Hochschule München University of Applied Sciences

Fakultät für Informatik
und Mathematik

Diese Erklärung ist zusammen mit der Bachelorarbeit bei dem/der PrüferIn abzugeben.

Bostanci, Manuel	München, 29.11.2021
(Familienname, Vorname)	(Ort, Datum)
16.02.1995 (Geburtsdatum)	IB7C / WS / 20 21 (Studiengruppe / WS/SS)

Erklärung

Hiermit erkläre ich, dass ich die Bachelorarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

M. Bostanci

(Unterschrift)

Bachelorarbeit

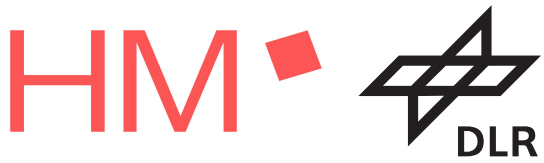
Laufzeitoptimierung von FAME-IO

Gesamtlaufzeit in Abhängigkeit der Agentenanzahl und Simulationsgranularität

Fakultät für Informatik und Mathematik
Hochschule München University of Applied Sciences

in Zusammenarbeit mit

Institut für vernetzte Energiesysteme
Deutsches Zentrum für Luft- und Raumfahrt (DLR)



Eingereicht bei:

Prof. Dr. Johannes Ebke

Betreut von:

Dr. Christoph Schimeczek

Eingereicht von:

Manuel Bostanci (296 492 18)
Studiengang: Bachelor of Science Wirtschaftsinformatik

München, 29. November 2021

Zusammenfassung

Das Ziel der vorliegenden Arbeit ist die Optimierung der Laufzeit von FAME-IO, einer Komponente des agentenbasierten Simulationsframeworks FAME. Die Laufzeit von FAME-IO hängt direkt von der Anzahl der zu verarbeitenden Parameter Ägentenünd deren SZeitreiheneinträgenäb. Daher wird die folgende Forschungsfrage untersucht: In wie weit kann die Laufzeit von FAME-IO in Abhängigkeit der Anzahl an Agenten und Zeitreiheneinträgen reduziert werden. Darüber hinaus wird die Frage, welche Auswirkungen die angewendeten Optimierungen auf den Arbeitsspeicherbedarf haben, beantwortet.

Zur Beantwortung der Forschungsfragen wurde der Ausgangszustand des Programms analysiert und anhand eines Profilings die Programmteile identifiziert, die den größten Anteil an der Gesamtlaufzeit haben. Zudem wurde der Bedarf an Arbeitsspeicher mit einem Memory Profiler ermittelt.

Die Analyse des Ausgangszustandes zeigte, dass die Umwandlung der Zeitreiheneinträge einen Großteil der Gesamtlaufzeit einnimmt. Zur Verbesserung dieses Programmteils wurde neben einer effizienteren Implementierung der Verarbeitung der Zeitreiheneinträge auch ein Mechanismus zu deren parallelen Verarbeitung angewendet. Durch die angewendeten Methoden konnte die Gesamtlaufzeit der Verarbeitung des umfangreichsten Simulationsszenarios von 232 Sekunden auf 133 Sekunden reduziert und somit nahezu halbiert werden. Die Auswirkung der parallelen Verarbeitung auf mehreren Prozessen auf den Arbeitsspeicher ist dabei abhängig von der verwendeten Methode zum Initialisieren neuer Prozesse. Werden acht Prozesse gestartet, benötigen diese bei Verwendung der "Fork" Methode gemeinsam etwa 35 GiB, bei Verwendung der "Spawn" Methode hingegen gemeinsam nur acht GiB an Arbeitsspeicher.

Eine weiterführende Untersuchung könnte auf die Erhöhung des parallelisierbaren Anteils des Programmcodes ausgerichtet sein, oder auch schnellere alternativen zu eingebundenen fremden Programmbibliotheken ermitteln, z. B. zum Einlesen von Datein im YAML Format.

Inhaltsverzeichnis

1 Einleitung

Auf der internationalen Klimakonferenz am 12. Dezember 2015 in Paris beschlossen Vertreter aller Staaten das Pariser Klimabakommen, welches die Staaten dazu verpflichtet, "[...]die Weltwirtschaft auf klimafreundliche Weise zu verändern"[2]. Die Emissionen durch den Einsatz fossiler Brennstoffe bei der Energiegewinnung stehen diesem Klimaziel jedoch entgegen. Um diese Auswirkungen zu reduzieren, sollte auf eine nachhaltige Energieerzeugung mit reduziertem Emissionsausstoß umgestellt werden [8]. Eine solche Umgestaltung des Energiesystems erfordert komplexe, multikriterielle Bewertungen, welche Umweltbelange berücksichtigen und gleichzeitig zu globaler Nachhaltigkeit führen. Zwar existieren eine Vielzahl an Energiesystemmodellen, allerdings verfolgen die meisten dieser Modelle volkswirtschaftliche Optimierungsansätze, welche häufig systemoptimale Zielszenarien konstruieren [8]. Es bleibt dabei jedoch oft unklar, wie dieses systemoptimale Ziel erreicht werden kann. In [21] zeigen Torralba-Díaz et al., wie Ergebnisse von agentenbasierten Modellsimulationen, welche sowohl komplexes Akteursverhalten als auch Marktverzerrungen aufgrund von politischen Instrumenten berücksichtigen, von denen aus Optimierungsmodellen abweichen und diese ergänzen. Dennoch existieren nur wenige ausgereifte und aktiv entwickelte agentenbasierte Simulationsmodelle, die umfangreich und komplex genug sind, um energiepolitische Forschungsfragen zu beantworten.

Mit dem Framework zur verteilten, agentenbasierten Modellierung von Energiesystemen namens FAME entwickeln die Wissenschaftler des Deutschen Zentrums für Luft- und Raumfahrt eine Software, die als quelloffener Standard für agentenbasierte Simulationsmodelle fungieren soll. FAME bietet ein Rahmenwerk an ausführlich getesteten Funktionen, die die Entwicklung neuer Modelle erleichtern sollen. Zudem soll durch den quelloffenen Ansatz und gemeinsame Schnittstellen eine höhere Transparenz der Modelle erreicht werden. Darüber hinaus zielt FAME darauf ab, den Bedarf an IT-Kenntnissen im Bereich der Programmparallelisierung zu minimieren, die Codebasis von agentenbasierten Modellen weniger komplex zu gestalten und die Dauer einer Simulationsdurchführung zu reduzieren.

FAME umfasst die in Abbildung 1 gezeigten Komponenten FAME-IO und FAME-CORE. FAME-IO wandelt die Simulationskonfiguration und die dafür benötigten zusätzlichen Daten wie z. B. Zeitreihen in Protobuffer Objekte um. Die erzeugten Protobuffer Objekte werden anschließend von FAME-CORE eingelesen und eine Simulation durchgeführt. Das Simulationsergebnis besteht ebenfalls aus Protobuffer Objekten, welche von FAME-IO in ein menschenlesbares Format wie z. B. CSV umgewandelt werden.

Diese Arbeit betrachtet die Umwandlung der Simulationskonfiguration in der FAME-IO Komponente und beschäftigt sich mit der Reduktion der Gesamtlaufzeit einer FAME basierten Simulation. Dies soll durch eine optimierte Verarbeitung der Simulationskonfiguration in der Komponente FAME-IO erreicht werden. Dazu wird der Einfluss zweier maßgeblicher Konfigurationsparameter von FAME-IO, nämlich die betrachtete Anzahl an Agenten sowie die Anzahl an zu berücksichtigenden Zeitreiheneinträgen auf die Gesamtlaufzeit des Systems

betrachtet.

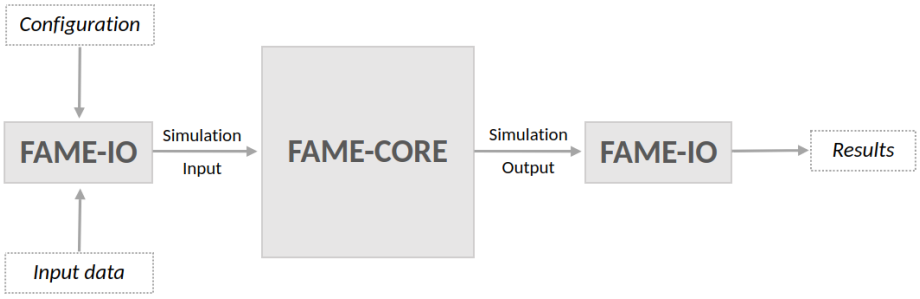


Abbildung 1: Komponenten einer FAME Simulation

Zu Beginn der Arbeit wird der Ausgangszustand von FAME-IO analysiert und so die laufzeitkritischsten Programmteile identifiziert. Anschließend werden die als relevant erkannten Programmteile durch Anwendung geeigneter Optimierungen hinsichtlich ihrer Laufzeit verbessert und die erzielte Laufzeitverbesserung für verschiedene Parametervariationen untersucht.

Somit ergeben sich die folgenden zentralen Forschungsfragen:

1. *Wie weit kann die Laufzeit von FAME-IO in Abhängigkeit sowohl der Anzahl an Agenten als auch Zeitreiheneinträge reduziert werden?*

2. *Welche Auswirkungen haben die angewendeten Optimierungen auf den Arbeitsspeicherbedarf von FAME-IO?*

2 Grundlagen

Im folgenden Kapitel werden die Grundlagen der Arbeit näher beleuchtet, die zum besseren Verständnis der Aufgabe und der Vorgehensweise beitragen sollen. Es wird sowohl auf die verwendete Programmiersprache Python eingegangen, als auch die damit verbundenen Möglichkeiten zur Analyse und Optimierung der Laufzeit einer in Python implementierten Anwendung. Weiter wird der Unterschied zwischen Parallelisierung und Multitasking erläutert und das Amdahlsche Gesetz näher betrachtet.

2.1 Die Programmiersprache Python

Die Programmiersprache Python wurde von Guido van Rossum federführend entwickelt und am 20. Februar 1991 in der Version 0.9 erstmals veröffentlicht.[16] In den Folgejahren wurde die Programmiersprache kontinuierlich weiterentwickelt und steht heute in der Version 3.10 zur Verfügung.[13] Auch die Verbreitung von Python nahm im Laufe der Jahre stetig zu und verzeichnet laut PYPL Popularity of Programming Language Index aktuell einen Anteil von 29,48%. Damit steht Python aktuell auf dem ersten Platz der Programmiersprachen, nach denen am häufigsten auf der Suchmaschinenplattform Google gesucht wird.[3]

Nicht nur die hohe Verbreitung von Python, auch das umfangreiche Ökosystem an verfügbaren Softwarebibliotheken und Frameworks, um die sich die Standardfunktionalität von Python unkompliziert erweitern lässt, macht die Sprache vielseitig einsetzbar. So stellt allein der Python Package Index (PyPI) aktuell mehr als 327.000 Erweiterungen zur Verfügung[15].„Insbesondere die Module NumPy, SciPy und Matplotlib bieten Naturwissenschaftlern und Ingenieuren eine perfekte Entwicklungsumgebung für Wissenschaftliches und Technisches Rechnen, für Anwendungen in der Physik, Chemie, Biologie und Informatik."[17]. Neben NumPy und SciPy wird auch die Datenanalysebibliothek *pandas* häufig im wissenschaftlichen Kontext eingesetzt. *Pandas* bietet eine Vielzahl an Funktionen und Klassen zur Verarbeitung und Manipulation von strukturierten Daten. Ausgangspunkt für diese Operationen sind die ein- und zweidimensionalen Datenstrukturobjekte *Series* und *DataFrame* [7].

Schlussfolgernd ist Python eine weit verbreitete und vielseitig einsetzbare Programmiersprache, welche auch mehr als 30 Jahre nach der erstmaligen Veröffentlichung noch den Anforderungen der Zeit entspricht. Eines der essentiellen Bestandteile für das Fortbestehen von Python ist das umfangreiche Modulsystem, wodurch die Programmiersprache um nützliche Funktionen erweitert werden und sich so stets auf neue Anwendungsbereiche und Problemstellungen anpassen kann. Im nächsten Kapitel wird das Modul *timeit* betrachtet, welches zur Messung von Programmlaufzeiten verwendet wird.

2.2 Laufzeitanalyse

Unter dem Begriff Laufzeitanalyse versteht man in der Softwareentwicklung die Analyse, wie lange die Ausführung eines Programms oder eines dedizierten Teil eines Programms in Anspruch nimmt. Ziel der Analyse ist es, lang laufende Programmteile zu identifizieren. Hierfür stellt die Python Standardbibliothek die Module *timeit*, *profile* und *cProfile* zur Verfügung.

Mit dem Modul timeit kann gemessen werden, wie lange ein Programm, oder eine Funktion innerhalb eines Programms, zur Ausführung benötigt. Dies ist besonders nützlich wenn man die Laufzeit unterschiedlicher Funktionen mit identischem Ergebnis untersuchen möchte. Neben der einfachen Berechnung der Laufzeit eines einmaligen Ausführens der entsprechenden Funktion, bietet *timeit* auch die Möglichkeit, die Funktion wiederholt auszuführen und anschließend die durchschnittliche Laufzeit inklusive Standardabweichung zu berechnen [20]. Besteht das Ziel jedoch darin, die Funktion mit dem größten Anteil relative zur Gesamtlaufzeit zu identifizieren, eignet sich ein Profiler.

