Numerische Reproduzierbarkeit und parallele Berechnungen

von
Karl Ihlenfeldt

für das Seminar
„Softwareentwicklung in der Wissenschaft“

Tutor:
Petra Nerge

31.08.2020
Universität Hamburg

Inhaltsverzeichnis

1. Einleitung

Wissenschaftliches Arbeiten baut darauf auf, aus empirisch gesammelten Daten mittels wissenschaftlicher Methoden Hypothesen und Theorien zu erzeugen oder zu widerlegen.
Auf dieser Grundlage gewinnt auch die Reproduzierbarkeit eines Experimentes oder einer Studie an Bedeutung. Sollte ein erarbeitetes Ergebnis nicht reproduzierbar sein, gilt es im Allgemeinen als nicht glaubwürdig.
Sowohl das Sammeln der Daten, wie durch Experimente am HERA, dem Teilchenbeschleuniger am DESY [Q 1.1], als auch die Ausführung von Experimenten, wie numerische Simulationen von Klimaszenarien des Deutschen Klimarechenzentrums [Q 1.2], erfordern leistungsstarke Computer. Die resultierenden oder auch verarbeiteten Datenmengen sind zu groß, um sie per Hand durchzugehen und zu verarbeiten oder auszuwerten. Dementsprechend gewinnen auch Supercomputer eine immer größere Bedeutung in der Wissenschaft. Darauf ausgelegt, Billionen bis Billiarden Berechnungen pro Sekunde auszuführen, sind sie ein wichtiges Werkzeug für wissenschaftliches Arbeiten.

Deshalb ist es umso wichtiger, dass die mit ihnen verarbeiteten Daten ein reproduzierbares Ergebnis liefern. Dies bringt allerdings seine ganz eigenen Schwierigkeiten und Hürden mit sich.
Um diese besser zu verstehen, wird im 2. Abschnitt (HPC) der Begriff und die Arbeitsweise eines Supercomputers näher beleuchtet. Nach einer genaueren Definition der numerischen Reproduzierbarkeit in Abschnitt 3 (Numerische Reproduzierbarkeit), wird in Abschnitt 4 (IEEE und Gleitkommazahlen) der genaue Aufbau einer Gleitkommazahl erklärt und warum dies gerade bei Berechnungen auf Hochleistungsrechnern zu Problemen führt.
Nach dem in Abschnitt 5 (FMA) eine Möglichkeit zur Präzisierung einer Gleitkommaberechnung und in Abschnitt 6 (Compilerflags) Probleme bei der Optimierung der Programmlaufzeit betrachtet werden, befassen sich Abschnitt 7 (Intervalle & ihre Arithmetik) und Abschnitt 8 (Genauigkeit) mit Möglichkeiten zur Begrenzung und Quantifizierung von Rundungsfehlern.

2. HPC

Der Begriff HPC steht für High Performance Computer / Computing, also Hochleistungsrechner und ihre Anwendungsgebiete.
High Performance / Hochleistung ist dabei ein relativer Begriff, der von den aktuellen Standards und Technologien abhängt, und sich somit konstant verändert.
Zum Beispiel war der „Fujitsu Numerical Wind Tunnel“ von 1993 bis 1996 mit einer theoretischen Spitzenleistung von 235.8 GFlops der leistungsstärkste Supercomputer der Welt [Q2.1], wobei moderne Grafikkarten diese Leistung deutlich übertreffen. So zum Beispiel auch die 2016 produzierte NVIDIA GeForce GTX 1060, welche bis zu 4.375 TFlops erreichen kann [Q2.2], ein mehr als 16-faches des Fujitsu's.
Im Allgemeinen wird deshalb davon abgesehen, spezifische Merkmale wie die Anzahl der verbauten Prozessorkerne, RAM-Speicher oder Flops in der Definition des Begriffes zu verwenden.

