Jül-4382

Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre (JSC)

Untersuchung des flexiblen Finite-Elemente-Toolkits FEniCS auf einer BlueGene/Q-Architektur im Hinblick auf parallele Effizienz

Thomas Breuer

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
FORSCHUNGSZENTRUM

Untersuchung des flexiblen Finite-Elemente-Toolkits FEniCS auf einer BlueGene/Q-Architektur im Hinblick auf parallele Effizienz

Thomas Breuer

Berichte des Forschungszentrums Jülich; 4382
ISSN 0944-2952
Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre (JSC)
Jül-4382

Vollständig frei verfügbar über das Publikationsportal des Forschungszentrums Jülich (JuSER)
unter www.fz-juelich.de/zb/openaccess

Forschungszentrum Jülich GmbH
Zentralbibliothek, Verlag
52425 Jülich
Tel.: +49 2461 61-5220
Fax: +49 2461 61-6103
E-Mail: zb-publikation@fz-juelich.de
www.fz-juelich.de/zb

Zusammenfassung

Zur Lösung von Differentialgleichungen mit Hilfe der Finite-Elemente-Methode (FEM) gibt es viele Implementierungen, die in unterschiedlichsten Simulationsprogrammen eingesetzt werden. Diese Software ist meistens auf ein bestimmtes Anwendungsgebiet spezialisiert und auch nicht sehr portabel für die Verwendung auf unterschiedlichen Computerarchitekturen. Im Gegensatz zu dieser Spezialisierung gibt es auf der anderen Seite Simulationssoftware wie das Open-Source-Projekt FEniCS, welches einen allgemeineren Ansatz verfolgt. Mit diesem Ansatz ist es möglich, verschiedenste Problemstellungen aus unterschiedlichen Bereichen mittels FEM weitgehend automatisch und effizient zu lösen.
Im ersten Teil der Arbeit wird die leichte Benutzbarkeit von FEniCS an einem einfachen Beispiel demonstriert. Im zweiten Teil konnte gezeigt werden, dass FEniCS auf der massiv parallelen BlueGene/Q-Architektur JUQUEEN installiert werden kann. Zudem hat die Auswertung einer großen Anzahl an Simulationen verschiedener Anwendungen, insbesondere der Navier-Stokes-Gleichungen, ein sehr gutes Skalierungsverhalten erkennen lassen. Des Weiteren konnten neben der deutlich schlechter werdenden I/O-Performance bei steigender Anzahl an MPI Prozessen, weitere kritische Bereiche ausgemacht werden, die in zukünftigen Arbeiten mittels spezieller Profiling Tools untersucht werden können.

Abstract

Various implementations are used in different simulation programs to solve differential equations on the basis of the finite element method (FEM). This kind of software is often limited to a certain area of application and cannot be transfered to other computer architectures. In contrast, simulation software like the open source project FEniCS traces a more general approach. FEniCS makes use of a method, which enables the solution of various problems from different areas in most cases automatically and efficiently by means of FEM.
The first part of the thesis comprises a demonstration of the convenient handling of FEniCS by a simple example. The installation of FEniCS on the massively parallel BlueGene/Q-architecture JUQUEEN will be discussed within the second part. The discussion is based on different applications focussing on the Navier-Stokes equations. Moreover, the evaluation of a large number of simulations indicates a very good scaling behavior. In addition to decreasing I/O-performance, which comes along with an increasing number of MPI processes, other critical areas exist. Those might be examined in future with the help of specific profiling tools.

Inhaltsverzeichnis

1. Einleitung

Simulationen haben in den letzten Jahrzehnten einen immer höheren Stellenwert in der Wissenschaft bekommen. Sie werden heutzutage als dritte Säule neben Experimenten und Theorie angesehen. Auf Basis von mathematischen Modellen helfen sie dabei Gesetzmäßigkeiten in der Natur nachvollziehen zu können. Beispiele für solche Modelle sind unter anderem die Maxwell-Gleichungen im Elektromagnetismus, die Lamé-Navier-Gleichungen in der Elastizität oder die Navier-Stokes-Gleichungen in der Strömungsmechanik.
Eine der Herausforderungen, vor der die Wissenschaftler stehen, ist das Entwickeln von Methoden, mit denen diese komplizierten Modelle (meist Differentialgleichungen) effizient gelöst werden können. Hierzu etablierten sich das Finite-Volumen-Verfahren, die Finite-Differenzen-Methode und die Finite-Elemente-Methode (FEM). Um die unterschiedlichen Phänomene simulieren zu können, müssen einige Parameter dieser Methoden angepasst werden, wie zum Beispiel die richtige Wahl der Diskretisierung. Daher existieren sehr viele hoch spezialisierte Simulationscodes.
Im Gegensatz zu dieser Spezialisierung gibt es auf der anderen Seite Simulationssoftware wie das Open Source Projekt FEniCS[1], welches einen allgemeineren Ansatz verfolgt. Mit diesem Ansatz ist es möglich, verschiedenste Problemstellungen aus unterschiedlichen Bereichen mittels FEM zu lösen, ohne die Simulationssoftware direkt verändern zu müssen.

Die Komplexität der betrachteten Gebiete und Phänomene kann soweit gesteigert werden, dass normale Computer nicht mehr in der Lage sind Ergebnisse zu liefern. Damit solche Problemstellungen trotzdem behandelt werden können, gibt es Supercomputer wie beispielsweise JUROPA und JUQUEEN im Forschungszentrum Jülich.
Diese Supercomputer haben spezielle Architekturen, auf die die jeweilige Simulationssoftware in der Regel angepasst werden muss. Auf Grund dessen ist die Portabilität des Programms und die Anzahl der Anwender eingeschränkt. Bei den Programmen, die auf verschiedenen Systemen lauffähig sind, muss jedoch üblicherweise der Anwendungsquellcode entsprechend angepasst werden. Im Gegensatz dazu steht FEniCS,

[1] http://fenicsproject.org/

das nicht nur flexibel in den Anwendungsgebieten ist, sondern auch auf unterschiedlichen Systemen läuft, ohne dass der Quellcode der Anwendung verändert werden muss. Die Idee ist es, eine Simulation auf dem Laptop zu implementieren und mit kleineren Gebieten zu testen, damit sie dann anschließend auch für rechenintensivere Fälle auf einem Supercomputer genutzt werden kann. Diese Portabilität wird erreicht, indem der Anwender das Problem und die Lösungsalgorithmen formuliert und das Programm die Partitionierung und Kommunikation übernimmt.

Im Rahmen dieser Arbeit wird ein Profiling von FEniCS durchgeführt. Dabei wird für die Poisson-Gleichung und für verschiedene Anwendungsfälle der Navier-Stokes-Gleichung die Skalierung auf JUQUEEN untersucht.
Ein zusätzlicher Aspekt, der die große Flexibilität von FEniCS untermauert, ist die Bereitstellung sowohl einer Python API als auch einer C++ API. Für diese Arbeit sind alle zu untersuchenden Anwendungen in Python und C++ programmiert worden. Allerdings werden nur die C++ Implementierungen für die Simulationen auf dem Supercomputer verwendet.
Auf Grund seiner Vielseitigkeit und der verhältnismäßig einfachen Benutzung ist FEniCS für viele Anwender aus unterschiedlichsten Wissenschaftsgebieten attraktiv, die in dieser Umgebung beispielsweise auch ihre eigenen Algorithmen einfach untersuchen können.

2. Das FEniCS Projekt

Im Rahmen dieser Arbeit wurde das Verhalten von FEniCS auf einem Großrechner anhand mehrerer Beispiele untersucht. In diesem Kapitel werden dazu zunächst das FEniCS Projekt und deren einzelne Module vorgestellt. Es basiert im Wesentlichen auf Logg et al. [7].

2.1. Allgemeines

Das FEniCS Projekt wurde 2003 mit der Idee gestartet, die Lösung mathematischer Modelle, basierend auf Differentialgleichungen, zu automatisieren und bestand ursprünglich aus den Bibliotheken DOLFIN und FIAT. Seitdem ist das Projekt immer weiter gewachsen, sodass es heute aus folgenden Hauptkomponenten besteht: DOLFIN, FFC, FIAT, Instant, UFL und UFC. Dabei zählen zu den Hauptentwicklern unter anderem das Simula Research Laboratory, die University of Cambridge, die University of Chicago, die Baylor University und das KTH Royal Institute of Technology.
Ein großer Vorteil von FEniCS ist, dass es für viele Anwender geeignet ist. Es werden neben der manuellen Installation des Quellcodes auch Binärpakete für verschiedene Betriebssysteme wie z.B. Debian, Ubuntu und OS X angeboten. Zusätzlich stehen dem Benutzer dieser Open Source Software sowohl eine C++ als auch eine Python Schnittstelle zur Programmierung zur Verfügung.

2.2. DOLFIN

DOLFIN [9] ist das größte Paket von FEniCS. Es stellt eine einfache Benutzerschnittstelle bereit, die die Funktionalität der anderen Komponenten und der externen Software vereinheitlicht und deren Kommunikation untereinander regelt.
Die Abhängigkeiten der einzelnen Komponenten sind in der Abbildung 2.1 dargestellt. Eine Anwendung, die entweder in Python oder C++ implementiert werden kann, setzt auf der DOLFIN Schnittstelle auf. Die Variationsformulierung des zu lösenden Problems wird in der Unified Form Language (Kapitel 2.6) beschrieben, welches vom

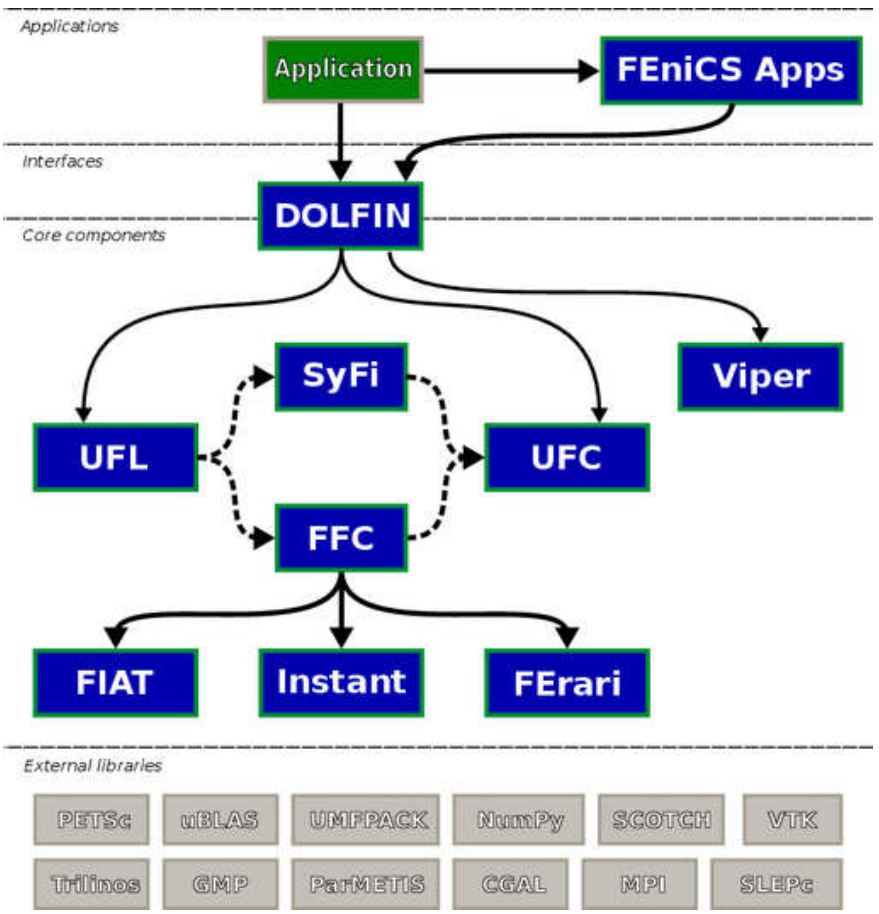

Abbildung 2.1.: Übersicht über die Beziehungen der einzelnen FEniCS Komponenten. Quelle: Logg et al. [9]

Form Compiler (Kapitel 2.3) in Unified Form-assembly Code (Kapitel 2.7) kompatiblen Code übersetzt wird. Dieser Code wird wiederum von DOLFIN verwendet, um die Matrizen für die Berechnung der Lösung zu konstruieren. Die weiteren Abhängigkeiten dieser Komponenten werden in den entsprechenden Kapiteln weiter behandelt.

