

FHTW

Fachhochschule für Technik und Wirtschaft Berlin
University of Applied Sciences

Thema:

Kürzester Weg **“All Pairs Shortest Path”**

Beleg Parallele Systeme
Wintersemester 2005/06

Bearbeiter: Frank Sommer (s 050 6636)
Nico Korf (s 050 6354)
Thomas Rega (s 038 9735)
Thilo Berndt (s 050 6575)

0. Inhaltsverzeichnis

0.	Inhaltsverzeichnis	2
1.	Einführung	3
2.	Algorithmus.....	3
3.	Parallelisierung	4
3.1.	Partitionierung.....	4
3.2.	Kommunikation.....	4
4.	Theoretische Analyse	5
4.1.	Speedup.....	6
4.2.	Effizienz	7
5.	Ergebnisse	8
5.1.	Fazit.....	9
6.	Profiling	10
7.	Visualisierung	11
8.	Quellen.....	16

1. Einführung

Das Problem des kürzesten Weges gehört zu der Graphentheorie, wobei alle Arten von Graphen betrachtet werden können. Für die Lösung des Problems gibt es verschiedene Algorithmen. Der wohl bekannteste Vertreter ist der Algorithmus von Dijkstra. Dieser Algorithmus wird auch im Internet zum Routing (beim Protokoll *OSPF*) benutzt. Erweiterungen dieses Algorithmus bzw. Alternativen sind der *-Algorithmus, der Floyd-Warshall-Algorithmus und der Bellman-Ford-Algorithmus.

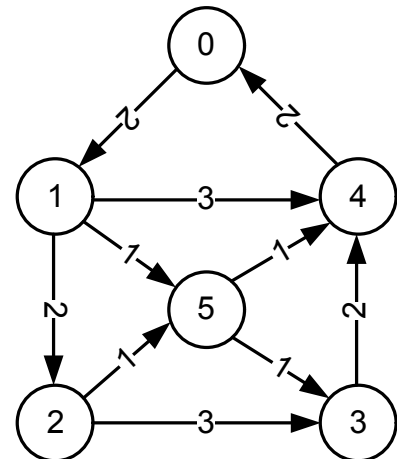


Abbildung 1 – Beispielnetz

2. Algorithmus

Der Floyd-Warshall-Algorithmus existiert in zwei Varianten, einmal nach Floyd für die kürzesten Wege und nach Warshall für die reflexive, transitive Hülle. Da uns die kürzesten Wege interessieren, betrachten wir an dieser Stelle nur die Variante nach Floyd. Diese findet die kürzesten Wege zwischen allen Paaren von Knoten innerhalb eines gewichteten Graphen (engl. *All Pairs Shortest Path*). Dieser Graph kann gerichtet, als auch ungerichtet sein.

Als Input dient eine Adjazenz- bzw. Gewichtsmatrix, welche die Verbindungen und die evtl. Gewichte beinhaltet. Die Funktionsweise des Algorithmus ist relativ einfach. Dabei wird stets für alle Paare an Knoten (i, j) und einem aktuellen Knoten (k) geprüft, ob der Weg zwischen i und j länger ist als der „Umweg“ über k. Falls das der Fall ist, ist die neue Entfernung zwischen i und j die Summe der Verbindungen über k (siehe Abbildung 3).

	0	1	2	3	4	5
0	0	∞	∞	∞	2	∞
1	2	0	∞	∞	∞	∞
2	∞	2	0	∞	∞	∞
3	∞	∞	3	0	∞	1
4	∞	3	∞	2	0	1
5	∞	1	1	∞	∞	0

Abbildung 2 – Adjazenzmatrix

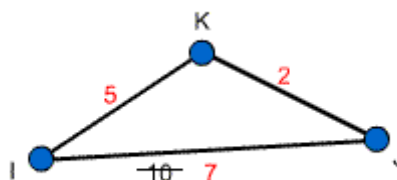


Abbildung 3 - Prüfung

Wie sich daraus schon erkennen lässt, sind für diesen Algorithmus drei verschachtelte Schleifen notwendig. Daraus ergibt sich folgender Aufwand.

$$\Theta(n^3)$$

3. Parallelisierung

Um einen Algorithmus zu parallelisieren bieten sich verschiedene Strategien an. Beim Floyd-Algorithmus ist es das Prinzip Divide-And-Conquer. Hierbei werden die Daten auf die verschiedenen Prozessoren aufgeteilt. Dies ist möglich da auf alle Knoten der Adjazenzmatrix dieselben Operationen ablaufen, nämlich ein Vergleich mit eventueller Zuweisung. Da aber für die Operationen Daten von anderen Prozessen benötigt werden, kommt es zu einem erhöhten Kommunikationsaufwand.

3.1. Partitionierung

Die Adjazenzmatrix kann auf verschiedene Arten auf die einzelnen Prozesse aufgeteilt werden. Hierbei gibt es eine zeilen-, eine spalten- und eine blockweise Aufteilung der Daten (siehe Abbildung 4). In unserer Implementierung haben wir auf die zeilenweise Aufteilung zurückgegriffen, da diese am einfachsten umzusetzen ist. Eine blockweise Aufteilung ist nur geringfügig performanter als die zeilenweise Aufteilung, jedoch gestaltet sich hierbei die Kommunikation zwischen den Prozessen durchaus schwieriger. Dazu jedoch mehr im nächsten Abschnitt. Für die zeilenorientierte Partitionierung spricht außerdem, dass Matrizen in C von vornherein zeilenweise implementiert sind.

	0	1	2	3	4	5
0	0	∞	∞	∞	2	∞
1	2	0	∞	∞	∞	∞
2	∞	2	0	∞	∞	∞
3	∞	∞	3	0	∞	1
4	∞	3	∞	2	0	1
5	∞	1	1	∞	∞	0

Abbildung 4 - Partionierungsmöglichkeiten

3.2. Kommunikation

Wie schon erwähnt benötigt jeder Prozess für die Ermittlung der kürzeren Wege Daten von anderen Prozessen. Da jeder Prozess für seinen Datenbereich die alternativen Wege, also den "Umweg" über k , für i und j prüft benötigt er alle Werte aus der k -ten Zeile (siehe Abbildung 5).

