

Softwarepraktikum Parallele Numerik

Rebecca, Joshua, Alexander

2019-08-28

Inhaltsverzeichnis

1	Poisson-Gleichung und OpenMP	2
1.1	OpenMP und Testtools	2
1.1.1	Aufgabe 1 - Hello World, π & Mandelbrot	2
1.1.2	Aufgabe 2	5
1.2	Parallelisierung	8
1.2.1	Aufgabe 3	8
1.2.2	Aufgabe 4	10
1.3	Partielle Differentialgleichungen	12
1.3.1	Aufgabe 5	12
1.3.2	Aufgabe 6	14
2	Partielle Differentialgleichungen und Cuda	18
2.1	Cuda	18
2.1.1	Aufgabe 1	18
2.1.2	Aufgabe 2	18
2.2	Parallelisierung Partieller Differentialgleichungen	19
2.2.1	Aufgabe 3	19
2.2.2	Aufgabe 4	22
2.3	Heizprozesssimulation einer Herdplatte	23
2.3.1	Aufgabe 5	23

Chapter 1

Poisson-Gleichung und OpenMP

1.1 OpenMP und Testtools

1.1.1 Aufgabe 1 - Hello World, π & Mandelbrot

a)

Bei dem parallelen "Hello World"-Programm, bei dem jeder Thread seine Thread-ID ausgibt, ist nicht vorherzusehen in welcher Reihenfolge dies geschieht und kann sich bei jeder Ausführung ändern.

b)

Um die Kreiszahl π parallel mittels numerischer Integration zu berechnen, kann man in OpenMP die einzelnen Summen entweder manuell oder mittels Reduktion aufaddieren. Die Unterschiede in der Laufzeit wurden in Table 1.1 dokumentiert.

Gemessen wurde auf dem i82sn02. Für die manuelle Parallelisierung haben wir uns für atomic entschieden, da uns das intuitiv vorkam. Hier hat sich die Rechenzeit in der parallelen Ausführung jedoch stark erhöht, weshalb die Messung nach der Messung mit 4 Threads beendet wurde. Für das Parallelisieren mit Reduktion ist das maximale Speedup ~ 8 .

Bei 16 und 32 Threads verschlechtert sich die Beschleunigung im Gegensatz zu 8 Threads bei kleineren N. Das könnte am Overhead der Verarbeitung der Threads liegen.

-	Number of Threads	N	Time (s)	Speedup
sequential	1	1000	0.002	-
	1	10^7	0.239	-
	1	10^8	2.388	-
	1	10^9	23.642	-
	1	10^{10}	239.508	-
manual	2	1000	0.002	1
	2	10^7	0.634	-
	2	10^8	6.461	-
	2	10^9	52.989	-
	4	10	0.003	-
	4	10^7	1.214	-
	4	10^8	13.491	-
	4	10^9	~ 120	-
reduction	2	1000	0.002	1
	2	10^7	0.121	1.9
	2	10^8	1.188	2
	2	10^9	11.845	1.9
	2	10^{10}	118.204	2
	4	1000	0.003	-
	4	10^7	0.062	3.9
	4	10^8	0.595	4
	4	10^9	5.959	3.9
	4	10^{10}	59.183	4
	8	1000	0.003	-
	8	10^7	0.032	7.5
	8	10^8	0.299	7.9
	8	10^9	2.967	7.9
	8	10^{10}	29.689	8.0
	16	1000	0.003	-
	16	10^7	0.040	5.9
	16	10^8	0.318	7.5
	16	10^9	3.028	7,8
	16	10^{10}	29.674	8.1
	32	1000	0.004	-
	32	10^7	0.037	6.4
	32	10^8	0.305	7,8
	32	10^9	2.987	7.9
	32	10^{10}	29.551	8.1

Table 1.1: OpenMP Parallelisierung mit verschiedenen Threadanzahlen und Synchronisationsmechanismen

Auflösung N	No. Threads	Time (s)	Speedup
1000	1	4.411	-
1000	2	2.228	1.980
1000	4	1.122	3.931
1000	8	0.563	7.835
1000	16	0.567	7.780
2000	1	17.625	-
2000	2	8.896	1.981
2000	4	4.493	3.923
2000	8	2.254	7.819
2000	16	2.248	7.840
4000	1	71.761	-
4000	2	35.559	2.010
4000	4	17.952	3.997
4000	8	8.978	7.991
4000	16	8.980	7.991
5000	1	110.143	-
5000	2	55.564	1.982
5000	4	27.951	3.94
5000	8	13.995	7.870
5000	16	14.020	7.856

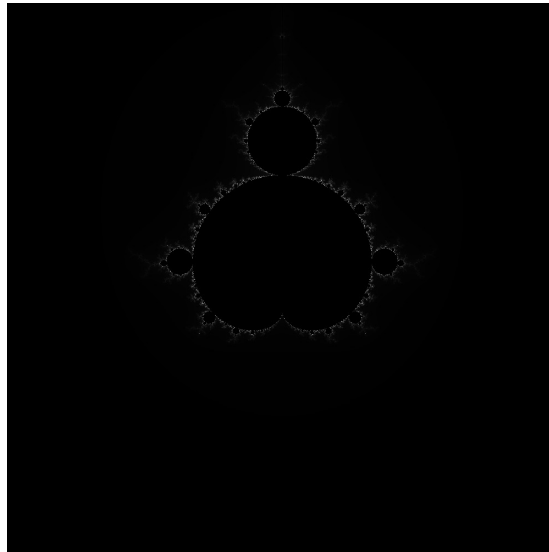
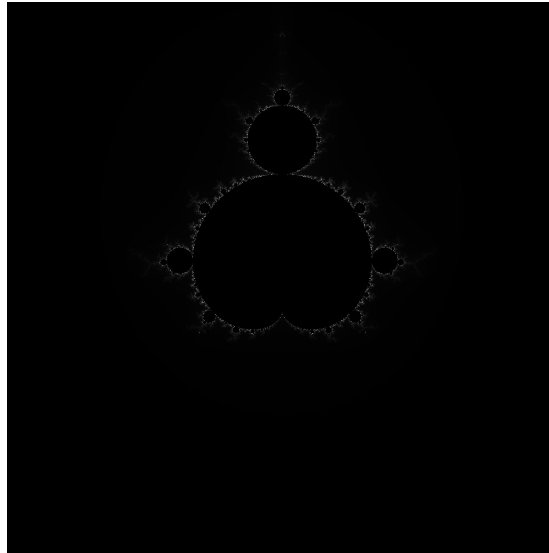
Table 1.2: Messungen zu Aufgabe 1c

c)

Die Berechnung der Mandelbrot-Menge kann parallelisiert werden, indem man die äußerste for-Schleife der Berechnung parallelisiert, mit privaten komplexen Zahlen z und $kappa$ und privater innerster Schleifenvariable k für jeden Thread. Die Laufzeiten wurden für die Auflösungen 1000, 2000, 4000 und 5000 gemessen mit jeweils 1, 2, 4, 8 und 16 Threads. Gemessen wurde wieder auf dem i82sn02 und die Ergebnisse sind in Tabelle 1.2 zu sehen..

Die Werte für das Speedup sind sehr nah an den gemessenen Werten in Aufgabenteil b).

Die folgenden Bilder zeigen verschiedene Outputs der Berechnung der Mandelbrotmenge. Bei beiden Bildern wurde die Auflösung 1000 berechnet. Das erste Bild zeigt den Output der sequentiellen Berechnung während das zweite Bild die Ausgabe der parallelen Berechnung zeigt.



