MatlabPool

Klaus Happacher
13. August 2020

1 Einführung

1.1 Motivation

Während eines Praktikums, Werkstudententätigkeit und Bachelorarbeit bei Daimler AG musste ich häufig Simulationen in Matlab/Simulink durchführen. Dabei suchte ich nach effizienten Optimierungsmethoden um optimale Parameter für die komplexen Modelle zu bestimmen. Eine Schwierigkeit dabei, waren zeitintensive Simulationen und daraus folgend, lange Optimierungszeiten. Um diese Optimierungszeit zu verringern, habe ich zunächst die Laufzeit des Modells optimiert. Das beschleunigte die Parameteroptimierungen zwar sehr, aber sie dauerten immer noch mehrere Stunden.

Der Computer, welcher für diese Aufgaben zur Verfügung stand, besaß einen Prozessor mit acht Kernen beziehungsweise 16 Threads. Es war also naheliegend dieses Potential auszunutzen um die Optimierung noch weiter zu beschleunigen. Es konnten zum Beispiel mehrere Simulationen bzw. Funktionsauswertungen der Kostenfunktion parallel in einem Iterationsschritt ausgewertet werden. Das lässt sich mit der Parallel Computing Toolbox schnell umsetzen, jedoch stand mir diese Toolbox wegen hoher Lizensgebühren nicht zur Verfügung. Von der Matlab Community gibt es dafür Alternativen, welche mehrere Matlab Instanzen starten, die dann über das Dateisystem kommunizieren. Das funktioniert zwar sehr gut, ist aber nicht besonders schön.

Matlab bietet seit der Version R2018a eine neue C++ API an, mit der man C++ Funktionen schreiben kann, welche wie normale Matlab Funktionen aufgerufen werden können. Außerdem ist es möglich C++ Programme zu schreiben, in denen Matlab Funktionen und Code ausgeführt wird. Mit diesen zwei Eigenschaften kann ein Matlab Pool entwickelt werden, welcher das Ziel dieser Arbeit ist und im folgenden als MatlabPool bezeichnet wird.

1.2 Ziel

Ziel dieser Hausarbeit ist es, einen Matlab Pool zu entwickeln. Dieser Pool soll beliebig viele Matlab Instanzen verwalten und diesen Instanzen einzelne Jobs zuweisen können. Ein Job beschreibt dabei einen Aufruf einer beliebigen Funktion in Matlab mit beliebig vielen Input sowie Output Argumenten. Des weiteren sollen Fehlermeldungen und Konsolenausgaben der einzelnen Jobs dem Benutzer angezeigt werden.

Damit die einzelnen Worker bzw. Matlab Instanzen des Pools alle möglichen Jobs bearbeiten können, ist es in manchen Fällen notwendig, gewisse Umgebungsvariablen zu setzen. Zum Beispiel wenn Funktionen oder Ressourcen in den Jobs benötigt werden, die sich nicht in dem stan-

dard Matlab Suchpfad befinden. Dafür soll es die Möglichkeit geben, Befehle auf allen Worker des Pools ausführen zu können.

Die Verwendung des MatlabPools soll über Matlab oder ein C++ Programm möglich sein. Beides soll möglichst plattformunabhängig funktionieren, also sowohl unter Linux als auch unter Windows (natürlich auch Mac, aber das kann ich aufgrund fehlender Ressourcen nicht testen)

1.3 Maltab C++ API

Matlab bietet drei APIs mit denen eine Schnittstelle zwischen C++ und Matlab geschaffen wird.

Matlab Data API

Im der Matlab Data API sind die Matlab-Datentypen als C++ Klassen definiert[1]. Eine Basis bildet die Klasse *matlab::data::Array* mit der jeder Matlab-Datentyp gehandhabt werden kann. Falls man jedoch zum Beispiel auf die Werte einer Matrix vom Type double zugreifen will, muss das *matlab::data::Array* Objekt in die abgeleitete Template Klasse *matlab::data:TypedArray<double>* gecastet werden. Für Matlab Structs, Strings, Cell-Arrays, ... sind im Namespace *matlab::data* noch weitere Klassen definiert, die auch von *matlab::data::Array* abgeleitet sind.

