Technische Universität München
Lehrstuhl für Datenverarbeitung
Prof. Dr.-Ing. Klaus Diepold

High Performance Computing for Machine Intelligence: Gruppe 3

Autoren: *Till Hülder*, *Tobias Klama*, *Tobias Krug*

***Zusammenfassung*— Verloren in Raum und Zeit? Nicht mehr! Für alle die regelmäßig eine Ausfahrt auf dem Weg von Terra nach Alpha Centauri verpassen und unterwegs mit leerem Tank auf einem leeren Planeten landen, haben wir eine optimale Lösung entwickelt: skalierbare asynchrone Value Iteration per Open MPI. Ziel dieser Ausarbeitung ist die Einführung in die relevanten Hintergründe zu Open MPI und darauf aufbauend die Motivation eines Projektaufbaus, der die Beurteilung verschiedener Kommunikationsschemata und Parametrierungen erlaubt. Mittels dieses Frameworks können wir aus drei MPI Schemata, sechs Ausführungsumgebungen und diversen Parameterkombinationen je nach Größe des Problems und zur Verfügung stehender Rechenumgebung eine zielführende Kombination ableiten. Die Kernergebnisse sind die Identifikation verschiedener Zusammenhänge zwischen MPI Kommunikationsschema, Rechenumgebung und Parametrierung und Qualitätsmetriken wie Rechenzeit, Speicherbedarf und Lösungsqualität. Diese erlauben eine optimale Anpassung des Projekts an die jeweiligen Rahmenbedingungen.**

Keywords*—*High Performance Computing, Parallel Processing, Reinforcement Learning, Machine Intelligence

I. Einführung

HIGH Performance Computing bezeichnet seit einiger Zeit eine Technik zur Verknüpfung einzelner Standardcomputer zu einem leistungsfähigen Konglomerat: „In 1988, an article appeared in the Wall Street Journal titled “Attack of the Killer Micros” that described how computing systems made up of many small inexpensive processors would soon make large supercomputers obsolete.“ [1, S. 3] Entsprechend dieser Vision können wir heute auf Systeme zurückgreifen, die aus Standardcomputern performante Cluster bilden.

Diese Arbeit befasst sich mit der Implementierung eines Optimierungsproblems aus dem Reinforcement Learning Umfeld auf genau solchen Clustern. Das Problem, welches wir lösen ist die Suche einer realisierbaren und – bezogen auf eine Kostenfunktion – optimalen Route zwischen zwei Planeten in einem theoretischen Raumfahrtnavigationsszenario. Hierzu wenden wir den bekannten Value Iteration Algorithmus (TODO: Referenz?) in seiner asynchronen Form an. Die Implementierung der Value Iteration (TODO: abbreviations einbauen) erfolgte mittels C++ und dem Open MPI (TODO: ref) Framework.

Die vorliegende Ausarbeitung befasst sich mit der abstrakten Idee der Umsetzung des oben genannten Projekts und der Struktur der Testautomatisierung. Weiterhin wird eine Analyse und Einordnung der Resultate vorgenommen. Für detaillierte Einblicke in die Implementierung verweisen wir auf die Softwaredokumentation in Form der Markdown Readme Datei und Doxygen Dokumentation.

Das Hauptmerkmal unserer Ausarbeitung ist die umfangreiche Durchführung von Benchmarks mittels Variation der Größen Datensatz, Testumgebung, MPI Schema und MPI Parametrierung.

Die Umsetzung fußt auf einer konkreten Formulierung eines Projektplans, welcher Inhalt und Umfang des Projekts absteckt. Um das Ziel einer funktionsfähigen Implementierung und einer aussagekräftigen Analyse zu erreichen, setzen wir auf Ansätze der SCRUM Methodik, um mittels regelmäßiger Meetings und ausgeprägter Nutzung von Issues und Branches regelmäßigen Fortschritt zu erreichen.

Diese Ausarbeitung startet in II mit einer Erläuterung der Projektstruktur und zeigt darauf aufbauend welche Testmöglichkeiten sich hiermit bieten. Anhand dreier Schemata validieren wir die automatisierte Erfassung und Verarbeitung von Messdaten. Die so gewonnenen Ergebnisse werden in III mit einer vergleichenden Perspektive auf getestete Schemata und Ausführungsumgebungen analysiert. In IV behandeln wir konkrete Thesen, welche im HPC (TODO: abbreviation) Kontext auftreten. Den inhaltlichen Abschluss bilden eine Aufstellung der wesentlichen Erkenntnisse in V und eine Darstellung unserer Beiträge in VI. Für weitergehende Einblicke in die Ergebnisse der Arbeit, schlüsseln wir im Anhang in -A die Ergebnisse je Datensatz, Testumgebung und MPI Schema auf.

II. Methodik

Zur erfolgreichen Durchführung umfassender Benchmarks setzen wir auf eine flexible Softwarearchitektur, welche eine einfache Parametrierung von vorhandenen MPI Schemata erlaubt. Diese Ansatz wird nachfolgend erläutert.

A. Softwarearchitektur

Die Softwarearchitektur des Projekts setzt auf zwei wesentlichen Prinzipien auf:

1) Parameter structs für Value Iteration (TODO: abbreviation), MPI und Logging dienen als leichtgewichtige Umsetzung des Flyweight Patterns [2, S.195ff.]. Dieses Vorgehen erlaubt das unkomplizierte Lesen und Schreiben von Parametern, Laufzeitgrößen zur Steuerung und Messwerten ohne Kopien.
2) Eine an MVC [3, S.125ff.] angelehnte und durch das Flyweight vernetzte Struktur. Diese besteht aus generischer Value Iteration (TODO: abbreviation) Implementierung (Model), flexibler Eingabe von Steuergrößen und Ausgabe von Messwerten (View) und verschiedenen, interfacekompatiblen MPI Schemata (Controller).

B. Automatisierung

Aufbauend auf der flexiblen Softwarearchitektur konnten wir eine weitgehende Automatisierung der Messdatenerfassung und -auswertung erzielen. Wie das Ablaufdiagramm (TODO: add ref) zeigt, bietet das Projekt verschiedene make Targets für die jeweiligen Ausführungsumgebungen. Diese werden in II-C genauer aufgeführt. Jedem dieser Testziele ist gemein, dass die Ausführung vollständig automatisiert auf Basis eines mehrstufigen Konfigurationsprozesses abläuft. In der ersten Stufe können die gewünschten Messzyklen und zu testenden Processorcounts je Datensatz vorgegeben werden. Diese werden schließlich auf jeder der Konfiguration angewandt, welche mittels eines Testauftraggenerators erzeugt werden. Dieser kann mittels YAML-Dateien (TODO: Referenz) konfiguriert werden und erlaubt es, die zu testenden Kombinationen aus Datensatz, MPI Kommunikationsintervall und MPI Schema zu definieren. Der Generator speichert hierzu je Auftrag eine YAML-Datei ab, welche sich über einen Vererbungsprozess einer definierten Standardkonfiguration des Testziels bedient.

Der eigentliche Test iteriert nun entsprechend der gewählten Anzahl Zyklen und Varianten Processorcount über die Testaufträge und arbeitet je einen Auftrag mittels eines Aufrufs von mpirun ab. Jeder dieser Durchläufe schreibt eine Reihe von Mess- und Kenngrößen in eine CSV-Datei in ein automatisch erzeugtes Verzeichnis. Dieses Verzeichnis wird automatisiert vor und nach einem Test mittels rsync auf eine HiDrive Instanz von Strato synchronisiert, sodass jeder Tester zu jeder Zeit alle aktuellen Messdaten zur Verfügung hat. Um eine hohe Reproduzierbarkeit zu gewährleisten, werden die Testdatei zudem mittels Metainformationen wie aktiviertem git Branch, git Commit Hash (mit dirty Erkennung), genutzter Testumgebung und git User-Email des Testers abgespeichert. Hiermit lässt sich stets auf den genutzten Softwarestand schließen, welcher den erzeugten Messdaten zugrundeliegt.

Eine Erkenntnis aus der Anwendung dieses Systems auf die HPC Cluster der TUM ist die Relevanz der Testausführung: werden Zyklen gleicher Testaufträge hintereinander abgearbeitet ergibt sich eine hohe Vergleichbarkeit der Daten durch gleiche Rahmenbedingungen. Liegen hierbei anhalten schwierige Bedingungen, z.B. durch eine hohe Auslastung durch andere Nutzer, vor, führt dies jedoch zu suboptimalen Messdaten für alle Zyklen. Die vorliegenden Daten sind zur Besserung Interpretierbarkeit daher mit der ersten Variante erzeugt worden.

C. Ausführungsumgebungen für Tests

Um plattformspezifische Einflüsse wie Betriebssystem, Hardware oder Netzwerktechnik vermeiden zu können, haben wir die Automatisierung auf eine Reihe von Testumgebungen angewandt. Die Daten von sechs dieser Umgebungen bilden die Grundlage für unsere Analysen und werden zusätzlich in Bezug mit der dieser Ausarbeitung zugrundeliegenden Implementierung der asynchronen Value Iteration mittels OpenMP verglichen. Die Testumgebungen sind folgende:

1) HPC Class A: Bis zu 5 (hpc01 - hpc05) PCs des LDV mit je einer Vierkern-CPU und 8GB RAM
2) HPC Class B: Bis zu 10 (hpc06 - hpc015) PCs des LDV mit je einer Vierkern-CPU und 8GB RAM
3) HPC Class Mixed: Bis zu 15 (hpc01 - hpc015) PCs des LDV mit je einer Vierkern-CPU und 8GB RAM
4) HPC NUC: 2 NUC Computer mit je einer Zweikern-CPU und 32GB RAM
5) HPC RPi: 4 RaspberryPi Single-Board-Computer mit je einer Vierkern-CPU und 8GB RAM
6) Lokal: 1 PC (hpc04) des LDV mit einer Vierkern-CPU und 8GB RAM

In Summe liegen dieser Ausarbeitung schließlich Messdaten entsprechend Abbildung 1 und Abbildung 2 für den Datensatz <small> und Abbildung 3 und Abbildung 4 für den Datensatz <normal> zugrunde.

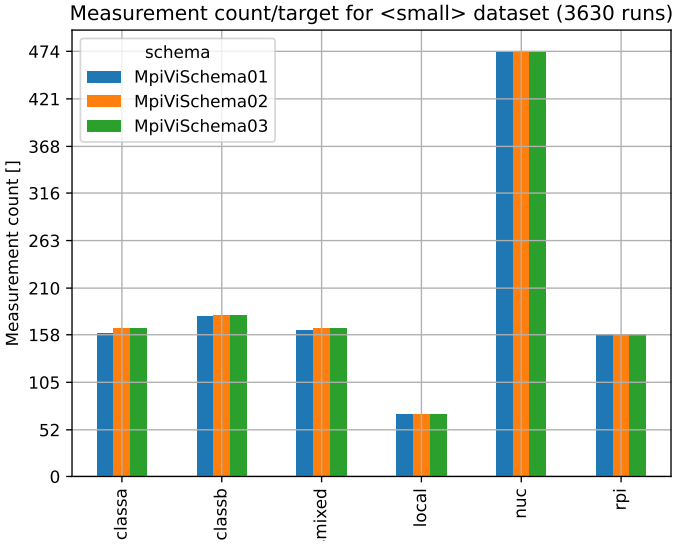


Abbildung 1. Anzahl an Messungen pro Rechenklasse mit kleinem Datensatz

D. Schemata

Den Kern dieser Arbeit bildet die Validierung und das Benchmarking verschiedener Kommunikationsschemata mittels Open MPI. Hierzu wurden drei Schemata implementiert, welche nachfolgend vorgestellt werden sollten. Außerhalb der eigentlichen Schemata ist eine Verteilung der Testkonfiguration und Aggregation von Messgrößen von/zu dem Processor mit Rank0 mittels verschiedener MPI Mechanismen realisiert.