1.1.2 Aufgabe 2

a)

1. Example:

Es besteht eine Race-Condition auf dem Array a über Index i , bei ungünstiger Ausführungsreihenfolge der Threads kann es dazu kommen, dass $a[i]$ durch einen Thread geschrieben wird und von einem anderen Thread versucht wird, darauf in der nachfolgenden Anweisung lesend zuzugreifen. Um dieses Problem zu lösen können zwei for-Schleifen verwendet werden. Eine

schreibt alle Werte in Array a und die zweite schreibt alle Werte unter Zugriff auf Array a in Array b. Beide Schleifen werden jeweils getrennt parallelisiert.

2. Example:

Threads existieren hier in der gesamten parallelen Region. Das `nowait`-Statement der ersten parallelisierten For-Schleife bewirkt, dass Threads bereits die nächste parallelisierte Schleife bearbeiten können, wenn sie ihren Teil der ersten Schleife fertig verarbeitet haben. Dies führt zu einer Race-Condition auf das Array a mit Index i, ähnlich zu Example 1. Mit dem entfernen des `nowait`-Statements der ersten for-Schleife warten alle Threads wieder auf die implizite Barriere bis alle Threads fertig sind und es kann problemlos auf die geschriebenen Werte in Array a zugegriffen werden.

3. Example:

Hier ist die Variable x zunächst global definiert und wird somit implizit zwischen den Threads geteilt. Somit besteht eine Race-Condition auf x da die Threads unabhängig voneinander sowohl lesend als auch schreibend auf x zugreifen. Indem man x explizit als `private` deklariert, erhält jeder Thread seine eigene Kopie der Variable und es besteht keine Race-Condition mehr.

4. Example:

f ist global definiert und wird durch jeden Thread `private` gesetzt. Allerdings erhält jeder Thread eine uninitialisierte Kopie der Variable f, was bedeutet dass initialisieren mit dem Wert 2 vor der Schleife nur für den Master-Thread sichtbar ist. Um diese Initialisierung auch für alle übrigen Threads sichtbar zu machen, ist es erforderlich, f mittels `firstprivate` zu deklarieren. Des Weiteren wird der Wert von x nicht aus der parallelen Region wieder rausgeschrieben, sondern gelöscht. Wodurch der zuletzt geschriebene Wert innerhalb der parallelen Region in x nach außen hin nicht sichtbar ist. Um dieses Verhalten zu erreichen, muss x als `lastprivate` deklariert werden.

5. Example:

Hier besteht eine Race-Condition auf die Variable sum. Da mehrere Threads versuchen hier den alten Wert von sum zu lesen, den i-ten Wert aus b auszuaddieren und damit sum wieder zu überschreiben, ist nicht garantiert, dass sum am Ende den korrekten Wert enthält. Indem man das Auslesen, aufaddieren und zuweisen mittels `critical`-Konstrukt schützt, ist garantiert dass immer nur ein Thread den Wert überschreiben kann. Allerdings ist die gesamte Schleife dann gleichbedeutend mit einer sequenziellen Ausführung.

Mode	N	Seq	2 Threads		4 Threads		16 Threads	
			Zeit	Speedup	Zeit	Speedup	Zeit	Speedup
gcc - IJK	10	0.0040	0.0040	1.00	0.0040	1.00	0.0040	1.00
gcc - IJK	100	0.0190	0.0140	1.36	0.0090	2.11	0.0070	2.71
gcc - IJK	1000	8.8990	4.5550	1.95	2.3770	3.74	0.7240	12.29
gcc - IKJ	10	0.0040	0.0040	1.00	0.0040	1.00	0.0040	1.00
gcc - IKJ	100	0.0166	0.0122	1.36	0.0088	1.89	0.0060	2.77
gcc - IKJ	1000	7.3744	7.8727	0.94	3.8862	1.90	1.6994	4.34
gcc - ... +Inv	10	0.0030	0.0030	1.00	0.0030	1.00	0.0040	0.75
gcc - ... +Inv	100	0.0160	0.0116	1.38	0.0090	1.78	0.0056	2.86
gcc - ... +Inv	1000	6.0300	3.1956	1.89	1.7114	3.52	0.5086	11.86
gcc - ... +O2	10	0.0040	0.0036	1.11	0.0040	1.00	0.0042	0.95
gcc - ... +O2	100	0.0060	0.0062	0.97	0.0050	1.20	0.0050	1.2
gcc - ... +O2	1000	6.0412	3.1912	1.89	1.7174	3.52	0.5078	11.9

Table 1.3: Verschiedene Varianten der Matrixmultiplikation

b)

Werden Matrizen in C zeilenweise im Speicher hinterlegt, ist es zur optimalen Ausnutzung von Caching-Effekten ideal, wenn auch zeilenweise über Einträge der Matrix iteriert wird. Dadurch lädt der Prozessor bei einem Cache-miss auf einen Eintrag nicht nur den einzelnen Eintrag in den Cache, sondern direkt mehrere nachfolgende Einträge, was weniger Zugriffe auf den RAM zu Folge hat und somit insgesamt die Performance verbessert. Wird jedoch spaltenweise über eine Matrix iteriert, müssen für einen Zugriff auf das (i+1)-te Element alle n-Einträge nach i ausgelassen werden. Bei entsprechend großen Matrizen werden die mit in den Cache geladenen Elemente gar nicht benötigt. Jeder Zugriff entspricht somit einem Cache-miss, was wiederum einen Speicherzugriff zur Folge hat und somit die gesamte Performance negativ beeinflusst.

Der klassische IJK-Algorithmus zur Matrixmultiplikation iteriert in der inneren K-Schleife für die rechte Matrix B für jede Addition+Multiplikation über die Spalten. Zum besseren Ausnutzen der Cache-Effekte kann durch Vertauschen der zwei Schleifen J und K, auch bekannt als IKJ-Optimierung, eine Verbesserung der Performance erzielt werden. So wird in der inneren J-Schleife nur über die Spalten aus B und C iteriert und die K-Schleife iteriert über die Spalten in A.

Da der einzelne Eintrag in A der inneren Schleife invariant ist, kann dieser zudem aus der Schleife herausgezogen und in einer temporären Variable gespeichert werden.

Die Messungen und Speedup für verschiedene Threadanzahlen sieht man in Table 1.3.

c)

Die Arbeitspakete pro Thread werden statisch vergeben. Die Berechnung der Mandelbrotmenge ist jedoch nur für einen bestimmten Bildbereich besonders

Mode	N, It., Chunk	Seq.	2 Threads	S.Up	4 Threads	S.Up	8 Threads	S.Up
static	2000, 500, -	8.528	6.188	1.38	5.453	1.56	3.908	2.18
static	4000, 500, -	34.080	24.881	1.37	22.193	1.54	15.061	2.26
static	8000, 500, -	135.882	98.576	1.38	86.120	1.56	60.522	2.25
dynamic	2000, 500, 1	8.524	4.375	1.95	2.284	3.73	1.216	7.01
dynamic	4000, 500, 1	33.955	17.016	2.00	8.600	3.95	4.803	7.07
dynamic	8000, 500, 1	135.966	67.987	2.00	34.344	3.96	19.110	7.11

Table 1.4: Auswirkung der Chunk-Size auf die Performance

Chunk	N	Seq.	2 Threads	4 Threads	8 Threads
1	8000	135.966	67.987	34.569	19.042
2	8000	-	68.069	34.569	19.049
4	8000	-	69.999	34.471	19.049
8	8000	-	68.027	34.533	19.096
16	8000	-	67.996	34.892	19.134

Table 1.5: Stabile Performance unabhängig von der Chunk-Size

Berechnungsintensiv. Threads welche diese Bereiche berechnen sollen, müssen mehr Aufwand betreiben im Vergleich zu den anderen Threads, wodurch diese insgesamt länger benötigen. Die anderen Threads werden kaum ausgelastet und die gesamte Parallelisierung ist ineffizient. Durch dynamisches Scheduling werden die Arbeitspakete erst an Threads verteilt, wenn diese keine Aufgabe haben oder eine bereits beendet haben. Dies führt zu einer optimaleren Verteilung der Arbeitslast und insgesamt schnelleren Berechnung des Gesamtproblems, da alle Threads möglichst gleich viel Arbeiten. Siehe Table 1.4.