Interessant ist hier, wie Matlab mit dem allokierten Speicher der Datentypen umgeht. In der Dokumentation wird die dafür verwendete Semantik mit *copy-on-write* bezeichnet[2]. Diese Technik lässt sich gut mit dem Smartpointer *std::shared_ptr* beschreiben. Man kann sich die Klasse *matlab::data::Array* als ein *std::shared_ptr* vorstellen, wenn ein Objekt dieser Klasse kopiert wird, wird ein Referenzzähler inkrementiert, und beim Abbau dekrementiert und ggf. der Speicher freigegeben. Der Unterschied bei *matlab::data::Array* ist, falls eine nicht *const* Funktion aufgerufen wird und der Referenzzähler größer als Eins ist, dann wird das Objekt mit den verwalteten Daten kopiert. Um unnötiges Kopieren der Matlab-Datentypen in C++ zu vermeiden, muss also darauf geachtet werden, nur *const* Funktionen aufzurufen, oder den Referenzzähler auf Eins zu halten falls nicht *const* Funktionen aufgerufen werden sollen. *matlab::data::Array* und die davon abgeleiteten Klassen unterstützen *std::move*, damit können Objekte dieser Klassen verschoben werden, ohne den Referenzzähler zu erhöhen.

Matlab Engine API for C++

Die Matlab Engine API bietet Funktionalitäten um eine oder mehrere Matlab Instanzen in einem C++ Programm zu steuern[3]. Grundsätzlich bietet die API Funktionen um eine Matlab Instanz zu starten oder sich mit einer bestehenden zu verbinden. Besteht eine Verbindung zu einer Matlab Instanz, können Variablen in oder aus dem Workspace geladen werden, Funktionen mit Parametern aufgerufen und Matlab Code als String ausgeführt werden. Diese genannten Funktionalitäten können entweder synchron oder asynchron ausgeführt werden.

[1] MathWorks, „Matlab Data API," 2018. Adresse: `https://de.mathworks.com/help/matlab/matlab-data-array.html`

[2] MathWorks, „Copy C++ MATLAB Data Arrays," 2018. Adresse: `https://ch.mathworks.com/help/matlab/matlab_external/copy-cpp-api-matlab-arrays.html`

[3] MathWorks, „Matlab Engine API for C++," 2018. Adresse: `https://de.mathworks.com/help/matlab/calling-matlab-engine-from-cpp-programs.html`

C++ MEX API

Mit der C++ MEX API können C++ Funktionen geschrieben werden, welche dann in Matlab aufgerufen werden können[4]. Dafür muss in dem C++ Code eine Klasse mit dem Namen *MexFunction* Implementiert werden, die von *matlab::mex::Function* abgeleitet ist. Weiter muss diese Klasse den virtuellen Funktionsoperator mit zwei Argumenten für den Input und Output überschreiben. Diese Funktion ist der Einstiegspunkt für den Aufruf aus Matlab. Der C++ Code muss dann zu einer Shared Library mit der Endung *.mexw64* (Windows), *.mexa64* (Linux) oder *.mexmaci64* (Mac) kompiliert werden. Solche Funktionen werden von Matlab häufig als Mex-Functions bezeichnet. Eine Minimalbeispiel könnte wie folgt aussehen.

```
#include "mex.hpp"
#include "mexAdapter.hpp"

class MexFunction : public matlab::mex::Function {
public:
    void operator()(matlab::mex::ArgumentList outputs, matlab::mex::ArgumentList inputs)
    {
        outputs[0] = std::move(inputs[0]);
    }
};
```

Hier wird lediglich das erste Argument der Funktion zurück gegeben.

Außerdem ist es möglich einen default Konstructor und Destruktor zu definieren. Diese werden dann beim Laden bzw. Schließen der Shared Library ausgeführt. Weiter ist es mit der Memberfunktion *getEngine* der Basisklasse möglich, einen Zeiger auf die Matlab Instanz, welche die Mex-Funktion aufruft, zu erhalten. Damit kann dann wie bei der Matlab Engine API Funktionen, und Matlab Code auf der Matlab Instanz ausgeführt werden.

Das kann zum Beispiel genutzt werden, wenn ein Fehler in der C++ Funktion auftritt bzw. erkannt wird. In diesem Fall kann dann die Matlab Funktion *error* aufgerufen werden, um eine passende Fehlermeldung auszugeben und die Ausführung der Mex Funktion zu unterbrechen.

2 Probleme

Sobald man sich mit der Matlab C++ API auseinander gesetzt hat, wird schnell klar, dass ein MatlabPool basierend auf dieser API sich nicht problemlos umsetzen lässt. Die ausschlaggebendsten Probleme sind die Inkompatibilität zwischen der Matlab Engine API und der C++ MEX API sowie die Synchronisierung von asynchronen Funktionen aus der Matlab Engine API.

