Modellierung und Meßunterstützung mit dem Werkzeug PEPP

Franz Hartleb, Peter Dauphin, Rainer Klar, Andreas Quick, Markus Siegle

Institut für Mathematische Maschinen und Datenverarbeitung VII
Universität Erlangen-Nürnberg, Martensstraße 3, 91058 Erlangen
{dauphin,klar,siegle}@informatik.uni-erlangen.de

Kurzfassung: PEPP erlaubt die ablauforientierte Modellierung paralleler Programme mit stochastischen Graphmodellen. Exakte und approximative Auswerteverfahren ermöglichen Laufzeitvorhersagen für Graphmodelle von unterschiedlichem Typus und Komplexität. Darüberhinaus bietet PEPP Funktionen zur Unterstützung der modellgesteuerten Messung und die Möglichkeit, Modelle mit beliebigen gemessenen Verteilungsfunktionen zu parametrieren.

Modelling and Monitoring Support with the Tool PEPP: PEPP is a tool for modelling parallel programs using stochastic graph models. It provides exact and approximate analysis methods for runtime prediction of different graph model types. PEPP also supports model-driven monitoring and allows to use arbitrary measured distributions for model parametrization.

1 Einleitung

Dieses Papier beschreibt das Modellierungswerkzeug PEPP (*P*erformance *E*valuation of *P*arallel *P*rograms) und die ihm zugrunde liegenden Analyseverfahren. PEPP entstand aus unserem Interesse an der Leistungsbewertung paralleler Programme. Dafür werden ablauforientierte Verfahren eingesetzt, die die Ablaufdynamik derart repräsentieren, daß die Modelle *Leistungsvorhersagen* für den kompletten Programmlauf auf einer gegebenen Rechnerkonfiguration erlauben und eine *modellgesteuerte Messung* unterstützen.

Zusammen mit der rasanten Entwicklung moderner Hardware und Software hat sich das Interesse des Leistungsbewerters von der Betrachtung der reinen Maschinenleistung immer mehr hin zur Bewertung von Rechensystemen unter komplexer, strukturierter Last verlagert [7]. Je differenzierter die Ablaufstruktur der Programme modelliert wird, desto stärker steht bei der Modellierung die Beschreibung der Last und ihrer Ablaufdynamik im Vordergrund. Bei parallelen Algorithmen ist die Dauer paralleler Rechenphasen meist nicht gleichlang. Zudem gibt es neben dem Verwaltungsaufwand für die Abwicklung der Parallelisierung auch das Problem funktionaler Abhängigkeiten zwischen den sequentiellen und parallelen Programmabschnitten. Hält man sich diese Tatsachen vor Augen, so wird deutlich, warum Graphmodelle [9, 10], die einer solchen Beschreibung entgegenkommen, betrachtet werden.

Zur Darstellung eines Programmlaufs als Graphmodell wird der Algorithmus in sequentiell und parallel ablauffähige Einheiten, sogenannte *Teilaufgaben* zerlegt, zwischen denen *Reihenfolgebeziehungen* angegeben werden, die die funktionalen Abhängigkeiten der Teilaufgaben widerspiegeln. Studien zum Zweck der Leistungsbewertung setzen voraus, daß neben dem funktionalen Verhalten auch zeitliche Aspekte durch das Graphmodell beschrieben werden. Den einzelnen Teilaufgaben werden daher *Ausführungszeiten* bzw. *Ausführungszeitverteilungen* in Form kontinuierlicher Zufallsvariablen zugeordnet. Dies ermöglicht die Berechnung der Gesamtausführungszeit.

Sei $V = \{v_1, \ldots, v_n\}$ die Menge der den Teilaufgaben entsprechenden *Knoten* v_i, und sei $E \subseteq V \times V$ die Menge der *gerichteten Kanten*, die die Reihenfolgebeziehungen ausdrücken. Der *gerichtete Graph* (directed graph, digraph) G ist dann definiert als $G = (V, E)$. Ist der gerichtete Graph G *endlich* und *azyklisch*, so heißt er *Präzedenzgraph*. Ein Präzedenzgraph, bei welchem jedem Knoten v_i eine stochastisch verteilte Laufzeit in Form einer kontinuierlichen Zufallsvariablen X_i zugeordnet ist, heißt *Stochastisches Graphmodell*.

Wir wollen uns hier auf Präzedenzgraphen beschränken und zunächst den scheinbaren Nachteil akzeptieren, daß das in Programmen wichtige Konstrukt der Schleife im Präzedenzgraphen nicht vorkommt. Wir beschreiben in Abschnitt 2 dann eine Möglichkeit, mit diesem Problem umzugehen, indem zyklische Knoten eingeführt werden.

2 Der Modellwerkzeugkasten PEPP — Übersicht

In Abb. 1 ist die graphische Bedienoberfläche von PEPP [2] mit dem Hauptmenü und einem Ausschnitt aus einem Graphmodell dargestellt. Die Oberfläche umfaßt einen graphischen Modelleditor, dient zur Auswahl der Analyseverfahren (s. Abschnitt 3) und hat Schnittstellenfunktion zu anderen Tools für die Integration von Messung und Modellierung (s. Abschnitt 4).

