Semesterprojekte - Parallel Systems

Sebastian Bauer*

April 9, 2013

Allgemeine Ziele

Die praktischen Inhalte der Vorlesung und Laborübung sollen von Zweiergruppen in den eingeteilten Themen angewandt, vertieft und aufbereitet werden. Die zu bearbeiten Algorithmen sind sowohl auf Grundlage von algorithmischer Beschreibung, durch fehlerfreie Implementationen in C/C++ als auch durch eine Diskussion der Ergebnisse am Ende des Semesters in der Laborübungsgruppe oder im Unterricht vorzustellen.

Die Programme sollen betriebsystemunabhängig geschrieben werden, mit `gcc` oder `clang` übersetzbar sein und `make` als Build-System nutzen. Ziel ist es, den gewählten Algorithmus mit mindestens zwei verschiedenen APIs für Parallelität zu implementieren, die gemeinsam mit der sequentiellen Variante miteinander verglichen werden. Als Schnittstellen kommen OpenMP, OpenMPI und OpenCL in Frage. Andere Schnittstellen sind in vorheriger Absprache mit dem Dozenten auch möglich. Zwar liegt der Hauptaugenmerk auf der Parallelisierung, allerdings sind Benutzerinteraktion (z.B. Wahl eines Ausschnittes oder Parametereingabe) und Visualisierung der berechneten Daten oftmals auch sinnvoll und werden positiv bewertet. Mögliche Schnittstellen sind SDL, OpenGL, Qt, wxWidgets oder GTK, die für die meisten Betriebssysteme verfügbar sind. Hierbei ist zu auf Modularität zu achten, d.h., die algorithmische Implementierung ist von der Benutzeroberfläche zunächst strikt zu trennen.

Als IDE empfiehlt es sich Eclipse mit CDT zu verwenden. Der Code soll in einem Projekt-eigenen Git- oder SVN-Repository über den Projekt-Server des Fachbereichs abgelegt werden (`http://studi.f4.htwberlin.de`). Sowohl der Programmtext als auch Änderungen im Repository sind sinnvoll zu kommentieren. Für die Dokumentation der Schnittstellen im Programmtesxt empfielt es sich, `doxygen` zu benutzen. Zu den Laborübungen wird der Bearbeitungsstand des Projektes erfragt und mögliche Lösungswege diskutiert.

Zusammengefasst sind die Anforderungen an einem erfolgreichen Abschluss der Seminararbeit:

- Das ausgewählte Problem mit mindestens einer, besser zwei parallelen Schnittstellen zu implementieren

*mail@sebastianbauer.info

- Die Implementation fehlerfrei zu gestalten (`valgrind` und `helgrind` bzw. *address sanitizer* und *thread-sanitizer*)
- Initialisierungen per Kommandozeileninterface (oder Konfigurationsdatei) zuzulassen
- Der berechneten Konfigurationen zu visualisieren
- Die Dauer der Ausführung der Implementationen in Abhängigkeit steigender Problemgröße auszuwerten
- Gute Dokumentation des Quelltextes (`doxygen`)
- Besonders positiv bewertet wird auch ein benutzerfreundliches Interface mit Eingriffsmöglichkeiten (Parameterveränderungen, Perturbationen, etc.)

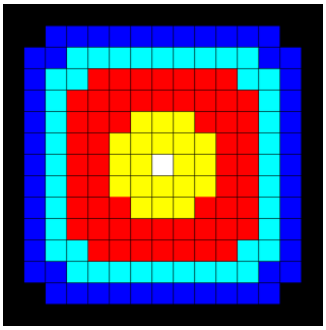
Der Vortrag soll ca. 25 Minuten dauern und von einer Diskussion begleitet werden. Neben der Implementation und dem Vortrag soll das Thema schriftlich in deutscher oder englischer Sprache ausgearbeitet werden. Ein guter Zielwert für den Umfang der schriftliche Ausarbeitung liegt zwischen 10 und 20 Seiten. Der erste Teil führt kurz in die theoretischen Grundlagen der Problemstellung (Anwendung, mathematische Herleitung, etc.) ein. Anschließend soll die eigene Vorgehensweise und die Implementation erläutert werden, wobei es sich empfiehlt, nur die Kernelemente der Implementation tatsächlich ins Dokument aufzunehmen. Im dritten Teil der Ausarbeitung werden Ergebnisse präsentiert. Dazu gehört z.B. ein Zeitvergleich zwischen sequenzieller und paralleler Programmausführung mit wachsender Problemgröße. Abschließend wird die Arbeit diskutiert und auf Probleme und Schwierigkeiten wird eingegangen. Das Schema kann auch für die Gliederung des Vortrags dienen. Bitte stellt im Text dar, wer was zur Arbeit beigetragen hat. Das kann eine separater Abschnitt sein oder direkt im Textfluss eingebaut werden. Es genügt, die Ausarbeitung dem Dozenten auf elektronischem Wege (per E-Mail an mail@sebastianbauer.info oder Zugriff auf Repository) bis spätestens zum 16. August 2013 zukommen zu lassen