	0	1	2	3	4	5
0	0	∞	∞	∞	2	∞
1	2	0	∞	∞	∞	∞
2	∞	2	0	∞	∞	∞
3	∞	∞	3	0	∞	1
4	∞	3	∞	2	0	1
5	∞	1	1	∞	∞	0

Abbildung 5 - Kommunikationsmuster

4. Theoretische Analyse

Wie schon festgestellt hat der sequentielle Algorithmus eine Komplexität von $\Theta(n^3)$. Beim parallelen Algorithmus ist erst einmal die innerste Schleife identisch.

$$\Theta(n)$$

Die zweite Schleife läuft über die auf die Prozessoren verteilten Daten. Das heißt über n/p Iterationen.

$$\Theta\left(\frac{n^2}{p}\right)$$

Vor den zwei Schleifen findet ein Broadcast von n Werten statt und hat daher eine Komplexität von $\Theta(n)$. Das Broadcasten auf die Prozessoren benötigt $(\log p)$ message-passing Schritte. Dies bedeutet der Broadcast schlägt mit einer Komplexität von $\Theta(n \log p)$ zu Buche.

Wie im vorigen Kapitel festgehalten, wird vom Besitzer der k -ten Zeile diese per Broadcast verteilt. Das Vorbereiten dieser zu verteilenden Zeile benötigt n Schritte. Das Ermitteln Besitzers der k -ten Zeilen ist ein Schritt.

$$\Theta\left(1 + n + n \log p + \frac{n^2}{p}\right)$$

Nun wird das ganze Prozedere für alle k durchgeführt. Das heißt also n -mal.

$$\Theta\left(n\left(1 + n + n \log p + \frac{n^2}{p}\right)\right) = \Theta\left(\frac{n^3}{p} + n^2 \log p + n^2 + n\right)$$

4.1. Speedup

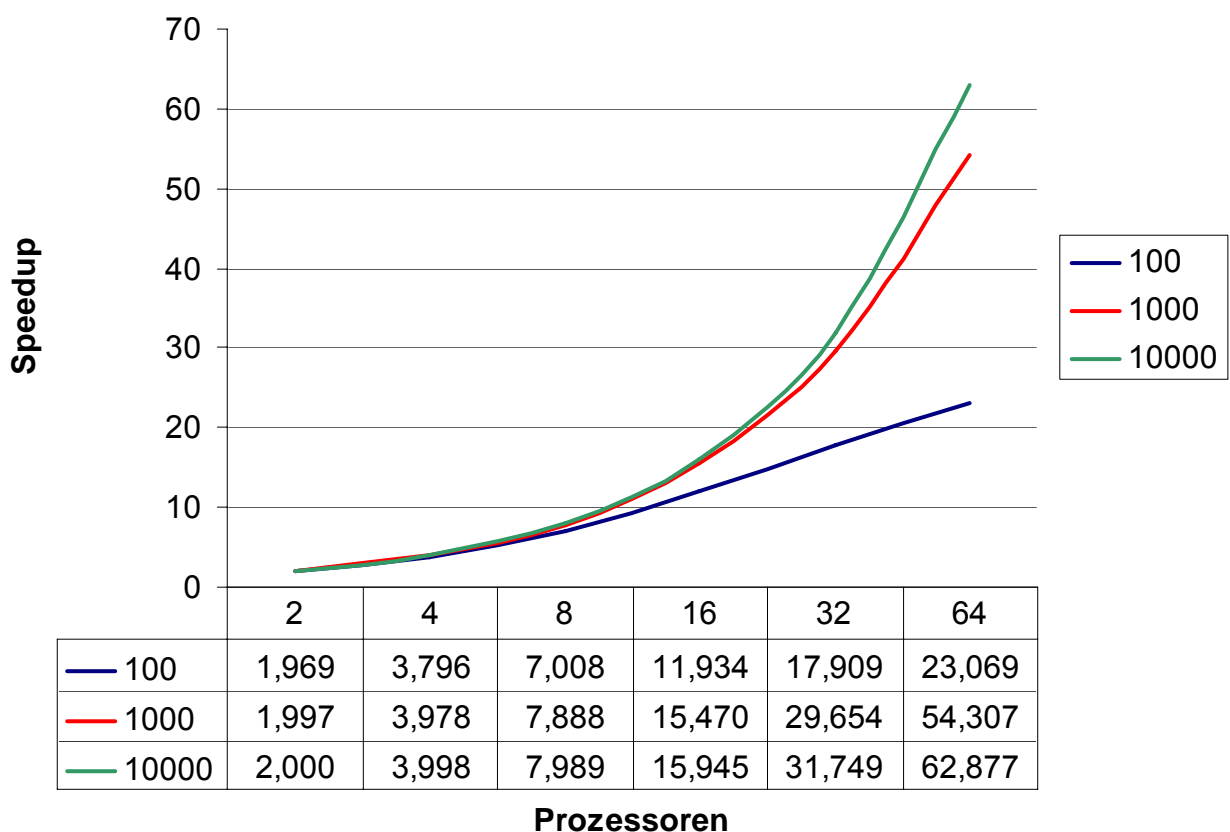
$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p + \kappa(n, p)}$$

D.h. Sequentielle Laufzeit durch Parallele Laufzeit. $\sigma(n)$ ist der sequentielle Bestandteil des Algorithmus, $\varphi(n)$ ist der parallel ablaufende Teil und $\kappa(n, p)$ ist die benötigte Kommunikation zwischen den Prozessoren.

Aus der Formel lässt sich erkennen, dass sich mit steigenden Prozessoren der Parallelanteil verringert, jedoch vergrößert sich der Kommunikationssaufwand.

