Ergebnisse der Parallelisierung eines Schnittkurvenalgorithmus

Jens Trapp
jens@ts.go.dlr.de

30. September 1994

Ziel der Arbeit war die Definition eines parallelisierbaren Schnittkurvenalgorithmus, also eines Algorithmus, der sowohl sequentiell als auch parallel eingesetzt werden kann. Dazu wurde zuerst ein (sequentieller) Algorithmus aufgestellt, der schnell und korrekt arbeitet. Dabei wurden Teilalgorithmen ausgewählt, die besonders für eine spätere Parallelisierung geeignet waren. In einem zweiten Schritt wurde das im ersten Teil entwickelte Verfahren nach dem *Teile–und–Herrsche*–Prinzip[1] parallelisiert. Bei der Implementation auf einer KSR1 von Kendall Square Research wurde die C–Programmiersprache mit der pthread–Bibliothek verwendet, die eine Portierung auf mehrere Systeme (z.B. Sun Solaris 2.1) ermöglicht. Es hat sich gezeigt, daß der verwendete Algorithmus bis zu einer vom jeweiligen Schnittproblem abhängigen Prozessoranzahl eine optimale Effizienz aufweist. Im allgemeinen ist diese Prozessorzahl jedoch niedrig, so daß sich eine massive Parallelisierung nur in besonderen Fällen lohnt.

1 Schnittkurven

Schnittkurven entstehen, wenn man Flächen oder Körper miteinander verschneidet. Mit Kenntnis der Schnittkurven kann man mehrere Flächen zu einem Körper verbinden oder aus gegebenen Körpern neue gewinnen. Hiervon wird beispielsweise beim *computer–unterstützten Konstruieren* (engl.: computer aided design, CAD) Gebrauch gemacht:

Beispiel: *Ein wesentlicher Bestandteil eines CAD–Programms ist die Objekt–Bibliothek. Darin sind allgemeine Objekte, beispielsweise Ebenen, Kugeln oder Zylinder, enthalten. Aus diesen Objekten baut der Konstrukteur sein Modell zusammen. Dazu schneidet er aus Objekten heraus, was er benötigt und verknüpft die Teile zu einem neuen Modell. Hierfür ist es notwendig, die Schnittkurven dieser Objekte zu berechnen.*

Auch für glatte Ausgangsflächen kann die Schnittmenge sehr kompliziert aussehen. Für zwei beliebige Flächen können die folgenden Fälle auftreten:

1. Die Flächen schneiden sich nicht.
2. Die Flächen schneiden sich in einem singulären Punkt.
3. Die Flächen schneiden sich in einer glatten Kurve.
4. Die Flächen schneiden sich in einem gemeinsamen Flächenstück.
5. Beliebige Kombinationen der Fälle (2.)–(4.), wobei jeder Fall beliebig oft auftreten kann.

Die Schnittkurven können Punkte mehrfach durchlaufen (siehe Abbildung 2(b)) oder in mehrere Kurven verzweigen. Diese Fälle komplizieren die Berechnung mit dem aufgestellten Algorithmus, sind aber lösbar. Dagegen

[1] Beim Teile–und–Herrsche–Prinzip wird ein gegebenes Problem in mehrere Teile aufgeteilt, die dann unabhängig voneinander gelöst werden können.

läßt sich der vierte Fall nicht mit den verwendeten Verfahren berechnen. Wenn im folgenden von Schnittkurven gesprochen wird, ist damit die Menge aller Schnittpunkte gemäß den Punkten (2.) und (3.) in beliebigen Kombinationen gemeint.

