UNIKASSEL
VERSITÄT

Universität Kassel

Bachelorarbeit

Vergleich der parallelen Programmierumgebungen Cilk-F und OpenMP bezüglich Benutzbarkeit und Effizienz

Fachbereich 16 Elektrotechnik/Informatik
Studiengang Bachelor Informatik

Vitali Wascher
35109035

Kassel, 6. Juli 2021

Betreuerin: Frau Prof. Dr. Claudia Fohry
Erstprüferin: Frau Prof. Dr. Claudia Fohry
Zweitprüferin: Frau Prof. Dr. Claude Draude

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet. Hiermit versichere ich, dass diese Version inhaltlich mit dem vorab elektronisch übersandten Exemplar übereinstimmt.

_____

Kassel, 6. Juli 2021

Vitali Wascher

Inhaltsverzeichnis

1 Einführung

Cilk-F ist eine Erweiterung von Cilk, welche Cilk um die Benutzung von Futures und ein besser an Futures angepasstes Workstealing erweitert. Cilk-F wurde, wie auch Cilk, am Massachusetts Institute of Technology (MIT) entwickelt.
Das Ziel von Cilk ist das parallele Programmieren auf Systemen mit gemeinsamen Speicher zu erleichtern. Mit wenigen Konstrukten soll Parallelität möglich werden, so beschränkt sich Cilk auf nur drei Befehle, die ein Programmierer braucht, um parallel zu programmieren. Damit soll die Benutzbarkeit von Cilk sehr einfach und schlicht gehalten werden.
Das Besondere an Cilk-F ist, dass zum einen Futures verwendet werden und zum anderen Proactive Workstealing eingesetzt wird, im Gegensatz zu Cilk. Folglich hat Cilk-F, nach Aussagen der Entwickler, eine ebenso schnelle Laufzeit in den mitgelieferten Benchmarks, wie Cilk. Des Weiteren sollen durch die Nutzung von Futures flexiblere parallele Algorithmen möglich sein.
OpenMP entstand als Spracherweiterung für C/C++ und hat in der heutigen Zeit große Bedeutung, wenn es um parallele Programmierung in Systemen mit gemeinsamen Speicher geht. Die Verbreitung von OpenMP beim Programmieren kommt im Übrigen daher, dass vorhandener sequentieller Code auf einfache Weise parallelisiert werden kann, damit eine einfache Benutzbarkeit von OpenMP gewährleistet ist. Die Bestandteile von OpenMP sind Compiler Direktiven, Bibliotheksfunktionen und Umgebungsvariablen für die Programmierung. Eine Unterstützung für Tasks ist mit den neueren Versionen (ab Version 3.0 2008) verfügbar.
Da Cilk-F sich, beim Anfertigen der Bachelorarbeit, im Prototypen Status befindet, sind grundlegende Fragen, welche parallelen Programme sich mit Cilk-F realisieren lassen, wie hoch der Aufwand bei der Einarbeitung in die Programmierumgebung Cilk-F ist.

In dieser Arbeit wird die Benutzbarkeit und Effizienz von Cilk-F und OpenMP verglichen. Die Benutzbarkeit einer Programmiersprache kann hierbei als eigene Erfahrungen, die gemacht wurden während der Implementierung der Benchmarks, angesehen werden und wie natürlich es ist, aus der Nutzersicht eines Programmierers, einen Algorithmus mit den Konzepten der Futures in Cilk-F und Threads/Tasks in OpenMP zu schreiben.
Die zu implementierenden Benchmarks konnten mit beiden Programmierumgebungen realisiert werden, mit Cilk-F war es allerdings aufwendiger einen Algorithmus zu programmieren im Gegensatz zu OpenMP. Es konnte festgestellt werden, dass die Effizienz der Benchmarks mit OpenMP und Cilk-F eine nahezu gleiche Laufzeit haben.
Der Unterschied, der bei der Programmierung zwischen den beiden parallelen Programmierumgebungen entsteht, soll festgestellt werden.
Die Bachelorarbeit gliedert sich folgendermaßen: in Kapitel 2 werden die Sprachkonzepte von OpenMP, Cilk und Cilk-F, mit einer etwas ausführlicheren Sicht auf Cilk-F und dessen Konstrukte, beschrieben. Zudem liegt der Schwerpunkt in Kapitel 2 auf einem allgemeinen Verständnis von grundlegenden Konzepten in Cilk, um die Programmierung mit Cilk-F besser verstehen zu können. In Kapitel 3 werden die programmierten Benchmarks beschrieben. Die Benchmarks wurden mit OpenMP und Cilk-F implementiert.
Kapitel 4 widmet sich der Benutzbarkeit zwischen den beiden Systemen mit Vorteilen sowie Nachteilen, die bei der Programmierung auftreten.
In Kapitel 5 wird die Performance von Cilk-F und OpenMP anhand der Benchmarks verglichen.
Das letzte Kapitel, Kapitel 6, fasst nochmals die Schwerpunkte zusammen und beinhaltet ein Fazit.

2 Programmierumgebungen

Dieses Kapitel gibt einen Überblick über OpenMP, Cilk und Cilk-F. Im Abschnitt 2.1 werden die verwendeten Sprachkonzepte von OpenMP erklärt. In Abschnitt 2.2 werden grundlegende Konzepte von Cilk zusammengefasst. Der Abschnitt 2.3 verschafft einen Überblick über Cilk-F und die verwendeten Konzepte. Durch die Unterkapitel werden die verwendeten Sprachkonstrukte von Cilk-F genauer erklärt, um eine Basis zu schaffen, die bei der Programmierung erforderlich ist. Des Weiteren wird eine Problematik durch Abschnitt 2.3.1 beschrieben, welches bei der Programmierung mit Cilk-F entstanden ist und wie es gelöst werden kann.

2.1 OpenMP

OpenMP (Open Specifications for Multi Processing) ist eine Programmierschnittstelle, die für die Programmierung von Rechnern mit gemeinsamem Speicher entwickelt wurde. Die Komponenten der Schnittstelle sind Compiler Direktiven, Bibliotheksfunktionen und Umgebungsvariablen. OpenMP entstand in Zusammenarbeit mit diversen Unternehmen im Jahre 1997 als Schnittstelle für C/C++ und FORTRAN.

Eine OpenMP Anweisung findet im Quellcode als Direktive statt, die in der Programmiersprache C/C++ durch das `#pragma` Schlüsselwort definiert ist. Das hat den Vorteil, dass ein Compiler ohne OpenMP Unterstützung dennoch ein gültiges, sequentielles Programm produziert, da die `#pragma` Direktive unter diesen Voraussetzungen als Kommentar gesehen wird und ein paralleler Bereich nur durch einen Thread ausgeführt wird.

Das Konzept von OpenMP basiert auf einem fork/join Modell. Die sequentielle Ausführung verzweigt (fork) an bestimmten Stellen auf mehrere Threads, dies kann als paralleler Abschnitt gesehen werden. Nachdem die Threads ihre Berechnungen abgeschlossen haben, werden sie zusammengeführt (join) und

nehmen die sequentielle Ausführung wieder auf. Ein sequentielles Programm kann schrittweise parallelisiert werden, indem parallele Abschnitte in inkrementeller Weise zum sequentiellen Code hinzugefügt werden, dies nennt man inkrementelle Parallelisierung.
Ein Ziel bei der Entwicklung von OpenMP ist es, benutzerfreundlich zu sein, indem OpenMP eine umfassende Literatur und Beispiele bietet, mit denen schnell Resultate in der Programmierung hervorgebracht werden sollen.

Die Beschreibung der Hauptbestandteile, die in den Benchmarks mit OpenMP verwendet wurden, erfolgt hier. Durch die Compiler Direktive `#pragma omp parallel num_threads(8)` wird ein paralleler Bereich mit Anzahl an Threads erzeugt, der von den erzeugten Threads ausgeführt wird. Die Anzahl der zu erzeugenden Threads wird als Parameter mitgegeben, in dieser Direktive wären es acht Threads. Innerhalb eines parallelen Bereichs `#pragma omp parallel` kann ein `#pragma omp single` Aufruf erfolgen, dieser Single Bereich wird nur von einem Thread ausgeführt.
In der Methode `omp_get_thread_num()` wird die Threadnummer des aufzurufenden Threads zurückgegeben. Eine `#pragma omp atomic` Direktive wird ausgeführt, wenn mehrere Tasks gleichzeitig z.B. eine Variable inkrementieren wollen, somit darf nur ein Task die Variable manipulieren.
Die Compiler Direktive `#pragma omp critical` definiert einen kritischen Abschnitt und wird verwendet, wenn mehrere Threads z.B. eine Variable inkrementieren wollen. In einem kritischen Bereich darf sich nur ein Thread aufhalten.
Seit der Version 3.0 (2008) unterstützt OpenMP Task-Direktiven und ab Version 4.0 (2013) wurde das Task-Konstrukt erweitert, um Abhängigkeiten zwischen Tasks definieren zu können.
Durch ein `#pragma omp task` wird ein Task erzeugt. Häufig erfolgt die Erzeugung eines Tasks in OpenMP in einem Single Bereich `#pragma omp single` durch ein

Thread. Der erzeugte Task kann sofort oder später von verfügbaren Threads ausgeführt werden.
In OpenMP können Reihenfolgebedingungen als Abhängigkeit zwischen Tasks mit der `depend` Klausel festgelegt werden. Als Parameter können in der Klausel `#pragma omp task depend (dependency-typ: var)` der Typ einer Abhängigkeit `dependency-typ` und eine Variable, Liste oder Bereich eines Arrays übergeben werden, die die Reihenfolgebedingungen festlegen. Die verwendeten `dependency-typen` sind zum einen `inout`, das definiert, dass der Task Lese- und Schreibzugriffe auf den Bereich von `var` haben muss. Zum anderen `in`, was bedeutet, dass der Task auf den Bereich `var` Lesezugriffe haben muss. Der Lesezugriff kann nur dann erfolgen, wenn ein Task mit Schreibzugriff z.B. `inout` seinen Wert berechnet hat. Ein Beispiel erfolgt mit `#pragma omp task depend (inout: A[i:k][j:k])` ein Lese- und Schreibzugriff durch den Typen `inout` auf ein zweidimensionales Array `A`. In den Bereichen des Arrays können die Variablen i und j als Zeile und Spalte angesehen werden und k als Größe des Bereiches. Weitere Informationen zu den Bestandteilen von OpenMP können der offiziellen Dokumentation [13] und Spezifikation [12] entnommen werden.

2.2 Cilk

Cilk wurde 1994 als Spracherweiterung von C++ für Rechner mit gemeinsamen Speicher am MIT entwickelt. Der Name Cilk ist kein Akronym, sondern eine Anspielung auf silk (Seide). In 2009 wurde es von der Intel Corporation erworben, seit 2018 steht Cilk als Open Source Implementierung (OpenCilk) zur Verfügung [5] [6]. Das Besondere an Cilk ist die Verwendung von lediglich drei Befehlen.

1. `cilk_for`
2. `cilk_spawn`
3. `cilk_sync`

`cilk_for` wird verwendet, um Schleifen zu parallelisieren. Durch `cilk_spawn` wird ein Task erstellt, der parallel zu anderen erstellten Tasks arbeiten darf. Dieser Befehl ist vergleichbar mit der Task-Direktive `#pragma omp task` von OpenMP. Mit dem Befehl `cilk_sync` wird auf alle Tasks gewartet, die innerhalb einer Methode erzeugt wurden. Ein implizites `cilk_sync` wird beim Verlassen einer Methode automatisch ausgeführt.
Der Compiler und die Laufzeitumgebung in Cilk sind für die Verteilung von Tasks auf die vorhandenen Prozessoren verantwortlich. Diese Technik wird als Workstealing bezeichnet. Jeder Worker, zur Vereinfachung Prozessor, arbeitet zuerst alle Tasks ab, die ihm zugewiesenen wurden. Sind alle Tasks innerhalb eines Workers abgearbeitet, versucht er als Dieb (thief) existierende Arbeit von anderen Workern (victim) zu stehlen (steal). Jeder Worker besitzt eine Struktur, um ein Programm auszuführen. Die Struktur besteht aus Elementen aus einer verketteten Liste mit Frames, die in einer `dequeue` (double ended queue) gespeichert sind, um verfügbare Arbeit stehlen zu können.

2.3 Cilk-F

Cilk-F wurde im Jahr 2019 entwickelt. Dabei bezieht sich das F auf Futures, die in Cilk-F verwendet werden können. Ein Future ist ein Platzhalter für ein zunächst unbekanntes, aber zu einem späteren Zeitpunkt (Zukunft) verfügbarem Rechenergebnis. Wenn auf ein Future durch z.B. einen Thread (A) zugegriffen wird, um eine Variable auszulesen, wird solange gewartet, bis der entsprechende Thread (B), der für die Berechnung des Wertes zuständig ist, den Wert errechnet hat.
Außerdem kann Thread (B) das Ergebnis solange bereithalten bis Thread (A) darauf zugreifen möchte. Dies kann, muss aber nicht, parallel ausgeführt werden. Durch dieses Verhalten wird Thread (A) nicht zwangsweise durch Thread (B)

geblockt. Durch die Benutzung von Futures ist es sowohl möglich Algorithmen mit den fork/join Modell zu programmieren sowie auch komplexere Algorithmen mit Abhängigkeiten.
Durch Futures können Werte beispielsweise von einem Thread an einen anderen übergeben werden. Dies könnte auch mit globalen Variablen bewerkstelligt werden. Durch die Benutzung von Futures kommt die Variablenübergabe ohne globale Variablen aus und gewährleistet Synchronisation.
Außerdem wird in Cilk-F Proactive Workstealing eingesetzt. Der Hauptunterschied zum Workstealing, welcher in Cilk eingesetzt wird, ist immer wenn ein Future/Task pausiert, stiehlt er Arbeit. Im Gegensatz dazu wird in Cilk das Stehlen von Arbeit sparsamer vorgenommen, damit der Synchronisationsaufwand klein gehalten wird.
In Cilk-F kann ein Future durch `fut-create` erstellt werden und durch eine Berührung (touch) des Futures durch ein `.get()` auf das Future gewartet werden. Dies hat Ähnlichkeit zu Cilk, in Cilk kann ein Task durch `cilk_spawn` erstellt werden und durch `cilk_sync` synchronisiert.

2.3.1 Cilk-F direkte Future Erzeugung

In Cilk-F existiert eine Methode mit der man ein Future direkt erzeugen kann. Die Methode `cilk_future_create__stack` Listing 2.1 (Zeile 3) kann wie ein `cilk_spawn` angesehen werden. Der Name des Futures und Datentyp des Rückgabewertes sind nach dem Beispiel aus Listing 2.1 gewählt worden, der Name eines Futures kann beliebig sein und andere Datentypen sind als Rückgabewerte möglich. Die Bezeichnungen in den Parametern beziehen sich auf Listing 2.1.

