Fallstudien zur fehlertoleranten Programmierung mit Erlang

Universität Kassel

Bachelorarbeit von

Christian Schaub

vorgelegt im

Fachgebiet Programmiersprachen/- methodik

Betreuerin: Prof. Dr. Claudia Fohry

Zweitgutachter: Prof. Dr. Albert Zündorf

Date: Sep 16, 2015

Selbständigkeitserklärung

Ich versichere hiermit, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.
Kassel, den 16. September 2015

Christian Schaub

Inhaltsverzeichnis

1 Einleitung

Die Recheneinheiten moderner Computer (CPU, GPU) werden heutzutage in Mehrkernbauweise gefertigt, da ein einzelner Kern auf Grund physikalischer Gegebenheiten nicht über eine bestimmte Verarbeitungsgeschwindigkeit hinaus betrieben werden kann. Dieses Konzept verlangt nach einer Möglichkeit die vorhandenen Kerne parallel und effizient zu nutzen. Programmiersprachen wie OpenMP, MPI, X10 oder Erlang haben dabei unterschiedliche Herangehensweisen. Das Konzept eines "shared memory" wird mit OpenMP [1] umgesetzt, hier können Prozesse auf einen gemeinsamen Adressraum zugreifen. Bei Message Passing mit MPI oder Erlang [2], [3], kommunizieren die Prozesse ihre Daten über Nachrichten. Die Sprache X10 nutzt einen "Asynchronous Partioned Global Adress Space" (APGAS), d.h. einen globalen Adressraum, der zur Verwendung durch mehrere Prozesse in Places eingeteilt wird. Im Vordergrund dieser Arbeit stehen die Sprachen Erlang und X10.

In herkömmlichen parallelen Programmen bewirkt ein Fehler in einem der parallelen Prozesse den Absturz der gesamten Anwendung, auch wenn andere Prozesse noch korrekt arbeiten. Wird die "Mean Time Between Failures" (MTBF) eines Prozesses mit 6 Monaten angegeben, hat ein 24h Stunden dauernder Auftrag mit 1000 Prozessen nur eine Chance von unter einem Prozent abgeschlossen zu werden [4]. Nur wenige Programmiersprachen, wie Erlang oder Hadoop [5], können Fehler behandeln, da der Absturz eines Prozesses nicht das Beenden der Anwendung bedeutet. In Erlang wird der Programmierer über das Beenden eines parallelen Prozesses, durch einen Fehler oder erfolgreich, mit einer Nachricht an den erzeugenden Prozess informiert. Mit Resilient X10 [4] wurde eine X10 Erweiterung entwickelt, die das Implementieren eines ähnlichen Verhaltens ermöglicht. Hier wird das vorzeitige Beenden eines Prozesses durch eine bestimmte Exception der Anwendung mitgeteilt, so dass der Programmierer auf den Fehler des Prozesses reagieren kann. Kawachiya [6] beschreibt drei Möglichkeiten mit einem Prozessabsturz umzugehen:

1. Ignorieren des Fehlers
2. Übernahme der Arbeit durch verbliebene Prozesse
3. Wiederherstellung der verlorenen Werte anhand periodisch gespeicherter Daten

Diese werden anhand mehrerer Algorithmen und X10-Beispielprogramme veranschaulicht.

Im Rahmen der Bachelorarbeit wurden die Algorithmen MontePi, KMeans und HeatTransfer aus [6] mittels der Sprache Erlang implementiert. Anschließend konnten die unterschiedlichen Herangehensweisen in Erlang und X10 hinsichtlich Effizienz und Programmieraufwand bewertet werden. Es zeigten sich einerseits die Vorteile der Sprache Erlang bezüglich der Prozessverwaltung und Fehlerbehandlung. Andererseits war die Sprache X10 bezüglich der Ausführungszeit im Vorteil.

Kapitel 2 gibt einen Überblick über Erlang und Resilient X10. Anschließend werden in den Kapiteln 3, 4 und 5 zunächst die Algorithmen MontePi, KMeans bzw. HeatTransfer erläutert, und danach die Resilient X10 und Erlang Implementierungen dargestellt. Die Resilient X10-Programme wurden aus [6] entnommen und werden kurz, die im Rahmen der Arbeit implementierten Erlang-Programme detaillierter beschrieben. Kapitel 6 enthält die Darstellungen der durchgeführten Fehlertoleranz-Experimente, deren Ergebnisse und die Bewertung des Programmieraufwandes. Abschließend wird die Arbeit in Kapitel 7 kurz zusammengefasst.

2 Grundlagen Programmierung

Im folgenden Kapitel werden die verwendeten Programmiersprachen Erlang und Resilient X10 beschrieben. Zur Einarbeitung in Erlang wurden das Buch [7] und die offizielle Erlang-HP [3] verwendet. Die Informationen zu X10 wurden aus [4],[6] und der offiziellen X10-HP [8] entnommen. Im Rahmen dieser Arbeit wurde mit Erlang gearbeitet. Erlang wird daher detaillierter dargestellt. Der Fokus der Beschreibung liegt auf Parallelität und Fehlertoleranz.

2.1 Resilient X10

Die imperative und objektorientierte Programmiersprache X10 wird seit 2004 von IBM entwickelt. Das Ziel beim Entwurf der Sprache, war ein einfaches und praktikables Programmiermodell für Parallelrechner anzubieten. Das verwendete APGAS Modell basiert auf zwei Grundbegriffen [4]:

1. Places
2. Asynchrony

Ein Place abstrahiert gemeinsam genutzte, veränderliche Daten und mit diesen arbeitende Threads. Das Hauptprogramm `main` läuft auf Place 0. Asynchrone Verarbeitung wird

durch das `async` Schlüsselwort ermöglicht. Die Zeile `async X` startet den Task X in einem parallelen Thread, im aktuellen Place. Das `at` Statement wird verwendet, um den Place zu wechseln. Der Datenaustausch zwischen Threads in verschiedenen Places erfolgt unter anderem über globale Referenzen. Das Schlüsselwort `finish` wurde vorgesehen, um auf die Beendigung mehrerer Threads warten zu können. Das Codebeispiel 1 veranschaulicht die Benutzung der Schlüsselwörter `async`, `at` und `finish`.

Das Beispiel demonstriert außerdem die Fehlertoleranz von Resilient X10. Die Zeilen 4 und 6-8 zeigen den notwendigen Code. Ein Fehler führt hier nicht zum Absturz, sondern wird dem Programmierer durch eine `DeadPlaceException` mitgeteilt. In diesem Beispiel wird der Absturz nicht weiter behandelt, es erfolgt die simple Ausgabe einer Fehlermeldung.

Listing 1: Ein einfaches fehlertolerantes Programm in Resilient X10 Quelle: [6]

```
1  class ResilientExample {
2    public static def main(Rail[String]) {
3      finish for (pl in Place.places()) async {
4        try {
5          at (pl) do_something(); // parallel distributed execution
6        } catch (e:DeadPlaceException) {
7          Console.out.println(e.place + " died"); // report failure
8        }
9      } // end of finish wait, for execution in all places
10   }
11 }
```

2.2 Erlang

Die funktionale Programmiersprache Erlang [3] wird seit 1987 von Ericsson entwickelt. Der Anwendungsbereich ist die Telekommunikation. Mit der Open Telecom Platform (OTP) wird eine umfangreiche Bibliothek zum Bau paralleler, hochverfügbarer Systeme mitgeliefert. Ziele beim Sprachentwurf waren:

- Parallelität
- hohe Verfügbarkeit
- Fehlertoleranz
- Auswechseln von Modulen zur Laufzeit

Ericsson beziffert die Verfügbarkeit ihrer Erlang-Systeme auf über 99% im Jahr. Ein Grund sind die integrierten Möglichkeiten zur fehlertoleranten Programmierung. Erlang ist eine

dynamisch typisierte Sprache[1]. Folgende Datentypen/- Strukturen werden verwendet:

- Variablen, z.B. `Points`, `Name`
- Skalare, z.B. `2`, `33`, `2003`
- Atome, z.B. `abc`, `table` (erster Buchstabe klein)
- Records, z.B. `R = #person{x="Hans", y="Schmid"}`
- Listen, z.B. `[a,b,c]`, `[33,557]`
- Tupel, z.B. `{abc, 3.6, [a,b,c]}`
- Funktionen, z.B. `fun() -> fwrite("hello\n") end.`

Erlang-Anwendungen werden in einer virtuellen Maschine (Erlang-VM) gestartet. Die Prozessverwaltung wurde ressourcenschonend implementiert, die Größe eines Prozesses umfasst nach [3] nur ca. 300 Wörter innerhalb dieser VM. Prozesse in Erlang können aus Betriebssystem-Sicht teils als Prozess, andererseits auch als Thread klassifiziert werden. Die Gemeinsamkeit mit einem Thread ist die Leichtgewichtigkeit und die dadurch hohe Performance beim Scheduling. Die Verwendung separater Speicherbereiche passt eher zu einem Prozess.