Neben den erwähnten FEniCS Modulen benutzt DOLFIN auch externe Software. Dazu zählen unter anderem die Linear Algebra Bibliotheken PETSc[1], Trilinos[2], uBLAS[3] und MTL4[4] sowie die Bibliotheken ParMETIS[5] und SCOTCH[6] zur Gitter Partitionierung.

[1] http://www.mcs.anl.gov/petsc/
[2] http://trilinos.org/
[3] http://www.boost.org/doc/libs/1_57_0/libs/numeric/ublas/doc/
[4] http://www.simunova.com/de/mtl4
[5] http://glaros.dtc.umn.edu/gkhome/views/metis/
[6] http://www.labri.fr/perso/pelegrin/scotch/

2.2.1. Exemplarische Implementierung der Poisson-Gleichung

Anhand der Implementierung der Lösung einer Poisson-Gleichung wird die einfache Benutzbarkeit von FEniCS für viele verschiedene Anwendungsgebiete verdeutlicht. Die mathematische Formulierung des gewählten Problems sieht wie folgt aus:

$$\begin{array}{rcll} -\Delta \mathbf{u} & = & f & \text{in } \Omega \\ u_0 & = & 1 + x^2 + y^2 & \text{auf } \partial\Omega \\ f & = & -6 & \\ \Omega & = & [0,1] \times [0,1] & \end{array}$$

Da FEniCS mit der Variationsformulierung arbeitet, muss diese noch hergeleitet werden. Für das angegebene Problem ergibt sich:
Finde ein $\mathbf{u} \in V$, so dass

$$\underbrace{\int_\Omega \nabla \mathbf{u} : \nabla \mathbf{v} \, \mathrm{d}x}_{a(\mathbf{u},\mathbf{v})} = \underbrace{\int_\Omega f \cdot \mathbf{v} \, \mathrm{d}x}_{L(\mathbf{v})} \quad \forall \mathbf{v} \in \hat{V}$$

mit

$$\hat{V} = \{\mathbf{v} \in H^1(\Omega) : \mathbf{v} = 0 \text{ auf } \partial\Omega\}$$
$$V = \{\mathbf{v} \in H^1(\Omega) : \mathbf{v} = u_0 \text{ auf } \partial\Omega\}$$

Gelöst werden muss $a(\mathbf{u}, \mathbf{v}) = L(\mathbf{v})$. Dabei ist $a(\mathbf{u}, \mathbf{v})$ eine Bilinearform und $L(\mathbf{v})$ eine Linearform.

Implementierung

Die gesamte Python-Implementierung des beschriebenen Poisson-Problems wird im Listing 2.1 dargestellt. In den nachfolgenden Abschnitten wird auf die einzelnen Zeilen näher eingegangen.

```
1  from dolfin import *
2
3  # Gitter erstellen und Funktionsraum definieren
4  mesh = UnitSquareMesh(8,8)
5  V = FunctionSpace(mesh, "Lagrange", 1)
6
7  # Randwerte definieren
8  u0 = Expression("1+x[0]*x[0]+2*x[1]*x[1]")
9  bc = DirichletBC(V, u0, "on_boundary")
10
11 # Variationsproblem definieren
12 f = Constant(-6.0)
13 u = TrailFunction(V)
14 v = TestFunction(V)
15 a = inner(grad(u), grad(v))*dx
16 L = f*v*dx
17
18 # Loesung berechnen
19 u = Function(V)
20 solve(a == L, u, bc)
21
22 # Loesung darstellen
23 plot(u)
24 interactive()
25
26 # Loesung in eine Datei im VTK Format schreiben
27 file = File("poisson.pvd")
28 file << u
```

Listing 2.1: Python Code zur Berechnung des Poisson-Problems

Gitter erstellen

DOLFIN stellt eigene Klassen zur Verfügung, um einfache Gitter zu erzeugen, wie beispielsweise das `UnitSquareMesh`.

```
1 mesh = UnitSquareMesh(8,8)
```

In diesem Fall handelt es sich um ein Einheitsquadrat, welches in 8×8 Rechtecke unterteilt wird, die wiederum jeweils in zwei Dreiecke geteilt werden. Somit ergeben sich insgesamt $8 * 8 * 2 = 128$ Dreieckszellen und $9 * 9 = 81$ Eckpunkte.
Zusätzlich gibt es die Möglichkeit komplexere Geometrien über eine Datei einzulesen. Hierbei werden verschiedene Dateiformate wie z.B. VTK[7] und HDF5[8] unterstützt. Sollte die Simulation parallel ausgeführt werden, wird das Gitter automatisch von ParMETIS oder SCOTCH auf die einzelnen Prozesse partitioniert.

Funktionsraum definieren

Nach dem Erstellen des Gitters kann der diskrete Funktionsraum für dieses Gitter erstellt werden.

```
1 V = FunctionSpace(mesh, "Lagrange", 1)
```

Dabei enthält das erste Argument die Gitterinformationen, das zweite Argument gibt den Elementtypen an und das dritte Argument die Ordnung der Basisfunktionen auf dem Element. FEniCS hält mehrere verschiedene Elementtypen bereit, die leicht ausgetauscht werden können, indem das zweite oder das dritte Argument in dem gezeigten Aufruf angepasst wird. Eine Auswahl der verschiedenen Elementtypen ist in Kapitel 2.4 zu sehen.
Bei der Definition der Funktionsräume und der Variationsformulierung gibt es wesentliche Unterschiede in der Implementierung in Abhängigkeit davon, ob die Anwendung in Python oder C++ geschrieben ist. Darauf wird in den Kapiteln 2.3 und 2.6 näher eingegangen.

Randwerte definieren

In dem Beispiel handelt es sich um Dirichlet Randwerte, die auf dem gesamten Rand durch die Funktion $u_0 = 1 + x^2 + y^2$ beschrieben werden.

[7] http://www.vtk.org/
[8] http://www.hdfgroup.org/HDF5/

```
1 u0 = Expression("1+x[0]*x[0]+2*x[1]*x[1]")
2 bc = DirichletBC(V, u0, "on_boundary")
```

Die Formel wird hier als String in C++ Syntax geschrieben. Das hat den Vorteil, dass der Ausdruck automatisch in eine effizient kompilierte C++ Funktion umgewandelt wird. Dieses Konstrukt wird der Funktion `DirichletBC` als zweites Argument übergeben. Das erste Argument enthält den Funktionsraum und das dritte Argument bestimmt den Bereich, auf den die Funktion u_0 als Randwerte angewendet werden soll. Das Argument `on_boundary` wird von DOLFIN bereitgestellt und liefert `True` zurück, falls ein Punkt auf dem physikalischen Rand des Gitters liegt. Es besteht allerdings auch die Möglichkeit, Randwerte nur auf bestimmten Teilen des Gebiets zu definieren.

Variationsproblem definieren

```
1 f = Constant(-6.0)
2 u = TrailFunction(V)
3 v = TestFunction(V)
4 a = inner(grad(u), grad(v))*dx
5 L = f*v*dx
```

Zunächst wird eine Funktion `f`, die konstant auf dem Gebiet ist, definiert. Dann folgen in Zeile zwei und drei die Definitionen der Ansatz- und Testfunktionen im Funktionsraum `V`. Damit sind alle Objekte erstellt, die benötigt werden, um die Variationsformulierung $a(\mathbf{u}, \mathbf{v}) = L(\mathbf{v})$ für die Gleichung $-\Delta\mathbf{u} = -6$ darzustellen, wie in Zeile vier und fünf zu sehen ist.
Eine der großen Stärken von FEniCS ist die Formulierung des Variationsproblems in einer Syntax, die der mathematischen Schreibweise sehr ähnlich ist und für viele PDE's und komplizierte Ausdrücke verwendet werden kann. Dies macht das FEniCS Modul UFL (siehe Kapitel 2.6) möglich. Wie bereits erwähnt, ist hier der Hauptunterschied zwischen der Python und der C++ Version. Die genauere Beschreibung dieses Unterschiedes findet in den Kapiteln 2.3 und 2.6 statt.

Lösung berechnen

Nachdem a und L definiert sind und die Dirichlet Randwerte vorliegen, kann die finite Elemente Lösung $\mathbf{u}$ berechnet werden.

```
1 u = Function(V)
2 solve(a == L, u, bc)
```

Dazu wird als erstes die Funktion **u** als Platzhalter für die Lösung definiert. Im zweiten Schritt folgt dann die Berechnung der Gleichung $a(\mathbf{u}, \mathbf{v}) = L(\mathbf{v})$. In diesem Beispiel wird eine Standardmethode gewählt, da keine weiteren Argumente dem `solve` Befehl übergeben wurden. Die Methode und die Prekonditionierer können allerdings auch explizit angegeben werden. Die hier zur Verfügung stehende Auswahl hängt von den Bibliotheken ab, mit denen DOLFIN konfiguriert worden ist. Neben PETSc werden auch Trilinos, uBLAS und MTL4 unterstützt. Zudem kann der Benutzer Parameter setzen, wie z.B. Toleranzen und Grenzwerte, die das Verhalten des Lösers beeinflussen.

Ergebnisse darstellen und speichern

Zur Nachbearbeitung der Ergebnisse kann man sich diese direkt im Programm anzeigen lassen oder in eine Datei speichern.

```
1 plot(u)
2 interactive()
```

Der einfachste Weg, sich schnell die Lösung und das Gitter anzusehen, geht über den `plot` Befehl. Dabei ist der `interactive()` Aufruf wichtig, damit das Fenster auf dem Bildschirm bleibt und sich nicht sofort wieder schließt.
Das Ergebnis des behandelten Poisson-Problems ist in der Abbildung 2.2 zu sehen.

Abbildung 2.2.: Darstellung der Lösung des Poisson-Problems

Die Ergebnisse können auch in eine Datei gespeichert werden, um sie anschließend mit speziellen Tools wie ParaView[9] zu visualisieren.

```
1 file = File("poisson.pvd")
2 file << u
```

Wie beim Laden des Gitters stehen auch hier verschiedene Datei Formate zur Auswahl, wie beispielsweise VTK und HDF5.

Programm Aufruf

Ein großer Vorteil von FEniCS ist, dass derselbe Quellcode sowohl für seriell als auch parallel ausgeführte Simulationen verwendet werden kann. Damit bleibt dem Entwickler das doppelte Implementieren erspart.
Wenn das Poisson-Problem aus Listing 2.1 in der Datei `poisson.py` gespeichert ist, kann das Programm auf folgende Weise seriell gestartet werden:

```
1 python poisson.py
```

Listing 2.2: Serieller Aufruf des Beispiels poisson.py

Zur parallelen Ausführung muss lediglich ein ‘‘`mpiexec -np #`’’ oder ‘‘`mpirun -np #`’’ dem seriellen Aufruf vorgestellt werden:

```
1 mpiexec -np 4 python poisson.py
```

Listing 2.3: Paralleler Aufruf des Beispiels poisson.py mit vier Prozessen

Dieses Beispiel kann als Konstruktionsbasis verstanden werden, anhand derer komplexere Simulationsprogramme aufgebaut werden können.