Die Funktion `cilk_future_create__stack` bekommt als Parameter:

1. Rückgabewert der aufzurufenden Funktion `int`
2. Name des Futures `x_fut`
3. Die aufzurufende Funktion `f`
4. Parameter der zu aufrufenden Funktion `n-1`, mehrere Parameter sind möglich

Mit einem `.get()` Aufruf auf dem Future `x_fut` kann auf den zu berechnenden Wert gewartet werden. Durch diesen `.get()` Aufruf wird der Worker, der den Wert aus dem Future `x_fut` benötigt, pausiert. Dieser Wert wird zurückgegeben, wenn die Berechnung abgeschlossen ist.

```
1 int fib(int n){
2   int x;
3   cilk_future_create__stack(int, x_fut, fib, n-1);
4   // berechnung
5   x = x_fut.get();
6   return x;
7
8 }
```

Listing 2.1: cilk_future_create___stack

Die Methode `cilk_future_create__stack` kann benutzt werden, um ein Future zu erstellen, ist aber in der Praxis nicht benutzbar, da man immer einen eingeschränkten Gültigkeitsbereich hat, wie in dem nächsten Beispiel zu sehen ist. In Listing 2.2 wird ein Future `x_fut` innerhalb einer `if` Abfrage erstellt. Dieses Future kann nur innerhalb des Gültigkeitsbereichs der `if` Abfrage (Zeilen 3-6) durch ein `.get()` verwendet werden, um einen Wert zurückzuliefern. Wenn ein `.get()` nach dem Gültigkeitsbereich der `if` Abfrage in Listing 2.2 (Zeile 7) nötig sein sollte, ist so etwas mit dem Future `x_fut` nicht möglich.

```
1  int f(int n){
2    int x;
3    if(a == true){
4    cilk_future_create__stack(int, x_fut, f, n-1);
5    // berechnung
6    }
7    x = x_fut.get(); // Fehler
8    return x;
9  }
```

Listing 2.2: cilk_future_create__stack falsch

Ein Möglichkeit sollte noch bestehen, um diese Problematik aus Listing 2.2 zu umgehen. Man kann ein Future außerhalb der `if` Abfrage definieren, siehe Listing 2.3 (Zeile 2), um dann auf dieses Future zugreifen zu können. Leider endet dieses Vorgehen in einem `Segmention fault`. Dies ist der Fall, wenn auf Speicher zugegriffen wird, der Speicher aber vor einem solchen Zugriff geschützt ist.

```
1  int f(int n){
2    cilk::future<int> x_fut = cilk::future<int>();
3    int x;
4    if(a == true){
5    cilk_future_create__stack(int, x_fut, f, n-1);
6    // berechnung
7    }
8    x = x_fut.get(); // Fehler
9    return x;
10 }
```

Listing 2.3: cilk_future_create__stack falsch 2

Im Elternsystem Cilk würde eine Implementierung aus Listing 2.3 wie in Listing 2.4 aussehen und würde korrekt parallel funktionieren. Die benötigen Tasks werden gestartet (Zeile 4) und erst nach der Schleife findet die Synchronisation statt (Zeile 6).

```
1  void f(int n){
2    int x;
3    for(int i = 0; i < n; i++) {
4        x = cilk_spawn g(i);
5    }
6    cilk_sync;
7  }
8  int g(int i){
9      // Berechnung
10 }
```

Listing 2.4: cilk_spawn

Das nächste Vorgehen, um die Problematik des eingeschränkten Gültigkeitsbereiches aus Listing 2.3 mit Futures zu lösen, wäre Listing 2.5, ein Future Array mit der Größe `n` zu erstellen (Zeile 2), um dieses Future Array nach der Schleife nach den Werten der Berechnung zu fragen. Dieses Vorgehen liefert einen korrekten, aber keinen effizienten Algorithmus. Dieses Vorgehen wurde in den NQueens Benchmarks getestet, die Laufzeit war äquivalent zu der sequentiellen Laufzeit.

```
1  void f(int n){
2  cilk::future<int> *future_array = new cilk::future<int>[n];
3      for(int i = 0; i < n; i++) {
4          cilk_future_create__stack(int, x_fut, g, i);
5          future_array[i] = x_fut;
6      }
7      for(int i = 0; i< n ; i++){
8          future_array[i].get();
9      }
10 }
11 int g(int i){
12     // Berechnung
13 }
```

Listing 2.5: Cilk-F Future Array

Warum ist also dieses Vorgehen ineffizient? Warum ist das gewünschte Verhalten von Futures in Cilk-F nicht so einfach durch die `cilk_future_create__stack` Methode umzusetzen? Und warum kann ein Future aus den Future Array nicht direkt gestartet werden?
Um diese Fragen auf eine vernünftige Art und Weise beantworten zu können, sollte man vorher wissen, dass ein Teil der Arbeit des Compilers in Cilk-F durch den Programmierer erledigt werden muss, aufgrund des unvollendeten Entwicklungsstandes von Cilk-F. Um die Arbeit des Compilers erledigen zu können, müssen Grundlagen im Elternsystem Cilk geschaffen werden. Zum einen welcher Code aus einem `cilk_spawn` in Cilk generiert wird und zum anderen welche Datenstrukturen durch die Cilk Laufzeitumgebung verwaltet werden. Diese Grundlagen werden im nächsten Unterkapitel thematisiert.

2.3.2 Cilk Grundlagen

Was der Compiler durch ein `cilk_spawn` generiert, wird nachfolgend an einem Beispiel erklärt. Die Funktion `f` Listing 2.6 (Zeile 1) ruft einen `cilk_spawn` auf und wird als aufrufende Funktion bezeichnet. Eine Funktion `g` (Zeile 3) wird als aufgerufene Funktion bezeichnet und durch einen `cilk_spawn` aufgerufen.

```
1 int f(int n){
2   int x;
3   x = cilk_spawn g(n);
4   cilk_sync;
5   return x;
6 }
```

Listing 2.6: cilk_spawn Beispiel

In Cilk werden unter anderem zwei Arten von Frames verwendet, eine Art ist der Cilk Stack Frame und die andere ist ein Full Frame. Ein Frame enthält Informationen über lokale Variablen, temporäre Variablen und Informationen für Rückgabewerte, ergo enthält ein Frame den Zustand der Variablen eines

Programmes.

In einem Cilk Stack Frame ist zusätzlich ein `context buffer` [`sf.ctx`] (Rücksprungadresse), welcher genügend Informationen enthält, um eine Funktion nach einem `cilk_spawn` oder `cilk_sync` fortzusetzen. Ein Full Frame enthält die selben Informationen wie ein Cilk Stack Frame und darüber hinaus noch weitere Informationen. Sonstige unterschiedliche Details braucht man zu den Frames für die Programmierung nicht zu wissen. Für weitere Details kann [15] verwendet werden.

Der Compiler generiert für einen Aufruf von `cilk_spawn` zusätzlichen Code, um die benötigte nachfolgende Struktur zu erstellen. Folgender Text bezieht sich auf Listing 2.7.

Als Erstes wird in der aufrufenden Funktion `f`, einmal pro Methodenaufruf von `f`, ein Full Frame erstellt und initialisiert (Zeilen 2 und 3). Anschließend wird die Fortsetzung von `f` in einem `context Buffer` (Zeile 5) gespeichert, wodurch die Funktion für einen `spawn` vorbereitet wird. Der Compiler generiert eine Hilfsfunktion `hilfs_funktion_g`, um die aufzurufende Funktion `g` aufzurufen, die Hilfsfunktion wird nach dem Speichern des `context Buffer` aufgerufen.

In der Hilfsfunktion (Zeile 16) wird ein Cilk Stack Frame erstellt und initialisiert. Diese erzeugte Hilfsfunktion ist nötig, um die erzeugten Frames zu trennen, damit jeder erzeugte Task seinen eigenen Frame hat. Beispielsweise um ein Wert zu inkrementieren, um auf diesen Wert race conditions zu vermeiden und damit das Workstealing in Cilk funktioniert. Auf das Workstealing in Cilk wird nicht weiter eingegangen. Race Conditions treten auf, wenn z.B. zwei Tasks gleichzeitig eine Variable inkrementieren wollen.

Durch das `detach` (Zeile 19) in der Hilfsfunktion wird die `dequeue` aktualisiert und der Cilk Stack Frame auf die verkettete Liste der Worker gelegt. Der Aufruf der aufzurufenden Funktion `g` wird nach dem `detach` ausgeführt. Nach der Ausführung der aufzurufenden Funktion `g` wird der Cilk Stack Frame aus der `dequeue` des Worker entfernt.

Nach dem Beenden der Hilfsfunktion `hilfs_funktion_g` in der aufrufenden Funktion `f` wird `cilk_sync` ausgeführt. Eine Berechnung der Synchronisation ist aufwendig, aus diesem Grund wird geprüft (Zeile 7), ob es wirklich nötig ist zu synchronisieren. Das `sf.flags` wird dann gesetzt, wenn ein Diebstahl stattgefunden hat. Zum Schluss der Funktion `f` wird der Full Frame aus der Liste der dequeue genommen und gelöscht.

```
1  int f(int n){ //aufrufende Funktion
2    __cilkrts_stack_frame_t sf; // Erstellen Full Frame
3    __cilkrts_enter_frame(&sf); // Initialisieren Full Frame
4    int x;
5    if(!setjmp(sf.ctx)) // Setzen der Ruecksprungadresse
6      hilfs_funktion_g(&x , n); // Rufe Hilfsfunktion auf
7    if(sf.flags & CILK_FRAME_UNSYNCHED)
8      if(!setjmp(sf.ctx))
9    __cilkrts_sync(&sf); // Synchronisation
10   int result = x;
11   __cilkrts_pop_frame(&sf); // Aus Liste entnehmen
12   if(sf.flags)
13     __cilkrts_leave_frame(&sf); // Loesche Full Frame
14   return result;
15 }
16 void hilfs_funktion_g(int * x, int n){  // Hilfsfunktion
17   __cilkrts_stack_frame sf; // Erstellen Cilk Stack Frame
18   __cilkrts_enter_frame_fast(&sf); // Initialisieren Cilk Stack Frame
19   __cilkrts_detach(); // Cilk Stack Frame auf Liste legen
20   *x = g(n); // aufzurufende Funktion
21   __cilkrts_pop_frame(&sf); // Aus Liste entnehmen
22   if(sf.flags)
23     __cilkrts_leave_frame(&sf); // Loesche Cilk Stack Frame
24 }
```

Listing 2.7: cilk_spawn Compiler

Zu der Arbeit, die in Cilk-F übernommen werden muss, zählt die Erstellung der beschriebenen Struktur mit den Frames und die Programmierung der Hilfsmethode. Durch die Entwickler wurden einige Makros definiert, so kann

der Aufwand bei der Programmierung mit Cilk-F reduziert werden. Anstatt z.B. beim Erstellen und Initialisieren eines Full Stack Frame zwei Befehle zu benutzen `__cilkrts_stack_frame_t sf;` und `__cilkrts_enter_frame(&sf);`, werden die Befehle durch ein Makro abgekürzt `CILK_FUNC_PREAMBLE`. Die Makros sind unter dem Pfad cilkrtssuspend/include/cilk/hancompiled-macros.cpp zu finden und werden in Abschnitt 2.3.3 genauer erklärt.
Durch die Makros kann Listing 2.7 vereinfacht werden, wie in Listing 2.8 zu sehen. Die Namen und Beschreibungen der Funktionen/Hilfsfunktionen werden aus Listing 2.7 übernommen.

```
1  int f(int n){
2    CILK_FUNC_PREAMBLE;
3    int x;
4    START_FIRST_FUTURE_SPAWN;
5      hilfs_funktion_g(&x , n);
6    END_FUTURE_SPAWN;
7    int result = x;
8    CILK_FUNC_EPILOGUE;
9    return result;
10 }
11 void hilfs_funktion_g(int * x, int n){
12   FUTURE_HELPER_PREAMBLE;
13   *x = g(n);
14   FUTURE_HELPER_EPILOGUE
15 }
```

Listing 2.8: cilk_spawn mit Makros

Durch das Verständnis der Grundlagen sollte nun die Beantwortung der Fragen am Ende des Abschnittes 2.3.1 möglich sein. Warum ist das gewünschte Verhalten von Futures in Cilk-F nicht so einfach durch die `cilk_future_create__stack` Methode umzusetzen? Warum ist also dieses Vorgehen aus Listing 2.5 ineffizient? Die `cilk_future_create__stack` Methode initialisiert die nötige Struktur und Hilfsfunktion, aber diese erzeugte Struktur und Hilfsfunktion ist nur im Gültigkeitsbereich der Schleife verfügbar Listing 2.9 (Zeilen 3-10), was

durch die Kommentare in Listing 2.9 nochmal verdeutlicht werden soll. Die Ineffizienz kommt daher, dass nach dem Löschen eines Full Frames eine implizite Synchronisation durch das Cilk Laufzeitsystem stattfindet. Diese Synchronisation findet in Zeile 9 statt. Somit ist das Abfragen des Wertes (Zeile 12) belanglos.

```
1  void f(int n){
2  cilk::future<int> *future_array = new cilk::future<int>[n];
3      for(int i = 0; i < n; i++) {
4          // Full Frame
5          cilk_future_create___stack(int, x_fut, g, i);
6          future_array[i] = x_fut;
7          // Hilfsfunktion hilfs_funktion_g
8          // Full Frame loeschen
9          // Implizite Synchronisation
10     }
11     for(int i = 0; i< n ; i++){
12         future_array[i].get();
13     }
14 }
15 int g(int i){
16     // Berechnung
17 }
```

Listing 2.9: Cilk-F create_stack falsch

Die richtige Initialisierung der Struktur und Hilfsfunktion, die in Cilk-F stattfinden muss, kann aus Listing 2.10 entnommen werden. Der Full Frame wird in (Zeile 2) angelegt. Ein Löschen des Full Frames und eine implizite Synchronisation sind am Ende der Funktion `f` nötig (Zeilen 11 und 12). Die Hilfsfunktion wird außerdem außerhalb der aufrufenden Funktion `f` definiert.

```
1  void f(int n){
2  // Full Frame
3  cilk::future<int> *future_array = new cilk::future<int>[n];
4      for(int i = 0; i < n; i++) {
5          cilk_future_create___stack(int, x_fut, g, i);
6          future_array[i] = x_fut;
7      }
8      for(int i = 0; i< n ; i++){
9          future_array[i].get();
10     }
11 // Full Frame loeschen
12 // Implizite Synchronisation
13 }
14
15 // Hilfsfunktion hilfs_funktion_g
16
17 int g(int i){
18     // Berechnung
19 }
```

Listing 2.10: Cilk-F create_stack richtig

Warum kann ein Future aus dem Future Array nicht direkt gestartet werden? Ein Future braucht einen eigenen Stack Frame, zudem muss es auf die verkette Liste eines Workers gelegt werden, damit der Worker dieses Future bearbeiten kann und Proactive Workstealing ermöglicht wird. Diese Arbeiten können durch die Makros und Programmierung einer Hilfsfunktion erledigt werden.

