Experimenteller Vergleich paralleler Programmiersprachen mit partitioniertem globalen Speicher anhand ausgewählter Beispielprogramme

Masterarbeit im Fachgebiet Elektrotechnik/Informatik
der Universität Kassel

vorgelegt von

Nikolas Luke
geb. am 20. März 1981 in Uslar

Eingereicht am 27. August 2012

Gutachter:
Prof. Dr. Claudia Fohry
Prof. Dr. Gerd Stumme

angefertigt beim
Fachbereich Programmiersprachen/-methodik
Wilhelmshöher Allee 73
34125 Kassel

Inhaltsverzeichnis

Einleitung

Seit die klassische Erhöhung der Taktfrequenz zur Leistungssteigerung von CPUs an physikalische Grenzen gestoßen ist [1], sind Mehrkernprozessoren zum Standard für Heimrechner geworden. Dadurch wird die parallele Programmierung wichtiger, da praktisch jedes leistungsorientierte Programm von Parallelisierung profitieren kann. Das Entwickeln von Programmen, welche von mehreren vernetzten Computern abgearbeitet werden, ist für die Programmierer gewöhnlich sehr aufwendig. In klassischen Programmiersprachen wird die Parallelisierung von mehreren Prozessorkernen mit gemeinsamem Speicher vom verteilten Rechnen traditionell getrennt betrachtet. Beim verteilten Rechnen kommunizieren die einzelnen vernetzten Computer (im Folgenden als „Knoten" bezeichnet, siehe auch Kapitel 1) über Nachrichten miteinander. Klassisch ist hierbei der Einsatz des Programmiersystems MPI (Message Passing Interface), welches Nachrichtenaustausch zwischen mehreren gestarteten Prozessen auf einem oder verschiedenen Knoten ermöglicht. Zur Parallelisierung von Berechnungen innerhalb eines Knotens wird statt MPI oft ein Programmiersystem wie OpenMP (Open Multi-Processing) oder pthreads (POSIX-Threads) eingesetzt, welche die Parallelisierung mittels Threads ermöglichen. Threads sind Teile eines Prozesses, welche vom Betriebssystem gleichzeitig bearbeitet werden können. Diese Programmiersprachen können mit MPI kombiniert werden, wobei MPI nur für die knotenübergreifende Kommunikation genutzt wird.

Um Programmabschnitte parallel ausführen lassen zu können, müssen diese in parallel ausführbare Teilaufgaben (im Folgenden Tasks genannt) zerlegt werden. Es ist die Aufgabe des Programmierers, solche Tasks mit deren eventuell vorhandenen Abhängigkeiten zu bestimmen und manuell oder automatisiert zum einen auf verschiedene Rechner, aber auch auf mehrere Threads für die CPUs zu verteilen. Dabei muss der Programmierer bei knotenübergreifender Kommunikation (MPI) stets darauf achten, welche Daten sich auf welchem Knoten befinden und welche Daten zwischen den Knoten ausgetauscht werden sollen. Die parallele Programmierung ist im Allgemeinen sehr fehleranfällig und kompliziert, weil bei Mehrfachausführung desselben Programms unterschiedliche Laufzeiten der parallelen Operationen und Kommunikationen auftreten, wobei diese deshalb in unterschiedlicher Reihenfolge beendet sein können. Auch der Quellcode ist mitunter schwer zu lesen, weil das Aufteilen des Problems auf mehrere Knoten, das Senden der Tasks an andere Knoten sowie das Empfangen von Nachrichten und ggf. eine Synchronisierung der Tasks logischerweise nicht an einer einzigen Stelle im Quellcode implementiert wird und die

Parallelisierung von lokalen Threads noch hinzu kommt. Einfachheit und Wartbarkeit von Programmen treten beim performanceorientierten parallelen Programmieren eher in den Hintergrund.

Diese Ausarbeitung beschäftigt sich mit dem Vergleich von PGAS Sprachen. PGAS steht für „partitioned global address space". Es handelt sich dabei um ein Programmiermodell, welches im Hochleistungsrechnen (engl. High Performance Computing, kurz HPC) eingesetzt werden soll. Im Gegensatz zu anderen Programmiersprachen, in denen Parallelität wie oben beschrieben durch MPI und OpenMP erreicht wird, befinden sich PGAS Sprachen auf einem höheren Abstraktionsniveau. Im PGAS Modell wird ein globaler Adressbereich logisch unterteilt, so dass jeder Knoten seinen eigenen Adressbereich erhält. Alle Prozessoren können zwar auf jede Speicherzelle im globalen (also knotenübergreifenden) Adressbereich zugreifen, jedoch wird davon ausgegangen, dass dem Prozessor auf dem ihm zugeteilten Adressbereich eine höhere Zugriffsgeschwindigkeit zur Verfügung steht [2]. Der Programmierer muss auf eine gleichmäßige Verteilung der Daten, aber auch des Rechenaufwands auf alle beteiligten Prozessoren achten, um gute Ergebnisse zu erzielen.

Das PGAS Modell ist in verschiedenen Sprachen implementiert worden. Dazu zählen Unified Parallel C (UPC), Co-array Fortran (CAF) welches in Fortran 2008 enthalten ist, Titanium, Fortress, Chapel und X10.

In dieser Ausarbeitung werden die PGAS Sprachen Chapel und X10 miteinander verglichen. Diese Sprachen wurden ausgewählt, weil sie noch weiterentwickelt werden und mit Cray und IBM große Firmen hinter der Sprache stehen, die die Zukunft der Sprachen sichern. Sie sind deutlich neuer als UPC und CAF und konnten auf die Erfahrungen aufbauen, welche dort bereits im PGAS Bereich gesammelt wurden. Damit präsentieren sie sehr gut den aktuellen Stand des PGAS Modells. Es soll untersucht werden, wie einfach, intuitiv und effizient mit den Sprachen parallele Programme entwickelt werden können. Zusätzlich wird auch eingeschätzt, wie aufwändig der Umstieg von einer klassischen Programmiersprache wie beispielsweise Java ausfällt. Um die zu untersuchenden Sprachen miteinander zu vergleichen, wurden verschiedene Beispielalgorithmen aus dem Bereich der Mathematik gewählt, welche auf unterschiedliche Weise parallelisiert werden können. Es wurde sich für mehrere „Cowichan Problems" entschieden. Dabei handelt es sich um eine Auswahl von Problemen, mit denen die Verwendbarkeit von parallelen Programmiersystemen eingeschätzt werden kann [3]. Benchmarks sind zu diesem Zeitpunkt noch wenig aussagekräftig, da sich Chapel und X10 noch in der Entwicklung befinden. So sind manche Konstrukte noch nicht

implementiert und müssen durch weniger effiziente workarounds ersetzt werden. Dadurch kann manchmal auch keine deutlich kürzere Programmlaufzeit mit mehreren Computern (auch als Speedup bezeichnet - nähere Erklärung im ersten Kapitel) in den aktuellen Versionen der Sprachen beobachtet werden.

In dieser Ausarbeitung werden zunächst in Kapitel 1 die wichtigsten Fachbegriffe erklärt, welche in der parallelen Programmierung wichtig sind. Anschließend werden in Kapitel 1 die verwendeten PGAS Sprachen vorgestellt und deren besondere Eigenschaften erklärt. In den Kapiteln 3 und 4 erfolgen Implementierungen der Beispielalgorithmen „Mandelbrot-Menge" und „Gaußsches Eliminationsverfahren" in den verschiedenen PGAS Sprachen. Dabei untersuchen wir verschiedene Implementierungsvarianten um festzustellen, ob sich ein vorgegebener Algorithmus unter Verwendung der PGAS Idee implementieren lässt und ob sich die Stärken der PGAS Sprachen in der Praxis überhaupt auswirken. Diese beiden Kapitel enden jeweils mit einer Bewertung der Sprachen in Bezug auf die Implementierung der Algorithmen. Nach diesen Praxistests wird in Kapitel 1 ein allgemeiner Vergleich der Sprachen durchgeführt, bei dem auch die Erfahrungen mit dem Umfang und der Qualität der Handbücher und Dokumentationen durchgeführt wird. Im 6. Kapitel wird der Inhalt dieser Masterarbeit noch einmal zusammengefasst, die Ergebnisse der experimentellen Vergleiche beurteilt und ein Fazit gezogen.

1. Grundbegriffe paralleler Programmierung

Im Bereich der parallelen Programmierung haben sich bestimmte Konzepte und Fachbegriffe etabliert. Es folgt eine Auflistung von Begriffen, welche zum Verständnis dieser Ausarbeitung wichtig sind:

Knoten

Als Knoten werden einzelne, untereinander vernetzte Computer bezeichnet, welche gemeinsam als „ein Rechner" angesprochen werden können. Solche Zusammenschlüsse von Knoten werden als Rechnerverbund oder auch Computercluster bezeichnet.

Task

Eine Aufgabe, welche von einem Algorithmus gelöst werden soll, muss für eine parallele Abarbeitung in einzelne Teilaufgaben zerlegt werden. Diese Teilaufgaben werden Tasks genannt. Ein Programm kann auch einen weitgehend sequentiellen Programmablauf haben, wobei nur die zeitaufwändigen Programmteile in Tasks zerlegt werden, um gezielt diese zu parallelisieren.

Prozess

In der Informatik tritt ein laufendes Programm als Prozess in Erscheinung, welcher vom Betriebssystem verwaltet wird. Insbesondere sorgt der Scheduler des Betriebssystems dafür, dass jedem Prozess Rechenzeit der CPUs zur Verfügung gestellt wird. Jeder Prozess hat seinen eigenen Speicher, wobei ein direkter Zugriff auf Speicher eines anderen Prozesses nicht möglich ist.

Thread

Threads erlauben die parallele Abarbeitung verschiedener Teile eines Prozesses, was auf Systemen mit mehreren CPUs Geschwindigkeitsvorteile bringt. Sequentielle Programme kommen aber auch mit einem einzigen Thread aus. Das Umschalten zwischen Threads eines Prozesses durch den Scheduler ist mit weniger Aufwand verbunden, als das Umschalten zwischen Prozessen, weil der Kontext sehr ähnlich ist. Deshalb werden Threads auch „leichtgewichtige Prozesse" genannt. Threads können auf Daten von anderen Threads desselben Prozesses zugreifen, weil sie sich dessen Speicher teilen.

Zeitkritischer Ablauf

Wenn mehrere Threads auf einen gemeinsamen Speicherbereich lesend und schreibend zugreifen, kann das Ergebnis (z.B. ein Rückgabewert) davon abhängen, wie schnell und in welcher Reihenfolge die Threads die Lese- und Schreiboperationen durchführen. Da der Programmierer auf die CPU Zuteilungen durch den Scheduler wenig Einfluss hat, kann es bei solchen „zeitkritischen Abläufen" zu unvorhergesehenen Programmzuständen und fehlerhaften Daten kommen, wobei die Fehlersuche sehr schwierig sein kann.

Deadlock

Eine „Verklemmung" von mehreren Prozessen oder Threads. Beispielsweise können zwei Prozesse gegenseitig auf Ressourcen warten, welche dem jeweils anderem Prozess zugeteilt sind. Bei Threads können Deadlocks gerade in der parallelen Programmierung auftreten, wenn diese beispielsweise auf Daten warten, wobei der Zugriff von anderen Threads mittels eines „Locks" gesperrt ist.

Overhead

Als Overhead wird in der parallelen Programmierung der zusätzliche Aufwand bezeichnet, welcher bei einem sequentiellen Programm nicht auftreten würde. Dazu gehören die Verwaltung und Synchronisation der Threads sowie der Nachrichtenaustausch zwischen mehreren Knoten. Dieser macht sich durch zusätzliche Zeit bemerkbar, die eine sequentielle Abarbeitung des gleichen Programmteils nicht benötigt hätte. Manche Programmteile, welche bereits sequentiell wenig Zeit brauchen, benötigen in einer parallelen Formulierung evtl. sogar länger, weil die Zeit für den Overhead die Zeitersparnis der Parallelisierung überwiegt. Dies kann beispielsweise beim Erstellen von sehr vielen Threads auftreten, die aufwendig verwaltet und synchronisiert werden müssen. Oft ist ein Angleichen der Anzahl der Threads an die Anzahl verfügbarer Prozessoren sinnvoll, so dass der Overhead der Threadverwaltung möglichst klein gehalten wird.

Speedup

Der Speedup ist das Verhältnis zwischen der sequentiellen Laufzeit eines Programms (*T sequentiell*) und der Laufzeit auf Mehrprozessorsystemen (*T parallel*). Er lässt sich als Funktionskurve in Abhängigkeit zur Anzahl verwendeter Prozessoren p darstellen (siehe Abbildung 1 links).

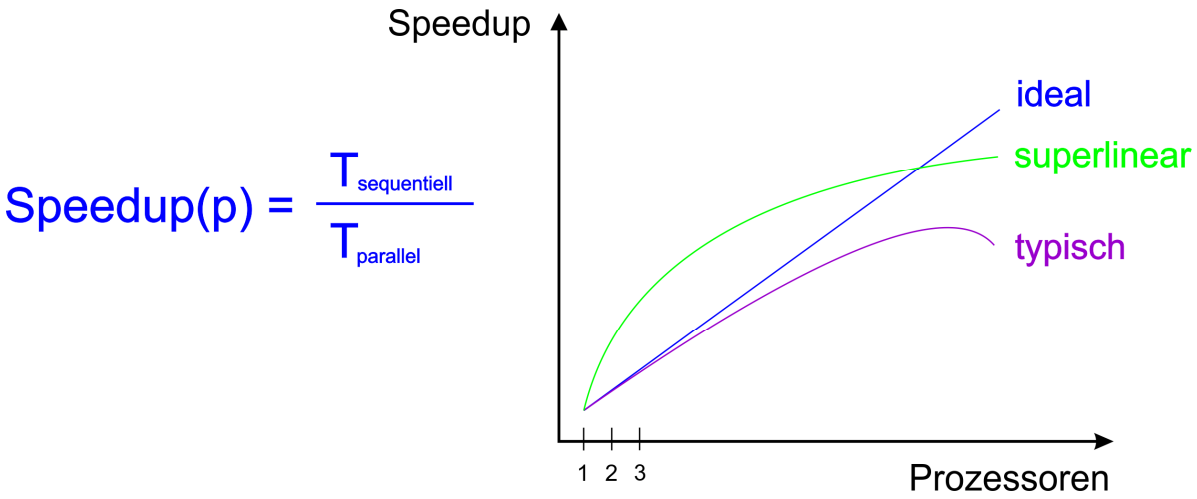


Abbildung 1 links: Speedupfunktion, rechts: Beispiele für Kurvenverläufe

Die Funktionskurve kann verschiedene Verläufe nehmen, denn das Verhältnis der Laufzeiten wird von Faktoren wie Art und Implementierung des Algorithmus und zeitlicher Höhe des Overheads beeinflusst. Letzterer ist je nach Übertragungsmedium mit größerem Zeitverlust verbunden. Außerdem haben Programme gewöhnlich einen sequentiellen Programmteil, welcher aus programmlogischen Gründen nicht parallelisiert werden kann.

Ein mit der Anzahl an Prozessoren linear steigender „idealer Speedup" (Abbildung 1 rechts in blau) wird nur selten erreicht, denn mit steigender Anzahl an CPUs und einer dementsprechend hohen Anzahl an Threads kommt es zu erheblichem Overhead. Ein „typischer Speedup" verläuft beim Erreichen einer bestimmten Prozessorenzahl zunehmend schlechter, weil nun unverhältnismäßig viel Aufwand für die Threadverwaltung betrieben werden muss. Je nach Programmanforderungen kann der Einsatz von mehreren Prozessoren durch den zusammen größeren zur Verfügung stehenden Cache Speicher zu besseren Laufzeiten führen, als es zu erwarten wäre. Solche „superlinearen" Speedups (Abbildung 1 rechts in grün) dürften bei steigender Prozessorenzahl dennoch durch die Nachteile des Overheads wieder negativ beeinflusst werden.

Cache-Effekte und Lokalität

Daten, welche vom Prozessor benötigt werden, liegen idealerweise im schnellen Cache-Speicher. Bei Zugriffen auf Speicheradressen, deren Daten sich nicht im Cache befinden, kommt es zu einem „Cache Miss". Die Daten müssen in diesem Fall aus dem langsameren Arbeitsspeicher in den Cache nachgeladen werden, womit andere Daten aus dem Cache entfernt bzw. überschrieben werden müssen. Programmierer können durch das Verhalten ihres Programms Einfluss darauf nehmen, häufiger „Cache Hits" zu bekommen, so dass auf Daten zugegriffen wird, welche sich wahrscheinlich noch im schnellen Cache Speicher befinden. Dieses für die Cachezugriffe relevante Programmverhalten wird „Lokalität" genannt und stellt einen wichtigen Performancefaktor dar. Muss auf Datenelemente öfters zugegriffen werden, sollten diese Zugriffe zeitlich nicht weit auseinander liegen, um die Wahrscheinlichkeit zu erhöhen, dass diese Elemente noch im Cache liegen. Der Programmierer nutzt dabei „zeitliche Lokalität". Je nach dem, wie die Verwaltung des Caches organisiert ist, können spekulativ weitere Daten, welche sich neben den angeforderten befinden, ebenfalls in den Cache geladen werden. Auf das Verhalten des Caches hat der Programmierer zwar keinen Einfluss, jedoch kann er darauf spekulieren, dass beispielsweise benachbarte Elemente eines Arrays, gerade in derselben Arrayzeile, gemeinsam in den Cache geladen werden. Ein zweidimensionales Array sollte demnach eher zeilenweise statt spaltenweise durchlaufen werden, um „räumliche Lokalität" zu nutzen [4][5].
Beim Zugriff auf Daten, welche sich auf demselben Knoten befinden, kann auch von lokalen Daten gesprochen werden, da kein Nachrichtenaustausch zwischen Knoten notwendig ist.

Mapping-Techniken

Um eine effektive Parallelisierung gewährleisten zu können, sollten möglichst alle Knoten mit ihren CPUs eine entsprechende Anzahl an Tasks zugeteilt bekommen, so dass eine etwa gleiche Laufzeit entsteht. Durch eine solche Lastenbalancierung soll vermieden werden, dass bei einer Synchronisierung, beispielsweise beim Zusammenführen von Teilergebnissen, CPUs oder ganze Knoten nicht mehr mitrechnen können, weil auf einige Tasks gewartet werden muss. Die Verteilung der Tasks auf die Knoten erfordert evtl. eine analoge Verteilung der Daten. Alternativ können auch Daten auf Knoten aufgeteilt werden, womit später erzeugte Tasks gemäß der Datenverteilung von den entsprechenden Knoten bearbeitet werden.

Es existieren bereits bestimmte „Standard Verteilungen". Wir betrachten diese nun anhand einfacher Beispiele mit der Aufteilung von Tasks auf mehrere Knoten. Man kann diese Verteilungen auch für lokale Prozesse und Threads innerhalb eines Knotens definieren, wobei

das Prinzip dieser Standard Verteilungen auch auf die Verteilung von Daten auf verschiedene Knoten anwendbar ist [6].

- **Zyklische Verteilung**

  Bei einer zyklischen Verteilung werden die Tasks abwechselnd auf alle zur Verfügung stehenden Knoten verteilt. Der erste Task befindet sich dabei auf dem ersten Knoten, der zweite Task auf dem zweiten. Hat auch der letzte Knoten einen Task erhalten, wird der nächste Task wieder dem ersten Knoten zugeteilt usw. Diese Verteilung bietet sich an, wenn die Tasks unterschiedlich groß sind und große aufeinander folgende Tasks auf verschiedene Knoten verteilt werden sollen. Durch die starke räumliche Verteilung der Tasks kann Lokalität nur wenig genutzt werden.

- **Blockverteilung**

  Jeder Knoten erhält ungefähr die gleiche Anzahl aufeinander folgender Tasks. Der erste Knoten erhält die ersten Tasks, die nächsten werden dem zweiten Knoten zugeteilt, bis jeder Knoten einmalig die ihm gemäß der Anzahl an Knoten zustehenden Tasks erhalten hat. Dies kann sinnvoll sein, wenn die Tasks gleich groß sind und Lokalität genutzt werden kann. Sind die Tasks unterschiedlich groß, können mehrere große, aufeinander folgende Tasks dem gleichen Knoten zugeteilt werden, was zu einer schlechten Lastenbalancierung führen würde. Bei der Verteilung eines zu bearbeitenden zweidimensionalen Arrays könnten jedem Knoten mehrere aufeinander folgende Zeilen (oder Spalten) zugeordnet oder eine schachbrettartige Aufteilung durchgeführt werden.

