INSTITUT FÜR
ANGEWANDTE INFORMATIK /
AUTOMATISIERUNGSTECHNIK
UNIVERSITÄT KARLSRUHE (TH)

Studienarbeit

Integration eines Giffler-Thompson-Schedulers in GORBA

Daniel Sonnleithner

Projektleiter: Dr. Karl-Uwe Stucky

INSTITUT FÜR
ANGEWANDTE INFORMATIK /
AUTOMATISIERUNGSTECHNIK
UNIVERSITÄT KARLSRUHE (TH)

Studienarbeit

Aufgabenstellung
für
Herrn cand.mech. Daniel Sonnleithner
Matrikel-Nr.: 11 55 713

Integration eines Giffler-Thompson-Schedulers in GORBA

Die Studienarbeit ist im engen Kontakt mit dem Institut auszuarbeiten und bleibt dessen Eigentum. Eine Einsichtnahme Dritter darf nur nach Rücksprache mit dem Institut erfolgen.

Ausgabedatum:	Abgabedatum:	Unterschriften
Verantwortlicher Hochschullehrer:	Prof. Dr.-Ing. habil. Georg Bretthauer	
Betreuer:	Dr. Michael Kaufmann	
Betreuer:	Dr. Karl-Uwe Stucky	
Betreuer:	Dr. Wilfried Jakob	
Bearbeiter: Anschrift:	cand.mech. Daniel Sonnleithner Ort: Karlsruhe Straße: Ebertstraße 26 Telefon: +49 (0) 179 77 44 555 Email: daniel.sonnleithner@web.de	

INSTITUT FÜR
ANGEWANDTE INFORMATIK /
AUTOMATISIERUNGSTECHNIK
UNIVERSITÄT KARLSRUHE (TH)

Aufgabenstellung

GORBA (Global Optimising Ressource Broker and Allocator) dient der Planung von Jobs in einer inhomogenen Grid-Umgebung. Verplant werden Applicationjobs, die aus Gridjobs bestehen, deren Reihenfolge durch Workflows beschrieben wird. Für jeden Gridjob steht dabei eine Menge an Ressourcen zur Auswahl, wobei diese Menge nicht leer sein darf und in der Regel größer als eins ist. Ressourcen haben einen normierten Leistungsfaktor und verursachen Kosten. Gridjobs kennen eine auf den Leistungsfaktor normierte Arbeitszeit. Das Ziel sind Belegungsplanungen, die in einem vom Anwender vorgegebenen Zeit- und Kostenrahmen durchgeführt werden, eine späteste Fertigstellungszeit einhalten und die Ressourcen möglichst gut auslasten.
Aus den Jobdaten werden nach einfachen Heuristiken Reihenfolgen für die Abarbeitung erzeugt, die dann nach weiteren Heuristiken den Ressourcen zugeordnet werden. Die Ergebnisse dieser heuristischen Planung werden bei Bedarf mit Hilfe eines Evolutionären Algorithmus weiter optimiert.
Die Aufgabe besteht darin, mit Hilfe des Giffler-Thompson-Algorithmus' eine Belegung zu erzeugen und diese durch eine vorhandene Bewertungskomponente bewerten zu lassen. Dazu ist es notwendig für den Giffler-Thompson-Algorithmus Erweiterungen für das Scheduling-Problem von GORBA zu entwerfen, diese zu implementieren und in GORBA zu integrieren. Der erstellte Schedule ist in eine XML-Datei zu überführen. Die zu entwickelnden Module sind in Java zu schreiben. Die Software ist gründlich zu testen.
Nach Entwicklung der Software sind einige Optimierungsszenarien mit vorhandenen Benchmarks durchzuführen. Die Ergebnisse sind mit der bisherigen heuristischen Planung zu vergleichen und die Unterschiede zu analysieren.
Über die durchgeführten Arbeiten ist ein Bericht anzufertigen.

Verantwortlicher Hochschullehrer	:	Prof. Dr.-Ing. habil. Georg Bretthauer
Betreuer	:	Dr. Michael Kaufmann
Betreuer	:	Dr. Karl-Uwe Stucky
Betreuer	:	Dr. Wilfried Jakob

Ich erkläre hiermit, die vorliegende Studienarbeit selbständig verfasst zu haben. Die verwendeten Quellen und Hilfsmittel sind im Text kenntlich gemacht und im Literaturverzeichnis vollständig aufgeführt.

Karlsruhe, 25. Januar 2008

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Im Jahre 1881 wurde in Godalming in der englischen Grafschaft Surrey das erste Elektrizitätswerk in Betrieb genommen. Der erzeugte Strom wurde für eine Lederfabrik und zur Straßenbeleuchtung verwendet. Da Beleuchtung mit Elektrizität zu dieser Zeit im Gegensatz zur Gasbeleuchtung noch teuer und wartungsaufwändig war, schloss das Kraftwerk bereits drei Jahre später.
Ab 1882 installierte die Firma „Siemens & Halske“ - zunächst versuchsweise - unter anderem am Potsdamer Platz in Berlin erste elektrische Straßenbeleuchtungen. Die elektrische Versorgung übernahmen dabei dezentral aufgestellte, dampfbetriebene Kleinkraftwerke, sogenannte „Centralen“. Dabei wurden diese so dimensioniert, dass sie in etwa die benötigte Leistung zur Verfügung stellen konnten. Wenige Jahre später, 1885, nahm in Berlin das erste öffentliche Kraftwerk den Betrieb auf. Dieses konnte nun einen größeren Kundenkreis mit Strom versorgen.
1891 wurde zwischen Lauffen am Neckar und Frankfurt am Main zum erstenmal mit einer Überlandleitung für Strom experimentiert. Aber erst gut 30 Jahre später wurde diese Technik in größerem Umfang eingesetzt, als um 1925 auf der ganzen Welt erste bedeutendere Verbundnetze entstanden. (siehe Paturi, 1989) Diese Verbundnetze wuchsen beständig zum heute bestehenden europäischen Verbundssystem der UCTE (Union for the Co-ordination of Transmission of Electricity) zusammen. Heute denkt niemand mehr darüber nach, woher der Strom kommt, wenn er das Licht anschaltet oder sich die Haare fönt.
Ähnliches lässt sich bei an der geschichtlichen Entwicklung der Computertechnologie beobachten: Nachdem 1941 Konrad Zuse den ersten Computer baute, wurden diese schrittweise verbessert. In den 70er Jahren waren Großrechner die vorherrschende Rechnertechnologie. Spätestens mit Einführung des IBM-PC wurde der Computer für nahezu jedermann erschwinglich.
Anfang der 60er stellte IBM bereits ein System vor, mit dem sich über das Telefon übertragene Daten mit dem Computer weiterverarbeiten ließen. Gleichzeit ließ die US-Luftwaffe das ARPANet (Advanced Research Projects Agency Network) entwickeln, den Vorläufer des heutigen Internets. Stand ist heute, dass es eine Vielzahl von Institutionen gibt, die Hochleistungsrechner betreiben. Diese sind vergleichbar mit den lokal vorhandenen Kraftwerken im vorletzten Jahrhundert. Es gibt zwar mit dem Internet eine Vernetzung der meisten Rechner, jedoch dient diese in erster Linie der Weiterleitung von Informationen. Rechenleistung, Arbeitsspeicher und Langzeitspeicherung wird nur in schwer zugänglicher Art und Weise angeboten.
Eine umfassende Infrakstruktur, vergleichbar dem Stromnetz, gibt es derzeit nicht. Ein Ansatz in diese Richtung ist „Grid Computing“. Ein Computergrid ist laut Foster (2002) eine Infrastruktur, die sowohl Hardware- als auch Softwarekomponenten umfasst. Es erlaubt einen verlässlichen, preiswerten und stabilen Zugang zu Hochleistungscomputerressourcen.
Es soll ein direkten Zugang zu Rechenleistung, Software und den zugehörigen Daten bereitgestllt werden. Diese Ressourcen werden von unterschiedlichen Systemen in unterschiedlichen Organisationen zur Verfügung gestellt. Um dennoch einen einheitlichen Zugang zu ermöglichen, ist eine Grid-Middleware nötig. Diese bildet eine Abstraktionsschicht über den heterogenen Ressourcen.

Das Institut für Angewandte Informatik (IAI) des Forschungszentrum Karlsruhes entwickelt Teile einer solchen Grid-Middleware. Der Schwerpunkt der Entwicklung liegt dabei auf dem *Resourcebroker*, der die vorhandenen Ressourcen verteilt. Dieser soll mittels eines Evolutionären Algorithmus die Verteilung besser als ein *Resourcebroker* mit klassischer heuristischer Planung durchführen. Um Verbesserungen messen zu können, ist ein Standardalgorithmus aus dem Bereich des Job-Shop-Scheduling heranzuziehen. Ein solcher wird in der folgenden Arbeit vorgestellt.

1.1 Aufgabenstellung

Die Verteilung zur Verfügung stehender Gridressourcen an verschiedene Nutzer eines Grids ist eine der Hauptaufgaben, die eine Grid-Middleware erfüllen muss. Diese Aufgabe wird von einem *Resourcebroker* übernommen. In der Grid-Middleware des IAI heißt der *Resourcebroker* GORBA. Er benutzt den Evolutionären Algorithmus „HyGLEAM“, um die Ressourcenverteilung zu optimieren. Um die Ergebnisse dieses neuartigen Algorithmus' bewerten zu können, ist es nötig, einen Standardalgorithmus zum Vergleich heranzuziehen.
Aufgabe ist es, die Arbeitsweise eines solchen Standardalgorithmus, des Algorithmus von Giffler und Thompson, zunächst zu erläutern. Dieser soll schließlich in GORBA implementiert werden, um schlussendlich erste Ergebnisse präsentieren zu können.

1.2 Definitionen

Es werden die im folgenden Abschnitt verwendeten Begriffe beschrieben. (in Anlehnung an Quinte u. a., 2007)

1.2.1 Applicationjob

Anwendungen, die auf dem mit GORBA verwalteten Grid ablaufen, werden als „Applicationjobs“ bezeichnet.
Diese können in einzelne Gridjobs unterteilt werden, die die einzelnen Arbeitsschritte innerhalb einer Anwendung darstellen. Der Kontrollfluss der Anwendung wird dabei von einem Workflow beschrieben.
Die Workflow-Beschreibung erlaubt sowohl sequentielle als auch parallele Ausführung von Gridjobs. Ein Gridjob kann mehrere Vorgänger haben, die zunächst bearbeitet sein müssen, bevor der Gridjob gestartet werden kann. Auch ist erlaubt, einem Gridjob mehrere Nachfolger zuzuordnen. (siehe Halstenberg u. a., 2004).
Dieser Workflow lässt sich als Graph mit folgenden Eigenschaften klassifizieren:

- Derzeit muss der Kontrollfluss bereits im Voraus festgelegt werden. Während der Laufzeit kann keine Änderung vorgenommen werden. Das bedeutet, dass der Graph statisch ist.
- Die Abfolge der einzelnen Gridjobs wird durch die Richtung des Graphen festgelegt.
- Um die Komplexität niedrig zu halten, werden derzeit keine Schleifen erlaubt, so dass der Graph azyklisch ist.

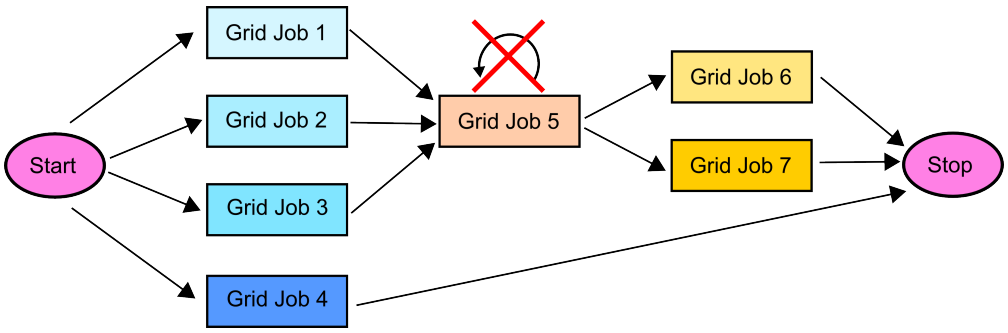

Abbildung 1.1: Applicationjob

Zusammen mit den virtuellen Knoten „Start“ und „Stop“, die keine Gridjobs sind, handelt es sich um einen gerichtet zusammenhängenden, azyklischen Graphen. Im Zusammenhang mit GORBA wird oft die Abkürzung *DAG*, basierend auf der englischen Übersetzung *directed acyclic graph*, verwendet.
Applicationjobs werden mittels der *Grid Application Definition Language (GADL)* (Hoheisel, 2002) definiert. Diese auf XML basierende Sprache wurde vom Fraunhofer Institut für Rechnerarchitektur und Softwaretechnik entwickelt und vom IAI den Anforderungen von GORBA angepasst.

1.2.2 Gridjob

Die Knoten eines Applicationjobgraphens sind Gridjobs. Mit ihnen wird beschrieben, welche Aktionen in einer Anwendung ausgeführt werden. Dazu beinhalten Gridjobs eine Liste von benötigten Ressourcen, die vom Anwender spezifiziert wird. Dabei ist es möglich, alternative, gleichwertige Ressourcen anzugeben.
Durch die Gliederung eines Applicationjob in Gridjobs kann die einem Grid eigene Heterogenität in GORBA abgebildet werden: ein Applicationjob kann sowohl aus Gridjobs bestehen, die auf einer Unix-Maschine ausgeführt werden, als auch aus solchen, die z.B. Windows benötigen. (siehe Halstenberg u. a., 2004)
Gridjobs werden unter Berücksichtigung ihrer vom Applicationjob gegebenen Abhängigkeiten und der benötigten Ressourcen vom Scheduler verplant. Dabei gilt, dass ein Gridjob aus Sicht von GORBA atomar ist, d.h. er kann nicht weiter unterteilt werden.

