MICROSOFT .NET: EINE NEUE ENTWICKLUNGSPLATTFORM – AUCH FÜR NUMERISCHE PROBLEME?

Jan Linxweiler, Sören Freudiger

Institut für Computeranwendungen im Bauingenieurwesen,
Technische Universität Braunschweig

Kurzfassung: *Mit dem .NET Framework stellt Microsoft seit knapp drei Jahren eine neue Plattform für die objektorientierte Entwicklung und Ausführung von Anwendungen zur Verfügung. Hierfür hält das Framework einige Neuerungen bereit, die die Anwendungsentwicklung komfortabler gestalten und die Ausführung des Codes u.a. sicherer machen. Am Beispiel des in der Programmiersprache C# implementierten interaktiven Strömungssimulators - FlowSim 2004 .NET - soll ermittelt werden, ob der Einsatz des .NET Frameworks für das wissenschaftliche Rechnen sinnvoll ist bzw. wie sich die Ausführungsgeschwindigkeit von .NET-Anwendungen im Vergleich zu "herkömmlichen" Anwendungen verhält.*

Diese Arbeit soll einen Einblick in die .NET-Technologie mit besonderem Focus auf performancerelevante Eigenschaften geben.

1 Einleitung

In der Forschung werden zur numerischen Simulation i. A. sequenzielle Programmcodes verwendet, die z.B. unter Verwendung der Programmiersprache *Fortran* oder *C* implementiert werden. Diese Programmiersprachen gewährleisten eine hohe Ausführungsgeschwindigkeit, die für rechenintensive numerische Simulation von besonderem Interesse ist. Zusätzlich bieten z. B. sequenzielle *Fortran*- bzw. *C*-Programme die Möglichkeit, diese zu parallelisieren und somit auf Multiprozessorsystemen auszuführen.

Seit einigen Jahren hat mit *C++, Java* und *C#* die objektorientierte Programmierung in der Informatik Einzug gehalten. *C++* stellt eine Erweiterung der Programmiersprache *C* dar. Mit dieser wird das objektorientierte Programmieren unter Verwendung der aus *C* und *Fortran* bekannten Zeiger ermöglicht. Mit *C++* erstellte Anwendungen erzielen somit eine ähnlich hohe Ausführungsgeschwindigkeit wie sequenzieller Code auf Basis von *C* oder *Fortran*.

In den jüngsten Sprachen *Java* und *C#* wurde das Prinzip der Zeiger aufgegeben. Zudem wurde die Speicherverwaltung, die bislang Aufgabe des Programmierers war, automatisiert. Im Unterschied zu Anwendungen, die in *C++* oder in einer der sequenziellen Sprachen implementiert sind, werden *Java-* und *C#* - Anwendungen nicht mehr zur Entwurfszeit in Maschinensprache übersetzt. Dieser Prozeß wird erst mit der Ausführung des Programms durchgeführt.

Die Programmierung unter Verwendung der modernen objektorientierten Sprachen bietet dem Entwickler einen großen Komfort im Vergleich zu früheren Programmierweisen. Allerdings führt dieser Zugewinn an Komfort teilweise zu Einbußen in der Ausführungsgeschwindigkeit.

Am Beispiel der Entwicklung eines interaktiven numerischen Strömungssimulators unter Verwendung der Programmiersprache *C#* und des *.NET Frameworks* soll untersucht werden, in wie weit sich der Komfort der Anwendungsentwicklung moderner Softwaretechnologien mit der Notwendigkeit hoher Ausführungsgeschwindigkeiten vereinbaren läßt.

2 Microsoft .NET Framework

Neben den *.NET Web Services* und den *.NET Enterprise Servern* ist das *.NET Framework* ein Teil der sogenannten *.NET Strategie*. Das *.NET Framework* enthält im Wesentlichen verschiedene Komponenten und Richtlinien, mit denen Anwendungen erstellt, kompiliert und ausgeführt werden können. Zu diesen zählen unter anderem die *Common Language Runtime* (Laufzeitumgebung) und die *Framework Class Library* (Klassenbibliothek) sowie eine Reihe Spezifikationen wie *das Common Type System* (Objektmodell) und die *Intermediate Language* (Zwischensprache) [7].