- **Blockzyklische Verteilung**

  Als Kompromiss kann jeweils eine kleinere Anzahl aufeinander folgender Tasks zyklisch auf die Knoten verteilt werden. Dadurch wird wie bei der Blockverteilung die Lokalität der zusammenhängenden Daten genutzt. Bei unterschiedlich großen Tasks kann es allerdings, wie bei der Blockverteilung, je nach Programmlogik, zu einer schlechten Lastenbalancierung kommen, da große aufeinander folgende Tasks wiederum dem gleichen Knoten zugeordnet werden können. Jedoch werden die Tasks häufiger voneinander getrennt, so dass eine größere Ansammlung rechenintensiver Tasks eher auf verschiedene Knoten verteilt wird, als bei der Blockverteilung.

2. Verwendete PGAS Sprachen

Aus dem Bereich der PGAS Sprachen wurden Chapel und X10 ausgewählt, da diese Sprachen eine vergleichsweise breite Unterstützung haben und weiterentwickelt werden. So wurde sich beispielsweise gegen die Programmiersprache Titanium entschieden, da diese seit November 2006 nicht mehr weiterentwickelt wird.

Chapel und X10 waren neben Fortress Teilnehmer des „High Productivity Computing Systems" (kurz HPCS) Projekts der DARPA, welches von 2002 bis Ende 2010 durchgeführt wurde. Die DARPA (Defense Advanced Research Projects Agency) gehört zum Verteidigungsministerium der USA und unterstützte in dieser Zeit die Finanzierung der Entwicklung der Sprachen. Mit dem HPCS Projekt sollte eine neue Generation von hochproduktiven Computersystemen für die nationale Sicherheit, aber auch für die Industrie geschaffen werden. Das klare Ziel war dabei die einfache und schnelle Möglichkeit für Programmierer, parallel programmieren zu können, so dass sich die Entwicklung von Programmen unter betriebswirtschaftlichen Gesichtspunkten lohnt. Dafür sollten die Programme leicht zu parallelisieren sein, was zu kürzeren Entwicklungszeiten, weniger Fehlerquellen und möglichst sicheren Programmen führen sollte. Diesem Ziel sollte der PGAS Ansatzes gerecht werden.

In den nächsten Kapiteln folgt eine Übersicht über die verwendeten Sprachen. Neben allgemeinen Informationen werden verschiedene Sprachdetails vorgestellt und anschließend die wichtigsten Parallelisierungskonstrukte beschrieben. Die Kapitel enden mit einer Auflistung der wichtigsten Eigenheiten der Sprachen.

2.1. Chapel

Die Sprache Chapel wird von der Firma Cray Inc. (ein Hersteller von Supercomputern) als Open Source Software unter der BSD Lizenz entwickelt und nahm als Teil des „Cray Cascade project" am HPCS Projekts teil.

Neben Supercomputern und Computerclustern können auch Mehrkern-Desktops und Laptops als Zielarchitekturen gewählt werden. Algorithmus und Daten-Mapping (siehe Kapitel 1) können weitgehend von einander getrennt werden. Chapel unterstützt eine vereinfachende

Datenparallelisierung (Verteilen der Daten auf mehrere Knoten), Task-Parallelisierung und „nested parallelism" im Sinne von Aufteilen von Tasks auf mehrere Knoten und wiederum auf mehrere CPUs innerhalb der Knoten - mehr dazu später. In Chapel heißen die Knoten **Locales**. Ein Locale ist nach Cray [7] in Chapel eine abstrakte Zielarchitektur in Form einer Ansammlung laufender Tasks und bestehender Variablen. In der Praxis handelt es sich normalerweise um einzelne Computer in Form von Prozessoren mit einem gemeinsamen Speicher. Gemäß dem PGAS Ansatz hat ein lokaler Task direkten Zugriff auf lokale Daten. Der Zugriff auf Daten, welche sich auf anderen Locales befinden, ist zwar auch möglich, kostet aber etwas Leistung und Zeit.

Die Syntax der Basisbefehle ähnelt etwas Java und C++. Neben der Organisation von Quelltext in **Modulen**, welche in anderen Dateien importiert werden können, kann auch objektorientiert mit Klassen ähnlich wie in Java und C# programmiert werden. Es ist auch ohne ein umgebendes Modul möglich den Quellcode einer Klasse oder einer Funktion in eine Datei zu schreiben. Chapel erzeugt aus diesem Quellcode implizit ein Modul. Quellcode, welcher außerhalb von Methoden geschrieben wurde, wird beim Programmstart direkt Zeile für Zeile von oben nach unten analog zu einer Scriptsprache abgearbeitet. Das kann auch zum Testen von Programmfragmenten sinnvoll sein.

Vordefinierte Konstanten im Quelltext können beim Programmaufruf vom Nutzer mit entsprechend gekennzeichneten **Befehlszeilenargumenten** überschrieben werden, ohne dass das Programm dafür neu kompiliert werden muss. So kann auch auf den parallelen Ablauf des Programms zur Laufzeit noch Einfluss genommen werden oder Dateinamen von Ein- und Ausgabedateien beliebig geändert werden, ohne dass auf eine bestimmte Anzahl oder Reihenfolge von Programmargumenten geachtet werden muss.

Das Durchlaufen von for-Schleifen wird in der klassischen Programmierung am häufigsten mit einer Zählvariablen realisiert. In Chapel kann neben der standardmäßigen Vorgabe der Inkrementierung oder Dekrementierung des Zählers eine solche Aufzählung als Variable vom Typ **Range** (=Bereich) abgespeichert werden. So kann ein einmal definierter Ablauf einer Zählvariablen (z.B. Zahlen von 20 aufwärts; davon nur jede dritte Zahl; insgesamt 15 Zahlen), einmalig als Muster definiert und in verschiedenen for-Schleifen an verschiedenen Stellen des Programms wiederverwendet und damit auch zentral geändert werden. For-Schleifen, welche ein Range-Muster benutzen, können mit datenparallelen Befehlen kombiniert werden, wodurch die Iterationen in definierbarer Weise auf mehrere Threads verteilt werden. Ranges können allerdings nur eindimensional angelegt werden.

Chapel verwendet aufbauend auf den Ranges ein interessantes Konzept, welches die Trennung von Datenstrukturen wie Arrays und deren logische Adressierung mittels Indizes vorsieht. Die folgenden Betrachtungen treffen neben Arrays auch auf einige Listen und vom Programmierer selbst definierte Datenstrukturen zu: Statt einer vorgegebenen Nummerierung der Speicherzellen von 0 oder 1 aufwärts zählend können alternativ **Domains** verwendet werden. Eine Domain hat für jede Dimension ihre eigene Range, also eine Zusammenstellung von Indizes, so dass eine spätere Datenoperation, im Fall von Arrays, nur auf deren in der Domain definierten Speicherzellen angewendet wird. Die Definition einer Domain kann mit derselben Syntax von Ranges erfolgen oder durch Verwendung von vorhandenen Ranges. Im Gegensatz zu einer Range kann eine Domain völlig frei definierte Bereiche eines Arrays, auch zweidimensionale, enthalten. Zusätzlich zur Range-Syntax können u. a. rechteckige Teilbereiche eines zweidimensionalen Arrays oder Teilbereiche einer bereits vorhandenen Domain als neue Domain definiert werden. Eine eigene Algebra für Domains erlaubt Schnittmengen von mehreren Domains, welche wie beschrieben eine Ansammlung von Indizes sind, zu bilden und als weitere Domain abzuspeichern. Die später erklärte Datenparallelisierung (u. a. forall Schleifen) können auch auf Domains angewendet werden, so dass der Programmierer interessante Möglichkeiten hat, Operationen gezielt auf durch Domains definierte Teilbereiche einer Datenstruktur anzuwenden.

Grundsätzlich kann die Aufteilung von Daten, oder auch nur deren Berechnungen mit mehreren Locales, schon beim Erstellen von Ranges und Domains mittels Definition einer **Domain Map** (vergleiche „Mapping-Techniken" in Kapitel 1) vorbereitet werden. Arrays und andere Datenstrukturen können so über die Locales verteilt abgelegt werden, dass je nach Programmlogik Lokalität genutzt werden kann. Auf den einzelnen Locales können gezielt eigene Tasks platziert werden. Der Vorteil der Sicht des Programmierers auf eine verteilte Datenstruktur wie ein Array, welches nun eigentlich auf verschiedenen Locales abgelegt ist, kann sich als Performancefalle erweisen. Durch den möglichen indirekten Zugriff auf alle Daten über deren Indizes ohne explizite Angabe des dazugehörigen Locales, können versehentlich Daten von anderen Locales gelesen werden. Zur Überprüfung, ob alle Zugriffe eines Programmstücks lokal auf einen Locale beschränkt waren, steht das **local** Statement zur Verfügung. Mit diesem Statement kann ein Programmteil umschlossen werden, worauf zur Ausführungszeit ggf. eine Fehlermeldung erscheint, wenn während dieses Programmteils auf andere Locales zugegriffen wurde. Die Implementierung dieses Statements ist in der untersuchten Chapel Version 1.5.0 noch nicht ausgereift, da Ausgaben am Bildschirm und

nach eigenen Experimenten einzelne lokale Variablenzugriffe zum Auslösen der Assertion führen - siehe auch Kapitel 4.3.1.

In Chapel werden die Parallelisierungskonstrukte in mehrere Kategorien unterteilt, welche miteinander kombinierbar sind. Es wird zwischen **Datenparallelisierung** und **Taskparallelisierung** unterschieden. Die **Taskparallelisierungskonstrukte** ermöglichen es mehrere Programmteile parallel in jeweils eigenen Tasks zu starten und bei Bedarf zu synchronisieren. Für den geschützten Zugriff auf Daten existieren in Chapel allerdings keine Lock Variablen oder Semaphore, wie sie aus vielen Programmiersprachen bekannt sind. Stattdessen können Daten mit Hilfe von Synchronisierungsvariablen vor gleichzeitigem Zugriff geschützt werden. Primitive Datentypen (mit Ausnahme von Arrays) können mit dem zusätzlichen Schlüsselwort **sync** als Synchronisierungsvariablen definiert werden. Eine solche Variable kann den Zustand „voll" oder den Zustand „leer" annehmen. Bevor auf eine Variable erstmals schreibend zugegriffen wird, blockieren alle Threads, welche auf dieser Variablen lesen wollen. Die Variable ist „leer". Nachdem ein anderer Thread schreibend auf die Variable zugegriffen hat, wird diese als „voll" definiert. Einem der wartenden Threads wird dann (nicht deterministisch ausgewählt) der lesende Zugriff gewährt, womit die Variable für alle anderen Threads wieder als „leer" markiert wird, so dass diese weiterhin blockierend warten müssen. Ein Thread kann auf eine solche Variable auch nur schreibend zugreifen, wenn diese gerade als „leer" definiert ist. Anderenfalls blockieren alle Threads, die die Variable beschreiben wollen, bis diese von anderen Threads gelesen wurde. Mit dem sync Statement kann leicht eine Barriere für mehrere Threads implementiert werden.

Als spezielle Alternative zum sync Statement kann eine Variable auch als **single** definiert werden. Diese kann nur einmal beschrieben werden. Analog zur als sync definierten Variable blockieren auch hier alle lesenden Threads, jedoch wird die Variable beim ersten und einzigen schreibenden Zugriff für immer als „voll" definiert, so dass diese für den Rest der Programmlaufzeit keine Threads mehr blockieren wird.

Nach der Verteilung der Daten auf die Locales werden Schleifen, welche parallel abgearbeitet werden sollen, als **forall** oder **coforall** implementiert. Während bei forall-Schleifen Chapel überlassen wird, in wie viele Tasks die Schleife zerlegt wird, ist bei coforall festgelegt, dass jede Iteration zwingend einen eigenen Task darstellt.

Mit Chapels Konzept der **Datenparallelisierung** kann ein Programmierer eine gleichartige Operation (=datenparallele Operation) auf eine Datenstruktur wie ein Array, aber auch eine Range und Domain anwenden, welche dann parallel von mehreren Threads abgearbeitet wird. Bei entsprechend gekennzeichneten forall- und coforall-Schleifen werden die beim Erstellen der Datenstruktur definierten Mappings automatisch berücksichtigt. Im Idealfall erfolgt auf den Locales eine lokale Berechnung der Daten. Bei Zuweisungen, welche Daten von anderen Locales benötigen, führt der Locale die Operation aus, deren Variable ein Wert zugewiesen wird. Man spricht bei dieser Auswahl vom „Leader-Prinzip".

Standardmäßig werden so viele Tasks bei einer Datenparallelisierung generiert, wie dem Locale physische Prozessoren zur Verfügung stehen [8]. Die Anzahl der CPUs wird von Chapel automatisch erkannt und beim Programmstart in eine globale Variable geschrieben. Diese kann vom Programmierer bei Bedarf überschrieben werden, so dass eine andere Anzahl von Tasks generiert wird.

Chapel besitzt keine Pointer, allerdings können einzelne Indizes oder Bereiche eines vorhandenen Arrays als **Array Alias** erneut deklariert werden. Arrays können statt „call by reference" auch als Kopie, also „call by value" an Funktionen übergeben werden.

Bei der Deklaration einer Variablen kann die Angabe eines expliziten **Datentyps**, ähnlich wie in vielen Scriptsprachen, entfallen. Die Variable nimmt dann implizit einen Datentyp an, welcher die erste Zuweisung ermöglicht. Teilweise ist allerdings eine explizite Angabe des Typs notwendig, weshalb zusätzlich zur besseren Typsicherheit die expliziten Typangaben vorzuziehen sind. Soll eine Variable, welche ohne Angabe eines Datentyps innerhalb eines if-else Konstrukts wahlweise mit einem int oder einem float-Wert initialisiert werden, kann Chapel zum Zeitpunkt der Deklaration nicht entscheiden, welcher Dateityp gewählt werden soll.

Die Entwickler von Chapel weisen in der Spezifikation darauf hin, dass sich die Sprache noch in der Entwicklung befindet und manche der dokumentierten Konstrukte noch nicht oder nur teilweise implementiert sind. Außerdem existieren noch viele Performanceprobleme, welche in der weiteren Entwicklung behoben werden sollen.

2.2. X10

Ursprünglich an Java orientiert bietet die Programmiersprache X10 viele Konstrukte welche inhaltlich und syntaktisch identisch mit Java sind. So wird Quellcode in Klassen organisiert und dem Programmierer stehen alle gängigen Konzepte der Objektorientierten Programmierung wie Vererbung, Interfaces und Zugriffskontrollen mit den aus Java bekannten Modifiern public, protected und private zur Verfügung.

X10 Code kann wahlweise mit einem **Java Backend** oder einem **C++ Backend** kompiliert werden. Darüber hinaus kann nativer Java und C++ Quellcode in Form von entsprechend markierter Methoden oder externen Quellcodedateien in den X10 Code eingebettet werden. Leider ist eine Interaktion des X10 Quellcodes mit dem eingebetteten nativen Code nur sehr eingeschränkt möglich. So können nur einige primitive Datentypen von dem „fremden" Quellcode verarbeitet werden. Die Übergabe von Arrays ist noch nicht möglich. Es kann allerdings dynamischer Quellcode geschrieben werden, wobei C++ Code mit Java Code nebeneinander, auch in derselben Methode, existieren kann. Beim Kompilieren mit Java Backend wird der C++ Code ignoriert, beim Kompilieren mit C++ Backend der Java Code. Die Möglichkeiten, welche sich durch nativen Quellcode anderer Programmiersprachen bieten, sind noch stark dadurch eingeschränkt, dass nur wenige für X10 modifizierte Bibliotheken im Java und C++ Code verwendet werden können, wobei diese Einschränkung auch für externe Quellcodedateien gilt. Es ist auch eine Erweiterung des C++ Backends verfügbar, welche es erlaubt CUDA Quellcode für die Ausführung auf NVidia GPUs zu generieren. Diese Möglichkeit wurde in dieser Ausarbeitung aber nicht untersucht.

Analog zur Trennung von Datenstrukturen von deren Adressierung mittels Domains in Chapel, existiert in X10 das Konzept von **Regionen**. Eine Region ist eine Liste von Indizes, welche in Form von Points abgespeichert werden. Für jede Dimension der Region muss eine eigene **IntRange** angegeben werden, welche zwar mit den Ranges aus Chapel vergleichbar sind, jedoch nur direkt aufeinander folgende Zahlen beinhalten können. IntRanges können aber auch analog zu den Ranges in Chapel in Zählschleifen verwendet werden, wobei die Eigenschaft der zusammenhängenden IntRanges der Region das Anwendungsgebiet einschränkt. Eine eigene Algebra, welche u.a. erlaubt Schnittmengen von Regionen als neue Region zu speichern, ist so konstruiert, dass weiterhin nur Regionen mit zusammenhängenden IntRanges entstehen können. Mit den Regionen können Arrays deklariert werden, wobei jedes Array nur eine Region haben kann.

Die Verteilung von Datenstrukturen über mehrere Locales wurde in Chapel mittels Domain Maps realisiert. In X10 heißt das entsprechende Pendant **Distribution** und wird im Quellcode Dist genannt. Bei den Standardverteilungen ist zum Zeitpunkt der Masterarbeit lediglich die Blockverteilung verfügbar. Die zyklische und blockzyklische Verteilung war 2010 noch in der Spezifikation dokumentiert, ist aber in der aktuellen Version des Dokuments nicht mehr vorhanden. In der API ist die Syntax zwar dokumentiert, jedoch sind die Konstrukte nach eigenen Tests nicht lauffähig.

In X10 werden einzelne Knoten als **Place** bezeichnet. Gemäß dem PGAS Ansatz und entsprechend eines Locales in Chapel wird ein globaler Speicher in Places aufgeteilt, wobei jedem Place seine eigenen Daten und Threads zugeordnet sind. Der Speicherzugriff im selben Place sollte wesentlich schneller stattfinden, als ein Zugriff zwischen den Places. Die Threads, welche einem Place vom Betriebssystem zur Verfügung stehen, werden in X10 von **Aktivitäten** (von activity) genutzt. Jeder Task wird von einer Aktivität abgearbeitet und diese werden wiederum auf die Threads abgebildet. Die Spezifikation spricht bei dieser Abstrahierung der wirklichen Threads bei Aktivitäten von „sehr leichtgewichtigen Threads". Aktivitäten können gestartet und bei Bedarf synchronisiert werden. Aktivitäten sind genau einem Place zugeordnet und können nur dessen Daten modifizieren.

Die Vorgehensweise beim Zugriff auf Daten anderer Places unterscheidet sich in X10 entscheidend vom in Chapel eingesetzten Konzept. Ruft ein Place 0 einen anderen Place 1 auf, um dort Rechenoperationen durchzuführen welche auch Variablen vom Place 0 benötigen, werden die Inhalte dieser Variablen, noch vor weiteren Programmausführungen, zum Place 1 kopiert und unter gleichen Dateinamen abgelegt. Es handelt sich um eine so genannte „deep copy", weil keine Verweise auf die Originale übergeben werden. Sämtliche Änderungen an diesen Kopien am Place 1 können am Place 0 nicht beobachtet werden. Selbst mittels des X10 Distribution Konstrukts verteilte Arrays unterliegen diesem Konzept. Sollen Aktivitäten eines Places Daten auf anderen Places verändern, muss dort eine neue Aktivität gestartet werden, welche die Modifikationen dann lokal durchführt. Um das zu erreichen kann eine Referenzierung der entfernten Daten über die Deklaration einer **GlobalRef** oder einem **PlaceLocalHandle** erfolgen. Bei beiden Konstrukten erhalten die Daten einen überall verfügbaren Namen. Die Programmausführung muss selbst bei verteilten Arrays weiterhin auf den Zielplace gewechselt werden, um die „fremden" Daten verändern zu können. In Chapel ist ein solcher Zugriff direkt über die Indizes eines Arrays möglich und der Programmierer muss Acht geben, nicht unbeabsichtigt entfernte Daten zu manipulieren.

Wie in Chapel können Variablen ohne Angabe des Datentyps deklariert werden. Dies führt analog zu Chapel oft zu Problemen mit weiteren Zuweisungen, weshalb eine generelle Angabe des Datentyps empfehlenswert ist. X10 verlangt oft die Verwendung von Konstanten. So beispielsweise, wenn von einem Place auf Daten eines anderen zugegriffen werden soll. Bei Konstanten treten Probleme mit fehlender Typangabe bei der Deklaration nach eigenen Erfahrungen nur selten auf, weil ohnehin keine weitere Zuweisung möglich ist.

