Universität Kassel
Fachbereich 16 Elektrotechnik/Informatik
Bachelorstudiengang Informatik

Parallelisierung des Simulationsprogramms HAUSer mit OpenMP

Bachelorarbeit
zur Erlangung des akademischen Grades
Bachelor of Science

vorgelegt von
Martin Heide
Matrikelnummer 880288
am 27.02.2024

Erstgutachter:	Prof. Dr. Claudia Fohry
Zweitgutachter:	Prof. Dr.-Ing. Anton Maas

Erklärung

Ich versichere hiermit, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Kassel, den *eigenhändige Unterschrift*

Inhaltsverzeichnis

1 Einleitung **1**

2 Ausgewählte Konzepte des OpenMP **5**

3 Simulationsprogramm HAUSer **9**

4 Konzept der Parallelisierung des Programms HAUSer mit OpenMP **13**

4.1 Konvertieren von Fortran77 in Fortran2008 13

4.2 Input-Buffer und Output-Buffer 14

4.3 Reader-Worker-Synchronisation 15

4.4 Worker-Writer-Synchronisation 16

5 Implementierung der Parallelisierung **19**

5.1 Initialisieren der Lock-Variablen durch den Master-Thread 19

5.2 Reader-Worker-Synchronisation 22

5.3 Worker-Writer-Synchronisation 26

6 Experimente und Ergebnisse **29**

7 Fazit **31**

A Akronyme **33**

B Glossar **35**

C Literaturverzeichnis **37**

1 Einleitung

Das Ziel dieser Bachelorarbeit war es, ein bestehendes Simulationsprogramm, das für rein serielle Ausführung entwickelt wurde, durch Parallelisierung mit OpenMP zu beschleunigen. Bei dem Simulationsprogramm handelt es sich um das Programm HAUSer. Es wurde mit Fortran77 programmiert, ohne die Erfordernisse der parallelen Verarbeitung zu berücksichtigen. Diese Fortran-Version beherrschte noch keine dynamische Speicherverwaltung wie z.B. zur Laufzeit allokierbare Arrays oder wachsende Listen. Weil die tatsächliche Größe der Arrays von Benutzer-Eingaben abhängt, wie Anzahl der definierten Bauteile, Räume, Nutzer, Wärmequellen, etc., ist das Programm HAUSer gezwungen, Arrays zu definieren, deren Längen in der Regel zu lang sind. Schließlich darf eine besonders lange Eingabe keine Arrays von HAUSer zum Überlaufen bringen. Die dynamische Speicherallokation wäre von großem Vorteil für Multithreading. Vor allem das Serialisieren der von Worker-Threads berechneten Simulationsdaten erfordert zwingend wachsende Listen. Ein einzelner Simulationslauf umfasst in der Regel nur ein Jahr an Klimadaten. Einige Simulationen erfordern es, den Simulationslauf bis zu zehn mal zu wiederholen. D.h. anstatt eines Datensatzes sind es zehn Datensätze, die auf den Festspeicher geschrieben werden sollen. Im Falle eines Single-Thread-Programms können die Datensätze einfach in die Ausgabedatei sequentiell geschrieben werden, so dass Array fester Größe für jede weitere Simulationsiteration verwendet werden können. Im Fall von mehreren Worker-Threads, dürfen die Daten unterschiedlicher Threads nicht einfach durcheinander in eine Datei geschrieben werden, deshalb müssen die Datensätze gesammelt und anschließen von einem Writer-Thread serialisiert werden. Soll der zehnfache Speicherbedarf vermieden werden, dann bedarf es `ALLOCATABLE` Fortran-Arrays, die erst mit Fortran90 eingeführt wurden. Weil der verwendete GNU-Fortran-Compiler den Code von Fortran 77 bis 2008 in einem Projekt kompilieren kann, wurde in dieser Arbeit Fortran2008 verwendet und die Ziele der Arbeit entsprechend erweitert. Auch musste so genanntes Refactoring vorgenommen wer-

den, um die Nutzererfahrung durch die Änderungen am Code vom Programm HAUSer nicht zu beeinträchtigen. Die Benutzer und Programmierer des HAUSer sind dieselben Personen. Es kommt häufig zu Änderungen am HAUSer-Code, deshalb musste die Struktur des Programms entsprechen angepasst werden. Das Programm wurde in Module unterteilt. Die Ziele der Arbeit wurden entsprechend angepasst.

Die Eingaben für das Programm HAUSer bestehen aus strukturierten ASCII-Dateien. Diese umfassen eine Steuerdatei, die aus Text-Zeilen besteht. Jede Zeile enthält einen Satz von Parametern, die für einen Simulationslauf benötigt werden. Darunter sind Datei-Pfade zu weiteren Eingabe-Dateien, wie Wand-Datei, Raum-Datei, Nutzungsprofil (Verhalten der Gebäude-Nutzer hinsichtlich Lüftung, Beleuchtung, Heizen etc.), TRY-Datei mit den Wetterdaten. Eine Zeile der HAUSer-Steuer-Datei enthält alle Parameter, die ein Simulationslauf braucht, um Ergebnisse zu Berechnen, wie im Sequenz-Diagramm 3.1 dargestellt.

Die Ausgaben des HAUSer-Programms sind Dateien mit 8760 Zeilen, eine Zeile entspricht einer simulierten Stunde eines Jahres, eine Datei entspricht einem simulierten Jahr. Zusätzlich wird eine Zeile mit den Energiebedarfswerten pro Jahr, bezogen auf verschiedene Gebäude-Parameter, die hier nicht weiter beschrieben werden, in eine weitere Ergebnisdatei geschrieben.

Das Programm HAUSer ist proprietär aber quelloffen und wird für thermisch-dynamische Simulationen von Gebäuden benutzt. Solche Simulationen werden sehr häufig in der Bauphysik eingesetzt, um thermisches Verhalten von Gebäuden und deren Energie-Effizienz zu beurteilen. Dies umfasst den simulierten Energiebedarf für Heizen und Kühlen, Voraussagen über die Überhitzung von Gebäuden angesichts des globalen Klimawandels, Beurteilung der Auswirkungen von neuen Technologien auf Nutzer-Komfort und Energiebedarf. Das ermöglicht den Ingenieuren und Ingenieurinnen das Zusammenspiel zwischen Gebäude, darin verbauten technischen Anlagen und der Umwelt zu verstehen und zu optimieren.

Das Ingenieurbüro Prof.-Dr. Hauser GmbH ist die Entwicklerin und Anwenderin des Simulationsprogramms HAUSer in einem. Es gibt zwei Anwendungsdomänen. Die Ingenieursdienstleistungen und Forschung im Bereich Bauphysik. Dabei stel-

len Ingenieursdienstleistungen keine erhöhten Anforderungen auf die Laufzeit des HAUSer-Programms. Weil sich das zu untersuchende Gebäude an einem Bestimmten Ort befindet, muss nur ein einziges Testreferenzjahr in Betrachtung gezogen werden. Handelt es sich um ein forschungsbezogenes Projekt, dann kann die Simulation einer einzigen Gebäudekonfiguration mehrere Tage dauern, weil dann die Wetterdaten pro Quadratkilometer der Bundesrepublik Deutschland berücksichtigt werden müssen.

