UNIKASSEL
VERSITÄT

Evaluierung der Fehlertoleranz in MPI / ULFM anhand von Beispielprogrammen

Bachelorarbeit

im Bachelorstudiengang Informatik am
Fachbereich 16 Elektrotechnik / Informatik
Fachgebiet Programmiersprachen / -methodik

Mario Richter
Kassel, 14. Dezember 2015

Erstgutachterin: Prof. Dr. Claudia Fohry

Zweitgutachter: Prof. Dr. Albert Zündorf

Erklärung

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

Kassel, den 14. Dezember 2015 _____

Inhaltsverzeichnis

1 Einleitung

Aufgrund des Bestrebens nach immer schnelleren Rechnens im Bereich des HPC (High Performance Computing), stieg die Rechenleistung von Supercomputern in den letzten Jahren stetig an. So hat sich etwa die Rechenleistung des schnellsten Supercomputers der Welt von Juni 2006 mit 2,4 TFlop/s auf 33,9 PFlop/s im Juni 2015 vervielfacht [9]. Da die Leistung einer einzelnen CPU bzw. GPU durch physikalische Grenzen jedoch stark begrenzt ist, werden beim HPC Mehrkern CPUs und GPUs verwendet und zusammengefasst.

Dies erfordert allerdings parallele Programmiersprachen, da sequentielle Programme Rechner mit mehreren Kernen nicht effizient auslasten können. Parallele Programmiersprachen müssen dabei die Möglichkeit der Kommunikation zwischen den verschiedenen Threads bieten. Sprachen wie z.B. X10, OpenMP oder MPI haben dabei unterschiedliche Konzepte entwickelt, so bietet OpenMP eine "shared Memory" genannte Möglichkeit. Dabei verwenden die einzelnen Threads einen gemeinsamen Adressraum, auf den alle Prozesse zugreifen können. Bei der Sprache X10 kommt "Asynchronous Partioned Global Adress Space" (AGPAS) zum Einsatz. Hier gibt es einen globalen Adressraum, welcher in Places eingeteilt wird und anschließend den Threads zur Verfügung steht. Die Kommunikation über Nachrichtenaustausch der einzelnen Threads zu realisieren stellt das Message Passing Interface (MPI) zur Verfügung.

Im Vordergrund dieser Arbeit stehen MPI und X10, insbesondere deren Fehlertoleranz. Weder MPI noch X10 bieten in ihrem Standard Möglichkeiten zur Fehlerbehandlung. Fällt ein Prozess bzw. ein Rechenknoten aus, führt dies in der Regel zum Absturz des gesamten Programms. Durch den starken Zuwachs an Rechenleistung von Supercomputern, werden immer mehr CPUs und GPUs zusammengefasst. Nimmt man eine Mean Time Between Failures (kurz MTBF) bei einer Recheneinheit von 100 Jahren an, sinkt diese auf nur ein Jahr herab, wenn man 100 Recheneinheiten zu einem System zusammenfasst. Bei einem Supercomputer mit 100.000 Recheneinheiten sinkt die MTBF auf acht Stunden, was drei Ausfälle pro Tag bedeutet [10]. Dies macht den Einsatz von Erweiterungen zur Fehlerbehandlung sinnvoll, um im Falle eines Ausfalls nicht alle Daten zu verlieren und dementsprechend Ressourcen sparen zu können. Bei MPI wird hierfür die Erweiterung "User Level Failure Mitigation" (ULFM) [8] entwickelt, bei X10 gibt es die Erweiterung Resilient X10 [3].

In [4] werden drei Möglichkeiten im Umgang mit Fehlern beschrieben:

1. Ignoranz - Hierbei werden die Fehler bzw. die Prozessausfälle ignoriert und der Verlust der entsprechenden Daten in Kauf genommen.
2. Verteilung der restlichen Arbeit auf die nach einem Ausfall verbliebenen Prozesse
3. Checkpointing – Periodische Sicherung aller wichtigen Daten, Wiederherstellung der Daten im Fehlerfall

Um diese drei Möglichkeiten zu veranschaulichen, wurden im Rahmen der vorliegenden Arbeit folgende drei Beispielprogramme gewählt:

- MontePi für das Ignorieren von Fehlern
- KMeans für die Übernahme der Arbeit durch die verbliebenen Prozesse
- Heat-Transfer für die Demonstration des Checkpointing

Diese drei Beispielprogramme wurden bereits von Herrn Schaub im Rahmen seiner Bachelorarbeit in Erlang implementiert und mit Resilient X10 verglichen [6]. Während Resilient X10 im Performancebereich punkten konnte, zeigten sich bei Erlang Vorteile bei der Prozessverwaltung und Fehlerbehandlung. Das Ziel dieser Arbeit ist die Implementierung der Algorithmen in der Programmiersprache C mit MPI und ULFM mit anschließendem Vergleich zu den Resilient X10 Implementierungen.

In Kapitel 2 werden zunächst die Grundlagen zur parallelen und fehlertoleranten Programmierung erläutert, darauf folgen die Beispielprogramme in den Kapiteln 3, 4 und 5. In Kapitel 6 folgt eine Präsentation der Ergebnisse hinsichtlich Fehlertoleranz, Programmieraufwand und Laufzeitgeschwindigkeit. Abschließend enthält Kapitel 7 eine Zusammenfassung.

2 Grundlagen parallele und fehlertolerante Programmierung

In diesem Kapitel werden zunächst die Grundlagen zur parallelen und fehlertoleranten Programmierung der jeweiligen Systeme erläutert. Dies betrifft im Rahmen der vorliegenden Arbeit die Programmiersprache X10 bzw. Resilient X10 und MPI / ULFM. Da die Quellcodes für die Beispielprogramme für X10 bereits aus [11] vorliegen, liegt der Fokus auf MPI, insbesondere ULFM.

2.1 X10

Die parallele Programmiersprache X10 wird hauptsächlich für den HPC Bereich entwickelt. Sie ist für bis zu 100.000 Hardware-Threads [5] ausgelegt und somit stark auf Parallelität ausgerichtet. IBM entwickelt X10 seit 2004 unter der Eclipse Public License 1.0, sodass sich auch Universitäten an der Entwicklung beteiligen [14].

X10 ist objektorientiert und klassenbasiert, neben Einfachvererbung bietet die Sprache auch einen Garbage Collector. Ihre Syntax ist stark an der weitverbreiteten Programmiersprache Java angelehnt. Durch die Verwendung einer asynchronen Taskverwaltung[1] erweitert die Sprache das Programmiermodell PGAS zu APGAS. Bei dem APGAS Modell von X10 wird der globale Adressbereich logisch unterteilt, sodass jeder Prozess einen eigenen lokalen Adressbereich zugeteilt bekommt. Teile dieses Adressbereichs können als „privat“ deklariert werden, sodass andere Prozesse auf diesen Adressbereich nicht zugreifen können. Auf nicht-„privat“ deklarierte Adressbereiche hingegen können alle anderen Prozesse zugreifen, wobei der entfernte Zugriff langsamer ist als der lokale Zugriff [15].

Ein Rechenknoten wird in X10 „Place“ genannt, auf denen wiederum „Activities“ ausgeführt werden können. Activities sind leichtgewichtige Threads, mit dem `at` Statement können diese auf einem Place ausgeführt werden. Listing 1 zeigt ein kurzes Hello World Beispiel: In Zeile 2 wird das Hauptprogramm `main` deklariert, welches auf Place 0 läuft. Das Schlüsselwort `finish` in Zeile 3 dient als Barriere am Ende des Blocks (Zeile 7), sodass der

[1] Eine Task ist eine Menge gleichwertiger Threads

Programmierer sicher sein kann, dass alle Threads diesen Punkt erreicht haben, bevor Zeile 8 ausgeführt wird. Die For-Schleife `for (pl in Place.places())` iteriert über alle Places, auf denen dann in Zeile 4 Activities gestartet werden (`at (pl)`). Das Schlüsselwort `async` in Zeile 4 startet die Activities asynchron. In Zeile 6 folgt schließlich die Ausgabe jedes Activities auf der Konsole.

Listing 1: Hello World in X10, Quelle: [4] Seite 2

```
1  class HelloWorld {
2    public static def main(args:Rail[String]) {
3      finish for (pl in Place.places()) {
4        at (pl) async { // parallel distributed exec in each place
5          Console.OUT.println( " Hello from " + here );
6        }
7      } // end of finish, wait for the execution in all places
8    }
9  }
```

2.2 Resilient X10

Fehlertolerante Programme können mit Resilient X10 ausgeführt werden. Resilient X10 ist in der Version 2.5 von X10 enthalten (Stand Dezember 2015), der Code kann normal kompiliert werden. Lediglich bei der Ausführung ist es nötig, dass die Umgebungsvariable `X10_RESILIENT_MODE` einen von Null verschiedenen Wert besitzt (z.B. Eins für „Most stable resilient mode“) [16].

In Resilient X10 wird im Falle des Ausfalls eines Places eine `DeadPlaceException` geworfen. Mittels einer `try – catch` Anweisung ist es möglich, solche Ausnahmen abzufangen. Listing 2 enthält ein einfaches fehlertolerantes Programm. In Zeile 4 beginnt der `try` Block, welcher in Zeile 6 endet. Sollte es in diesem Block zu einer Ausnahme vom Typ `DeadPlaceException` kommen, wird diese von der `catch` Anweisung in Zeile 6 abgefangen. Im `catch` Block erfolgt die Fehlerbehandlung. In diesem Beispiel erfolgt lediglich die Ausgabe, welcher Place ausgefallen ist.

Listing 2: Minimales fehlertolerantes Programm in Resilient X10, Quelle [4] Seite 2

```
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 }
```

2.3 Messsage Passing Interface

Das Message Passing Interface (MPI) beschreibt einen Standard zum Nachrichtenaustausch zwischen Prozessen. Dieser Standard ist als Programmierschnittstelle zu verstehen, in ihm werden Operationen und ihre Semantik definiert. Die Entwicklung von MPI begann 1992, die Version MPI 1.0 erschien 1994 [17]. Die zurzeit aktuellste Version ist MPI-3.1 vom 4. Juni 2015 [13] (Stand Dezember 2015). Die Entwicklung von MPI wird von Firmen, Forschungseinrichtungen und Universitäten vorangetrieben.

Als Implementierungen stehen MPICH [19], OpenMPI [20], MVAPICH [21] und weitere zur Verfügung. MPICH konzentriert sich auf hohe Performance und hohe Portabilität. OpenMPI zählt zusammen mit MPICH zu den beiden meist verbreiteten Implementierungen und wird z.B. auf dem ITS Cluster der Universität Kassel eingesetzt. MVAPICH bietet z.B. Kommunikation über InfiniBand und steht ebenfalls auf dem ITS Cluster zur Verfügung.

Im Folgenden werden einige wichtige Begriffe und Funktionen von MPI erläutert [22], welche auch für ULFM wichtig sind:

2.3.1 Kommunikatoren und Gruppen

Kommunikatoren spielen eine zentrale Rolle in MPI und ULFM. So ist etwa jegliche Art von Kommunikation zwischen Prozessen ausschließlich über Kommunikatoren möglich. Jedem Kommunikator liegt eine Gruppe zugrunde, welche wiederrum aus Prozessen besteht. Ein Prozess kann Mitglied in mehreren Gruppen und Kommunikatoren sein. Er verfügt in jeder Gruppe bzw. jedem Kommunikator über eine Nummer (genannt Rang), welche über die Funktion `MPI_Group_rank(gruppe, &meinRang)` für Gruppen, bzw. `MPI_Comm_rank(komm, &meinRang)` für Kommunikatoren abrufbar ist. `gruppe` und

komm sind jeweils der Name der Gruppe bzw. des Kommunikators, in der Variable meinRang wird der Rang des entsprechenden Prozesses gespeichert. Die Anzahl von Prozessen in einer Gruppe bzw. einem Kommunikator lässt sich über die Funktion MPI_Group_size(gruppe, &anzProzesse), bzw. MPI_Comm_size(komm, &anzProzesse) ermitteln. Kommunikatoren können zusätzlich über Attribute sowie einen Errorhandler verfügen. Über die Funktion MPI_Comm_create(komm, gruppe, neuerKomm) kann ein neuer Kommunikator erzeugt werden. Dabei ist komm der ursprüngliche Kommunikator, gruppe eine Untermenge (Gruppe) von komm und neuerKomm der erzeugte Kommunikator. Der zugehörige Destruktor zum Löschen eines Kommunikators lautet MPI_Comm_free(komm). MPI_COMM_WORLD ist ein vordefinierter Intrakommunikator, welcher die Menge aller Prozesse bezeichnet. Intrakommunikatoren sind für die Kommunikation innerhalb einer Prozessgruppe zuständig, während Interkommunikatoren die Kommunikation zwischen Prozessgruppen ermöglichen.