Als Profiler wird eine Komponente bezeichnet, die Statistiken über jede aufgerufene Funktion eines Programms und deren Ausführungszeit bereitstellt. Standardmäßig beinhaltet Python die Profiler *cProfile* und *profile*. Während *cProfile* in der Programmiersprache C implementiert ist und damit einen nur sehr geringen laufzeitrelevanten Overhead zur Gesamtlaufzeit des zu analysierenden Programms hinzufügt, ist *profile* in Python implementiert, was einen signifikanten, laufzeitrelevanten Overhead generiert. Somit ist die Verwendung von *cProfile* in den meisten Fällen die bevorzugte Implementierung und wird auch für das Profiling im Rahmen dieser Arbeit verwendet. Ein Profiling mit *cProfile* kann entweder über die Kommandozeile oder durch das explizite Importieren des *cProfile* Moduls in die zu analysierende Anwendung erfolgen. Ein Profiling der Anwendung test.py kann beispielsweise mit dem Kommandozeilenbefehl `python -m cProfile test.py` durchgeführt werden [19]. Nach erfolgreicher Durchführung eines Profilings, bieten die meisten Profiler eine visuelle Darstellung der Ergebnisse an.

Die Anwendung der hier beschriebenen Methoden und Hilfsmittel dient dazu, ein bestehendes Programm hinsichtlich der Laufzeit zu analysieren und etwaige Engpässe zu identifizieren. Welche Arten von Engpässen in einem Python Programm entstehen können, wird im nächsten Kapitel behandelt.

2.3 Laufzeitengpässe

Betrachtet man die Laufzeit eines Programms können verschiedene Arten von Engpässen auftreten. In diesem Kapitel werden sowohl I/O- als auch CPU-bezogene Engpässe näher beleuchtet.

Als I/O-Engpass wird ein Laufzeitproblem bezeichnet, bei dem ein Programm mehr Zeit damit verbringt auf Ein- oder Ausgaben zu warten als mit der Verarbeitung der Informationen. Beispiel für einen I/O-Engpass stellt ein Web-Browser dar. Dieser benötigt signifikant mehr Zeit für das Laden von Webinhalten wie beispielsweise JavaScript und CSS Dateien als das Ausführen von

selbigen. Wenn also die Rate, mit der Daten angefordert werden, langsamer ist als die Geschwindigkeit, mit der sie verarbeitet werden, besteht ein I/O-Engpass [6].

Neben dem I/O Engpass kann auch ein CPU-Engpass auftreten. Dieser entsteht, wenn die Geschwindigkeit, mit der Daten verarbeitet werden deutlich geringer ist als die Rate, mit der Daten bereitgestellt werden. Wird beispielsweise ein Programm auf einer Umgebung mit einer deutlich schnelleren Recheneinheit ausgeführt, so sollte sich die Laufzeit reduzieren. Dieser Engpass steht also in direkter Relation zur Rechengeschwindigkeit des Prozessors [6]. Besteht ein solcher CPU-Engpass, kann eine parallele Verarbeitung von Datensätzen den Engpass beheben. Im nächsten Kapitel wird deshalb auf die Begrifflichkeit der Gleichzeitigkeit und Parallelität näher eingegangen.

2.4 Gleichzeitigkeit vs. Parallelität

Im Alltag werden die Begriffe Gleichzeitigkeit, Multi-Tasking und Parallelität häufig synonym verwendet, dabei unterscheiden sich die Begriffe deutlich. Gleichzeitigkeit oder auch Multitasking ist im Allgemeinen die Praxis, mehrere Dinge gleichzeitig zu tun, jedoch nicht zwingend parallel. Eine telefonierende Person, die gleichzeitig ein Essen zubereitet, betreibt Multitasking. Hingegen ist das gleichzeitige Telefonieren von zwei Angestellten eines Call-Centers eine parallele Verarbeitung. Werden also Aufgaben von mehrere Personen unabhängig voneinander abgearbeitet, so spricht man von paralleler Verarbeitung. Gibt es jedoch mehrere Aufgaben, an denen eine Person gleichzeitig arbeitet, so ist die Rede von Multi-Tasking [6].

Bezogen auf Software liegt der Unterschied zwischen Gleichzeitigkeit (Concurrency) und Parallelität (Parallelism) darin, dass in parallelen Programmen mehrere Verarbeitungsströme (z. B. Prozesse) unabhängig voneinander und zur selben Zeit arbeiten, während bei einer gleichzeitigen Verarbeitung mehrere Verarbeitungsströme (z. B. Threads) auf eine gemeinsame Ressource zugreifen und diese nutzen [10].

Sowohl für eine gleichzeitige als auch parallele Ausführung von Operationen stellt die Programmiersprache Python verschiedene Module bereit, welche im nächsten Kapitel näher erläutert werden.

2.5 Gleichzeitigkeit und Parallelität in Python

Nachfolgend wird auf verschiedenen Möglichkeiten zur gleichzeitigen Verarbeitung von Operationen in einer Python-Anwendung und deren Vor- und Nachteile eingegangen.

2.5.1 Multi-Threading

In der Computerwissenschaft wird ein Thread als die kleinste Code-Einheit bezeichnet, die ein Scheduler verarbeiten bzw. managen kann. Als Scheduler bezeichnet man die Logik, welche die zeitliche Ausführung von Prozessen in ei-

nem Betriebssystem steuert. Ein Thread existiert innerhalb eines Prozesses und hat somit Zugriff auf dessen Ressourcen wie beispielsweise den Speicher oder auch den Kontext [10, S. 44–45]. Als Multi-Threading wird die Ausführung von mehreren Threads innerhalb eines Prozesses bezeichnet. Dies kommt dadurch zustande, dass Prozessoren bzw. Prozessorkerne in sehr kurzer Zeit zwischen einzelnen Threads hin und her wechseln können. Dieser Mechanismus ermöglicht, dass mehrere Aufgaben gleichzeitig ausgeführt werden können und so die Gesamtlaufzeit eines Programmes verkürzt werden kann. Eine solche Aufteilung von verschiedenen Aufgaben auf mehrere Threads ist beispielsweise bei einem I/O-bezogenen Engpass wie in Kapitel 2.3 beschrieben vorteilhaft. Durch die Verarbeitung der Ein- bzw. Ausgabe in einem eigenen Thread, können weitere Aufgaben abgearbeitet und so die Gesamtlaufzeit eines Programms verkürzt werden [6, S. 9].

Allerdings birgt die gleichzeitige Verarbeitung von Aufgaben in mehreren Threads auch Nachteile. Durch den Zugriff mehrerer Threads auf geteilte Ressourcen kann es zu unvorhersehbaren Konflikten wie beispielsweise Deadlocks und Race-Conditions kommen. Um dies zu verhindern, muss eine Synchronisation der Lese- und Schreiboperationen auf eine geteilte Ressource erfolgen[10, S. 47].

Um unter anderem einen sicheren Zugriff mehrerer Threads auf eine gemeinsam genutzte Ressource zu gewährleisten, wurde der Global Interpreter Lock (GIL) in Python eingeführt. Der GIL ist ein Mutual Exclusion Lock, der verhindert, dass mehrere Threads parallel CPU-gebundene Operationen, wie beispielsweise die Veränderung einer Variable, ausführen dürfen. Bevor ein Thread eine Operationen ausführen darf, muss dieser einen Lock erwerben, wordurch garantiert wird, dass keine andere Operation in weiteren Threads ausgeführt wird. Der Vorteil von Multi-Threading liegt somit lediglich in der parallelen Ausführung von nicht CPU-gebundenen Operationen, zu denen unteranderem I/O-gebundene Aufgaben wie beispielsweise das Lesen einer Datei von der Festplatte zählt [10, S. 278–284].

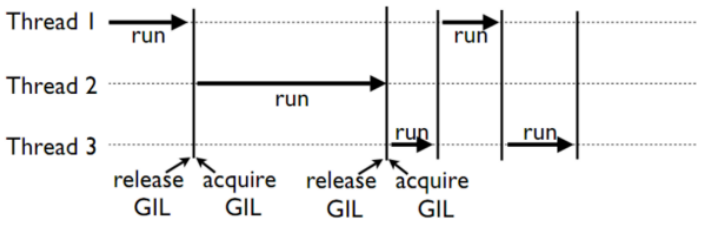


Abbildung 2: Funktionsweise des Python Global Interpreter Locks [6, S. 21]

Trotz der Limitation durch den GIL bestehen in Python mehrere Möglichkeiten, CPU-gebundene Aufgaben zu parallelisieren. So erlaubt beispielsweise die CPython Implementierung des Interpreters die Verwendung von sogenannten Erweiterungsmodulen, welche ebenfalls in C geschrieben werden. Damit ist

neben der Implementierung neuer Objekttypen auch die Verwendung von Bibliotheksfunktionen und Systemaufrufen möglich. Dies macht sich beispielsweise das Modul NumPy zu Nutze, welches zu den elementaren Modulen für wissenschaftliche Datenverarbeitung zählt. NumPy bietet eine Vielzahl an laufzeitoptimierten Funktionen, in welchen eine manuelle Freigabe des GILs vorgenommen wird und somit Berechnungen parallel durchführen kann [10, S. 287].

Festzuhalten ist, dass trotz der strikten Sequentialisierung von gleichzeitigen Operationen über mehrere Threads durch den GIL, die Verwendung des Multi-Threading Ansatzes bei einem IO-Engpass eine positive Auswirkung auf die Gesamtlaufzeit einer Python Anwendung herbeiführen kann. Eine weitere Möglichkeit, Operationen parallel auszuführen ist die Implementierung eines Multi-Processing Ansatzes anstelle von Multi-Threading. Dieses Vorgehen wird im folgenden Kapitel näher betrachtet.

2.5.2 Multi-Processing

Da der GIL nur die parallele Ausführung mehrerer Threads verhindert, spielt dieser bei der Ausführung von Prozessen keine Rolle. Ein Prozess ist im Vergleich zu einem Thread eine größere Programmeinheit und kann je nach Betriebssystem einen oder mehrere Threads enthalten. Während Threads auf geteilte Ressourcen zugreifen und somit untereinander kommunizieren können, besteht bei Prozessen lediglich die Möglichkeit sich über systemweite Prozesskommunikation auszutauschen. Dies hat jedoch den Vorteil, dass im Vergleich zu Threads keine Verwaltung der Speicherzugriffe stattfinden muss, da jeder Prozess seinen eigenen Speicherbereich zugewiesen bekommt.

Werden mehrere Prozesse zur parallelen Verarbeitung von voneinander unabhängigen Operationen verwendet, so spricht man von Multi-Processing. Führt man das Programm auf einer Hardware mit mehreren Prozessorkernen aus, so können die Prozesse auf alle zur Verfügung stehenden Kerne verteilt und damit parallel ausgeführt werden, wodurch eine Optimierung der Laufzeit des Programms erzielt werden kann [10, S. 105–107, 285].

Bei der Implementierung von Multi-Processing sollte jedoch beachtet werden, dass die Erzeugung von weiteren Prozessen (Worker-Prozesse) im Vergleich zur Erzeugung von Threads deutlich aufwändiger ist [10, S. 108]. Die in dieser Arbeit verwendete Klasse *multiprocessing.Pool* verwendet zur Erzeugung weiterer Prozesse standardmäßig die "Fork"-Mehtode. Dabei wird eine identische Kopie des Hauptprozesses inklusive aller Ressourcen erzeugt. Dieser neu erzeugte Prozess bekommt einen eigenen Adressbereich und eine Kopie der Daten des Hauptprozesses [6, S. 55–56]. Wird die Verarbeitung von Daten beispielsweise auf acht Worker-Prozesse verteilt, vervielfacht sich der Speicherbedarf daher um diese Anzahl.

Neben der als Standard für Unix Systeme definierten Startmethode „Fork" bietet das *mutliprocessing* Modul zudem die Methoden „Spawn" und „Forkserver". Bei Verwendung der „Spawn" Methode wird anstatt der Duplizierung des aktuellen Hauptprozesses ein neuer Prozess mit einer eigenen Python Interpreter Instanz erzeugt und das aktuell verwendete Python Modul geladen. Zudem

werden nur die Ressourcen an den neuen Prozess vererbt, die zur Ausführung der Funktion `Process.run()` benötigt werden. Somit benötigen die mit dieser Methode erzeugten Worker-Prozesse deutlich weniger Arbeitsspeicher im Vergleich zur „Fork" Methode. Allerdings ist dieser Vorgang laut offizieller Python Dokumenation deutlich langsamer als die „Fork" Methode [9]. Die dritte Startmethode „Forkserver" stellt eine spezielle Art der Prozessbereitstellung dar. Zu Beginn wird dabei ein Server-Prozess gestartet, welcher anschließend dupliziert wird, sobald ein neuer Worker-Prozess benötigt wird. Dieser Mechanismus ist jedoch nur auf ausgewählten Unix Plattformen verfügbar, welche die Übergabe von Dateideskriptoren über Unix-Pipes unterstützen [6, S. 192]. Aufgrund dieser speziellen Voraussetzung wird diese Methode nachstehend nicht untersucht.

2.6 Amdahlsches Gesetz

Das Amdahlsche Gesetz drückt die maximal zu erwartende Laufzeitverbesserung eines Systems aus, wenn ein Teil des Systems parallel ausgeführt wird. Im Bezug auf parallele Datenverarbeitung kann so die theoretisch maximale Geschwindigkeitssteigerung bei Verwendung mehrerer Prozessoren vorhergesagt werden [11]. Dies hilft bei der Entscheidung, ob weitere Ressourcen (z. B. Prozessoren) hinzugefügt, oder vielmehr die Optimierung von sequentiellen Programmteilen verfolgt werden soll, um die Laufzeit eines Programms zu beschleunigen. Grundlage des Gesetzes ist die Annahme, dass ein Programm sowohl aus parallelisierbaren als auch sequentiell abzuarbeitenden Teilen besteht. Ein Programm kann dadurch nie schneller werden, als die Ausführung des sequentiellen Teils in Anspruch nimmt. Darüber hinaus kann aus der Anwendung des Gesetzes abgeleitet werden, dass mit zunehmender Anzahl an Ressourcen, die Geschwindigkeitssteigerung abnimmt, was sich mit dem steigenden Verwaltungsaufwand von Ressourcen wie z. B. Prozessen begründen lässt.

