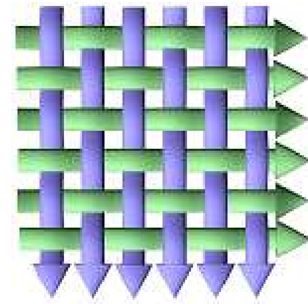
Sonderforschungsbereich 559
Modellierung großer
Netze in der Logistik



Technical Report 03012

ISSN 1612-1376

Numerische Verfahren zur quantitativen Analyse von *ProC/B* und Petri Netz Modellen

Teilprojekt M2:
Markus Fischer
Informatik 4,Universität Dortmund

Dortmund, 03.05.2003

Inhaltsverzeichnis

1 Numerische Markov-Ketten-Analyse für Logistische Systeme **3**

2 Modellierung logistischer Systeme **9**

2.1 Prozesse mit ausfallbehafteten Betriebsmitteln 9

2.2 Komplexe Bedienstation 13

3 Test numerischer Analyseverfahren **15**

3.1 Ergebnisse Fallbeispiel 1 16

3.2 Ergebnisse Fallbeispiel 2 22

4 Fazit **28**

Kapitel 1

Numerische Markov-Ketten-Analyse für Logistische Systeme

Eine quantitative und qualitative modellbasierte Analyse technischer Systeme (Computer, Produktion, Logistik) ist für die Planung neuer Systeme und die Reorganisation existierender Systeme bedeutsam. Durch sie können verschiedene Planungsszenarien am Modell und nicht durch Experimentserien am realen System bewertet werden. Dies ist vorteilhaft, weil dann riskante und teure Eingriffe in reale Systeme oder zeitaufwendige prototypische Implementierungen neuer Systeme obsolet sind.
Für die Reproduktion eines Systems im Modell (Modellbildung) müssen zunächst grundlegende Systemeigenschaften identifiziert werden. Viele technische Systeme sind als **Diskrete Ereignisgesteuerte Dynamische Systeme** (*DEDS*) interpretierbar, deren Dynamik durch diskrete Systemzustände und diskrete und asynchron auftretende Ereignisse, die Zustandswechsel auslösen, gekennzeichnet ist. Die *DEDS* **Modellbildung** kann mit hochsprachlichen, an Anwendungen adaptierten Notationen (Petri-Netze, Warteschlangennetze) oder analytischen Analysemethoden leicht zugänglichen Notationen (Automaten, stochastische Prozesse) realisiert werden. Die übliche Methodik hierbei ist, die manuelle Modellbildung durch den Modellierer mit einer hochsprachlichen Notation zu unterstützen, die in ihrer Semantik hinreichend formal und eindeutig ist, um aus ihr automatisiert ein analytisches Modell zu erzeugen.
Sofern die Modellbildung kein Selbstzweck ist steht sie im Zielkonflikt zwischen Behandelbarkeit bzw. Lösung des analytischen Modells einerseits und Ausdrucksmächtigkeit des hochsprachlichen Modells andererseits. Ein Kompromiss (neben anderen) ist, bestimmte - sogenannte Markovsche - Annahmen bezügliches des zeitlichen Verhaltens im *DEDS* zu machen, die dazu führen, dass das analytische Modell eine **Markov-Kette** ist. Viele technische Systeme erfüllen die Markovsche Annahme. Der Vorteil einer Markov-Kette liegt in einer sie auszeichnenden Eigenschaft - der sogenannten Gedächtnislosigkeit - wodurch Markov-Ketten prinzipiell analytisch

handhabbar sind. Die rechnergestützte Lösung **großer Markov-Ketten**, wie sie gewöhnlich aus realen technischen Systemen resultieren, ist jedoch sehr platz- und rechenintensiv, wodurch die Verwendung von Markov-Ketten limitiert wird.
Eine Markov-Kette ist vollständig durch die sogenannte **Generatormatrix** spezifiziert, deren Dimension mit der Anzahl der Systemzustände übereinstimmt und deren Einträge Raten exponentiell-verteilter (=Markovsche Annahme) Zustandstransitionen sind. Die Markov-Kette bzw. die Generatormatrix wird aus einer hochsprachlichen Notation durch Exploration des Zustandsraums via Erreichbarkeitsanalyse generiert. Das Ergebnis ist eine explizite Enumeration erreichbarer Zustände im Erreichbarkeitsgraphen, der durch zeitliche Bewertung (Raten) möglicher Zustandsübergänge in einer isomorphen Generatormatrix resultiert. Bedingt durch den enormen Platzbedarf für die Speicherung der Generatormatrix haben sich implizite bzw. generische Darstellungen etabliert, die Struktureigenschaften und Ergebnisse gewisser Voranalysen des hochsprachlichen Modells verwerten.
Die **stationäre Verteilung** der Markov-Kette ist ein Vektor, dessen Einträge Aufenthaltswahrscheinlichkeiten für Zustände repräsentieren und aus denen für das hochsprachliche Modell technische Kennzahlen wie Durchsatz und Auslastung ableitbar sind. Die stationäre Verteilung ist als (normierte) Lösung eines homogenen, linearen Gleichungssystems (LGS) charakterisiert, dessen singuläre Koeffizientenmatrix genau die Generatormatrix ist. Insofern erfordert die stationäre Markov-Ketten Analyse das Lösen eines LGS, ein Berechnungsproblem, für das auf einen prinzipiell breiten Fundus numerischer Verfahren zurückgegriffen werden kann. Allerdings erzwingt das Größenproblem und der damit einhergehende hohe Platz- und Rechenbedarf Weiterentwicklungen sowohl im Bereich mathematischer Modelle numerischer Verfahren als auch bei deren algorithmischer Umsetzung.
Eine wichtige Klasse **mathematischer Modelle** numerischer Verfahren sind **iterative Verfahren** (Jacobi, Gauss-Seidel etc), in denen sukzessive ein Iterationsvektor (=Approximation stationäre Verteilung) aktualisiert wird. Mathematische Modelle iterativer Verfahren resultieren aus einem Splitting der Koeffizientenmatrix, das das ursprüngliche LGS nach algebraischen Umformungen in eine Fixpunktgleichung mit expliziter Angabe eines Iterations-Operator (=Abbildungsvorschrift für Iterationsschritt) transformiert. Weiterentwicklungen und Verallgemeinerungen klassischer Iterationen betreffen u.a. blockorientierte Iterationen, hierarchische Iterationen und asynchrone Iterationen. **Blockorientierte Iterationen** adaptieren an einer Blockstruktur der Koeffizientenmatrix und aktualisieren Teilvektoren und nicht nur einzelne Einträge des Iterationsvektors in einem Schritt. **Hierarchische** bzw. **Multi-Level Iterationen** basieren auf einer Blockstrukturierung und betten in jeden Schritt einer äußeren Iteration (z.B. auf Block-Ebene agierend) eine innere Iteration (z.B. innerhalb von Blöcken) ein. Sie sind u.a. bedeutsam, wenn eine direkte Lösung der Blöcke zu teuer ist. **Iterative Aggregierungs- Disaggregierungsverfahren** (IAD) separieren die Behandlung unterschiedlicher Kopplungsgrade in NCD-Generatormatrizen, indem sie abwechselnd alle Blocksysteme starker Zusammenhangskomponenten lösen (z.B. im Rahmen einer Block-Jacobi Iteration) und

anschließend mit einem reduzierten bzw. aggregierten System die schwache inter-Block-Kopplung bearbeiten. In **asynchronen Iterationen** ist die Iteration von Teilvektoren in gewisser Hinsicht entkoppelt, obwohl Datenabhänigkeiten vorliegen. Für mathematische Modelle iterativer Verfahren, die eine oder mehrere der oben aufgezählten Eigenschaften vereinen, sind **Kovergenzresultate** bekannt. Das Vorgehen hierbei ist zu prüfen, inwieweit Struktureigenschaften der Koeffizientenmatrix in Abhängigkeit von der Auswahl des Koeffizientenmatrix-Splittings auf den Iterations-Operator übertragbar und dort für die Bewertung der Konvergenz(geschwindigkeit) nützlich sind. In diesem Zusammenhang sind zwei Zielkonflikte bedeutsam, erstens die Ausdrucksmächtigkeit von Splittings, die notwendig ist, um bestimmte Eigenschaften von Iterationen (hierarchisch, asynchron, eingebettete AD-Schritte) zu erfassen versus der analytischen Handhabbarkeit generischer Klassen von Iterations-Operatoren und zweitens schwache Annahmen bezüglich der Koeffizientenmatrix versus ableitbare Aussagen zum Iterations-Operator (Konvergenz). Bedingt durch diese Zielkonflikte sind Konvergenzaussagen für kombiniert asynchrone und hierarchische Iterationen (= komplexes Splitting bzw. Iterations-Operator) angewendet auf singuläre *LGS* (schwache Annahme bzgl. Koeffizientenmatrix) schwer ableitbar.
Weiterentwicklungen **algorithmischer Modelle** streben eine effiziente (Platz und Zeit) Ausführbarkeit eines mathematischen Modells im Rechner an. Ein Schwerpunkt ist die Implementierung problemspezifischer Datenstrukturen zur Speicherung der Generatormatrix und/oder des Iterationsvektors und die Bereitstellung effizienter Basisoperationen für diese. Die Datenstrukturen adaptieren am Berechnungsproblem, um Strukturen und Redundanzen zu erfassen und Daten strukturiert und kompositionell bzw. generisch (Kronecker-Algebra, diverse Varianten von Entscheidungsbäumen) zu speichern.
Der inhärent hohe Rechen- und Platzaufwand der numerischen Lösung von Markov-Ketten macht die Nutzung leistungsfähiger, paralleler **Rechnerarchitekturen** attraktiv. Wissenschaftliches Rechnen fokussiert gegenwärtig weniger auf monolithische Mehrprozessor-Rechner, sondern zunehmend auf preiswerte, flexibel vernetzbare (Local/Wide Area Networks), vollständig ausgestattete, heterogene Einzelrechner.
In dieser Arbeit wird eine Implementierung numerischer Verfahren (*ASYNC*) experimentell getestet, die bestimmte Aspekte mathematischer Modelle, algorithmischer Modelle und paralleler Rechnerarchitekturen kombiniert, die einzeln betrachtet bekannt sind, deren Zusammenspiel in einer kombinierten Realisation aber unbekannt ist. *ASYNC* zielt auf eine verbesserte Beherrschbarkeit großer Markov-Ketten ab. Hierfür soll der Nachweis erbracht werden, dass *ASYNC* die Performance (=Lösungszeit) bekannter numerischer Verfahren verbessert und dabei eine angemessene Beschleunigung erzielt. Desweiteren soll das *ASYNC* Instrumentarium für die quantitative Analyse logistischer Systeme verfügbar gemacht und in seiner Eignung für dieses Anwendungsfeld untersucht werden. Die essentiellen Eigenschaften von *ASYNC* sind:

1. **Mathematisches Modell**: *ASYNC* basiert auf einer Block-Jacobi (*BJAC*)-

Iteration, deren Schritte auf Blockebene durch eingebettete, auf intra-Block Ebene agierende Iterationen hierarchisch verfeinert sind. Die *BJAC*-Iteration schreitet asynchron voran und erlaubt Aggregierungsschritte.

