OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

Diskussion der Parallelisierung und Leistungsbetrachtung des Relaxationsverfahren

An der Fakultät für Informatik und Mathematik der
Ostbayerischen Technischen Hochschule Regensburg
im Studiengang
Techniche Informatik

eingereichte

Studienarbeit

Vorgelegt von: Korbinian Federholzner
Matrikelnummer: 3114621

Gutachter: Prof. Dr. Rudolf Hackenberg

Inhaltsverzeichnis

1 Einleitung

In dieser Arbeit wird die Performance von skalierbaren Anwendungen analysiert und bewertet. Bei der betrachteten Anwendung handelt es sich um die Implementierung des Relaxationsverfahrens, unter der Verwendung der Jakobi-Methode. Dazu wird die sequenzielle Version des Programmes mit der parallelen Version verglichen. Für die Parallelisierung wird das Open Message Passing Interface (OpenMPI) verwendet. Die Ausführung findet am Supercomputer "SuperMU" des Leibniz-Rechenzentrums in München statt.

2 Verfahren und Parallelisierungsstrategie

2.1 Relaxationsverfahren

Im Beispiel der Studienarbeit soll die Wärmeverteilung auf einer Metallplatte berechnet werden. Um das Relaxationsverfahren zu verwenden, wird die Metallplatte in eine Matrix eingeteilt und diese an den Ecken mit Werten je nach Hitzegrad belegt. Um einen inneren Punkt der Matrix berechnen zu können, müssen alle direkten Nachbareinträge betrachtet werden. Dies wird iterativ mehrmals für alle Zellen durchgeführt bis der Fehler nicht mehr sehr groß ist. Dieser Fehler kann mittels der Norm des Residuums berechnet werden. Je genauer die Norm des Residuums gewählt wird, desto genauer das Ergebnis. Bei Abbildung 1 kann man erkennen, dass die mittleren Zellen gegen 0.5 konvergieren und dies bei einem exakten Ergebnis auch erreichen würden.

1	0.875	0.75	0.625	0.5	0.375	0.25	0.125	0
0.875	0.781216	0.687443	0.593681	0.499931	0.406191	0.312458	0.218729	0.125
0.75	0.687443	0.624902	0.562382	0.499882	0.437399	0.374929	0.312464	0.25
0.625	0.593681	0.562382	0.531108	0.499858	0.468628	0.437414	0.406207	0.375
0.5	0.499931	0.499882	0.499858	0.499858	0.499878	0.499914	0.499957	0.5
0.375	0.406191	0.437399	0.468628	0.499878	0.531146	0.562427	0.593713	0.625
0.25	0.312458	0.374929	0.437414	0.499914	0.562427	0.624948	0.687474	0.75
0.125	0.218729	0.312464	0.406207	0.499957	0.593713	0.687474	0.781237	0.875
0	0.125	0.25	0.375	0.5	0.625	0.75	0.875	1

Abbildung 1: Die 8x8 Ergebnismatrix bei $||r|| = 0.0001$

2.2 Lösungsverfahren des Linearen Gleichungssystems

Zur Lösung des Relaxationsverfahrens kann entweder das Gauß-Seidel- oder das Jakobi-Verfahren verwendet werden. Zwar ist das Gauß-Seidel-Verfahren schneller als das Jakobi-Verfahren, jedoch ist es wesentlich komplexer zu parallelisieren. Da beim Gauß-Seidel-Verfahren die Zwischenergebnisse der anderen Zellen in derselben Iteration eine Rolle spielen und beim Jakobi-Verfahren diese Ergebnisse erst nach dem Abschließen einer kompletten Iteration relevant sind. Daher wird eine Art Hybridvariante verwendet, bei welcher in den Submatrizen wie bei Gauß-Seidel die Zwischenergebnisse eine Rolle spielen. Wenn alle Submatrizen berechnet worden sind werden diese wie im Jakobi Verfahren in die Originalmatrix zurückgeschrieben, ohne dass sie in der ersten Iteration noch von ihren Nachbarprozessen beeinflusst werden.

