Praktikum - High Performance Computing for Machine Intelligence in C++/Python

Abschlussbericht

Authored by
Ala Fnayou
Daniel Duclos-Cavalcanti
Efstathios Zafeiriadis Petridis

1. Einführung

Der folgende Abschlussbericht stellt die Arbeit dar, die von der Gruppe 5 im Rahmen des Hauptprojekts vom Praktikum High Performance Computing an der TUM durchgeführt wurde.

1.1 Problemstellung

Das Projekt liefert eine Lösung für ein spezifisches Navigationsproblem. Bei einer Menge von Sternen im Weltraum zielt das Problem darauf ab, die optimale Trajektorie eines Raumschiffs zwischen einem Quell- und einem Zielstern zu berechnen. Es handelt sich dabei um ein stochastisches Navigationsproblem, das auf iterative Weise gelöst werden kann. In diesem Zusammenhang wird ein Asynchronous Value Iteration (**AVI**) Modell berechnet. Aus der Sicht des AVI-Modells wird ein Satz von Zuständen zusammen mit einem Satz von Aktionen definiert. Wenn sich das Raumschiff beispielsweise auf einem bestimmten Stern befindet, dann befindet es sich auch in einem bestimmten Zustand. Das Raumschiff kann Aktionen ausführen, die es in einen neuen Zustand/Stern bringen. Die Durchführung einer Aktion hängt direkt von einer bestimmten Wahrscheinlichkeitsmatrix **P** ab, die jedem Start-Zustand eine bestimmte Wahrscheinlichkeit für die Durchführung einer Aktion zuweist. Nach der Ausführung einer Aktion können die Kosten berechnet werden. Das Ziel ist es, die optimale Trajektorie **J** (minimale Kosten für jeden Zustand) und die optimale Strategie **Pi** (die optimale Aktion für jeden Zustand) zu generieren.

2. Methoden und Implementierungen

Im Rahmen dieses Projekts wurde eine hochleistungsfähige Lösung des Problems gefordert. Dies wurde durch die Aufteilung der zu verarbeitenden Arbeitslast und die Parallelisierung ihrer Berechnung auf mehrere Rechner der LDV erreicht. Diese parallele Durchführung wurde mit Hilfe einer Message Passing Interface Library erreicht, nämlich OpenMPI. Darüber hinaus wurden von jedem Mitglied der Gruppe verschiedene Implementierungen mit Hilfe verschiedener OpenMPI-Funktionen und -Methoden entwickelt. Schließlich wurde ein Benchmarking-Prozess durchgeführt, um die vorgeschlagenen Implementierungen zu bewerten sowie zu vergleichen.

2.1 Projekt Architektur

Die Architektur des Projekts lässt sich nach dem Paradigma der objektorientierten Programmierung Paradigma beschreiben. Genauer gesagt wurde hier der *bridge pattern* benutzt. Diese Abstraktion implementiert eine Basisklasse, wovon verschiedenen Sub-Klassen erben und dadurch die interne Implementation von einem Algorithmus auf ihre eigene Weise umsetzen. So kann

der selbe API/Klasse verschiedene Implementationen von der selben Funktion haben. Die Basis Klasse in dem Fall ist *ValueIteration*, die für die Ausführung der Value Iteration modell alle wichtige Parameter und Methoden enthält. Außerdem besitzt die Basisklasse zwei virtuelle Funktionen, nämlich *implementation* und *partialValueIteration*. Diese zwei sehr wichtige Funktionen werden in dem Abschnitt **2.1.1** weiter erklärt.

Diese Architektur ermöglicht dass, nicht nur jede geerbte Klasse alle notwendige Variablen und Methoden schon besitzt, sondern auch nur die virtuelle Methoden überschrieben werden müssen, um ein komplettes Model zu implementieren. Noch dazu hat jede neue Subklasse, die Möglichkeit Ihre eigene Methoden und Variablen zu implementieren, sowie für jeden Fall gebraucht wird. Diese Beziehungen können durch den Figure 1 gut dargestellt werden.

