

2. Hausübung - MPI

Dokumentation von Leon Bohmann (2493657) und Jonathan Stollberg (247775)
Tag der Einreichung: 15. Januar 2023

Darmstadt



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Inhaltsverzeichnis

1	Einführung	3
2	Sequentieller Algorithmus	4
3	Cannon-Algorithmus	5
4	Zeitmessung	7
5	Benchmark und Fazit	8

1 Einführung

Auf die 1. Hausübung aufbauend werden in dieser Ausarbeitung weitere Methoden zur parallelen Berechnung von Hauptproblemen der linearen Algebra berechnet. Genauer sollen Matrizen-Multiplikationen beschleunigt werden, indem ihre Berechnung auf mehrere Anwendungs-Instanzen aufgeteilt und mittels des Message Passing Interfaces (MPI) verknüpft wird. Zur Berechnung des Speedups werden sequentielle wie auch die vorher verwendeten beschleunigten Algorithmen herangezogen. Zuletzt, wird der Code auf dem Lichtenberg Hochleistungsrechner ausgeführt.

Das implementierte Programm löst das folgende Problem:

$$\mathbf{C} = \mathbf{AB} \tag{1.1}$$

2 Sequentieller Algorithmus

Zur Vergleichbarkeit wird ein sequentieller Algorithmus implementiert, der das oben beschriebene Problem lösen kann:

```
1  solution = new short[m * m];
2
3  // compute matrix-vector product
4  for (int i = 0; i < m; i++) {
5      for (int j = 0; j < m; j++) {
6          for (int k = 0; k < m; k++) {
7              solution[i * m + k] += A[i * m + j] * B[j * m + k];
8          }
9      }
10 }
```

Listing 2.1: Sequentieller Algorithmus zur Berechnung Matrix-Matrix-Produkte

Zum weiteren Vergleich wurde auch ein einfacher paralleler Algorithmus aus der ersten Hausübung übernommen:

```
1  solution = new short[m * m];
2  IntStream.range(0, m).parallel().forEach(i -> {
3      for (int j = 0; j < m; j++) {
4          for (int k = 0; k < m; k++) {
5              solution[i * m + k] += A[i * m + j] * B[j * m + k];
6          }
7      }
8  });
```

Listing 2.2: Paralleler Algorithmus mit Streams-API

Um die Richtigkeit der durch Streams und MPI parallel berechneten Ergebnisse zu gewährleisten, werden diese am Ende eines jeden Programmdurchlaufs mit dem sequentiell berechneten Ergebnis abgeglichen.

3 Cannon-Algorithmus

Um die Matrix aufzuteilen und mit einzelnen Programm-Instanzen berechnen zu können, bedarf es einem passenden Algorithmus. Hierfür wurde der Cannon-Algorithmus implementiert, da dieser die parallele Berechnung mit mehreren Programm-Instanzen erlaubt. Die Besonderheit des Cannon-Algorithmus ist, dass die einzelnen Prozesse auf dedizierte Speicheradressen zugreifen und somit Adress-Blockierungen umgangen werden können.

Ablauf Beim Cannon-Algorithmus wird die zugrundeliegende Matrix reihen- beziehungsweise spaltenweise verschoben. Die einzelnen Prozesse arbeiten dann immer mit der Submatrix an ihren Koordinaten. Der Prozess 1 einer Matrix mit 3x3 Einträgen arbeitet dann immer mit der Submatrix an der Position (0,0). Nach jeder Iteration, also der Berechnung der Submatrizen aller Prozesse werden die Matrixeinträge zyklisch vertauscht (siehe 3.1).