[9] http://www.paraview.org/

2.3. FEniCS Form Compiler (FFC)

Eins der Hauptmerkmale von FEniCS ist die automatische Code Generierung für individuelle Variationsformulierungen. Diese beruht auf einem sogenannten Form Compiler für offine oder just-in-time Kompilierung von Quellcode. In diesem Kapitel wird der FEniCS Form Compiler [8], kurz FFC, beschrieben.

2.3.1. Grundidee

Das Erstellen von linearen und nichtlinearen Gleichungssystemen ist ein Hauptbestandteil von finite Elemente Programmen. FEniCS verwendet hierzu einen allgemeinen Algorithmus, der auf der Berechnung des Elementtensors und dessen globaler Zuordnung basiert. Sowohl der Elementtensor als auch seine Zuordnung unterscheiden sich stark bei verschiedenen finiten Elementen und Variationsformulierungen. Deswegen wird spezieller Code benötigt, der für die jeweilige Anwendung per Hand entwickelt werden muss.

Für den FEniCS Benutzer entfällt dieses sehr mühsame und fehleranfällige Implementieren per Hand, da es vom Form Compiler FFC vorgenommen wird. Dieser generiert Quellcode zur Berechnung des Elementtensors und dessen globaler Zuordnung, der wiederum zur Konstruktion der finiten Element Matrizen und Vektoren verwendet werden kann.

Konkret erwartet der Form Compiler als Eingabe die mathematische Variationsformulierung im UFL Format (siehe Kapitel 2.6) und erzeugt anhand derer als Ausgabe C++ Code, welcher zur UFC Schnittstelle (siehe Kapitel 2.7) konform ist. Dieser Prozess ist schematisch in Abbildung 2.3 dargestellt.

Abbildung 2.3.: Schematische Darstellung der Verarbeitung einer Variationsformulierung mittels des Form Compilers FFC

2.3.2. Verwendung

FFC stellt drei verschiedene Schnittstellen zur Verfügung: eine Python, eine just-in-time (JIT) und eine Kommandozeilen Schnittstelle. Die Python Nutzer verwenden in der Regel die Python Schnittstelle nicht direkt, sondern stützen sich auf DOLFIN, das sich um die Kommunikation mit dem Form Compiler kümmert. DOLFIN wiederum nutzt intern das JIT Interface als Verbindung zum Form Compiler. Es erlaubt DOLFIN, nahtlos Quellcode für eine in Python geschriebene Anwendung zu generieren und zu kompilieren. Damit werden die Leistungsvorteile von generiertem C++ Code mit der Einfachheit einer Skriptsprache kombiniert. Somit werden viele Python Benutzer nur sehr selten mit den einzelnen Schnittstellen direkt konfrontiert.
Die Kommandozeilen Schnittstelle muss von den C++ Programmierern benutzt werden. Der C++ Benutzer schreibt seine Anwendung in C++, muss aber die Variationsformulierung in einer eigenen Datei im UFL Format codieren, was nichts anderes ist als Python Code für bestimmte abgeleitete Klassen. Das hat den Vorteil, dass die Gleichungen in einer Syntax beschrieben werden können, die der mathematischen Schreibweise sehr ähnlich ist. Der über die Kommandozeile aufgerufene Form Compiler (siehe Listing 2.4) verlangt als Eingabe die erstellte UFL Datei und erzeugt daraufhin eine UFC kompatible C++ Header Datei. Diese kann dann wiederum in der Anwendung importiert werden, um die finite Elemente Matrizen zu konstruieren. Ein konkretes Beispiel ist in Kapitel 2.6 zu finden.

```
1 ffc FormFile.ufl
```

Listing 2.4: Beispielhafter Aufrufe des Form Compilers FFC

2.4. FInite element Automatic Tabulator (FIAT)

Der FInite element Automatic Tabulator (FIAT) [6] ist ein Python Programm, welches die Vorteile von NumPy[10] zur Steigerung der Effizienz ausnutzt. Es dient der automatischen Konstruktion von verschiedenen finiten Elementen mit unterschiedlicher Ordnung. Unterstützt werden u.a. folgenden Elementtypen: Brezzi-Douglas-Marini, Crouzeix-Raviart, (Discontinuous) Lagrange, Nédélec, Raviart-Thomas und weitere (siehe Abbildung 2.4), eine übersichtliche Darstellung aller Elementtypen zeigt die `Periodic Table of the Finite Elements`[11].

[10] http://www.numpy.org/
[11] http://femtable.org/

Damit stellt es die Basisfunktionen für den Form Compiler FFC zur Verfügung. Die meisten Benutzer werden FIAT nie direkt verwenden, da die Interaktion über den Form Compiler FFC geregelt wird.

Abbildung 2.4.: Darstellung einiger Basiselemente, die von FEniCS unterstützt werden

2.5. Instant

Instant ist ein kleines Python Modul, das unmittelbares Inlining von C und C++ Code in Python Programmen ermöglicht. Es nutzt dazu die Programme SWIG[12] und Distutils[13]. Innerhalb von FEniCS wird Instant vom Form Compiler FFC und DOLFIN verwendet um C++ Code zu einem Python Modul zu kompilieren. Zudem besitzt Instant eine Art Cache, sodass der Quellcode nicht neu kompiliert werden muss wenn dieser sich im Vergleich zur vorherigen Ausführung nicht geändert hat. Dazu berechnet und speichert es die SHA1 Summe [4] des Codes. Des Weiteren verfügt Instant über die Funktion, NumPy Arrays zwischen Python Code und C/C++ Code zu transferieren.

[12] http://www.swig.org/
[13] https://docs.python.org/2/distutils/

2.6. Unified Form Language (UFL)

Die Unified Form Language (UFL) [1] ist eine in Python eingebettete Sprache zur Definition von finiten Element Diskretisierungen für Variationsformulierungen. Sie definiert ein flexibles Interface für die Wahl von finiten Element Räumen und definiert Ausdrücke für die schwache Form in einer Art und Weise, die sehr mathematischen Schreibweise ähnelt.
Damit ist UFL eine weitere Hauptkomponente des FEniCS Projekts, die dem Benutzer eine Schnittstelle bietet, um die zu lösende PDE sehr einfach zu implementieren. Zudem unterstützt es automatische Differentiation [3].
Die Unified Form Language ist das Eingabeformat des Form Compilers FFC, der die definierte Variationsformulierung weiter verarbeitet (vergleiche Kapitel 2.3). In der Python Version von DOLFIN sind UFL und FFC nahtlos eingebunden, sodass die Definition der Funktionsräume und die Variationsformulierungen direkt in der Anwendung implementiert werden können, wie im Beispiel 2.2.1 gezeigt wurde. Wenn die Anwendung in C++ implementiert wird, werden diese Definitionen in eine eigene Datei mit der Endung `.ufl` geschrieben und der Form Compiler explizit aufgerufen (siehe Listing 2.4). Der Inhalt einer solchen Datei ist im Listing 2.5 zu sehen. Dieses zeigt die Definition des Funktionsraumes und die Variationsformulierung im UFL Format für die Poisson-Gleichung aus Kapitel 2.2.1. Als finite Elemente werden Lagrange Dreiecke erster Ordnung verwendet.

```
1 element = FiniteElement("Lagrange", triangle, 1)
2
3 u = TrialFunction(element)
4 v = TestFunction(element)
5 f = Coefficient(element)
6
7 a = inner(grad(u), grad(v))*dx
8 L = f*v*dx
```

Listing 2.5: Variationsformulierung im UFL Format für die Poisson-Gleichung aus Kapitel 2.2.1. Gespeichert in der Datei `poisson.ufl`

Diese Datei kann dann mittels des Kommandozeilen Interfaces des Form Compilers FFC in eine C++ Header Datei übersetzt werden (siehe Listing 2.6).

```
1 ffc poisson.ufl
```

Listing 2.6: Aufruf des Form Compilers FFC für die Datei `poisson.ufl`

Diese Header Datei enthält UFC kompatiblen Code, der in die C++ Anwendung importiert werden kann, damit auf die Funktionsräume und Variationsformulierungen zugegriffen werden kann.

Auf Grund dieser einfachen Syntax kann der Benutzer ohne großen Aufwand die Elementtypen für die Simulation verändern. So zeigt Listing 2.7 die Anwendung eines Brezzi-Douglas-Marini Tetraeder dritter Ordnung und Listing 2.8 die Anwendung eines Crouzeix-Raviart Dreiecks erster Ordnung.

```
1 element = FiniteElement("BDM", tetrahedron, 3)
```

Listing 2.7: Brezzi-Douglas-Marini Tetraeder dritter Ordnung

```
1 element = FiniteElement("Crouzeix-Raviart", triangle, 1)
```

Listing 2.8: Crouzeix-Raviart Dreieck erster Ordnung

Darüber hinaus werden deutlich komplexere Variationsformulierungen unterstützt als die in Listing 2.5 dargestellte.
Zusammengefasst nimmt die Unified Form Language dem Anwender viel Arbeit ab, indem sie ein einfaches Interface zur Definition des mathematischen Problems bereitstellt. In Kombination mit dem FFC, der unter anderem das Erstellen der Basisfunktionen übernimmt, findet eine automatisierte finite Elemente Berechnung statt.

2.7. Unified Form-assembly Code (UFC)

Der Unified Form-assembly Code (UFC) [2] ist die Schnittstelle zwischen den problemspezifischen und den allgemeinen Komponenten eines finite Elemente Programms. Insbesondere definiert das UFC Interface die Struktur und Signatur des Codes, der vom Form Compiler FFC für DOLFIN generiert wird. Diese Schnittstelle kann auf einen großen Bereich von finiten Elementen angewendet werden. Dies beinhaltet zum Beispiel auch gemischte finite Elemente. Zudem kann es mit Bibliotheken verwendet

werden, die sich stark in ihrem Design unterscheiden. Hierzu hängt UFC von keiner anderen FEniCS Komponente ab und besteht lediglich aus einer einzigen Header Datei `ufc.h`, die ein C++ Interface spezifiziert, welches vom Form Compiler FFC implementiert werden muss.

UFC innerhalb einer FEniCS Simulation

Zunächst definiert der Benutzer die finiten Elemente und die Variationsformulierung im UFL Format. Dieses dient als Eingabe für den Form Compiler FFC, der daraus den Elementtensor und dessen globale Zuordnung bestimmt und dies im UFC kompatiblen Code implementiert. Dieser automatisch generierte Quellcode kann dann wiederum von DOLFIN in Kombination mit den Gitterinformationen verwendet werden, um die entsprechenden globalen Matrizen zu konstruieren. Diese Matrizen werden dabei ebenfalls automatisch direkt in der Datenstruktur erstellt, die die lineare Algebra Bibliothek verlangt, mit der der Benutzer die Simulation instrumentiert hat. Durch das Definieren eines bestimmten Parameters in der Anwendung kann der Benutzer dieses sogenannte linear algebra backend für die Simulation festlegen. Der beschriebene Ablauf ist in Abbildung 2.5 veranschaulicht.