2. **Algorithmisches Modell**: *ASYNC* nutzt als Basis-Datenstruktur eine hierarchische Kronecker-Darstellung der Generatormatrix (Buchholz), fokussiert auf die verteile Realisation von *BJAC* Iterationsschritten und verwendet Kommunikationsroutinen mit asynchroner Sende- und Empfangssemantik.

3. **Rechnerarchitektur**: *ASYNC* zielt auf ein lokales Netzwerk von Rechnern mit verteiltem Speicher ab, das auf Ethernet-Verbindungen basiert und in einer Mehrbenutzer-Umgebung betrieben wird.

Die Zusammenführung und Integration dieser Aspekte ist neu und durch zahlreiche Faktoren begründet:

1. Die Kronecker-Darstellung der Generatormatrix profitiert von einer parallelen Rechnerarchitektur, weil leicht verteuerte Rechenoperationen auf der Generatormatrix (generische Darstellung) bei einer verteilten Ausführung zusätzliche Rechenressourcen erschließen. Zudem wird in einem Rechnernetzwerk ein großer, wenn auch verteilter Arbeitsspeicher verfügbar gemacht. Eine Verteilung des Iterationsvektors ist attraktiv, weil der Platz-Flaschenhals in Form des Iterationsvektors behoben wird.

2. Die Kronecker-Darstellung liefert eine Blockstruktur der Generatormatrix, an der block-orientierte Verfahren adaptieren können. Eine *BJAC*-Iteration liefert eine Dekomposition des Berechnungsproblems in Teilproblme, die auf der gegebenen Blockstruktur skalierbar ist und eine parallele Ausführung ermöglicht. Zudem reflektiert die Blockstruktur etwaige NCD-Charakteristika im System, so dass Teilprobleme numerisch gering gekoppelt sind, ein Umstand, von dem parallele Ausführungen profitieren und der IAD Methoden rechtfertigt.

3. Eine verteilte Ausführung der *BJAC*-Iteration profitiert von der Kronecker-Darstellung, weil in jedem verteilten Arbeitsspeicher die vollständige Generatormatrix als Duplikat gehalten werden kann, was Datenabhängigkeiten verringert und gleichzeitig die Kommunikation flexibilisiert, sowie Lastumverteilungen erleichtert.

4. Verteilte, asynchrone Iterationen sind unabhängig von der Verfügbarkeit "aktueller" kommunizierter Daten und deswegen geeignet für Kommunikationsmedien mit hohen und lastabhängig-schwankenden Kommunikationszeiten. Das Kommunikationsmedium profitiert von verteilten, asynchronen Iterationen, die zu einer gleichmäßigeren Belastung der Kommunikationsressourcen führen, weil Rechenphasen (geringe Kommunikation) und Kommunikationsphasen überlappend ablaufen.

5. Asynchrone und hierarchische Iterationen sind flexibel ausführbar und können ggf. an Performance-Charakteristika des Kommunikationsmediums (adaptiv) angepasst werden, um Stabilität und Robustheit zu erzielen. Dies beinhaltet die Steuerung von Kommunikationsraten und selektive, lokale Iterationen, die an der Verfügbarkeit kommunizierter Daten adaptieren.

Die Besonderheit von *ASYNC* ist die **Flexibilität** der **Iteration** und **Kommunikation**. Rechenprozesse führen asynchron und selektiv Iterationen zur approximativen Lösung "ihrer" - aus der *BJAC*-Iteration resultierenden - Teilprobleme aus und kommunizieren asynchron und selektiv Lösungen "ihrer" Teilprobleme. Es ist eine offene Frage, ob und wenn ja wie diese Flexibilität zur Optimierung der Performance (=Lösungszeit der Markov-Kette) ausnutzbar ist. Mittel zum Zweck soll eine Selbst-Anpassung der Ausführung des algorithmischen Modells sein, die zum Zeitpunkt der Übersetzung des Quellcodes und dynamisch während der Laufzeit realisiert wird. Die Selbst-Anpassung zielt auf eine effektive und moderate Inanspruchnahme der Rechen- und Kommunikationsressourcen und eine gute Konvergenzrate der Iteration ab.

Das Konzept von "**Self-adapting numerical software**" ist verbreitet, insbesondere auch für iterative Verfahren. Als Beispiele seien die Optimierung des mathematischen Modells (dynamische Parameter-Schätzer für Relaxationparameter; ..) des algorithmischen Modells (intelligente Auswahl zwischen algorithmischen Varianten für Kernoperationen, wie z.B. Vektor-Matrix Multiplikation; Aggregation in Datenstrukturen; ..) oder der Inanspruchnahme von Ressourcen (Lastumverteilung, intelligente Auswahl zwischen Varianten kollektiver Kommunikationsroutinen; ..) genannt. Die Variation eines Relaxationsparameters hat keinen Einfluß auf die Performance des algorithmischen Modells, in *ASYNC* ist die Optimierung bedingt durch Zielkonflikten multikritiell. Beispielsweise sind Restriktionen bzgl. der Asynchronität von Iterationen notwendig für eine verbesserte Konvergenzrate, allerdings erfordert dies bedingte Synchronisationspunkte im algorithmischen Modell, die potentiell die Performance verschlechtern. Eine verbesserte Approximationsgüte (Anzahl innerer Iterationen) beschleunigt die Konvergenz hierarchischer Iterationen, verteuert aber die Ausführungszeit äußerer Iterationsschritte.

Die **Bewertung der Performance** von *ASYNC* (mit und ohne Selbst-Anpassung) kann experimentell (Messung am realen "System *ASYNC*´) oder mit relativ starken Einschränkungen auch modellbasiert mit formalen Methoden durchgeführt werden. Dieser Bericht ist auf die experimentelle Messung der Performance beschränkt.

Die hier zu analysierenden Markov-Ketten beschreiben **logistische Systeme**. Die Verfügbarmachung von *ASYNC* für die quantitative Analyse logistischer Systeme ist durch die Vorgehensweise des SFBs 559 determiniert, vgl. Flussdiagramm in Abb. 1.1. Logistische Systeme werden als *DEDS* (s.o.) interpretiert, weil variable Systemgrößen (Lagerbestand, Auftragsvolumen) diskret und Werttransitionen der Variablen durch diskrete Ereignisse (Einlagerung, Auftragseingang) ausgelöst werden, deren Auftreten zudem mit Unsicherheiten (stochastisch schwankende Nachfra-

Abbildung 1.1: Vorgehensmodell zur Analyse logististischer Systeme

ge) verbunden ist. Der Ausgangspunkt ist ein logistisches System, das Mithilfe eines graphisch-basierten Editors in der *ProC/B* Notation modelliert wird. Die ***ProC/B* Notation** unterstützt eine objekt-orientierte, prozess-orientierte und hierarchische Sichtweise bei der Modellbildung und hat sowohl Konzepte des Prozessketten-Paradigmas nach Kuhn adoptiert, wodurch Anforderungen und Bedarfe des Anwendungsbereiches berücksichtigt sind, als auch Konzepte des Modellierungs- und Analyse-Tools HIT von Beilner, wodurch eine hinreichend formale Beschreibung und eindeutige Interpretation des Modells als Voraussetzung für eine automatisierte Übersetzung in ein Leistungsmodell erreicht wird. Eine automatisierte Abbildung einer Teilklasse der *ProC/B* Modellwelt in Petri Netze wurde konzipiert und implementiert. Das Petri Netz kann nach einer manuellen Strukturierung auf eine Markov-Ketten Beschreibung (Kronecker) automatisiert transformiert werden (Buchholz), die als Eingabe für *ASYNC* dient. Alle Transformationen und ein Petri-Netz Editor sind in der APNN-Toolbox verfügbar, *ASYNC* ist neu in die Toolbox integriert und kann vom Frontend aufgerufen werden.

Kapitel 2

Modellierung logistischer Systeme

In diesem Kapitel werden zwei Modelle vorgestellt, anhand derer im nachfolgenden Kapitel die Performance numerischer Analysemethoden experimentell untersucht wird.
Das Modell in Abschnitt 2.1 ist in Kooperation mit Teilprojekt A11 entstanden [11, 12] und reproduziert Prozesse für deren Ausführung ausfallbehaftete Betriebsmittel in Anspruch genommen werden. Die Modellbildung beinhaltet als Basis ein *ProC/B* Modell aus dem in einem zweiten Schritt ein Petri Netz Modell abgeleitet wird.
Das Modell in Abschnitt 2.2 ist [2] entnommen und modelliert eine komplexe Bedienstation. Dieses Modell liegt oft der Bewertung numerischer Analysemethoden zugrunde und hat somit den Charakter eines Referenzmodells.