Die Erzeugung nebenläufiger Prozesse wird mittels der Methode `spawn` ermöglicht. Die spawn-Funktion gibt als Rückgabewert den Prozessidentifikator (Pid) des erzeugten Prozesses zurück. Prozesse können in Erlang nicht direkt auf Daten anderer Prozesse zugreifen, die Kommunikation erfolgt ausschließlich über Nachrichten. Die Adressierung erfolgt über die Pid der Prozesse. Nachrichten werden durch einen `receive -> ... end` Block empfangen und mit dem Sende-Symbol "!" versendet. Inhalt der Nachrichten können beliebige Erlang-Datenstrukturen sein. Grundsätzlich werden Nachrichten der Form `{'Atom', Daten}` versendet. Das Atom wird zur Unterscheidung der Nachrichten verwendet.

Eine in dieser Arbeit oft verwendete Kontrollstruktur ist die `if/else` Anweisung. Die Struktur des `else` Zweiges in Erlang ist etwas ungewöhnlich: Mit `if x == 0` wird die gewohnte Prüfung eingeleitet, anschließend folgt der `true` Zweig, der als Pendant zum gewohnten `else` Zweig gesehen werden kann.

[1] Typprüfungen werden zur Laufzeit eines Programms durchgeführt

Die Funktionalität des folgenden Beispiels ist angelehnt an Beispiel 1, Listing 1. Die Implementierung dieses Beispiels wird in Listing 2 dargestellt. Das Programm startet in Zeile 3 und 8ff eine vorgegebene Anzahl an Prozessen. Der Erzeugerprozess startet anschließend mit der Funktion `getMessages` das Warten auf Nachrichten mittels des `receive`-Blockes. Diese Funktion ruft sich nach der Auswertung einer empfangenen Nachricht rekursiv auf, bis `ProcessCount` Bestätigungen (normal beendet oder Absturz) der Prozesse eingegangen sind. Die Definition der Funktion mit festem Parameter 0 (Zeile 14) dient in Erlang als eine Abbruchbedingung.

Der letzte Prozess schickt keine Nachricht zurück an den Erzeuger. Der Prozess beendet fehlerhaft mit einer von `normal` abweichenden `Reason`. Durch den Aufruf von `exit(Reason)` (Zeile 27) versendet das Erlang-System automatisch die Nachricht `{'DOWN', Ref, process, Pid2, Reason}` (Zeile 21) an den überwachenden Prozess.

Listing 2: Ein einfaches fehlertolerantes Programm in Erlang

```
1  start (ProcessCount) ->
2    process_flag(trap_exit, true),
3    createProcesses(ProcessCount, self()),
4    getMessages(ProcessCount).

6  %% create the processes and monitor them
7  createProcesses (0, _) -> io:format("Started all processes.~n");
8  createProcesses (ProcessCount, Pid) ->
9    PidNew = spawn(simple, do_something, [Pid, ProcessCount]),
10   erlang:monitor(process, PidNew),
11   createProcesses(ProcessCount - 1, Pid).

13 %% Wait for the msg, result or error
14 getMessages (0) -> io:format("Waiting for messages done ~n");
15 getMessages (ProcessCount) ->
16   receive
17     %% got a result msg
18     {'result', Count} ->
19       getMessages(ProcessCount -1);
20     %% ignore error, go further receiving messages
21     {'DOWN', Ref, process, Pid2, Reason} ->
22       getMessages(ProcessCount -1)
23   end.

25 %% Sends the result msg back, arithmetic failure when process count is smaller 1
26 do_something (Pid , ProcessCount) ->
27   if ProcessCount < 1 -> erlang:raise(exit, "Boom!", []);
28      true ->
29      Pid ! {'result', "123"}
30   end.
```

Zur Programmierung fehlertoleranter Anwendungen ist es erforderlich Fehler in einem Prozess zu kommunizieren. In Erlang stehen zwei Möglichkeiten zur Interprozesskommunikation (IPC) zur Verfügung:

- Link

  Ein Link ist eine bidirektionale Verbindung zwischen zwei Prozessen. Die Verbindung wird mit einer speziellen `spawn` Funktion hergestellt, oder mittels `link(Pid)`. Dieses Konzept wird in Erlang dazu verwendet, Fehler durch eine Kette von Prozessen zu propagieren, die an einer Aufgabe arbeiten oder anderweitig in Abhängigkeit stehen. Die Kommunikation der Fehler erfolgt in Form eines "exit"-Signales. Eine weitere Verbreitung des Fehlers im System, soll durch die Versendung eines "exit"-Signales an alle verlinkten Prozesse gewährleistet werden. Diese Signale können weder ignoriert noch direkt empfangen werden. Möchte ein Prozess derartige Nachrichten empfangen, anstatt direkt beendet zu werden, muss dieses Verhalten mit `process_flag('trap_exit', true)` aktiviert werden (Zeile 2).

- Monitor

  Der Monitor stellt eine unidirektionale Verbindung zu einem zu überwachenden Prozess her. Der Überwacher wird bei Beendigung des überwachten Prozesses durch eine Nachricht über die Pid und das Beenden (normal oder fehlerhaft) des Prozesses informiert. Anschließend können Maßnahmen zur Fehlerbehandlung getroffen werden, beispielsweise eine der Herangehensweisen a, b oder c aus [6]. Mit `erlang:monitor (process, Pid)` (Zeile 10) wird, wie eingangs erwähnt, die Überwachung eines Prozesses aktiviert. Der überwachende Prozess wird durch eine Nachricht der Form `{'DOWN', Ref, process, Pid2, Reason}` (Zeile 21) über das Beenden eines mit aktiviertem Monitor überwachten Prozesses informiert. Innerhalb der Erlang-VM beendet ein Prozess entweder mit der `Reason` 'normal', was eine fehlerfreie Ausführung kennzeichnet. Jede andere `Reason` bedeutet einen Fehler, wie z.B. 'badarith' im Fall einer Division durch 0.

2.2.1 Erlang Term Storage (ETS)

Der Erlang Term Storage (ETS) ist eine in der Erlang-Laufzeitumgebung integrierte Datenbank. Es ist möglich sehr große Datenmengen zu speichern. Diese werden als dynamische Tabellen (Table) organisiert, die Erlang-Tupel speichern können. Das erste Element

innerhalb dieser Tupel wird als Schlüssel verwendet, um die Elemente zu unterscheiden. ETS-Tabellen werden in vier Typen eingeteilt:

1. set - Nur ein Element pro Schlüssel
2. ordered_set - Elemente geordnet, sonst wie set
3. bag - Viele Elemente pro Schlüssel, nur eine Instanz pro Element
4. duplicate bag - Mehrere Instanzen eines Elementes möglich, sonst wie bag

Der Prozess der die Tabelle erzeugt, wird dessen Eigentümer. Beendet der Eigentümer-Prozess wird automatisch jede Tabelle gelöscht, der von diesem Prozess erzeugt wurde. Ein Tabelle-Erbe kann durch die Option `heir` definiert werden. Existiert ein Erbe, wird dieser beim Beenden des Tabelle-Erzeugers der neue Eigentümer der Tabelle. Bei der Erzeugung mit `ets:new(name, Options)` können der Name der Tabelle, sowie verschiedene Optionen, wie Zugriffsrechte , festgelegt werden. Diese Funktion gibt einen `table identifier` zurück, der an andere Prozesse gesendet werden kann um die Tabelle auch verwenden zu können. Die Zugriffsrechte der Tabelle müssen in diesem Fall auf `public` gesetzt werden. Eine weitere Möglichkeit des gemeinsamen Zugriffs auf eine Tabelle ist die Vergabe eines Atoms `name` zur Identifikation, mit gleichzeitiger Aktivierung der Option `named_table`.