Veränderungen in der Chunk-Size beim Scheduling wirken sich in diesem Fall nicht auf die Performance aus. Dies sieht man in Table 1.5.

1.2 Parallelisierung

1.2.1 Aufgabe 3

a)

$P(x)$: Anzahl auszuführender Operationen auf x Prozessoren.

$T(x)$: Ausführungszeit auf x Prozessoren.

Speedup: $S(n) = T(1)/T(n)$.

Der Zusammenhang zwischen serieller und paralleler Ausführungszeit eines Programmes. Der Wertebereich ist $1 \leq S(n) \leq n$.

Effizienz: Die Effizienz $E(n) = S(n)/n$ gibt die relative Verbesserung der

-	GPUs	CPUs	FPGAs
Energieeffizienz	Gut	Mittel	Gut
Anwenderfreundlichkeit	<ul style="list-style-type: none"> • Braucht Einarbeitungszeit. • Es gibt Bibliotheken. 	<ul style="list-style-type: none"> • Am einfachsten zu programmieren. • Viele Bibliotheken vorhanden. • Kurze Compilierzeit. 	<ul style="list-style-type: none"> • Aufwändig zu programmieren. • Lange Compilierzeit. • Wenig Bibliotheken.

Table 1.6: Bewertung verschiedener Programmierplattformen

Verarbeitungsgeschwindigkeit an.

Auslastung: $R(n)/(n \cdot T(n))$. Gibt an, wie viele Operationen (Tasks) jeder Prozessor im Durchschnitt pro Zeiteinheit ausgeführt hat.

Mehraufwand: $R(n) = P(n)/P(1)$. Beschreibt den bei einem Multiprozessorsystem erforderlichen Mehraufwand für die Organisation, Synchronisation und Kommunikation der Prozessoren.

b)

Race-Conditions: Wenn zwei Threads unabhängig voneinander auf eine Ressource lesend oder auch schreibend zugreifen können, spricht man von einer Race-Condition. Hierbei kann es bspw. beim Zugriff auf Variablen bei ungünstiger Ausführungszeiten dazu kommen, dass am Ende der Berechnungen ein falscher Wert in der Variable enthalten ist, als wenn die Berechnung sequentiell ausgeführt worden wäre. Um eine Race-Condition zu vermeiden, können die kritischen Abschnitte in der Art und Weise gesichert werden, dass immer nur ein Thread gleichzeitig innerhalb des kritischen Abschnitts sein darf.

In diesem Zusammenhang kann es auch zu Deadlocks, Lifelocks oder auch Starvation kommen.

c)

In diesem Praktikum verwenden wir CPUs und GPUs da sie universeller einsetzbar sind als FPGAs und die Programmierung deutlich leichter ist. Eine ausführlichere Bewertung ist in Table 1.6 aufgeführt.

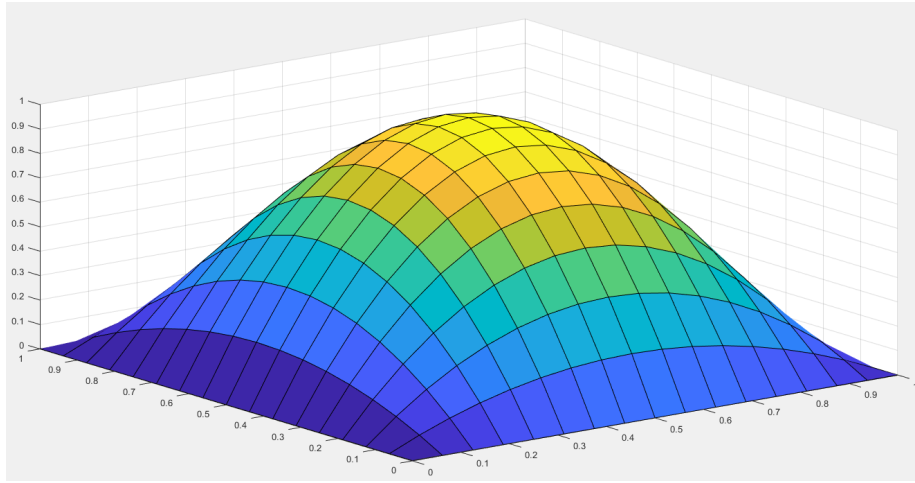


Figure 1.1: $l=4$; $h=1/16$;

-	$l=4$; $h=1/16$	$l=5$; $h=1/32$	$l=6$; $h=1/64$
	0.050s	1.429s	37.296s
	0.041s	1.426s	37.014s
	0.040s	1.392s	37.061s
	0.037s	1.428s	37.416s
	0.051s	1.424s	37.295s
	=0.044s	=1.420s	=37.216s

Table 1.7: Laufzeiten mit verschiedenen l -Werten

1.2.2 Aufgabe 4

a)

Alle Programmdurchläufe für die Messungen hatten eine Fehlerschranke von 0.000001. Die Laufzeiten bei verschiedenen l und daraus resultierenden h Werten sieht man in Table 1.7.

Diese Lösungswerte in Figure 1.1, 1.2 und 1.3 ergeben sich mit $l=4,5,6$;

b)

Die Schleifenabläufe sind direkt voneinander abhängig und müssen in Reihenfolge ablaufen. Die naive Parallelisierung liefert keine korrekten Ergebnisse. Alternativ könnte man mit einer naiven Parallelisierung auch Parallelisierung der beiden inneren Summen meinen. Diese hätte zur Folge, dass man nur bis zu maximal vier parallele Threads hat, welche jeweils nur einen Wert aufaddieren.