Insbesondere wird HAUSer in der Forschung für Studien zu Auswirkungen des globalen Klimawandels und der damit verbundenen Überhitzung von Gebäuden eingesetzt. Weil das thermische Verhalten von Gebäuden ortsabhängig ist, stellt der Deutsche Wetterdienst (DWD) für jeden Quadratkilometer der Bundesrepublik Deutschland so genannte Testreferenzjahre (TRY) zur Verfügung. Bei einem Testreferenzjahr handelt es sich um eine Zusammenstellung meteorologischer Daten für ein spezifisches geographisches Gebiet von einem Quadratkilometer Fläche der Bundesrepublik Deutschland. TRY Daten repräsentieren typische Wetterbedingungen für den jeweiligen Ort. Die Idee hinter dem Testreferenzjahr ist es, standardisierte Jahresklimadaten zu schaffen, die ortsabhängig für energietechnische und bauklimatische Simulationen im Bereich Gebäudetechnik verwendet werden können [1]. Ein TRY ist ein Synonym für eine Referenz auf ein hypothetisches, typisches Jahresvorrat an Wetterdaten (Lufttemperatur, relative Feuchte, solare Bestrahlung, etc.), die Stündlich vom 01. Januar 00:00 Uhr bis zum 31. Dezember 00:00 Uhr in einer strukturierten ASCII-Datei gelistet sind. TRY wird stellvertretend für klimatische Daten aller Jahre eines Orts benutzt. Ein Testreferenzjahr ist ein statistisches Median über Zeit aber nicht über Ort. Der Bedarf an flächenbezogener Auflösung der Klimadaten im Zusammenhang mit den Studien über die Auswirkungen des globalen Klimawandels auf die Überhitzung von Gebäuden wuchs in den vergangenen Jahren. Seit 2017 wendet der DWD neue statistische Verfahren an, um die flächenbezogene Auflösung der Klimadaten von einem Quadratkilometer der Bundesrepublik Deutschland zu erreichen, so dass die Anzahl an verfügbaren TRY-Klima-Dateien von einigen Dutzend auf zwei Millionen wuchs.

Auf einem aktuellen Prozessor benötigt Programm HAUSer für eine Simulationsrechnung bzw. die Verarbeitung einer Zeile der Steuer-Datei etwa 0,3 Sekunden.

Sollen alle für Deutschland verfügbaren TRY berücksichtigt werden, dann steigt die Berechnungszeit auf über 166 Stunden. Das heißt die Berechnung von nur einer Gebäude-Konfiguration dauert eine Woche. Weil die modernen CPUs über viele Kerne verfügen (mehr als zehn), bietet sich die Parallelisierung des HAUSer-Programms als Mittel zur Beschleunigung der Berechnungszeit an.

Fortran verfügt über kein eigenes Multithreading. Die OpenMP-API bietet eine Möglichkeit zur Parallelisierung von Fortran-Programmen. Voraussetzung ist ein OpenMP-fähiger Compiler. Für diese Arbeit wurde gFortran verwendet, das ein Teil der GNU-Compiler-Collection ist. Durch die Verwendung von Pragmas kann im Code des Fortran-Programms angegeben werden, welche Teile des Programms parallel ausgeführt werden sollen, so dass die Anwendungslogik von mehreren Threads parallel mit dem Ziel der Performance-Steigerung verarbeitet werden kann. Für die Parallelisierung des Programms HAUSer, das im Fortran-Code vorliegt, wurde OpenMP benutzt.

Kapitel 2 geht auf die in dieser Arbeit verwendeten Pragmas und Funktionen des OpenMP näher ein. Kapitel 3 beschreibt die Struktur des Programms HAUSer in kurzen Zügen. Kapitel 4 geht auf das Konzept der Parallelisierung des HAUSer-Programms ein, gefolgt von der Beschreibung der Implementierung im Kapitel 5. Kapitel 6 beschreibt schließlich die Performance-Messungen und zeigt die Messergebnisse. Kapitel 7 fasst die aus dieser Arbeit abgeleiteten Schlussfolgerungen zusammen.

2 Ausgewählte Konzepte des OpenMP

Eine parallele Region beginnt in Fortran mit dem Ausdruck `!$OMP PARALLEL` gefolgt von optionalen Klauseln und endet mit `!$OMP END PARALLEL`. Keine zwei Pragmas dürfen in der selben Zeile kombiniert werden. Wird vom Programmierer keine explizite Anzahl an Threads definiert, wählt die zugrundeliegende OpenMP-Implementierung diese Anzahl implizit. In der Regel entspricht die implizite Anzahl der Threads der Anzahl der vorhandenen Prozessor-Kerne. Der OpenMP-Standard legt jedoch nicht fest, wie die Anzahl an Thread vom OpenMP-System gewählt werden soll, deshalb ist die explizite Definition der Anzahl von Threads durch den Programmierer eine bessere Vorgehensweise [2].

Es gibt zwei Möglichkeiten die Anzahl der Threads zur Laufzeit eines Programms festzulegen. Zum Einen per Umgebungsvariable des Betriebssystems `OMP_NUM_THREADS=N`, mit `N` ist gleich der Anzahl Threads. Zum Anderen per Fortran-Subroutine `omp_set_num_threads(N)`, mit dem Parameter vom Typ `INTEGER N` gleich der Anzahl der Threads.

Mit der Funktion `omp_get_max_threads()` lässt sich die Anzahl der Threads ermitteln, die echt-parallel von der zugrundeliegenden CPU ausgeführt werden können. D.h. die Anzahl der Threads entspricht der Zahl der CPU-Kerne. Diese Zahl ist unabhängig von der Stelle im Programm, in der `omp_get_max_threads()` Aufgerufen wird.

Soll die Anzahl der Threads innerhalb der parallelen Region ermittelt werden, dann ist die Funktion `omp_get_num_threads()` das Mittel der Wahl.

Trifft ein Thread auf die OpenMP-Anweisung (`OMP PARALLEL`), dann erstellt dieser einen neuen Datenbereich. Dessen Variable werden von Klauseln auf den neuen, gerade erzeugten Datenbereich abgebildet. Die Klauseln legen die Eigenschaften der Datenfreigabe fest, während die Variablen in den neuen Bereich übertragen werden. Der Datenfreigabe-Attribut wird standardmäßig auf "geteilt" gesetzt. Das Bedeutet, dass es eine Kopie der Variablen gibt, die für alle Threads im Team sichtbar ist. Die Syntax der Klausel für geteilte Variablen ist:

```
!$OMP PARALLEL SHARED(Liste_der_Variablen)
```

Alle gelisteten Variablen werden zwischen Mitgliedern des Thread-Teams geteilt. Jede Änderung des Inhalts der geteilten Variablen wird für alle Threads sichtbar. Änderung an diesen Variablen sollten durch Barrieren, Locks oder Atomic-Anweisungen synchronisiert werden, weil es zu Rennbedingungen (Race Conditions) kommen kann [2].

Eine `PRIVATE(Liste_der_Variablen)` Klausel führt zu Erzeugung lokaler Kopien der gelisteten Variablen für jeden Thread des parallelen Abschnitts. Jede Kopie ist unabhängig von den anderen Kopien. Änderungen an einer privaten Variable innerhalb eines Threads haben keinen Einfluss auf die gleiche Variable in anderen Threads. Sie existiert für die Dauer des parallelen Blocks, in dem sie definiert wurde [2].

Die Anweisung `!$OMP BARRIER` der OpenMP-API definiert eine Synchronisationsbarriere, an der die Threads der parallelen Region anhalten und warten müssen, bis alle Threads die Barriere erreicht haben, bevor sie fortfahren können. Diese Synchronisation wird verwendet, um sicherzustellen, dass bestimmte Teile des Programms von allen Threads vollständig abgeschlossen worden sind, bevor die Ausführung nach der Barriere fortgesetzt wird. Sie verhindert Race-Conditions und kann die Synchronisation zwischen Threads gewährleisten.