Daraus folgt:

Speedup Floyd-Algorithmus

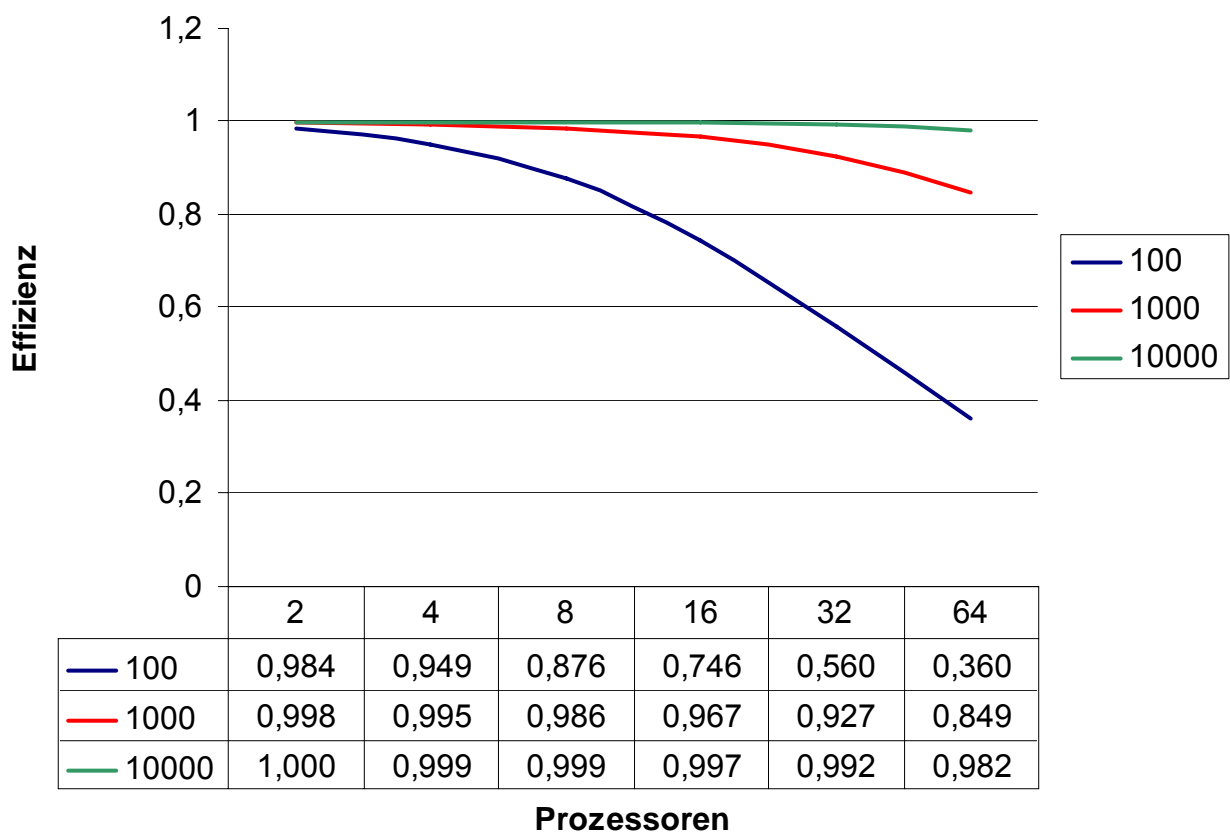


4.2. Effizienz

Zur Berechnung der Effizienz wird Formel für den Speedup auf den einzelnen Prozessor erweitert. Daraus folgen folgende Formel und Diagramm für die Effizienz.

$$\varepsilon(n, p) \leq \frac{\sigma(n) + \varphi(n)}{p \cdot (\sigma(n) + \varphi(n)/p + \kappa(n, p))} = \frac{\sigma(n) + \varphi(n)}{p \cdot \sigma(n) + \varphi(n) + p \cdot \kappa(n, p)}$$

