Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Projekt-Ausarbeitung
Parallelrechnerevaluation

Capture & Replay für Fortran- und MPI-Programme

verfasst von

Marcel Heing-Becker

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Informatik (M. Sc.)
Matrikelnummer:	7004013
Betreuer:	Christian Hovy
	Dr. Julian Kunkel

Hamburg, 2017-09-04

Inhaltsangabe

Programme im HPC-Umfeld sind auf massive parallele Arbeit ausgelegt, die beispielsweise durch eine MPI-Implementierung realisiert wird. Stellt sich im Zuge des Testens heraus, dass Tests nicht für einen Einzelknotenbetrieb in einer günstigen Testumgebung ausgelegt sind – oder überhaupt nicht existieren, sind Lösungen gefordert, die auch ohne tiefgreifende Code-Kenntnisse Referenzergebnisse liefern können. Für Fortran-Programme existiert ein Werkzeug namens *FortranTestGenerator*, das aus dem Produktivbetrieb heraus Ein- und Ausgabedaten von Subroutinen abspeichern und Testtreiber-Programme dazu erstellen kann. Dieser Bericht untersucht am Beispiel der Klimasimulation *ICON*, welche Maßnahmen nötig sind, diesen Ansatz analog auf MPI-Kommunikation auszuweiten.

Inhalt

1 Motivation

Programmcode, ganz gleich welchen Alters oder Zweck, unterliegt in der Praxis oft dem Problem fehlender oder bloß geringer Test-Abdeckung im Sinne von Unit-Tests. Insbesondere im Zuge von Alterung und Entwickler-Fluktuation steigt die Schwierigkeit des Nachreichens von Tests bei nicht-trivialem Code.

Ein Hilfsansatz liegt in der Annahme, dass der Programmcode innerhalb seiner beanspruchten Wertebereiche von Entwicklern wie Nutzern als akzeptiert erachtet wird. Aus dieser Prämisse ergibt sich eine Grundlage für Regressionstests, wenn angenommen werden kann, dass die Aus- oder Rückgabe einer Code-Subroutine in Abhängigkeit der produktiv genutzten Eingaben korrekt ist.

Es verbleibt die Schwierigkeit, dass eine Code-Subroutine nicht ohne Weiteres isoliert von der Anwendung aufgerufen werden kann: Es existieren Abhängigkeiten von globalen oder modulfremden Daten, und auch die Eingaben müssen zuvor berechnet oder angeordnet werden.

Der in diesem Bericht vorgestellte Lösungsansatz nennt sich *Capture & Replay*: Für Fortran kann der *FortranTestGenerator* (Kapitel 2) Subroutinen-spezifisch die Instrumentierung des Codes vornehmen, sodass Eingabedaten, Abhängigkeiten und Ausgaben zum Zeitpunkt ihres jeweiligen Aufrufs auf die Festplatte persistiert werden. Von dort aus können diese nun für das Setup eines isolierten Durchlaufs verwendet werden.

Kommuniziert ein Programm innerhalb einer solchen Routine hingegen zusätzlich auch über MPI zu anderen Prozessen innerhalb eines Netzwerks, so fehlen maßgebliche Eingaben für die lokale Rechnung. Darüber hinaus ist zu befürchten, dass ein Programm im Testbetrieb ein zum Produktivdurchlauf äquivalentes MPI-Setup benötigt, da der MPI-Standard grundsätzlich Reflexion erlaubt. Ein erstrebenswertes Ziel ist daher, zusätzlich die Möglichkeit herzustellen, zu Testzwecken einen einzigen Prozess lokal mit derselben Datensicht zu versorgen, die er produktiv hätte.

In den nachfolgenden Kapiteln wird die Nutzung von MPI innerhalb ICONs beleuchtet (Kapitel 3), die Entwicklung und Fähigkeiten eines im Zuge dieses Projekts entwickelten MPI-Wrappers vorgestellt (Kapitel 4) und anhand eines Beispiel-Programmcodes aus ICON in Kombination mit FTG präsentiert (Kapitel 5). Weitere Ansätze und Implementierungen finden sich in Kapitel 6 erwähnt.

2 FortranTestGenerator (FTG)

Der *FortranTestGenerator* (FTG) ist ein Python-Scriptsatz[1], dessen Zweck die Herstellung der Capture & Replay-Funktionalität (C&R) für Subroutinen von Fortran-Programmen ist [HK16].

2.1 Capture

Im *Capture*-Teil werden Eingaben der Subroutine, Abhängigkeiten von anderen Modulen und Berechnungsergebnisse aufgezeichnet. Die Funktionsweise von FTG wird anhand Abb. 2.1 erläutert.