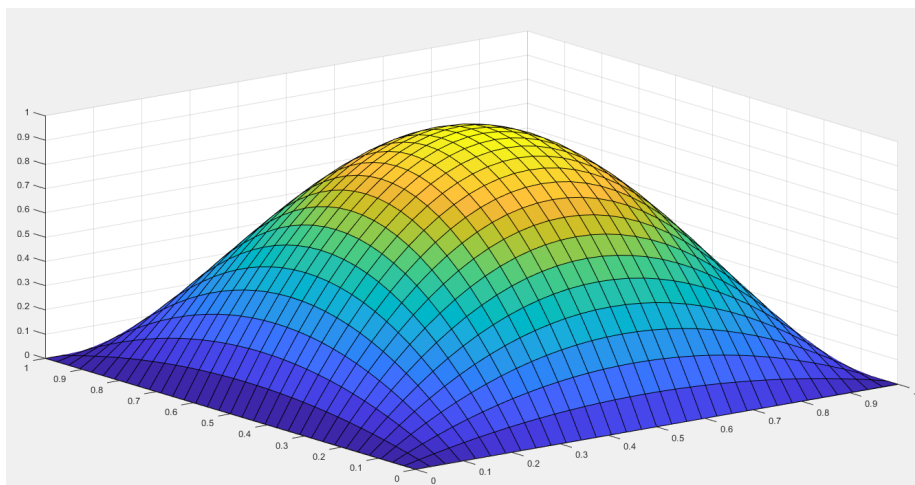


Figure 1.2: $l=5$; $h=1/32$;

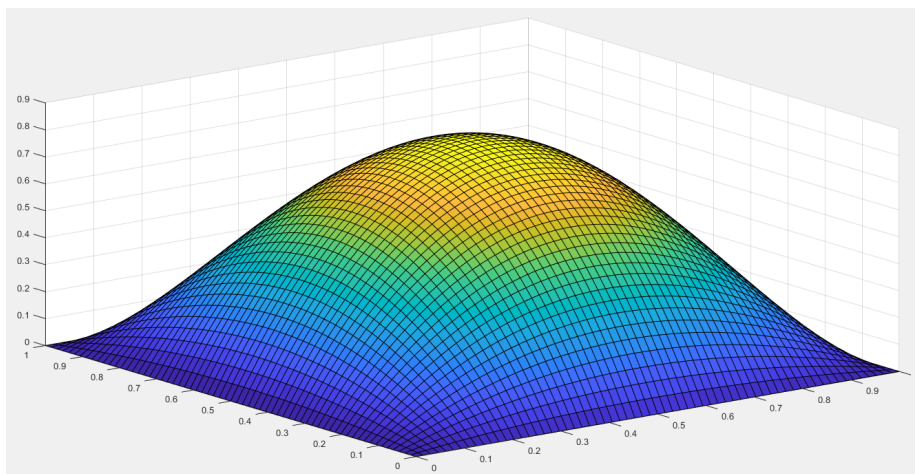


Figure 1.3: $l=6$; $h=1/64$;

	Laufzeit(sequentiell)	Laufzeit(parallel)	SpeedUp	Efficiency
$l=4; h=1/16$	0.044s	0.242s	0.182	0.004
$l=5; h=1/32$	1.420s	1.154s	1.231	0.026
$l=6; h=1/64$	37.216s	26.884s	1.384	0.029

Table 1.8: Laufzeiten und Speedup des parallelisierten Gauß-Seidel-Verfahren

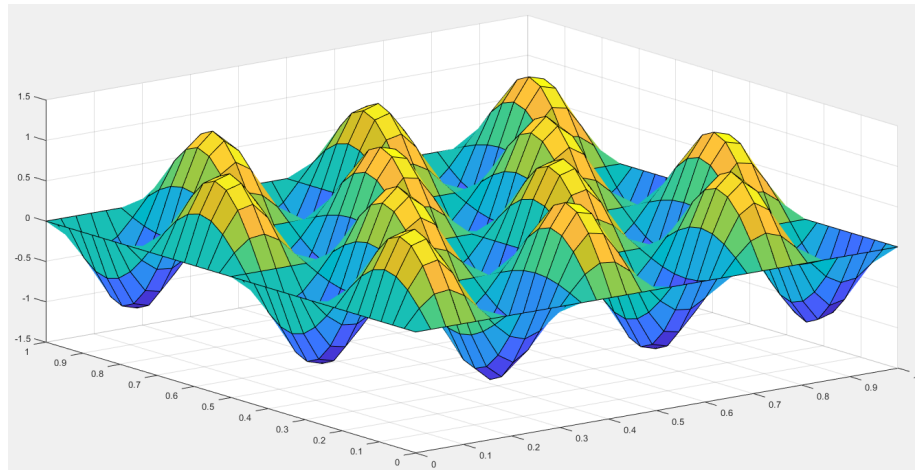


Figure 1.4: $l=5; h=1/32$;

c)

Methodik: Es gibt verschiedene Wege der effizienteren Parallelisierung der Werte. Man kann die zu berechnenden Werte zeilen- oder spaltenweise in Gruppen einteilen, sowie auch in einem Schachbrettmuster. Dann berechnet man die verschiedenen Gruppen parallel. Dies wird umgangssprachlich "Rot-Schwarz" genannt und konvergiert zum korrekten Ergebnis. Eine andere Variante der Berechnung ist die sogenannte "Wavefront". Dabei werden die Werte diagonal ausgerechnet. Dadurch entsteht eine ungleiche Last über die Ausführungszeit, welche jedoch ausgeglichen werden kann, indem man die Matrix zusätzlich in Blöcke einteilt. Den Vergleich zwischen der sequentiellen und parallelisierten Version sieht man in Table 1.8.

1.3 Partielle Differentialgleichungen

1.3.1 Aufgabe 5

a)

$f(x,y)$ muss stetig definiert sein auf Ω
 Γ ist der Rand von Ω und lipschitzstetig

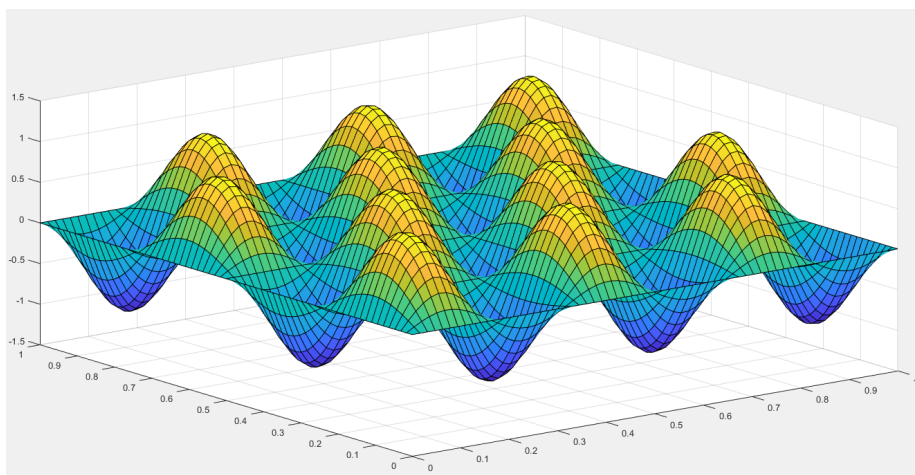


Figure 1.5: $l=6$; $h=1/64$;

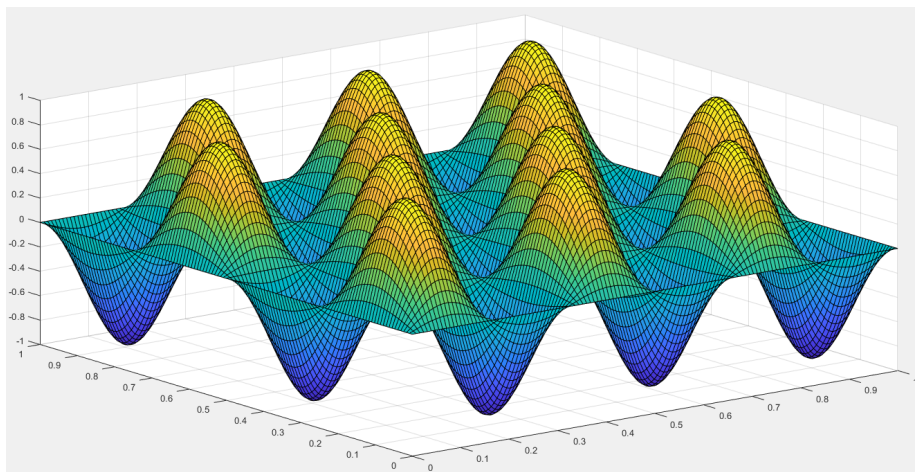


Figure 1.6: $l=7$; $h=1/128$;