Effizienz Floyd-Algorithmus

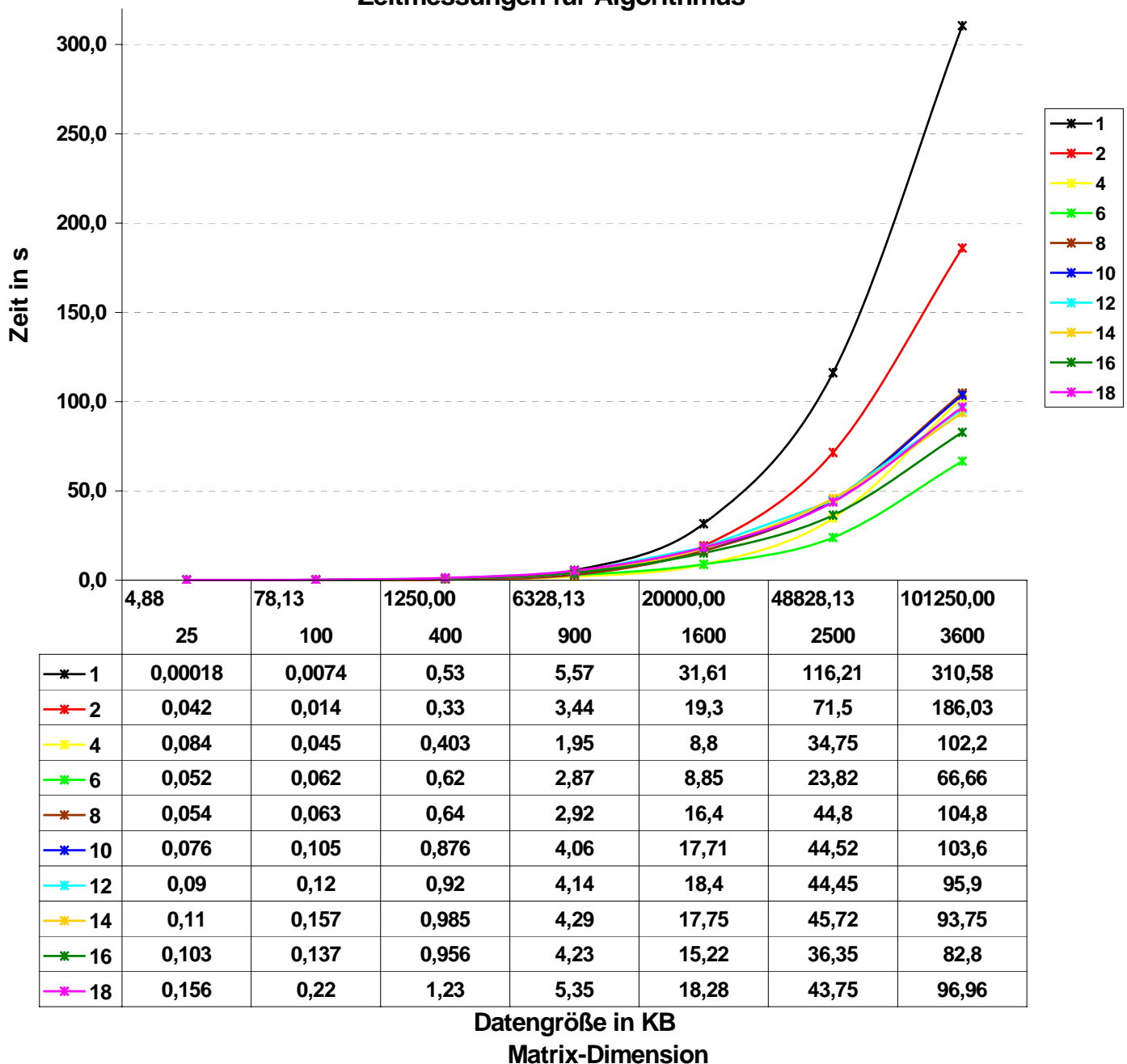


5. Ergebnisse

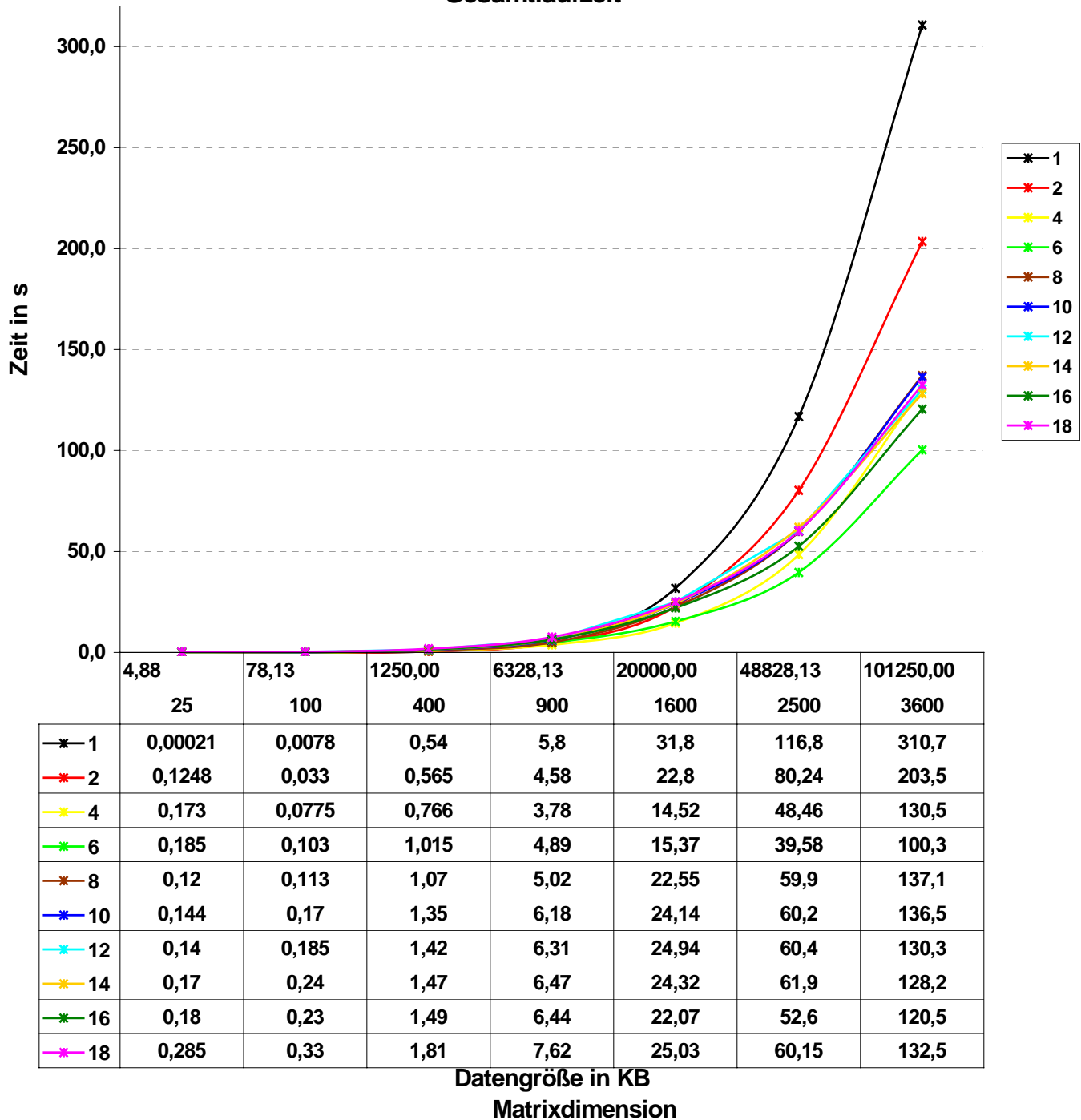
Das Programm wird verschiedenen Zeitmessungen unterzogen. Bei jeder einzelnen Konfiguration (Anzahl der Prozessoren und Größe der Adjazenzmatrix) werden fünf Zeitmessungen durchgeführt, um eventuelle Störungen (hohe Netzwerklast etc.) herauszufiltern.

Die Messungen werden für 1, 2, 4, 6, 8, 10, 12, 14, 16 und 18 Prozessoren und sechs unterschiedlichen Adjazenzmatrizen (Dimension 25, 100, 400, 900, 1600, 2500 und 3600) durchgeführt. Größere Matrizen sind aufgrund der Speicherbegrenzung auf den Laborrechnern nicht möglich. Daraus ergeben sich dann folgende Messergebnisse und Diagramme. Des Weiteren wird in Laufzeit des Algorithmus und in Gesamtlaufzeit des Programms unterschieden.