2.3 Parallelisierungsstrategie

Die gewählte Parallelisierungsstrategie ist ein Master-Slave-Verfahren, bei dem ein Master-Prozess die aktuelle Matrix hält und diese dann gerecht in Submatrizen aufteilt und seine Slaves schickt. Nach dem Abschließen einer Iteration über die komplette Submatrix senden alle Slaves ihre fertig berechneten Submatrizen und ihr berechnetes Residuum zurück an den Master. Der Master berechnet nun das maximale Residuum und vergleicht dieses mit dem Abbruchkriterium. Gilt die Berechnung als abgeschlossen, so broadcastet der Master eine Message, dass das Programm beendet werden kann und alle Slaves stoppen ihre Berechnungen.

3 Implementierung

Für die Parallelisierung des Programmes wurde OpenMPI gewählt. OpenMPI bietet mehrere Funktionen für die parallelisierte Kommunikation von Prozessen. Da alle MPI-Funktionen mit Memoryblöcken, also in Array-Form arbeiten und nicht mit mehrdimensionalen Matrizen, wird eine Matrix-Wrapper-Klasse verwendet, die die Zugriffe auf das Array wie bei einer Matrix aussehen läst. Die einfachste Implementierung der MPI Kommunikation ist mit MPI_Send und MPI_Recv, welche es erlauben, beliebig große Nachrichten zwischen Prozessen auszutauschen. Dabei wird die Matrix horizontal in Submatrizen zerlegt, welche außerdem zwei Zeilen mehr als eigentlich zum berechnen benötigt, erhalten. Diese Zeilen sind die Werte, die vom Prozess davor und danach berechnet wurden. Sie sind für das Relaxationsverfahren von Relevanz, da dieses alle Nachbarzellen betrachtet. Siehe die gelbe Markierung in Abbildung 2.

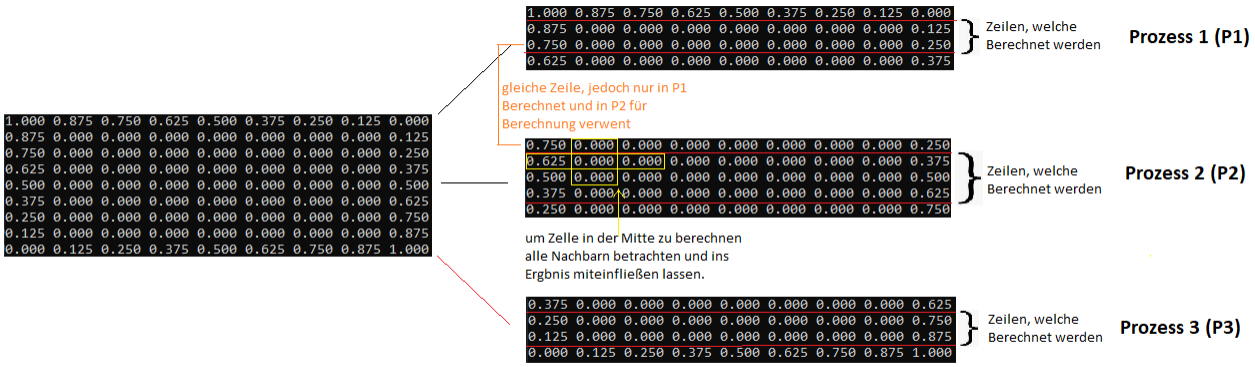

Abbildung 2: Aufteilung einer 9x9 Matrix auf 3 Prozesse

Eine bessere Implementierung, erreicht man mit MPI_Scatterv und MPI_Gatherv. Diese MPI Funktionen erhalten einen Offset vom Anfang der Originalmatrix und die Größe der neu erzeugten Submatrix und verschicken diese an die einzelnen Slave-Prozesse. Der Vorteil zu MPI_Send und MPI_Recv ist, dass MPI_Scatterv und MPI_Gatherv effizienter implementiert sind und dadurch dasselbe schneller erledigen können. Der Ablauf der Implementierung sieht wie folgt aus:

- MPI_Scatterv teilt die gegebene Matrix auf und verteilt Submatrizen an alle Prozesse.
- Jeder Prozess berechnet für jeden Zellenwert der Submatrix einen neuen Wert.
- MPI_Gatherv sammelt alle Ergebnismatrizen der Prozesse und integriert diese in die Originalmatrix.
- MPI_Gather sammelt die berechneten maximalen Residuen.
- Der Master-Prozess bildet das maximale Residuum aus allen Ergebnissen.
- Ist die gegebe Genauigkeit unterschritten, wird mit MPI_Bcast eine Broadcast-Meldung an alle Prozesse geschickt.
- Erhält ein Prozess den Broadcast, dann beendet er sich, ansonsten wiederholt sich der Algorithmus bei Punkt 1 erneut.

4 Analyse des Programmes

4.1 Analyse Tool ITAC

Für die Analyse des Programmes wird das Intel Trace Analyser Tool (ITAC) verwendet. Diese Anwendung erlaubt es das Verhalten der Kommunikation von MPI Programmen zu analysieren. Performance Probleme die mit MPI zusammenhängen, können durch Analysieren von Timelines und Statistiken identifiziert und behandelt werden. Um eine Analyse des Programmes durchführen zu können, muss erst eine stf-Datei erzeugt werden. Dazu muss das Programm mit den nötigen Compiler-Flags kompiliert und ausgeführt werden. Der Nachteil des kompilierens mit den ITAC Compiler-Flag ist, dass es das Programm mit mehr Overhead füllt und daher mehr Rechenzeit benötigt.

4.2 Betrachtete Testfälle

Für die Analyse des Programmes werden drei verschiedene Größen gewählt und die Ergebnisse aus dem Intel Trace Analyser Tool abgelesen.

- Matrix 150x150 Genauigkeit 0.0001
- Matrix 200x200 Genauigkeit 0.001
- Matrix 8x8 Genauigkeit 0.0001

Für jede der Größen werden jeweils 1, 2, 4, 8, oder 16 Prozesse gewählt und die Ergebnisse in eine Tabelle eingetragen. Siehe Abbildung 3.

Matrix-Dimension	150x150	Genauigkeit	0.0001					
Nodes	Prozesse	Ges	Dauer ohne MPI (s)	Zeit MPI	Zeit Gesamt (s)	Serial Code	MPI calls Percent	MPI #calls	Application Calls
1	1	1	630	1	631	0.998415214	0.001584786	24015	2237566969
1	2	2	446	47	493	0.904665314	0.095334686	48254	2250213817
1	4	4	383	60	443	0.864559819	0.135440181	96892	2263062010
1	8	8	322	102	424	0.759433962	0.240566038	19538	2291247175
1	16	16	325	90	415	0.78313253	0.21686747	371265	2200878513
Matrix-Dimension	200x200	Genauigkeit	0.001					
Nodes	Prozesse	Ges	Dauer ohne MPI (s)	Zeit MPI	Zeit Gesamt (s)	Serial Code	MPI calls Percent	MPI #calls	Application Calls
1	1	1	124	0.01	124.01	0.999919361	8.06387E-05	4603	726860236
1	2	2	104	1.4	105.4	0.986717268	0.013282732	8718	726595523
1	4	4	95	6	101	0.940594059	0.059405941	96892	2263062010
1	8	8	89	10	99	0.898989899	0.101010101	19538	2291247175
1	16	16	81	17	98	0.826530612	0.173469388	371265	2200878513
Matrix-Dimension	8x8	Genauigkeit	0.0001					
Nodes	Prozesse	Ges	Dauer ohne MPI (ms)	Zeit MPI	Zeit Gesamt (ms)	Serial Code	MPI calls Percent	MPI #calls	Application Calls
1	1	1	12	1	13	0.923076923	0.076923077	223	66820
1	2	2	18	11	29	0.620689655	0.379310345	502	98452
1	4	4	31	36	67	0.462686567	0.537313433	1084	156400
1	8	8	59	147	206	0.286407767	0.713592233	2504	298216
1	16	16	Bei einer 8x8 Matrix, bei Aufteilung in 16 Reihen nicht möglich						