2.1 Intermediate Language (IL)

.NET-Sprachcompiler erzeugen anders als herkömmliche Compiler keinen Maschienencode sondern Code in einer maschinennahen Zwischensprache, der Intermediate Language. Diese wird zur Laufzeit vom *JIT-Compiler (Just In Time)* in plattformspezifischen und –optimierten Maschinencode überführt [7]. In Hinblick auf die Ausführungsgeschwindigkeit ist an dieser Stelle zu erwähnen, daß der *JIT-Compiler* die zu Grunde liegende Rechnerarchitektur des jeweiligen Systems berücksichtigt. Das bedeutet, daß der *JIT-Compiler* Maschinencode generiert, der z.B. auf die vorhandene *CPU* zugeschnitten ist [6].

2.2 Common Type System (CTS)

Als Spezifikation, der alle .NET-Sprachen unterworfen sind, definiert das *Common Type System* sämtliche verfügbaren Basistypen und Vorschriften zur Erstellung eigener Typen an zentraler Stelle. Das *CTS* stellt somit ein gemeinsames Typsystem für die .NET-

Sprachen auf Ebene der *IL* dar. Es ist allerdings nicht bindend, daß jede Sprache das vollständige Typsystem des *CTS* verwendet. Zwingend unterstützen alle .NET-Sprachen jedoch eine Untermenge des *CTS,* die als *Common Language Specification (CLS)* bezeichnet wird. Das *CTS* bildet somit zusammen mit der *CLS* die Basis für eine Sprachunabhängigkeit. Durch das gemeinsame Typesystem und die ebenfalls gemeinsame Zwischensprache kann somit eine Integration auf Codeebene stattfinden. Das bedeutet, daß einzelne Programmteile in verschiedenen Programmiersprachen entwickelt und anschließend in einer Anwendung zusammengeführt werden können.

Oberstes Ziel hinter dem Konzept des *CTS* ist die Typsicherheit. Nach dem Vorbild von *Java* sind potentiell unsichere Operationen wie Zeiger, Arrayzugriffe oder Rückruffunktionen als objektorientiertes Konzept neu gefaßt und Teil der Spezifikation des *CTS*. Die Typüberprüfung findet zudem nicht nur zur Kompilierungs- sondern auch zur Ausführungszeit statt. Aus der Typsicherheit resultiert zum großen Teil die Ausführungssicherheit unter *.NET* [7].

2.3 Common Language Runtime (CLR)

Die *CLR* ist die Laufzeitumgebung des *.NET Frameworks* und stellt gewissermaßen den Kern des Frameworks dar. Zu dem Aufgabenbereich der *Common Language Runtime* gehören Verwaltungsaufgaben, die im Zusammenhang mit der Ausführung von .NET-Anwendungen stehen. So sind sowohl der oben genannte *JIT-Compiler* als auch der *Type Checker* Komponenten der *CLR*. Eine weitere Aufgabe der Runtime ist die automatisierte Speicherverwaltung, die von dem so genannten *Garbage Collector* übernommen wird. Dieser im Hintergrund laufende Prozeß entfernt nicht mehr benötigte Objekte vom Heap und organisiert die Belegung des Speichers durch Umschichtung. Die aus den Zeiten von *C* und *C++* bekannten „Speicherlecks“ können daher nicht mehr entstehen. Zusätzlich wird durch die Umschichtung eine hohe Lokalität häufig verwendeter Daten erzielt, die schnelle Zugriffszeiten zur Folge hat [7].

2.4 Unmanaged Code versus Unsafe Code

Die wahrgenommenen Verwaltungsaufgaben der Laufzeitumgebung sind Grund dafür, daß der von der Runtime ausgeführte *IL-Code* im Allgemeinen auch als *Managed Code* bzw. verwalteter Code bezeichnet wird. Im Gegensatz dazu werden Anwendungen, die nicht unter der Aufsicht der *CLR* ausgeführt werden als *Unmanaged* oder unverwaltet auch bezeichnet. Ein weiterer Ausdruck aus dem .NET-Sprachgebrauch ist *Unsafe Code*. Dieser ist nicht zu verwechseln mit dem Begriff *Unmanaged*. Obwohl man sich in *.NET* von dem Prinzip der Zeiger verabschiedet hat, stehen sie jedenfalls in *C#* weiterhin zur Verfügung [1]. Allerdings werden Programmteile in denen Zeiger verwendet werden von der Laufzeitumgebung prinzipiell als „unsicher“ eingestuft, da derartige Codeabschnitte die Typsicherheit umgehen. Unsicherer Code ist aber dennoch verwalteter bzw. *Managed Code*, der unter der Aufsicht der *CLR* ausgeführt wird [8].