2.1 Inkompatibilität Matlab Engine API und C++ MEX API

Includiert man die Header Dateien *mex.hpp* und *MatlabEngine.hpp* der beiden APIs gleichzeitig, dann wird man beim Kompilieren mit nahezu endlos vielen Fehlermeldungen überschüttet. Ein Grund dafür ist, dass in *mex.hpp* ein Teil der Matlab Engine API implementiert ist und dadurch mehrere Klassen und Funktionen doppelt definiert sind. Die *mex.hpp* enthält jedoch nicht die

[4] MathWorks, „Matlab MEX API," 2018. Adresse: `https://de.mathworks.com/help/matlab/cpp-mex-file-applications.html`

gesamte Funktionalität der Matlab Engine API, zum Beispiel fehlt die Funktion eine Matlab Instanz zu starten. Für den MatlabPool werden deshalb beide Header Dateien benötigt.

Um beide APIs nutzen zu können, werden deshalb im Folgenden alle Klassen und Funktionen, welche die Matlab Engine API benötigen, in eine Shared Library ausgelagert.

2.2 Synchronisierung von asynchrone Funktionen

Mithilfe der Matlab Engine API können Funktionen bzw. die eingehenden Jobs des MatlabPools asynchron gestartet werden. Die Synchronisierung erfolgt dabei mit einem *matlab::engine::FutureResult* Objekt. Diese Klasse ist von *std::future* abgeleitet und bietet eine weitere Funktion *cancel* um Jobs unterbrechen zu können.

Nun kann man sich Fragen, wie ein Worker aus dem MatlabPool nach Beendigung eines Jobs mitteilt, dass er bereit für weitere Jobs ist. Eine Möglichkeit wäre, für jede Matlab Instanz einen Thread zu erstellen, der Jobs an seine Matlab Instanz übergibt und konkurrierend auf eine Job-Queue zugreift. Diese Lösung wird hier jedoch nicht umgesetzt wegen den zusätzlichen Threads für jede Matlab Instanz und weil dadurch das Abbrechen von Jobs sehr umständlich wird.

Eine weitere Möglichkeit ist, die Matlab Engine API zu erweitern, um die fehlende Funktionalität zu erhalten. Zum Beispiel kann mit der Memberfunktion *fevalAsync* aus der *matlab::engine::MATLABEngine* Klasse eine Funktion in einer Matlab Instanz asynchron gestartet werden. Dabei ist *fevalAsync* ein Wrapper für die Funktion *cpp_engine_feval_with_completion*, welcher insgesamt 13 Parameter übergeben werden. Dazu gehören unter anderem zwei Funktions-Pointer für Funktionen die Daten oder eine Exception im *std::promise* Objekt setzen. In der Matlab API ist das wie folgt umgesetzt.

```
// engine_execution_interface_impl.hpp
FutureResult<std::vector<matlab::data::Array>> fevalAsync(
    const std::u16string &function,
    const size_t nlhs,
    const std::vector<matlab::data::Array> &args)
{
    // ...
    uintptr_t cpp_engine_feval_with_completion(...,
        &set_feval_promise_data,
        &set_feval_promise_exception, ...)
    // ...
}

void set_feval_promise_data(void *p, size_t nlhs, bool straight,
                            matlab::data::impl::ArrayImpl** plhs)
{
    // ...
    using Promise = std::promise<std::vector<matlab::data::Array>>;
    Promise* prom = reinterpret_cast<Promise*>(p);
    // ...
}
void set_feval_promise_exception(void *p, size_t nlhs, bool straight,
                                 size_t excTypeNumber, const void* msg);
```

Es fällt auf, dass das *std::promise* Objekt als *void** übergeben wird. Das kann ausgenutzt werden, in dem man *set_feval_promise_data* und *set_feval_promise_exception* umgeht und mit dem *void** Argument zusätzliche Informationen übergibt, wie zum Beispiel hier.

```
struct MatlabPromiseHack
{
    std::promise<std::vector<matlab::data::Array>> *prom;
    std::function<void()> notifier;
};
void set_feval_promise_data_hack(void *p, size_t nlhs, bool straight,
                                 matlab::data::impl::ArrayImpl** plhs)
{
    MatlabPromiseHack *p_hack = reinterpret_cast<MatlabPromiseHack *>(p);
    matlab::execution::set_feval_promise_data(p_hack->prom, nlhs, straight, plhs);
    p_hack->notifier();
    delete p_hack;
}
```

Die *std::promise* Variable steht bewusst an erster Stelle im dem *MatlabPromiseHack* Struct, damit ein Pointer auf ein *MatlabPromiseHack* Objekt mit *reinterpret_cast* zu einem *std::promise* Objekt gecastet werden kann.