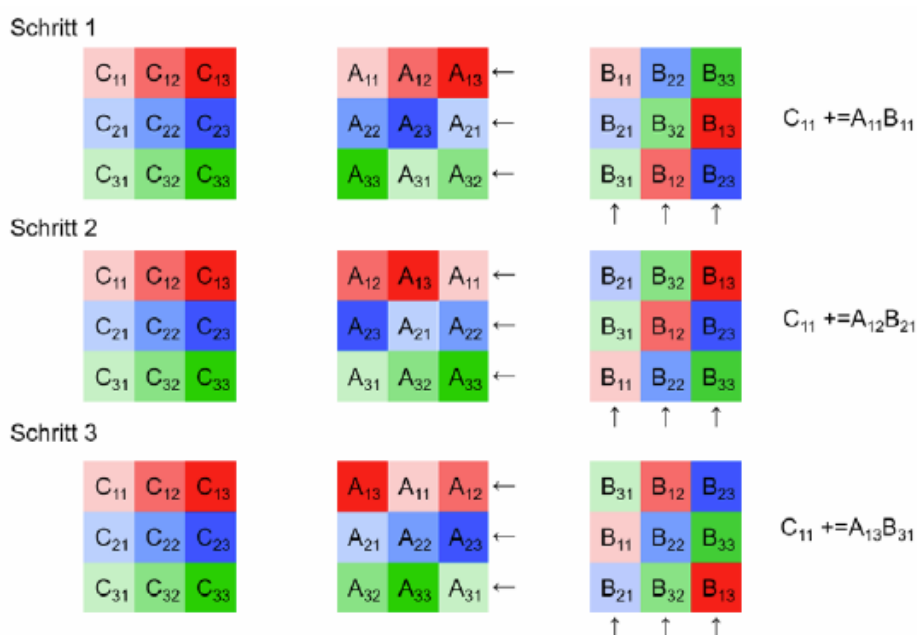


Abbildung 3.1: Zyklisches Vertauschen nach jeder Iteration

In der darauf folgenden Iteration werden die neuen Daten an den Koordinaten des Prozesses von vorher verwendet und auf das Ergebnis summiert. Diese Operation kann man wie folgt formulieren:

$$\mathbf{C}_{ij} = \sum_{n=0}^{size} \mathbf{A}_{(il)} \cdot \mathbf{B}_{(lj)} \quad (3.1)$$

Durch die Klammern um die Indizes ist angedeutet, dass es sich um Submatrizen von **A** und **B** handelt. Die Besonderheit in der Formel ist (wie bereits erwähnt) in der Summation verborgen. Diese erfolgt nicht im Prozess (i, j) sondern global für die zugrundeliegenden Matrizen **A** und **B**. Man erkennt aber leicht, dass die Formel analog der sequentiellen Matrizenmultiplikation ist. Angenommen, die Submatrizen sind Skalare, kann man die Formel also auch schreiben als:

$$\mathbf{C}_{ij} = \sum_{l=0}^n \mathbf{A}_{il} \mathbf{B}_{lj} = \mathbf{A}_{11} \mathbf{B}_{11} + \mathbf{A}_{12} \mathbf{B}_{21} + \mathbf{A}_{13} \mathbf{B}_{31} \quad (3.2)$$

Damit ist die Analogität leicht ersichtlich.

Implementierung Um die Matrix aufzuteilen muss zunächst die Anzahl an gleichzeitigen Prozessen sowie ihre jeweilige Kennung bestimmt werden. Dafür können die Message Passing Interface (MPI) Methoden `MPI.COMM_WORLD.Rank()` und `MPI.COMM_WORLD.Size()` verwendet werden. Mit den so bestimmten Größen können die Matrix-Blöcke definiert werden.

Nun werden die Submatrizen an die einzelnen Prozesse verteilt. Dafür nutzt man die `MPI.COMM_WORLD.Scatter(...)` Methode. Für eine korrekte Initialisierung des Algorithmus müssen die Submatrizen anschließend allerdings auf dem Prozessgitter zyklisch verschoben werden, wie in Abb. 3.2 dargestellt. Die Submatrizen der Matrix **A** werden zyklisch horizontal verschoben, die Submatrizen von **B** vertikal.

