of the force matrix and hence a first and last particle index: `first_i` and `last_i`, representing the first row and last row in the force matrix managed by `local_id`. Here `local_id` $\in \{0, P-1\}$ with P being the number of processors. Since rows are combined to blocks, each index i with `first_i` $\leq i \leq$ `last_i` is also managed by `local_id`. All particle coordinates and velocities are updated on every processor using an `mpi_allgatherv` command. Following a check weather a resort needs to be done (now arbitrarily done every 250 time steps), a verlet neighbor list is filled by taking advantage of the linked cell method, i.e. create *linked cell lists* and only check neighbor cells for interacting particles. Information about every interacting partner from remote Processors is stored and to optimize communication, only those particles are send to `rank` $<$ `local_id` (remember that because of Newton's third law only the upper triangle of the force matrix is considered). In the next step, forces are calculated on each Processor and zero entries of the force matrix are saved, so no redundant data is send in the following. After calculation, the forces

are send back to the processors, particle coordinates originated from.
For load balancing, the calculation time in force-routine is measured and `first_i` and `last_i` are moved up, or down respectively to achieve equal workload (See also [1])

Improvement Idea

To minimize the data volume, presently gathered in neighbor routine the basic idea is to create a not particle but cell based communication list, i.e. every processor determines its locally managed cells by converting the particle indices `first_i` and `last_i` to cell indices (detecting the cells, particles are located in), now stored in `first_cell` and `last_cell`. All cells i with `first_cell` $\leq i \leq$ `last_cell` are considered as local cells for the following neighbor list update, i.e. local particles are located in this cells, which is true for a sorted system and therefore the beginning of the simulation.
To keep track of particles that move out of those cells in the next integration steps a *linked cell list* of local particles is created in each neighbor list update. The lowest and highest cell index found in this process is considered as `first_cell` and `last_cell`,respectively. Hence, only the `_cell` arrays need to be gathered on all processors, which are two arrays with size of integer times N. With this information each processor calculates the neighbor cells of local managed cells and stores them, together with the processor id in a send and receive list for later communication. In the communication step later on, whole cells, i.e. all particles located in this cell are transfered.

Algorithm

To go into detail, the procedure-steps of the new neighbor routine are presented here. The first step is to sort all local managed particles into a *linked cell list*, the indices are calculated according to the used sorting method. Since not all cell indices for Lebesgue and Hilbert sorting are referring to a cell of the simulation box, the `list_head` array has to be allocated with a size $(2^n)^3$, again with $2^n \geq \max M_i$. The lowest and highest determined cell index in stored in `first_cell` and `last_cell`. For a sorted system, this is equal to the cell indices of the cells where `first_i` and `last_i` are located in. Every cell i with `first_cell` $\leq i \leq$ `last_cell` is treated as a local cell.
In the next step, this two arrays with length P (equal to the number of processors) are communicated to every processors using an `mpi_allgather` command, i.e. only two integers for each processor have to be send with a global communication call, instead of $N\times$ three real, or double variable for coordinates and one integer as index. After communication, another array `cell_list` is created, which stores information about which processors manages which cells. This again has to be allocated with size *number of cells* $\times$ *number of processors*, since in worst case, every processors manages every cell. It is done by every participating processor, at present for all cells.
Now the actual cell based communication lists for processor p_i are created by looping all local cells and determine their neighbor cells. If a neighbor cell is not local, or managed by more than one processor, it is stored in the communication list `recv_list` which has *number of processors* $\times$ *number of cells* entries. Additionally, the present cell is stored in a similar array `send_list`, to store which cells have to be send to which PE. To proceed, this two lists can be used in a *linked cell* force calculation, i.e. the whole cells are communicated in the beginning of every force routine, each cell is check for interacting particles and forces are calculated, if necessary. It might be to much computational effort, to check all particles in neighbor cells for interaction partners, so for the new implementation, in addition to `send` and `recv_list`, a verlet neighbor list list created in the neighbor routine, which stores potentially interacting particles in advance,i.e. particles within a distance r_l(list radius of verlet routine). Although this takes more computational time in the neighbor routine it remarkably speeds up the force routine.

Results

Communication statistics for different sorting methods are made. To compare these methods, the number of communication calls are represented in Fig. 3, for an example of $P = 64$, $N = 100000$. A random particle configuration is chosen and there are about 20 particles within one cutoff sphere, resulting into total cell number of 1331 (11 in each dimension). It shows quiet well, that for linked cell sorting the number of send and receive call per processor is minimal, compared to Hilbert and Lebesgue sorting. This is caused by the layer structure of the simulation box (see Fig. 1), created with this sorting. In best case, processors only communicate with their two neighbors (above and below layers).
In Fig. 4 the data-volume (number of particles to be transfered) of point to point communication for each processor is shown. Although linked cell sorting achieved the lowest number of communication calls per processor, the number of particles that need to be send in each call is higher than for Hilbert and Lebesgue sorting.