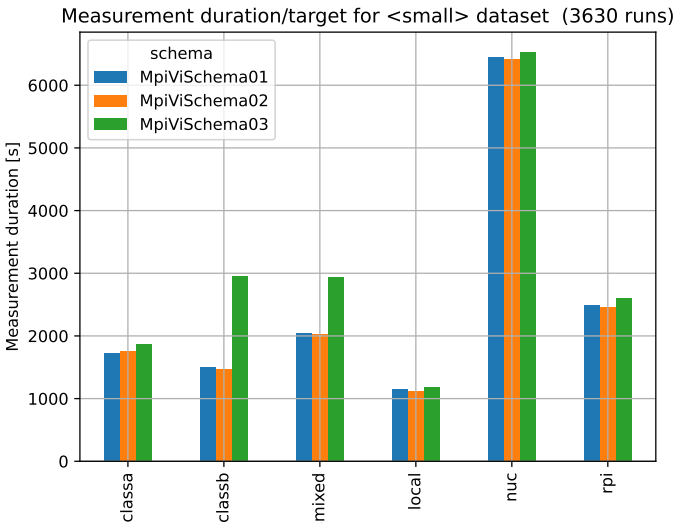


Abbildung 2. Gesamtdauer an Messungen pro Rechenklasse mit kleinem Datensatz

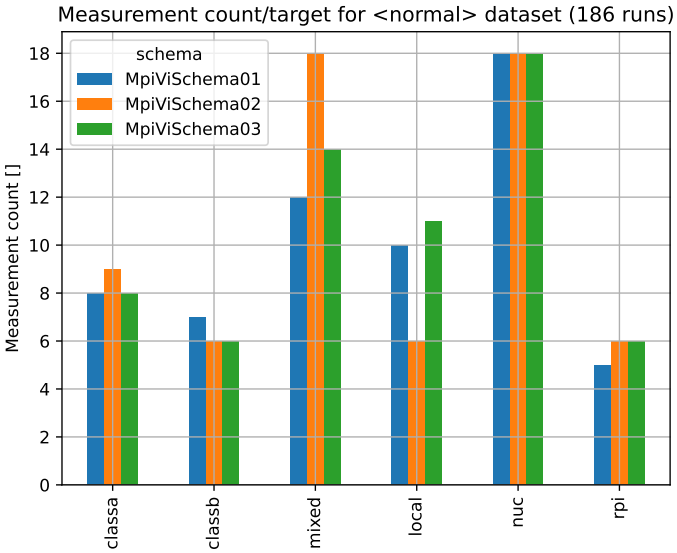


Abbildung 3. Anzahl an Messungen pro Rechenklasse mit normalen Datensatz

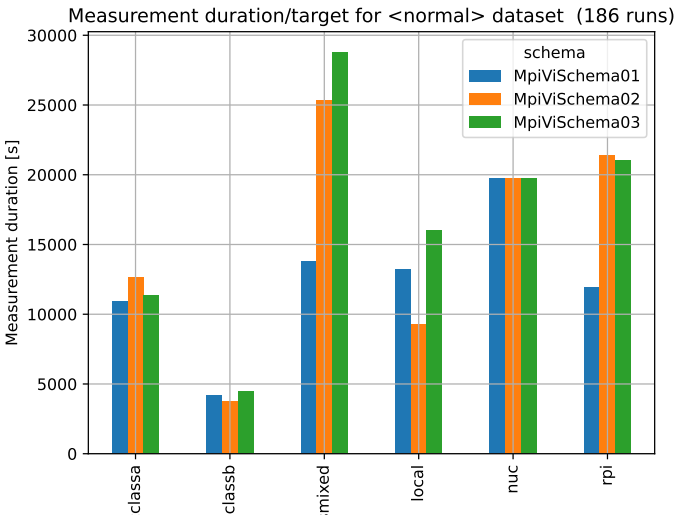


Abbildung 4. Gesamtdauer der Messungen pro Rechenklasse mit normalen Datensatz

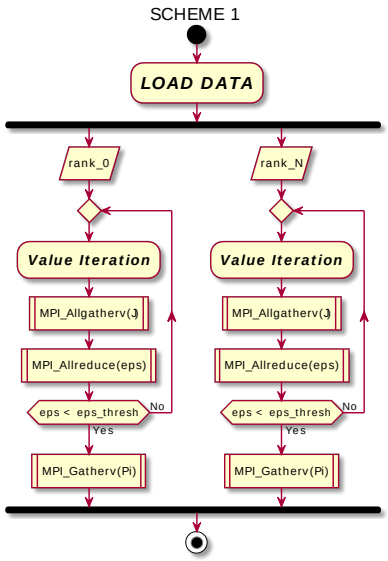


Abbildung 5. MpiViSchema01

1) MpiViSchema01: Das erste Schema Abbildung 5 ist ein synchronisiertes Schema, welches einen Zugriff auf den Datensatz bei jedem Processor voraussetzt. Dieses Schema nutzt die MPI Mechanismen Allgatherv zur Synchronisation des aktuellen Kostenvektors, Allreduce zur Bestimmung eines globalen Konvergenzkriteriums eps und Gatherv zur Bereitstellung des globalen Aktionsvektors.

2) MpiViSchema02: Das Schema Nummer zwei Abbildung 6 ist ebenfalls ein synchronisiertes Schema, welches jedoch als wesentlichen Unterschied eine Verteilung des Datensatzes ausgehend vom Processor mit Rank0 umsetzt. Hierbei müssen andere Processors mitunter keinen Zugriff auf die Datensätze haben und können somit mit weniger Arbeits- und Festspeicher arbeiten. Dieser Vorgang wird mittels des MPI Mechanismus Scatterv umgesetzt und implementiert eine aufwändige Spaltung des Datensatz in abgeschlossene Blöcke, wovon jedem Processor nur einer vorliegt. Die eigentliche Ablauf der Value Iteration erfolgt vergleichbar zu Schema01.

3) MpiViSchema03: Das dritte und letzte Schema Abbildung 7 dieser Ausarbeitung ist ebenso ein synchrones Schema, nutzt statt optimierter Routinen für den großflächige Datenaustausch jedoch die trivialen MPI Methoden wie Sendv und Recv zum Austausch des Kostenvektors J und des Konvergenzkriteriums eps.

III. Analyse & Diskussion

Ziel dieses Kapitels ist es Parameter die Einfluss auf die Berechnung nehmen hervorzuheben und die drei oben erwähnten implementierten Schemen zu analysieren. Dabei soll der Fokus vorallem auf der Rechenzeit, den Speicherbedarfs und den Rechenfehler liegen.
Um die Schemta zu Vergleichen wurden Testläufe mit unterschiedlichen Parametern gemessen. Diese Ergebnisse werden in Unterkapitel A erörtert. Um erworbene Erkenntnisse auf anderen Systemen zu verifizieren wurden Messungen auf

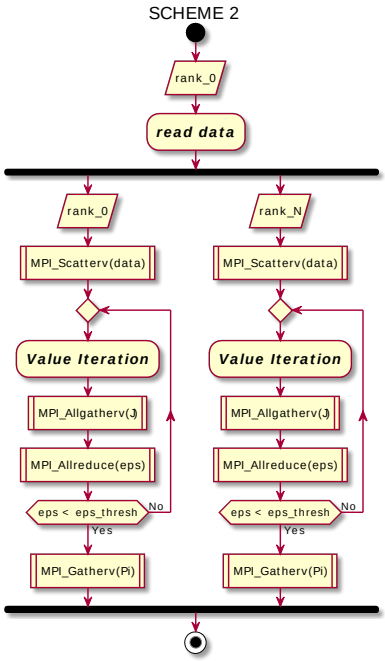


Abbildung 6. MpiViSchema02

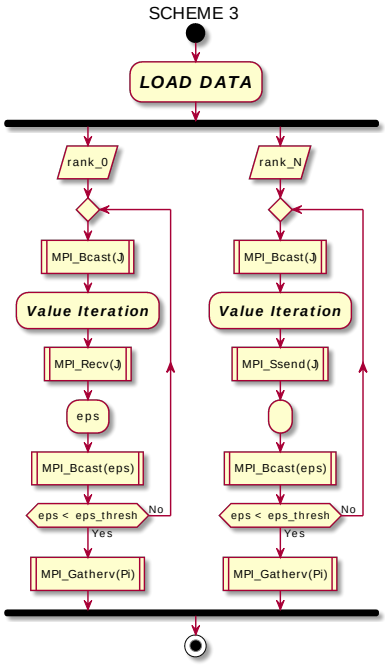


Abbildung 7. MpiViSchema03

verschiedenen Klassen an Recheneinheiten ausgeführt. Dies wird in Unterkapitel B beschrieben. Zu den verwendeten Klassen gehören: HPC Klasse A (HPC 1 - HPC 5), HPC Klasse B (HPC 6 - HPC 15), eine gemischte HPC Klasse (HPC 1 - HPC15) und aus privat stammendem Besitz Respberry Pi Klasse, NUC Rechnerklasse und eine lokale Rechnerklasse.
Da es teilweise auf den Messgeräten zu einer ungleichmäßigen Auslastung kam und damit Datenausreißer generiert wurden, wurden pro Messzyklen mehrere Messungen durchgeführt. Die Anzahl und Messzeiten pro Gerät und Schmema können der Abbildung (1 und 3) entnommen werden. Auf allen Messgeräten wurden Messungen mit je dem klein und normal großen Datensatz vorgenommen. Die ins diesem Kapitel angesprochenen Grafiken und weitere Grafiken sind der Übersicht halber im Anhang abgebildet.

A. *Vergleich der Schemata*

Bei den Messdaten die über die Anzahl an Recheneinheiten und dem Kommunikationsintervall variieren, kann gesehen werden, dass es zwischen den einzelnen Schemen, in Bezug auf Rechenzeit und Konvergenzschritten, zu keinen großen Unterschieden kommt. Dies kann den Messungen auf den Nuc Rechnern aus der Grafik (12a) und der Grafik (14a) besonders gut entnommen werden. Dennoch können mit steigender Anzahl der Recheneinheiten etwas schnellere Ergebenisse erziehlt werden, siehe Grafik (11b). Allerdings kann der Gewinn an Rechenzeit durch die Parallelisierung von Rechenschritten bei einer zu großen Anzahl an Recheneinheiten, durch den den großen Kommunikationsaufwand, schnell wieder zunichte gemacht werden, wie in Abbilung (11c) gesehen werden kann. Die Anzahl der Recheneinheiten hat außerdem eine Auswirkung auf die Anzahl der Iterationsschritte. So steigt mit der Anzahl der Recheneinheiten auch die Anzahl der benötigten Iterationsschritte. Einen großen Einfluss auf die Rechenzeit hat das Kommunikationsintervall, siehe Grafik (12d). So kann beobachtet werden, dass ganz am Anfang die Rechenzeit mit zunehmendem Kommunikationsintervall verkürzt werden kann. Doch tritt schon früh nach einer weitere Erhöhung des Kommunikationsintervalls eine Zunahme der Rechenzeit ein. Im von uns gewählten Kommunikationsintervall ist gegen Ende hin eine lineare Zunahme der Rechenzeit zu sehen, Grafik (11f). Diese Zunahme der Rechenzeit resultiert vor allem aus einer höheren Anzahl an benötigten Iterationsschritten bis zur Konvergenz, siehe Grafik (14d). Es wird außerdem aus der Grafik (14d) sichtbar, dass mit einem höherem Kommunikationsintervall eine höhere Varianz bei den Iterationsschritten entsteht. Diese entstehende Varianz ist bei allen gemessenen Schemen gleich ausgeprägt.
Auch bei der Frage des Speicherbedarfs können einige Erkenntnisse gewonnen werden. Generell ist zu sehen, dass Schema 1 und Schema 3 beim Speicherbedarf nahe beieinander liegen. Schema 2 benötigt auf der Recheneinheit mit dem Rang 0 einen deutlich höheren Speicherbedarf als die anderen beiden Schemata. Wenn man jedoch den gesamten Speicher für die Recheneinheiten über die Anzahl von Recheneinheiten anschaut, wie in Grafik (12l), so sieht man dass mit höherer Anzahl an Recheneinheiten der Speicherbedarf steigt. Bei Schema 2 jedoch nicht so stark wie bei den anderen Schemata. Daher ist etwa ab 4 Recheneinheiten besser das Speicherärmer Schema 2 zu verwenden. Das könnte mit dem Schemaaufbau erklärt werden, da hier nur ein Rang alle Daten einliest und erst danach auf die anderen Rechner weiterveteilt.
Bei der Analyse des Rechenfehlers ist es schwieriger anhand der gewonnenen Messdaten eine Aussage zu treffen, da die Messergebnisse je nach Rechnerklasse variieren können. Jedoch lässt sich sagen, dass der Mittelwert bei gleicher