2.1 Prozesse mit ausfallbehafteten Betriebsmitteln

Die Abbildung 2.1 zeigt das **konzeptionelles Modell**. Es basiert auf Vorüberlegungen [11, 12] von TP A11. Die grundlegende Annahme ist, dass Aktivitäten eines Prozesses zu ihrer Ausführung eine oder mehrere Betriebsmittel (Ressourcen) beanspruchen, die ausfallbehaftet sind. Wenn mehrere Betriebsmittel durch eine Aktivität gleichzeitig beansprucht werden, bestimmt ein Fehlerbaum bzw. eine boolesche Funktion in Abhängigkeit vom Zustand der Betriebsmittel, ob die Aktivität ausführbar ist. Die damit verbundene Flexibilität erlaubt die Modellierung von optional und selektiv nutzbaren Betriebsmitteln. Eine weitere wichtige Annahme ist, dass das Ausfall- und Reparaturverhalten der einzelnen Betriebsmittel durch unabhängige stochastische Prozesse (konkret Poisson'sche Ausfall- und Reparaturraten) beschreibbar ist. Das **Verfügbarkeitsmodell** gleichzeitig beanspruchter **Betriebsmittel** ('r' viele) ist somit eine r-dimensionale Markov-Kette $\{(X_1, \ldots, X_r)_t : t \in T\}$ mit T als Zeitparameter und $X_1, \ldots, X_r \in \{OK, \overline{OK}\}$. Der Zustand OK beschreibt Verfügbarkeit und $\overline{OK}$ Nichtverfügbarkeit (Ausfall) des jeweiligen Betriebsmittels.
Das Verfügbarkeitsmodell der Betriebsmittel muss unabhängig vom Zustand der

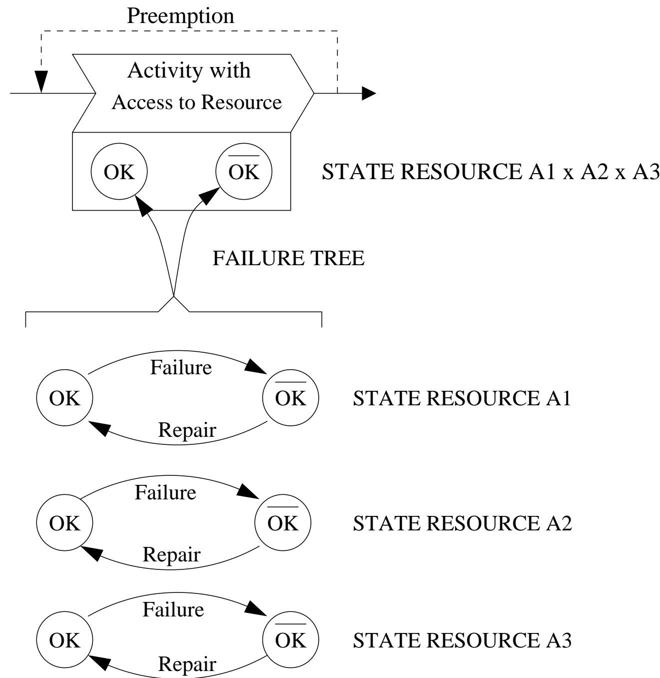


Abbildung 2.1: Prozesselement bzw. Aktivität mit Zugriff auf fehlerbehaftete Betriebsmittel ‘A1’, ‘A2’ und ‘A3’. Jedes Betriebsmittel ist durch 2 Zustände OK und $\overline{OK}$ und Poisson‘sche Fehler- und Reparaturraten gekennzeichnet. Ein Fehlerbaum (boolsche Funktion) legt fest, ob Aktivität ausführbar ist. Aktivitäten werden im Fall der Nichtausführbarkeit blockiert und somit verzögert. Tritt der Ausfall des Betriebsmittels während des Zeitraums der Inanspruchnahme auf, muss die Aktivität wiederholt werden (preemption dispatching).

Prozesse und seiner Aktivitäten sein. Dies schließt die Situation ein, dass der Ausfall während der Inanspruchnahme auftritt. In diesem Fall wird angenommen, dass die Aktivität wiederholt werden muss (preemption dispatching).
Das konzeptionelle Modell aus Abbildung 2.1 wird nachfolgend in ein *ProC/B* Modell überführt. Die Abbildung 2.2 zeigt eine Funktionseinheit, die das Verfügbarkeitsmodell 3 gleichzeitig beanspruchter Betriebsmittel und den Zugriff auf die Betriebsmittel zeigt. Die wesentlichen Bestandteile der Funktionseinheit sind: zwei Prozessketten (Zugriff auf Betriebsmittel und Ausführung (Trajektorie) des Markov Verfügbarkeitsmodells der Betriebsmittel), eine vordefinierte Funktionseinheit vom Typ ‘Prio-Server’ (ausfallbehaftete Betriebsmittel), etwas umfangreichere textuelle Modellierungsvorschriften im HiSlang Programm-Code von HIT [8] (Modellierung der Trajektorie des Markov Verfügbarkeitsmodells) und globale Konstanten/Variablen (Spezifikation der Markov-Kette, d.h. Zustände, Zustandsübergangsraten). Die obere Prozesskette beschreibt den Zugriff auf Betriebsmittel (‘Prio-Server’) und wird durch eine übergeordnete Aktivität aufgerufen. Die Zeit der Inanspruchnahme des Betriebsmittels ist hier durch eine Erlang-Phasenverteilung ge-

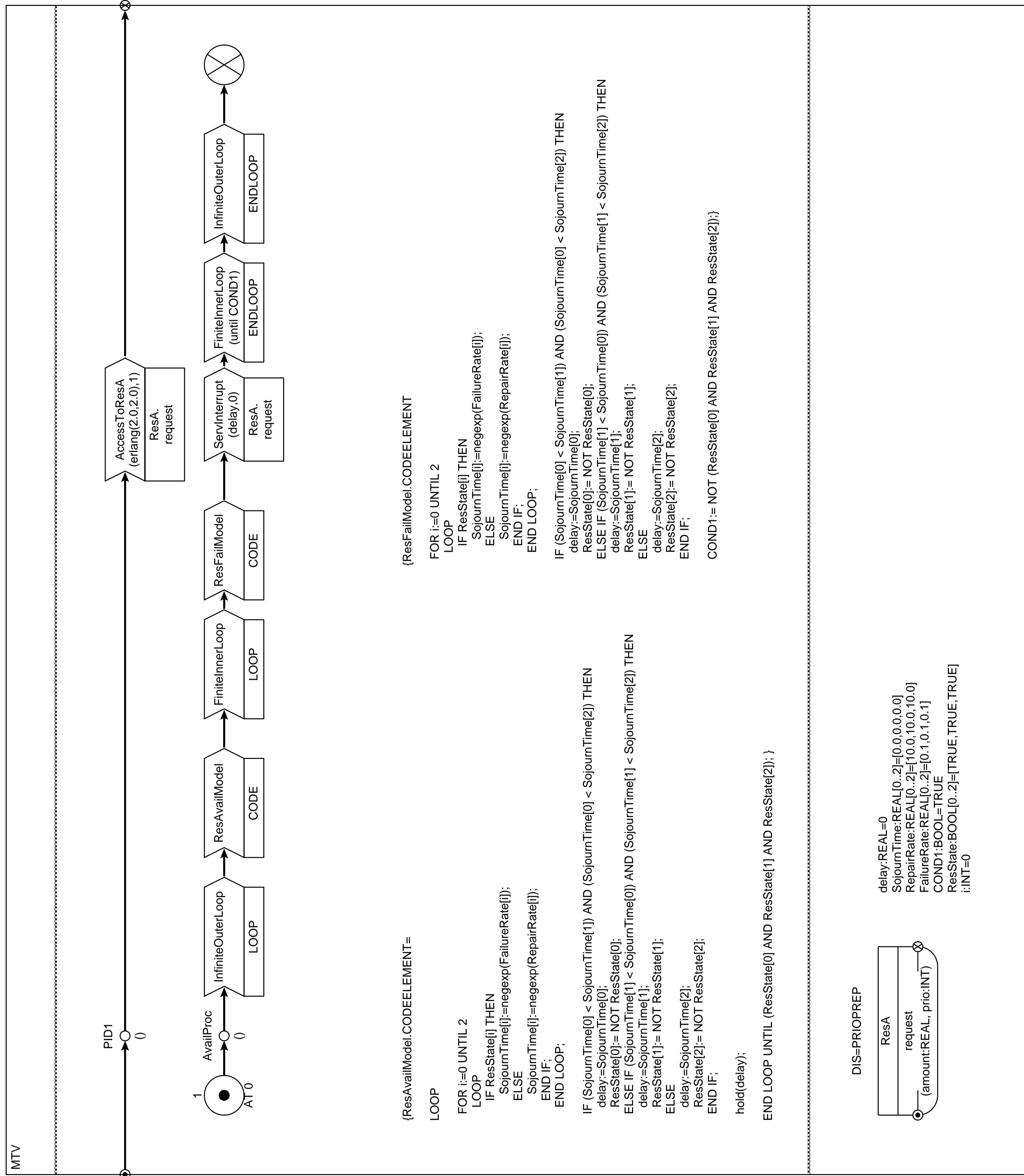


Abbildung 2.2: *ProC/B* Modell (Ausschnitt)

geben. Die zweite Prozesskette wird einmalig generiert und modelliert in einer Endlosschleife die Trajektorie der Markov-Kette. Der Rumpf der Endlosschleife besteht inhaltlich aus 2 Teilen:

1. CODE-PCE 'ResAvailModel': hier werden die Phasen der Trajektorie modelliert, in denen die Markov-Kette in Zuständen ist, in denen das Verfügbarkeits-

modell im Zustand OK ist. Der Programmcode in Abbildung 2.2 links unten, realisiert Zustandsübergänge (erster Teil) und die Aufenthaltsdauer in einem Zustand (`hold(delay)`) innerhalb einer bedingten Schleife, die verlassen wird, wenn nicht der Zustand (OK, OK, OK) vorliegt [1].

2. Bedingte innere Schleife: Hier ist die bedingte Schleife aus dem ersten Teil auf die Ebene der graphischen *ProC/B* Modellbildung hochgezogen worden, um den wesentlich Unterschied zu illustrieren: die Aufenthaltsdauer in einem Zustand (vorher `hold(delay)`) wird nun durch den Aufruf des Dienstes 'request' des Prio-Servers ersetzt. Er bewirkt, dass bereits in Bearbeitung befindliche Inanspruchnahmen durch Aktivitäten unterbrochen und zurückgesetzt werden oder zukünftige Inanspruchnahmen für maximal `delay` Zeiteinheiten blockiert werden.