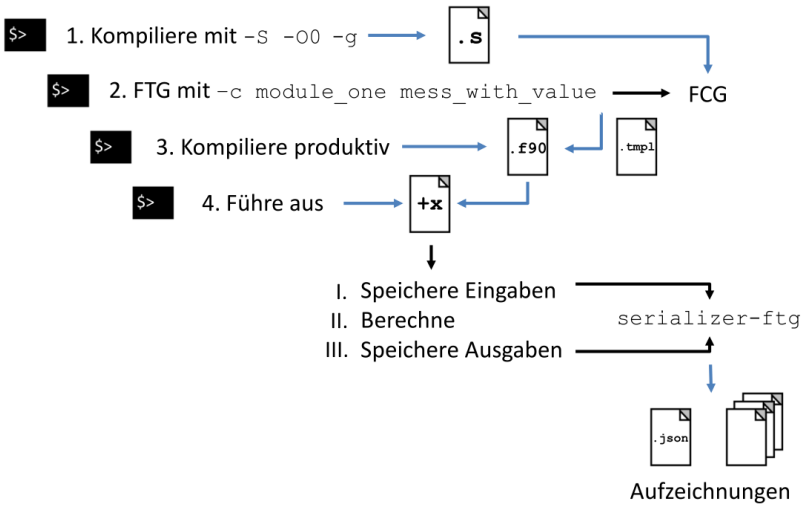

Abb. 2.1: FTG: Herstellung Capture-Funktionalität

Der erste notwendige Schritt besteht im Kompilieren des Fortran-Quellcodes, wobei ein Durchlauf die Assemblerausgabe und Debug-Informationen (GNU Fortran: `-S -g`) unter Ausschluss von Optimierungen (`-O0`) zu erzeugen hat.

Im zweiten Schritt ist der Aufruf von FTG zu tätigen, welcher den Parameter `-c` unter Angabe des Zielmoduls und dessen Subroutine zu tragen hat. FTG bedient sich eines Programms mit dem Namen *FortranCallGraph*[2], dessen Zweck wesentlich aus zwei Teilen besteht: 1.) Es ermittelt aus dem Programmcode die Abhängigkeiten von Eingabevariablen und modulfremden Variablen, 2.) es analysiert den effektiv kompilierten

[1] Repository: `https://github.com/fortesg/fortrantestgenerator`
[2] Repository: `https://github.com/fortesg/fortrancallgraph`

Aufrufgraphen einer Subroutine anhand der Assembler-Dateien, sodass auch für die aufgerufenen Subroutinen die Abhängigkeiten ermittelt werden können.

War dieser Schritt erfolgreich, so ist der Quellcode der Subroutine instrumentiert: Dieser erhält nun Methoden zum Initialisieren des Aufzeichnens, zum Aufzeichnen aller Eingaben und Abhängigkeiten und zum Aufzeichnen aller Ausgaben und möglicherweise veränderter Abhängigkeiten. Die Serialisierung der Daten wird von der hinzuzulinkenden Bibliothek *serialbox*[3] übernommen, alle durch FTG vorgenommenen Modifikationen am Code sollen sich dabei transparent für die Subroutinen-Berechnungen verhalten.

Die Besonderheit von FTG besteht in der Vorlagenverwendung für den einzufügenden Code: In spezifizierten `tmpl`-Dateien finden sich verschiedene Sektionen, beispielsweise für die Initialisierung oder die Subroutinen-Präambel. Dies erlaubt die Erstellung von Vorlagensätzen für spezifische Anwendungen oder Zwecke, die zusätzlichen Anforderungen gerecht werden müssen, etwa dem Schalten von Zuständen genau um den Aufruf herum oder dem vorzeitigen Beenden des Aufzeichnens. In Abhängigkeit der Vorlage wird die Zielmodul-Datei entsprechend modifiziert.

Im dritten Schritt kann der instrumentierte Quellcode daraufhin produktiv kompiliert und im vierten Schritt gestartet werden. Für jeden Aufruf der instrumentierten Subroutine werden Dateien in einem in Schritt zwei spezifizierten Verzeichnis angelegt, welche den Namen der Subroutine, die Nummer des Aufrufs und einen eindeutigen Variablenbezeichner tragen. Zum Ende des Aufrufs geschieht dasselbe für die Ausgabedaten.

Ist das Programm beendet, erlauben Werkzeuge der *serialbox*-Bibliothek die Inspektion und den Vergleich der aufgezeichneten Daten.

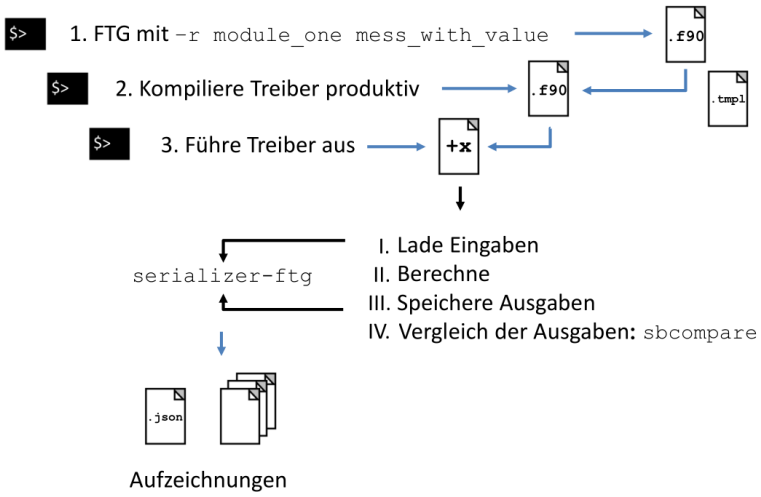

Abb. 2.2: FTG: Herstellung Replay-Funktionalität

[3]Repository: `https://github.com/fortesg/serialbox-ftg`

2.2 Replay

Ist der Capture-Vorgang reibungslos verlaufen, so kann der *Replay*-Teil von FTG getestet werden. Siehe dazu Abb. 2.2.