ETS-Tabellen bieten eine begrenzte Unterstützung für gleichzeitigen Zugriff. Alle Veränderungen an einzelnen Objekten sind garantiert atomar und isoliert. Das bedeutet eine Veränderung ist entweder erfolgreich oder fällt komplett aus (atomar). Des weiteren können Zwischenergebnisse nicht von anderen Prozessen gesehen werden (isoliert).

3 Fallstudie Monte Pi

Dieser Algorithmus zur Berechnung einer Näherung der Zahl Pi dient zur Darstellung des Verfahrens a) einfaches Ignorieren eines Prozessabsturzes.

3.1 Algorithmus

Um die Zahl Pi mittels einer Monte-Carlo-Simulation näherungsweise zu berechnen, werden Punkte in einem 2×2 Quadrat randomisiert erzeugt und deren Lage überprüft. Auf den Mittelpunkt dieses Quadrates wird ein Kreis mit Radius 1 gelegt. Punkte im Kreis gelten als Treffer und werden gezählt. Abbildung 1 illustriert diese Idee.

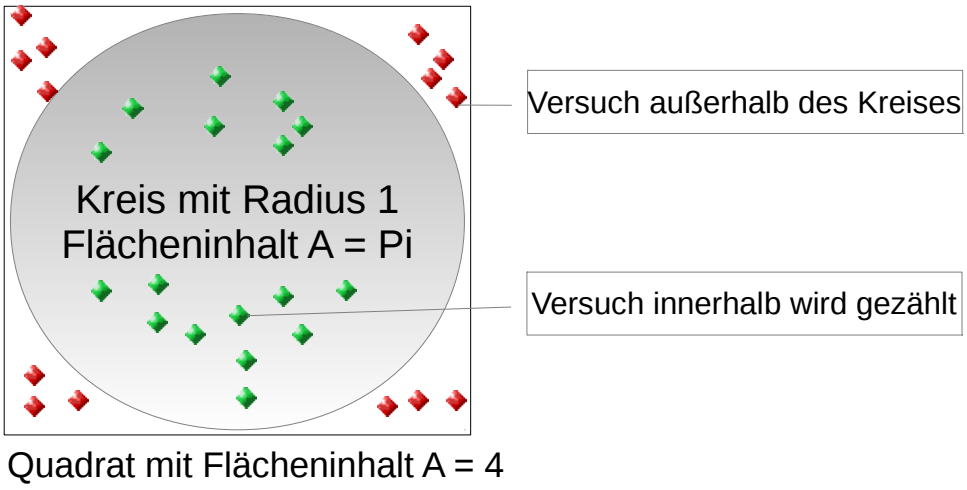

Abbildung 1: Veranschaulichung zum Algorithmus Monte Pi

Das Verhältnis der Versuche zu den Treffern wird anschließend verwendet um den Flächeninhalt $A = 4$ des Quadrates mittels Multiplikation von $\pi = A * \frac{Treffer_{gesamt}}{Versuche_{gesamt}}$ auf den des Kreises mit $A = \pi * r^2 = \pi(r = 1)$ zu reduzieren. Die Genauigkeit der Zahl π steigt mit der Anzahl der Versuche. Zu Beginn sind die Anzahl möglicher Versuche einen Punkt im Kreis zu finden und die Prozessanzahl gegeben. Die Versuche werden auf die Prozesse verteilt und parallel ausgeführt. Die Ausführung verläuft folgendermaßen:

1. Zufällige Wahl eines Punktes innerhalb eines 2×2 Quadrates
2. Prüfung, ob Punkt innerhalb des Kreises liegt
3. Falls ja, diesen Versuch zählen (Treffer_lokal)
4. Falls Anzahl Versuche erreicht: weiter mit 5., sonst zurück zu 1.
5. Zähler (Punkte im Kreis) der Prozesse summieren (Treffer_global)
6. Verhältnis der Anzahl der Punkte im Kreis und der Versuche bilden
7. Zahl Pi ergibt sich durch Multiplikation dieses Verhältnisses mit dem Flächeninhalt des 2×2 Quadrates

Da ein Prozess hier eine bestimmte Anzahl Punkte bearbeitet, fehlen diese in der abschließenden Berechnung sollte der Prozess abstürzen. Deshalb muß außerdem die Gesamtzahl

der Versuche aktualisiert werden. Die Zahl Pi wird durch das Fehlen dieser Versuche ungenauer. Die folgenden Abschnitte 3.2 und 3.3 beschreiben die Umsetzung in Resilient X10 und Erlang.

3.2 Implementierung in Resilient X10

Der Programmcode in Listing 3 wurde aus [4] entnommen. Es folgt eine kurze Beschreibung des Codes: Zeile 3 enthält die Definition der globalen Referenz `result`. In Zeile 4 wird mit `finish for ... async` die parallele Verarbeitung gestartet. Ein Place wird mittels `at(p)` ausgewählt (Zeile 6). Die Berechnung der Punkte findet in den Zeilen 9-12 statt. Die Anzahl der Versuche pro Place wird mit `ITERS` festgelegt. Abschließend wird in Zeile 17 die globale Referenz des Trefferzählers `result` aktualisiert. Durch den Ausdruck `atomic` (Zeile 15) wird sichergestellt, dass die Aktualisierung in einer atomaren Operation durchgeführt wird. Wurden in allen Prozessen `ITERS` Punkte berechnet und geprüft, wird das Ende des `finish`-Blockes (Zeile 21) erreicht. Hier kann nun der Wert der Zahl Pi, wie in 3.1 beschrieben, berechnet und ausgegeben werden. Sollte ein Place ausfallen, wirft das `at` Statement (Zeile 6) eine `DeadPlaceException`. Diese wird durch das `catch` Statement (Zeile 20) gefangen und einfach ignoriert. Da der Place ausgefallen ist, wird Zeile 17 nicht ausgeführt und die globale Referenz nicht aktualisiert. Dadurch ergibt sich eine ungenauere Näherung der Zahl Pi, da die diesem Place zugeordneten Versuche den Kreis zu treffen verfallen.

Listing 3: Monte Pi Implementierung in Resilient X10 Quelle: [6]

```
1  class ResilientMontePi {
2    public static def main (args:Rail[String]) {
3      val result = GlobalRef(new Cell(Pair[Long,Long](0l, 0l)));
4      finish for (p in Place.places()) async {
5        try {
6          at (p) {
7            val rnd = new x10.util.Random(System.nanoTime());
8            var c:Long = 0;
9            for (iter in 1..ITERS) {
10             // ITERS trials per place
11             val x = rnd.nextDouble(), y = rnd.nextDouble();
12             if (x*x + y*y <= 1.0) c++; // if inside the circle
13           }
14           val count = c;
15           at (result) atomic {
16             // update the global result
17             val r = result(); r() = Pair(r().first+count, r().second+ITERS);
18           }
19         }
20       } catch (e:DeadPlaceException) { /* just ignore place death */ }
21     } // end of finish, wait for the execution in all places
22 /* calculate the value of Pi and print it */
23 }
```

3.3 Implementierung in Erlang

Das folgende Codebeispiel 4 zeigt die Implementierung des Monte Pi Algorithmus in Erlang. Einzelne Programmteile werden in Erlang in Modulen organisiert, dieses Programm besteht nur aus einem Modul `errorp` (Zeile 1). Die Anwendung wird in einer Erlang-Konsole mittels der `run` Funktion (Zeile 4) und Angabe des zugehörigen Moduls gestartet, die Anzahl der arbeitenden Prozesse, sowie möglicher Versuche werden als Parameter übergeben. Hier gilt die Annahme, dass die Versuche ohne Rest auf die Prozesse verteilt werden können.