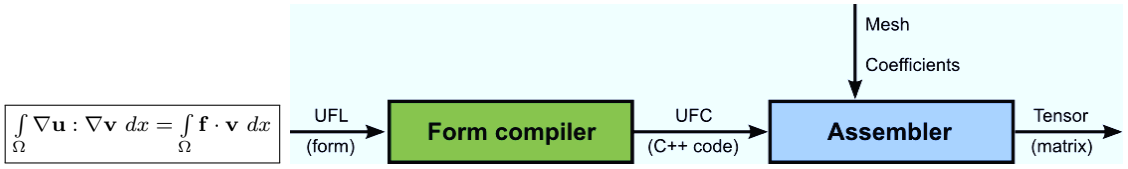

Abbildung 2.5.: Flussdiagramm zur Konstruktion von finiten Elementen in FEniCS

2.8. Zusammenfassung

In diesem Kapitel wurden die wichtigsten FEniCS Module und der Bedeutung innerhalb einer Simulation anhand eines einfachen Beispieles verdeutlicht. Allerdings wurden in dieser Beschreibung nicht alle Details der einzelnen Komponenten einbezogen, da diese für das allgemeine Verständnis von FEniCS nicht wichtig sind. Fortgeschrittene Benutzer und Entwickler können sich an dieser Stelle mit dem FEniCS Buch [7] weiterhelfen.
Abschließend lässt sich feststellen, dass FEniCS eine sehr benutzerfreundliche Schnittstelle zur Verfügung stellt, die eine einfache Definition und Lösung eines mathematischen Problems mit Hilfe von finiten Elementen ermöglicht.

3. Portierung auf JUQUEEN

Ein großer Vorteil von FEniCS ist, dass es unter verschiedenen Betriebssystemen und Computerarchitekturen lauffähig ist, ohne Veränderungen an den einzelnen Komponenten oder seinem Anwendungsquellcode vornehmen zu müssen. Dabei werden im High Performance Computing (HPC) Bereich sowohl Shared als auch Distributed Memory Systeme unterstützt. Der Benutzer kann folglich seine Anwendung für eine kleine Problemgröße auf einem Laptop implementieren und testen und diese anschließend auf einen Supercomputer rechnen. Auf Grund dessen kann doppelte Arbeit vermieden werden, wenn das Verhalten von FEniCS auf einem Großrechner wie JUQUEEN untersucht werden soll. Dazu müssen die einzelnen Komponenten zunächst einmal installiert werden.

3.1. JUQUEEN Konfiguration

Für diese Arbeit wurden Simulationen auf dem Supercomputer JUQUEEN[1] im Jülich Supercomputing Centre durchgeführt. Hierbei handelt es sich um eine IBM BlueGene/Q Architektur mit 1.6 GHz IBM PowerPC A2 Prozessoren, die mit Simultaneous Multithreading arbeiten. Dies bedeutet, dass ein Prozessor in diesem Fall bis zu vier Hardware-Threads gleichzeitig bearbeiten kann. Die Anzahl dieser Hardware-Threads wird für einen Node angegeben und über den Parameter mit dem Namen `ranks-per-node` gesteuert. Von diesen Prozessoren befinden sich jeweils 16 auf einem Node inklusive 16 GB SDRAM-DDR3. 32 dieser Compute Nodes werden zu einem Nodeboard zusammengefasst. Eine Midplane besteht wiederum aus 16 Nodeboards und zwei Midplanes bilden ein Rack. Insgesamt besteht JUQUEEN aus 28 Racks und damit 28672 Nodes beziehungsweise 458752 Prozessoren. Das gesamte System kommt auf 448 TB Hauptspeicher. Ein Rack entspricht einem Schrank, die in sieben Reihen mit jeweils vier Racks in einer Rechnerhalle aufgestellt sind. Diese Anordnung ist in Abbildung 3.1 zu sehen. Die Kommunikation findet über eine 5D-Torus-Netzwerk-Topologie statt. Mit dieser Architektur kommt JUQUEEN auf einen Linpack[2] Wert

[1] http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/JUQUEEN_node.html
[2] http://www.top500.org/project/linpack/

von 5.0 Petaflops und eine theoretische Peak Performance von 5.9 Petaflops. Mit dieser Leistung ist das System zur Zeit (November 2014) auf dem achten Platz der Top500 Liste[3] der weltweit schnellsten Supercomputer gelistet.

Abbildung 3.1.: Der Supercomputer JUQUEEN in der Rechnerhalle des Jülich Supercomputing Centres. Quelle: Forschungszentrum Jülich

3.2. Installation

Dem Benutzer werden neben der manuellen Installation des Quellcodes auch Binärpakete für verschiedene Betriebssysteme wie unter anderem Debian, Ubuntu und Mac OS X angeboten. Die Installation auf der Workstation mit einem OpenSuse Betriebssystem und auf dem Supercomputer wurde jeweils manuell über den Quellcode vorgenommen. Insbesondere ist hierbei wichtig, dass alle Komponenten eine konsistente Versionsnummer haben und auch exakt derselbe Quellcode auf beiden Systemen installiert wurde. Ansonsten kann es zu Problemen kommen, da die auf der Workstation erstellten C++ Header Dateien nicht mit dem erwarteten Format auf dem Supercomputer übereinstimmen.
Der Anwendungsquellcode wurde zunächst auf der Workstation entwickelt, wozu alle in Kapitel 2 vorgestellten Komponenten installiert worden sind. Da die Module UFL, FIAT, Instant und FFC Python Programme sind, die dazu dienen, die im UFL Format

[3] http://www.top500.org/

definierte Variationsformulierung in eine C++ Header Datei zu übersetzen, war es lediglich erforderlich, DOLFIN und UFC auf dem Supercomputer zu installiert. Diese C++ Header Dateien wurden im Laufe der Entwicklung für die jeweilige Anwendung auf der Workstation erstellt. Diese Version ist bereits für hoch paralleles Rechnen geeignet, sodass es ausreicht, diese Dateien auf dem Supercomputer für die Anwendung bereit zu stellen.
Zur Installation war der Standard IBM XL Compiler[4] nicht geeignet, da DOLFIN C++11 Elemente enthält und diese nicht unterstützt werden. Auf Grund dessen wurde hier der clang[5] Compiler eingesetzt. Dies hat wiederum dazu geführt, dass viele mit dem XL Compiler bereits installierte Bibliotheken nicht verwendet werden konnten. Somit mussten diese Bibliotheken ebenfalls neu compiliert werden. Die Tabelle 3.1 zeigt die mit dem clang Compiler erstellten Bibliotheken, die zur Durchführung der Simulationen benötigt wurden.

Bibliotheksname	**Versionsnummer**
DOLFIN	1.4.0
UFC	1.4.0
Boost	1.55.0
ParMetis	4.0.2
PETSc	3.5.1
LAPACK	3.4.2
HDF5	1.8.12
Eigen3	3.2.90
libXML	2.9.1
zlib	1.2.8

Tabelle 3.1.: Auf JUQUEEN installierte Bibliotheken

[4]http://www-03.ibm.com/software/products/en/xlcc+forbluegene
[5]http://clang.llvm.org/

4. Anwendungen und deren Skalierungsverhalten

Zum Testen der FEniCS Installation auf JUQUEEN wurden verschiedene Anwendungen in der Programmiersprache C++ implementiert. Für den letzten Testfall wurde die komplexe Geometrie einer Düse mit dem Gittergenerator Gmsh[1] erzeugt. Die restlichen Gitter sind Einheitsquadrate oder Einheitswürfel, die mit den DOLFIN built-in Funktionen `UnitSquareMesh` und `UnitCubeMesh` erstellt wurden. Es wurde mit der Poisson-Gleichung ein stationäres Problem und mit der Navier-Stokes-Gleichung mehrere instationäre Probleme gelöst.
Zur Untersuchung des Skalierungsverhaltens wurde jede Anwendung mit verschiedenen Anzahlen von MPI Prozessen auf dem Supercomputer gestartet. Für die Zeitmessungen wurde jede Anwendung mit vier DOLFIN `Timern`[2] instrumentiert. Diese messen das Preprocessing, die Berechnung, das Postprocessing und die gesamte Laufzeit. Zusätzlich verwendet DOLFIN auch intern `Timer`, um die Zeit, die in bestimmten Programmabschnitten während einer Simulation verbracht wird, zu messen. Die so gesammelten Daten können dann pro Prozess über die Funktion `list_timings` ausgegeben werden. Zur Auswertung wurde ein Python Skript geschrieben, welches diese Daten in Tabellen Form und in Boxplots darstellt.

Erläuterung der Skalierungsbilder

Die in den folgenden Kapiteln gezeigten Skalierungsbilder der einzelnen Testfälle beinhalten die gemessenen Zeiten der einzelnen `Timer`. Diese Zeiten werden zur besseren Darstellung auf Simulationen pro Stunde ($\frac{3600}{\text{gemessene Zeit in Sekunden}}$) normiert und in Abhängigkeit der Anzahl der MPI Prozesse aufgetragen. Pro Bild werden immer mehrere Linien dargestellt. Alle Punkte, die sich auf derselben Linie befinden sind Messergebnisse, die mit derselben Nodeboard-Größe, aber unterschiedlicher Anzahl an `ranks-per-node` erhalten wurden. Somit spiegeln die verschiedenen Linien auch

[1] http://geuz.org/gmsh/
[2] http://fenicsproject.org/documentation/dolfin/dev/cpp/programmers-reference/common/Timer.html

die Kosten für eine Simulation wieder, da diese über die Anzahl der verwendeten Nodeboards abgerechnet wird. Aus wirtschaftlicher Sicht ist es deswegen sinnvoll, eine kleinere Anzahl an Nodeboards für die Simulation zu verwenden, auch wenn sich bei einer höheren Anzahl die Laufzeit gering verbessert.

4.1. Poisson-Gleichung

Als einfachstes Beispiel wurde die Poisson-Gleichung auf einem Einheitsquadrat mit Dirichlet- und Neumann-Randwerten gelöst. Die mathematische Formulierung dieses Problems ist in Formel 4.1 dargestellt.

$$
\begin{array}{rcll}
-\Delta u & = & 0 & \text{in } \Omega \\
u & = & 1 & \text{für } x \in [0.7, 0.8] \ \wedge \ y = 0 \\
u & = & 0 & \text{für } x \in [0.2, 0.3] \ \wedge \ y = 0 \\
\nabla u \cdot \mathbf{n} & = & 0 & \text{auf } \partial\Omega \\
\Omega & = & [0,1] \times [0,1] &
\end{array}
$$

Formel 4.1.: Definition des Poisson-Problems auf dem Einheitsquadrat

Wie schon bei der Poisson-Gleichung beim Einstiegsbeispiel im Kapitel 2.2.1 werden auch hier zur Diskretisierung Lagrange-Dreiecke erster Ordnung verwendet. Da sich die Variationsformulierung im Vergleich zum Einstiegsbeispiel ebenfalls nicht ändert, stellt Listing 4.1 die Input Datei für den Form Compiler dar.
Das Ergebnis der Rechnung zeigt Abbildung 4.1.

```
1 element = FiniteElement("Lagrange", triangle, 1)
2
3 u = TrialFunction(element)
4 v = TestFunction(element)
5 f = Coefficient(element)
6
7 a = inner(grad(u), grad(v))*dx
8 L = f*v*dx
```

Listing 4.1: Variationsformulierung im UFL Format für die Poisson-Gleichung

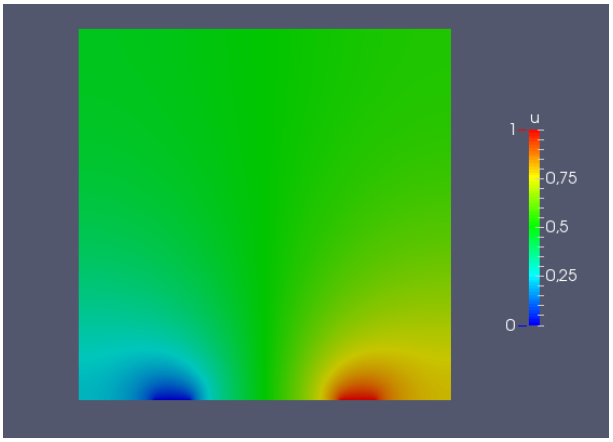

Abbildung 4.1.: Darstellung der Lösung des Poisson-Problems

4.1.1. Skalierungsverhalten des Poisson-Problems

Zur Untersuchung des Skalierungsverhaltens wurden Simulationen mit Gittern von 20 Millionen, 40 Millionen und 80 Millionen Gitterzellen auf JUQUEEN durchgeführt. Da die Rechenzeit auf JUQUEEN über die Anzahl der verwendeten Nodeboards abgerechnet wird, wurden für jede der drei Gittergrößen mehrere Simulationen mit unterschiedlicher Anzahl an Nodeboards und MPI Prozessen gerechnet. Eine Simulation wurde dazu in drei Bereiche eingeteilt, die jeweils durch eine Zeitmessung separat gemessen wurden. Der erste Bereich ist das Preprocessing, dem vor allem das Einlesen des Gitters und das Erstellen des Funktionsraumes zugeteilt sind. Darauf folgt der zweite Bereich, der die Zeit für die Konstruktion der Matrizen und der Berechnung der Lösung misst. In den abschließenden dritten Bereich fällt das Schreiben der Ergebnisse in eine Datei. Zusätzlich wird die gesamte Laufzeit einer Simulation noch bestimmt.
Die erzielten Messergebnisse werden in den folgenden Abschnitten dargestellt.