Für das Modul, das zuvor zum Aufzeichnen von FTG instrumentiert wurde, lässt sich nun eine neue Programmcode-Datei erzeugen. Der Aufruf von FTG im ersten Schritt wird dazu mittels `-r` und den Modul- und Subroutinenangaben durchgeführt. Vor demselben Hintergrund wie in Abschnitt 2.1 wird der Replay-Programmcode auch über Vorlagen erzeugt. Darin zu finden ist vor dem Aufruf der Subroutine das Laden der Eingaben und Abhängigkeiten sowie die Ausgabe in ein zusätzliches Verzeichnis für die Ergebnisdaten.

Ist der Replay-Code im zweiten Schritt produktiv erfolgreich kompiliert worden, kann er anschließend ausgeführt werden. Nach Beendigung kann nun geprüft werden, ob alle Ausgaben des aufzeichnenden Produktivlaufs aus Abschnitt 2.1 mit denen des Replay-Treibers übereinstimmen: Das Programm `sbcompare`[4] ist hilfreich zum Ermitteln von Differenzen.

Im Idealfall sind keine Unterschiede auszumachen, sodass mit dem Replay-Treiber für einen Satz Eingabedaten jetzt eine synthetisierte Unit-Test-Basis zur Verfügung steht.

[4]Repository: `https://github.com/fortesg/serialbox-compare`

3 MPI & ICON

Für diese Projektarbeit ist ein Abstecken des umfangreichen Felds von MPI für die zielführende Anwendung mit ICON notwendig.

MPI

MPI (für *Message Passing Interface*) ist eine Spezifikation für C- und Fortran-Programme, die der Organisation von verteilten Rechenkonten und ihrem Datenaustausch untereinander dient. Es existieren gängige offene Implementierungen (z. B. *OpenMPI*, *MPICH*) wie auch proprietäre (etwa von *IBM* oder *Microsoft*). Der Standard beschränkt sich auf Interface-Deklarationen, sodass die Implementierung beliebig spezialisiert werden kann.

Der MPI-Standard in Version 3.1 legt mehr als 400 zu implementierende Funktionen fest, welche sich grob in folgende Gruppen aufteilen lassen können: Netzwerk-Topologie, Datenaustausch, Datei-E/A, entfernten Speicherzugriff und Datentyp-Konfiguration nebst etlicher Hilfsfunktionen [For15].

ICON

ICON ist ein Software-Modell für Klimaprognosen und Wettervorhersagen, dessen Entwicklung hauptsächlich vom *Max-Planck-Institut für Metereologie* und dem *Deutschen Wetterdienst* getragen wird. Der Quellcode ist in Fortran geschrieben und benutzt, wie im HPC-Umfeld üblich, eine MPI-Schnittstelle zur Verteilung der Rechenlast auf eine nahezu beliebige Zahl von Rechenknoten.

Um dieser Menge sinnvoll im Projektrahmen entgegnen zu können, ist zuerst eine Untersuchung des Nutzungsrahmens durch ICON interessant. Mit einem Versionsstand von Juni 2017 wurden mithilfe von `grep` alle Codezeilen mit `CALL MPI_` beginnend gesammelt, aufgrund der Case-Unempfindlichkeit von Fortran zunächst normalisiert und dann summiert. Zu berücksichtigen ist bei ICON jedoch, dass MPI-Aufrufe weitestgehend gekapselt sind (Modul `mo_mpi`) und sogar ein Compiler-Makro existiert, das den Programmcode mutmaßlich ohne MPI-Unterstützung kompilieren kann (`-DNOMPI`). Die reine Summe der Aufruf lässt insofern kaum Schlüsse zu, in Abb. 3.1 sind die jedoch die fünf am häufigsten gezählten Aufrufe von ICON gelistet. Insgesamt wurden 72 verschiedene MPI-Aufrufe gezählt.

Dennoch eignet sich diese Liste als Leitfaden, verwendete MPI-Konzepte zu erkennen und Versuche mit diesen zu unternehmen. Von Bedeutung ist dabei jeweils, ob ein Aufruf eingehende Daten erwartet (wie `MPI_Recv`), welche von einem Tool aufzuzeichnen wären, Daten versendet (etwa `MPI_Send`), welche im Standalone-Betrieb wegzuwerfen wären, oder hybride Aufrufe (zum Beispiel `MPI_Allreduce`). Die Spezifikation der Semantik der

MPI-Aufruf	#
`MPI_Send`	44
`MPI_Recv`	43
`MPI_Irecv`	41
`MPI_Isend`	40
`MPI_Allreduce`	39

Abb. 3.1: Top 5 der gezählten Codezeilen für MPI-Aufrufe

Aufrufe findet sich sowohl im MPI-Standard als auch in den Dokumentationen gängiger Implementierungen.

ICON war bereits Untersuchungsgegenstand des FTG, sodass dieses speziell für ICON entwickelte Vorlagensätze bereitstellt (`icon_standalone`), die die MPI-Initialisierung vornehmen und unbedingte Module laden.