Stattdessen wird der Begriff HPC über die Rechnerarchitektur definiert, welche auf parallele Verarbeitung ausgerichtet ist. [Q2.3]
Moderne Hochleistungsrechner sind verteilte Systeme und bestehen aus einer großen Anzahl miteinander als Grid oder Cluster vernetzter Computer oder Server, welche auch als „Knoten“ bezeichnet werden. Koordiniert werden die einzelnen Knoten von einer Zentralstelle, welche den jeweiligen Recheneinheiten ihre Aufgabe zuweist und den Datenaustausch zwischen den Knoten dirigiert. Realisiert wird dies durch eine sogenannte „Middleware“, einer Abstraktionsschicht

zwischen den auf dem Knoten vorhandenen Hardwarekomponenten, Betriebssystem und der jeweils ausgeführten Anwendung. Die Middleware stellt dadurch das Betriebssystem des HPCs dar und ermöglicht große Unterschiede im Aufbau der einzelnen Knoten, welche sogar aus Hardwarekomponenten verschiedener Hersteller bestehen können. [Q2.4]

Hochleistungsrechner werden meistens individuell für ihren Einsatzzweck hergestellt, was dazu führt, dass kaum je zwei HPCs exakt gleich sind. Dies spielt vor allem bei der Portabilität eines Programms eine große Rolle.
Die Grid- und Cluster-Architekturen ermöglichen einen hohen Grad an Parallelisierung und damit eine enorme Effizienzsteigerung: Anstatt ein Programm seriell abzuarbeiten, kann es in mehrere kleinere Threads aufgeteilt werden, welche dann parallel von verschiedenen Knoten gleichzeitig verarbeitet werden.
Gegenüber einer seriellen Bearbeitung benötigt die parallele nur einen Bruchteil der Zeit, bringt aber seine eigenen Probleme mit sich, wie zum Beispiel Verklemmungen oder Wettlaufsituationen.
Während Verklemmungen zwar dringend vermieden werden müssen, sind sie jedoch recht gut feststellbar und lassen sich oft beheben.
Wettlaufsituationen beschreiben Konstellationen, in der das Ergebnis von der zeitlichen Abfolge der Operationen abhängt. Ihr Auftreten ist nicht immer offensichtlich, sie sind schwerer zu beheben und ein häufiger Grund für nichtdeterministische Programmfehler. Damit sind sie auch ein großes Hindernis für die numerische Reproduzierbarkeit.

3. Numerische Reproduzierbarkeit

Numerische Reproduzierbarkeit stellt das Problem dar, das gleiche Ergebnis bei mehreren Durchläufen von ein und der selben Berechnung zu erhalten. Wenn die Berechnung mehrmals auf ein und derselben Maschine ausgeführt wird, spricht man auch von numerischer Wiederholbarkeit.

Numerische Reproduzierbarkeit ist vor allem wegen der Wechselwirkungen von Gleitkommazahlen, nichtdeterministischer Programmabfolge, und der Notwendigkeit einer schnellen Ausführung des Programms schwer zu erreichen.
So wird zum Beispiel die Aufteilung der Berechnung in kleinere Threads, für den Benutzer meist unsichtbar, vom Scheduler übernommen. Je nach Auslastung des Systems, oder auch dem Programm zugeteilten Ressourcen, kann sich dabei die Reihenfolge der durchgeführten Operationen verändern. Der Programmablauf verhält sich also nicht-deterministisch. Durch die fehlende Assoziativität der Gleitkommazahlen (siehe 4. IEEE 754 und Gleitkommazahlen) kann dies zu Wettlaufsituationen führen und sowohl das Ergebnis verfälschen, als auch das Finden und Beheben von Fehlern erschweren.

Um solche Wettlaufsituationen zu bemerken, ist eine hohe Genauigkeit der Berechnung wichtig. Hierbei muss eine Abwägung zwischen Genauigkeit und Geschwindigkeit getroffen werden:
Ein Ergebnis mit einer niedrigen Genauigkeit hat zwar weniger Aussagekraft, allerdings steigt die für die Berechnung benötigte Zeit mit der Genauigkeit an.