1.2.3 Ressource

Ressourcen werden von Gridjobs benötigt, um mit ihnen Aufgaben innerhalb eines Applicationjobs zu bearbeiten.
GORBA verwendet für die Beschreibung von Ressourcen die in der GADL (Hoheisel, 2002) definierte *Grid Ressource Definition Language (GResourceDL)*.
In der GResourceDL werden sechs Arten von Ressourcen unterschieden:

- Hardware Ressourcen,

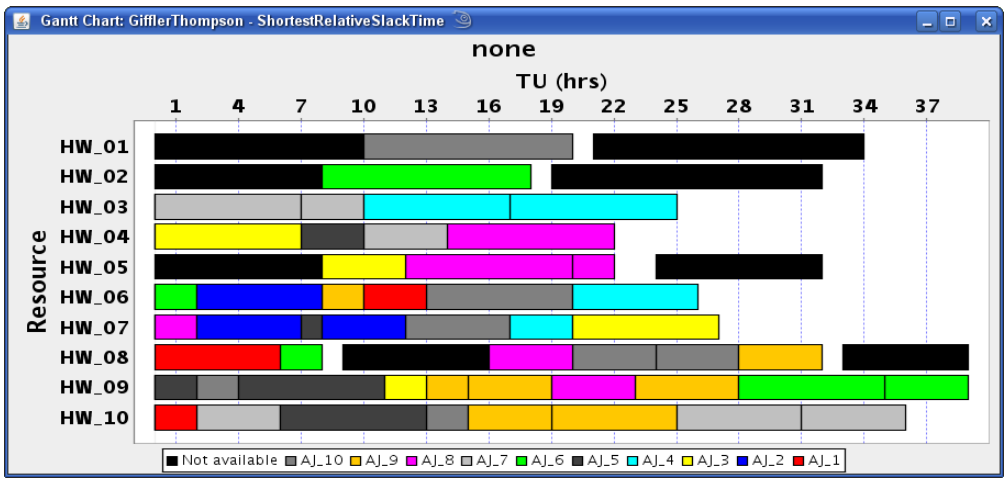

Abbildung 1.2: Gantt-Diagramm eines Schedules

- Hardware-Ressourcenklassen,
- Daten Ressourcen,
- Daten Ressourcenklassen,
- Software Ressourcen und
- Software Ressourcenklassen.

GORBA verwendet derzeit nur Hardware- und Software-Ressourcen, sowie deren Ressourcenklassen.

1.2.4 Schedule

Das Ziel eines Scheduling ist es, einen gültigen Schedule zu erzeugen. Ein Schedule beinhaltet folgende Daten: jeder Gridjob, der in einem der zu verplanenden Applicationjobs enthalten ist, gibt vor, auf welchen Ressourcen er verplant werden kann bzw. verplant werden muss. In einem Schedule wird jeder Gridjob mit den für ihn notwendigen Ressourcen verknüpft. Dabei wird jedem Gridjob eine Start- sowie Endzeit auf der mit ihm verknüpften Ressource zugeteilt. Diese Verknüpfung eines Gridjobs auf einer Ressource wird auch Allokation genannt.
Gültig ist ein Schedule, wenn die Belegungszeiten (je begrenzt durch Start- und Endzeit) verschiedener Gridjobs auf einer Ressource sich nicht überschneiden. Weiter ist darauf zu achten, dass die Reihenfolgeabhängigkeiten zwischen den Gridjobs eines Applicationjobs erhalten bleiben. So darf nie ein Gridjob vor dem Endzeitpunkt eines Vorgängers starten.
Ein Schedule kann als Gantt-Diagramm dargestellt werden. Abbildung 1.2 zeigt einen mit GORBA visualisierten Schedule.

2 Der Giffler-Thompson-Algorithmus

Im Folgenden wird der Giffler-Thompson-Algorithmus beschrieben. Dieser Scheduling-Algorithmus wurde in den späten 50er-Jahren des vergangenen Jahrhunderts für Scheduling-Probleme in der Produktion entwickelt.
Zunächst wird das betrachtete Scheduling-Problem beschrieben; anschließend wird die Funktionsweise des Algorithmus aufgezeigt. Schlussendlich werden die nötigen Anpassungen an das Scheduling-Problem von GORBA angeführt.

2.1 Grundlagen

Um innerhalb der GORBA-Arbeitsgruppe eine einheitliche Schreibweise zu verwenden, wird im Folgenden die Notation an Hahnenkamp (2007) angepasst; die ursprünglichen englischen Bezeichnungen sind in Klammern und in *kursiver* Schrift angegeben.
Der Giffler-Thompson-Algorithmus ist ein klassischer Scheduling-Algorithmus der in Giffler u. Thompson (1960) zum ersten Mal beschrieben wurde. Er wurde für Probleme in der Produktion entwickelt.
Baker (1974) teilt Algorithmen, die Schedules erstellen, in zwei Klassen ein: zum einen gibt es Mechanismen, die einen Schedule in einem Durchlauf erstellen. Dabei kann die Arbeitsweise am besten anhand des Gantt-Diagramms verdeutlicht werden: dieses wird während der Algorithmus arbeitet von links nach rechts „aufgefüllt“. Ist eine Operation einmal auf einer Maschine verplant, ändert sich deren Start- und Endzeit nicht mehr.
Zum anderen beschreibt Baker (1974) Algorithmen, die die Operationen innerhalb des Schedules umsortieren. Während der Algorithmus arbeitet, kann sich dabei sowohl die Start- als auch die Endzeit einer Operation mehrmals ändern.
Der hier beschriebene Giffler-Thompson-Algorithmus gehört zur ersten Algorithmenklasse, die den Schedule in einem Durchlauf erstellen.
Nach Neumann u. Morlock (2002, S. 505) besitzt er einen Rechenaufwand von $O(mn^2)$ wobei m die Zahl der zur Verfügung stehenden Maschinen und n die Anzahl der zu verplandenden Jobs ist.

Der Algorithmus betrachtet Scheduling-Probleme unter folgendem Kontext:

- Es gibt n Jobs $J_i(i = 1, ..., n)$ *(commodity)*, die abgearbeitet werden sollen. Zur Verdeutlichung wird in GORBA von „Applicationjobs“ gesprochen.
- Jeder Job wird auf einer oder mehreren von m Maschinen $M_j(j = 1, ..., m)$ *(facility)* bearbeitet. Die zum Einsatz gelangenden Maschinen sind nicht notwendigerweise unterschiedlich. Maschinen werden im Kontext von GORBA allgemeiner als Ressource bezeichnet.

- Der Arbeitsschritt des i. Jobs auf der j. Maschine wird als Operation O_{ij} bezeichnet. Für jede Operation ist eine endliche Ausführungszeit t_{ij} definiert. Diese Zeit ist unabhängig von der Reihenfolge, in der die Operationen ausgeführt werden.

  Eine Operation, die einmal gestartet ist, kann nicht unterbrochen werden. Jede Operation wird abgeschlossen, bevor ihre Nachfolgeoperation gestartet wird.

  Operationen werden in GORBA als „Gridjobs“ benannt.

- Zwei Operationen A und B können in der Beziehung „B muss nach A ausgeführt werden“ stehen. Weiter gibt es die Beziehung „B kann nach A ausgeführt werden“.

  Gibt es eine solche Beziehung, ist auch eine Verzögerung zwischen den beiden Operationen zu definieren. Im Kontext des Grids könnte diese z.B. wegen Kommunikationszeiten zwischen Ressourcen notwendig sein. Diese werden derzeit bei der Planung in GORBA noch nicht berücksichtigt.

- Der Einfachheit halber wird angenommen, dass die Abfolge von Operationen, die nötig sind einen Job zu bearbeiten, sequentiell ist. Außerdem wird kein Job mehrmals auf derselben Maschine bearbeitet.

Nachdem die Randbedingungen des Scheduling-Problems definiert wurden, werden nun die verschiedenen Schedules als Ergebnis des Giffler-Thompson-Algorithmus beschrieben. Es wird zwischen zulässigen *(feasible)*, aktiven *(active)* und optimalen *(optimal)* Schedules unterschieden.
Zulässige Schedules sind Schedules, die in endlicher Zeit abgearbeitet werden können. Außerdem berücksichtigen sie folgende Aspekte:

- Alle durch den Job vorgegebenen Folgeabhängigkeiten zwischen Operationen sind berücksichtigt.

- Jede Operation bekommt ihre benötigte Bearbeitungszeit zugeteilt.

- Eine vorgegebene Pause zwischen zwei Folgeoperation wird nicht unterschritten.

Diese Vorgaben werden als technologische Anforderungen bezeichnet.
Aktive Schedules sind eine Untermenge der zulässigen Schedules. Sie zeichnen sich durch die Eigenschaft aus, dass auf jeder Maschine alle Zeiten zwischen Folgeoperationen minimal sind. Das bedeutet, keine Operation kann bei vorgegebenem Schedule früher beginnen, ohne die technologischen Anforderungen zu verletzen.
Nach Giffler u. Thompson (1960) sind **optimale** Schedules wiederum aktive Schedules, deren Gesamtlaufzeit minimal ist. [1]
Ziel eines jeden Schedulings ist nun, einen Schedule in vertretbarer Zeit zu erzeugen, der optimal ist, bzw. dem theoretischen Optimum möglichst nahe kommt.

[1] Da Giffler u. Thompson (1960) nur die Laufzeit berücksichtigt, ist der kürzeste Schedule hier der optimale. Optimale Schedules können aber auch diejenigen sein, die die geringsten Kosten verursachen, oder diejenigen, die die höchste Auslastung der Maschinen garantieren. Kurz: der Begriff der Optimalität ist immer an ein Optimalitätskriterium geknüpft, das definiert sein muss.

2.2 Funktionsweise

Im folgenden Abschnitt wird der grundsätzliche Ablauf des Giffler-Thompson Algorithmus beschrieben.
Dem hier beschriebenen Verfahren ist eine Variante des Giffler-Thompson-Algorithmus, die sich in Schwindt (1994) findet, zu Grunde gelegt. Zur Visualisierung ist das Flussdiagramm (Abbildung 2.1) abgedruckt.
Die Abarbeitung erfolgt anhand von zwei, mit R und S bezeichneten, Mengen. R enthält die zu belegenden Maschinen, wohingegen S die als nächstes zu verplanenden Operationen umfasst.
Jede Maschine hat ein Attribut g_j in dem die Startzeitgrenze gespeichert ist. Diese gibt den frühesten Zeitpunkt an, an dem eine Operation auf der Maschine beginnen kann.
Operationen sind neben der Laufzeit t_{ij} durch den (vorläufigen) Startzeitpunkt s_{ij} charakterisiert.

1 Zunächst wird das System initialisiert.

  Die ersten Operationen $O_{iM(O_{i1})}$ eines jeden Jobs J_i werden in die Menge S aufgenommen.

  Von jeder Maschine wird die Startzeitgrenze g_j auf 0 gesetzt.

  Die vorläufigen Startzeitpunkte s_{ij} aller Operationen in S werden ebenfalls auf 0 gesetzt.

2 Die folgenden Schritte werden nun so lange ausgeführt, bis die Menge S leer ist:

  2.1 Die Maschinen M_j, die als nächstes belegt werden sollen, werden ausgewählt. Dazu werden zunächst die Operationen O_{kr} bestimmt, deren aktuell vorläufige Endzeitpunkte am frühesten liegen. Alle Maschinen, auf denen diese ausgewählten Operationen bearbeitet werden sollen, werden in die Menge R aufgenommen. Die Operation O_{kr}, die die Auswahl ausgelöst hat, wird mit der Maschine gespeichert.

  2.2 Die folgenden Schritte werden so lange ausgeführt, bis R leer ist:

    2.2.1 Es wird eine Maschine r aus R ausgewählt. Diese wird aus R entfernt. Der Auswahlmechanismus ist nicht näher bestimmt.

    2.2.2 Alle Operationen aus S, die auf der Maschine r ausgeführt werden sollen, bilden die Menge K. Aus ihr wird eine Operation mittels einer Prioritätsregel ausgewählt.
    Die ausgewählte Operation O_{lr} hat folgende Eigenschaft zu erfüllen:

$$s_{lr} < (s_{kr} + t_{kr})$$

    Diese Eigenschaft ist nötig, da sonst die Operation O_{kr} noch vor der Operation O_{lr} ausgeführt werden könnte. (Neumann u. Morlock, 2002, S.504).
    Welche Prioritätsregel angewandt wird, liegt beim Benutzer. Später werden verschiedene mögliche Regeln vorgestellt.

    2.2.3 Die ausgewählte Operation wird aus S genommen. Damit gilt sie als verplant. Der vorläufige Startzeitpunkt s_{ij} wird somit endgültig. Die vorläufigen Startzeitpunkte derjenigen Operationen, die auch auf der Maschine r ausgeführt werden sollen, verschieben sich um die Laufzeit t_{ij} der ausgewählten Operation nach hinten. Auch auf die Startzeitgrenze g_r, der gerade betrachteten Maschine r, wird derselbe Wert addiert.