Das Amdahlsche Gesetz stellt einen Spezialfall des Ertragsgesetzes dar. Dieses besagt, dass bei Hinzunahme von weiteren Produktionsfaktoren der Gesamtertrag zuerst überproportional und anschließend unterproportional zunimmt, bis ein Maximum erreicht ist. Nach diesem Maximum nimmt der Ertrag ab [18]. Bringt man die verschiedenen Verbesserungen eines Programms jedoch in eine optimale Reihenfolge, so ergibt sich eine monoton abfallende Ausführungszeit wie im Fall des Amdahlschen Gesetztes. Besteht jedoch keine optimale Reihenfolge, so hat das Hinzufügen weiterer Ressourcen ab einem bestimmten Punkt einen negativen Effekt auf die Gesamtlaufzeit [10, S. 35–36].

Zur Berechnung der Geschwindigkeitssteigerung (S) in Abhängigkeit der Anzahl verwendeter Prozessoren (j) kann folgende Formel verwendet werden, wobei B den Anteil parallel auszuführender Programmteile bezeichnet.

$$S = \frac{1}{(1 - B) + \frac{B}{j}} \tag{1}$$

Betrachtet man ein Programm mit einem parallelisierbaren Anteil von 40%, das auf einer Hardware mit vier Prozessoren ausgeführt wird,

$$S = \frac{1}{(1 - 0.4) + \frac{0.4}{4}} = \frac{10}{7} \approx 1.43 \tag{2}$$

und setzt man für die Anzahl an Ressourcen $j = \{1, 4, 8, ..., 128\}$ ein, so erhält man folgenden Verlauf der Geschwindigkeitssteigerung.

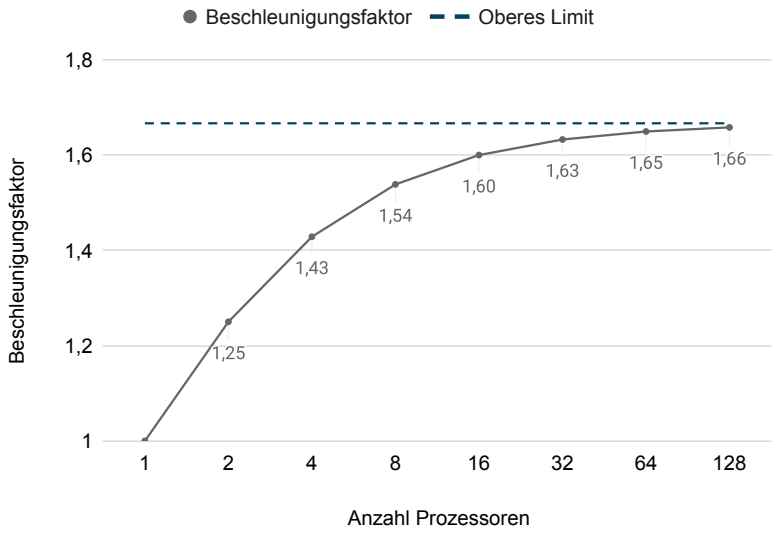


Abbildung 3: Geschwindigkeitssteigerung in Abhängigkeit der Ressourcenanzahl

In Abbildung 3 wird verdeutlicht, dass sich die Geschwindigkeitssteigerung, durch Hinzufügen weiterer Ressourcen zwar erhöht, jedoch jede einzelne Steigerung geringer als die Vorherige ausfällt. Im Bezug auf das obere Limit $\lim_{j \to \infty} S = \frac{1}{B}$ ergibt sich für das obige Beispiel das Limit $S = \frac{1}{0.6} = 1.\overline{6}$, das besagt, dass die Anwendung im Vergleich zu einer sequentiellen Ausführung maximal um den Faktor $1.\overline{6}$ beschleunigt werden kann [10, S. 32–35].

3 Projektumfeld

Während der Entwicklung der Simulationssoftware AMIRIS, einer agentenbasierten Simulation des deutschen Elektrizitätsmarktes, erkannten die Wissenschaftler des Deutschen Zentrums für Luft- und Raumfahrt (DLR), dass die Simulation von Entscheidungsprozessen von beispielsweise Energieversorgungsunternehmen erhebliche Rechenleistung erfordert. Eine parallele Verarbeitung solch komplexer Simulationen kann diese deutlich beschleunigen, erfordern jedoch ein tiefes Verständnis der Bibliotheken und Verfahren für paralleles Rechnen, um z. B. Race Conditions zu vermeiden. Die Wissenschaftler des DLR entwickelten daher das Framework FAME, um die Implementierung einer effizienten Parallelisierung von der fachlichen Modellierung eines Simulationsszenarios zu entkoppeln. FAME (öpen Framework for distributed Agent-based Modelling of Energy systems") besteht aus den Komponenten FAME-IO zur Umwandlung der Simulationskonfiguration in das von Google entwickelte sprachunabhängige Serialisierungsformat Protocol Buffer [14] und FAME-Core zur parallelen Durchführung der Simulation in Java. Das Zusammenspiel der Komponenten von FAME veranschaulicht Abbildung 1.

Diese Arbeit analysiert die Möglichkeiten zur Beschleunigung der Serialisierung von Simulationskonfiguration in FAME-IO. Dabei gilt es die in einer Simulationskonfiguration enthaltene Anzahl sowohl an Agenten als auch Einträgen einer Zeitreihe zu beachten. Als Agent wird eine Konfigurationseinheit bezeichnet, die einen Akteur innerhalb des zu simulierenden Energiesystems beschreibt. Dies kann z. B. ein Stromproduzent, ein Handelsplatz oder auch ein Stromverbraucher sein. Damit die Eigenschaften eines Agenten, wie beispielsweise die produzierte Menge an Energie, im zeitlichen Verlauf der Simulation veränderbar sind, enthält eine Simulationskonfiguration sogenannte Zeitreihen. Eine Zeitreihe besteht aus einer eindeutigen ID und einer Anzahl von Zeitreiheneinträgen. Jeder Zeitreiheneintrag setzt sich aus einem textuellen Zeitstempel im Format `YYYY-mm-dd_HH:MM:SS` [4] und einem numerischen Wert zusammen. Dabei gibt der Zeitstempel den Zeitpunkt an, ab dem das entsprechende Attribut des Agenten den Wert des Zeitreiheneintrages annimmt. Somit ist die Anzahl der Einträge einer Zeitreihe abhängig von der Schwankung des Agentenattributs und der Länge des zu simulierenden Zeitraumes. Gespräche mit Wissenschaftlern des DLR ergaben, dass Simulationskonfigurationen typischerweise zwischen 10^1 und 10^5 Agenten sowie 10^3 bis 10^7 Zeitreiheneinträge enthalten. Diese Größenordungen sowohl an Agenten als auch Zeitreiheneinträgen stecken daher den Rahmen für die nachfolgend erläuterten Optimierungen ab.

4 Ausgangszustand

In diesem Kapitel wird der Ist-Zustand von FAME-IO im Bezug auf die Laufzeit analysiert. Hierfür werden die in Kapitel 2.2 beschriebenen Hilfsmittel eingesetzt, um eine Aussage über die Gesamtlaufzeit der Komponente in Abhängigkeit von der Agentenanzahl und der Länge der Zeitreihen zu erhalten. Darüber hinaus werden mittels eines Profilers die Programmteile identifiziert, die in Relation zur Gesamtdauer am meisten Zeit in Anspruch nehmen und somit im Fokus für Optimierungen stehen.

4.1 Laufzeitanalyse

Um ermitteln zu können, ob ein Optimierungsversuch zu einer Verkürzung der Laufzeit führt, muss die Ausgangsgeschwindigkeit als Vergleichswert ermittelt werden. Hierfür wird das in Kapitel 2.2 beschriebene Modul *timeit* verwendet. Dieses bietet die Möglichkeit, einen Ausdruck oder auch eine Funktion auszuführen und dessen Laufzeit zu messen. Da die Funktion nicht auf einem isolierten System ausgeführt wird, auf dem keine weiteren Prozesse laufen, unterliegt die Laufzeit einer gewissen Schwankung. Aus diesem Grund wurde das *timeit* Modul mit einer Anzahl an Ausführungen parametrisiert, wodurch der Durchschnittswert und die Standardabweichung aller Ausführungen ermittelt werden kann. Bezogen auf FAME-IO wurde die Laufzeit anhand von 20 Wiederholungen gemessen.

Da der Fokus dieser Arbeit auf der Laufzeitoptimierung sowohl in Abhängigkeit der Agentenanzahl als auch der Anzahl an Einträgen einer Zeitreihe liegt, wird die Ausgangsgeschwindigkeit in Abhängigkeit dieser Werte ermittelt. Für die Laufzeitmessung in Bezug auf die Agentenanzahl werden die Werte 10^1, 10^2, ... ,10^5 herangezogen, für die Anzahl der Zeitreiheneinträge werden die Werte 10^3, 10^4, ..., 10^7 verwendet. Diese Werte entsprechen den in realistischen Simulationsszenarien verwendeten Größenordnungen und decken somit den Hauptteil der Anwendungsbereiche ab. Simulationskonfigurationen außerhalb dieser Größenordnungen stellen einen Sonderfall dar und werden in dieser Arbeit nicht berücksichtigt.

Durchgeführt werden die Messungen auf einer Elastic Compute Cloud (EC2) Instanz des Cloud-Anbieters Amazon Web Services (AWS). Dabei handelt es sich um eine Virtuelle Maschine, die von AWS bereitgestellt wird. Dem Nutzer stehen eine Vielzahl an Konfigurationsmöglichkeiten zur Verfügung. Über die Auswahl des Instanztyps kann unter anderem definiert werden, mit wie vielen Prozessorkernen die virtuelle Maschine bereitgestellt wird. Für den hier vorliegenden Anwendungsfall wird das Image m5.2xlarge gewählt, wodurch der virtuellen Maschine acht virtuelle Prozessorkerne und 32 GiB Arbeitsspeicher zugewiesen werden. Bei der damit bereitgestellten Hardware handelt es sich um Intel Xeon® Platinum 8175M-Prozessoren mit bis zu 3,1 GHz [1]. Durch die Verwendung einer virtuellen Maschine kann im Vergleich zu einem herkömmlichen Desktop PC der Einfluss von nebenläufigen Prozessen auf die Messung minimiert werden. Dem gegenüber steht der fehlende Einfluss auf die konkret verwendete

Hardware. Zwar gibt AWS an, welche Prozessoren verwendet werden und welche maximale Geschwindigkeit erreicht werden kann, allerdings handelt es sich bei den Angaben um Maximalwerte, welche in der Praxis auch niedriger ausfallen können. Dies hat zur Folge, dass die Messewerte über die durchgeführten Messreihen hinweg schwanken.

Für das Durchführen von Messungen wurde ein Hilfsskript entwickelt, welches ein definiertes Set an Konfigurationsszenarien ausführt. Die Konfigurationsszenarien unterscheiden sich ausschließlich in der Anzahl von Agenten und Zeitreiheneinträgen. Für jedes Konfigurationszenario wird die FAME-IO Funktion
`make_config()` eine zuvor definierte Anzahl an Wiederholungen aufgerufen und deren Laufzeiten in einer Liste zurückgegeben. Der Durchschnittswert aller Einzellaufzeiten wird ermittelt. Das Diagramm in Abbildung 4 visualisiert die Ergebnisse der Laufzeitmessung des Ausgangszustandes von FAME-IO. Sowohl bei dieser als auch bei allen folgenden Diagrammen werden basierend auf der Standardabweichung die gesicherten Stellen des Mittelwerts dargestellt.

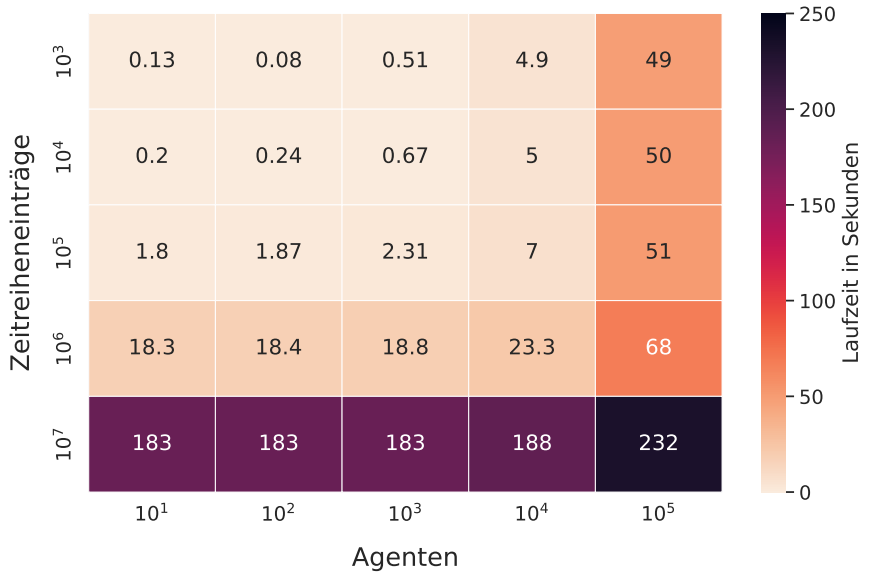


Abbildung 4: Laufzeitmittelwerte von FAME-IO im Ausgangszustand in Abhängigkeit der Agentenanzahl und Anzahl an Zeitreiheneinträgen

Abbildung 5 zeigt die Gesamtlaufzeit der Simulationskonfigurationen mit 10^1 Agenten in Abhängigkeit der Anzahl an Zeitreiheneinträgen. Dabei ist bei steigender Anzahl an Zeitreiheneinträgen ein linearer Anstieg der Gesamtlaufzeit zu beobachten, was auf eine lineare Komplexität des Verarbeitungsalgorithmus' schließen lässt. Der rot eingefärbte Bereich stellt die Abweichung des der Messergebnisse vom Mittelwert dar.