2.3.2 Punkt-zu-Punkt Kommunikation

Die Kommunikation in MPI erfolgt über das Senden und Empfangen von Nachrichten zwischen zwei Prozessen. Dabei sendet ein Prozess eine Nachricht an einen anderen Prozess, welcher sich im selben Intrakommunikator befinden muss. Da dieser Themenkomplex sehr umfangreich ist, wird an dieser Stelle nur auf einen Teil der Funktionen eingegangen.

Bei dem Nachrichtenaustausch unterscheidet man zwischen blockierender und nichtblockierender Kommunikation. Bei der blockierenden Variante wartet der Sender, bis der Empfänger die zugehörige Empfangsfunktion aufgerufen hat. Bei der nichtblockierenden Variante erhält der Sender lediglich ein Request-Objekt, anhand dessen er den Status der Nachrichtenübertragung prüfen und sofort weiterrechnen kann. Eine Sendeoperation ist wie folgt definiert: MPI_Send(* puffer, anzahl, datentyp, ziel, tag, komm). puffer beschreibt dabei die Anfangsadresse des Sendepuffers, anzahl ist folglich Anzahl der Elemente des Sendepuffers. datentyp beschreibt den Datentyp des Sendepuffers. Der Rang des Empfängers wird in ziel abgelegt, tag ist eine vom Programmierer festzulegende Markierung der Nachricht. komm legt den Kommunikator der Prozessgruppe fest, z.B. MPI_COMM_WORLD. Analog erfolgt das Senden einer nichtblockierenden Nachricht, hier

wird jedoch zusätzlich ein Zeiger auf ein `Request`-Objekt mit übergeben. Anhand dieses Zeigers kann z.B. mit `MPI_Test(MPI_Request* request, int* flag, MPI_Status* status)` der Fortschritt der Operation abgefragt werden (`flag=1` wenn die Operation abgeschlossen ist).

Das Empfangen von Nachrichten erfolgt analog, z.B. `MPI_Irecv(...)` für das nichtblockierende Empfangen einer Nachricht.

2.3.3 Kollektive Operationen

In parallelen Anwendungen benötigt man häufig Kommunikationsmuster, an denen mehrere oder alle Prozesse beteiligt sind. Im Folgenden werden die wichtigsten kollektiven Operationen vorgestellt.

Eine häufig benötigte Funktion ist die Synchronisation. Mit `MPI_Barrier(MPI_Comm komm)` blockiert der aufrufende Prozess solange, bis alle Prozesse aus `komm` diese Routine aufgerufen haben. Eine weitere Funktion ist der Broadcast: `MPI_Bcast(void *puffer, int anzahl, MPI_Datatype datentyp, int root, MPI_Comm komm)`. Der Prozess mit dem Rang `root` sendet seine Daten aus `puffer` an alle anderen Prozesse des Kommunikators, wobei alle Prozesse diese Funktion aufrufen müssen. Des Weiteren gibt es die Möglichkeit des Einsammelns von Daten aller beteiligten Prozesse durch `root` mit der `MPI_Gather`-Funktion. Auch eine Verknüpfung mit logischen oder arithmetischen Operationen der eingesammelten Daten ist mit `MPI_Reduce` möglich.

Listing 3 zeigt ein einfaches MPI Programm in C. In Zeile 5 wird die MPI Umgebung initialisiert. Die Zeilen 6 und 7 bestimmen den Rang des jeweiligen Prozesses sowie die Anzahl aller Prozesse. In Zeile 8 folgt dann eine Ausgabe von jedem Prozess, anschließend wird MPI in Zeile 9 beendet.

Listing 3: Hallo Welt in C mit MPI

```
1  #include <mpi.h>
2  #include <stdio.h>
3  int main (int argc, char* argv[]) {
4    int meinRang, anzahlProzesse;
5    MPI_Init (&argc, &argv);
6    MPI_Comm_rank (MPI_COMM_WORLD, &meinRang);
7    MPI_Comm_size (MPI_COMM_WORLD, &anzahlProzesse);
8    printf( "Hallo Welt von Prozess %d von %d\n", meinRang, anzahlProzesse);
9    MPI_Finalize();
10   return 0;
11 }
```

2.4 User Level Failure Mitigation

Bisher wird Fehlertoleranz für MPI Anwendungen auf der System Level Ebene praktiziert. Bei System Level Fehlertoleranz wird mittels Checkpointing der Speicherbereich der einzelnen Prozesse automatisch periodisch gesichert und im Bedarfsfall wiederhergestellt [2]. Dabei muss der Programmierer die Ausfallsicherheit bei der Entwicklung von Anwendungen nicht berücksichtigen. Im Fall eines Ausfalls wird die Anwendung beendet und durch das System vom letzten erfolgreichen Checkpoint neugestartet. Das Sichern des gesamten Speicherbereichs führt jedoch zu einem hohen Overhead: Je geringer die MTBF eines Clusters ist, desto höher ist die Checkpoint- und Neustartfrequenz und damit der Overhead [18]. Dies erfordert neue Möglichkeiten im Umgang mit Ausfällen, gerade bei Hochleistungsrechnern mit einer geringen MTBF. Ein Ansatz hierfür ist die User Level Fehlertoleranz. Bei diesem Ansatz muss der Programmierer mittels bereitgestellter Funktionen Ausfälle selbst behandeln[2] und ist für die Sicherheit von wichtigen Daten verantwortlich.

Um diesen Anforderungen gerecht zu werden, wird von der Fault Tolerance Working Group des MPI Forums User Level Failure Mitigation (kurz ULFM) entwickelt. Dabei werden folgende Ziele verfolgt [8]:

1. Einfachheit: Die API soll leicht verständlich und von Programmierern einfach einzusetzen sein.

2. Flexibilität: ULFM soll als Basis für weitere fehlertolerante Modelle dienen können, z.B. Fenix [18].

[2] D.h. im Gegensatz zur System Level Fehlertoleranz wird die von einem Ausfall betroffene Anwendung nicht automatisch beendet

3. Keine Deadlocks: MPI Funktionen dürfen keinen Deadlock verursachen, eine Funktion muss immer einen Wert zurückgeben, entweder einen Erfolg (`MPI_SUCCESS`) oder einen Fehler, siehe 2.4.2.

ULFM wird zurzeit noch entwickelt und ist aktuell in der Version 1.1 (Stand Dezember 2015) verfügbar. Im MPI Forum wird ULFM mit dem Ziel der Integration in den künftigen MPI 4.0 Standard diskutiert [23], [24]. Im Folgenden werden einige Grundkonzepte erläutert [1]:

2.4.1 Fehlererkennung

Der Ausfall eines Threads wird von MPI Funktionen erkannt und einem oder mehreren Threads mitgeteilt. Die Fehlererkennung erfolgt grundsätzlich nur lokal, da nicht immer alle Prozesse von jedem anderen abhängig sind und somit nicht immer jeder Prozess über den Ausfall informiert werden muss. Welche Prozesse informiert werden, hängt davon ab, ob sie mit dem ausgefallenen Prozess über die Funktion, welche den Fehler erkennt, in Kommunikation stehen. Hierfür muss eine der folgenden Bedingungen erfüllt sein [1]:

- Es handelt sich um eine kollektive Funktion und beide[3] Prozesse befinden sich im selben Kommunikator.
- Beide Prozesse sind Teil einer Punkt-zu-Punkt Kommunikation, entweder als Ziel oder als Quelle.
- Die Funktion ist eine `MPI_ANY_SOURCE` Empfangsoperation und der ausgefallene Prozess ist der sendenden Gruppe zugeordnet.

Dementsprechend erfolgt eine globale Fehlererkennung lediglich in Sonderfällen, etwa bei einer kollektiven Funktion über einen Kommunikator mit allen Prozessen. All jene Funktionen, an denen kein ausgefallener Prozess beteiligt ist, dürfen auch keinen Fehler erzeugen. Kommunikationsfunktionen müssen ihre eigentliche Aufgabe nicht erfüllen, wenn sie einen Fehler berichten. So ist etwa der Empfangspuffer einer Nachricht undefiniert oder eine Synchronisation ist möglicherweise nicht erfolgt.

2.4.2 Fehlerklassen

In ULFM wurden folgende drei Fehlerklassen definiert:

[3] Mit „beide“ sind der ausgefallene und der (die) informierte (n) Prozess(e) gemeint

- `MPI_ERR_PROC_FAILED`: Die Funktion konnte aufgrund eines Prozessausfalls nicht ausgeführt werden. Kollektive Funktionen erzeugen immer diese Fehlerklasse wenn ein Ausfall vorliegt. Bei Punkt-zu-Punkt Kommunikation wird ebenfalls diese Fehlerklasse erzeugt, es gibt jedoch eine Ausnahme (siehe nächsten Punkt `MPI_ERR_PROC_FAILED_PENDING`). Sollte ein `Request`-Objekt einer Punkt-zu-Punkt Kommunikation angehörig sein, so besitzt dieses den Status abgeschlossen.
- `MPI_ERR_PROC_FAILED_PENDING`: Die Funktion wurde aufgrund eines Prozessausfalls unterbrochen, kann aber später noch beendet werden. Diese Fehlerklasse zeigt bei einer nicht-blockierender Empfangsoperation von `MPI_ANY_SOURCE` an, dass es keine passende Sendeoperation gibt. Dies ist ein Indiz dafür, dass der Sendeprozess abgestürzt ist. In diesem Fall ist weder die Empfangsoperation, noch das `Request`-Objekt abgeschlossen und können später noch beendet werden.
- `MPI_ERR_REVOKED`: Das von der Funktion genutzte Kommunikationsobjekt wurde widerrufen (z.B. der Kommunikator bei kollektiven Operationen). Mehr zu dieser Fehlerklasse im Punkt 2.4.3.

2.4.3 Fehlerbehandlung

Da eine globale Fehlererkennung nicht garantiert wird, erhalten lediglich jene Prozesse einen Fehlercode, welche in die Kommunikation mit einem abgestürzten Prozess verwickelt sind. Ein solcher Prozess kann bei Bedarf andere oder alle Prozesse über einen Fehler informieren. Mit der Funktion `MPI_Comm_revoke(MPI_Comm komm)` wird der Kommunikator `komm` widerrufen. Diese Funktion muss nur von einem Prozess aufgerufen werden, darf aber auch von beliebig vielen Prozessen aufgerufen werden, was jedoch keinen Unterschied macht. Alle nicht-lokalen MPI Operationen, welche `komm` als Kommunikator nutzen, erzeugen eine Ausnahme der Klasse `MPI_ERR_REVOKED`. Dadurch werden alle zu `komm` gehörenden Prozesse informiert. Nicht kommunizierende Prozesse oder nur lokal kommunizierende Prozesse werden demnach nicht über einen Ausfall informiert. Um auch solche Prozesse zu informieren, muss der Programmierer an geeigneter Stelle eine kollektive Funktion (z.B. `MPI_Barrier(komm)` aufrufen. Im Anschluss kann die Fehlerbehandlung beginnen:

Durch das Aufrufen der kollektiven Funktion `MPI_Comm_Shrink(MPI_Comm komm, MPI_Comm* neuerKomm)` wird einer neuer Kommunikator erzeugt. Dieser enthält alle noch lebenden Prozesse aus `komm`, sofern alle

noch lebenden Prozesse diese Funktion aufrufen. Rufen einige noch lebende Prozesse `MPI_Comm_Shrink` nicht auf, sind sie im neuen Kommunikator nicht mehr enthalten. Die Eigenschaft Intra- oder Interkommunikator wird ebenfalls von `neuerKomm` übernommen. Dies ist die übliche Vorgehensweise, um ausgefallene Prozesse zu eliminieren.

Eine weitere Möglichkeit ist das Bestätigen eines Prozessausfalls mit `MPI_Comm_failure_ack(MPI_Comm komm)`. `MPI_ANY_SOURCE` Empfangsoperationen, welche einen Fehler der Klasse `MPI_ERR_PROC_FAILED_PENDING` erzeugen würden, fahren anschließend ohne Fehlermeldung mit der Ausführung fort. Somit werden durch den Aufruf von `MPI_Comm_failure_ack` Fehler ignoriert, der Kommunikator bleibt dabei jedoch defekt. Kollektive Funktionen erzeugen jedoch nach wie vor Fehlermeldungen, z.B. der Klasse `MPI_ERR_PROC_FAILED` nach einem Prozessausfall. Durch den Aufruf der Funktion `MPI_Comm_failure_get_acked(MPI_Comm komm, MPI_Group* ausgefalleneProzesse)` erhält man eine Gruppe von Prozessen, deren Ausfall mit `MPI_Comm_failure_ack` bestätigt wurde.