4. IEEE 754 und Gleitkommazahlen

Gleitkommazahlen spielen eine elementare Rolle bei wissenschaftlichen Berechnungen. Denn selbst wenn die für die Berechnung verwendeten Daten als ganze Zahlen darstellbar sind, werden doch Konstanten benutzt oder Zwischenergebnisse errechnet, welche nur als Gleitkommazahlen vorhanden sind. Ein Beispiel hierfür wäre die Division zweier ganzer Zahlen, welche in vielen Fällen nicht als ganze Zahl darstellbar ist.

Die Standarddarstellung für Gleitkommazahlen wird im Standard des IEEE 754 definiert [Q4.1].
Eine Gleitkommazahl $x = s*m*2^e$ besteht aus:

- Dem Vorzeichen s
- Der Mantisse m
- Dem Exponenten e

Es gibt verschiedene Gleitkommaformate, wie zum Beispiel float, double oder longdouble, welche sich nur darin unterscheiden, wie viele Bits für jeweils die Mantisse und den Exponenten reserviert werden. Für das Vorzeichen wird genau ein Bit reserviert.
Der Exponent wird implizit mit einem Bias subtrahiert, welcher sich je nach ausgewähltem Datenformat ändert. Für float zum Beispiel ist der Bias 127, für double 1023. Allgemeiner: Der Bias beträgt $2^{r-1}-1$, wobei r die Anzahl der für den Exponenten reservierten Bits ist. Dies ermöglicht Darstellungen von sowohl positiven als auch negativen Exponenten, ohne das Vorzeichen extra speichern zu müssen oder Kollisionen mit dem 2er-Komplement in Kauf zu nehmen.
Für die Mantisse wird implizit eine 1 vorangestellt und die Nachkommastellen eingespeichert, wodurch ein weiteres Bit für eine höhere Präzision genutzt werden kann.

Um eine beliebige Zahl in die binäre Gleitkommadarstellung umzuwandeln, muss sie „normalisiert“ werden.
Zuerst wird das Vorzeichen der Zahl abgespeichert.
Als nächstes wird die Zahl in Binärdarstellung gebracht, wobei sich die Nachkommastellen als Summe von Zweierpotenzen mit negativem Exponenten zusammensetzen.
Danach wird das Komma verschoben, so dass nur eine einzige Eins vor dem Komma steht. Diese Eins wird impliziert und nicht extra abgespeichert. Der Exponent wird für jede verschobene Stelle erhöht, beziehungsweise verringert, je nachdem in welche Richtung das Komma geschoben wurde.

Diese Darstellung von Gleitkommazahlen benötigt nur wenige Bits, hat aber auch ihre Grenzen.
Ein großes Problem ist die fehlende Assoziativität. Um zum Beispiel zwei Gleitkommazahlen miteinander zu addieren, müssen sie den gleichen Exponenten haben. Um dies zu erreichen, wird einer der beiden Exponenten erhöht, wobei das Komma in der Mantisse entsprechen verschoben werden muss. Dies kann dazu führen, dass Bits verloren gehen und sich der tatsächliche Wert der Zahl ändert.
Ein Beispiel hierfür wäre $2^{100} + 1$. Die Darstellung der beiden Zahlen wäre $1{,}0*2^{100} + 1{,}0*2^0$. Um den Exponenten von 1, beziehungsweise $1{,}0*2^0$, auf 100 zu erhöhen, wird auch das Komma verschoben. Aus $1{,}0*2^0$ wird $0{,}1*2^1$, dann $0{,}01*2^2$, bis schlussendlich 99 Nullen und eine Eins hinter dem Komma stehen. Da die meisten Gleitkommaformate deutlich weniger Bits für die Mantisse zur Verfügung stellen, geht dabei die Genauigkeit verloren und der Wert der Zahl ändert sich zu 0. Im Endeffekt wird also $2^{100} + 0$ berechnet und die Eins wird absorbiert.
Die Reihenfolge der Operationen kann also das Ergebnis der Berechnung verändern. So ergibt zum Beispiel $(2^{100} - 2^{100}) + 1 = 1$, aber $2^{100} - (2^{100} + 1) = 0$. Berechnungen mit Gleitkommazahlen sind also nicht assoziativ!