Zeitmessungen für Algorithmus



Gesamtlaufzeit



Bei der Gesamtlaufzeit wird das Verteilen und Einsammeln der Daten ebenfalls berücksichtigt.

5.1. Fazit

Bei kleinen Matrizen (von 25 bis 400) zeigt sich bei der Parallelisierung noch keine Vorteile. Für diese Fälle ist der Aufwand der Kommunikation größer als die Einsparung durch die Parallelisierung der Datenauswertung. Bei größeren Matrizen kommt die Parallelisierung erst zur Geltung, wobei die ideale Prozessoranzahl mit Größe der Eingabematrix steigt. In unserem Messbereich lässt sich ein Optimum von 6 Prozessoren für diesen

Algorithmus aus den Messdaten ablesen. Des Weiteren ragen die Werte für 16 Prozessoren hervor, welche wohl erst bei größeren Matrizen ihre wirklichen Vorteile ausspielen.

6. Profiling

MPICH bietet die Möglichkeit die Laufzeit einer verteilten Applikation in einem Logfile zu protokollieren. Jeder Prozess erzeugt dazu in seiner Umgebung ein Logfile, in dem er alle MPICH-Aktionen mit Zeitstempel mitprotokolliert. Am Ende der Laufzeit werden die Daten an den Root-Prozess gesendet und zu einem Logfile gemergt. In der Grundeinstellung verwendet MPICH das RLOG-Format, welches wir in unserer Laufzeitumgebung auch verwenden.

Mit dem Logfile hat man nun die Möglichkeit die Laufzeit des Programms nach Beendigung zu analysieren und eventuelle Probleme beim Kommunikationsverhalten zu erklären.

Für die Analyse haben wir auf die JAVA-Applikation Jumpshot-4 zurückgegriffen. Diese ist Bestandteil des MPE (MultiProcessing Environment), welches sich im MPICH-Paket befindet. Diese Environment gibt dem Entwickler Werkzeuge in die Hände, mit der er die Performance der geschriebenen Applikation überprüfen kann.

Jumpshot-4 benötigt für die Visualisierung der Daten das SLOG-2-Format. Im Gegensatz zum RLOG-Format werden die Daten in SLOG-2 skalierbar abgespeichert. Das bedeutet, dass Jumpshot-4 oder jeder beliebige andere Viewer nicht die komplette Logfile sequentiell einlesen muss um die Daten darzustellen. Je nach Detailgrad der Darstellung werden nur bestimmte Daten aus der Logfile gelesen. Das kann bei Logfiles mit Größen weit über 100 MB einen großen Performanzvorteil bringen.

In Jumpshot-4 werden alle Prozesse entlang einer Zeitachse visualisiert, wobei die einzelnen Funktionen innerhalb der Prozesse durch verschiedene Farben voneinander getrennt sind. Kommunikationen zwischen Prozessen (MPI_Send, MPI_Recv) werden durch Pfeile dargestellt. Mit Hilfe der Zoomfunktion kann der Benutzer bequem die Zeitachse modifizieren und per Maus entlang der Zeitachse scrollen.

Weil wir mit MPICH RLOG-Files erzeugt haben, müssen diese vor der Nutzung mit Jumpshot-4 in das SLOG-2-Format konvertiert werden. Zu unserem Glück haben die Entwickler mitgedacht und einen Konverter für die häufigsten Formate integriert.

Es ist wichtig zu nennen, dass Jumpshot-4 oder vergleichbare Viewer den Entwickler beim Profiling der geschriebenen Programme unterstützen, jedoch nicht Fehler in der Implementierung beseitigen. Sie geben dem Entwickler aber ein wichtiges Werkzeug in die Hand um Performance-Engpässe zu erkennen und zu beheben.

7. Visualisierung

Für die praktische Umsetzung des parallelen Programms und dessen Visualisierung, kamen verschiedene Scriptsprachen, ANSI C und Java zur Anwendung. In Abbildung 6.1 sieht man einen kurzen Überblick:

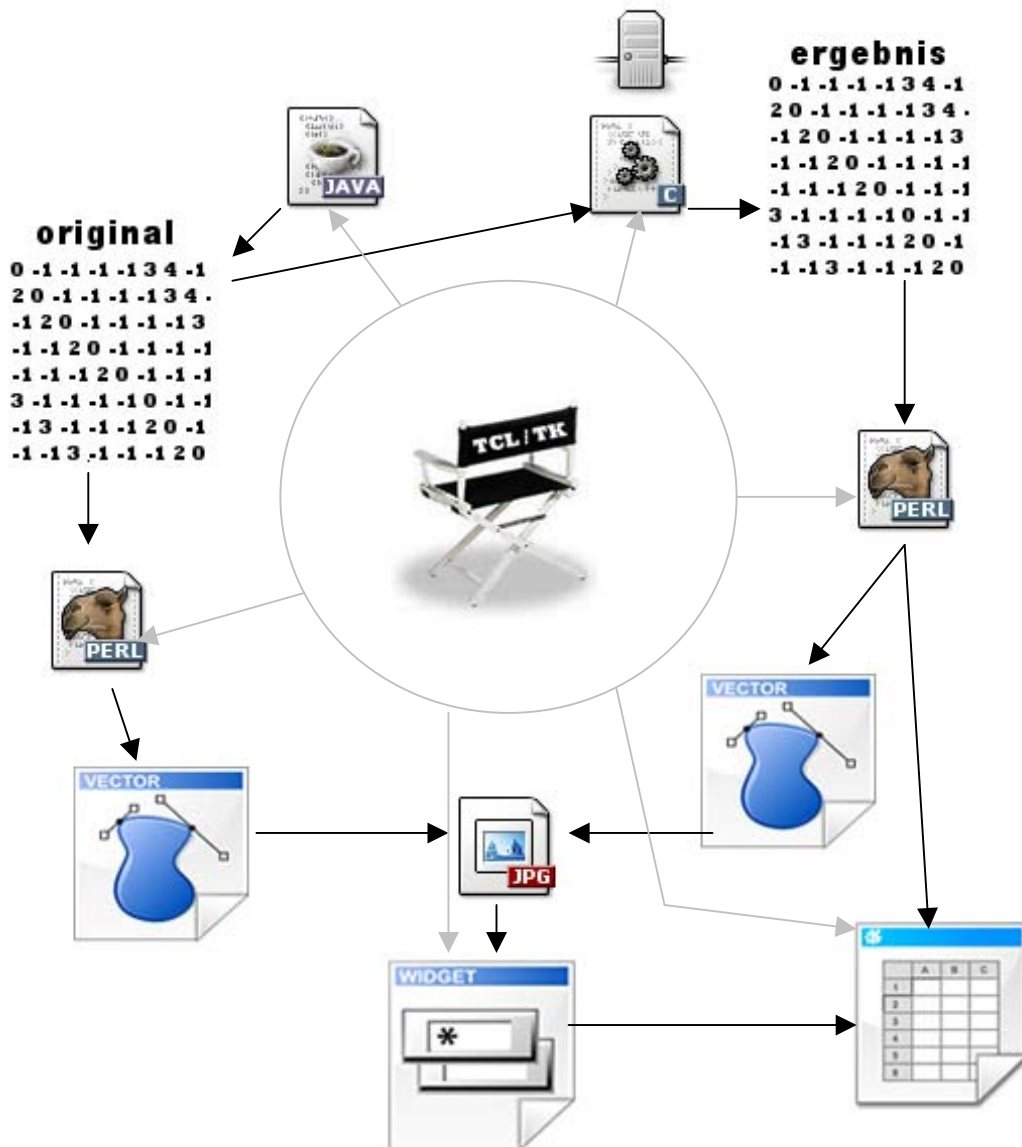


Abbildung 6 - Übersicht – Zusammenspiel Software

Als Skriptsprachen wurden Perl und Tcl verwendet, wobei dem in Tcl „verfassten“ Teil eine Art Manager-Funktion zu kommt. Die einzelnen verwendeten Komponenten und ihr Zusammenspiel sollen im Folgenden der Reihenfolge nach erläutert werden.

Jede Komponente wurde einzeln für sich entwickelt, wobei die je nach gestellter Aufgabe, die Programmiersprache gewählt wurde, die für diesen Zweck am geeignetsten schien. Den „Zusammenschluss“ fanden die einzelnen Komponenten in einem Tcl-Skript, welches die separaten Programm-Teile „in Szene setzt“.

Das Tcl Script erledigt im eigentlichen Sinne nicht viel Arbeit. Vielmehr verteilt es wie ein Drehbuchautor (engl. script -> Drehbuch) die einzelnen Rollen an die „Schauspieler“, die anderen Programme (C, Perl, Java), und „führt dann nur noch Regie“, in dem es z. B. den Zeitpunkt und die Art und Weise des Programmaufrufes festlegt. Mit der „eigentlichen Arbeit“, die verrichtet wird, hat es jedoch weiter nichts „am Hut“.

Doch der Reihe nach.

Die **Generierung** einer **Adjazenzmatrix**¹ übernimmt ein **Java** Programm, welches bequem über Parameter aufgerufen werden kann.

Diese Adjazenzmatrix bildet die Grundlage für die **parallele Berechnung** der kürzesten Wege mittels des Floyd-Algorithmus.

Der **Floyd Algorithmus** wurde in **C** umgesetzt und mittels des **MPI**-Protokolls für die parallele Berechnung der Lösung, die Berechnung aller kürzesten Wege in dem jeweiligen Graphen, benutzt.

Zur **Visualisierung** des **Graphen** wurde „**Graphviz**“ benutzt. Dabei handelt es sich um „ein on AT&T und den Bell-Labs entwickeltes Open-Source-Programmpaket zur Visualisierung von gerichteten und ungerichteten Graphen“ [wikipedia]. Da Graphviz selbst mit Skripten als Eingabe zur Generierung arbeitet, war es nötig sich solches Skript, in Abhängigkeit zum aktuellen Graphen, automatisch generieren zu lassen.

Hier kam Perl ins Spiel. Die Perl „innewohnenden“ Eigenschaften, die es ermöglichen, auf einfache Art und Weise mittels regulärer Ausdrücke eine Datei zu parsen und deren Inhalt durch entsprechende Ausdrücke zu ersetzen, prädestinieren Perl für diese Aufgabe.

Mittels **Perl** wird die **generierte Adjazenzmatrix geparkt** und aus den gegebenen Daten das **Eingabeskript** für die **Graphengenerierung erzeugt**. Dieses Skript kommt dann zur Ausführung und generiert die Visualisierung des Graphen als **jpg** Datei.

Das erzeugte Bild wird dann in die mittels **Tcl/Tk** erzeugte **graphische Benutzeroberfläche** geladen und durch diese dargestellt.

¹ „**Adjazenz** bezeichnet eine Beziehung zwischen Knoten oder Kanten in einem Graphen. Zwei Knoten heißen in einem Graph **adjazent** oder **benachbart**, wenn sie in diesem durch eine Kante verbunden sind.“
[<http://de.wikipedia.org/wiki/Adjazenz>]