Die Funktion `createWorker` (Zeile 6, 11) startet rekursiv die angegebenen Prozesse und aktiviert jeweils einen Monitor zur Überwachung. Der Funktion `spawn` (14), zum Starten neuer Prozesse, werden beim Aufruf einige Parameter mit folgender Bedeutung übergeben: Der erste Parameter gibt das Modul an, in dem die Funktion liegt, die in einem separaten Prozess gestartet werden soll. Anschließend wird der Name der Funktion angegeben, gefolgt von einem Tupel. Das Tupel enthält die Parameter, die die Funktion zum Start erwartet. Jedem Prozess wird willkürlich, zur Überprüfung der Fehlertoleranz, als letzter Parameter ein Indikator zur vorzeitigen Beendigung übergeben. Innerhalb der

Funktion `process_points` wird dieser geprüft, im Fall einer `1` wird der Prozess abrupt mit einer Fehlermeldung beendet. Ansonsten übernimmt die nicht dargestellte Funktion `check_points_in_circle` (Zeile 49) die Berechnung der Punkte und möglicher Treffer pro Prozess. Die Anzahl dieser Treffer wird abschließend in einer Nachricht zum angegebenen Prozess (Pid) (Zeile 50) geschickt.

Der überwachende Prozess wartet durch das Starten der Funktion `getMessages` (Zeile 28) auf Nachrichten seiner Arbeiter. In dieser Funktion liegt die `receive` Anweisung, die auf Nachrichten der Form `{'result', Count}` oder ein vorzeitiges Beenden, signalisiert durch `{'DOWN', Ref, process, Pid2, Reason}`, wartet. In Zeile 34 wird der Reason Parameter geprüft, um falls nötig Fehler behandeln zu können. Die Definition der Funktion `getMessages` (Zeile 22) mit einem festen Parameter, dient auch hier als Abbruchbedingung der Rekursion. Wird ein Prozess fehlerhaft beendet, müssen diese Versuche von der Gesamtanzahl abgezogen werden, um das korrekte Ergebnis zu erhalten (Zeile 37). Die abschließende Berechnung der Zahl π findet erst statt, nachdem eine Rückmeldung aller Arbeiterprozesse erhalten wurde (Zeile 23f).

Um ein endloses Warten durch den `receive`-Block zu verhindern, wurde mit `after 3000` (Zeile 39) ein Zeitlimit von drei Sekunden zwischen dem Eintreffen neuer Nachrichten eingerichtet. Dieser Fall könnte hier nur durch einen Programmierfehler eintreten und dient daher Debug-Zwecken.

Listing 4: Monte Pi Algorithmus in Erlang

```
-module (errorp).
-export ([run/2, runTimer/2, process_points/3]).

run (ProcessCount, Points) ->
  PointsPerProcess = Points div ProcessCount,
  createWorker (ProcessCount, PointsPerProcess),
  getMessages (ProcessCount, Points, PointsPerProcess,0).

createWorker (0, PointsPerProcess) ->
  io:format("Started all processes.~n~n");
createWorker (ProcessCount, PointsPerProcess) ->
  Pid = self(),
  if ProcessCount rem 2 == 0 ->
    PidNew = spawn(errorp, process_points, [PointsPerProcess, Pid, 1]);
     true ->
     PidNew = spawn(errorp, process_points, [PointsPerProcess, Pid, 0])
  end,
  erlang:monitor(process, PidNew),
  io:format("Worker created: ~p PID ~p~n", [ProcessCount, PidNew]),
  createWorker(ProcessCount - 1, PointsPerProcess).

getMessages (0, Points, PointsPerProcess, Sum) ->
  Pi = (Sum / Points) * 4,
  Error = abs(math:pi() - Pi),
  io:format("Pi: ~p~n", [Pi]),
  io:format("Error: ~p~n", [Error]);

getMessages (ProcessCount, Points, PointsPerProcess, Sum) ->
  receive
    {'result', Count} ->
      io:format("Sum current: ~p~n", [Sum]),
      getMessages (ProcessCount -1, Points, PointsPerProcess, Sum + Count);
    {'DOWN', Ref, process, Pid2, Reason} ->
      if Reason == 'normal' ->
        getMessages (ProcessCount, Points, PointsPerProcess, Sum);
      true ->
        getMessages (ProcessCount -1, Points - PointsPerProcess, PointsPerProcess, Sum)
      end;
    after 3000 ->
      io:format("timeout in getMessages !!! ~n~n")
  end.

process_points(Points, Pid, Kill) ->
  if Kill == 1 -> exit("boom");
     true -> {ok}
  end,
  % start the random calculator with seed-value of the current time
  apply(random, seed, tuple_to_list(now())),
  Count = check_points_in_circle(Points, 0),
  Pid ! {result, Count}.
```

4 Fallstudie K-Means

Der K-Means Algorithmus dient der Veranschaulichung des Verfahrens b) Übernahme der Arbeit durch verbliebene Prozesse

4.1 Algorithmus

Das Verfahren wird verwendet, um eine Menge von Punkten in k Cluster zu unterteilen. Die Clusteranzahl k wird vorgegeben, kann aber frei gewählt werden[2]. Die Punkte liegen in dieser Anwendung in einem 2D-Raum. Abbildung 2 zeigt eine Punktmenge, unterteilt in zwei Cluster. Der Algorithmus läuft wie folgt ab:

1. k Punkte zufällig[3] aus Punktmenge als Startzentren der k Cluster wählen
2. Punkte dem Cluster mit dem geringstem Euklidischen Abstand zum Clusterzentrum zuordnen
3. Clusterzentren neu berechnen:

   $X_{neu} = Summe(X_1...X_n)/Punktanzahl_{Cluster}$
   $Y_{neu} = Summe(Y_1...Y_n)/Punktanzahl_{Cluster}$
4. Bei Änderung der Zentren: zurück zu 2.; vorzeitiger Abbruch durch Kriterium; sonst: Ende

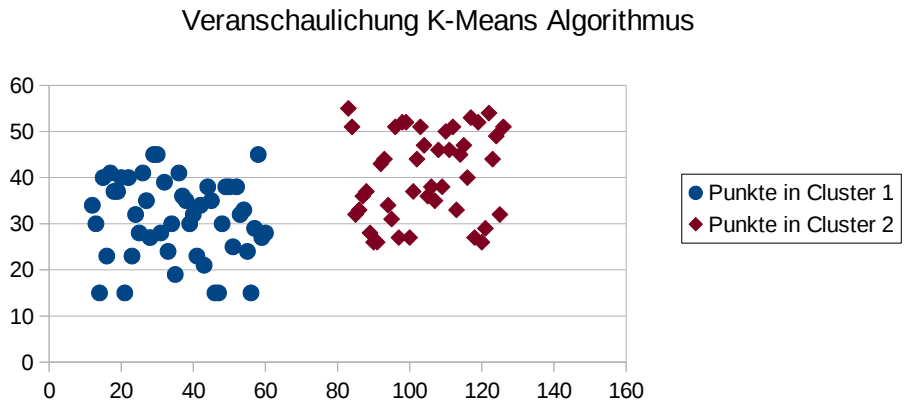

Abbildung 2: Punktmenge unterteilt in zwei Cluster

[2]Wahl eines neuen k kann Ergebnis verändern
[3]Auswahl der Startpunkte kann Ergebnis verändern

4.2 Implementierung in Resilient X10

Das folgende Listing 5 zeigt die aus [6] entnommene Implementierung des K-Means Algorithmus. Die Fehlerbehandlung wurde nach Verfahren b) implementiert. Dieses Verfahren kann angewendet werden, da zu Beginn die gesamte Punktmenge an alle Places übermittelt wird (Zeile 3). Dadurch ist jeder Place in der Lagew die gesamten Berechnungen zu übernehmen, ohne Abhängigkeiten zu einem eventuell abgestürzten Place.

In jeder Iteration werden zuerst die Places und die damit verbundene Aufteilung der Punktmenge, in Intervalle pro Prozess, bestimmt (Zeile 9ff). Die zur Fehlerbehandlung wichtigste Anweisung (Zeile 10) bestimmt die Anzahl aktiver Places. Anschließend werden an jedem Place die Punkte im Interval des Place lokal den Clustern zugeordnet (Zeile 14ff). Diese lokalen Berechnungen werden, sobald alle Places die Verarbeitung abgeschlossen haben, zu zentralen Clustern vereinigt (Zeile 23) und mittels der globalen Referenz (Zeile 5) gespeichert. Das Warten auf das Abschließen aller Places wird mittels der `finish` Anweisung (Zeile 15) erreicht.