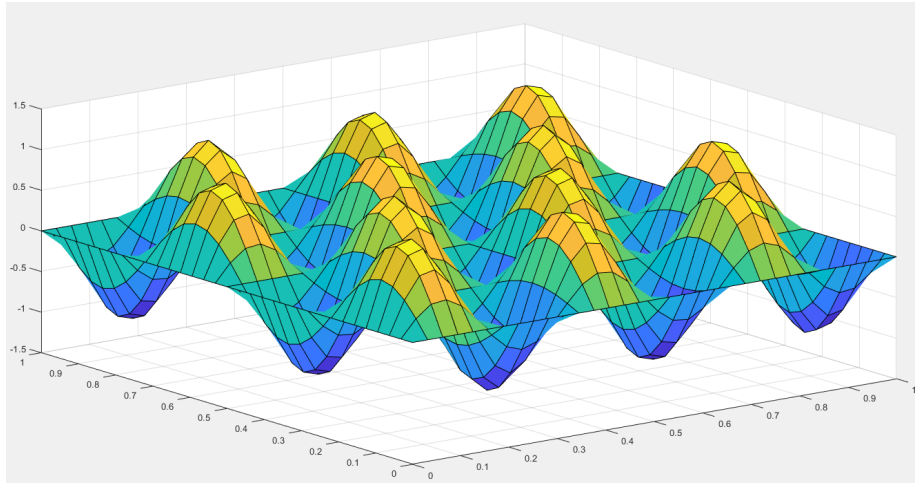


Figure 1.7: GMRES; $l=5$; $h=1/32$;

Lösung $u(x,y)$ muss zweifach differenzierbar sein

b)

$$f(x,y) = (N^2 + M^2) * 4 * \pi^2 * \sin(2 * M * \pi * x) * \sin(2 * N * \pi * y)$$

c)

Es handelt sich um eine h-FEM.

Die Lösungswerte in Figure 1.4, 1.5 und 1.6 ergeben sich mit $M=3$; $N=2$; $l=5,6,7$;

1.3.2 Aufgabe 6

a)

Liste: CG Verfahren; PCG-Verfahren; Verfahren der minimalen Residuen(GMRES); GCR-Verfahren; Arnoldi-Verfahren; FOM, ORTHORES;

Das Gleichungssystem ist dünnbesetzt und alle Krylow-Unterraumverfahren sind gut geeignet für dünnbesetzte Gleichungssysteme. Diese Lösungswerte in Figure 1.7, 1.8 und 1.9 ergeben sich mit der Gleichung aus Aufgabe 5 und $M=3$; $N=2$; $l=5,6,7$;

b)

Die Verwendung des Residuums als Abbruchbedingung ist eventuell problematisch, da man dadurch in jedem Schleifendurchlauf einen Lösungsvektor x^k

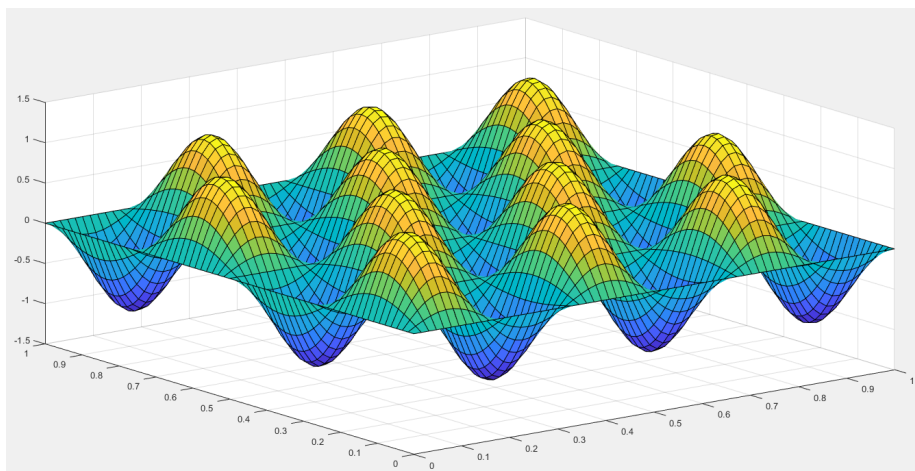


Figure 1.8: GMRES; $l=6$; $h=1/64$;

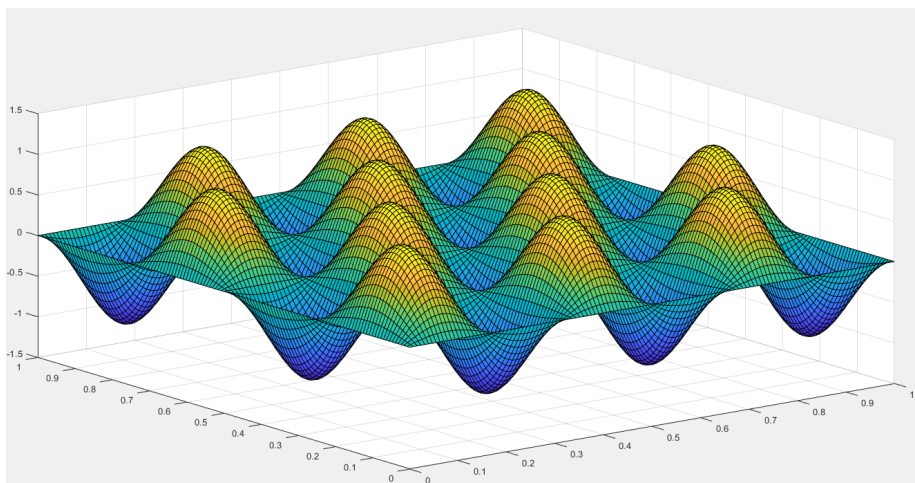


Figure 1.9: GMRES; $l=7$; $h=1/128$;

-	Aufgabe 5(s)	Aufgabe 6(s)	Speedup
l=5; h=1/32	+1.450	+0.028	56.96
	+1.420	+0.021	
	+1.371	+0.026	
	+1.475	+0.026	
	+1.408	+0.026	
	=1.424	=0.025	
l=6; h=1/64	+15.427	+0.140	98.488
	+12.546	+0.148	
	+12.561	+0.089	
	+12.766	+0.142	
	+13.081	+0.155	
	=13.276	=0.135	
l=7; h=1/128	+107.183	+1.289	81.113
	+105.581	+1.188	
	+104.562	+1.175	
	+105.834	+1.703	
	+104.077	+1.148	
	=105.447	=1.300	

Table 1.9: Laufzeiten und Speedup des GMRES gegenüber dem parallelisierten Gauß-Seidel-Verfahren

berechnen muss. Man kann sich eine feste Anzahl von Iterationen setzen, allerdings hat man dann keine garantierte Genauigkeit. Außerdem kann man das Residuum mit weniger Rechenaufwand in jeder Iteration abschätzen und diesen Schätzwert als Abbruchbedingung nutzen. Auch diese Vorgehensweise hat keine garantierte Genauigkeit, da die Schätzgenauigkeit variiert. Im Zuge des Praktikums haben wir bis jetzt nur eine stabile Version mit abgeschätztem Residuum implementiert(gmres.c). Die Version mit dem Residuum als Abbruchbedingung funktioniert nicht verlässlich(gmresResiduum.c).

c)

Die Efficiency ist bei jedem Vergleich zwischen Aufgabe 6 und Aufgabe 5c gleich dem Speedup, da die Anzahl N der Kerne auf denen das Programm läuft nicht variiert. Der Speedup ist in Table 1.9 dokumentiert.