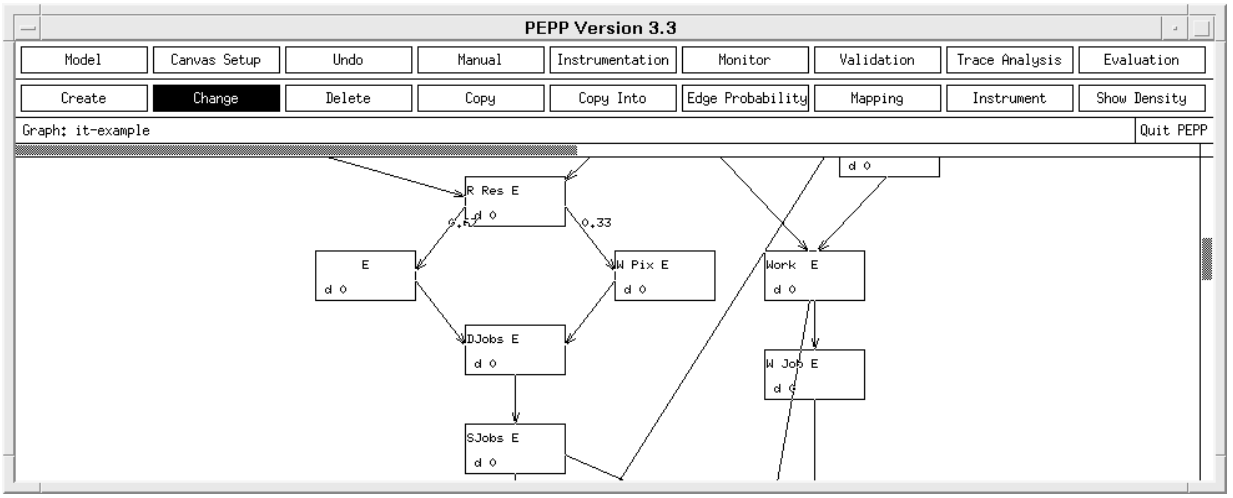


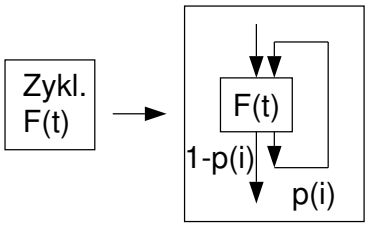
Abbildung 1: Das PEPP-Hauptmenü und Ausschnitt aus einem PEPP-Modell

2.1 Modellierungsmöglichkeiten

Durch die Bereitstellung der folgenden Knotentypen unterstützt PEPP die Modellierung typischer immer wiederkehrender Programmkonstrukte.

- **Elementarknoten**
  repräsentiert eine in sich nicht weiter strukturierte Teilaufgabe.
- **Parallelknoten**
  steht für n parallel ausführbare Elementarknoten, die alle die gleiche Laufzeitverteilung haben.
- **Zyklischer Knoten**
  Programme, die wesentliche Schleifen haben, lassen sich dennoch durch einen azyklischen Programmgraphen modellieren, wenn die Schleifen komplett durch einen zyklischen Knoten dargestellt werden. Dieser repräsentiert das mehrfache Durchlaufen einer Teilaufgabe.

Durch Angabe eines Schleifentyps wird die Wahrscheinlichkeit $p(i)$ für den Wiedereintritt in die Schleife nach dem i-ten Schleifendurchlauf bestimmt. Möglich ist z.B. konstante oder linear abnehmende Wahrscheinlichkeit für den Wiedereintritt.



Bei den Knotentypen Elementarknoten, Parallelknoten und zyklischer Knoten wird neben Name, Kurzbezeichnung und Knotentyp auch noch eine Laufzeitangabe gemacht, wobei der Typ der Laufzeitverteilung und die zugehörigen Bestimmungsgrößen der Verteilung angegeben werden.

d Deterministische Verteilung mit dem Parameter $e = \mathrm{E}[T]$ (Erwartungswert).

e Exponentielle Verteilung mit der Verteilungsfunktion $F(t) = 1 - e^{-\lambda t}$, $t \geq 0$, Parameter $\lambda = 1/\mathrm{E}[T]$.

a Approximative Verteilung mit den Parametern e (Erwartungswert) und v (Varianz), vgl. Abschnitt 3.2.

b Gemischte Erlang-Verteilung mit den Parametern $(p_1, k_1, \lambda_1, ..., p_n, k_n, \lambda_n)$.

- **Hierarchischer Knoten**
  Die in PEPP zulässigen Graphen sind hierarchische Graphen, d.h. jeder Knoten kann selbst wieder ein Graph sein. Das Konstrukt *Hierarchischer Knoten*, das einen Untergraphen repräsentiert, ermöglicht eine hierarchische Graphentwicklung und -darstellung. Die Untergraphen können wiederum hierarchische Knoten enthalten, so daß mehrstufige Hierarchien entstehen. Zur Auswertung wird ein hierarchischer Graph in einen nichthierarchischen ausgebreitet, vgl. Abb. 2. Um dabei beliebige Untergraphen problemlos in den expandierten Graphen einbetten zu können, werden sie um zwei zeitlose Synchronisationsknoten *Sync* ergänzt.

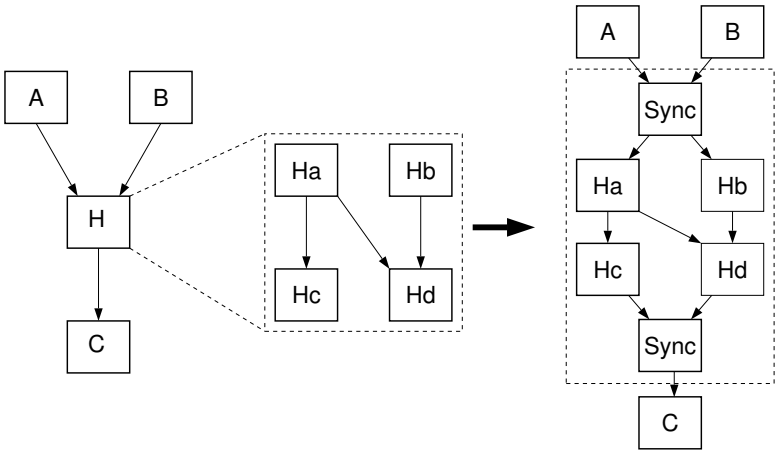


Abbildung 2: Expandieren eines hierarchischen Knotens

- **Loop-Knoten** (hierarchischer Schleifenknoten)
  entspricht einem mehrfach zu durchlaufenden Untergraphen, kombiniert also die Eigenschaften eines hierarchischen und eines zyklischen Knotens.