2.5 Interoperabilität

Unter Verwendung der Programmiersprache *C++ Managed Extensions* ist es möglich, verwaltete Anwendungen zu erstellen, die unverwaltete Anteile enthalten. Dies betrifft sowohl die Daten als auch den Programmcode. Auf diese Weise ist es auch weiterhin möglich, bestehende Quelltexte bzw. Bibliotheken in verwalteten Projekten zu verwenden. Besondere Aufmerksamkeit gilt den Schnittstellen zwischen verwalteten und unverwalteten Programmteilen, da es hier zu einem sogenannten *Marshalling* kommt. Das bedeutet es müssen beim Übergang zwischen den verwalteten und unverwalteten Programmteilen Datentypen umgewandelt werden.

2.6 Arrays in .NET

In *.NET* stehen drei verschiedene Arraytypen zur Verfügung: eindimensionale Arrays, sequenzielle Arrays *(rectangled arrays)* und verschachtelte Arrays *(jagged arrays)*. Wie in C beginnen eindimensionale Arrays mit dem Index Null und sind sequenziell im Speicher abgelegt. Für eindimensionale Arrays existiert ein Satz von IL-Instruktionen, die es den Compilern ermöglichen, die Arrayzugriffe zu optimieren. Zudem kann für eindimensionale Arrays das *Range Checking* unterdrückt werden. Verschachtelte Arrays sind mehrdimensionale Arrays, die sich aus mehreren eindimensionalen Arrays zusammensetzen. Nachteilig ist allerdings, daß die einzelnen Arrays unzusammenhängend im Heap abgelegt sind. Diesem Problem begegnen die ebenfalls mehrdimensionalen sequenziellen Arrays, bei denen eine hohe Lokalität der Daten besteht. Von Seiten des Compilers kann die Verwendung von sequenziellen Arrays nicht in dem Maße optimiert werden wie bei verschachtelten Arrays. Auch die Indizierung ist bei sequenziellen Arrays nicht so hoch optimiert [3].

3 Der interaktive Strömungssimulator

Das Programm *FlowSim 2004 .NET* ist ein interaktiver numerischer Strömungssimulator auf Basis der Lattice-Boltzmann-Methode [5]. Die Anwendung bietet die Möglichkeit zur Simulation laminarer, schwach kompressibler Strömungen auf uniformen, zweidimensionalen, kartesischen Gittern. Für die Simulation stehen zwei Berechnungsmodelle zur Verfügung. Die Berechnung kann wahlweise mittels der LBGK-Methode oder mittels des Momentenmodells (MRT) erfolgen. Die Besonderheit des Strömungssimulators besteht darin, daß die Anwendung die Möglichkeit bietet, die laufende Simulation seitens des Anwenders interaktiv zu beeinflussen. Dies kann z.B. über ein Verändern der Geometrie des Strömungsgebietes geschehen.

Die Interaktivität der Simulation macht es notwendig, eine möglichst hohe Ausführungsgeschwindigkeit der performancekritischen Anwendungsteile (Rechenkern) zu erzielen, um ein großes Maß an Benutzerfreundlichkeit zu gewährleisten.

3.1 Die Lattice-Boltzmann-Methode

Zur numerischen Berechnung der Strömungs- und Transportprozesse wird die Lattice-Boltzmann-Methode herangezogen. Der grundlegende Gedanke dieser Methode besteht darin, die makroskopischen Eigenschaften eines Fluides durch einen mikroskopischen kinetischen Ansatz zu lösen, der die Bewegung und Interaktion auf Teilchenebene beschreibt.

Die Basis bildet die Finite-Differenzen-Approximation der Diskreten-Boltzmann-Gleichung [4] mit dem vereinfachen BGK-Kollisionsoperator [1]:

$$f_i(\vec{x}+\vec{e}_i\cdot\Delta t, t+\Delta t) - f_i(\vec{x},t) = \frac{\Delta t}{\tau}[f_i(\vec{x},t) - f_i^{eq})] \tag{1}$$

mit

f	- Verteilungen	*t*	- Zeit
i	- Verteilungsrichtungen	τ	- Relaxationszeit
f^{eq}	- Gleichgewichtsverteilung (z.B. nach Maxwell)	*x*	- Ort

Die linke Seite der Gleichung beschreibt den physikalischen Strömungsprozeß, während die rechte Seite die Annäherung an die Gleichgewichtsverteilung modelliert. Dies bedeutet, daß das BGK-Schema aus zwei numerischen Schritten besteht: dem Propagationsschritt und dem Kollisionsschritt.