Abbildung 7 - Welcome

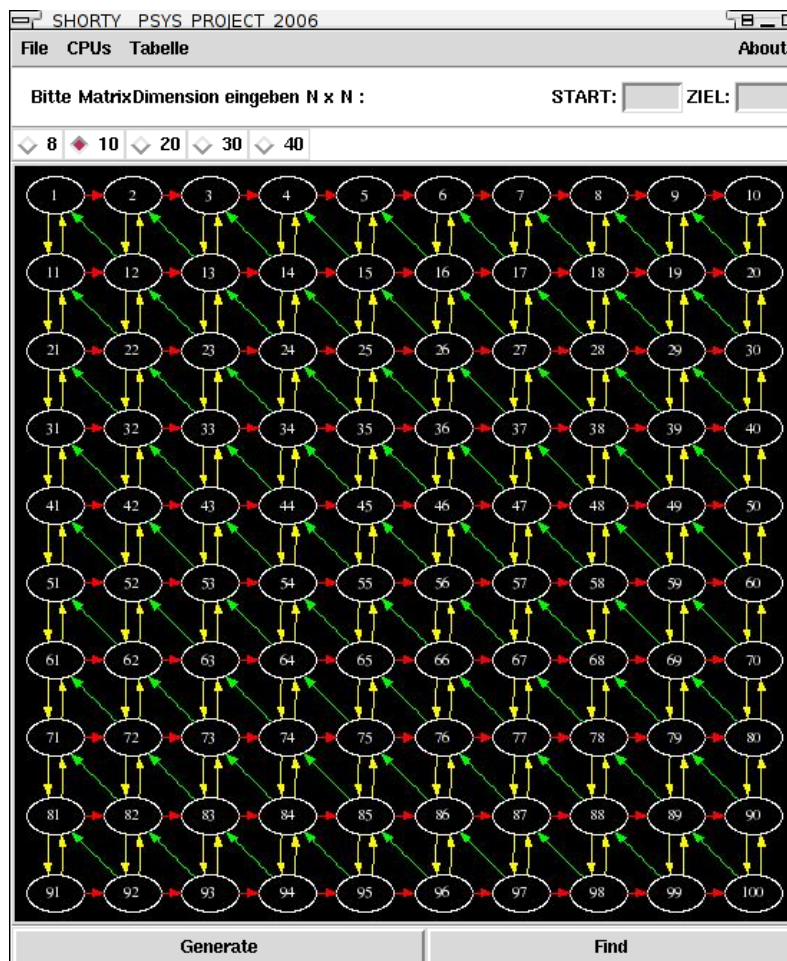


Abbildung 8 - generierter Graph

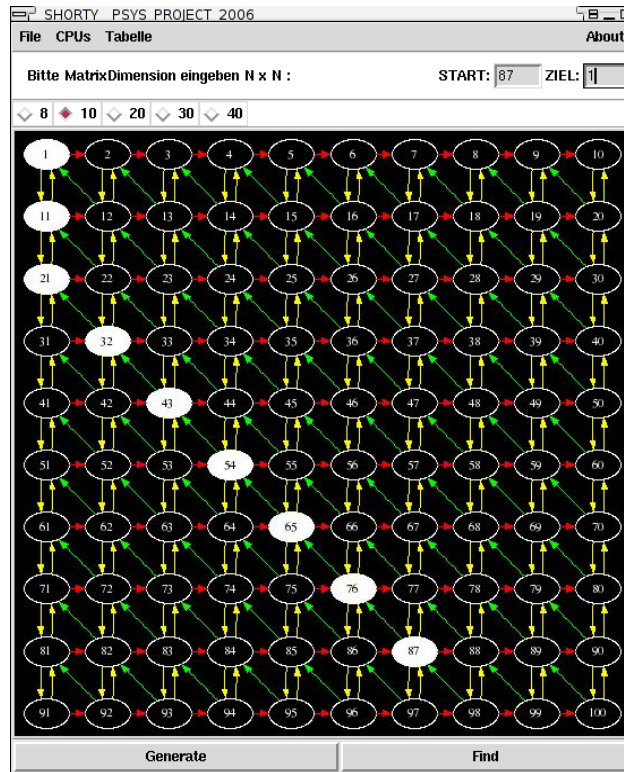


Abbildung 9 - Visualisierung der Lösung



Abbildung 10 - Darstellung der Zeitdauer des Programmlaufs

	k0	k1	k2	k3	k4	k5	k6	k7	k8	k9
k1	-	7,00	14,00	21,00	28,00	35,00	42,00	49,00	56,00	
k2	2,00	-	7,00	14,00	21,00	28,00	35,00	42,00	49,00	
k3	4,00	2,00	-	7,00	14,00	21,00	28,00	35,00	42,00	
k4	6,00	4,00	2,00	-	7,00	14,00	21,00	28,00	35,00	
k5	8,00	6,00	4,00	2,00	-	7,00	14,00	21,00	28,00	
k6	10,00	8,00	6,00	4,00	2,00	-	7,00	14,00	21,00	
k7	12,00	10,00	8,00	6,00	4,00	2,00	-	7,00	14,00	
k8	14,00	12,00	10,00	8,00	6,00	4,00	2,00	-	7,00	
k9	16,00	14,00	12,00	10,00	8,00	6,00	4,00	2,00	-	
k10	18,00	16,00	14,00	12,00	10,00	8,00	6,00	4,00	2,00	-
k11	3,00	10,00	17,00	24,00	31,00	38,00	45,00	52,00	59,00	
k12	5,00	3,00	10,00	17,00	24,00	31,00	38,00	45,00	52,00	
k13	7,00	5,00	3,00	10,00	17,00	24,00	31,00	38,00	45,00	
k14	9,00	7,00	5,00	3,00	10,00	17,00	24,00	31,00	38,00	
k15	11,00	9,00	7,00	5,00	3,00	10,00	17,00	24,00	31,00	
k16	13,00	11,00	9,00	7,00	5,00	3,00	10,00	17,00	24,00	
k17	15,00	13,00	11,00	9,00	7,00	5,00	3,00	10,00	17,00	
k18	17,00	15,00	13,00	11,00	9,00	7,00	5,00	3,00	10,00	
k19	19,00	17,00	15,00	13,00	11,00	9,00	7,00	5,00	3,00	
k20	21,00	19,00	17,00	15,00	13,00	11,00	9,00	7,00	5,00	
k21	6,00	13,00	20,00	27,00	34,00	41,00	48,00	55,00	62,00	
k22	8,00	6,00	13,00	20,00	27,00	34,00	41,00	48,00	55,00	
k23	10,00	8,00	6,00	13,00	20,00	27,00	34,00	41,00	48,00	
k24	12,00	10,00	8,00	6,00	13,00	20,00	27,00	34,00	41,00	
k25	14,00	12,00	10,00	8,00	6,00	13,00	20,00	27,00	34,00	
k26	16,00	14,00	12,00	10,00	8,00	6,00	13,00	20,00	27,00	
k27	18,00	16,00	14,00	12,00	10,00	8,00	6,00	13,00	20,00	
k28	20,00	18,00	16,00	14,00	12,00	10,00	8,00	6,00	13,00	
k29	22,00	20,00	18,00	16,00	14,00	12,00	10,00	8,00	6,00	
k30	24,00	22,00	20,00	18,00	16,00	14,00	12,00	10,00	8,00	
k31	9,00	16,00	23,00	30,00	37,00	44,00	51,00	58,00	65,00	
k32	11,00	9,00	16,00	23,00	30,00	37,00	44,00	51,00	58,00	