2 Der Schnittkurvenalgorithmus und seine Parallelisierung

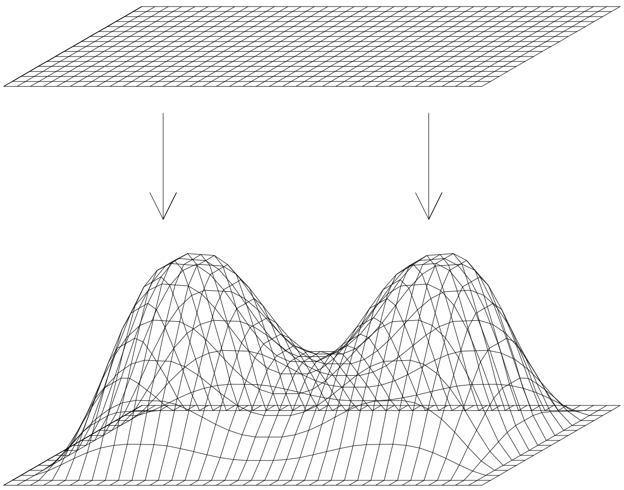


Abbildung 1: Schnitt einer Ebene mit einer Fläche. Wenn die Ebene verschoben wird, entstehen unterschiedliche Schnittkurven.

Während Schnittprobleme für Körper in expliziter Darstellung analytisch berechnet werden können, ist dies für Freiformflächen[2] nur mit sehr hohem Aufwand möglich. Der Algorithmus berechnet die Schnittkurve von zwei parametrisch gegebenen Flächen implizit durch Bestimmung einzelner Kurvenpunkte, mit denen man den Kurvenverlauf interpolieren kann. Das Verfahren wurde für Flächen implementiert, die in Bernstein–Bézier–Tensorproduktdarstellung vorliegen.

Die dem Algorithmus zugrunde liegenden Verfahren sind noch recht jung. Zum einen wurden Ideen eines sogenannten *Teile–und–Herrsche*–Algorithmus aus dem Jahr 1985 von Houghton ([Houghton '85]) benutzt. Hierbei handelt es sich hauptsächlich um globale Strategien zur Eingrenzung des Problems. Ein zweiter Programmteil berechnet dann für das eingegrenzte Problem die Kurvenpunkte. Ausgehend von einem Startpunkt werden mit Hilfe eines Newton–Verfahrens sukzessive weitere Kurvenpunkte ermittelt. Basis für diesen Teil des Algorithmus sind Arbeiten von Barnhill ([Barnhill '87] und [Barnhill '90]), sowie die Arbeiten von Schaback ([Schaback '89] und [Schaback '92]) und Bürger ([Bürger '92]). In den beiden zuletzt genannten Arbeiten wurden die globalen und lokalen Techniken, die ursprünglich zwei getrennte Ansätze zur Berechnung von Schnittkurven bildeten, vereinigt. Im Algorithmus entspricht jede Strategie einer Phase. Globale und lokale Phase werden wechselseitig aufgerufen. In einer weiteren Phase wird die gefundene Kurve auf Eindeutigkeit untersucht.