Einsatz eines Vorkonditionierers: Der Einsatz ist unsinnvoll, da wir bereits eine dünn besetzte Matrix gegeben haben. Falls die Matrix anders besetzt wäre, könnte ein Vorkonditionierer von Nutzen sein.

d)

MFEM; deal.II; libMesh; JuliaFEM; FEniCS; Hermes Project;...

Liste weiterer FEM Packages/Libraries: https://en.wikipedia.org/wiki/List_of_finite_element_software_packages

Fast alle sind Open Source und kostenlos nutzbar. "Hermes Project" scheint zum Beispiel eine leicht nutzbare C/C++ Bibliothek.

Chapter 2

Partielle Differentialgleichungen und Cuda

2.1 Cuda

2.1.1 Aufgabe 1

Es gibt sehr viele Eigenschaften die man auslesen kann. Hier sind mögliche Ausgaben:

```
Number of Devices: 2
Device Number: 0
Device Name: Tesla K80
Memory Clock Rate (KHz): 2505000
Memory Bus Width (bits): 384
Global Memory size (bytes): 11996954624
Device Number: 1
Device Name: Tesla K80
Memory Clock Rate (KHz): 2505000
Memory Bus Width (bits): 384
Global Memory size (bytes): 11996954624
```

2.1.2 Aufgabe 2

Arraygröße Bei kleinen Arrays ist die Transferrate auf der GPU nicht besser als auf der CPU. Das ändert sich aber mit wachsender Arraygröße, denn dann ist die Rate "Device to Device" deutlich besser als "Host to Host". Hier wurde mit nvprof gemessen. Siehe Table 2.1.

N	Device to Host	Host to Device	Device to Device	Host to Host
10	2.1120us	2.0160us	3.7440us	3.000us
100	2.3360us	2.0800us	3.8400us	4.000us
1000	2.6880us	2.5280us	3.5200us	3.000us
10000	7.4240us	10.368us	3.2640us	22us
100000	55.679us	58.240us	7.7760us	156us
1000000	624.25us	553.79us	74.015us	2.171ms
10000000	5.4335ms	5.3853ms	621.02us	25.503ms
100000000	53.996ms	54.383ms	6.1003ms	266.338ms

Table 2.1: Vergleich der Transferraten zwischen Device und Host

Werte um X erhöhen Im folgender Messung wurde in dem Array jeder Wert um 1 erhöht. Die Laufzeit wurde wieder mit nvprof gemessen und in Table 2.2 dokumentiert.

Speedup Wenn die CPU jeden Wert des Vektors um 1 erhöht (mit OpenMP) ergeben sich folgende Laufzeiten:

- N = 10000000: Im Schnitt 49.6942 ms (zwischen 194 und 440 ms)
- N = 100000000: Im Schnitt 488.84 ms

Verglichen mit Table 2.1 ergeben sich die in Table 2.3 aufgeführten Speedups für die verschiedenen Blockgrößen.

2.2 Parallelisierung Partieller Differentialgleichungen

2.2.1 Aufgabe 3

a)

Running with 32 threads.
 $h = 0.015625$, $n = 63$, $l = 6$
6.725816

Es wurde eine Implementierung des Jakobi-Iterationsverfahren zur Lösung des Gauss-Seidel-Verfahrens implementiert. Hier im Vergleich befindet sich die äußerst naive Implementierung, in welche über die Summen des Gauss-Seidel-Verfahrens mittels OMP parallelisiert wurde, der bereits erwähnten Jakobi-Iteration mittels OMP und eine Implementierung der Jakobi-Iteration auf der CUDA-GPU.

Die Messung sind in Table 2.4 und der Speedup ergibt sich aus dem Vergleich mit der sequentiellen Laufzeit des naiven Ansatzes.

blocksize	N	Time
4	10000	14.752us
	100000	121.47us
	1000000	1.4293ms
	10000000	14.122ms
	100000000	141.23ms
8	10000	8.9920us
	100000	62.207us
	1000000	7.0312ms
	100000000	70.966ms
16	10000	5.8880us
	100000	32.896us
	1000000	357.12us
	10000000	3.5412ms
	100000000	35.236ms
24	10000	4.8000us
	100000	23.936us
	1000000	251.20us
	10000000	2.5063ms
	100000000	24.878ms
32	10000	4.0960us
	100000	17.920us
	1000000	184.29us
	10000000	1.8227ms
	100000000	18.020ms

Table 2.2: Arraywerte um 1 erhöht

blocksize	Speedup	
	N = 10 ⁷	N = 10 ⁸
4	3.52	3.49
8	7.07	6.89
16	14.03	13.87
24	19.82	19.64
32	27.19	27.74

Table 2.3: Speedup mit verschiedenen Block-Größen

Implementierung	l,h	Laufzeit(seriell)	Laufzeit(parallel)	Speedup
OMP naiv	l=4; h=1/16	0.044s	0.242s	0.182
OMP naiv	l=5, h=1/32	1.42	1.154	1.231
OMP naiv	l=6; h=1/64	37.216s	26.884s	1.384
OMP Jacobi	l=4; h=1/16	-	0.391s	0.113
OMP Jacobi	l=5, h=1/3	-	1.475s	0.963
OMP Jacobi	l=6, h=1/64	-	6.394s	5.82

Table 2.4: Vergleich - Paralleles mit sequentiellm Gauß-Seidel-Verfahren

Implementierung	l,h	It./Block	Laufzeit(parallel)	Speedup (vs. Seq.)
CUDA synch.	4, 1/16	-	1.020	0.043
CUDA synch.	5, 1/32	-	1.055	1.346
CUDA synch.	6, 1/64	-	1.318	28.237
CUDA async	4, 1/16	10	0.989	0.044
CUDA async	4, 1/16	15	0.973	0.045
CUDA async	4, 1/16	20	0.950	0.046
CUDA async	4, 1/16	25	0.976	0.045
CUDA async	5, 1/32	10	0.986	1.440
CUDA async	5, 1/32	15	0.979	1.450
CUDA async	5, 1/32	20	1.002	1.417
CUDA async	5, 1/32	25	1.018	1.395
CUDA async	6, 1/64	10	1.088	34.206
CUDA async	6, 1/64	15	1.108	33.588
CUDA async	6, 1/64	20	1.077	34.555
CUDA async	6, 1/64	25	1.056	35.242

Table 2.5: Vergleich - implementiertes synchrones Jacobi-Verfahren mit block-asynchronem Verfahren

b)

Block-asynchrone Relaxation ist eine Beschleunigungstechnik bei welcher das Jacobi-Iterationsverfahren asynchron auf einem Teilblock der Gesamtlösung ausgeführt wird. Dabei wird zunächst die Lösungsmatrix in mehrere Teilblöcke unterteilt und ein Thread-Block holt sich einen Block inklusive direkter Nachbarn in den Shared-Memory. Auf diesem Block wird dann asynchron eine bestimmte Anzahl an Jacobiiterationen durchgeführt und im Anschluss erst der Global-Memory geupdatet. Die Besonderheit ist das hier zwischen den Threadblöcken keinerlei Synchronisation stattfindet.