Abbildung 11 - Darstellung aller kürzesten Entfernungen

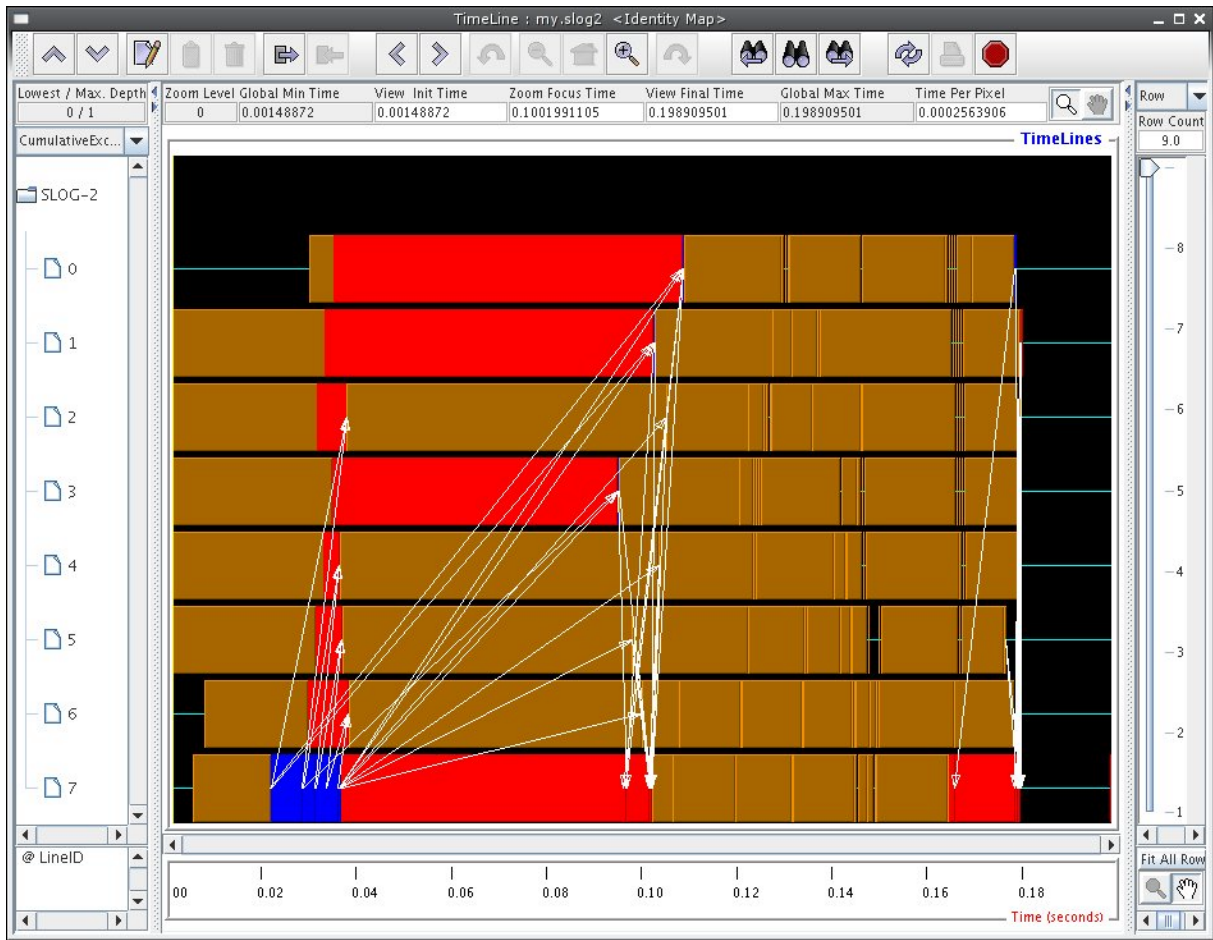


Abbildung 12 - Jumpshot-4 Profiling

8. Quellen

Wikipedia – Floyd-Algorithmus

<http://de.wikipedia.org/wiki/Floyd-Warshall-Algorithmus>

Universität Stuttgart – Floyd-Algorithmus

[http://www.ifp.uni-stuttgart.de/\[...\]/gfe_MO_NA3_de_4.html](http://www.ifp.uni-stuttgart.de/[...]/gfe_MO_NA3_de_4.html)

Universität Leipzig - Parallelverarbeitung und Komplexe Systeme

[http://pacosy.informatik.uni-leipzig.de/\[...\]/unterlagen](http://pacosy.informatik.uni-leipzig.de/[...]/unterlagen)

FH Bonn Rhein Sieg – Verteilte und parallele Systeme

[http://www2.inf.fh-bonn-rhein-sieg.de/\[...\]/ws0506/vups2/V12.pdf](http://www2.inf.fh-bonn-rhein-sieg.de/[...]/ws0506/vups2/V12.pdf)

Parallel Programming in C with MPI and OpenMP, Michael J. Quinn
McGraw-Hill Education (ISE Editions), September 2003

MPICH2 – Homepage

<http://www-unix.mcs.anl.gov/mpi/mpich2/>

MPI Course

https://www.thoreg.org/misc/Mpi_Course.pdf

Graphenvisualisierungstool

<http://www.graphviz.org/>



linux rulez ;)