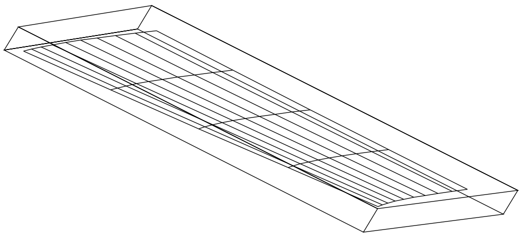


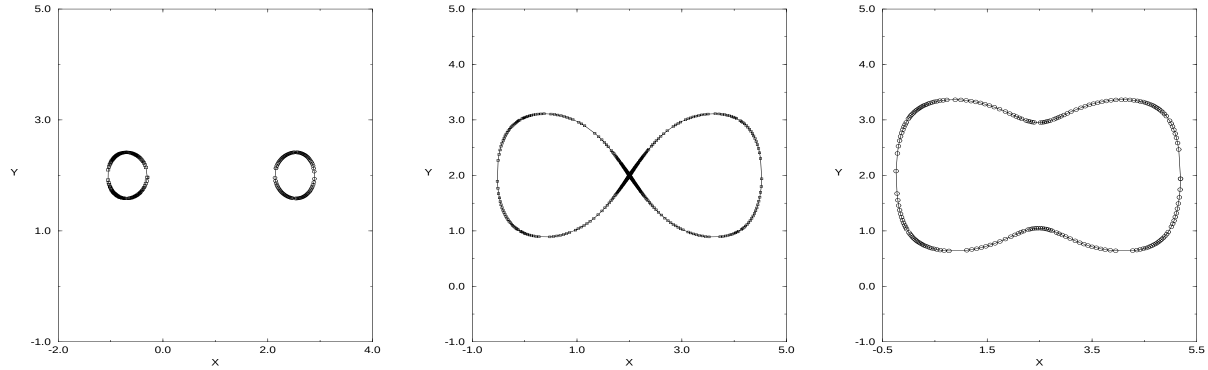
Abbildung 2: Ein Flächenstück wird in der globalen Phase in eine Bounding Box eingeschlossen. Die Bounding Boxes der verschiedenen Flächen werden dann auf Disjunktheit getestet.

Kernstück des globalen Teils ist eine Subdivisionsfunktion. Diese zerteilt das Schnittproblem in kleinere Flächenstücke. Bei jedem Subdivisionsschritt entstehen für beide Ausgangsflächen vier Teilflächen. Stellt man je eine Teilfläche allen Teilflächen der anderen Flächen gegenüber, so erhält man 16 neue Flächenpaare, bzw. Probleme, von denen allerdings viele Paare schnell als disjunkt erkannt werden können. Für den Test auf Disjunktheit wird um jede Teilfläche eine Bounding Box[3] gelegt (siehe Abbildung 2). Anhand der Bounding Boxes kann sehr schnell festgestellt werden, daß zwei Teilflächen sich nicht schneiden[4]. Je öfter die Flächen subdividiert werden, desto mehr

[2] Freiformflächen sind Flächen, die sich nicht in expliziten Gleichungen ausdrücken lassen.

[3] Die Kontrollnetze von Bernstein–Bézier–Flächen bilden eine konvexe Hülle. Darum kann man Bounding Boxes sehr einfach berechnen.

[4] Die Umkehrung gilt im allgemeinen nicht.



(a) Zwei geschlossene, reguläre Schnittkurven, die unabhängig voneinder, also parallel, berechnet werden können.

(b) Schnittkurve mit Doppelpunkt. Wegen fehlender Eindeutigkeit im Schnittpunkt müssen dort viele Punkte berechnet werden.

(c) Eine geschlossene, reguläre Schnittkurve.

Abbildung 3: Schnittkurven zu Beispiel 1 wie in Abbildung 1. Schnitt einer Fläche mit zwei lokalen Maxima und einer Ebene in drei verschiedenen Abständen der Randkurven.

Flächenstücke können als disjunkt erkannt werden. Denn zum einen wird das Problem durch die Aufteilung eingegrenzt; zum anderen umschließen die Bounding Boxes die Teilflächen viel enger als die Ausgangsfläche.

Trotz der Disjunktheitstests entstehen durch die Subdivision, besonders bei hohen Genauigkeitsanforderungen, sehr viele neue Schnittprobleme. Allerdings sind viele dieser Teilprobleme voneinander unabhängig[5]. Alle Flächenpaare, die nicht–benachbarte oder nicht–identische Flächenstücke behandeln, können gleichzeitig, sprich parallel, berechnet werden.

Die lokale Phase, in der mittels Newton–Verfahren und sukzessiver Vorgehensweise die Schnittkurve auf einem Teilstück errechnet wird, ist kaum zu parallelisieren. Es handelt sich dabei um ein streng iteratives Verfahren. Ausgehend von einem Startpunkt[6] wird in einer zu errechnenden Schrittweite[7], tangential zu beiden Flächen ein neuer Punkt *geschätzt*. Das Newton–Verfahren dient zur Verfeinerung dieses Punktes. Anschließend geht der gefundene Punkt als Startpunkt wieder in das Verfahren ein. Die Kurve wird solange *verfolgt*, bis entweder der Rand einer Fläche erreicht wird, oder bis das Newton–Verfahren abbricht.