Parameterwahl und gleicher Rechnerklasse zwischen den Schemen wenig variiert. Dies gilt sowohl für die l2, die Maxnorm und die mittlere quadratische Abweichung. Außerdem bleibt der Fehler je nach Recheneinheit mit varrierender Rechenanzahl und Kommunikationsintervall gleich, siehe Grafik (14g) oder Grafik (14j).

B. *Vergleich der Ausführungsumgebungen*

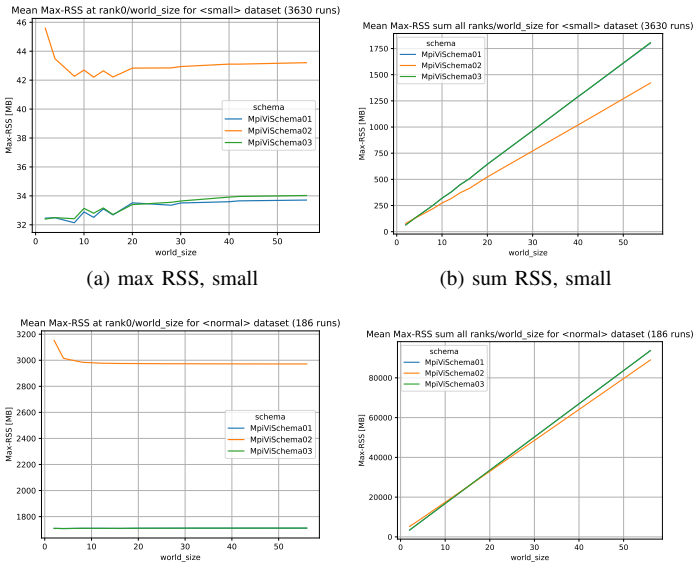
Beim Vergleich der verschiedenen Ausführungsrechenklassen fällt vorallem auf, dass die Rechenzeit auf den Nuc, Lokalen und Raspberry PI Rechnern zwischen den implementierten Schemen weniger variiert. Da die Auslastung auf den HPC Rechnern, je nach Anzahl der Benutzer stark variiert, wird hier auch eine Varianz in den Rechenzeiten sichtbar. Da die Rechnergruppen jedoch unterschiedliche Rechenleistungen aufweisen, kann man keinen direkten Vergleich der Rechenzeit vornehmen. Dennoch können bei der Analyse der Rechenzeit auf den verschiedenen Messgerätklassen, Eigenschaften der verschiedenen Schemata aufgezeigt werden. So sieht man dass der Mittelwert der Rechenzeit bei größeren Kommunikationsintervallen in der Mixed Klasse größer ist als in Klasse B. Die Mixed Rechnerklasse HPC Rechner beinhaltet Rechner aus Klasse A und Klasse B. Dabei weist die Rechnerklasse A eine leicht schlechtere Rechenleistung auf, wie der Vergleich der mittleren Laufzeiten von Klasse A und Klasse B sich zeigt. Da nun in den implementierten Schemen bei der Kommunikation auf das langsamste Glied gewartet werden muss, kann die leicht homogen perfomantere Rechnerklasse schneller zu einem Ergebniss kommen.
Auch bei der Betrachtung des Rechenfehlers gab es Unterschiede zwischen den Rechnerklassen. So die wird Berechnungen auf Rechnerklasse A mit einem größer Fehler ausgeführt als auf Rechnerklasse B.
Beim Vergleich der unterschiedlichen Ausführungsergebnissen konnte jedoch meistens die Erkenntnisse aus dem Unterkapitel A auf allen Rechnerklassen bestätigt werden.

IV. Thesen

Der folgende Abschnitt behandelt Thesen bezüglich der Zusammenhänge zwischen Messgrößen und Parametern. Die Thesen werden anhand der Messergebnisse, der zugrunde liegenden Schema-Architektur und Hardware erörtert.

A. *Es besteht eine Korrelation von RAM mit world_size*

Wie zu erwarten steigt der summierte RAM-Bedarf über alle Processors mit steigender world_size (Fig 8b und Fig 8d sowie Fig 11j-l, Fig 12j-l, Fig 15j-l und Fig 16j-l). Insbesondere bei Schema 1 und 3 liegt jedem Processor die gesamte Datenmenge an Parametern und P-Matrix im Arbeitsspeicher vor. Schema 2 teilt die P-Matrix in Blöcke auf und scattert diese an alle Processors. Diese Aufteilung und dadurch, dass rank_0 auch an sich selbst scattert führt dazu, dass rank_0 von Schema 2 einen höheren RAM Bedarf hat als bei Schema 1 und 3. Weiterhin kann den Messungen entnommen werden, dass ab einer world_size von 4 beim kleinen Datensatz und 16 beim normalen Datensatz der gesamt benötigte RAM Bedarf von Schema 2 niedriger als bei den anderen beiden Schemas ist und darüber hinaus langsamer ansteigt. Das liegt daran, dass jeder rank nur einen Bruchteil entsprechend der world_size der Daten erhält und somit jede Vergrößerung der world_size einen niedrigeren durchschnittlichen RAM Bedarf ergibt. In Fig 8a und Fig 8c sowie Fig 11g-l, Fig 12g-l, Fig 15g-l und Fig 16g-l ist der maximal benötigte RAM-Bedarf von rank_0 in jedem Schema dargestellt. Der Bedarf bleibt über alle world_sizes konstant, da jeder rank_0 unabhängig von Schema und world_size die gesamte Datenmenge im Arbeitsspeicher vorliegen hat.



(a) max RSS, small (b) sum RSS, small

(c) max RSS, normal (d) sum RSS, normal

Abbildung 8. Verlauf des RSS-Bedarfs

B. *Es besteht eine Korrelation runtime mit com_interval*

Das com_interval ist der Parameter, der angibt wie oft Ranks miteinander kommunizieren. Anhand der Diagramme Fig 11d-f, Fig 12d-f, Fig 15d-f und Fig 16d-f ist eine klare Korrelation zwischen der benötigten runtime zur Konvergenz und com_interval erkennbar. Zur Darstellung eines eindeutigeren Verlaufs sind Messungen mit einer höheren com_interval-Auflösung in Fig 9 und 10 dargestellt. Die runtime ist bei allen drei Schemas sehr ähnlich und die Iterationsanzahl sogar meist identisch, daher überdecken die Messpunkte von Schema 3 zum Großteil die anderen beiden Schemata. Die beiden nebeneinander verlaufenden Kurven resultieren aus den zwei unterschiedlichen world_sizes 2 & 4. In Fig 9 gehört die Kurve mit niedrigerer runtime zu world_size 4 und in Fig 10 gehört der Verlauf mit höherer benötigter Iterationsanzahl zu world_size 4. Eine durch höheres com_interval geringere Häufigkeit der Kommunikation zwischen den Processors führt dazu, dass die Processors mehr Iterationen der Value Iteration durchführen bevor die Ergebnisse untereinander ausgetauscht werden. Im Idealfall würde durch selteneres Austauschen weniger Zeit für eben diese Kommunikation verwandt werden und die runtime dadurch sinken. Tatsächlich führt ein größeres com_interval dazu, dass durch das seltenere Update des J-Vektors die Konvergenz beinträchtigt wird. Das führt zu einer höheren benötigten Iterationsanzahl was schlussendlich wieder zu einer höheren Anzahl an benötigten Kommunikationen und

dadurch zu einer längeren Laufzeit führt. Der ansteigende Bedarf an Iterationen bei steigendem com_interval ist in Fig 10 dargestellt. Für die dargestellten NUC-Messungen haben diese beiden sich gegensätzlichen Effekte in Summe bei com_interval 3 ihr Minimum. Bei den anderen Targets liegt das Minimum ebenfalls in dieser Größenordnung. Ohne explizite Messung mit com_interval 3 kann jedoch kein Schluss daraus gezogen werden ob das Minimum bei com_interval 3 hardware-unabhängig ist.