Abbildung 3: Ergebnisse nach der Analyse des Programmes mit ITAC

Die Matrix von 8x8 ist ein Negativbeispiel, welches darstellen soll, wenn man zu viele Aufteilungen bei nur einer kleinen Matrize vornimmt. Bei ihr ist zu erwarten, dass der Overhead der durch die Parallelisierung entsteht, die Laufzeit des Programmes verschlechtert. Der Graph 4 zeigt, dass sich mit steigender Anzahl an Prozessen sich alle verbessern, nur die 8x8 Matrix nicht. Die Orange Linie verfolgt den optimalen Verlauf, falls sich die Laufzeit mit jedem Prozess verdoppeln würde. Siehe Abbildung 4

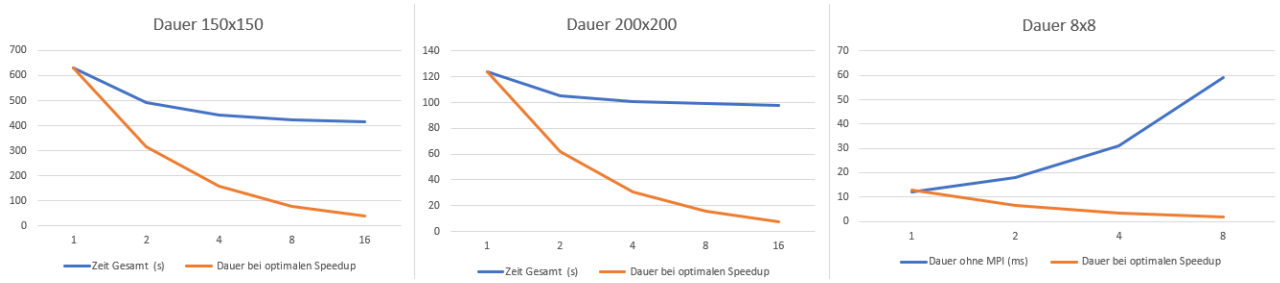

Abbildung 4: Die Laufzeiten der Programme als Graphen

4.3 Speedup

Der Speedup ist eine Maß für den Geschwindigkeitszusatz, der durch die Parallelisierung erreicht wird.

$$S = \frac{T(1)}{T(n)} \tag{1}$$

Dabei steht S für den Speedup, $T(1)$ die sequentielle Zeit bei einem Prozess und $T(n)$ die Zeit bei n Prozessen.
Ein Optimaler Speedup wäre z.B., wenn sich die Ausführungszeit bei doppelter Anzahl an

Prozessen halbieren würde. Ein Superscalarer Speedup wäre, falls sich die Ausführungszeit dann mehr als verdoppeln würde. Der Speedup für die Appliaktion in den gewählten Testgrößen hat den Graphen in Abbildung 5

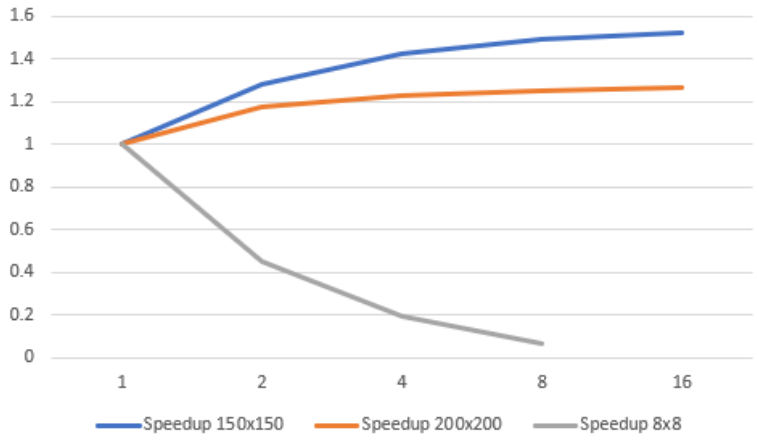

Abbildung 5: Speedup für die jeweilig gewählten Testgrößen

Bei dem Verlauf der 8x8 Matrix nimmt der Speeup ab, was daran liegt, dass das Programm mit steigenden Parallelismus durch Overhead langsamer wird. Bei allen anderen bringt das Hinzufügen von Prozessen eine Verbesserung der Rechenzeit.