Die einzige Ansatzmöglichkeit für eine parallele Bearbeitung dieser Phase wäre die Vektorisierung einiger Schleifen, wobei kaum mehr als ein Zeitgewinn von Faktor 3 oder 4 (Dimension der verwendeten Räume) zu erwarten ist. In der lokalen Phase fällt der Großteil der Bearbeitungszeit[8] an. Durch die oben angesprochene Aufteilung in viele unabhängige Flächenstücke ist es möglich, die lokale Phase für viele Teilprobleme gleichzeitig zu durchlaufen. Offenbar fällt bei dieser Strategie kaum Kommunikation an.

Bei der Parallelisierung müssen Schwierigkeiten, die sich aus dem dynamischen und unausgeglichenem Wachstum des Problems ergeben, überwunden werden. Erst bei der Berechnung der Kurve ergibt sich, in welchen Teilbereichen eine genauere Berechnung erforderlich ist. Deswegen ist es nicht sinnvoll, den Algorithmus mit

[5] Beispielsweise sind benachbarte Flächen nur auf der gemeinsamen Randkurve voneinander abhängig.

[6] Zur Ermittlung eines Startpunktes werden die Flächen stückweise linearisiert und die entstehenden Ebenen verschnitten. Liegt ein Schnittpunkt hinreichend nah an beiden Flächen, dann kann über ein Newton–Verfahren ein tatsächlicher Schnittpunkt errechnet werden.

[7] Durch eine adaptive Steuerung der Schrittweite wird sichergestellt, daß die errechneten Kurvenpunkte auf derselben Schnittkurve liegen.

[8] Eine Untersuchung durch die UNIX-Kommandos `prof` bzw. `gprof` hat für die angegebenen Beispiele ergeben, daß etwa 75% der Rechenzeit nur für die Berechnung von Flächenpunkten und -ableitungen nach dem Horner–Schema verbraucht werden.

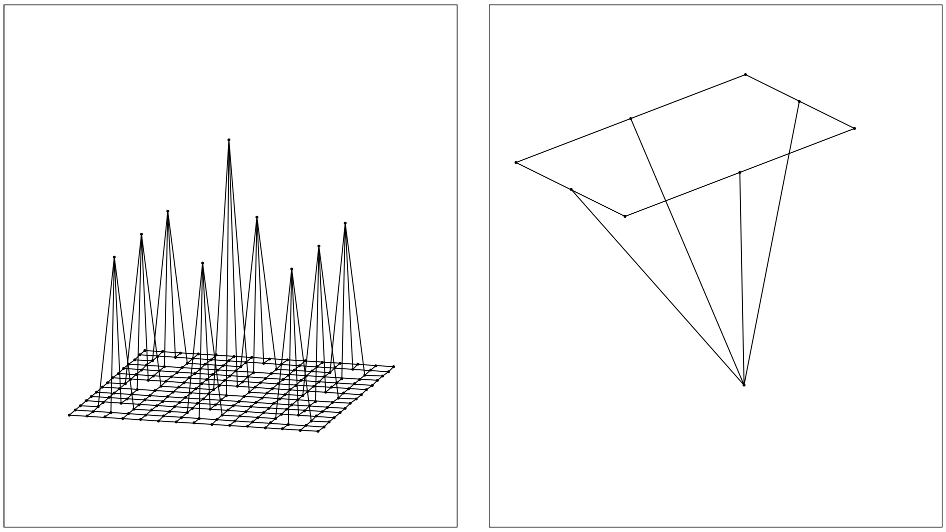


Abbildung 4: Kontrollnetze für Beispiel 2. Die erste Fläche hat ein Kontrollnetz von 15×15 Kontrollpunkten, die zweite von 3×3. Durch die Wahl der Kontrollpunkte entstehen 9 voneinander unabhängige Schnittkurven. Darum sollten sich mindestens 9 Prozessoren, bzw. *Threads* effektiv einsetzen lassen.