Wie bereits erwähnt, ist die Reihenfolge der Operationen nicht immer deterministisch und die fehlende Assoziativität der Gleitkommazahlen ist ein häufiger Grund für Wettlaufsituationen. Eine höhere Präzision kann zwar mehr Fälle abdecken und beim Erkennen der Fehler helfen, ist aber keine Garantie für eine korrekte Berechnung.

Ein weiteres Problem ist die endliche Menge an darstellbaren Zahlen.
Nicht jede Zahl lässt sich als endliche Summe von Zweierpotenzen beschreiben. Neben den offensichtlich irrationalen Zahlen wie die Eulersche Zahl „e“ oder „pi“, sind auch Zahlen wie 0,2 = 0,001100110011... in binärer Darstellung nicht rational und müssen approximiert werden.
Hierfür wird die Zahl gerundet, sodass der für 0,2 eingespeicherte Wert tatsächlich größer als 0,2 ist.
Für die Rundung von Gleitkommazahlen definiert der IEEE 754 vier verschiedene Modi:

- Richtung $+\infty$, also Aufrunden
- Richtung $-\infty$, also Abrunden
- Richtung 0, so dass der Betrag der Zahl kleiner wird
- Zur nächsten darstellbaren Zahl

Der letzte Modus, Runden zur nächsten darstellbaren Zahl, beinhaltet eine Fallunterscheidung. Wenn die zu rundende Zahl genau zwischen 2 darstellbaren Zahlen liegt, kann entweder von der 0 weg gerundet werden, so dass der Betrag der Zahl größer wird, oder es wird zur nächsten geraden Zahl gerundet. Gerade bedeutet in diesem Zusammenhang, dass das niederwertigste Bit 0 wird.
Dieser letzte Fall, Rundung zur nächsten geraden darstellbaren Zahl, ist der Standard für die meisten Programmiersprachen.

5. FMA

Die Auswahl des richtigen Rundungsmodus ist wichtig für die numerische Reproduzierbarkeit, allerdings bedeutet jede Rundung auch eine Verfälschung des Ergebnisses, die im Nachhinein nicht mehr nachvollziehbar ist. Denn selbst wenn die Richtung der Rundungen bekannt ist, ist es sehr aufwendig bis unmöglich die Größe des Fehlers abzuschätzen.

Um die Anzahl der Rundungen zu verringern und damit den Fehler zu minimieren, sollte, wo es möglich ist, auf FMA zurückgegriffen werden.
FMA steht für „fused-multiply-addition“ und wurde 2008 in den Standard IEEE 754 aufgenommen.
Bei einer normalen Verrechnung von Gleitkommazahlen wird nach jeder Rechenoperation gerundet. Bei der Rechnung a*b + c wird also erst a*b verrechnet, das Ergebnis gerundet und dann c addiert. Das Ergebnis wird ein zweites Mal gerundet.
FMA ermöglicht die gleiche Rechnung mit einer einzigen Rundung: Das Produkt von a*b wird mit voller Genauigkeit berechnet, c wird addiert und dann wird einmalig gerundet.
Dies ermöglicht nicht nur eine Minimierung von Rundungsfehlern, sondern auch eine höhere Geschwindigkeit der Berechnung, da für die Operation ein spezieller Befehlssatz im Prozessor zur Verfügung steht, der die Multiplikation und Addition zu einem Schritt vereinigt. [Q5.1, Seite 24]