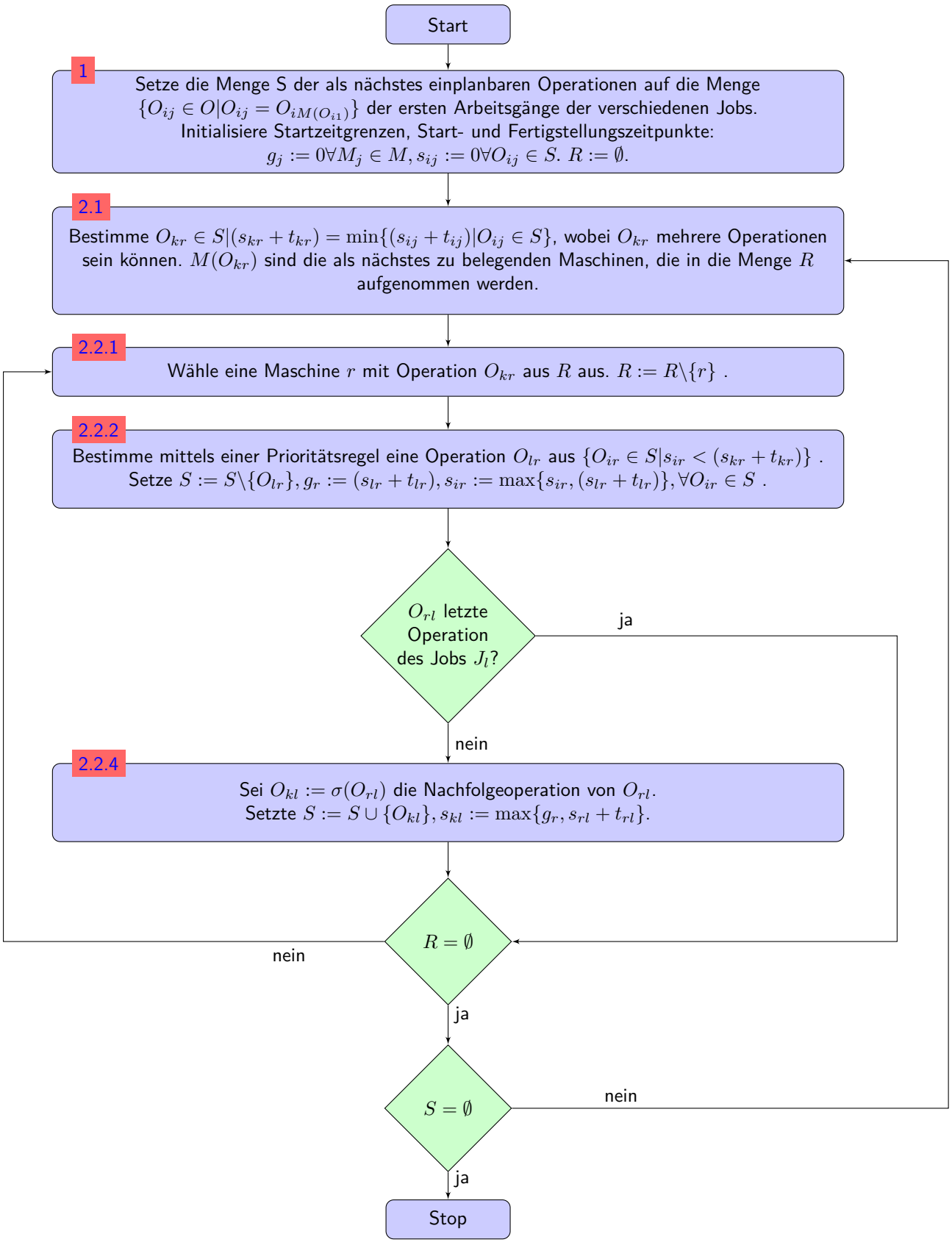

Abbildung 2.1: Flussdiagramm einer Variante des Verfahrens von Giffler und Thompson (aus Schwindt, 1994, S. 20, (Bezeichnungen angepasst))

2.2.4 Hat die ausgewählte Operation eine Nachfolgeoperation, so wird diese in S aufgenommen.

Die entscheidenden Arbeitsschritte dieses Algorithmus sind das Auswählen der als nächstes zu belegenden Maschine (Schritt 2.1) in Kombination mit der Auswahl der Operation, die schlussendlich verplant wird (Schritt 2.2.2). Durch die Wahl der Maschine nach dem frühesten Endzeitpunkt wird gewährleistet, dass alle Maschinen gleichmäßig Operationen zugeteilt bekommen. Da aber nicht zwangsläufig diejenige Operation mit dem frühesten Endzeitpunkt auch verplant wird, ermöglichen die Prioritätsregeln eine Gewichtung anderer Faktoren.
Welche Faktoren berücksichtigt werden können, zeigt der nächste Abschnitt.

2.3 Prioritätsregeln

In Schritt 2.2.2, des oben beschriebenen Alogrithmus', ist aus der Menge K eine Operation auszuwählen, die dann verplant wird. Die Menge K ist eine Teilmenge von S mit der Eigenschaft, dass alle in ihr enthaltenen Operationen auf derjenigen Maschine bearbeitet werden sollen, die in Schritt 2.1 ausgewählt wurde. Um eine Auswahl aus K zu treffen, werden Prioritätsregeln verwendet. Mittels dieser Regeln lassen sich bei gleichen Eingangsdaten unterschiedliche Schedules erzeugen.
In Schwindt (1994) wurden die im Folgenden genannten Prioritätsregeln auf ihre Leistungsfähigkeit bezüglich der Zielkriterien „Zykluszeit“, „mittlere Durchlaufzeit“, „mittlere Belegungszeit“, „maximale Verspätung“, „mittlere Verspätung“, „Anzahl verspäteter Aufträge“ und „Laufzeit“ untersucht.
Derzeit sind die Regeln „Shortest-Processing-Time“ und „Longest-Processing-Time“ aufgrund ihrer Einfachheit in der derzeitigen Implementierung des Giffler-Thompson-Algorithmus enthalten. Die Regel „Shortest-Relative-Slack-Time“wurde wegen ihres guten Abschneidens bei genannter Untersuchung hinsichtlich Verspätungen implementiert. Außerdem wurde die Regel „Most-Work-Remaining“ programmiert, da sie bei Neumann u. Morlock (2002) als Standardregel aufgeführt wird.
Zusätzlich sind die Regeln „Last-in-S“ sowie „Planed-Application-Preferred“ entwickelt und implementiert worden.

2.3.1 Shortest Processing Time

Aus der Menge K wird diejenige Operation O_{lr} ausgewählt, die die kürzeste Ausführungszeit t_{ir} auf der Maschine r beansprucht:

$$O_{lr} \in K | t_{lr} = \min\{t_{ir} | O_{ir} \in K\}$$

2.3.2 Longest Processing Time

Aus der Menge K wird diejenige Operation O_{lr} ausgewählt, die die längste Ausführungszeit t_{ir} auf der Maschine r beansprucht:

$$O_{lr} \in K | t_{lr} = \max\{t_{ir} | O_{ir} \in K\}$$

2.3.3 Most Work Remaining

Aus der Menge K wird diejenige Operation O_{lr} ausgewählt, deren Nachfolgeoperationen in Summe die meiste Restbearbeitungszeit benötigen.

$$O_{lr} \in K | \sum_{\mu \in R_{lr}} t_{l\mu} = \max \left\{ \sum_{\mu \in R_{ir}} t_{i\mu} | O_{ir} \in K \right\}$$

R_{ir} sei dabei die Menge der Maschinen, auf denen J_i in Anschluss an r noch bearbeitet werden muss.

2.3.4 First in S

O_{lr} ist diejenige Operation aus K, die zuerst in S aufgenommen wurde. Das bedeutet, dass diejenige Operation ausgewählt wird, die am längsten in der Menge S vorhanden ist.
Angenommen, alle Operationen würden auf nur einer Maschine verplant werden, würde das bedeuten, dass die Jobs „reihum“ verplant werden. Sind - wie üblich - mehrere Maschinen zu verplanen, werden Jobs nicht mehr streng der Reihe nach alloziert, aber eine Tendenz dazu ist zu erwarten.

2.3.5 Last in S

Aus der Beobachtung, dass in GORBA Schedules besser bewertet werden, bei denen die Operationen (Gridjobs) eines (Application-)Jobs zeitlich gruppiert sind, wurde die Regel „Last-in-S“ entwickelt.
Im Gegensatz zur „First-in-S“-Regel wählt sie diejenige Operation aus K aus, die am kürzesten in der Menge S enthalten ist. Dadurch wird versucht, die Operationen eines Jobs im Schedule „enger beieinander“ zu halten.

2.3.6 Planed Application Preferred

Diese sehr auf GORBA zugeschnittene Prioritätsregel ist eine Erweiterung der „Last-in-S“-Regel. Bevor die Logik aus der „Last-in-S“-Regel angewandt wird, wird die Menge K dahingehend gefiltert, dass nur noch Operationen enthalten sind, bei denen die Verplanung ihres Jobs bereits begonnen hat. Dies soll zusätzlich die einzelnen Jobs im Schedule zeitlich enger halten und die parallele Ausführung fördern.

2.3.7 First Come - First Served

Es wird diejenige Operation ausgewählt, deren Vorgängerjob zuerst fertig geworden ist.

$$O_{lr} \in K | (s_{l\pi_{lr}} + t_{l\pi_{lr}}) = \min\{(s_{i\pi_{ir}} + t_{i\pi_{ir}}) | O_{ir} \in K\}$$

π_{ir} beizeichnet dabei die Maschine, auf der Job J_i vor Maschine r ausgeführt worden ist.

2.3.8 Shortest Slack Time

Die Kürzeste-Schlupfzeit-Regel ermittelt die auszuwählende Operation nach der Schlupfzeit eines Jobs. Die Schlupfzeit ist die Zeit, die dem Job bis zum Erreichen seiner Deadline (Liefertermin) d_{ij} bleibt.

$$O_{lr} \in K | d_l - s_{lr} - \sum_{\mu \in R_{lr}} t_{l\mu} = \min \left\{ d_i - s_{ir} - \sum_{\mu \in R_{ir}} t_{i\mu} | O_{ir} \in K \right\}$$

Dabei ist d_i der Liefertermin bzw. der späteste Fertigstellungszeitpunkt für den Job J_i.

2.3.9 Shortest Relative Slack Time

Aus der Shortest-Slack-Time-Regel leitet sich die Kürzeste-Relative-Schlupfzeit-Regel ab:

$$O_{lr} \in K | st_{lr} = \min\{st_{ir} | O_{ir} \in K\}$$

mit

$$st_{ir} := \begin{cases} \frac{d_i - s_{ir} - \sum_{\mu \in R_{ir}} t_{i\mu}}{|R_{ir}|} & , \text{für} \quad d_i - s_{ir} - \sum_{\mu \in R_{ir}} t_{i\mu} > 0 \\ \left(d_i - s_{ir} - \sum_{\mu \in R_{ir}} t_{i\mu} \right) \cdot |R_{ir}| & , \text{sonst} \end{cases}$$

Diese Regel wurde so konzipiert, dass Jobs mit einer großen Anzahl noch durchzuführender Operationen bevorzugt eingeplant werden, um hierdurch Stauungen von Jobs mit großen Schlupfzeiten zu vermeiden, wie sie bei der klassischen Shortest-Slack-Time-Regel zu beobachten sind. (Schwindt, 1994, S.21)

2.3.10 Random

Aus K wird eine Operation O_{lr} zufällig ausgewählt, wobei die Auswahl jeder Operation aus K mit jeweils gleicher Wahrscheinlichkeit erfolgt.

2.4 Erweiterungen für GORBA

Nachdem nun die Funktionsweise des Giffler-Thompson-Algorithmus und die möglichen Prioritätsregeln erläutert wurden, soll nun auf die Eigenheiten des Scheduling-Problems von GORBA eingeangen werden.
Da sich die folgenden Ausführungen speziell auf GORBA beziehen, werden nun die Begriffe aus GORBA verwendet:
Dies sind im Einzelnen:

- Ressource für Maschine

- Applicationjob für Job
- Gridjob für Operation

Die Formelzeichen werden weiter in der oben vereinbarten Systematik verwendet.

2.4.1 Aufbau von Applicationjobs

Operationen haben bei Giffler u. Thompson (1960) maximal eine Vorgänger- bzw. Nachfolgeoperation. Wie bereits im Abschnitt 1.2.1 beschrieben, lässt ein Applicationjob die Parallelität seiner Gridjobs zu. So kann ein Gridjob sowohl mehrere Vorgänger- als auch Nachfolgegridjobs besitzen. Dies ist insbesondere bei der Ermittlung von Nachfolgergridjobs zu beachten (Schritt 2.2.4).

2.4.2 Variable Laufzeit eines Gridjobs

Nach Giffler u. Thompson (1960) ist es notwendig, die Laufzeit eines Gridjobs vorab zu kennen. Da GORBA alternative Ressourcen für jeden Gridjob zulässt, kann die Laufzeit mit den unterschiedlichen Ausführungseinheiten differieren. Um dennoch verschieden Gridjobs miteinander verleichen zu können, wird die Laufzeit auf einer „Standardcpu“ angegeben. Um die effektive Laufzeit eines Gridjobs zu berechnen, ist es notwendig, die normierte Laufzeit über den Benchmark b_j einer Ressource zu skalieren:

$$t_{eff} = \lceil \frac{t_{ij}}{b_j} \rceil$$

2.4.3 Kommunikationszeiten

In einem Grid laufe Gridjob 1 auf Ressource A. Nun bestehe die Möglichkeit, den Nachfolger von Gridjob 1, Gridjob 2, entweder auf Ressource A oder Ressource B zu allozieren. Würde die Ressource B ausgewählt, müssten die Ergebnisse von Gridjob 1, die die Eingangsdaten von Gridjob 2 sind, zuerst von Ressource A nach Ressource B übermittelt werden, damit Gridjob 2 starten könnte. Diese Kommunikation benötigt die Zeit t_{com}. Würde Ressource A ausgewählt, entfiele diese Kommunikationszeit, da die Daten bereits auf der Ressource vorhanden wären. Je nach Länge der Kommunikationszeit, ist es möglicherweise sinnvoller, Gridjob 2 auf Ressource A zur Ausführung zu bringen, da der Start auf Ressource B um t_{com} verzögert ist.
Kommunikationszeiten sind zwar in GORBA implementiert, doch werden sie von den derzeitigen Schedulern, so auch dem hier vorgestellten, nicht berücksichtigt.

2.4.4 Ressourcen

Bei Giffler u. Thompson (1960) wird vorausgesetzt, dass bereits bei der Initialisierung klar ist, auf welcher Maschine eine Operation ausgeführt werden muss. Bei GORBA ist ein Gridjob nicht zwangsläufig an eine Ressource gebunden. Vielmehr können mehrere Ressourcen zur Ausführung benötigt werden (Koallokation). Außerdem kann es zu einer Ressource mehrere alternative Ressourcen geben, auf denen der Gridjob ebenso ausgeführt werden kann.

Bei der Auswahl der als nächstes zu belegenden Ressource (Schritt 2.1), ist dieser Umstand zu beachten. Deswegen wird bei der Implementation nicht nur eine Ressource ausgewählt, sondern immer eine Gruppe an zu belegenden Ressourcen (Ressourcengruppe).