4 Projekt: MPI-Wrapper

Das Ziel dieser Projektarbeit ist die erfolgreiche Demonstration von C&R mit einem MPI-Mechanismus in Kombination mit FTG anhand von Beispielen aus ICON. Zuerst werden daher die technischen Rahmenbedingungen von MPI zur Herstellung von C&R beleuchtet und anschließend die Implementierung vorgestellt.

4.1 MPI-Wrapping

MPI bietet als integralen Bestandteil neuerer Versionen einen *Profiling*-Modus für Werkzeuge an, die sich in MPI-Aufrufe von Programmen einhaken möchten. Für alle C- und Fortran-MPI-Aufrufe, die den `MPI_`-Präfix besitzen, existiert ein Aufruf mit `PMPI_`-Präfix. Dabei wird die MPI-Implementierung so kompiliert, dass die `MPI_`-Aufrufe *schwache Symbole* der `PMPI_`-Aufrufe sind, wodurch sie als überschreibbare Aliase fungieren. Im Regelfall ruft die Neudefinition eines existierenden Compiler-Objekt-Symbols im Linkvorgang einen Fehler hervor. Anders bei schwachen Symbolen, deren Prozedureinstiegsadressen überschreibbar sind. Damit ist der Betrieb eines MPI-Wrappers auf zwei Arten möglich:

- Durch Voranstellen eines Compiler-Objekts mit den abzufangenden Symbolen im Linkprozess, sodass dieses Teil der Anwendung wird.
- Durch Erzeugen einer dynamischen Bibliothek mit `MPI_`-Symbolen, welche über die Umgebungsvariable `LD_PRELOAD` den dynamischen MPI-Bibliotheken vorangestellt wird.

Letzter Ansatz zeichnet sich durch eine höhere Portabilität zwischen verschiedenen MPI-Anwendungen aus, ist jedoch nicht – wie gezeigt werden wird – für jeden Zweck exklusiv einsetzbar.

Der grundlegende Aufbau einer zu wrappenden C-MPI-Prozedur gestaltet sich wie in Listing 4.1. Bei der Implementierung ist stets zu beachten, dass die Symbole zwischen C und Fortran nicht kompatibel sind, maßgeblich aufgrund der folgenden Eigenschaften:

- C-Prozeduren, wenn sie aus Fortran unter demselben Namen aufrufbar sein sollen, müssen auf das `_`-Suffix enden.
- Während die C-Prozeduren von MPI meist einen Rückgabewert als Erfolgsindikator besitzen, wird dieser in Fortran über einen `INTENT(OUT)`-attributierten Aufrufparameter gesetzt.

```
1 int MPI_Xyz (/* args */) {
2   /* Vorarbeit. */
3   int return_value = PMPI_Xyz(/* args */);
4   /* Nacharbeit. */
5
6   return return_value;
7 }
```

Listing 4.1: MPI-Wrapping

- Etliche Objekte von MPI (etwa `MPI_Type` oder `MPI_Request`) müssen zwischen den Sprachkontexten konvertiert werden. Der MPI-Standard stellt für die allermeisten Fälle jedoch `MPI_..._f2c`- und `MPI_..._c2f`-Methoden für diesen Zweck bereit.

Es existiert ein Werkzeug namens `wrap`[1], das die Erzeugung sämtlicher MPI-Aufrufdelegationen aus Fortran an C übernimmt und auch die Konvertierungsaufrufe selbstständig einfügt. Einem Wrapper-Entwickler wird insoweit das händische Erzeugen sämtlicher Signaturen und Umwandlungen erspart, sodass das erzeugte Gerüst auch für den im folgenden vorgestellten Wrapper verwendet wird.

4.2 Implementierung

In diesem Abschnitt werden die Implementierungen der für das Projekt gewählter MPI-Wrapper-Aufrufe vorgestellt und mit den einhergehenden Problemen oder speziellen Anforderungen an ein Test-Setup erläutert. Der Versionskontext von MPI ist 3.1, die Signaturen und Implementierungen sind vorrangig in C beschrieben.

`MPI_Send` & `MPI_Recv`

Bei diesem Aufrufpaar handelt sich um die einfachste in MPI zu realisierende Punkt-zu-Punkt-Kommunikation zwischen zwei Prozessen. Auf einen mittels `MPI_Send` (Listing 4.2) sendenden Prozess mit Zieldeskriptor hat der empfangende Prozess über `MPI_Recv` (Listing 4.3) die Daten entgegenzunehmen. Die beiden Aufrufe blockieren den jeweiligen Prozess bis zum erfolgreichen Austausch.

Für den C&R-Modus eines Wrappers ist der Aufruf von `MPI_Send` uninteressant. Ganz anders `MPI_Recv`: Die zu empfangenden Daten befinden sich nach dem Aufruf im vom `buf` referenzierten Speicher, welche sich als bis zu `count`-viele Elemente vom Typ `datatype` beschreiben. Mit `source` und `tag` können Empfangsfilter über Quelle und Nachrichtentyp an den Empfangsaufruf gekoppelt werden. Die zu diesem Aufruf assoziierten Metadaten finden sich in der Referenz `status`. Das `struct MPI_Status` ist nicht über den Standard definiert, allerdings existieren Getter- und Setter-Methoden,

[1] Repository: `https://github.com/LLNL/wrap`

die die Anzahl der tatsächlich übermittelten Elemente und einen Abbruch indizieren beziehungsweise ermitteln können. Erlaubt man die Annahme, dass `struct MPI_Status` flach definiert ist, lässt es sich jedoch in Abhängigkeit seiner Bytegröße serialisieren.