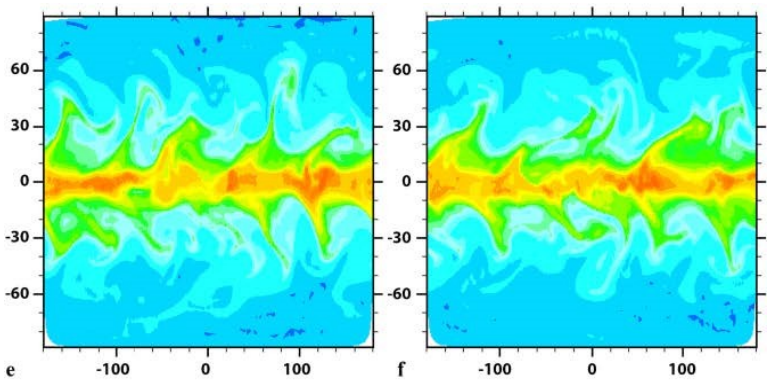
FMA muss sowohl von der Hardware als auch von der Software unterstützt und implementiert werden. Obwohl die Technologie schon Ende der 1980er Jahre von IBM Research entwickelt wurde [Q5.2], verbreitete sie sich nur langsam. So haben zum Beispiel Intel und AMD erst 2006, beziehungsweise 2009, angefangen die FMA-Architekturen großflächig zu implementieren.
Es gibt 2 Varianten des FMA-Befehlssatzes: FMA3 und FMA4. Beide sind Teil der 128- und 256-bit Streaming Single-Input-Multiply-Data Extension, einer Befehlssatzerweiterung für Prozessoren der x86-Architektur, welche eine Verbesserung der Programmlaufzeit „durch Parallelisierung auf

Instruktionslevel“ [Q5.3] ermöglicht.
FMA3 und FMA4 unterscheiden sich hauptsächlich in der Anzahl der genutzten Register: FMA3 speichert das Ergebnis der Rechnung a*b + c im Register von c, während FMA4 ein weiteres Register nutzt.
Auf der Softwareebene müssen sowohl die Programmiersprache als auch der Compiler FMA unterstützen. Bei den Programmiersprachen gibt es eine breite Implementierung. In den meisten Fällen gibt es einen FMA-Befehl in der jeweiligen Mathe-Bibliothek, oder wenigstens eine alternative Bibliothek mit der entsprechenden Implementierung.
Auch die Implementierung durch den Compiler ist weitestgehend gegeben, eine Liste von Compilern mit FMA-Unterstützung findet sich unter [Q5.4].
Um auf den FMA-Befehlssatz zugreifen zu können, müssen bei vielen dieser Compiler extra Flags gesetzt werden. Was genau eine Flag ist, oder wofür sie verwendet werden kann, wird im folgenden Abschnitt näher erläutert.

6. Compilerflags

Compilerflags sind vom Entwickler gesetzte Anweisungen, die das Verhalten des Compilers beim Übersetzen des Programmcodes beeinflussen. Je nach Programmiersprache oder Compiler gibt es viele verschiedene Flags die gesetzt werden können, wobei nicht jede Sprache die Möglichkeit anbietet. Java zum Beispiel erlaubt keine Flags.
Flags haben viele verschiedene Funktionalitäten und einen vielfältigen Anwendungsbereich.
Für die numerische Reproduzierbarkeit sind dabei vor allem die Anweisungen wichtig, die Gleitkommaberechnungen regulieren oder die Laufzeitgeschwindigkeit optimieren.
Ein Beispiel hierfür wäre die -Ofast Flag der GNU Compiler Collection. Einmal gesetzt, gibt sie dem Compiler mehrere Anweisungen auf einmal und aktiviert, unter anderem, Flags wie -fno-rounding-math oder -fassociative-math. Die -fno-rounding-math - Flag deaktiviert Umwandlungen und Optimierungen, welche auf dem Rundungsmodus „zur nächsten Darstellbaren Zahl“ (siehe 4. IEEE 754 und Gleitkommazahlen) aufbauen, die -fassociative-math - Flag erlaubt Optimierungen unter der Annahme, die Berechnungen wären assoziativ. Wie in Abschnitt 4 erklärt, verstößt -Ofast damit gegen mehrere Normen des IEEE Standards, ermöglicht allerdings eine schnellere Ausführungszeit des Programms. [Q6.1]
Solche auf Laufzeitoptimierung ausgelegte Flags sind häufig ein Risiko und können sogar eine Reproduktion eines Ergebnisses unmöglich machen.