Das *ProC/B* Modell wird nun händisch in ein äquivalentes Petri Netz Modell übersetzt, um für diese Notation verfügbare nicht-simulative Analysetechniken anwenden zu können. Eine automatische Erzeugung ist nicht möglich, weil das *ProC/B* Modell zahlreiche Modellierungskonstrukte beinhaltet (Programm-Code etc), die nicht der Klasse (automatisch) abbildbarer Konstrukte angehören. Trotzdem legt die Semantik der Anwendung eine Modellierung mit Petri-Netzen nahe, da die Dynamik des Verfügbarkeitsmodells der Betriebsmittel und der Zugriff auf sie mit Petri Netzen kompakter und einfacher zu beschreiben ist im Vergleich zur *ProC/B* Notation. Alternativ kann das *ProC/B* Modell automatisch in die Einagbesprache des HIT-Simulators übersetzt und dort simulativ analysiert werden. Somit wäre es prinzipiell möglich, die Effizienz unterschiedlicher Analyseansätze mit unterschiedlichen Modellbildungen der gleichen Anwendung zu vergleichen.
Das **Petri-Netz Modell** in Abbildung 2.3 beschreibt eine Sequenz von nun 4 Aktivitäten, für deren Ausführung ausfallbehaftete Betriebsmittel beansprucht werden. Es ist zu beachten, dass bei der Einführung des *ProC/B* Modells oben auf eine einzelne Aktivität fokussiert wurde, weil dieser Ausschnitt bereits die wesentlichen Eigenschäften des Gesamtmodells offenbart. Die oberen 4 Rechtecke in 2.3 beschreiben die Aktivitäten des Prozesses und den Zugriff auf die Betriebsmittel, der mittlere Teil beschreibt die boolsche Funktion, welche festlegt, welche Konstellationen bei gleichzeitiger Beanspruchung mehrerer Betriebsmittel die Ausführbarkeit der Aktivität erlauben und der untere Teil stellt das Verfügbarkeitsmodell der Betriebsmittel dar. Die zweite und vierte Aktivität des Prozesses und die darunterliegenden Teile sind eine 1:1 Umsetzung des *ProC/B* Modells von oben. Die erste und dritte Aktivität beanspruchen ein einzelnes und geteiltes Betriebsmittel. Hier entfällt die Modellierung der booleschen Funktion. Die Prozessbeschreibung wird im Kurzschluss modelliert, d.h. es zirkuliert eine feste Anzahl (hier =12) von Prozessen. Mit

[1] O.b.d.A wird hier angenommen, dass alle Betriebsmittel für die Ausführbarkeit der Aktivität verfügbar sein müssen. Natürlich können auch beliebig andere boolesche Funktionen modelliert werden, etwa dass mindestens zwei Betriebsmittel verfügbar sind etc.

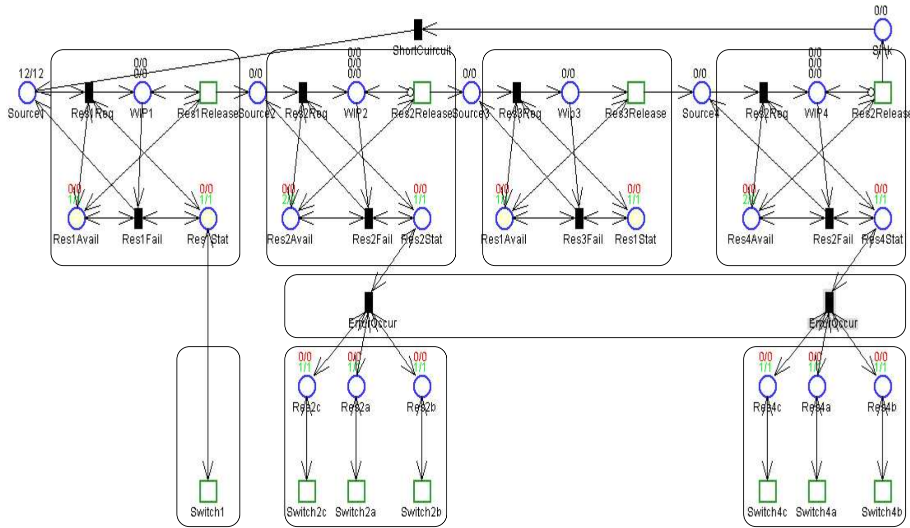


Abbildung 2.3: Petri Netz Modell eines Prozesses mit ausfallbehafteten Betriebsmitteln: Prozess mit Zugriff auf Betriebsmittel (oben), Fehlerbaum (boolsche Funktion) (mittig) und Verfügbarkeitsmodell mit Poisson Ausfall- und Reparaturraten (unten)

dieser Belegung der Modellparameter hat die unterliegende Markov-Kette 1 134 672 Zustände.

2.2 Komplexe Bedienstation

Ein ‘Multi-Server Multi-Queue’ (`'MSMQ'`) Bediensystem [2] besteht aus $J > 1$ Warteschlangen Stationen, die von S Bedienern zirkulierend besucht werden, vgl. Abbildung 2.4 links. Jede Station j $(1 \leq j \leq J)$ hat einen Puffer mit endlicher Kapazität C_j. Die Last (Kunden etc) wird durch stations-spezifische Poisson-Prozesse mit Rate λ_j generiert. Bei einem Puffer-Überlauf gehen Kunden verloren. Die Kunden haben Erlang-2 verteilte Bedienwünsche mit Mittelwert μ_j^{-1}. Der Übergang eines Bedieners von der Station j zur Station $(j \bmod J) + 1$ dauert im Mittel ω_j^{-1} Zeiteinheiten. Wenn bei Ankunft oder nach Beendigung der Bedienung die Warteschlange leer ist, geht der Bediener zur nächsten Warteschlange über. Wenn nach der Bedienung die Warteschlange nicht leer ist, geht entweder der Bediener trotzdem zur nächsten Station oder er bleibt in der aktuellen Station (Modellvarianten). An einer Station können mehrere Bedienvorgänge simultan stattfinden, sofern $S > 1$ ist. Die Abbildung 2.4 rechts zeigt das GSPN-Modell einer einzelnen Station. Das SGSPN-Modell für die gesamte `'MSMQ'` Anwendung besteht aus J GSPN-Modellen einzelner Stationen, die über die Transitionen `t_Sw_j` und `t_Sw_j+1` kommunizieren. Die Transitionen `t_Buf`, `t_Proc`, `t_Sw_j` und `t_Sw_j+1` beschreiben die Ankunft/Bearbeitung von Kunden bzw. Ankunft/Weggang eines Bedieners.

Für den Performance-Test numerischer Verfahren im nächsten Kapitel werden die

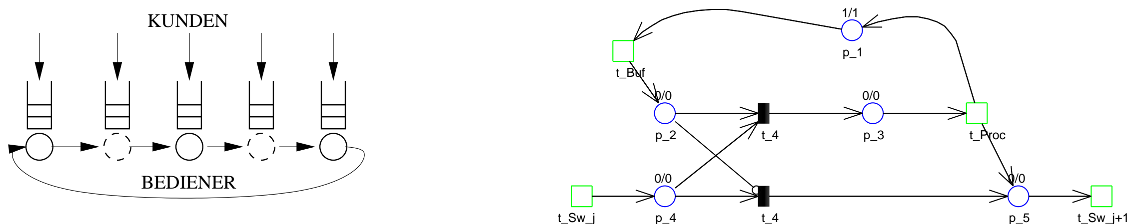


Abbildung 2.4: <u>Links</u>: `'MSMQ'` Bedienstation mit $J = 5$ Warteschlangen und $S = 3$ Bedienern; <u>Rechts</u>: *GSPN*-Modell der einzelnen Warteschlange mit polling-Bedienung (Komponente j)

Modellparameter wie folgt belegt:

$J = 6$	Anzahl der Warteschlangen
$S = 3$	zwischen Warteschlangen zirkulierende Bediener
$C_1, \ldots, C_6 = 3$	Kapazität der Warteschlangen
$\lambda_1 = 0.2, \lambda_2, \ldots, \lambda_5 = 0.5, \lambda_6 = 1.5$	Ankunftsraten
$\omega_1, \ldots, \omega_6 = 10.0$	Stationsübergangsrate
$\mu_1, \ldots, \mu_6 = 1.0$	Erlang-2 Bedienraten

Aus dieser Konfiguration resultiert eine Markov-Kette mit 11 417 500 Zuständen.

Kapitel 3

Test numerischer Analyseverfahren

Das vorherige Kapitel führte *ProC/B* und Petri Netz Modelle logistischer Anwendungen ein. Die Modelle sind so konstruiert, dass der unterliegende stochastische Prozess eine Markov-Kette (*CTMC*) ist. Die APNN-Toolbox [5] beinhaltet Algorithmen, die eine *CTMC* Darstellung aus einem Petri-Netz generieren. Die Datenstruktur zur Speicherung der *CTMC* kann in zwei Ausprägungen erzeugt werden: eine modular-strukturierte Kronecker-basierte Darstellung [10] oder eine hierarchisch-strukturierte Kronecker-basierte Darstellung [4, 3, 6]. In der Experimentreihe wird neben der *CTMC* Darstellung der Typ des numerischen (iterativen) Verfahrens und die verwendete Rechnerarchitektur variiert:

- ***CTMC* Darstellung**: *MK_X*= modulare Kronecker-Darstellung, 'X' verweist auf unterschiedliche Implementierungen zur Darstellung erreichbarer Zustände; *HierKron*= hierarchische Kronecker-Darstellung;

- **Iterative Verfahren**: *JOR*= (punktweise) Jacobi Iteration mit Relaxation; *BSOR*= (blockweise) Multi-Level Iteration mit Jacobi Iteration auf Blockebene (=äußere Iteration) und Gauss-Seidel Iteration mit Relaxation innerhalb der Blöcke (=innere Iteration); *ASYNC*= Erweiterung von *BSOR* mit paralleler und asynchroner Block-Jacobi Iteration und geschachtelter Gauss-Seidel Iteration;

- **Rechnerarchitektur**: *SBl*=**S**UN **Bl**ade 100, UltraSparc IIe Prozessor, 500MHz, 512MB Hauptspeicher; *SEp*=**S**UN **E**nter**p**rise 250, 2 x UltraSparc II Prozessor, 400MHz, 2GB Hauptspeicher;

In der Experimentreihe werden verschiedene Performance-Maße gemessen:

- Anzahl von **Iterationenschritten** (*ITER*), verbrauchte **CPU-Zeit** je Rechenprozess für alle ausgeführten Iterationsschritte (*CPU-T*) und verbrauchte **User/Real-Zeit** je Rechenprozess für alle ausgeführten Iterationsschritte(*USR-T*);

- sowie spezielle Maße, die das stochastische Laufzeitverhalten von *ASYNC* beschreiben.

Hierbei werden die folgenden Fragestellungen beantwortet:

F1 Laufzeit mit unterschiedlichen Implementierungen *MK_Stree*, *MK_Fold*, *MK_Shuf* der modularen Kronecker-Darstellung für ein festes iteratives Verfahren (=*JOR*). *MK_Shuf* ist eine neue Implementierung, bei der Wahrscheinlichkeiten erreichbarer Zustände (=Einträge im Iterationsvektor) über probabilistische Entscheidungsdiagramme multiplikativ erzeugt werden [7], siehe auch TP M2, AP 2 in [1].