Listing 5: K-Means Implementierung in Resilient X10 Quelle: [6]

```
1  class ResilientKMeans {
2    public static def main(args:Rail[String]) {
3      val points_local = PlaceLocalHandle.make[Array[Float]{region==points_region}]
4        (Place.places(), ()=>points_master);
5      val local_new_clusters = PlaceLocalHandle.make[Rail[Float]](Place.places(),
6        ()=>new Rail[Float](CLUSTERS*DIM));
7      val central_clusters_gr = GlobalRef(central_clusters);
8      for (iter in 1..ITERATIONS) { // iterate until convergence
9        /* deliver current cluster values to other places by using the PLaceLocalHandle*/
10       val numAvail = Place.MAX_PLACES - Place.numDead();
11       val div = POINTS / numAvail;  // share for each place
12       val rem = POINTS % numAvail; // extra share for Place 0
13       var start:Long = 0;
14       try {
15         finish for (pl in Place.places()) {
16           if (pl.isDead()) continue; // skip dead place(s)
17           var end:Long = start+div; if (pl==place0) end+=rem;
18           at (pl) async {
19             /* process [start,end), and return the data */ }
20             start = end;
21         } // end of finish, wait for the execution in all places
22       } catch (es:DeadPlaceException) { /* just ignore place death */ }
23       /* compute new cluster values and write to the GlobalRef  */
24     } // end of for (iter)
25     /* print the result */
26 }
```

4.3 Implementierung in Erlang

Das folgende Listing 6 zeigt, in gekürzter Form, die Implementierung des K-Means Algorithmus in Erlang. Die Fehlerbehandlung wurde auch hier nach Verfahren b) implementiert. Die Übermittlung der gesamten Punkte erfolgt direkt bei der Erzeugung des Prozesses (Zeile 4, 8). Die Methode `startProc` wird direkt mit dem Prozess gestartet (Zeile 8). Die ETS-Methode `ets:insert(table, Data)` (Zeile 9) wird verwendet, um neue Werte (Data) in eine ETS-Tabelle (table) einzutragen. Hier werden die Pid der aktiven Prozesse gespeichert. Stürzt ein Prozess ab, wird er aus der Tabelle entfernt (Zeile 33).

Die Iteration (s. Listing 5, Zeile 8) der Anwendung, wurde durch die Methode `iterate` implementiert. Zuerst werden der Pid der aktiven Prozesse in eine Liste geschrieben. Die zur Fehlerbehandlung benötigte Anweisung (Zeile 14) bestimmt die Anzahl der aktiven Prozesse in der Liste, mittels der Erlang-Funktion `lists:foldl(fun(X, Sum), Acc, List)`. Diese Funktion ruft für jedes Element der Liste List die gegebene Erlang-fun (fun) auf. In diesem Fall wird zu dem Startwert Acc (0) pro Element eins addiert. Eine Erlang-Funktion mit dieser Funktionalität existiert nicht. Mit der aktuellen Prozessanzahl werden die neuen Intervalle der Punktmenge pro Prozess bestimmt (Zeile 15). Im nächsten Schritt, werden die neuen Cluster und Intervalle mittels der Methode `sendDataToProcs` (Zeile 16, 21) an die aktiven Prozesse versendet. Die Methode wird rekursiv aufgerufen, bis alle Prozesse abgearbeitet wurden. Mittels der Erlang-Funktion `element(Position, Tuple)` werden die jeweiligen Pid aus der Prozessliste gelesen, um anschließend die aktuellen Clusterzentren und das Interval des Prozesses zu übertragen (Zeile 23). Anschließend werden pro Prozess die Cluster für das Punkt-Interval lokal berechnet (Zeile 42, 46ff). Diese Berechnungen starten, wenn im `receive`-Block der Methode `startProc` (Zeile 39) neue Werte für Cluster und Interval in Form der Nachricht `{'newClusters', [NewClusterList, Start, End]}` eintreffen. Die Funktion `calcLocalClusters` (Zeile 42, 46) berechnet die lokalen Cluster, diese werden anschließend an `Pid` versendet (Zeile 43). Diese lokalen Berechnungen werden zu zentralen Clustern vereinigt, wenn alle Prozesse fertig sind und in der `central_cluster` ETS-Tabelle gespeichert. Abschließend werden die neuen zentralen Cluster mittels `calcCentral Clusters` (Zeile 18, 46) berechnet.

Listing 6: K-Means Algorithmus in Erlang (Auszüge)

```
start () ->
  ITERATIONS = 2, PROCESSES = 2, POINT_COUNT = 15,
  createKMeansTables:createPointTable(), createKMeansTables:createCentralCluster(),
  spawnProcesses(PROCESSES, ets:tab2list(points)), %% every process with own pointlist
  iterate(ITERATIONS, POINT_COUNT).

spawnProcesses (ProcessCount, Points) ->
  PidNew = spawn(resKTest, startProc, [Points, self()]), erlang:monitor(process, PidNew),
  ets:insert('processTable', [{PidNew}]), %% add process and to process-table
  spawnProcesses(ProcessCount -1, Points ).

iterate (ITERS, POINT_COUNT) ->
  ProcInfo = ets:tab2list(processTable),
  NumAvail = lists:foldl(fun(X, Sum) -> 1 + Sum end, 0, ProcInfo),% available procs
    Div = POINT_COUNT div NumAvail, Rem = POINT_COUNT rem NumAvail, Start = 1,
  sendDataToProcs(NumAvail, ProcInfo, POINT_COUNT, Start, Div),
  getMessages(NumAvail),
  calcCentralClusters(),
  iterate(ITERS -1, POINT_COUNT).

%% send the current clusters to the processes, triggers the calc of new clusters
sendDataToProcs (Procs, ProcInfo, PointCount, Start, Div) ->
  element(1,lists:nth(Procs,ProcInfo)) !
    {'newClusters' , [ets:tab2list('centralClusters'), Start, Start+Div]},
  sendDataToProcs(Procs-1, ProcInfo, PointCount, Start+Div, Div).

getMessages(Procs) ->
  receive
    {'result', ClusterList} -> % got a result msg
      getMessages(Procs -1);
    {'DOWN', Ref, process, Pid2, Reason} -> % delete dead process from processtable
      if Reason == 'normal' -> getMessages(Procs -1);
      true -> ets:delete(processTable, Pid2), getMessages(Procs -1)
      end
  end.

%% waits on new cluster values, compute new clusters and sends result back to Pid
startProc (Points, Pid) ->
  receive
    {'newClusters', [NewClusterList, Start, End]} -> %% write cluster in local storage
      First = lists:nth(1, NewClusterList), Second = lists:nth(2, NewClusterList),
      LocalClusters = calcLocalClusters(First, Second, Points, Start, End),
      Pid ! {'result', LocalClusters}, %% send result back
      startProc (Points, Pid) end. %% restart waiting for new cluster values

calcLocalClusters (FirstC, SecondC, Points, Start, End) ->
  %% calc every point in process-range
  for_points(FirstC, SecondC, Points, Start, End, [], []).
```

5 Fallstudie HeatTransfer

Am Beispiel dieses Algorithmus wird zum Einen das, mit X10 nicht unmittelbar umsetzbare, Verfahren des neu-starten eines Prozesses nach einem Absturz dargestellt. Zum Anderen wird die Herangehensweise c) Wiederherstellung der Berechnung anhand periodisch gespeicherten Daten, mit X10 gezeigt.

5.1 Algorithmus

Der Algorithmus berechnet die Diffusion von Temperaturen über angrenzende Bereiche in einem zweidimensionalem Raum. Jeder Punkt im Raum speichert einen Temperaturwert, dieser wird innerhalb jedes Durchlaufes durch das arithmetische Mittel der in den umgebenden Punkten gespeicherten Werten ersetzt. Die folgenden Abschnitte 5.2 und 5.3 stellen die Implementierung dieses Algorithmus in Resilient X10 und Erlang dar.

5.2 Implementierung in Resilient X10

Das folgende Listing 7 zeigt die aus [6] entnommene Implementierung des HeatTransfer Algorithmus. Das Temperaturfeld wird in dieser Resilient X10 Implementierung mittels eines `ResilientDistArray` gespeichert. Diese Datenstruktur ermöglicht die Verteilung eines Arrays auf mehrere Places, sowie die Fehlerbehandlung. Die `DistArray`-Funktion `snapShot` (Zeile 7, 26) wird zur Fehlerbehandlung verwendet, indem die gesamte Datenstruktur gespeichert und ggf. wiederhergestellt wird. Ein Ausfall eines Prozesses verursacht daher kaum Datenverlust, die Verluste hängen von der Häufigkeit eines Snapshots ab, dies beeinflusst aber auch die Geschwindigkeit der Anwendung.