Aufzuzeichnen und entsprechend wiederherstellbar zu machen sind gleichermaßen Nutz- wie Metadaten. Bei allen aufzuzeichnenden MPI-Daten ist zu berücksichtigen, dass diese nicht zwangsläufig zusammenhängend im Speicher vorliegen: Für den Aufruf von `MPI_Type_size` auf einen vorliegenden `MPI_Datatype` wird eine Bytegröße zurückgegeben, die sich typbedingt aus verschiedenen Liegenschaften im Speicher summieren kann und damit nicht im Sinne der Zeigerarithmetik zusammenhängend bei `buf` beginnend vorliegen muss. Intern sind zum Abspeichern und der Wiedergabe daher für alle Methoden Aufrufe von `MPI_Pack` und `MPI_Unpack` auf diesen Speicherbereichen unabdingbar.

```
1 int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int
    ↪ dest, int tag, MPI_Comm comm)
```

Listing 4.2: `MPI_Send`

```
1 int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int
    ↪ source, int tag, MPI_Comm comm, MPI_Status *status)
```

Listing 4.3: `MPI_Recv`

`MPI_Isend` & `MPI_Irecv`

Wenngleich sie in der Kommunikationssemantik zu den Aufrufen aus dem vorherigen Abschnitt identisch sind, blockieren diese Aufrufe nicht – entsprechend ihres für MPI üblichen `I`-Präfixes (für *immediate*) im Namen. Ihnen ist ein zusätzlicher Parameter vom Typ `MPI_Request` gegeben, welcher für einen beliebig später, aber notwendigerweise folgenden Aufruf von `MPI_Wait` verwendbar ist.

`MPI_Wait` erlaubt ein spezifisches Warten für einen anzufragenden `MPI_Status`, wohingegen `MPI_Waitall` auf alle schwebenden Aufrufe wartet. In jeden Fall darf der im initialen Aufruf referenzierte Speicher erst nach Beendigung der Aufrufe freigegeben oder gelesen werden.

In diesem Projekt wurde nur auf `MPI_Wait` eingegangen, `MPI_Waitall` bleibt unangetastet. Im Unterschied zu den blockierenden Varianten ist eine zusätzliche Herausforderung im Umgang mit asynchronen Daten, dass der Wrapper sich die Metadaten des dazugehörigen `MPI_Request` des ersten Aufrufs zu merken hat, und erst nach Abschluss des spezifischen `MPI_Wait` zusammen mit den empfangenen Daten vollständig aufzeichnen kann.

`MPI_Reduce` & `MPI_Allreduce`

Hierbei handelt es sich um kollektive Rechenoperationen: Auf allen Teilnehmern des angegebenen Kommunikators werden Daten eines Typs erfasst und mit einer MPI-nativen Aggregation berechnet. Das Ergebnis steht bei `MPI_Reduce` (Listing 4.4) im angegebenen Prozess `root` im Speicher `recvbuf`, bei `MPI_Allreduce` (Listing 4.5) erhalten alle Teilnehmer dieses.

```
1 int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
    ↪ MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

Listing 4.4: `MPI_Reduce`

```
1 int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,
    ↪ MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Listing 4.5: `MPI_Allreduce`

Die Speicherung und Wiedergabe unterliegen keinen besonderen Umständen, dennoch erlaubt der MPI-Standard eine Eigenart, von der auch ICON Gebrauch macht: das Flag `MPI_IN_PLACE`. Bei Intra-Kommunikatoren ist es unter der Angabe von `MPI_IN_PLACE` im Parameter `sendbuf` zulässig, dass sich gleichermaßen Sende- wie Empfangsdaten im `recvbuf` verwiesenen Speicher befinden. Dabei von Bedeutung ist die Definition von `MPI_IN_PLACE`: Je nach Sprachkontext handelt sich dabei um das Makro einer Konstante (C) oder einen Zeiger (Fortran). Wird der MPI-Wrapper in C geschrieben, aber jedoch aus Fortran aufgerufen, kann der C-Wrapper das Argument nicht fehlerfrei an die `PMPI_`-Fassung durchreichen, da diese mit einem zu Fortran inkompatiblen Symbol die Prüfung auf `MPI_IN_PLACE` vornehmen würde. Der korrekte Adressat für diesen speziellen Fall wäre für Aufrufe aus Fortran-Programmcode der `pmpi_..._`-Aufruf.

Glücklicherweise wies ein ähnliches Programm namens *ReMPI*[2] das Problem in der Art auf, dass dieser Sonderfall einst ausgeklammert wurde, um den damit entstehenden `SEGFAULT`s vorzubeugen. Zusammen mit den Entwicklern konnte aber eine Lösung für OpenMPI gefunden werden – ohne Anspruch auf globale Gültigkeit[3].