`!$OMP ATOMIC` ist ein Pragma der OpenMP-API, die die atomare Ausführung einer spezifischen Operation auf eine zwischen den Threads geteilte Variable gewährleistet. Die Operation wird als eine unteilbare Einheit behandelt, die von den anderen Threads nicht unterbrochen werden kann. Es wird sichergestellt, dass die atomaren Lese- und Schreibvorgänge auf eine Variable vollständig abgeschlossen sind, bevor

ein anderer Thread darauf zugreift. In dieser Arbeit wurden `ATOMIC WRITE` zum Schreiben von Werten in geteilte Variablen und `ATOMIC READ` zum Lesen von Werten aus geteilten in lokale Variablen benutzt. Das Listing 2.1 demonstriert das Lesen einer geteilten Variablen in eine lokale Variable eines Threads. Variable `pcount` wird im lokalen Bereich inkrementiert und danach per `ATOMIC WRITE` in die geteilte Variable geschrieben. Es ist zu beachten, dass nicht alle Prozessoren eine per `ATOMIC` veränderte Variable sofort vom CPU-Cache in den Arbeitsspeicher übertragen und vice versa. Es ist sicherer vor dem Lesen bzw. nach dem Schreiben ein `FLUSH` Pragma zu verwenden. Dann wird der Prozessor angewiesen das Cache mit dem Arbeitsspeicher zu synchronisieren.

```
1  integer :: counter, pcount
2  !$OMP PARALLEL shared(counter) private(pcount)
3  !$OMP FLASH
4  !$OMP ATOMIC READ
5  pcount = counter
6  pcount = pcount + 1
7  !$OMP ATOMIC WRITE
8  counter = pcount
9  !$OMP FLASH
10 !$OMP END PARALLEL
```

Listing 2.1: ATOMIC WRITE

OpenMP-Lock-Variablen bieten die Möglichkeit zur paarweisen Synchronisation zwischen den Threads. In Fortran haben solche Variablen den Datentyp `integer(kind=omp_lock_kind)` und werden per Subroutine `omp_init_lock()` initialisiert. Ein Thread kann Lock per Fortran-Subroutine `omp_set_lock()` erwerben. Ist das Lock erworben, dann kann der Lock-Eigentümer im kritischen Block des parallelen Programms geteilte Variablen thread-sicher lesen oder in diese schreiben. D.h. es treten keine Race Conditions auf. Ein Thread, das nicht freigegebene Lock erwerben will, wird angehalten bis der Eigentümer-Thread `omp_unset_lock()` aufruft [2].

3 Simulationsprogramm HAUSer

Beim HAUSer handelt es sich um ein Computerprogramm zur thermisch-dynamischen Simulation von Gebäuden. D.h. es handelt sich um ein computergestütztes, mathematisches Modell zur Berechnung des thermischen Verhaltens von Bauwerken. Dieses setzt sich aus Teilmodellen zusammen, wie

- der Fourierschen dynamischen Wärmeleitungsgleichung, die die Wärmeleitung von Festkörpern und nichtströmenden Fluiden beschreibt,
- der Gleichung von Navier-Strokes, die die Konvektion in Räumen behandelt (Verluste durch Lüften und undichte Bauteile),
- dem Plankschen Strahlungsgesetzt, der den Strahlungsaustausch der Festkörper beschreibt.

Weitere Teilmodelle sind

- Regelung der Heizung und der Kühlung,
- Interne Gewinne (Wärmequellen, die nicht zum Heizen des Bauwerks genutzt werden),
- Nutzerprofile (Einflüsse durch das Verhalten der Gebäude-Nutzer),
- Meteorologie.

Die Letztgenannten Modelle werden nicht von HAUSer berechnet, sondern werden als Eingabe-Parameter in das Programm durch den Nutzer eingegeben [3]. Z.B. ein Testreferenzjahr kommt als eine vorberechnete Datei vom DWD. Das in dieser Arbeit verwendete Beispiel-Gebäude besteht aus einer Zone von acht Bauteilen. Die Innenwände sind Bauteile, die von mehreren Räumen geteilt werden und nicht mehrfach berechnet werden. Außenwände werden immer berechnet, wobei mehr als

ein Raum auf eine Außenwand-Definition in der HAUSer-Wand-Datei referenzieren darf. Die Zuordnung der Außen- und Innenwände wird in der HAUSer-Raum-Datei beschrieben.

Ein Simulationslauf des HAUSer-Programms berechnet ein Gebäude mit einem spezifischen Nutzer-Profil und einer Wetter-Datei. Gibt es mehr Nutzer-Profile und Wetter-Dateien, dann werden von dem Bediener oder der Bedienerin des HAUSer-Programms entsprechende Simulationsläufe definiert. Alle Eingaben sind strukturierte, kontextsensitive ASCII-Dateien. HAUSer wird in der Regel über einen Skript aufgerufen wie im Listing 3.1 dargestellt.

```
1 #!/bin/bash
2 ./hauser-programm ./steuerdatei.txt
```

Listing 3.1: Beispiel für Aufruf des HAUSer-Programms per Linux-bash

Ein HAUSer-Simulationslauf über nur einer Gebäude-Konfiguration und einem Testreferenzjahr dauern 0,3[s]. Eine Ingenieursdienstleistung mit nur wenigen Gebäude-Konfigurationen und einer Wetterdaten-Datei erfordert keine langen Rechenzeiten. Das erzeugt keinen wirtschaftlichen Druck, um die Parallelisierung des HAUSer-Programms zu motivieren. Eine forschungsbezogene Simulation über einem Bauwerk und allen für Deutschland verfügbaren Testreferenzjahren dauert $0,3[\text{Sekunden}] \cdot 2 \cdot 10^6 = 6 \cdot 10^5[\text{Sekunden}] = 166,7[\text{Stunden}]$. Das sind etwa sieben Tage für nur eine Gebäudekonfiguration. Innerhalb einer Woche können neue Fragen auftauchen, die es erfordern, die laufende Simulationsrechnung abzubrechen und neu zu starten. Im Worst-Case-Szenario können mehrere Wochen vergehen bis zufriedenstellende Simulationsergebnisse vorliegen. Die Eingabedaten können in Teilmengen aufgeteilt werden, so dass das HAUSer-Programm mehrfach mit jeweils verschiedener Eingabe-Teilmengen ausgeführt werden kann. Das erfordert jedoch händisches Eingreifen des Benutzers oder der Benutzerin oder das Programmieren eines Skripts, der die Eingabe-Daten aufteilt. Dieses Vorgehen ist nicht effizient. Benutzer müssen viel Zeit mit Vorbereitungen der Simulationsrechnung verbringen. Die intrinsische Fähigkeit zu Parallelisierung hätte als Nebeneffekt einfachere Bedienung des Programms zur Folge.

Die Steuer-Datei enthält, neben einigen Simulationsparametern, auf die in dieser Arbeit nicht näher eingegangen wird, die Definition der Pfade für Eingabe - und Ausgabe - Dateien. Die Zeile 2 im Listing (3.2) enthält eine verkürzt dargestellte Eingabe-Dateien für einen Einzellauf. Falls sich die Eingabe-Dateien in der nächsten Zeile der Steuerdatei wiederholen, dann werden diese erneut geladen. Die Läufe werden stapel-verarbeitet bis zum Ende der Steuerdatei. Die aus1.txt ist die Ausgabe-Datei für stündliche Werte des simulierten Jahres.

BATCH-ERGEBNISSE.txt ist Ausgabe-Datei für verdichtete Ergebnisse wie z.B. Jahressummen der Energieumsätze. Diese werden pro Zeile in die Ergebnisdatei geschrieben.

```
1 'BATCH_ERGEBNISSE.txt'  0
2 'n/nutzung.txt'  'r/raumdatei.txt'  'w/wanddatei.txt'  'aus1/aus1.txt'
     'TRY/trydatei.txt'
```

Listing 3.2: Kurz-Beispiel der Steuer-Datei des HAUSer-Programms

Im Sequenzdiagramm (3.1) ist ein Simulationslauf des single-threaded HAUSer-Programms dargestellt. Der strukturelle Programm-Aufbau ist sehr einfach programmiert. Die Funktion `control` ist als `PROGRAM` deklariert. Diese Deklaration weist eine Funktion als Hauptfunktion aus, die bei der Ausführung des Programms als erste aufgerufen wird. Funktion `control` liest die Steuerdatei. Eine Zeile der Steuerdatei wird von der Fortran `read()` Funktion als eine Liste von Variable interpretiert. Darunter sind Pfade zu weiteren Eingabedateien wie Wanddatei, Raumdatei, etc.

Die Funktion `haus()` wird von `control` aufgerufen. Die Eingabevariablen werden als Parameter an `haus()` übergeben, wobei es sich bei dieser Funktion um eine Fortran-Subroutine handelt. Diese Subroutine ist eine Funktion, die ihre Parameter by-reference übernimmt. Die Subroutine `haus` enthält die gesamte für eine Gebäudesimulation benötigte Funktionalität und sie schreibt die Simulationsergebnisse in mehrere Ausgabedateien. Für diese Arbeit ist nur diejenige Ausgabedatei relevant, die später für Multithreading serialisiert werden muss. Kapitel 4 beschreibt die Serialisierung und das Konzept der Parallelisierung.