Um zu überprüfen, ob alle Prozesse den Ausfall eines anderen Prozesses bestätigt haben, kann die Funktion `MPI_Comm_agree(MPI_Comm komm, int* flag)` verwendet werden. Mit dieser Funktion kann darüber hinaus abgefragt werden, ob alle anderen Prozesse noch leben. Sollte ein Prozess vor oder während des Aufrufs von `MPI_Comm_agree` ausgefallen sein (und nicht zuvor mit `MPI_Comm_failure_ack` bestätigt worden sein), erzeugt die Funktion einen Fehler der Klasse `MPI_ERR_PROC_FAILED`. Anschließend bestätigt diese den Fehler mit `MPI_Comm_failure_ack`. Nach dem Aufruf kann der Programmierer sicher sein, dass alle Prozesse entweder `MPI_SUCCESS` oder `MPI_ERR_PROC_FAILED` zurückgeben, alle Prozesse verfügen somit über einen konsistenten Status. Analog existiert eine nicht-blockierende Variante `MPI_Comm_iagree(MPI_Comm komm, int* flag, MPI_Request* anfrage)`.

`MPI_Comm_Shrink` und `MPI_Comm_(i)agree` dürfen auch auf Kommunikatoren angewendet werden, welche zuvor mittels `MPI_Comm_revoke` widerrufen wurden. Beide erzeugen in diesem Fall keinen Fehler der Klasse `MPI_ERR_REVOKED`.

Listing 4 zeigt ein einfaches fehlertolerantes Programm (Auszug) in C mit MPI und ULFM. In Zeile 1 wird die für Fehlertoleranz benötigte Bibliothek importiert. Zeile 8 zeigt das

Ändern des Errorhandlers von `MPI_ERRORS_ARE_FATAL` auf `MPI_ERRORS_RETURN`, dies ist nötig, da der Standard Errorhandler bei einem Prozessausfall das Programm beenden würde. In Zeile 9 wird ein Prozess absichtlich zum Absturz gebracht, dies wird in Zeile 10 bemerkt. Der Fehlercode wird in `rc` gespeichert, eine weitere Fehlerbehandlung folgt nicht. Es wird lediglich eine Fehlermeldung in Zeile 12 ausgegeben. In diesem Fall bemerken alle Prozesse den Fehler, da `MPI_Barrier` kollektiv ist und `MPI_COMM_WORLD` alle Prozesse enthält.

Listing 4: Ein einfaches fehlertolerantes MPI Programm in C, Auszug aus Quelle [10]

```
1  #include <mpi-ext.h>
2  int main(int argc, char *argv[]) {
3    int rank, size, rc, len;
4    char errstr[MPI_MAX_ERROR_STRING];
5    MPI_Init(NULL, NULL);
6    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
7    MPI_Comm_size(MPI_COMM_WORLD, &size);
8    MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
9    if( rank == (size-1) ) raise(SIGKILL);
10   rc = MPI_Barrier(MPI_COMM_WORLD);
11   MPI_Error_string(rc, errstr, &len);
12   printf("Rank %d / %d: Notified of error %s. Stayin' alive!\n", rank,
13     size, errstr);
14   MPI_Finalize();
15 }
```

3 Beispielprogramm Monte Pi

In diesem Beispielprogramm wird mit Hilfe eines Monte-Carlo-Algorithmus die Zahl Pi annäherungsweise berechnet. Dazu wird zunächst ein Kreis mit dem Radius $r = 1$ um den Mittelpunkt eines Quadrates mit der Kantenlänge $a = 2$ gelegt. Anschließend werden zufällige Punkte in diesem Quadrat generiert und geprüft, ob der Punkt im Kreis liegt. Punkte innerhalb des Kreises gelten als Treffer, Punkte außerhalb des Kreises werden ignoriert. Abbildung 1 veranschaulicht dieses Konzept. Die Zahl Pi wird im Anschluss aus dem Flächeninhalt des Quadrates, der Gesamtanzahl der Treffer sowie der Gesamtanzahl der Versuche berechnet: $\pi = A * \frac{Treffer_{gesamt}}{Versuche_{gesamt}}$. Da Monte-Carlo-Algorithmen kein korrektes Ergebnis liefern müssen, ist für die Genauigkeit der Zahl Pi die Anzahl der Versuche entscheidend.

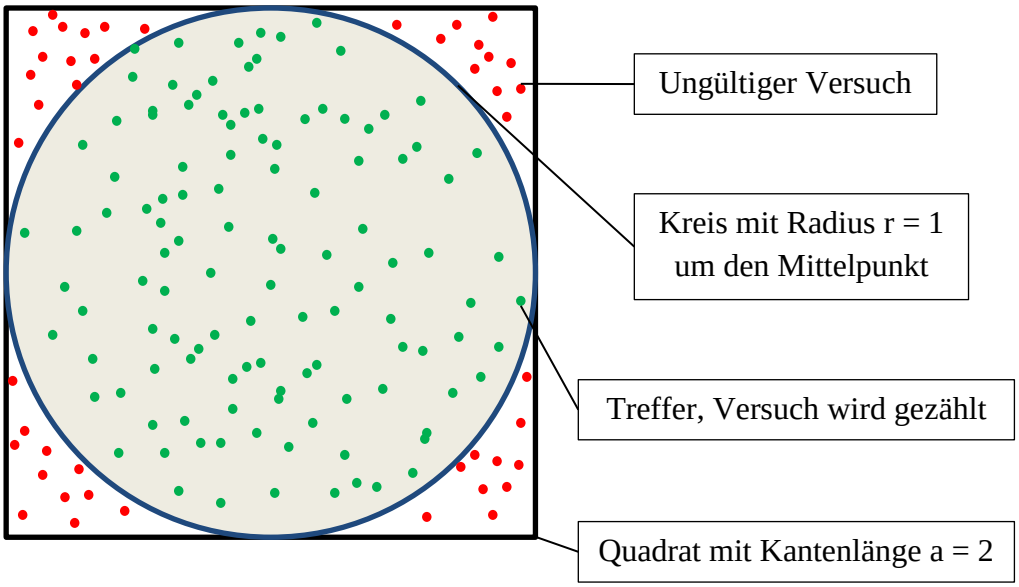


Abbildung 1: Monte Pi

Die Programmausführung erfolgt parallel mit einer vorab angegebenen Anzahl von Prozessen. Jeder Prozess bekommt zunächst eine Anzahl auszuführender Versuche zugeteilt. Anschließend generiert jeder Prozess eine zufällige x- und y-Koordinate pro Versuch innerhalb des Quadrates mit $-1 \leq x \leq 1$ und $-1 \leq y \leq 1$ und prüft, ob es sich um einen Treffer handelt. Die Anzahl der Treffer speichert jeder Prozess zunächst lokal. Nach

Abarbeitung aller Versuche werden die Treffer und Versuche global summiert. Der root Prozess berechnet abschließend die Zahl Pi. Ausgefallene Prozesse beteiligen sich nicht an der Summierung der Ergebnisse, der Verlust ihrer Ergebnisse wird ignoriert. Dies beeinflusst die Berechnung der Zahl Pi entsprechend negativ, da $\frac{Prozzese_{ausgefallen}}{Versuche_{gesamt}}$ Versuche fehlen und sich somit Gesamtzahl der Versuche verringert.

3.1 Implementierung in Resilient X10

Listing 5 zeigt einen Auszug der Implementierung in Resilient X10. In Zeile 2 wird die Anzahl der Versuche für jeden Prozess festgelegt, Zeile 4 enthält das Tupel für das Ergebnis (Anzahl Treffer und Anzahl Versuche). Diese Variable wird als globale Referenz definiert, jeder Prozess erhält Zugriff auf diese Variable. Mit der Iteration über alle Places in Zeile 5 startet die asynchrone Verarbeitung. Im `try`-Block ab Zeile 6 erfolgt die Berechnung auf jedem Place `p` (Zeile 7). Nach Durchführung aller Versuche in den Zeilen 8 bis einschließlich 15 erfolgt eine Ausgabe, im Anschluss werden ab Zeile 17 die Ergebnisse zusammengetragen. Das Schlüsselwort `atomic` gewährleistet dabei einen atomaren Zugriff auf die globale Referenz `result`. Im `catch` Statement in Zeile 22 werden Ausnahmen vom Typ `DeadPlaceException` abgefangen. Diese werden vom `at` Statement in Zeile 7 im Falle eines Ausfalls eines Places erzeugt. Da der Zugriff auf die Ergebnisvariable `result` atomar ist und ausgefallene Prozesse ignoriert werden dürfen, erfolgt keine weitere Fehlerbehandlung. Lediglich eine Information über den Ausfall des Places erfolgt als Ausgabe auf der Konsole. In Zeile 25 erfolgt eine Synchronisation aller Activities durch das Schlüsselwort `finish` in Zeile 5, anschließend berechnet Place 0 die Zahl Pi (Zeile 26) und gibt diese aus (Zeilen 27f).

Listing 5: Implementierung Monte Pi in Resilient X10, Auszug aus Quelle [25]

```
1  public class ResilientMontePi {
2    static val ITERS = 10000000001 / Place.numPlaces();
3    public static def main (args : Rail[String]) {
4      val result = GlobalRef(new Cell(Pair[Long,Long](0l, 0l)));
5      finish for (p in Place.places()) async {
6        try {
7          at (p) {
8            val rand = new Random(System.nanoTime());
9            var total : Long = 0l;
10           for (iter in 1..ITERS) {
11             val x = rand.nextDouble();
12             val y = rand.nextDouble();
13             if (x*x + y*y <= 1.0) total++;
14           }
15           val total_ = total;
16           Console.OUT.println("Work done at: "+here);
17           at (result) atomic {
18             result()(Pair(result()().first+total_,
19               result()().second+ITERS));
20           }
21         }
22       } catch (e:DeadPlaceException) {
23         Console.OUT.println("Got DeadPlaceException from "+e.place);
24       }
25     }
26     val pi = (4.0 * result()().first) / result()().second;
27     Console.OUT.println("pi = "+pi+"   calculated with
28       "+result()().second+" samples.");
29   }
30 }
```

3.2 Implementierung in MPI / ULFM

Listing 6 enthält einen Auszug der Implementierung des Monte Pi Algorithmus in C mit MPI und ULFM. Bis einschließlich Zeile 9 finden einige Initialisierungen statt, etwa die Anzahl aller Versuche in Zeile 2. Die Integer Variable `rc` in Zeile 9 speichert den Rückgabecode aus MPI Funktionen, etwa `MPI_SUCCESS` oder einen der Fehlerklassen aus Punkt 2.4.2. In Zeile 10 wird die MPI Umgebung initialisiert, direkt darauf wird der Errorhandler für den Kommunikator `komm` auf `MPI_ERRORS_RETURN` geändert, um Fehlertoleranz zu ermöglichen. Die Anzahl der Versuche für jeden Prozess wird in Zeile 17 festgelegt. Sollte die Anzahl er Versuche nicht gleichmäßig auf alle Prozesse aufgeteilt werden können, werden die übrig gebliebenen Versuche analog zur Resilient X10 Implementierung ignoriert. In den folgenden sieben Zeilen erfolgt die Durchführung der Versuche. Mittels der MPI Funktion `MPI_Reduce` werden in Zeile 27 die Ergebnisse zusammengetragen. In Zeile 28 wird mit Hilfe der Funktion `MPIX_Comm_agree` der Status aller Prozesse überprüft. Sollte es zu einem Prozessausfall gekommen sein, würde die Funktion `MPIX_Comm_agree` einen

Fehlercode in `rc` zurückgeben. Nach 2.4.3 erhält jeder Prozess aus `komm` den Fehlercode, somit startet jeder Prozess die Fehlerbehandlung in Zeile 29. Zunächst wird in Zeile 30 sichergestellt, dass die `MPI_Reduce` Funktion nach der Fehlerbehandlung über die `do-while`-Schleife erneut ausgeführt wird. Anschließend wird der Kommunikator widerrufen (Zeile 31) und ein neuer Kommunikator mittels `MPIX_Comm_shrink` erzeugt. Dieser enthält alle Prozesse des alten Kommunikators, exklusive den ausgefallenen Prozessen. Da `MPIX_Comm_shrink` eine kollektive Funktion ist, muss jeder noch lebende Prozess diese Funktion aufrufen um im neuen Kommunikator enthalten zu sein. Damit alle noch lebenden Prozesse diese Funktion aufrufen können, müssen alle Prozesse diese Fehlerbehandlung durchlaufen. Um dies wiederum zu gewährleisten, müssen alle Prozesse den Ausfall erkennen.

Obwohl `MPI_Reduce` eine kollektive Funktion ist, gewährt diese Ausnahmsweise keine globale Fehlererkennung, laut Aussage der Entwickler liegt dies an dem internen Aufbau der Funktion. Somit hätte eine Fehlerbehandlung der `MPI_Reduce` Funktion (`rc = MPI_Reduce(&meineTuV[0], &anzahlTuV, 2, MPI_LONG, MPI_SUM , 0, komm);` ohne Zeile 28) zur Folge, dass lediglich der root Prozess den Ausfall erkennt, da ULFM in diesem Fall keine globale Fehlererkennung garantiert. Drauf hin würde lediglich der root Prozess eine Fehlerbehandlung durchführen und die Funktion `MPIX_Comm_shrink` als einziger Prozess aufrufen. In Folge dessen gäbe es nur noch einen Prozess in `komm`, was den Aufruf einer zusätzlichen kollektiven Funktion erforderlich macht. In dieser Arbeit wurde dazu die `MPIX_Comm_agree` Funktion verwendet, prinzipiell genügt aber auch eine einfache `MPI_Barrier`.