Zeile 6 zeigt die Initialisierung des verteilten Arrays `A`, das Array `BigD` (Initialisierung nicht im Listing) dient als Vorgabe des Aufbaues. Anschließend wird ein erster Snapshot des verteilten Arrays gespeichert (Zeile 7). Die Snapshot Funktion des des `ResilientDistArray` wird verwendet, um in bestimmten Abständen ein Backup der Daten durchzuführen. Diese Anwendung speichert in jeder 10. Iteration einen Snapshot (Zeile 26). Sollte ein Place abstürzen wird `restore_needed` (Zeile 4), innerhalb der nicht gezeigten Funktion `processException` (Zeile 27), true gesetzt und dadurch die Wiederherstellung aktiviert. Durch die gespeicherten Snapshots ist es möglich die verteilten Daten wiederherzustellen und unter den verbliebenen Places neu zu verteilen(Zeile 13ff).

Die zur Berechnung des arithm. Mittels der umliegenden Temperaturen benötigten Daten müssen, falls die Werte auf einem anderen Place liegen, übertragen werden (Zeile 22f).

Listing 7: HeatTransfer Implementierung in Resilient X10 Quelle: [6]

```
class ResilientHeatTransfer {
  static val BigR = Region.make(0..(n+1), 0..(n+1));
  static val livePlaces = new ArrayList[Place]();
  static val restore_needed = new Cell[Boolean](false);
  public static def main(args:Rail[String]) {
    val A = ResilientDistArray.make[Double](BigD, ...); // create a DistArray
    A.snapshot(); // create the initial snapshot
    for (iter in 1..ITERATIONS) {
      // iterate until convergence
      try {
        if (restore_needed()) {
          // if some places died
          val livePG = new SparsePlaceGroup(livePlaces.toRail());
          BigD = Dist.makeBlock(BigR, 0, livePG); // recreate Dist
          A.restore(BigD);
          // restore elements from the snapshot
          restore_needed() = false;

        }
        finish ateach (z in D_Base) { // distributed processing
          /* compute new heat values for A's local elements */
          Temp = ((at (A.dist(x-1,y)) A(x-1,y)) + (at (A.dist(x+1,y)) A(x+1,y))
            + (at (A.dist(x,y-1)) A(x,y-1)) + (at (A.dist(x,y+1)) A(x,y+1)))/4;
        }
        /* if converged, exit the for loop */
        if (iter \% 10 == 0) A.snapshot(); // create a snapshot at every 10th iter.
      } catch (e:Exception) { processException(e); }
    } // end of for (iter)
  /* print the result */
  }
}
```

5.3 Implementierung in Erlang

Zur Implementierung des HeatTransfer Algorithmus sind drei Varianten denkbar, von denen aus Zeitgründen nur eine implementiert wurde.

1. Abgestürzten Prozess neu starten, Datenspeicherung in zentraler ETS-Tabelle

   Diese Variante wurde im Rahmen dieser Arbeit implementiert und wird anschließend beschrieben.

2. Bei Absturz eines Arbeiterprozesses werden sämtliche anderen Arbeiter auch beendet

   Diese Variante wäre zum Einen durch das Setzen von Erlang-Links unter den Prozessen und den Einsatz eines Überwacherprozesses (hält die Daten) einfach umzusetzen.

Die Prozesse werden anschließend neu gestartet und die Arbeit durch den Überwacher verteilt.

3. Datenspeicherung mittels einer Mnesia-Datenbank[4]

   Mittels dieser Variante wäre ein zur Resilient X10 Implementierung (DistArray) ähnlicher Aufbau denkbar.

Die Anwendung (Variante 1) startet mit der Definition der Feldgröße, sowie der Iterations/- und Prozessanzahl. Die Prozesse wurden hier analog zu X10 Places genannt (Zeile 2). Anschließend wird das pro Prozess verarbeitete Intervall im Feld bestimmt (Zeile 3). Dieses Feld wird mittels der Methode `createDataTable` (Zeile 4) mit Zufallswerten (Interval 0-100) initialisiert. Die Speicherung des Feldes erfolgt eindimensional, um den Zugriff zu vereinfachen. Die Prozesse werden in einer Prozesstabelle in Form einer ETS-Tabelle `processTable` gespeichert. Die Methode `createProcessTable` erstellt diese Tabelle. Mit `spawnProcesses` (Zeile 7) werden die Prozesse gestartet und die Überwachung aktiviert (Zeile 19f). Das verarbeitete Intervall und der Pid des Prozesses werden in Zeile 21 zu einem Tupel zusammengefasst, um anschließend in der Prozesstabelle gespeichert zu werden.

Die im Prozess ablaufende Funktion `processPoints` (Zeile 12) startet rekursiv die, nicht gezeigte, Methode `diffuseHeat` bis `Iters` den Wert 0 erreicht, das Intervall des Prozesses wird als Parameter übergeben. Mittels `diffuseHeat` werden anschließend die Werte aus dem zentral gespeicherten `dataTable` gelesen, die neuen Temperaturwerte berechnet und gespeichert.

Die Fehlerbehandlung durch Neustarten eines abgestürzten Prozesses wird in der Methode `handleMsg` umgesetzt. Trifft eine `DOWN` Nachricht mit von einer von `normal` abweichenden `Reason` ein, wird das dem Prozess zugeordnete Intervall aus dem `processTable` gelesen und der Prozess neu gestartet (Zeile 38ff). Anschließend wird der Eintrag des abgestürzten Prozesses im `processTable` mit der Pid des neu gestarteten überschrieben und die Methode `handleMsg` gestartet, um weitere Nachrichten empfangen zu können.

[4]Verteilte Datenbank in Erlang, die auf ETS-Tabellen aufbaut

Listing 8: HeatTransfer Algorithmus in Erlang

```
start () ->
  NUMBER_OF_LOOPS = 5,  SIZE_OF_GRID = 20, PLACES = 4,
  INDEX_STEP_SIZE = SIZE_OF_GRID*SIZE_OF_GRID div PLACES,
  createDataTable(SIZE_OF_GRID * SIZE_OF_GRID), %% square table
  createProcessTable(PLACES),
  StartVal = 1,
  spawnProcesses(PLACES, StartVal, INDEX_STEP_SIZE, self(), NUMBER_OF_LOOPS),
  handleMsg(PLACES).

processPoints (Start,End ,Pid ,Kill ,0) -> Pid ! {result, self()};

processPoints (Start, End, Pid, Kill, Iters)  ->
  diffuseHeat(Start, (End - Start)),
  processPoints(Start, End, Pid, Kill, Iters -1).

spawnProcesses (0, _, _,_,_) -> io:fwrite("All Processes spawned \n");

spawnProcesses (ProcessCount, Start, StepSize, PidMsg, Iters) ->
  PidNew = spawn(heatTransfer, processPoints, [Start, Start+StepSize-1, PidMsg]),
  erlang:monitor(process, PidNew),
  Insert = {Start, Start+StepSize, PidNew},
  %% add process and index values to table
  ets:insert('processTable', [Insert]),
  spawnProcesses(ProcessCount -1, Start+StepSize, StepSize, PidMsg, Iters).

handleMsg (0) -> printData(1, 20 * 20);

handleMsg (ProcessCount) ->
  io:format("Startet msg handling ~n"),
  receive
    %% got a result msg
    {'result', Pid} ->
      handleMsg(ProcessCount -1);
    {'DOWN', Ref, process,Pid2, Reason} ->
      if Reason == 'normal' ->
        handleMsg (ProcessCount);
      true ->
        %% Code needed to restart process -> get process index using pid and restart it
        [StartIndex, EndIndex] = lists:nth(1, ets:match('processTable', {'$1','$2',Pid2})),
        PidNew = spawn(heatTransferRestartErrors, processPoints,
          [StartIndex, EndIndex-1, self(), 0, 5]),
        Insert = {StartIndex, EndIndex, PidNew},
        ets:insert('processTable', [Insert]),
        handleMsg(ProcessCount)
      end
  end.

%% create HeatValues from 0 - 100 into the data table
createData (Index, Size) -> ...
```

6 Evaluierung von Programmieraufwand und Effizienz

Das folgende Kapitel enthält die Ergebnisse der Fehlertoleranz-/ und Effizienzexperimenten, sowie deren Bewertung. Abschließend wird der jeweilige Programmieraufwand bewertet. Die Experimente zum Vergleich der Effizienz der Programme in Erlang und X10 wurden auf zwei unterschiedlichen Linux-Systemen durchgeführt. Die Werte wurden mittels cpu-/ meminfo ermittelt:

1. Laptop:

   Thinkpad T61 (Ubuntu, 3.2.0-51), 3 GB RAM, 2 Prozessoren 1,8 Ghz mit je 2 Kernen

2. ITS Cluster:

   Linux Enterprise , 64 GB RAM, 12 Prozessoren 2,6 Ghz mit je 6 Kernen

   Es wurden die jeweils aktuellsten Versionen 18.0 (Erlang) und 2.4.5 (X10) der Programmiersprachen verwendet.

