ulm university universität ulm

Institut für Numerische Mathematik

Dr. Andreas F. Borchert

29. Mai 2015
Blatt 5

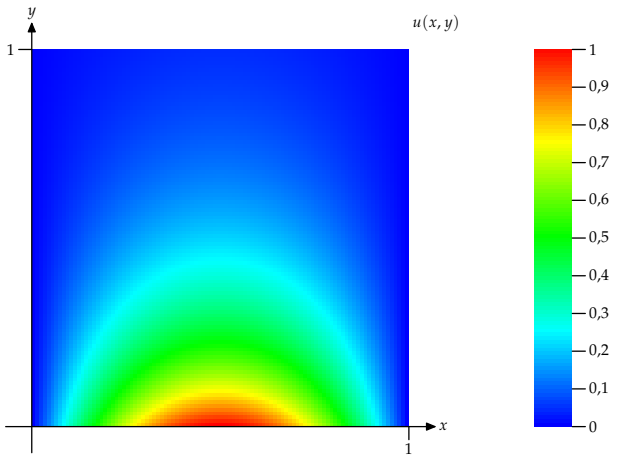
Parallele Programmierung mit C++ (SS 2015)

Abgabe bis zum 12. Juni 2015, 10:00 Uhr

Lernziele:

- Implementierung von Synchronisierungs-Barriers
- Partitionierung einer Matrix für mehrere Threads

Aufgabe 8: Jacobi-Verfahren



Gesucht sei eine numerische Näherung einer Funktion $u(x,y)$ für $(x,y) \in \Omega = [0,1] \times [0,1]$, für die gilt: $u_{xx} + u_{yy} = 0$ mit der Randbedingung $u(x,y) = g(x,y)$ für $x,y \in \delta\Omega$. Das obige Beispiel zeigt eine numerische Lösung für die Randbedingungen $u(x,0) = sin(\pi x)$, $u(x,1) = sin(\pi x)e^{-\pi}$ und $u(0,y) = u(1,y) = 0$. Numerisch lässt sich das Problem lösen, wenn das Gebiet Ω in ein $N \times N$ Gitter gleichmäßig zerlegt wird. Dann lässt sich $u(x,y)$ auf den Gitterpunkten durch eine Matrix A darstellen, wobei

$$A_{i,j} = u(\frac{i}{N}, \frac{j}{N})$$

für $i, j = 0 \ldots N$.

Hierbei lässt sich A schrittweise approximieren durch die Berechnung von $A_0, A_1 \ldots$, wobei A_0 am Rand die Werte von $g(x,y)$ übernimmt und ansonsten mit Nullen gefüllt wird.

Es gibt mehrere iterative numerische Verfahren, wovon das einfachste das Jacobi-Verfahren ist mit dem sogenannten 5-Punkt-Differenzenstern:

$$A_{k+1_{i,j}} = \frac{1}{4}\left(A_{k_{i-1,j}} + A_{k_{i,j-1}} + A_{k_{i,j+1}} + A_{k_{i+1,j}}\right)$$

für $i, j \in 1 \ldots N-1, k = 0, 1, 2, \ldots$ (Zur Herleitung siehe Alefeld et al, *Parallele numerische Verfahren*, S. 18 ff.) Die Iteration wird solange wiederholt, bis

$$\max_{i,j=1\ldots N-1} \left| A_{k+1_{i,j}} - A_{k_{i,j}} \right| \leq \epsilon$$

für eine vorgegebene Fehlergrenze ϵ gilt.

Das numerische Verfahren lässt sich parallelisieren, indem die Matrix A eindimensional in Gruppen aufeinanderfolgender Zeilen oder zweidimensional in Blöcken zerlegt wird. Im Rahmen dieser Übungsaufgabe können Sie die etwas einfachere eindimensionale Partitionierung verwenden.

Wenn Sie das Verfahren parallelisieren, können Sie die gesamte Matrix A_k in einer für alle Threads gemeinsamen Datenstruktur unterbringen. Hinzu kommt die Matrix der nachfolgenden Iteration A_{k+1}. Sobald alle Threads die Berechnung ihres Anteils an A_{k+1} abgeschlossen haben, können Sie die zuvor für A_k verwendete Matrix für A_{k+2} verwenden. Wichtig ist hier jedoch die Synchronisierung aller Threads, da mit der Berechnung von A_{k+2} erst begonnen werden darf, wenn die von A_{k+1} abgeschlossen ist. Diese Synchronisierung kann gleichzeitig dazu genutzt werden, das globale Maximum der Abweichung aufeinanderfolgenden Iterationen festzustellen.

Es steht Ihnen frei, ob Sie Ihre Lösung direkt mit Threads oder mit Hilfe von OpenMP umsetzen.