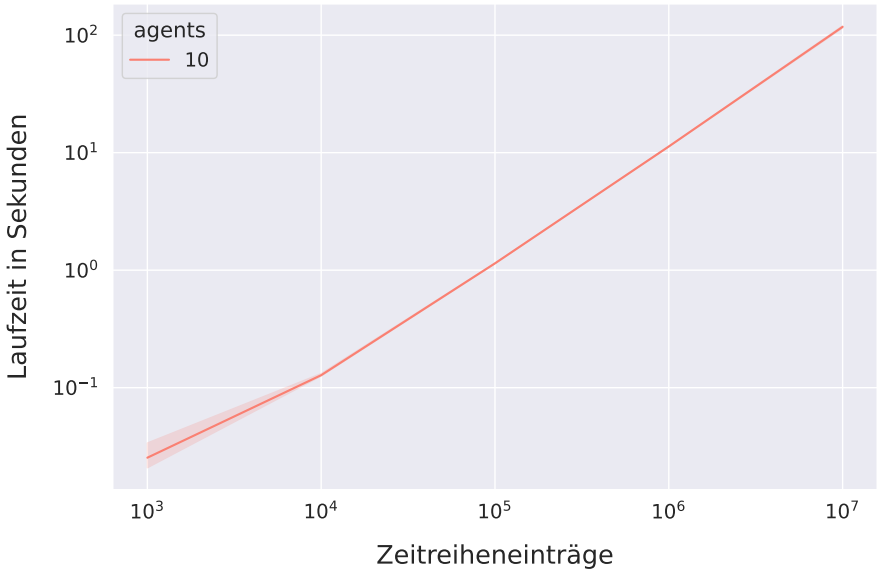


Abbildung 5: Laufzeitmittelwerte und Standardabweichung des Ausgangszustands von FAME-IO in Abhängigkeit der Anzahl an Zeitreiheneinträgen bei 10^1 Agenten

4.2 Profiling

Die Laufzeitmessung des Ausgangszustandes in Kapitel 4.1 hat gezeigt, dass die Verarbeitung der Zeitreiheneinträge einen hohen Anteil der Gesamtlaufzeit von FAME-IO ausmachen kann. In diesem Kapitel wird daher der Programmteil, der für die Verarbeitung der Zeitreiheneinträge zuständig ist, näher analysiert. Dafür wird der in Kapitel 2.2 beschriebene Profiler *cProfile* eingesetzt.

Dieser führt FAME-IO mit einer Simulationskonfiguration mit 100 Agenten und 100.000 Zeitreiheneinträgen aus. Dabei liegt der Fokus auf der Verteilung der Gesamtlaufzeit auf die aufgerufenen Funktionen. Jede Funktion wird als Box dargestellt und abhängig von ihrem prozentualen Anteil an der Gesamtlaufzeit farblich markiert. Rot eingefärbte Funktionen weisen auf einen hohen, grün eingefärbte Funktionen auf einen geringen Anteil an der Gesamtlaufzeit hin. Es wird für jede aufgerufene Funktion zwischen der Gesamtzeit (Total) und dem Eigenanteil (Own) unterschieden. Die Gesamtzeit gibt an, wie lange die Ausführung der Funktion, inklusive der Dauer aller weiteren, innerhalb der Funktion aufgerufenen Funktionen, in Anspruch nimmt. Der Eigenanteil einer Funktion gibt die Laufzeit an, die die Ausführung der funktionseigenen Statements exklusive der Laufzeit aufgerufener Funktionen benötigt. Um die Gesamtlaufzeit eines Programms zu optimieren, müssen Funktionen mit einem hohen Eigenanteil beschleunigt werden.

Abbildung 6 zeigt die Ergebnisse des Profilings und den Teil von FAME-IO, der am meisten Laufzeit in Anspruch genommen hat. Die Gesamtlaufzeit der

Einstiegsfunktion `make_config()` beläuft sich auf etwa 23 Sekunden. Grob 88% der Gesamtlaufzeit der Anwendung entfallen auf die Umwandlung der Zeitreiheneinträge zu Protobuffer Objekten in der Funktion `_add_rows_to_series()`. In der Funktion `_convert_to_datetime()` wird jeder textuelle Zeitstempel eines Zeitreiheneintrages mit Hilfe des Python-Moduls "datetimeïn ein DateTime Objekt umgewandelt, was insgesamt etwa 55% der Gesamtlaufzeit in Anspruch nimmt.

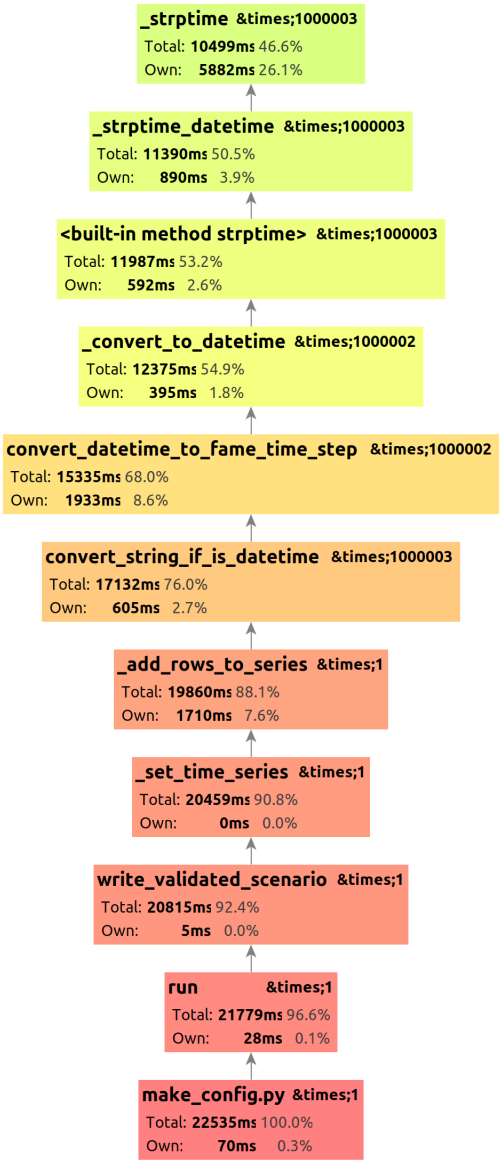


Abbildung 6: Profiling-Ergebnis für FAME-IO im Ausgangszustand mit 10^2 Agenten und 10^5 Zeitreiheneinträgen: Funktionspfad mit längster Laufzeit

Neben dem Profiling der Funktionsaufrufe ist auch die Betrachtung des Speicherbedarfs, auch Memory Profiling genannt, ein wesentlicher Bestandteil der Analyse des Ausgangszustandes. Zur Ermittlung des benötigten Arbeitsspeichers wird das Python Modul *memory_profiler* [12] eingesetzt. Betrachtet wird das Simulationsszenario mit 10^5 Agenten und 10^7 Zeitreiheneinträgen, da dieses

Szenario bezogen auf die Dateigrößen der Konfigurationsdateien am größten ist. Abbildung 7 zeigt den Speicherbedarf, den FAME-IO zur Umwandlung des zuvor beschriebenen Simulationsszenarios benötigt. Der rot eingefärbte Bereich zeigt den benötigten Arbeitsspeicher der Funktion `load_yaml()`, welche die beim Aufruf der Funktion `make_config()` übergebene Szenariokonfiguration einliest. Diese Datei enthält unter anderem sämtliche Agentendefinitionen, was sich im Anstieg des Speicherbedarfs widerspiegelt. Der orange markierte Bereich visualisiert den benötigten Arbeitsspeicher innerhalb der Funktion `_set_time_series()`, welche die benötigten Zeitreihen und deren Einträge sequenziell in Protocol Buffer Objekte umwandelt. Aufgrund der Vielzahl an Zeitreiheneinträgen in dem verwendeten Simulationsszenario steigt der Speicherbedarf auf über 4000 MiB an. Dieses Maximum visualisiert die rot gestrichelte Linie.

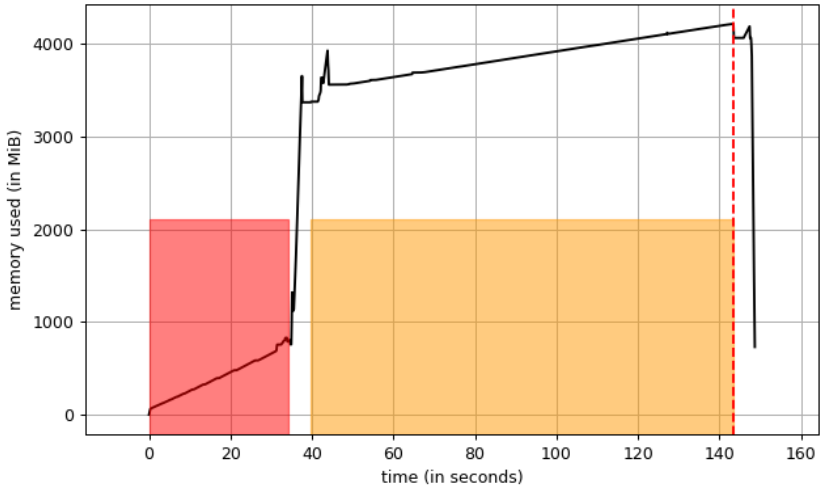


Abbildung 7: Memory-Profiling-Ergebnis für FAME-IO im Ausgangszustand mit 10^5 Agenten und 10^7 Zeitreiheneinträgen. Hervorhebung des Speicherbedarfs zum Einlesen der Simulationskonfiguration (rot) und der Umwandlung von Zeitreiheneinträgen in Protocol Buffer Objekte (orange)

4.3 Erkenntnisse

Bei der Code-Analyse des Ausgangszustandes von FAME-IO zeigt sich, dass die Anwendung keine Mechanismen zur gleichzeitigen Verarbeitung von Daten wie in Kapitel 2.5 näher beschrieben, aufweist. eine sequentielle Verarbeitung der Daten hat bei einer hohen Anzahl an Agenten oder vielen Zeitreiheneinträgen eine direkte Auswirkung auf die Gesamtlaufzeit des Programms. Dieses Verhalten spiegelt sich auch in der Laufzeitanalyse in Kapitel 2.2 wieder. Eine Beschleunigung des Programmablaufs durch den Einsatz mehrerer Prozesse zur parallelen Verarbeitung ist somit bisher nicht möglich.

Darüber hinaus zeigt das Ergebnis des Profilings von FAME-IO, dass die

Umwandlung von textuellen Zeitstempeln zu DateTime Objekten innerhalb der Funktion `_convert_to_datetime()` mit 55% die laufzeitkritischste einzelne Programmfunktion darstellt.

5 Optimierungsansätze

Nachdem in Kapitel 4 der Ausgangszustand identifiziert und analysiert wurde, geht das folgende Kapitel auf verschiedene Optimierungsansätze ein, die zu einer Verbesserung der Gesamtlaufzeit von FAME-IO führen sollen.

5.1 Parallele Umwandlung von Zeitreiheneinträgen

Ein erster Ansatz zur Optimierung der Laufzeit von FAME-IO ergibt sich aus der Analyse des Ausgangszustands in Kapitel 4. Dabei ist aufgefallen, dass die Einträge einer Zeitreihe sequentiell in der Funktion `_add_rows_to_series()` zu Protobuffer Objekten umgewandelt werden. Eine `for` Schleife erzeugt für jede Zeile des übergebenen DataFrames ein neues Protobuffer Objekt vom Typ `Row`, das anschließend dem übergeordneten `TimeSeries` Objekt hinzugefügt wird.

Zur Optimierung dieser Umwandlung eignet sich der in Kapitel 2.5.2 beschriebene Multi-Processing Ansatz. Dabei wird die Verarbeitung der Daten auf mehrere Prozesse aufgeteilt, welche auf einer Mehrkern-Rechnerarchitektur parallel ausgeführt werden können. Im Vergleich zu einem Multi-Threading Ansatz entsteht beim Multi-Processing keine Sequenzialisierung durch den GIL, wie in Abbildung 2 dargestellt. Zudem handelt es sich bei der Umwandlung von Zeitreiheneinträgen um einen CPU-gebundenen Engpass, welcher durch die Anwendung eines Multi-Processing Ansatzes reduziert werden kann.

5.1.1 Zielsetzung

Um eine Abschätzung zu erhalten, wie sich die parallele Verarbeitung von Zeitreiheneinträgen auf die Gesamtlaufzeit von FAME-IO auswirkt, wird das in Kapitel 2.6 beschriebene Amdahlsche Gesetz angewendet. Hierfür wird der prozentuale Anteil des parallel ausgeführten Programmcodes (B) und die Anzahl der verwendeten Prozesse benötigt. Ersteres wird ermittelt, indem die Dauer des parallelisierbaren Programmteils (T_{par}) aus Abbildung 8 durch die Gesamtlaufzeit (T_{ges}) aus Abbildung 4 dividiert wird.

Abbildung 9 zeigt den Laufzeitanteil des parallelisierbaren Programmanteils an der Gesamtprogrammdauer in Abhängigkeit der Agentenanzahl und der Anzahl der Zeitreiheneinträge.