`MPI_Comm_rank` & `MPI_Comm_size`

Die Aufrufe dieser Methoden erlauben dem MPI-Programm Reflexion:

- `MPI_Comm_rank`: Diese Methode teilt dem Programm mit, welcher MPI-Rang ihm zugeteilt wurde. Der MPI-Rang und -Kommunikator erlauben die eindeutige Identifikation eines Prozesses zur MPI-Laufzeit, weshalb diese Information gleichermaßen für den Wrapper zum Zeitpunkt des Aufzeichnens von Bedeutung ist, im

[2]Repository: `https://github.com/PRUNERS/ReMPI`
[3]Bug-Tracker: `https://github.com/PRUNERS/ReMPI/issues/8`

Replay-Modus aber unbedingt einen vom Nutzer gewählten Wert für die Testeinheit mitteilen muss.

- `MPI_Comm_size`: Erfordert die Programmlogik zu wissen, wie viele andere Prozesse sich zusätzlich im angegebenen Kommunikator befinden, kann es diesen Aufruf verwenden. Für den Fall, dass sich in Abhängigkeit des Rückgabewerts der Programmfluss ändert, muss dieser Aufruf im Replay-Modus einen vom Nutzer spezifizierten Wert zurückgeben können, um dem erklärten Ziel des Einzelprozessdurchlaufs gerecht werden zu können. Es handelt sich insofern um die kleinste Form von vorgetäuschter MPI-Topologie.

Die Projekt-Implementierungen geben bisweilen nur pauschale Rückgabewerte, die nicht nach verschiedenen Kommunikatoren unterscheiden können.

4.3 Interface

Zur Kontrolle des Wrappers zur Programmlaufzeit stellt dieser ein Interface für verschiedene Zwecke zur Verfügung. Da der Wrapper erst aktiv aktiviert werden muss, ist es derzeit unabdingbar, das zu untersuchende Programm im Kompilierungs- und Linkprozess auf diese zusätzliche Abhängigkeit umzustellen.

Wrapper-Aufruf	**Funktion**
`rwi_reset_counter()`	Setzt den Capture- oder Replay-Modus an den Anfang zurück.
`rwi_set_comm_size(int v)`	Setzt die Rückgabe von `MPI_Comm_size` auf `v`.
`rwi_set_enabled(int v)`	Aktiviert/deaktiviert den Wrapper.
`rwi_set_replay_data(int8_t*, const size_t)`	Setzt eigene Nutzdaten – siehe nächsten Abschnitt.
`rwi_set_replay_filter(void (*f)(void *buf, const size_t s))`	Wende `f` auf die Rückgabedaten `buf` bei `s` Bytes an.
`rwi_set_replay_enabled(int m, int v)`	Wechsele in oder aus dem Replay-Modus, setze Replay-Rang auf `v`.

Abb. 4.1: Kontroll-Methoden des Wrappers

4.4 Eigene Daten

Ein gewünschtes Feature für die Projektarbeit war außerdem die Möglichkeit, die Wiedergabe mit eigenen MPI-Nutzdaten aus dem Programmcode heraus zu versorgen. Möchte

der Benutzer des Testtreibers etwa andere MPI-Eingabewerte ausprobieren, so muss dies durch Veränderung der Binäraufzeichnung des Wrappers erfolgen. Alternativ bietet der Projekt-Wrapper mittels `rwi_set_replay_data(int8_t *tape, const size_t s)` an, ein Quasi-Band `tape` der Länge von `s` Bytes einzulegen.

Ein Modus der Verwendung ist die Überlagerung mit aufgezeichneten Daten: So wird anstelle der aufgezeichneten Nutzdaten das Band um die erforderliche Menge Bytes gelesen, während die Metadaten jedoch weiterhin von der Festplatte bezogen werden.

Darüber hinaus ist nicht einmal eine Aufzeichnung erforderlich, solange der Datenstrom hinreichend groß ist, jedoch werden die Metadaten vom Wrapper auf unkritische Standardwerte gesetzt, weshalb die Verwendung ohne jegliche MPI-Aufzeichnung nur mit Vorsicht zu genießen ist. Ein Beispiel findet sich in Listing 4.6, `rwi_set_replay_data_vi` ist eine Methode, die die Größe für gegebene `INTEGER(4)`-Vektoren im Fortran-Kontext vorberechnet.

```
1 INTEGER :: i = 0
2 CALL rwi_set_replay_data_vi((/4,5,6/))
3
4 DO WHILE (i .LT. 3)
5   CALL MPI_Recv(value, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
      ↪ MPI_COMM_WORLD, status, ierr)
6   i = i + 1
7   PRINT *, value ! Ausgaben: 4,5,6
8 END DO
```

Listing 4.6: Eigene Daten

4.5 Interna

Der Projekt-MPI-Wrapper zeichnet sämtliche MPI-Eingangsdaten in einem Pfad relativ zur Anwendung auf, für jeden MPI-Rang in einen eigenen Unterordner. Innerhalb eines MPI-Prozesses wird für jeden empfangenen Programmaufruf eine Aufzeichnungsdatei angelegt. Die total-geordnete Aufzeichnungsreihenfolge bedingt damit auch die Reihenfolge der Wiedergabe der Daten für diesen Prozess. Im Wiedergabemodus von dieser Ordnung abweichende MPI-Aufrufe führen damit zu korrumpierten Daten, weshalb etwa die Notwendigkeit für die Nutzerbestimmung der Rückgabe von `MPI_Comm_size` besteht, und möglicherweise noch weiterer topologischer Informationen in Zukunft.