F2 Vergleich Laufzeit eines iterativen Verfahrens (=*JOR*) auf unterschiedlichen Implementierungen (*MK_X* vs. *HierKron*) der *CTMC* Darstellung.

F3 Leistungsfähigkeit eingesetzter Rechnerarchitekturen *SBl* vs. *SEp*.

F4 Stochastischer Speed-up und Effizienz bei *ASYNC*, siehe auch TP M2, AP 4 in [1].

F5 Stochastisches Laufzeitverhalten in *ASYNC* (ausgewählte Beispiele), siehe auch TP M2, AP 4 in [1]. Das stochastische Laufzeitverhalten von *ASYNC* wird durch 3 Faktoren beeinflusst: geteilte Rechen- und Kommunikationsressourcen, stochastische Kommunikationsmuster und die selektive, asynchrone Iterationen. Es sind keine auswertbaren analytischen Modelle bekannt, die den Einfluss dieser Faktoren auf die Konvergenz und Performance beschreiben. Deswegen muss sich die Analyse auf experimentelle Untersuchungen beschränken, in denen Heuristiken zum Einfluss der Faktoren beurteilt werden. Hierfür ist es notwendig, den dynamischen Ablauf der Iteration transparent zu machen. Zu diesem Zweck beinhaltet *ASYNC* Monitoring-Funktionalitäten. Es werden u.a. Traces generiert, in denen Informationen zum dynamischen Ablauf protokolliert werden. Die Daten können ex-post betrachtet werden.

Die *CTMC* des Modells von Prozessen mit ausfallbehafteten Betriebsmitteln (Abschnitt 2.1) hat 1 134 672 Zustände und in der *HierKron* Implementierung eine Blockstruktur mit 91 $\times$ 91 Blöcken (maximale Blockgröße 42 768 Zustände). Die *CTMC* der `'MSMQ'` Bedienstation ist deutlich (etwa zehnfach) größer, sie hat 11 417 500 Zustände und in der *HierKron* Implementierung 56 $\times$ 56 Blöcke (maximale Blockgröße 274 625). Derartig große, keine Symmetrien oder extrem dünne Besetzungsgrade aufweisende *CTMC* Modelle induzieren einen Platz- und Berechnungsaufwand, der an die Leistungsgrenzen verfügbarer numerischer Analysemethoden stößt.

3.1 Ergebnisse Fallbeispiel 1

Die Tabelle 3.1 beinhaltet die Ergebnisse der Experimentreihe zum Test numerischer Analyseverfahren (angewendet auf Modell aus Abschnitt 2.2).

No	Method	CTMC	P	*ITER*	*CPU-T*	*USR-T*
1	*JOR*	*MK_Stree*	$1 \times SBl$	1260	5334	5435
2	*JOR*	*MK_Fold*	$1 \times SBl$	1260	3149	3394
3	*JOR*	*MK_Shuf*	$1 \times SBl$	1260	$\gg$50000	$\gg$50000
4	*JOR*	*HierKron*	$1 \times SBl$	1260	4328	4362
5	*BSOR*	*HierKron*	$1 \times SBl$	304/4	2627	2644
6	*BSOR*	*HierKron*	$1 \times SEp$	304/4	2317	2318
7	*ASYNC*	*HierKron*	$1 \times SBl$	311/4	2875	2905
8	*ASYNC*	*HierKron*	$1 \times SEp$	311/4	2588	2604
9	*ASYNC*	*HierKron*	$1 \times SEp$	dynam[1)]	1387	1445
			$1 \times SEp$	dynam[1)]	1393	1443
10	*ASYNC*	*HierKron*	$1 \times SEp$	dynam[1)]	923	1088
			$1 \times SEp$	dynam[1)]	909	1097
			$1 \times SBl$	dynam[1)]	1009	1099
			$1 \times SBl$	dynam[1)]	1013	1099

Tabelle 3.1: Performance numerischer Analyseverfahren im Vergleich

In Bezug auf den Fragekatalog aus der Einleitung des Kapitels können folgende Ergebnisse festgehalten werden:

F1 (Quelle Experimente No 1..3): In No 2 wird mit 3149/3394 Sekunden die beste Laufzeit erreicht. In No 3 wurde die Iteration nach 50000 Sekunden abgebrochen (ohne dass die Genauigkeitsschranke erreicht wurde), weil die verbrauchte Rechenzeit den Vergleichswert aus No 2 drastisch übersteigt. Leider erzeugt das Modell ein Szenario, in dem keine Platzersparnis bei der probabilistischen Darstellung des Iterationsvektors und zugleich teure Zugriffe auf den Iterationsvektor zu beobachten sind.

F2 (Quelle No 2+4): 3149/3394 Sekunden benötigt die beste *JOR* Iteration (mit *MK_Fold*) im Vergleich zu 4328/4362 Sekunden der besten *JOR* Iteration mit *HierKron CTMC*-Darstellung. Die Anzahl der Iterationsschritte ist stets 1260, weil die Darstellung der *CTMC* nicht das numerische Berechnungsmodell (*JOR*) verändert.

F3 (Quelle No 5-8): Die CPU-Zeiten in Experiment 5 vs. 6 und 7 vs. 8 zeigen, dass die SUN Enterprise 250 trotz etwas kleinerer Taktfrequenz 10 Prozent schneller ist.

F4 (Quelle No 5+6 vs. No 7 bis 10) In der *BSOR*-Iteration (No 5+6) ist die Auswahlreihenfolge der Blöcke und die Anzahl innerer Iterationsschritte deterministisch festgelegt. Die Genauigkeitsschranke wird nach 304 äußeren mit jeweils 4 eingebetteten inneren Iterationsschritten erreicht. Das gleiche numerische Berechnungsmodell liegt der *ASYNC* Implementierung in No 7 und 8

zugrunde. Die Anzahl der ausgeführten Schritte ist in *ASYNC* etwas höher, weil in *ASYNC* nicht nach jedem Schritt die Terminierungsbedingung getestet wird und die Überprüfung der Terminierungsbedingung dezentral ist (hier zwischen Master-Prozess und einem Rechenprozess), verbunden mit einem Kommunikationsprotokoll, das zu einem fehlerfreien Abbruch führt. Bis zur Beendigung des Protokolls iterieren und kommunizieren die Prozesse weiter, was eine gewisse Verzögerung bewirkt. In *ASYNC* ist die CPU-Zeit je Iterationsschritt um etwa 7 Prozent höher. Hierin drückt sich ein Mehraufwand aus, der für die verteilte, flexible, gesteuerte und dynamische Ausführung der Iteration notwendig ist. Beispielsweise werden Trace-Informationen protokolliert, Kommunikationsbefehle aufgerufen und zusätzliche Bedingungen zur Steuerung der Iteration getestet.

In No 9 wird *ASYNC* in seiner vollständigen Funktionalität mit 2 Prozessen getestet. Der **Speed-Up** ist **1.86** und **1.67** (CPU-Zeit im Vergleich zur sequentiellen Variante von *ASYNC* in No 8 und *BSOR* in No 6) bzw. **1.80** und **1.60** (USER-Zeit im Vergleich zu No 8 und 6). Hierzu sind zwei Anmerkungen wichtig: Das numerische Berechnungsmodell in No 9 ist nicht identisch mit dem in 6 und 8, weil die Iteration randomisiert ist. Insofern bezieht sich der Speed-up auf unterschiedliche Berechnungen und kann den Wert 2 übersteigen. Die Randomisierung bewirkt zudem, dass die Anzahl der Iterationen bis zum Erreichen der Genauigkeitsschranke in wiederholten Experimenten variiert. Der Speed-up bezüglich der CPU-Zeit ist immer besser als der Speed-up bzgl. der User-Zeit, weil bei der verteilten Berechnung neben dem Rechenprozess stets ein daemon-Prozess (Abwicklung der Kommunikation) um CPU-Zeit konkurriert. Erfahrungsgemäß benötigt der daemon-Prozess 10-15 Prozent der CPU-Zeit. Deswegen ist in No 9+10 die User-Zeit der einzelnen Prozesse stets um 10-15 Prozent höher als ihre CPU-Zeit.

In No 10 mit $P = 4$ kann der Speed-Up gegenüber No 9 zwar verbessert werden, allerdings auf Kosten einer deutlich verschlechterten Effizienz (=gesamte CPU Zeit aller Prozesse), der Wert steigt in No 8,9,10 von 2588-2780-3854 Sekunden.

F5: Stochastisches Laufzeitverhalten von *ASYNC* Die nachfolgenden Ausführungen beziehen sich aus das Experiment No 10. Die Abbildung 3.1 zeigt einen Ereignis-Trace, in dem Kommunikationsereignisse (diagonal verlaufende Linien), sowie Prozesszustände (horizontal verlaufende Bänder) und weitere prozessinterne Ereignisse (senkrechte Linien in den Bändern) über der Zeit visualisiert werden[1]. Der unterste Balken (weiß) repräsentiert den Master-Prozess, der nur Kontrolldaten zur Überwachung empfängt und wenig kommuniziert. Der Trace zeigt deutlich den chaotisch/stochastischen Charakter des Kommunikationsmusters.