3. Beispielproblem: Mandelbrot

Die so genannte Mandelbrot-Menge ist eine Teilmenge des Zahlenraums der komplexen Zahlen. Durch eine Analyse, ob eine Folge von komplexen Zahlen divergiert oder konvergiert, wird entschieden, ob eine komplexe Zahl *c* eines zweidimensionalen Ergebnisraums in der komplexen Ebene entweder die Farbe Schwarz, im Fall der Konvergenz, oder Weiß zugewiesen bekommt. Es ist auch eine, in dieser Ausarbeitung nicht verwendete, Variante möglich, in denen die nach unterschiedlich vielen Iterationen abgebrochenen Punktiterationen in Graustufen oder verschiedenen Farben darstellt werden. Die Mandelbrot-Menge lässt sich in der komplexen Ebene als optisch interessantes Bild wie in Abbildung 2 darstellen.

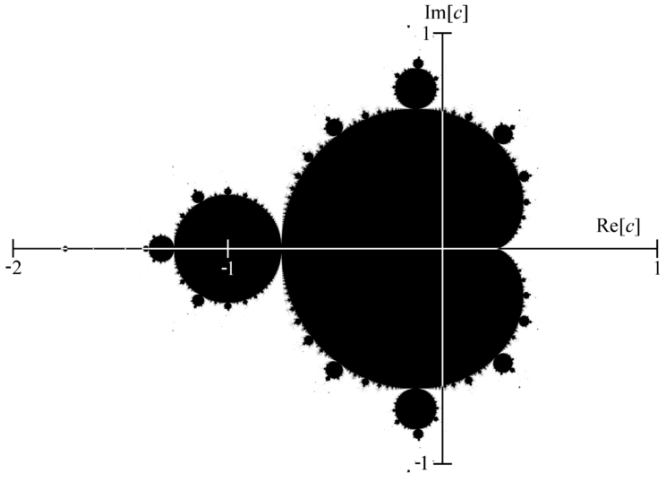


Abbildung 2 Mandelbrot in der komplexen Ebene [9]

Für das Bild in Abbildung 2 wurden die komplexen Zahlen aus dem reellen Bereich [-2, 1] und den imaginären Bereich [-1, 1] untersucht, auf dem die Figur des Mandelbrots im näheren Sinne entsteht. Die Mandelbrot-Menge ist spiegelsymmetrisch zur waagerechten reellen Achse. Bei höherer Auflösung eines Bildes, mit entsprechend mehr zu berechnenden Bildpunkten, werden viele weitere Details sichtbar.

3.1. Beschreibung der Problemlösung

Um eine grafische Darstellung zu erhalten, wird ein Ergebnisraum definiert, welcher aus den Indizes eines zweidimensionalen Arrays oder den Pixeln eines Bildes besteht. Die einzelnen x und y Koordinaten dieser späteren Bildpunkte werden als komplexe Zahlen eines vorgegebenen Bereichs interpretiert, wobei die vertikale Koordinate als Realteil und die horizontale Koordinate als Imaginärteil der komplexen Zahl definiert wird. Diese werden nun vor deren Berechnung in den Zahlenraum der komplexen Zahlen gemappt, in dem das zentrale Bild des Mandelbrots entsteht. Alle Zahlen werden nun auf ihre Divergenz bzw. Konvergenz nach der bei $z_0=0$ beginnenden rekursiven Iterationsvorschrift

$$z_{n+1} = z_n^2 + c$$

getestet. Die komplexen Zahlen, deren Folgeglieder nach der Iterationsvorschrift nicht nach einer definierten maximalen Iterationsanzahl über eine vorher festgelegte obere Schranke wachsen, werden als divergierend gegen unendlich definiert und fallen in die optisch interessante Menge des Mandelbrots. Als obere Schranke wird oft definiert, dass das Quadrat des Betrags der komplexen Zahl nicht größer als zwei werden darf. Die Anzahl der Iterationen sollte mindestens 100 betragen.

Ein Algorithmus, welcher sich an die Iterationsvorschrift hält, führt eine Punktiteration für jeden Index aus, bis die obere Schranke oder die Iterationsgrenze erreicht wird. Die Ergebnismenge kann in einem Integer- oder boolean Array abgelegt werden, wobei lediglich gespeichert werden muss, ob ein Punkt zur Mandelbrotmenge gehört oder nicht. Eine Implementierung der seriellen Vorgehensweise (ohne umgebendes Beispielprogramm) ist in Abbildung 3 zu sehen. Während `calculate_mandelbrot` die Punktiteration (Zeile 9) für jeden Index eines übergebenen Arrays ausführt, führt die Methode `point_iteration` die Berechnung der Folgeglieder der komplexen Zahl anhand der Iterationsvorschrift durch. Wurde die vorgegebene Iterationsgrenze `max_iteration` erreicht, wird davon ausgegangen, dass diese komplexe Zahl gegen unendlich divergiert und mit einer 1 im Array eingetragen. Ein Integerarray kann einem Boolean-Array vorgezogen werden, weil sich aus Nullen und Einsen eine Bilddatei im pbm Format erstellen lässt, welches als schwarz/weiß Bild betrachtet werden kann. Der else-Zweig mit dem Beschreiben des Arrays mit einer Null in Zeile 14 könnte in Chapel entfallen, da Arrays dort implizit mit Nullen initialisiert sind.

```
1  proc calculate_mandelbrot(max_iteration, width, height, myArray) {
2      for y in 0..height-1 {
3          for x in 0..width-1 {
4              // for the pixel x, y calculate the corresponding complex number
5              var cx: real = -2.0 + ( x * (4.0 / ( width - 1 ) ) ) ;
6              var cy: real = -1.5 + ( y * (3.0 / ( height - 1 ) ) ) ;
7
8              // calculate max iterations to reach the limit
9              var iteration_count: int = point_iteration(cx, cy, max_iteration);
10
11             if (iteration_count == max_iteration) {
12                 myArray[y][x] = 1;
13             } else {
14                 myArray[y][x] = 0;
15             }
16         }
17     }
18 }
19
20 proc point_iteration(cx: real, cy: real, max_iteration: int) {
21     var iteration: int = 0;
22     var cxi: real = 0;
23     var cyi: real = 0;
24
25     // calculate next complex number until the absolute value of the complex number
26     // is greater then 2 or the max iterations limit is reached
27     while ( cxi * cxi + cyi * cyi <= 4 && iteration < max_iteration ) {
28         var xTemp: real = cxi * cxi - cyi * cyi + cx;
29         var yTemp: real = 2 * cxi * cyi + cy;
30         cxi = xTemp;
31         cyi = yTemp;
32         iteration += 1; // there is no increment operator in chapel like i++
33     }
34     return iteration;
35 }
36
```

Abbildung 3 Serielle Mandelbrot-Implementierung in Chapel

Die Spiegelsymmetrie der Mandelbrot-Menge würde es auch erlauben, die Berechnung eines Bildes ausschließlich im positiven imaginären Bereich durchzuführen. Anschließend würde es ausreichen, den berechneten Bereich beim Erstellen eines Bildes auf die untere Hälfte gespiegelt zu übertragen.

3.2. Allgemeine Parallelisierungsmöglichkeiten

Zwischen den einzelnen Bildpunkten bestehen keinerlei Abhängigkeiten. Deshalb kann die Berechnung der einzelnen Bildpunkte in beliebiger Reihenfolge und parallel erfolgen. Da die schwarze Fläche des Mandelbrots bis zum Iterationsmaximum berechnet werden muss, fällt hier der größte Rechenaufwand an. Ein Array für das Berechnen des Mandelbrots hat eine beachtliche Größe, da ein zu generierendes Bild mindestens VGA Auflösung haben sollte. Eine gleichzeitige Berechnung aller Bildpunkte mit einer entsprechend hohen Menge an

Threads wäre daher mit sehr viel Overhead verbunden. Deshalb sollten die einzelnen Tasks (=Berechnung eines Bildpunkts) mit nur wenigen Threads, angepasst an die Anzahl der CPUs, bearbeitet werden. Als erster Ansatz sollte eine Aufteilung des Arrays gewählt werden, welche den Bereich des Mandelbrots möglichst gleichmäßig auf verschiedene Knoten verteilt. Dadurch soll vermieden werden, dass Knoten ihre Berechnungen bedeutend früher beendet haben als andere und dann untätig bleiben. Dazu kommen die in Kapitel 1 beschriebenen Mapping-Techniken „zyklische Verteilung" und „blockzyklische Verteilung" in Frage. Die Blockverteilung wäre keine sinnvolle Wahl, denn bei einer Blockverteilung mit waagerechten Streifen hätte der Kern, der den Bereich am oberen Rand zugeteilt bekäme, weniger Arbeit, als ein Knoten, welcher einen mittleren Bereich bekommt. Bei einer Blockverteilung mit senkrechten Streifen oder einer Blockverteilung im Schachbrettmuster tritt dieses Problem in ähnlicher Weise auf. Da im Fall von zyklischer und blockzyklischer Verteilung eine faire Verteilung der Mandelbrotfläche gewährleistet wird, wurde sich für die Implementierungen willkürlich für waagerecht blockzyklisch verteilte Streifen entschieden, wobei die einzelnen Blöcke, wie in Abbildung 4 links ersichtlich, die Größe eines solchen Streifens haben. Alternativ könnte die Aufteilung des Mandelbrots mit blockzyklischer Verteilung auch über ein Schachbrettmuster erfolgen, welches das Mandelbrot in rechteckige oder quadratische kleine Teile zerlegt, welche zyklisch auf die Knoten verteilt werden (Abbildung 4 rechts). So würde jeder Knoten Teile des Mandelbrots aber auch Teile mit „freier Fläche" erhalten [9].

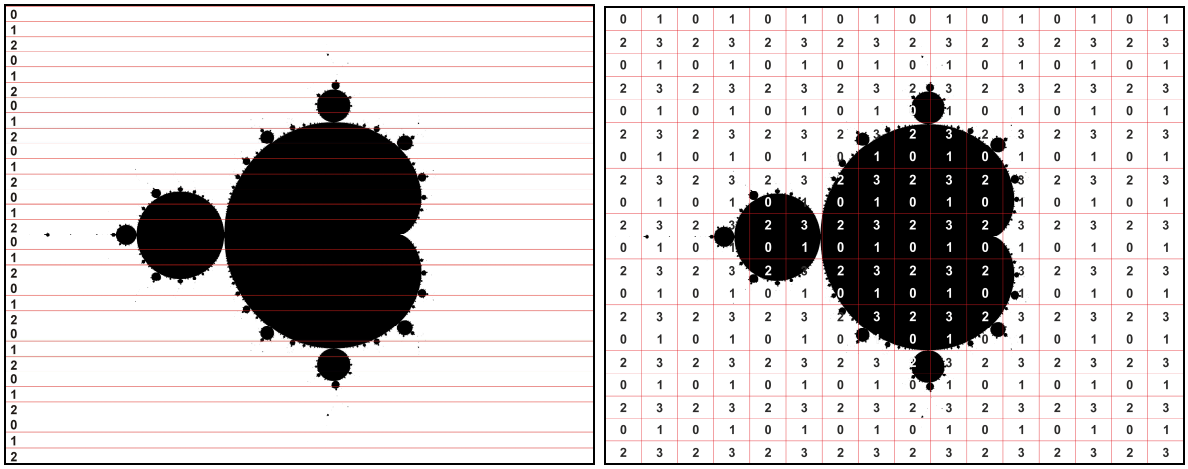


Abbildung 4 Mandelbrot-Menge mit blockzyklischer Verteilung: Links zeilenweise verteilt auf drei Knoten; Rechts Schachbrettmuster auf vier Knoten verteilt

Die auf die Knoten verteilten waagerechten Streifen können dort wiederum lokal mit mehreren Threads bearbeitet werden, so dass alle CPUs ausgelastet werden können. Jede Punktiteration stellt dabei einen eigenen Task dar. Die Threadanzahl kann dabei gering gehalten und der Anzahl der CPUs angeglichen werden.

3.3. Implementierung in den einzelnen Sprachen

Implementierungen des parallelen Algorithmus zur Lösung der Mandelbrot-Menge wird nun in Chapel und X10 vorgestellt. Dabei wird zunächst auf die Verteilung der Daten auf mehrere Knoten eingegangen und anschließend die Verteilung von Tasks auf Threads behandelt.

3.3.1. Mandelbrot in Chapel

In Chapel kann ein Array mit Hilfe von Domain Maps blockzyklisch auf mehrere Locales (Knoten) verteilt werden. Im Gegensatz zur im Kapitel 1 beschriebenen Definition einer blockzyklischen Verteilung, kann diese klassischerweise zeilenweise, spaltenweise oder mit einem schachbrettartigen Muster erfolgen. Leider ist die blockzyklische Verteilung in Chapel lediglich mit Schachbrettmuster implementiert, welches aber mit verschiedenen Parametern beeinflusst werden kann. Um Chapels Implementierung genauer zu verstehen, sehen wir uns eine Testausgabe eines Arrays, welches bereits auf vier Locales verteilt wurde, mittels folgenden Codefragments an:

```
forall a in A do
    a = a.locale.id;
writeln(A);
```

Mit der „forall" Schleife werden alle Locales, auf die das Integer Array A verteilt wurde, aufgefordert, die Operationen des Body auf die ihnen zugeteilten Arraybereiche anzuwenden. Im Codefragment besteht der Body nur aus einer Zeile und schreibt in jedes Element a des Arrays die ID des Locales, auf welchem dieses Element abgelegt ist. Die Locales sind automatisch von 0 aufwärts durchnummeriert, so dass sich bei der Ausgabe des Arrays auf der Konsole mittels writeln(), bei der von uns eingesetzten blockzyklischen Verteilung, folgende Bilder ergeben können:

```
0 0 0 1 1 1 0 0
0 0 0 1 1 1 0 0
2 2 2 3 3 3 2 2
2 2 2 3 3 3 2 2
0 0 0 1 1 1 0 0
0 0 0 1 1 1 0 0
2 2 2 3 3 3 2 2
2 2 2 3 3 3 2 2

Blockgröße (2,3)
```

```
0 0 0 0 0 0 0 1
2 2 2 2 2 2 2 3
0 0 0 0 0 0 0 1
2 2 2 2 2 2 2 3
0 0 0 0 0 0 0 1
2 2 2 2 2 2 2 3
0 0 0 0 0 0 0 1
2 2 2 2 2 2 2 3

Blockgröße (1,7)
```

```
0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2
0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2
0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2
0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2

Blockgröße (1,8)
```

Abbildung 5 Testausgaben einer blockzyklischen Verteilung

Die Blockgröße, welche zyklisch auf das zweidimensionale Array verteilt werden soll, ist im ersten Bild von Abbildung 5 auf einen 2*3 Bereich eingestellt. Es handelt sich um eine schachbrettartige blockzyklische Verteilung mit überwiegend 2*3 großen Feldern. Weil es sich um die einzige Implementierung eines blockzyklischen Verteilungsmusters in Chapel handelt, sehen wir uns noch an was passiert, wenn die Blockgröße des Verteilungsmusters auf die Höhe und Breite einer Zeile eingestellt wird, um evtl. doch eine zeilenweise Verteilung zu erreichen. Im zweiten Bild ist vorerst zu sehen, dass die Größe der einzelnen Blöcke, welche einem Locale zugeordnet werden, die entsprechende Größe annimmt, auch wenn diese fast so breit wie das Array sind. Dabei bleiben für die Locales 1 und 3 kaum Elemente des Arrays übrig. Im dritten Bild ist die Blockgröße genau an eine Zeile des Array angepasst. Statt mit diesem Muster in der Reihenfolge 0, 1, 2, 3 die Zeilen aufzuteilen erhalten nur noch die Locales 0 und 2 Elemente. Die Verteilung auf die Locales wird aufgrund dieser Beobachtungen grob etwa mit dem folgenden Muster durchgeführt:

L0 L1
L2 L3

Abbildung 6 Blockzyklisches „schachbrettartiges“ Verteilungsmuter

Es kann mit diesem Verteilmuster also ausschließlich eine schachbrettartige Verteilung realisiert werden, wobei eine blockzyklische Verteilung weder zeilenweise noch spaltenweise möglich ist. Das ist überraschend, da zeilenweise Verteilungen von Arrays aufgrund von Cacheeffekten wie dem Vorausladen von nachfolgenden Elementen eines Arrays zum Standard gehören. Würden wir auf die zeilenweise Verteilung verzichten und die beschriebene blockzyklische Verteilung von Chapel nutzen, würde der Quelltext, wie in Abbildung 7 ersichtlich, überraschend kurz ausfallen. Die Domain wird dabei in Zeile 6 (ab dem ersten Index) mit Blöcken der Dimension 5x8 verteilt und mit dieser das Array „theArray“ angelegt.

```
1  // dimensions of the new Array
2  const Space = [0..#height, 0..#width];
3
4  // distribute a domain with the default "chessboard" distritution with
5  // 5x8 fields and create a new array named "theArray" with this domain
6  const D: domain(2) dmapped
7      BlockCyclic(startIdx=Space.low, blocksize=(5, 8)) = Space;
8  var theArray: [D] int;
```

Abbildung 7 Default blockzyklische Verteilung eines Arrays auf mehre Locales

Statt dieser Default-Verteilung kann aber ein eigenes Verteilungsmuster implementiert werden. Diese Möglichkeit ist in der Spezifikation von Chapel allerdings (noch) nicht dokumentiert und ist damit vorerst als unsicherer „Hack" einzustufen. Um dennoch wie vorher geplant eine zeilenweise Verteilung auf alle Locales zu erreichen, wurde ein neues Verteilungsmuster implementiert. Statt des Musters mit vier Feldern in einer 2*2 Anordnung, wie in Abbildung 6, hat das neue Muster nur ein Feld, mit dem die Locales verteilt werden sollen und ist daher lediglich eindimensional (Abbildung 8 Zeile 2). Das hat den Effekt, dass die Verteilung über ein vorgegebenes Muster praktisch außer Kraft gesetzt wird und die gewünschte zeilenweise Verteilung implementiert werden kann.

```
1   // write an own 1x1 pattern and override the default distribution pattern (Grid)
2   const GridShape = [1..numLocales, 1..1];
3
4   // transfer the new Pattern to an array with the expected format of
5   // the BlockCyclic distribution
6   const Grid: [GridShape] locale = reshape(Locales, GridShape);
7
8   // dimensions of the new Array
9   const Space = [0..#height, 0..#width];
10
11  // distribute a domain with the own distritution pattern "Grid" with the width
12  // of a line and create a new array named "theArray" with this domain
13  const D: domain(2) dmapped
14      BlockCyclic(startIdx=Space.low, blocksize=(10, width), Grid) = Space;
15  var theArray: [D] int;
```

Abbildung 8 Manuelle blockzyklische Verteilung eines Arrays auf mehrere Locales [10]

In den Zeilen 13-14 von Abbildung 8 wird nun eine zweidimensionale Domain „D" erstellt, welche eine blockzyklische Verteilung des gesamten definierten Adressraums (Space aus Zeile 9) ab dem ersten Index vorsieht. Ein Array, welches mit dieser Domain instanziiert wird, hat die Dimensionen von „Space" und wird in waagerechten Streifen in der Breite des Arrays und der Höhe von jeweils 10 Indizes aufgeteilt. Die einzelnen Tasks beinhalten also die Einträge von 10 Zeilen.

Speziell bei der Berechnung des Mandelbrots wäre die schachbrettartige Defaultverteilung des Arrays ähnlich sinnvoll gewesen wie die zeilenweise Verteilung. Es wurde sich aber hier aus Demozwecken für die zeilenweise Verteilung entschieden. Eine weitere Möglichkeit, die zeilenweise Verteilung zu erreichen, wird im späteren Kapitel Gaußsches Eliminationsverfahren besprochen und implementiert.