20 Millionen Gitterzellen

Auch beim kleinsten Beispiel mit 20 Millionen Gitterzellen ist das Hauptproblem von FEniCS deutlich sichtbar. Wie in Abbildung 4.2 zu erkennen ist, nimmt die I/O-Performance mit zunehmender Anzahl Prozesse ab.
Im Gegensatz dazu ist in Abbildung 4.3 zu erkennen, dass das Laufzeitverhalten von FEniCS für die Berechnung der Lösung mit zunehmender Anzahl Prozesse immer besser wird. Allerdings ist es nicht sinnvoll, mehr als 32 Nodes, beziehungsweise ein Nodeboard, zu verwenden, da ab mehr als 2048 MPI Prozessen die Dauer der Berechnung wieder zunimmt.

Abbildung 4.2.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für das Preprocessing (oben) und das Postprocessing (unten) von Simulationen der Poisson-Gleichung mit 20 Millionen Gitterzellen.

Abbildung 4.3 zeigt zudem ein untypisches Verhalten. Bei Betrachtung der blauen Linie stellt sich ab 256 MPI Prozessen ein Sättigungsverhalten ein und trotzdem steigt die Linie ab 1024 MPI Prozessen noch einmal sehr stark an. Dieses Verhalten tritt bei allen Linien zwischen den beiden letzten Punkten auf. Dieses Merkmal ist auch bei anderen Testfällen aufgefallen und wird in Kapitel 4.2.1 näher beschrieben.

In Hinblick auf die gesamte Laufzeit, die in Abbildung 4.4 dargestellt ist, lässt sich festhalten, dass das günstigste und effektivste Ergebnis mit einem Nodeboard und 2048 MPI Prozessen erreicht wird. Jeweils die zweiten Punkte der roten und der

grünen Linien haben dasselbe Laufzeitverhalten, kosten dem Benutzer aber deutlich mehr, da hier vier beziehungsweise acht Nodeboards verwendet wurden. Zudem kann man in Abbildung 4.4 an den fallenden Linien erkennen, dass ab einschließlich zwei Nodeboards der Zeitbedarf für das I/O in einer Anwendung den Zeitbedarf für die Berechnung deutlich übersteigt. Somit steigt die gesamte Laufzeit mit zunehmender Anzahl MPI Prozesse an und die erreichten Simulationen pro Stunde nehmen ab.

Abbildung 4.3.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für das Berechnen der Lösung von Simulationen der Poisson-Gleichung mit 20 Millionen Gitterzellen.

Abbildung 4.4.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für die gesamte Laufzeit von Simulationen der Poisson-Gleichung mit 20 Millionen Gitterzellen.

40 Millionen Gitterzellen

Die Zeitmessungen für die Simulationen mit 40 Millionen Gitterzellen zeigen im Wesentlichen dasselbe Verhalten wie die Messergebnisse mit 20 Millionen Gitterzellen. Mit zunehmender Anzahl an MPI Prozessen nimmt die I/O-Performance ab (vergleiche Abbildung 4.5).

Obwohl sich zunächst wieder ab einer Größe von 512 beziehungsweise 1024 MPI Prozessen ein Sättigungsverhalten einstellt, steigt die erreichte Anzahl an Simulationen pro Stunde bei einer Verdoppelung von 32 auf 64 `ranks-per-node` noch einmal an, wie Abbildung 4.6 zeigt.
Für Simulationen mit mehr als einem Nodeboard und mehr als 512 MPI Prozessen zeigt Abbildung 4.7, dass der Zeitbedarf des I/O das Verhalten der gesamten Laufzeit dominiert und damit insgesamt weniger Simulationen pro Stunde erreicht werden können. Auch hier wird das schnellste und kostengünstigste Ergebnis mit einem Nodeboard und 64 `ranks-per-node` erzielt.

Abbildung 4.5.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für das Preprocessing (oben) und das Postprocessing (unten) von Simulationen der Poisson-Gleichung mit 40 Millionen Gitterzellen.

Abbildung 4.6.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für das Berechnen der Lösung von Simulationen der Poisson-Gleichung mit 40 Millionen Gitterzellen.

Abbildung 4.7.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für die gesamte Laufzeit von Simulationen der Poisson-Gleichung mit 40 Millionen Gitterzellen.

80 Millionen Gitterzellen

Die Messergebnisse für die Simulationen mit 80 Millionen Gitterzellen zeigen denselben Verlauf wie schon bei den kleineren Gittergrößen zu sehen war.
Das I/O-Verhalten zeigt Abbildung 4.8. In Abbildung 4.9 werden die Simulationen pro Stunde für die Berechnung der Lösung dargestellt und Abbildung 4.10 veranschaulicht die gesamte Laufzeit.

Abbildung 4.8.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für das Preprocessing (oben) und das Postprocessing (unten) von Simulationen der Poisson-Gleichung mit 80 Millionen Gitterzellen.

Abbildung 4.9.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für das Berechnen der Lösung von Simulationen der Poisson-Gleichung mit 80 Millionen Gitterzellen.

Abbildung 4.10.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für die gesamte Laufzeit von Simulationen der Poisson-Gleichung mit 80 Millionen Gitterzellen.

4.2. Lid-Driven Cavity Problem in 2D und 3D

Für das sogenannte Lid-Driven Cavity Problem wird die Navier-Stokes-Gleichung in einem Einheitsquadrat und einem Einheitswürfel gelöst. Dazu wird am oberen Rand, beziehungsweise der oberen Fläche, die Geschwindigkeit $\mathbf{u}$ in x-Richtung auf 1 und für den restlichen Rand gleich 0 gesetzt.
In Abhängigkeit von der Viskosität ν und der gerechneten Zeitspanne bilden sich im Inneren des Gebietes Wirbel. Zur Untersuchung des Skalierungsverhaltens wurde allerdings nur ein Zeitschritt gerechnet. Weitere physikalische Größen, die in die Gleichung mit eingehen, sind der Druck p und die äußere Kraftdichte $\mathbf{f}$.
Als Lösungsalgorithmus wurde die `Chorins Projektions Methode` [10] gewählt, die auf Basis von gemischten Elementen für die Geschwindigkeit und den Druck arbeitet. Für die Geschwindigkeit werden Lagrange-Dreiecke zweiter Ordnung verwendet und für den Druck aus Stabilitätsgründen eine Ordnung weniger als für die Geschwindigkeit. Zur Berechnung der Lösung wird in jedem Zeitschritt zunächst eine vorläufige Geschwindigkeit unter Vernachlässigung des Druckes berechnet. Danach wird der korrigierte Druckterm ermittelt und schließlich die Geschwindigkeit auf den für diesen Zeitschritt berechneten Druckterm angepasst.

4.2.1. 2-dimensionaler Fall

Die mathematische Formulierung des Lid-Driven Cavity Problems für den 2-dimensionalen Fall ist in Formel 4.2 dargestellt.

$$
\begin{array}{rcll}
\dot{\mathbf{u}} + \mathbf{u} \cdot \nabla \mathbf{u} - \nu \Delta \mathbf{u} + \nabla p &=& \mathbf{f} & \text{in } \Omega \times (0, T] \\
\nabla \cdot \mathbf{u} &=& 0 & \text{in } \Omega \times (0, T] \\
\mathbf{u} &=& (1,0)^\top & \text{auf } \partial\Omega \cap \{\mathrm{y} = 1\} \\
\mathbf{u} &=& (0,0)^\top & \text{auf } \partial\Omega \cap \{\mathrm{y} < 1\}
\end{array}
$$

Formel 4.2.: Definition des Lid-Driven Cavity Problems auf dem Einheitsquadrat

Zur Durchführung der Simulationen wurde ein Einheitsquadrat mit 40 Millionen Gitterzellen und mit 80 Millionen Gitterzellen erstellt. Die berechnete Lösung für eine höhere Anzahl an Zeitschritten auf einem groben Gitter ist in Abbildung 4.11 dargestellt.

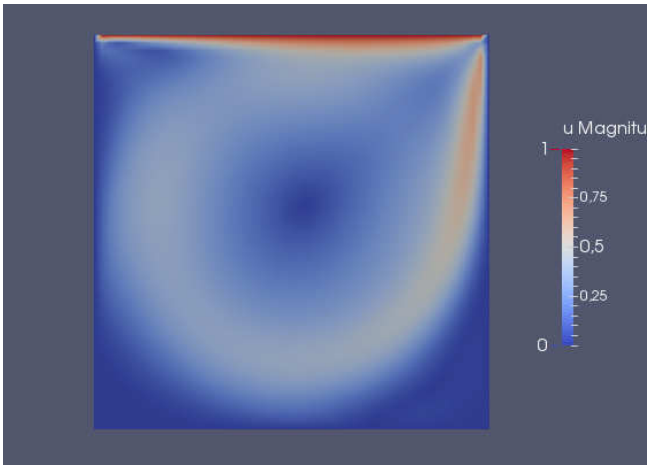

Abbildung 4.11.: Darstellung der Lösung des Lid-Driven Cavity Problems auf dem Einheitsquadrat

40 Millionen Gitterzellen

Die bei der Poisson-Gleichung erkannten Trends und Merkmale lassen sich auch in diesem Anwendungsfall wiederfinden. So auch die immer schlechter werdende I/O-Performance mit steigender Anzahl MPI Prozesse, wie Abbildung 4.12 zeigt. Allerdings gibt es auch hier wieder zwei extreme Ausreißer beim Preprocessing. Entgegen der fallenden Anzahl an Simulationen pro Stunde, steigen die gerechneten Fälle mit zwei (gelbe Linie), beziehungsweise vier (rote Linie) Nodeboards und 64 `ranks-per-node` noch einmal stark an.
Die Ursache für diese Unterschiede kann mit Hilfe der internen Zeitmessungen von DOLFIN, die die Bearbeitungsdauer für einen bestimmten Programmteil messen, näher bestimmt werden. Dazu wurden die gemessenen Zeit der einzelnen `Timer` pro MPI Prozess in einem Simulationslauf in eine Datei geschrieben und anschließend von eigenen Python-Skripten bildlich aufbereitet. Daraus sind für jeden `Timer` Boxplots entstanden, die die Verteilung der Bearbeitungsdauer über alle MPI Prozesse widerspiegeln.
Die dargestellten Zeiten in der Abbildung 4.13 wurden mit einer JUQUEEN Partitionsgröße von zwei Nodeboards und 32 `ranks-per-node`, also 2048 MPI Prozessen, gemessen. Hierbei handelt es sich um den vorletzten Punkt auf der gelben Linie in den Abbildungen 4.12, 4.15 und 4.16. Im Vergleich dazu zeigt Abbildung 4.14 die Zeiten, die mit einer JUQUEEN Partitionsgröße von zwei Nodeboards und 64 `ranks-per-node`, also 4096 MPI Prozessen, gemessen wurden. Diese Zeiten fließen in den letzten Punkt der gelben Linie in den Abbildungen 4.12, 4.15 und 4.16 ein.