4.4 Amdahlsches Gesetz

Das Gesetz nach Amdahl beschreibt den Geschwindigkeitszusatz, beim parallelisieren von Programmen, der maximal erreicht werden kann. Da ein Programm nie vollständig parallel ausgeführt werden kann, da einige Teile wie Initialisierungen und Speicherzugriffe immer sequentiell sein müssen, wird beim Amdahlschen Gesetz das Programm in einen parallelisierbaren und einen nicht parallelisierbaren Teil zerlegt. Die Formel für den maximal erreichbaren Speedup nach Amdahl lautet:

$$S = \frac{1}{(1-P) + (\frac{P}{N})} \tag{2}$$

Hier steht S für den Speedup der maximal erreicht werden kann, P für den parallelisierbaren Anteil der Applikation und N für die Anzahl der parallelen Prozesse.
Da dieses Gesetz auch für die entwickelte Anwendung gilt, sind die Ergebnisse aus Abbildung 5, den Erwartungen entsprechend.

4.5 Effizienz

Die Effizienz E ist eine Maßzahl, die den erzielten Speedup mit dem maximal möglichen Speedup vergleicht.

$$E(n) = \frac{T(1)}{n \cdot T(n)} \tag{3}$$

Wobei E der Effizenz, $T(1)$ der Zeit im sequentiellen Fall und $T(n)$ der Zeit bei n Prozessen entspricht.
Die Ergebnisse der Testfälle der Applikation ergeben den Graphen 6.

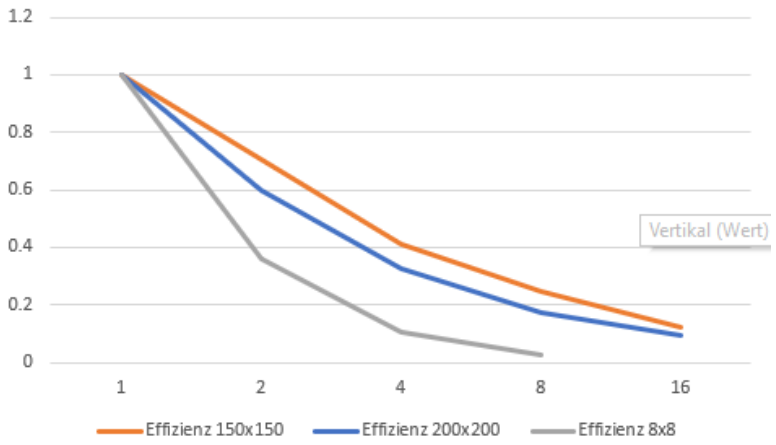

Abbildung 6: Die Effizienz der Testfälle

Hier ist zu sehen, dass die Effizienz mit steigendem Parallelismus fällt.

4.6 Redundanz

Die Redundanz beschreibt die Verträglichkeit zwischen Software- und Hardwareparallelismus. Je höher der Aufwand der Synchronisation und Parallelisierung der Kommunikation ist, desto höher ist die Redundanz. Code der vom Entwickler parallelisiert worden ist, wird auf der Hardware manchmal anders abgebildet wie erwartet. Das heißt es kann sein, dass wesentlich mehr Befehle als erwartet erzeugt werden, (MPI-Calls).

$$R(n) = \frac{O(n)}{O(1)} \tag{4}$$

R ist die Redundanz, $O(n)$ ist die Anzahl der Operationen bei n Prozessen und $O(1)$ die Anzahl der Operationen bei einem Prozess.
Der Graph der Testfälle in Abbildung 7 zeigt eine verhältnismäßig hohe Redundanz vom Negativbeispiel zu den anderen auf, weil das Verhältnis von MPI-Calls zu normalen Applikationsaufrufen beim Negativbeispiel höher ist. So folgen beim Fall mit 8 Prozessen auf 298216 normale Aufrufe 2504 MPI-Aufrufe. In den anderen Fällen ist dieser Unterschied viel größer, vergleiche

Werte aus Abbildung 3.