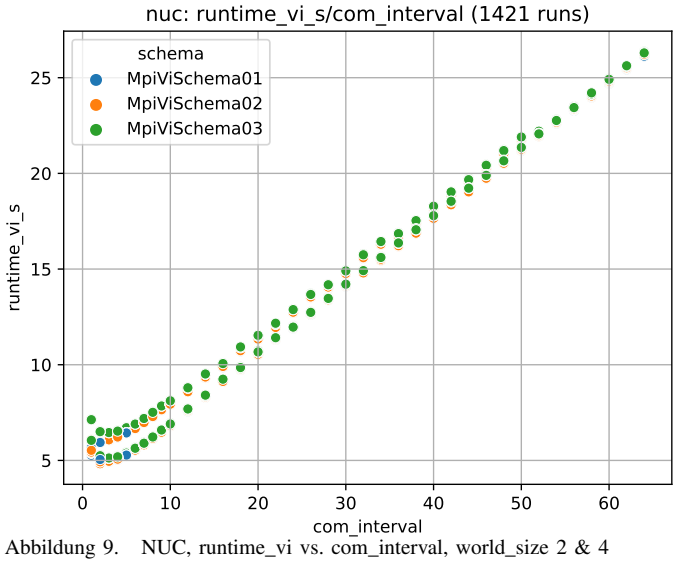


Abbildung 9. NUC, runtime_vi vs. com_interval, world_size 2 & 4

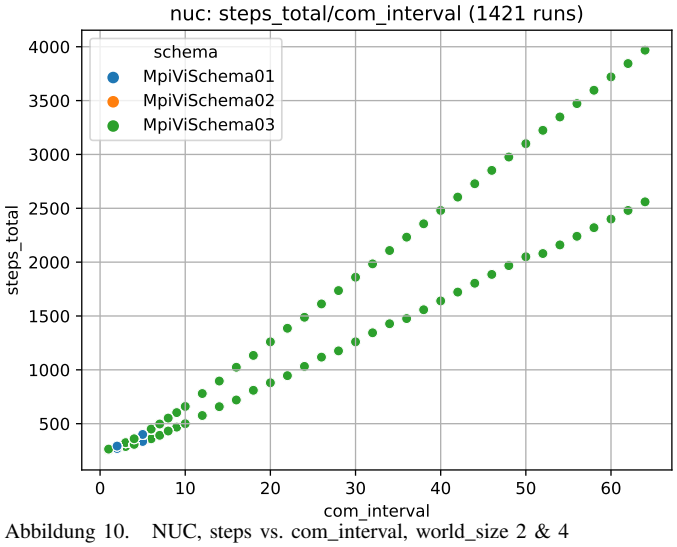


Abbildung 10. NUC, steps vs. com_interval, world_size 2 & 4

C. Es besteht eine inverse Korrelation zwischen world_size und runtime

Eine größere world_size erhöht die Anzahl an Berechnungen, die parallel durchgeführt werden können. Sind die Berechnungen pro Processor komplex/lange genug um den Mehraufwand an inter-Processor Kommunikation zu rechtfertigen so führt dies zu einer veringerten runtime. In der vorliegenden Value-Iteration ist der Effekt nicht besonders stark, da die Berechnungen für die nötige Konvergenz nicht unabhängig voneinander durchgeführt werden können. Ein Austausch der Ergebnisse während des Algorithmus ist für ein richtiges Ergebnis zwingend nötig. Das führt zu einer notwendigen Kommunikation zwischen den Processors, die dem Effekt der Zeitersparnis durch Parallelisierung entgegenwirkt. Anhand der Messergebnisse in Fig 15a-c und Fig 16a-c kann besonders beim normalen Datensatz kein Zusammenhang zwischen der world_size und der runtime festgestellt werden. Die Auswirkung einer größeren world_size fällt von Target zu Target unterschiedlich aus. Bei den isolierten Targets NUC, RPi und Local bleibt die Zeit weitgehend gleich mit einer Tendenz zu geringfügig schnellere Ausführung bei größerer world_size. Aufgrund der Varianz der Messdaten ist es jedoch nicht möglich eine zuverlässige Aussage darüber zu treffen. Beim kleinen Datensatz (siehe Fig 11a-c und Fig 12a-c) ist im Allgemeinen, bis auf world_size 56 bei HPC Class mixed, eine leichte Tendenz zur schnelleren Ausführung bei größerer world_size zu beobachten. Das liegt vermutlich daran, dass die HPCs frei zugänglich sind und die Wahrscheinlichkeit weiterer Nutzer, die die runtime stören, mit steigender world_size und grundsätzlich längerer Berechnungsdauer beim größeren Datensatz steigt. Weiterhin sind die Schemas mit blockierenden MPI-Funktionen implementiert. Das bedeutet, dass in jeder Kommunikations-Iteration auf den langsamsten Processor gewartet wird. So führt einerseits die Heterogenität beim mixed-cluster zu Performance Einbußen, andererseits müssen alle genutzten Processors warten, falls ein Processor durch einen zusätzlichen Nutzer am HPC verlangsamt wird. Für eine eindeutige Aussage der genauen Korrelationen sind Messungen mit garantiert freiem Cluster und kontrollierten Störungen nötig. Im Vergleich zur asynchronen Value Iteration ohne MPI-Kommunikation (siehe Tabelle I) zeigt sich, dass im vorliegenden Fall der Value Iteration beim großen Datensatz MPI zu keiner Verbesserung der Performance führt. Beim kleinen Datensatz konnte bei HPC class A mit world_size 12 und HPC class B mit world_size 30 der openMP-Wert unterschritten werden. Allerdings ist die Varianz der Zeiten so hoch, dass im Mittel kein Performance Gewinn erreicht wird.

Tabelle I. VERGLEICH DER RUNTIME VON OPENMP UND MPI

Datensatz	HPC class A		HPC class B	
	openMP	MPI	openMP	MPI
small	~5s	~4-11s	~3.5s	~2-10s
normal	~490s	~600-800s	~360s	~400-600s

V. ERKENNTNISSE

Im Rahmen dieser Arbeit konnten wir zeigen, dass automatisiertes Ausführen und Testen des vorliegenden Optimierungsproblems auf unterschiedlichen Targets mit unterschiedlichen Parametern realisierbar ist. Unterschiede zwischen verschiedenen MPI-Schemata und deren Einflüsse auf Performance und Messwerte der Value Iteration wurden dargelegt. Weiterhin wurde der Einfluss verschiedener Hardware-Strukturen sowie Parameter auf die Laufzeit und Qualität der Value Iteration dargestellt.

Wir konnten zeigen, dass der benötigte Arbeitsspeicher pro Processor vom verwendeten Schema und dessen Implementierung abhängt, die benötigte Rechenzeit hängt beim vorliegenden Problem in erster Linie von der Größe des

com_intervals und der gewählten Hardware ab. Hierbei zeigt sich, dass langsame Ausführungsumgebungen mit guter Parametrierung ggfs. deutlich bessere Laufzeiten erzielen können, als es schnellere Hardware mit schlechter Parametrierung vermag.

Es konnte kein global gültiger Zusammenhang zwischen runtime und world_size festgestellt werden. Dies und weitere Erkenntnisse aus der Analyse der Laufzeiten führen zu dem Schluss, dass die Lösung des vorliegenden Value Iteration Problems mittels MPI Kommunikation gegenüber einer lokalen Implentierung mittels OpenMP nur unter besonderen Bedingungen einen Mehrwert mit sich bringen kann. Die hier umgesetzten Varianten mittels synchroner MPI Kommunikation bieten diese Bedingungen nicht.

Bezüglich einer Parameterempfehlung lässt sich jedoch anmerken, dass die Laufzeit und die Schritte bis Konvergenz linear vom gewählten Kommunikationsintervall (Intervall > 6) abhängen. Kleinere Kommunikationsintervalle scheinen also wünschenswert.

Zudem variiert die Qualität der Value Iteration in kleinem Rahmen mit der gewählten Ausführungsumgebung und der Anzahl Processors. Je nach Anforderung an das System ist dies zu berücksichtigen.

VI. Beiträge

- Till Hülder: III
- Tobias Klama: IV, V
- Tobias Krug: Zusammenfassung, I, II, V

Literatur

[1] Dowd, K.: *High performance computing*, O'Reilly & Associates, Cambridge Sebastopol, CA, 1998. – ISBN 9781565923126

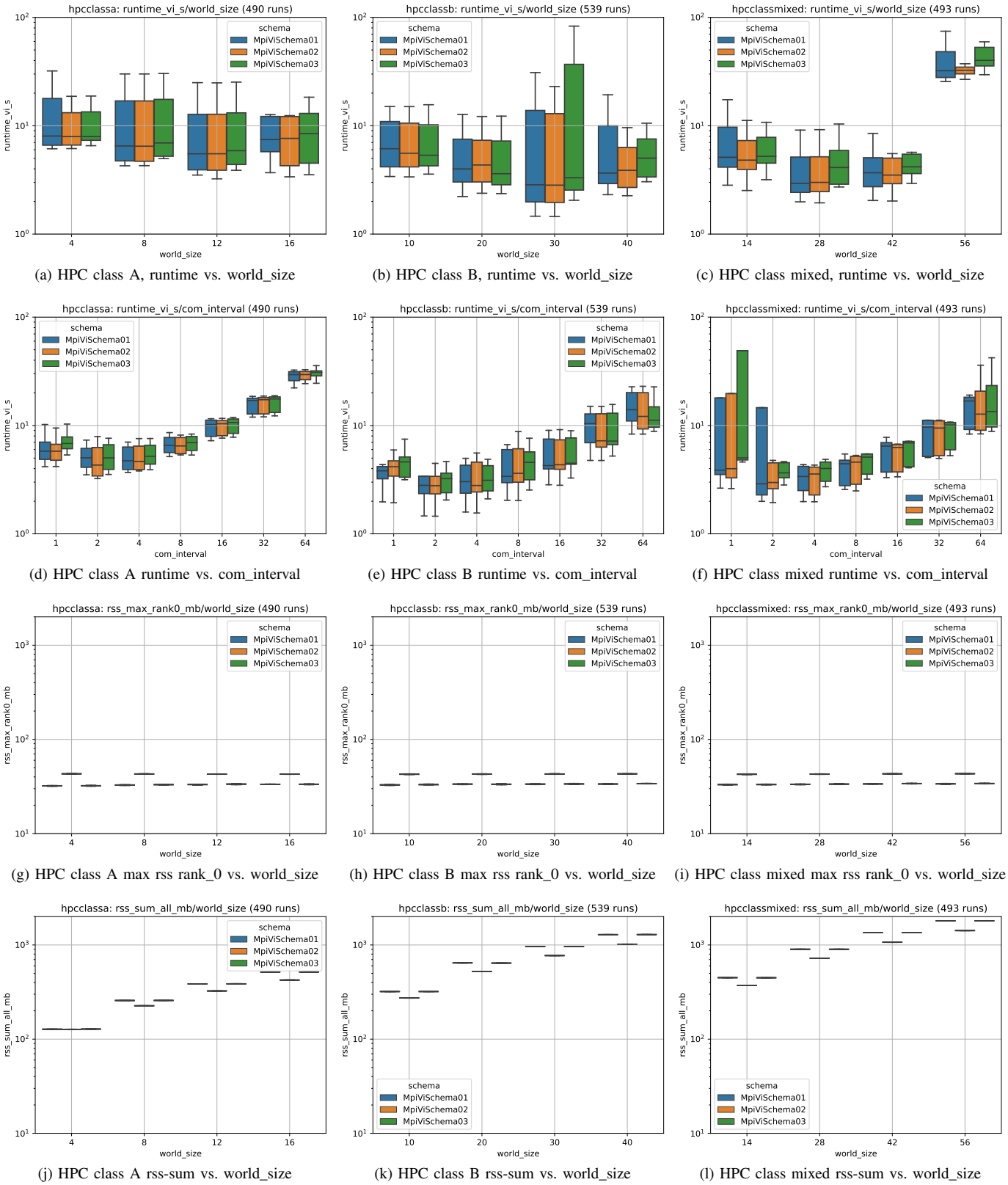
[2] Gamma, E.: *Design patterns : elements of reusable object-oriented software*, Addison-Wesley, Reading, Mass, 1995. – ISBN 9780201633610

[3] Buschmann, F.: *Pattern-oriented software architecture : a system of patterns*, Wiley, Chichester New York, 1996. – ISBN 9780471958697

A. Gprahiken, Benchmark

1) Benchmak Datensatz small: Die nachfolgenden Graphiken zeigen die Ergebnisse der Benchmarks für den Datensatz small.

2) Benchmark Datensatz normal: Die nachfolgenden Graphiken zeigen die Ergebnisse der Benchmarks für den Datensatz normal.