3 Umsetzung

3.1 Jobs

Die Schnittstelle zwischen MATLAB und C++ unterstützt nur Datentypen aus der Matlab Data API. Durch diese Einschränkung lassen sich nur primitive Datentypen, bzw. Arrays und Structs aus primitiven Datentypen übertragen. Deshalb macht es Sinn, einen Job für den MatlabPool in C++ zu verwalten. Mit einem Index kann dann ein Zugriff auf den Job von der Matlab Instanz ermöglicht werden. Abbildung 1 zeigt einen groben Entwurf der Job Klassen, mit den wichtigsten Inhalten (getter und setter Methoden sind z.B. nicht enthalten).

In Abbildung 1 fällt auf, dass für die Jobs eine *StreamBuf* Klasse für Fehlermeldungen und Konsolenausgaben verwendet wird. Im Gegensatz zur *std::ostringstream* Klasse aus der Standard Bibliothek wird in *StreamBuf* der Datentyp *char16_t* statt *char* verwendet, und eine zusätzliche Methode *get* liefert einen Shared Pointer auf den Buffer. Diese beiden Unterschiede sind sehr hilfreich bei der Verwendung von Funktionen aus Matlab Engine API.

Die Namen der Job Klassen sind an die Bezeichnungen der Matlab Engine API Funktionen angelehnt. Zum Beispiel ist in der API die Funktion *eval* bzw. *evalAsync* definiert, um Matlab Code auszuführen, oder *feval* bzw. *fevalAsync* um Matlab Funktionen aufzurufen. Bei *feval* können beliebig viele Argumente (*args*) übergeben werden und die Funktion kann beliebig viele Werte zurück geben (*result*).

In der Basisklasse *JobBase* wird mit einer statischen Variable dafür gesorgt, dass alle Jobs durch einen Index *id* eindeutig sind. Damit das während der Laufzeit so bleibt, ist das Kopieren der Job Klassen untersagt. Mit der Funktion *toStruct* kann der Inhalt eines Jobs in ein Matlab Struct kopiert oder ggf. verschoben werden. Dieses Struct erhält der Benutzer in Matlab nach Beendigung seines Jobs.

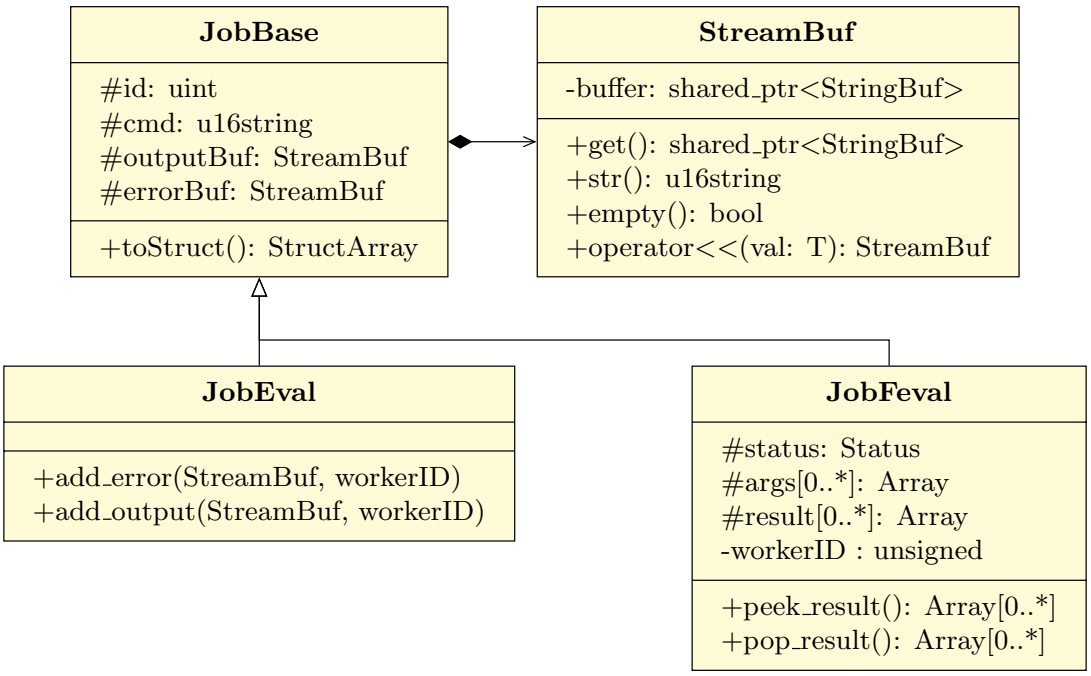

Abbildung 1: Job Klassen (*u16string*, *shared_ptr* sind aus *std* und *Array*, *StructArray* aus *matlab::data*)

3.2 Shared Library

Wie bereits erwähnt, müssen einige Teile der Implementierung in eine Shared Library ausgelagert werden, um doppelte Definitionen und somit Fehler beim Linken des Programms zu vermeiden. Abbildung 2 zeigt grob den Inhalt dieser Shared Library.