6.1 Monte Pi

Die folgenden Tabellen zeigen zum Einen die Abweichungen der Monte Pi Anwendungen von der zu errechnenden Zahl π, in Abhängigkeit von der gewählten Anzahl möglicher Versuche. Des Weiteren werden die, mit unterschiedlicher Prozessanzahl, erfassten Ausführungszeiten der Erlang Anwendung dargestellt. Die Ausführungszeiten der Implementierung in X10 wurde ausschließlich mit einer Versuchsanzahl von einer Millarde und 1 bis 10 Places erfasst, da selbst mit dieser weitaus höheren Anzahl, eine niedrigere Ausführungszeit erreicht werden konnte.

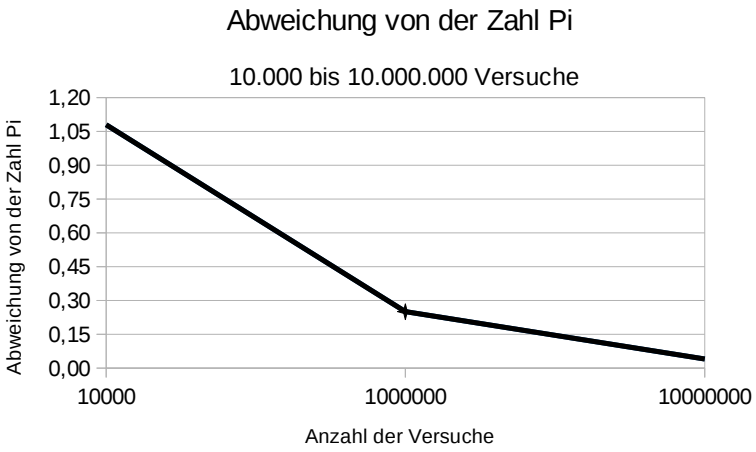

Abbildung 3: Abweichungen von π Monte Pi Algorithmus

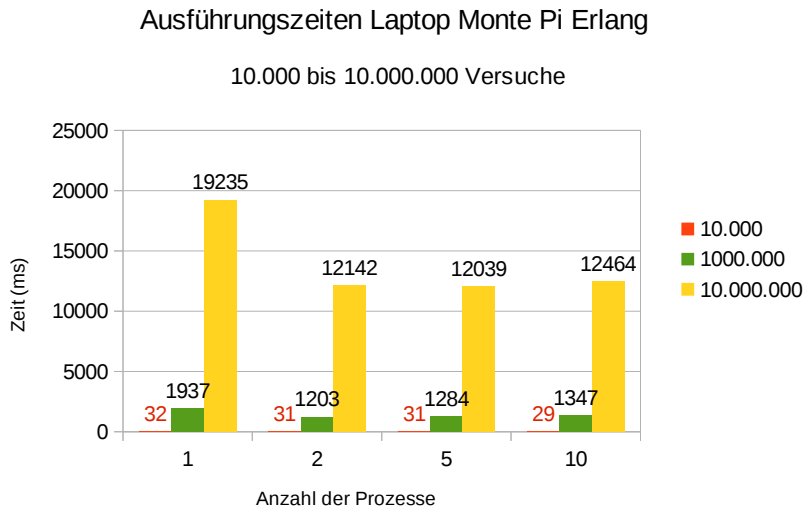

Abbildung 4: Ausführungszeiten Monte Pi mit Erlang (Laptop)

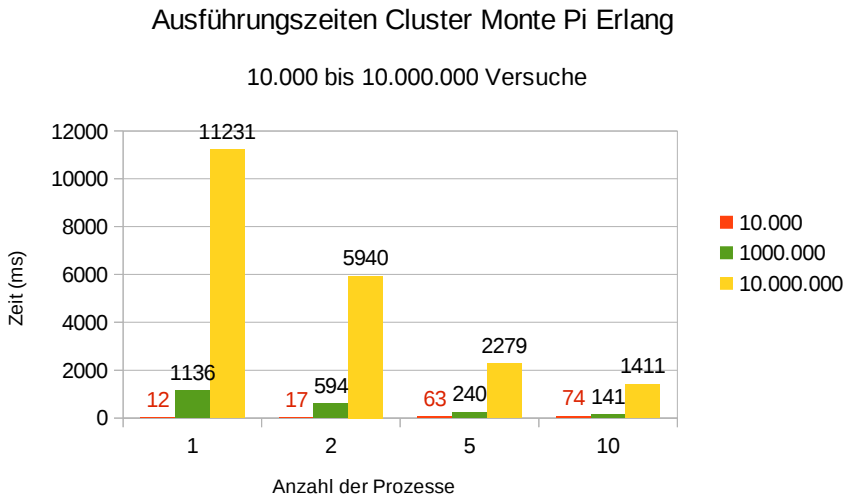

Abbildung 5: Ausführungszeiten Monte Pi mit Erlang (Cluster)

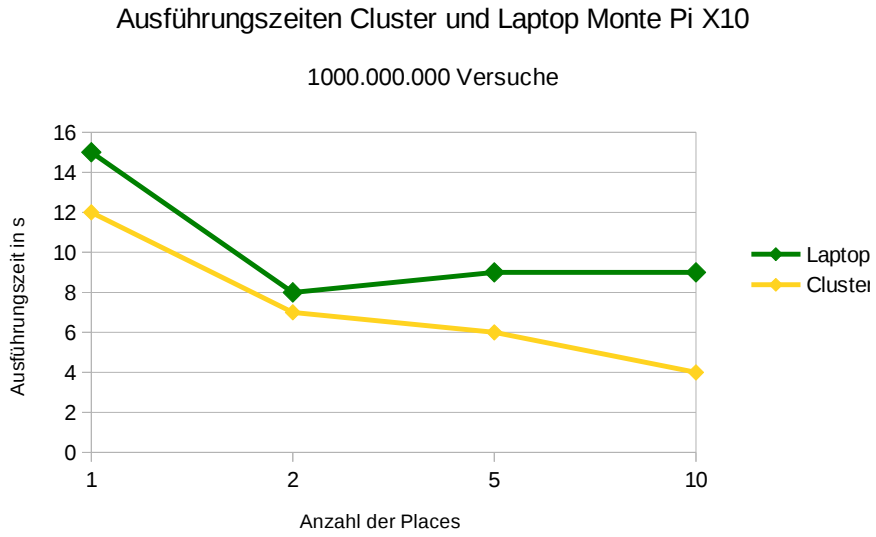

Abbildung 6: Ausführungszeiten Monte Pi mit X10 (Laptop und Cluster)

Die zur Fehlerbehandlung benötigte Zeit wurde ebenfalls bestimmt. Im Rahmen dieser Experimente wurde jeder zweite Prozess mittels Error-Seeding[5] zum Absturz gebracht. Der Fehler wurde direkt nach der Erzeugung des Prozesses ausgelöst. Die Experimente wurden wie folgt durchgeführt:

[5]Ein, zu Testzwecken, absichtlich eingebauter Fehler führt Absturz der Anwendung

1. Anwendungen ohne Error-Seeding: 100 Prozesse, Berechnete Punkte: 1.000.000

   Erlang: Dauer 2,24 s
   X10: Dauer 4,61 s
   Zeitdifferenz: 2,4 s
   Prozessverwaltung in Erlang, im Rahmen dieser Experimente effizienter.