einer Anfangsverteilung der Arbeit ohne ausgleichende Maßnahmen (engl.: *Load Balancing*) zu entwickeln. Diese Ausgleichsmaßnahmen erfordern nochmals Kommunikation. Allerdings dauert diese Verteilung im Verhältnis zur Berechnung nicht lange. Durch einfache Synchronisation kann erreicht werden, daß arbeitende *Threads* nicht merklich gestört werden.

Da der Algorithmus scheinbar mit einem Minimum an Kommunikation auskommt und keine Arbeit doppelt durchgeführt wird, ist die Parallelisierung sehr vielversprechend. Es ist zu erwarten, daß die Beschleunigung bei Erhöhung der Prozessoranzahl fast linear ansteigt, während die Effizienz annähernd konstant bleibt. Allerdings wird der lineare Anstieg nur bis zu einer vom Problem abhängenden Prozessoranzahl effektiv arbeiten. Danach wird der Anstieg der Beschleunigung abflachen, bis sie nahezu konstant wird. Für die Effizienz bedeutet dies, daß die Effizienz erst annähernd konstant bei 1 liegt und dann, bei konstanter Beschleunigung gegen 0 geht. Nimmt man beispielsweise zwei sich nicht schneidende Flächen, so wird der Algorithmus nur mit einem Prozessor effizient arbeiten. Nimmt man umgekehrt ein Flächenpaar mit vielen singulären Schnittpunkten, so können auch viele Prozessoren eingesetzt werden. Anhand der theoretischen Überlegungen läßt sich nur schätzen, wie viele Prozessoren für ein gegebenes Flächenpaar effektiv eingesetzt werden können.

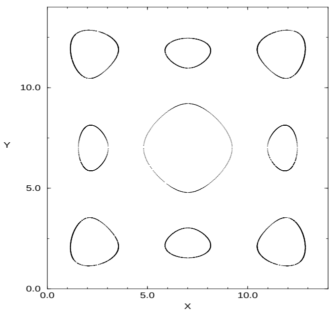


Abbildung 5: Schnittkurven des zweiten Beispiels für die in Abbildung 4 gezeigten Kontrollnetze(Schnitt durch die XY–Ebene).

Neben theoretischen Überlegungen wurden bei der Umsetzung des Algorithmus auch praktische Eigenschaften berücksichtigt. Hierzu zählen unter anderem Verfügbarkeit und Bereitstellung von Kontrollnetzdaten am jeweiligen Prozessor, sowie eine Minimierung des Speicheraufwandes durch eine *Garbage Collection*[9]. Der Algorithmus wurde in der Programmiersprache C implementiert, die es erlaubt, komplexe Datenstrukturen, die im Algorithmus benötigt werden, umzusetzen. Das Programm wurde erst auf einem Arbeitsplatzrechner ausgetestet. In das sequentielle Programm wurden

[9]Freigabe des Speichers von nicht mehr benötigten Daten.

dann Synchonisationselemente der pthread–Library für die Parallelisierung eingebaut. Zur Synchronisierung wurden hauptsächlich *Mutex locks* (siehe [KSR C '92]) verwendet.

3 Ergebnisse

Im vorigen Abschnitt konnten nur tendenzielle Aussagen aufgestellt werden. Zeitmessungen mit mehreren Testbeispielen sollen die tendenziellen Aussagen bestätigen. Dazu wurden Flächenpaare mit unterschiedlichen *Thread*–Anzahlen getestet (siehe Abbildung 7).