In Zeile 35 ermittelt jeder Prozess seinen Rang erneut. Dies erlaubt in diesem Fall auch den Ausfall des root Prozesses, da die übrig gebliebenen Prozesse wieder ab `0` durchnummeriert werden und es somit einen neuen root Prozess gibt. Grundsätzlich lässt sich jedes Programm gegen einen Ausfall des root Prozesses absichern, der Programmierer muss nur sicherstellen, dass der root Prozess keine relevanten Daten exklusiv enthält. Gegebenenfalls müssen solche Daten auf einen persistenten Speicher ausgelagert werden, sodass mit Hilfe einer Wiederherstellungsroutine die Daten durch den neuen root Prozess erneut eingelesen werden können. Nachdem die Prozesse einen neuen Rang erhalten haben, wird im Anschluss der alte Kommunikator mittels seines Destruktors freigegeben (Zeile 33) und erneut zugewiesen (Zeile 34). Sollte ein Fehler während der Fehlerbehandlung auftreten, so wird dies spätestens in der nächsten Iteration erkannt und die Fehlerbehandlung erneut durchgeführt.

Abschließend wird in den Zeilen 38f die Zahl Pi vom root Prozess berechnet und ausgegeben. Die Abfrage `if(meinRang == 0)` macht deutlich, dass eine Neunummerierung der Prozesse erforderlich ist, da nach einem Ausfall des root Prozesses sonst kein Prozess mit `meinRang = 0` existieren würde. Abschließend wird MPI in Zeile 42 beendet.

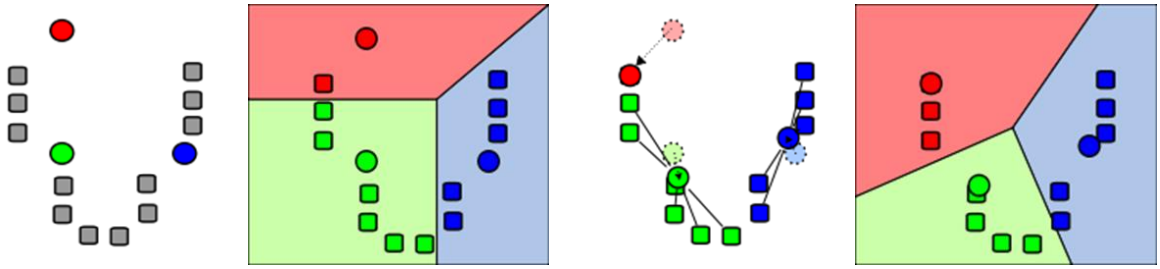
Listing 6: Implementierung Monte Pi in MPI / ULFM

```
1  #include <mpi-ext.h>
2  const long versuche = 1000000000;
3  int main(int argc, char *argv[]) {
4    double x, y, pi;
5    long meineTuV[2];
6    long anzahlTuV[2];
7    MPI_Comm neuerKomm, komm = MPI_COMM_WORLD;
8    int meinRang, anzProzesse;
9    int rc;
10   MPI_Init(&argc, &argv);
11   MPI_Comm_set_errhandler(komm, MPI_ERRORS_RETURN);
12   MPI_Comm_rank(komm, &meinRang);
13   MPI_Comm_size(komm, &anzProzesse);
14   srand(time(NULL) * meinRang + 1);
15   meineTuV[0] = 0;
16   long i;
17   meineTuV[1] = versuche / anzProzesse;
18   for(i = 0; i < meineTuV[1]; i++) {
19     x = (double)rand() / RAND_MAX;
20     y = (double)rand() / RAND_MAX;
21     if(x*x + y*y <= 1.0) {
22       meineTuV[0]++;
23     }
24   }
25   int alleErfolgreich;
26   do {
27     MPI_Reduce(&meineTuV[0], &anzahlTuV, 2, MPI_LONG, MPI_SUM, 0, komm);
28     rc = MPIX_Comm_agree(komm, &alleErfolgreich);
29     if(rc != MPI_SUCCESS){
30       alleErfolgreich = 0;
31       MPIX_Comm_revoke(komm);
32       MPIX_Comm_shrink(komm, &neuerKomm);
33       MPI_Comm_free(&komm);
34       komm = neuerKomm;
35       MPI_Comm_rank(komm, &meinRang);
36     } else alleErfolgreich = 1;
37   } while(!alleErfolgreich);
38   if(meinRang == 0) {
39     pi = 4.0 * (double)anzahlTuV[0] / (double)anzahlTuV[1];
40     printf("Die berechnete Zahl pi: %f\n", pi);
41   }
42   MPI_Finalize();
43   return 0;
44 }
```

4 Beispielprogramm K-Means

Der K-Means-Algorithmus wird zur Clusteranalyse verwendet. Das Verfahren wird häufig zur Gruppierung von Objekten eingesetzt, da es schnell die Zentren der Cluster findet [26]. In diesem Beispielprogramm werden Objekte durch Punkte in einem beliebig dimensionalen Raum dargestellt. Die Anzahl der Punkte kann frei gewählt werden, ebenso die Anzahl der Cluster, denen die Punkte zugeordnet werden. Abbildung 2 zeigt ein Beispiel aus [26].

In dem vorliegendem Algorithmus werden zunächst die Punkte zufällig in dem beliebig dimensionierten Raum bestimmt. Im Anschluss werden die Startzentren der Cluster ebenfalls zufällig gewählt. Die folgenden zwei Schritte werden bis zu einem Abbruchkriterium[4] wiederholt: Berechnung des nächstliegendem Clusterzentrum eines jeden Punktes mit Hilfe des Euklidischem Abstands und anschließende Neuberechnung der Clusterzentren.



Drei Clusterzentren wurden zufällig gewählt.	Die durch Rechtecke repräsentierten Objekte werden jeweils dem Cluster mit dem nächsten Clusterzentrum zugeordnet.	Die Zentren (jeweilige Schwerpunkte) der Cluster werden neu berechnet.	Die Objekte werden neu verteilt und erneut dem Cluster zugewiesen, dessen Zentrum am nächsten ist.

Abbildung 2: Beispiel K-Means, Darstellung angelehnt an [26]

[4] Anzahl der maximalen Iterationen erreicht oder Änderung der Clusterzentren kleiner als 0.0001

4.1 Implementierung in Resilient X10

Listing 7 zeigt eine Implementierung des K-Means Algorithmus in Resilient X10 aus [4] Seite 3. Zunächst werden die Punkte und Clusterzentren generiert und auf alle Places übertragen (Zeile 5 und 7, nur mit Text dargestellt). In den Zeilen 9 bis 17 erfolgt die Zuweisung der Arbeit für die Places. Diese Zuweisung erfolgt in jeder Schleifeniteration neu, Zeile 9 ermittelt die Anzahl der zur Verfügung stehenden Places. In Zeile 10f wird die Anzahl der zu bearbeiteten Punkte für jeden Place berechnet, anschließend erfolgt die parallele Verarbeitung ab Zeile 14. Ausgefallene Places werden in Zeile 15 einfach übersprungen, in Zeile 17ff werden die euklidischen Abstände für jede Punkt – Cluster Kombination berechnet. Exceptions vom Typ `DeadPlaceException`, welche im Fall eines Ausfalls in Zeile 14 des `finish` Statements geworfen werden, werden in Zeile 23 abgefangen und ignoriert. Andere Exceptions werden weiter geworfen (Zeile 25). Abschließend werden die neuen Clusterzentren berechnet (Zeile 27) und ausgegeben (Zeile 29).

Listing 7: Implementierung K-Means in Resilient X10 (Auszug), Quelle [4] Seite 3

```
1  class ResilientKMeans {
2    static val POINTS = 10000000; // number of points
3              :
4    public static def main(args:Rail[String]) {
5      /*prepare a set of points, and deliver it to other places*/
6      for (iter in 1..ITERATIONS) { // iterate until convergence
7        /*deliver current cluster values to other places*/
8        //process some part of the points at each place
9        val numAvail = Place.MAX_PLACES - Place.numDead();
10       val div = POINTS / numAvail; // share for each place
11       val rem = POINTS % numAvail; // extra share for Place 0
12       var start:Long = 0; // next point to be processed
13       try {
14         finish for (pl in Place.places()) {
15         if (pl.isDead()) continue; // skip dead place(s)
16         var end:Long = start+div; if (pl==place0) end+=rem;
17         at (pl) async { // compute at live places in parallel
18           /* process points [start,end), and return the data
19              necessary for updating cluster vals to Place 0 */
20         }
21         start = end;
22       } //end of finish, wait for the execution in all places
23     } catch (es:MultipleExceptions) {
24       for (e in es.exceptions()) { // just ignore place death
25         if (!(e instanceof DeadPlaceException)) throw e; }
26       }
27       /* compute new cluster values, and exit if converged */
28     } // end of for (iter)
29     /* print the result */
30   } // end of main
31 }
```

4.2 Implementierung in MPI / ULFM

Listing 8 zeigt eine Implementierung von K-Means in MPI / ULFM. Zunächst werden in den Zeilen 1 – 4 und 6 – 10 einige Variable definiert, etwa das zweidimensionale Array `punkte`, welches alle Punkte enthält. `meinePunkte` hingegen speichert nur die Punkte des jeweiligen Prozesses. Zeile 8 enthält den Kommunikator sowie einen Hilfskommunikator, welcher im Fall eines Prozessausfalls benötigt wird, Zeile 9 speichert die Rückgabe- / Fehlercodes der MPI Funktionen. In Zeile 16 beginnt eine für Fehlertoleranz wichtige Schleife und endet in Zeile 58. Diese `do-while` Schleife wird verlassen, wenn ein vollständiger Schleifendurchlauf ohne Prozessausfall gelang. Zu Beginn der Schleife wird in den Zeilen 17 bis 20 die Job Größe für jeden Prozess berechnet. Anschließend verteilt der root Prozess das `punkte` Array auf alle anderen Prozesse mittels `MPI_Send`, diese empfangen die Daten im `meinePunkte` Array (Zeilen 26 - 35). Ab Zeile 36 beginnt die Schleife über Berechnungsiterationen. Zunächst werden die aktuellen Clusterzentren via `MPI_Bcast` an alle Prozesse verteilt (Zeile 37f). Nachdem jeder Prozess die ihm zugewiesenen Punkte abgearbeitet hat (Zeile 39), werden die lokalen Ergebnisse der Clusterinformationen (`summeACLokal`) mittels `MPI_Reduce` in den Zeile 40 - 43 zusammengefasst. Mittels `MPIX_Comm_agree` wird überprüft, ob alle Prozesse noch funktionsfähig sind und somit die `MPI_Reduce` Funktion erfolgreich war. Im Fall eine Prozessausfalls wird die Variable `ausfall` auf `1` gesetzt. Dies bewirkt ein erneutes Durchlaufen der `do-while`-Schleife in Zeile 16 und somit ein erneutes verteilen der Arbeit auf alle Prozesse. Zusätzlich wird in den Zeilen 21 - 24 der Speicher der verwendeten lokalen Punkte Arrays freigegeben und `ausfall` wieder auf `0` gesetzt.

Anschließend wird analog zur Monte Pi Implementierung der Kommunikator `komm` widerrufen (Zeile 47) und ein neuer Kommunikator ohne die ausgefallenen Prozesse erzeugt (Zeile 48). In Zeile 49f wird der alte Kommunikator freigegeben und erneut zugewiesen. Anschließend werden der Rang und die Anzahl der Prozesse im neuen Kommunikator für jeden Prozess neu bestimmt, um eine korrekte Neuverteilung der Arbeit zu gewährleisten. In Zeile 53 wird mittels einer break Anweisung die Berechnungsschleife zunächst verlassen und über die darüber liegende `do-while` Schleife mit der Neuverteilung der Punkte auf die übrigen Prozesse begonnen. Im Fall eines störungsfreien Durchlaufens der Berechnungsschleife und Erreichen der maximalen Iterationen bzw. Konvergenz, wird diese sowie die darüber liegende, arbeitsverteilende Schleife verlassen und MPI beendet (Zeile 58).