2.3.3 Cilk-F Makros

Nachfolgend werden die verwendeten Makros beschrieben. Die Beschreibung erfolgt sehr allgemein und es wird nur das beschrieben, was für die Programmierung wichtig ist, zusätzlich wird die verwendete Stelle der Makros im Programm erwähnt. Der Sinn der Makros ist es, den Aufwand bei der Programmierung mit Cilk-F zu reduzieren.

1. CILK_FUNC_PREAMBLE

Dieses Makro `CILK_FUNC_PREAMBLE` wird verwendet, um den Full Stack Frame zu erstellen und zu initialisieren. Es wird am Anfang der aufrufenden Funktion `f` verwendet.

```
1 int f(int n){ //aufrufende Funktion
2   CILK_FUNC_PREAMBLE;
3   int x;
4   // weiterer code
5 }
```

Listing 2.11: CILK_FUNC_PREAMBLE

2. CILK_FUNC_EPILOGUE

Das zweite Makro `CILK_FUNC_EPILOGUE` wird benutzt, um am Ende der aufrufenden Funktion `f` den Full Frame aus der Liste der dequeue zu nehmen und zu löschen. Außerdem findet durch das Makro eine implizite Synchronisation beim Verlassen der Methode statt.

```
1 int f(int n){ //aufrufende Funktion
2   // weiterer code
3   CILK_FUNC_EPILOGUE;
4   return x;
5 }
```

Listing 2.12: CILK_FUNC_EPILOGUE

3. START_FIRST_FUTURE_SPAWN

Vor dem Aufruf der Hilfsfunktion wird das Makro `START_FIRST_FUTURE_SPAWN` benutzt, demzufolge wird ein `cilk_fiber`, ein Task in Cilk, gestartet. Des Weiteren wird durch dieses Makro der `Context Buffer` (Rücksprungadresse) definiert. Um dieses Makro benutzen zu können, muss vorher das

Makro `CILK_FUNC_PREAMBLE` benutzt werden, damit das Makro `START_FIRST_FUTURE_SPAWN` Zugriff auf den Full Stack hat.

```
1 int f(int n){ //aufrufende Funktion
2   CILK_FUNC_PREAMBLE;
3   // weiterer code
4   START_FIRST_FUTURE_SPAWN;
5   hilfs_funktion_g(&x , n);
6   // weiterer code
7   return x;
8 }
```

Listing 2.13: START_FIRST_FUTURE_SPAWN

4. END_FUTURE_SPAWN

Eine Benutzung dieses Makros `END_FUTURE_SPAWN` ist nach dem Aufruf der Hilfsfunktion notwendig. Dadurch wird der Task aus der dequeue des Worker genommen und es werden Referenzen/Pointer des Tasks aus der dequeue des Worker entfernt.

```
1 int f(int n){ //aufrufende Funktion
2   // weiterer code
3   START_FIRST_FUTURE_SPAWN;
4   hilfs_funktion_g(&x , n);
5   END_FUTURE_SPAWN;
6   return x;
7 }
```

Listing 2.14: END_FUTURE_SPAWN

5. FUTURE_HELPER_PREAMBLE

Dieses Makro `FUTURE_HELPER_PREAMBLE` wird zu Beginn der Hilfsfunktion verwendet, somit wird ein Cilk Stack Frame erstellt und initialisiert. Zudem wird in diesem Makro ein `detach` aufgerufen, daher wird der Cilk Stack Frame auf die verkettete Liste der Worker gelegt. Ein Code-Beispiel ist in Listing 2.15 (Zeile 2) zu sehen.

`6. FUTURE_HELPER_EPILOGUE`

Makro sechs `FUTURE_HELPER_EPILOGUE` wird am Ende einer Hilfsfunktion verwendet, da nach Ausführen der Hilfsfunktion der Cilk Stack Frame aus der dequeue des Worker entfernt werden sollte. Ein Code-Beispiel ist in Listing 2.15 (Zeile 7) zu sehen.
Durch diese Makros wird die Programmierung mit Cilk-F vereinfacht, da man nicht alle Anweisungen von Hand schreiben muss.

2.3.4 Cilk-F Hilfsfunktion

Durch dieses Unterkapitel wird die Benutzung der Hilfsfunktion beschrieben. Bei einem `cilk_spawn` in Cilk wird die Hilfsfunktion automatisch erzeugt (Abschnitt 2.3.2). In Cilk-F ist dies, durch den aktuellen Entwicklungsstand nicht der Fall, außerdem werden in der Hilfsfunktion in Cilk-F mehr Code benötigt als in Cilk. Es gibt zwei Arten von Hilfsfunktionen, einmal ohne Rückgabewert und einmal mit. Des Weiteren setzt sich dieses Kapitel mit den Bausteinen, die in der Hilfsfunktion verwendet werden, auseinander. Die Eigenschaft `__attribute__((noinline))` (Zeile 1) unterdrückt das direkte Einfügen dieser Funktion in dem Aufrufpunkt der Hilfsfunktion. Diese Eigenschaft wird verwendet, wenn ein Funktionsaufruf durch die Cilk Laufzeitumgebung verwaltet werden soll, statt durch den Compiler.
Beschreibung der Hilfsfunktion ohne Rückgabewert Listing 2.15, wobei die Abkürzung 'Z.' Zeile bedeutet:

Z.1) Die Hilfsfunktion bekommt als Parameter ein Future sowie alle anderen Parameter der zu aufrufenden Funktion `nqueens(j, queens)` (Zeile 3).

Z.2) Hier wird das Makro `FUTURE_HELPER_PREAMBLE;` angewendet.

Z.3) Der Funktionsaufruf der Funktion `nqueens(j, queens)` erfolgt hier.

Z.5) Füge das Future zur deque des Workers hinzu. Dies geschieht durch das Aufrufen von `.put()` ohne Parameter auf den Future.

Z.6) Definiere die deque als eine deque, die von jedem Worker fortgesetzt werden kann.

Z.7) Am Ende der Hilfsfunktion wird das Makro `FUTURE_HELPER_EPILOGE;` benutzt.

```
1 void __attribute__((noinline)) function_helper(cilk::future<int> *fut, int
    j, char *queens) {
2 FUTURE_HELPER_PREAMBLE;
3 nqueens(j, queens);
4 __asm__ volatile ("" ::: "memory");
5 void pointer __cilkrts_deque = fut->put();
6 if (__cilkrts_deque) __cilkrts_resume_suspended(__cilkrts_deque, 2);
7 FUTURE_HELPER_EPILOGUE;
```

Listing 2.15: Future Hilfsfunktion ohne Rückgabebwert

Beschreibung der Hilfsfunktion mit Rückgabewert Listing 2.16: Wenn das Future einen Rückgabewert besitzt, wird in das `.put()` des Futures die aufzurufende Methode geschrieben `fut->put(nqueens(j, queens);` (Zeile 3).

```
1 void __attribute__((noinline)) function_helper(cilk::future<int> *fut, int
    j, char *queens) {
2 FUTURE_HELPER_PREAMBLE;
3 void *__cilkrts_deque = fut->put(nqueens(j, queens));
4 if (__cilkrts_deque) __cilkrts_resume_suspended(__cilkrts_deque, 2);
5 FUTURE_HELPER_EPILOGUE;
```

Listing 2.16: Future Hilfsfunktion mit Rückgabebwert

3 Benchmarks und Implementierungen

3.1 NQueens

Das NQueens-Problem berechnet alle Möglichkeiten N Damen auf einem Schachbrett der Größe N * N zu positionieren, ohne dass sich die Damen laut den Zugmöglichkeiten der Schachregeln schlagen können. Anschaulicher ausgedrückt: Auf derselben Reihe, Linie oder Diagonale stehen niemals zwei Damen. Der Benchmark kalkuliert alle möglichen Positionen von N Damen auf dem Schachbrett. Die Zahl N wird als Parameter mitgegeben. Ausgabe ist die Anzahl aller möglichen Positionen der Damen. Es wurden vier parallele Algorithmen entwickelt, um das Problem zu lösen.

a) OpenMP mit Threads/Cilk-F mit Futures.

b) OpenMP/Cilk-F mit einstellbarer Anzahl an Tasks/Futures.

Bei der Implementierung der Benchmarks a) in OpenMP wurde eine feste Anzahl an Threads verwendet. In Cilk-F wurde eine feste Anzahl an Futures verwendet, die Anzahl der Threads und Futures hängt von der Eingabe N ab. In den Benchmarks aus b) kann man die Anzahl der Tasks/Futures einstellen. Der Benchmark aus b) in Cilk-F ist komplizierter zu implementieren als der Benchmark aus a) mit Cilk-F, da man die Verwaltung der Futures übernehmen muss, so wurden zwei verschiedene Algorithmen geschrieben.
Die Beschreibung des sequentiellen Algorithmus erfolgt der Übersichtshalber vor Abschnitt 3.1.4.

3.1.1 OpenMP mit Threads

Mit OpenMP lässt sich NQueens auf mehrere Arten programmieren. Der erste Ansatz ist die Benutzung von OpenMP-Threads. Eine Referenz Implementierung

kann auf der Internetseite [7] gefunden werden. Diese Implementierung wurde auf Grund der intelligenten Umsetzung der Schachregeln und der sparsamen Benutzung von Speicherplatz genommen. Diese Implementierung wurde genutzt, da dadurch hohe Effizienz bezüglich der Laufzeit erwartet wurde.
Die Grundidee ist, so viele Threads zu starten, wie die Eingabe N groß ist, jeder Thread bekommt ein Array, auf dem er die Platzierungen der Damen überprüfen kann, wie in Abbildung 3.2 dargestellt. Die sparsame Version des Speicherplatzes kommt daher, dass in einem eindimensionalen `int` Array die Spalte, auf der eine Dame steht, gespeichert wird. Die Zeile ist die Stelle des Arrays. Abbildung 3.1 zeigt ein Beispiel, wo die Damen auf den Feldern (Zeile, Spalte), (0,1), (1,2) und (2,0) sind.

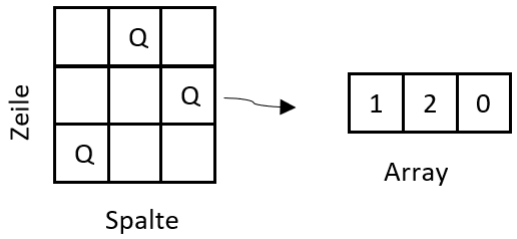

Abbildung 3.1: Damen in Array

In der `Main` Methode, der Startpunkt einer Anwendung (Zeile 1), wird die Methode aufgerufen, die die parallele Ausführung startet (Zeile 5). Der Einfachheitshalber wird in den nächsten Listings die `Main` Methode weggelassen, da nur geringe Unterschiede existieren. Außerdem wird um diesen Aufruf `rec_nqueens_par_premble();` die Zeitmessung durchgeführt (Zeilen 4 und 6) für das Kapitel 5. Diese Zeitmessung erfolgt in den nächsten Listings analog. Die Variable `nr_of_solutions` wurde als globale Variable definiert.
Folgendes bezieht sich auf Listing 3.1. Es wird ein paralleler Bereich erstellt (Zeile 12). Die Anzahl der gestarteten Threads beträgt dabei die Eingabe N durch `num_threads(FIELDS)` (Zeile 13). Jeder Thread ruft die Methode `rec_nqueens_par` (Zeile 13) mit den Parametern auf: ein neues eindimensionales

Array, die Zeile, in der gestartet wird und die Spalte, in der die erste Dame platziert werden soll. Die Methode `omp_get_thread_num()` (Zeile 13) liefert die Threadnummer des aufzurufenden Threads, da N Threads erstellt worden sind, werden alle Zahlen von 0 bis N - 1 an die Methode `rec_nqueens_par` (Zeile 15) übergeben. Diese Zahlen sind die Platzierung einer Dame auf der ersten Zeile mit Spaltennummer 0 bis N - 1.

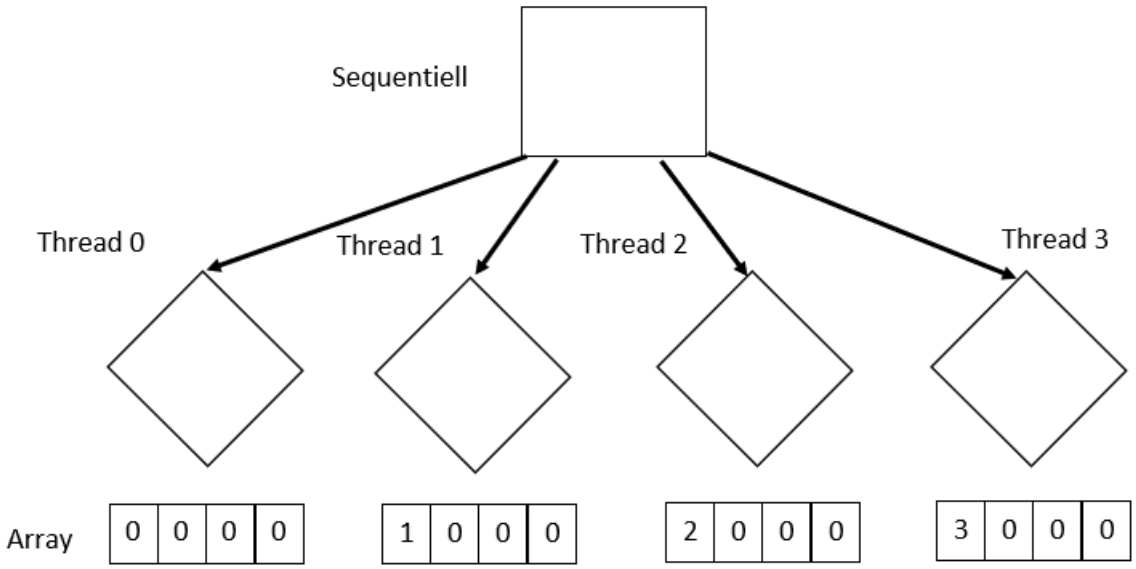

Abbildung 3.2: Erstelltes Modell OpenMP Threads

Eine Iteration bis zur aktuellen Zeilenposition `row` findet statt (Zeile 16). In der Iteration wird für jede Zeile (Stelle im Array) geprüft, ob der Wert der Spalte doppelt vorkommt (Zeile 17). Das würde bedeuten, dass sich zwei Damen in der selben Zeile oder Reihe befinden. Bei der nächsten Überprüfung wird die Diagonale der Damen geprüft (Zeile 19), dabei gibt es zwei Möglichkeiten wie die Damen sich in der Diagonalen schlagen können: b - a = v[b] - v[a]
(Abbildung 3.3) oder b - a = v[a] - v[b] (Abbildung 3.4). Diese Überprüfung kann mit b - a = absoluter Betrag |v[a] - v[b]| vereinfacht werden.