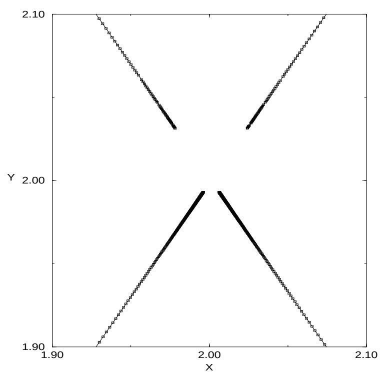


Abbildung 6: Fall 3 (Ausschnitt): Wie die Vergrößerung zeigt, zerfällt die Kurve in zwei nicht geschlossene Kurven.

Zur Bewertung der Parallelisierung wurden die Laufzeiten des Programms mit dem UNIX–Kommando `time` gemessen. Als Ergebnis von `time` werden die Real–, User– und System–Time ausgegeben. Die Real–Time, die die Zeitdauer des Programmlaufs angibt, ermöglicht, die theoretisch berechneten, dimensionslosen Größen, Beschleunigung und Effizienz, zu überprüfen. Die User–Time ist die Summe der Zeiten, der an der Berechnung beteiligten Prozessoren. Wenn deutlich mehr CPU–Zeit benötigt wird, spricht dies dafür, daß, im Vergleich zum sequentiellen Programm, Berechnungen mehrfach durchgeführt werden. System–Time fällt an, wenn Speicher allokiert wird, oder wenn Daten im Speicher hin– und hergeschoben oder ausgegeben werden müssen. Zu dieser Zeit addieren sich bei einem parallelen Programm auch die Wartezeiten an Mutexes und Barriers. Wie man in Abbildung 7 erkennt, lohnt sich eine Parallelisierung für das erste Testbeispiel (wie in Abbildung 1, wobei die Ebene so liegt, daß die Kurve einen Doppelpunkt hat) kaum. Der Algorithmus arbeitet nur bis zum Einsatz von 4 *Threads* effizient. Danach steigt die Beschleunigung nur noch unwesentlich, so daß die Effizienz langsam gegen 0 strebt. Dieses Ergebnis entspricht den Erwartungen. Die Schnittkurve in Beispiel 1 hat einen Doppelpunkt. Da der Doppelpunkt selbst nicht erreicht wird, laufen (lokal gesehen, wie in Abbildung 6) vier Geraden auf diesen Doppelpunkt zu. Die Berechnung der vier Geraden benötigt einen Großteil der Rechenzeit. Für jede dieser vier Kurven wird im wesentlichen nur ein *Thread* benötigt.

Das zweite Beispiel (siehe Abbildung 4) ist aufgrund der theoretischen Betrachtungen so gewählt, daß möglichst viele voneinander unabhängige Schnittkurven entstehen. Das Ergebnis bestätigt die theoretischen Überlegungen: Die Anzahl der effizient nutzbaren *Threads* läßt sich beliebig vergrößern. Der Algorithmus arbeitet mit einer beliebigen Anzahl von *Threads* effizient, wenn die Flächen entsprechend ausgewählt werden[10]. Im Vergleich mit Beispiel 1 fällt auf, daß die Systemzeit mit wachsender *Thread*–Anzahl in diesem Beispiel nicht so stark ansteigt (was zu erwarten wäre, wenn *Threads* sich gegenseitig behindern). Der Grund für den starken Anstieg in Beispiel 1 ist somit als Wartezeit der *Threads*, und nicht als programm–interner Verwaltungsaufwand zu bewerten. In Beispiel 1 müssen die *Threads* warten, weil nicht genügend Arbeit anliegt.

Wenn man die beiden Testbeispiele miteinander vergleicht, fällt ein zweiter, wichtiger Faktor auf, der für das bessere Verhalten der Effizienz in Beispiel 2 verantwortlich ist. Statt eines 9×5 großen Netzes werden im zweiten Beispiel 15×15 Kontrollpunkte verwendet. Diese Größe beeinflußt nicht die Subdivisionstiefe, sondern nur die Zeit, die pro Subdivisionsschritt notwendig ist. Während bei kleinen Kontrollnetzen die Rechenzeiten der Verteilungsroutine und der *Thread*–Verwaltung stark ins Gewicht fallen, sind sie bei größeren Netzen vernachlässigbar.