Durch dynamische Polymorphie bzw. virtuelle Memberfunktion in der *Pool* Klasse wird hier das Laden der Shared Library vereinfacht. Die *PoolImpl* Klasse beinhaltet einen Vektor von *EngineHack* Objekten (Worker), einen Master-Thread für die Verteilung der Jobs, sowie eine Queue und Map für wartende und abgeschlossene Jobs. Nachdem ein Pool erstellt wurde, können neue Worker hinzugefügt oder Bestehende entfernt werden mit der Funktion *resize*. Der Grund dafür ist, dass die Matlab Instanzen in dem Pool nicht unnötig neu gestartet werden müssen. Das folgende Beispiel verdeutlicht warum das sinnvoll ist.

---

clear
close all

MatlabPool.resize(4);

result = *MatlabPool.eval*("Hello␣World");

Beim ersten Aufruf dieses Skriptes wird die Mex-Funktion des MatlabPools geladen (falls diese zuvor noch nicht benutzt wurde) und ein Pool mit vier Worker erstellt. Führt man dieses Skript im Anschluss ein weiteres mal aus, existiert der Pool bereits mit vier Worker, wodurch die *resize* Funktion nahezu keine Laufzeit kostet. Hierbei sei noch angemerkt, dass die Matlab-Funktion *clear* zu Beginn des Skriptes nur alle Variablen aus dem Workspace entfernt und freigibt, jedoch

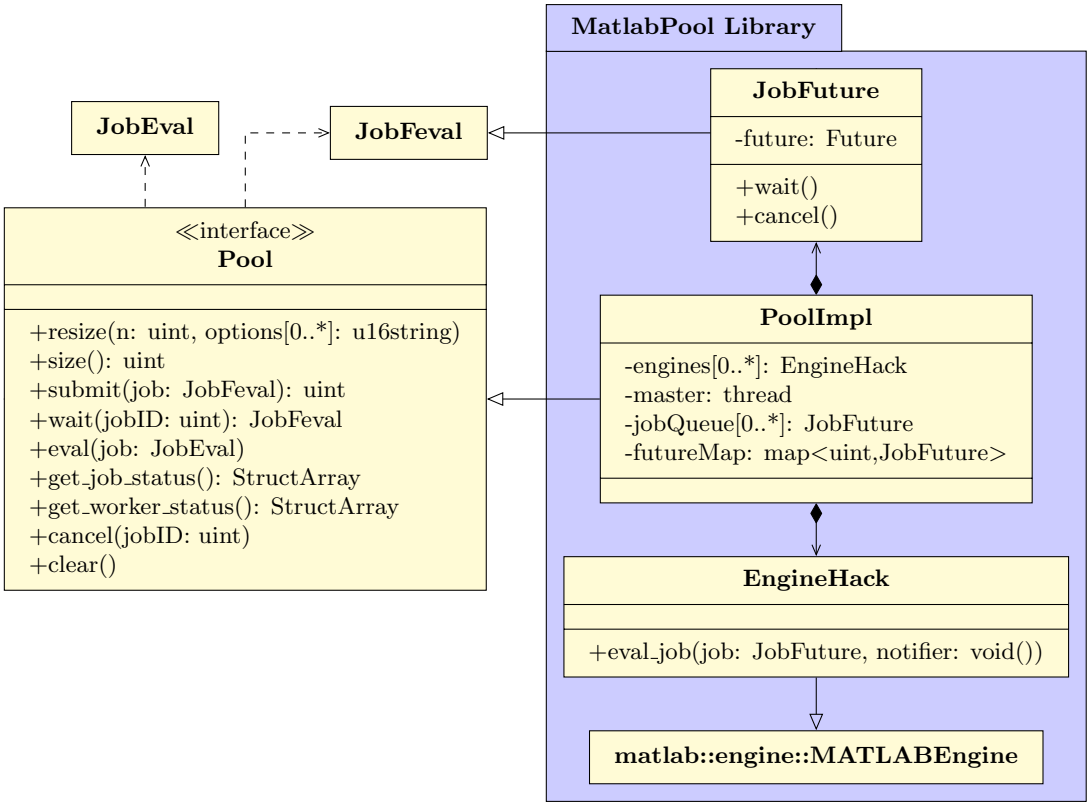

Abbildung 2: MatlabPool Library (*u16string*, *map*, *thread* sind aus *std* und *Future*, *StructArray* aus *matlab::data*)

nicht Mex-Funktionen, diese müssen explizit freigegeben werden[5].