Kollision: $$f_i^{new}(\vec{x},t) - f_i(\vec{x},t) = \Omega_i \tag{2}$$

Propagation: $$f_i(\vec{x}+\vec{e}_i\cdot\Delta t, t+\Delta t) = f_i^{new}(\vec{x},t) \tag{3}$$

Der Vorteil der LBM, der für die Anwendung in computergestützter Simulation spricht, ist das einfach gehaltene System an Gleichungen, das dementsprechend leicht zu implementieren ist. Zudem beinhaltet der Propagationsschritt sehr wenig Rechenaufwand und der Kollisionsschritt findet ausschließlich lokal statt.

3.2 Die Berechnungsdaten

Die der Berechnung zu Grunde liegenden Daten werden innerhalb der Anwendung vorwiegend in mehrdimensionalen Arrays gehalten. Die Geometrie des Strömungsgebietes wird auf eine Knotenmatrix abgebildet. Zu jedem Knoten existieren beispielsweise Daten über die Geschwindigkeit, die Dichte und die Teilchenverteilungen.

3.3 Die verschiedenen Berechnungskerne

In der Anwendung stehen mehrere Berechnungskerne zur Auswahl, die eine Simulation sowohl mittels der LBGK-Methode als auch mittels des Momentenmodells (MRT) ermöglichen. Alle Kerne implementieren diese Funktionalität auf dieselbe Weise. Sie un-

terscheiden sich jedoch hinsichtlich bestimmter Details. Es wird unterschieden, ob der Programmcode bzw. die Daten der Verwaltung der Laufzeitumgebung unterliegen *(managed-unmanaged)*. Des Weiteren wird seitens der verwalteten Kerne unterschieden, welche Arraytypen Verwendung finden *(jagged-rectagled)* und welche Zeigerstrukturen verwendet werden *(safe-unsafe).* Die Hierarchie der Berechnungskerne ist in

dargestellt.

Die unterschiedlichen Implementierungen werden in einem Benchmark miteinander verglichen, um abschließend eine Aussage über die Ausführungsgeschwindigkeiten von .NET-Anwendungen zu treffen.

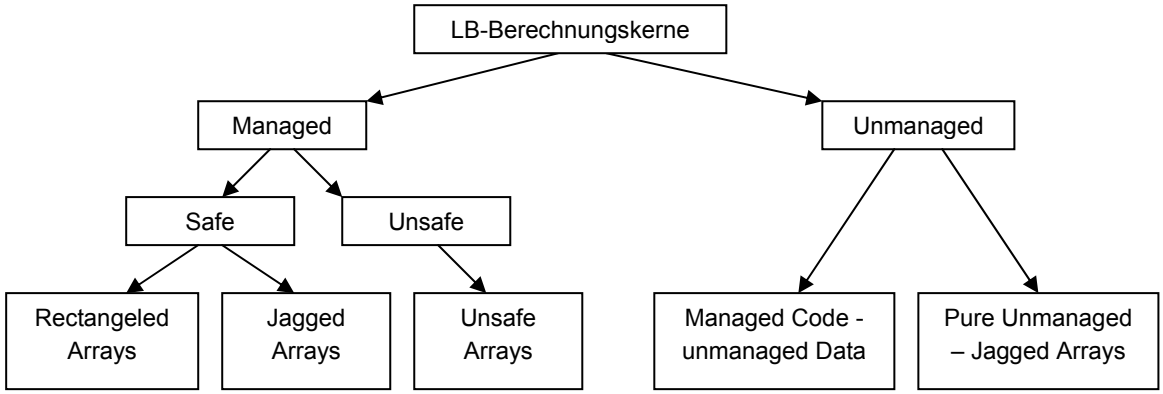

Abb. 1: Hierarchie der Berechnungskerne

4 Benchmark

Um eine Aussage über die Ausführungsgeschwindigkeiten der Berechnungskerne machen zu können, wurde auf mehreren Testrechnern ein Gebiet mit 30.000 Knoten unter Verwendung beider Berechnungsmethoden (LBGK & MRT) gerechnet. Als Bewertungskriterium für die Ausführungsgeschwindigkeit werden die so genannten *nups* (nodal updates per second) herangezogen. Die Ergebnisse der Testläufe sind den Tabellen 1 und 2 zu entnehmen.