1 Einfache Wärmeleitungsgleichung

Viele Probleme in paralleler Programmierung benötigen eine Kommunikation zwischen Tasks, die Teilprobleme bearbeiten. Einige spezielle Subklassen von Problemen benötigt dabei Kommunikation ausschließlich zwischen "'Nachbar-Tasks"'. Die Lösung der einfachen *Wärmeleitungsgleichung* ist ein solches Problem.

Die Wärmeleitungsgleichung beschreibt die zeitliche Temperaturänderung eines Körpers ausgehend von einem *Anfangszustand* (Zustand zum Zeitpunkt $t = 0$) und *Grenzbedingungen*. Bei dieser Aufgabe geht es um ein Objekt, das die Form eines Quadrates mit beliebiger Kantenlänge a hat. Zu Beginn ist die Temperatur des Quadrates in seiner Mitte am höchsten, während die Temperatur der Ränder 0 beträgt. Außerdem wird die Temperatur der Ränder dauerhaft auf 0 gehalten.

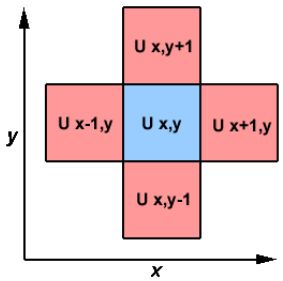
Um den weiteren zeitlichen Verlauf der Temperaturverteilung zu bestimmen, soll die Finite-Differenzen-Methode angewendet werden, um die Wärmeleitungsgleichung nummerisch zu lösen. Dabei wird der Temperaturzustand des Quadrates durch Temperaturzustände vieler kleinerer Subquadrate repräsentiert.



Der Zustand eines Subquadrates $u_{x,y}$ zum Zeitpunkt $t+1$ (geschrieben $u^{t+1}_{x,y}$) hängt von den Zuständen des selben Elements und seiner Nachbarelementen zum Zeitpunkt t ab. Die Formel

$$u^{t+1}_{x,y} = u^t_{x,y} + c_x(u^t_{x+1,y} + u^t_{x-1,y} - 2u^t_{x,y}) + c_y(u^t_{x,y+1} + u^t_{x,y-1} - 2u^t_{x,y})$$

beschreibt den Zusammenhang. Es werden solange neue Zeitpunkte berechnet, bis das Ergebis konvergiert (d.h., der Zustand jedes einzelnen Subquadrates ändert sich nicht mehr bis auf ein bestimmtes ϵ).



- Schreibe zunächst ein serielles Programm, das für jeden Zeitpunkt den Zustand berechnet, solange keine Konvergenz vorliegt. Dabei soll der Parameter ϵ vom Benutzer (als Kommandozeilenparameter oder Benutzeroberfläche) vorgegeben werden.
- Die Zeit und die Anzahl der Zeitschritte, die das Programm für die Bearbeitung benötigt, soll ausgegeben und ausgewertet werden.
- Der Kernalgorithmus ist mit zwei Schnittstellen zu parallelisieren, die zusammen mit der sequentiellen Variante verglichen werden.
- Idealerweise wird der aktuelle Zustand zusätzlich auf dem Bildschirm graphisch ausgegeben, eine gleichzeitige oder zusätzliche Speicherung als Bild ist auch denkbar.
- Pertubationen (z.B. zusätzliche Hitzequellen per Mausklick) können auch eingebaut werden.

2 Simulationen von Netzplänen

Netzpläne beschreiben temporale und finale Verkettung von Aktionen, die Bestandteile einer Aufgabe oder eines Prozesses sind. Sie finden ihre Anwendung insbesondere in der Terminplanung von Projekten. Ein Netzplan kann mathematisch als diskreter gerichteter Graph dargestellt werden. Knoten des Graphs repräsentieren Teilaufgaben und Kanten die Abhängigkeiten. Wenn sich eine Kante zwischen zwei Knoten a und b befindet, dann bedeutet dies, dass die Ausführung von b vom Abschluss der Teilaufgabe a abhängt.