In Table 2.5 wird das block-asynchrone Verfahren mit einer synchronen Implementierung des Jacobi-Verfahrens verglichen.

c)

2.2.2 Aufgabe 4

a)

Als Indexmenge eignen sich Indizes für i von 0 bis $n-1$ und für j jeweils Indizes von i bis $i+n$. Man braucht nicht mehr Indizes solange die Matrix A die Struktur aus den vorherigen Aufgaben hat. Falls die Matrix voll besetzt ist sollte man i zwischen 0 und $n-1$ variieren und j zwischen i und $n-1$.

b)

Nein, eine Implementierung mit OpenMp wäre nicht geeigneter. Gerade bei der Matrix der vorherigen Aufgaben (Finite Differenzen Matrix) kann man die Indizes in n gleichgroße Arbeitspakete aufteilen, bei OpenMP hat man aber meist sehr viel weniger Kerne. Das ganze verfahren lässt sich mit Cuda sehr gut parallelisieren. Man kann sogar einen Thread pro Index kombination im GPU-Kernel nutzen und hat so weit mehr parallel laufenden Code.

c)

Bei der Linksvorkonditionierung multipliziert man die Matrix $(LU)^{-1}$ von links an die Gleichung die vorkonditioniert werden soll. Dadurch muss man zum Beispiel beim GMRES Verfahren das Residuum beim starten und in jedem Schritt neu bestimmen mit $LUr = r'$. Bei der Rechtsvorkonditionierung multipliziert man $(LU)^{-1} * LU$ von rechts. Dadurch braucht man zusätzlich noch die Vorkonditionierer, wenn man am Ende den Lösungsvektor bestimmt. Die beiden Vorkonditionierungsmethoden sind gut, aber ich würde Linksvorkonditionierung empfehlen, da man dadurch die Vorkonditionierungsmatrizen nicht noch beim Lösungsaufbau braucht. Außerdem wird Linksvorkonditionierung mehr verwendet und dadurch findet man mehr Hilfestellungen und Beispiele im Internet.

d)

Implementierung: Linksvorkonditioniertes GMRES-Verfahren

Vergleichsimplementierung: GMRES-Verfahren

Das vorkonditionierte Verfahren konvergiert schneller als das gewöhnliche GMRES-Verfahren. Allerdings ist die Laufzeit trotzdem je nach der gewählten Genauigkeit langsamer als im gewöhnlichen GMRES. Dies liegt daran, dass die Bestimmung des Vorkonditionierers am Anfang und dessen Anwendung in jeder Iteration die Laufzeit verlängert. Mit $l=5$ gilt, wenn der globale Fehler kleiner als $0.8 \cdot 10^{-5}$ ist ist die vorkonditionierte Version schneller. Dieser Fehlergrenzwert erhöht sich mit der Erhöhung von l .

Die Messungen wurden mit $l=5$ $M=3$ $N=2$ und variierender Fehlerschranke f durchgeführt. Die Messwerte können in Table 2.6 abgelesen werden. In den Datenzellen steht jeweils die Anzahl der Iterationen und durch ein Semikolon

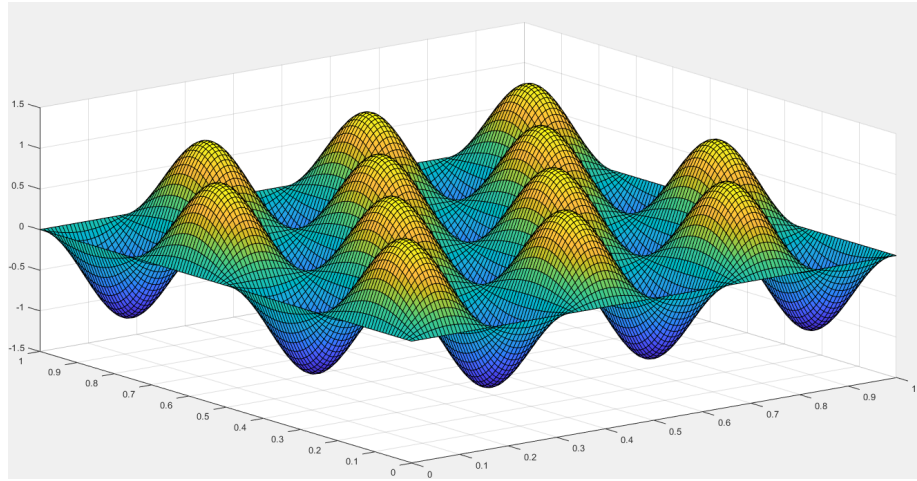


Figure 2.1: Poisson-Problem: Lösung

Algorithmus	$f=10^{-5}$	10^{-6}	$9 * 10^{-7}$	$8 * 10^{-7}$	$7 * 10^{-7}$
GMRES	33;0.423	173;1.181	213;1.378	269;1.953	351;2.683
Vorkonditioniert	18;0.725	21;1.760	29;1.921	36;2.111	36;2.106

Table 2.6: Vergleich zwischen GMRES und vorkonditioniertem GMRES

getrennt die Laufzeit in Sekunden.

2.3 Heizprozesssimulation einer Herdplatte

2.3.1 Aufgabe 5

Mathematische Grundlage

Im folgenden Abschnitt werden folgende Funktionen und Parameter verwendet:

$u(t_n)$:= Wärme an allen räumlichen Diskretisierungspunkten zum Zeitpunkt

t_n

$u'(t_n)$:= Wärmeänderungen an allen räumlichen Diskretisierungspunkten zum Zeitpunkt t_n

$u_{i,j}(t_n)$:= Wärme am räumlichen Diskretisierungsschritt (i,j) zum Zeitpunkt t_n

$u'_{i,j}(t_n)$:= Wärmeänderungen am räumlichen Diskretisierungsschritt (i,j) zum Zeitpunkt t_n

$u(x,y,t_n)$:= Wärme am räumlichen Punkt (x,y) zum Zeitpunkt t_n

$u'(x,y,t_n)$:= Wärmeänderungen am Punkt (x,y) zum Zeitpunkt t_n

f := Heizfunktion an allen räumlichen Diskretisierungspunkten (zeitlich invari-

$\text{ant)} f(x, y) = 32 * [x * (x - 1) + y * (y - 1)]$
 $h_t :=$ Zeitlicher Diskretisierungsparameter (Zeitschrittweite)
 $h_s :=$ Räumlicher Diskretisierungsparameter (Schrittweite zwischen den räumlichen Diskretisierungspunkten)
 $a :=$ Temperaturleitfähigkeit des Materials
 $A :=$ Matrix mit besonderer Struktur für Finite Differenzen
 $t_n :=$ Zeitpunkt nach n Zeitschritten
 $t_0 :=$ Startzeitpunkt
 $N :=$ Menge der natürlichen Zahlen