Computer	Managed Code rectangled Arrays	managed Code jagged Arrays	managed Code unsafe	managed Code unmanaged Data	unmanaged Code - unmanaged Data
(CPU / Ram)	x1000 [nups]	x1000 [nups]	x1000 [nups]	x1000 [nups]	x1000 [nups]
Mobile P4 1.6 GHz 768 MB	722	1.055	1.168	1.246	**1.668**
s.o.	721	1.050	1.169	1.245	**1.678**
Athlon XP 2400+ 2.0 Ghz 1,5 GB	1.084	1.467	1.616	1.505	**1.766**
P4 1.7 GHz - 1,5 GB DDR	803	1.165	1.286	1.417	**1.955**
Athlon XP 2500+ 1.84 GHz 1,5 GB	914	1.220	1.392	1.276	**1.425**
P4 2.4 GHz 1,0 GB	924	1.100	1.350	1.310	**1.500**
P4 mobile 1.6 GHz 500 MB	680	963	1.077	1.155	**1.530**
Opteron 2.0 Dual 2x 2,75 GB DDR	1.385	2.194	2.227	1.966	**2.699**
P4 2.4 GHz 1,0 GB	929	1.200	1.340	1.315	**1.509**
Athlon XP 1800+ 1.53 GHz 1,5 GB	698	845	980	910	**1.014**

Tab. 1: Benchmark unter Verwendung der LBGK-Methode

Computer	managed Code rectangled Arrays	managed Code jagged Arrays	managed Code unsafe	managed Code unmanaged Data	unmanaged Code unmanaged Data
(CPU / Ram)	x1000 [nups]	x1000 [nups]	x1000 [nups]	x1000 [nups]	x1000 [nups]
Mobile P4 1.6 GHz 768 MB	629	**1.207**	1.005	916	1.159
Athlon XP 2400+ 2.0 Ghz 1,5 GB	967	1.474	1.515	1.389	**1.559**
P4 1.7 GHz 1,5 GB DDR	700	**1.402**	1.145	1.031	1.315
Athlon XP 2500+ 1.84 GHz 1,5 GB	827	1.235	1.375	1.176	**1.325**
P4 2.4 GHz 1,0 GB	825	1.215	1.250	1.061	**1.270**
P4 mobile 1.6 GHz 500 MB	595	**1.159**	960	835	1.065
Opteron 2.0 Dual 2x 2,75 GB	1.267	**2.480**	2.360	2.250	2.340
P4 2.4 GHz 1,0 GB	843	**1.328**	1.240	1.065	1.260
Athlon XP 1800+ 1.53 GHz 1,5 GB	640	850	**935**	840	920

Tab. 2: Benchmark unter Verwendung des Momentenmodells

4.1 Ergebnisse

Aus Tabelle 1 ist ersichtlich, daß der unverwaltete Berechnungskern (unmanaged Code – unmanaged Data) in allen Testläufen für die LBGK-Methode die besten Ergebnisse erzielt. An zweiter Stelle folgt der verwaltete Berechnungskern, der unverwaltete Daten verwendet (managed Code – unmanaged Data). Dieser liegt in etwa gleich auf mit dem verwalteten Kern, der Zeiger verwendet (managed Code - unsafe). Anschließend folgt der verwaltete Kern, der verschachtelte Arrays verwendet und zuletzt derjenige, der auf sequenziellen Arrays arbeitet. Zwischen den letztgenannten Berechnungskernen liegt die Differenz der Ausführungsgeschwindigkeiten bei ca. 25%. Der Unterschied zwischen dem unverwalteten und dem schnellsten verwalteten Kern liegt bei ca. 19% im Mittel.

Tabelle 2 zeigt die Ergebnisse, die für die MRT-Methode erzielt wurden. Im Bereich der mittleren Ausführungsgeschwindigkeiten werden die Ergebnisse aus Tabelle (1) bestätigt. Im oberen Bereich liegt in diesem Fall jedoch der verwaltete Berechnungskern unter Verwendung der jagged Arrays mit dem vollständig unverwalteten Kern in etwa gleich auf. Die Differenz zwischen dem weiterhin langsamsten verwalteten Kern mit sequenziellen Arrays und dem Kern mit verschachtelten Arrays vergrößert sich auf durchschnittlich ca. 40%.