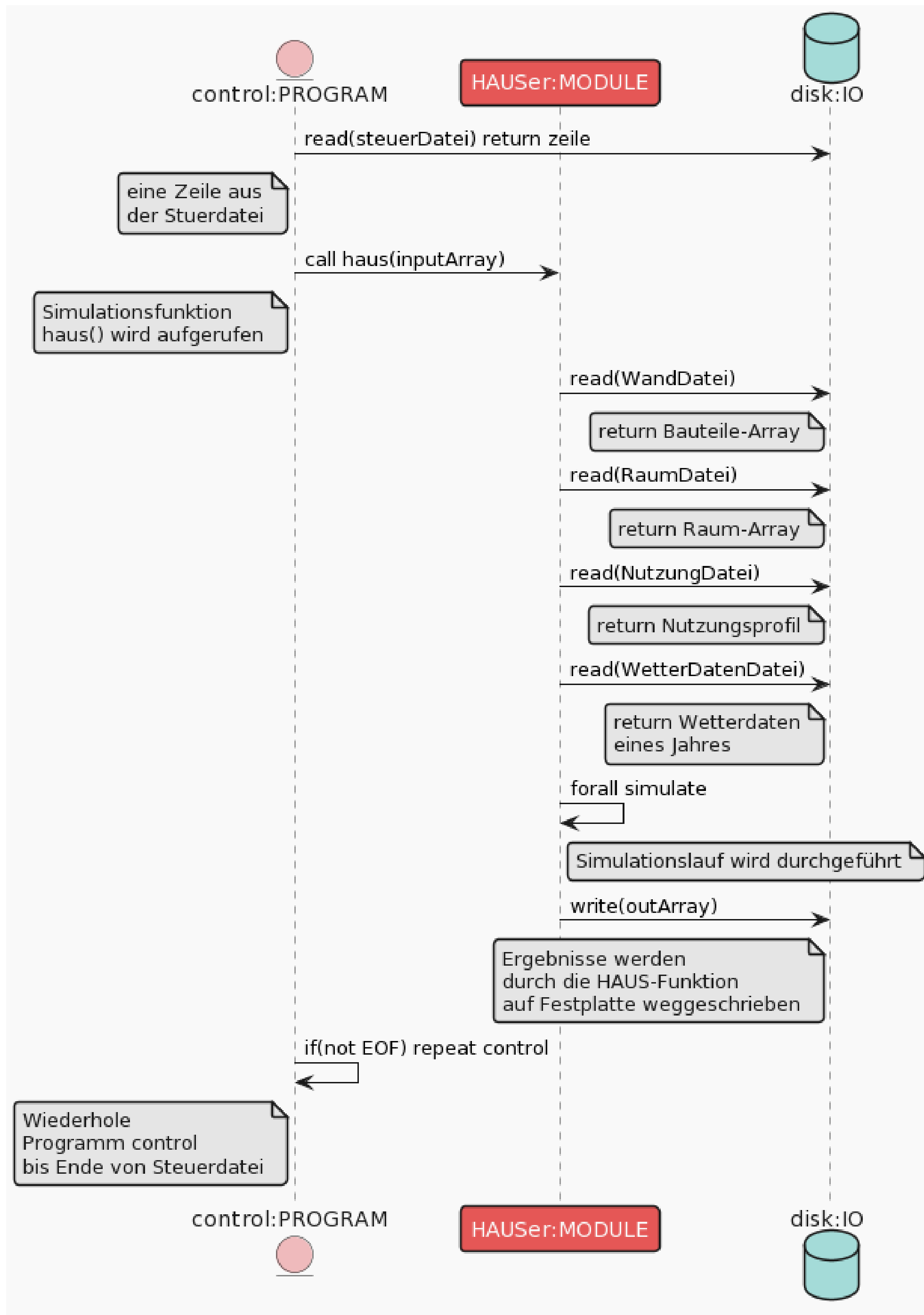

Abbildung 3.1: Sequenzdiagramm eines Laufe des Single-Thread HAUSer-Programms

4 Konzept der Parallelisierung des Programms HAUSer mit OpenMP

4.1 Konvertieren von Fortran77 in Fortran2008

Das Simulationsprogramm HAUSer wurde entwickelt ohne die Erfordernisse der Parallelverarbeitung zu berücksichtigen. Ferner liegt das Programm HAUSer in Fortran77 vor. Die Konvertierung zu einem neueren Fortran-Standard ist einfach, jedoch fehlt die dynamische Speicherverwaltung. Diese ist jedoch für Multithreading erforderlich, weil sonst keine Input-/Output-Buffer anhängig von Anzahl der Threads zur Laufzeit des Programms allokiert werden können. Die Anzahl der Threads hängt vom Prozessor des Rechners oder von der Benutzer-Eingabe ab und kann nicht statisch als Parameter vorgegeben werden, ohne das Programm HAUSer erneut zu kompilieren. Da die HAUSer-Simulationsfunktion komplex ist und die Änderungen am Code sehr spezielles Wissen erfordern, wurde diese im Laufe dieser Arbeit nur minimal Verändert. Lediglich ein Teil der Eingabefunktionen und der Ausgabefunktionen wurden mit dynamisch allokierbaren Feldern (Arrays) nach Fortran2008-Standard ausgestattet, so dass die Worker-Threads von einem Reader-Thread mit Daten versorgt und die Simulationsergebnisse von einem Writer-Thread auf den Festspeicher weggeschrieben werden können.

Die Schreibfunktion für die HAUSer-Ergebnisdatei musste für Multithreading serialisiert werden, denn durch das Overhead der Thread-Syhchronisation würde die Performance der parallelen Ausführung des HAUSer-Programms beeinträchtigt werden. Außerdem müssen die Worker-Threads ihre Ergebnisdaten im Multi-Thread-Betrieb zusammenhängend, zeilenweise ausgeben, ohne dass Zeilen durcheinander ausgeschrieben werden.

4.2 Input-Buffer und Output-Buffer

Eine Struktur fasst alle für einen Simulationslauf benötigten Parameter der Steuerdatei zusammen. (Zur Erinnerung: die Steuerdatei hält eine Zeile pro Simulationslauf bereit. Die Worte der Zeile sind Parameter einer einzelnen Simulation einer einzelnen Gebäude-/Wetter-Konfiguration). Diese Struktur wurde `input_buffer` genannt. Die Struktur wird in einem Array instanziiert, dessen Länge der Anzahl der Worker-Threads entspricht, so dass jeder Worker eine Instanz des `input_buffer` zugeordnet bekommt. Auf jede dieser Instanzen wird paarweise vom Worker mit der entsprechenden Thread-Nummer und vom Reader zugegriffen. Während ein Worker nur auf die für ihn reservierte `input_buffer`-Instanz zugreifen darf, darf Reader auf alle Instanzen zugreifen. So wird vom Reader jede Zeile der Steuerdatei einer Instanz des `input_buffer` zugeordnet.

Ein Simulationsergebnis kann mehr als einen Datensatz beinhalten. Ein HAUSer-Programm ohne die Parallelisierung schreibt die Datensätze als Zeilen in eine Ergebnisdatei. Für Parallelverarbeitung werden wachsende Listen oder ähnliche Datenstrukturen benötigt, weil die Ausgaben eines Thread serialisiert, zusammenhängend in die Ergebnisdatei geschrieben werden müssen.