Die Versuche haben gezeigt, daß der Algorithmus im allgemeinen nicht effizient massiv (also etwa 64 Prozessoren) parallelisierbar ist, da für normale Schnittprobleme mit niedrigem Kontrollnetzgrad nur wenig Prozessoren benötigt werden. Die Beschleunigung hat sich bei den behandelten Problemen schon bei wenigen *Threads*

[10] Allerdings werden dazu immer größere Kontrollnetze benötigt. Bei Beispiel 2 wurde ein Kontrollnetz mit 15×15 Kontrollpunkten verwendet (siehe Abbildung 4).

Rechenzeiten für Beispiel 1, (siehe Abbildung 1, 3(b)). Schnitt von Flächen mit Kontrollnetzen der Größe 9×5 und 2×2. Die Schnittkurve besitzt ein Doppelpunkt.

Rechenzeiten für Beispiel 2, (siehe Abbildung 4).Schnitt von Flächen mit Kontrollnetzen der Größe 15×15 und 3×3. Die Schnittkurve zerfällt in 9 geschlossene Schnittkurven.

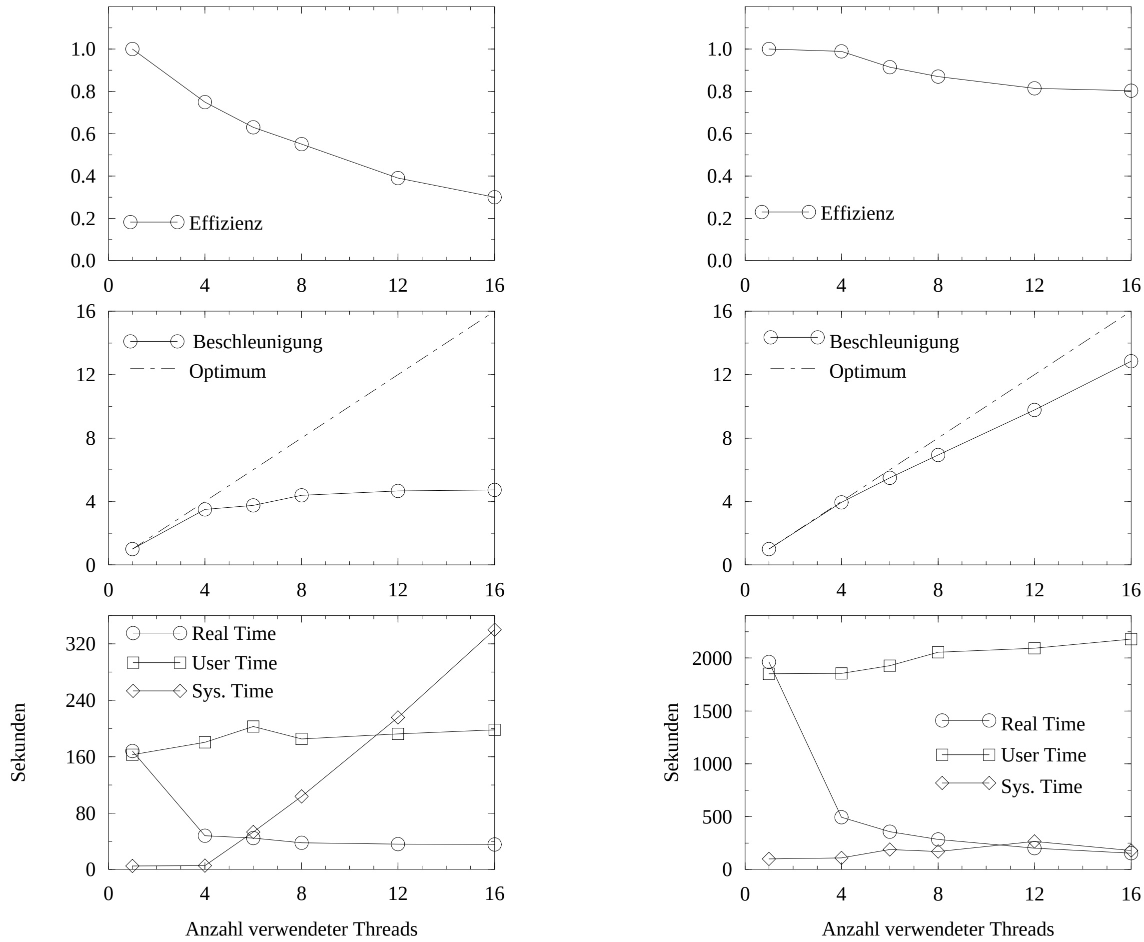


Abbildung 7: Mit `time` gemessene Rechenzeiten. Aus der Real–Time kann man die dimensionslosen Größen Beschleunigung und Effizienz berechnen. Die Beschleunigung fällt im ersten Beispiel ab, weil nicht genügend voneinander unabhängige Arbeit für alle Prozessoren vorliegt.

beziehungsweise Prozessoren nur noch geringfügig verbessert. Massive Parallelisierung lohnt sich mit diesem Algorithmus nur für besondere Fälle, z.B. für Kontrollnetze mit vielen Kontrollpunkten. Allerdings zeigen die Ergebnisse, daß sich der hier vorgestellte Algorithmus für viele Schnittkurvenberechnungen sehr gut mit einer kleineren Anzahl von *Threads* parallelisieren läßt. Die Anzahl, für die alle Prozessoren noch effizient arbeiten ist stark vom Problem abhängig. Je mehr Kurvenpunkte berechnet werden müssen, desto mehr Prozessoren lassen sich sinnvoll einsetzen. Es hat sich herausgestellt, daß kein Algorithmus, dessen Parallelisierung auf der rekursiven Subdivision basiert, eine bessere Effizienz erwarten läßt.