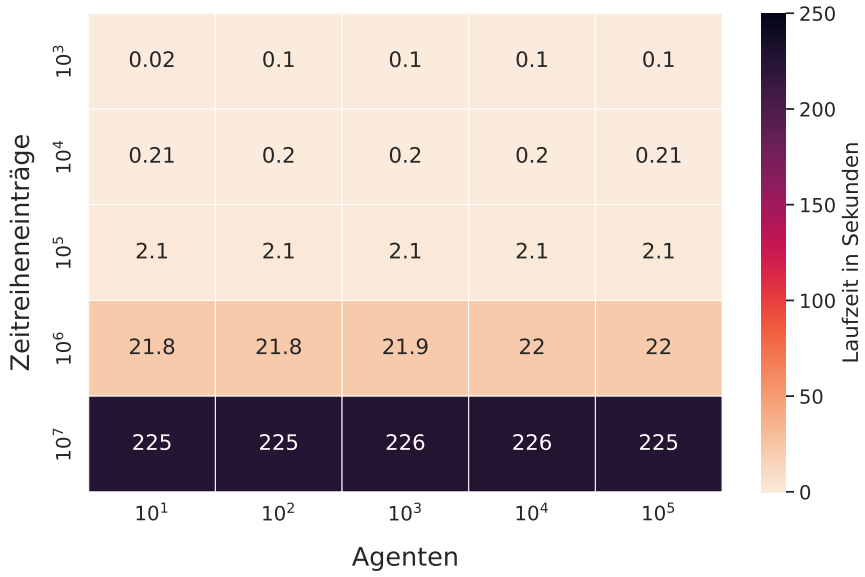


Abbildung 8: Laufzeit des parallel ausführbaren Programmcodes

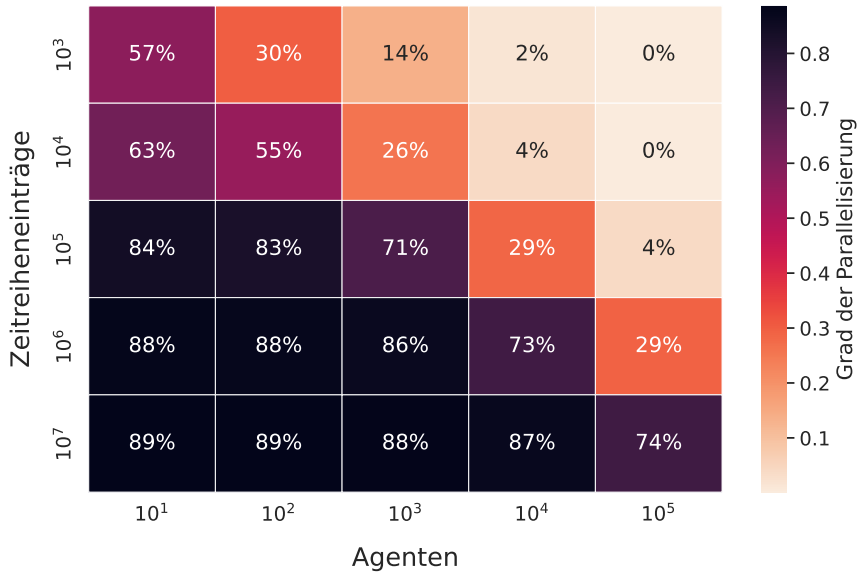


Abbildung 9: Prozentualer Laufzeitanteil des zu parallelisierenden Programmcodes an der Gesamtprogrammlaufzeit

Sowohl in Abbildung 8 als auch in Abbildung 9 wird deutlich, dass die Verarbeitung der Zeitreiheneinträge für die Mehrzahl der betrachteten Szenariokonfigurationen einen dominanten Anteil an der Gesamtlaufzeit des Programms einnimmt. Aus diesem Grund liegt der Fokus der weiteren Arbeit auf der optimierten Verarbeitung der Zeitreiheneinträge. Die Anzahl der Agenten wird für alle weiteren Betrachtungen auf 10^1 festgelegt.

Wendet man die in Kapitel 2.6 beschriebene Formel 1 an, so ergibt sich für die jeweilige Kombination aus Zeitreiheneinträgen und der Anzahl an verwendeten Prozessen der in Abbildung 10 abgebildete theoretische Beschleunigungsfaktor.

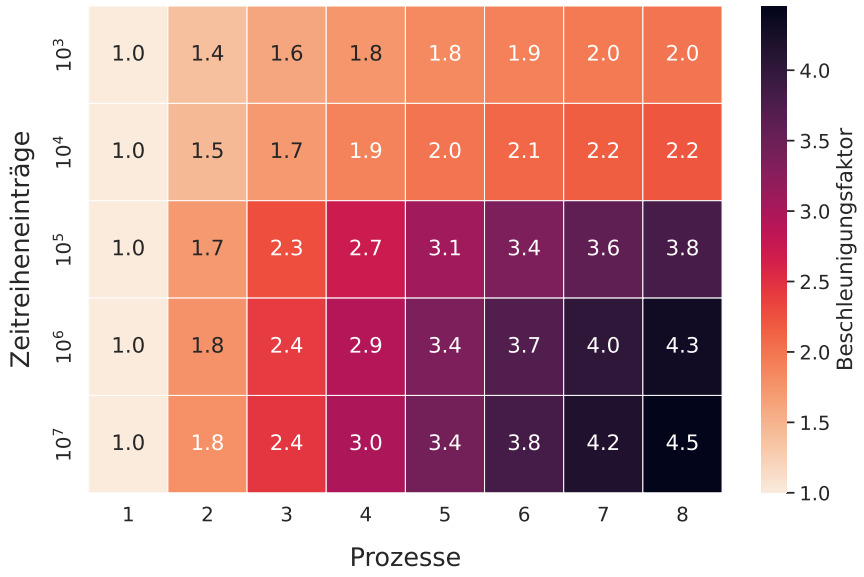


Abbildung 10: Beschleunigungsfaktor nach Amdahlschem Gesetz in Abhängigkeit der Zeitreiheneinträge und Anzahl verwendeter Prozesse bei Verwendung von 10^1 Agenten

Bei der Verwendung von acht Prozessen ist demzufolge im Optimum eine Beschleunigung der Gesamtlaufzeit um den Faktor $4,5$ zu erwarten. Bezogen auf die Ausgangslaufzeit aus Abbildung 4 entspricht dies einer Laufzeit von ≈ 41 Sekunden für das hier betrachtete Simulationsszenario mit 10^1 Agenten und 10^7 Zeitreiheneinträgen.

Die berechneten Laufzeiten in Abhängigkeit von der Anzahl der verwendeten Prozesse stellt Abbildung 11 dar. Diese entsprechen dem Soll-Zustand, welcher nach Implementierung des Multi-Processing Mechanismuses optimalerweise erreicht werden soll.

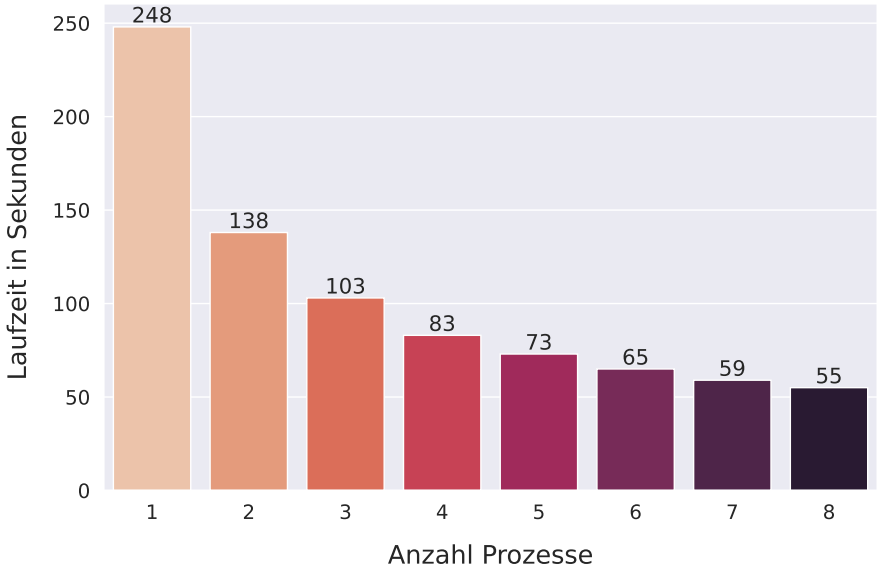


Abbildung 11: Simulierte Laufzeit von FAME-IO nach dem Amdahlschen Gesetz bei Verwendung von 10^1 Agenten und 10^7 Zeitreiheneinträgen

5.1.2 Umsetzung

Um die Verarbeitung der Zeitreiheneinträge auf mehrere Prozesse zu verteilen, bietet Python die Module *multiprocessing.Pool* und *concurrent.futures.ProcessPoolExecutor* an. Diese unterscheiden sich nur in der bereitgestellten Schnittstelle. Während auf einem *multiprocessing.Pool* Objekt verschiedene Varianten von `map()` und `apply()`-Operationen aufgerufen werden können, bietet der *ProcessPoolExecutor* lediglich die Operationen `submit()` zur Übergabe eines einzelnen Tasks an den Executor und `map()` für die Verarbeitung einer Liste von Argumenten an. Damit soll der Einstieg in die gleichzeitige Programmierung für Anwender erleichtert werden [6, S. 199–200]. Für den vorliegenden Anwenungsfall wird aufgrund der umfangreicheren Schnittstelle die *multiprocessing.Pool* Implementierung verwendet.

Bei der Erzeugung eines *multiprocessing.Pools* definiert der Parameter `processes` die Anzahl der Prozesse, die für die gleichzeitige Verarbeitung der Aufgaben verwendet werden soll. Ist keine Prozessanzahl definiert, wird die Anzahl der verfügbaren Prozessorkerne verwendet. Für den hier vorliegenden Anwendungsfall wird die Verteilung auf bis zu acht Prozesse betrachtet.

Damit eine parallele Verarbeitung der Zeitreiheneinträge erfolgen kann, muss die existierende Funktion `_add_rows_to_series` durch eine Funktion ersetzt werden, welche anstatt eines kompletten DataFrames, nur einen einzelnen Zeitreiheneintrag entgegen nimmt und in ein Protobuffer `Row` Objekt umwandelt. Die neue Implementierung ist in Listing 1 zu sehen. Als Parameter werden der tex-

```
1         @staticmethod
2         def _create_row(key, value):
3             row = InputFile_pb2.InputData().TimeSeriesDao().Row()
4             row.timeStep = FameTime.convert_string_if_is_datetime(key)
5             row.value = value
6             return row
```

Listing 1: Funktion create_row()

tuelle Zeitstempel `key` und der zugehörige Wert `value` übergeben.

Anschließend wird innerhalb des *multiprocessing.Pool* Kontextes die Funktion `starmap()` auf dem Pool-Objekt aufgerufen. Diese nimmt eine Funktion und ein `Iterable` von Tupeln als Parameter entgegen. Für den vorliegenden Anwendungsfall wird die Funktion `_create_row()` und ein Iterable des DataFrames übergeben. Letzteres kann durch den Aufruf der Funktion `DataFrame.itertuple()` erzeugt werden. Für jeden Eintrag des `Iterables` wird nun auf einem der zur Verfügung stehenden Prozesse die übergebene Funktion ausgeführt und die Werte des Tupels als Parameter übergeben. Der Rückgabewert jedes Funktionsaufrufs, in diesem Fall ein Objekt vom Typ `Row`, wird in einer Liste gesammelt und nach erfolgreicher Verarbeitung aller Einträge zurückgegeben. Abschließend wird die Liste von `Row` Objekten mit der Funktion `.extend()` dem übergeordneten `TimeSeries` Protobuffer Objekt hinzugefügt.

Die anschließend erneut durchgeführte Laufzeitmessung mit dem Simulationsszenario bestehend aus 10^1 Agenten und 10^7 Zeitreiheneinträgen ergab die in Abbildung 12 abgebildeten Laufzeiten.

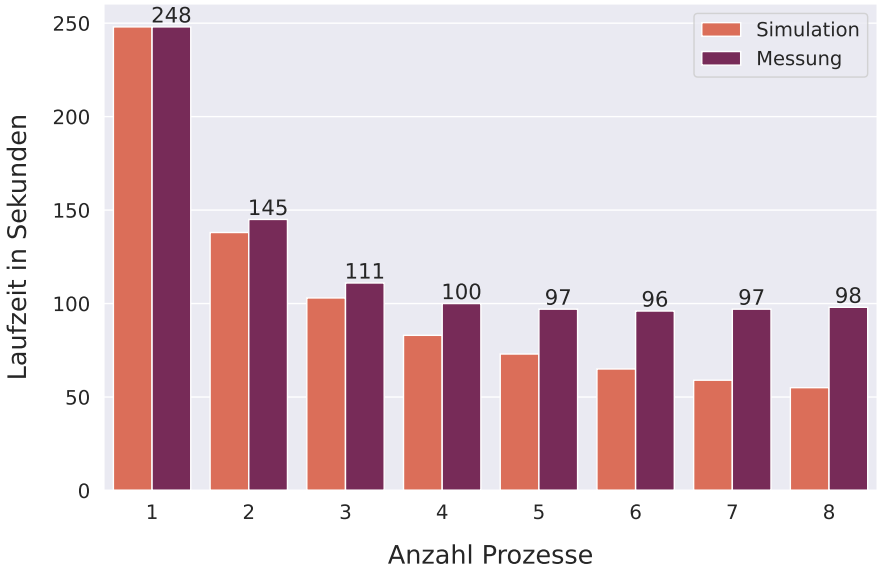


Abbildung 12: Vergleich der simulierten Laufzeit von 10^1 Agenten und 10^7 Zeitreiheneinträgen mit der gemessenen Laufzeit von FAME-IO in Abhängigkeit der verwendeten Prozesse

5.1.3 Validierung

Bei der Betrachtung der Messergebnisse fällt auf, dass die Gesamtlaufzeit bei Verwendung des *multiprocessing.Pool* mit nur einem Prozess im Vergleich zur Laufzeit des Ausgangszustandes in Abbildung 4 deutlich ansteigt. Dauerte die sequentielle Ausführung des Simulationsszenarios mit 10^1 Agenten und 10^7 Zeitreiheneinträgen 232 Sekunden, beträgt sie bei Verwendung eines Worker-Prozesses insgesamt 248 Sekunden. Daraus erschließt sich, dass die Erzeugung und Verwaltung von Prozessen einen nennenswerten laufzeitrelevaten Aufwand mit sich bringt.