Abbildung 4.12.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für das Preprocessing (oben) und das Postprocessing (unten) von Simulationen des Lid-Driven Cavity Problems auf einem Einheitsquadrat mit 40 Millionen Gitterzellen.

Wie bereits erwähnt, wurden die Simulationsprogramme in drei Teile eingeteilt: das Preprocessing, das Berechnen der Lösung und das Postprocessing. Diese drei Bereiche werden durch die beiden dicken, schwarzen, senkrechten Linien in den Abbildungen 4.13 und 4.14 markiert. Alle `Timer`, die sich links der linken Linie befinden, messen DOLFIN interne Programmabschnitte, die dem Preprocessing zuzuordnen sind. Diese beeinflussen folglich die Lage der Punkte im oberen Bild der Abbildung 4.12.
Beim Vergleich der beiden Boxplot-Abbildungen 4.13 und 4.14 fällt insbesondere auf, dass ein großer Unterschied zwischen den gemessenen Zeiten des `Timers` mit dem Namen `Init dofmap` besteht. Dies ist kein Einzelfall, sondern die Regel bei allen Testfällen, bei denen im Preprocessing-Bereich ein großer Sprung zwischen den Simulationen mit 32 und 64 `ranks-per-node` festzustellen ist. Weitere Boxplot-Abbildungen von anderen Testfällen sind in Anhang A zu finden.
Der Versuch, per Hand im Quellcode weitere Informationen über die von den verschiedenen `Timern` gemessenen Programmabschnitte zu erhalten, hat die Vermutung aufkommen lassen, dass einzelne `Timer` komplett in einem anderen `Timer` enthalten sein könnten. Damit würden bestimmte Abschnitte doppelt gemessen. Um diese Vermutung genauer zu untersuchen, müsste DOLFIN mit einem Profiling Tool wie beispielsweise HPCToolkit[3] instrumentiert werden. Auf Grund des höheren Zeitaufwandes konnte dies im Rahmen dieser Arbeit nicht mehr durchgeführt werden.

[3] http://hpctoolkit.org/

Abbildung 4.13.: Boxplots der gemessenen Rechenzeit interner DOLFIN `Timer` zur Darstellung der Variation über alle MPI Prozesse.
(40 Millionen Lid-Driven Cavity, zwei Nodeboards / 32 `ranks-per-node`)

Abbildung 4.14.: Boxplots der gemessenen Rechenzeit interner DOLFIN `Timer` zur Darstellung der Variation über alle MPI Prozesse. (40 Millionen Lid-Driven Cavity, zwei Nodeboards / 64 `ranks-per-node`)

Nachdem bei den I/O Zeitmessungen ein wiederholendes Merkmal festgestellt werden konnte, zeigt auch der Verlauf der Messungen für die Berechnung der Lösung ein ähnliches Verhalten wie beim Poisson-Beispiel.
Zunächst steigen alle Linien der unterschiedlichen Nodeboard-Konfigurationen in Abbildung 4.15 mit zunehmender Anzahl MPI Prozesse an, ehe sie leicht abflachen. Bei der roten und grünen Linie nehmen die erreichten Simulationen pro Stunde sogar ab, bevor sich dann die Performance für eine Konfiguration von 64 `ranks-per-node` auf allen Nodeboardgrößen noch einmal stark verbessert.
Der Programmabschnitt, der diese großen Unterschiede verursacht, kann wieder mit Hilfe der Boxplot-Abbildungen 4.13 und 4.14 näher bestimmt werden. Beim Vergleich fallen insbesondere die `Timer` mit den Namen `Build sparsity` und `Assemble cells` auf, da hier die größten Zeitunterschiede zwischen den beiden Konfigurationen zu sehen sind.
Wenn nun alle Testfälle betrachtet werden, bei denen ein so ausgeprägter Zeitgewinn zwischen den Simulationen mit 32 und 64 `ranks-per-node` besteht, dann fällt auf, dass vor allem der Bereich, den der `Build sparsity Timer` misst, entscheidend ist.

Abbildung 4.15.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für das Berechnen der Lösung von Simulationen des Lid-Driven Cavity Problems auf einem Einheitsquadrat mit 40 Millionen Gitterzellen.

Diese deutlichen Sprüngen beim Preprocessing und der Berechnung der Lösung spiegeln sich auch in der gesamten Laufzeit wider, wie Abbildung 4.16 zeigt. Bishin zu 512 MPI Prozessen ist an den steigenden Linien zu erkennen, dass der Zeitaufwand für die Berechnung der Lösung das Laufzeitverhalten bestimmt. Mit größer werdender

Partitionsgröße nimmt die Anzahl erreichter Simulationen pro Stunde immer mehr ab, was darauf zurückzuführen ist, dass die Dauer für den I/O die Simulationen dominiert. Allerdings sind die erwähnten großen Sprünge dafür verantwortlich, dass sich die Laufzeit-Performance für zwei, beziehungsweise vier Nodeboards noch einmal stark verbessert für 4096 beziehungsweise 8192 MPI Prozesse.

Abbildung 4.16.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für die gesamte Laufzeit von Simulationen des Lid-Driven Cavity Problems auf einem Einheitsquadrat mit 40 Millionen Gitterzellen.

Da zur Durchführung der Benchmarks lediglich ein Zeitschritt gerechnet wurde, ist bei längeren Simulationsläufen davon auszugehen, dass sich vor allem der Verlauf der Linien in Abbildung 4.16, die die gesamte Laufzeit widerspiegeln, ändert. Mit steigender Dauer der Berechnung wird der Einfluss des I/O auf die komplette Simulation immer geringer. Somit ist zu erwarten, dass der lokale Hochpunkt bei 512 MPI Prozessen zu einer höheren Anzahl an MPI Prozessen bewegt und die absolute Spitze bei 4096 MPI Prozessen, also der Simulationskonfiguration mit zwei Nodeboards und 64 `ranks-per-node`, noch deutlicher ausgeprägt ist.

80 Millionen Gitterzellen

Die Verdoppelung der Problemgröße von 40 auf 80 Millionen Gitterzellen unterstützt die bereits gewonnen Erkenntnisse.
Das in Abbildung 4.17 dargestellte I/O-Laufzeitverhalten wird unabhängig von der Anzahl der Nodeboards mit zunehmender Anzahl MPI Prozesse immer schlechter.

Abbildung 4.17.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für das Preprocessing (oben) und das Postprocessing (unten) von Simulationen des Lid-Driven Cavity Problems auf einem Einheitsquadrat mit 80 Millionen Gitterzellen.

Im Gegensatz dazu zeigt Abbildung 4.18 das sehr gute Skalierungsverhalten der Zeitmessungen für die Berechnung der Lösung. Des Weiteren ist auch das Sprungmerk-

mal in diesem Testfall vorhanden. So ist ein deutlicher Performance-Gewinn beim Übergang von 32 auf 64 `ranks-per-node` bei Simulationen mit vier (rote Linie) und acht (grüne Linie) Nodeboards zu erkennen. Der Hauptgrund dafür ist auch hier wieder der Programmabschnitt, der vom `Timer` mit dem Namen `Build sparsity` gemessen wird.

Abbildung 4.18.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für das Berechnen der Lösung von Simulationen des Lid-Driven Cavity Problems auf einem Einheitsquadrat mit 80 Millionen Gitterzellen.

Im Hinblick auf die gesamte Laufzeit in Abbildung 4.19 wird sich der lokale Hochpunkt bei 512 MPI Prozessen noch weiter nach rechts verschieben, wenn mehr als nur ein Zeitschritt simuliert wird. Damit würde das Skalierungsverhalten einer längeren Simulation noch besser aussehen. Die schnellsten und kostengünstigsten Ergebnisse werden dann mit einer Konfiguration von vier Nodeboards und 64 `ranks-per-node`, also insgesamt 8192 MPI Prozessen, erreicht.

Abbildung 4.19.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für die gesamte Laufzeit von Simulationen des Lid-Driven Cavity Problems auf einem Einheitsquadrat mit 80 Millionen Gitterzellen.

4.2.2. 3-dimensionaler Fall

Im 3-dimensionalen Fall wird die Navier-Stokes-Gleichung auf einem Einheitswürfel gelöst und dazu die Geschwindigkeit in x-Richtung auf der oberen Fläche des Würfels gleich eins gesetzt. Die mathematische Formulierung des Lid-Driven Cavity Problems für den 3-dimensionalen Fall ist in Formel 4.3 dargestellt.

$$
\begin{array}{rcll}
\dot{\mathbf{u}} + \mathbf{u} \cdot \nabla \mathbf{u} - \nu \Delta \mathbf{u} + \nabla p & = & \mathbf{f} & \text{in } \Omega \times (0, T] \\
\nabla \cdot \mathbf{u} & = & 0 & \text{in } \Omega \times (0, T] \\
\mathbf{u} & = & (1, 0, 0)^\top & \text{auf } \partial\Omega \cap \{\mathrm{y} = 1\} \\
\mathbf{u} & = & (0, 0, 0)^\top & \text{auf } \partial\Omega \cap \{\mathrm{y} < 1\}
\end{array}
$$

Formel 4.3.: Definition des Lid-Driven Cavity Problems auf dem Einheitswürfel

Die berechnete Lösung für eine höhere Anzahl an Zeitschritten auf einem gröberen Gitter ist in Abbildung 4.11 zu sehen.
Zur Durchführung der Simulationen auf JUQUEEN wurde ein Einheitswürfel mit 40 Millionen Gitterzellen und mit 80 Millionen Gitterzellen erstellt. Während der Programmausführung sind öfters Speicherprobleme aufgetreten, sodass die Anzahl der Messpunkte im Vergleich zu allen anderen Testbeispielen geringer ausfällt. In folgenden Arbeiten kann untersucht werden, ob es für dieses Beispiel mathematisch effizi-

enter ist, einen anderen Lösungsalgorithmus als die `Chorin Projektions Methode` zu wählen, oder ob ein Wechsel der Lösungsmethode für die Gleichungssysteme die Speicherprobleme behebt.