In dieser Abbildung ist die Problematik gut zu erkennen.
Sie ist das Ergebnis einer Klimamodellierung vom Deutschen Wetterdienst und dem Max-Planck-Institut für Meteorologie. Beide Graphiken sind das Ergebnis von ein und derselben Berechnung auf dem selben Datenset. Für die Lauzeitoptimierung wurden dabei die Compiler-Optionen -O3 und -xAVX genutzt, welche aber auch einen nichtdeterministischen Programmablauf verursachen [Q6.2].



Berechnung des Gewichtes der Wasserdampfsäule in einem globalen Klimamodell mit Optimierungsstufe -O3 -xAVX und nicht bitweise reproduzierbarem Berechnungsergebnis. Quelle: https://gi.de/informatiklexikon/reproduzierbarkeit

Wie in den Abschnitten 2 und 4 erläutert könnte dies Wettlaufsituationen verursacht haben und somit verschiedene Ergebnisse erzeugen.

Obwohl die in der Abbildung simulierte Zeit gerade einmal 100 Tage beträgt, sind deutliche Abweichungen zu erkennen und die Ergebnisse sind nicht bitweise reproduzierbar. Abweichungen dieser Art sind keinesfalls offensichtlich und werden häufig erst nach mehrfacher Ausführung des Programms bemerkt. [Q6.2]

7. Intervalle & ihre Arithmetik

Der numerischen Reproduzierbarkeit stehen also einige Hürden im Weg. Doch selbst wenn eine Berechnung ein reproduzierbares Ergebnis liefert, ist dieses nicht exakt, sondern durch Rundungen verfälscht. Eine Interpretation des Ergebnisses wird zusätzlich dadurch erschwert, dass die Größe und Richtung der Abweichung im Nachhinein nicht mehr nachvollziehbar ist.

Um diese Abweichungen besser abschätzen und begrenzen zu können, bietet sich die Verwendung von Intervallen an [Q7.1].
Es gibt verschiedene Darstellungsformen von Intervallen, wobei jede ihre eigenen Vor- und Nachteile hat. So ermöglicht zum Beispiel die Mittelpunkt-Radius-Darstellung eines Intervalls eine exakte Addition, aber Multiplikation gibt nur eine Begrenzung der Intervallgröße.
Je nach Anwendungsgebiet werden also verschiedene Darstellungen und die damit verbundenen Arithmetiken genutzt.
Bei numerischen Berechnungen bietet sich die Darstellung als Endpunktnotation an.
Hierfür werden die Daten zu Punktintervallen umgewandelt, aus einem double-Wert x wird also das Intervall [x, x]. Bei jeder Rechenoperation mit dem Intervall wird nun gerichtet gerundet: die obere Grenze des Intervalls wird gegen $+\infty$, also aufgerundet, die untere Grenze wird gegen $-\infty$, also abgerundet. Dadurch wird sichergestellt, dass das exakte Ergebnis im Intervall liegt. [Q7.1]

Die Addition und Subtraktion zweier Intervalle ist über die Addition, beziehungsweise Subtraktion, ihrer Endpunkte definiert:

$$[a, b] + [c, d] = [a + c, b + d]$$
$$[a, b] - [c, d] = [a - d, b - c]$$

Für die Multiplikation müssen die Maxima und Minima der möglichen Endpunktmultiplikationen betrachtet werden:

$$[a, b] * [c, d] = [\min S, \max S]$$

wobei

$$S = \{a*c, a*d, b*c, b*d\}$$

Die Division zweier Intervalle ist nur eindeutig definiert, wenn der Divisor nicht die 0 umschließt. In diesem Fall wäre

$$[a, b] / [c, d] = [a, b] * [1/c, 1/d]$$

Andernfalls gibt es verschiedene Möglichkeiten die Division zu definieren und zu interpretieren, wobei je nach Anwendung eine andere Definition möglich oder sogar notwendig ist.

