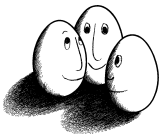
**Diplomarbeit**

# Adaptives Scheduling für verteiltes Data Mining

Maxim Martens



Diplomarbeit
am Fachbereich Informatik
der Universität Dortmund

Dortmund, 14. November 2007

**Betreuer:**

Prof. Dr. Katharina Morik
Dipl.-Inform. Michael Wurst

# Inhaltsverzeichnis

# Tabellenverzeichnis

# Abbildungsverzeichnis

# 1. Einleitung

In der heutigen Zeit schreitet die Technisierung unserer Lebenswelt immer weiter voran. Vor allem das Internet erlebte in den letzten Jahren einen sehr großen Boom. Durch diesen Fortschritt fallen bei unseren Aktivitäten immer mehr Daten an. Oft gewollt, aber auch als Abfallprodukt unserer Tätigkeiten. Egal, ob wir im Internet surfen, einkaufen oder einfach spazieren. Unsere Kommunikation wird geloggt. Beim Einkaufen die Einkaufsliste mit der Bayback/EC-Karte gespeichert. Unsere Bewegungen werden von den zahlreichen Kameras erfasst. Die so entstandenen Datenberge sind erst mal nutzlos. Die daraus gewonnen Informationen können sich aber für manche Interessengruppen als sehr nützlich erweisen. Hier kommt das Data Mining ins Spiel. Mit Hilfe immer Leistungsfähiger Rechner wird versucht aus den Daten verwertbare Informationen zu gewinnen. Diese Datenanalyse ist aber sehr rechenintensiv. Und sie wird durch die wachsende Datenflut immer aufwendiger. Die Anforderung an die Rechenleistung der Maschinen steigt in Folge sehr stark an.

Auf der anderen Seite haben wir die rasante Entwicklung der Rechnervernetzung. Nicht nur das Internet wird immer schneller. Auch die Vernetzung im lokalen Netz hat die Gigabit-Grenze überschritten. Es wird immer vorteilhafter mehrere Rechner zu einem sogenannten „verteiltem System“ zusammen zu schließen. So soll die Leistung eines High-End Rechners erreicht werden, mit nur einem Bruchteil der Kosten. Im Laufe der Zeit haben sich viele Ausprägungen dieser Systeme entwickelt. Dabei entstanden große Unterschiede in der Qualität und Quantität der verwendeten Hardware und der organisatorischen Sicht auf das System. Es entstanden sowohl dedizierte Systeme, als auch Systeme die sich die Rechner mit anderen Nutzern teilen und nur einen Bruchteil der Gesamtleistung zur Verfügung kriegen. Und so hört man immer öfter die Schlagworte wie Grid- und Clustercomputing. Die Vorteile von verteiltem Rechnen scheinen enorm zu sein, und so ist es nahe liegend diese Vorteile bei dem Data Mining zu nutzen.

Bevor man in einer verteilten Umgebung arbeiten kann, müssen zuerst einige Hürden überwunden werden. Erst muss sichergestellt werden, dass die Aufgabe sich parallelisieren lässt, damit sie gleichzeitig auf mehreren Rechnern ausgeführt werden kann. Dann muss der sogenannte Schedulingvorgang statt finden. Dabei werden die einzelnen Teilaufgaben an die verschiedenen Rechner verteilt. Diese Verteilung bestimmt die Effizienz der parallelen Ausführung. Das Finden der optimalen Lösung für das allgemeine Schedulingproblem braucht eine exponentielle Zeit. Damit gehört diese Aufgabe zu den NP-harten Problemen. Im Laufe der Jahre wurden viele Verfahren entwickelt, um eine möglichst gute Approximation für die optimale Lösung zu finden. Dabei stand man aber auch von dem Problem, überhaupt korrekte Daten für die Eingabe zu erhalten. So ging die

Forschung in viele verschiedene Richtungen, die alle unterschiedliche Annahmen über die Rechenumgebung und Eigenschaften der zur verteilenden Aufgabe hatten. So wurden viele Verfahren entwickelt, die nur in bestimmten Spezialfällen eingesetzt werden können. Nebenbei aber auch Algorithmen, die für allgemeine Verwendung geeignet sind.

Das Problem dabei ist, das richtige Verfahren für das Scheduling auszuwählen. Es ist nicht klar welche Anforderungen die Aufgaben des Data Minings an die Schedulingalgorithmen stellen, in welchen Fällen es sinnvoll ist, die Parallelisierung vorzunehmen und welche Art von Rechnerumgebungen für die Ausführung der Aufgaben in Frage kommen. Ist eine dedizierte Umgebung unbedingt nötig? Welche Eigenschaften sollte das verteilte System haben? Können mehrere Nutzer das gleiche System zur selben Zeit in Anspruch nehmen? Was ist mit fremden Benutzern? Können auch externe Systeme zeitgleich auf den gleichen Rechnern eingesetzt werden? Es stellen sich viele Fragen, von denen der Geschwindigkeitsgewinn bei der Ausführung der Data Mining Aufgaben abhängt.