Wenn Sie explizit mit Threads arbeiten, benötigen Sie eine Template-Klasse *AggregatingBarrier* in C++, die

- von dem für die Aggregierung zu verwendenden Datentyp T abhängt,
- mit der Zahl der Threads n, einem Aggregierungsoperator und einem Ausgangswert für die Aggegrierung initialisiert wird,
- eine Methode *wait* offeriert, die einen Wert des Typs T entgegennimmt, den Thread solange blockiert, bis *wait* von n Threads aufgerufen worden ist und den aggregierten Wert zurückgibt, und die
- nach n Aufrufen von *wait* wiederholt verwendet werden kann und dabei die Zahl der Iterationen zählt und jeweils den Aggregierungswert auf den Ausgangswert zurücksetzt.

Der Aggregierungsoperator ist dabei ein Objekt mit einem ()-Operator, der zwei Argumente des Typs T entgegennimmt und einen Wert des Typs T zurückliefert. Der erste Parameter ist dann jeweils der bislang aggregierte Wert und der zweite Parameter der neu hinzukommende zu aggregierende Wert. Verpacken können Sie solche polymorphen Objekte mit der Klasse *std*::*function*:

```
#include <functional> // fuer std::function
// ...
template <typename T>
class AggregatingBarrier {
    public:
        typedef std::function<T(T, T)> AggregatingOperator;
        AggregatingBarrier(const AggregatingOperator& op, /* ... */) :
            aggregating_op(op), /* ... */ {
        }
        // ...
    private:
        AggregatingOperator aggregating_op;
        // ...
};
```

Sie können dann einen Lambda-Ausdruck verwenden, um den Aggregierungs-Operator zu spezifizieren. Hier ist ein Beispiel für den Aufruf des Konstruktors mit einem summierenden Aggregierungsoperator:

```
AggregatingBarrier([](double a, double b) -> double {
        return a + b;
    }, /* ... */)
```

Als nächstes wird dann eine Klasse *JacobiThread* benötigt, die einen einzelnen Thread des Jacobi-Verfahrens repräsentiert. Deren Konstruktor benötigt die Zeiger auf die beiden Matrizen, einen Hinweis auf die zu bearbeitende Partition, eine Referenz oder Zeiger auf den *AggregatingBarrier* und das Abbruchkriterium ϵ. Damit sie dann *std::thread* übergeben werden kann, benötigt sie einen ()-Operator, der die Jacobi-Iterationen durchführt, bis das Abbruchkriterium erreicht wird.

Schließlich ist eine Klasse *Jacobi* zu implementieren, die die Randbedingung $g(x, y)$ erhält (wieder als polymorphes Funktionsobjekt mit einem entsprechenden ()-Operator), die Dimensionierung des Gitters N, das Abbruchkriterium ϵ und die Zahl der Threads n. Diese belegt dann die Speicherflächen für A_0 und A_1, initialisiert A_0, erzeugt dann die n auf *JacobiThread* basierenden Threads und liefert dann das Näherungsresultat zurück. Anhand des Iterationszählers kann dann festgestellt werden, welche der beiden Matrizen das Endergebnis hält.

Alternativ, wenn Sie die Lösung mit OpenMP entwickeln möchten, können Sie als Barrier die OpenMP-Anweisung „**#pragma** omp barrier“ verwenden. Diese bietet allerdings keine Aggregierungsfunktion, so dass sie das Abbruchkriterium auf andere Weise umsetzen müssen.

Wenn Sie alles unter Solaris zusammenbauen, vergessen Sie bitte nicht mit der Option „-lm“ die Mathematik-Bibliothek hinzuzunehmen. (Bei Linux benötigen Sie diese Option nicht.)

Für eine erste Überprüfung, ob die Endmatrix gut aussieht, empfiehlt sich eine Visualisierung. Hierfür steht das Perl-Skript *gen-mp.pl* zur Verfügung. Dies erwartet in der ersten Zeile die Dimensionierung der Ausgabematrix. Danach sollten zeilenweise die einzelnen Zeilen der Matrix ausgegeben werden, wobei die einzelnen Werte durch Leerzeichen getrennt werden. Das Skript gibt dann auf der Standardausgabe ein METAPOST-Programm aus, das mit

Hilfe von *mpost* PostScript erzeugt (in einer Datei mit der Endung „.1“) und das danach mit *mptopdf* in PDF konvertiert werden kann. Leider ist *mpost* auf Theseus fehlerhaft, es lohnt sich hier, die ältere Version von *mpost* auf Turing oder die neuere auf der Thales zu verwenden.

Verpacken Sie all Ihre Quellen mit *tar* in ein Archiv und reichen dies ein:

```
tar cvf jacobi.tar *.?pp [mM]akefile
submit pp 8 jacobi.tar
```

Viel Erfolg!