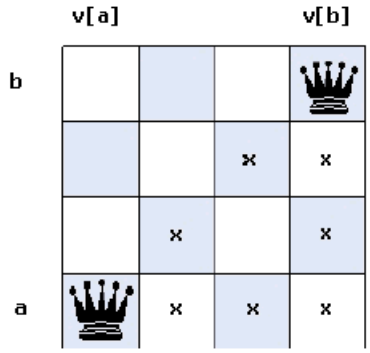

Figure 12

Abbildung 3.3: Diagonale 1

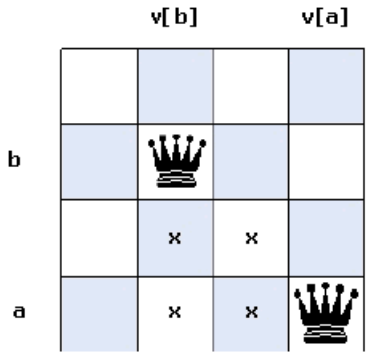

Figure 13

Abbildung 3.4: Diagonale 2

Danach wird in das `queens` Array die Spaltennummer der Dame platziert (Zeile 22). Eine Überprüfung, ob N Damen platziert worden sind, findet hier statt (Zeile 23). Falls die Überprüfung erfolgreich ist, wird der Wert `nr_sol_par` mit Hilfe einer `#pragma omp atomic` Direktive inkrementiert (Zeile 25). Die Direktive (Zeile 24) wird benutzt, da die Möglichkeit besteht, dass mehrere Threads gleichzeitig den Wert inkrementieren möchten. Wenn noch nicht N Damen platziert wurden, findet ein weiterer Aufruf der Methode statt (Zeile 29). Als Parameter wird die nächste Zeile übergeben und die Laufvariable in der Schleife (Zeile 28), die über die Spalten, iteriert.

```
1  int main(int argc, char *argv[])
2  {
3      double t_par1, t_par2;
4      t_par1 = omp_get_wtime();
5      rec_nqueens_par_premble();
6      t_par2 = omp_get_wtime();
7      printf("Time=%f\n", t_par2 - t_par1);
8      return 0;
9  }
10 int nr_of_solutions = 0;
11 void rec_nqueens_par_premble(){
12 #pragma omp parallel num_threads(FIELDS){
13   rec_nqueens_par(new int[FIELDS], 0, omp_get_thread_num());
14 }
15 void rec_nqueens_par(int queens[], int row, int col) {
16   for(int i = 0; i  < row; i++) {
17       if (queens[i] == col ) {
18           return; }
19       if (abs(queens[i]-col) == (row-i) ) {
20           return; }
21     }
22     queens[row] = col;
23     if(row == FIELDS-1) {
24 #pragma omp atomic
25           nr_of_solutions++;
26     }
27     else {
28         for(int i = 0; i < FIELDS; i++) {
29             rec_nqueens_par(queens, row + 1, i);
30         }
31     }
32 }
```

Listing 3.1: NQueens OpenMP mit Threads

3.1.2 Cilk-F mit Futures

In dieser Implementierung war es das Ziel den Code so einfach wie möglich in Cilk-F zu programmieren, um ein Ziel der Entwickler von Cilk-F zu verdeutlichen, eine einfache Benutzung zu ermöglichen, wie mit dem Elternsystem Cilk, indem

wenige Schlüsselwörter genügen, um parallel zu programmieren.
Die Grundidee ist mit der Art und Weise, aus dem vorherigen Listing 3.1 zu arbeiten in Bezug auf die Überprüfung der Platzierungen der Damen und die Benutzung eines Arrays für jedes Future. Es sollen Eingabe N viele Futures gestartet werden. Diese Futures sind für die Berechnung eines Teilergebnisses zuständig. Nachdem die Futures gestartet wurden, soll auf die Teilergebnisse durch einen `.get()` Aufruf auf alle Futures gewartet werden. Nachdem die Teilergebnisse berechnet worden sind, sollen sie zu einem Endergebnisse addiert werden, wie in Abbildung 3.5 zu sehen.

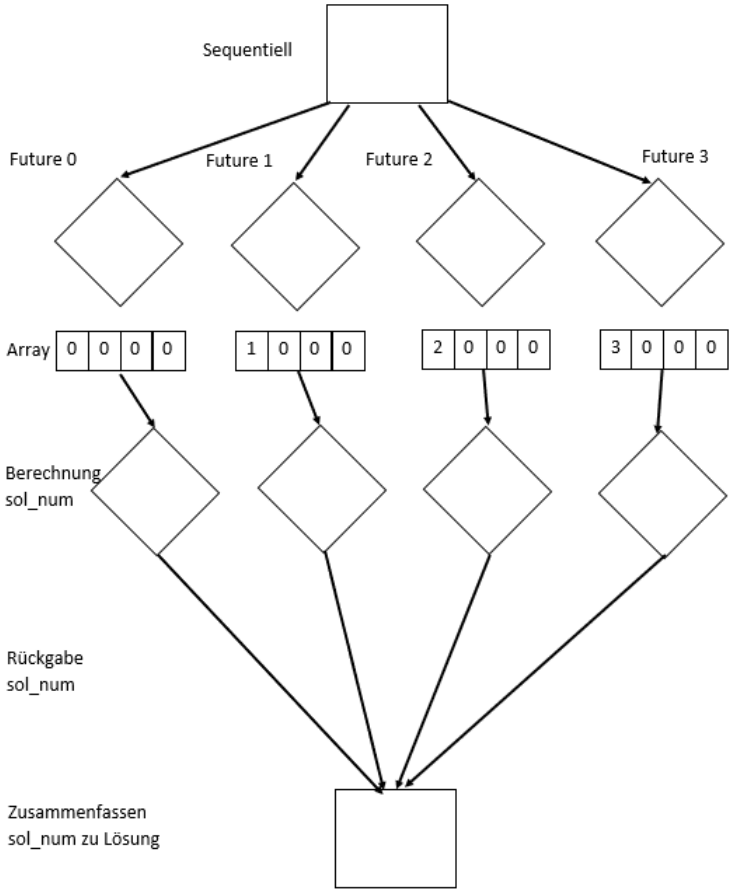

Abbildung 3.5: Erstelltes Modell Cilk-F

Das Folgende bezieht sich auf Listing 3.2. Die Hauptunterschiede zu Listing 3.1 sind, dass beim Aufruf der Methode `rec_nqueens_par_premble()`

ein Array `count_all` initialisiert wird (Zeile 9), die Größe des Array ist dabei die Anzahl der noch zu startenden Futures. Nachdem die Futures gestartet wurden, hat jedes Future ein Rückgabewert für eine Teileberechnung. Das Ergebnis wird in dem erzeugten Array `count_all` gespeichert und im weiteren Verlauf (Zeilen 17-20) durch eine Schleife über alle Futures gewartet, bis der Wert der Teilebrechungen verfügbar ist. Wenn der Wert verfügbar ist, wird er auf die Variable `nr_of_solutions` addiert und als Ergebnis zurückgegeben.
In der Methode `rec_nqueens_par` (Zeile 28) wird, nach der Überprüfung, ob sich die Damen schlagen, eine `int` Variable `sol_num` und ein neues Array `count` initialisiert. In diesem Array werden die Lösungen gespeichert, die durch den Rückgabewert der Methode `rec_nqueens_par` berechnet werden. Wenn die benötigte Anzahl `N` an Damen auf dem Schachbrett steht, wird eine Eins zurückgegeben (Zeile 41), wenn die Damen sich schlagen dann eine Null
(Zeilen 31 und 33).
Die Lösungen werden auf die Variable `sol_num` addiert und beim Beenden der Funktion an die Variable `sol_num` zurückgegeben. Es werden Eingabe N viele Futures verwendet. Weil die Verwendung eines Future Array und die Verbesserung der Laufzeit ein Ziel ist, muss dafür die nötige Struktur durch die Makros aus Abschnitt 2.3.3 und nötige Hilfsfunktion Abschnitt 2.3.4 programmiert werden.
Als Erstes wird die Hilfsfunktion definiert, da das Future einen Rückgabewert besitzt, so wird eine Hilfsfunktion mit Rückgabewert aus Abschnitt 2.3.4 Listing 2.16 verwendet. In dieser Hilfsfunktion wird der Rückgabewert der Methode `premble_method` an das `fut->put` als Parameter übergeben (Zeile 3).
In der Methode `rec_nqueens_par_premble()`, die eine Teillösungen berechnet, wird die nötige Struktur durch die Makros geschaffen.

Eine kurze Zusammenfassung über die Erstellung der Struktur, in Zeile 8 wird der Full Stack erstellt und durch das Makro `CILK_FUNC_PREAMBLE;` initialisiert, am Ende der `rec_nqueens_par_premble()` Methode wird er

wieder durch das Makro `CILK_FUNC_EPILOGUE` gelöscht. Um die Hilfsfunktion `function_helper` werden zwei Makros benötigt, und zwar am Anfang das Makro `START_FIRST_FUTURE_SPAWN;` um einen Task zu erstellen und die Rücksprungadresse zu setzen, am Ende `END_FUTURE_SPAWN` um die Referenzen des Tasks zu entfernen.

Wurde die nötige Struktur erstellt, können nun die benötigten Futures als Array (Zeile 10) initialisiert werden, es werden `N` viele Futures benötigt. Mit dem Aufrufen der Hilfsfunktion `function_helper` wird ein Future mitgegeben und gestartet. Nachdem alle Futures gestartet worden sind, kann außerhalb des Gültigkeitsbereich der for Schleife (Zeilen 11-15) auf die Berechnung des Wertes durch die Futures gewartet werden. Dies geschieht indem eine for Schleife über das Future Array iteriert und auf jedem Future `.get()` aufgerufen (Zeile 18) wird. Dadurch wird sichergestellt, dass der berechnete Wert in der Zusammenfassung der Ergebnisse (Zeile 19) verfügbar ist.

```
1  void __attribute__((noinline)) function_helper(cilk::future<int> *fut, int
     i) {
2      FUTURE_HELPER_PREAMBLE;
3      void *__cilkrts_deque = fut->put(premble_method(i));
4      if (__cilkrts_deque) __cilkrts_resume_suspended(__cilkrts_deque, 2);
5      FUTURE_HELPER_EPILOGUE;
6  }
7  int rec_nqueens_par_premble() {
8      CILK_FUNC_PREAMBLE;
9      int count_all[FIELDS];
10     cilk::future<int> *future_array = new cilk::future<int>[FIELDS];
11     for (int i = 0; i < FIELDS; i++) {
12         START_FIRST_FUTURE_SPAWN;
13         function_helper(&future_array[i], i);
14         END_FUTURE_SPAWN;
15     }
16     int nr_of_solutions = 0;
17     for(int i = 0; i< FIELDS ; i++){
18         count_all[i] = future_array[i].get();
19         nr_of_solutions += count[i];
```

```
20      }
21      CILK_FUNC_EPILOGUE;
22      return nr_of_solutions;
23  }
24  int premble_method(int i) {
25      int nr_of_solutions = rec_nqueens_par(new int[FIELDS], 0, i);
26      return nr_of_solutions;
27  }
28  int rec_nqueens_par(int queens[], int row, int col) {
29      for(int i = 0; i  < row; i++) {
30          if (queens[i] == col ) {
31              return 0;}
32          if (abs(queens[i] - col) == (row - i) ) {
33              return 0;}
34      }
35      int sol_num = 0;
36      int *count;
37      count = (int *) alloca(FIELDS * sizeof(int));
38      (void) memset(count, 0, FIELDS * sizeof (int));
39      queens[row] = col;
40      if(row == FIELDS - 1) {
41         return 1;}
42      else {
43          for(int i = 0; i < FIELDS; i++) {
44             count[i] = rec_nqueens_par(queens, row + 1, i);
45          }
46      }
47      for(int i = 0; i < FIELDS; i++) {
48          sol_num += count[i];
49      }
50      return sol_num;
51  }
```

Listing 3.2: NQueens Cilk-F N-Futures

3.1.3 NQueens Sequentiell

Sequentiell wurde das Problem als rekursives Backtracking implementiert, dabei wird bei jedem rekursiven Aufruf eine Dame gesetzt und überprüft, ob die gesetzte Dame keine anderen Damen schlägt. Als Schachbrett wurde ein

zweidimensionales Bool Array gewählt.

Die folgenden Schritte beziehen sich auf Listing 3.3. Die Variable `nr_of_solutions` wurde als globale Variable definiert (Zeile 1). Außerdem wird in der `Main` die Funktion `rec_nqueens` Listing 3.3 (Zeile 2) aufgerufen.
Eine Überprüfung, ob N Damen platziert worden sind, findet statt (Zeile 3). Wenn das der Fall ist, wird `nr_of_solutions` inkrementiert (Zeile 4). Anschließend wird über jede Zeile des Array iteriert (Zeile 5) und durch die `check` Methode (Zeile 6) überprüft, ob eine Dame nach Schachregeln platziert werden kann. Die `check` Methode (Zeile 13) prüft die Zeile, Spalte und Diagonale des Schachbrettes.
Die `check` Methode prüft mit der ersten Schleife (Zeile 15), ob sich auf der mitgegebenen Zeile zwei Damen als Parameter auf derselben Spalte befinden. Die zweite Schleife (Zeile 18) prüft, ob sich Damen auf einer Diagonalen befinden. Angefangen von links unten bis rechts oben. Durch die dritte Schleife wird auch die Diagonale geprüft, allerdings von rechts unten nach links oben.
Wenn sich in der Überprüfung keine Damen schlagen, wird eine Dame gesetzt (Zeile 7). Die Funktion `rec_nqueens` (Zeile 8) wird mit den Parametern aufgerufen: Schachbrett mit der neu platzieren Dame und der nächsten Spalte `col + 1`. Nach dem Aufruf wird die Dame wieder deplatziert (Zeile 9).

```
1  int nr_of_solutions = 0;
2  void rec_nqueens(bool **A, int col)
3  {   if (col == FIELDS){
4          nr_of_solutions++;}
5      for (int i = 0; i < FIELDS; i++){
6          if (check(A, i, col) == 1){
7              A[i][col] = 1;
8              rec_nqueens(A, col + 1);
9              A[i][col] = 0;
10         }
11     }
12 }
13 bool check(bool **A, int row, int col){
14     int i, j;
15     for (i = 0; i < col; i++)//Spalten
16         if (A[row][i]){
17             return 0; }
18     for (i = row, j = col; i >= 0 && j >= 0; i--, j--)//Diagonale links
19         if (A[i][j]){
20             return 0; }
21     for (i = row, j = col; j >= 0 && i < FIELDS; i++, j--)//Diagonale rechts
22         if (A[i][j]){
23             return 0; }
24     return 1;
25 }
```

Listing 3.3: NQueens Sequentiell

3.1.4 OpenMP mit einstellbarer Anzahl an Tasks

Bei der OpenMP Implementierung mit einstellbaren Tasks wurde die Idee des sequentiellen Algorithmus verwendet. Die Compiler Direktive `#pragma omp parallel` definiert einen parallelen Bereich, indem ein `#pragma omp single` Aufruf erfolgt. Ein Single Bereich wird nur von einem Thread ausgeführt. Der Thread, der den `#pragma omp single` Bereich ausführt, generiert alle Tasks, die an alle verfügbaren Threads verteilt und ausgeführt werden.