Fortran2008 hat keine wachsenden Listen. Dieser Nachteil konnte jedoch leicht umgangen werden. Ein Fortran `TYPE output_buffer` wurde definiert. Es handelt sich um eine Datenstruktur, die alle für die Ausgabe in die HAUSer-Ergebnisdatei erforderlichen Variablen enthielt. Darüber hinaus wurde eine Fortran-Struktur `TYPE list_out_buffer` definiert, die die Struktur `output_buffer` mit Attributen `allocatable` und `dimension(:)` deklarierte. So konnte eine Struktur vom Typ `list_out_buffer` definiert werden, die leere Referenzen vom Typ enthielt. Diese Struktur konnte per Klausel `shared` als geteilt deklariert werden und die darin enthaltenen Verweise auf Output-Buffer-Struktur später innerhalb der Simulationssubroutine instanziiert werden, abhängig von der Anzahl zu iterierender meteorologischer Jahre.

4.3 Reader-Worker-Synchronisation

Obwohl es möglich wäre die Zugriffe auf `input_buffer` per OpenMP-Pragmas `OMP CRITICAL` oder `OMP BARRIER` zu synchronisieren, wurden OpenMP-Lock-Variablen für die Synchronisation verwendet. Der Grund dafür war, dass sich die Laufzeit eines Simulationslaufs stark unterscheiden kann. Diese kann zwischen 0,25 und 3 Sekunden betragen, weil, abhängig von Simulationsparametern, ein Lauf bis zu zehn mal wiederholt werden kann. (Solche Wiederholungen werden benutzt, um numerische Berechnungen an Übergängen von Jahren zu verfeinern.) Im Worst-Case Szenario kann es ein Thread geben, das 3 anstatt 0,3 Sekunden für die Berechnung braucht. In so einem Fall würden alle anderen Threads an der Barriere warten, was zu Performance-Einbußen führen würde. Die Worker-Threads sollen asynchron laufen und nur mit dem Reader- oder Writer-Thread paarweise synchronisiert werden. Die Grafik 4.1 stell die Synchronisation zwischen dem Reader- und dem i-ten Worker-Thread dar. Jeder Worker mit Thread-Nummer ungleich i darf auf die für ihn zuständige Zelle im Input-Buffer zugreifen. Sobald ein Worker seine Daten aus dem Input-Buffer abgeholt hat, wird die Variable `stale` (vom Typ `logical`) der entsprechenden Buffer-Instanz auf `.TRUE.` gesetzt, um dem Reader anzuzeigen, dass er diesen Buffer wieder beschreiben darf. Der Reader setzt die Variable `stale` in der entsprechenden Input-Buffer-Zelle auf `.FALSE.`, falls Daten aus der Steuerdatei gelesen werden konnten. Falls nicht, verbleibt der Daten-Zustand im Stale und der Reader signalisiert über eine geteilte Variable, dass End-Of-File der Steuerdatei erreicht wurde, nachdem die Lock-Variable freigeschaltet worden war. Der Reader darf die Schleife verlassen, bleibt jedoch am Ende der parallelen Region des Programms an der Barriere im Wartezustand. Hat Reader die Schleife verlassen, dann prüfen die Worker ihren geteilten Input-Buffer auf Stale-Zustand. Gibt es noch Daten, die nicht veraltet (not stale) sind, dann wird Simulationsrechnung gestartet, sonst wird die Schleife verlassen und an der Barriere gewartet bis alle Threads ihre Arbeit abgeschlossen haben.

Weil es Use-Cases gibt, in denen die Worker viel langsamer sind als der Reader, wurde eine Zählvariable, `worker_cnt`, vom Typ `integer` benutzt, um aktive Worker zu zählen. Die `read_lck` Variable dient dazu den Reader zu blockieren, falls alle Worker aktiv Rechnen und alle Input Buffer frische Daten enthalten, die noch nicht

von Workern abgeholt wurden. Befindet sich der Reader im blockierten Zustand, dann wird er freigegeben, sobald ein Worker seine Simulationsrechnung beendet hat. Diese Art der Benutzung von Locks gilt als schlechter Programmierstil, weil so Dead-Lock-Zustände auftreten können. In diesem Fall wurde sichergestellt, dass der Reader immer vor Workern die parallele Region im HAUSer-Programm verlassen darf. Dadurch wurde sichergestellt, dass ein Dead-Lock nicht auftreten kann, und die oben beschriebenen Nachteile von Barrieren wurden vermieden.

4.4 Worker-Writer-Synchronisation

Ein analoges Verfahren wurde zur Synchronisation zwischen Writer- und Worker-Threads benutzt. Mit dem Unterschied, dass der Worker die vom Writer referenzierten Output-Buffer abhängig von der Anzahl berechneter Jahre instanziiert. Das kopieren des Output-Buffer aus dem lokalen Bereich des Worker-Threads in den geteilten Bereich des Writer-Threads wurde analog zum oben beschriebenen Verfahren designt. Zusätzlich enthält der Writer-Thread eine Nachlauf-Variable, denn sobald die Worker ihre Arbeit abgeschlossen haben, muss der Writer noch ein letztes mal den Output-Buffer auf nicht veraltete Daten (not stale) durchsuchen und diese in die Ergebnisdatei wegschreiben. Der Writer-Thread ist immer der letzte Thread, der die Schleife verlässt.

Die Worker-Threads haben lokale Kopien der jeweiligen geteilten Input- und Output-Buffer-Instanz. Das soll die Verweildauer in der kritischen Region begrenzen und so genanntes False Sharing [2] vermeiden. Beim False Sharing handelt es sich um ein Performance-Problem, das in Multithreading-Umgebungen auftritt. Das geschieht, wenn mehrere Threads auf unterschiedliche Zellen eines geteilten Array parallel zugreifen. Scheinbar stehen diese Zellen nicht in Konkurrenz zueinander. Weil jeder Kern eines modernen Prozessors seinen lokalen Speicher, das Cache, hat und dieses in Cache-Lines unterteilt ist, müssen die Daten derselben Line zwischen Kernen kopiert werden, sobald diese geändert wurden. Dadurch wird Synchronisation vom Prozessor erzwungen, was paralleles Arbeiten verhindert. Lokale Kopien der Daten sind vom False-Sharing nicht betroffen.

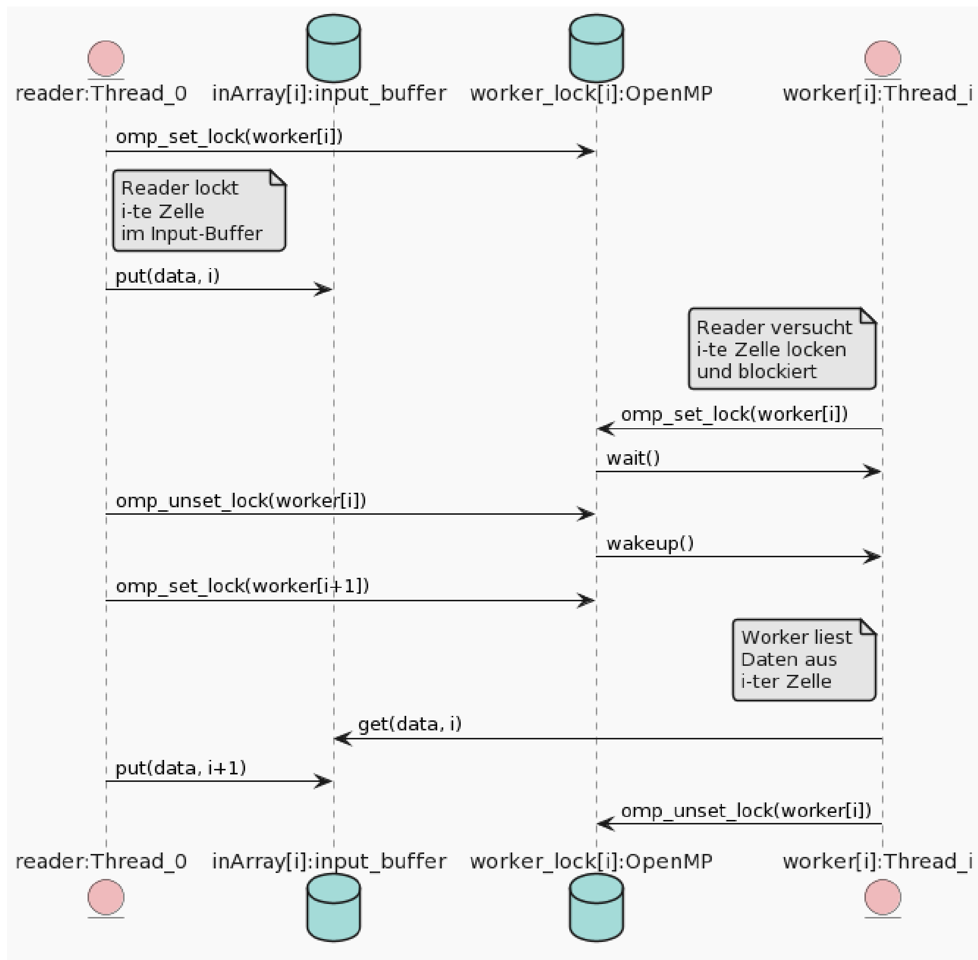

Abbildung 4.1: Paarweise Synchronisation zwischen dem Reader- und dem i-ten Worker-Thread, Lock-Variable ist das Array worker[i], mit i = Thread-Nummer