3 Auswertemethoden

Den derzeitigen Umfang von PEPP bezüglich der Auswertung von Programmgraphen zeigt die Abb. 3. Sie stellt drei Klassen von *Auswertewerkzeugen* dar.

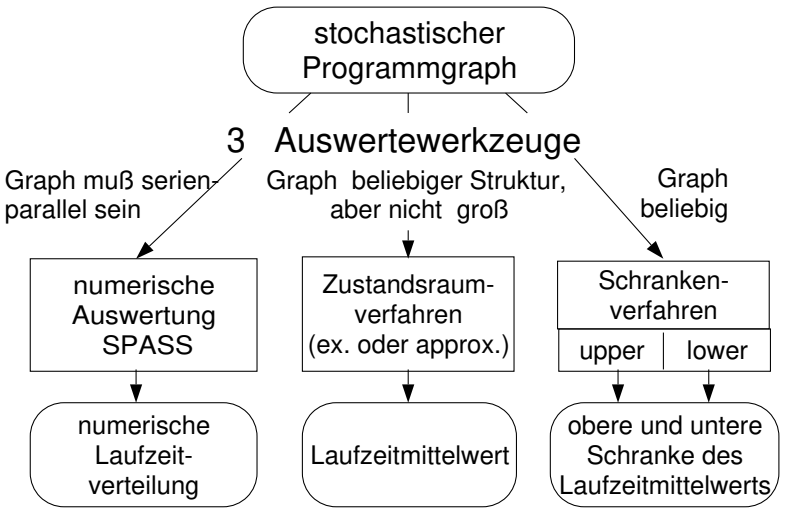


Abbildung 3: Die Auswertewerkzeuge des PEPP-Werkzeugkastens

3.1 Serienparallele Graphen: Auswertung mit SPASS

Das Werkzeug SPASS (*S*eries *PA*rallel *S*tructures *S*olver) ist geeignet für *serienparallel reduzierbare Graphstrukturen*. Es benutzt durchgehend eine *numerische Darstellung der Laufzeitverteilung*, während numerisch vorgegebene Laufzeitverteilungen bei anderen Auswerteverfahren meist durch ihre Momente approximiert werden. Damit ist der Weg frei, den Knoten realistische Bearbeitungszeiten in Form von beliebigen Laufzeitverteilungen zuzuordnen. Insbesondere ist es möglich, Ergebnisse von Messungen — wie immer sie aussehen — in Form einer empirischen Laufzeitverteilung als Knotenbewertung heranzuziehen, also die Rückkopplung von der Messung zum Modell konkret zu verwirklichen. Das Auswerteergebnis, die Gesamtlaufzeit eines mit SPASS analysierten Programmgraphen, ist nicht einfach eine Zahl, sondern eine Verteilung in numerischer Darstellungsform, deren Mittelwert den Erwartungswert der Gesamtlaufzeit darstellt.

Das Prinzip der serienparallelen Reduktion

Zwei Teilaufgaben v_i, v_j heißen *seriell verknüpft*, wenn die einzige Ausgangskante von v_i zugleich die einzige Eingangskante von v_j ist. Zwei oder mehrere Teilaufgaben heißen *parallel verknüpft*, wenn sie dieselben Vorgänger und Nachfolger haben. Ein stochastisches Graphmodell heißt *serienparallel reduzierbar*, wenn es mittels sukzessiver Ersetzung zweier seriell verknüpfter Knoten (bzw. zweier oder mehrerer parallel verknüpfter Knoten) durch einen Ersatzknoten schließlich auf *einen* Knoten reduziert werden kann. Alle Bearbeitungszeiten müssen dabei unabhängig voneinander sein. Für die Laufzeitverteilung des reduzierten Knotens gilt:

Serielle Reduktion: Bearbeitungszeitdichte für Summe (Faltung, Convolution)

$$f_S(t) = \mathrm{conv}(f_X, f_Y) = \int_0^t f_X(\tau) f_Y(t-\tau) d\tau \tag{1}$$

Parallele Reduktion: Bearbeitungszeitdichte für Maximum

$$f_M(t) = \max(f_X, f_Y) = f_X(t) \int_0^t f_Y(\tau) d\tau + f_Y(t) \int_0^t f_X(\tau) d\tau \tag{2}$$

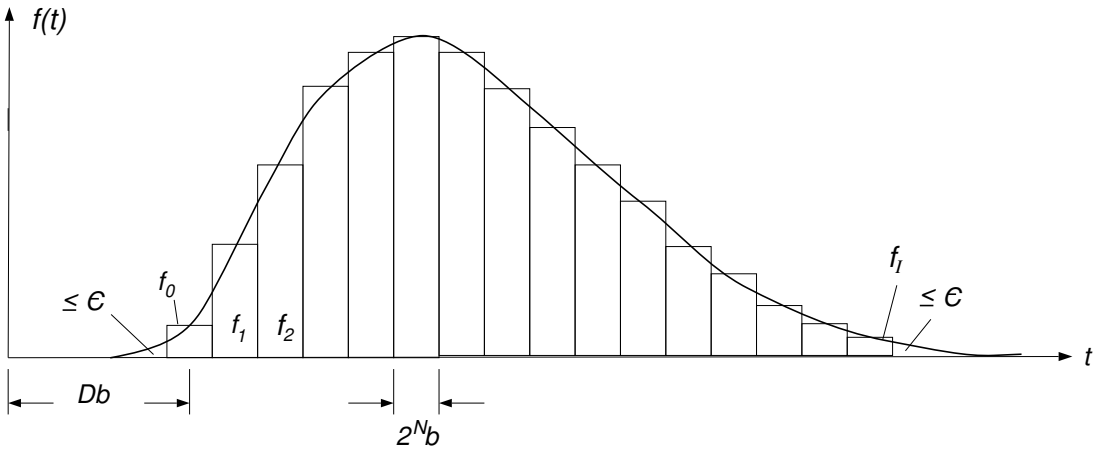


Abbildung 4: Kontinuierliche Dichtefunktion und zugehörige numerische Dichtefunktion

Numerische serienparallele Reduktion in SPASS

Die von SPASS verwendete numerische Darstellung der Dichtefunktion f_X einer Bearbeitungszeit X wird durch Abb. 4 illustriert.