2. Anwendungen mit Error-Seeding: 200 Prozesse, Berechnete Punkte: 1.000.000

   Erlang: Dauer 2,26 s
   X10: Dauer 11,24 s
   Zeitdifferenz: 9 s
   Prozessverwaltung mit Fehlerbehandlung in Erlang, im Rahmen dieser Experimente effizienter.

Hier zeigten sich die Vorteile der Sprache Erlang bezüglich der Prozessverwaltung und Fehlerbehandlung. Die Verwaltung der doppelten Anzahl Prozesse und die Behandlung der eingebauten Fehler dauerte weniger als 100 ms, die X10-Anwendung benötigte zur Fehlerbehandlung einige (6) Sekunden länger.

Der Aufwand zur Programmierung ist bei beiden Sprachen gleich hoch anzusetzen.

6.2 K-Means

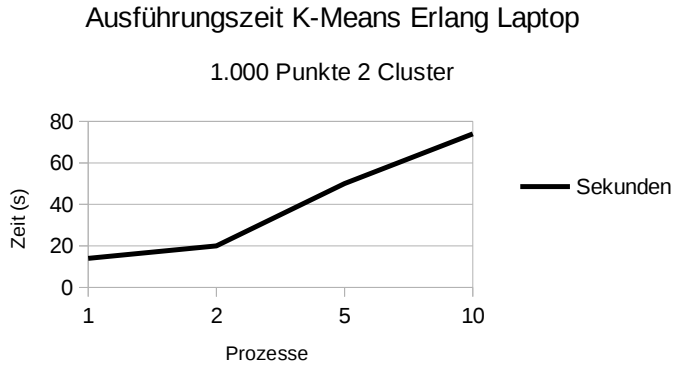

Abbildung 7: Ausführungszeiten K-Means mit Erlang (Laptop)

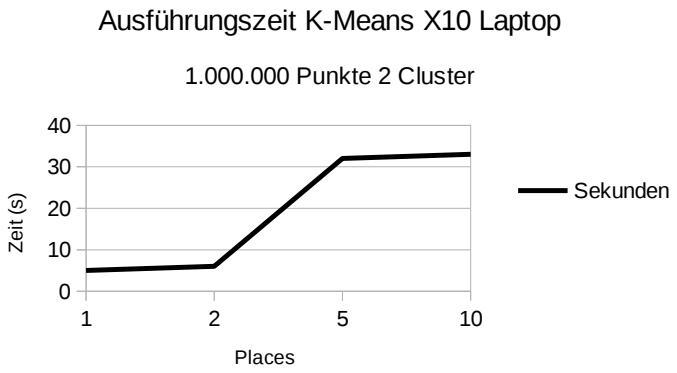

Abbildung 8: Ausführungszeiten K-Means mit X10 (Laptop)

Die zur Fehlerbehandlung benötigte Zeit wurde ebenfalls bestimmt. Im Rahmen dieser Experimente wurde jeder zweite Prozess mittels Error-Seeding zum Absturz gebracht. Die Experimente wurden wie folgt durchgeführt:

1. Anwendungen ohne Error-Seeding: 5 Prozesse, Berechnete Punkte: 1.000

   Erlang: Dauer 50 s
   X10: Dauer 32 s
   Zeitdifferenz: 18 s

2. Anwendungen mit Error-Seeding: 10 Prozesse, Berechnete Punkte: 1.000

   Erlang: Dauer 63 s
   X10: Dauer 34 s
   Zeitdifferenz: 29 s

Der Aufwand zur Programmierung ist in diesem Fall in Erlang höher, der Grund ist die Eignung einer imperativen Sprache, wie X10, für derartige Anwendungsfälle. Der Einsatz der `for(...)` Schleife nimmt dem Programmierer viel Arbeit ab.

6.3 HeatTransfer

Aus zeitlichen Gründen konnten nur die Ausführungszeit und die zur Fehlerbehandlung benötigte Zeit des Erlang-Programmes erfasst werden.

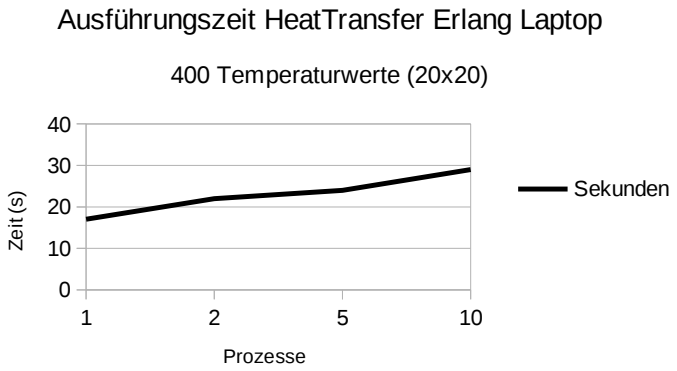

Abbildung 9: Ausführungszeiten HeatTransfer mit Erlang (Laptop)

Die zur Fehlerbehandlung benötigte Zeit wurde ebenfalls bestimmt. Im Rahmen dieser Experimente wurde jeder zweite Prozess mittels Error-Seeding zum Absturz gebracht. Die Experimente wurden wie folgt durchgeführt:

1. Anwendung ohne Error-Seeding:

   5 Prozesse, Berechnete Temperaturwerte: 400, Iterationen: 10

   Dauer: 24 s

2. Anwendung mit Error-Seeding:

   10 Prozesse, Berechnete Temperaturwerte: 400, Iterationen: 10

   Dauer: 25 s

   Auch hier zeigt sich die effiziente Prozessverwaltung und Fehlerbehandlung mit Erlang.

7 Zusammenfassung

Im Rahmen dieser Bachelorarbeit wurden die Algorithmen MontePi, K-Means und HeatTransfer, mit verschiedenen Ansätzen zur Fehlerbehandlung implementiert. Verwendet wurden die Programmiersprachen Erlang und X10, wobei die X10-Programme zum Vergleich bereits vorlagen. Die Erlang-Anwendungen wurden selbst implementiert.

Anschließend konnten die Laufzeiten der Programme, mit und ohne Fehlerbehandlung, verglichen werden. Dieser Performancevergleich fiel klar für X10 aus. Die Stärken der Sprache Erlang lagen in den Bereichen Prozessverwaltung und Fehlerbehandlung, besonders hervorgetreten im Vergleich der Ausführungszeiten des Monte Pi Algorithmus mit 10 und 200 Prozessen. Vor allem die Konzepte Links und Monitor zum Prozessmonitoring und die in Erlang Nachrichten-basierte IPC traten positiv hervor. Die Erzeugung der Prozesse, sowie das Reagieren auf einen Absturz ist in Erlang intuitiv umsetzbar.

Die Entwicklung mit der funktionalen Programmiersprache Erlang gestaltete sich anfangs ungewohnt und der Einstieg erwies sich als schwierig. Die Verwendung der Entwicklungsumgebung "Eclipse", mit Erlang-plugin "Erlide" war dabei sehr hilfreich, die Syntax-Prüfung ist hier besonders hervorzuheben.

Der Entwurf fehlertoleranter Anwendungen stellte sich generell als schwierig heraus. Es Bedarf einerseits der Kenntnis über das verwendete System bzw. die Programmiersprache, andererseits müssen genaue Kenntnisse über den zu implementierenden Algorithmus vorhanden sein, da dieser ggf. zur Fehlerbehandlung erweitert werden muss. Ein Beispiel wäre die Verteilung der gesamten Daten auf alle Prozesse, um Datenverlust bei einem Prozessabsturz vorzubeugen (s. 4.1).

Listings

Abbildungsverzeichnis

Literatur

[1] OpenMP. The OpenMP: API for Parallel Programming. http://www.openmp.org.

[2] MPI Forum. Message passing interface forum. http://www.mpi-forum.org.

[3] Erlang. Erlang documentation. http://www.erlang.org.

[4] D. Cunningham, D. Grove, B. Herta, A. Iyengar, K. Kawachiya, H. Murata, V. Saraswat, M. Takeuchi, O. Tardieu. Resilient X10. pages 67–80, Orlando, Florida, USA, 2014. Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming.

[5] Hadoop. Apache hadoop. http://www.hadoop.org.

[6] Kiyokuni Kawachiya. Writing Fault-Tolerant Applications Using Resilient X10. Technical report, IBM Research, Tokyo, 2015.

[7] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007.

[8] IBM. X10 website. http://www.x10-lang.org.