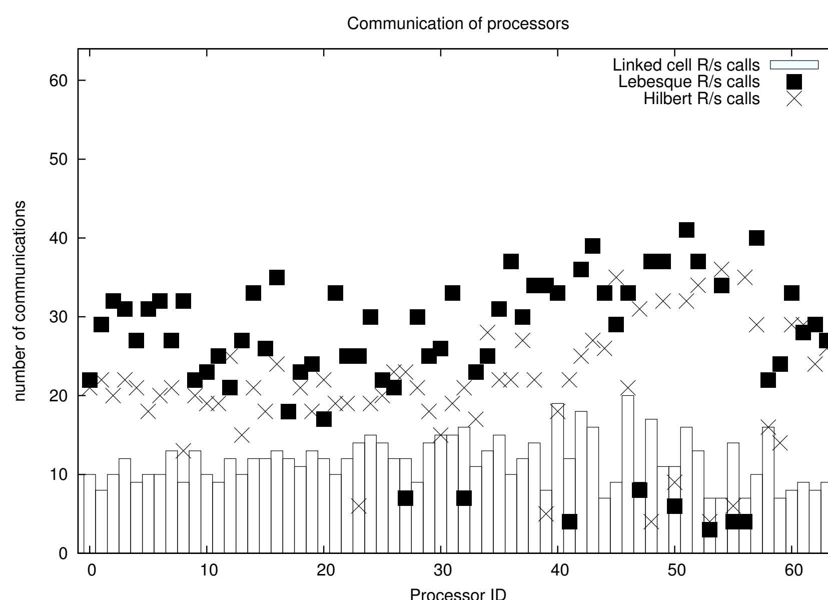
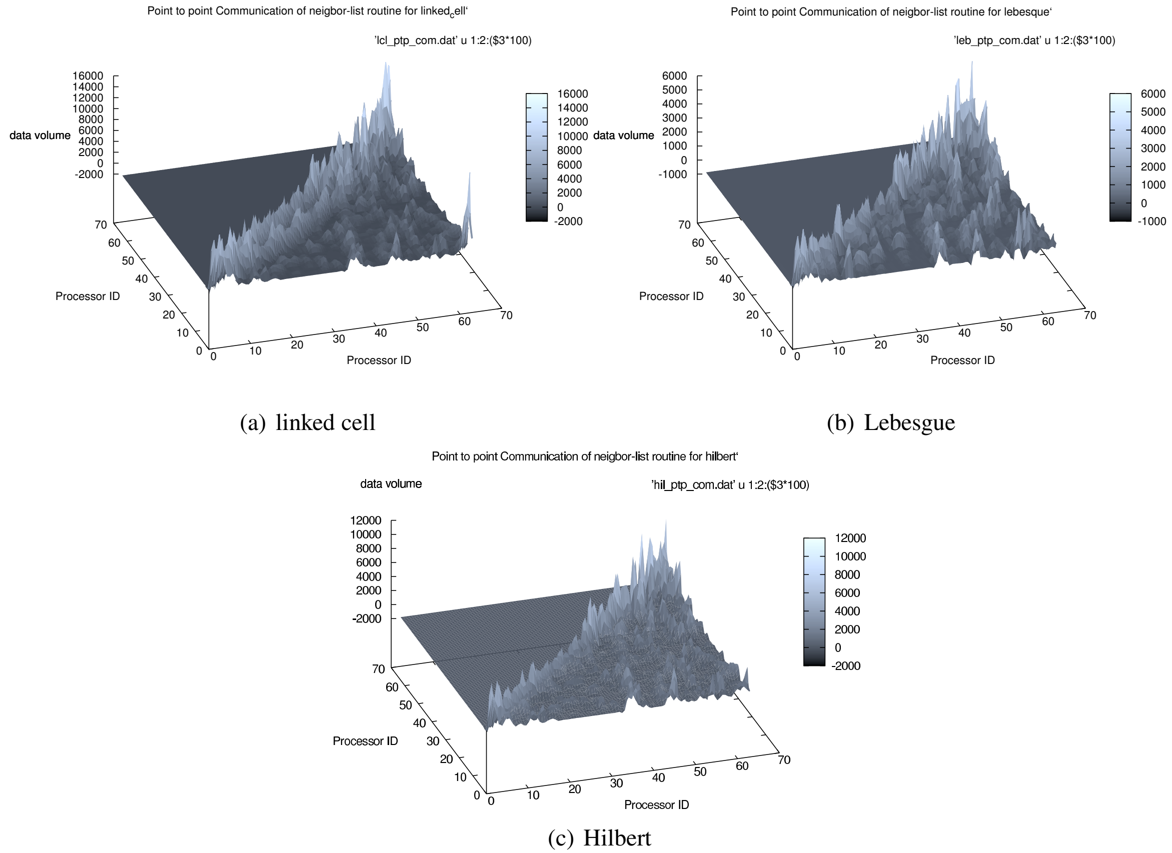