[1] Dieser Trace ist bereits Bestandteil des PVM-Kommunikationstools

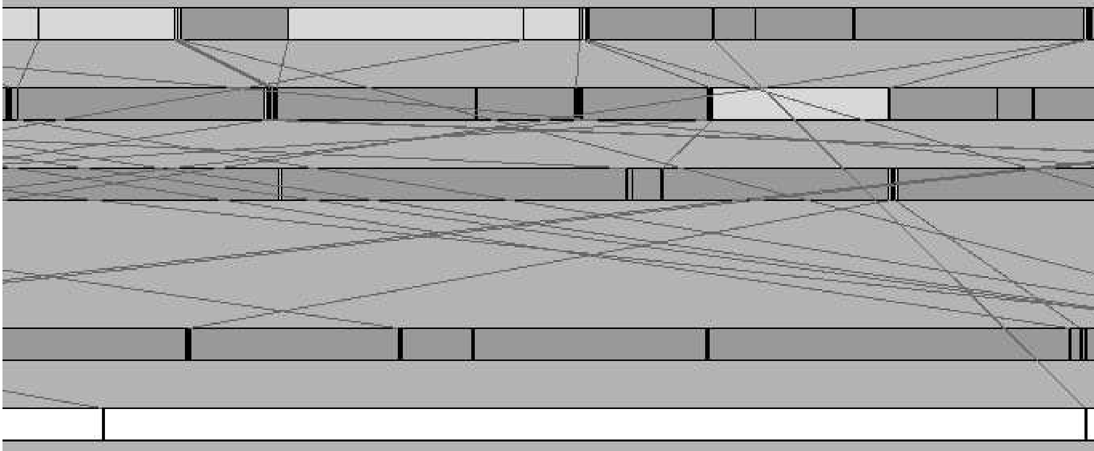


Abbildung 3.1: Ereignis Trace der Prozesse in *ASYNC* (mit Kommunikation)

Die effiziente Zuordnung der Rechenprozesse (hier 4) zu den Blöcken (hier 91) des Iterationsvektors ist als Partitionierungsproblem eines knoten- und kantengewichteten Graphs interpretierbar, wenn Effizienz durch Balanziertheit der Rechenlast und geringen Kommunikationsaufwand definiert wird[2]. Die in diesem Sinne beste Lastverteilung ist in der Abbildung 3.1 angegeben. Sie wurde mit Techniken aus der Chaco Toolbox [9]. berechnet. Da die Modellstruktur mit geeigneter Dekomposition in die Erzeugung der Blockstruktur der *HierKron*-Darstellung einfließt, ergibt sich oft bereits aus der gegebenen Anordnung der Blöcke eine geeignete Aufteilung des Berechnungsproblems. Es kann oft beobachtet werden, dass aufeinanderfolgende Blöcke einem Prozess zugeordnet, vgl. Abbildung 3.1.

Prozess	zugeordnete Blöcke
1	1..5, 14..18, 26..30, 37..40, 47..50, 56..58, 64, 65
2	6..13, 19..25, 31..36, 41..46, 51, 53..55, 59
3	52, 60..63, 67..70, 74..76, 80, 81, 85, 88
4	0, 66, 71..73, 77..79, 82..84, 86, 87, 89..90

Tabelle 3.2: Lastverteilung (induzierter Kommunikationsaufwand für einen synchronen *BJAC*-Schritt: 5.637 MB verteilt auf 36 Nachrichten)

In der Tabelle 3.3 ist die Laufzeit (User) der einzelnen Prozessen hinsichtlich bestimmter Phasen unterteilt. Idle-Phasen treten ein, wenn ein Prozess nicht weiter iterieren soll, weil nicht hinreichend viele aktualisierte Daten, die in die Iteration einfließen, empfangen werden können. Idle-Phasen treten hier praktisch nicht auf. Die eigentliche Iteration wird konzeptionell in 3 Phasen (Comp1,Comp2 und Comp3) unterteilt: 1) Iteration von Blöcken, in die nur Daten einfließen, die aufgrund der

[2]Neben dem technologisch-orientierten Maß des Kommunikationsaufwands kann alternativ auch der numerische Kopplungsgrad (NCD-Charakteristikum) zwischen den Blöcken benutzt werden

Lastverteilung lokal verfügbar sind (Datenlokalität), 2) Iteration von Blöcken, in die zu kommunizierende Daten einfließen und 3) die Berechnung/Aufbereitung von Zwischenergebnissen, die lokal nicht benötigt werden, aber in Zusammenarbeit mit anderen Prozessen an diese verschickt werden. Desweiteren wird die Laufzeit durch CTRL-Phasen verlängert, in denen die Iteration überwacht und gesteuert wird (Berechnung der lokalen Genauigkeit der Iteration, Auswahl zu iterierender Komponenten, Generierung des Trace etc.).

TOTAL	IDLE	COMP1	COMP2	COMP3	CTRL
1.087711e+03	8.680165e-02	5.907314e+02	3.054483e+02	1.369738e+01	1.748169e+02
1.097276e+03	9.386086e-02	4.492999e+02	4.331722e+02	2.312042e+01	1.878242e+02
1.099210e+03	3.787518e-02	3.138726e+02	5.681394e+02	2.091269e+01	1.827242e+02
1.098905e+03	4.177916e-02	3.978402e+02	4.862754e+02	2.102424e+01	1.809173e+02

Tabelle 3.3: Unterteilung der Laufzeit (User) von *ASYNC* in NO 10: TOTAL=Gesamtzeit; IDLE=Wartezeiten (treten nur auf, wenn alle empfangenen Daten so 'alt' sind, dass eine Weiterberechnung auf ihnen sinnlos ist); COMP1=Iterationen, in die nur Daten einfließen, die lokal berechnet werden (Datenlokalität); COMP2=Iterationen, in die empfangene Daten einfließen, die von anderen Prozessen 'erzeugt' werden; COMP3=Berechnung von Zwischenergebnissen, die Daten generieren, die ausschließlich in die Berechnung anderer Prozesse einfließen; CTRL=Steuerung und Überwachung der Iteration

ASYNC hat eine Wahlfreiheit bei der Auswahl zu iterierender Blöcke (selektive Iteration) und nach getroffener Auswahl des Blocks Wahlfreiheit für die Anzahl von *JOR* Iterationsschritten innerhalb des Blocks (nicht-stationäre, geschachtelte Iteration). In der Tabelle 3.4 ist die Gesamtanzahl innerer Iterationsschritte bzgl. einzelner Blöcke aufgelistet - hier exemplarisch für alle Blöcke, die Prozess 1 zugeordnet sind. Es wird deutlich, dass die Anzahl innerer Iterationsschritte block-abhängig variiert. Die Auswahl der Anzahl innerer Iterationen ist dynamisch und basiert hier auf dem Aktualisierungsgrad einfließender Daten - nachfolgend kurz als URD-Kriterium (Update-Ratio-Driven) bezeichnet (insgesamt stehen 9 Auswahlstrategien zur Verfügung). Das Auswahlkriterium basiert auf der Heuristik, dass viele Iterationen auf veralteten Daten kontraproduktiv für die Konvergenz sind, der Trace der global erreichten Genauigkeit zeigt für solche Szenerien oft starke Schwankungen. Das URD-Kriterium bewirkt, dass Blöcke tendenziell mehr iteriert werden, wenn nur Daten einfließen, die lokal verfügbar sind, weil sie bei Bedarf aktualisiert werden können. Im vorliegenden Szenario wird genau für diese Blöcke (mit Stern gekennzeichnet) eine maximale Anzahl von 1828 Iterationen erreicht. Auffällig ist die deutlich nach unten abweichende Iterationszahl für Block 65. Sie zeigt an, dass die Rate, mit der aktualisierte Daten verfügbar gemacht werden, hier zu gering ist.
Die Spalte ACCUPRE in Tabelle 3.4 gibt die akkumulierte Anzahl von Anfragen an, mit denen andere Prozesse bzgl. der einzelnen Blöcke nach Aktualisierungen

Block	InnerIter	MOpUpRe	AccUpRe
1*	1828	0.000000e+00	0
2*	1828	0.000000e+00	0
3*	1828	0.000000e+00	0
4*	1828	0.000000e+00	0
5*	1828	0.000000e+00	0
14*	1828	0.000000e+00	0
15*	1828	0.000000e+00	0
16*	1828	0.000000e+00	0
17*	1828	0.000000e+00	0
18	1664	1.310044e-02	310
26*	1828	0.000000e+00	0
27*	1828	0.000000e+00	0
28*	1828	0.000000e+00	0
29*	1828	0.000000e+00	0
30	1266	6.113537e-01	451
37*	1828	0.000000e+00	0
38*	1828	0.000000e+00	0
39*	1828	0.000000e+00	0
40*	1828	0.000000e+00	0
47*	1828	0.000000e+00	0
48*	1828	0.000000e+00	0
49*	1828	0.000000e+00	0
50	1014	9.497817e-01	456
56*	1828	0.000000e+00	0
57*	1828	0.000000e+00	0
58	1792	6.026201e-01	444
64	1780	4.759825e-01	443
65	460	2.641921e-01	435

Tabelle 3.4: Iterations-Szenario von Prozess 1: InnerIter=block-abhängige Anzahl innerer Iterationen; MOpUpRe= mittlere Anzahl offener Aktualisierungs-Anfragen initiiert durch andere Prozesse (Maß für Befriedigung des Bedarfs an aktualisierten Werten);AccUpRe=akkumulierte Anzahl von Aktualisierungs-Anfragen (Maß für Nachfragebedarf)

anfragen (Maß für Nachfragebedarf). Die Spalte MOpUpRe ist eine über die Zeit gemittelte Anzahl von Aktualisierungs-Anfragen anderer Prozesse, die nicht bearbeitet werden können, weil der Prozess 1 gemäß des URD-Auswahlkriteriums keine Aktualisierung der betreffenden Daten berechnen kann (Maß für Befriedigung des Bedarfs an aktualisierten Werten).

3.2 Ergebnisse Fallbeispiel 2

Die Ergebnisse der Experimentreihe zum Modell aus dem Abschnitt 2.1 sind in der Tabelle 3.5 zusammengefasst.