Die *Pool* Klasse beschreibt weitere Funktionen wie z.B. *submit* und *wait* womit einzelne Jobs eingereicht und auf deren Fertigstellung gewartet werden kann. Mit *eval* kann Matlab-Code auf allen Worker ausgeführt werden um z.B. den aktuellen Pfad der Worker/Matlab-Instanzen anzupassen. Zum Abbrechen von Jobs kann die Funktion *cancel* (einzelner Job) oder *clear* (alle Jobs) verwendet werden. Dabei kann ein Job jederzeit abgebrochen werden, es spielt also keine Rolle ob sich der Job noch in der Job-Queue befindet, gerade einem Worker zugewiesen wird, bearbeitet wird, oder bereits abgeschlossen ist.

In der *EngineHack* Klasse sind die Ideen aus dem Abschnitt 2.2 umgesetzt, damit die Worker die Beendigung eines Jobs den Master-Thread mitteilen. Die *EngineHack* Klasse ist von der Matlab Klasse *matlab::engine::MATLABEngine* abgleitet. Diese Matlab Klasse unterstützt weder Kopieren noch Verschieben, weshalb Objekte dieser Klasse bzw. der *EngineHack* Klasse bei der Benutzung in einen *std::unique_ptr* verpackt werden, um das Verschieben zu ermöglichen.

Die Job Klasse *JobFuture* befindet sich ebenfalls in der MatlabPool Library, da sie ein Future-Objekt aus dem *matlab::engine* Namespace enthält, um das Synchronisieren sowie Abbrechen der Jobs zu ermöglichen.

[5]MathWorks, „Matlab Clear Funktion," 2018. Adresse: `https://de.mathworks.com/help/matlab/ref/clear.html`

3.3 Zuladen der Library

Das Zuladen der Library wird über eine weitere RAII Klasse gehandhabt, siehe Abbildung 3. Hierfür wird eine Klasse Implementiert, die allgemein Shared Libraries sowie deren Funktionen laden kann, unabhängig von dem verwendetem Betriebssystem. Eine davon abgeleitete Singleton Klasse übernimmt dann das Laden der MatlabPool Library. Durch die Eigenschaften von Singleton Klassen wird gewährleistet, dass die Bibliothek nur einmal geladen wird.

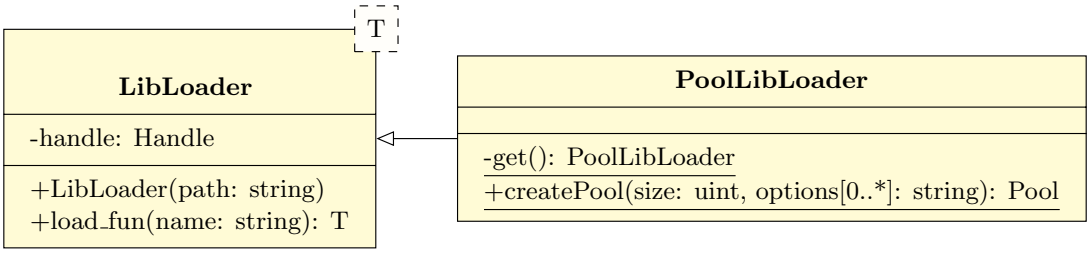

Abbildung 3: Laden und Entladen der MatlabPool Library

3.4 Mex-Funktion

Die Umsetzung der Mex-Funktion bzw. der Schnittstelle zwischen Matlab und dem MatlabPool ist relativ einfach, weil dort Jobs sowie Ergebnisse lediglich durch gereicht werden und Exception ggf. als Fehlermeldungen in Matlab angezeigt werden. Alle Exceptions die speziell für dieses Projekt implementiert wurden, sind von einer Exception Klasse mit einer zusätzlichen rein virtuellen Funktion abgeleitet. Diese zusätzliche Funktion soll einen Identifier (String) zurückgeben, mit welchem die jeweilige Exception eindeutig beschrieben ist. Das ist zwar nicht zwingend notwendig, aber dadurch lassen sich lesbare Matlab Fehlermeldungen erzeugen. Außerdem können die Fehlermeldungen dann ähnlich wie in C++ mit mehreren *catch* Blöcken gefiltert beziehungsweise getrennt voneinander behandelt werden.

4 Testen

Zum Testen der einzelnen Funktionen des MatlabPools wurden Tests in C++ mit einer Testsuite implementiert sowie Tests in Matlab. Die Tests in Matlab sind sehr ähnlich zu den C++ Tests und wurden mit einer von Matlab vordefinierten Testsuite umgesetzt. Die Testsuite für die C++ Tests ist dagegen selbst implementiert. Abbildung 4 zeigt den groben Entwurf der *TestSuit* Klasse.