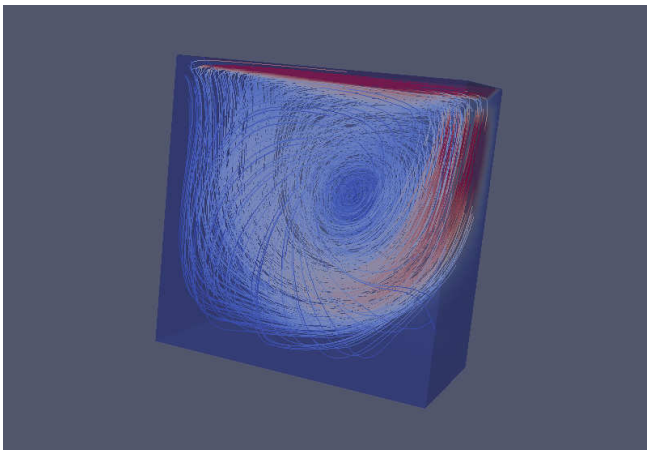

Abbildung 4.20.: Beispielhafte Darstellung der Lösung des Lid-Driven Cavity Problems auf dem Einheitswürfel

40 Millionen Gitterzellen

Trotz der geringen Anzahl an Messergebnissen, sind auch beim Anwendungsfall von 40 Millionen Gitterzellen die bisher erkannten Merkmale anderer Anwendungsfälle erkennbar. So zeigt Abbildung 4.21, dass die I/O-Performance mit zunehmender Anzahl von MPI Prozessen immer schlechter wird.
Ein gegenläufiger Trend zeichnet sich bei den Zeitmessungen für die Berechnung der Lösung in Abbildung 4.22 ab. Hier lässt sich nur vermuten, dass die Simulationen pro Stunde zunächst ansteigen und ab einer Grenze bei 4096 MPI Prozessen nicht mehr signifikant zunehmen.
Über das gesamte Laufzeitverhalten kann an dieser Stelle auch nur gesagt werden, dass das I/O bei Berechnung eines Zeitschrittes die Simulationsdauer dominiert und die Linien in Abbildung 4.23 stark abfallen.

Abbildung 4.21.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für das Preprocessing (oben) und das Postprocessing (unten) von Simulationen des Lid-Driven Cavity Problems auf einem Einheitswürfel mit 40 Millionen Gitterzellen.

Abbildung 4.22.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für das Berechnen der Lösung von Simulationen des Lid-Driven Cavity Problems auf einem Einheitswürfel mit 40 Millionen Gitterzellen.

Abbildung 4.23.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für die gesamte Laufzeit von Simulationen des Lid-Driven Cavity Problems auf einem Einheitswürfel mit 40 Millionen Gitterzellen.

80 Millionen Gitterzellen

Auch bei der Diskretisierung des Einheitswürfels mit 80 Millionen Gitterzellen wird das I/O-Verhalten mit zunehmender Anzahl an MPI Prozessen immer schlechter, wie Abbildung 4.24 zeigt.

Abbildung 4.24.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für das Preprocessing (oben) und das Postprocessing (unten) von Simulationen des Lid-Driven Cavity Problems auf einem Einheitswürfel mit 80 Millionen Gitterzellen.

Abbildung 4.25.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für das Berechnen der Lösung von Simulationen des Lid-Driven Cavity Problems auf einem Einheitswürfel mit 80 Millionen Gitterzellen.

Im Vergleich dazu stellt Abbildung 4.25 das außerordentlich gute Skalierungsverhalten des Berechnungsanteils dar.
Bei Betrachtung der gesamten Laufzeit für einen Zeitschritt in Abbildung 4.26 lässt sich erkennen, dass bis 2048 MPI Prozesse die Dauer der Berechnung der Lösung den Hauptanteil ausmacht. Für eine größere Anzahl Prozesse dominiert das I/O wieder, sodass die gesamte Performance wieder deutlich schlechter wird. Unter Berücksichtigung der Tatsache, dass in einem Produktionslauf sehr viele Zeitschritte gerechnet werden, ist davon auszugehen, dass mit einer größeren Anzahl als 2048 Prozesse insgesamt schneller ein Ergebnis erreicht werden kann.

Abbildung 4.26.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für die gesamte Laufzeit von Simulationen des Lid-Driven Cavity Problems auf einem Einheitswürfel mit 80 Millionen Gitterzellen.

4.3. Navier-Stokes-Gleichung im L-Gebiet

Für einen weiteren Anwendungsfall wurde ein FEniCS Demo[4] Beispiel abgeändert. Es wird die inkompressible Navier-Stokes-Gleichung auf einem Gebiet in L-Form unter Verwendung von `Chorins Projektions Methode` gelöst. Am Einlass, am oberen Rand des L-Gebietes ($y = 1$), ist ein oszillierender Druck vorgegeben. Der Druck im Auslass am rechten Rand bei $x = 1$ ist auf null gesetzt. Die Formulierung des Problems ist in Formel 4.4 dargestellt.

$$
\begin{array}{rcll}
\dot{\mathbf{u}} + \mathbf{u} \cdot \nabla \mathbf{u} - \nu \Delta \mathbf{u} + \nabla p & = & \mathbf{f} & \text{in } \Omega \times (0, T] \\
\nabla \cdot \mathbf{u} & = & 0 & \text{in } \Omega \times (0, T] \\
p_{\text{in}}(t) & = & \sin(3t) & \text{für } y = 1 \\
p_{\text{out}} & = & 0 & \text{für } x = 1 \\
u & = & (0, 0)^{\top} & \text{für } \partial\Omega \setminus (\{x = 1, y\} \cup \{x, y = 1\})
\end{array}
$$

Formel 4.4.: Definition des Navier-Stokes-Problems auf einem L-Gebiet

[4] http://fenicsproject.org/documentation/dolfin/dev/cpp/demo/documented/navier-stokes/cpp/documentation.html

Zur Durchführung des Benchmarks, wurde auch in diesem Anwendungsfall ein Zeitschritt auf einem Gitter mit etwa 60 Millionen Elementen gerechnet. Die berechnete Lösung für die Geschwindigkeit und den Druck auf einem gröberen Gitter für eine höhere Anzahl an Zeitschritten ist in Abbildung 4.27 zu sehen.

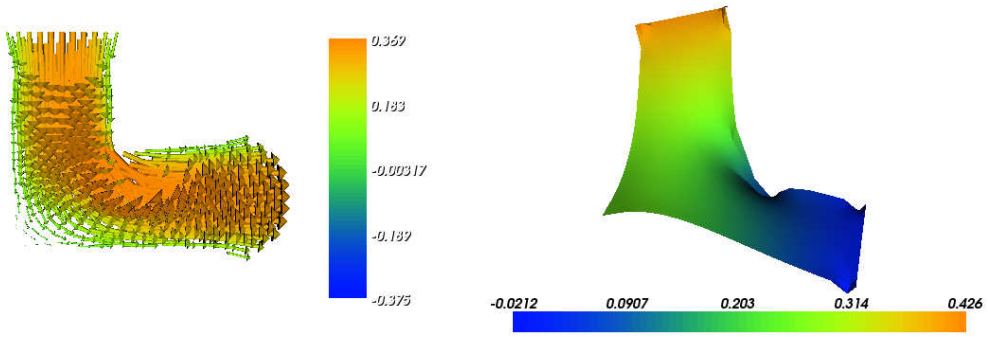

Abbildung 4.27.: Darstellung der Lösung der Geschwindigkeit (links) und des Druckes (rechts) in Wert und Farbe für die Navier-Stokes-Gleichung auf einem L-Gebiet

4.3.1. Skalierungsverhalten für das L-Gebiet mit 60 Millionen Gitterzellen

Die Messergebnisse dieses Testbeispiels zeigen dieselben Merkmale auf, wie sie auch schon in den bisher diskutierten Fällen aufgetreten sind.
So wird das I/O-Verhalten in Abbildung 4.28 mit zunehmender Anzahl MPI Prozesse mit einer Ausnahme immer schlechter. Der einzige Ausreißer ist die Simulation mit zwei Nodeboards und 64 `ranks-per-node` (letzter Punkt der gelben Linie). Bei Betrachtung des Boxplots dieser Simulation (siehe Abbildungen A.1 und A.2 im Anhang) ist zu erkennen, dass der Programmabschnitt, der vom `Timer` mit dem Namen `Init dofmap` gemessen wird, deutlich schneller bearbeitet wird, als bei anderen Systemkonfigurationen.
Die Zeitmessungen für die Berechnung der Lösung sind in Abbildung 4.29 dargestellt und zeigen ein sehr gutes Skalierungsverhalten. Auch hier sind wieder die Sprünge bei den Simulationen mit vier und acht Nodeboards von 32 auf 64 `ranks-per-node` erkennbar. Die Boxplots (siehe Abbildung A.3 und A.4 im Anhang) der entsprechenden Simulationen stellen heraus, dass der `Build sparsity Timer` den Hauptanteil dieses Laufzeitunterschiedes ausmacht.
Insgesamt ist an den zunächst steigenden Linien in Abbildung 4.30 festzumachen, dass bis einschließlich 512 MPI Prozesse der Berechnungsteil einer Simulation den zeitlich

größten Anteil einnimmt. Bei einer größeren Anzahl MPI Prozesse nehmen die erreichten Simulationen pro Stunde immer weiter ab. Dieses Verhalten ist auf die schlechte I/O-Performance zurückzuführen. Der beobachtete Ausreißer in den Zeitmessungen für zwei Nodeboards mit 64 `ranks-per-node` ist deutlich ausgeprägt, sodass es sich hierbei um die schnellste Konfiguration handelt. Es ist jedoch zu erwarten, dass sich der lokale Hochpunkt bei 512 MPI Prozessen zu einem höheren Wert verschiebt, wenn mehr als ein Zeitschritt gerechnet wird.

Abbildung 4.28.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für das Preprocessing (oben) und das Postprocessing (unten) von Simulationen der Navier-Stokes-Gleichung auf einem L-Gebiet mit 60 Millionen Gitterzellen.

Abbildung 4.29.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für das Berechnen der Lösung von Simulationen der Navier-Stokes-Gleichung auf einem L-Gebiet mit 60 Millionen Gitterzellen.

Abbildung 4.30.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für die gesamte Laufzeit von Simulationen der Navier-Stokes-Gleichung auf einem L-Gebiet mit 60 Millionen Gitterzellen.

4.4. Navier-Stokes-Gleichung in einer Düse

Der komplexeste der hier untersuchten Anwendungsfälle ist die Navier-Stokes-Gleichung in einer Düse. Die Geometrie der Düse wurde aus einer Studie[5] der U.S. Food and Drug Administration[6] entnommen und mit dem Gittergenerator Gmsh erzeugt. Sie besteht aus vier Elementen: einem zylinderförmigen Einlass mit einem Radius von 0.6 cm, einem konischen Verbindungstück, das sich auf einen Radius von 0.2 cm verengt, einen daran anschließenden zylinderförmigen Hals und einem ebenfalls zylinderförmigen Auslass mit einem Radius von 0.6 cm. Insgesamt besteht das Gitter aus etwa 26 Millionen Elementen. Die errechnete Geschwindigkeit ist in Abbildung 4.31 dargestellt.
Zur Durchführung der Simulationen wurde ein parabolisches Einflussprofil so gewählt, dass im Hals eine Reynoldszahl von 500 vorliegt. Die kinematische Viskosität wurde auf den Wert 0.03314 cm^2/s gesetzt und entspricht damit dem Wert für Blut. Die Formulierung des Problems ist in Formel 4.5 dargestellt.

$$
\begin{aligned}
\dot{\mathbf{u}} + \mathbf{u} \cdot \nabla \mathbf{u} - \nu \Delta \mathbf{u} + \nabla p &= \mathbf{f} && \text{in } \Omega \times (0, T] \\
\nabla \cdot \mathbf{u} &= 0 && \text{in } \Omega \times (0, T] \\
\mathbf{u} &= (0, 0, 0))^\top && \text{für } \partial\Omega \setminus \{x, y, z = -0.088\} \\
\mathbf{u} &= (0, 0, w(x, y))^\top && \text{für } \partial\Omega \cap \{x, y, z = -0.088\} \\
\\
w(x, y) &= 9.2067 \cdot (0.6^2 - x^2 - y^2)/0.6^2
\end{aligned}
$$