Aufgezeichnete Metadaten und Daten werden nach `MPI_Pack` serialisiert, Metadaten wie etwa `MPI_Status` stehen stets am Dateianfang.

Für die Vormerkung von asynchronen Aufrufen (`MPI_Irecv`) werden zunächst die erforderlichen Metadaten – Speicheradresse, Datengröße, weitere MPI-Metadaten – in einem eigenen Verzeichnis abgelegt, deren Dateiname aus der SHA256-Prüfsumme von `MPI_Request` besteht. Dieses Vorgehen ermöglicht ein implementierungsunabhängiges Verfahren, die vorgemerkten Daten beim Aufruf von `MPI_Wait` wiederzufinden.

5 Beispielroutine in ICON

Mit der bestehenden Implementierung kann nun ein Beispiel von ICON unter die Lupe genommen werden, zuvor ist jedoch eine Modifikation der FTG-Vorlagen nötig.

5.1 FTG

Als Grundlage wird der Vorlagensatz `icon_standalone` gewählt. Zunächst ist bei der Initialisierung des Aufzeichnens, welche sich in `capture_aftersubroutine.tmpl` in der Subroutine befindet, die den Suffix `_init_for_capture` trägt, der Programmcode aus Listing 5.1 einzufügen. Dieser aktiviert den Wrapper für einen gewählten Prozess `my_mpi_id`, kurz vor der Ausführung von Interesse.

```
1 IF (my_mpi_id .EQ. 0) THEN
2   CALL rwi_set_enabled(1)
3 END IF
```

Listing 5.1: Capturing

Damit ist der Aufzeichnung genüge getan, da der Wrapper aktiv in den Replay-Modus umgeschaltet werden muss. Das weitere Vorgehen ist dem Kapitel 2 zu entnehmen.

Zum Erzeugen eines geeigneten Testtreibers sind weitere Maßnahmen nötig: In der Datei `replay.test.tmpl` wird ein Fortran-Programm vorgelegt, das vor dem Aufruf der zu testenden Subroutine die in Listing 5.2 gezeigte Präambel erfordert. Nach dem Aktivieren des Wrappers wird dieser in den Replay-Modus umgeschaltet und gibt dabei einen gewählten MPI-Rang vor. Darüber hinaus hat dieser eine MPI-Topologiegröße vorzugeben, die möglicherweise größer ist als die, die aus dem einzigen gestarteten Testprozess besteht.

```
1 CALL rwi_set_enabled(1)
2 CALL rwi_set_replay_enabled(1, 0)
3 CALL rwi_set_comm_size(4)
```

Listing 5.2: Replay

Um dem Kompiliervorgang das Laden weiterer Modulabhängigkeiten zu ersparen, ist zusätzlich an geeigneter Stelle ein Fortran `INTERFACE` einzufügen, das die Signaturen aller benötigten `rwi_`-Aufrufe enthält.

5.2 ICON

Die Suche nach einer geeigneten Routine als demonstrationsfähiges Beispiel erfordert zunächst die Wahl eines geeigneten Experiments. Als günstig im Zeitaufwand wie in der Größe der Eingabedaten erwies sich das im ICON-Codeverzeichnis befindliche `run/exp.atm_amip_noforcing_notransport_test`, das mit vier Prozessen durchgeführt wird.

Es konnte schließlich anhand der Subroutine `calculate_maxwinds` im Modul `mo_nh_supervise` ein erfolgreiches Experiment durchgeführt werden: Zunächst war der Umgang mit FTG ohne die entsprechenden Anpassungen in Abschnitt 5.1 auf Erfolg zu testen, das entsprechend der ursprünglichen Schwierigkeit weiterhin mit vier MPI-Prozessen zu starten ist.

Nachdem die Ausgabedaten als miteinander übereinstimmend bestätigt werden konnten, wurde das Experiment darauf mit den Projektanpassungen durchgeführt. Als Zielprozess diente dazu derjenige mit MPI-Rang 0. Beobachtet man die Aufrufe des Wrappers, so bezieht dieser Prozess in der Subroutine Daten aller Prozesse aus dem Aufruf von `MPI_Reduce`.

Ist auch der Durchlauf mit FTG- und MPI-Aufzeichnungen abgeschlossen, kann der Testtreiber mit `mpiexec -np 1` gestartet und seine Ausgabe mit den Referenzwerten verglichen werden. Für die erwähnte Testroutine ist es erheblich, entsprechend des ursprünglichen Durchlaufs vorzugeben, dass vier MPI-Prozesse im Setup vorhanden wären.

6 Andere Ansätze

Im Kontext dieser Projektarbeit wurden auch andere Ansätze für C&R von MPI untersucht, die einen maßgeblichen Einfluss auf die Entwicklung nahmen.