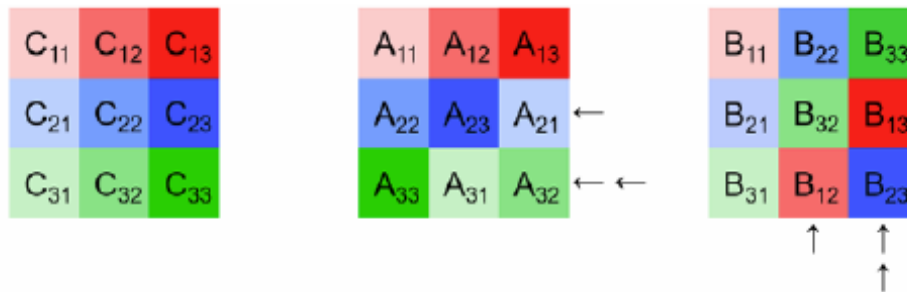


Abbildung 3.2: Initialisierung des Algorithmus

Zur Realisierung der Verschiebungen nutzt man die `MPI.COMM_WORLD.Sendrecv_replace(...)` Methode bzw. alternativ die Methoden `MPI.COMM_WORLD.Isend(...)` und `MPI.COMM_WORLD.Irecv(...)`. Anschließend können die Submatrizen auf den jeweiligen Prozessen sequentiell miteinander multipliziert werden. Durch jede weitere Verschiebung bauen sich wie zuvor gezeigt die korrekten Submatrizen der Lösungsmatrix **C** auf.

Nach erfolgreicher Berechnung der Submatrizen kann das Ergebnis nun mittels `MPI.COMM_WORLD.Gather(...)` wieder zurück in die globale Lösungs-Matrix geschrieben werden.



4 Zeitmessung

Die Zeitmessung wird von einem einzigen Prozess kontrolliert. Nachdem eine Barrieroperation alle laufenden MPI-Prozesse synchronisiert, misst dieser die Walltime. Melden alle Prozesse einen Abschluss, so wird erneut die Walltime erfasst und so die gesamte Berechnungsdauer ermittelt.

5 Benchmark und Fazit

Auf dem Lichtenberg Hochleistungsrechner wurden die Algorithmen nacheinander ausgeführt. In den folgenden Darstellungen werden der sequentielle, durch Streams parallelisierte und der durch MPI parallelisierte Algorithmus gegenüber gestellt.

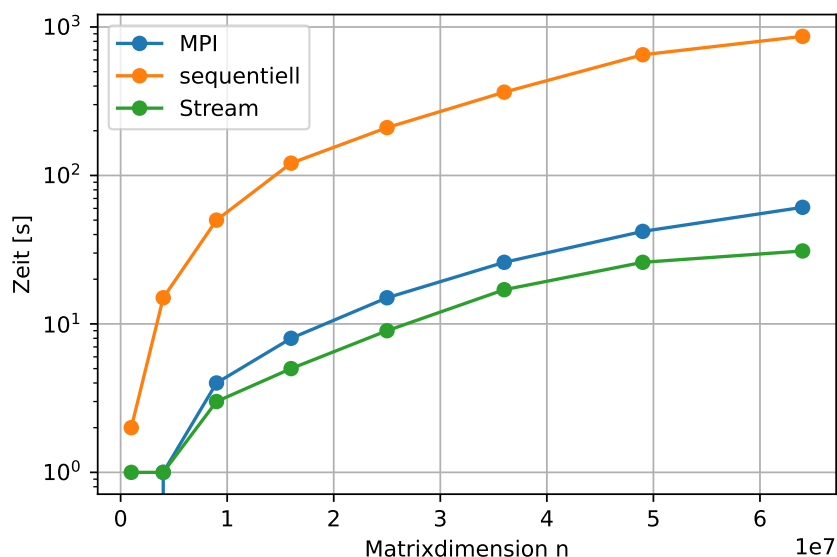


Abbildung 5.1: Berechnungsdauer in Abhängigkeit der Matrixdimension n , 16 Prozesse (bzw. Kerne)