Aus diesen Definitionen folgen sowohl die Assoziativität als auch Kommutativität, allerdings gibt es keine additiven oder multiplikativen Inversen. Auch die Distributivität ist nicht gegeben.
So gilt zum Beispiel

$$[1, 2] * ([1, 1] - [1, 1])$$
$$= [1, 2] * ([0, 0])$$
$$= [0, 0]$$

wobei die Anwendung des Distributivgesetzes

$$[1, 2] * ([1, 1] – [1, 1])$$
$$= [1, 2] * [1, 1] - [1, 2] * [1, 1]$$
$$= [1, 2] – [1, 2]$$
$$= [–1, 1]$$

ergibt.
Anstatt der Distributivität gilt nur eine Subdistributivität: Die Anwendung des Distributivgesetzes resultiert in einem Ergebnis, dass das exakte Intervall umschließt, aber überschätzt. [Q7.2, Seite 32]

Ein weiterer Grund für Überschätzungen ist das Abhängigkeitsproblem.
Für die quadratische Funktion $f(x) = x^2$ ergibt sich mit klassischer Intervallarithmetik

$$f([–1, 1]) = [–1, 1]^2 = [–1, 1] * [–1, 1] = [–1, 1]$$

Dieses Ergebnis ist offensichtlich falsch, beziehungsweise ungenau. Die Quadrierung einer reellen Zahl kann keine negative Zahl ergeben. Der Fehler entsteht, weil die beiden Faktoren [–1, 1] und [–1, 1] als unabhängige Parameter betrachtet werden. Auch hier findet eine Überschätzung des Intervalls statt; das exakte Ergebnis von $[–1, 1]^2 = [0, 1]$ ist im überschätzten Intervall enthalten [Q7.2, Seite 38].

8. Genauigkeit

Bei korrekter Anwendung der Arithmetik wird zwar das Inklusionsprinzp eingehalten, sodass das exakte Ergebnis im überschätzten Intervall enthalten ist, allerdings lässt sich die Abweichung im Nachhinein nicht mehr nachvollziehen.
Diese Problematik verschärft sich weiter: Die Endpunkte der Intervalle sind selbst Gleitkommazahlen und leiden unter den kombinierten Problemen. Sowohl Assoziativität als auch Distributivität sind nicht gegeben, sodass auch eine Berechnung mit Intervallen unter der nicht-deterministischen Programmabfolge zu verfälschten Ergebnissen führt [Q8.1, Abschnitt 5.1].

Ein Ansatz um den Betrag eines Intervalls zu verringern und somit ein Ergebnis zu präzisieren sind iterative Algorithmen. Hierfür wird mit linearen Gleichungssystemen eine wiederholte Matrizenmultiplikation auf dem Intervall ausgeführt, um dieses so schrittweise zu verringern.
Hier ist wieder eine Abwägung von Geschwindigkeit gegen Genauigkeit notwendig. Das Ziel der iterativen Präzisierung ist eine Konvergenz gegen einen festen Wert, der sich nur um einen Rundungsfehler vom exakten Ergebnis unterscheidet.
Dies ist in vielen Fällen auch erreichbar [Q8.1, Abschnitt 4.2], setzt aber voraus, dass der iterative Algorithmus genug Schritte durchlaufen hat. Die Anzahl der dafür benötigten Schritte kann dabei stark variieren, sodass eine Begrenzung der Schritte zu Gunsten der Geschwindigkeit die numerische Reproduzierbarkeit gefährdet.