Weil zwischen den Iterationen keine Abhängigkeiten bestehen und deshalb egal ist, in welcher Reihenfolge die Iterationen bearbeitet werden, habe ich mich für die forall-Schleife

entschieden (verg. Kapitel 2.1). Den Einsatz der forall-Schleife kann in Zeile 3 von Abbildung 9 nachvollzogen werden. Statt dem seriellen Durchlaufen des Arrays mittels zweier geschachtelter for-Schleifen wie in den auskommentierten Zeilen 4 und 5 ist bei dem über die Locales verteiltem Array nur noch die Iteratorform der for-Schleife möglich. Das führt überraschenderweise zu umständlichen workarounds, da das verteilte Array zwar die zweidimensionalen x und y Koordinaten in seiner Domain enthält, aber keine Länge und Breite des Arrays kennt, wie bei lokalen zweidimensionalen Arrays. Diese müssen bei Bedarf schon beim Anlegen des Arrays gespeichert und ggf. Methoden übergeben werden. Ohne die Breite des Arrays zu kennen, könnte ein Array beispielsweise nicht zweidimensional am Bildschirm ausgegeben werden, da der Zeilenumbruch nicht implementiert werden könnte. Weil die Methode `calculate_mandelbrot` die x und y Koordinaten als Variablen benötigt, wurde in der for-Schleife die Domain des Arrays übergeben, um die entsprechenden Koordinaten zum aktuellen Arrayelement zu erhalten.

```
1  proc calculate_mandelbrot(max_iteration, width, height, theArray, debugDistArray) {
2
3      forall (a, (y, x), t) in (theArray, theArray.domain, debugDistArray) {
4      //for y in 0..height-1 {
5          //for x in 0..width-1 {
6              // calculate the complex value of the given pixel
7              var cx: real = -2.0 + ( x * (4.0 / ( width - 1 ) ) ) ;
8              var cy: real = -1.5 + ( y * (3.0 / ( height - 1 ) ) ) ;
9
10             // calculate depth of the point iteration of the complex value
11             var iteration_count: int = point_iteration(cx, cy, max_iteration);
12
13             if (iteration_count == max_iteration) {
14                 //theArray[y][x] = 1;
15                 a = 1;
16             } else {
17                 //theArray[y][x] = 0;
18                 a = 0;
19             }
20             if (debug) {
21                 t = a.locale.id;
22             }
23         //}
24     }
25
26 }
```

Abbildung 9 Parallele calculate_mandelbrot Methode in Chapel

Zur optischen Überprüfung, ob alle Berechnungen auf den richtigen Locales durchgeführt wurden, wurde das Array debugDistArray übergeben, welches mit dem formalen Parameter *t* in der forall-Schleife genutzt wird. Dieses erhält in Zeile 21 an jedem Index die eindeutige ID des Locales zugewiesen, welche in der aktuellen Iteration aktiv ist. So können etwa die Beispiele aus Abbildung 5 bestätigt werden. Man beachte in Abbildung 9, dass ein Zugriff auf das verteilte Array in den Zeilen 15 und 18 nicht mehr über die auskommentierte Index-Schreibweise möglich ist.

Die Berechnungen der einzelnen Elemente werden von den Locales vorgenommen, auf welchen sie sich lokal befinden. In diesem speziellen Fall der Mandelbrotberechung müssen die Indizes der einzelnen Elemente im Array für die Berechnungen zur Verfügung gestellt werden, weswegen der Schleifenkopf von parallel abzuarbeitenden Schleifen (vergl. spätere Abbildung 9) komplizierter ausfällt, als es in Chapel normal nötig ist.

Die forall-Schleife nutzt nun nicht nur auf jedem Locale die lokalen Daten für ihre entsprechenden Berechnungen, sondern verteilt die einzelnen Tasks zusätzlich auf die Threads. In den Programmiersprachen MPI, OpenMP und pthreads wäre dafür weiterer Quelltext nötig gewesen. Die Anzahl der für die Schleife verwendeten Tasks kann vom Programmierer bei Bedarf beeinflusst werden (vergl. Kapitel 2.1).

Die Übertragung des Arrays in eine neue Datei vom Typ .pbm ist mit der in Chapel implementierten File IO möglich, deren Syntax der Programmiersprache C ähnelt.

3.3.2. Mandelbrot in X10

Wie in Chapel wollen wir auch in X10 die geplante blockzyklische Verteilung implementieren. Ähnlich wie in Chapel gibt es auch hier ein Problem der Implementierung in X10. So sind die zyklische Verteilung sowie die blockzyklische Verteilung zwar in der Spezifikation von 2010 dokumentiert, aber inzwischen wieder entfernt worden. Die durch die Dokumentation der API beschriebene Syntax kann von X10 zwar kompiliert werden, ist aber nach eigenen Tests nicht lauffähig. Sehen wir uns daher die einzige fertig implementierte Verteilung in Abbildung 10 an, um eine typische X10 Implementierung zu erhalten. Es handelt sich um die Blockverteilung. In Zeile 2 wird zuerst eine zweidimensionale Region mittels zweier IntRanges erstellt. Aus dieser Region wird in Zeile 3 nun eine X10 Distribution mit einer Blockverteilung erstellt. Das neue Array in Zeile 7, welches auf dieser Verteilung basiert, wird zuletzt mit Nullen initialisiert. Dafür werden alle zweidimensionalen Points auf Nullen abgebildet.

```
1  // Create a Region and a Block-Distribution
2  val R <: Region = 0..(height-1) * 0..(width-1);
3  val D <: Dist = Dist.makeBlock(R);
4
5  // Create a new int array named "theArray" with the
6  // Distribution D initialized with Points of zeros
7  val theArray <: DistArray[Int] =
8      DistArray.make[Int](D, (Point(2))=>0);
```

Abbildung 10 X10 Blockverteilung eines Arrays

Wie bereits erwähnt, eignet sich die Blockverteilung eigentlich nicht, das Array der Mandelbrotmenge sinnvoll aufzuteilen. Sehen wir uns deshalb die in der API beschriebene Syntax der blockzyklischen Verteilung, welche nicht lauffähig ist, im Vergleich an:

```
1  // Create a Region and a Block-Distribution
2  val R <: Region = 0..(height-1) * 0..(width-1);
3  val D <: Dist = Dist.makeBlockCyclic(R, 1, width*10);
4
5  // Create a new int array named "theArray" with the
6  // Distribution D initialized with Points of zeros
7  val theArray <: DistArray[Int] =
8      DistArray.make[Int](D, (Point(2))=>0);
```

Abbildung 11 X10 blockzyklische Verteilung eines Arrays

In Zeile 3 von Abbildung 11 wird diesmal eine blockzyklische Verteilung mit drei Parametern erstellt. Neben der Region wird als zweites Argument die Dimension der Blöcke angegeben, welche zyklisch verteilt werden sollen. Als letztes Argument wird die Größe der Blöcke als ganze Zahl angegeben. Die Angabe eindimensionale Blöcke in der Breite des Arrays und Höhe von 10 Zeilen zu verteilen, sollte zur gewünschten zeilenweisen Blockverteilung führen, wobei dies natürlich nicht getestet werden kann.

Sehen wir uns nun die zentrale Methode zur Berechnung des Mandelbrots in Abbildung 12 an. Um die Methode erst zu verlassen, wenn alle Aktivitäten ihre Arbeit beendet haben, wird die äußere for-Schleife in Zeile 4 mit dem finish-Statement umgeben. In den Zeilen 4-6 läuft die Schleife über alle Places und erstellt für jeden Place mit dem async Statement eine neue Aktivität, so dass direkt mit der nächsten Iteration der Schleife begonnen werden kann und die Places somit praktisch parallel gestartet werden. Jeder Place *p* führt nun die innere for-Schleife aus und filtert dabei durch die Pipe-Schreibweise in Zeile 7 aus der X10 Distribution des Arrays „theArray" nur die Points heraus, welche dem Place *p* zugeordnet sind. Man beachte, dass sich ein Point als eindimensionales Array darstellen lässt, womit leicht auf die Indizes zugegriffen werden kann.

```
1  public static def calculate_mandelbrot(val max_iteration: Int,
2                      val width: Int, val height: Int, theArray: DistArray[Int](2)) {
3
4      finish for (pid in 0..(Place.MAX_PLACES-1)) {
5          val p = Place(pid);
6          async at(p) {
7              for ([y, x] in theArray.dist | p) {
8              //for ([y, x] in theArray) {
9                  async {
10                     // calculate the complex value of the given pixel
11                     var cx: double = -2.0 + ( x * (4.0 / ( width  - 1 ) ) ) ;
12                     var cy: double = -1.5 + ( y * (3.0 / ( height - 1 ) ) ) ;
13
14                     // calculate depth of the point iteration of the complex value
15                     var iteration_count: int = point_iteration(cx, cy, max_iteration);
16
17                     if (iteration_count == max_iteration) {
18                         theArray(y,x) = 1;
19                     } else {
20                         theArray(y,x) = 0;
21                     }
22                 }
23             //}
24             }
25         }
26     }
27 }
```

Abbildung 12 Parallele calculate_mandelbrot Methode in X10

Die mittlere for-Schleife kann nicht, wie in Chapels Datenparallelisierung automatisiert, verteilt werden. Deshalb erstellen wir in Zeile 9 mittels async eine Aktivität für den Body jeder Iteration der Schleife, so dass diese parallel von Threads bearbeitet werden können, um mehrere Prozessoren nutzen zu können.

Im Vergleich zur sequentiellen Version der Methode calculate_mandelbrot kamen die finish und async Statements hinzu sowie der Tausch der auskommentierten for-Schleife über alle Points des Arrays gegen die for-Schleife über das verteilte Array. Der Original Quellcode ist immer noch gut lesbar, was die Einfachheit und Übersichtlichkeit der Parallelisierung demonstriert.

Das Übertragen des Arrays in eine Datei war leider nicht möglich. Konstrukte für die Eingabe und Ausgabe in bzw. von Dateien sind in X10 noch nicht implementiert. Es besteht zwar die Möglichkeit, nativen Java oder C++ Code einzubinden (siehe auch Kapitel 2.2), jedoch kann eine Datenstruktur wie ein Array noch nicht an eine Java bzw. C++ Methode übergeben werden.

3.4. Bewertung der einzelnen Sprachen in Bezug auf die Implementierung des Mandelbrot-Problems

Die Mandelbrot-Menge ist ein Beispiel einer einfachen Parallelisierung von Berechnungen der Arrayelemente ohne Datenabhängigkeiten zwischen diesen Elementen. Die Formulierung in Chapel fällt sehr kurz aus und ist leicht zu lesen. Die Vorüberlegungen zur Parallelisierung müssen einmalig beim Erstellen der Domain getroffen werden und werden automatisch in allen „forall"-Schleifen die diese Domain betreffen angewendet, so dass jeder Locale gezielt nur auf seine lokalen Daten zugreift. In X10 ist die Parallelisierung mit mehr Denkaufwand verbunden, denn es muss vom Programmierer für jeden Place eine Aktivität erstellt werden. Die Syntax zum Filtern der lokalen Elemente aus dem verteilten Array ist praktisch, jedoch ist auch das in Chapel nicht nötig.

Das Überschreiben von Programmargumenten über die Befehlszeile (in willkürlicher Reihenfolge) macht in Chapel Tests unterschiedlichster Einstellungen möglich, ohne den Quelltext zu ändern oder das Programm neu zu kompilieren. So können Mandelbrotbilder in niedriger und hoher Auflösung mit verschiedenen Dateinamen erstellt werden und optional die maximale Iterationszahl eingestellt werden, ohne dass die Parameterübergabe zusätzlich zur Markierung von überschreibbaren Variablen speziell implementiert.

Negativ fällt in Chapel das unnötig komplizierte Ausgeben eines Arrays auf der Konsole oder die identisch implementierte Übertragung in eine Datei aus. Auf die einzelnen Indizes des über eine Domain instanziierten Arrays kann nämlich nicht mit der selben Syntax zugegriffen werden wie bei einem klassischen Array, das durch einfache Angabe von zwei Dimensionen instanziiert wurde. Umgekehrt ist der Zugriff auf ein klassisches Array nach eigenen Experimenten mit der Domainschreibweise nicht möglich.

Die zeilenweise Verteilung des Arrays fällt in Chapel unnötig kompliziert aus. Trotz der Möglichkeit mit copy&paste des Quellcodes aus Abbildung 8 die Defaultverteilung jederzeit außer Kraft zu setzen, verdoppeln sich die Codezeilen, mit welchen die Verteilung definiert wurde, was das Lesen und Warten des Codes schwerer macht. Andererseits wurde sich bereits vor der Implementierung für eine Aufteilung des Arrays entschieden. Wenn Chapels Möglichkeiten schon vor der Parallelisierung bekannt sind und in die Parallelisierungsüberlegungen einfließen, kann sicherlich häufig, so wie im Fall der

Mandelbrot-Menge, die default-Verteilung gewählt werden um durch weniger Quellcode einfacheren und kompakteren Quellcode zu erhalten.

Die Verteilung des Arrays in X10 ist in der blockzyklischen Variante zwar nicht mal verfügbar, jedoch lässt die angekündigte Syntax auf eine einfache Nutzung des Statements mit vielen Optionen der Verteilung schließen.

4. Beispielproblem: Gaußsches Eliminationsverfahren

Das Gaußsche Eliminationsverfahren (oder auch Gauß-Algorithmus) ist ein Algorithmus zum Lösen von linearen Gleichungssystemen. Die folgende Beschreibung des Algorithmus orientiert sich an Dankert [12]. Gegeben ist ein lineares Gleichungssystem mit n Gleichungen und n Unbekannten. Diese stehen in einer Beziehung der Form

$$\boldsymbol{A\,x = b}$$

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}, \qquad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \qquad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

Abbildung 13 Matrix A mit Ergebnisvektor b und Unbekannte x

wobei A eine quadratische Matrix ist. Ausführlich geschrieben sieht das Gleichungssystem wie in Abbildung 14 aus.

$$\begin{array}{rcrcrcrcl}
a_{11}x_1 & + & a_{12}x_2 & + & a_{13}x_3 & + & \ldots\ldots & + & a_{1n}x_n = b_1 \\
a_{21}x_1 & + & a_{22}x_2 & + & a_{23}x_3 & + & \ldots\ldots & + & a_{2n}x_n = b_2 \\
a_{31}x_1 & + & a_{32}x_2 & + & a_{33}x_3 & + & \ldots\ldots & + & a_{3n}x_n = b_3 \\
\vdots & & \vdots & & \vdots & & & & \vdots \quad \vdots \\
\vdots & & \vdots & & \vdots & & & & \vdots \quad \vdots \\
a_{n1}x_1 & + & a_{n2}x_2 & + & a_{n3}x_3 & + & \ldots\ldots & + & a_{mn}x_n = b_n
\end{array}$$

Abbildung 14 Lineares Gleichungssystem

Zu gegebenen A und b werden die Unbekannten x gesucht. Für den Fall dass die Matrix A regulär ist (d.h. ihre Determinante ist ungleich Null) hat das lineare Gleichungssystem eine eindeutige Lösung.

4.1. Beschreibung der Problemlösung

Durch Umformungen der Gleichungen, welche die Lösung nicht beeinflussen, kann eine Stufenform erreicht werden, bei der alle Elemente der Diagonalen von links oben nach rechts unten mit Einsen belegt sind. Alle Elemente unterhalb dieser Diagonalen werden dabei zu Nullen. Diese wird in Matrixschreibweise auch Dreiecksmatrix genannt. Um von dieser Dreiecksmatrix ausgehend anschließend die endgültigen Lösungen der Unbekannten zu

berechnen, können bereits bekannte Variablen durch Rückwärtseinsetzung, die so genannte Rückwärtssubstitution, von der unteren bis zur obersten Zeile genutzt werden. Da in der untersten Zeile der Dreiecksmatrix nur eine Variable steht, kann deren Wert trivial abgelesen werden. In Abbildung 15 ist ein Beispielquellcode, zum besseren Verständnis in sequentieller Form und ohne Rückwärtssubstitution in Chapel, zu sehen. Dieser formt das Gleichungssystem A*x = b in ein System U*x = y um, bei dem U Dreiecksform hat und die Lösungsseite b in die Werte y umgeformt ist. Bei dem Beispielquelltext wird zur leichteren Lesbarkeit davon ausgegangen, dass eine eindeutige Lösung existiert und sich in den Zeilen 8 und 11 keine Nullen an der Position der Matrix befinden, durch die dividiert wird. Außerdem wird die so genannte „Pivotisierung" nicht durchgeführt, welche für kleinere Rundungsfehler sorgen kann. Dabei wird die Arrayzeile mit dem betragsmäßig größtem vorderen Element mit der Zeile getauscht, welche den nächsten Divisionsschritt durchführt um möglichst nicht durch eine sehr kleine Zahl teilen zu müssen.

```
1  proc gaussian_elimination(A, b, y) { // A[][]real b[]real y[]real
2      var n: int = A.numElements;
3
4      for k in 0..n-1 {   // for every line do...
5
6          // Division step
7          for j in k+1..n-1 {
8              A[k][j] /= A[k][k];      // Divide whole line except
9                                       //  A[k][k] and leading zeros
10         }
11         y[k] = b[k] / A[k][k];       // Division step: b value
12         A[k][k] = 1;                 // don't divide A[k][k] by itself because of
13                                      //  rounding errors
14
15         // Elimination step
16         for i in k+1..n-1 {          // for every line below the k-th do...
17             for j in k+1..n-1 {
18                 A[i][j] -= A[i][k] * A[k][j];   // Substract scaled values of the
                   k-th row
19                                                 //  from the j-th row except
20                                                 //  A[k][k] and leading zeros
21             }
22             b[i] -= A[i][k] * y[k]; // Elimination step: b value
23             A[i][k] = 0;            // don't subtract to calculate the zero because
24                                     //  of rounding errors
25         }
26     }
27 }
```

Abbildung 15 Serieller Gauß Algorithmus in Chapel

Während sich die Zeilen 7-12 mit dem Erzeugen einer 1 in der aktuell zu berechnenden Zeile k beschäftigen, wird in den Quellcodezeilen 16-25 die Zeile k genutzt, um eine 0 in allen

Folgezeilen der Matrix zu erzeugen. Nach dem Ablauf dieser Methode befindet sich die Matrix in der gewünschten Dreiecksform, wie in Abbildung 16 gezeigt ist [13].

$$
\begin{array}{ccccccccccc}
1 & + & a_{12}x_2 & + & a_{13}x_3 & + & \ldots\ldots & + & a_{1n}x_n & = & b_1 \\
0 & + & 1 & + & a_{23}x_3 & + & \ldots\ldots & + & a_{2n}x_n & = & b_2 \\
0 & + & 0 & + & 1 & + & \ldots\ldots & + & a_{3n}x_n & = & b_3 \\
0 & + & 0 & + & 0 & + & \ldots\ldots & + & \vdots & & \vdots \\
\vdots & & \vdots & & \vdots & & & & \vdots & & \vdots \\
0 & + & 0 & + & 0 & + & \ldots\ldots & + & \ldots\ 1 & = & b_n
\end{array}
$$

Abbildung 16 Lineares Gleichungssystem in Dreiecksform

Die Rückwärtssubstitution zur Berechnung von x und das Speichern dieser Ergebnisse in y, kann durch den Beispielquellcode (in Chapel) aus Abbildung 17 durchgeführt werden. Die Eingabematrix U für diese Methode muss sich dafür in Dreiecksform befinden, während der Ergebnisvektor x mit Nullen initialisiert ist.

```
1  proc backSubstitution(U, x, y) {
2      var n: int = U.numElements;
3      for k in 0..n-1 by-1 { // for loop from n-1 to 0
4          x[k] = y[k];
5          forall i in 0..k-1 by-1 {
6              y[i] = y[i] - x[k] * U[i][k];
7          }
8      }
9  }
```

Abbildung 17 Rückwärtssubstitution in Chapel

4.2. Allgemeine Parallelisierungsmöglichkeiten

Die folgende Diskussion verschiedener Parallelisierungsmöglichkeiten basiert auf Grama [13], wurde jedoch zur Übersichtlichkeit und Anpassung an die Implementierung in den PGAS Sprachen modifiziert. Dabei handelt es sich um Lösungsansätze, welche in Hinblick auf eine Implementierung in MPI beschrieben worden sind. Da MPI länger existiert als die PGAS Sprachen und deutlich weiter entwickelt ist, kann hier auf eine umfangreiche Literatur zurückgegriffen werden, während diese bei den PGAS Sprachen noch fehlt. Deshalb wird versucht, die Algorithmen in der Parallelisierung mit MPI auf die PGAS Sprachen zu portieren und ggf. durch die zusätzlich zur Verfügung stehenden Konstrukte zu ergänzen oder zu ersetzen. „Prozesse" entsprechen in den weiteren Ausführungen, unabhängig von der

Programmiersprache, einzelnen Tasks, welche auf jeweils einem Knoten lokal abgearbeitet werden.