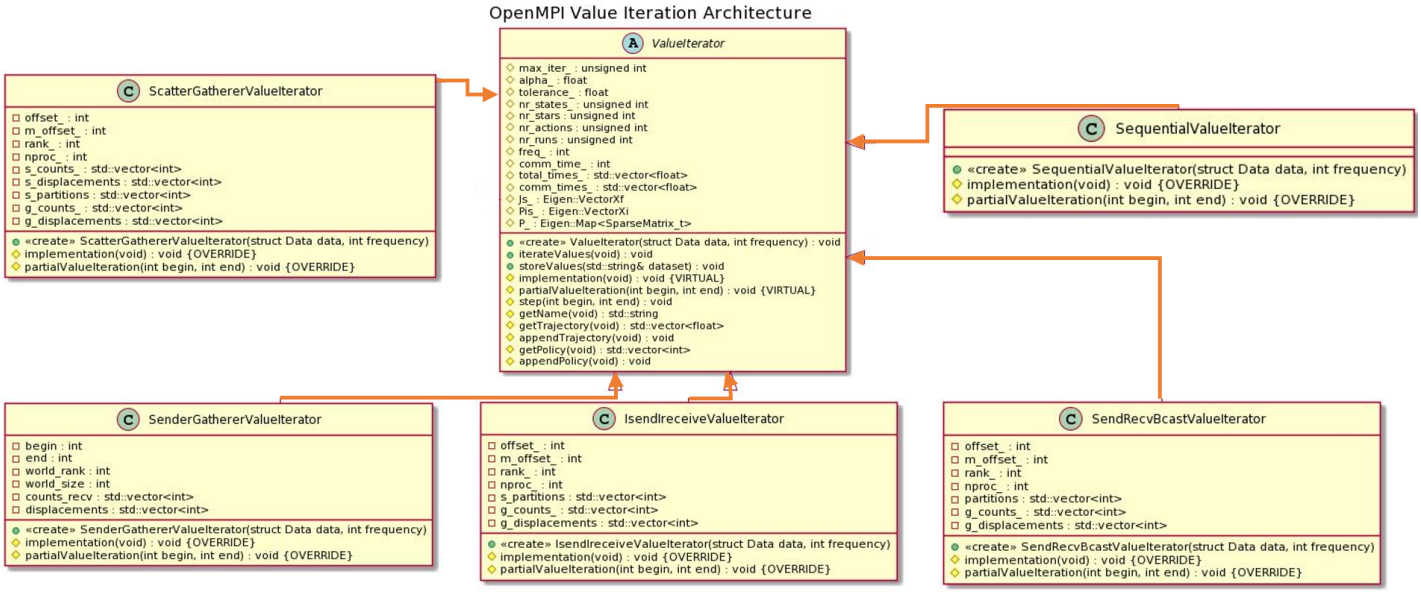

Figure 1: Projekt Architektur

Zusätzlich ist jede Klasse/Objekt ihre eigene Welt. Jedes Objekt hat Zugang zu allen notwendigen Variablen wie (`J_`), (`Pi_`) und (`P_`) und kann andere Parameter wie (`nr_runs_`) oder (`freq_`) auch aktualisieren. Alle Sub-Klassen speichern die notwendige Ergebnisse genau gleich. Das wird von der geerbten *storeValues()* Funktion ausgeführt. Diese Methode speichert als `.npz` file:

1. Wie oft dieses Objekt sein Value Iteration Modell gelaufen hat
2. Ausführungszeiten pro Lauf.
3. Kommunikationszeiten pro Lauf.
4. *Trajektorie* **J** und *Strategie* **Pi** pro Lauf
5. Die Anzahl von Slots/Prozessoren.
6. Die Anzahl von States und Actions.
7. Die Frequenz während Value Iteration Ergebnisse mit anderen Prozesse geteilt wurde.

Noch zwei Prozesse sind wichtig zum erklären der Architektur. Die sind die *struct Data* und *struct Transformer* Datenstrukturen. Als Zusammenfassung diese Datenstrukturen sind kompakte Formen wie alle notwendige Daten zum initialisieren der *ValueIteration* Klasse oder eine Sub-Klasse davon gegeben werden können. Sie werden am Anfang des Programms initialisiert und dann als *Constructor* Parameter zu jeder Klasse gegeben. Sie enthalten die notwendige Parameter für das Value Iteration Modell. Allerdings gibt es noch eine Variable, die besonders erklärt werden muss, was auch durch den *struct Transformer* transportiert wird. Das ist der `coord_map`.