Jedem Knoten ist zusätzlich eine Dauer zugeordnet. Da die Dauer einer Teilaufgabe im Voraus nicht exakt bestimmbar ist, wird die Dauer mit Hilfe von Wahrscheinlichkeitsverteilungen angegeben. Beispiele für Wahrscheinlichkeitsverteilungen sind die uniforme Gleichverteilung, Exponentialverteilung oder Normalverteilung.

Netzpläne können auch in Form eines sogenannten Gantt-Diagramms dargestellt werden. Ein Beispiel findet sich in Abbildung 1.

Die Aufgabe dieses Projektes ist, aus einem gegebenen, vollständig spezifizierten Netzplan, die Wahrscheinlichkeitsverteilung der Gesamtdauer zu bestimmen und u.a. darauf die zu erwartende Gesamtdauer zu bestimmen. Hierbei soll ein Monte-Carlo-Ansatz Verwendung finden.

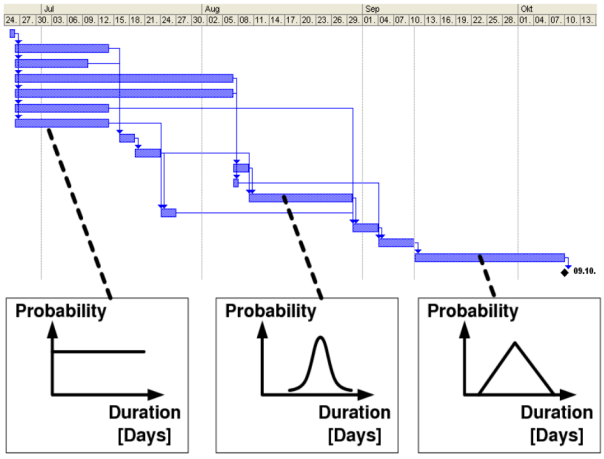


Figure 1: **Netzplan in Form eines Gantt-Diagramms**. Die horizontale Achse ist die Zeitachse. Jeder Balken definiert einen Teilprozess und die Länge des Balkens die zu erwartende Dauer. Jedem Prozess ist eine Wahrscheinlichkeitsverteilung zugeordnet. Die Abhängigkeiten zwischen Teilprozessen werden durch Pfeile dargstellt.

- Definiere eine geeignete Datenstruktur zur Repräsentation eines Netzplans.
- Implementiere den Monte-Carlo-Ansatz für den Algorithmus, der auf der Datenstruktur agiert.
- Parallelisiere den Algorithmus mit Hilfe von zwei Schnittstellen für Parallelität.
- Plotte die Berechnungsdauer in Abhängigkeit von der Problemgröße für verschiedene Anzahl von Rechnenprozessen.

3 Mandelbrotmenge

Ziel dieser Aufgabe ist es ein Apfelmännchen- oder Mandelbrot-Generator zu entwickeln, der von Parallelität profitiert. Die Mandelbrot-Menge ist mathematisch gesehen eine Menge derjenigen komplexen Zahlen c, für die Rekursionsfolge

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

mit der Anfangsbedingung $z_0 = 0$ konvergiert. Eine Darstellung dieser Menge erfolgt in der Regel in der komplexen Ebene. Dabei werden Punkte, die in der Menge zugehörig sind, also für die die Zahlenfolge konvergiert, schwarz dargestellt. Punkte, die nicht zur Menge gehören, die beim Einsetzen in die

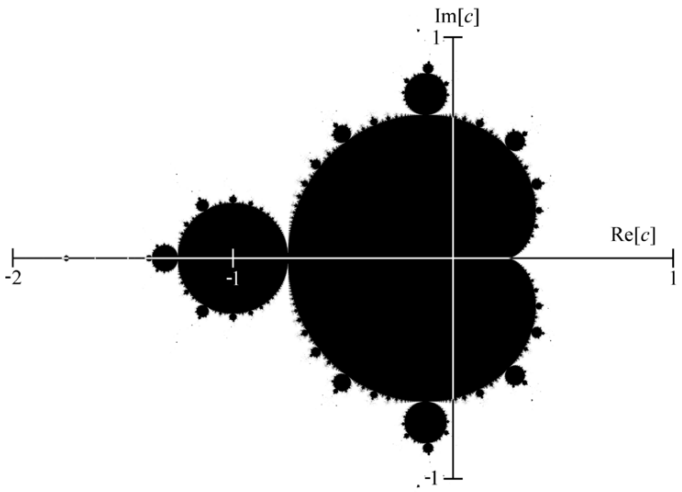


Figure 2: **Grafische Darstellung der Mandelbrotmenge in der komplexen Ebene**. Üblicherweise bildet die waagerechte Achse den Realteil und die senkrechte Achse den Imaginärteil einer komplexen Zahl. Um die Mandelbrotmenge zu visualisieren, werden zugehörige Punkte schwarz kodieren, während bei dieser Darstellung alle anderen weiß kodiert werden.