Weil die Bediener des Programms HAUSer auch Programmierer sind und Änderungen am Code häufig auftreten, wurden die für Simulationsrechnung zuständigen Funktionen des Programms in das Fortran Modul `HAUSer` verschoben. Die beiden Strukturen, Input-Buffer und Output-Buffer, sind im Modul `BUFFER` deklariert. Für Simulationen zuständige Ingenieure und Ingenieurinnen können die Simulationsfunktionen des Programms HAUSer verändern, ohne sich mit der parallelen Programmierung auseinander zu setzen. Das erleichtert das Arbeiten mit dem parallelisierten HAUSer-Programm.

5 Implementierung der Parallelisierung

5.1 Initialisieren der Lock-Variablen durch den Master-Thread

Wie im Abbildung 3.1 dargestellt, liest das HAUSer-Programm jeweils eine Zeile der Steuerdatei, konvertiert die darin gespeicherten Parameter in entsprechend typisierte Variablen und ruft die Subroutine `haus(input_buffer)` mit den aus der Steuerdatei übernommenen Variablen als Funktionsparameter auf. Bestünde die Steuer-Datei aus nur vier Zeilen, dann könnten diese auf vier Threads verteilt und parallel berechnet werden. In der Praxis sind jedoch Steuer-Dateien mit mehrere Millionen Zeilen zu erwarten. Diese Zeilen müssen auf einige Arbeiter-Threads verteilt werden. Das heißt der Reader-Thread muss die Zeilen auf die Worker-Threads verteilen, so dass diese parallel und asynchron die Simulationsrechnungen durchführen können.

Weil HAUSer die Anzahl der Threads dynamisch zur Laufzeit, abhängig von der Anzahl der Prozessor-Kerne definieren soll, musste Code des Fortran77 zum Fortran2008 erweitert werden. Die Speicherallokation des Input- und Output-Array wurde in Fortran2008 programmiert, so dass der Speicherbedarf linear-proportional zu Anzahl der Kerne des Prozessors steigt. Im Listing 5.2 ist eine Fortran2008-Struktur dargestellt, die alle Parameter der HAUSer-Steuerdatei umfasst.

Um dem Worst-Case-Szenario mit stark abweichenden Laufzeiten der Worker-Threads Rechnung zu tragen, wurde entschieden, auf Barrieren zu verzichten. Um aktives Warten der Worker-Threads zu vermeiden, wurde für die Reader-Worker-Synchronisation die Lock-Variable der OpenMP-API verwendet. Eine Lock-Variable wird in Fort-

ran2008 per Datentyp `INTEGER` mit dem Attribut (`kind=omp_lock_kind`) verwendet (Listing 5.1, Zeile 4). Für die Worker bietet sich ein allokierbares Array an, um Lock-Variablen zu speichern, dessen Länge der Anzahl der Worker-Threads entspricht (Listing 5.1, Zeile 6).

Das Pragma `!$OMP MASTER` definiert einen Code-Block, der vom Master-Thread ausgeführt wird. Alle anderen Threads setzen ihre Arbeit im Code-Abschnitt fort, der dem Master-Block folgt. `!$OMP MASTER` impliziert keine Barriere [2]. Im Listing 5.1 wird ein Master-Konstrukt der OpenMP-API verwendet, um die Variable `worker_read_lck` durch Master-Thread zu initialisieren und zu beanspruchen.

Die Worker-Threads überspringen das Master-Konstrukt und warten an der Barriere, bis der Master-Thread das Konstrukt durchlaufen und die Barriere passiert hat (Zeile 18 des Listing 5.1). Die Variable `stStart` wurde eingeführt, um den ersten Durchlauf der parallelen Schleife anzuzeigen. Da der Reader-Thread beim ersten Durchlauf bereits alle Lock-Variablen besetzt hält, darf er diese nicht erneut beanspruchen. Variable `stStart` verhindert, dass Dead-Locks beim ersten befüllen des Input-Buffer durch den Reader-Thread auftreten. Die die Lock-Variable wird bei jeder Befüllung einer Input-Buffer-Zelle vom Reader freigegeben, so dass der entsprechende Worker die Daten lesen und eine Simulationsrechnung starten kann.

```
1  logical :: stStart
2  integer :: i, first
3  integer :: numThreads
4  integer(kind=omp_lock_kind), allocatable, dimension(:) ::
       worker_read_lck
5  numThreads = omp_get_max_threads()
6  allocate(worker_read_lck(2:numThreads-1)
7  first = 2 !Index des ersten Worker-Threads im Team
8  !$OMP PARALLEL shared(numThreads, worker_read_lck, stStart) &
9  !$OMP &private(i, first)
10 !$OMP MASTER
11 init_locks:do i=first, numThreads-1
12   call omp_init_lock(worker_read_lck(i))
13   call omp_set_lock(worker_read_lck(i))
14 end do init_locks
15 stStart = .TRUE.
```

```
16 !$OMP FLUSH
17 !$OMP END MASTER
18 !$OMP BARRIER
19 outerloop:do while(private_ios)
20   !$OMP MASTER
21   innerloop:do i=2, numThreads-1
22     if(not stStart) then
23       call omp_set_lock(worker_read_lck(i))
24     end if
25   end do innerloop
26   !$OMP END MASTER
27 end do outerloop
```

Listing 5.1: Reader-Worker-Locks

```
1 module BUFFER
2   type input_buffer
3     !input buffer Pfad Nutzungsdatei
4     character(len=255) :: nutzung
5     !input buffer Pfad Raumdatei
6     character(len=255) :: raum
7     !input buffer Pfad Wanddatei
8     character(len=255) :: wand
9     !input buffer Drehwinkel
10    double precision :: winkel
11    !input buffer Anzahl zu untersuchender Jahre
12    integer :: anzahl
13    !input buffer Pfad Wetterdatendatei
14    character(len=255) :: try
15    !integer switch Format des DWD
16    integer :: dwdtry
17    !integer switch Luftwechsel
18    integer :: iswlw
19    double precision :: fcfix
20    !Passive Kuehlung [W/m^2]
21    double precision :: qpass
22    !Faktor interne Waermequellen
23    double precision :: qintfak
24    !Faktor Strahlung
25    double precision :: strfak
26    !integer Switch Lueftungsanlage
```

```
    integer :: iswlanlag
    !input buf Luftungsanlage vorhanden
    character(len=255) :: lanlag
    !integer Switch Theta Luft/Operativ
    integer :: iswtemp
    !integer switch langwellig
    integer :: iswlang
    !integer switch Interpolation im Z-Intevall
    integer :: iswinter
    !integer switch, steuert Ausgabe in AUS2
    integer :: iswpri
    !Anfangsstunde
    integer :: anz
    !Endstunde
    integer :: ez
    character(len=255) :: aus1, aus2, aus3
    !stale Indikator
    logical :: stale
  end type
contains
end module BUFFER
```

Listing 5.2: Definition des INPUT_BUFFER des HAUSer-Programms

```
!Deklaration des dynamisch allokierbaren array
type (input_buffer), allocatable, dimension(:) :: in_buf
integer numThreads
numThreads = omp_get_max_threads()
allocate(in_buf(2:numThreads-1))
```

Listing 5.3: Dynamische Speicher-Allokation des INPUT_BUFFER

5.2 Reader-Worker-Synchronisation

Die Interaktionen zwischen Reader und Worker, so wie zwischen Worker und Writer wurde konform zum Producer-Consumer Entwurfsmuster implementiert. Im Listing 5.3 wird ein Array abhängig von der Anzahl der Threads definiert, wobei jeder Array-Index jeder Thread-Nummer der Worker zugeordnet ist. Fortran2008-Array dürfen mit einem Index ungleich eins beginnen. Im Listing 5.3 beginnt der Array-Index mit der Nummer zwei, weil die Threads mit den Nummern null und eins fürs

Lesen und Schreiben der IO-Daten reserviert wurden.