2.1.1 Coordinates Map

Die Koordinationskarte wird in der Datenstruktur *struct Transformer* gespeichert. Diese stark verschachtelte Datenstruktur würde auf einer ersten Ebene eine Reihe von Vektoren für jeden Zustand in der Wahrscheinlichkeitsmatrix enthalten. Auf der zweiten Ebene der Verschachtelung würde jeder Zustand nur die Aktionen enthalten, die er ausführen kann. Mit anderen Worten: Jeder Zustand enthält nur Aktionen mit einer Wahrscheinlichkeit ungleich Null. Auf der letzten Ebene der Verschachtelung werden schließlich die Koordinaten dieser Elemente, gespeichert. Die in der Initialisierungsphase des Programms berechnete Koordinatenkarte würde es den Prozessen ermöglichen, die Zustände so effizient wie möglich zu iterieren, was zu einer drastischen Abnahme der Ausführungszeit führen würde.

The Coordination map will be stored within the *struct Transformer* data structure. This highly nested data structure would hold on a first level, a set of vectors for each state in the probability matrix. On the second level of nesting, each state would hold only the actions that it is capable of taking. In other words, each state would hold only the non-zero actions. Finally, in the last level of nesting, coordinates of the non-zero elements are stored. Once computed at the initialization phase of the program, the coordinates map would enable the processes to iterate the states in the most efficient way possible which resulted in a dramatic decrease in execution time.

2.1.2 Virtuelle Methoden

Partial Value Iteration: *partialValueIteration(int begin, int end)*

Diese Funktion sollte immer einen Start- und Endindex aus dem Zustandsvektor erhalten, der die genauen Zustände angibt, die es zu berechnen hat. Deshalb muss dann diese Methode die optimale *Trajektorie* **J** und die optimale *policies* **P** für diese Bandbreite von Zuständen berechnen. Das Ziel von Projekt ist natürlich das ganze Value Iteration durch MPI und verschiedene Prozesse zu parallelisieren. Jedoch, damit der AVI erfolgreich und richtigerweise kalkuliert wird, müssen Parameter wie der *error* und der *Trajektorie* **J** unter alle Prozesse verteilt werden. Deswegen ist diese Methode eine virtuelle Methode. Jedes MPI Kommunikationschema kann oder sollte individuell bzw. anders implementiert werden und daher sollte der Mechanismus zur Aktualisierung der genannten Parameter auch dazu passen.

MPI Implementation: *implementation()*

Diese Funktion ist der kern von jeder Sub-Klasse. Hier ist wo jede Implementation ihre eigene Art von Verteilung und sammeln der Arbeitslastung mithilfe der MPI API umsetzt. Diese Methode wird intern von *iterateValues()* gerufen und kümmert sich um die Berechnung der Ergebnisse. Die Ergebnisse müssen dann auf dem *master* Prozess am Ende gesammelt werden, damit sie richtigerweise gespeichert werden können. Wie jede Klasse ihre *implementation()* Funktion strukturiert hat, wird in Kapitel **2.2** erklärt werden.

2.2 Sub-Klassen und MPI Schemas

Die folgenden Unterabschnitte erklären wie jede Sub-Klasse bzw. Kommunkationsschema gebaut wurde. Diese Klassen sind: *ScatterGatherer*, *SendRecvBcast*, *SenderGatherer* und *IsendIreceive*. Die *Sequential* Klasse wurde eher als Beispiel implementiert und wird nur als einen möglichen basis Vergleich zur Parallelisierung der Value Iteration Modell genutzt.

2.2.1 ScatterGatherer

Dieses Schema wurde von Daniel Duclos-Cavalcanti geschrieben. Es teilt die J (`nr_states_`) unter alle Prozessoren mithilfe der Funktion `MPI_Scatterv`. Die Verteilung wird von *master* Prozess gerechnet und verbreitet. Danach wird jeder Prozess seine Bandbreite von Zustände rechnen und mit einer bestimmten Frequenz *freq_* den *error* und die *Trajektorie* **J** unter alle Prozesse aktualisieren. Für die Aktualisierung der Fehler wird die `MPI_Allreduce` Funktion benutzt, damit der größte Fehler zu Jedem Prozess synchronisiert wird. Für die Aktualisierung der Trajektorie wird die `MPI_Allgatherv` benutzt, um die gesamte Trajektorie unter alle Prozesse zu verteilen. Die Politik Pi wird am Ende bei *master* `MPI_Gatherv` mitgeteilt.