In den folgenden Betrachtungen beschränken wir uns auf die Berechnungen des Arrays. Der Vektor b und der Ergebnisvektor y müssen natürlich analog verteilt und berechnet werden. Als naiven Ansatz betrachten wir den Spezialfall, dass die zu berechnende Matrix nur so viele Zeilen enthält wie wir parallel laufende Prozesse erzeugen wollen bzw. uns in Form von Knoten zur Verfügung stehen. Dann kann jedem Prozess genau eine Zeile zugeordnet werden. Die aktuell zu berechnende Zeile, nennen wir die k-te Zeile.

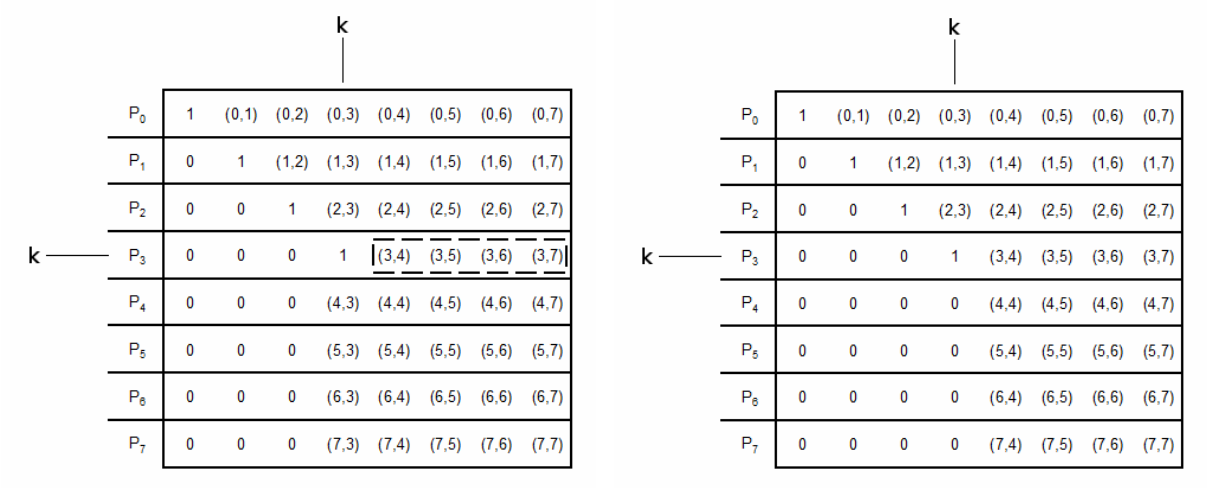


Abbildung 18 Links: Matrix nach Divisionsschritt k. Rechts: Matrix nach Eliminationsschritt k

In Abbildung 18 ist links diese Aufteilung zu sehen. Prozess 3 hat soeben die markierten Elemente der Zeile k durch das Element (k,k) geteilt, wodurch dieses zu einer 1 geworden ist. Dieser Berechnungsschritt konnte ohne Abhängigkeiten zu den anderen Zeilen und ohne Kommunikation zwischen den Prozessen durchgeführt werden. Nun benötigen die Prozesse, welche die Zeilen unterhalb von Zeile k berechnen sollen, die markierten Daten aus Zeile k. Also sendet Prozess 3 seine Zeile (oder nur die markierten Elemente) per Broadcast an die Prozesse k+1 bis n. Nachdem diese die Zeile empfangen haben, können sie durch je eine Multiplikation und eine Subtraktion mit deren entsprechenden Elementen eine Null auf den Positionen unterhalb von (k,k) wie in Abbildung 18 rechts zu erzeugen.

Anschließend wäre der nächste Prozess mit der k+1-ten Zeile an der Reihe. Diese einfache Methode der Parallelisierung wird als 1-D Partitionierung bezeichnet. Es existiert auch eine in dieser Ausarbeitung nicht verwendete 2-D Variante, in der die Prozesse als n x n Matrix analog zur Matrix *A* angeordnet werden, wobei jedem Prozess genau ein Element zugeordnet wird.

Bei der 1-D Parallelisierungsmethode fällt auf, dass die Prozesse `0` bis `k-1`, oberhalb der aktuell zu berechnenden Zeile, untätig sind und bis zum Ende des Algorithmus keine Arbeit mehr erhalten werden. Während die 1 der aktuellen Zeile `k` von ihrem Prozess berechnet wird, sind sogar alle anderen Prozesse untätig. Die einzige parallele Aktion ist die anschließende Berechnung der Folgezeilen durch die Prozesse `k+1` bis `n`. Neben den offensichtlichen Schwächen der 1-D Partitionierung, nämlich der Untätigkeit vieler Prozesse zu verschiedenen Zeitpunkten, kommt auch ein hoher Kommunikationsaufwand im Verhältnis zur Anzahl der Zeilen hinzu, da jedem Prozess nur eine Zeile zugeordnet wurde.

Die äußere Schleife der Implementierung aus Abbildung 15 wurde in der 1-D Partitionierung sequentiell abgearbeitet, so dass alle Prozesse an derselben Iteration arbeiteten. Die nächste Iteration wurde erst gestartet, wenn alle Prozesse mit ihren aktuellen Berechnungen fertig waren. Als verbesserte Variante dieses Ansatzes bietet sich die Pipelined- oder auch asynchrone Version des Gauß-Algorithmus an. Hierbei können die Prozesse bereits die nächste Iteration beginnen, während andere Prozesse noch an der aktuellen Iteration arbeiten. Dafür werden die Zeilen des Arrays zyklisch auf die Knoten verteilt, so dass bei 4 Prozessen das linke Bild aus Abbildung 19 entsteht. Dort ist bereits eine Verteilung einer größeren Anzahl von Arrayzeilen auf wenige Prozesse vorgesehen. Eine Blockverteilung, bei der Prozess `0` im Beispiel die ersten beiden Zeilen zugeordnet bekäme, hätte den Nachteil, dass nach der zweiten Iteration Prozess `0` für den Rest des Algorithmus keine Berechnungen mehr an seinen Zeilen durchführen müsste.

P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
P_1	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_2	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
P_3	0	0	0	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
P_0	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
P_1	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_2	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
P_3	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
P_1	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_2	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
P_3	0	0	0	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

Abbildung 19 Matrix zyklisch auf Prozesse verteilt (in zwei verschiedenen Ansichten)

Durch die zyklische Verteilung wechseln sich die Prozesse nun beim Divisionsschritt ab und haben dadurch immer in etwa die gleiche Anzahl an fertig gestellten Zeilen, wie im rechten

Bild von Abbildung 19 zu sehen ist und werden dadurch nicht vorzeitig untätig. Hierbei handelt es sich um die gleiche Verteilung der Zeilen auf die Prozesse, nur sind die Zeilen in dieser Ansicht nach Prozessen sortiert.

Jeder Prozess führt also in jeder Iteration entweder selbst den Divisionsschritt mit anschließendem, lokalem Eliminationsschritt seiner weiteren Zeilen durch, oder ist alternativ mit dem Eliminationsschritt aller unbearbeiteten Zeilen beschäftigt, nachdem er die aktuelle Zeile vom zugehörigen Prozess empfangen hat. Dabei fällt auf, dass der Prozess, welcher bei der nächsten Iteration mit dem Divisionsschritt an der Reihe ist, diesen prinzipiell schon durchführen kann noch bevor die anderen Prozesse mit dem Eliminationsschritt fertig sind. Dabei kann es passieren, dass dieser Prozess die neue Einserzeile an die Folgeprozesse senden will, während viele dieser Prozesse noch nicht in dieser Iteration angekommen sind.

Nachrichten mit der aktuellen Einserzeile werden bei der Pipelined Version des Gauß Algorithmus nicht mehr per Broadcast an alle Prozesse versendet, sondern nur an den Folgeprozess, welcher wie in Abbildung 19 rechts ersichtlich, die Zeilen unterhalb des vorigen Prozesses bearbeitet. Der letzte Prozess sendet hierbei seine berechneten Einserzeilen an den ersten Prozess, welcher dessen Nachfolger ist.

So erhält bei dieser Pipelined Version des Gauß Algorithmus der Prozess, welcher den nächsten Divisionsschritt durchführen wird und dessen verzögerte Berechnung direkte Auswirkungen auf die Laufzeit des Algorithmus haben wird, direkt die Zeile von seinem Vorgänger. Das führt in MPI zu kürzeren Wartezeiten, als wenn ein Prozess per Broadcast alle Folgeprozesse beliefern muss. Je nach Implementierung findet ein Broadcast in MPI über einen Baumalgorithmus statt, welcher die Nachricht an einzelne Prozesse sendet, welche diese wiederum an weitere Prozesse weitergeben [14]. Dieses Problem tritt in den PGAS Sprachen zwar nicht in gleicher Weise auf, da statt dem Broadcast Remote-Zugriffe von allen Knoten erfolgen, jedoch kann die Reihenfolge dieser Zugriffe kaum beeinflusst werden. Außerdem existiert der Aufwand des Broadcast auch in den PGAS Sprachen, auch wenn er dem Programmierer verborgen bleibt.

Jeder Prozess befindet sich in seinem eigenen, von den anderen Prozessen unabhängigen Iterationsschritt. Zu jeder Zeit ist ein Prozess (auch in den PGAS Sprachen) mit den folgenden Handlungsschritten beschäftigt:

- Ist eine der Zeilen des Prozesses in dieser Iteration mit dem Divisionsschritt an der Reihe, wird dieser berechnet, die Zeile an den Folgeprozess versendet und der Eliminationsschritt mit den weiteren Zeilen des Prozesses durchgeführt.
- Besitzt der Prozess in seiner eigenen Iteration nicht die Zeile mit dem Divisionsschritt, wartet der Prozess blockierend auf die aktuelle Einserzeile des Vorgängerprozesses. Wird diese empfangen, wird sie direkt an den Folgeprozess weitergeleitet und erst jetzt der Eliminationsschritt bei allen eigenen Zeilen auf Grundlage der empfangenen Einserzeile durchgeführt.

Die „Prozesse" dieser Beschreibung entsprechen, wie vorher definiert, einzelnen Tasks, welche einem Knoten zugeordnet wurden. Um mehrere Threads eines Computers und damit mehrere Prozessoren bei der Berechnung der Matrixzeilen nutzen zu können, kann die Berechnung der einzelnen Elemente einer Zeile als einzelne Tasks angesehen werden, welche lokal abgearbeitet werden können. Um den hohen Overhead zu vermeiden, welcher durch so viele kleine Tasks entstehen kann, reicht es bei sehr großen Arrays vermutlich, die Arrayzeilen, welche einem Knoten zugeteilt wurden, als Tasks zu definieren.

4.3. Implementierung in den einzelnen Sprachen

In den PGAS Sprachen können einfache Parallelisierungen oft mit wenigen Zeilen Quellcode realisiert werden. So können sogar Programmteile parallelisiert werden, die in den klassischen Programmiersprachen vermutlich seriell gelassen würden, weil der Programmieraufwand als unwirtschaftlich angesehen würde. Dazu kämen die hinzugekommenen Fehlerquellen und der zusätzliche Quellcode müsste später auch gewartet werden. Daher wollen wir in diesem Kapitel zusätzlich zur offensichtlich effektiveren Pipelined Version des Gauß Algorithmus auch die potentiell kürzere und einfachere Implementierung einer 1D Verteilung betrachten, welche gerade in den PGAS Sprachen kurz unter übersichtlich ausfallen sollte. Für letztere ist dann natürlich ebenfalls eine Verteilung von mehreren Matrixzeilen auf einzelne Knoten möglich.

4.3.1. Gaußsches Eliminationsverfahren in Chapel

Sowohl für die 1D Verteilung als auch für die Pipelined Version ist eine zeilenweise zyklische Verteilung des Arrays (die Vektoren x, y und b werden analog betrachtet) vorgesehen. Analog zur Verteilung von mehreren Zeilen auf Prozesse in MPI, werden wir in Chapel den einzelnen Locales Arrayzeilen zuordnen. Als Alternative zur im Kapitel 3.3.1 vorgestellten 2D Domainmap, welche für 2D Arrays laut Chapel Spezifikation vorgesehen ist, werden wir hier als praktische Alternative eine eindimensionale Range (vergleiche Kapitel 2.1), entsprechend der ersten Dimension des Arrays, als Domain Map verteilen. Damit werden zweidimensionale Arrays, welche in der ersten Dimension mit dieser Range definiert werden, zeilenweise auf Locales verteilt. Das hat den Vorteil, dass im Vergleich zur 2D Domain Map zusätzlich zum deutlich kürzeren Quellcode auch die einfachere, gewöhnlichere Syntax beim Zugriff auf das Array möglich ist. Allerdings bewegen wir uns hier außerhalb des in der Spezifikation vorgegebenen Implementierungsansatzes, welcher ausschließlich die Nutzung von 2D Domain Maps vorsieht.

Das Vorgehen ist in Abbildung 20 dargestellt. Die Range „AllLinesSpace" hat so viele Einträge, wie das Array der Matrix Zeilen hat und wurde mit zyklischer Verteilung definiert. Alle for-Schleifen, welche diese Range nutzen, werden ihre Iterationen auf den durch die Verteilung vorgegebenen Locales durchführen. Um Matrizen berechnen zu können, deren Ergebnisse bekannt sind, wurde ein lokales Array „matrixTemp" angelegt, welches später in das Verteilte Array kopiert wird. Das neue Array „matrix" wird nun in Zeile 3 mit zwei Dimensionen angelegt, wobei es sich bei der ersten um die zyklisch verteilte Range handelt. Mit den Vektoren b und y wurden analog verfahren. Obwohl die Syntax akzeptiert wird und das Array mit der Verteilten Range angelegt werden kann, gibt das lokal Statement bei dieser Vorgehensweise eine Warnung aus.

```
1  // a Range with cyclic distribution
2  const AllLinesSpace = [0..matrixTemp.numElements-1] dmapped Cyclic(startIdx=0);
3  var matrix: [AllLinesSpace][0..matrixTemp[0].numElements-1] real;
4  var b: [AllLinesSpace] real;
5  var y: [AllLinesSpace] real;
```

Abbildung 20 Erstellung von Arrays mit einer zyklisch verteilten Range

Da wie in Chapel üblich, jeder Locale auf alle Daten des verteilten Arrays zugreifen kann, kommt nun der vereinfachte Quellcode in Abbildung 21 zum Einsatz. Dieser wird von allen Locales parallel ausgeführt. Nehmen wir an, es existieren genau 4 Locales. In Zeile 1 wird

dann eine Range angelegt, welche von der ID des jeweiligen Locales in 4er Schritten bis zur Größe des Arrays läuft. Sie enthält damit die Indizes der Zeilen, welche durch die zyklische Verteilung dem Locale zugeordnet sind. Den Divisionsschritt führt nur ein Locale durch. Deshalb fragt jeder Locale in Zeile 5, ob die Zeile des Divisionsschritts in seiner Range vorhanden ist. Alle anderen Locales fahren mit dem Eliminationsschritt fort, müssen aber an einer mit sync Variablen realisierten Wall „Line Ready" in Zeile 16 warten bis diese, nach fertig gestelltem Divisionsschritt in Zeile 13, freigegeben wurde. Diese Synchronisation der Zugriffe ist nötig, da die Locales jederzeit direkt auf die aktuelle Zeile zugreifen können, die sie für den Eliminationsschritt benötigen, auch wenn diese noch nicht bearbeitet wurde.

```
1   var lokaleZeilenRange = here.id..n-1 by numLocales;
2
3   for k in 0..n-1 {
4       // division step
5       if (lokaleZeilenRange.member(k)) {
6           for j in k+1..n-1 {
7               matrix[k][j] /= matrix[k][k];
8           }
9           y[k] = b[k] / matrix[k][k];
10          matrix[k][k] = 1;
11
12          // end of division step - the line can be read
13          LineReady$[k] = true;
14      }
15
16      LineReady$[k];
17
18      // Elimination step
19      coforall i in lokaleZeilenRange[k+1..n-1] {
20          for j in k+1..n-1 {
21              matrix[i][j] -= matrix[i][k] * matrix[k][j];
22          }
23          b[i] -= matrix[i][k] * y[k];
24          matrix[i][k] = 0;
25      }
26  }
```

Abbildung 21 Paralleler Gauß Algorithmus

Die Broadcastvariante des Gauß Algorithmus kann so mit wenigen Modifikationen des Original Quelltextes implementiert werden. Die Nutzung einer Range in Zeile 1 erlaubt das Filtern nach den Iterationen der Schleifen, welche auf dem Locale überhaupt durchgeführt werden müssen und lässt die Schreibweise der Schleife von `0` bis `n-1` bestehen. Mit der späteren Rückwärtssubstitution wurde nur ein Locale beschäftigt, weil es sich hierbei um einen weitgehend sequentiellen Algorithmus handelt. Lediglich die for-Schleife wurde als forall-Schleife geschrieben, womit bei großen Arrays jede Zeile in mehrere Tasks zerlegt und parallel bearbeitet werden kann.

Sehen wir uns nun die Pipelined Variante des Gauß Algorithmus in Chapel an. Wir beschränken uns hierbei erneut auf den Teil des Quellcodes, welcher von allen Locales parallel gestartet wird. Aufgrund der Komplexität des Algorithmus betrachten wir erst nur den Divisionsschritt:

```
1   var lokaleZeilenRange = here.id..n-1 by numLocales;
2
3   for k in 0..n-1 {
4       // Division step (with local elimination and send)
5       if (lokaleZeilenRange.member(k)) {
6           coforall j in k+1..n-1 {
7               matrix[k][j] /= matrix[k][k];
8           }
9           y[k] = b[k] / matrix[k][k];
10          matrix[k][k] = 1;
11          // Send line A and y to the next locale, if this is not the last
12          if (k != n-1) {
13              if (numLocales > 1) then SyncA$[here.id] = true; // wait for sync...
14              // write message
15              LocaleDataMatrixLine[here.id] = matrix[k];
16              LocaleDataY[here.id] = y[k];
17              if (numLocales > 1) then SyncB$[here.id] = true; // say ready for read
                of the message
18              // elimination step
19              coforall i in lokaleZeilenRange[k+1..n-1] {
20                  writeln("L" + here.id + " eliminiere selber mit eigener Zeile " + i);
21                  for j in k+1..n-1 {
22                      // for every line below the k-th do...
23                      matrix[i][j] -= matrix[i][k] * matrix[k][j];
24                  }
25                  b[i] -= matrix[i][k] * y[k];
26                  matrix[i][k] = 0;
27              }
28          }
29          // end division step
```

Abbildung 22 Pipelined Version des Gauß Algorithmus: Divisionsschritt in Chapel

Als große Veränderung zur kurzen Broadcast Variante, müssen in der Pipelined Variante des Gauß Algorithmus nun ähnlich wie in MPI Nachrichten verschickt werden. Die Range hilft hier analog zum vorigen Algorithmus bei der Auswahl des Locales, welcher beim Divisionsschritt an der Reihe ist. Nach beendetem Divisionsschritt wird in den Zeilen 15-16 eine Nachricht in Form der Arrayzeile, sowie dem dazugehörigen y-Wert, lokal zwischengespeichert. Damit der empfangende Locale nicht zu früh diese Nachricht ausließt, aber der aktuelle Locale sich auch nicht durch erreichen der nächsten Iteration diese Nachricht zu früh mit der nächsten überschreibt, besteht eine wechselseitige Synchronisation zwischen den Locales. Diese wurde für jedes Paar Locales, welche aufeinander folgen, mittels zwei sync Variablen syncA$ und syncB$ realisiert. Bei der Variable syncA$ wartet ein Locale, bis der Speicher der Nachricht vom nächsten Locale ausgelesen wurde, was er dadurch erfährt, dass auf der Variable geschrieben werden kann (Zeile 13). Nach erfolgreichem Schreiben beschreibt

dieser nun in Zeile 17 die Variable syncB$, auf welche der folgende Locale blockierend versucht zu Lesen, bis er dadurch das Signal erhält, dass die Nachricht abrufbar ist. Nachdem die Nachricht versendet wurde, führt der Locale den Eliminationsschritt seiner eigenen Zeilen mit seiner lokalen Originalzeile durch.