Der folgende Text nimmt Bezug auf Listing 3.4. Ein paralleler Bereich mit

`#pragma omp parallel` und `#pragma omp single` wird gestartet
(Zeilen 4 und 5), ein Thread führt dann die Methode `solve_queens` (Zeile 6) aus. Als Parameter wird ein zweidimensionales Array und der Anfangswert für die Spalte übergeben.
Jeder Aufruf der Methode `solve_queens` erzeugt ein temporäres Array für die Berechnung (Zeilen 9 und 10). Die Überprüfung, ob N Damen platziert worden sind, findet statt (Zeile 11). Wenn das der Fall ist, wird die globale Variable `nr_sol_par` mit Hilfe einer `#pragma omp atomic` Direktive inkrementiert.
Im nächsten Schritt wird mit der aktuellen Spalte über alle Zeilen (Zeile 14) iteriert und geprüft, ob eine Dame geschlagen wird (Zeile 15). Die `check` Methode ist dabei die Gleiche wie in Listing 3.3 (Zeilen 13-25). Nach der Überprüfung wird eine Dame auf das getestete Feld gesetzt (Zeile 16). In diesem Schritt wird die Anzahl der Tasks beschränkt (Zeile 17), indem ein Taskzähler (Zeile 22) existiert, der bei jedem erzeugten Task die globale Variable `task` inkrementiert.
Durch diesen Taskzähler kann die maximale Anzahl an Tasks beschränkt werden. Die globale Variable `task` wird in einer `#pragma omp atomic` Direktive ausgeführt, weil mehrere Tasks gleichzeitig diese Variable inkrementieren können. Dadurch könnte während des Lesevorgangs die zu lesende Variable verändert werden und ein falscher Wert gelesen werden.
Wenn die maximale Anzahl an Tasks überschritten wird, wird die Methode sequentiell ausgeführt (Zeile 25). Der letzte Schritt ist es, die Dame von dem Feld zu entfernen (Zeile 26). Das wird benötigt, da jeder Methoden Aufruf `solve_nqueens` ein temporäres Array hat. Über dieses eine Array wird durch eine for Schleife iteriert und N Möglichkeiten einer Spaltenbelegung geprüft.

```
1  int tasks = 0;
2  int max_tasks = 200000;
3  void rec_nqueens_par_premble(bool queens[FIELDS][FIELDS]){
4  #pragma omp parallel
5  #pragma omp single
6          solve_queens(queens, 0);
7  }
8  void solve_queens(bool queens[FIELDS][FIELDS], int col)
9  {    bool new_board[FIELDS][FIELDS];
10     memcpy(new_board, queens, sizeof(new_board));
11     if (col == FIELDS){
12 #pragma omp atomic
13         nr_sol_par++; }
14     for (int i = 0; i < FIELDS; i++)
15     { if (check(queens, i, col))
16         { new_board[i][col] = 1;
17             if(tasks < max_tasks ) {
18 #pragma omp task
19                 {
20                     solve_queens(new_board, col + 1);
21 #pragma omp atomic
22                     tasks++;
23                 }
24             }
25             else { solve_queens(new_board, col + 1);}
26             new_board[i][col] = 0;
27         }
28     }
29 }
```

Listing 3.4: NQueens OpenMP mit einstellbaren Tasks

Die maximale Anzahl an Tasks wurde eingeführt, da durch die rekursive Art des Algorithmus bei hoher Eingabe N zu viele Tasks erstellt werden. Auf diese Weise benötigen die Tasks viel Speicherplatz und die Laufzeit verschlechtert sich, da die Erstellung eines Tasks Kosten verursacht. Ohne die Beschränkung, wie viele Tasks maximal existieren dürfen, funktioniert der Algorithmus. Allerdings mit einer schlechteren Laufzeit im Vergleich zur einstellbaren Variante, deshalb wurde die Möglichkeit realisiert die Anzahl an Tasks einzustellen.

Weitere Alternativen, um einen Algorithmus zu programmieren, der einstellbare Tasks hat, wäre es nach einer festgelegten Tiefe der Rekursion den Aufruf der Methode sequentiell ausführen zu lassen z.B. Listing 3.4 (Zeile 17)
`if(col < 13 )` zu ersetzen. In der Implementierung aus Listing 3.4 kann es vorkommen, dass die erstellten Tasks eine ungleiche Verteilung an Arbeit haben, weil die sequentielle Ausführung der Methode erst ab erreichen einer maximalen Anzahl an Tasks, stattfindet. Somit kann es sein, dass ein Task noch z.B. 20 Aufrufe der Methode `solve_queens` bekommt und ein anderer Tasks nur drei Aufrufe.

3.1.5 Cilk-F mit einstellbarer Anzahl an Futures

Die Idee von den benutzten Algorithmen für den Benchmark NQueens mit Cilk-F stammt von einem Mitarbeiter des MIT, welcher in die Entwicklung von Cilk involviert war [8]. Dieser Algorithmus für die Berechnung des NQueens-Problem wurde im Elternsystem Cilk umgesetzt. Der Algorithmus wurde aufgrund der intelligenten Umsetzung der Schachregeln und der sparsamen Benutzung von Speicherplatz genommen. Die Grundidee ist Teilberechnungen der Lösung durch Futures auszuführen, es soll eine einstellbare Anzahl an Futures erzeugt werden können, d.h. es existieren mehrere kleine Teilebrechungen, als in Listing 3.2, die zusammengefasst werden müssen.
Wenn die maximale Anzahl erreicht ist, soll die Methode sequentiell ausgeführt werden. Das Prinzip der Verwendung eines eindimensionalen Arrays und die Definition der Methode für die Überprüfung der Schachregel kann in Kapitel 3.1.1 nachgeschlagen werden.
Kurz Zusammengefasst: es wird ein eindimensionales `int` Array verwendet, indem die Spaltennummer, ein Element des Arrays, ausdrückt auf welcher Spalte eine Dame steht, die Zeile ist die Stelle des Arrays. Es wird geprüft, ob in dem Array die gleichen Spaltennummern existieren und ob sich die Damen diagonal

schlagen, wenn die Differenz der absoluten Zeile und Spalte gleich ist.
Um in dieser Implementierung in Cilk-F Futures zu benutzen, werden die Makros aus Kapitel 2.3.3 und die Hilfsfunktion aus Kapitel 2.3.4 zur Hilfe genommen. Die Hilfsfunktion mit dem Rückgabewert aus Listing 2.16 wird hier gebraucht, aus Platzgründen wurde sie nicht ins Listing 3.5 übernommen.
Das anschließend Beschriebene bezieht sich auf Listing 3.5. Die Methode `nqueens` wird in der `Main` aufgerufen und bekommt als Parameter ein zweidimensionales Array `queens` und den Anfangswert für die Spalte.
In der `nqueens` Methode wird ein Full Frame initialisiert (Zeile 16), dies erfolgt durch das Makro `CILK_FUNC_PREMBLE;`. Danach wird ein temporäres Array, ein Array um die Ergebnisse zu zählen und ein Array um den Future Index zu speichern (Zeilen 17-21) erstellt. Es wird überprüft, ob N Damen platziert worden sind (Zeile 22). Die Arrays `count` und `future_index` werden initialisiert und mit dem Wert Null befüllt (Zeilen 24-27).
Ein Eingabe N großes Future Array (Zeile 28) wird erstellt. Dieses Future Array wird an dieser Stelle erstellt, damit der Gültigkeitsbereich des Future Arrays für den Aufruf mit `.get()` (Zeile 50) verfügbar ist.
Der Wert `future_count` wird mit einer Null initialisiert, dieser Wert zählt die gestarteten Futures innerhalb eines Methodenaufrufes `nqueens`.
Nun wird über Zeilen iteriert und ein temporäres Array, um Damen zu setzen, initialisiert (Zeile 31) (die Größe des Arrays beträgt dabei Spalte + 1). Der Wert des `queens` Array aus dem Methodenaufruf wird in das temporäre Array
(Zeile 32) kopiert.
Eine Dame wird auf dem temporären Array `temp_queens` platziert (Zeile 33). Dann wird durch die `check` Methode geprüft, ob Konflikte mit der neu gesetzten Dame existieren. Die maximale Anzahl an Futures wird hier überprüft (Zeile 35), wenn die Anzahl unter einem Wert `max_future` liegt, dann wird ein Future gestartet. Die Iterationsnummer (Zeile 36) wird gespeichert, um im weiteren Verlauf zu wissen welches Future aus dem `future_array` gestartet wurde.

Der Wert `future_count` wird inkrementiert und zählt die gestarteten Futures innerhalb eines Methodenaufrufes `nqueens`.
Ein Task wird durch das Makro `START_FIRST_FUTURE_SPAWN;` (Zeile 39) gestartet und die Hilfsfunktion wird mit den benötigten Parametern aufgerufen (Zeile 40). Nach dem Aufruf der Hilfsfunktion folgt das Makro `END_FUTURE_SPAWN;` (Zeile 41). Wenn die maximale Anzahl an Futures überschritten worden ist, startet die Methode sequentiell (Zeile 44) und die zurückgegebenen Werte werden in dem Array `count` gespeichert (Zeilen 44 und 50). Während bei dem sequentiellen Aufruf der Wert direkt zurückgegeben wird, wird bei dem Aufruf der Funktion `nqueens` durch ein Future gewartet, bis das Future den Wert zurückliefern kann.
Deshalb war es vorher notwendig die gestarteten Futures zu zählen `future_count` und den Index der Futures, die im Future Array `future_array` gestartet wurden, zu speichern `future_index`. Am Ende wird mit einer Schleife nur über die gestarteten Futures iteriert (Zeile 49) und der Wert durch ein `.get()` angefordert (Zeile 50). Das Array `count` erhält die nötigen Rückgabewerte, diese werden in einer Variablen `sol_num` (Zeilen 53-55) addiert und zurückgegeben (Zeile 58).
Zum Schluss der Funktion erfolgt die Benutzung des Makros `CILK_FUNC_EPILOGE;` (Zeile 57), um den Full Frame zu löschen. Wenn die Futures nicht mehr genutzt werden, können sie mit `delete [] futures;` gelöscht werden.

```
1  int max_future = 200000;
2  int futures = 0;
3  int check (int actual_col, int *queens) {
4      int i, j;
5      int p, q;
6      for (i = 0; i < actual_col; i++) {
7          p = queens[i];
8          for (j = i + 1; j < actual_col; j++) {
9              q = queens[j];
```

```
10              if (q == p || q == p - (j - i) || q == p + (j - i))
11                  return 0;
12          }
13      } return 1;
14  }
15  int nqueens (int col, int *queens) {
16      CILK_FUNC_PREAMBLE;
17      int *temp_queens;
18      int i;
19      int *count;
20      int *future_index;
21      int solNum = 0;
22      if (FIELDS == col) {
23          return 1;}
24      count = (int * ) alloca(FIELDS * sizeof(int));
25      (void) memset(count, 0, FIELDS * sizeof (int));
26      future_index = (int *) alloca(FIELDS * sizeof(int));
27      (void) memset(future_index, 0, FIELDS * sizeof (int));
28      cilk::future<int> * future_array = new cilk::future<int>[FIELDS];
29      int future_count = 0;
30      for (i = 0; i < FIELDS; i++) {
31          temp_queens = (int * ) alloca((col + 1) * sizeof (int));
32          memcpy(temp_queens, queens, col * sizeof (int));
33          temp_queens[col] = i;
34          if(check (col + 1, temp_queens)) {
35              if(col < max_future) {
36                  future_index[future_count] = i;
37                  future_count ++;
38                  futures++;
39                  START_FIRST_FUTURE_SPAWN;
40                  function_helper(&future_array[i], col + 1, temp_queens);
41                  END_FUTURE_SPAWN;
42              }
43              else {
44                  count[i] = nqueens(col + 1, temp_queens);
45              }
46          }
47      }
48    if(future_count > 0) {
49        for(int k =0; k < future_count; k++) {
50            count[k] = future_array[future_index[k]].get();
51        }
52    }
```

```
53      for(i = 0; i < FIELDS; i++) {
54          solNum += count[i];
55      }
56      delete [] future_array;
57      CILK_FUNC_EPILOGUE;
58      return solNum;
59  }
```

Listing 3.5: NQueens Cilk-F einstellbare Futures

Es kann eine andere Möglichkeit für eine Lösung gewählt werden: Listing 3.6. Die Futures können direkt in der `Main` erstellt werden und das Warten auf den zu berechnenden Wert kann in der `Main` erfolgen. Allerdings wird der Verwaltungsaufwand der Futures dadurch höher. Da es nötig ist, zu wissen welches Future welchen Teilbereich berechnet hat und welcher Teilbereich durch die sequentielle Ausführung berechnet worden ist. Dadurch wird der Aufwand bei der Programmierung erhöht.

```
1  int max_futures = 200000;
2  int main()
3  {
4    cilk::future<int> *farray = new cilk::future<int>[max_futures];
5    nqueens (0, queens, farray);
6    for(int i =0; i < future_count; i++) {
7            farray[k].get();
8    }
9  }
10 int nqueens (int col, int *queens, c cilk::future<void> *farray) {
11    // berechnung
12 }
```

Listing 3.6: NQueens Cilk-F einstellbare Futures

Eine Implementierung mit einem Algorithmus aus dem Elternsystem Cilk zu erstellen ist vorteilhaft, da es dadurch klarer wird an welcher Stelle die Futures verwendet werden sollen. Allerdings kann nicht einfach `cilk_spawn` in Cilk durch `fut-create` und `cilk_sync` durch `fut-get` ersetzen werden. Es muss zusätzlicher

Aufwand betrieben werden, um genau zu wissen welches Future durch ein `.get()` angesprochen wird.

3.2 Stencil

Stencil, beschreibt eine Berechnung, bei der es darum geht auf einem Spielfeld mit Zeilen und Spalten zu erfahren, ob eine Zelle tot oder lebendig ist. Je nach Nachbarschaft bleiben die Zellen am Leben, sterben oder bringen neues Leben hervor. Bei der nachfolgenden Implementierung geht es um die Berechnung der Zellen aus den Nachbarn. Als Eingabeparameter werden die Spielfeldgrößen mit der Größe N * N Zellen und Zeit `t` mitgegeben.
Die Zeit wird als eine Iteration der Berechnung des kompletten Feldes definiert. In dem sequentiellen Algorithmus existieren drei Schleifen. Schleifen für die Zeile, Spalte und die Zeit (Listing 3.7). Eine Zelle wird als `double` Wert angesehen, dies dient nur der Lasterzeugung und eine Zelle könnte ein `bool` Wert sein. Eine Zelle ist am Leben, wenn sie einen Wert besitzt, der größer als null ist.

```
1 for (int t = 0; t < T_MAX; ++t) {
2     for (int i = 1; i < FIELDS - 1; i++) {
3         for (int j = 1; j < FIELDS - 1; j++) {
4             A[i][j] = berechne_neuen_zustand(A, i, j);
5         }
6     }
7 }
```

Listing 3.7: Stencil Schleifen sequentiell

Um zu berechnen, ob eine Zelle tot oder lebendig ist, wird die Methode `anzahl_nachbarn` (Listing 3.8) (Zeile 2) aufgerufen mit den Parametern: das Spielfeld und die Position der Zelle durch `i` Zeile, `j` Spalte. In der Methode `anzahl_nachbarn` Listing 3.9 werden die `double` Werte der Nachbarn oben, unten, links, rechts und der Wert der Zelle addiert (Zeilen 4-8) und anschließend durch fünf geteilt (Zeile 9). Dieser errechnete Wert wird zurückgegeben und in

der Variablen `check_value` in Listing 3.8 (Zeile 2) für den weiteren Verlauf zur Verfügung gestellt.