(a) HPC class A, runtime vs. world_size

(b) HPC class B, runtime vs. world_size

(c) HPC class mixed, runtime vs. world_size

(d) HPC class A runtime vs. com_interval

(e) HPC class B runtime vs. com_interval

(f) HPC class mixed runtime vs. com_interval

(g) HPC class A max rss rank_0 vs. world_size

(h) HPC class B max rss rank_0 vs. world_size

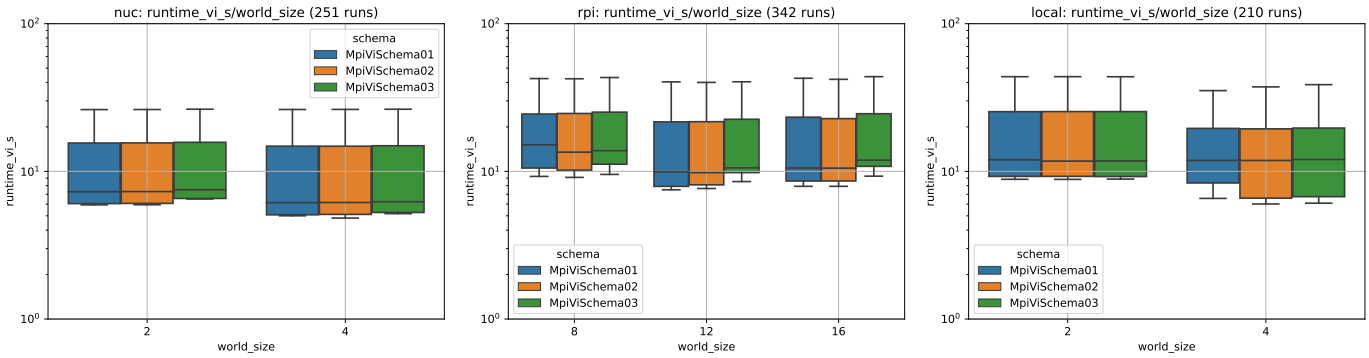
(i) HPC class mixed max rss rank_0 vs. world_size

(j) HPC class A rss-sum vs. world_size

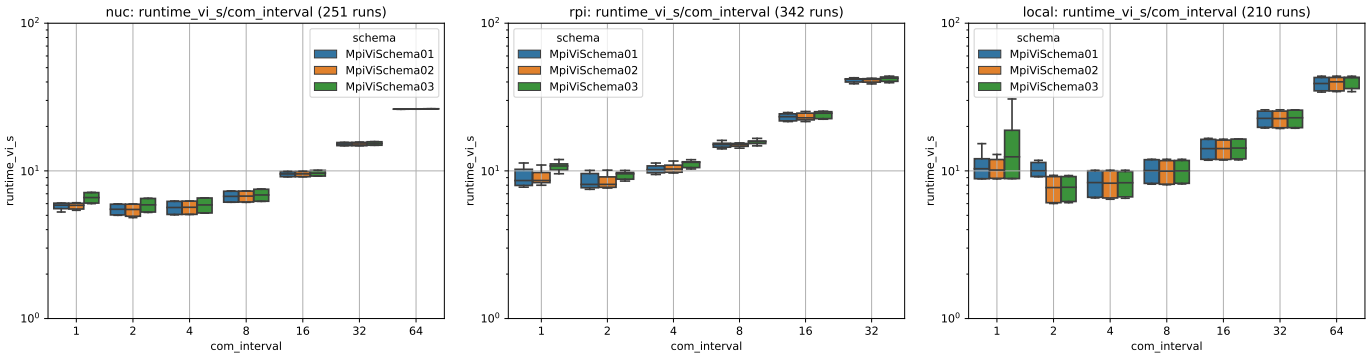
(k) HPC class B rss-sum vs. world_size

(l) HPC class mixed rss-sum vs. world_size

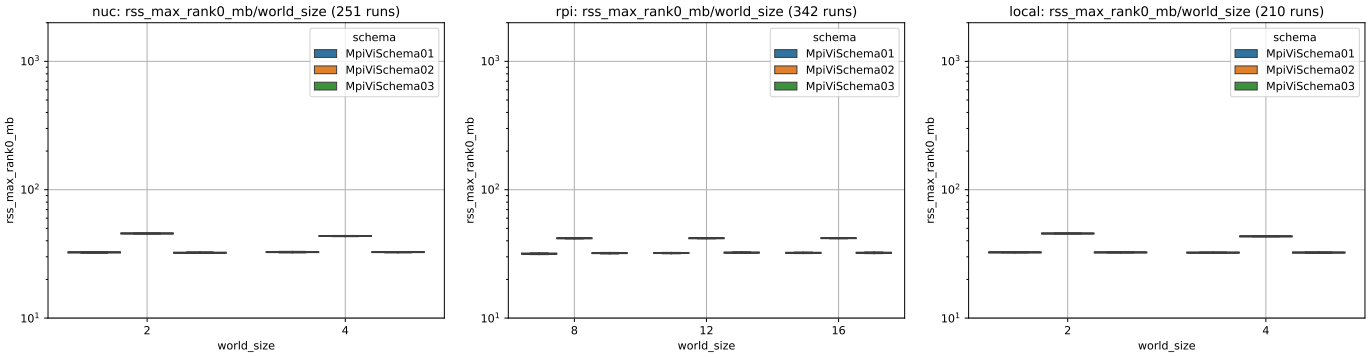
Abbildung 11. Vergleich der Ausführungsumgebungen HPC Cluster mit dem Datensatz <small>, Teil 1/2



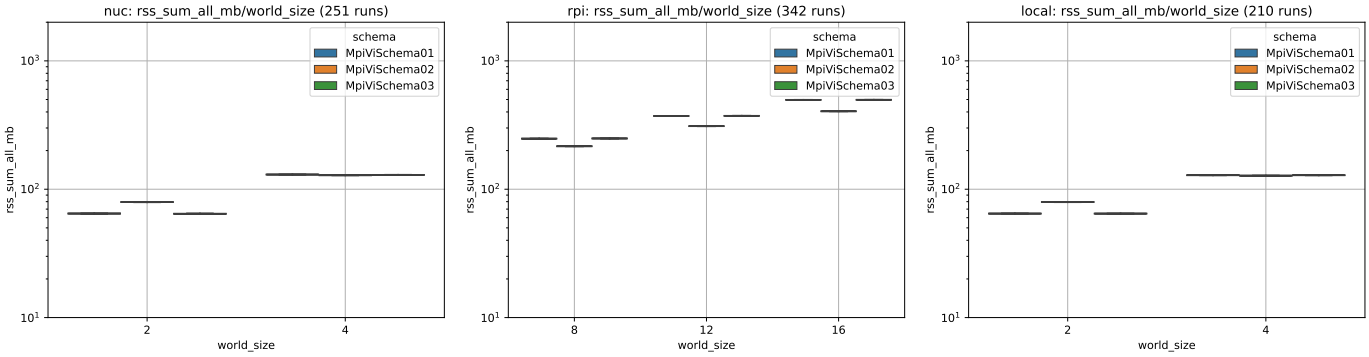
(a) NUC, runtime vs. world_size (b) RPi, runtime vs. world_size (c) Local, runtime vs. world_size



(d) NUC runtime vs. com_interval (e) RPi runtime vs. com_interval (f) Local runtime vs. com_interval

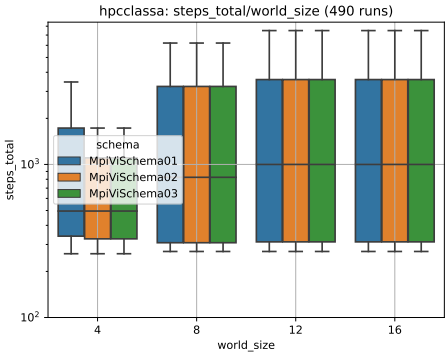


(g) NUC max rss rank_0 vs. world_size (h) RPi max rss rank_0 vs. world_size (i) Local max rss rank_0 vs. world_size

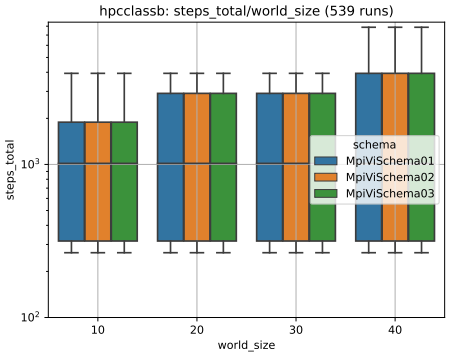


(j) NUC rss-sum vs. world_size (k) RPi rss-sum vs. world_size (l) Local rss-sum vs. world_size

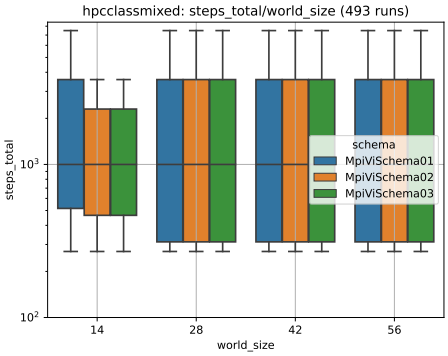
Abbildung 12. Comparison between NUC, RPi and Local with dataset small, part 1/2



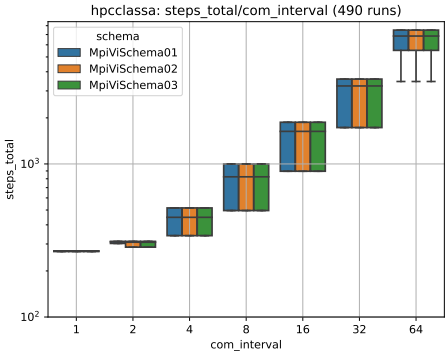
(a) HPC class A, Iterations vs. world_size



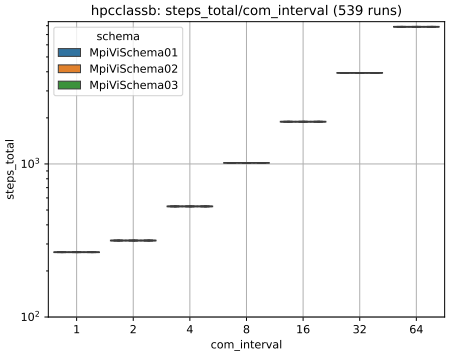
(b) HPC class B, Iterations vs. world_size



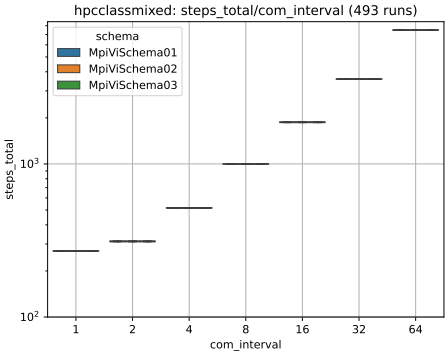
(c) HPC class mixed, Iterations vs. world_size



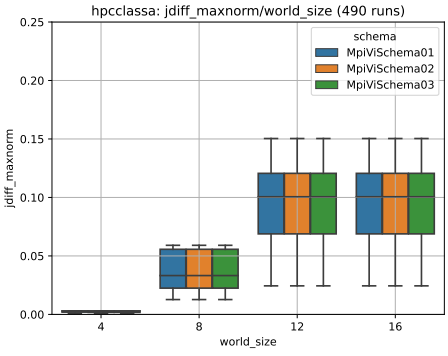
(d) HPC class A Iterations vs. com_interval



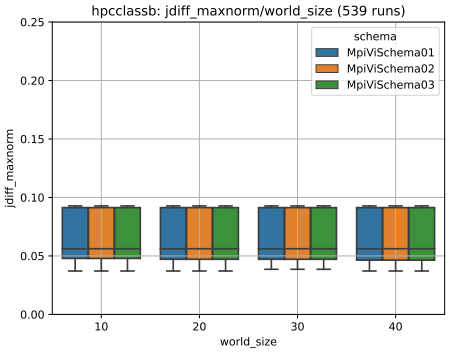
(e) HPC class B Iterations vs. com_interval



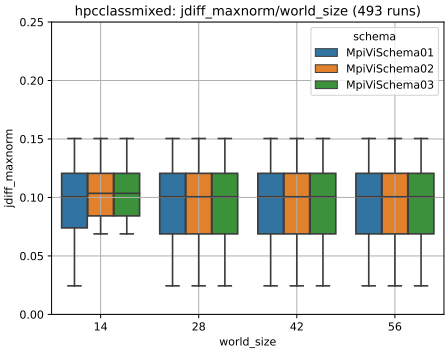
(f) HPC class mixed Iterations vs. com_interval