Diese Arbeit wurde mit dem Ziel geschrieben, zumindest einige dieser offenen Fragen zu beantworten. Daher wurden die Ziele wie folgt festgelegt:

1. Erstellung eines verteilten Systems, auf dessen Grundlage die Data Mining Verfahren parallelisiert ausgeführt werden können.

2. Auswahl geeigneter Data Mining Verfahren und Parallelisierung dieser.

3. Auswahl und Implementierung geeigneter Schedulingverfahren und Einsatz dieser in dem oben erstellten System. Evaluierung ihrer Leistung in verschiedenen Einsatzgebieten. Identifizierung von Schwächen und Stärken der einzelnen Algorithmen.

Die Arbeit ist in mehrere Teile gegliedert. Im zweiten Kapitel werden die verteilten Systeme im allgemeinen vorgestellt. Es wird auf die Hard- und die Softwareseite eingegangen und einige Lösungen für Kommunikation und Aufgabenverteilung vorgestellt. Im dritten Kapitel wird auf die Grundlagen des Data Minings eingegangen. Zuerst werden einige Grundlegende Verfahren vorgestellt. Danach einige ausgewählte Verfahren im Detail besprochen. Anschließend wird der Aufbau des RapidMiners vorgestellt. Im vierten Kapitel geht es dann um das Scheduling selber. Nach einigen Definitionen wird auf die einzelnen für diese Arbeit ausgewählten Algorithmen eingegangen und mögliche Ansätze für ihre Implementierung aufgezeigt.

Im zweiten Teil der Arbeit wird das von mir implementierte verteilte System vorgestellt. Danach die ausgewählten Testumgebungen und erstellten Experimente für die Evaluierung der Algorithmen beschrieben und anschließend die Ergebnisse der Tests zusammengefasst.

# 2. Einführung in verteilte Systeme

Dieses Kapitel soll als eine Einführung in die verteilten Systeme dienen. Zuerst werden kurz die Ziele bei der Entwicklung von verteilten Systemen vorgestellt. Danach wird die Hardwareseite beleuchtet und auf die organisatorischen Eigenschaften eingegangen. Zum Schluss werden exemplarisch einige Beispiele von einsatzfähigen Systemen vorgestellt.

Seit der Erfindung der Computer gab es stetige Verbesserungen in diesem Bereich. Die Rechner wurden immer kleiner, kompakter und vor allem leistungsfähiger. Der Fortschritt machte nicht bei den Rechnern selbst halt. Auch die Möglichkeiten die einzelnen Rechner zu vernetzen wurden immer vielfältiger. Als einen großen Durchbruch kann man das Aufkommen von LANs (Local Area Network) sehen, die eine sehr leistungsfähige Verbindung zwischen räumlich nahen Rechnern ermöglichten. Mit der immer besseren Vernetzung entstand auch die Idee, mehrere Maschinen gemeinsam für die Lösung einer Aufgabe einzusetzen. So wurden die „verteilten Systeme“ geboren.

## 2.1. Definition und Ziele

Als verteiltes System bezeichnet man dabei „eine Menge unabhängiger Computer, die dem Benutzer wie ein einzelnes, kohärentes System erscheint.“[31] Dabei handelt jeder Rechner autonom und dieses Handeln ist für den Benutzer unsichtbar. Die Tatsache, dass es um ein verteiltes und nicht um ein lokales System handelt, soll so von dem Benutzer verborgen werden. Dabei spricht man von einer „Transparenz“.[26] Die Eigenschaften des Systems sollten transparent ⇒ durchsichtig ⇒ unsichtbar sein. Die Art der Transparenzen ist vielfältig. Hier sind einige davon:

- **Zugriffstransparenz:** Es soll keinen Unterschied zwischen den Zugriff auf die lokalen und die externen Ressourcen geben. Die Zugriffsschnittstelle soll für beide gleich sein. Zum Beispiel sollte es für beide Datenarten die gleichen Manipulationsmöglichkeiten (speichern, löschen) geben.

- **Ortstransparenz:** Der genaue Ort der benutzten Ressource soll dem Nutzer unbekannt bleiben. Es soll keinen Unterschied machen, auf welchem externen Rechner sich die Daten befinden. So soll es möglich sein, die Daten zwischen den Rechnern zu verschieben, ohne dass es sich für den Nutzer etwas verändert.

- **Leistungstransparenz** Dem Nutzer sollte automatisch die gesamte freie Leistung des Systems zur Verfugung stehen. Er sollte sich nicht selber um die Leistungsanforderung kümmern.

- **Ausfalltransparenz** Der Ausfall eines Rechners im System soll die Ausführung der Aufgabe des Nutzers nicht beeinträchtigen. Das ist eine schwierige Aufgabe und benötigt einige zusätzliche Ressourcen für die Umsetzung. Sogar wenn man die Vorkehrungen wie Replikation der Daten und Backups trifft, kann die Ausfalltransparenz nicht garantiert werden. Wenn der von dem Ausfall betroffene Systemteil hinreihend groß ist, kann die korrekte Ausführung der Nutzeraufgabe nicht gewährleistet werden. In diesem Fall wird ein Fehler in der Ausführung an den Nutzer gemeldet.