Man erhält eine Treppenfunktion mit der Stufenbreite $\Delta t = 2^N b$ und mit:

b Zeiteinheit, sinnvollerweise die Zeitauflösung des verwendeten Meßgeräts.
N Ordnung, sie bestimmt die Breite der Stufen in Zweierpotenzen von b.
D Displacement, meist ungleich 0. D ist stets Vielfaches von 2^N.
I Zahl der Stützstellen.

Für die *diskrete numerische Faltung* in SPASS wird die kontinuierliche Formel (1) nach dem in Abb. 4 angedeuteten Prinzip diskretisiert. Man stellt zunächst Displacement D, Ordnung N und Stützstellenzahl I fest und ermittelt darauf aufbauend die diskreten Werte von $f_S(t)$. Die Zahl der Stützstellen I_S ergibt sich als Summe der Einzelstützstellen. I_S darf dabei nicht größer werden als ein vorgegebenes $I_{\max}$. Dies erreicht man gegebenenfalls durch Vergröberung der Ordnung N. Die Rechenzeit für die diskrete Faltung wächst quadratisch mit der Stützstellenzahl. Es empfiehlt sich deshalb nicht nur wegen der durch $I_{\max}$ vorgegebenen Begrenzung der Stützstellenzahl, die Ordnung N und damit die Stufenbreite nicht zu fein zu wählen. Für das *diskrete numerische Maximum* in SPASS wird die kontinuerliche Formel (2) diskretisiert. Bei der Maximumbildung wächst die Rechenzeit linear mit der Zahl der Stützstellen.

3.2 Nichtserienparallele Graphen: Approximative Auswertung mittels de-Verfahren

Die in der Realität vorkommenden Programmstrukturen sind leider oft nicht serienparallel reduzierbar. Man denke z.B. an einen Algorithmus für ein Multiprozessorsystem mit Nachbarschaftskommunikation. Hier hängt die i-te Teilaufgabe T_i^k auf Rechnerknoten R_k von den Vorgängerresultaten des Knotens R_k und denen seiner beiden Nachbarn $R_{(k-1)}$ und $R_{(k+1)}$ ab, was zu Programmgraphen mit der in Abb. 5 gezeigten Struktur führt. PEPP bietet die Möglichkeit, die mittlere Laufzeit nicht serienparallel reduzierbarer Graphen mit Hilfe eines Zustandsraumverfahrens [12] zu ermitteln.

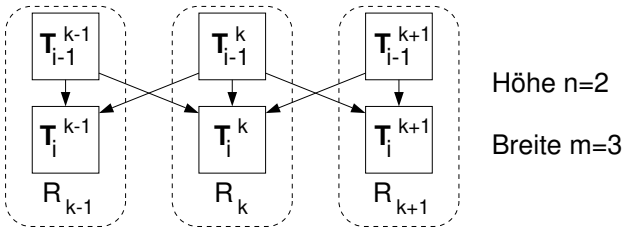


Abbildung 5: Programmgraph für Nachbarschaftskommunikation (Ausschnitt)

Die Methode der Zustandsreihenfolge

Ein *Zustand* z ist charakterisiert durch die Menge der Teilaufgaben $T\ (z)\ =\ \{T_1, \ldots, T_n\}$, die sich in Bearbeitung befinden. Wird eine Teilaufgabe fertig, so tritt ein Zustandsübergang ein und zwar $z \rightarrow z_i'$, wenn die Teilaufgabe T_i als erste fertig ist. Die Menge der im Zustand z_i' laufenden Teilaufgaben $T(z_i')$ ergibt sich aus der Menge $T(z) \setminus \{T_i\}$ vereinigt mit der Menge $S_z(T_i)$, die die als Nachfolger von T_i lauffähig gewordenen Teilaufgaben darstellt:

$$T\big(z_i'\big) = T(z) \setminus \{T_i\} \ \cup \ S_z(T_i)$$

Sei $\mathrm{E}[T_i(z)]$ die mittlere Bearbeitungszeit von T_i, falls Teilaufgabe T_i im Zustand z als erste fertig wird, und bezeichne s den Anfangszustand. Ferner sei $p(T_i(z))$ die Wahrscheinlichkeit dafür, daß die Teilaufgabe T_i im Zustand z als erste fertig wird. Die mittlere Gesamtbearbeitungszeit kann dann rekursiv folgendermaßen berechnet werden:

$$E[s] = \sum_{T_i \in T(s)} p(T_i(s))\Big(E[T_i(s)] + E\Big[s_i^{'}\Big]\Big)$$

Die Berechnung der Gesamtbearbeitungszeit ist i.a. kompliziert, da Restbearbeitungszeiten zu berücksichtigen sind. Eine Ausnahme sind Programmgraphen mit exponentiell verteilten Bearbeitungszeiten. Die "Gedächtnislosigkeit" dieser Verteilung hat den Vorteil, daß einer teilbearbeiteten Teilaufgabe bei ihrer Fortsetzung die ursprüngliche Laufzeitverteilung erneut zugewiesen werden kann.

Das de-Verfahren

Da rein exponentiell verteilte Knotenlaufzeiten das reale Verhalten der Teilaufgaben in der Regel schlecht modellieren, möchte man auch andere Verteilungsfunktionen zulassen. Um die Eigenschaft der Gedächtnislosigkeit nicht zu verletzen, könnte man zunächst daran denken, mit aus exponentiellen Phasen zusammengesetzten Verteilungen zu arbeiten. Bei Teilaufgaben, deren Laufzeiten eine geringe Varianz haben, erhält man allerdings Erlang-k-Verteilungen mit sehr hoher Phasenzahl k und damit einen Programmgraphen mit sehr hoher Knotenzahl, was zu einer "Zustandsraumexplosion" führt.