4.2 Diskussion der Ergebnisse

Aus den Ergebnissen wird, wie erwartet, deutlich, daß unverwaltete Programmteile in den häufigsten Fällen die höchsten Ausführungsgeschwindigkeiten erzielen. Der Zugewinn an Komfort verwalteter Anwendungen (Ausführungssicherheit, automatische Speicherverwaltung, Plattformunabhängigkeit, etc.) führt, bedingt durch den zusätzlichen Verwaltungsaufwand, zu gewissen Einschränkungen bzgl. der Ausführungsgeschwindigkeit. Dies gilt insbesondere dann, wenn der Programmablauf, wie im Fall der Berechnungskerne, in einem großen Maße von Speicherbewegungen bestimmt ist (*Range Checking*, Typüberprüfung). Die Berechnungsschleife vollzieht sich im Wesentlichen wie in Abschnitt 3.1 beschrieben. Dabei werden in der Kollision vornehmlich mathematische Operationen durchgeführt, wohingegen in der Propagation lediglich Speicherbewegungen stattfinden. Zu beachten ist, daß der Anteil an Rechenoperationen im Verhältnis zu den Speicherbewegungen bei der Simulation mittels Momentenmodell etwas höher ist, als bei der LBGK-Methode. Dies hat zur Folge, daß in diesem Fall die Optimierungsfunktion der Laufzeitumgebung zu höheren Ausführungsgeschwindigkeiten führt, die in einigen Fällen sogar die des unverwalteten Kerns übertreffen. Der vornehmlich sequenzielle Zugriff auf die Arraydaten innerhalb der Berechnungsschleife bedingt, daß sich die verschachtelten Arrays schneller verhalten als die sequenziellen Arrays (siehe Abschnitt 2.6). Bei weniger zusammenhängenden Zugriffen auf die Arrays wird die hohe Lokalität der Daten in sequenziellen Arrays in Form von schnelleren Zugriffen deutlich [3]. Die Verwendung der *unsafe-Option* zur Erhöhung der Ausführungsge-

schwindigkeit von verwalteten Anwendungen ist nur bedingt empfehlenswert. Zum einen führt die Verwendung des Schlüsselwortes *unsafe* dazu, daß die Runtime in ihrer Optimierungsfunktion eingeschränkt wird (Schlüsselwort: *fixed*). Zum anderen steht der zusätzliche Aufwand nicht im Verhältnis zu den Ergebnissen. Außerdem geht aus dem Benchmark hervor, dass annähernd dieselben Ergebnisse unter Verwendung der *Managed Extensions* für die Sprache *C++* erzielt werden können (managed Code - unmanaged Data). Diese Erkenntnis ist naheliegend, da im Wesentlichen in beiden Fällen eine verwaltete Anwendung mittels Zeiger auf den jeweiligen Arrays agiert.

5 Zusammenfassung

Aus den vorangegangenen Ergebnissen ist ersichtlich, daß das *.NET Framework* durchaus für numerische Probleme eingesetzt werden kann. Dabei können unter bestimmten Voraussetzungen auch von verwalteten Anwendungen hohe Ausführungsgeschwindigkeiten erzielt werden. Auch innerhalb verwalteter Anwendungen können bei Bedarf vorhandene unverwaltete Bibliotheken verwendet werden. Moderne Programmiersprachen wie *Java* oder *C#* ermöglichen die Entwicklung von Anwendungen, die den Ansprüchen heutiger objektorientierter Entwürfe gerecht werden. Der hohe Komfort der Anwendungsentwicklung mittels moderner Programmiersprachen macht sich nicht zuletzt auch in Bezug auf die Entwicklungszeit bemerkbar.

Literatur

[1] T. Archer, A. Whitechapel: Inside C# 2nd ed., Microsoft Press Deutschland, Unterschleißheim, 2002

[2] P. L. Bhatnagar, E. P. Gross and M. Krook: A model of collision processes in gases, Physical Review, 94:511, 1954

[3] F. Gilani: Harness the Features of C# to Power Your Scientific Computing Projects, MSDN Magazine, März 2004

[4] M. Krafczyk: Gitter-Boltzmann-Methoden: Von der Theorie zur Anwendung, Habilitationsschrift, Lehrstuhl für Bauinformatik, TU-München, 2001

[5] J. Linxweiler: Entwicklung eines auf der Lattice-Boltzmann-Methode basierenden interaktiven Strömungssimulators in C# mit verschiedenen Berechnungskernen, Studienarbeit, Institut für Computeranwendungen im Bauingenieurwesen, TU Braunschweig, 2004

[6] A. McNaughton: Boosting the Performance of the Microsoft .NET Framework, Intel Corporation, MSDN Magazine, August 2002

[7] J. Richter: Microsoft .NET Framework Programmierung, Microsoft Press Deutschland, Unterschleißheim, 2002

[8] Standard ECMA-334, C# Language Specification, Dezember 2001