Figure 3: Number of communication calls for each processor

Conclusion and outlook

A new way of communication in neighbor routine of a molecular dynamic simulation is introduced. It avoids a `mpi_allgatherv` of coordinates by creating a cell based communication list and hence reduces communication calls in the neighbor routine. In order to avoid a linked cell force routine, which would need more computational time, a additional verlet neighbor list is created in the neighbor routine to speed up the force calculation. Compared with the old implemented routines, the time spend in `mpi` calls has decreased, but the calculation time in neighbor routine has increased, which is a result of coding and decoding algorithms and the additional linked cell force routine to built up the verlet neighbor list.
It works quiet well for sorted systems, however if particles mix up, communication increases until all processors communicate with each other. This is one of the further improvements to be considered: For systems with periodic boundary condition, particles from processor p might move from first cell, with the lowest index, to a cell with a high index and all cells in between are considered as local cells, although most cells in between are not occupied by local managed particles. This leads to unnecessary communication, since all other Processors send their cells to p. To obviate this communication, ghost cells at the edge of the box might be used to store particles that have jumped to the other side of the



(a) linked cell

(b) Lebesgue

(c) Hilbert

Figure 4: Illustrations of point to point communication for different sorting algorithms

simulation box and consider this to determine `first` and `last_cell`.
To improve scaling and make use of the advantaged one might implement a parallel sorting algorithm. This would enable to perform a sorting more often, keep the communication low and profit of a good domain decomposition respectively. Especially for very large systems, which might not be handled by a verlet neighbor routine, the presented algorithms promises a good way to improve communication, by reducing the time spend in `mpi` calls and use computation time more efficiently.

Algorithm for decoding and coding Lebesgue key

```
function leb2coord(key) result(coord)

  integer (kind = 8)                  ::key
  integer (kind = 8)                  ::k_ix, k_iy, k_iz, key_tmp
  integer                             ::dim,level,j
  integer ,dimension(3)               ::coord

  k_ix = 1_8                 ! = ...0 001
  k_iy = 2_8                 ! = ...0 010
  k_iz = 4_8                 ! = ...0 100

  dim = 3
  key_tmp = key
  level = 0

  !deduce level - dynamic leveling
  do
     key_tmp = ishft(key_tmp,-dim)
     level = level + 1
     if( key_tmp < 1_8 ) exit
  enddo

  !calculate cell coordinates
  do j = level,1,-1
     coord(1) = ishft(coord(1),1)+iand(ishft(key,-(j-1)*dim),k_ix)
     coord(2) = ishft(coord(2),1)+ishft(iand(ishft(key,-(j-1)*dim),k_iy),-1)
     coord(3) = ishft(coord(3),1)+ishft(iand(ishft(key,-(j-1)*dim),k_iz),-2)
  enddo


end function leb2coord




function coord2leb(ix,iy,iz) result(key)

  integer (kind = 8)                 ::key_leb
  integer (kind = 8)                 ::placebit
  integer                            ::ix,iy,iz
  integer                            ::jx,jy,jz
  integer                            ::j

  key_leb =0

   placebit=2_8**63

      do j = 0, bit_size(placebit)-1
         if( btest(ix,j) )then
            jx=1
         else
            jx = 0
         endif
         if( btest(iy,j) )then
            jy=1
         else
            jy = 0
         endif
         if(btest(iz,j) )then
            jz=1
         else
            jz = 0
         endif

         key_leb = key_leb + 8**j * (4*jz + 2*jy + jx)
      enddo

end function coord2leb
```