Der dem de-Verfahren [12] zugrundeliegende Gedanke wertet die häufig sehr kleine Varianz der Teillaufzeiten und die von Null verschiedenen Mindestbearbeitungszeiten als einen Vorteil und stellt die Mindestbearbeitungszeit durch eine deterministische und die Laufzeitschwankungen durch eine exponentielle Phase dar. Ein beliebiger Knoten mit dem Erwartungswert E und der Varianz V wird auf einen zweiphasigen Knoten reduziert, dessen deterministische Phase mit dem Parameter d und exponentielle Phase mit Parameter λ das ursprüngliche Laufzeitverhalten

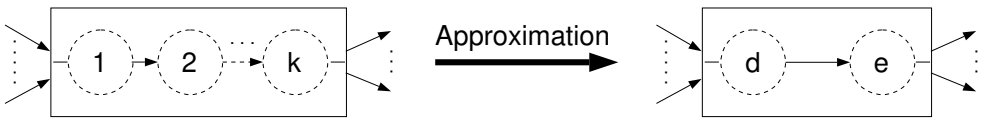


Abbildung 6: de-Approximation eines Knotens mit der Laufzeitverteilung Erlang-k

approximieren, vgl. Abb. 6. Wählt man $\lambda = \sqrt{1/V}$ und $d = E - \sqrt{V}$, so stimmen Erwartungswert und Varianz der ursprünglichen und der approximierten Verteilung überein.

Eine exakte Analyse ist mit dieser Methode i.a. wegen der Verletzung der Gedächtnislosigkeit nicht möglich. Enthält ein Zustand nämlich Teilaufgaben T_i mit deterministisch verteilten Laufzeiten, so hängt das künftige Verhalten von allen Vorgängerzuständen bis zu dem Zustand, in dem ausschließlich exponentiell verteilte Phasen aktiv waren, ab. Diese Abhängigkeiten des Systemverhaltens von seiner Vergangenheit sind zu komplex, als daß sie bei der Berechnung der mittleren Verweilzeit des Systems im Zustand z berücksichtigt werden könnten. Das mittlere Systemverhalten wird daher näherungsweise bestimmt. Die Näherung besteht darin, daß im Folgezustand z' die mittlere Verweilzeit in z' nur in Abhängigkeit von $T(z')$ bestimmt wird und dann die Parameter der Teilaufgaben $T_i \in T(z')$ berechnet werden.

Das de-Verfahren macht durch Verkleinerung des Zustandsraums weit komplexere Graphen analysierbar als das exakte Zustandsraumverfahren (siehe Tab. 1), aber es gibt auch für die de-Approximation ein Limit, ab welchem die Rechenzeiten zu hoch werden. Dies veranlaßte uns zur Entwicklung der nachfolgend dargestellten Schrankenverfahren.

3.3 Nichtserienparallele Graphen: Approximative Auswertung durch Schranken

Grundidee der Schrankenverfahren ist die Tatsache, daß das Hinzufügen von Kanten und Knoten die Gesamtbearbeitungszeit erhöht und das Weglassen von Kanten und Knoten diese verringert. Für PEPP wurden 4 veröffentlichte Schrankenverfahren genauer untersucht [6], eine verwandte theoretische Arbeit ist in [13] beschrieben. Wir favorisieren Schrankenmethoden, die die volle verfügbare Information, nämlich die Verteilung der Teilaufgabenlaufzeiten berücksichtigen. Das tun die drei Verfahren, die auf Ideen von Kleinöder [8], Shogan [11] und Dodin [5] zurückgehen. Sie modifizieren gegebene Graphen solange, bis sie serienparallel werden. Die gesuchte Laufzeitschranke ergibt sich als Mittelwert der Laufzeitverteilung des serienparallelen Graphen. Diese wird mit Hilfe der numerischen serienparallelen Reduktion wie beim SPASS-Verfahren berechnet. Die Verwendung numerischer Dichtefunktionen erlaubt auch in diesem Fall die Modellierung der Laufzeit mit beliebigen realitätsnahen Verteilungsfunktionen.

Heuristischer Algorithmus für die obere Schranke (Idee von Kleinöder)

Die beiden Schritte des folgenden heuristischen Algorithmus zum Hinzufügen von Kanten werden solange wiederholt, bis nur noch ein Knoten übrigbleibt.

1. Soweit als möglich serienparallele Reduktion des vorliegenden Graphen.
2. Suche nach einem Paar von Knoten, das sich parallel reduzieren läßt, wenn neue Kanten hinzugefügt werden, wobei folgende Bedingungen zu beachten sind:
   - Es dürfen keine Kanten zwischen Knoten auf derselben Ebene hinzugefügt werden.
   - Möglichst geringe Zahl neuer Kanten, die die nächste serienparallele Reduktion ermöglicht.
   - Ist diese Bedingung für mehr als ein Paar von Knoten erfüllt, so wählen wir jenes, für das der resultierende Graph minimal wird.