Rekursionsgleichung also eine divergente Zahlenfolge erzeugen, werden farblich anders kodiert. In Abbildung 2 ist die Menge dargestellt.

In Literatur findet man farbliche Mandelbrotdarstellungen. Hier werden die Punkte in Abhängigkeit vom sogenannten Divergenzwinkels farblich kodiert. Ob eine Zahlenfolge für ein c konvergiert, lässt sich ermitteln, indem die Zahlenfolge für dieses c angewendet wird. Sobald der Betrag eines Gliedes den Wert 2 überschreitet, also $|z_n| \geq 2$, divergiert die Zahlenfolge. Die Anzahl der Schritte bis zu diesem Ereignis, lässt sich als Divergenzgrad verstehen und damit zur Farbkodierung verwenden.

Sehr nützlich für das Verständnis der Theorie sind die Wikipedia-Einträge:

- http://de.wikipedia.org/wiki/Komplexe_Zahl
- http://de.wikipedia.org/wiki/Mandelbrot-Menge

Die Aufgabenstellung lautet konkret:

- Schreibe eine Programm, das die Mandelbrotmenge berechnet. Dabei soll die maximale Rekursionstiefe vom Benutzer spezifiziert werden.
- Das entstandene Bild soll graphisch ausgegeben werden.
- Der Algorithmus soll von mehreren Prozessorkernen profitieren.

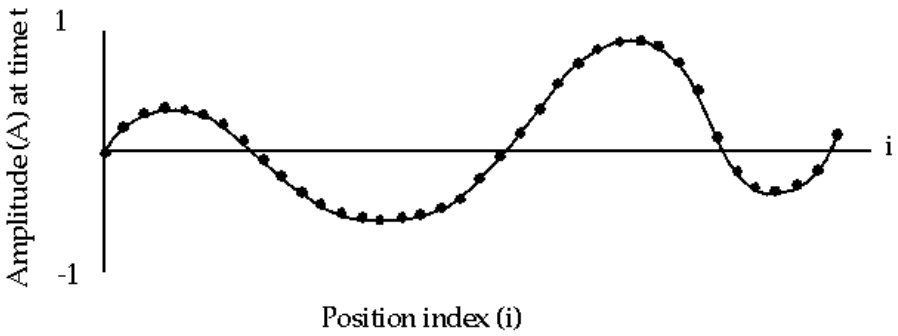


Figure 3: Welle

- Der Speedup ist für eine verschiedene Kernanzahl und Implementationen für verschiedene Ausschnitte zu messen und zu diskutieren.
- Idealerweise lässt sich der Ausschnitt direkt in der Anzeige der Grafik durch den Benutzer verändern.

4 Wellengleichung

In dieser Aufgabe sollen die Amplituden einer vibrierenden Saite berechnet und visualisiert werden. Das für diese Aufgabe zu Grunde liegende Modell liefert die Wellengleichung, von der wir den eindimensionalen Fall annehmen. Zum Finden eines numerischen Approximation der Amplitude zu einem beliebigen diskreten Zeitpunkt $t+1$ am Knotenpunkt i, bietet sich folgende Rechenvorschrift an:

$$A(i,t+1) = 2A(i,t) - A(i,t-1) + c(A(i-1,t) - 2.0A(i,t) + A(i+1,t))$$

Ein Beispiel findet sich in Abbildung 3.

- Implementiere das vorgestellte numerisches Approximationsverfahren.
- Erweitere die Implementation, so dass von mehreren Kernen profitiert werden kann.
- Visualisiere die berechneten Wellen.
- Der Initialzustand und die Parameter sind vom Benutzer zu spezifizieren, z.B. direkt in der Benutzeroberfläche und optional in einer Konfigurationsdatei.
- Zum Testen kann auch eine zufällige Welle erzeugt werden.
- Perturbationen (z.B. Benutzer klickt auf ein Teil der Saite, um sie festzuhalten) können auch eingebaut werden.