Formel 4.5.: Definition des Navier-Stokes-Problems in einer Düse

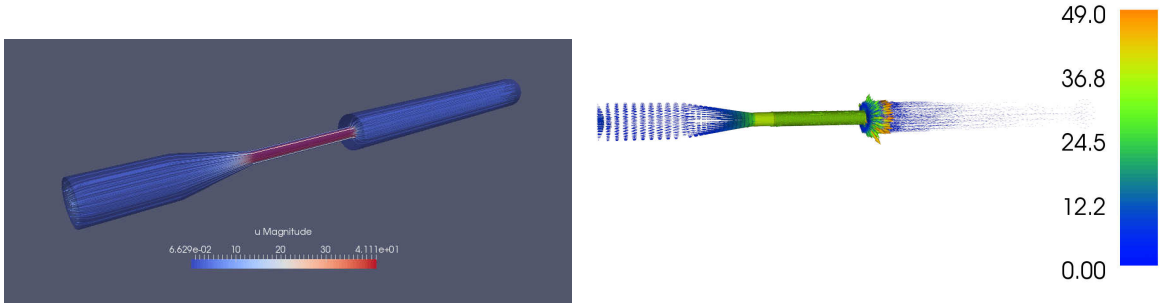

Abbildung 4.31.: Darstellung der Geschwindigkeit mit Hilfe von ParaView (links) und der DOLFIN `plot`-Funktion (rechts) für die Navier-Stokes-Gleichung in einer Düse

[5] http://www.fda.gov/scienceresearch/specialtopics/criticalpathinitiative/spotlightoncpiprojects/ucm149414.htm
[6] http://www.fda.gov/

Der verwendete Lösungsalgorithmus wurde für diese Arbeit aus Houzeaux et al. [5] übernommen. Dieser Algorithmus wurde speziell für die Verwendung auf einem Supercomputer entwickelt. Für die Geschwindigkeit wurden als Basiselemente Lagrange-Tetraeder zweiter Ordnung gewählt und für den Druck Lagrange-Tetraeder erster Ordnung.

4.4.1. Skalierungsverhalten für eine Düse mit 26 Millionen Gitterzellen

Dieser Anwendungsfall zeigt wieder, dass die I/O-Performance von FEniCS verbesserungswürdig ist, da in Abbildung 4.32 zu sehen ist, dass mit zunehmender Anzahl an MPI Prozessen die Simulationen pro Stunde für das Pre- und Postprocessing immer weiter abnehmen. Entgegen diesem Trend wird die Laufzeit bei zwei Nodeboards und 4096 MPI Prozessen noch einmal besser. Dieser Sprung ist ebenfalls auf den `Timer` mit dem Namen `Init dofmap` zurückzuführen.
Abbildung 4.33 zeigt dagegen das sehr gute Skalierungsverhalten des verwendeten Algorithmus zur Berechnung der Lösung. Bis hin zu 4096 MPI Prozesse steigt die Anzahl der Simulationen pro Stunde immer weiter an. In diesem Anwendungsbeispiel sind die bei der `Chorins Projektions Methode` auftretenden Sprünge nicht vorzufinden. Dies lässt darauf schließen, dass die Ursache für diese Sprünge auf Eigenschaften des verwendeten Algorithmus zurückzuführen sind. Die Verwendung eines Profiling Tools an dieser Stelle würde genauere Rückschlüsse zulassen.
Die Messergebnisse der gesamten Laufzeit bilden wieder einen konkaven Bogen, wie Abbildung 4.34 zu sehen ist. Bis zu einem Hochpunkt bei 1024 MPI Prozessen nimmt die Anzahl an Simulationen pro Stunde zu, da in diesem Bereich die Dauer für die Berechnung der Lösung überwiegt. Für eine höhere Anzahl an MPI Prozessen wird die Performance schlechter, da hier die I/O-Dauer deutlich dominiert. Allerdings gibt es eine Ausnahme bei der Simulationskonfiguration von zwei Nodeboards mit 64 `ranks-per-node`, also 4096 MPI Prozessen. Der Sprung im I/O-Verhalten dieser Partitionsgröße macht sich hier bemerkbar, indem, entgegen dem allgemein schlechter werdenden Trend, mehr Simulationen pro Stunde erreicht werden. Auch in diesem Anwendungsfall ist wieder zu erwarten, dass sich bei einer größeren Anzahl berechneten Zeitschritten die Laufzeitkurve verschiebt und die schnellsten Ergebnisse mit mehr als 1024 MPI Prozessen erreicht werden können.

Abbildung 4.32.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für das Preprocessing (oben) und das Postprocessing (unten) von Simulationen der Navier-Stokes-Gleichung in einer Düse mit ungefähr 26 Millionen Gitterzellen.

Abbildung 4.33.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für das Berechnen der Lösung von Simulationen der Navier-Stokes-Gleichung in einer Düse mit ungefähr 26 Millionen Gitterzellen.

Abbildung 4.34.: Auf Simulationen pro Stunde normierte Zeit in Abhängigkeit von der Anzahl der MPI Prozesse für die gesamte Laufzeit von Simulationen der Navier-Stokes-Gleichung in einer Düse mit ungefähr 26 Millionen Gitterzellen.

5. Zusammenfassung und Ausblick

Diese Arbeit hat das finite Elemente Toolkit FEniCS vorgestellt. Es besteht aus einzelnen Komponenten, die zusammen eine flexible, benutzerfreundliche Schnittstelle bereitstellen, um verschiedenste Anwendungsprobleme auf Basis der finiten Elemente Methode effizient lösen zu können.
Ein weiterer großer Vorteil von FEniCS ist, dass es auf vielen Computerarchitekturen und Betriebssystemen läuffähig ist. In dieser Arbeit ist in einem ersten Schritt gezeigt worden, dass FEniCS auch auf dem IBM BlueGene/Q-Supercomputer JUQUEEN im Jülich Supercomputing Centre installiert werden kann. Auf Grund der speziellen Systemarchitektur und da FEniCS eine große Anzahl verschiedenster Bibliotheken verwendet, ist die erste Installation auf JUQUEEN sehr kompliziert gewesen. Im Rahmen dieser Arbeit wurde die Instrumentalisierung des Installationsprozesses sehr gut verstanden, sodass weitere Installationen einen geringeren Aufwand haben werden.
Da der implementierte Anwendungsquellcode sowohl für serielle, als auch parallele Programmausführung ohne Modifikationen verwendet werden kann, wurden verschiedene Testbeispiele zunächst auf einer Workstation entwickelt und anschließend auf dem Supercomputer zur Verfügung gestellt. In einem zweiten Schritt wurden unter Verwendung dieser Beispiele Benchmark-Simulationen auf JUQUEEN durchgeführt.
Unter Berücksichtigung aller Simulationen, haben die Zeitmessungen ergeben, dass die I/O-Performance mit zunehmender Partitionsgröße deutlich schlechter wird. Bei einzelnen Konfigurationen wurden jedoch größere Verbesserungen beobachtet, die eine verbesserte Laufzeit anzeigen. Diese Sprünge werden im Wesentlichen durch den Programmabschnitt verursacht, der von dem internen `Timer` mit dem Namen `Init dofmap` gemessen wird.
Im Gegensatz dazu zeigen die Messergebnisse für die Berechnung der Lösung ein sehr gutes Skalierungsverhalten. Allerdings wurden auch in diesem Bereich unerwartet hohe Sprünge für bestimmte Simulationen festgestellt. Diese Sprünge sind hauptsächlich auf den Programmabschnitt zurückzuführen, der von dem `Timer` mit dem Namen `Build sparsity` gemessen wird.
Mit Hilfe der `Timer` konnten insbesondere zwei kritische Bereiche ausgemacht werden, die in weiterführenden Arbeiten mit einem speziellen Profiling Tool wie beispielsweise

`HPCToolkit` genauer untersucht werden können. Die Analyse der Ergebnisse eines Profilings würde detaillierte Hinweise liefern, anhand derer den FEniCS-Entwicklern konkrete Verbesserungsmöglichkeiten aufgezeigt werden könnten.
Als eine weitere Möglichkeit bietet sich im Forschungszentrum Jülich die Option an, FEniCS auf dem Supercomputer JUROPA zu installieren, dessen Architektur sich deutlich von JUQUEEN unterscheidet. Damit könnte FEniCS im Hinblick auf das Verhalten auf unterschiedlichen Supercomputer-Architekturen genauer untersucht werden.
Insgesamt ist FEniCS in der aktuell installierten Version mit allen verwendeten Bibliotheken geeignet, um Simulationen auf dem Hochleistungsrechner JUQUEEN durchzuführen. Da die Installation von Programmen auf einem Supercomputer sehr komplex ist, könnte in einem weiteren Schritt untersucht werden, ob die Bibliotheken optimal installiert wurden. Des Weiteren könnte FEniCS auch mit bisher noch nicht installierten Bibliotheken instrumentiert werden, um die Funktionalität zu erhöhen oder noch bessere Laufzeitergebnisse zu erzielen.

Literaturverzeichnis

[1] Martin S. Alnæs. *UFL: a Finite Element Form Language*, chapter 17. Springer, 2012.

[2] Martin S. Alnæs, Anders Logg, and Kent-Andre Mardal. *UFC: a Finite Element Code Generation Interface*, chapter 16. Springer, 2012.

[3] Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA, 2000.

[4] Tony Hansen and Garrett Wollman. *US Secure Hash Algorithm 1 (SHA1)*.

[5] G. Houzeaux, R. Aubry, and M. Vázquez. Extension of fractional step techniques for incompressible flows: The preconditioned orthomin(1) for the pressure schur complement. *Computers & Fluids*, 44(1):297–313, 2011.

[6] Robert C. Kirby. *FIAT: Numerical Construction of Finite Element Basis Functions*, chapter 13. Springer, 2012.

[7] Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[8] Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. *FFC: the FEniCS Form Compiler*, chapter 11. Springer, 2012.

[9] Anders Logg, Garth N. Wells, and Johan Hake. *DOLFIN: a C++/Python Finite Element Library*, chapter 10. Springer, 2012.

[10] Kristian Valen-Sendstad, Anders Logg, Kent-Andre Mardal, Harish Narayanan, and Mikael Mortensen. *A comparison of finite element schemes for the incompressible Navier-Stokes equations*, chapter 21. Springer, 2012.

A. Verschiedene Boxplots

Abbildung A.1.: Boxplots der gemessenen Rechenzeit interner DOLFIN `Timer` zur Darstellung der Variation über alle MPI Prozesse. (60 Millionen Navier-Stokes-Gleichung auf L-Gebiet, zwei Node-boards / 32 `ranks-per-node`)

Abbildung A.2.: Boxplots der gemessenen Rechenzeit interner DOLFIN `Timer` zur Darstellung der Variation über alle MPI Prozesse. (60 Millionen Navier-Stokes-Gleichung auf L-Gebiet, zwei Nodeboards / 64 `ranks-per-node`)

Abbildung A.3.: Boxplots der gemessenen Rechenzeit interner DOLFIN `Timer` zur Darstellung der Variation über alle MPI Prozesse. (60 Millionen Navier-Stokes-Gleichung auf L-Gebiet, vier Nodeboards / 32 `ranks-per-node`)

Abbildung A.4.: Boxplots der gemessenen Rechenzeit interner DOLFIN `Timer` zur Darstellung der Variation über alle MPI Prozesse. (60 Millionen Navier-Stokes-Gleichung auf L-Gebiet, vier Nodeboards / 64 `ranks-per-node`)

Jül-4382
Januar 2015
ISSN 0944-2952