Bei Verwendung von acht Prozessen dauert die Ausführung des zuvor genannten Simulationsszenarios durchschnittlich 98 Sekunden. Vergleicht man die tatsächlichen Laufzeiten mit den mittels des Amdahlschen Gesetzes errechneten Werten, so erkennt man, dass die Laufzeit durch Erhöhung der Prozesse zuerst kontinuierlich sinkt, jedoch jeder weitere Prozess eine geringere Laufzeitsteigerung herbeiführt. Bei Verwendung von sechs Prozessen ist die Laufzeit mit 96 Sekunden am niedrigsten. Die Hinzunahme weiterer Prozesse wirkt sich negativ auf die Gesamtlaufzeit aus. Dies lässt sich durch den in Kapitel 2.6 beschriebenen Zusammenhang zwischen dem Amdahlschen Gesetz und dem Ertragsgesetz erklären. Dieser besagt, dass bei nicht optimalen Bedingungen das Hinzufügen von weiteren Produktionsfaktoren, in diesem Fall die Verwendung weiterer Prozesse, ab einem bestimmten Punkt einen negativen Einfluss auf das Ergebnis

hat. Betrachtet man die Laufzeiten in Abbildung 12, so ist zu erwarten, dass sich bei Hinzunahme weiterer Prozesse die Gesamtlaufzeit kontinuierlich erhöht. Dies lässt sich durch den linear steigenden Aufwand zur Erzeugung und Verwaltung der Prozesse erklären, welcher die immer kleiner werdende Beschleunigung übersteigt.

Zum Vergleich der durch Verwendung mehrerer Prozesse erziehlten Laufzeiten mit dem in Kapitel 4 beschriebenen Ausgangszustand zeigt Abbildung 13 die Gesamtlaufzeit von FAME-IO bei Verwendung von sechs Prozessen.

Vergleicht man den Durchschnittswert von 10^1 Agenten und 10^7 Zeitreiheneinträgen aus Abbildung 13 mit der Laufzeit bei Verwendung von sechs Prozessen aus Abbildung 12, so fällt eine Differenz von zwei Sekunden auf. Dieser Unterschied entsteht, da die Messungen wie in Kapitel 2.2 beschrieben, auf einer Cloud-Umgebung durchgeführt werden. Diese sichert lediglich eine maximal erreichbare Rechenleistung zu, garantiert diese jedoch nicht permanent.

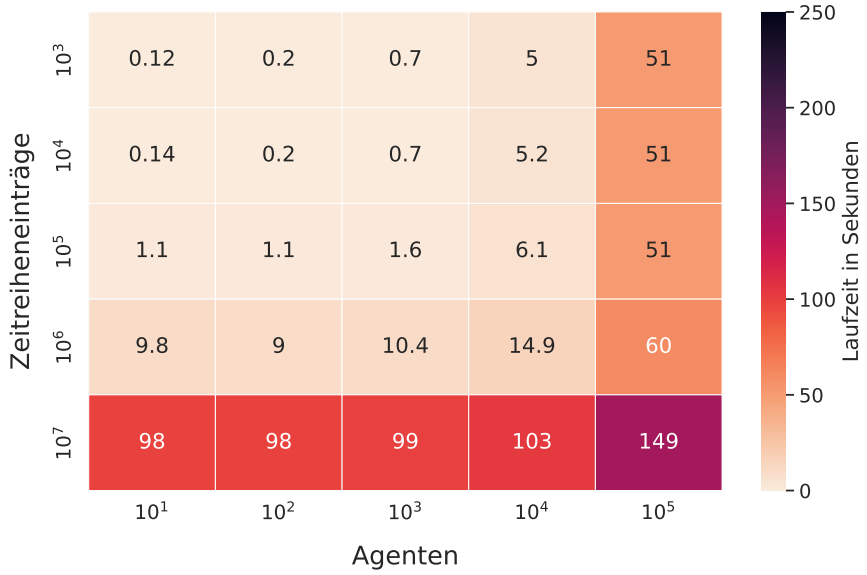


Abbildung 13: Laufzeit von FAME-IO bei Verwendung von sechs Prozessen zur Verarbeitung von 10^1 Agenten und 10^7 Zeitreiheneinträgen

Wie in Kapitel 2.5.2 beschrieben, wird bei Verwendung der *multiprocessing.Pool* Klasse auf Unix Systemen die "Fork" Methode zur Erzeugung von Worker-Prozessen verwendet [9]. Dabei wird der Haupt-Prozess vollständig dupliziert, was einen deutlichen Anstieg des Speicherbedarfs erwarten lässt. Aufgrund dessen wurde ein erneutes Memory Profiling durchgeführt, dessen Ergebnis Abbildung 14 zeigt.

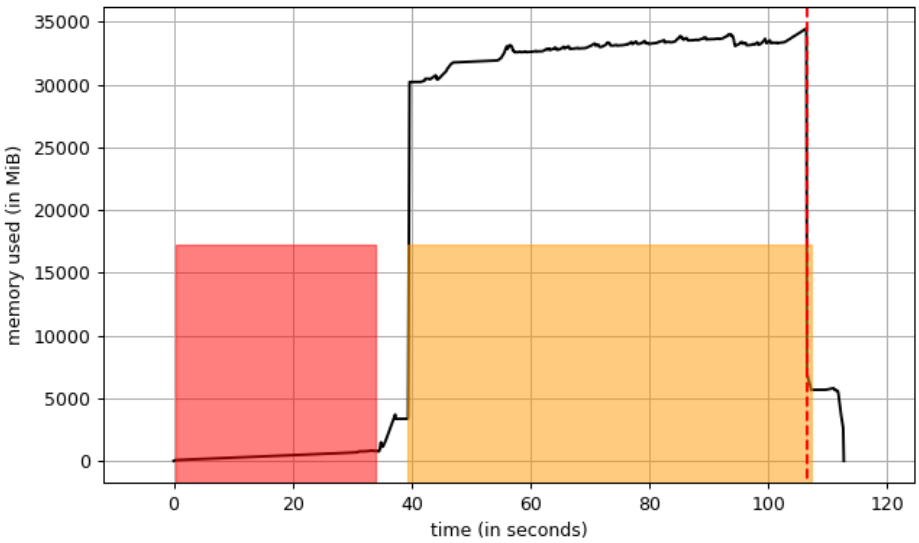


Abbildung 14: Bedarf an Arbeitsspeicher bei Verwendung von acht Worker-Prozessen zur Verarbeitung des Simulationsszenarios mit 10^1 Agenten und 10^7 Zeitreiheneinträgen

Analog zu Abbildung 7 zeigt der in rot bzw. orange eingefärbte Bereich den Speicherbedarf der Funktionen `load_yaml()` bzw. `_set_time_series()`. Betrachtet man den Verlauf des Speicherbedarfs in Abbildung 14, so markiert der rote Bereich den Anstieg durch das Laden und Verarbeiten der Agentenkonfiguration. Zu Beginn des orangen Bereichs steigt der Speicherbedarf von etwa 4 GiB auf fast 35 GiB stark an, was sich auf die Erzeugung der Worker-Prozesse zurückführen lässt.

Um einen derartigen Anstieg des Arbeitsspeichers zu vermeiden, wurde die in Kapitel 2.5.2 beschriebene Startmethode "Spawn" verwendet. Dies führte erwartungsgemäß zu einer deutlichen Reduktion des benötigten Arbeitsspeichers. Bei Verwendung von ebenfalls acht Worker-Prozessen wurde ein maximaler Speicherbedarf von 8179 MiB gemessen. Abbildung 15 stellt den benötigten Arbeitsspeicher bezogen auf die Start-Methoden "Fork" und "Spawn" gegenüber.

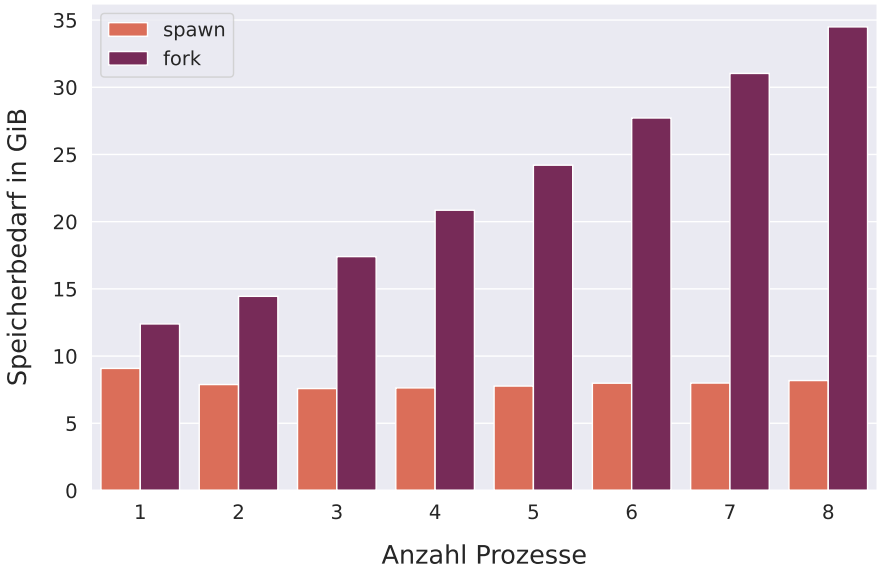


Abbildung 15: Vergleich der Start-Methoden "Fork" und "Spawn" hinsichtlich des maximalen Bedarfs an Arbeitsspeicher während der Programmlaufzeit (Einzelmessung)

Während bei Verwendung der 'Fork' Methode ein linearer Anstieg des Arbeitsspeichers zu erkennen ist, fällt die Auswirkung bei der "Spawn" Methode auf den Arbeitsspeicherbedarf bei etwa gleichbleibender Programmgeschwindigkeit nur gering aus. Betrachtet man den Speicherverbrauch der mittels "Spawn" Methode gestarteten Worker-Prozesse, so fällt auf, dass bei Verwendung eines Worker-Prozesses der Peak des Arbeitsspeichers mit 9.082 MiB höher ist als bei Verwendung von zwei Worker-Prozessen, wo der Bedarf an Arbeitsspeicher in der Spitze 7.881 MiB beträgt. Dies lässt sich mit dem Python Garbage Collecting, also dem Freigeben von nicht mehr benötigtem Speicherplatz, erklären. Bei der Verteilung der zu konvertierenden Daten mittels der Funktion `starmap()` auf die Worker-Prozesse wird die übergebene Liste an Argumenten in sog. Chunks aufgeteilt. Die Größe eines solchen Chunks (Chunksize) kann optional beim Aufruf der Funktion übergeben werden. Ist dies nicht der Fall, berechnet sich die Chunksize (M_{chunk}) aus der Anzahl der übergebenen Argumente (N) und der Anzahl der verwendeten Worker-Prozesse (j) wie folgt:

$$M_{chunk} = \frac{N}{(4 * j)} = \frac{1000}{(4 * 1)} = 250 \tag{3}$$

Jeder Chunk wird anschließend in einem sog. Task an den Worker-Prozess zur Verarbeitung übergeben. Betrachtet man Abbildung 16, welche den Speicherbedarf für die Verarbeitung von 10^5 Agenten und 10^7 Zeitreiheneinträgen bei Verwendung von einem Worker-Prozess darstellt, sind exakt vier Peaks zu er-

kennen. Dies lässt darauf schließend, dass jeder Peak einem Task entspricht. Der nach jedem Peak fallende Bedarf an Arbeitsspeicher deutet darauf hin, dass der in Python enthaltene Garbage Collector den nicht mehr benötigten Speicherplatz freigibt, wodurch der Gesamtbedarf nicht linear steigt. Diese Beobachtung zeigt sich ebenfalls bei Betrachtung von Abbildung 17, welche den Speicherverlauf des identischen Simulationsszenarios wie im vorherigen Abschnitt, jedoch mit zwei Worker-Prozessen darstellt. In Abbildung 17 sind insgesamt acht Peaks zu erkennen, was erneut der Anzahl der Chunks bzw. Tasks entspricht. Da sich bei Verwendung von zwei Worker-Prozessen die Chunksize halbiert, benötigt jeder einzelne Task zur Ausführung weniger Arbeitsspeicher. Das Freigeben des nicht mehr benötigten Speicherplatzes durch den Garbage Collector unmittelbar nach Beendigung des Tasks führt somit zu einer Reduzierung des maximal benötigten Arbeitsspeichers.