9. Fazit

Alles in allem sind Intervalle kein Ersatz für Gleitkommazahlen.
Zum einen kommt es aufgrund von Abhängigkeiten häufig zu einem, zwar korrektem, aber nutzlosem Ergebnisintervall von $(-\infty, +\infty)$, zum anderen kommen je nach Anwendungsbereich und Situation verschiedene Algorithmen für die Intervallberechnungen zum Einsatz.
Als grobe Regel sollten Gleitkommaberechnungen verwendet werden, falls eine Verifizierung nicht notwendig ist und Intervallberechnungen für die strategische Begrenzungen der Rundungsfehler. [Q7.1, Minute 47:58]

Trotz fortschreitender technischer Entwicklung und Verbesserungen einzelner Hardwarekomponenten werden auch in Zukunft HPCs und ihre damit verbundene parallele Architektur eine große Rolle in der Wissenschaft spielen.
Für die numerische Reproduzierbarkeit sieht es also auch in Zukunft schwierig aus.
Probleme wie die fehlende Assoziativität von Gleitkommazahlen oder Wettlaufsituationen können dabei bestenfalls ausgeglichen, aber nicht eliminiert werden.

Die von den HPCs errechneten Ergebnisse sind weder exakt noch endgültig, sondern erfordern ein hohes Maß an rigoroser Einschätzung und Interpretation der jeweiligen Wissenschaftler.
Somit stellt sich die Frage, ob die numerische Reproduzierbarkeit eines Ergebnisses überhaupt immer zwingend erforderlich ist, oder ob Garantien für die Genauigkeit ausreichen könnten.
Diese Frage hat keine allgemein gültige Antwort, aber, wie in [Q9.1] vertieft, eine hohe praktische Bedeutung. Sowohl der Energiebedarf des Systems als auch die Ausführungsgeschwindigkeit eines Programms könnten von niedrigeren Anforderungen an die Reproduzierbarkeit profitieren.

10. Quellen

[Q1.1] https://www.desy.de/forschung/anlagen__projekte/hera/index_ger.html
[Q1.2] https://www.dkrz.de/kommunikation/klimasimulationen
[Q2.1] https://en.wikipedia.org/wiki/Numerical_Wind_Tunnel
[Q2.2] https://www.techpowerup.com/gpu-specs/geforce-gtx-1060-6-gb.c2862
[Q2.3] https://de.wikipedia.org/wiki/Supercomputer#Geschichte_und_Aufbau
[Q2.4] https://svs.informatik.uni-hamburg.de/teaching/VSS-Screencast.pdf, Folie 19
[Q4.1] https://ieeexplore.ieee.org/document/8766229
[Q5.1] E. C. Quinell, „Floating-Point Fused Multiply-Add Architectures“, 2007, University of Texas at Austin, https://repositories.lib.utexas.edu/bitstream/handle/2152/3082/quinnelle60861.pdf
[Q5.2] https://de.wikipedia.org/wiki/Fused_multiply-add
[Q5.3] https://de.wikipedia.org/wiki/Streaming_SIMD_Extensions
[Q5.4] https://en.wikipedia.org/wiki/FMA_instruction_set
[Q6.1] https://gcc.gnu.org/wiki/FloatingPointMath
[Q6.2] https://gi.de/informatiklexikon/reproduzierbarkeit
[Q7.1] R. B. Kearfott, „Interval Arithmetic: Fundamentals, Successes, and Pitfalls“, Universidad EAFIT, 27. July 2018 - https://www.youtube.com/watch?v=SLHnQuAPLsI&t=1815s
[Q7.2] R. Moore, R.B. Kearfott, und M. J. Cloud, „Introduction to Interval Analysis“, Philadelphia, PA, USA: SIAM, 2009 - http://www-sbras.nsc.ru/interval/Library/InteBooks/IntroIntervAn.pdf
[Q8.1] N. Revol und P. Théveny, „Numerical Reproducibility and Parallel Computations: Issues for Interval Algorithms“, August 2014
[Q9.1] J.Dongarra, „Algorithmic and Software Challenges For Numerical Libraries at Exascale“, University of Tennessee, 13. September 2013 - https://www.youtube.com/watch?v=CPuWKT6_vwk&t=14m58s

Weitere Quellen:

https://accurate-algorithms.readthedocs.io/en/latest/ch03fma.html
https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Alle Quellen wurden am 31.08.2020 um 18:00 Uhr abgerufen