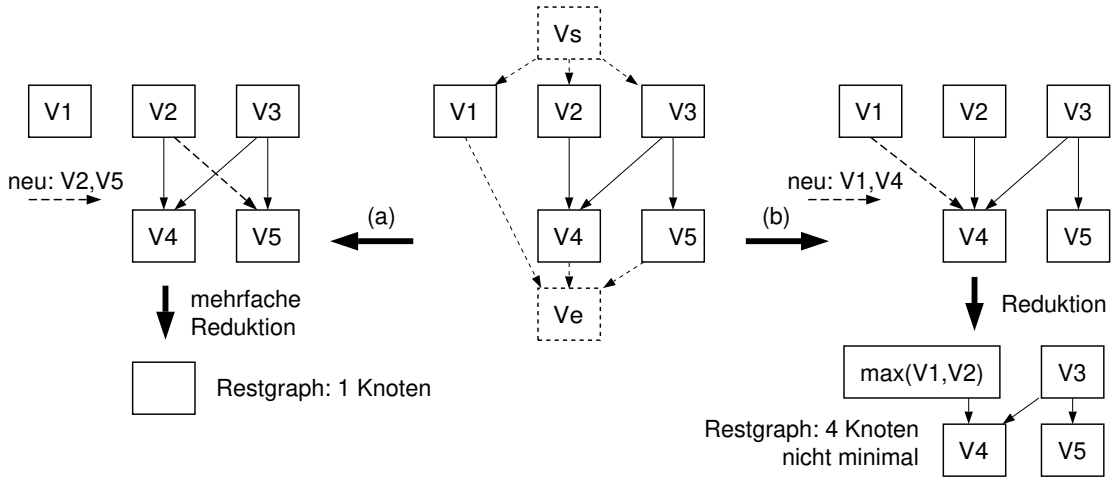


Abbildung 7: Ermittlung der oberen Schranke: Verschiedene Möglichkeiten, einen Graphen durch Hinzufügen von Kanten serienparallel zu reduzieren

Für einen kleinen Graphen ist dies im Beispiel der Abb. 7 veranschaulicht. Zur Vereinfachung der Darstellung sind dort (außer beim Originalgraphen) die Start- und Endeknoten v_s und v_e weggelassen. Man erkennt, daß in den Fällen (a) und (b) zwar jeweils genau eine neue Kante eingeführt wird, das Resultat jedoch sehr verschieden ausfällt. Die zusätzliche Kante (v_2, v_5) ermöglicht die vollständige Reduktion auf *einen* Knoten, während im Fall (b) mit (v_1, v_4) nur die Knoten v_1 und v_2 verschmolzen werden können. Entsprechend ist auch die Güte der Schranken: Unter der Annahme, daß alle Knoten exponentiell verteilte Laufzeiten mit Erwartungswert 1 haben erhält man im Fall (a) eine obere Schranke von 3.11 und im Fall (b) von 3.33.

Weitere Schrankenverfahren

Grundsätzlich ist klar, daß das Weglassen von Kanten untere Schranken liefert, weil Vorbedingungen, die den Start der betrachteten Teilaufgabe verzögern könnten, wegfallen. PEPP enthält einen entsprechenden heuristischen Algorithmus für die untere Schranke. Auch die Verfahren von Shogan und Dodin sind in PEPP implementiert. Sie gewinnen Schranken durch das Hinzufügen/Weglassen von *Knoten*. Das Verfahren von Dodin liefert nur eine obere Schranke, die jedoch immer besser ist als die obere Schranke von Shogan.

3.4 Vergleich der Auswerteverfahren

3.4.1 Die Genauigkeit der de-Approximation

Um die Genauigkeit der de-Approximation exemplarisch vorzustellen, wollen wir uns wieder auf den Graphen aus Abb. 5 beziehen (Nachbarschaftssynchronisation). Solche reguläre Graphen können durch Höhe n und Breite m beschrieben werden. In Tab. 1 werden die approximierten Ergebnisse mit den mit Hilfe des exakten Zustandsraumverfahrens gewonnenen Werten für ein 4×5 Nachbarschaftsmodell verglichen. Als Last nehmen wir identisch verteilte Laufzeiten für alle Tasks an, und zwar Erlang-k-Laufzeitverteilungen $E_k(\lambda)$. Um die Ergebnisse vergleichbar zu machen, wurden λ und k so gewählt, daß das erste Moment der Knotenlaufzeit konstant ist.

Zur Ermittlung der Genauigkeit der de-Approximation wurden außerdem mit Hilfe des Werkzeugs GreatSPN [1] viele Simulationen durchgeführt und mit der de-Approximation verglichen. Die Unterschiede zwischen approximierten und simulierten Modellaufzeiten lagen in allen untersuchten Beispielen zwischen 0 und 5 Prozent.

Knotenlaufzeit (Last)	exakt		de-Approximation
	Zust.	Wert	
$E_1(0.2)$	260	40.80	40.80
$E_2(0.4)$	6052	34.72	34.70
$E_5(1)$	zu aufwendig		29.24
$E_{10}(2)$			26.47

Tabelle 1: Vergleich der exakten und der approximierten Ergebnisse für ein 4×5 Nachbarschaftsmodell

3.4.2 Die Genauigkeit der Schrankenverfahren

Bei der Betrachtung der Nachbarschaftsstrukturen stellt sich heraus, daß die Güte der berechneten Schranken sowohl von der gegebenen Graphstruktur als auch von den Laufzeitverteilungen der Knoten abhängt. Wir betrachten zunächst den Strukturaspekt anhand von $(8 \times m)$-Nachbarstrukturen mit $m \in \{4, 6, 8, 10, 12\}$. Alle Task-Laufzeiten seien identisch $\mathrm{E}_{10}(2)$ verteilt. Die Ergebnisse dieser Betrachtung zeigt Abb. 8. Die "extensions" .u und .l stehen für die obere (upper) und untere (lower) Schranke.