2.2.2 SendRecvBcast

Dieses Schema wurde von Daniel Duclos-Cavalcanti geschrieben. Es teilt die J (`nr_states_`) unter alle Prozessoren, aber hier wird das bei jedem Prozess selbst kalkuliert. Danach wird jeder Prozess seine Bandbreite von Zustände rechnen und mit einer bestimmten Frequenz *freq_* den *error* und die *Trajektorie* **J** unter alle Prozesse aktualisieren. Für die Aktualisierung der Fehler wird jeder Prozess, der nicht *master* ist, seine Trajektorie und seinen Fehler zu *master* mithilfe der blockierenden `MPI_Send` Funktion senden. Dann wird *master* den größten Fehler mit `MPI_Bcast` verbreitet und auch die gesamte Trajektorie mit der selben Funktion gemacht.

2.2.3 IsendIreceive

Dieses Schema wurde von Ala Fnayou entwickelt und es beginnt mit der Berechnung der Arbeitslast für den Hauptprozess und mit Hilfe von `MPI_Bcast`, wird das Ergebnis an alle anderen Prozesse übertragen. Dies würde es jedem anderen Prozess ermöglichen, die eigene Bandbreite

der Zustände zu identifizieren. Während der Ausführung von *partialValueIteration()* wird jeder J-Vektor in jedem Prozess mit allen anderen Prozessen mittels `MPI_Allgatherv` ausgetauscht, um die Genauigkeit und Konsistenz der Ergebnisse zu gewährleisten. Zusätzlich wird der maximale Fehler durch den Masterprozess mittels `MPI_Allreduce` von den anderen Prozessen abgeholt. Schließlich werden der Strategie-Vektor Pi und der Trajektorien-Vektor J von jedem Prozess durch eine Kombination von nicht-blockierenden Sende- und Empfangsfunktionen, nämlich `MPI_Isend` und `MPI_Irecv`, an den Masterprozess gesendet. Die Synchronisation der beiden Prozesse wurde mit der Funktion `MPI_Wait` und einer for-Schleife im Master-Prozess erreicht.

2.2.4 SenderGatherer

Dieses Schema wurde von Efstathios Zafeiriadis Petridis geschrieben und verwendet den `MPI_Gatherv`. Die Größe von J, `j_size` , wird durch die Anzahl der Prozesse, `world_size` , geteilt. Dann wird die ceil-Funktion verwendet, um die nächste ganze Zahl zu finden, und jeder Prozess nimmt die gleiche Anzahl von Elementen *ceil(j_size / world_size)*, aber der letzte Prozess nimmt auch die verbleibenden Zustände, falls es welche gibt. Nachdem die partialValueIteration konvergiert hat, d.h. der Fehler kleiner als die Toleranz ist, sendet jeder Prozess seinen Teil der Strategie an den Master-Prozess mit `MPI_Gatherv`. J wird an allen Prozessen mit `MPI_Allgatherv` kommuniziert. Der maximum error über alle Prozessen wird mit `MPI_AllReduce` kommuniziert.

2.3 Ergebnisse

In den folgenden Abschnitten werden die wichtigsten Ergebnisse der Arbeit dargestellt.

2.3.1 Gesamtausführungszeit

Die Gesamtausführungszeit wurde unter Variation zweier Parameter gemessen, nämlich der Anzahl der Prozesse (`nr_proc_`) und die Kommunikationsfrequenz (`freq_`). Zunächst wurde die Kommunikationsfrequenz auf 14 festgelegt, während die Anzahl der Prozesse erhöht wurde. Die folgende Abbildung 2 zeigt, dass eine Erhöhung der Anzahl der Prozesse zu einer deutlichen Verringerung der Gesamtausführungszeit führt.

Abbildung 3 zeigt eine Zunahme der Ausführungszeit, die proportional zur Zunahme der Kommunikationsdauer ist.