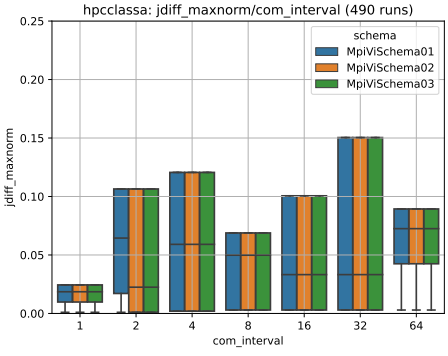
(g) HPC class A J-diff maxnorm vs. world_size



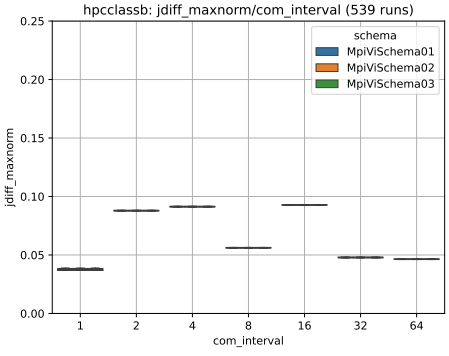
(h) HPC class B J-diff maxnorm vs. world_size



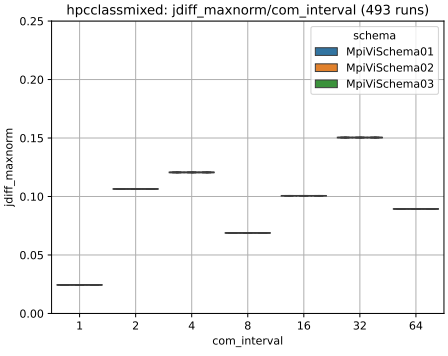
(i) HPC class mixed J-diff maxnorm vs. world_size



(j) HPC class A J-diff maxnorm vs. com_interval

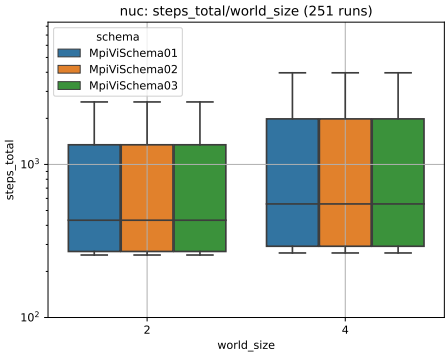


(k) HPC class B J-diff maxnorm vs. com_interval

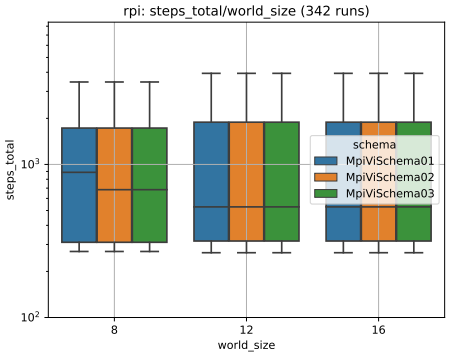


(l) HPC class mixed J-diff maxnorm vs. com_interval

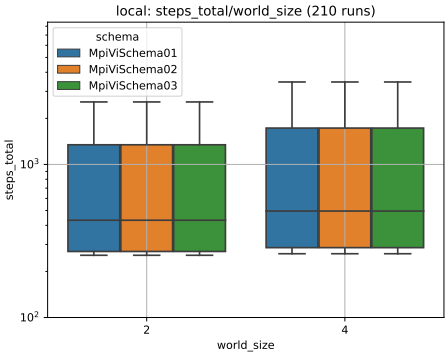
Abbildung 13. Vergleich der Ausführungsumgebungen HPC Cluster mit dem Datensatz <small>, Teil 2/2