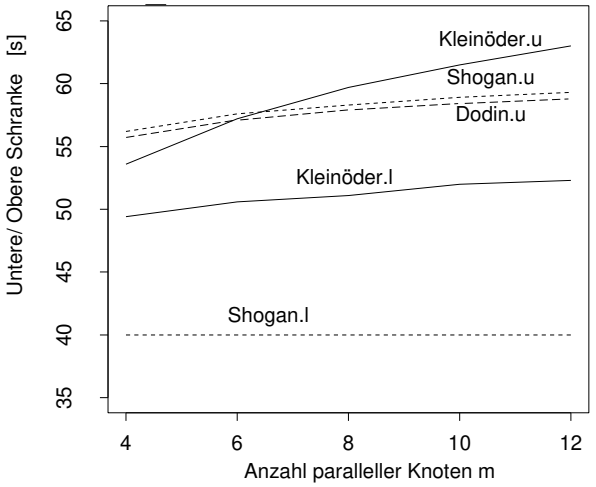


Abbildung 8: Schranken bei variierendem Parallelitätsgrad m

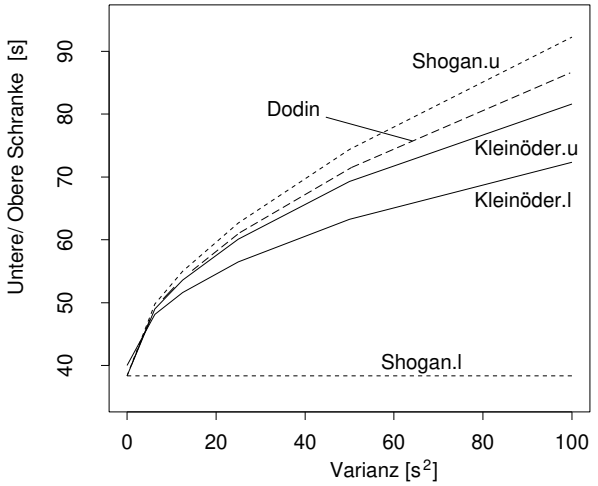


Abbildung 9: Schranken bei variierenden Task-Laufzeiten

Sobald die in einem parallelisierten Ablauf parallel laufenden Tasks unterschiedliche Laufzeiten haben, erhöht sich der Einfluß wechselseitiger Wartezeiten. Wir erwähnten früher, daß die Varianz der Laufzeitverteilungen eine wichtige Rolle spielt und beobachten, daß niedrige Varianz wie erwartet zu engen Schranken führt, vgl. Abb. 9. Diese Betrachtung fußt auf einem (4×4)-Nachbarschaftsmodell mit mittleren Tasklaufzeiten von 10 s.

3.4.3 Vergleich des Berechnungsaufwands

Tab. 2 zeigt in den mit T_c bezeichneten Spalten den Berechnungsaufwand für die de-Approximation und die verschiedenen Schrankenverfahren exemplarisch für den $(8 \times m)$-Programmgraphen. Exakte Ergebnisse kann man für ein Modell dieser Größe wegen explodierender Rechenzeit überhaupt nicht mehr berechnen. Auch bei der de-Approximation zeigt sich ab $m = 6$ die Wirkung der Zustandsraumexplosion. Entscheidend aber ist die Leistungsfähigkeit der Schrankenverfahren bei größeren Modellen: Der Berechnungsaufwand ist klein und wächst nur polynomial mit m.

m	de-Approximation			Schrankenverfahren									
				Kleinöder				Shogan				Dodin	
				untere		obere		untere		obere		obere	
	Wert	Zust.	T_c [s]	Wert	T_c [s]	Wert	T_c [s]	Wert	T_c [s]	Wert	T_c [s]	Wert	T_c [s]
4	51.1	4252	15	**49.4**	2	**53.6**	1	40.0	1	56.2	1	54.8	1
5	52.4	62329	1430	**49.2**	3	**55.6**	1	40.0	1	57.0	2	55.8	1
6	zu aufwendig			**50.6**	3	57.2	2	40.0	1	57.6	2	**56.3**	2
10				**52.0**	9	61.5	10	40.0	2	58.9	3	**57.8**	3
12				**52.3**	14	63.0	18	40.0	2	59.3	4	**58.2**	3

Tabelle 2: Vergleich des Berechnungsaufwands T_c und der ermittelten Werte.
Die besten unteren und oberen Schranken sind durch Fettdruck hervorgehoben.

4 Integration von Messung und Modellierung: Modellgesteuertes Monitoring mit PEPP

Messung und Modellierung sollen als integrales Ganzes der Leistungsbewertung und dem Debugging zur Verfügung stehen. Die folgenden, im Umfang von PEPP enthaltenen Werkzeuge dienen dazu, die in Modellen verfügbaren Abstraktionsüberlegungen der Messung nutzbar zu machen. Sie unterstützen die Vorbereitung der Messung, ihre Steuerung, die Prüfung der Konsistenz zwischen Meßspur und Modell und die Spurauswertung.

Modellgesteuerte Instrumentierung mit AICOS
Eine Messung beginnt mit der Auswahl der zu instrumentierenden Stellen im Programm. Dies geschieht über die Erstellung eines PEPP-Graphmodells und dessen Verwendung als Monitoringmodell. PEPP erzeugt eine Kommandodatei, aus der das Instrumentierungswerkzeug AICOS (*A*utomatic *I*nstrumentation of *C* pr*o*gram*s*) die für die Instrumentierung benötigte Information liest [2]. Die Instrumentierung wird in den folgenden Schritten durchgeführt:

1. Ausführen aller C-Präprozessor-Anweisungen (mit cpp).
2. Syntaktische Analyse des C-Programms.
3. Erzeugung von Ableitungsbaum und Symboltabelle.
4. Einfügen der Monitoring-Anweisungen in den Ableitungsbaum.
5. Transformation des modifizierten Ableitungsbaums in ein instrumentiertes C-Programm.

Software für den Hardwaremonitor ZM4
Nach der Instrumentierung eines Programms kann die Messung gestartet werden, wozu Kenntnis über den Hardwaremonitor ZM4 [3] notwendig ist. PEPP unterstützt die Messung, indem Konfigurationsdateien mit Monitoring-Anweisungen angeboten werden und indem automatisch Meßablaufsteuerungsprogramme für den Hardwaremonitor ZM4 generiert werden. Der Sinn der PEPP-Konfigurationsbeschreibung liegt darin, das Einstellen des Monitors ZM4 und die Abwicklung der Messung zu erleichtern und Fehlbedienungen zu vermeiden.