Listing 8: Implementierung K-Means (Auszug) in MPI / ULFM

```
const long anzDimensionen = 4;
const long anzPunkte = 1000000;
const long anzIterationen = 1000;
const float konvergenz = 0.0001;
int main(int argc, char *argv[]) {
  float **punkte;
  float **meinePunkte;
  MPI_Comm neuerKomm, komm = MPI_COMM_WORLD;
  int rc;
  long jobGroesse, meineJobGroesse, iteration;
  MPI_Init(&argc, &argv);
  MPI_Comm_set_errhandler(komm, MPI_ERRORS_RETURN);
  MPI_Comm_rank(komm, &meinRang);
  MPI_Comm_size(komm, &anzProzesse);
        /* Punkte und Cluster initialisieren */
  do {
    jobGroesse = anzPunkte / anzProzesse;
    meineJobGroesse = jobGroesse;
    if(meinRang == anzProzesse-1 && anzPunkte % anzProzesse != 0)
      meineJobGroesse += anzPunkte % anzProzesse;
    if(ausfall) {
      free(meinePunkteH);
      free(meinePunkte);
      ausfall = 0;
    }
    if(meinRang == 0) {
      int tempJobGroesse;
      for(i = 1; i < anzProzesse; i++) {
        tempJobGroesse = (i == anzProzesse-1 ? meineJobGroesse + (anzPunkte
          % anzProzesse) : meineJobGroesse);
        MPI_Send(&(punkte[tempJobGroesse*i][0]), tempJobGroesse *
          anzDimensionen, MPI_FLOAT, i, 1, komm);
      }
    } else MPI_Recv(&(meinePunkte[0][0]), meineJobGroesse * anzDimensionen,
             MPI_FLOAT, 0, 1, komm, &status);
    do {
      MPI_Bcast(&(cluster[0][0]), ANZCLUSTER * anzDimensionen, MPI_FLOAT,
        0, komm);
                      :
      for(i = 0; i < ANZCLUSTER; i++) {
        MPI_Reduce(&summeACLokal[i], &neueCluster[i], anzDimensionen+1,
          MPI_FLOAT, MPI_SUM, 0, komm);
      }
      rc = MPIX_Comm_agree(komm, &ausfall);
      if(rc != MPI_SUCCESS){
        ausfall = 1;
        MPIX_Comm_revoke(komm);
        MPIX_Comm_shrink(komm, &neuerKomm);
        MPI_Comm_free(&komm);
        komm = neuerKomm;
        MPI_Comm_rank(komm, &meinRang);
        MPI_Comm_size(komm, &anzProzesse);
        break;
      }
      iteration++;
    } while(iteration <= anzIterationen && !konvergiert);
  } while(ausfall);
  MPI_Finalize();
  return 0;
}
```

5 Beispielprogramm Heat Transfer

Der Heat Transfer Algorithmus dient zur Berechnung der Diffusion von Temperaturen in einem Raum. Bei dem in dieser Arbeit verwendeten Algorithmus handelt es sich analog zur Implementierung aus [4] um einen zweidimensionalen Raum. Die Darstellung des Raumes erfolgt durch ein zweidimensionales Array. Jedes Element des Arrays enthält einen Temperaturwert und repräsentiert somit einen Punkt im Raum. Bei jeder Iteration wird jeder Temperaturwert durch das Bilden des arithmetischen Mittels der umliegenden Werte neu berechnet. Abbildung 3 illustriert diesen Gedanken. Nach Erreichen der maximalen Iterationen oder Konvergenz wird die Berechnung beendet.

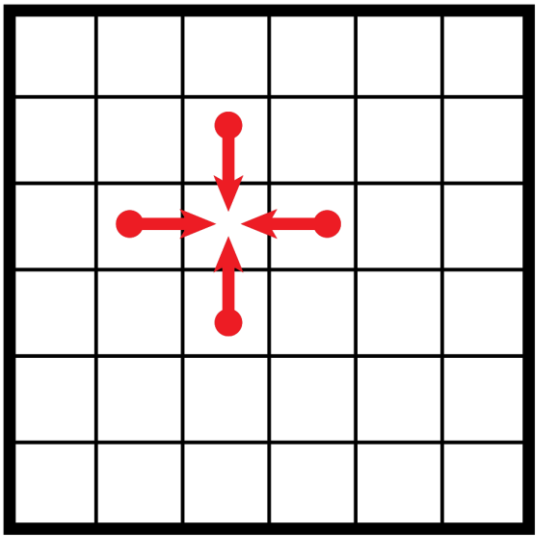


Abbildung 3: Berechnung einer Heat Transfer Zelle, entnommen aus [27]

5.1 Implementierung in Resilient X10

Listing 9 zeigt einen Auszug der Implementierung des Heat Transfer Algorithmus aus [4] Seite 4. Nach einigen Initialisierungen in den Zeilen 2 – 4 werden in Zeile 7 alle vorhandenen und lebenden Places ermittelt. In Zeile 9 wird ein zweidimensionales Array initialisiert und auf die ermittelten Places verteilt (Zeile 10f). Anschließend wird in Zeile 13 das Heat Transfer Array `A` als `ResilientDistArray` aus dem `DistArray` aus Zeile 10f erstellt. Die Datenstruktur `ResilientDistArray` ermöglicht eine Fehlerbehandlung, etwa die

Erstellung eines Snapshots mit `A.snapshot();` in Zeile 14. Ab Zeile 15 bis einschließlich 29 wird mittels einer `for`-Schleife über alle Iterationen bzw. bis zur Konvergenz iteriert. Die Berechnung der neuen Temperaturwerte erfolgt in den Zeilen 23 – 25, jeder Place berechnet die bei ihm gespeicherten Werte. Die Randwerte müssen bei jeder Iteration von dem jeweiligen Place beschafft werden, der Zugriff auf den entfernten Speicher erfolgt mittels `at (A.dist(x-1,y)) A(x-1,y)` (nicht im Listing enthalten). Jede zehnte Iteration wird ein Snapshot erstellt (Zeile 27), die Snapshot Frequenz beeinflusst die Laufzeitgeschwindigkeit sowie die Fehlertoleranz. Ein häufiges Erstellen von Snapshots reduziert den Verlust von Informationen bei einem Ausfall, erhöht jedoch die Laufzeit des Programms. Idealerweise wird die Häufigkeit der Snapshot-Erstellung in Abhängigkeit der MTBF der verwendeten Hardware gewählt. Im Rahmen dieser Arbeit wird davon jedoch zugunsten von Demonstrationszwecken abgesehen.

Sollte im `try`-Block ab Zeile 16 ein Fehler auftreten, wird dieser im `catch`-Block in Zeile 28 behandelt. Der Fehler wird der Funktion `processException(e:Exception)` (Zeile 33 – 39) übergeben. Diese behandelt Fehler von Typ `DeadPlaceException` indem sie den ausgefallenen Place von der Liste der lebenden Places streicht und die Variable `restore_needed()` auf `true` setzt. In der nächsten Iteration wird in Zeile 17 das Setzen von `restore_needed()` auf `true` bemerkt und anschließend mit der Wiederherstellung von `A` begonnen. Zunächst wird in Zeile 18f das verteile Array `BigD` auf alle noch lebenden Places neu verteilt. In Zeile 21 wird `A` mittels `A.restore(BigD);` wiederhergestellt. Die Wiederherstellung der Daten erfolgt aus dem letzten erfolgreichen Snapshot, der maximal mögliche Datenverlust beläuft sich somit auf zehn Iterationen. Nach Durchlaufen aller Iterationen bzw. Erreichen der Konvergenz wird in Zeile 30 das Ergebnis ausgegeben.

Listing 9: Implementierung Heat Transfer in Resilient X10 (Auszug), Quelle [4] Seite 4

```
1  class ResilientHeatTransfer {
2    static val N = 20; // size of grid
3    static val livePlaces = new ArrayList[Place]();
4    static val restore_needed = new Cell[Boolean](false);
5           :
6    public static def main(args:Rail[String]) {
7      for (pl in Place.places()) livePlaces.add(pl);
8      // initialize Region and Dist
9      val BigR = Region.make(0..(N+1), 0..(N+1)); // +surroundings
10     var BigD:Dist(2) = Dist.makeBlock(BigR, 0,
11             new SparsePlaceGroup(livePlaces.toRail()));
12     // create a DistArray, each element holds a heat value
13     val A = ResilientDistArray.make[Double](BigD, ...);
14     A.snapshot(); // create the initial snapshot
15     for (iter in 1..ITERATIONS) { // iterate until convergence
16       try {
17         if (restore_needed()) { // if some places died
18           BigD = Dist.makeBlock(BigR, 0, // recreate Dist, and
19               new SparsePlaceGroup(livePlaces.toRail()));
20           A.restore(BigD); // restore elements from the snapshot
21           restore_needed() = false;
22         }
23         finish ateach (z in D_Base) { // distributed processing
24           /* compute new heat values for A's local elements */
25         }
26         /* if converged, exit the for loop */
27         if (iter % 10 == 0) A.snapshot(); // create a snapshot
28       } catch (e:Exception) { processException(e); }
29     } // end of for (iter)
30     /* print the result */
31   } // end of main
32
33   private static def processException(e:Exception) { // exception
34     if (e instanceof DeadPlaceException) {
35       val deadPlace = (e as DeadPlaceException).place;
36       livePlaces.remove(deadPlace); restore_needed() = true;
37     } else ... /* handle MultiPlaceExceptions recursively */
38   }
39 }
```

5.2 Implementierung in MPI / ULFM

Listing 10 zeigt die Implementierung des Heat Transfer Algorithmus in C mit MPI und ULFM. Um das Listing übersichtlich zu halten, wurde es an verschiedenen Stellen gekürzt und auf das wesentliche reduziert. Anfangs finden einige Variablen-Definitionen bis einschließlich Zeile 13 statt. Darunter befinden sich z.B. die verschiedenen Arrays, etwa das große Heat Transfer Array `ht` mit allen Werten (Zeile 5). `htTemp` dient als Hilfs-Array für die Erstellung eines Snapshots, `meinHt` dient als lokales Arbeitsarray. Die Variablen `rc` und `ausfall` sowie `neuerKomm` dienen analog zur K-Means Implementierung der Fehlerbehandlung. Nach der Initialisierung der MPI Umgebung (Zeile 14) wird der Errorhandler auf `MPI_ERRORS_RETURN` gesetzt um eine Fehlerbehandlung zu ermöglichen.

Nach dem Ermitteln des Rangs und der Größe des Kommunikators durch jeden Prozess (Zeile 16f) wird das `ht`-Array durch root mit Zufallsvariablen in Zeile 18 initialisiert. Die Zeilen 19 – 118 enthalten die Schleife für die Verteilung der Arbeit auf die noch lebenden Prozesse und werden im Folgenden näher beschrieben.

Nach der Ermittlung der individuellen Aufgabengröße (Zeilen 20 – 26) durch jeden Prozess verteilt der root Prozess das `ht`-Array auf alle Prozesse in `komm`. Die Verteilung erfolgt dabei zeilenweise mittels der `MPI_Send` Funktion (Zeile 35f). In Zeile 38f kopiert root seinen eigenen Aufgabenbereich in sein lokales Arbeitsarray `meinHt`, alle anderen Prozesse empfangen ihren Aufgabenbereich in Zeile 41f. Im Anschluss beginnt jeder Prozess mit Berechnung des Heat Transfers von `meinHt` in `meinHtTemp` (Zeilen 45 – 50) und überträgt die Ergebnisse in `meinHt` (Zeile 51). Ab Zeile 52 beginnt die Übertragung der Randdaten an die jeweiligen Prozesse, welche diese Daten benötigen. Um keine Deadlocks zu verursachen wird jeweils nicht-blockierend gesendet (mittels `MPI_Isend)` aber blockierend empfangen (`MPI_Recv`).

Im Fall eines Ausfalls wird dieser in Zeile 72 erkannt. Ab Zeile 73 beginnt im Bedarfsfall die Fehlerbehandlung. Diese erfolgt weitgehend analog zur K-Means Implementierung, zunächst wird die Variable `ausfall` auf `1` gesetzt, um die `do-while`-Schleife für die Neuverteilung der Arbeit erneut zu durchlaufen (Zeile 74). Anschließend wird der Kommunikator `komm` repariert (Zeilen 75 – 78) und der Rang sowie die Anzahl der übrig gebliebenen Prozesse durch jeden Prozess neu ermittelt (Zeile 79f). Mittels der `break`-Anweisung wird die aktuelle Iteration der Berechnung des Heat Transfers abgebrochen (Zeile 81). Während der Neuverteilung der Arbeit wird ab Zeile 27 der Speicherbereich der lokalen Arbeitsarrays freigegeben und die Variable `ausfall` wieder auf `0` gesetzt.

Mittels der Funktion `MPI_Allreduce` (Zeile 83f) sowie den Zeilen 85 – 87 findet eine Konvergenzprüfung statt. Eine komplexe Fehlerbehandlung ist hier nicht erforderlich, es wird lediglich geprüft, ob ein Ausfall vorliegt (Zeile 85). Damit wird verhindert, dass einzelne Prozesse im Fall des Erreichens der Konvergenz die Iterations-Schleife verlassen. Ab Zeile 88 wird jede zehnte Iteration ein Snapshot erstellt. Dabei empfängt der root Prozess mittels `MPI_Recv` die Daten der anderen Prozesse (Zeile 95). Sollte es hierbei zu einem Ausfall kommen, wird die Snapshot Erstellung sofort abgebrochen (Zeile 99). Da in diesem Fall ein Ausfall nur von root erkannt werden kann, wird mittels `MPIX_Comm_agree` ein konsistenter Zustand für alle Prozesse (s.o.) hergestellt.