2.4.5 Kosten

Kostenfunktion

Im Artikel von Giffler und Thompson wird der optimale Schedule als der kürzestmögliche beschrieben. Die Intention des Algorithmus' ist also ein Optimum hinsichtlich der Laufzeit zu erreichen. Nicht beachtet wird z.B., dass meist sehr schnelle Maschinen auch höhere Kosten, hinsichtlich Beschaffung bzw. Wartung verursachen. Ein besonders kurzer Schedule ist also meist auch mit hohen Kosten verbunden.
Um die von einem Schedule verursachten Kosten untersuchen zu können, wird bei GORBA jeder Ressource eine Kostenfunktion zugeordnet. Dabei werden zwei Berechnungsmodi unterschieden:

1 Kosten können abhängig von der Dauer der Belegung einer Ressource entstehen. Kosten werden also pro Zeiteinheit (*per Time*) berechnet.

2 Beim anderen Modell werden Kosten pro Nutzung (*per Use*) erhoben. Dieser Modus kann z.B. bei der Abrechnung von Softwarelizenzen herangezogen werden.

Es ergibt sich somit - neben der Zeit - ein weiteres Kritierium, auf das hin optimiert werden kann.

Kostenlimits

Genauso wie bei Applicationjobs Endtermine definiert sind, zu denen der Applicationjob abgearbeitet sein muss, kann den Applicationjobs ein Kostenlimit vorgeschrieben werden. Dieses wird eingehalten, wenn die Summe aller verursachten Kosten kleiner als dieses Kostenlimit ist.
Der Giffler-Thompson-Algorithmus berücksichtigt bei seiner Planung entstehende Kosten in keiner Weise. Dennoch geht bei der hier vorgestellten Implementation dieses Kriterium ein: bei der Auswahl der als nächstes zu belegenden Ressourcengruppe wird bei sonst gleichwertigen Alternativen, diejenige ausgewählt, die die geringeren Kosten verursacht.
Neben dieser untergeordneten Bewertung einer Ressourcengruppe ist es denkbar, eine Prioritätsregel zu entwerfen, die die Kosten bei ihrer Auswahl berücksichtigt: soll eine teure Ressource belegt werden, werden Gridjobs von Applicationjobs, die größere Kosten zulassen, vor Gridjobs von Applicationjobs, deren Budget kleiner ist, bevorzugt.

3 Implementierung

Die folgenden Abschnitte beschreiben die Implementierung des Giffler-Thompson-Schedulers in GORBA.
Zunächst wird eine Übersicht der wichtigsten Klassen gegeben. Im zweiten Teil wird die Verwaltung der Ressourcen näher betrachtet. Deswegen werden die dafür erforderlichen Klassen *GTResource*, *GTResourceClass*, *GTResourceInstance* sowie *TimeLine* im zweiten Teil beschrieben. Eine komplette Übersicht bietet die Tabelle im Abschnitt 3.1.2.
Die gesamte Implenentierung ist in Java Version 6 erfolgt. Bis auf einige *@Override*-Annotations, die im Bedarfsfall auskommentiert werden können, compiliert der Code auch mit der bei GORBA verwendeten Version 5. Es wurden außer den mit Java mitgelieferten Klassenbibliotheken keine externen Bibliotheken benutzt.
Wert gelegt wurde auf eine logische und konsequente Umsetzung des Programmierparadigmas der Objektorientierung. Um möglichst typsicher zu programmieren, wurden, soweit möglich, generische Klassen benutzt.
Prinzipiell wurde der Code relativ separat vom GORBA-Code erstellt. Diese Separierung wird im folgenden Giffler-Thompson-Umgebung genannt. Durch die geringe Verflechtung mussten wenige Anpassungen an GORBA gemacht werden und der Giffler-Thompson-Code ist einfacher zu warten. Ermöglicht wird das durch das Wrappen einiger GORBA-Klassen. Um eine klare Zuordnung der Klassen zu ermöglichen, sind den Namen gewrappter Klassen die Buchstaben „GT“ vorangestellt.

3.1 Klassenübersicht

Es wird eine Übersicht über die wichtigsten verwendeten Klassen mit ihren Attributen und Methoden gegeben. Naheliegenderweise werden nicht alle vorhandenen Methoden erwähnt. Eine vollständige Auflistung bietet die Dokumentation des Codes, die mit *javadoc* erstellt wird.

3.1.1 Package-Struktur

Für die Giffler-Thompson-Umgebung wurden drei Packages dem GORBA-Code hinzugefügt:

- *gorba.gifflerThompson*: Das Hauptpackage, in dem alle Klassen außer Exceptions und Testklassen zusammengefasst sind.
- *gorba.gifflerThompson.exceptions*: Die benötigten Exceptions sind in diesem Package zusammengefasst.
- *gorba.gifflerThompson.test*: In diesem Package sind Testklassen für verschiedene Klassen aus dem Hauptpackage. Auf die Testklassen wird im Folgenden nicht näher eingegangen.

3.1.2 Tabellarische Übersicht aller Klassen

Klassenname	Package gorba.*	Beschrieben in Abschnitt	Kurzbeschreibung
CalculatingCostMode	gifflerThompson	–	Enumeration; gibt an, ob eine Ressource *per Time* oder *per Use* abgerechnet wird.
GifflerThompsonMain	gifflerThompson	3.1.3	Einstiegspunkt der Giffler-Thompson-Umgebung.
GifflerThompson-Schedule	gifflerThompson	3.1.4	Enthält die Giffler-Thompson-Umgebung.
GTApplicationJob	gifflerThompson	3.1.6	Repräsentiert Application-jobs; wrappt die Klasse ApplicationJob aus GORBA.
GTGridJob	gifflerThompson	3.1.5	Repräsentiert Gridjobs; wrappt die Klasse GridJob aus GORBA.
GTResource	gifflerThompson	3.2.2	Abstrakte Klasse zur Beschreibung aller Ressourcen; wrappt die Klasse Resource aus GORBA.
GTResourceClass	gifflerThompson	3.2.4	Implementierung von GTResource. Repräsentiert Ressourcenklassen.
GTResourceInstance	gifflerThompson	3.2.3	Implementierung von GTResource. Repräsentiert alle Instanzen von Ressourcen.
GTResourceType	gifflerThompson	–	Enumeration; gibt den Typ einer Ressourcen an (*CLASS*, *INSTANCE*)
LastInSSSet	gifflerThompson	3.1.8	Implementierung von SSet nach der Last-in-S - Prioritätsregel (siehe Abschnitt 2.3.5)
LongestJobFirstSSet	gifflerThompson	3.1.8	Implementierung von SSet nach der Longest-Processing-Time - Prioritätsregel (siehe Abschnitt 2.3.2)
MostWorkRemaining-SSet	gifflerThompson	3.1.8	Implementierung von SSet nach der Most-Work-Remaining - Prioritätsregel (siehe Abschnitt 2.3.3)

Fortsetzung der Tabelle:

Klassenname	Package gorba.*	Beschrieben in Abschnitt	Kurzbeschreibung
PlanedApplication-PreferrdSSet	gifflerThompson	3.1.8	Implementierung von SSet nach der Planed-Application-Preferred - Prioritätsregel (siehe Abschnitt 2.3.6)
NoGifflerThompson-SchedulingException	gifflerThompson.-exception	3.1.9	Kritische Ausnahme; wird geworfen, wenn kein Giffler-Thompson-Schedule erstellt werden kann.
NotAllocateable-Exception	gifflerThompson.-exception	3.1.9	Ausnahme wird geworfen, wenn ein Gridjob nicht auf einer Ressource alloziert werden kann.
NotAllocatedException	gifflerThompson.-exception	3.1.9	Wird geworfen, wenn ein Gridjob noch nicht alloziert ist.
NotImplemented-Exception	gifflerThompson.-exception	3.1.9	Diese Ausnahme wird geworfen, wenn versucht wird, eine Funktion aufzurufen, die noch nicht implementiert ist.
ResourceGroup	gifflerThompson	3.1.7	Ressourcengruppe, die nötig ist, um einen Gridjob ausführen zu können.
ShortestJobFirstSSet	gifflerThompson	3.1.8	Implementierung von SSet nach der Shortest-Processing-Time - Prioritätsregel (siehe Abschnitt 2.3.1)
ShortestRelative-SlackTimeSSet	gifflerThompson	3.1.8	Implementierung von SSet nach der Shortest-Relative-Slack-Time - Prioritätsregel (siehe Abschnitt 2.3.9)
SortedLinkedList	gifflerThompson	–	Klasse, die eine immer sortierte Liste bereitstellt.
SSet	gifflerThompson	3.1.8	Abstrakte Klasse, die die Menge S im Algorithmus von Giffler und Thompson repräsentiert.
TimeLine	gifflerThompson	3.2.3	Die Timeline verwaltet die verfügbaren Zeitabschnitte in Ressourcen.

3.1.3 GifflerThompsonMain

Die Klasse *GifflerThompsonMain* ist der Einstiegspunkt des Giffler-Thompson-Schedulers.
Der Konstruktor der Klasse benötigt eine Referenz auf das aufrufende GORBA-Objekt. Damit wird die gesamte Giffler-Thompson-Umgebung aufgebaut.
Einzige Methode ist *main*. Diese initialisiert zunächst die Giffler-Thompson-Umgebung, lässt dann den Algorithmus ablaufen und wandelt schließlich die Ergebnisse so um, dass ein Objekt der Klasse *GorbaSchedule*, das die Ergebnisse speichert, GORBA übergeben werden kann.

3.1.4 GifflerThompsonSchedule

Die Klasse *GifflerThompsonSchedule* enthält die gesamte Giffler-Thompson-Umgebung. Diese Umgebung besteht zum einen aus allen zur Verfügung stehenden Ressourcen, die im Attribut *m_resourcePool* gespeichert werden. Zum anderen werden im Attribut *m_applicationJobsToSchedule* alle zu verplanenden Applicationjobs hinterlegt.
Dem Konstruktor wird außer einer Referenz auf das GORBA-Objekt eine Startzeit übergeben. Diese gibt in Zeiteinheiten den Startpunkt der Planung wieder. Ist sie „0“, handelt es sich um einen Montag um 0:00 Uhr.
Nach der Planung kann mit der Methode *transformToGorbaSchedule* die gesamte Umgebung mit ihren Ergebnissen in ein Objekt der Klasse *GorbaSchedule* umgewandelt werden. Dieses Objekt wird dann von GORBA bewertet und als möglicher konventioneller Schedule gespeichert.

3.1.5 GTGridJob

Die Klasse *GTGridJob* hüllt die Klasse *GridJob* aus GORBA ein.
Wichtiges Attribut ist folglich *m_gridJob*, in dem das zu wrappende Objekt gespeichert wird. Neben einer Referenz auf den Applicationjob (*m_applicationJob*), sind alle Vorgänger und Nachfolger in den Attributen *m_prevJobs* und *m_nextJobs* gespeichert. Gibt es keine Vorgänger bzw. Nachfolger, sind diese Sets nicht *null*, sondern existieren, haben aber keinen Eintrag. Ein weiteres Attribut ist der Status des Gridjobs *m_jobStatus*, der entweder *INIT* oder *PLANED* sein kann. Wichtig für die Verplanung ist das Set *m_resourceToUse*, das die notwendigen Ressourcen enthält. Da die Giffler-Thompson-Umgebung derzeit keine Koallokation unterstützt, sollte nur ein Eintrag existieren. Nach der Verplanung wird die verplante Ressourcengruppe im Attribut *m_planedResourceGroup* abgelegt.

Zeitrechnung

In der Giffler-Thompson-Umgebung gibt es bei der Verplanung keine globale Zeitrechnung. Vielmehr beinhaltet jeder Applicationjob eine eigene Zeit, die von Gridjob zu Gridjob „weitergereicht“ wird. Diese Zeit wird im Attribut *m_earliestStartTime* gespeichert.
Die Zeit wird bei den ersten Gridjobs zunächst auf „0“ gesetzt. Wird ein Gridjob verplant, wird anschließend die Methode *getNextJobsToSchedule* aufgerufen.
Diese ruft bei allen Nachfolgern des Gridjobs die Methode *getMaxEndTimeOfPredessors* auf. Zurückgegeben wird die größte Endzeit aller ihrer Vorgänger, sofern alle verplant wurden. Ist dies