No	Method	CTMC	P	*ITER*	*CPU-T*	*USR-T*
1	*JOR*	*MK_Stree*	$1 \times SBl$	n.e.	n.e.	n.e.
2	*JOR*	*MK_Fold*	$1 \times SBl$	460	13836	14280
3	*JOR*	*MK_Shuf*	$1 \times SBl$	n.e.	n.e.	n.e.
4	*JOR*	*HierKron*	$1 \times SBl$	460	15760	16010
5	*BSOR*	*HierKron*	$1 \times SBl$	247/4	27880	28570
6	*BSOR*	*HierKron*	$1 \times SEp$	247/4	20550	20690
7	*ASYNC*	*HierKron*	$1 \times SEp$		11479	11930
		HierKron	$1 \times SEp$		11471	11939
8	*ASYNC*	*HierKron*	$1 \times SBl$	dynam[1)]	7788	8993
			$1 \times SEp$	dynam[1)]	8482	9049
			$1 \times SEp$	dynam[1)]	8531	9043
			$1 \times SBl$	dynam[1)]	7704	9066
			$1 \times SBl$	dynam[1)]	7571	8995
			$1 \times SBl$	dynam[1)]	7910	9044
			$1 \times SBl$	dynam[1)]	7347	8991
			$1 \times SBl$	dynam[1)]	7236	8988
9	*ASYNC*	*HierKron*	$7 \times SBl$	dynam[1)]	4780	7136
			$2 \times SEp$	dynam[1)]	..6264	..7174
10	*ASYNC*	*HierKron*	$8 \times SBl$	dynam[1)]	2380	8588
			$2 \times SEp$	dynam[1)]	..7494	..8632

Tabelle 3.5: Performance numerischer Analyseverfahren im Vergleich

Hinsichtlich des Fragekatalogs aus der Einleitung des Kapitels können folgende Beobachtungen dokumentiert werden:

F1 (Quelle No 1..3): In No 2 wird mit 13836/14280 Sekunden eine *JOR* Laufzeit erreicht, die alle sequentiellen Iterationen aus No 4 bis 6 unterbietet. Leider konnten die verbleibenden *MK_X* Implementierungen nicht getestet werden, weil die Generierung des Zustandsraums fehlschlug. Die potentielle Größe des Zustandsraums ($=183^6 \approx 3.8E13$) übersteigt die handhabbare Obergrenze für die *MK_Stree* und *MK_Shuf* Implementierung. Für eine Konfiguration des Modells, die zu einem erzeugbaren (kleineren) Zustandsraum führt, stieg die Zeit je Iterationsschritt um etwa 4 Größenordnungen. Ähnliche Beobachtungen sind auch in [7] dokumentiert.

F2 (Quelle No 2+4): Die *JOR* Iteration benötigt mit der *MK_Fold* Implementierung 13836/14280 Sekunden und ist damit schneller als mit der *HierKron*

Implementierung, die 15760/16010 Sekunden benötigt (+12 Prozent).

F3 (Quelle No 5+6): Die CPU-Zeiten in No 5 vs. No 6 zeigen, dass die SUN Enterprise 250 trotz etwas kleinerer Taktfrequenz 26 Prozent schneller ist.

F4 (Quelle No 5+6 vs. No 7-9): In No 7-10 wird *ASYNC* in seiner vollständigen Funktionalität mit 2, 8,9 und 10 Prozessen getestet. Mit 2 Prozessen ist der **Speed-Up** gleich **1.73** im Vergleich zu *BSOR* (No 6) und **1.20** (No 2) im Vergleich zu *JOR* (alle Angaben bzgl. *USR-T*). Mit 8 Prozessen konnte der **Speed-Up** kaum verbessert werden, er beträgt nun **2.30** bzw. **1.59**. Mit 9 Prozessen ist der **Speed-Up 2.89** bzw. **2.0**. Im Experiment mit 10 Prozessen wird der **Speed-Up** schlechter, er beträgt nun **2.41** und **1.66**. Die **Effizienz** (=gesamte CPU Zeit aller Prozesse) ist 22950 Sekunden (P=2), 62569 Sekunden (P=8) 52884 Sekunden (P=9) und 61614 Sekunden (P=10). Im Vergleich dazu stehen 20550 Sekunden (bester *BSOR*) und 13836 Sekunden (bester *JOR*).

F5: Stochastisches Laufzeitverhalten von *ASYNC* Die Größe der untersuchten *CTMC* verursacht einen hohen **Kommunikationsaufwand** in *ASYNC*. Ein (äußerer) Iterationsschritt, der verteilt von 2 Prozessen durchgeführt wird, induziert bereits 19 Nachrichten (Vektoren) mit einem Datenumfang von 33.3 MB. Der Aufwand steigt auf 59 Nachrichten und 98.2 MB (P=8), 66 Nachrichten und 105.8 MB (P=10) und 71 Nachrichten und 114.1 MB (P=12, hier nicht betrachtet). Dies führt zu einem erhöhten Speicherplatzaufwand bei den Rechenprozessen für lokale Datenpuffer. Andererseit wird dieser Mehraufwand durch die Verteilung des Iterationsvektors kompensiert. In Zahlen ausgerückt: der Iterationsvektor hat eine Dimension von 11417500 Einträgen, d.h. der Speicherverbrauch (double-Einträge mit je 8 Byte) ist 87.1 MB. Somit ist der mittlere Speicherverbrauch der Rechenprozesse (87.1+33.3)/2=60.2 MB (P=2), 23.2 MB (P=8), 19.3 MB (P=10) und 16.8 MB (P=12).
Nachfolgend wird das *ASYNC* **Kommunikationssystem** mit Messdaten aus dem Experiment No 9 untersucht. Die Kommunikation wird durch ein sogenanntes 'Demand Driven Messaging' Protokoll und ein 'Supply Driven Messaging' Protokoll gesteuert.
Das verwendete **Demand Driven Messaging** (DDM) Protokoll steuert die Kommunikation aus der Sicht eines Datenkonsumenten. Wenn ein Konsument Daten aus dem Kommunikationspuffer liest, signalisiert er dem Produzenten den Bedarf nach einer Aktualisierung, indem eine Anforderung an den Produzenten verschickt wird. Bis die aktualisierten Daten vorliegen, kann der Konsument (asynchron) fortfahren und auf veraltete Daten sukzessive zugreifen (ohne erneut eine Anforderung zu verschicken). Dabei ist eine selektive Vorgehensweise, bei der Daten mit hohem Aktualisierungsgrad bevorzugt benutzt werden, möglich. Das Protokoll bewirkt, dass alle vom Produzenten berechneten und verschickten Daten auch tatsächlich verwendet werden. Die Verfügbarkeitsrate, d.h. die Häufigkeit je Zeitintervall mit der

Daten aktualisiert werden ist durch die Konsumrate beschränkt. Bei einer Entkopplung zwischen Datenproduzent und -empfänger würden partiell hohe Kommunikationsraten (Verfügbarkeitsrate > Konsumrate) und damit verbunden Instabilitäten auftreten.

Das DDM Protokoll ist mit einem **Supply Driven Messaging** (SDM) Protokoll kombiniert, das die Kommunikation aus der Sicht eines Datenproduzenten steuert. Die wesentlich Aufgabe des Protokolls ist es, keine Daten zu kommunizieren, die hinreichend ähnlich oder sogar Duplikate sind. Wenn beispielsweise bei einem Produzenten angeforderte Daten seit der letzten Anfrage nicht ausreichend aktualisiert wurden, so wird die Anforderung nicht unmittelbar bedient und ggf. bereits verschickte Daten nicht nochmals verschickt.

Da jeder Rechenprozess gleichermaßen als Datenkonsument und -produzent fungiert, ist das Zusammenspiel aller Prozesse gekoppelt über das indeterministisch wirkende Kommunikationsmedium ein komplexes System. Nachfolgend werden **Messdaten** für eine konkrete **Lastsituation** und beobachtete **Leistungskennzahlen** vorgestellt, vgl. Tabelle 3.7. Die Zeilen in Tabelle 3.7 repräsentieren zu kommunizierende Daten (=Teilvektoren mit Index 'BLOCK'), die während der Iteration von einem Prozess mit dem Index 'P' iteriert (aktualisiert) und kommuniziert werden. Die Werte in Spalte DATAREQ sind die Häufigkeiten, mit der ein Datenkonsument auf den Teilvektor zugreift (Konsumrate), in der Spalte MSGRECV steht die Anzahl der Aktualisierungen für den jeweiligen Teilvektor (Aktualisierungsrate). Die Werte in den Klammern in Spalte DATAREQ sind der prozentuale Anteil von Zugriffen auf aktuelle Daten, die eine Aktualisierung anfordern (DDM). Kritische Werte, bei denen der Anteil unter 33 Prozent liegt, sind durch Unterstreichung gekennzeichnet. Im Optimalfall, wenn stets auf aktuelle Werte zugegriffen und eine Anforderung generiert wird, ist die Konsumrate=Verfügbarkeitsrate, sonst gilt Verfügbarkeitsrate<Konsumrate. Die Werte in Spalte RESPTIME beschreiben die mittlere Dauer zwischen Datenanforderung und -empfang. Alle Werte die 30 Sekunden überschreiten sind durch Unterstreichung gekennzeichnet. Die Werte in Spalte 'OUTDATETIME' sind die absoluten User-Zeiten, in dem ein Teilvektor bei einem Datenkonsumenten in nicht-aktueller Form vorliegt, obwohl eine aktualisierte Fassung angefordert wurde (DDM). Absolute Zeiten, die 75 Prozent der Gesamtzeit (=8600 Sekunden) überschreiten, sind durch Unterstreichung hervorgehoben.

Aus dem Spektrum der Leistungskennzahlen des Kommunikationssystems ist für die numerische Berechnung in *ASYNC* der prozentuale Anteil von Zugriffen auf aktualisierte Daten bedeutsam. Die Messwerte in Spalte DATAREQ zeigen, dass dieser Anteil teilweise gering ist. Andererseits ist es schwierig Handlungsempfehlungen zu formulieren, die auf eine Verbesserung dieser Leistungskennzahl abzielen. Ein Ansatzpunkt hierfür ist die mittlere Dauer zwischen Datenanforderung und -empfang (RESPTIME) und die Länge der Zeitintervalle, in denen die Daten nicht aktualisiert sind (OUTDATETIME). Die unterstrichenen Werte zeigen, dass diese Messwerte mit den Werten in Spalte DATAREQ korrelieren, vgl. auch Tabelle 3.2. Dies ist insofern intuitiv, als dass dass die Wahrscheinlichkeit für den Zugriff auf veraltete Daten

steigt, wenn vermehrt alte Daten vorliegen oder die Aktualisierung länger dauert. Dieser Zusammenhang besteht jedoch nicht zwingend und gilt nicht bei passender Taktung der Datenzugriffe (hier ist Steuerung der Datenzugriffe möglich, im Beispiel aber nicht praktiziert).