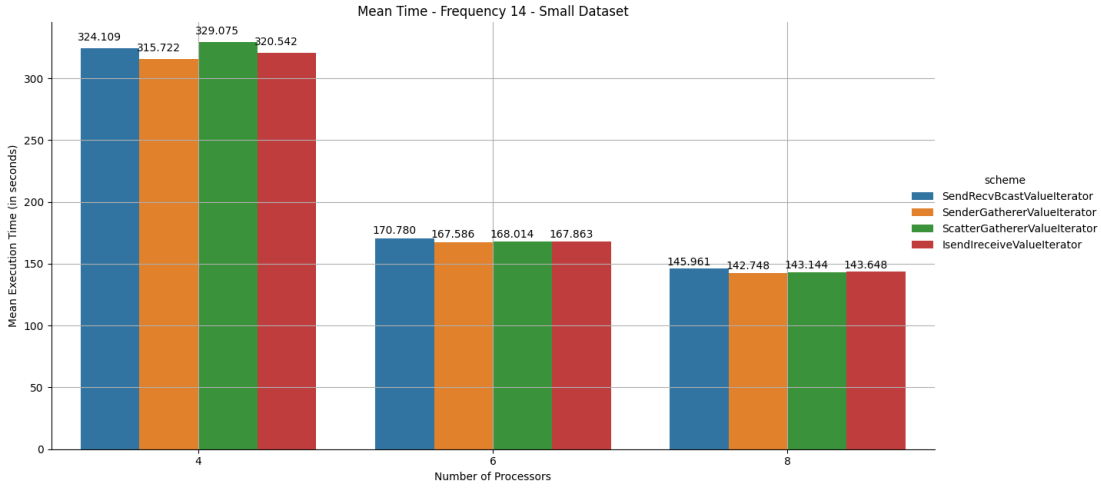

Figure 2: Gesamtausführungszeiten mit 14 als Kommunikationsfrequenz

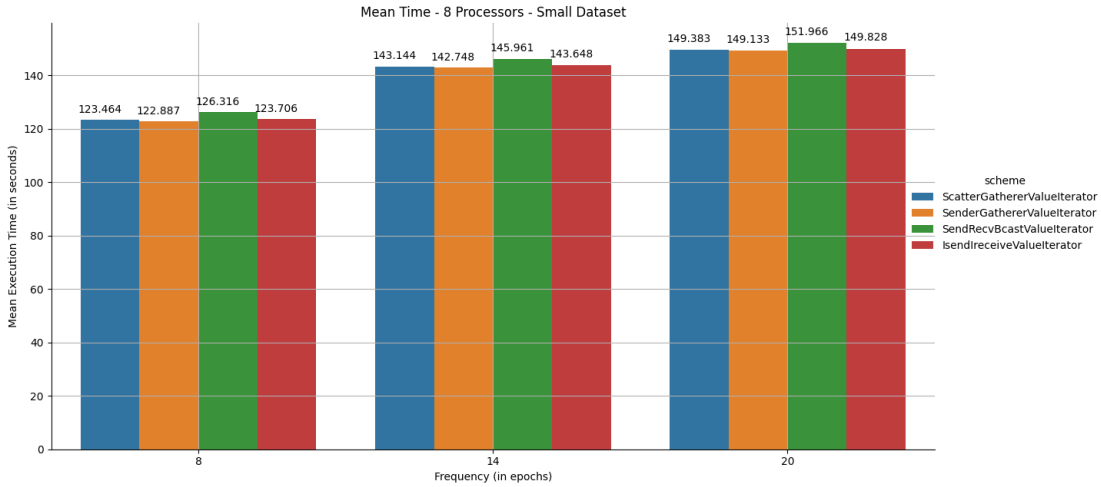

Figure 3: Gesamtausführungszeiten mit 8 Prozesse

Das beobachtete Ergebnis ist insofern etwas kontraintuitiv. Das kann daran liegen dass der Algorithmus mehr Zeit für die Konvergenz benötigt, da weniger Kommunikation zwischen den Prozessen stattfindet. Wenn die Funktion *partialValueIteration* beispielsweise, wie in Abbildung 4 dargestellt ist, nach 89 Epochen konvergieren sollte, dann konvergiert sie bei `freq_ = 8` in Epoche 96, bei `freq_ = 14` in Epoche 98 und bei `freq_ = 20` konvergiert sie bei Epoche 100. Daher wird `freq_ = 8` weniger Zeit für die Ausführung benötigen.

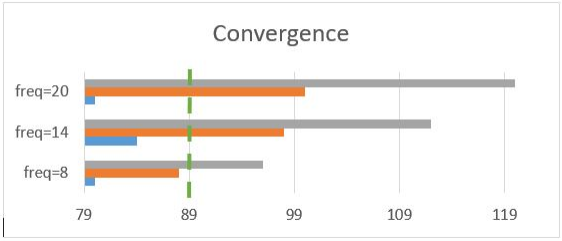

Figure 4: Konvergenz Rate

2.3.2 Kommunikationszeit

Zusammen mit der Gesamtausführungszeit wurde auch die Kommunikationszeit gemessen, um aufzuzeigen, wie jedes Kommunikationsschema auf eine Veränderung der Anzahl der Prozesse oder der Kommunikationshäufigkeit reagieren würde. Dies dient auch dazu, die verschiedenen Kommunikationsschema miteinander zu vergleichen und zu bewerten.