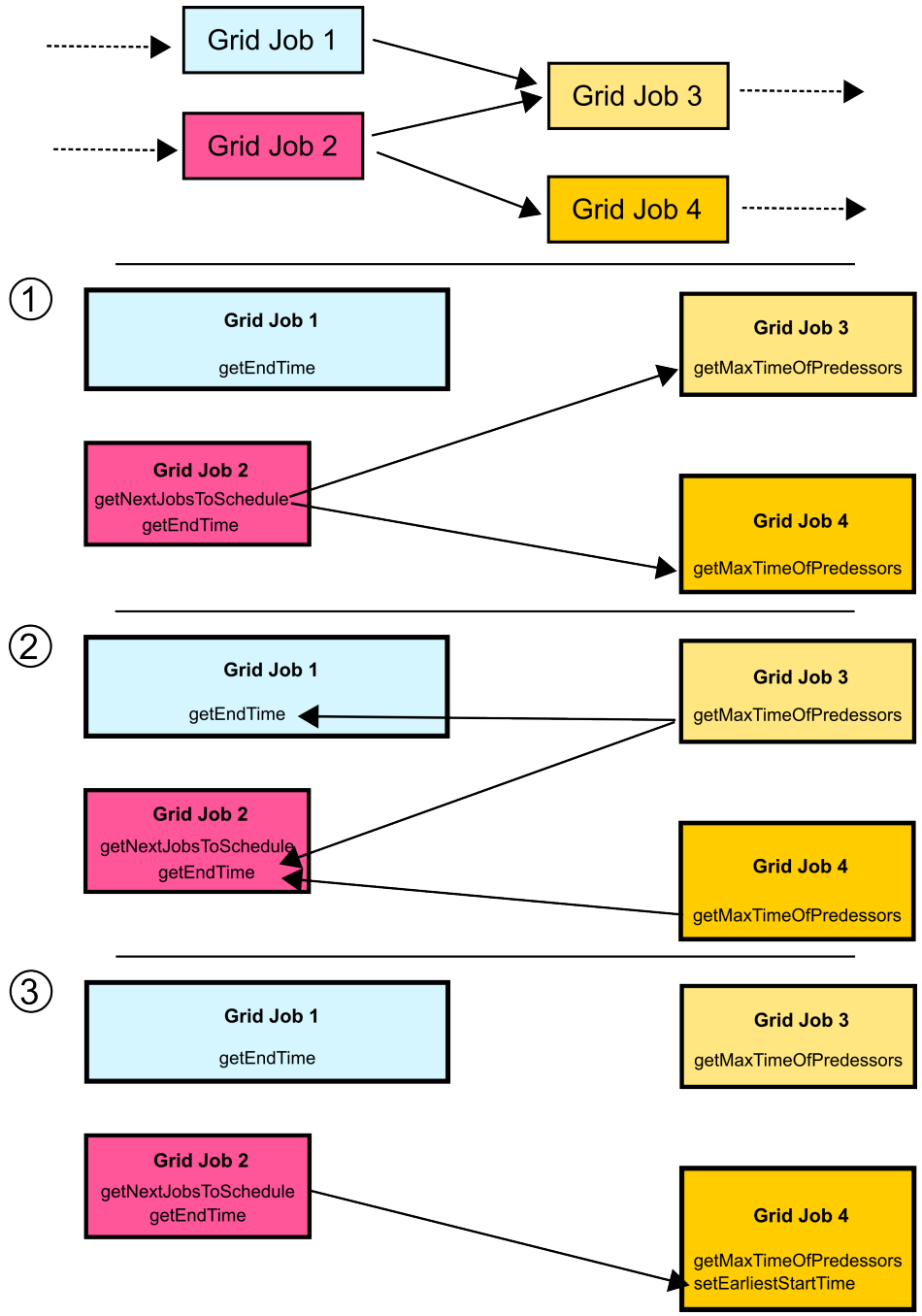

Abbildung 3.1:
oben: Abhängigkeiten zwischen Gridjobs; Ausschnitt aus einem Applicationjob;
1: Gridjob 2 wurde gerade verplant; bei ihm wird die Methode *getNextJobsToSchedule* aufgerufen; diese Methode ruft bei allen Nachfolgern die Methode *getMaxTimeOfPredessors* auf.
2: Die Methode *getMaxTimeOfPredessors* erfragen bei den Vorgängergridjobs die Endzeiten. Ist ein Gridjob noch nicht verplant (wie Gridjob 1) wird eine Exception geworfen, die vom Aufrufer gefangen wird. Daraufhin gibt die Methode *getMaxTimeOfPredessors* „-1“ zurück, unabhängig, ob ein anderer Gridjob schon verplant ist. Da der Vorgänger von Gridjob 4 nur Gridjob 2 ist, wird die Endzeit von Gridjob 2 zurückgegeben.
3: Diese Endzeit wird als früheste Startzeit in Gridjob 4 gesetzt; die Methode *getNextJobsToSchedule* gibt als Ergebnis ein Set mit Gridjob 4 zurück, da dieser nun verplant werden kann.

nicht der Fall, wird „-1“ zurückgegeben.
Bekommt die Methode *getNextJobsToSchedule* einen Zeitwert zurück, setzt sie bei dem zugehörigen Gridjob mittels *setEarliestStartTime* den frühestmöglichen Startzeitpunkt auf den Wert der Endzeit dieses Gridjobs und fügt sie der Rückgabemenge hinzu.
Gibt es entweder keine Nachfolger oder geben alle Nachfolger „-1“ zurück, gibt die Methode ein leeres Set zurück.
Diese Methodik stellt sicher, dass kein Gridjob der Planung zugeführt wird, bevor nicht alle seine Vorgänger verplant sind.

Bestimmen der als nächstes zu belegenden Ressourcengruppe

In Schritt 2.1 des Giffler-Thompson-Algorithmus' werden die als nächstes zu belegenden Maschinen ermittelt. Dazu werden zunächst die Operationen selektiert, deren vorläufige Endzeitpunkte am frühesten liegen. Die Maschinen, auf denen diese Operationen bearbeitet werden sollen, sind dann die als nächstes zu belegenden Maschinen.
Um in der Giffler-Thompson-Umgebung für jeden Gridjob den vorläufig, frühesten Endzeitpunkt zu finden, wird die Methode *getNextAllocatableResourceGroup* verwendet. Diese sucht bei allen benötigten Ressourcen nach dem frühestmöglichen Endzeitpunkt. Da derzeit Koallokation nicht unterstützt wird, beschränkt sich diese Suche auf eine erforderliche Ressourcenklasse. Ausgehend von dieser Ressourcenklasse wird der Ressourcenbaum rekursiv durchsucht.
Kann keine Ressource gefunden werden, auf die der Gridjob alloziert werden könnte, wird eine *NoGifflerThompsonSchedulingException* geworfen.
Als Ergebnis der rekursiven Suche wird ein Objekt der Klasse *ResourceGroup* zurückgegeben; dieses enthält diejenigen Ressourcen, auf denen der Gridjob zur Ausführung mit der frühesten Endzeit gebracht werden könnte.

Verplanen eines Gridjobs

Ist eine *ResourceGroup* zur Belegung ausgewählt, müssen aus der Menge S diejenigen Gridjobs gefiltert werden, die auf dieser Ressourcengruppe ausgeführt werden können. Dazu dient die Methode *isExecutableOn*. Sie überprüft, ob die nötigen Ressourcen in der gegebenen *ResourceGroup* enthalten sind und gibt als Ergebnis *true* oder *false* zurück.
Um aus den ausgewählten Gridjobs dann denjenigen zu ermitteln, der verplant wird, kann sich die Prioritätsregel der Methode *getMaxRemainingWorkTime* bedienen. Diese Methode gibt, ausgehend von diesem Gridjob, die maximal noch verbleibende Restarbeitszeit zurück. Dabei muss beachtet werden, dass bei der Berechnung die normierte Laufzeit herangezogen wird. Die effektive Restlaufzeit kann sich je nach verwendeten Ressourcen verlängern bzw. verkürzen. Zum Vergleich verschiedener Gridjobs in der Menge K ist diese Zeit aber geeignet.
Ist der gesamte Giffler-Thompson-Schedule erzeugt, wird er in ein Objekt der Klasse *GorbaSchedule* überführt. Dazu müssen die Gridjobs in *GorbaScheduleJob*s umgewandelt werden. Diese Aufgabe übernimmt die Methode *transformToGorbaScheduleJob*.

3.1.6 GTApplicationJob

Die Klasse *GTApplicationJob* repräsentiert Applicationjobs in der Giffler-Thompson-Umgebung. Sie umhüllt die Klasse *ApplicationJob* aus GORBA. Das gewrappte Objekt wird im Attribut *m_applicationJob* gespeichert.
Wie im Abschnitt 3.1.5 beschrieben, sind die Gridjobs eines Applicationjobs in einer Kette sich referenzierender Gridjobs verwaltet. Die jeweiligen „Enden“ der Kette sind in den Attributen *m_firstGridJobs* und *m_lastGridJobs* gespeichert.
Die „Gridjobkette“ wird bei der Erzeugung eines *GTApplicationJob*-Objekts mit Hilfe der privaten Methode *generateSuccessors* generiert.
Wird der Applicationjob verplant, werden die Attribute *m_startTime* und *m_endTime* von „-1“ auf den jeweiligen Wert gesetzt. Diese Daten sind für die Umwandlung in ein Objekt der Klasse *GorbaScheduleApplication* nötig. Diese Umwandlung erfolgt in der Methode *transformToGorbaScheduleApplication* und ermöglicht die Erstellung eines *GorbaSchedule*, der an GORBA zurückgegeben wird.

3.1.7 ResourceGroup

Wie in Abschnitt 2.4.4 bereits angedeutet, wird bei GORBA nicht nur eine Maschine, sondern immer eine Gruppe von Ressourcen alloziert. Diese Gruppe besteht sowohl aus Ressourceninstanzen, als auch aus Ressourcenklassen, die die Instanzen verwalten. Selbst wenn nur eine Ressourceninstanz alloziert werden soll, gibt es mindestens eine dazugehörige Ressourcenklasse, die diese Ressourceninstanz verwaltet. Das folgt aus der GORBA-Vorgabe, dass in einem Gridjob immer nur Ressourcenklassen und nicht einzelne Ressourceninstanzen zur Verplanung definiert werden können.
Diese Gruppe von Ressourcen wird von der Klasse *ResourceGroup* verwaltet. Die Ressourceninstanzen in dieser Gruppe werden im Attribut *m_resourceInstances* hinterlegt. Dagegen werden alle übergeordneten Ressourcenklassen im Attribut *m_resourceClasses* gespeichert. Näheres zur Ressourcenverwaltung im Abschnitt 3.2.
Alle Ressourcen einer Ressourcengruppe haben ab mindestens einem Zeitpunkt einen freien Zeitbereich, der mindestens so lange ist, wie die effektive Laufzeit des Gridjobs. Dieser Zeitpunkt wird in *m_startTime* gespeichert. Um die effektive Laufzeit zu berechnen ist einerseits die Speicherung des gemeinsamen Benchmarks dieser Ressourcengruppe in *m_benchmark* nötig. Andererseits wird die normierte Laufzeit im Attribut *m_duration* abgelegt.
Mittels dieser Daten lässt sich mit der Methode *getEndTime* der Endzeitpunkt der Abarbeitung auf dieser Ressourcengruppe berechnen. Im Algorithmus von Giffler und Thompson, ist dieser Zeitpunkt ausschlaggebend für die Auswahl der als nächstes zu belegenden Ressourcengruppe. (Schritt 2.1)

Vergleichen von Ressourcengruppen

Bei Erzeugung der Menge R (Schritt 2.1) werden zunächst alle möglichen Resourcengruppen ausgewählt, die zum gleichen minimalen Endzeitpunkt mit der Bearbeitung fertig sind. Da eine Ressourceninstanz nur einmal pro Belegungszyklus alloziert werden soll, müssen doppelte Einträge

entfernt werden. Dazu wird geprüft, ob zwei Ressourcengruppen gleiche Ressourceninstanzen besitzen. Dies erfolgt mit der Methode *hasSameInstances*. Gibt die Methode *true* zurück, umfasst das gegebene Objekt der Klasse *ResourceGroup* die gleichen Ressourceninstanzen wie die aktuelle Ressourcengruppe.

Unterstützung für Prioritätsregeln

Manche Prioritätsregeln benötigen die aktuelle Laufzeit eines Gridjobs. Dies gilt z.B. für die „Shortest-Relative-Slack-Time"-Regel (Abschnitt 2.3.9). Da es in der Giffler-Thompson-Umgebung keine globale Laufzeit gibt, wird die Bearbeitungsstartzeit eines Gridjobs auf einer Ressource herangezogen. Bevor ein Gridjob alloziert ist, existiert diese Startzeit nicht. Deswegen wird die Methode *calculatePossibleStartTime* herangezogen, die die mögliche Startzeit eines gegebenen Gridjobs auf der Ressourcengruppe berechnet und ausgibt. Kann der gegebene Gridjob nicht auf dieser Ressourcengruppe ausgeführt werden, wird eine *NotAllocateableException* geworfen.

Allokation eines Gridjobs

Steht die zu allozierende Ressourcengruppe und der darauf zu allozierende Gridjob fest, wird die Methode *allocate* aufgerufen. Diese versucht den gegebenen Gridjob zu allozieren. Dazu wird rekursiv auf allen Ressourceninstanzen der Ressourcengruppe mit der effektiven Laufzeit des Gridjobs rekursiv nach einer gemeinsamen Startzeit gesucht.
Ist eine geeignete Startzeit gefunden, wird diese mit der Endzeit der Ressourcengruppe verglichen. Liegt die gefundene Startzeit nach der Endzeit wird eine *NotAllocateableException* geworfen. Diese Abfrage leitet sich von der im Algorithmus von Giffler und Thompson in Schritt 2.2.2 nötigen Bedingung ab.
Kann der Gridjob auf dieser Ressourcengruppe alloziert werden, wird er auf alle Ressourceinstanzen belegt. Dadurch wird der Status des Gridjobs auf *PLANED* gesetzt. Dieser nun verplante Gridjob wird zurückgegeben.
Kann keine gemeinsame Startzeit gefunden werden, wird eine *NotAllocateableException* geworfen.

Vorbereitung zur Koallokation

Die gegenwärtig verfügbare Implementierung des Giffler-Thompson-Algorithmus sieht keine Koallokation vor. Dennoch sind die Klassen so ausgelegt, dass sie zur Implementierung dieser Funktionalität vorbereitet sind
Sollen mehrere Ressourcen gemeinsam alloziert werden, wird für jede parallel zu allozierende Ressourcenklasse ein Objekt der Klasse *ResourceGroup* erstellt. Um die weitere Funktionalität der derzeitigen Implementierung verwenden zu können, müssen diese Ressourcengruppen zu einer gemeinsamen Ressourcengruppe „verschmolzen" werden. Die dafür geschriebene Methode *meltResourceGroups* überprüft zunächst, ob die zu vereinigenden Ressourcengruppen die Voraussetzungen für eine Koallokation erfüllen: es muss die gleiche Startzeit, die gleiche normierte Laufzeit und der gleiche Benchmark verwendet werden. Dann werden die verwendeten Ressourcenklassen und Ressourceninstanzen kopiert. Sind die Voraussetzungen nicht erfüllt, wird eine *IllegalArgumentException* geworfen.

3.1.8 SSet

Die abstrakte Klasse *SSet* repräsentiert die Menge S im Algorithmus von Giffler und Thompson. In dieser Implementierung des Algorithmus' übernimmt sie einen Großteil der Scheduling-Logik. Wichtigstes Attribut der Klasse ist *m_collection*, in dem die Gridjobs während des Scheduling gespeichert werden. Das Attribut ist vom Typ *Collection<GTGridJob>*, wobei *Collection<>* ein generisches Interface ist. Erst durch den Aufruf des Konstruktors einer Implementierung der abstrakten Klasse *SSet* mit einem geeigneten Objekt, wird die Klasse dieses Attributs festgelegt. Dadurch kann bei der Implementierung der abstrakten Klasse auf die Art und Weise der Speicherung der Gridjobs Einfluss genommen werden.