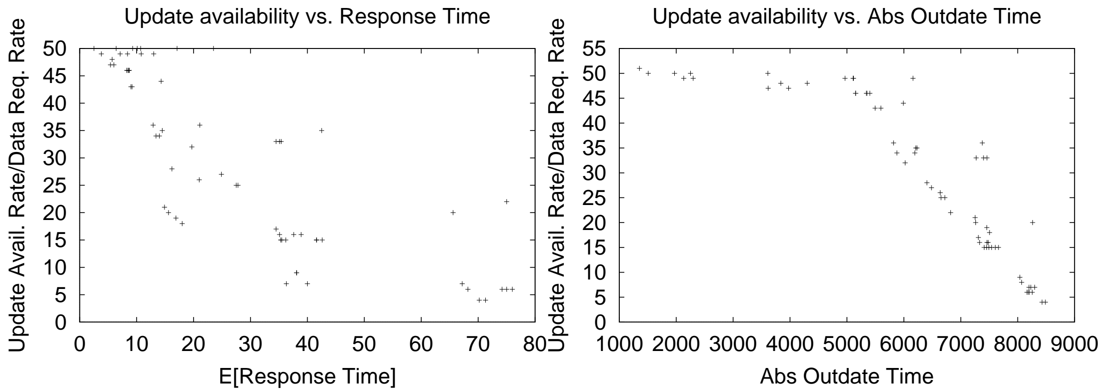


Abbildung 3.2: Links: Verfügbarkeitsrate (MsgRecv) bzgl. Dauer zwischen Datenanforderung und -empfang (RespTime); Rechts: Verfügbarkeitsrate bzgl. Zeitdauer, in denen nicht-aktuelle Daten vorliegen (OutDateTime)

Die Dynamik von Datenzugriff, Datenanforderung und Datenempfang kann als eine **G/G/1/1 Bedienstation** interpretiert werden, bei der Kunden (Datenzugriff) mit einer Ankunftsrate (Konsumrate,DataReq) an einer Station mit Kapazität 1 eintreffen. Ist die Station leer (Daten sind aktualisiert), findet eine Bedienung mit einer spezifischen Bedienrate (Zeit zwischen Datenzugriff und Aktualisierung, RespTime) statt. Ist die Station nicht leer (Daten sind nicht aktualisiert), wird der Kunde abgewiesen (zeitloser Zugriff auf veraltete Daten). Der so erzielte Durchsatz (Verfügbarkeitsrate, MsgRecv) ist kleiner oder gleich der Konsumrate (Ankunftsrate). Die Zeit zwischen Datenzugriffen und die Zeit zwischen Datenzugriff und Aktualisierung wird vermutlich inadäquat durch exponentiell-verteilte Zufallsvariablen beschrieben. Im Beispiel folgt der Datenzugriff einem deterministischen Muster (kein adaptives Verhalten aktiviert) und die Varianz der Zeiten zwischen den Zugriffen ist vermutlich gering (Messung implementierungsbedingt nicht möglich). Die Zeit zwischen Datenzugriff und Aktualisierung setzt sich aus den Kommunikationszeiten und der Reaktionszeit beim Datenproduzenten zusammen. Es wurde nun überprüft, ob die gemessenen Werte des Durchsatzes und der Auslastung durch ein (analytisch leicht handhabbares) M/M/1/1 Modell reproduzierbar sind. Dabei konnte festgestellt werden, dass einige Messwerte überraschend gut reproduzierbar sind, einige aber auch deutlich abweichen, siehe Tabelle 3.6.

Block	λ	μ	D_{me}	D_{MM11}	Fehler	ρ_{me}	ρ_{MM11}
0	0.14372	0.14184	0.07093	0.07139	+0.6%	50	50
1	0.14163	0.02660	0.02674	0.02239	-16.3%	85	84
29	0.36395	0.01425	0.01395	0.01371	-1.8%	98	96
37	0.04907	0.10752	0.02465	0.03369	+36.7%	23	31

Tabelle 3.6: Überprüfung des M/M/1/1 Modells mit Messdaten: λ= (Ankunfts/Konsum)-Rate = $DataReq/8600$; μ= (Bedien/Verfügbarkeits)-Rate = $1/RespTime$; D_{me} = gemessener Durchsatz = $1/(8600 - MsgRecv)$; D_{MM11} = berechneter Durchsatz = $\lambda * (1/(1 + \lambda/\mu))$; ρ_{me} = gemessene Auslastung = $OutDateTime/8600$; ρ_{MM11} = berechnete Auslastung = $1 - (1/(1 + \lambda/\mu))$

Block	P	OutDateTime	MsgRecv	DataReq	RespTime
0	6	4303	610	1236(49)	7.05
1	10	7331	195	1218(16)	37.6
2	10	7662	184	1218(15)	41.6
4	1	5108	474	964(49)	10.8
5	10	7542	177	1218(15)	42.6
7	1	5991	420	964(44)	14.3
8	1	6192,6022	463,305	1408(34),964(32)	13.4, 19.74
11	9	6230,6485	430,260	1236(35),964(27)	14.5, 24.9
12	9	6720	242	964(25)	27.8
13	1	7252	488	2308(21)	14.9
14	1	6405,7457	394,440	1408(28),2308(19)	16.2,16.9
16	2	8262	126	640(20)	65.6
17	4	2678	635	640(99)	4.2
18	3	7379,7481	230,213	640(36),1348(16)	32.1, 35.1
19	2	8486	119	3130(4)	71.3
20	4	8199	122	1858(7)	67.2
21	5	7310,8069	212,212	1236(17),2308(9)	34.5,38.1
22	5	7454,8038	211,211	1408(15),2308(9)	35.3, 38.1
23	5	7399	211	640(33)	35.1
24	5	7270,7494	211,211	640(33),1408(15)	34.5,35.5
25	5	7460	211	640(33)	35.4
26	2	6821	91	422(22)	75.0
27	4	2252	211	422(50)	10.7
28	3	7411,6209	205,146	1348(15),422(35)	36.2, 42.5
29	2	8427	120	3130(4)	70.2
30	4	8206	108	1858(6)	76.0
31	7	5352	619	1348(46)	8.6
32	8	8230,5818	227,451	3130(7),1236(36)	36.3, 12.9
33	8	8163	110	1858(6)	74.2
34	4	3616	667	1408(47)	5.4
35	7	5150	619	1348(46)	8.3
36	6	1355	212	422(50)	6.4
37	6	1971	212	422(50)	9.3
38	10	3610,7460	211,192	422(50),1218(16)	17.1, 38.9
39	6	2132	211	422(50)	10.1
40	10	4966,7606	211,183	422(50),1218(15)	23.5, 41.6
41	8	8254,6160	110,476	1858(6),964(49)	75.0, 13.0
42	7	5402,2298	619,610	1348(46),1218(50)	8.7,3.8
43	9	8298,6638,6652	208,316,242	3130(7),1236(26),964(25)	40.0,21.0,27.5
44	8	8184,7261	120,464	1858(6),2308(20)	68.2,15.6
45	4	3838	669	1408(48)	5.7
47	8	5877,7505	419,418	1236(34),2308(18)	14.0,18.0
49	4	3975	660	1408(47)	6.0
50	7	5155	619	1348(46)	8.3
51	6	5115	610	1236(49)	8.4
52	6	5495	610	1408(43)	9.0
53	6	5596	610	1408(43)	9.2
54	7	5342,1509	619,609	1348(46),1218(50)	8.6, 2.5

Tabelle 3.7: Messungen zur Performance der Kommunikation

Kapitel 4

Fazit

Verteilt implementierte numerische Verfahren (Arbeitspaket 4, Teilprojekt M2) liegen in einer stabilen Implementierung vor und erzielen eine verbesserte Performance im Vergleich zu sequentiellen Implementierungen. Allerdings ist mit zunehmenden Grad der Parallelisierung (> 2 Rechenprozesse) die Effizienz nicht zufriedenstellend, d.h. die erhöhte Rechenleistung wird nicht effizient genug eingesetzt. Das unterliegende mathematische Modell der numerischen Berechnung ist randomisiert. Insofern beziehen sich die Aussagen zur Effizienz nur auf eine von potentiellen vielen Trajektorien/Abläufen des Berechnungsmodells und sind deswegen nicht verallgemeinerbar.
Sequentielle numerische Verfahren, die auf einer baumartigen Datenstruktur des Iterationsvektors arbeiten (Arbeitspaket 4, Teilprojekt M2), erkaufen die Platzersparnis insofern sehr teuer, als dass die Zeiten zur Durchführung eines Iterationsschritts deutlich (um mehrere Größenordnungen) ansteigen. In den durchgeführten Experimenten konnten sie deswegen nicht mit anderen Verfahren konkurrieren. Eine offene Frage ist, welche Modelle aus dem Kontext logistischer Netze geeigneter sind, d.h. eine so platz-effiziente Darstellung des Iterationsvektors aufgrund der ihrer Größe zwingend benötigen und die gleichzeitig eine Dynamik besitzen, die eine weniger zeitlich-aufwendige Verwaltung der Datenstruktur bewirken.

Literaturverzeichnis

[1] Sonderforschungsbereich 559. *Modellierung großer Netze in der Logistik - Finanzierungsantrag 01-05*. Universität Dortmund, 2001.

[2] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with generalized stochastic Petri nets*. Wiley, New York, USA, 1995.

[3] P. Buchholz. Hierarchical structuring of superposed GSPNs. *IEEE Transactions on Software Engineering*, 25(2):166–181, 1999.

[4] P. Buchholz. Structured analysis approaches for large Markov chains. *Applied Numerical Mathematics*, 31(4):375–404, 1999.

[5] P. Buchholz, M. Fischer, P. Kemper, and C. Tepper. New features in the APNN-Toolbox. In P. Kemper (ed.), editor, *Tools at Int. Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems*, Technical Report 760, University of Dortmund, pages 62–68, 2001.

[6] P. Buchholz and P. Kemper. On generating a hierarchy for GSPN analysis. *ACM Performance Evaluation Review*, 26(2):5–14, 1998.

[7] P. Buchholz and P. Kemper. Compact representations of probability distributions in the analysis of superposed GSPNs. In *Proc. 9th Int. Workshop Petri Nets and Performance Models (PNPM'01)*, pages 81–90, 2001.

[8] H. Beilner, J. Mäter, C. Wysocki. The Hierarchical Evaluation Tool HIT. 7th Int. Conference on Modelling Techniques and Tools for Computer Performance Evaluation, 1994.

[9] B. Hendrickson and R. Leland. *The Chaco user's Guide, Version 2.0.* Technical Report SAND94–2692, Sandia Nat. Lab., 1995.

[10] P. Kemper. Numerical analysis based on superposed GSPNs. *IEEE Transactions of Software Engineering*, 22(9):615–628, 1996.

[11] M. Ohlbrecht. Das zustandsänderungsmodell für mtv-systeme. *Interner SFB Bericht*, 2002.

[12] M. Ohlbrecht. Fehlerbäume für ausfallbehaftete ressourcen für den prozess 'bereitstellung' im depot innerhalb eines mehrweg-kreislaufs (exel-tabelle). *Interner SFB Bericht*, 2002.