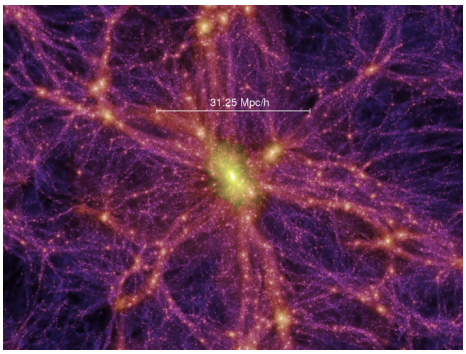


Figure 4: **N-Körper-Simulation**. Dieses Bild entstammt aus dem Millennium Ru, einer Simulation des Universiums von seiner Entstehung bis heute. Quelle: Wikipedia

5 N-Körper-Simulation

Das N-Körper-Problem beschäftigt sich mit der Simulation der Bewegung einer fixen Menge von N Körpern, die durch Kräfte miteinander interagieren. Dies könnten z.B. unsere Himmelskörper sein. Dabei ist für jeden Körper i eine Masse m_i, ein initialer Positionsvektor $\vec{p}_i$ sowie ein initialer Geschwindigkeitsvektor $\vec{v}_i$ gegeben.

Die Beschleunigung, die ein Körper i von den anderen Körpern im System erfährt, ergibt sich näherungsweise aus

$$\vec{a}_i = \gamma \sum_{j=1}^{N} \frac{m_j \vec{r}_{ij}}{(\|\vec{r}_{ij}\|^2 + \epsilon^2)^{\frac{3}{2}}},$$

wobei γ die Gravitationskonstante, ϵ ein *softening factor*[1] und $\vec{r}_{ij} = \vec{p}_j - \vec{p}_i$ ist. Mit Hilfe der errechneten Beschleunigung, die Körper i erfährt, lässt sich seine Position sowie Geschwindigkeit iterativ durch ein Zeitschrittverfahren aktualisieren.

Ziel dieser Aufgabe ist es, ein Programm zu erstellen, die eine N-Körper-Simulation ausführt. Der Simulationsalgorithmus soll parallelisiert werden und in OpenCL und OpenMP oder OpenMPI implementiert werden. Die Startwerte sind von einer Datei einzulesen, die einem selbst gewählten Format folgt.

Die Werte für die Anfangsbedingungen der Planeten finden sich zum Beispiel in `http://iau-comm4.jpl.nasa.gov/XSChap8.pdf` wieder. Werte für weitere Himmelskörper lassen sich über `http://ssd.jpl.nasa.gov/?horizons` abrufen. Sinnvoll ist ein Skript für das Telnet-Interface zu schreiben, das die Eingabe-Datei erzeugt. Der Benchmark soll für mehre, auch hohe N durchgeführt werden.

[1] Damit wird verhindert, dass bei einer Kollision eine Division durch 0 stattfinden. Sinnvolle Werte sind z.B. $\epsilon = 0.01$.

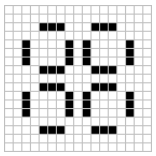


Figure 5: **Game of Life**. Dieses Muster stellt einen sogenannten Pulsar dar. Quelle: Wikipedia

6 Conways Spiel des Lebens

In Conways Spiel des Lebens (engl. *Conway's Game of Life*) wird die Evolution einer Zellenpopulation mit Hilfe eines zweidimensionalen zellulären Automaten beschrieben. Jedes Gitterquadrat des Spielfeldes stellt eine Zelle dar, die entweder tot oder lebendig ist. Jede Zelle hat insgesamt 8 Nachbarzimmer. Der Spieler legt zu Beginn eine Initialkonfiguration fest. Es folgen die Evolutionsschritte. Weitere Interaktionen seitens des Spielers sind normalerweise nicht vorgesehen.

Die Entwicklung der Zellen läuft nach definierten Regeln ab, wobei der Zustand einer Zelle in der neuen Generation ausschließlich von den Zuständen der Nachbarzellen der alten Generation abhängt. Die genauen Regeln finden sich z.B. auf `http://www.mathematische-basteleien.de/gameoflife.htm` oder `http://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens`.

Ziel dieser Aufgabe ist es, ein Programm für das Spiel des Lebens zu implementieren, das für beliebige Eingabegrößen funktioniert und von Mehrkernprozessoren und verteilten Rechnern profitiert. Der parallelisierte Algorithmus soll in mindestens zwei APIs (OpenCL, OpenMP oder OpenMPI) implementiert werden und zusammen mit der seriellen Version gegenübergestellt werden.