Erzeugen der Menge R

Mit der Methode *getNextResourceGroupsToAllocate* wird die Menge R des Giffler-Thompson-Algorithmus' erzeugt. Dazu werden von allen Gridjobs in der Menge S die Ressourcengruppen gesammelt, auf denen diese Gridjobs ausgeführt werden sollen. Von diesen Ressourcengruppen werden nur diejenigen gespeichert, die mit der Abarbeitung des Gridjobs am frühesten fertig sind. Da jede Ressourceninstanz nur einmal pro Zyklus belegt werden soll, wird die Menge der Ressourcengruppen nochmals gefiltert: von allen Ressourcengruppen, die dieselben Ressourceninstanzen verwenden, werden die Dubletten gelöscht.

Allozieren von Gridjobs

Zur Allokation eines Gridjobs wird die Methode *allocate* ausgeführt:
Ist klar, welche Ressourcengruppe belegt werden soll, wird nun die Menge K erzeugt. Diese beinhaltet alle Gridjobs, die auf der Ressourcengruppe, die belegt werden soll, ausgeführt werden können. Aus der Menge K wird mittels Prioritätsregel ein Gridjob ausgewählt. Dieser wird auf der Ressourcengruppe alloziert. Dann werden alle allozierbaren Nachfolgegridjobs der Menge S zugeführt.

Implementierung von Prioritätsregeln

Die Klasse ist als abstrakte Klasse ausgeführt, um die in Schritt 2.2.2 erwähnten Prioritätsregeln implementieren zu können. Um eine neue Prioritätsregel zu implementieren, sind zwei Schritte nötig:

1 Die Unterklasse benötigt einen Konstruktor, der den Konstruktor der Klasse *SSet* aufruft. Dem Konstruktor der Oberklasse wird eine Beschreibung der verwendeten Prioritätsregel und ein Logger-Objekt übergeben. Außerdem wird ihm ein Objekt einer Unterklassen von *Collection<>* übergeben. In dem Objekt der Unterklasse *Collection<>* werden die Gridjobs der Menge S hinterlegt.

2 Der Methodenrumpf *chooseGridJob*, den die abstrakte Klasse *SSet* bereitstellt, ist zu implementieren.

Diese Methode wählt aus einer gegebenen Menge K von Gridjobs mittels einer Regel einen Gridjob aus, der alloziert werden soll. In Kapitel 2.3 werden einige mögliche Regeln vorgestellt.

Bei der derzeitigen Implementierung sind die Regeln „Shortest-Processing-Time“ (Abschnitt 2.3.1), „Longest-Processing-Time“ (Abschnitt 2.3.2), „Most-Work-Remaining“ (Abschnitt 2.3.3) und „Shortest-Relative-Slack-Time“ (Abschnitt 2.3.9) enthalten. Für GORBA wurden weiterhin die Regeln „Last-in-S“ (Abschnitt 2.3.5) und „Planed-Application-Preferred“ (Abschnitt 2.3.6) entwickelt und implementiert.
Welche dieser Regeln aktiv ist, wird über den Aufruf der Methode *main* des *GifflerThompsonMain*-Objekts in der Methode *planGTJobAction* im *GorbaJFrame* bestimmt.

3.1.9 Exceptions

In Ausnahmesituationen werden *Exceptions* geworfen. Je nach Situation, können unterschiedliche Ausnahmen auftreten:

- Stellt die Scheduling-Logik fest, dass es nicht mehr möglich ist, einen gültigen Giffler-Thompson-Schedule zu erzeugen, wirft sie eine *NoGifflerThompsonSchedulingException*. Diese kritische Ausnahme führt zum Abbruch des Schedulers.
- Bei der Suche nach geeigneten Ressourcengruppen, wird getestet, ob ein gegebener Gridjob auf einer Ressourceninstanz prinzipiell alloziert werden könnte. Ist dies nicht der Fall wird eine *NotAllocateableException* geworfen. Dies ist so lange unkritisch, so lange es eine alternative Ressourceninstanz gibt, die diesen Gridjob ausführen kann.
- Wird versucht, einen nicht verplanten Gridjob nach seinen Planungsdaten zu fragen, wird eine *NotAllocatedException* geworfen, da er diese Anfrage nicht beantworten kann. Diese Exception sollte in der derzeitigen Implementierung nicht auftreten.
- Wird eine Funktion aufgerufen, die derzeit nicht implementiert ist, wird eine Exception vom Typ *NotImplementedException* geworfen. Dies ist derzeit beim Versuch der Koallokation eines Gridjobs der Fall.

3.2 Organisation der Ressourcen

Im folgenden Abschnitt wird beschrieben, wie in der Giffler-Thompson-Umgebung die zur Verfügung stehenden Ressourcen verwaltet werden. Dies wurde weitgehend in Anlehnung an die Ressourcenverwaltung in GORBA realisiert.
Zunächst wird ein Überblick über die Ressroucenverwaltung gegeben. Die einzelnen Klassen werden dann näher erläutert.

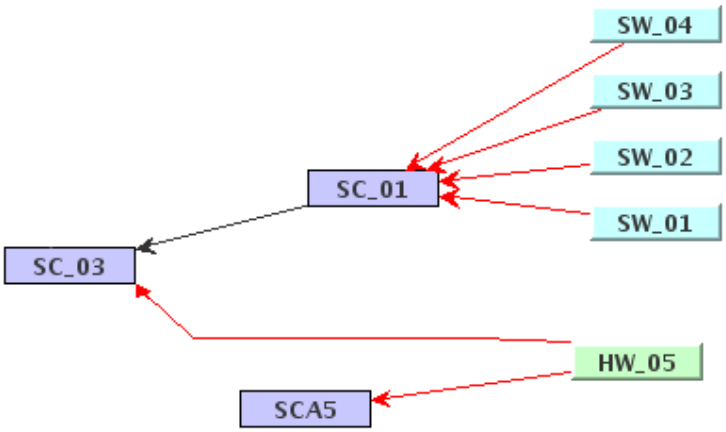

Abbildung 3.2: Ressourcenverwaltung in GORBA

3.2.1 Überblick über die Ressourcenverwaltung

Im Gegensatz zu GORBA, wo zwischen

- Hardware Ressourcen,
- Hardware-Ressourcenklassen,
- Software Ressourcen und
- Software-Ressourcenklassen

unterschieden wird, wird in der Giffler-Thompson-Umgebung nur zwischen Ressourcenklassen (Klasse *GTResourceClass*) und Ressourceninstanzen (Klasse *GTResourceInstance*) unterschieden. Beide implementieren die abstrakte Klasse *GTResource*.
Ressourceninstanzen sind dabei echt vorhandene Ressourcen, auf denen ein Gridjob zur Ausführung gebracht werden kann.
Haben mehrere Ressourceninstanzen gleiche Eigenschaften, so können sie zu einer Ressourcenklasse zusammengefasst werden. Dabei ist von Hoheisel (2002) vorgesehen, dass die Eigenschaften der Klassen an die Instanzen vererbt werden. Diese Funktionalität ist, wie in GORBA auch, zur Zeit nicht implementiert.
Mehrere Ressourceninstanzen stellen also eine Ressourcenklasse zur Verfügung; diese Eigenschaft wird von Hoheisel (2002) mit dem Begriff „provide“ verknüpft.
Neben dem Zusammenfassen einzelner Ressourceninstanzen zu einer Klasse, besteht auch die Möglichkeit, dass eine Ressourcenklasse eine zweite Ressourcenklasse zur Verfügung stellt: so könnte z.B. eine Hardwareklasse „Computer“ die Ressourcenklasse „Linux-Betriebssystem“ zur Verfügung stellen. Auch diese Eigenschaft wird von Hoheisel (2002) unter „provide“ subsummiert.
Diese Zusammenhänge lassen sich in der Abbildung 3.2 erkennen. Rote Pfeile stellen dabei die Beziehnung „provide“ dar. Der schwarze Pfeil steht für die Beziehung „depend“. Letzteres wird im nachfolgenden Abschnitt näher erläutert.

Vorgabe ist, dass einem Gridjob immer nur eine Ressourcenklasse zugewiesen werden kann, niemals also eine konkrete Instanz einer Ressource. Vorteil dieser Vorgehensweise ist, dass während der Planung der Scheduler aus dieser Ressourcenklasse die für seine Planung günstigste konkrete Instanz heraussuchen und allozieren kann. Um einen Gridjob dennoch auf genau einer Ressource zur Ausführung zu bringen, ist es möglich für eine einzelne Ressourceninstanz eine Klasse zur Allokation zu erstellen.

Abhängigkeiten zwischen Ressourcen

Außer der Möglichkeit, mehrere Ressourcen mit gemeinsamen Eigenschaften in einer Ressourcenklasse zusammenzufassen, können Ressourcen andere Ressourcen zu deren Ausführung benötigen. So könnte z.B. eine bestimmte Software auf einer bestimmten Hardware installiert sein. Wird diese Software benötigt, muss auch die zugehörige Hardware vom Scheduler verplant werden. Angelehnt an die Bezeichnung bei Hoheisel (2002), wird diese Eigenschaft in der Giffler-Thompson-Umgebung mittels des Attributs *m_resourceDependsOnResources* in der Klasse *GTResource* bereitgestellt.
Durch diese Abhängigkeit wird eine Koallokation der abhängigen Ressourcen nötig. Die Abhängigkeiten können zwar abgebildet werden, doch ist diese parallele Belegung mehrerer Ressourcen in der derzeitigen Version des Giffler-Thompson-Schedulers nicht vorgesehen.
Dies hat folgenden Grund: Alle an der Koallokation beteiligten Ressourcen müssen ab einem gemeinsamen, festzulegenden Startzeitpunkt für eine gewisse Dauer frei sein. Diese Dauer hängt einerseits von der normierten Laufzeit des zu verplanenden Gridjob ab. Andererseits hängt die Laufzeit von einem Benchmark ab. Im Gegensatz zur Allokation einer Ressource, setzt sich dieser Benchmark nun aus Benchmarks mehrerer Ressourcen zusammen. Derzeit gibt es in der GORBA-Arbeitsgruppe keine Vorgaben, wie dieser Benchmark berechnet wird.
Könnte so ein gemeinsamer Benchmark berechnet werden, ist ein iterativer Prozess nötig, der bei allen beteiligten Ressourcen einen gemeinsamen freien Zeitraum findet, zu dem ein Gridjob auf allen Ressourcen ausgeführt werden kann.
Im Folgendem werden Überlegungen zur Berechnung eines gemeinsamen Benchmarks vorgestellt.

Grundsätzlich kann bei der Koallokation zwischen zwei Fällen unterschieden werden:

1 Eine Ressource wird „echt“ parallel zu einer anderen Ressource alloziert. „Echt“ parallel meint, dass nicht die eine Ressource auf der anderen zur Ausführung gebracht wird, sondern beide wirklich parallel benötigt werden. Als Beispiel sind zwei Knoten eines Computerclusters zu nennen.

   Da alle allozierten Ressourcen auf das Ergebnis der langsamsten Ressource warten müssen, bestimmt diese die Dauer der Allokation. Es wird der kleinste auftretende Benchmark zur Berechnung der effektiven Laufzeit herangezogen.

   Ein solcher Vergleich von Benchmarks ist aber nur sinnvoll, wenn beide Ressourcen von einem Typ sind, sei es Hard- oder Software.

2 Eine Ressource wird „auf“ einer anderen Ressource zur Ausführung gebracht. Denkbar ist

z.B. eine Anwendungssoftware auf einem Betriebssystem, das wiederum auf einem Computer läuft.

Hierbei entsteht eine Abhängigkeit aller verwendeten Ressourcen. Wird z.B. eine schnelle Implementation einer Software auf einer langsamen Hardware ausgeführt, wird die Laufzeit des Gridjobs von beiden Ressourcen beeinflusst. Zur Berechnung der effektiven Laufzeit wird das Produkt der Benchmarks aller beteiligten Ressourcen herangezogen.

Um die Ermittlung eines gemeinsamen Benchmarks zu erleichtern, wird eine Anpassung des von Hoheisel (2002) übernommenen Ressourcenmodells vorgeschlagen. So sollte im angepassten Modell zwischen paralleler Ausführung wie in Fall 1, abhängiger Ausführung (Fall 2) und alternativen Ressourcen (oben) unterschieden werden können. Ein Algorithmus, der aus dem gegebenen Modell ein angepasstes Modell erzeugt, erscheint möglich, ist aber zu entwickeln.

3.2.2 GTResource

Die Klasse *GTResource* ist die Basisklasse zur Beschreibung aller Ressourcen. Das wichtigste Attribut dieser Klasse ist *m_resource*, in dem das umhüllte Objekt der GORBA-Klasse *Resource* gespeichert wird.
Die Klasse stellt die Methode *createDependentAndProvidingResources* zur Erzeugung aller Giffler-Thompson-Ressourcenobjekte bereit. Die Methode wertet die Abhängigkeiten zwischen den Ressourcen aus, die in den *DependencyList*-Objekten der Klasse *Resource* gespeichert sind. Anders als die Ressourcenverwaltung in GORBA, erzeugt die Giffler-Thompson-Umgebung für die *provide*-Abhängigkeit aber Referenzen sowohl von einer Ressourceninstanz zur Ressourcenklasse, als auch von der Ressourcenklasse zur Ressourceninstanz.
In Anlehnung an die Bezeichnungen bei Hoheisel (2002) werden in Ressourcen die bereitgestellten Ressourcen im Attribut *m_resourceProvidesResources* gespeichert.
Unterklassen von *GTResource* müssen die abstrakte Methode *getBestProvidingResourceGroup* implementieren. Diese Methode wird bei der Suche nach der als nächstes zu belegenden Ressourcengruppe aufgerufen und gibt auf die Eingabe von Startzeit und Dauer die am besten passende *ResourceGroup* zurück. Gibt es keine passende Ressourcengruppe wird eine *NotAllocateableException* geworfen.

3.2.3 GTResourceInstance

Die Klasse *GTResourceInstance* repräsentiert alle zur Verfügung stehenden Ressourceinstanzen. Sie erweitert die abstrakte Klasse *GTResource*.

Zeitliche Verfügbarkeit

Bei der Definition von Ressourceinstanzen ist die zeitliche Verfügbarkeit zu berücksichtigen. Dies kann verschiedene Gründe haben:

- Hardware benötigt gewisse Standzeiten, in denen Wartungsarbeiten durchgeführt werden können.