ReMPI

Erwähnenswert ist insbesondere das Projekt *ReMPI*, das dem *Lawrence Lawford National Laboratory* entstammt und ein vollständiges C&R-Framework für MPI darstellt. Als HPC-Plattform für dieses Tool wird auch der *IBM Blue Gene/Q* genannt.

Die Steuerung der MPI-Prozesse wird über Umgebungsvariablen vorgenommen, so auch das Laden von ReMPI über `LD_PRELOAD`. Eine Steuerung des Wrappers zur Programmlaufzeit ist allerdings nicht möglich. Viel mehr zielt der Anwendungszweck darauf ab, die gesamte verwendete MPI-Landschaft nachzustellen. So ist es weder möglich den Replay mit einer kleineren Anzahl von Prozessen zu starten als die Aufzeichnung, noch lässt sich lediglich ein gewählter MPI-Prozess isoliert für C&R betrachten.

Darüber hinaus verfügt es über strenge Plausibilitätskontrollen der MPI-Aufrufe, die ein Manipulieren der aufgezeichneten Daten nahezu unmöglich machen. Der Fokus von ReMPI liegt weiterhin im Umgang mit großen MPI-Clustern und Datenmengen, wie die Verfügbarkeit etlicher Kompressionsmechanismen und Effizienzuntersuchungen zeigt [SAL+15].

Seine Open Source-Natur ist dennoch auch für dieses Projekt ein großer Vorteil: Nicht nur befinden sich in seinem Umfeld weitere nützliche Werkzeuge wie das bereits erwähnte `wrap`, sondern es ermöglicht darüber hinaus ein Abschauen im Umgang mit Schwierigkeiten bei Aufzeichnung und Wiedergabe für die Behandlung noch fehlender MPI-Aufrufe, auch als Kreuzvalidierung der geleisteten Implementierung erweißt es sich hilfreich.

HDF5

Eine Erweiterung der Aufzeichnung findet sich in der Idee, MPI-Nutzdaten in HDF5-Containern abzuspeichern: Beide Standards verfügen über Grund-Datentypen, aus denen sich abgeleitete Compound-Datentypen bilden lassen. Ein Wrapper, der in der Lage ist, die Abbildung von MPI auf HDF5 und unmgekehrt vorzunehmen, eröffnet dem Benutzer die Möglichkeit, die ausgetauschten Daten in einem selbstbeschreibenden Format vorliegen zu haben und mit Werkzeugen der HDF5-Suite benutzen zu können.

Im Unterschied zu dem vorgestellten Wrapper senkt ein solches Konzept die Hemmschwelle für die Bearbeitung von Testdaten für Nutzer massiv, die sich in diesem Projekt sonst mit Binärserialisierung auseinandersetzen müssen [Kel17].

7 Ausblick

MPI-Wrapping zur Realisierung von C&R ist keine grundsätzliche Neuerscheinung auf dem Markt für MPI-Werkzeuge. Dennoch können bestimmte Anwendungsfälle – so etwa die prozessisolierte Wiedergabe von produktiven MPI-Durchläufen in Fortran – noch nicht zufriedenstellend abgedeckt werden.

In dieser Projektarbeit konnte gezeigt werden, dass ein MPI-Wrapper mit den gewünschten Anforderungen im Zusammenspiel mit FTG anhand der Klimaatmosphären-Simulation ICON verwendet werden kann. Aufgrund der Fülle des MPI-Standards ist zu diesem Zeitpunkt aber nur ein Bruchteil der erforderlichen Funktionalität für eine generische Verwendung implementiert.

Ein möglicher Ansatz zur Fortführung bestünde bottom-up darin, anhand eines Anwendungsfalls erforderliche MPI-Methoden oder Parameterserialisierung bei Bedarf nachzuimplementieren, oder aber top-down die Standard-Spezifikation zur Hand zu nehmen und das Projekt sukzessive zu einem Ende zu führen.

Gleichermaßen interessant wäre die Möglichkeit, das C&R-Verfahren zu modularisieren, um verschiedene Speicherverfahren realisieren zu können. Anstatt lediglich einen Binärdump zu schreiben und zu lesen, könnte etwa zusätzlich eine Anbindung an das bestehende HDF5-Projekt realisiert werden. Das Augenmerk sollte dabei auf dem Interface zwischen dem Wrapper mit seinen implementierten MPI-Aufrufen und dem Backend liegen, sodass dieses mit einem minimalen Spektrum von Methoden alle notwendigen Speicher- und Wiedergabeaufrufe bedienen kann.

Literaturverzeichnis

[For15] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.1, June 2015. `http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`.

[HK16] C. Hovy and J. Kunkel. Towards Automatic and Flexible Unit Test Generation for Legacy HPC Code. In *2016 Fourth International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE)*, pages 1–8, Nov 2016.

[Kel17] T. Kellyeh. Enabling Single Process Unit-Testing of MPI in Massive Parallel Applications. Masterarbeit in Bearbeitung, Universität Hamburg, 2017.

[SAL+15] Kento Sato, Dong H. Ahn, Ignacio Laguna, Gregory L. Lee, and Martin Schulz. Clock Delta Compression for Scalable Order-replay of Non-deterministic Parallel Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 62:1–62:12, New York, NY, USA, 2015. ACM.