7 Voronoi-Diagramme

Ein Voronoi-Diagramm ist eine Zerlegung des Raumes in Zellen, die durch eine gegebenen Punktmenge $P = \{p_1, \ldots, p_n\}$ induziert wird. Eine Zelle k ist durch genau einen Punkt $p_k \in P$ und allen Punkten des Raumes definiert, deren Abstand zu anderen Punkten aus P größer ist als zu p_k.

Ziel dieser Aufgabe ist, ein Programm zu entwickeln, das aus einer gegebenen Menge von Punkten das zugehörige Voronoi-Diagramm konstruiert. Der Ablauf eines Algorithmus findet sich z.B. auf `http://de.wikipedia.org/wiki/Voronoi-Diagramm`. Der Algorithmus ist nach der Vorgaben im Abschnitt zu parallelisieren. Für den Benchmark ist es sinnvoll zu erlauben, dass der Benutzer die zu unterteilenden Punkte in einer Datei festlegt.

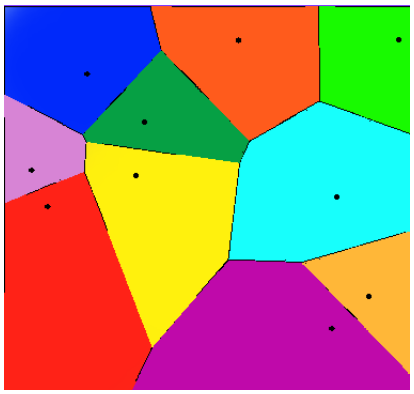


Figure 6: **Voronoi-Diagram**. Quelle: Wikipedia

8 Kürzeste Pfade

Ein grundlegendes Problem in der Graphentheorie ist das Suche eines kürzesten Pfades ausgehend von einem Startknoten zu allen Knoten in einem Graph. Das Problem wird von Dijkstra Algorithmus effizient gelöst. Ziel dieser Aufgabe ist es, den Algorithmus zu parallelisieren und in OpenCL zu implementieren. Als Grundlage kann eine Arbeit von Harish und Narayanan (2008) mit dem Titel *Accelerating large graph algorithms on the GPU using CUDA* dienen.

9 Zählen von Kreisen

Gegeben ist ein zweidimensionales Bild, z.B. ein Foto von Geldmünzen wie in Abbildung 7. Es soll algorithmisch die Anzahl der Kreise bestimmt werden, die auf dem Bild zu sehen sind. Hierfür eignet sich die sogenannte Hough-Transformation, die den Bildraum in einem Parameterraum der zu suchenden Figurklasse überführt. Bei Kreisen sind dies Position und Radius.

Als Eingabe der Hough-Transformation dient häufig ein Binärbild, auf dem nur die Umrisse der Objekte zu sehen sind. Aus einem Foto kann dieses durch einen Kantendetektionsfilter wie z.B. dem Sobel-Filter gewonnen werden. Für die Arbeit genügt es, den Hough-Transformation auf dem Binärbild zu implementieren. Den Filter wendet man in diesem Fall vorher manuell an, oder man zeichnet ein paar eigene Testbilder mit verschiedenen Kreisen. Wer möchte, kann natürlich auch den Sobel-Filter implementieren, der sich ebenso parallelisieren lässt. Wie bei den anderen Aufgaben ist es Ziel, den Algorithmus zu parallelisieren und die Ausführungszeit mit der Ausführungszeit der sequentiellen Implementierung zu vergleichen.

Ein guter Einstiegspunkt für mehr Information ist der Wikipedia-Artikel `http://de.wikipedia.org/wiki/Hough-Transformation`. Das Poster `http://developer.download.nvidia.com/GTC/PDF/GTC2012/Posters/P0438_ht_poster_`

Figure 7: **Münzenbild**.

`gtc2012.pdf` stellt eine GPU-Implementation mit CUDA vor.

10 Freestyle

In dieser Aufgabe kann das Thema des Projektes in vorheriger Absprache mit dem Dozenten frei gewählt werden. Eine wichtige Bedingung ist, dass es sich bei dem gewähltem Problem um ein rechenintensives und parallelisierbares Problem handelt. Wie bei den anderen Projekten, sollten mindestens zwei Implementation basierend auf unterschiedlichen Frameworks (OpenMP, OpenMPI oder OpenCL) miteinander verglichen werden.