Listing 10: Implementierung Heat Transfer (Auszug) in MPI / ULFM

```
1  const long n = 200;
2  const long anzIterationen = 1000;
3  const double konvergenz = 0.0001;
4  int main(int argc, char *argv[]) {
5    double **ht; // Heat Transfer Array
6    double **htTemp; // Temporäres Array für Snapshot
7    double **meinHt; // kleines Heat Transfer Array für jeden Prozess
8    double **meinHtTemp; // Hilfsarray für Berechnungen
9    MPI_Comm neuerKomm, komm = MPI_COMM_WORLD;
10   int meinRang, anzProzesse, konvergiert;
11   int rc, ausfall = 0; // Fehlervariablen
12   long jobGroesse, meineJobGroesse;
13   long iteration = 0; // Anzahl der abgearbeiteten Iterationen
14   MPI_Init(&argc, &argv);
15   MPI_Comm_set_errhandler(komm, MPI_ERRORS_RETURN);
16   MPI_Comm_rank(komm, &meinRang);
17   MPI_Comm_size(komm, &anzProzesse);
18   /*Initialisierung von ht mit Zufallswerten*/
19   do { // Schleife für Aufgabenverteilung
20     jobGroesse = n / anzProzesse;
21     meineJobGroesse = jobGroesse;
22     if(meinRang == anzProzesse-1 && n % anzProzesse != 0)
23       meineJobGroesse += n % anzProzesse;
24     if(meinRang == 0 || meinRang == anzProzesse-1)
25       meineJobGroesse += 1;
26     else meineJobGroesse += 2;
27     if(ausfall) {
28       /*Speicher lokal genutzer Arrays freigeben*/
29       ausfall = 0;
30     }
31     if(meinRang == 0) { // Daten verteilen
32       for(i = 1; i < anzProzesse; i++) {
33         tempJobGroesse = (i == anzProzesse-1 ? jobGroesse+1 + (n %
34           anzProzesse) : jobGroesse+2);
35         MPI_Send(&(ht[jobGroesse*i-1][0]), tempJobGroesse * n, MPI_DOUBLE,
36           i, 0, komm);
37       }
38       memcpy(&(meinHt[0][0]), &(ht[0][0]), n * meineJobGroesse *
39         sizeof(double));
40     } else { // Daten empfangen
41       MPI_Recv(&(meinHt[0][0]), meineJobGroesse * n, MPI_DOUBLE, 0, 0,
42         komm, &status);
43     }
44     do { // große Schleife fuer die Berechnung des Heat Tranfers
45       for(i = 1; i < meineJobGroesse-1; i++) { // Berechnung des HTs
46         for(j = 1; j < n-1; j++) {
47           meinHtTemp[i][j] = (meinHt[i-1][j] + meinHt[i+1][j] +
48                               meinHt[i][j-1] + meinHt[i][j+1]) / 4;
49         }
50       }
51       /*Temporäre Daten aus meinHtTemp in meinHt speichern*/
52       if(meinRang == 0) { // Randdaten neu verschicken
53         MPI_Isend(&(meinHt [meineJobGroesse-2][0]), n, MPI_DOUBLE,
54           meinRang+1, meinRang, komm, &request);
55         MPI_Recv(&(meinHt [meineJobGroesse-1][0]), n, MPI_DOUBLE,
56           meinRang+1, meinRang+1, komm, &status);
57       } else if(meinRang == anzProzesse-1) {
58         MPI_Isend(&(meinHt [1][0]), n, MPI_DOUBLE, meinRang-1, meinRang,
59           komm, &request);
60         MPI_Recv(&(meinHt [0][0]), n, MPI_DOUBLE, meinRang-1,
61           meinRang-1, komm, &status);
```

```
62        } else {
63          MPI_Isend(&(meinHt [1][0]), n, MPI_DOUBLE, meinRang-1, meinRang,
64            komm, &request);
65          MPI_Recv(&(meinHt [0][0]), n, MPI_DOUBLE, meinRang-1,
66            meinRang-1, komm, &status);
67          MPI_Isend(&(meinHt [meineJobGroesse-2][0]), n, MPI_DOUBLE,
68            meinRang+1, meinRang, komm, &request);
69          MPI_Recv(&(meinHt [meineJobGroesse-1][0]), n, MPI_DOUBLE,
70            meinRang+1, meinRang+1, komm, &status);
71        }
72        rc = MPIX_Comm_agree(komm, &ausfall); // Ausfall erkennen
73        if(rc != MPI_SUCCESS){
74          ausfall = 1;
75          MPIX_Comm_revoke(komm);
76          MPIX_Comm_shrink(komm, &neuerKomm);
77          MPI_Comm_free(&komm);
78          komm = neuerKomm;
79          MPI_Comm_rank(komm, &meinRang);
80          MPI_Comm_size(komm, &anzProzesse);
81          break;
82        }
83        rc = MPI_Allreduce(&maxAbweichungLokal, &maxAbweichung, 1,
84          MPI_DOUBLE, MPI_MAX, komm);
85        if(maxAbweichung < konvergenz && rc == MPI_SUCCESS) {
86          konvergiert = 1;
87        }
88        if(iteration % 10 == 0 || konvergenz) {
89          if(meinRang == 0) {
90            memcpy(&(htTemp[0][0]), &(meinHt[0][0]),
91              (meineJobGroesse-1) * n * sizeof(double));
92            for(i = 1; i < anzProzesse; i++) {
93              tempJobGroesse = (i == anzProzesse-1 ? jobGroesse +
94                                (n % anzProzesse) : jobGroesse);
95              rc = MPI_Recv(&(htTemp[jobGroesse*i][0]), tempJobGroesse * n,
96                MPI_DOUBLE, i, i, komm, &status);
97              if(rc != MPI_SUCCESS) {
98                ausfall = 1;
99                break;
100             }
101           }
102           if(!ausfall) { // Keine Snapshot Erstellung bei Ausfall
103             memcpy(&(ht[0][0]), &(htTemp[0][0]), n * n * sizeof(double));
104           }
105         } else {
106           tempJobGroesse = (i == anzProzesse-1 ? jobGroesse +
107             (n % anzProzesse) : jobGroesse);
108           MPI_Send(&(meinHt[1][0]), tempJobGroesse * n, MPI_DOUBLE, 0,
109             meinRang, komm);
110         }
111       }
112       rc = MPIX_Comm_agree(komm, &ausfall);
113       if(rc != MPI_SUCCESS){
114         /*Fehlerbehandlung analog zu oben*/
115       }
116       iteration++;
117     } while(iteration <= anzIterationen && !konvergiert);
118   } while(ausfall);
119   /*Ausgabe des Ergebnisses*/
120   MPI_Finalize();
121   return 0;
122 }
```

6 Auswertung

Dieses Kapitel enthält eine kurze Erläuterung zur Fehlertoleranz und Programmieraufwand für jedes Beispielprogramm sowie eine kurze Präsentation der Ergebnisse der Laufzeitmessungen. Die Laufzeitmessungen wurden auf zwei Clustern des ITS durchgeführt:

ITS Cluster [28]:
its-cs200.its.uni-kassel.de
its-cs201.its.uni-kassel.de
Scientific Linux 6
Prozessoren: AMD Opteron, 12 Kerne mit je 2,3 GHz
Hauptspeicher: 128GB
OpenMPI Version 1.8.1
ULFM Version 1.0
X10 Version 2.5.3

Jede Messung wurde zehnmal durchgeführt und jeweils die beste gewertet. Bei Messungen mit Ausfällen wurde jeder zweite Prozess / Thread zum Absturz gebracht bzw. beendet. Alle Messungen wurden mit aktivierter Fehlertoleranz durchgeführt.

Bei MPI / ULFM wurden Prozesse mittels der Funktion `raise(SIGKILL);` gezielt zum Absturz gebracht, jeweils in der 100. Iteration des Programms. Bei Resilient X10 wurden Threads mit dem Befehl `kill` manuell beendet. Dies war leider nicht immer bei exakt der Hälfte der Threads möglich, sodass sich vor allen bei vielen Threads gegeben falls leichte Vorteile für die Resilient X10 Implementierungen ergaben

6.1 Monte Pi

6.1.1 Fehlertoleranz

Die Fehlertoleranz ist bei den Resilient X10 und MPI / ULFM Implementierungen ähnlich gut, Ausfälle werden jeweils einfach ignoriert. Als Unterschied ist zu erwähnen, dass bei Resilient X10 der Place 0 nicht ausfallen darf ([4] Seite 2).

6.1.2 Programmieraufwand

Der Programmieraufwand ist bei MPI etwas höher als bei X10 einzustufen. Während bei Resilient X10 lediglich eine Ausnahme abgefangen werden muss, muss bei MPI der Fehler behandelt werden, da sonst die Funktion `MPI_Reduce` kein Ergebnis liefern würde. Bei der Behandlung von Fehlern zeigte sich der Nachteil, dass es bei ULFM keine globale Fehlererkennung gibt und der Programmierer für eine globale Fehlererkennung (falls benötigt) selbst sorgen muss.

6.1.3 Laufzeit

Abbildung 4 zeigt die einen Leistungsvergleich zwischen Resilient X10 und MPI / ULFM. Dieser fällt klar zugunsten von Resilient X10 aus. Eine Erklärung dafür könnte sein, dass die ULFM Entwicklung bisher ausdrücklich auf Korrektheit fokussiert ist und nicht auf Leistung. Positiv für die MPI / ULFM Implementierung ist der geringe Zeitaufwand für die Fehlerbehandlung, Resilient X10 benötigt gerade bei vielen Threads deutlich mehr Zeit. Der Geschwindigkeitsvorsprung der MPI / ULFM Implementierung bei Ausfällen mit mehr als 12 Prozessen kann damit begründet werden, dass die verbliebenen Prozesse das Cluster (mit insgesamt 12 Kernen) effizienter auslasten können.

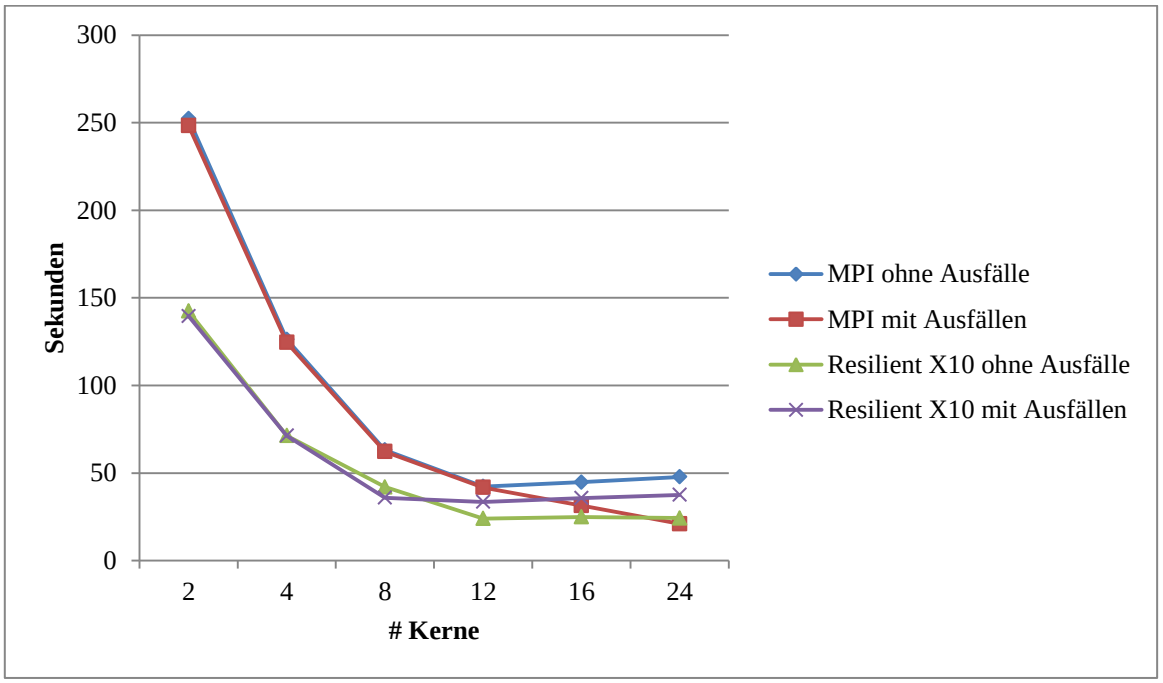


Abbildung 4: Laufzeit Monte Pi auf dem ITS Cluster mit 10^{10} Versuchen

6.2 K-Means

6.2.1 Fehlertoleranz

Analog zu Monte Pi ist auch bei K-Means ein ähnliches Verhalten bei Ausfällen zwischen Resilient X10 und MPI / ULFM festzustellen. Während bei Monte Pi auch der root Prozess bei der MPI / ULFM Implementierung ausfallen darf, ist dies bei K-Means nicht so. Der Grund hierfür liegt darin, dass der root Prozess alle Punkte in einem großen Array speichert und bei Bedarf neu verteilt, es gibt kein Backup. Abhilfe könnte man durch das Auslagern des Arrays auf einen Festspeicher schaffen, was jedoch zu Lasten der Laufzeit gehen würde. Alternativ könnte man analog zur Resilient X10 Implementierung das Punkte-Array jedem Prozess vollständig zur Verfügung stellen, jedoch mit dem Nachteil eines höheren Ressourcenverbrauchs. Hierin liegt ein Vorteil der MPI / ULFM Implementierung, da jeder Prozess nur die tatsächlich benötigten Punkte erhält und damit weniger Kommunikationszeit und Arbeitsspeicher benötigt. Die implementierte Variante wurde gewählt, da sie mit MPI im Gegensatz zu X10 einfach umzusetzen ist. Eine dritte Möglichkeit besteht darin, das Punkte-Array nicht ausschließlich von root speichern zu lassen, sondern auf alle Prozesse zu verteilen[5]. Der Implementierungsaufwand hierfür ist jedoch enorm, da ULFM für ein solches Szenario keine Schnittstellen oder Funktionen bietet. Dies ist jedoch auch nicht gewollt, denn ULFM ist eine Erweiterung zur Fehlerbehandlung und keine Sicherungs- / Wiederherstellungsstrategie.