Wir benutzen die nicht homogene Wärmeleitungsgleichung, da wir nicht nur die Wärmeverteilung simulieren wollen, sondern mit der Funktion f Wärme hinzugefügt werden soll. Damit lautet die Gleichung:

$$u'(t) - a * \Delta u(t) = f$$

Damit unser Modellproblem korrekt gestellt ist definieren wir folgendes:

$$\begin{aligned}
 -\Delta u(x, y, t_n) &= \frac{f(x, y) - u'(x, y, t_n)}{a}, (x, y) \in \Omega = (0, 1)^2, n \in N \setminus 0; \\
 u(x, y, t_n) &= 20, (x, y) \in \Gamma, n \in N; \\
 u(x, y, t_0) &= 20, (x, y) \in \Omega; \\
 f(x, y) &\text{ ist stetig, } (x, y) \in \Omega.
 \end{aligned}$$

Damit sind unsere Rand- und Anfangsbedingungen gesetzt. Wir haben also $u(t_0)$ gegeben und kennen f . Damit können wir $u'(t_0)$ bestimmen indem wir unsere Basisgleichung umstellen.

$$\begin{aligned}
 u'(t_n) &= f + a * \Delta u(t_n) \\
 \text{mit } \Delta u(t_n) &= \frac{4 * u_{i,j}(t_n) - u_{i-1,j}(t_n) - u_{i+1,j}(t_n) - u_{i,j-1}(t_n) - u_{i,j+1}(t_n)}{h_s^2}
 \end{aligned}$$

Im Fall von unserem $u(t_0)$ ist $\Delta u(t_n) = 0$, da es keine Hitzeunterschiede in $u(t_0)$ gibt. Anschließend können wir $u(t_1)$ bestimmen und damit den Zeitschritt vorziehen. Dafür eignet sich das explizite Eulerverfahren und als Gleichung ergibt sich:

$$u(t_{n+1}) = u(t_n) + h_t * u'(t_n)$$

Nun haben wir alles um das Poissonproblem zu lösen. Die oben definierte Gleichung konvertieren wir um sie numerisch zu lösen. Da wir u' von unserem neuen Zeitschritt noch nicht kennen ersetzen wir es durch $\frac{u(t_n) - u(t_{n-1}))}{h_t}$.

$$A * u(t_n) = \frac{h_s^2}{a} * (f - \frac{u(t_n) - u(t_{n-1}))}{h_t})$$

Die Lösung des Problems gibt uns $u(t_1)$ und dadurch können wir wieder mit dem ersten Schritt weiter machen. Damit lässt sich jeder beliebige Zeitschritt

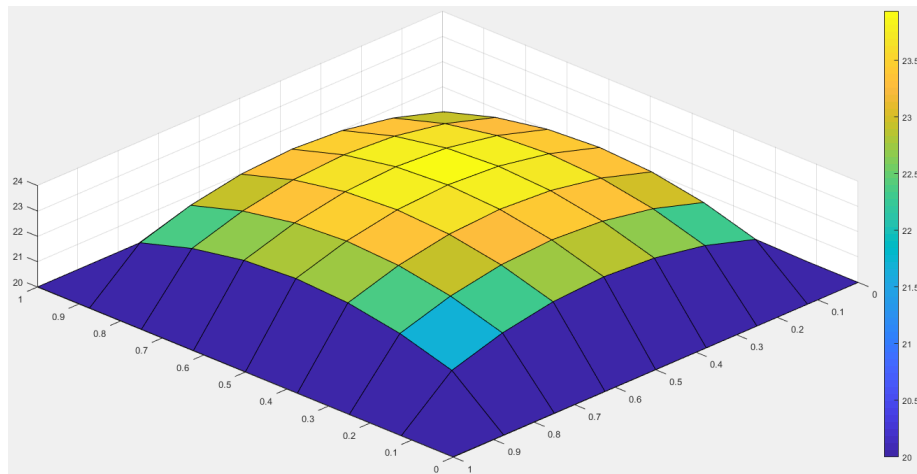


Figure 2.2: $u(t_5)$

berechnen.

Implementierung

Wir haben uns für die Implementierung des Poisson-Gleichungs-Lösers für einen vorkonditionierten GMRES-Algorithmus entschieden, da die Matrix A konstant bleibt und man somit den Vorkonditionierer nur ein Mal bestimmen muss und trotzdem in jeder Iteration davon profitiert. Wir haben $a=1$ angenommen in unserer Implementierung und alle anderen Parameter sind konfigurierbar. Außerdem werden die Hitzedaten in jedem x-ten Zeitschritt ausgegeben, wobei man x frei wählen kann. Bei den Zeitschritten 5, 30 und 60 ergeben sich die in den Grafiken dargestellten Hitzewerte.

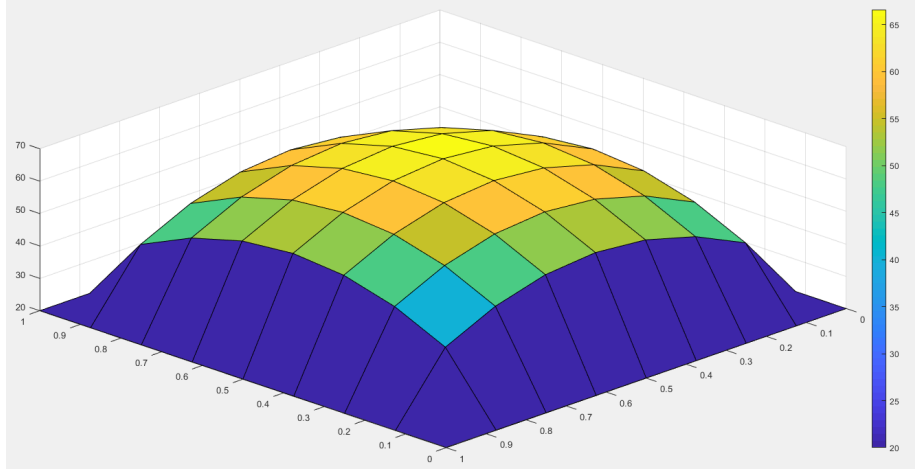


Figure 2.3: $u(t_{30})$

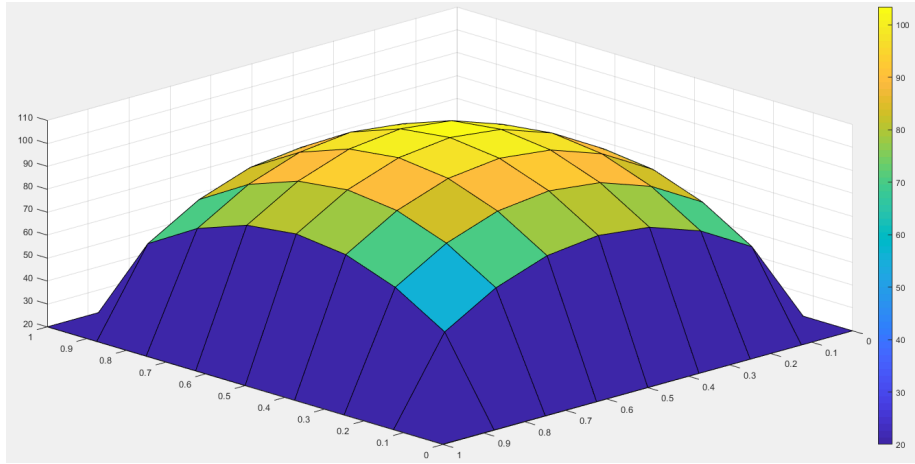


Figure 2.4: $u(t_{60})$