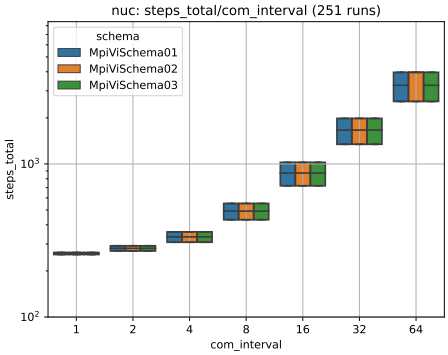
(a) NUC, Iterations vs. world_size



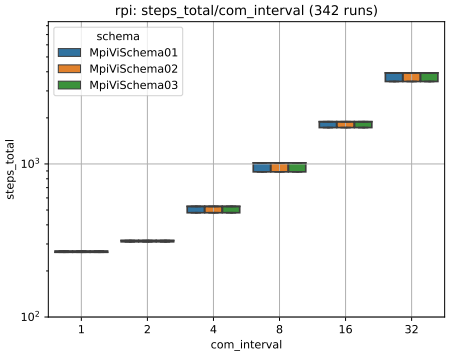
(b) RPi, Iterations vs. world_size



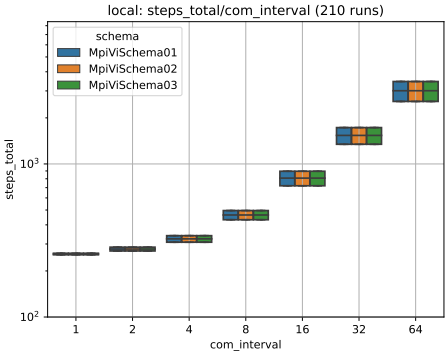
(c) Local, Iterations vs. world_size



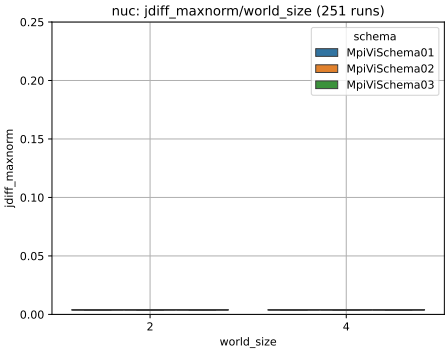
(d) NUC Iterations vs. com_interval



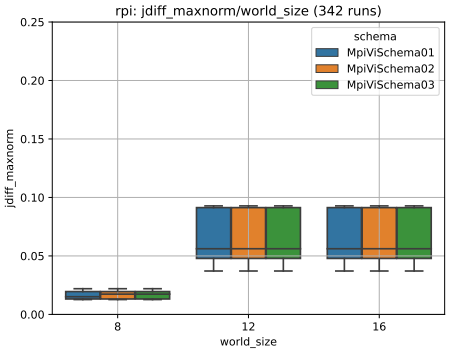
(e) RPi Iterations vs. com_interval



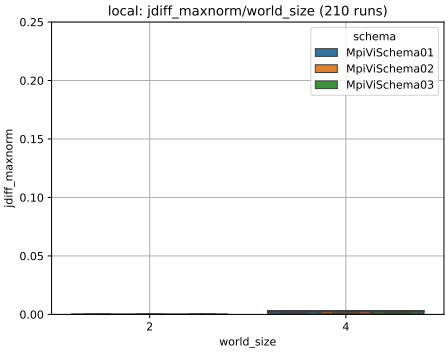
(f) Local Iterations vs. com_interval



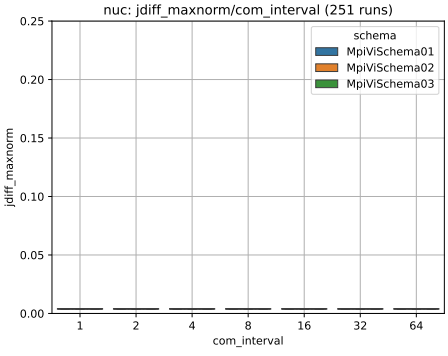
(g) NUC J-diff maxnorm vs. world_size



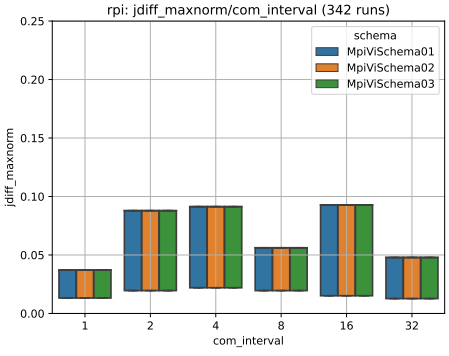
(h) RPi J-diff maxnorm vs. world_size



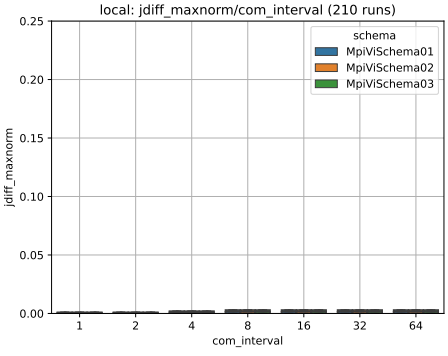
(i) Local J-diff maxnorm vs. world_size



(j) NUC J-diff maxnorm vs. com_interval

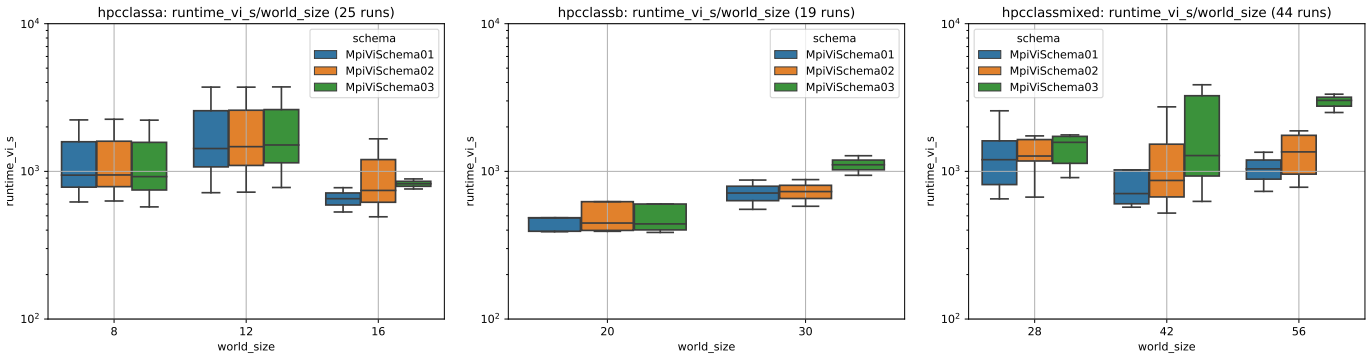


(k) RPi J-diff maxnorm vs. com_interval

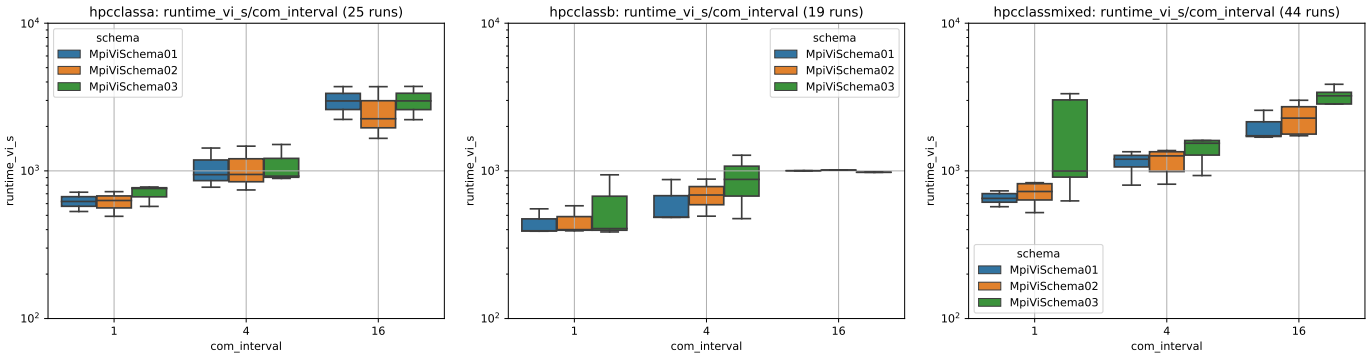


(l) Local J-diff maxnorm vs. com_interval

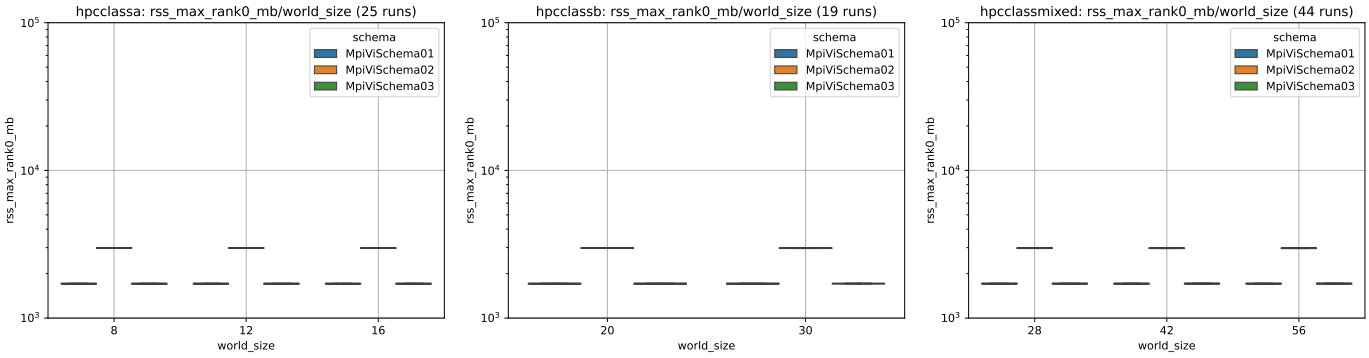
Abbildung 14. Comparison between NUC, RPi and Local with dataset small, part 2/2



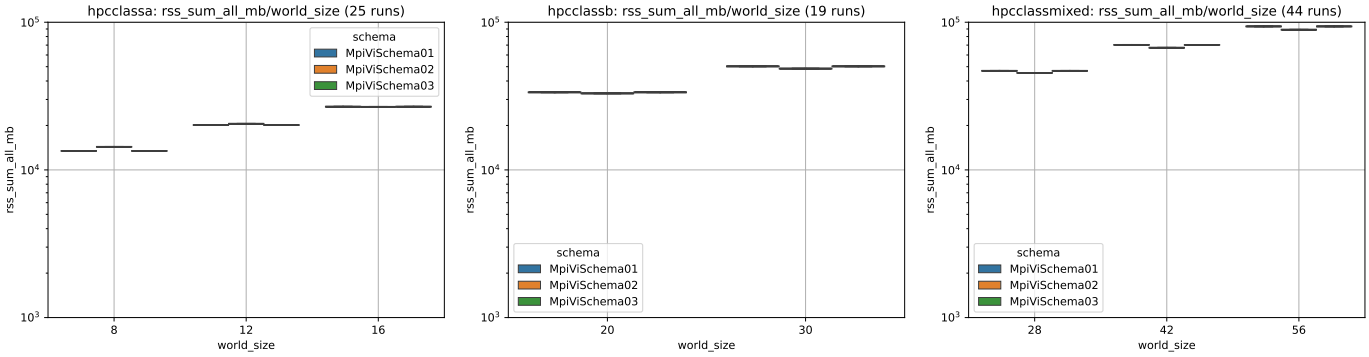
(a) HPC class A, runtime vs. world_size (b) HPC class B, runtime vs. world_size (c) HPC class mixed, runtime vs. world_size



(d) HPC class A runtime vs. com_interval (e) HPC class B runtime vs. com_interval (f) HPC class mixed runtime vs. com_interval

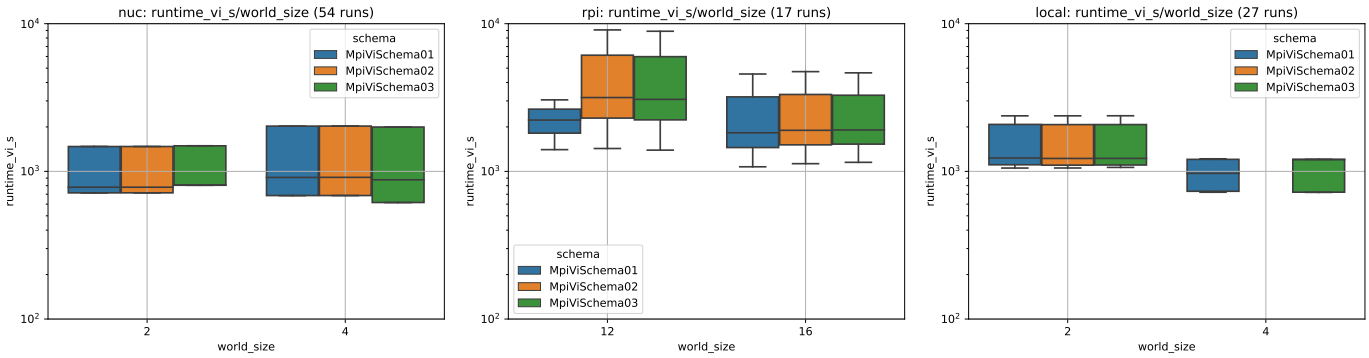


(g) HPC class A max rss rank_0 vs. world_size (h) HPC class B max rss rank_0 vs. world_size (i) HPC class mixed max rss rank_0 vs. world_size

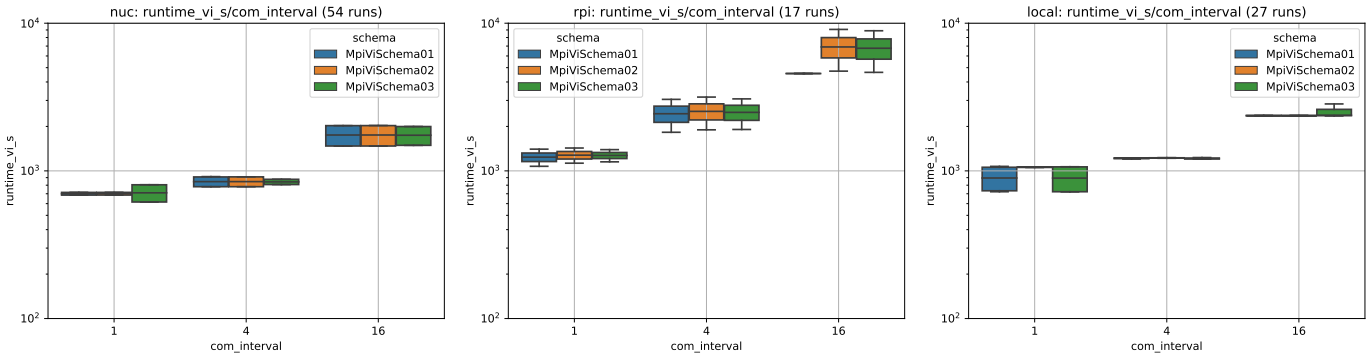


(j) HPC class A rss-sum vs. world_size (k) HPC class B rss-sum vs. world_size (l) HPC class mixed rss-sum vs. world_size

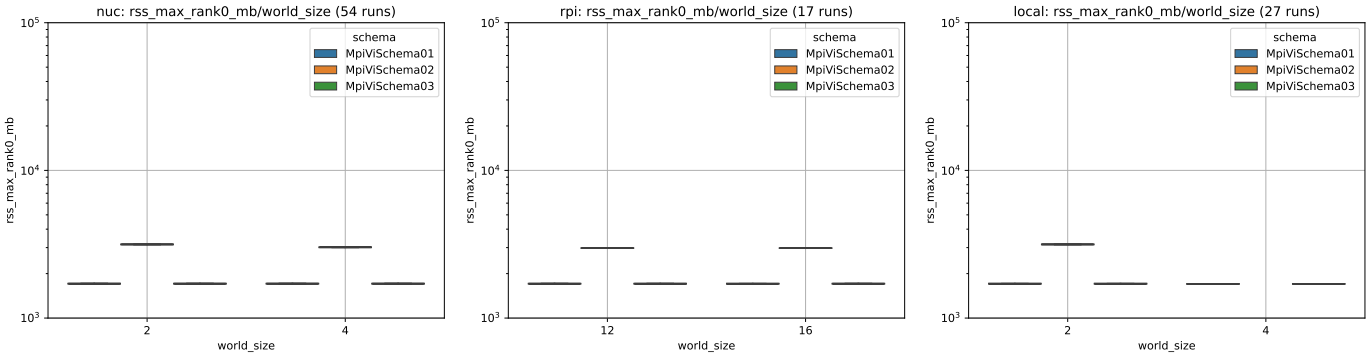
Abbildung 15. Vergleich der Ausführungsumgebungen HPC Cluster mit dem Datensatz <normal>, Teil 1/2



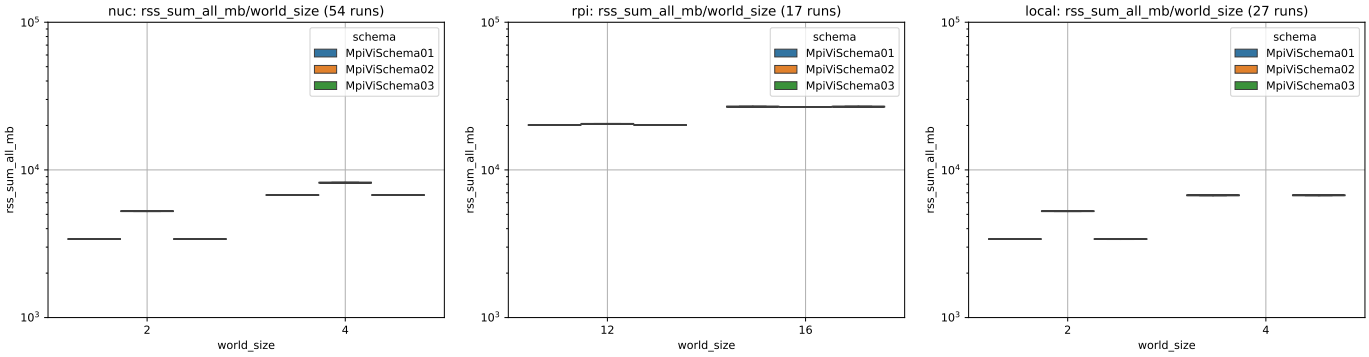
(a) NUC, runtime vs. world_size (b) RPi, runtime vs. world_size (c) Local, runtime vs. world_size



(d) NUC runtime vs. com_interval (e) RPi runtime vs. com_interval (f) Local runtime vs. com_interval