Algorithm for decoding and coding Hilbert key

```
integer,parameter          :: dim=3
integer,parameter          :: n_level

function calc_J(P) result (J)

  integer                    :: i, J, P

  J = dim
  do i =1,dim-1
     if(iand(ishft(P,-i),1)==iand(P,1)) then
        cycle
     else
        exit
     endif
  enddo

  if(i/=dim)J=J-i

end function calc_J

!---------------------------------------------------

function calc_T(P) result(PE)

  integer                      ::P,PE


  if(P<3) then
     PE=0
     return
  endif

  if(modulo(P,2)==1)then
     PE= ieor(P-1,(P-1)/2)

  else
     PE= ieor(P-2,(P-2)/2)

  endif

end function calc_T

!---------------------------------------------------

function calc_tS_tT(xJ,val) result(retval)

  integer                       :: xJ,val,retval,tmp1,tmp2

  retval = val

  if(modulo(xJ,dim)/=0) then
     tmp1 = ishft(val,-modulo(xJ,dim))
     tmp2 = ishft(val,dim-modulo(xJ,dim))
     retval=ior(tmp1,tmp2)
     retval= iand(retval,ishft(1,dim)-1)
  endif

end function calc_tS_tT



!==================================================================



function hil2coord(hilkey) result(coord)


  integer                      :: i,k,J,xJ=0,T,tT,S,tS, &
                                  W,tT_old,x,      &
                                  key_i
  integer(kind =  8)           :: hilkey
  integer,dimension(dim)       :: coord, coord_dummy

  xJ=0; tT_old=0; coord=0;W=0


  do i = 1, n_level

     key_i= iand(ishft(hilkey,-(n_level-i)*dim),(2_8**dim)-1)
     J=calc_J(key_i)
     T=calc_T(key_i)
     S= ieor(key_i,key_i/2)
     tS= calc_tS_tT(xJ,S)
     tT= calc_tS_tT(xJ,T)
     W = ieor(W,tT_old)
     tT_old=tT
     x=ieor(W,tS)
     xJ=xJ+(J-1)


     !Calculate coordinate
```

```
      do k=1,dim
         coord(k)= ishft(coord(k),1) + iand(ishft(x,-(k-1)),1)!,n_level-i)
      enddo

   enddo


   coord_dummy=coord
   coord(1)=coord_dummy(2)
   coord(2)=coord_dummy(1)
   coord(3)=coord_dummy(3)


 end function hil2coord


 function coord2hil(ix,iy,iz) result(hilkey)


   integer,dimension(3)                   :: ncell
   integer                                :: i,j,cell,dir,level
   integer(kind = integer_8)              :: hilkey,key,key_tmp,k
   integer                                :: ix,iy,iz
   integer                                :: jx,jy,jz
   integer,dimension(0:7,0:11)            :: DirTable,HilTable


   DirTable = reshape((/ 8,10, 3, 3, 4, 5, 4, 5,   2, 2,11, 9, 4, 5, 4, 5,    &
        7, 6, 7, 6, 8,10, 1, 1,   7, 6, 7, 6, 0, 0,11, 9,    &
        0, 8, 1,11, 6, 8, 6,11,  10, 0, 9, 1,10, 7, 9, 7,    &
        10, 4, 9, 4,10, 2, 9, 3,   5, 8, 5,11, 2, 8, 3,11,    &
        4, 9, 0, 0, 7, 9, 2, 2,   1, 1, 8, 5, 3, 3, 8, 6,    &
        11, 5, 0, 0,11, 6, 2, 2,   1, 1, 4,10, 3, 3, 7,10 /),(/8,12/))
   HilTable = reshape((/ 0,7,3,4,1,6,2,5,  4,3,7,0,5,2,6,1,  6,1,5,2,7,0,4,3, &
        2,5,1,6,3,4,0,7,  0,1,7,6,3,2,4,5,  6,7,1,0,5,4,2,3, &
        2,3,5,4,1,0,6,7,  4,5,3,2,7,6,0,1,  0,3,1,2,7,4,6,5, &
        2,1,3,0,5,6,4,7,  4,7,5,6,3,0,2,1,  6,5,7,4,1,2,0,3 /),(/8,12/))

   .....
   !Calculation of Lebesgue key
   ....

   k = ishft(1,DIM)-1

   hilkey = 0
   dir = 0
   level = bit_size(ix)


   do j = level,1,-1
    cell = iand(ishft(key,-(j-1)*DIM),k)
    hilkey = ishft(hilkey,dim) + HilTable(cell,dir)
    dir = DirTable(cell,dir)
   enddo


   end function coord2hil
```

References

1. G. Sutmann and F. Janoschek Communication and Load Balancing of Force-Decomposition Algorithms for Parrallel Molecular Dynamics
2. M. Griebel, S. Knapek, G. Zumbusch adn A. Caglar. Numerische Simulationen in der Moleküldynamik. Springer, Berlin, 2004.
3. Computer Simulation of Liquids M.P. Allen and D.J. Tildesley Oxford Science Publications, Oxford, 1987
4. A.R. Butz Alternativ Algorithm for Hilbert's Space filling curve IEEE Transactions on Computers, 20:424-426, April 1971
5. J.K. Lawder Calculations of Mappings Between One and n-dimensional Values Using the Hilbert Space filling Curve Research Report BBKCS-00-01, August 2000