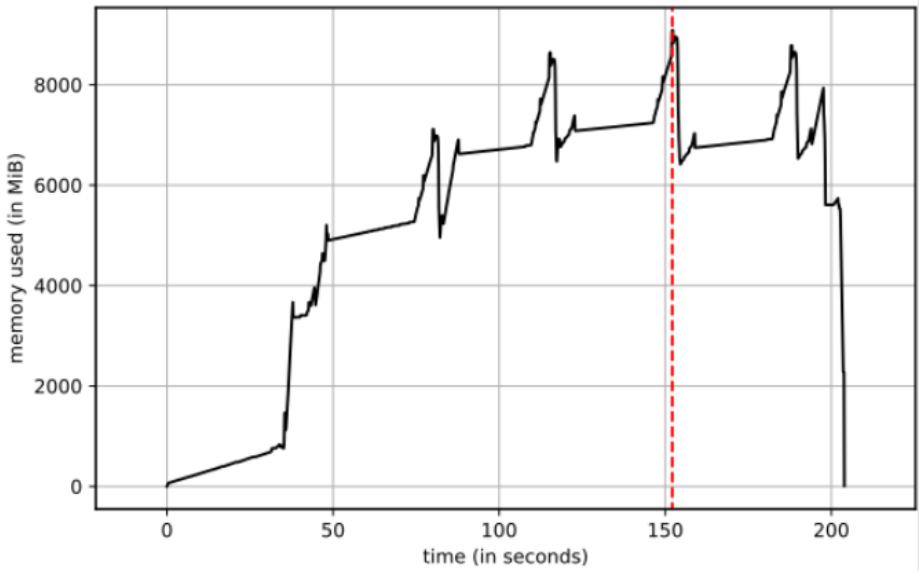


Abbildung 16: Arbeitsspeicherbedarf bei Verwendung der "Spawn" Startmethode und einem Worker-Prozess zur Umwandlung von 10^5 Agenten und 10^7 Zeitreiheneinträge

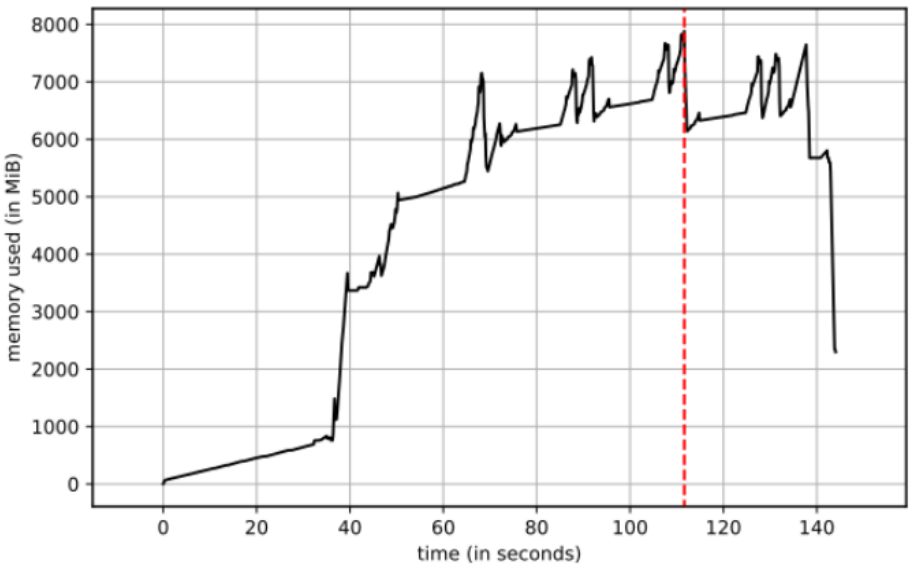


Abbildung 17: Arbeitsspeicherbedarf bei Verwendung der "Spawn" Startmethode und zwei Worker-Prozessen zur Umwandlung von 10^5 Agenten und 10^7 Zeitreiheneinträge

Vergleicht man den Speicherplatzbedarf mit dem in Kapitel 4 gemessenen Ausgangszustand, so verdoppelt sich dieser nahezu, bleibt jedoch nahezu unabhängig von der Gesamtzahl der eingesetzten Worker-Prozesse.

Schlussfolgernd konnte die Laufzeit von FAME-IO durch die parallele Verarbeitung von Zeitreiheneinträgen in mehreren Prozessen deutlich reduziert und gleichzeitig der Bedarf an Arbeitsspeicher stabilisiert werden. Für das Simulationsszenario mit 10^1 Agenten und 10^7 Zeitreiheneinträgen ergab die gleichzeitige Verarbeitung auf sechs Prozessen die größte Beschleunigung.

Das folgende Unterkapitel betrachtet eine weitere Optimierungsmöglichkeit mit dem Ziel, die Gesamtlaufzeit von FAME-IO weiter zu reduzieren.

5.2 Konvertierung von Zeitstempeln

Bei der Betrachtung des Ausgangszustandes von FAME-IO in Kapitel 4 fiel neben der sequentiellen Umwandlung der Zeitreiheneinträge, welche in Kapitel 5.1 behandelt wurde, auf, dass die Konvertierung des Zeitstempels eines Zeitreiheneintrages von einer textuellen in eine numerische Repräsentation mit 55% einen erheblichen Anteil an der Gesamtlaufzeit von FAME-IO hat. Aus diesem Grund widmet sich das folgende Unterkapitel der Optimierung der Zeitstempelkonvertierung. Wie auch in Kapitel 5.1 bezieht sich die Modifikation nur auf die Verarbeitung der Zeitreiheneinträge, weshalb bei den folgenden Betrachtungen nur Simulationskonfigurationen mit einer Agentenanzahl von 10^1 verwendet werden.

5.2.1 Zielsetzung

Das Modul `FameTime` enthält nützliche Methoden zur Umwandlung von textuellen Zeitstempeln in eine FAME-spezifische numerische Repräsentation. FAME verwendet eine vereinfachte Chronologie, die auf einer fest definierten Anzahl an Zeitschritten (Steps) pro Sekunde basiert. In der aktuellen Version entspricht jede Sekunde genau einem Step. Des Weiteren ist definiert, dass ein Tag 24 Stunden und ein Jahr 365 Tage hat. Für die numerische Repräsentation eines Zeitpunktes wird die Anzahl an Steps seit dem Startdatum 01.01.2000 verwendet.

Um die Anzahl der Steps für einen textuellen Zeitstempel wie beispielsweise `12-05-2015_13:25:56` zu errechnen, wird die Funktion `convert_datetime_to_fame_time_step()` verwendet. Diese ruft die Funktion `_convert_to_datetime()` auf, um aus dem textuellen Zeitstempel ein DateTime Objekt zu erstellen. Hierfür verwendet `FameTime` die Funktion `_strptime()` aus dem Python Standardmodul *datetime*, welche bei dem in Kapitel 4.2 durchgeführten Profiling eine sehr hohe Eigenlaufzeit aufwies. Zeil der Optimierung ist es, die Funktion `_strptime()` durch eine performantere Implementierung zu ersetzen.

5.2.2 Umsetzung

Da die weiteren Berechnungen in der Funktion `convert_datetime_to_fame_time_step()` nur auf den konkreten Werten für Tag, Monat, Jahr sowie Stunde, Minute und Sekunde basieren, können diese Informationen auch direkt aus dem übergebenen textuellen Zeitstempel extrahiert werden. Hierfür wird ein regulärer Ausdruck verwendet, der die benötigten Datums- und Uhrzeitinformationen gruppiert bereitstellt. Die Funktion `groupdict()` stellt die gruppierten Informationen mittels eines `Dict` Objekts zur Verfügung. Anschließend werden die extrahierten Informationen einem neu erzeugten DateTime Objekt als Parameter übergeben. Schließlich wird das neu erzeugte DateTime Objekt an die aufrufende Funktion `convert_datetime_to_fame_time_step()` zurückgegeben.

Damit die entsprechenden Informationen aus dem textuellen Zeitstempel entnommen werden können, muss dieser einem in FAME-IO definierten Format entsprechen. Zur Prüfung dieses Formats stellt das `FameTime` Objekt die Hilfsfunktion `is_datetime()` zur Verfügung, welche zu Beginn der Funktion `convert_datetime_to_fame_time_step()` aufgerufen wird.

Listing 2 zeigt die modifizierte Implementierung der Funktion `_convert_to_datetime()`

```
    @staticmethod
    def _convert_to_datetime(datetime_string: str):
        match = DATE_REGEX.fullmatch(datetime_string)
        dt_dict = match.groupdict()
        return dt.datetime(year=int(dt_dict["year"]),
                           month=int(dt_dict["month"]),
                           day=int(dt_dict["day"]),
                           hour=int(dt_dict["hour"]),
                           minute=int(dt_dict["minute"]),
                           second=int(dt_dict["second"]))
```

Listing 2: Funktion _convert_to_datetime()

5.2.3 Validierung

Um die Optimierung der Laufzeit der modifizierten Implementierung zu validieren, wurde erneut das Modul *timeit* herangezogen. Zuerst wurde die Laufzeit der ursprünglichen Implementierung anhand von 10^6 Aufrufen der Funktion `_convert_to_datetime()`, welche zehn mal wiederholt wurden, ermittelt. Dies ergab auf der in Kapitel 4.1 beschriebenen AWS Umgebung eine durchschnittliche Ausführungszeit von 12,1 Sekunden bei einer Standardabweichung von 0,072. Anschließend wurde die optimierte Implementierung mit der identischen Konfiguration aufgerufen, was durchschnittlich 4,24 Sekunden in Anspruch nahm und eine Standardabweichung von 0,007 aufwies. Somit ergibt sich aus der modifizierten Erzeugung des DateTime Objekts eine deutliche Beschleunigung der Funktion `_convert_to_datetime()`, wodurch sich die Gesamtlaufzeit von FAME-IO stark reduziert. Tabelle 1 stellt die ursprüngliche und optimierte Laufzeit von FAME-IO abhängig von der Anzahl der verarbeiteten Zeitreiheneinträge für 10^1 Agenten gegenüber.

Zeitreiheneinträge	**ursprüngliche Laufzeit**	**optimierte Laufzeit**
10^3	0,13	0,02
10^4	0,2	0,12
10^5	1,8	1,2
10^6	18,3	11
10^7	183	112

Tabelle 1: Vergleich der Laufzeit von FAME-IO mit Ausgangsimplementierung und verbesserter Implementierung der Zeitstempelkonversion

Eine Gesamtbetrachtung der Durchschnittslaufzeit von FAME-IO in Abhängigkeit von der Agentenanzahl und der Anzahl verwendeter Zeitreiheneinträge stellt Abbildung 18 dar.

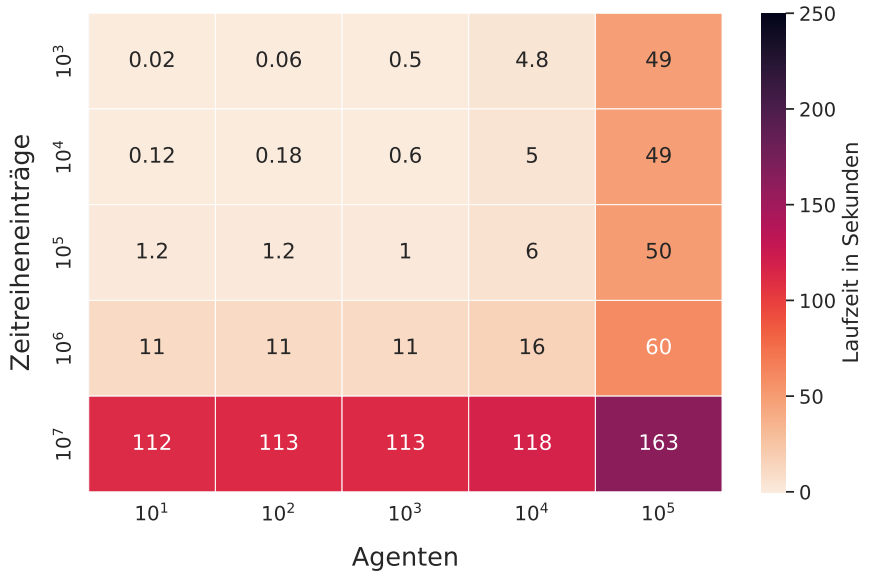


Abbildung 18: Laufzeiten von FAME-IO durch optimierte Zeitstempelumwandlung mittels regulärem Ausdruck

Vergleicht man die aus der in in diesem Kapitel beschriebenen Optimierung resultierende Laufzeit mit der Ausgangslaufzeit in Abbildung 4, erkennt man eine deutliche Reduzierung der Gesamtlaufzeit. Im folgenden Kapitel werden alle in Kapitel 5 beschriebenen Optimierungen angewendet und die resultierende Gesamtlaufzeit ermittelt.

5.3 Optimierte Laufzeit von FAME-IO

Die Anwendung der in Kapitel 5.1 und Kapitel 5.2 beschriebenen Optimierungen führen jeweils zu einer deutliche Reduktion der Gesamtlaufzeit von FAME-IO. Die Anwendung des Multi-Processing Ansatzes mittels der von der Python Standardbibliothek bereitgestellten Klasse *multiprocessing.Pool* zur parallelen Umwandlung von Zeitreiheneinträgen reduziert die Laufzeit des Simulationsszenarios mit 10^5 Agenten und 10^7 Zeitreiheneinträgen, bei Verwendung von sechs Prozessen, von 232 Sekunden auf 149 Sekunden. Dies entspricht einem Beschleunigungsfaktor von 1,5.

Die Erzeugung des DateTime Objektes durch direkte Zuweisen der zuvor mittels regulärem Ausdruck extrahierten Werte beschleunigt die Gesamtlaufzeit von FAME-IO von anfänglich 232 Sekunden auf nun 163 Sekunden, entsprechend einem Beschleunigungsfaktor von 1,4.

Wendet man beide Optimierungen auf das anfänglich beschriebene Simulationsszenario an, so ergibt sich bei Verwendung von sechs Prozessen eine redu-

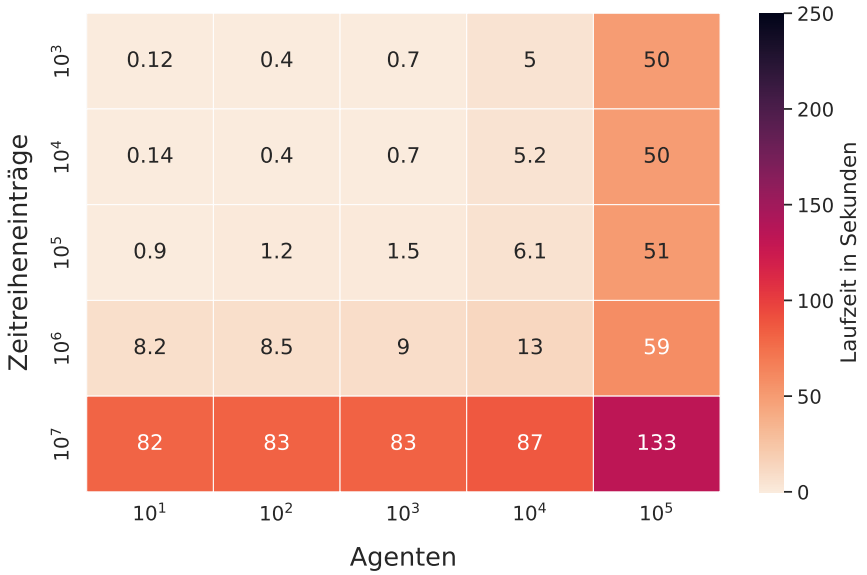


Abbildung 19: Gesamtlaufzeit bei Anwendung beider Optimierungen bei Verwendung von sechs Prozessen

zierte Gesamtlaufzeit von 133 Sekunden und damit ein Gesamtbeschleunigungsfaktor von 1,7.