- Bei Software kann es dazu kommen, dass eine Vielzahl an Software-Hardware-Ressourcen zur Verfügung stehen, aber nur eine begrenzte Anzahl an Lizenzen vorhanden ist. Wenn die Lizenzen erschöpft ist, sind die verbleibenden Ressourcen zeitlich nicht verfügbar.

Jede Ressourceinstanz definiert Zeitbereiche (Klasse *TimeSlots* aus dem Package *gorba.timeevent*), in denen sie zu definierten Kosten zur Verfügung steht. Zur Verwaltung steht das Attribut *m_timeLine* der Klasse *TimeLine* zur Verfügung. Wie der Name sagt, kann man sich *TimeLine* als Zeitstrahl vorstellen, auf dem die Zeitbereiche markiert sind, innerhalb deren die Ressourcen verfügbar sind. Technisch handelt es sich bei der Klasse um eine nach den Anfangszeiten sortierte Liste von Zeitbereichen. Die *TimeLine* wird aus der *GorbaKapList* mittels ihrer Methode *createTimeLine* erzeugt:
Die *GorbaKapList* kann Objekte vom Typ *GorbaKapDateWeekly*, *GorbaKapDateDaily* und *GorbaKapDateOnce* enthalten. Ersteres beschreibt Zeitbereiche, die sich wöchentlich wiederholen; *GorbaKapDateDaily* beschreibt Zeitbereiche, die sich täglich wiederholen; letzteres beschreibt schließlich einmalige Zeitbereiche. Dabei werden jeweils die mit dem Zeitbereich verknüpften Kosten gespeichert. Sind keine Objekte enthalten, steht die Ressource jederzeit kostenlos zur Verfügung. Sonst werden die Objekte in der genannten Reihenfolge in die Zeitleiste eingetragen. Sollten sich Zeitbereiche überschneiden, überschreiben später eingestellte Zeitbereiche die schon existierenden.

Kosten, zu denen die Ressource zur Verfügung steht

Die Nutzung einer Ressource ist in der Regel mit Kosten verbunden. Kosten können entweder *per Time* oder *per Use* abgerechnet werden. Im ersten Fall werden Kosten pro Zeiteinheit erhoben. Im zweiten Fall spielt die Nutzungsdauer keine Rolle; vielmehr werden Kosten pro Nutzung berechnet. Die Methode *calculateCosts* der Klasse *GTResourceInstance* berechnet die entstehenden Kosten. Zur Abrechnung nach dem *per Time*-Verfahren werden die Zeitabschnitte aus der *TimeLine* mit ihren verknüpften Kosten herangezogen und diese für die beanspruchte Dauer aufsummiert. Bei der *per Use*-Abrechnung wird das Attribut *m_costsPerUse* ausgelesen und an den Aufrufer zurück gegeben.
Da die Abrechnung *per Use* derzeit in GORBA nicht implementiert ist, konnte dies auch in der Giffler-Thompson-Umgebung nicht realisiert werden. Sobald GORBA diesen Abrechnungsmodus unterstützt, ist der Konstruktor der Klasse *GTResourceInstance* entsprechend anzupassen.

Benchmarks

Außerdem ist es möglich, jeder Ressourceinstanz einen Benchmark zu zuordnen. Je größer dieser ist, desto schneller kann die Ressource eine gegebene Aufgabe lösen. Bei Hardwareressourcen spiegelt dieser Index die Leistungsfähigkeit einer Ressource wieder. Bei Software kann der Wert folgendermaßen interpretiert werden: für eine Aufgabe gibt es verschiedene Implementierungen. Der Index sagt aus, wie schnell die jeweilige Softwarelösung vergleichsweise zu einem Ergebnis kommt.
Mittels dieses Benchmarks und der normierten Laufzeit eines Gridjobs lässt sich die effektive Laufzeit dieses Gridjobs auf einer Ressource berechnen (Siehe Abschnitt 2.4.2).

Allokation einer Ressourceinstanz

Bis ein Gridjob auf einer Ressource alloziert werden kann, sind eine Reihe von Schritten nötig, die in Kapitel 2.2 erläutert werden. Um diese Aufgaben lösen zu können, sind in der Klasse *GTResourceInstance* folgende Methoden implementiert:

getBestProvidingResourceGroup Die Methode *getBestProvidingResourceGroup*, die die abstrakte Methode in *GTResource* überschreibt, wird bei der Suche nach der als nächstes zu belegenden Ressourcengruppe von einer übergeordneten Ressource aufgerufen. Es wird untersucht, ob die Ressourceinstanz ab der gegebenen Startzeit für die gegebene Dauer freie Zeitbereiche hat. Es ist aber nicht zwingend nötig, dass die Ressource ab genau der gegebenen Startzeit frei ist. Wichtig ist, dass es überhaupt der Ressource möglich ist, die Anfrage zu behandeln. Es wird also nach der nächstmöglichen Startzeit bei gegebener Dauer gesucht.
Kann die Anfrage erfüllt werden, wird ein neues Objekt der Klasse *ResourceGroup* instanziert. Diesem wird die aktuelle Ressourceinstanz, die nächstmögliche Startzeit, die gegebene Dauer und der Benchmark der Ressourceinstanz mitgegeben.
Kann für die gegebene Anfrage kein freier Zeitbereich gefunden werden, wird eine *NotAllocateableException* geworfen.

tryToAllocate Die Methode *tryToAllocate* arbeitet ähnlich der zuvor beschriebenen Methode *getBestProvidingResourceGroup*. Im Unterschied zu dieser wird sie aber bei der Allokation von der zu belegenden *ResourceGroup* aufgerufen. Es ist also schon klar, welche Ressourcengruppe belegt werden soll, nur wird nach einer gemeinsamen Startzeit gesucht.
Da derzeit die Giffler-Thompson-Umgebung keine Koallokation unterstützt, ist die Suche nach einer gemeinsamen Startzeit mehrerer Ressourcen eigentlich unnötig, da es nur eine Ressource zu belegen gilt. Um einer Implementierung dieser Funktion vorzugreifen, ist diese Methode im Zusammenspiel mit der Methode *allocate* in der Klasse *ResourceGroup* bereits dafür vorbereitet. Im Gegensatz zur Methode *getBestProvidingResourceGroup* gibt die Methode *tryToAllocate* die der Anfrage nächsten Startzeit zurück. Ein weiterer Unterschied ist, dass in der Anfrage nicht mehr die normierte Laufzeit eines Gridjobs gegeben wird, sondern die effektive.

allocate Wird ein Gridjob schlussendlich auf einer Ressourceninstanz alloziert, wird die Methode *allocate* aufgerufen. Neben dem zu allozierenden Gridjob ist die Startzeit und die effektive Laufzeit gegeben. Aus der *TimeLine*, die die freien Zeitbereiche speichert, wird diese Zeit herausgelöscht. Außerdem wird der Gridjob der Liste *m_planedGridJobs* hinzugefügt. Diese Liste wird derzeit in der Giffler-Thompson-Umgebung nicht wieder ausgelesen. Möglicherweise ist das Speichern der auf dieser Ressourceinstanz auszuführenden Gridjobs für eine spätere Version nützlich.

3.2.4 GTResourceClass

In der Giffler-Thompson-Umgebung wird die Abbildung von Ressourcenklassen durch die Klasse *GTResourceClass* bereitgestellt. Sie implementiert die abstrakte Klasse *GTResource*.
Die Ressourcen, die von dieser Ressourcenklasse zusammenfasst werden, werden im Attribut *m_resourceIsProvidedByResources* abgelegt.

getBestProvidingResourceGroup

Die Methode *getBestProvidingResourceGroup* wird bei der Suche nach der als nächstes zu belegenden Ressourcengruppe von der übergeordneten Ressourcenklasse aufgerufen. Die Startzeit, ab der nach freien Zeitbereichen gesucht werden soll, ist als Eingabeparameter neben der Dauer der zu beanspruchenden Zeit zu übergeben. Diese Dauer entspricht der normierten Laufzeit eines zu verplanenden Gridjobs.
Zunächst wird die Anfrage an alle Ressourcen weitergegeben, die von dieser Ressourcenklasse zusammengefasst werden. Dabei kann entweder ein Objekt der Klasse *ResourceGroup* zurückgegeben werden oder es wird eine *NotAllocateableException* geworfen. Im zweiten Fall wird diese Ausnahme gefangen und solange nach einer Ressource, die die Anfrage erfüllen kann, weitergesucht, bis es keine weiteren Alternativen mehr gibt.
Gibt es keine Ressource, die die Anfrage erfüllen kann, wird eine *NotAllocateableException* geworfen.
Kann eine Ressourcengruppe eine Anfrage erfüllen, bedeutet das, dass alle enthaltenen Ressourcen ab einem gemeinsamen (Start-)Zeitpunkt für die nötige Dauer freie Zeitbereiche enthalten. Die nötige Dauer entspricht der effektiven Laufzeit eines Gridjobs auf dieser Ressourcengruppe; sie errechnet sich aus einem gemeinsamen Benchmark und der normierten Laufzeit des Gridjobs. Aus diesen Daten lässt sich der Endzeitpunkt der Bearbeitung ableiten. Aus den zurückgegebenen Ressourcengruppen, wird die Ressourcengruppe ausgewählt, deren **Endzeitpunkt** am frühesten liegt. Wie in Schritt 2.1 des Giffler-Thompson-Algorithmus beschrieben, ist dieses Kriterium vom Algorithmus vorgegeben. Gibt es mehrere Ressourcengruppen mit dem gleichen Endzeitpunkt, wird diejenige ausgewählt, die bei gegebenen Daten die geringsten Kosten verursacht.
Das aufgerufene Objekt der Klasse *GTResourceClass* wird der ausgewählten Ressourcengruppe hinzugefügt. Diese Ressourcengruppe wird schließlich an den Aufrufer der Methode zurückgegeben.

4 Ergebnis

Im Folgenden wird zunächst beschrieben, wie ein Schedule bewertet werden kann. Dann wird eine erste Analyse des Giffler-Thompson-Algorithmus mit den bestehenden heuristischen Planungen des GORBA-Teams gegeben.

4.1 Benchmarks

Um einen Schedule zu bewerten, wird dieser an das Modul *gorba_kp_bew* übergeben. Die Bewertungsgrößen werden nachfolgend entsprechend der GORBA-Notation als Note bezeichnet. Das Bewertungsmodul gibt zwei Bewertungsgrößen aus, deren Wert jeweils zwischen 0 und 100 000 betragen kann, wobei ein größerer Wert ein besseres Ergebnis wiederspiegelt. Der erste Wert ist die Rohnote, die den Schedule an sich bewertet, ohne vom Nutzer angegebene Beschränkungen hinsichtlich Lieferzeit oder Kosten zu berücksichtigen. Der zweite Wert, die Endnote, basiert auf der Punktzahl der Rohnote, bezieht aber genannte Beschränkungen mit ein. Wird eine Zielvorgaben übertreten, wird die Rohnote eines Schedules mittels eines Faktors verkleinert. Dieser Faktor ergibt sich aus Häufigkeit und Umfang der Überschreitung.

Die Berechnung der Rohnote basiert auf folgenden Kriterien:

- Die Bearbeitungszeiten der verplanten Applicationjobs.
- Die entstehenden Kosten.
- Die Gesamtlaufzeit des gesamten Schedules.
- Die Auslastung der verfügbaren Maschinen. Dabei werden zur Bewertung nur die ersten 75 % der Gesamtlaufzeit herangezogen. Die Auslaufphase, die oft deutlich weniger „dicht“ belegt ist, wird dabei ausgespart.

Weiterhin gibt es ein fünftes Kriterium, das als Hilfskriterium zur Erfüllung der zuvor genannten Kriterien dient: Die Wartezeit der inneren Gridjobs eines Applicationjobs. Innere Gridjobs zeichnen sich dadurch aus, dass sie keine Nachfolger haben. Das Hilfskriterium soll die evolutionäre Suche des nachfolgenden Optimierungsschritts dabei unterstützen, die eigentlichen vier Kriterien zu erfüllen.
Ein Scheduling-Algorithmus wird bewertet, indem er eine Reihe von vorgegebenen beispielhaften Planungssituationen bearbeitet. Diese vom GORBA-Team entwickelten Planungen werden im Folgenden Benchmarks [1] genannt. Benchmarks enthalten jeweils einen Ressourcenpool und eine

[1] Die hier genannten Benchmarks haben nichts mit den Benchmarks einer Ressourceninstanz gemein, die in Kapitel 2.4.2 erläutert werden.

Reihe von Applicationjobs, die es zu verplanen gilt. Um einen Eindruck zu gewinnen, mit welchen Scheduling-Problemen ein Scheduler gut und mit welchen Problemen er weniger gut zurecht kommt, sind die Benchmarks in verschiedene Klassen eingeteilt:

- Die Benchmarks sind nach der Zahl der zu verplanenden Gridjobs klassifiziert. Es gilt Scheduling-Probleme mit 50, 100 und 200 Gridjobs zu bewältigen. Der Benchmarkklasse „200d“ stehen **d**oppelt soviele Ressorcen zur Verfügung wie den anderen Benchmarkklassen.
- Weiter werden Benchmarks hinsichtlich der Beschaffenheit der Applicationjobs eingeteilt:

  Dabei wird unterschieden zwischen Applicationjobs, deren Gridjobs viele (**g**roße **A**bhängigkeiten: gA) bzw. wenige (**k**leine **A**bhängigkeiten: kA) Vorgängerrelationen besitzen.
- Schließlich gibt es die Unterscheidung ob einem Gridjob viele (**g**roße **R**essourcenzahl: gR) oder wenige (**k**leine **R**essourcenzahl: kR) alternative Ressourcen zur Verfügung stehen.