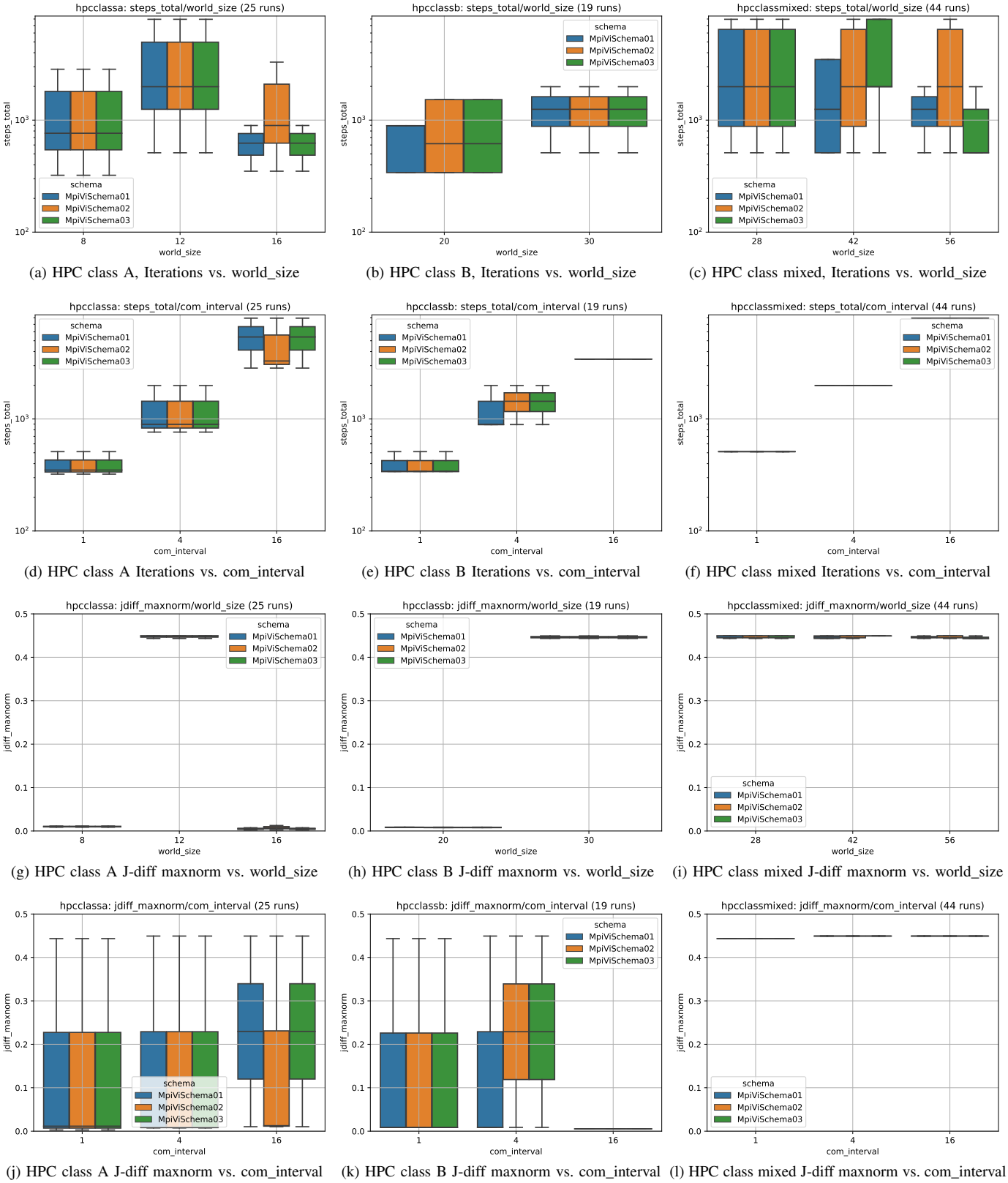


(g) NUC max rss rank_0 vs. world_size (h) RPi max rss rank_0 vs. world_size (i) Local max rss rank_0 vs. world_size



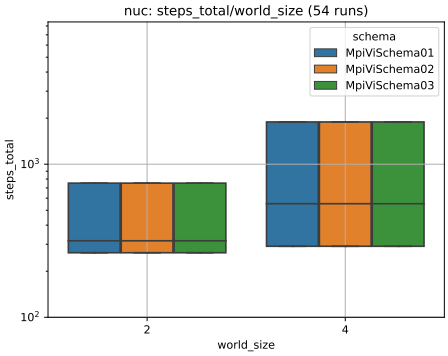
(j) NUC rss-sum vs. world_size (k) RPi rss-sum vs. world_size (l) Local rss-sum vs. world_size

Abbildung 16. Vergleich der Ausführungsumgebungen NUC, RPi und HPC lokal mit dem Datensatz <normal>, Teil 1/2

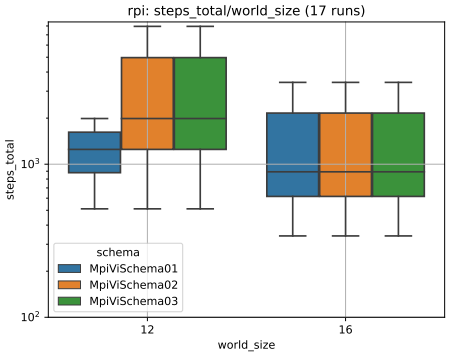


(a) HPC class A, Iterations vs. world_size
(b) HPC class B, Iterations vs. world_size
(c) HPC class mixed, Iterations vs. world_size
(d) HPC class A Iterations vs. com_interval
(e) HPC class B Iterations vs. com_interval
(f) HPC class mixed Iterations vs. com_interval
(g) HPC class A J-diff maxnorm vs. world_size
(h) HPC class B J-diff maxnorm vs. world_size
(i) HPC class mixed J-diff maxnorm vs. world_size
(j) HPC class A J-diff maxnorm vs. com_interval
(k) HPC class B J-diff maxnorm vs. com_interval
(l) HPC class mixed J-diff maxnorm vs. com_interval

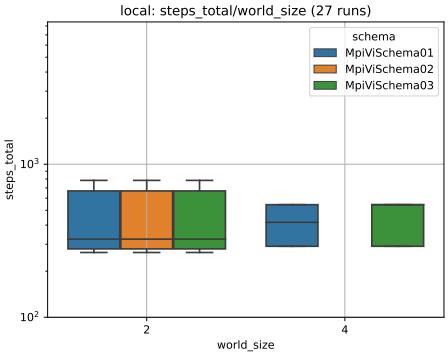
Abbildung 17. Vergleich der Ausführungsumgebungen HPC Cluster mit dem Datensatz <normal>, Teil 2/2



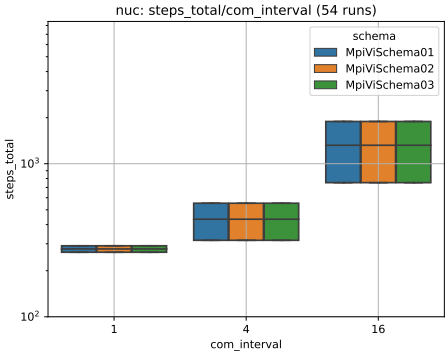
(a) NUC, Iterations vs. world_size



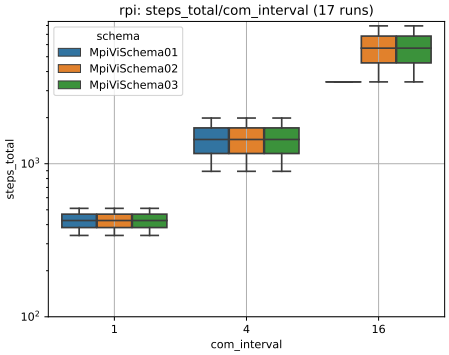
(b) RPi, Iterations vs. world_size



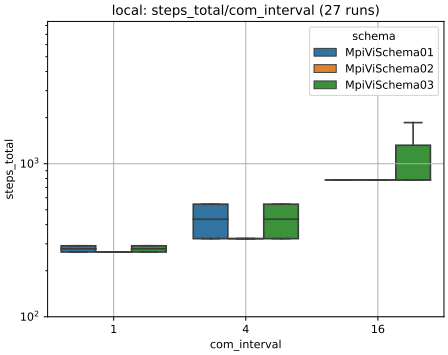
(c) Local, Iterations vs. world_size



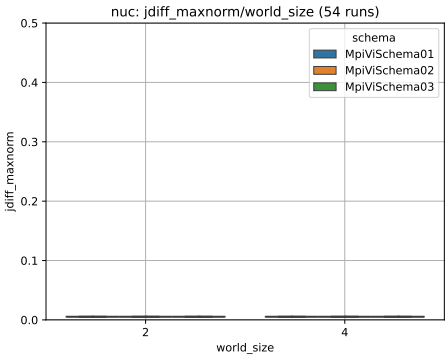
(d) NUC Iterations vs. com_interval



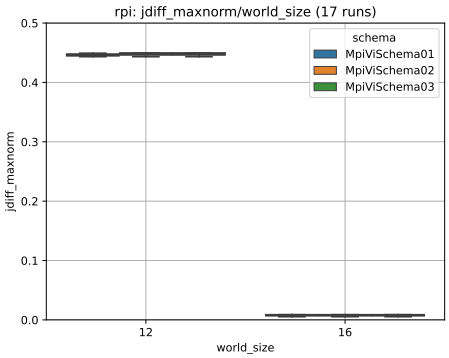
(e) RPi Iterations vs. com_interval



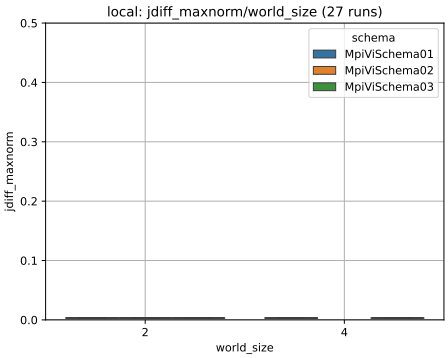
(f) Local Iterations vs. com_interval



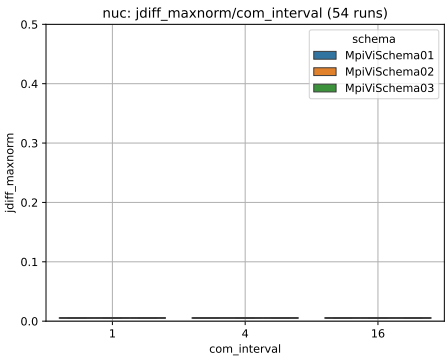
(g) NUC J-diff maxnorm vs. world_size



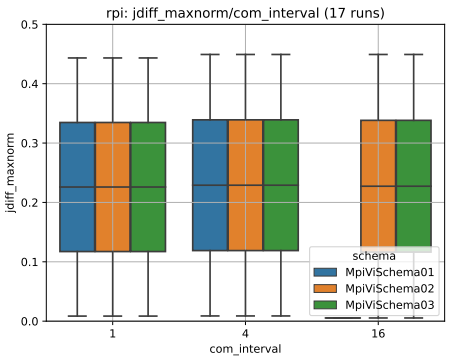
(h) RPi J-diff maxnorm vs. world_size



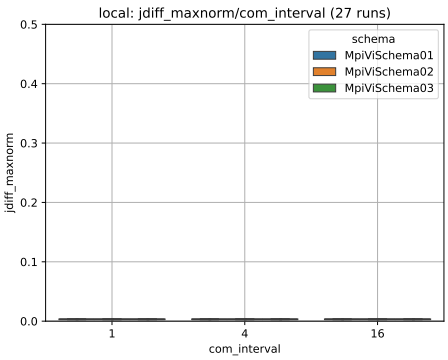
(i) Local J-diff maxnorm vs. world_size



(j) NUC J-diff maxnorm vs. com_interval



(k) RPi J-diff maxnorm vs. com_interval



(l) Local J-diff maxnorm vs. com_interval

Abbildung 18. Vergleich der Ausführungsumgebungen NUC, RPi und HPC lokal mit dem Datensatz <normal>, Teil 2/2