Es folgt nun der Eliminationsschritt bei den Places, welche nicht den Divisionsschritt durchgeführt haben. Dabei empfängt ein Locale die Nachricht seines Vorgängers (Zeilen 9-11) und sendet diese gegebenenfalls weiter. Anschließend wird der Eliminationsschritt mit der empfangenen Nachricht durchgeführt.

```
1   } else {
2           // if this local has not the one-line, it maybe have to wait for the message
            "k"
3           if (k < lokaleZeilenRange.last) { // if I have to receive a message
4               // calculate predecessor
5               var meinVorgaengerID: int = here.id-1;
6               if (meinVorgaengerID == -1) then meinVorgaengerID = numLocales-1;
7               if (numLocales > 1) then SyncB$[meinVorgaengerID]; // wait for sync
8               //receive line
9               var myReceivedLocaleDataMatrixLine: [0..#n] real = LocaleDataMatrixLine[
                meinVorgaengerID];
10              var myReceivedY: real = LocaleDataY[meinVorgaengerID];
11              if (numLocales > 1) then SyncA$[meinVorgaengerID]; // wait for sync
12              // maybe send the message to the next locale
13              var senderID: int = k % numLocales;
14              var meinNachfolger: int = (here.id+1) % numLocales;
15              if (senderID != meinNachfolger && k != n-2) {
16                  if (numLocales > 1) then SyncA$[here.id] = true; // say ready to read
17                  // write message
18                  LocaleDataMatrixLine[here.id] = myReceivedLocaleDataMatrixLine;
19                  LocaleDataY[here.id] = myReceivedY;
20                  if (numLocales > 1) then SyncB$[here.id] = true; // say ready to read
21              }
22              // Eliminationsstep with received message
23                  coforall i in lokaleZeilenRange[k+1..n-1] {
24                      for j in k+1..n-1 {
25                          var matrixIK: real = matrix[i][k];
26                          var myReceivedJ: real = myReceivedLocaleDataMatrixLine[j];
27                          matrix[i][j] -= matrixIK * myReceivedJ;
28                      }
29                      b[i] -= matrix[i][k] * myReceivedY;
30                      matrix[i][k] = 0;
31                  }
32          }  // end of: if I have to receive a message
33      }
```

Abbildung 23Pipelined Version des Gauß Algorithmus: Eliminationsschritt in Chapel

4.3.2. Gaußsches Eliminationsverfahren in X10

In X10 gestaltet sich die Implementierung des Gaußschen Eliminationsverfahrens durch das Fehlen der zyklischen Verteilung als schwierig. So könnte die Aufteilung des Arrays analog zu einer MPI Implementierung komplett manuell erfolgen, worauf sich auch das Versenden von Arrayzeilen durch Synchronisation noch komplizierter gestalten würde, als in Chapel. Diese Art der Implementierung widerspricht aber dem PGAS Ansatz, mit verteiltem gemeinsamem Speicher zu Arbeiten.

Um möglichst ein praxisnahes Beispiel des Gauß Algorithmus zu erhalten, solange die zyklische Verteilung noch nicht zur Verfügung steht, wurde ein lokales Array angelegt, in welches die Zeilen der zu berechnenden Matrix systematisch in einer neuen Reihenfolge übertragen werden. Die Reihenfolge wurde so gewählt, dass die in X10 verwendete Blockverteilung genau die Zeilen des Arrays an jeden Place zuweißt, welche dieser bei einer zyklischen Verteilung des Originalarrays erhalten hätte. So konnte zwar eine zyklische Verteilung realisiert werden, jedoch gestaltete sich die Implementierung des Algorithmus durch das gemischte Array unnötig kompliziert, so dass eher die MPI artige Implementierung vorzuziehen wäre. Sehen wir uns deshalb die Broadcastvariante des Gauß Algorithmus in Abbildung 24 an, welche zwar ohne zyklische Verteilung implementiert wurde und damit nur lokale Parallelisierung mittels Aktivitäten aufweißt:

```
1  async for (k in 0..(n-1)) {     // for every line do...
2      finish {
3          for (j in (k+1)..(n-1)) {
4              A(k,j) /= A(k,k);   // Division step: whole line except
5                                  //  A(k,k) and zeros
6          }
7          y(k) = b(k) / A(k,k);       // Division step: b value
8          A(k,k) = 1;                 // don't divide A(k,k) by itself because of
9                                      //  rounding errors
10
11         for (i in (k+1)..(n-1)) {
12             async for (j in (k+1)..(n-1)) {
13                 // for every line below the k-th do...
14                 A(i,j) -= A(i,k) * A(k,j);  // Elimination step: whole line except
15                 //  A(k,k) and
                   zeros
16             }
17             b(i) -= A(i,k) * y(k); // Elimination step: b value
18             A(i,k) = 0;            // don't subtract to calculate the zero because
19             //  of rounding errors
20         }
21     }
22 }
```

Abbildung 24 Gauß Algorithmus parallel in X10

Jeder Place würde in Zeile 1 seine eigene Aktivität starten und die Zeilen analog zur Mandelbrotimplementierung filtern, welche ihm lokal zugeteilt wurden. Nach jeder Iteration der äußeren Schleife muss allerdings eine Synchronisation erfolgen, welche mit dem finish-Statement realisiert wurde. In Zeile 12 wird für jede Zeile, für die der Eliminierungsschritt durchgeführt werden muss ebenfalls eine eigene Aktivität gestartet. Die Synchronisation der Places untereinander könnte ebenfalls durch ein weiteres umgebendes finish-Statement erfolgen, da dieses nicht nur auf die Beendigung der eigenen Aktivitäten wartet, sondern auch auf die, welche innerhalb des Bodys auf weiteren Places gestartet wurden.

Die Pipelined Variante des Gauß Algorithmus kann mit dem Konstrukt der Clocks implementiert werden, mit der die Arbeitsschritte der Places synchronisiert werden können. Das hat einen gewissen Vorteil gegenüber der Chapel Version, weil eine schrittweise Synchronisation mittels sync Variablen entfällt und die Clocks mit nur wenig Zeilen implementiert werden können (siehe auch Anhang B). Leider kann durch die fehlende zyklische Verteilung zu diesem Zeitpunkt kaum eine effiziente Lösung dieser anspruchsvolleren Variante implementiert werden.

4.4. Bewertung der einzelnen Sprachen in Bezug auf die Implementierung des Gaußschen Eliminationsverfahrens

Mit der Verteilung des zweidimensionalen Arrays mittels Ranges, statt mit der in der Spezifikation dokumentierten 2D Domain Map, konnte in Chapel eine deutlich einfachere Implementierung erreicht werden. Die Auswirkungen auf das lokal-Statement sind schwer nachzuvollziehen und deuten auf evtl. Schwächen der Implementierung hin, welche bei der Verteilung laut Spezifikation nicht aufgetreten wären. Da die Sprache noch stark weiterentwickelt wird, könnten Kompromissgedanken zwischen Spezifikation und eigenen riskanten Tests in Zukunft wegfallen und für diesen Vergleich von PGAS Sprachen nicht mehr relevant sein.

Anders sieht es bei der Implementierung in X10 aus. Verschiedenartige Tests und Implementierungsansätze zeigen, dass zwar durch das Konzept der Clocks sehr sauberer Quellcode entsteht, jedoch stellt die fehlende zyklische Verteilung hier ein KO Kriterium dar. Tatsächlich ist zu diesem Zeitpunkt eine manuelle Implementierung der Arrayverteilung, wie sie analog in MPI durchgeführt werden müsste, vorzuziehen, worauf kaum noch Vorteile gegenüber dem Senden und Empfangen eines MPI Systems gefunden werden könnten.

5. Allgemeiner Vergleich

Programmierer, welche Chapel und X10 lernen, werden sicherlich Erfahrungen in anderen Programmiersprachen wie C, C++ und Java mitbringen. Da als Zielplattform der PGAS Sprachen neben Computerclustern auch Desktops und Laptops in Frage kommen, sind die Sprachen allerdings auch für Programmierer interessant, welche keine Erfahrungen im verteilten Rechnen mittels MPI oder der Thread-basierten Parallelisierung mittels OpenMP und pthreads mitbringen. Wir betrachten in diesem Vergleich der Sprachen zunächst die besonderen Eigenschaften der einzelnen Sprachen, wobei auch in dieser Ausarbeitung nicht genutzte Möglichkeiten erwähnt werden. Im Anschluss betrachten wir den Umfang von Handbüchern und Dokumentationen, welche über den Webauftritt der jeweiligen Sprache erhältlich sind um einschätzen zu können, wie aufwändig der Umstieg von einer klassischen Programmiersprache zu einer PGAS Sprache ausfällt.

5.1. Sprachkonstrukte und Möglichkeiten

Teilweise fehlen gängige Programmierfeatures, welche bei einer Programmiersprache die Klassen und abstrakte Parallelisierungskonstrukte vorausgesetzt werden könnten. So ist eine Modulodivison in Chapel nur mit Integerzahlen möglich, womit vom Programmierer ein performancetechnisch ineffizienter Workaround geschaffen werden muss. Während in Chapel der „sleep" Befehl, zum pausieren von Threads, nur mit ganzen Sekunden aufgerufen werden kann, fehlt dieser in X10 gänzlich.

Die Fehlersuche gestaltet sich bei beiden Sprachen aufgrund von sehr allgemein gehaltenen Fehlermeldungen oft als schwierig. In Chapel erscheint am häufigsten die Meldung „unresolved call", was allerdings zusammen mit der Zeilennummer die Suche örtlich einschränkt. In X10 treten verschiedene Fehlermeldungen auf, jedoch kommt es öfter zu Laufzeitfehlern, weil verteilte Arrays bzw. allgemein entfernte Daten leicht inkorrekt angesprochen werden können.

5.2. Handbücher und Dokumentationen

Für Chapel und X10 stehen sehr umfangreiche Spezifikationen zur Verfügung, welche durch Umfang und Qualität als Handbücher angesehen werden können. Diese werden in Abständen von etwa zwei Monaten aktualisiert wobei neue Features der aktuellen Compilerversionen und neue Kapitel zu bereits implementierten Spracheigenheiten aufgenommen werden. Darüber hinaus können verschiedene Paper und Präsentationsfolien eingesehen werden. Letztere sind teilweise als Tutorial aufgebaut und können einen Einstieg in die Sprachen erleichtern. Weitergehende Recherchen über Suchmaschinen wie google.de werden durch den Umstand erschwert, dass Chapel u. a. das englische Wort für „Kapelle" ist und der Name „X10" auch für ein Netzwerkprotokoll verwendet wird. Um die wichtigsten Konstrukte in der Vielfalt an Quellen mit deren eigenen Beispielen und Schwerpunkten an einem Ort zu zentrieren, wurde in Chapel sowie in X10 jeweils ein eigenes Handbuch geschrieben, welches in Anhang A bzw. Anhang B zu finden ist. Ohne diese Vorgehensweise wäre es kaum möglich gewesen, relevante Informationen später erneut aus der Spezifikation, den Präsentationsfolien, Papers und anderer pdf Dateien wieder zu finden.

Beide Sprachen werden mit Beispielquelltexten ausgeliefert, welche u.a. parallele Varianten von „hello world!" beinhalten. Diese Beispiele demonstrieren bereits, wie in wenig Codezeilen Tasks an andere Kerne verteilt und dort bearbeitet werden können.

Werden Chapels Spezifikation, Tutorials und mit dem Compiler mitgelieferten Beispielprogramme zusammengefasst als Dokumentation betrachtet, ist ein Einstieg in die Sprache, speziell zum Programmieren von kleinen, sequentiellen Programmen relativ leicht möglich. Problematisch ist die teilweise nicht intuitive Syntax. So war zum Beginn dieser Masterarbeit weder dokumentiert wie ein Array zweidimensional mit Zeilenumbrüchen ausgegeben werden kann, noch wie die mit den vorhandenen Mapping Techniken auf zweidimensionale Arrays angewendet werden können. Diese Probleme mussten mit try&error gelöst werden. Zum Zeitpunkt der Abgabe sind gerade bei Chapel neue Kapitel vorhanden, welche die Einarbeitung in die Handhabung verteilter Arrays deutlich erleichtern. Die aktuellen Dokumentationen stellen zwar die meisten Konstrukte vor, müssen aber für die gewünschten Anwendungsfälle anders genutzt werden, als die raren Beispielquellcodes es zeigen. So ist der Programmierer gezwungen, vieles selbst auszuprobieren und dabei die korrekte Syntax zu finden.

Die Spezifikation von X10 ist deutlich kürzer als die von Chapel und es fehlen ganze Kapitel. Der Umfang an Erklärungen und Beispielen für die einzelnen Konstrukte fällt dementsprechend weniger umfangreich aus.

Interessant ist, dass die Dokumentationen von Chapel und X10 in Beispielquelltexten Konstrukte verwenden, welche in dieser Weise nicht mehr verwendet werden. Ein identischer Fehler ist kurioserweise das Auftauchen der Möglichkeit, vom ersten Element eines zweidimensionalen Arrays die Länge der Arrayzeile mittels .length bzw. .size abzufragen, was in beiden Sprachen nicht mehr möglich ist. Fehlerhafte Beispiele können daher beim Programmierer Verwirrung hervorrufen. So halten sich bei X10 großgeschriebene und kleingeschriebene Namen von Konstanten die Waage, so dass kein einheitlicher Programmierstiel abgeleitet werden kann.

6. Zusammenfassung

In dieser Ausarbeitung wurden die Grundlagen der parallelen Programmierung beschrieben. Es wurde betont, dass die klassische Programmierung mittels MPI für die knotenübergreifende Kommunikation sowie OpenMP oder pthreads für die lokale Parallelisierung innerhalb eines Knotens kompliziert und fehleranfällig sind.

Das Gebiet der PGAS Sprachen beschäftigt sich mit der Vereinfachung des parallelen Programmierens. Dadurch soll das Programmieren gerade im verteilten Rechnen schneller und weniger fehleranfällig werden, so dass sich die parallele Programmierung auch unter Betrachtung betriebswirtschaftlicher Aspekte lohnt.

Die untersuchten PGAS Sprachen Chapel und X10 konnten das Gebiet der PGAS Sprachen gut repräsentieren, da deren Entwickler auf die Erfahrungen beim Programmieren in den anderen Sprachen zurückgreifen konnten. Interessant ist die sehr unterschiedliche Philosophie des Verteilens von Datenstrukturen auf mehrere Knoten. Chapel vereinfacht den Zugriff auf Daten anderer Locales entscheidend, indem ein direkter Zugriff möglich ist. Dies kann aber auch unabsichtlich passieren und der Programmierer gerät in eine Performancefalle. In X10 werden Daten von anderen Places standardmäßig kopiert. Das umgeht das Problem, bei Zugriffen auf ein entferntes Array jedes einzelne Element durch das Netzwerk anzufordern. Dafür muss mit deutlich mehr Quellcode und komplizierter Modifikation gerechnet werden.

Das Mandelbrotproblem ließ sich in beiden Sprachen, bis auf die Einschränkungen bei den Mappings, übersichtlich implementieren. Die Datenparallelisierung von Chapel konnte hier ihre volle Stärke zeigen, da in X10 die Kontrolle über die Parallelisierung innerhalb der Places und die Synchronisation der Aktivitäten vom Programmierer übernommen werden musste.

Bei der Implementierung des „Cowichan Problems" Gaußsche Elimination wurden zunächst die klassischen Programmieransätze betrachtet, die eine Implementierung in MPI annehmen. Da diese Implementierungsvorschläge gut erforscht waren, bildeten sie eine gute Ausgangsbasis, um die Implementierung der Algorithmen in den PGAS Sprachen durchzuführen. Überraschend war die fehlende Implementierung gängiger Varianten der „blockzyklischen Verteilung" in Chapel, womit eine eigene, etwas komplizierte Verteilung implementiert werden musste, um die gewünschte Datenverteilung zu erreichen. In X10 war

nur die Blockverteilung lauffähig, bei die zyklische und blockzyklische Verteilung gänzlich fehlten und analog zu Chapel bei Bedarf eigene Implementierungen erforderlich sind.

Das Gaußsche Eliminationsverfahren ist in seinen parallelen Formulierungen anspruchsvoll und kompliziert zu implementieren. Während in den PGAS Sprachen für viele Standardprobleme kurze und schnelle Lösungen zur Verfügung stehen, bieten sie für diesen speziellen Algorithmus auf den ersten Blick weniger Unterstützung an. Allerdings kann in Chapel eine gegebene sequentielle Lösung des Gaußschen Eliminationsverfahrens mit nur wenigen Modifikationen in die parallele Formulierung „1-D Partitionierung" umgewandelt werden. So ist der ursprüngliche Quelltext weiterhin gut lesbar und nicht schwerer zu verstehen als ohne diese Modifikationen. Das stellt einen riesigen Vorteil gegenüber einer MPI Implementierung dar, welche das Verteilen der Daten und die Kommunikation zwischen den Knoten als deutlich umfangreichere Programmierarbeit beinhalten würde. Die Möglichkeit, Daten direkt von anderen Knoten anfordern zu können, erforderte eine Synchronisation der gegenseitigen Zugriffe. Diese Vorgehensweise wäre bei MPI durch das eigentlich umständlich zu programmierende Senden und Empfangen von Nachrichten, was in Chapel transparent geschieht, nicht nötig gewesen.

Chapel kann durch Datenparallelisierung oft den sequentiellen Originalquellcode erhalten und mit nur wenigen Modifikationen die Daten auf mehreren Locales ablegen. Entsprechend markierte Schleifen werden automatisch so zerlegt, dass die Locales möglichst nur die Daten brauchen, welche ihnen lokal vorliegen. Zusätzlich wird die Schleife in Tasks zerlegt, so dass diese auch gleich mit mehreren Prozessoren abgearbeitet wird. Datenparallelisierung beherrscht also lokale und verteilte Parallelisierung in einem Schritt. Dieses Konzept steht in X10 nicht zur Verfügung, so dass parallele Operationen an Arrays weiterhin manuell implementiert werden müssen. Kann ein Problem allerdings durch komplizierte Abhängigkeiten durch die Standardverteilungen zerlegt werden, muss dies auch in PGAS wie in MPI manuell programmiert werden. Teilweise helfen in Chapel dann die dynamisch erstellbaren Ranges, wobei das analoge Konstrukt der Regionen in X10, mit der Einschränkung nur aufeinander folgende Indizes aufzunehmen, selten geeignet ist.

Benchmarks, die den Speedup der Sprachen anhand von einzelnen Beispielen vergleichen könnten, wären wenig sinnvoll gewesen, da bereits bei den Standardverteilungen eigene evtl. ineffiziente Workarounds geschaffen wurden, die einen fairen Vergleich der Geschwindigkeit der Programme ausschlossen.

Die Aufgabe Datenabhängigkeiten und Möglichkeiten von Lokalität auszunutzen liegt nach wie vor beim Programmierer. Damit fallen bei der Definition von einzelnen Tasks die gleichen Überlegungen an, welche auch bei der klassischen verteilten Programmierung nötigt sind. Jedoch ist die Formulierung in den PGAS Sprachen um ein Vielfaches kürzer und einfacher gehalten. Deshalb können in den untersuchten Sprachen Programmteile parallelisiert werden, welche in MPI evtl. wegen zu großem Aufwand nicht verändert wüden. In Chapel und X10 handelt es sich dabei oft nur um minimale Modifikationen des Quellcodes. Um bei der Programmierung möglichst produktiv zu sein, ist vielleicht auch nicht immer der effizienteste Algorithmus nötig, dessen Parallelisierung selbst in den PGAS Sprachen ähnlich kompliziert ausfällt, wie eine Implementierung in MPI. Eine einfache und damit besser wartbare Lösung, welche in den PGAS Sprachen leicht zu implementieren ist und ebenfalls eine gute Parallelisierung erlaubt, kann eine gute Alternative sein.