6.2.2 Programmieraufwand

Auch bei K-Means ist der Programmieraufwand bei MPI / ULFM etwas höher als bei Resilient X10. Während bei der Resilient X10 Implementierung bei jeder Iteration eine Neuverteilung der zu bearbeiteten Punkte in Abhängigkeit von den verfügbaren Places erfolgt, ist dies bei der MPI / ULFM Implementierung nur nach einem tatsächlichen Ausfall vorgesehen. Dies erlaubt bei Resilient X10 ein Ignorieren der Ausnahmen, es muss lediglich eine Ausführung von Activities auf defekten Places (mittels `if (pl.isDead()) continue;`) verhindert werden. Bei der MPI / ULFM Implementierung findet nach einem Prozessausfall eine Neuberechnung der Arbeitsgröße und ein neu Versenden der entsprechenden Punkte statt. Bei sehr vielen Prozessausfällen führt

[5] D.h. jeder Prozess speichert zusätzlich zu seinen zu bearbeiteten Punkten weitere Punkte um Ausfälle andere Prozesse abfangen zu können.

dies zu einem höheren Kommunikationsaufwand. Außerdem muss analog zur Monte Pi Implementierung der Fehler behandelt werden, andernfalls ist keine kollektive Kommunikation möglich.

6.2.3 Laufzeit

Bei dem K-Means Algorithmus benötigt die MPI / ULFM Implementierung deutlich mehr Zeit für die Berechnung bei Ausfällen als dies noch bei Monte Pi der Fall war, zu sehen in Abbildung 5. Dies liegt daran, dass die gleiche Menge an Arbeit nun von weniger Prozessen abgearbeitet werden muss. Auch hier fällt der Leistungsvergleich zugunsten der Resilient X10 Implementierung aus, bei vielen Prozessen bzw. Threads zeigten sich jedoch Vorteile bei der MPI / ULFM Implementierung.

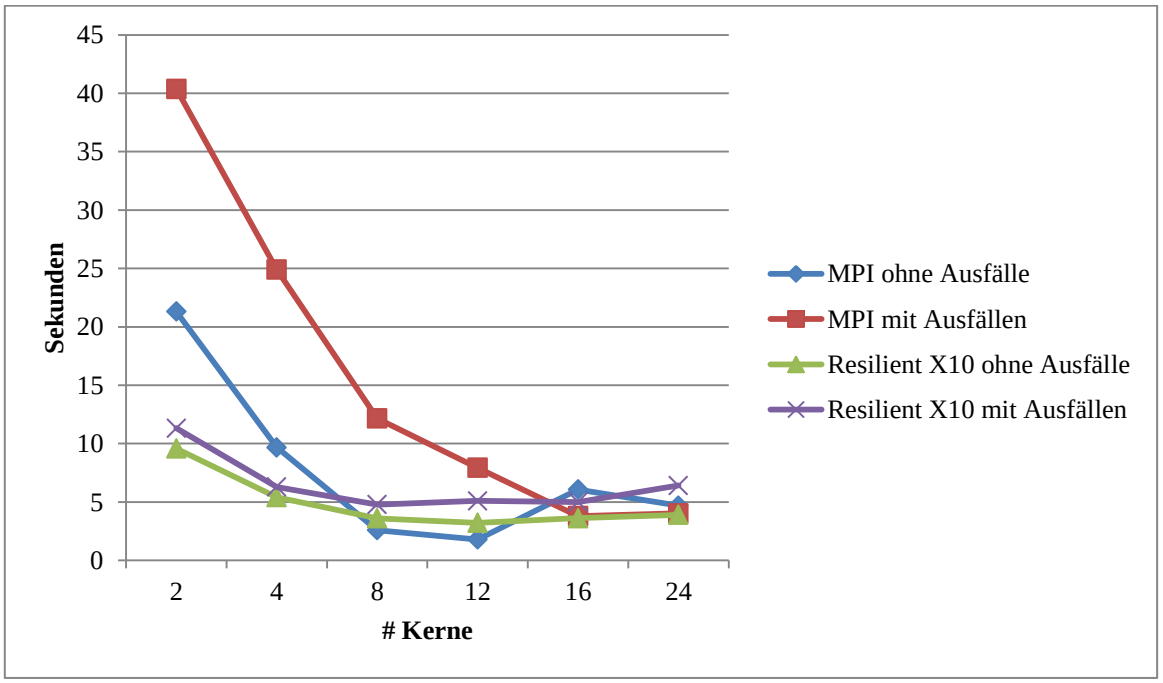


Abbildung 5: Laufzeit K-Means auf dem ITS Cluster mit 10^6 Punkten

6.3 Heat Transfer

6.3.1 Fehlertoleranz

Das Verhalten der Resilient X10 Implementierung sowie der MPI / ULFM Implementierung bei Ausfällen ist weitgehend analog zu den jeweiligen K-Means Implementierungen.

6.3.2 Programmieraufwand

Der Programmieraufwand der Implementierung des Heat Transfer Algorithmus ist bei MPI / ULFM deutlich höher als bei Resilient X10. Anders als bei den MontePi und K-Means Implementierungen erfolgt bei der Heat Transfer Implementierung bei Resilient X10 eine Fehlerbehandlung des Typs `DeadPlaceException`. Während dieser Fehlerbehandlung wird der ausgefallene Place von der Liste der lebenden Places entfernt und die Wiederherstellungsvariable `restore_needed()` auf `true` gesetzt. Bei der MPI / ULFM Implementierung erfolgt die Fehlerbehandlung zunächst analog zu den Implementierungen von MontePi und K-Means.

Der größte Unterschied zwischen der Resilient X10 Implementierung und der MPI / ULFM Implementierung liegt in der Verteilung der Daten sowie in der Sicherungs- und Wiederherstellungsstrategie. Resilient X10 bietet die Möglichkeit, ein Array mittels der Funktion `Dist.makeBlock` auf verschiedene Places zu verteilen. Der Zugriff auf die Daten eines anderen Places erfolgt mittels `at (A.dist(x-1,y)) A(x-1,y)`. Bei MPI erfolgt eine Verteilung der Daten durch das Verschicken von Nachrichten, insbesondere der Austausch der Randdaten ist deutlich aufwändiger als die Verwendung des `at`-Statements bei Resilient X10. Einen weiteren Vorteil bietet Resilient X10 bei der Sicherung und Wiederherstellung von Daten. Ein Array von Typ `ReslilientDistArray` kann mittels einer `snapshot` bzw. `restore` Funktion gesichert und bei Bedarf wiederhergestellt werden. Somit wird in der Resilient X10 Implementierung jede zehnte Iteration ein Snapshot mittels `A.snapshot();` erstellt und im Bedarfsfall eine Wiederherstellung von `A` mittels `A.restore(BigD);` durchgeführt.

Bei MPI / ULFM muss sich der Programmierer selbst um die Sicherung und Wiederherstellung der Daten bemühen. In der vorliegenden MPI / ULFM Implementierung wird analog zur Resilient X10 Implementierung in jeder zehnten Iteration eine Sicherung durchgeführt. Dies geschieht mittels der `MPI_Send` und der `MPI_Recv` Funktionen. Dabei senden alle Threads (außer root) ihren Aufgabenbereich an root. Wichtig dabei ist, dass der Zeiger des Empfangspuffers von root nicht direkt auf das Zielarray verweist. Die Daten der anderen Threads werden also nicht direkt in `ht` geschrieben sondern zunächst in ein dafür vorgesehenes Array `htTemp` zwischengespeichert. Der Hintergrund dafür ist, dass im Falle eines Ausfalls vor oder während der Empfangsfunktion (Listing 10 Zeile 95f: `rc = MPI_Recv(&(htTemp[jobGroesse*i][0]), …)`) der Empfangspuffer

undefiniert ist. Im Fehlerfall würde bei einem Empfang der Daten direkt in `ht` der aktuelle Snapshot zerstört. Gibt die Empfangsfunktion einen Fehler zurück, wird die Snapshot-Erstellung abgebrochen und anschließend der Fehler behandelt (Listing 10 Zeile 112ff).

Da ULFM als Basis für andere Fehlertolerante Modelle verwendet kann (siehe Seite 11), empfiehlt sich die Verwendung der Erweiterung Namens Fenix, welche auf ULFM basiert. Fenix bietet eine ähnliche Funktionalität wie Resilient X10 bezüglich der Sicherung von Arrays [18]. Mehr zu Fenix in Kapitel 7.

6.3.3 Laufzeit

Abbildung 6 zeigt die Ausführungszeiten der MPI / ULFM Implementierung des Heat Transfer Algorithmus. Leider traten bei der Kompilierung der Resilient X10 Implementierung einige Fehler auf, welche nicht kurzfristig behoben werden konnten. Somit kann nur die Leistung der MPI / ULFM Implementierung betrachtet werden. Analog zu K-Means Implementierung benötigt die MPI / ULFM Implementierung bei wenigen Prozessen mehr Zeit für die Berechnung, da bei Ausfällen mit weniger Prozessen gearbeitet wird. Bei der Verwendung von mehr als 12 Prozessen zeigte sich, dass die Messvariante mit Ausfällen das ITS Cluster effizienter auslasten kann.

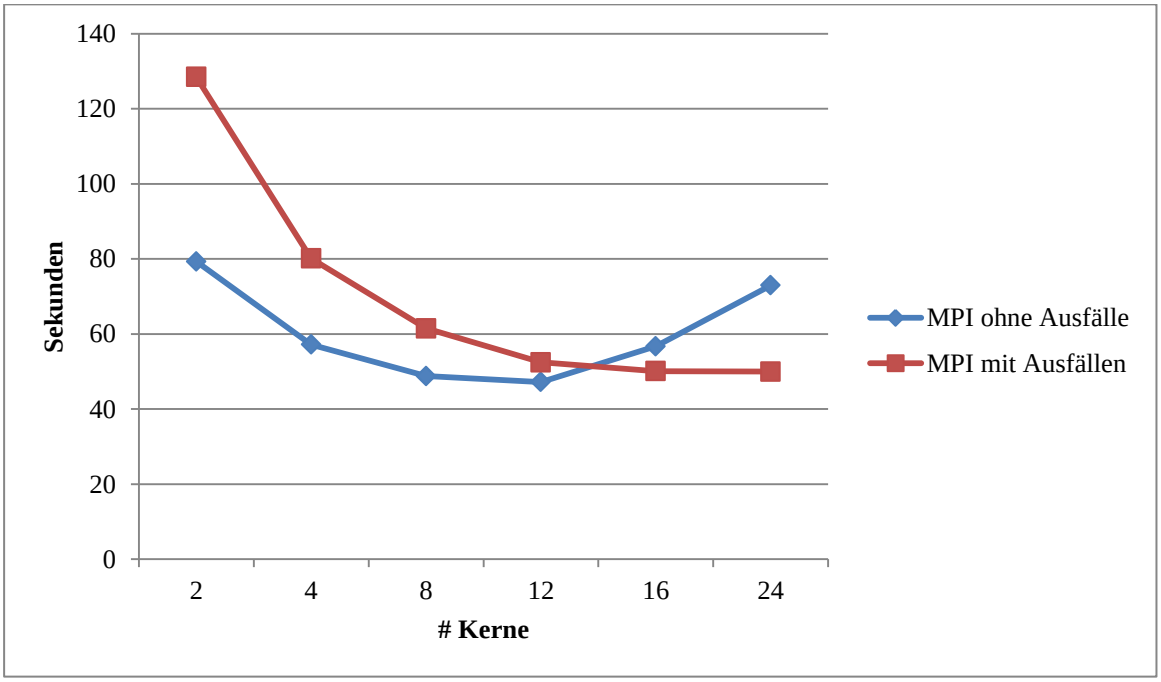


Abbildung 6: Laufzeit Heat Transfer auf dem ITS Cluster mit einer Matrix von 2000 x 2000 Punkten

7 Zusammenfassung

Bei der Implementierung der Beispielprogramme zeigte sich im Vergleich zu Resilient X10 der höhere Programmieraufwand für die Datensicherung und Wiederherstellung. Wie in 6.2.1 erwähnt, ist ULFM eine Erweiterung zur Fehlerbehandlung und keine Sicherungs- / Wiederherstellungsstrategie. Eine interessante Erweiterung Namens Fenix bietet eine solche Strategie [18]. Dabei wird zunächst analog zu den Implementierungen dieser Arbeit nach einem erkannten Ausfall der Kommunikator widerrufen und verkleinert. Anschließend werden entsprechend der Anzahl der ausgefallenen Prozesse neue Prozesse in einem temporären Kommunikator erzeugt. Für diese neuen Prozesse stehen im Idealfall Hardware Reserve Knoten zur Verfügung. Zusätzlich werden beide Kommunikatoren anschließend vereinigt, sodass nun gleich viele Prozesse wie vorher existieren. Die Daten der ausgefallenen Prozesse werden für die neu erzeugten Prozesse aus dem letzten erfolgreichen Checkpoint wiederhergestellt. Dieses Prinzip nennt sich Local Failure, Local Recovery, es werden somit nicht alle Prozesse wiederhersgestellt. Das sogenannte Diskless Checkpointing Verfahren verhindert dabei kostenintensive Festspeicherzugriffe. Die Daten werden dazu z.B. bei dem jeweiligen Nachbarn des Prozesses gesichert und von dort wiederhergestellt.