Der Reader-Thread speichert Parameter jeder Zeile der HAUSer-Steuer-Datei jeweils in einer Struktur des Typs `input_buffer`. Dabei kann jeder Worker-Thread über die eigene Thread-Nummer den für ihn reservierten Buffer im Array `in_buf` finden. Die Thread-Nummer lässt sich über die OpenMP Funktion `omp_get_tread_num()` ermitteln Listings 5.3 und 5.4.

```
1 integer thID
2 input_buffer :: private_buffer
3 !$OMP PARALLEL private(thID, private_buffer) shared(in_buf)
4 thID = omp_get_thread_num()
5 if(thID > 1)
6   private_buffer = in_buf(thID)
7 end if
8 !$OMP END PARALLEL
```

Listing 5.4: Beispiel für das Referenzieren der Array-Position durch Thread-Eigennummer

Wie im Kapitel 4 beschrieben, soll False-Sharing vermieden werden. Listing 5.4 demonstriert das Kopieren einer geteilten Input-Buffer-Instanz in den lokale Bereich des Thread mit der entsprechenden Thread-Nummer. Die Variable `private_buffer` referenziert den lokalen Bereich des Threads.

```
1  outerloop: do while (private_ios == 0)
2    !$OMP MASTER
3    innerloop:do i=2, numThreads-1
4      call omp_set_lock(worker_read_lck(i))
5      !$OMP FLUSH
6      stale_temp = input_buffer(i)%stale
7      if(stale_temp) then
8        !read-Funktion liest eine Zeile aus file ins input_buffer
9        !iostat ist ungleich 0, falls EOF erreicht wurde
10       read(file,*,iostat=privatio_ios) input_buffer
11       shared_ios = private_ios
12       if(private_ios == 0) then
13         input_buffer(i)%stale = .FALSE.
14       else
```

```
15          !EOF erreicht , Reader darf den innerloop vorzeitig
     verlassen
16          input_buffer(i)%stale = .TRUE.
17          call omp_unset_lock(worker_read_lck(j))
18          exit innerloop
19        end if
20        !$OMP FLUSH
21      end if
22      call omp_unset_lock(worker_read_lck(i))
23    end do innerloop
24    !$OMP END MASTER
25  end do outerloop
```

Listing 5.5: Der Reader im parallelen Bereich des Programms

Im Listing 5.5 ist die Funktionsweise des Reader-Threads verkürzt dargestellt. Nach verlassen des Code-Abschnittes `outerloop` und bei Variable `stState` gleich Wahr, werden alle gesetzten Variablen `worker_read_lock` vom Reader freigegeben, da keine Zeile der Steuerdatei gelesen wurde. Ansonsten wird die Lock-Variable nur paarweise mit dem jeweiligen Worker genutzt.

```
1  thID = omp_get_thread_num()
2  outerloop: do while (private_ios == 0)
3    if(thID > 1) then
4      !worker darf erst dann die seine Arbeit starten ,
5      !wenn sein , mit dem Writer geteilter , Output-Buffer
6      !leer ist oder veraltete Daten hat
7      call omp_set_lock(worker_read_lck(thID))
8      !$OMP FLUSH
9      if(not input_buffer%stale) then
10       private_buffer = input_buffer(i)
11       input_buffer%stale = .TRUE.
12       call omp_unset_lock(worker_read_lck(thID)
13       !$OMP ATOMIC
14       worker_count = worker_count + 1
15       !$OMP FLUSH
16       !Zur Erinnerung: Fortran subroutine's bekommen
17       !die Felder by reference
18       call HAUS(private_buffer, private_out_buffer)
19       !$OMP ATOMIC
```

```
20        worker_count = worker_count - 1
21        !$OMP FLUSH
22        !Daten fuer Writer vorbereiten
23        call omp_set_lock(worker_write_lck(thID))
24        !HAUS subroutine hat die Laenge des Buffer-Array
25        !abhaengig von Anzahl zu iterierender Jahre
26        !definiert
27        length=size(private_out_buffer)
28        if(list_output_buffer(thID)%stale) then
29          allocate(list_output_buffer(thID)%output_buffer(length))
30          list_output_buffer(thID)%output_buffer = private_out_buffer
31          list_output_buffer(thID)%stale = .FALSE.
32          deallocate(private_out_buffer)
33        end if
34        call omp_unset_lock(worker_write_lck(thID))
35      else
36        !Abfrage des EOF-Zustands des Reader
37        !private_ios ungleich 0 bewirkt, dass
38        !Worker outerloop-Region verlaesst
39        !$OMP FLUSH
40        !$OMP ATOMIC READ
41        private_ios = shared_ios
42      end if
43      !Reader wird geweckt
44      !dieser blockiert sich selbst,
45      !falls keine Worker bereit sind
46      call omp_unset_lock(reader_lck)
47    end if
48  end do outerloop
```

Listing 5.6: Code-Block der Worker im parallelen Bereich des Programms

Das Listing 5.6 Zeigt die Funktionalität des Worker-Thread. Ein Worker verwendet Lock-Variablen, um kritische Abschnitte zu schützen. Das sind das Abholen der Daten aus dem Input-Buffer und wegschreiben in den Output-Buffer. Die Variable `private_ios` ungleich null unterbricht die Schleife `outerloop`. Wichtig ist die Variable `private_ios`, die das Ende der HAUSer-Steuerdatei anzeigt, erst nach dem Abholen der Daten durch den Worker aus dem Input-Buffer abzufragen. Ansonsten würde ein Worker die Schleife `outerloop` zu früh verlassen.

5.3 Worker-Writer-Synchronisation

```
1  outerloop: do while (private_ios == 0)
2    if(thID == 1) then
3      !Worker-id beginnt mit einer 2
4      writeloop:do i=2, numThreads-1
5        call omp_set_lock(worker_write_lck(i))
6        if(not list_output_buffer(i)%stale) then
7          length=size(list_output_buffer(i)%output_buffer)
8          do j=1, length
9            !eine Zeile in Erbensisdatei schreiben
10           write(Ergebnis_File, format) list_output_buffer(i))%
     output_buffer(j)
11         end do
12         deallocate(list_output_buffer(i)%output_buffer)
13         list_output_buffer(i)%stale = .TRUE.
14       end if
15       call omp_unset_lock(worker_write_lck(i))
16     end do writeloop
17     if(worker_count == 0) then
18       if(shared_ios != 0) then
19         if(nachlaufZaehler == 0) then
20           !Es gibt nichts mehr zu tun
21           !Writer darf outerloop verlassen
22           private_ios = -1
23         else
24           !beginnt mit der Anzahl Worker
25           !wird dekrementiert mit jedem
26           !Wegschreiben der Daten, falls
27           !EOF der Steuerdatei erreicht ist
28           nachlaufZaehler = nachlaufZaehler - 1
29         end if
30       else nachlaufZaehler = numberOfWorkers
31       end if
32     end if
33   end if
34 end do outerloop
```

Listing 5.7: Code-Block des Writer im parallelen Bereich des Programms

Writer-Thread schreibt die von Workern berechneten Ergebnisse der Simulationen

in die Ergebnisdatei. Der Nachlauf-Zähler wird benutzt, um die letzten Daten zu schreiben, nachdem alle Worker ihre Arbeit beendet haben und der Reader das End-Of-File der HAUSer-Steuerdatei erreicht hat.