Nun wird in Listing 3.8 entschieden, welchen Wert der neue Zustand einer Zelle annimmt. Dabei gibt es drei Möglichkeiten: der Wert `check_value` ist kleiner als 0.5 (Zeile 3), der neue Wert ist drei. Der Wert `check_value` ist größer 0.5 und kleiner als 6 (Zeile 5), der neue Wert ist `check_value` oder der Wert `check_value` ist größer als 6 (Zeile 7), so wird der neue Wert zu einer Null.

Die `if` Abfragen wurden so gewählt, damit eine Lasterzeugung für alle Tasks gewährleistet ist. Zudem werden auf den Spielfeld Randbereiche definiert, die nicht berechnet werden, jedoch für die Berechnung einer Zelle notwendig sind, da der linke und obere Nachbar der Zelle oben links existieren muss. Die Anzahl der Felder steigt auf Eingabe Felder + 2, wie in Abbildung 3.6 mit der Spielfeldgröße 4 zu sehen ist.

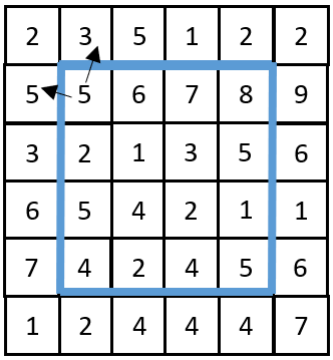

Abbildung 3.6: Randbereiche

```
1  double berechne_neuen_zustand(double **A, int i, int j){
2      double check_value =  anzahl_nachbarn(A,i,j);
3      if( check_value < 0.5 ) {
4          return 3; }
5      if (check_value < 6) {
6          return check_value; }
7      if (check_value >= 6) {
8          return 0; }
9      return 0;
10 }
```

Listing 3.8: Stencil Berechnung tot oder lebendig

```
1  double anzahl_nachbarn(double **A, int i, int j){
2      double summe = 0;
3      int div_value = 5;
4      summe += A[i][j];
5      summe += A[i-1][j];
6      summe += A[i][j+1];
7      summe += A[i+1][j];
8      summe += A[i][j-1];
9      return summe / div_value;
10 }
```

Listing 3.9: Stencil Berechnung Zelle

Für die parallele Implementierung wurden zwei Arten betrachtet, einmal ohne und einmal mit Zeitschleife.

3.2.1 Stencil OpenMP ohne Zeitschleife

Bei der parallelen Implementierung besteht die Schwierigkeit darin, das Feld so zu teilen, dass jeder Task nicht zu viel Arbeit erhält und dass die Abhängigkeiten zwischen den geteilten Feldern korrekt erfolgen.
Die Aufgabe sollte durch Tiling mit paarweiser Synchronisation gelöst werden, somit ist sichergestellt, dass der Algorithmus effizient arbeitet. Tiling bedeutet eine Aufteilung des Spielfeldes in mehrere gleiche Bereiche, dadurch kann

davon ausgegangen werden, dass ein jeder Task nicht zu viel Arbeit erhält. Dementsprechend wird bei der parallelen Version von Stencil als Parameter zusätzlich ein Tiling Wert mitgegeben, somit wird das Problem, die Felder korrekt zu teilen, gelöst.

Durch die Nutzung von Tasks kann der Programmieraufwand reduziert werden, im Gegensatz zur Benutzung von Threads. Tasks bieten die Möglichkeit Abhängigkeiten/Synchronisation (`depend`) übersichtlicher und effizienter zu definieren als Threads durch `busy Waiting`.

In Abbildung 3.7 werden die Abhängigkeiten der Tasks verdeutlicht. Die Größe der möglichen Felder wird so beschränkt, sodass nur Zweierpotenzen als Eingabe für die Felder ohne Rand möglich sind. Dies wird geprüft und bei Bedarf inkrementiert. Des Weiteren muss das Tiling Value durch die Größe der Felder teilbar sein z.B. 1024 / 128 = 8 entspricht Feldgröße/Tiling. Das Ergebnis wird mit Zwei potenziert z.B. 8^2, dadurch ergeben sich die gestarteten Tasks, die das Spielfeld berechnen.

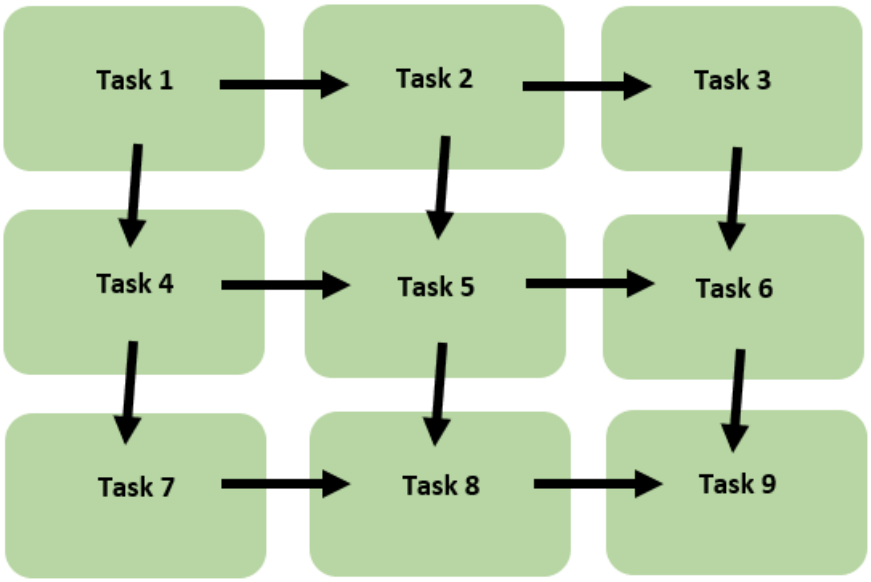

Abbildung 3.7: Task Abhängigkeiten

Die Methode `iteration_stencil_par` Listing 3.10 (Zeile 1) wird in der `Main` aufgerufen. Der nachfolgenden Text bezieht sich auf Listing 3.10.
Ein paralleler Bereich mit `#pragma omp parallel` und `#pragma omp single` wird gestartet (Zeilen 2 und 4). Innerhalb dieses parallelen Bereichs werden, durch zwei Schleifen (Zeilen 6 und 7) die Bereiche für die Berechnung der Tasks festgelegt. Nachdem die Tasks ihre Bereiche für die Berechnung kennen, werden sie durch `#pragma omp task` (Zeile 8) gestartet.
In OpenMP können die Abhängigkeit sowie die Reihenfolgebedingungen zwischen Tasks mit der `depend` Klausel festgelegt werden `#pragma omp task depend (dependency-typ: list)`. Folglich können die Abhängigkeiten als Bereiche definiert werden. Das anfangs `inout` (Zeile 9) definiert, dass ein Task Lese- und Schreibzugriff auf den Bereich des Feldes `A[i:tiling_value][j:tiling_value]` haben muss.
Dann folgt das `in`, was bedeutet, dass der Task auf den Bereich `A[i-tiling_value:tiling_value][j:tiling_value]` (Zeile 11) und `A[i:tiling_value][j-tiling_value:tiling_value]` (Zeile 12) Lesezugriffe haben muss. Der Lesezugriff kann erfolgen, wenn der Task mit dem Schreibzugriff seinen Wert berechnet hat.
Anschließend entsteht eine Ausführungsreihenfolge der Tasks. Diese Reihenfolge wird eingehalten, dennoch können die Tasks optimal auf die verfügbaren Prozessoren/Threads verteilt werden.
Die zu berechnenden Zellen für die Tasks werden durch zwei weitere Schleifen definiert (Zeilen 14 und 15).

```
1  void iteration_stencil_par( double **A) {
2  #pragma omp parallel
3      {
4  #pragma omp single
5      {
6       for (int i = 1 ; i < FIELDS - 1; i += tiling_value) {
7         for (int j = 1 ; j < FIELDS - 1; j += tiling_value) {
8  #pragma omp task depend\
9  (inout: A[i:tiling_value][j:tiling_value]) \
10 depend\
11 (in: A[i-tiling_value:tiling_value][j:tiling_value] , \ // oben
12     A[i:tiling_value][j-tiling_value:tiling_value] )   // links
13          {
14              for (int ii = i; ii < (tiling_value + i); ++ii) {
15                 for (int jj = j; jj < (tiling_value + j); ++jj) {
16                    A[ii][jj] = berechne_neuen_zustand(A, ii, jj);
17                 }
18              }
19          }
20 }}}}}
```

Listing 3.10: Stencil OpenMP ohne Zeitschleife

Durch das Konstrukt der Tasks und `depend` Klausel konnte die Implementierung des Stencil-Benchmarks in OpenMP übersichtlich erfolgen.

3.2.2 Cilk-F ohne Zeitschleife

Die Basisidee war es, durch die Benutzung von Futures die gleichen Abhängigkeiten zu schaffen, wie bei der Implementierung mit OpenMP. Auf Grund der Notwendigkeit, um die Abhängigkeiten zwischen mehreren Futures zu schaffen und die dadurch verbundene Nutzung eines Futures Array, werden die Makros Kapitel 2.3.3 und die Hilfsfunktion ohne Rückgabewert aus Kapitel 2.3.4 benötigt. Es reicht nicht aus nur ein Future zu haben, durch `cilk_future_create_stack` um mehrere Abhängigkeiten zu definieren.
Die Futures können, wie bei der Implementierung in OpenMP als Tasks angesehen und verwendet werden, damit die nötigen Abhängigkeiten geschaffen werden

können.

Die Funktion `iteration_stencil_par` wird in der `Main` aufgerufen.

Folgender Text nimmt Bezug auf Listing 3.11. Beim Aufruf der Funktion `iteration_stencil_par` wird die genaue Anzahl an Futures errechnet (Zeilen 2-4), diese berechnet sich aus

$$Anzahl\_Futures = (n/tiling\_value)^2 \tag{3.1}$$

Durch die Randfelder muss die Zahl mit Zwei subtrahiert werden (Zeile 2). Im Anschluss daran wird ein Futurezähler `future_count` (Zeile 6) und ein Future Array (Zeile 7), mit der vorher errechneten Größe, initialisiert. Dieser Zähler wird benötigt, um auf das zu benutzende Future zuzugreifen. Durch zwei Schleifen folgt die Aufteilung des Spielfeldes in Bereiche (Zeilen 8 und 9). Der Aufruf der Hilfsfunktion findet statt (Zeile 11).

Als zusätzliche Parameter werden in der Hilfsfunktion das Future Array `farray`, die Nummer des aktuellen Future `future_count` und der Wert `n/tiling power_value` mitgegeben. In der Hilfsfunktion wird die `spawn_function` (Zeile 23) mit den Parametern: `i` Zeile, `j` Spalte, das Feld `A`, das Future Array `farray`, die aktuelle Futurenummer `future_count` und der Wert `n/tiling power_value` aufgerufen.

Die ersten Abhängigkeiten werden erstellt, wenn `i` und `j` größer als Eins sind, da die erste Zeile keine Abhängigkeit von oben hat und die erste Spalte keine Abhängigkeit von links. In der `spawn_function` werden die Abhängigkeiten erzeugt, indem auf dem übergebenen Future Array `farray` auf ein Future des Arrays mittels `.get()` (Zeilen 32 und 34) aufgerufen wird. Das benötigte Future, welches durch das `.get()` für die Abhängigkeit oben angesprochen wird, wird aus dem Array ermittelt mit: `future_count - power_value` (Zeile 32). Für die Abhängigkeit links wird eine Eins vom aktuellen Wert der Futurenummer `future_count` subtrahiert (Zeile 34). Somit bekommt man den linken Nachbarn des Futures. Nach Erzeugung der Abhängigkeiten werden die Zellen berechnet.

```
1  void iteration_stencil_par(double **A) {
2      int f = FIELDS - 2;
3      int power_value = f / tiling_value;
4      int number_futures = (int) pow(power_value, (double) 2);
5      CILK_FUNC_PREAMBLE;
6      int future_count = 0;
7      cilk::future<void> * farray = new cilk::future<void>[number_futures];
8          for (int i = 1; i < FIELDS - 1; i += tiling_value) {
9              for (int j = 1; j < FIELDS - 1; j += tiling_value) {
10                 START_FIRST_FUTURE_SPAWN;
11                 function_helper(&farray[future_count], farray, i, j, A,
                       future_count, power_value);
12                 END_FUTURE_SPAWN;
13                 future_count++;
14             }
15         }
16         future_count = 0;
17     delete [] farray;
18     CILK_FUNC_EPILOGUE;
19 }
20 void __attribute__((noinline)) function_helper(cilk::future<void> *fut,\
21 cilk::future<void> *farray, int i, int j, double **A,int future_count, \
       int power_value) {
22     FUTURE_HELPER_PREAMBLE;
23     spawn_function(i, j, A, farray, future_count, power_value);
24     __asm__ volatile ("" ::: "memory");
25     void pointer __cilkrts_deque = fut->put();
26     if (__cilkrts_deque) __cilkrts_resume_suspended(__cilkrts_deque, 2);
27     FUTURE_HELPER_EPILOGUE;
28 }
29 void spawn_function(int i, int j, double **A, \
30 cilk::future<void> *farray, int future_count, int power_value){
31     if(i > 1) {
32         farray[future_count - power_value].get();}// dependency up
33     if(j > 1) {
34         farray[future_count - 1].get();}// dependency left
35     for (int ii = i; ii < (tiling_value + i); ++ii){
36         for (int jj = j; jj < (tiling_value + j); ++jj) {
37             A[ii][jj] = berechne_neuen_zustand(A, ii, jj);}}}
```