Literatur

[Barnhill '87] R. E. Barnhill, G. Farin, M. Jordan, B. R. Piper: *Surface/surface intersection,* Computer Aided Geometric Design, volume 4, 3–16, North–Holland, 1987

[Barnhill '90] R. E. Barnhill, S. N. Kersey: *A marching method for parametric surface/surface intersection,* Computer Aided Geometric Design, volume 7, 257–280, North–Holland, 1990

[Bürger '92] H. Bürger: *Ein Mehrphasenalgorithmus für Parallelrechner zu Bestimmung von Schnittkurven zweier Flächen*, Dissertation, Göttingen, 1992

[Farin '90] G. Farin: *Curves and surfaces for computer aided geometric design, a practical guide,* Second Edition, Academic Press, Inc., New York, 1990

[Houghton '85] E. G. Houghton, R. F. Emnett, J. D. Factor, C. L. Sabharwal: *Implementation of a divide–and–conquer method for intersection of parametric surfaces,* Computer Aided Geometric Design, volume 2, 173–183, North–Holland, 1985

[JáJá '92] J. JáJá: *An Introduction to Parallel Algorithms,* Addison Wesley, 1992

[KSR C '92] Kendall Square Research: *KSR C Programming Manual*, Waltham, 1992

[Schaback '89] R. Schaback: *A local tracing method for computing intersection curves of surfaces in computer aided design,* Vortragsskizzen, Göttingen, 1989

[Schaback '92] R. Schaback: *Error Estimates for Approximationsfrom Control Nets* Computer Aided Geometric Design, North–Holland,1992

[Trapp '94] J. Trapp: *Ein parallelisierbarer Schnittkurvenalgorithmus,* Diplomarbeit (in Vorbereitung), Göttingen, 1994

[Weberpals '92] H. Weberpals: *Grundlagen paralleler Algorithmen,* Vorlesungsmitschrift, Göttingen, 1992/93