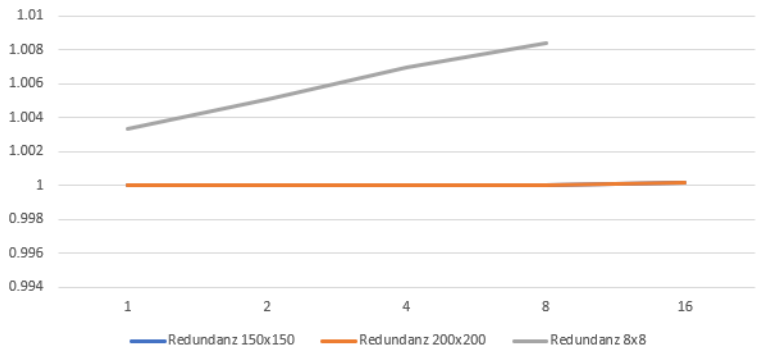

Abbildung 7: Die Redundanz der Testfälle

4.7 Auslastung

Die Auslastung, auch Utilization genannt, ist eine Kennzahl für den Anteil der Auslastung der Betriebsmittel.

$$U(n) = R(n) \cdot E(n) \tag{5}$$

Wie man in Abbildung 8 sehen kann, fällt die Auslastung der Betriebsmittel mit steigender Prozessanzahl ab, da durch die Synchronisation der Prozesse diese oftmals aufeinander warten müssen und so nicht vollständig ausgelastet werden können.

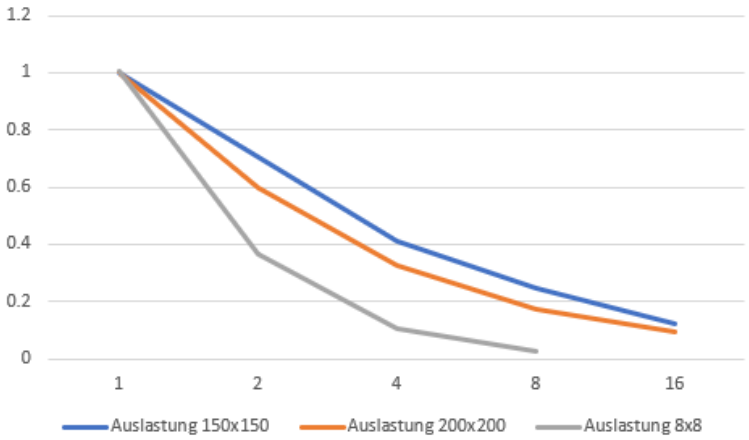

Abbildung 8: Die Auslastung der Testfälle

4.8 Qualität

Die Qualität ist eine Kennzahl für die Qualitätsverbesserung durch Abbildung der Software auf die Hardware. Dadurch erhält man ein Maß für die Güte der Software.

$$Q(n) = \frac{S(n) \cdot E(n)}{R(n)} \tag{6}$$

$$Q(n) = \frac{T(1)^3}{n \cdot T(n)^2 \cdot O(n)} \tag{7}$$

Wie auch in allen anderen Leistungsbetrachtungen, ist auch bei der Qualität der Negativfall extrem schlecht. Siehe Abbildung 9. Man kann hier feststellen, dass der Testfall der 150x150 Matrix von der Qualität der Beste und der 8x8 Fall der Schlechteste ist. Normal würde man erwarten, dass 200x200 hier besser ist, da im Verhältnis zu 150x150 der Overhead geringer sein müsste. Jedoch ist bei der 200x200 Matrix die Genauigkeit des Residuums 0.001 anstatt 0.0001, sodass sie deshalb auch weniger Operationen hat. Siehe Abbildung 3.