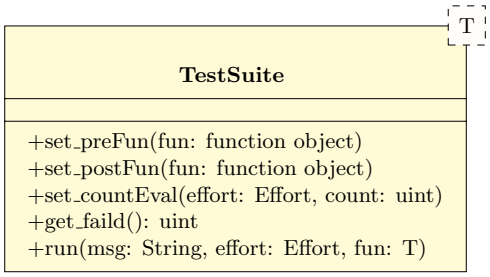

Abbildung 4: TestSuite Klasse

Mit *run* können einzelne Tests ausgeführt werden. Ein Test kann dabei mit einem Text beschrieben werden und mit der Enumeration Klasse *Effort* kann die Laufzeit des Tests ungefähr angegeben werden. Mit *set_countEval* wird den Definitionen aus der Enumeration Klasse *Effort* Werte zugewiesen. Zum Beispiel wird mit *obj.set_countEval*(*Effort:Small,100*) angegeben, dass alle Tests die mit *Effort::Small* definiert sind, 100 mal durchgeführt werden sollen. Die Idee dahinter ist, dass in manchen Funktionen Fehler enthalten sein können, die zu einem undefinierten Verhalten führen. Während der Debug-Phase dieser Arbeit ist zum Beispiel ein Test für das Abbrechen von Jobs erst nach mehreren Aufrufen fehlgeschlagen bzw. gelangte in einen Deadlock. Der Fehler trat genau dann auf, wenn ein Job während der Zuweisung zu einen Worker abgebrochen werden sollte. Durch das mehrfache ausführen des Tests steigt die Wahrscheinlichkeit, dass dieser Spezialfall auftritt und somit getestet wird. Es gibt jedoch sehr zeitintensive Tests in denen z.B. neue Worker gestartet werden, diese Tests sollten dann natürlich nicht so häufig Wiederholt werden.

Außerdem können mit *set_preFun* und *set_postFun* Funktionen definiert werden, die vor oder nach den Tests aufgerufen werden sollen. Damit wird hier nach jedem Test überprüft, ob der Pool noch Jobs enthält die eventuell in dem Test vergessen wurden oder auf einen Fehler im MatlabPool zurückzuführen sind.

4.1 Memory Leaks

Das Überprüfen der Implementierung auf Memory-Leaks wurde mit dem Tool *valgrind* unter Ubuntu 20.04 durchgeführt. Dabei schien es, dass die Matlab Engine API selbst mehrere Memory Leaks aufweist. Um diese Vermutung zu bestätigen, kann folgendes Minimalbeispiel betrachtet werden.

```
#include "MatlabEngine.hpp"

int main()
{
    using namespace matlab::engine;
    std::unique_ptr<MATLABEngine> engine = startMATLAB({u"−nojvm"});
}
```

Hier wird die Matlab Engine API verwendet um eine Matlab Instanz zu starten, diese Vorgehensweise ist in der Matlab Dokumentation zu finden[6].

Ein Test mit *valgrind* liefert für das Minimalbeispiel folgends Ergebnis.

```
==62694== LEAK SUMMARY:
==62694== definitely lost: 936 bytes in 3 blocks
==62694== indirectly lost: 0 bytes in 0 blocks
==62694== possibly lost: 1,854 bytes in 18 blocks
==62694== still reachable: 812,144 bytes in 2,072 blocks
```

Aus diesem Grund werden mit dem *valgrid* Flag *suppressions* alle Matlab Libraries ignoriert. Die erneute Durchführung des Tests liefert dann das Ergebnis

```
==62409== LEAK SUMMARY:
==62409== definitely lost: 0 bytes in 0 blocks
```

[6]MathWorks, „Start MATLAB Sessions from C++,“ 2018. Adresse: `https://de.mathworks.com/help/matlab/matlab_external/start-and-connect-to-a-matlab-sessions-from-c.html`

==62409== *indirectly lost*: 0 *bytes in* 0 *blocks*
==62409== *possibly lost*: 448 *bytes in* 1 *blocks*
==62409== *still reachable*: 0 *bytes in* 0 *blocks*

Bei der Überprüfung der Tests bzw. des MatlabPools auf Memory-Leaks konnte mit dem *suppressions* Flag wie zuvor das gleiche Ergebnis erzielt werden. Daraus lässt sich schließen, dass bei den Tests keine Memory-Leaks durch die Implementierung des MatlabPools aufkamen. Das heißt jedoch nicht, dass die Implementierung frei von möglichen Memory-Leaks ist. Vor allem kann die Klasse *EngineHack* diesbezüglich Probleme bereiten, weil dort an mehreren Stellen Speicher mit einem reinen *new* allokiert wird, der dann teilweise erst nach Beendigung eines Jobs wieder freigegeben wird. Die Funktionen in der *EngineHack* Klasse stammten jedoch aus der Matlab Engine API und wurden teilweise nur leicht angepasst. Weil Matlab in seiner API diese möglichen Memory-Leaks zulässt, wird dieses Problem in der *EngineHack* Klasse übernommen, um die Abweichung zur Implementierung in der Matlab Engine API gering zu halten.