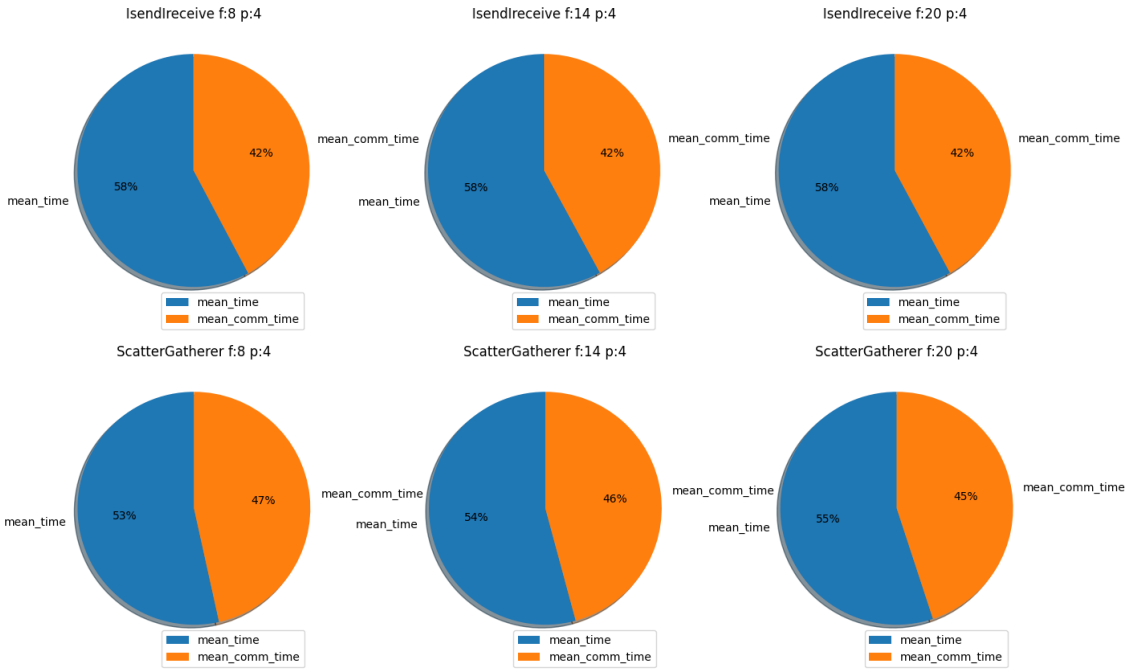

Figure 5: Vergleich von den Prozentsätze der Kommunikationzeiten bei ScatterGatherer and IsendIreceive Schemas

Wie in Abbildung 5 zu sehen ist, weist das ScatterGatherer-Schema bei gleicher Anzahl von Prozessen und gleicher Kommunikationshäufigkeit einen höheren Prozentsatz an Kommunikationszeit von bis zu 47% auf, im Vergleich zu einem Prozentsatz von 42% für das IsendIreceive-Schema. In einer Umgebung mit 4 Prozessen und 8 als Kommunikationshäufigkeit könnte dieser Unterschied von 5% stark mit der Tatsache zusammenhängen, dass das IsendIreceive-Schema nicht blockierend ist, während das ScatterGatherer-Schema einen zusätzlichen MPI-Aufruf hat, um die Daten über die Prozesse zu verteilen.

3. Beitrag im Abschlussbericht/Projekt

- 1. Einführung: Ala Fnayou
- 1.1 Problemstellung: Ala Fnayou
- 2. Methoden und Implementierungen: Ala Fnayou
- 2.1 Projekt Architektur: Daniel Duclos-Cavalcanti
- 2.1.1 Coordinates Map: Ala Fnayou
- 2.1.2 Virtuelle Methoden: Daniel Duclos-Cavalcanti
- 2.2 Sub-Klassen und MPI Schemas Daniel Duclos-Cavalcanti
- 2.2.1 ScatterGatherer Daniel Duclos-Cavalcanti
- 2.2.2 SendRecvBcast Daniel Duclos-Cavalcanti
- 2.2.3 IsendIreceive Ala Fnayou
- 2.2.4 SenderGatherer: Efstathios Zafeiriadis Petridis
- 2.3 Ergebnisse: Efstathios Zafeiriadis Petridis
- 2.3.1 Gesamtausführungszeit Efstathios Zafeiriadis Petridis
- 2.3.2 Kommunikationszeit Efstathios Zafeiriadis Petridis