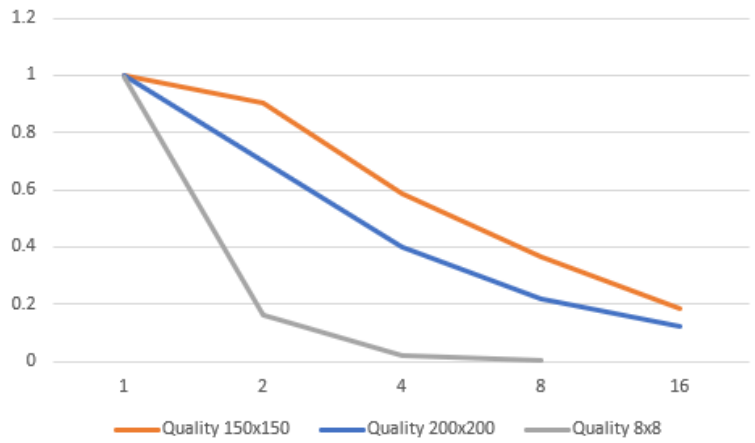

Abbildung 9: Die Qualität der Testfälle

5 Zusammenfassung und Verbesserungsmöglichkeiten

Die einfachste Anwendung für das Relaxationsverfahren kann man mit MPI_Send und MPI_Recv implementieren. Diese MPI Funktionen verursachen jedoch Overhead, der die Laufzeit der Applikation schlecht macht. Vor allem sind MPI_Send und MPI_Recv blockierende Funktionen, das heißt, nachdem sie aufgerufen worden sind, macht der zugehörige Prozess nichts, bis die Nachricht ankommt. Besser geeignet sind hier MPI_Isend und MPI_Irecv, da diese Funktionen, nachdem sie aufgerufen worden sind, die Kontrolle zum Programm zurückgeben. Nun muss nur noch mit MPI_Test überprüft werden, ob eine Nachricht verschickt worden oder angekommen ist. Der Prozess kann parallel zum Senden auch noch Berechnungen durchführen. Da in der Geschichte des High Performance Computings oft mit Matrizen gearbeitet wurde, gibt es auch spezielle Funktionen, die nur für diesen Zweck existieren. Es handelt sich um MPI_Gather und MPI_Scatter, welche es erlauben, Matrizen in Speicherblöcke zu unterteilen und an die jeweiligen Unterprozesse zu verteilen. Die Verwendung der Funktion ist nicht nur einfacher, sondern diese Funktionen sind auch schneller als ihr Send und Recv Gegenstück. Beim Relaxationsverfahren wird jedoch immer ein Teil der Vorgängermatrix mit in der Berechnung verwendet, sodass man die MPI Funktionen MPI_Gatherv und MPI_Scatterv benötigt. Im Gegensatz zum normalen Gather und Scatter kann man hier einen Offset angeben, um die Zerteilung der Matrix in Submatrizen besser beeinflussen zu können. So kann man die Matrix so zerlegen, dass man eine Reihe der Vorgängermatrix zur nächsten mitgibt. Dieser Ansatz wird auch in der momentanen Applikation verwendet. Besser ist, entweder MPI_Igatherv und MPI_Iscatterv zu verwenden, da diese Operationen nicht blockierend sind. Eine weitere Verbesserung würde man erreichen, wenn nicht ganze Submatrizen geschickt werden, sondern nur die Randwerte die vom Vorgänger berechnet wurden. Siehe Abbildung 2 (orange Markierung). Ist am Ende das Residuum erreicht, können die fertigen Submatrizen zusammengesetzt werden. Eine Anwendung mit nicht blockierenden Funktionen, wie MPI_Igatherv und MPI_Iscatterv, die nur die Randwerte übergibt, sollte in Speedup, Redundanz, Effizienz, Auslastung und Qualität nochmal etwas besser sein.

6 Verwendeter Anwendungscode

Der verwendeten Anwendungscode:
`https://github.com/JustARand0m/RelaxationMethod`