Abbildung 19 stellt die durchschnittliche Gesamtlaufzeit der optimierten FAME-IO Implementierung in Abhängigkeit der verwendeten Anzahl an Agenten und Zeitreiheneinträgen dar.

Vergleicht man die durchschnittlichen Laufzeiten in Abbildung 19 mit den Laufzeiten des Ausganzszustandes, so fällt auf, dass die Optimierungen bei Simulationsszenarien mit 10^3 Zeitreiheneinträgen zu keiner Beschleunigung der Gesamtlaufzeit führen. Dies lässt sich dadurch begründen, dass die Verarbeitung von Agenten im Vergleich zur Verarbeitung von Zeitreiheneinträgen bei dem in Kapitel 4.2 durchgeführten Profiling einen geringeren Anteil an der Gesamtlaufzeit ausgemacht hat. Damit fokussierten sich die Optimierungen auf die Umwandlung der Zeitreiheneinträge, die beim Profiling den größten Anteil an der Gesamtlaufzeit darstellten.

Betrachtet man die Simulationsszenarien mit 10^7 Zeitreiheneinträgen, ist eine hohe Beschleunigung der Gesamtlaufzeit zu erkennen. Bei einer parallelen Verarbeitung des Szenarios mit 10^1 Agenten auf sechs Prozessen beträgt der Beschleunigungsfaktor 2,2.

Bei den durchgeführten Messung zeigte sich jedoch, dass der Verlauf der Gesamtlaufzeit bei paralleler Verarbeitung der Zeitreiheneinträge auf mehr als sechs Prozessen sich umkehrt und wieder ansteigt. Dieses Verhalten lässt sich durch das Ertragsgesetz wie in Kapitel 2.6 beschrieben, erklären.

6 Weiterführende Forschung

Die in diesem Kapitel implementierten Optimierungen basieren auf den Erkenntnissen der in Kapitel 2.2 durchgeführten Laufzeitanalyse. Da nun die dabei erkannten laufzeitkritischsten Programmteile optimiert wurden, bietet sich eine erneute Durchführung einer Laufzeitanalyse an, um die Programmteile zu identifizieren, welche nach der ersten Optimierungsiteration den größten Einfluss auf die Gesamtlaufzeit haben. Hierfür wird erneut auf den in Kapitel 2.2 beschriebenen Profiler *cProfile* und ein Simulationsszenario mit 10^5 Agenten und 10^7 Zeitreiheneinträgen zurückgegriffen. Das Ergebnis des Profilings wird mit der Visualisierungsbibliothek *yFile* visualisiert. Der Ausschnitt des Profiling-Ergebnisses in Abbildung 20 zeigt die beiden Funktionen mit dem größten Anteil an der Gesamtlaufzeit, die damit Ausgangspunkt für weitere Optimierungen sind.

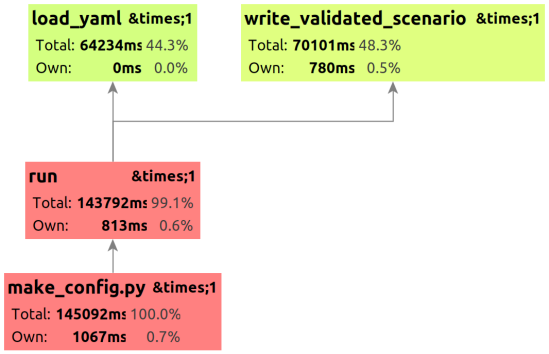


Abbildung 20: Ergebnis Profiling nach erster Optimierungsiteration

Die Funktion `load_yaml()` liest die bei der Ausführung von FAME-IO übergebene Szenario YAML Datei ein und gibt ein Objekt vom Typ `Dict` zurück. Hierfür wird das Python Modul *pyyaml* verwendet, welches eine effiziente Schnittstelle zum Lesen und Schreiben von YAML Konfigurationsdateien bereitstellt. Da die Funktion `load_yaml()` lediglich die `load()` Funktion des *pyyaml* Moduls aufruft, kann die Laufzeit lediglich durch Verwendung einer effizienteren Bibliothek reduziert werden. Die Recherche nach einer effizienteren Bibliothek zum Lesen und Schreiben von YAML Konfigurationsdateien wird jedoch in dieser Arbeit nicht weiter verfolgt.

Bei der weiteren Analyse wird nun die Funktion `write_validated_scenario()` betrachtet. Diese nimmt mit einem prozentualen Anteil von 48,3% auch nach Anwendung der in Kapitel 5 beschriebenen Optimierung den größten Teil der Gesamtlaufzeit in Anspruch. Einen weiteren Optimierungsansatz liefert das in Kapitel 2.6 beschriebene Amdahlsche Gesetz. Das Gesetz gibt den Beschleunigungsfaktor für die Laufzeit eines zum Teil parallel ausführbaren Programms in Abhängigkeit von den verwendeten Prozessen an. Während sich die Optimierung in Kapitel 5.1 auf die letztere Variable, die Anzahl der Prozesse, fokussiert,

bietet auch die Erhöhung des zweiten Einflussfaktors, dem Anteil des parallel ausführbaren Programmcodes, die Möglichkeit zur Optimierung.

7 Fazit

In Kapitel 4 wurde der Ausgangszustand von FAME-IO analysiert und der Programmteil, in dem Zeitreiheneinträge in Protobuffer Objekte umgewandelt werden, als der mit dem größten Einfluss auf die Gesamtlaufzeit von FAME-IO identifiziert. Diese Umwandlung hatte einen Anteil von etwa 90% an der Gesamtlaufzeit, weshalb dieser in Kapitel 5 optimiert wurde. Durch Anwendung der in Kapitel 5.1 beschriebenen parallelen Umwandlung von Zeitreiheneinträgen in Protobuffer Objekte auf sechs Prozessen, reduziert sich die Gesamtlaufzeit bei Verwendung eines Simulationsszenarios mit 10^5 Agenten und 10^7 Zeitreiheneinträgen von anfänglich 232 Sekunden auf 149 Sekunden. Da die Verwendung mehrerer Prozesse einen erheblichen Einfluss auf den benötigten Arbeitsspeicher haben kann, wurde dieser ebenfalls betrachtet. Es zeigte sich, dass die standardmäßig auf Unix Systemen verwendete "Fork" Methode zum Erstellen von Worker-Prozessen zu einem erheblichen Anstieg des Arbeitsspeicherbedarfs führt. Bei Verwendung von acht Prozessen wurden fast 35 GiB Speicherplatz beansprucht. Dieser konnte durch die Umstellung auf die Startmethode "Spawn" auf etwa 8 GiB reduziert werden. Die Frage nach den Auswirkungen auf den Arbeitsspeicherbedarf ist somit abhängig von der verwendete Startmethode zur Prozessinitialisierung.

Darüber hinaus wurde die Gesamtlaufzeit weiter reduziert, indem die Funktion `_strptime()` durch eine alternative Implementierung, welche Listing 2 zeigt, ersetzt wurde. Durch das Extrahieren der Datums- und Uhrzeitinformationen mittels regulärem Ausdruck aus den textuellen Zeitstempeln der Konfigurationsdateien, können die Werte direkt einem neu erzeugten DateTime Objekt zugewiesen werden. Diese Optimierung reduziert die Gesamtlaufzeit von FAME-IO mit eine Konfiguration von 10^5 Agenten und 10^7 Zeitreiheneinträgen von anfänglich 232 Sekunden auf 163 Sekunden.

Wendet man beide Optimierungen gleichzeitig an, ergibt sich für die genannte Simulationskonfiguration bei Verwendung von sechs Worker-Prozessen eine optimierte Gesamtlaufzeit von 133 Sekunden. Bezogen auf die erste Forschungsfrage aus Kapitel 1, konnte eine Geschwindigkeitssteigerung um Faktor 1,7 erzielt werden. Bei Verwendung eines Simulationsszenario mit 10^1 Agenten und 10^7 Zeitreiheneinträgen konnte eine Beschleunigung um Faktor 2,2 erreicht werden.

Die zuvor beschriebenen Optimierungen werden in das unter der Open Source Lizenz veröffentlichte Projekt FAME [5] integriert, wodurch zukünftige auf FAME basierende Simulationen von einer optimierten Laufzeit von FAME-IO profitieren.

Literaturverzeichnis

[1] *Amazon EC2 Instance-Typen*. Amazon Web Services, Inc. URL: https://aws.amazon.com/de/ec2/instance-types/ (besucht am 05. 10. 2021).

[2] Bundesumweltministerium. *Die Klimakonferenz in Paris*. Bundesministerium für Umwelt, Naturschutz und nukleare Sicherheit. URL: https://www.bmu.de/themen/klimaschutz-anpassung/klimaschutz/internationale-klimapolitik/pariser-abkommen (besucht am 26. 10. 2021).

[3] Pierre Carbonnelle. *PYPL PopularitY of Programming Language index*. PYPL PopularitY of Programming Language. URL: https://pypl.github.io/PYPL.html (besucht am 16. 09. 2021).

[4] *datetime — Basic date and time types — Python 3.10.0 documentation*. URL: https://docs.python.org/3/library/datetime.html#strftime-and-strptime-format-codes (besucht am 21. 11. 2021).

[5] *FAME-Framework*. GitLab. URL: https://gitlab.com/fame-framework (besucht am 02. 09. 2021).

[6] Elliot Forbes. *Learning Concurrency in Python*. 1. Aufl. Birmingham: Packt Publishing Limited, 2017. 1 Online-Ressource (360 Seiten). ISBN: 978-1-78728-316-9.

[7] Matt von Harrison und Theodore Petrou. *Pandas 1.x Cookbook*. Pages: 626. 2020. ISBN: 978-1-83921-891-0.

[8] Martin Klein, Ulrich J. Frey und Matthias Reeg. „Models Within Models – Agent-Based Modelling and Simulation in Energy Systems Analysis“. In: *Journal of Artificial Societies and Social Simulation* 22.4 (2019), S. 6. ISSN: 1460-7425.

[9] *multiprocessing — Process-based parallelism — Python 3.9.7 documentation*. URL: https://docs.python.org/3/library/multiprocessing.html (besucht am 02. 09. 2021).

[10] Quan Nguyen. *Mastering Concurrency in Python*. 1. Aufl. Birmingham: Packt Publishing Limited, 2018. 1 Online-Ressource (446 Seiten). ISBN: 978-1-78934-136-2.

[11] Gerard O'Regan. „Gene Amdahl“. In: *Giants of Computing: A Compendium of Select, Pivotal Pioneers*. Hrsg. von Gerard O'Regan. London: Springer London, 2013, S. 13–16. ISBN: 978-1-4471-5340-5. DOI: 10.1007/978-1-4471-5340-5_3. URL: https://doi.org/10.1007/978-1-4471-5340-5_3.

[12] Fabian Pedregosa. *memory-profiler: A module for monitoring memory usage of a python program*. Version 0.58.0. URL: https://github.com/pythonprofilers/memory_profiler (besucht am 02. 11. 2021).

[13] *PEP 619 – Python 3.10 Release Schedule*. Python.org. URL: https://www.python.org/dev/peps/pep-0619/ (besucht am 16. 09. 2021).

[14] *Protocol Buffers*. Google Developers. URL: https://developers.google.com/protocol-buffers (besucht am 21. 11. 2021).

[15] *PyPI · The Python Package Index*. PyPI. URL: https://pypi.org/ (besucht am 16. 09. 2021).

[16] Guido Van Rossum. *The History of Python: A Brief Timeline of Python*. The History of Python. 20. Jan. 2009. URL: http://python-history.blogspot.com/2009/01/brief-timeline-of-python.html (besucht am 16. 09. 2021).

[17] Christoph Schäfer. „Überblick über die Programmiersprache Python“. In: *Schnellstart Python: Ein Einstieg ins Programmieren für MINT-Studierende*. Hrsg. von Christoph Schäfer. Wiesbaden: Springer Fachmedien Wiesbaden, 2019, S. 1–2. ISBN: 978-3-658-26133-7. DOI: 10.1007/978-3-658-26133-7_1. URL: https://doi.org/10.1007/978-3-658-26133-7_1.

[18] Prof Dr Marion Steven. *Definition: Ertragsgesetz*. Ertragsgesetz. Publisher: Springer Fachmedien Wiesbaden GmbH Section: economy. 19. Feb. 2018. URL: https://wirtschaftslexikon.gabler.de/definition/ertragsgesetz-34979/version-258470 (besucht am 30. 09. 2021).

[19] *The Python Profilers — Python 3.9.7 documentation*. URL: https://docs.python.org/3/library/profile.html (besucht am 17. 09. 2021).

[20] *timeit — Measure execution time of small code snippets — Python 3.9.7 documentation*. URL: https://docs.python.org/3/library/timeit.html#module-timeit (besucht am 17. 09. 2021).

[21] Laura Torralba-Díaz u. a. „Identification of the Efficiency Gap by Coupling a Fundamental Electricity Market Model and an Agent-Based Simulation Model“. In: *Energies* 13.15 (Jan. 2020). Number: 15 Publisher: Multidisciplinary Digital Publishing Institute, S. 3920. DOI: 10.3390/en13153920. URL: https://www.mdpi.com/1996-1073/13/15/3920 (besucht am 26. 10. 2021).

Abbildungsverzeichnis