Listing 3.11: Stencil Cilk-F ohne Zeitschleife

3.2.3 Stencil OpenMP mit Zeitschleife

Bei der Programmierung mit OpenMP mit einer zusätzlichen Zeitschleife (Listing 3.12) konnte die Implementierung aus Abschnitt 3.2.1 genutzt werden. Ein Unterschied ist, dass ein Parameter als Zeit eingegeben werden kann. Demnach iteriert eine Zeitschleife über den Algorithmus (Zeile 7). Des Weiteren wird der erste Task (oben links) vor allen anderen Tasks bei jeder Iteration der Zeitschleife gestartet, um sicherzustellen, dass er wirklich zu Beginn einer jeden Zeititeration gestartet wird. Bei den übrigen Tasks, die anschließend gestartet werden, wird durch ein If-Statement (Zeile 17) sichergestellt, dass der erste Tasks nicht nochmal gestartet wird und eine zusätzliche `in` Abhängigkeit für den ersten Task wird in den restlichen Tasks definiert `A[1:tiling_value][1:tiling_value])` (Zeile 23).
Das Problem, welches entstehen könnte, wenn der erste Task nicht zuerst gestartet wird, ist, dass durch die Iterationen der Zeitschleife die Abhängigkeit verloren gehen könnte, da die letzten Tasks mit den ersten der nächsten Zeitschleifeniteration verbunden sind.

```
1  void iteration_stencil_par( double **A) {
2  #pragma omp parallel
3      {
4  #pragma omp single
5          {
6              int i, j, ii, jj, t;
7              for (t = 0; t < T_MAX; ++t) {
8  #pragma omp task depend (inout: A[1:tiling_value][1:tiling_value],t)
9                      { for (int i = 1; i < tiling_value + 1; ++i) {
10                             for (int j = 1; j < tiling_value + 1; ++j) {
11                                 A[i][j] = berechne_neuen_zustand(A, i, j);
12                             }
13                         }
14                     }
15             for (i = 1 ; i < FIELDS - 1; i += tiling_value) {
16                 for (j = 1 ; j < FIELDS - 1; j += tiling_value) {
17                     if(!(i == 1 && j == 1)) {
18 #pragma omp task depend\
19 (inout: A[i:tiling_value][j:tiling_value]) \
20 depend\
21 (in: A[i-tiling_value:tiling_value][j:tiling_value] /* oben */, \
22      A[i:tiling_value][j-tiling_value:tiling_value] /* links */, \
23      A[1:tiling_value][1:tiling_value])             /* first task */
24                 {
25                 for (ii = i; ii < (tiling_value + i); ++ii) {
26                     for (jj = j; jj < (tiling_value + j); ++jj) {
27                        A[ii][jj] = berechne_neuen_zustand(A, ii, jj);
28                     }
29                 }}}}}}}}}
```

Listing 3.12: Stencil OpenMp mit Zeitschleife

3.2.4 Cilk-F mit Zeitschleife

In Cilk-F wurde der Algorithmus aus Abschnitt 3.2.2 verwendet. Die Grundidee ist dabei die Gleiche wie in Abschnitt 3.2.2 beschrieben. Die Futures werden als Tasks gestartet und die nötigen Abhängigkeiten werden in der Berechnung der Felder des Spielfeldes geschaffen. Unterschiedlich ist, dass eine Zeitschleife über den Algorithmus iteriert. Das führt dazu, dass die Stelle der Initialisierung

und die Anzahl der Futures in einem Array durchdacht werden sollte. Der Gültigkeitsbereich des Future Arrays muss an die Synchronisation angepasst werden.

Die erste Möglichkeit ist es, ein Future Array mit der Größe der Berechnung (3.1) aus Abschnitt 3.2.2 vor der Zeitschleife zu initialisieren. Nachstehendes bezieht sich auf Listing 3.13. Das Future Array wird vor der Zeitschleife initialisiert (Zeile 7). Dann werden die Futures mit der Hilfsfunktion gestartet und die Zellen wie in 3.2.2 berechnet. Zum Schluss der Iteration über die Zeit wird auf das letzte Future gewartet (Zeile 10), indem ein `.get()` Aufruf auf dem letzten Future erfolgt, um sicherzustellen, dass alle Futures ihre Berechnungen abgeschlossen haben. Mit der nächsten Iteration über die Zeitschleife wird gewartet, weil die Anzahl an Futures, die gestartet werden können, auf eine Zeititeration begrenzt sind.

Eine andere Möglichkeit wird in einer Alternativen im weiteren Verlauf erklärt. Es existiert eine Methode, die Futures nach Beendigung einer Iteration zurückzusetzen (Zeile 12) anstatt sie zu löschen. Dadurch kann das Future Array außerhalb der Zeitschleife angelegt werden. Somit spart man sich das häufige Anlegen und Löschen eines Future Arrays in der Zeitschleife.

```
1  void iteration_stencil_par(double **A) {
2      int f = FIELDS - 2;
3      int power_value = f / tiling_value;
4      int number_futures = (int) pow(power_value, (double) 2);
5      CILK_FUNC_PREAMBLE;
6      int future_count = 0;
7      cilk::future<void> farray = new cilk::future<void>[number_futures];
8      for (int t = 0; t < T_MAX; ++t) {
9        // Algorithmus aus 3.2.2 (Zeilen 8 - 15)
10         farray[future_count - 1].get();
11         for(int k = 0; k < future_count; k++) {
12           farray[k].reset();
13         }
14     }
15     CILK_FUNC_EPILOGUE;
16 }
```

Listing 3.13: Stencil Cilk-F mit Zeitschleife

Eine alternative Möglichkeit wäre die Definition der Futures innerhalb der Zeitschleife und nach Beendigung eines Durchlaufes zu löschen. Diese Möglichkeit bietet dieselbe Laufzeit, wie die Implementierung in Listing 3.13, sonst aber keine weiteren Vorteile. Eine weitere Alternative wäre es die Anzahl an Futures mit `number_futures * T_MAX` zu definieren, für jede Zeititeration eigene Futures. Dadurch könnten alle verfügbaren Futures in der Zeitschleife gestartet werden und müssten nicht auf eine Zeititeration warten. Diese Methode führt bei sehr großen Werten zur Überlastung des Speichers. Auf dem getesteten System tritt die Überlastung bei einem Wert von ungefähr 16 Milliarden Futures auf, um dieses Problem zu umgehen, könnte ein Maximalwert für die Anzahl der Futures implementiert werden.

```
1  cilk::future<void> farray = new cilk::future<void>[number_futures * T_MAX];
2      for (int t = 1; t <= T_MAX; ++t) {/* Algorithmus aus 3.2.2 */}
```

Listing 3.14: Stencil Cilk-F mit Zeitschleife futures * T_MAX

4 Vergleich der Benutzbarkeit

Die beiden Programmierumgebungen OpenMP und Cilk-F verwenden die Sprache C++ als Basis. Ein Vergleich der Benutzbarkeit ist nur sehr bedingt möglich, da Cilk-F sich im Prototypen Status befindet und OpenMP eine stabile Version aufweist. Außerdem werden in OpenMP Updates und Verbesserungen durchgeführt. Des Weiteren kommt hinzu, dass OpenMP viel mehr Sprachkonstrukte als Cilk-F besitzt, was aber keinen Vorteil darstellen muss.
Der Vergleich der Benutzbarkeit von der Handhabung der Futures gegenüber der Handhabung von Threads/Tasks lässt sich dennoch durchführen. Der Vergleich der Benutzbarkeit der beiden Programmierumgebungen gliedert sich in zwei Unterkapitel:
In Abschnitt 4.1 werden eigene Erfahrungen, die bei der Installation von Cilk-F und OpenMP gemacht wurden, erklärt. Durch Abschnitt 4.2 werden Nutzererfahrungen, die mit der Programmierumgebung von Cilk-F und OpenMP gemacht wurden, beschrieben. In dem anschließenden Abschnitt 4.3 wird auf die Benutzbarkeit eingegangen, eigene Erfahrungen, die gemacht wurden, während der Implementierung der Benchmarks und wie natürlich es ist, aus Nutzersicht eines Programmierers, einen Algorithmus mit den Konzepten der Futures gegenüber den Konzepten aus OpenMP zu schreiben.

4.1 Einrichtung der Programmierumgebung

OpenMP wird unter anderem von dem Compiler von Intel, Microsoft und GCC unterstützt, GCC ist eine Sammlung von Compilern und bietet eine einheitliche Schnittstelle zur Erstellung von Programmen in C/C++. Somit konnte unter dem benutzen System Microsoft Windows 10 eine IDE installiert und OpenMP

genutzt werden. Eine integrierte Entwicklungsumgebung (IDE) ist ein Programm, mit dem Software entwickelt werden kann, vorteilhaft ist, dass Korrekturen des Quellcodes und Vorschläge existieren und somit das Programmieren erleichtert wird.
Cilk-F wurde in einer Docker Umgebung auf den NV Fachgebietsrechnern des FB 16 (Programmiersprachen/-methodik) zur Verfügung gestellt. Eine Docker Umgebung legt Programme samt ihrer Abhängigkeiten in Images ab. Diese Images sind auf verschiedenen Betriebssystem lauffähig [19].
Durch die Docker Umgebung wurde ein funktionierender Compiler für
Cilk-F bereitgestellt, außerdem wurden die implementierten Benchmarks auf der Docker Umgebung kompiliert und ausgeführt. Auf dem verwendeten System Windows 10 konnte der Docker Container mit vertretbarem Aufwand nicht zum Laufen gebracht werden. Eine IDE war für die Programmierung mit Cilk-F nicht vorhanden. Die implementierten Benchmarks wurden in einem GIT-Repository hinterlegt und sind unter der Docker Umgebung von Cilk-F zu finden.
In OpenMP war es möglich in wenigen Minuten mit dem Programmieren zu beginnen, die Unterstützung einer IDE war von Vorteil, da die Fehleranzahl durch Syntaxfehler geringer war als mit Cilk-F. Mit Cilk-F war der Zeitaufwand bei der Einrichtung höher als in OpenMP, jedoch durch die schon installierte Docker Umgebung war der Zeitaufwand angemessen und mit grundlegendem Verständnis von Docker nicht kompliziert.

4.2 Nutzererfahrung mit Programmierumgebung Cilk-F/OpenMP

Das Erlernen der Programmierung mit OpenMP war einfacherer, da bereits Grundlagen vorhanden waren, aber auch Konstrukte, in denen keine Grundlagen vorhanden waren (Task), konnten mit vertretbarem Aufwand erlernt werden. In Cilk-F war das Erlernen der Programmierung mit deutlich höherem Zeitaufwand

verbunden, da zum einen keine Vorkenntnisse vorhanden waren und zum anderen diese Programmierumgebung durch Ausprobieren erlernt wurde. Es wurden Vorkenntnisse vom Programmierer im Elternsystem Cilk vorausgesetzt, diese waren nicht verfügbar und mussten recherchiert werden. Zu diesen Vorkenntnissen gehörten: Was passiert mit dem Cilk Code beim Kompilieren?, Wie sieht die erstellte Struktur aus?, Was macht die Hilfsfunktion?, Wie funktioniert das Cilk Runtime System?

OpenMP liefert viele Beispiele, Implementierungen und eine Dokumentation, in der bei Problemen nachgeschlagen werden kann, zudem gibt es viel Literatur und viele Beispiele durch die weite Verbreitung von OpenMP. Für Cilk-F existiert nur wenig Dokumentation, da es nur für einen akademischen Rahmen entwickelt wurde [1]. Für Cilk-F sind verschiedene Benchmarks vorhanden, die von den Entwicklern implementiert worden sind, so gibt es einige Beispiele zur Implementierungen. Diese Beispielimplementierungen sind teilweise schwierig zu verstehen, weil sie nur ungenügend kommentiert sind und Kenntnisse in Cilk voraussetzen. Diese Kenntnisse in Cilk können jedoch mit Zeitaufwand recherchiert werden.

Die Fehlermeldungen in Cilk-F sind unverständlich, der Programmierer weiß nicht welchen Fehler er gemacht hat. Außerdem ist die Struktur des Algorithmus in OpenMP übersichtlicher als in Cilk-F, da keine Makros und Hilfsfunktionen implementiert werden müssen.

Cilk-F bietet keinen Schutz bei verschachtelter Parallelität, wenn zu viele Futures erstellt werden, das Programm terminiert ohne Ergebnis. Bei dem getesteten System konnte bei der Anzahl von circa 16 Milliarden Futures eine Grenze ermittelt werden. OpenMP bietet den Schutz, die Laufzeit wird bei zu vielen Tasks schlechter. Der Nachteil beim Arbeiten mit Cilk-F ist, dass man die Verwaltung der Futures übernehmen muss, also wissen muss welches Future gestartet wird und wie viele Futures gestartet werden müssen. In OpenMP muss dieser Aufwand der Verwaltung der Tasks nicht übernommen werden.

4.3 Benutzbarkeit/Diskussion

Es konnten korrekte Algorithmen mit OpenMp und Cilk-F implementiert werden. Ein störender Faktor bei der Programmierung in Cilk-F ist der Entwicklungsstand, dadurch muss der Programmierer die Aufgaben des Compiler übernehmen. Zudem muss der Programmierer erstmal wissen, was diese Aufgaben sind.

Dieser Mehraufwand spiegelt sich im Aufwand der Implementierung wieder, da eine Aufgabe mit Cilk-F nicht fertiggestellt werden konnte, und zwar Algorithmen zu implementieren, die von Futures profitieren.

Die Idee mit Futures zu programmieren, um Parallelität zu Programmieren ist vom Grund auf gut, als Programmierer hat man intuitiv viele Ideen, um schnelle Programme mit Futures zu verwirklichen. Die Umsetzung dieser Ideen wird von Cilk-F in dem aktuellen Stand nicht unterstützt, wie am Beispiel von NQueens mit einstellbarer Anzahl an Futures (Abschnitt 3.1.4.) bei der Laufzeitmessung im nächsten Kapitel, Kapitel 5, festgestellt worden ist. Die Nutzung der Futures wurde in der Implementierung NQueens mit einstellbarer Anzahl an Futures, nach den Beispielen in den mitgelieferten Benchmarks, versucht bestmöglich umzusetzen, dennoch konnte kein performantes Programm im Vergleich zur Implementierung mit OpenMP erzielt werden. Um dennoch ein performantes Programm in Cilk-F schreiben zu können, wäre es nötig gewesen, noch mehr Wissen in der Laufzeit Umgebung von Cilk aufzubauen, was mit Zeitaufwand verbunden wäre, der nicht mehr zur Verfügung stand.

Ein Nachteil der Futures ist es, dass man genau wissen muss wie viele Futures erzeugt werden müssen und welche Futures aktuell mit welcher Arbeit beschäftigt sind, damit man einen Wert abfragen kann. Das heißt, die Verwaltung der Futures muss vom Programmierer übernommen werden. In OpenMP muss man nicht exakt wissen wie viele Tasks existieren und die Verwaltung der Tasks muss nicht übernommen werden. Dennoch muss man in OpenMP bei der Verwendung von

Task `dependencies` wissen, welcher Task welchen Bereich abzuarbeiten hat.
Die Implementierung des Stencil Benchmark war mit Cilk-F vorteilhafter als mit OpenMP. Es erforderte weniger Abstraktion auf die Beendigung von Futures zu warten, als die Teilbereiche in dem Task `dependencies` zu definieren. Außerdem ist die Implementierung mit der zusätzlichen Zeitschleife in Cilk-F einfacher gewesen, weil man die Anzahl der Futures nur erhöhen muss, um ein funktionierendes Ergebnis zu erhalten.
Zusammenfassend gesagt: das Konzept und die Handhabung der Futures ist gut und intuitiv, allerdings kann durch den Entwicklungsstand von Cilk-F nicht klar gesagt werden, ob Futures eine Einschränkung mitbringen oder die Programmierumgebung Cilk-F eingeschränkt ist.
Bei der Verwendung von Rekursion, wenn der Programmierer nicht weiß, wie viele Futures erstellt werden, ist die Nutzung der Konzepte aus OpenMP vorteilhafter als die Nutzung von Futures aus Cilk-F.
Wenn man die Anzahl der Futures kennt und die Verwaltung der Futures einfach ist, wie z.B. in Benchmark Stencil, ist es vorteilhafter die Konzepte aus Cilk-F zu nutzen, als die aus OpenMP. Es ist leichter zu programmieren, dass auf einen Bereich gewartet wird, als diesen Bereich zu definieren. Die Programmierung mit Cilk-F ist nicht trivial, kann aber mit dem nötigen Wissen umgesetzt werden.