7. Literaturverzeichnis

[1] Elektronik Kompendium, http://www.elektronik-kompendium.de/sites/com/1203171.htm

[2] V. Saraswat, B. Bloom, X10 Language Specification Version 2.2 S. 13, http://x10.sourceforge.net/documentation/languagespec/x10-222.pdf

[3] University Amsterdam, http://www.cs.vu.nl/~bal/cowichan.html

[4] Fohry, Vorlesung Parallelverarbeitung 1 WS2011 - Uni-Kassel Folie 35

[5] Gorlatch, Vorlesung Parallele Systeme 2006 Folie 5, http://pvs.uni-muenster.de/pvs/lehre/SS06/ps/folien/3-06Print.pdf

[6] Fohry, Vorlesung Parallelverarbeitung 1 WS2011 - Uni-Kassel Folien 93ff

[7] Cray Inc., Chapel Specification, http://chapel.cray.com/spec/spec-0.91.pdf

[8] Cray Inc., Chapel Specification/Configuration Constants, http://chapel.cray.com/spec/spec-0.91.pdf S. 219

[9] Mandelbrot Menge auf Wikipedia, http://de.wikipedia.org/wiki/Mandelbrot-Menge

[10] Chapel bugs-report, http://sourceforge.net/mailarchive/forum.php?forum_name=chapel-bugs

[11] „Introduction to Concurrency in Programming Languages, Sottile/Mattson

[12] Dankert/Dankert, Der Gaußsche Algorithmus, http://www.rzbt.haw-hamburg.de/dankert/WWWErgVert/html/der_gausssche_algorithmus.html

[13] A. Grama, A. Gupta , Introduction to Parallel Computing (Second Edition), S. 252ff, Addison Wesley (2003)

[14] TU Chemnitz, MPI Broadcast, http://www.tu-chemnitz.de/informatik/RA/projects/mpihelp/bcast.html

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Hofgeismar, den 26.08.2012

_____

Nikolas Luke

Anhang A

Eigenes Chapel Handbuch

„Hello World"

Erstelle Datei hello.chpl mit folgendem Inhalt:
writeln("hello, world");

Ausgaben (write und writeln)

var zahl: int = 5;
writeln("Zahl: ", zahl);

Compilen und ausführen

> chpl hello.chpl
> ./a.out
hello, world
>

Weitere Compiler Optionen

./a.out --numLocales=8
./a.out -nl 8

// höchstens I threads auf jedem Locale. Default = 0, wodurch es vom System festgelegt wird (Anzahl CPUs)
--numThreadsPerLocale=<i>
--numThreadsPerLocale=0

Comments

// Einzeiliges Kommentar
/* zweizeiliges
Kommentar */

Primitive Typen

Type	Default size	Other sizes	Default init
bool	impl. dep.	8, 16, 32, 64	false
int	32	8, 16, 64	0
uint	32	8, 16, 64	0
real	64	32, 128	0.0
imag	64	32, 128	0.0i
complex	128	64, 256	0.0+0.0i
String	Variable		""

Variables, Konstanten and Configuration

var x: **real** = 3.14; variable of type real set to 3.14
var isSet: **bool**; variable of type bool set to false
var z = -2.0i; variable of type imag set to -2.0i
const epsilon: **real** = 0.01; runtime constant
param debug: **bool** = **false**; compile-time constant
config const n: **int** = 100; > ./a.out --n=4
config param d: **int** = 4; > chpl -sd=3 x.chpl

weitere Beispiele:
config paramintSize= 32;
config type elementType= real(32);
config const epsilon = 0.01:elementType;
config var start = 1:int(intSize);

% chplmyProgram.chpl -sintSize=64
-selementType=real
% a.out --start=2 --epsilon=0.00001

Modules

module M1 { code; } module definition
module M2 {
use M1; module use
proc main() { body(); } main definition
}

% chplM1.chpl M2.chpl --main-module M1

Zuweisungen

Einfache Zuweisung: =
Kurzschreibweisen mit Rechenschritt:
+= -= *= /= %= **=
&&= ||= &= |= <<= >>=
Swap: <=>

Casts

var i: int = 2.0:int; cast real to int
var x: real = 2; coerce int to real

If-Anweisungen

if cond then stmt1(); else stmt2(); // nur ein Befehl
if cond { stmt1(); } else { stmt2(); } // mehrere Bef.
var half = if i%2 then i/2+1 else i/2;

Switch-Anweisung

select expr {
when equiv1 **do** stmt1(); // nur ein Befehl
when equiv2 { stmt2(); } // mehrere Befehle
otherwise stmt3();
}

type select actual {
when type1 **do** stmt1();
when type2 { stmt2(); }
otherwise stmt3();
}

Schleifen

```
while condition { … }
do { … } while condition;
writeln(for i in 1..n do i**2);
for index in aggregate do …;
for index in aggregate { … }

varA: [1..3] string= (" DO", " RE", " MI");
for i in 1..3 {write(A(i)); } // DO RE MI
for a in A { a += "LA"; } write(A); // DOLA RELA
MILA

label outer for …
break; or break outer;
continue; or continue outer;
```

Für for-Schleifen kann die Syntax von Ranges auch direkt genutzt werden. Hier zum Beispiel eine rückwärts zählende Schleife:
for t in 0..4 by-1 do write(t);
Ausgabe: 43210

```
// Mit mehreren Zählern:
forall (a, (y, x), t) in (myArray, myArray.domain,
debugDistArray) { …
```

do Statement für einzelne Anweisungen

```
while cond do
    compute();

for indices in iteratable-expr do
    compute();

select key {
when value1 do compute1();
when value2 do compute2();
otherwise do compute3();
}
```

Zipper Iteration

Über alle angegebenen Indeces paarweise (auch trippelweise etc)

```
varA: [0..9] real;
for (i,j,a) in (1..10, 2..20 by2, A) do
    a = j + i/10.0;writeln(A);

// 2.1 4.2 6.3 8.4 10.5 12.6 14.7 16.8 18.9 21.0
```

Procedures

```
proc bar(r: real, i: imag): complex {
        var c: complex = r + i;
        return c;
}

proc writeCoord(x: real= 0.0, y: real= 0.0) {
    writeln((x,y));
}

proc arrayUebergabe(A: []) {…

writeCoord(2.0); // (2.0, 0.0)
writeCoord(y=2.0); // (0.0, 2.0)
writeCoord(y=2.0, 3.0); // (3.0, 2.0)

proc foo(i) return i**2 + i + 1;
```

Argumente und Returntypen

Arguments can optionally be given intents
- (blank): varies with type; follows principle of least surprise
  - most types: const
  - arrays, domains, sync vars: passed by reference
- const: disallows modification of the formal
- in: copies actual into formal at start; permits modifications
- out: copies formal into actual at procedure return
- inout: does both of the above
- param/type: formal must be a param/type (evaluated at compile-time)

- **Return types can also have intents**
- (blank)/const: cannot be modified (without assigning to a variable)
- var: permits modification back at the callsite
- type: returns a type (evalutedat compile-time)
- param: returns a paramvalue (evaluated at compile-time)

Ausdrücke und Assoziativität

(alle nichtgekennzeichneten Ausdrücke sind links-assoziativ)

Operators	Uses
. () []	member access, call and index
new (right)	constructor call
:	cast
****** (right)	exponentiation
reduce scan dmapped	reduction, scan, apply domain map
! ~ (right)	logical and bitwise negation
*** / %**	multiplication, division, modulus
unary **+ -** (right)	positive identity, negation
+ -	addition, subtraction
<< >>	shift left, shift right
<= >= < >	ordered comparison
== !=	equality comparison
&	bitwise/logical and
^	bitwise/logical xor
|	bitwise/logical or
&&	short-circuiting logical and
||	short-circuiting logical or
..	range construction
in	loop expression
by #	range/domain stride and count

if forall [for sync single	conditional expression, parallel iterator expression, serial iterator expression, synchronization type
,	comma separated expression

Mathamatische Ausdrücke

x**y x hoch y

Formal Argument Intents

Intent	Semantics
in	copied in
out	copied out
inout	copied in and out
blank	formal arguments are constant except arrays, domains, syncs are passed by reference

Named Formal Arguments

```
proc foo(arg1: int, arg2: real) { … }
foo(arg2=3.14, arg1=2);
```

Default Values for Formal Arguments

```
proc foo(arg1: int, arg2: real = 3.14);
foo(2);
```

Records

```
record Point { record definition
        var x, y: real; declaring fields
}
var p: Point; record instance
writeln(sqrt(p.x**2+p.y**2)); field accesses
p = new Point(1.0, 1.0); assignment
```

Classes

```
class Circle {
        var p: Point;
        var r: real;
}

var c = new Circle(r=2.0); class construction
proc Circle.area() method definition
        return 3.14159*r**2;
writeln(c.area()); method call
class Oval: Circle { inheritance
        var r2: real;
}
proc Oval.area() method override
        return 3.14159*r*r2;
delete c; free memory
c = new Oval(r=1.0,r2=2.0); polymorphism
writeln(c.area()); dynamic dispatch
```

Eigener Konstruktor:

```
class MessagePoint {
        var x, y: real;
        proc MessagePoint(x: real, y: real) {
                this.x = x;
                this.y = y;
        this.message = "a point";
}
```

Unions

```
union U { union definition
        var i: int; alternatives
        var r: real;
}
```

Tuples

```
var pair: (string, real); heterogeneous tuple
var coord: 2*int; homogeneous tuple
pair = ("one", 2.0); tuple assignment
(s, r) = pair; destructuring
coord(2) = 1; tuple indexing

var coord: (int, int, int) = (1, 2, 3);
var coordCopy: 3*int= coord;
var (i1, i2, i3) = coord;
var triple: (int, string, real) = (7, "eight", 9.0);
```

Enumerated Types

```
enum day {sun,mon,tue,wed,thu,fri,sat};
var today: day = day.fri;
```

Ranges

```
var every: range = 0..n;  // range definition
var evens = every by 2; // strided range
var R = evens # 5; // counted range
var odds = evens align 1; // aligned range

1..6 // 1, 2, 3, 4, 5, 6
6..1 // empty
3.. // 3, 4, 5, 6, 7, …
1..6 by2 // 1, 3, 5
1..6 by-1 // 6, 5, 4, …, 1
1..6 #4 // 1, 2, 3, 4
1..6[3..] // 3, 4, 5, 6

1.. by 2 // 1, 3, 5, …
1.. by 2 #3 // 1, 3, 5
1.. #3 by 2 // 1, 3
0..#n // 0, …, n-1
```

Das # Zeichen zeigt die Anzahl, an Indices an, welche generiert werden!

```
[0..#4] // 0,1,2,3
[1..#4] // 1,2,3,4
[6..#4] // 6,7,8,9
```

Es kann fast jede Methode, welche für Domains implementiert und dokumentiert wurde auch auf Ranges angewendet werden, auch wenn das nicht dokumentiert ist. Z.B.

```
myRange.first // erster Index
```

myRange.high // letzer Index
meineRange.member(3) // ist 3 Index in meineRange

Domains and Arrays

var D: **domain**(1) = [1..n]; // domain
var A: [D] **real**; array
var Set: **domain**(**int**); associative domain
Set += 3; add index to domain
var SD: **sparse subdomain**(D); sparse domain

var A: [0..9] real;
A[0] = 4;
var A: [1..3] int = (5, 3, 9); // 1D direct instanzieren
var A: [1..2][1..2] int = ((5, 3),(7, 8)); // 2D direkt
B: [1..3, 1..5] real, // 2D array of reals
C: [1..3][1..5] real; // array of arrays of reals

Kopieren eines Arrays: Eine Zuweisung ist nur eine Kopie
var array2 = array1;

Kopieren von Teilen eines Arrays (Beispiel das Locales-Array):
var ErsteLocalesArray = Locales[0..1];
var RestLocalesArray = Locales[2..numLocales-1];

// Umkopieren von eindimensionalem Array zu zweidimensionalem Array:
varMeinGridIn2D = Locales.reshape([1..2, 1..4]);
Macht aus L0, L1, L2, L3, L4, L5, L6, L7
Ein Grid mit L0, L1, L2, L3,
L4, L5, L6, L7

A.numElements // Größe des Arrays
A.domain.numIndices // funktioniert evtl auch
A.domain.low // erster Index des Arrays
A.domain.member(4) // ist 4 ein Index von A?

Zugriff auf die Indices von Domains:
for i in A.domain.dim(1) do
for j in A.domain.dim(2) do
writeln(i, " ", j);

Arrays aus nicht zusammen hängenden Ranges erstellen:
var meineRange = 0..10 by 2;
for i in meineRange do write(i + " "); // 0 2 4 6 8 10
var A: [meineRange] int;
for i in A.domain do write(i + " "); // 0 2 4 6 8 10
write("numElements: " + A.numElements); // 6
Nun nur auf die Indices zugreifen, welche von 0-5 liegen:
for i in meineRange[0..5] do writeln(A[i]);
Auch offene Teilmenge ist möglich. Zum Beispiel vom Index 5 bis zum Ende des Arrays (man beachte, dass es den Index 5 nicht gibt):
for i in meineRange[5..] do writeln(i); // 6 8 10

myDomain.first // erster Index
myDomain.high // letzer Index

Domain Maps

Domainmaps werden verwendet, um Datenstrukturen wie Arrays aufzuteilen, so dass später auch eine Parallelisierung in der gewünschten Weise (Auf locales verteilt in block, zyklischer, block-zyklischer oder dynamischer Verteilung oder nur auf einem locale, wie viele Tasks etc.
Ohne spezifizierte Domain Map ist als default eingestellt, dass der aktuelle local alle Indices und Werte besitzt, womit auch nur lokal gerechnet wird.

use BlockDist;
config const numIters= 20;
const WorkSpace= [1..numIters] dmapped
Block([1..numIters]);

proc main() {

forall i in WorkSpace do

writeln("Hello, world! ",
"from iteration ", i, " of ",
numIters,
" on locale ", here.id, " of ",
numLocales);
}

// Ausgabe:
1-5 auf locale 0
6-10 auf locale 1
11-15 auf locale 2
6-17 auf locale 3

Der zweite Adressspace, wecher Block übergeben wird, ist die Range, welche Blockverteilt wird. Alles außerhalb des Blocks wird ein und demselben Locale gegeben, da wir mit „forall" alle Indices abklappern und damit auch die, die nicht als Block Verteilung definiert sind.

ODER

const ProblemSpace = [1..m] **"Verteilung"**;

var A: [ProblemSpace] real;

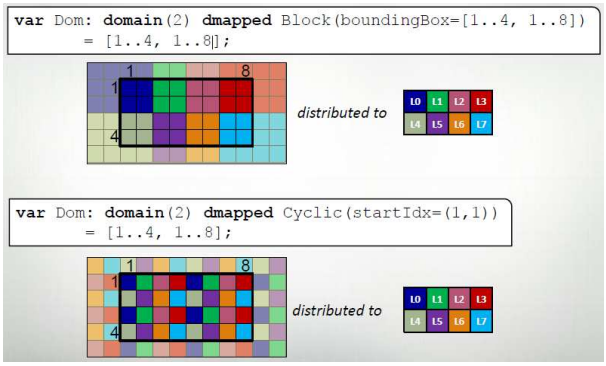
A = B + alpha * C;

Als Verteilungen (auf 4 Locales) kommen in Frage:

Block
use BlockDist;
dmapped Block(boundingBox=[1..m]);
Teilt Array bzw ProblemSpace in 4 große zusammenhängende Blöcke auf

Cyclic
use CyclicDist;
dmapped Cyclic(startIdx=1);
Teilt Array in einzelne(!!!) Indices auf:
01230123012301230123…

BlockCyclic
use BlockCycDist;
const Space = [1..8, 1..8];
dmapped BlockCyclic(startIdx=Space.low,
blocksize=(2,3)) = Space;
2D Version:



Jedes Array wird über die Domain gemappt, über welches es deklariert wurde.

Beispiel:

use BlockDist;
var Dom: domain(2) dmapped Block([1..5,1..6]) = [1..5,1..6];
var MyArray: [Dom] real;

Hier wurde das Array MyArray mit der Blockverteilung von Dom angelegt und später auch so gemappt.

Wie kommt man an den Index eines Elements während einer for-Schleife?

use BlockCycDist;
const Space = [1..8, 1..8];
const D: domain(2) dmapped
BlockCyclic(startIdx=Space.low, blocksize=(2,3)) = Space;

var A: [D] real;
forall (a, (x, y)) in (A, D) do
a = x + y/10.0; // Index wird eingetragen

writeln(A);

Wie kann ich mein mit einer Domain erstelltes 2DArray ohne writeln(A) ausgeben?

```
for i in D.dim(1) {
        for j in D.dim(2) {
                write(A[i, j] + " ");
        }
        writeln();
}
```

Replicated
use replicatedDist;
Eine mögliche Verteilung. Implementiert aber noch nicht dokumentiert…

Es ist auch möglich Domainmaps unabhängig von einer Domain einzeln zu erzeugen. Man könnte dann viele Domains mit der selben Map erzeugen (siehe Specification).

var B = **new dmap**(**new** Block([1..n]))**;** block distribution
var D: **domain**(1) **dmapped** B; distributed domain
var A: [D] **real**; distributed array
var D2: **domain**(1) **dmapped** Block([1..n]); domain map sugar

Data Parallelism

forall
Wenn eine sehr große Schleife mit sehr viel weniger Tasks ausgeführt werden soll. Es bringt ja nichts für eine Schleife bis 50000 einfach mal 50000 kleine Mini-Tasks "aufzumachen".

forall i **in** D **do** A(i) = 1.0; domain iteration
[i **in** D] A(i) = 1.0; "
forall a **in** A **do** a = 1.0; array iteration
[a **in** A] a = 1.0; "
A = 1.0; array assignment

Reductions and Scans

Pre-defined: + * & | ^ && || min max minloc maxloc
var sum = + **reduce** A; 1 2 3 => 6
var pre = + **scan** A; 1 2 3 => 1 3 6
var ml = minloc **reduce** (A, A.**domain**);

Iterators

iter squares(n: **int**) { serial iterator
for i **in** 1..n **do**
yield i**2; yield statement
}
for s **in** squares(n) **do** …; iterate over iterator

Task Parallelism

begin
"Feuern und vergessen": Der in begin gestartete Task wird in einem eigenen Thread gestartet und es geht im original Thread auch direkt weiter

begin writeln("hello1");
writeln("hello2");

// Ausgabe:
// hello1 oder hello2
// hello2 hello1

cobegin

Erstellt für jedes Statement im Body einen eigenen Task. Hinter dem Body wird mit der Programmausführung gewartet, bis diese Tasks abgeschlossen sind (aber auf keine geschachtelten, mit „begin" gestarteten Tasks. Das würde „sync" tun). Eignet sich für viele, sehr unterschiedliche Tasks oder sehr wenige, gleiche.

```
cobegin {
    task1();
    task2();
} // Warte hier, bis alle Tasks fertig sind
```

coforall
Erstelle für wirklich **jede** Iteration einen eigenen Task. Wartet bis alle Tasks abgeschlossen sind. Ist die Schleife zu lang, sollte man wegen dem Overhead über „**forall**" nachdenken. Einsatzgebiet ist eher, wenn jede Iteration ein beträchtlicher, langer Arbeitsaufwand ist.

```
coforall i in aggregate do task(i);

var numTasks: int = here.numCores; // z.B. 4
coforall t in 0..#numTasks do {
    writeln("Hello from task ", t, " of ",
    numTasks);
} // warte hier, bis alle Tasks fertig sind
writeln("All tasks done");

// Hello from task 2 of 4
// Hello from task 0 of 4
// Hello from task 3 of 4
// Hello from task 1 of 4
// All tasks done
```

serial
Wertet ersten Ausdruck aus und führt den Anschließenden Befehl oder Body aus. Verhindert dabei jeden dynamisch auftretenden gleichzeitigen Zugriff

```
serial condition do stmt();

proc search(N: TreeNode, depth = 0) {
    if (N != nil) then
        serial (depth > 4) do cobegin{
            search(N.left, depth+1);
            search(N.right, depth+1);
    }
}
search(root);
```

sync
Wartet nach der Folgenden Anweisung bzw Block darauf, dass selbst die dynamischen, mit begin gestarteten Tasks (auch verschachtelt aufgerufene), abgeschlossen sind. Gibt es keine mit „begin" gestarteten Tasks, verhält es sich praktisch wie „cobegin".

```
sync { begin task1(); begin task2(); }

sync{
    for I in1..numConsumers {
        begin consumer(i);
    }
    producer();
}


proc search(N: TreeNode) {
    if (N != nil) {
        begin search(N.left);
        begin search(N.right);
    }
}
sync{ search(root); }
```

Synchronization Variables (sync und single)
sync und single Variablen werden in Chapel nach Konvention mit einem abschließenden $ Zeichen benannt. Dadurch wird der Programmierer u. a. daran erinnert, dass es zu Deadlocks kommen kann.
Das „normale" Lesen auf einer syncronisierten Variablen ist nur möglich, wenn deren Status „voll" ist. Das „normale" Schreiben ist nur möglich, wenn der Status der Variable „leer" ist.
Wird eine sync-Variable gelesen, ist ihr Status anschließend „leer". Wird eine single-Variable gelesen, ändert sich ihr Status nicht.
Wird auf einer sync oder single-Variablen geschrieben, wechselt der Status immer auf voll. Single Variablen werden gewöhnlich nur einmalig beschrieben, weil dessen Status dann sowieso für immer auf „voll" bleibt.
Sync und single sind „nur" auf alle primitive Datentypen (außer strings und complex) anwendbar: bool, int, uint, real, imag locale
Versucht ein Task eine Variable zu lesen oder zu beschreiben, welche nicht im richtigen Status dafür ist, reiht er sich blockierend in eine Warteliste ein. Ist die Variable wieder im richtigen Status, wird nicht-deterministisch einer der wartenden Tasks zum weiterarbeiten ausgesucht, während die anderen weiter warten müssen.

```
var count$: sync int = 0;
begin count$ = count$ + 1;
begin count$ = count$ + 1;
begin count$ = count$ + 1;
```

Normalerweise wäre die Incrementierung nicht sicher. Da es aber eine sync Variable ist, welche nach dem Lesen den Status auf leer setzt, ist das Schreiben anschließend sicher. Vom Effekt her sind die Zuweisungen dadurch atomic, weil nicht zwei Tasks in folge von der Variablen lesen können.

```
var count$: sync int = n; // Zähler
var release$: single bool; // Barriere
forall t in 1..n do begin {
      work(t);
      var myc = count$; // count wird empty
      if myc!=1 {
            write(".");
            count$ = myc-1; // count wird full
            release$; // Alle Tasks blockieren hier
            beim Versuch die Variable zu lesen
      } else {
      release$ = true; // Beschreibt die single
      Variable
      writeln("done");
      }
}
```

der letzte Task betritt den else-Fall und lässt alle anderen Tasks dadurch frei, dass die single Variable release das erste mal beschrieben wurde und nun alle anderen lesen können.