5 Anwendungen

5.1 Latenzzeit

Eine kurze Latenzzeit für das Bearbeiten von Jobs ist eine wichtige Voraussetzung für einen effizienten Threadpool bzw. hier MatlabPool. Diese Zeit kann mit einem Skript gemessen werden, das Jobs ohne Bearbeitungsaufwand einreicht und auf deren Fertigstellung wartet. Die durchschnittliche Zeit für das Einreichen und Zurückgeben der Jobs im MatlabPool beträgt circa 1-2 Millisekunde. Dieses Ergebnis ist nicht wirklich berauschend, durch ignorieren der Konsolenausgabe der Worker kann eventuell eine kleine Verbesserung erzieht werden, dies wurde hier jedoch nicht getestet.

5.2 Julia-Menge

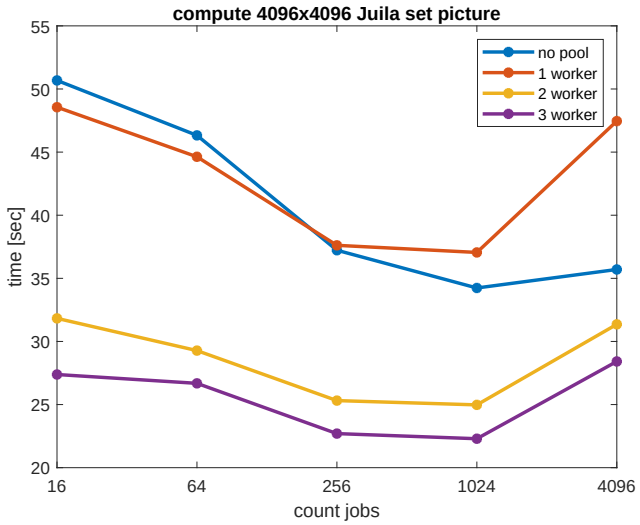

Abbildung 5: Vergleiche MatlabPools mit verschiedenen Einstellungen

Mit einer Julia Menge kann die Performance des MatlabPools gut veranschaulicht werden. Dafür wird ein Bild von einer Juila Menge mit der Auflösung 4096×4096 und verschiedenen Job-Größen berechnet. In einem Job wird immer ein Teil des Bildes berechnet. Abbildung 5 vergleicht die wall-time für die Berechnung der Bilder mit unterschiedlicher Anzahl an Worker bzw. ohne dem MatlabPool.

Werden die Jobs kleiner gewählt, folgt daraus automatisch, dass insgesamt mehrere Jobs benötigt werden, wodurch die Latenzzeit eine größere Rolle spielt. Bei größeren, also weniger Jobs, steigt die wall-time, weil die verwendete Funktion zur Berechnung der Pixel dafür nicht ausgelegt ist. In der Abbildung 5 ist außerdem zu erkennen, dass der Unterschied zwischen zwei und drei Worker nicht mehr besonders groß ist, der Grund darin liegt wahrscheinlich an dem verwendeten Computer, da dieser nur einen Prozessor mit zwei Kernen bzw. vier Threads besitzt.

6 Ausblick

Der MatlabPool lässt sich an einigen Stellen erweitern oder anpassen. Mögliche Erweiterungen wären Funktionen mit denen Variablen in den Workspace der Worker gespeichert und geladen werden können. Damit könnte man zum Beispiel Messdaten in die Worker laden, die für die Bearbeitung der Jobs notwendig sind. Des Weiteren kann eine zusätzliche Job Klasse implementiert werden, die der *JobFeval* Klasse ähnelt jedoch Konsolenausgaben sowie Fehlernachrichten der Worker ignoriert und nur das Ergebnis des Jobs speichert. Ziel dieser Erweiterung wäre eine Verringerung der Latenzzeit.

Der MatlabPool kann auch angepasst werden, indem Jobs zum Beispiel eine Priorität zugewiesen werden könnte, oder Funktionen bzw. ein Skript das vor/nach der Bearbeitung eines Jobs ausgeführt werde soll, um die Umgebung entsprechen anzupassen. Dafür müsste wahrscheinlich der bestehende Code an manchen Stellen angepasst werden.