In 5.1 ist die Berechnungsdauer einer Matrixmultiplikation bis 64 Mio. Einträgen dargestellt. Es ist zu erkennen, dass die parallelen Algorithmen um ein vielfaches schneller sind, als der Sequentielle. Auffällig ist aber, dass die durch Streams realisierte Parallelisierung schneller ist, als die MPI-Methode. Das liegt vor allem daran, dass MPI eigentlich für mehrere Nodes zum Einsatz kommt. Dabei kann die Berechnung auf mehreren Rechnern mit wiederum mehreren Kernen ausgeführt werden, was im Rahmen dieser Versuche leider nicht möglich war, da auf dem Lichtenberg nur ein Knoten zur Verfügung stand. Außerdem ist der Zusatzaufwand der Datenverteilung durch MPI auf nur einem Knoten deutlich höher, als dass ihn der Speedup der eigentlich Berechnung wieder gut machen könnte.

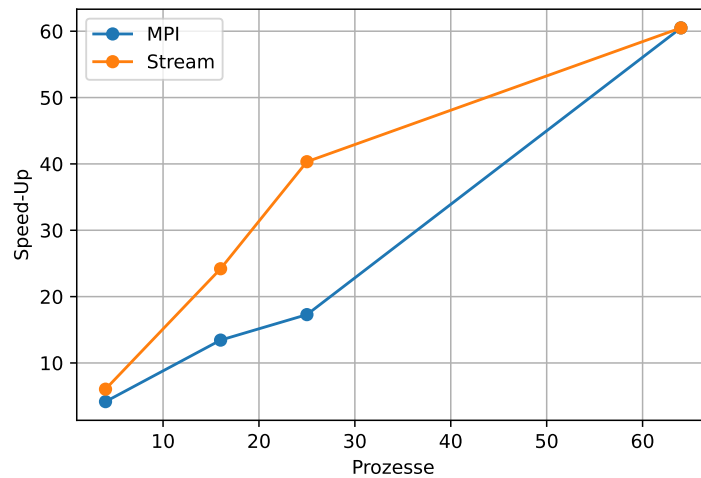


Abbildung 5.2: Speedup gegenüber der sequentiellen Berechnung bei 16 Mio. Einträgen

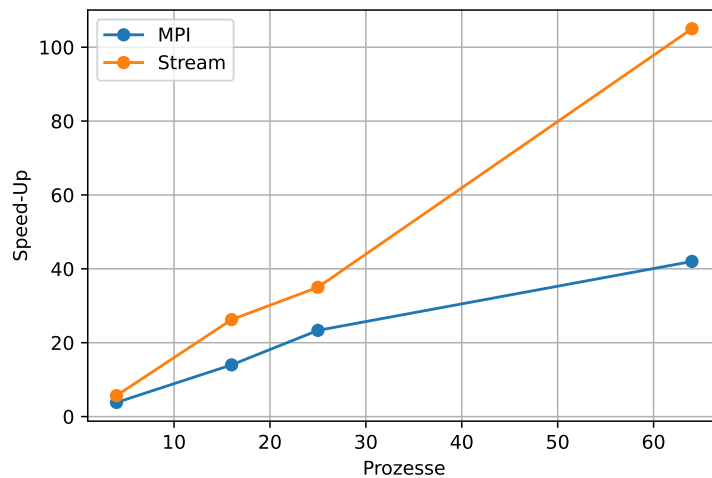



Abbildung 5.3: Speedup gegenüber der sequentiellen Berechnung bei 25 Mio. Einträgen

In 5.2 und 5.3 sind die relativen Speedups gegenüber der sequentiellen Berechnung dargestellt. Auffällig ist, dass die Berechnung der größeren Matrix mit Streams einen fast doppelt so großen Speedup erzielt als die der kleineren Matrix. Das deutet weiter darauf hin, dass die Datenverteilung durch MPI einen erheblichen Zeitaufwand birgt.

Zusammenfassend kann festgehalten werden, dass die mögliche Beschleunigung durch die Verwendung von für MPI optimierte Parallelisierungen nicht ausgereizt werden konnte. Für einknotige Maschinen mit mehreren Kernen waren die durch Streams implementierten Algorithmen in jeder Hinsicht schneller und deutlich



einfacher zu entwickeln.