6 Experimente und Ergebnisse

Die Experimente zum Beschleunigen des Programms HAUSer mit OpenMP wurden mit einem Gebäude durchgeführt, das aus einer so genannter Zone mit acht Bauteilen bestand. Die Wand-Bauteile bestanden aus drei bis vier Bauteilschichten. Die Fensterbauteile bestanden aus doppelt-verglasten Fenstern. Die Gebäudedaten wurden mehrfach in die Steuerdatei kopiert, so dass pro Steuerdatei 170 Simulationen ausgeführt wurden.

Tabelle 6.1 zeigt die gemessenen Ausführungszeiten des parallelen Programms HAUSer und die erzielten Speed-Up's. Für die Messungen wurde der Prozessor Apple M3 Pro verwendet. Dieser hat zehn so genannte Performance Kerne und vier Efficiency Kerne. Ab zehn Worker-Threads steig die Ausführungsgeschwindigkeit nicht mehr signifikant.

Anzahl Worker	Ausführungszeit[Sekunden]	Speed-Up
1	41,094	1
2	21,21	1,9
4	10,742	3,8
6	7,37	5,6
8	5,8	7,1
10	5,27	7,8
12	5,4	7,6
14	5,494	7,5
16	4,95	8,3

Tabelle 6.1: Speed-Up

7 Fazit

Ziel dieser Arbeit war es, das Programm HAUSer mit OpenMP zu parallelisieren. Außerdem musste Programm-Code in Fortran2008 umgesetzt und in Module aufgeteilt werden. Beide Ziele konnten erreicht werden. Die für Experimente zugrundeliegende Gebäudezone für das gesamte Gebiet von Bundesrepublik Deutschland zu simulieren, mit allen vom DWD zur Verfügung gestellten Testreferenzjahren als Wetterdaten, bei 300 Millisekunden pro Simulationslauf, würde 166 Stunden dauern. Ein Speed-Up um einen Faktor 7,8 würde die gesamte Simulationsdauer auf 21,3 Stunden reduzieren. Das Ergebnis spricht für sich.

Dennoch ginge noch mehr. Nicht alle Eingabedaten wurden per Reader-Thread verarbeitet. Es gab vier Dateien, deren Lesen und Verarbeiten nur sehr schwer von den Simulationssubroutinen zu entkoppeln gewesen wären. Der Autor dieser Arbeit hat eine Woche für den Versuch der Entkopplung aufgewendet und musste aufgeben, weil der Aufwand den Zeitrahmen einer Bachelorarbeit gesprengt hätte. Das gleiche gilt für die Ausgabedaten. Nur die HAUSer-Ergebnisdatei wurde per Writer-Thread verarbeitet. Die Zeitersparnis wäre nicht extrem gewesen. Nur 0,1 Millisekunden pro Team aus zehn Worker-Threads. Durch die vollständige Serialisierung der Eingabe- und Ausgabedaten hatten, für alle zwei Millionen Simulationsläufe, 5,5 Stunden eingespart werden können. Ein Speed-Up-Faktor von 1,3 beim geschätzt höherem Arbeitsaufwand eines Programmierers, einer Programmiererin als fürs Erreichen des Speed-UP-Faktors von 7,8. Was lässt sich aus dieser Schätzung schlussfolgern? Arbeitsökonomisch ungünstig.

Der Gedanke lässt sich weiter spinnen. Wie in Kapiteln 4 und 5 beschrieben, wurde der Reader-Thread so implementiert, dass Reader sich selbst per doppeltem Setzen eines Lock sperrt, wenn alle Input-Buffer belegt und alle Worker mit Rechnen beschäftigt sind. Angesichts der geringen Beschleunigung arbeitsökonomisch

ungünstig. Signifikante Speed-Up-Faktoren wurden lediglich auf den Performance-Kernen des Prozessors erreicht. Zwei auf Efficiency-Kernen aktiv wartende Threads (Reader und Writer) würden die Ausführungsgeschwindigkeit des HAUSer-Programms nicht beeinflussen.

OpenMP stellt eine interessante Möglichkeit dar, Simulationsprogramme durch das Parallelisieren zu beschleunigen. Im Fall des Programms HAUSer waren nur wenige grundsätzliche Änderungen notwendig. Das Programm kann unabhängig vom Betriebssystem und dem zugrundeliegenden Multithreading-System genutzt werden. Eine direkte Programmierung von parallelen Threads (z.B. Posix-Threads) würde, trotz der Platformunabhängigkeit einer Programmiersprache, zu Abhängigkeit vom Thread-System des Betriebssystems führen. Platformunabhängigkeit, Einfachheit und gute Beschleunigen von Berechnung zeichnen OpenMP als eines der nützlichsten Tools im Arsenal des Programmierers, der Programmiererin.

A Akronyme

ASCII American Standard Code for Information Interchange. 3

DWD Der Deutsche Wetterdienst. 3, 9, 31

OpenMP Open Multi-Processing. 4

TRY Testreferenzjahr. 3, 4

B Glossar

Anwendungslogik Der Begriff Anwendungslogik beschreibt den Teil eines Software-Systems, der für die Durchführung der zentralen Aufgaben der Anwendung zuständig ist. Ein Beispiel für Anwendungslogik ist eine Simulationsrechnung eines Gebäudes des Programms HAUSer. 4

dynamische Speicherverwaltung Die Fähigkeit einer Programmiersprache und deren Compiler, die Länge eines Feldes dynamisch zur Laufzeit des von der Programmiersprache erzeugten Programms festzulegen. Beispiel dafür ist die ALLOCATE-Funktion des Fortran2008. 1

gFortran Fortran Compiler der GNU Compiler Collection. 4

Gleichung von Navier-Strokes Dient u.a. der Beschreibung von Lüftungswärmeverlusten in Gebäuden. 9

Interne Gewinne Wärmequellen, die primär nicht der Heizung dienen. Z.B. Menschen, Haustiere, Haushaltsgeräte, etc. 9

Reader-Thread Ein Thread, der für das Lesen der Steuerdatei und Aufgabenverteilung an Worker-Thread zuständig ist. 13

Refactoring Ein Prozess der Umstrukturierung des bestehenden Codes, ohne dessen externes Verhalten zu änder. Ziel des Refactoring ist es, die Verständlichkeit, Lesbarkeit und Erweiterbarkeit des Codes zu verbessern. 1

Strahlungsaustausch Das Planksche Strahlungsgesetzt beschreibt den Austausch der Energie zwischen Festkörpern durch Strahlung. 9

Testreferenzjahr Ein Testreferenzjahr des DWD ist eine Zusammenstellung meteorologischer Daten für ein spezifisches geographisches Gebiet für den Zeitraum eines Jahres. 3

Worker-Thread Ein Thread, der für die Anwendungslogik eines Programms zuständig ist. Z.B. für Simulationsrechnung. 1, 13, 14

Writer-Thread Ein Thread, der für das Serialisieren der von den Worker-Threads berechneten Daten und das Schreiben dieser Daten auf den Festspeicher zuständig ist. Dieser Thread übernimmt keine Aufgaben der Anwendungslogik des Programms. 1, 13

Zone Die Zone einer Gebäudesimulation entspricht einem typischen Raum des zu untersuchenden Bauwerks. Durch die Verwendung einer Zone wird der Berechnungsaufwand gesenkt und die Simulationsrechnung produziert weniger aber genauere Daten. Das erleichtert die Auswertung der Ergebnisse für Forschende. 29

C Literaturverzeichnis

[1] Deutscher Wetterdienst, "Testreferenzjahre." `https://www.dwd.de/DE/leistungen/testreferenzjahre/testreferenzjahre.html`, accessed on 08.02.2024.

[2] Timothy G. Mattson, Yun (Helen) He, Alice E. Koniges, *The OpenMP Common Core*. The MIT Press, 2019.

[3] W. Feist, *Thermische Gebäudesimulation*. C. F. Müller, 1994.