Sync und single Variablen können als Formale Parameter übergeben werden. In dem Fall ist die Übergabe by reference

```
            1) var lock$: sync bool;
lock$ = true; lock$ = true; //fill lock
critical1(); critical2();
lock$; lock$; //empty lock
            2) var data$: sync int;
data$ = produce1(); consume(data$);
data$ = produce2(); consume(data$);
            3) var go$: single real;
go$=set(); use1(go$); use2(go$);

// critical sections
var lock$: sync bool;
lock$ = true;
critical();
var lockval = lock$; // lock$; müsste auch gehen

// futures
var future$: sync real;
begin future$ = compute();
computeSomethingElse();
useComputedResults(future$);
```

Single Variables

```
var future$: single real;

begin future$ = compute();
begin computeSomethingElse(future$);
begin computeSomethingElse(future$);
```

Bounded Buffer Producer/Consumer Example

```
var buff$: [0..#buffersize] sync real;
cobegin {
      producer();
      consumer();
}

proc producer() {
      var i= 0;
      for … {
            i= (i+1) % buffersize;
            buff$(i) = …;
      }
}

proc consumer() {
      var i= 0;
      while …{
            i= (i+1) % buffersize;
            …buff$(i)…;
      }
}
```

Locales

1) Built-in Constants (welche genutzt werden können):

```
config const numLocales: int; //set via -nl
config const numLocales: int = …;

// Locales: L0, L1, L2,…L(n-1)
const Locales: [LocaleSpace] locale;
const Locales: [0..numLocales] locale = ...;
const LocaleSpace: domain(1) = [0..numLocales-
1];
```

2) Locale Prozeduren

Here zeigt immer auf den locale, auf welchem der Task läuft:

```
writeln(here.id); // Ausgabe: 0
onLocales(1) do
      writeln(here.id); // Ausgabe: 1
```

Zugriff mit here.id oder meineLocaleVariable.id

here.id	0 (der Index in LocaleSpace)
here.name	SLATERLAPTOP
here.numCores	4
here.callStackSize	0

Argument 1: Einheit
Argument 2: Rückgabetyp

```
here.physicalMemory(MemUnits.Bytes, int(64))
here.physicalMemory(MemUnits.KB, int(64))
here.physicalMemory(MemUnits.MB, int(64))
here.physicalMemory(MemUnits.GB, int(64))

consttotalPhysicalMemory=
      + reduceLocales.physicalMemory();
```

Multi-Locale HelloWorld:
Ob coforall oder forall eingesetzt wird, macht keinen Unterschied. Die „do" Schlüsselwörter sagen nur, dass nur ein Befehl und keine Sequenz im Body kommt. Das „on" sagt, auf welchem Locale der folgende Befehl oder Block ausgeführt werden soll. Die Variable numLocales gibt die Größe des eingebauten Locales Arrays zurück.

```
coforall loc in Locales do
        on loc do
        writeln("Hello, world!",
              "from node ", loc.id, " of ",
              numLocales);

// Ausgabe (beispielsweise):
// Hello, world! from node 0 of 4
// Hello, world! from node 2 of 4
// Hello, world! from node 1 of 4
// Hello, world! from node 3 of 4
```

On Statement
Mit dem "on" Statement kann der Locale angegeben werden, auf dem der Folgebefehl ausgeführt werden soll. Dabei wird keinerlei automatische Parallelisierung durchgeführt.

```
writeln("L", here.id, ": nur auf locale 0");
on Locales[1] do
        writeln("L", here.id, ": jetzt nur locale 1");
writeln("L", here.id, ": und wieder locale 0");

// Ausgabe:
// L0: nur auf locale 0
// L1: jetzt nur locale 1
// L0: und wieder locale 0
```

Natürlich können auf den Zielrechnern, welche mit „on" benannt wurden, parallel verschiedene Aufträge bearbeitet werden:

```
writeln("start on locale 0");
begin onLocales(1) do
      writeln("now on locale 1");
on Locales(2) do begin
      writeln("now on locale 2");
writeln("on locale 0 again");
```

Es wurde durch das erste begin nahezu gleichzeitig an locale 1 und locale 2 ein Befehl zur Ausgabe gegeben. Da auch der zweite Befehl mit „begin" gestartet wurde und direkt mit der letzten lokalen Ausgabe begonnen werden kann, könnten die insgesamt 3 Ausgaben in jeder möglichen Reihenfolge erscheinen

Ähnlich wie bei MPI könnte man schreiben:

```
proc main() {
      coforall loc in Locales do
            on loc do
                  MySPMDProgram(loc.id,
            Locales.numElements);
}

proc MySPMDProgram(me, p) {
      ...
}
```

Was liegt auf welchem Locale?

```
const LocaleSpace = [0..numLocales-1]; // build in
var c: Circle;
on Locales(i) { //migrate task to new locale
          writeln(here.id);
          c = new Circle(); //allocate class on locale
}
writeln(c.locale); //query locale of class instance
on c do { … } //data-driven task migration
```

Frage Variable nach ihrem Locale

```
var i: int;
on Locales(1) {
      var j: int;
      writeln(i.locale.id, " ", j.locale.id);
}
// Ausgabe:
// 0 1
```

Serielles Beispiel mit impliziter Kommunikation

```
var x, y: real; // x and y allocated on locale 0
on Locales(1) {    // migrate task to locale 1
      var z: real;   // z allocated on locale 1
      z = x + y;    // remote reads of x and y

      on Locales(0) do       // migrate back to locale
      0
      z = x + y;             // remote write to z
            // migrate back to locale 1
      on x do // data-driven migration to locale 0
                  // tue auf dem Locale, wo sich x
                // befindet folgendes:
            z = x + y; // remote write to z
                        // migrate back to locale 1
} // migrate back to locale 0
```

"local" statement
Asserts to the compiler, that all operations are local Gilt nur für den nächsten Befehl, wenn kein Body angegeben ist. Das Schlüsselwort „do" wird hier nicht genutzt. Es handelt sich um eine Überprüfung, ob alle Operation im body lokal verlaufen, ansonsten erscheint eine Fehlermeldung. Ausgabe mit write(„Test") und writeln(„Test") auf anderen

Locales als Locale 0 schlagen hier auch an, obwohl nicht einmal eine fremde Variable gelesen wird!

```
on Locales(1) {
        local {
                x = y;
    }
    writeln(x);
}
```

Datei IO

Chapel 1.4.0
Datei zum Schreiben öffnen:

```
var datei: file = new file();
datei.filename = name_ausgabedatei;
datei.mode = FileAccessMode.write;
datei.open();
```

Chapel 1.5.0
Datei zum Lesen öffnen:

```
var f = open(name_ausgabedatei,
iomode.r).reader();
```

Datei zum Schreiben öffnen:

```
var datei = open(name_ausgabedatei,
iomode.cw).writer();
```

Wenn eine Datei zum Schreiben geöffnet wurde, werden alle write() und writeln() Ausgaben in die Datei umgeleitet.

Mehr Infos zur IO unter doc/technotes/README.io

Hilfe und Informationen

www: http://chapel.cray.com/
contact: chapel_info@cray.com
bugs: chapel-bugs@lists.sourceforge.net
discussion: chapel-users@lists.sourceforge.net

$CHPL_HOME/README.tasks

Anhang B

Eigenes X10 Handbuch

Hello World

```
public class Mandelbrot {
   public static def main(Array[String]) {
      Console.OUT.println("Hello, World!");
   }
}
```

Kommandozeilenparameter

```
public static def main(args: Array[String](1)) {
        Console.OUT.println("Anzahl Argumente: "
+ args.size);
        if (args.size >=1) {
                var size:Int;
                size = Int.parse(args(0));
                Console.OUT.println(size);
        }
}
```

Variablendeklaration var vs val

```
var m : Int = 5;// Variable in Methode
public var m : Int = 0; // Instanzvariable
```

val macht Variablen „final", also unveränderbar.
Man nennt sie immutables oder auch constants

```
val i : Int = 5; // Kann nicht verändert werden
h : Int = 6; // ist implizit final, wenn h nicht existiert

val i : Int; // möglich: final Variable deklarieren und
i = 4;       // später einmalig beschreiben
```

Objektfelder dürfen Variablen sein. Bei Klassenfeldern sind nur Konstanten erlaubt.

Type Cast

```
(value as Int) // floatingpoint to int

Int.parse(value); // String to int
```

Assertions

```
assert j == 0;
```

Methoden

```
static def inc(var i:Int) {
        i += 1;
}
```

Ranges und Regions

Ranges sind IntRanges, welche ein Set von Integern sind. Auch zweidimensionale Ranges sind erlaubt. Sie werden so definiert wie Regions, so dass kein Unterschied festgestellt werden kann. Für Regions sind viele Methoden angegeben. Eine Region ist ein Set von Points mit gleichem Rank. Es existiert eine umfangreiche Algebra, mit der verschiedene Regionen verknüpft werden können

```
val R1 = 1..10; // IntRange
for(i in R1) Console.OUT.println(i);

// Zweidimensionale Region
val R1 = (1..2)*(1..5); // IntRange
for ([i, j] in R1) Console.OUT.println(i + " / " + j);

// Alle Koordinaten einer Region um einen
bestimmten Betrag erhöhen:
val R1 = 0..2*0..3 // [0,0] [0,1] [0,2] …
val p: Point = [1, 2];
val R2 = R1 + p; //  [1,2] [1,3] [1,4] …
```

Arrays

Lokale, also auf einen Place bezogene Arrays, werden über mehrdimensionale "Points" adressiert. Verteilte, sogenannte DistArrays mit Dists verteilt und adressiert.
Points sind mehrdimensionale Indices.
Jedes Array hat eine Region.

```
Array[T] a // die Region des Arrays ist A.region
a.region // Region von a
a.size // Anzahl Elemente
a.rank // das selbe wie a.region.rank


val A1 = new Array[Int](1..10);
A1(4) = 4;
// Gibt Index und entsprechende Speicherzelle aus:
for (i in A1) Console.OUT.println(i + " " +  A1(i));

Mit Nullen initialisiert instanziieren
val A1 = new Array[Int](1..10, 0);
val A4 = new Array[Int]((1..2)*(1..3), 0);

direkte Manipulation von Elementen
A1(4) = A1(4) + 1;

// Direkt instanzieren
val a = [[1,2,3],[7,8,9]];
```

// Ausgeben von lokalem Array:
Die Syntax a.size gibt bei zweidimensionalen Arrays nur die gesamte Anzahl an Elementen zurück. Die Syntax a(0).size taucht zwar in der Spezifikation stellenweise auf, ist aber nicht implementiert. Eine Ausgabe mit Zeilenumbruch sollte also mit Vorgabe der Arraybreite möglich:

```
for (idx in localArray) {
```

```
        Console.OUT.print(localArray(idx) + " ");
        if (idx(1) == width-1) {
                Console.OUT.println();
        }
}
```

```
// Workaround, wenn Array aus Region entstanden
ist:
var actualLine : Int = 0;;
for ([i, j] in a.region) {
   actualLine = i;
   break;
}
for ([i, j] in a.region) {
   if (i != actualLine) {
        actualLine = i;
        Console.OUT.println();
   }
   Console.OUT.print(a(i, j));
}
```

Distributions

Eine Region wird auf mehrere places gemapped. Die Verteilungen „Dist" liegen in x10.array.Dist, welches aber bereits implizit importiert ist. Der rank der Verteilung ist auch der rang der Region mit dem diese erstellt wurde.

```
// Mapped alle Punkte der Region R zum Place P
val D <: Dist = Dist.makeConstant(R,P)

// Blockverteilung aller Punkte der Region R über
alle Locales. Weitere Optionen können in der API
Dokumentation nachlesen werden.
val D <: Dist = Dist.makeBlock(R)
```

Die zyklische Verteilung sowie die blockzyklische Verteilung sind noch nicht nutzbar, jedoch waren die Konstrukte in einer Spezifikation von 2010 dokumentiert und sind auch jetzt noch in der API aufgeführt.

Funktionen zu Distributions D und Verteilten Arrays „da" mit Beispielausgaben:

```
D.region // [0..10] // Region von D
da.region // [0..10] // Region vom Array
D.rank // 1 // Anzahl Dimensionen von D
da.rank // 1 // Anzahl Dimensionen vom Array
D(5) // Place(1) // Place von Index 5 von D
da.dist(5) // Place(1) // Place von Index 5
D(5).id // 1 // Place ID von Index 5 von D
da.dist(5).id // 1 // Place ID von Index 5 vom Array
```

Beispiel: val p: Place = da.dist(el);

Möchte man nur eine Region R ansprechen, welche aber nur eine Teilmenge der Region ist, mit der die Distribution D erstellt wurde, kann man mit
D | R
alle Points (Typ Distribution) erhalten, welche zu R gehören

Möchte man von einer Distribution D alle Punkte ansprechen, welche auf einem speziellen Place P liegen, kann man diese mit
D | P
filtern. Bei einer zyklischen Verteilung sollte man eine Funktion implementieren, welche alle entsprechenden (lokalen) Indices innerhalb eines Arrays auflisten kann.

Alle Elemente eines Verteilten Arrays ausgeben, welche Place(2) zugeordnet sind:

```
at (Place.places()(2)) {
        for (i in da.dist | Place(2)) {
                Console.OUT.print(da(i) + " ");
        }
        Console.OUT.println();
}
```

Schleifen

```
for (i in 0..10) Console.OUT.println(i);
```

for (i in A1) …ist identisch mit
for (i in A1.region) …

```
for (var i: Int = 0; i < a.size; i++) …
```

Nativer Code

Es ist möglich, java oder c++ Code in x10 Dateien einzufügen.

Native Methoden:

Dafür kann eine eigene Methode geschrieben werden, wobei der entsprechende Quellcode VOR der Methode eingefügt wird. Der C++ Code wird dann ausgeführt, wenn mit C++ Backend compiliert wird und der Java Code, wenn mit Java Backend compiliert wird. Ist nur C++ oder nur Java Code angegeben, wird im Fall des Backends, für den kein Code eingefügt wurde, der Body des Methode, welcher nur X10 Code enthalten kann, ausgeführt:

```
@Native("c++",   "printf(\"Hi c++1!\");" +
                 "printf(\"Hi c++2!\");")
        @Native("java", "System.out.println(\"Hi!
java\")")
static def printNatively():void = {
        Console.OUT.println("Error: The x10 Code
        is used!");
};
```

Soll erzwungen werden, dass sowohl Java aber auch C++ Code implementiert wird, kann das mit dem Modifier "native" erzwungen werden. Dann ist auch kein Body mehr erlaubt:

```
@Native("c++",   "printf(\"Hi c++1!\");" +
                 "printf(\"Hi c++2!\");")
```

```
	@Native("java", "System.out.println(\"Hi! java\")")
public static native def myMethod():void;
```

Parameterübergabe ist auch möglich, aber die Variablennamen werden in der Reihenfolge in der sie übergeben wurden mit #1, #2… benannt.

Nativer Quellcode innerhalb von X10 Methoden:

Es kann auch Java oder C++ Code mitten in einer X10 Methode eingefügt werden. Dieser kann auch auf die Variablen der Methode zugreifen. Wird nur für Java oder nur für C++ nativer Code geschrieben, wird im Falle des „falschen" Backends der anschließende Body (in geschweiften Klammern) ausgeführt:

```
public static def dateiSchreiben(x: Int) {
	@Native("c++", "printf(\"Hi c++! x: %d\", x);")
	{Console.OUT.println("Error - c++ was not acepted");}
	}
```

Parallelisierung

Places

```
pl.id // Nummer des Place zwischen 0 und
      // Place.MAX_PLACES-1
here.next(); // Nachfolger Place
Place.MAX_PLACES // Anzahl Places
Place.FIRST_PLACE // Place(0)
Place.places()(2) // Place mit ID 2
```

```
Hello word parallel:
finish for (p in Place.places()) {
	async at (p) Console.OUT.println("Hello World from place "+p.id);
    }
```

Der „at" Operator

Ein Task (=Aktivität) kann auf einem speziellen Place platziert werden. Man spricht von „Place-shifting". Mit dem at Schlüsselwort wird eine neue Aktivität gestartet, welche sich am selben Ort befindet, wie das Element. Man kann auch für jeden Ort eine Aktivität starten die alle dortigen lokalen Elemente manipuliert:

```
at (Place.places()(1)) Console.OUT.println("Hello from place: " + here); // place 1
at (here) Console.OUT.println("Hello World from place: " + here); // place 0 (wenn der das gestartet hat)
```

Vorsicht: Alle Werte, welche am Zielplace benötigt werden, werden zu diesem kopiert und unter gleichem Variablennamen abgelegt. Veränderungen können am Place, dem die Daten eigentlich gehören, nicht beobachtet werden. Es handelt sich um eine „deep copy", weil nur die Werte kopiert werden, nicht aber verweise auf die Originale.

```
at (P) S

for (place in places) {
	at(place) {
	val pointsAtP : Region{rank == D.rank} =
		D.get(place)
```

async und finish

Mittels async werden neue Aktivitäten gestartet. Ohne umgebendes finish geht der Programmablauf direkt weiter und das async statement wird parallel dazu abgearbeitet.

```
async {s1();} // Hauptthread wartet nicht auf s1()…
async {s2();} // sondern startet s2()
```

Mit finish wartet der Aufruber, bis alle Aktivitäten (auch die auf fremden Places gestartet wurden) beendet sind

```
finish async {s1();} // wartet hier bis s1 fertig ist
async {s2();}
```

Eine Aktivität auf einem anderen Place starten und parallel weiter arbeiten, indem dieser Vorgang in einer eigenen Aktivität gestartet wird:
async at(p) S

Auf dem entfernten Place p eine eigene Aktivität starten, um S auszuführen
at(p) async S

atomic

Zugriffe auf Variablen innerhalb eines atomic Blocks schließen gleichzeitigen Zugriff von anderen Aktivitäten aus, wenn diese auch aus einem atomic Block darauf zugreifen.

Clocks

Sollen mehrere Aktivitäten gemeinsam nach einzelnen Rechenschritten aufeinander warten, bietet sich das Konzept der Clocks an. Jede Aktivität gibt an, wann sie mit ihrem aktuellen Schritt fertig ist und wartet, bis alle Aktivitäten auf diesem Stand sind. Dann arbeiten die Aktivitäten automatisch weiter.

Im folgenden Beispiel warten die Aktivitäten A und B nach jedem Schritt aufeinander, so dass die Schritte A1 und B1 in beliebiger Reihenfolge vor A2 und B2 ausgeführt werden usw.

```
val cl = Clock.make();
async clocked(cl) {// Activity A
say("A-1");
Clock.advanceAll();
```

```
say("A-2");
Clock.advanceAll();
say("A-3");
}// Activity A
async clocked(cl) {// Activity B
say("B-1");
Clock.advanceAll();
say("B-2");
Clock.advanceAll();
say("B-3");
}// Activity B
```