Der Leistungsvergleich fiel zugunsten von Resilient X10 aus, bei vielen Prozessen bzw. Threads konnten die MPI / ULFM Implementierungen jedoch mithalten oder waren teilweise schneller. Bei der MPI / ULFM Implementierung von Monte Pi fiel auf, dass diese praktisch keine Zeit für die Fehlerbehandlung benötigt.

Bei Hochleistungsrechnern mit sehr vielen Recheneinheiten empfiehlt sich die Aufteilung der Prozesse in verschiedene Kommunikatoren. Dies verhindert, dass bei Prozessausfällen innerhalb eines Intrakommunikators alle Prozesse (`MPI_COMM_WORLD`) zwingend eine Fehlerbehandlung durchführen müssen. Es müssen somit immer nur maximal alle Prozesse innerhalb des defekten Kommunikators eine Fehlerbehandlung durchführen.

Die Installation von ULFM verlief auf dem heimischen Server ohne Schwierigkeiten, es existiert eine gut verständliche Installationsanleitung in [8]. Lediglich die Installation auf dem ITS-Cluster bereitete Probleme, da einige für die Installation von ULFM benötigten Programmversionen nicht aktuell waren. An dieser Stelle sei ein Dankeschön für die freundliche und kompetente Hilfe an das ITS Team ausgesprochen.

Literaturverzeichnis

[1] Message Passing Interface Forum, Document for a Standard Message-Passing Interface, Draft, NFS contract CDA-9115428, 15. September 2014

[2] Wesley B. Bland. Toward Message Passing Failure Management, Dissertation, University of Tennessee, Knoxville, Mai 2013

[3] David Cunningham, David Grove, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Hiroki Murata, Vijay Saraswat, Mikio Takeuchi, Olivier Tardieu. Resilient X10: Efficient failure-aware programming, pages 67-80, Orlando, Florida, USA, 2014

[4] Kiyokuni Kawachiya. Writing Fault-Tolerant Applications Using Resilient X10, IBM Research, Tokyo, Japan, 2015

[5] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, David Grove. X10 Language Specification Version 2.5., 3. Oktober 2014

[6] Christian Schaub. Fallstudien zur fehlertoleranten Programmierung mit Erlang, Bachelorarbeit, Universität Kassel, 16. September 2015

[7] Keita Teranishi, Michael A. Heroux. Toward Local Failure Local Recovery Resilience Model using MPI-ULFM, Kyoto, Japan, September 2014

Webseiten

[8] Fault Tolerance Research Hub, http://fault-tolerance.org/

[9] TOP500 Supercomputer Sites, http://top500.org/

[10] Fault Tolerance Research Hub. SC'15 tutorial, http://fault-tolerance.org/2015/10/02/sc15-tutorial/

[11] X10, Productivity and Performance at Scale, sourceforge, resiliency samples http://sourceforge.net/p/x10/code/HEAD/tree/trunk/x10.dist/samples/resiliency/

[12] The X10 Programming Language, http://x10-lang.org/

[13] Message Passing Interface Forum, http://www.mpi-forum.org/

[14] Wikipedia. X10 (Programmiersprache), https://de.wikipedia.org/wiki/X10_%28Programmiersprache%29

[15] Wikipedia. Partitioned Global Address Space, https://de.wikipedia.org/wiki/Partitioned_Global_Address_Space

[16] X10, Productivity and Performance at Scale, sourceforge, Readme, http://sourceforge.net/p/x10/code/HEAD/tree/trunk/x10.dist/samples/resiliency/README.txt

[17] Wikipedia. Message Passing Interface, https://de.wikipedia.org/wiki/Message_Passing_Interface

[18] Marc Gamell, Keita Teranishi, Manish Parashar. Fenix: Online Failure Recovery on top of ULFM, http://fault-tolerance.org/downloads/ulfm-bof-sc15/Fenix%20-%20GAMELL,Marc.pdf

[19] MPICH, https://www.mpich.org/

[20] Open MPI, Open Source High Performance Computing, http://www.open-mpi.org/

[21] MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE, http://mvapich.cse.ohio-state.edu/

[22] Hilfesystem für den Message Passing Interface Standard MPI, https://www.tu-chemnitz.de/informatik/RA/projects/mpihelp/mpihelp.html

[23] Martin Scholz, The Message Passing Interface: MPI 3.1 and Plans for MPI 4.0, http://meetings.mpi-forum.org/2014-11-scbof-intro.pdf

[24] George Bosilca, Keita Teranishi, Marc Gamell, Tsutomu Ikegami, Sara Salem Hamouda. FAULT TOLERANT MPI APPLICATIONS WITH ULFM, Austin, TX, USA, 2015, http://fault-tolerance.org/downloads/ulfm-bof-sc15/ULFM%20Bof%20SC15.pdf

[25] X10, Productivity and Performance at Scale, sourceforge, Monte Pi, http://sourceforge.net/p/x10/code/HEAD/tree/trunk/x10.dist/samples/resiliency/ResilientMontePi.x10

[26] Wikipedia. k-Means-Algorithmus, https://de.wikipedia.org/wiki/K-Means-Algorithmus

[27] Wikipedia. Stencil code, https://en.wikipedia.org/wiki/Stencil_code

[28] ITS Handbuch: Wissenschaftliche Datenverarbeitung, https://www.uni-kassel.de/its-handbuch/daten-dienste/wissenschaftliche-datenverarbeitung.htm

Anhang A - Messergebnisse

Monte Pi:

Tabelle 1: MPI / ULFM Laufzeiten ohne Ausfälle auf ITS Cluster

24 Cores	16 Cores	12 Cores	8 Cores	4 Cores	2 Cores
51,647059	46,601462	42,405687	63,757259	127,393063	254,351741
52,441438	45,257244	42,402064	63,437147	127,356439	252,557011
50,714862	45,397314	42,396712	63,589453	127,186423	252,768574
53,499404	45,384123	42,406949	63,602809	126,264137	254,174615
47,788305	45,643769	42,563654	63,599836	126,91671	254,487738
51,42547	44,76778	42,40337	63,720937	126,658046	252,368254
50,409171	45,072158	42,784937	63,499691	126,414584	254,576576
49,962419	45,764717	42,416998	63,590881	126,717927	254,627214
51,01471	45,443813	42,388163	63,476518	127,616963	255,260355
49,088345	46,322951	42,321599	63,161305	127,09294	253,051403
47,788305	44,76778	42,321599	63,161305	126,264137	252,368254

Tabelle 2: MPI / ULFM Laufzeiten mit Ausfällen ITS Cluster

24 Cores	16 Cores	12 Cores	8 Cores	4 Cores	2 Cores
21,127855	31,651264	41,792652	63,354245	124,639537	251,071409
21,014529	31,508096	41,860524	62,737111	124,737867	248,445426
20,981633	31,639058	42,223591	62,33381	125,435435	251,302355
21,117122	31,458833	42,016821	62,731471	125,915306	250,673067
21,062899	31,483829	42,052372	62,910955	125,355348	249,328516
20,996904	31,679495	42,119159	62,25449	124,535821	249,575889
21,109419	31,580321	41,943659	63,040677	126,363047	248,433927
21,094896	31,662074	42,011947	62,550413	125,423852	248,327349
21,125415	31,383218	41,884178	63,074062	125,550348	248,442714
21,113428	31,551027	41,924694	62,78125	125,22745	248,933279
20,981633	31,383218	41,792652	62,25449	124,535821	248,327349

K-Means:

Tabelle 3: MPI / ULFM Laufzeiten ohne Ausfälle ITS Cluster

24 Cores	16 Cores	12 Cores	8 Cores	4 Cores	2 Cores
5,544326	9,079453	7,095972	3,059515	10,629826	45,111572
7,353619	8,195909	8,935182	4,839807	14,706303	29,743028
8,970316	7,366852	5,283525	9,055454	9,680433	24,828273
5,553762	7,339651	8,816165	5,849454	20,353151	24,828273
4,68378	8,019066	7,29094	9,77378	17,963256	28,435666
10,795947	15,661559	6,749724	2,587556	21,576371	36,349876
8,55108	8,559379	1,79478	5,918834	15,285236	42,576839
8,124066	6,066831	8,345854	8,997544	13,961769	21,307433
7,095621	10,995504	8,551154	3,133207	13,695474	68,037777
6,599737	12,238686	2,062862	8,760343	26,867188	35,19089
4,68378	6,066831	1,79478	2,587556	9,680433	21,307433

Tabelle 4: MPI / ULFM Laufzeiten mit Ausfällen ITS Cluster

24 Cores	16 Cores	12 Cores	8 Cores	4 Cores	2 Cores
12,50534	14,75412	12,728821	14,953261	80,995478	40,379565
8,593963	8,720112	13,570968	18,278448	29,087174	53,119938
6,254612	10,010885	12,745718	24,45203	24,90004	53,809442
11,052144	4,171329	14,930615	22,644178	41,982887	68,852152
4,819133	4,964518	7,932706	21,906254	42,146322	54,700427
6,466897	9,167306	14,564366	18,670229	37,177329	50,544754
6,994696	9,791689	24,949164	15,554807	36,03178	66,495345
4,022166	12,45688	15,026805	12,155517	40,867736	45,967342
9,015198	12,294262	8,9356	12,35118	25,795803	144,683836
7,811179	3,796175	21,315066	19,82869	27,047364	62,658563
4,022166	3,796175	7,932706	12,155517	24,90004	40,379565

Heat Transfer:

Tabelle 5: MPI / ULFM Laufzeiten ohne Ausfälle ITS Cluster

24 Cores	16 Cores	12 Cores	8 Cores	4 Cores	2 Cores
72,959572	56,706304	47,462152	51,62472	57,999218	80,147958
79,440381	59,602427	48,842405	51,715217	63,409385	85,38664
80,553051	62,63703	47,191621	50,873431	58,742394	86,686141
79,853053	59,187717	47,446766	53,869013	57,757863	79,704714
83,95614	60,339233	49,258812	49,627865	57,591794	79,672811
81,621256	63,345641	47,272008	48,769799	58,258131	79,563141
78,393941	61,650002	47,313551	52,059285	57,169956	79,297169
82,482985	63,529796	47,422945	55,146972	67,928463	87,17059
82,41122	61,139166	49,637522	63,759036	69,085522	79,472592
78,394914	61,681269	47,332767	59,888756	58,104124	79,64768
72,959572	56,706304	47,191621	48,769799	57,169956	79,297169

Tabelle 6: MPI / ULFM Laufzeiten mit Ausfällen ITS Cluster

24 Cores	16 Cores	12 Cores	8 Cores	4 Cores	2 Cores
51,877496	50,377631	54,233999	61,482563	89,700423	129,603079
55,407619	51,691323	57,992723	61,846184	80,266207	131,551786
51,966767	50,722398	58,157118	61,834557	85,866691	129,472403
50,132203	51,417449	62,185145	63,390698	80,690494	128,737607
52,019674	50,081685	53,317681	62,530003	83,623992	132,199525
50,401699	50,233965	54,648794	65,919044	80,245357	128,676468
52,467558	50,822396	52,464749	64,776027	85,7098	136,032389
49,963556	51,3569	53,683788	63,523961	80,127093	128,499004
51,553986	50,723447	54,516601	62,821868	84,488265	129,667336
50,12818	51,285269	53,845653	62,841385	83,654972	129,846117
49,963556	50,081685	52,464749	61,482563	80,127093	128,499004