Validierung
PEPP unterstützt die automatische modellgesteuerte Validierung von Ereignisspuren zur Beantwortung der Frage: Sind die Spuren konsistent mit dem Modell? Es wird überprüft, ob die in einer gemessenen Ereignisspur vorliegende Ereignisreihenfolge eine mögliche Ereignisreihenfolge in bezug auf das Graphmodell ist [4]. Da das Programm nicht automatisch aus dem Monitoringmodell generiert wird, ist eine Konsistenzprüfung nötig. Bei der hier gewählten Betrachtungsweise wird das Modell als korrekt vorausgesetzt. Somit bedeutet diese Art der

Validierung in der Regel eine Suche nach Programmierfehlern. Stellt PEPP eine Inkonsistenz zwischen Modell und Ereignisspur fest, so kann für Debuggingzwecke abgefragt werden, in welcher Programmzeile der Fehler auftrat, welche Ereignisnummer und welcher Ereigniszeitpunkt betroffen sind und wie häufig das inkriminierende Ereignis auftrat.

Auswertung mit dem Ereignisspuranalysesystem SIMPLE
Für die Ereignisspuranalyse mit SIMPLE [3] wird semantisches Wissen aus dem Monitoringmodell benutzt. Eine zentrale Grundidee von SIMPLE ist es, mittels einer Spurbeschreibung in der Sprache TDL beliebige Ereignisspuren den Auswertewerkzeugen zugänglich zu machen. Da PEPP eine systematische und automatische Instrumentierung ermöglicht, liegt es nahe, zugleich auch die spätere Auswertung zu unterstützen. Dazu erzeugt man automatisch eine TDL-Spurbeschreibungsdatei, die daraus abgeleitete Schlüsseldatei und die Werkzeugaufrufe. Es existieren derzeit folgende Möglichkeiten, die Spurauswertung modellgesteuert mit PEPP zu unterstützen:

- Statistische Auswertung mit dem Werkzeug TRCSTAT.
- Ablauforientierte Auswertung mit Gantt-Diagrammen (Werkzeug GANTT).
- Animation dynamischer Programmabläufe (Werkzeug VISIMON).

5 Schlußbemerkung

Wir haben eine Übersicht über die Funktionalität des Werkzeugs PEPP gegeben. Zusammenfassend seien noch einmal die wichtigsten Merkmale von PEPP erwähnt: PEPP beinhaltet vielfältige Analyseverfahren für stochastische Graphmodelle, vom exakten Zustandsraumverfahren über die de-Approximation bis zu den äußerst effizienten Schrankenverfahren. Darüberhinaus unterstützt PEPP die Methode der modellgesteuerten Messung und Spurauswertung.

Literatur

[1] G. Chiola. GreatSPN 1.5 Software Architecture. In G. Balbo and G. Serazzi, editors, *Proceedings of the 5th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Torino, Feburary 1991*, pages 117–132. Elsevier Science Publisher B.V., 1992.

[2] P. Dauphin, F. Hartleb, M. Kienow, V. Mertsiotakis, and A. Quick. PEPP: Performance Evaluation of Parallel Programs — User's Guide – Version 3.3. Technical Report 17/93, Universität Erlangen–Nürnberg, IMMD VII, September 1993.

[3] P. Dauphin, R. Hofmann, R. Klar, B. Mohr, A. Quick, M. Siegle, and F. Sötz. ZM4/SIMPLE: a General Approach to Performance-Measurement and -Evaluation of Distributed Systems. In T.L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*, chapter 6, pages 286–309. IEEE Computer Society Press, Los Alamitos, California, Jan 1994.

[4] P. Dauphin, M. Kienow, and A. Quick. Model-driven Validation of Parallel Programs Based on Event Traces. In Topham, Ibbett, and Bemmerl, editors, *Proceedings of the "Working Conference on Programming Environments for Parallel Computing", Edinburgh 6–8 April*, pages 107–125. North-Holland, Amsterdam, 1992.

[5] B. Dodin. Bounding the Project Completion Time Distributions in PERT Networks. *Operations Research*, 33(4):862–881, 1985.

[6] F. Hartleb and V. Mertsiotakis. Bounds for the Mean Runtime of Parallel Programs. In R. Pooley and J. Hillston, editors, *Proceedings of the Sixth International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 197–210, Edinburgh, 1992.

[7] U. Herzog. Leistungsbewertung und Modellbildung für Parallelrechner. *Informationstechnik (it)*, 31(1):31–38, 1989.

[8] W. Kleinöder. *Stochastische Bewertung von Aufgabenstrukturen für hierarchische Mehrrechnersysteme*. Dissertation, Universität Erlangen–Nürnberg, IMMD VII, Arbeitsbericht Bd. 15, Nr. 10, August 1982.

[9] D.F. Martin and G. Estrin. Models of Computations and Systems — Evaluation of Vertex Probabilities in Graph Models of Computations. *Journal of the ACM*, 14(2):281–299, April 1967.

[10] R. Sahner and K. Trivedi. Performance Analysis and Reliability Analysis Using Directed Acyclic Graphs. *IEEE Transactions on Software Engineering*, SE-13(10), October 1987.

[11] A.W. Shogan. Bounding Distributions for a Stochastic PERT Network. *Networks*, 7:359–381, 1977.

[12] F. Sötz. A Method for Performance Prediction of Parallel Programs. In H. Burkhart, editor, *CONPAR 90–VAPP IV, Joint International Conference on Vector and Parallel Processing. Proceedings*, pages 98–107, Zürich, Switzerland, September 1990. Springer–Verlag, Berlin, LNCS 457.

[13] N. Yazici-Pekergin and J.-M. Vincent. Stochastic Bounds on Execution Times of Parallel Programs. *IEEE Transactions on Software Engineering*, 17(10):1005–1012, October 1991.