5 Vergleich der Performance

5.1 Testumgebung

Alle Messungen wurden auf den NV Fachgebietsrechnern des FB 16 (Programmiersprachen/-methodik) durchgeführt, diese Rechner enthalten 8 Intel Core i7 CPU mit 2.67 GHz [17] Prozessoren. Die Messungen für Cilk-F und OpenMP wurden unter der Docker Umgebung, die mit Cilk-F zur Verfügung gestellt wurde, durchgeführt. Die Docker Umgebung wurde für OpenMP benutzt, um einen besseren Vergleich bezüglich der Effizienz zwischen den Programmierumgebungen zu ermöglichen. Diese Docker Umgebung hat die GCC Version 5.4.0 (Release 03 Juni 2016). Die OpenMP Benchmarks wurden mit dieser Version kompiliert.

Es wurden zwei Benchmarks ausgeführt, wobei in den Benchmarks von NQueens zwei verschiedene Versionen existieren und jeder Benchmark mit Versionen fünfmal ausgeführt wurde. Aus diesen fünf Aufrufen wurde der Durchschnitt ermittelt.

Die Zeitmessung wurde in der `Main` Methode durchgeführt, um den ersten Aufruf einer Methode, die die Parallelität startet.

Der Parameter für die Anzahl an Tasks/Futures in NQueens für OpenMP und Cilk-F wurde so gewählt, dass die Laufzeit am Besten ist. Der Tiling Wert in Stencil, in OpenMP und Cilk-F wurde so gewählt, dass die Laufzeit optimal ist. Der Parameter `f` in den NQueens Benchmarks bezeichnet dabei die maximal mögliche Anzahl an Tasks/Futures. Die Parameter in den Stencil Benchmark bezeichnen dabei mit `v` den Tiling Wert und `t` als Zeit.

Die Benchmarks sind nachstehend beschrieben.

- NQueens
  - `nqueens_seq` NQueens sequentiell Listing 3.3.
  - `nqueens_openmp` NQueens Listing 3.1.
  - `nqueens_cilkf` NQueens Listing 3.2.
  - `nqueens_adj_openmp` Einstellbare Tasks mit OpenMP Listing 3.4.
  - `nqueens_adj_cilkf` Einstellbare Futures mit Cilk-F Listing 3.5.
- Stencil
  - `stencil_seq` Stencil sequentiell Listing 3.8.
  - `stencil_openmp` Stencil mit Zeitschleife OpenMP Listing 3.13.
  - `stencil_cilkf` Stencil mit Zeitschleife Cilk-F Listing 3.14.

5.2 Ergebnisse und Diskussion

Benchmark	Laufzeit	Parameter
nqueens_seq	57.619ms	$N = 15$
	425.298ms	$N = 16$
nqueens_openmp	8.797ms	$N = 15$
	65.363ms	$N = 16$
nqueens_cilkf	13.201ms	$N = 15$
	94.176ms	$N = 16$
nqueens_adj_openmp	11.668ms	$N = 15$, $f = 1000000$
	80.928ms	$N = 16$, $f = 300000000$
nqueens_adj_cilkf	81.276ms	$N = 15$, $f = 1000000$
	657.204ms	$N = 16$, $f = 2100000000$

Tabelle 5.1: NQueens Benchmark

Benchmark	Laufzeit	Parameter
stencil_seq	89.413ms	$N = 2^{14}$, $t = 30$
stencil_openmp	21.875ms	$N = 2^{14}$, $v = 128$, $t = 30$
	Segmentation Fault	$N = 2^{14}$, $v = 256$, $t = 30$
	72.558ms	$N = 2^{14}$, $v = 128$, $t = 100$
stencil_cilkf	17.749ms	$N = 2^{14}$, $v = 128$, $t = 30$
	17.601ms	$N = 2^{14}$, $v = 256$, $t = 30$
	58.705ms	$N = 2^{14}$, $v = 128$, $t = 100$

Tabelle 5.2: Stencil Benchmark

Der Speedup Wert wird mit Ausführungszeit des sequentiellen Algorithmus durch Ausführungszeit des parallelen Algorithmus ermittelt.

$$Speedup = \frac{Tseq}{Tpar} \tag{5.1}$$

1. `nqueens_openmp` Speedup: 6,55, N = 15 und 6,5, N = 16
2. `nqueens_cilkf` Speedup: 4,36, N = 15 und 4,51, N = 16
3. `nqueens_adj_openmp` Speedup: 4,93, N = 15 und 5,25, N = 16.
4. `nqueens_adj_cilkf` Speedup: 0,71, N = 15 und 0,65, N = 16.
5. `stencil_openmp` Speedup: 4,08, N = 16386, t = 30, tiling_wert = 128 Seqmentation Fault, N = 16386, t = 30, tiling_wert = 256.
6. `stencil_cilkf` Speedup: 5,04 N = 16386, t = 30, tiling_wert = 128

Bei der Berechnung von NQueens konnte in beiden Programmierumgebungen Laufzeitverbesserung in Bezug zum sequentiellen Algorithmus festgestellt werden, dabei ist die schnellste Implementierung mit OpenMP schneller als die schnellste Implementierung mit Cilk-F. Die Berechnung von NQueens

`nqueens_openmp` ist schneller als die Implementierung mit Cilk-F `nqueens_cilkf`. Ein Grund dafür könnte sein, dass in Cilk-F mehr Speicherplatz durch die Teileberechnungen benötigt wird. Bei der Berechnung von NQueens mit OpenMP `nqueens_adj_openmp` und Cilk-F `nqueens_adj_cilkf` mit einstellbaren Tasks/Futures ist die Berechnung mit OpenMP schneller als die Berechnung mit Cilk-F.

Der Algorithmus in Cilk-F ist langsamer, als die sequentielle Berechnung, eine Vermutung ist, dass der Algorithmus nicht parallel funktioniert und sequentiell ausgeführt wird und außerdem durch die Erstellung und Verwaltung der Futures langsamer ist als der sequentielle Algorithmus. Ein weiterer Grund für die schlechte Laufzeit könnte die häufige Erstellung eines Full Frames in der `nqueens` Methode durch `CILK_FUNC_PREAMBLE;` sein. Dieses Makro kann an keiner anderen Stelle gesetzt werden, da der erzeugte Full Frame in den nachfolgenden Makros `START_FIRST_FUTURE_SPAWN;` benötigt wird. Eine mögliche Lösung wäre es, den Full Frame einmal vor dem ersten Methodenaufruf `nqueens` zu initialisieren ohne das Makro `CILK_FUNC_PREAMBLE;` zu verwenden, damit er als Parameter in der Methode `nqueens` verwendet werden kann. Diese Lösung konnte nicht implementiert werden, da dafür weitere Einarbeitungszeit nötig gewesen wäre.

Bei den Implementierungen für den Stencil Algorithmus war der Speedup von Cilk-F höher als der Speedup von OpenMP. Das kommt vermutlich daher, dass der Aufwand der Synchronisation in Cilk-F geringer ist als in OpenMP. Ein möglicher Vorteil der Futures kommt zum Tragen, da ein Future nur solange warten muss, wie ein anderes Futures Zeit für die Berechnung braucht. In OpenMP ist vermutlich der Aufwand für die Synchronisation höher, weil zunächst überprüft werden muss, ob der Bereich von der Berechnung fertig ist.

Der erzielte Speedup mit beiden Programmierumgebungen Cilk-F und OpenMP weist kleine Unterschiede auf, im Vergleich steht effektivste Cilk-F Implementierung NQueens, Stencil und effektivste OpenMP Implementierung NQueens, Stencil.

6 Zusammenfassung und Fazit

In dieser Bachelorarbeit wurde die Benutzbarkeit und Effizienz der parallelen Programmierumgebungen OpenMP und Cilk-F verglichen. Beide Programmierumgebungen sind für die Programmierung von Rechnern mit gemeinsamen Speicher konzipiert worden.
OpenMP hat eine größere Verbreitung als Cilk-F, Cilk-F wird hauptsächlich im akademischen Rahmen benutzt. In Cilk-F werden Futures verwendet, die Verwendung von Futures bietet Vorteile. Da Cilk-F ein Prototyp ist, ist die Programmierung zeitaufwendig. Ohne Wissen und Informationen über das Elternsystem Cilk ist es schwierig zu verstehen, was in den mitgelieferten Benchmarks passiert.
Wenn das Wissen geschaffen wurde, steht man vor nicht umsetzbaren Ideen, da die Benutzung und Verwaltung der Futures aufwendig und fehleranfällig im aktuellen Entwicklungsstand von Cilk-F ist.
In OpenMP können diese Ideen umgesetzt werden, auch weil bereits Erfahrungen vorhanden sind und Beispiele aus dem Internet zahlreich.
In dieser Bachelorarbeit wurden mehrere Benchmarks implementiert, anhand der Erfahrung bei der Implementierung dieser Benchmarks, wurden allgemeine Informationen/Grundlagen zur Programmierung mit Cilk-F verfasst.
Zusätzlich wurden die Benchmarks beschrieben. Die Benutzbarkeit zwischen OpenMP und Cilk-F wurde nach den gesammelten Erfahrungen verglichen.
Die Effizienz bei Cilk-F und OpenMP in den implementierten Benchmarks zeigt nur einen kleinen Unterschied bezüglich der Laufzeit auf.
So kommt es zum Resümee, dass mit Cilk-F programmiert werden kann, allerdings mit höherem Zeitaufwand beim Lernen und aufwendigeren Implementierungen, durch die nötige Nutzung der Makros, als in OpenMP. Grundlegend ließen sich alle Benchmarks gut mit beiden Programmierumgebungen implementieren,

beim Benchmark NQueens mit einstellbaren Futures mit Cilk-F kann, aufgrund mangelnder Informationen, nur schwer gesagt werden, ob ein besserer Algorithmus für die Lösungen des Benchmarks in Cilk-F existieren oder das System einfach nicht für die Nutzung bereit ist.

In allen Unterpunkten bei der Wahl der Programmierumgebung liegt der Vorteil bei OpenMP. Dennoch, wenn es um die natürliche Benutzung von Konzepten geht, ist die Benutzung von Futures naheliegender, da weniger Abstraktion, um mit Futures Parallelität zu erzeugen, benötigt wird. Wenn der Entwicklungsstand von Cilk-F ausgereifter wäre, könnte die Programmierung mit Futures einfacher sein, als mit OpenMP Task `dependencies`. Somit ist das Konzept der Futures in Cilk-F interessant und liefert bei einigen Implementierungen Laufzeitverbesserungen und weniger Komplexität. Nachfolgende Arbeiten könnten sein: Die Implementierung von Cilk-F in eine benutzerfreundlichere Programmierumgebung, wie das Elternsystem Cilk.

Die Dokumentation der Programmierumgebung Cilk/Cilk-F und die Implementierung besserer Fehlermeldungen.

Die Implementierung von Benchmarks, die durch die Benutzung von Futures profitieren, da die Zeit für die Einarbeitung höher war als anfangs angenommen.

Literaturverzeichnis

[1] Singer, Kyle ; Xu, Yifan ; Lee, I-Ting Angelina: *P*roactive Work Stealing for Futures, Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP). 2019. https://dl.acm.org/citation.cfm?id=3295735

[2] Singer, Kyle;Agrawal, Kunal;Lee, I-Ting Angelina: *s*cheduling I/O Latency-Hiding Futures in Task-Parallel Platforms https://doi.org/10.1137/1.9781611976021.11

[3] Muller, Stefan K.; Singer, Kyle;Agrawal, Kunal; Goldstein, Noah; Acar, Umut A.; Lee, I-Ting Angelina: *R*esponsive Parallelism with Futures and State arXiv:2004.02870

[4] Frigo, M; Leiserson, C. E.; K. H., Randall: *The Implementation of the Cilk-5 Multithreaded Language.* Implementation of the Cilk-5 Multithreaded Language. ACM PLDI, 1998, pp. 212-223., https://doi.org/10.1145/277650.277725

[5] Schardl, Tao B.; Leiserson, C. E.; Lee, I-Ting Angelina: July 2018 *Announcement: Open Cilk. In SPAA '18: 30th ACM Symposium on Parallelism in Algorithms and Architectures, July 16–18, 2018, Vienna, Austria. ACM, New York, NY, USA, 3 pages.* https://doi.org/10.1145/3210377.3210658

[6] http://opencilk.org

[7] Parallelizing N-Queens with the Intel Parallel Composer https://www.drdobbs.com/go-parallel/article/print?articleId=214303519

[8] T. B. Schardl Benchmarks for Cilk https://github.com/neboat/cilkbench

[9] T. B. Schardl Cilk Presentation http://web.mit.edu/neboat/www/6.S898-sp17/cilkrts.pdf

[10] T. B. Schardl The Cilk Runtime System https://www.youtube.com/watch?v=Z7r4aAZ9Vqo

[11] Voss, Michael; Reinders, James; Asenjo, Rafael: *Pro TBB*. Published by Apress, July 2019, ISBNs 978-1-4842-4397-8, 978-1-4842-4398-5

[12] OpenMP Application Programming Interface https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[13] OpenMP Application Programming Interface Examples https://www.openmp.org/wp-content/uploads/openmp-examples-5.0.0.pdf

[14] Herlihy, Maurice; Liu, Zhiyu: Well-Structured Futures and Cache Locality February 2016 https://arxiv.org/abs/1309.5301v1

[15] Frigo, Matteo; Halpern, Pablo; Leiserson, Charles E.; Reducers and other Cilk++ hyperobjects 11 August 2009 PAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architecturesAugust 2009 Pages 79–90 https://doi.org/10.1145/1583991.1584017

[16] Intel® Cilk++ SDK Programmer's Guide https://www.clear.rice.edu/comp422/resources/Intel_Cilk++_Programmers_Guide.pdf

[17] Intel® Core™ i7-920 Prozessor https://ark.intel.com/content/www/de/de/ark/products/37 core-i7-920-processor-8m-cache-2-66-ghz-4-80-gt-s-intel-qpi.html

[18] Oppermann, Reinhard; Reiterer, Harald: Einführung in die Software-Ergonomie 1994 http://nbn-resolving.de/urn:nbn:de:bsz:352-252951

[19] https://www.docker.com/