4.2 Auswertung der Benchmarkergebnisse

In Anhang A sind die Benchmarkergebnisse sowohl des Giffler-Thompson-Algorithmus', als auch der vom GORBA-Team entwickelten drei heuristischen Planungsverfahren festgehalten. Die Ressourcenzuordnung erfolgt mit drei weiteren Heuristiken, so dass insgesamt neun Schedules zu bewerten sind. Die Anzahl der Überschreitungen der Vorgaben hinsichtlich Zeit und Kosten bezieht sich auf die beste Qualität dieser neun Schedules. Wie die Tabelle in Anhang A zeigt, liefert die Heuristik „shortestDueTime“ die besten Ergebnisse und damit auch die angegebenen Überschreitungswerte.
Beobachtet werden kann, dass die Giffler-Thompson-Planung bei der Rohnote meist im Mittelfeld der Planungsergebnisse liegt. Aber die vom Nutzer definierten Vorgaben hinsichtlich einzuhaltender Liefertermine bzw. Kosten werden eklatant verfehlt. Anders können die durchweg niedrigen Punktzahlen der Endnote nicht interpretiert werden. Selbst mit der Prioritätsregel „Shortest-Relative-Slack-Time“, die formuliert wurde, um ein Lieferterminüberschreitung zu verhindern, kommt der Giffler-Thompson-Algorithmus selten in die Größenordnung der besten heuristischen Planung des GORBA-Teams.
Eine erste Vermutung, dass diese schlechten Ergebnisse dadurch verursacht werden könnten, dass das Giffler-Thompson-Verfahren die Gridjobs eines Applicationjobs meist weit über den Schedule „verteilt“, konnte nicht bestätigt werden. Die beiden Prioritätsregeln „Last-in-S“ und „Planed-Application-Preferred“, die entwickelt wurden um Gridjobs zeitlich enger zu gruppieren, bewirken - wie die Ergebnisse zeigen - keine Verbesserung.
Dabei muss beachtet werden, dass der Giffler-Thompson-Algorithmus, obwohl für das GORBA-Problem erweitert, nicht im eigentlichen Sinn für das gegebene Problem entwickelt wurde. So sind ihm in seiner ursprünglichen Form alternative Ressourcen sowie parallele Jobausführung fremd. Auch der Kostenaspekt wird von ihm völlig außer Acht gelassen. Letzteres gilt allerdings auch für die bereits in GORBA vorhandenen Heuristiken.
Ziel der Arbeit war ferner, einen Standardalgorithmus aus dem Job-Shop-Planung-Bereich zu implementieren, um eine Referenz zum Vergleich mit den Ergebnissen der vorhandenen Heuristiken

und vor allem den Ergebnissen der nachfolgenden Planung mit dem Evolutionären Algorithmus zu schaffen.

5 Zusammenfassung und Ausblick

Nach einer Einführung in die Sprache von GORBA und in die Begriffe, die Scheduling-Probleme beschreiben, wurde die Funktionsweise eines Standardalgorithmus aus dem Bereich der Job-Shop-Planung erläutert. Im Gegensatz zu anderen Algorithmen sucht sich der Giffler-Thompson-Algortihmus zunächste eine zu belegende Maschine aus, auf die dann eine Operation verplant wird. Die Anpassungen, die zur Bearbeitung des GORBA-Scheduling-Problem mit dem genannten Algorithmus nötig sind, sind vorgestellt worden. Anschließend ist die für GORBA entwickelte Implementierung eingehend dargestellt. Dabei wurde auch Lösungsvorschläge zu derzeit nicht implementierten Funktionen angeführt. Schließlich ist der implementierte Scheduler mittels Benchmarks im Vergleich zur bestehenden heuristischen Planung bewertet worden.
Insgesamt konnte die zum Vergleich mit der evolutionären Planung notwendige Referenz beschrieben und implementiert werden.

5.1 Ausblick

Die Fortsetzung vorliegender Arbeit kann unter Anderem in drei Richtungen erfolgen:

1 Ausgehend von der aktuellen Implementierung können weitere Prioritätsregeln zugeschnitten auf das GORBA-Problem entwickelt werden. Es kann z.B. versucht werden, den Kostenaspekt mit in die Auswahl eines Gridjobs aus K mit einzubeziehen. Die Ergebnisse der Benchmarks, die mit den neuen Regeln erstellt werden, sind zu analysieren. Auch die in Kapitel 2.3 vorgestellten, aber noch nicht implementierten Prioritätsregeln, sind Kandidaten für eine Implementierung und weiterführende Analyse.

2 Aufbauend auf den vorliegenden Benchmarkergebnissen und weiterer Untersuchungen, ist die Arbeitsweise des Giffler-Thompson-Algorithmus im Vergleich zur vom GORBA-Team entwickelten heuristischen Planung näher zu analysieren. Ziel einer solchen Analyse könnte sein, das verhältnismäßig schlechte Abschneiden des Giffler-Thompson-Algorithmus zu erklären.

3 In dieser Arbeit wurde auf mehrere vorbereitete, aber noch nicht implementierte Features der vorliegenden Giffler-Thompson-Implementierung hingewiesen. So fehlt die Einbeziehung von Kommunikationszeit derzeit völlig. Wichtiger erscheint aber, dass eine zukünftige Version eine Koallokation von Ressourcen unterstützt. Über die Vorschrift, wie ein gemeinsamer Benchmark bei einer Koallokation von Ressourcen berechnet werden muss, ist noch zu disskutieren. Erste Ansätze wurden in Kapitel 3.2.1 skizziert.

5.2 Dank

Mein Dank gilt meinen Betreuern Dr. Wilfried Jakob, Dr. Alexander Quinte und Dr. Karl-Uwe Stucky vom Institut für Angewandte Informatik, sowie Dr. Michael Kaufmann vom Institut für Angewandte Informatik/Automatisierungstechnik für die Unterstützung beim Erstellen dieser Arbeit. Meiner Familie möchte ich für die generelle Unterstützung im Studium und meinem Vater für seine nützlichen Korrekturvorschläge danken. Schließlich danke ich Inken Marie, die sich des öfteren meine „Theorien“ und gedanklichen Ergüsse bezüglich verschiedener Algorithmen und Implementierungen antun musste.

Literaturverzeichnis

[Baker 1974] Baker, Kenneth R.: *Introduction to sequencing and scheduling*. Wiley, 1974. – ISBN 0–471–04555–1

[Foster 2002] Foster, Ian: What Is The Grid? - A Three Point Checklist. In: *GRIDtoday* 1 (2002), Nr. 6

[Giffler u. Thompson 1960] Giffler, B. ; Thompson, G.L.: Algorithms for solving production scheduling problems. In: *Operations Research* 8 (1960), S. 487 – 503

[Hahnenkamp 2007] Hahnenkamp, Björn: *Integration anwendungsneutraler lokaler Suchverfahren in den adaptiven memetischen Algorithmus HyGLEAM für die komplexe Reihenfolgeoptimierung*, Universität Karlsruhe (TH), Diplomarbeit, März 2007

[Halstenberg u. a. 2004] Halstenberg, Silke ; Stucky, Karl-Uwe ; Süss, Wolfgang: A Grid Environment for Simulation and Optimization and a First Implementation of a Biomedical Application. In: *Grid Computing and Its Application to Data Analysis*. Postfach 3640, D-76021 Karlsruhe, Germany, 2004

[Hoheisel 2002] Hoheisel, Andreas: Grid Application Definition Language / Fraunhofer-Institut für Rechnerarchitektur und Softwaretechnik. Kekulestraße 7,12489 Berlin, September 2002 (0.2). – Internal Report

[Neumann u. Morlock 2002] Neumann, Klaus ; Morlock, Martin: *Operations Research*. 2. Aufl. Carl Hanser Verlag, 2002. – ISBN 3–446–22140–9

[Paturi 1989] Paturi, Felix R. ; Brocks, Manfred (Hrsg.) ; Matthes, Dr. M. (Hrsg.) ; Uhlmannsiek, Bernd (Hrsg.): *Chronik der Technik*. 3. Aufl. Gütersloh / München : Chronik Verlag, 1989

[Quinte u. a. 2007] Quinte, Alexander ; Jakob, Wilfried ; Stucky, Karl-Uwe ; Süss, Wolfgang: Optimised planning and scheduling of grid resources. In: *GES Baden-Baden*. Hermann-von-Helmholtz-Platz 1, 76344 Eggenstein-Leopoldshafen, Germany, 2007

[Schwindt 1994] Schwindt, Christoph: Vergleichende Beurteilung mehrerer Varianten der Heuristik von Lambrecht und Vanderveken zur Lösung des integrierten Losgrößen- und Ablaufplanungsproblems / Institut für Wirtschaftstheorie und Operations Research, Universität Karlsruhe. 1994 (437). – Report WIOR

A Auswertung der konventionellen Planung

Überschreitungen Zeit/Kosten:
Anzahl der GridJobs (d=doppelte ResAnz):

Endnote

	2/0		2/1		1/1		1/0		2/1		3/1		3/1		3/2		2/1		4/1		2/2		1/3		2/0		4/0		3/2		1/1	
					50						**100**								**200**								**200d**					
Scheduling-Algorithmus	**kRkA**	**kRgA**	**kRkA**	**gRkA**	**gRkA**	**gRgA**			**kRkA**	**kRgA**	**kRgA**	**gRkA**	**gRkA**	**gRgA**	**gRgA**		**kRkA**	**kRgA**	**kRgA**	**gRkA**	**gRkA**	**gRgA**	**gRgA**		**kRkA**	**kRgA**	**kRgA**	**gRkA**	**gRkA**	**gRgA**	**gRgA**	
GifflerThompson - LongestJobFirstSSet	252	2344	500	509					139	1071	169	115					34			22	44				268	1069	147	281				
GifflerThompson - ShortestRelativeSlackTime	1007	45267	479	289					495	2913	139	166					205			117	80				524		324	505				
GifflerThompson - Most Work Remaining	363	2302	209	99					185	203	49	41					25			10	17				143	350	72	131				
GifflerThompson - ShortestJobFirst	547	4371	143	489					132	244	44	49					37			7	20				180	872	67	133				
GifflerThompson - LastInSSSet	101	619	237	124					22	664	54	29					10			17	14				120	287	65	103				
GifflerThompson – PlanedApplPreferred	155	619	200	124					20	664	29	29					22			16	14				157	287	96	103				
shortestDueTime (individual)	14266	20551	10630	13163					13639	9552	12168	7769					18872	20528	19615	19292					28404	20948	21376	7220				
shortestDueTime (globalCheap)	12676	44198	23895	15842					16351	7339	15302	4227					14750	13636	23666	10405					23636	20369	12598	9442				
shortestDueTime (globalFast)	14266	20551	10171	5345					13101	9552	11972	2181					18044	22605	18339	11938					18321	23233	18528	26093				
shortestWorkTimeGridJob (individual)	702	43403	112	453					142	346	51	58					32	193	9	24					193	847	64	132				
shortestWorkTimeGridJob (globalCheap)	566	43179	226	386					182	371	51	47					32	154	10	22					194	762	69	174				
shortestWorkTimeGridJob (globalFast)	702	43403	176	349					139	349	49	53					34	192	14	29					171	526	55	193				
shortestWorkTimeAppJob (individual)	1100	2331	2423	4491					393	349	1071	693					163	371	425	505					648	944	1678	1409				
shortestWorkTimeAppJob (globalCheap)	1089	1956	4613	6578					357	348	1900	574					163	368	586	444					531	812	1552	1623				
shortestWorkTimeAppJob (globalFast)	1100	2331	3568	5179					391	437	1115	508					171	526	416	469					678	1610	1469	1869				

Rohnote

	kRkA	**kRgA**	**gRkA**	**gRgA**	**kRkA**	**kRgA**	**gRkA**	**gRgA**	**kRkA**	**kRgA**	**gRgA**	**kRkA**	**kRgA**	**gRgA**
		50				**100**				**200**			**200d**	
Scheduling-Algorithmus														
GifflerThompson - LongestJobFirstSSet	34089	36815	29768	24153	29535	27444	29888	23678	28009		26294	23718	29896	28135	27059	26514
GifflerThompson - ShortestRelativeSlackTime	31939	45266	28710	22460	30133	28995	29529	22948	28815		28169	24245	30031		29454	26074
GifflerThompson - Most Work Remaining	31992	36164	28643	21283	29858	28310	29680	21832	29261		29323	24101	29837	28831	29853	25922
GifflerThompson - ShortestJobFirst	33954	38730	29969	24505	29393	27915	28732	21688	29533		28854	22211	29808	29148	28649	25509
GifflerThompson - LastInSSSet	31759	24304	27324	22679	28779	26812	29049	20435	28059		26857	22774	28672	26468	28564	24829
GifflerThompson - PlanedApplicationPreferred	32413	24304	27682	22679	27794	26812	27843	20435	27876		25524	22774	28316	26468	28036	24829
shortestDueTime (individual)	38596	44066	41091	38945	36380	38429	40170	36859	36199	39598	38600	37715	39076	40696	39806	36855
shortestDueTime (globalCheap)	37168	44198	40625	39744	36408	36244	38251	35322	35547	38432	37245	36937	38175	40768	38818	38266
shortestDueTime (globalFast)	38596	44066	39320	36575	36164	38429	38788	22567	35819	39411	36739	35007	38370	41165	40492	37875
shortestWorkTimeGridJob (individual)	33795	43402	27916	22776	29156	27314	27987	20929	28995	28158	27810	21184	29629	27956	28293	23404
shortestWorkTimeGridJob (globalCheap)	33954	43179	29923	22288	29429	27208	28004	19514	29041	27894	28026	20540	29651	28450	28371	24019
shortestWorkTimeGridJob (globalFast)	33795	43402	29728	21385	28645	27404	28809	20842	29222	28037	28508	21722	29154	27640	28387	24674
shortestWorkTimeAppJob (individual)	31636	34977	37239	35708	27059	25986	35119	34488	25495	25369	34259	34925	34615	27315	37930	36542
shortestWorkTimeAppJob (globalCheap)	31297	33662	38722	36345	26905	25929	36773	23361	25495	24896	35322	34357	33504	27190	38297	36637
shortestWorkTimeAppJob (globalFast)	31636	34977	38708	23779	26920	26285	35140	23056	25464	26100	34146	33314	34199	27871	38359	37595

rot: kleinster Wert der Spalte

grün: größter Wert der Spalte

Größte und kleinste Werte werden getrennt nach End- und Rohnote ermittelt.
nicht unterstrichene, aber eingefärbte Werte: Rohnote: Unterschied zum Minimum/Maximum < 100
Endnote: Unterschied zum Minimum/Maximum < 10