- **Sprachtransparenz** Die Architektur und das Betriebssystem der externe Maschinen soll für die Ausführung der Aufgabe keine Rolle spielen. Der Benutzer muss sich darauf verlassen können, dass alle angebotenen Dienstleistungen auch von jedem Teil des Systems unterstützt werden. Zu Problemen kann es zum Beispiel kommen, wenn der Nutzer Texte bearbeiten möchte, die auf einer Windowsmaschine erstellt wurden, aber das Zielsystem das Linux-Betriebssystem verwendet. Dann müssen bestimmte Zeichen im Text angepasst werden. Auch kann es vorkommen, das bestimmte Prozessortypen die Zahlen anders abspeichern (High-Byte, Low-Byte Problem), und so die Daten nicht kompatibel sind.

Als Beispiel für ein verteiltes System kann man eine Distributed Hash Table wie CAN [27] nehmen. Bei einer Hash Table wird für jedes Datum ein Schlüssel generiert und diese Daten sind dann über den Schlüssel eindeutig identifizierbar und abrufbar. Die Funktion ist einer Datenbank sehr ähnlich. Bei der verteilten Version werden die Daten nicht zentral, sondern auf mehreren Rechnern gespeichert. Der Nutzer kriegt aber davon nichts mit. Er kann wie gewohnt die Daten über den eindeutigen Schlüssel abrufen, und muss sich keine Gedanken über den tatsächliche Standort der Daten machen.
Interessanterweise ist das Internet kein verteiltes System, da hier die Daten eindeutig einem Server zugeordnet sind und die Netzwerkverbindungen für den Benutzer sichtbar sind.
Durch die Schaffung von verteilten Systemen hat man vor allem das Ziel verfolgt Nutzer mit Ressourcen zu verbinden. Wenn die lokale Leistung nicht reicht, sollte man so Hilfe von externen Maschinen erhalten. Doch die Auslagerung bringt ein großes Risiko mit sich. Man kann für die Sicherheit der Daten nicht mehr garantieren. Schon bei dem Transport der Daten ist es möglich diese abzuhören. Außerdem könnte der Zielrechner kompromittiert sein, und die übermittelten Daten weiterreichen. Für die Übertragung kann Verschlüsselung benutzt werden, es ist aber sehr schwer festzustellen, ob der Zielrechner auch das tut, was er vorgibt. Diese Sicherheit versucht man mit den Entwicklungen wie TCPA bzw. deren Nachfolger TCG[1] zu erreichen.
Um die Benutzung fremder Maschinen zu ermöglichen, müssen natürlich Schnittstellen definiert werden, um die Portabilität zu gewährleisten. Diese Schnittstellen unterschieden sich ziemlich stark von den normalen Programmschnittstellen. Hier müssen nämlich zwei Sichten auf das System berücksichtigt werden. Die Sicht des Nutzers, der ein vorhandenes System benutzen möchte und eine Schnittstelle für die Anbindung an das

[1] www.trustedcomputinggroup.org

Netz braucht und die Sicht des Programmierers, der das System für seine Applikation anpassen möchte und deswegen eine ganz andere Schnittstelle für diese Erweiterungen benötigt. So entsteht eine mehrschichtige Sicht auf das System.
Diese Schnittstellen werden normalerweise von einer „Middleware“ („Verteilungsplattform“) implementiert, die als Schnittstelle zwischen einer Applikation und dem System darunter dienen soll.

Der Vorteil von einem verteilten System gegenüber einer einzelnen Maschine ist vor allem die Skalierbarkeit. Falls ein Rechner an seine Leistungsgrenze stößt, ist es sehr aufwendig ihn durch einen besseren zu ersetzen. Bei einem verteilten System kann man dagegen meistens einfach mehrere zusätzliche Rechner dazunehmen. Das steigert nicht nur die Leistung, sondern verhindert auch den Fehlschlag der ganzen Anwendung bei einem Rechnerausfall. Der „single Point of Failure“ ist bei einem solchen System nicht mehr gegeben. Dadurch kann die Verfügbarkeit des Gesamtsystems beträchtlich erhöht werden, da einzelne Rechner während des Betriebs gewartet werden können. Allerdings steigt auch der Verwaltungsaufwand stark an. Es ist vor allem schwierig die Daten auf allen Rechnern konsistent zu halten. Auch der Einsatz von heterogenen Rechnern verkompliziert die Sache, da man durch Tests an einem bestimmten Rechner nicht mehr auf das Verhalten der anderen schließen kann.
Bei der Entwicklung oder Anpassung von Programmen an verteilte Umgebungen müssen viele Sachen beachtet werden. Die wichtigste Erkenntnis ist, dass man nur Zugriff auf die lokalen Daten hat. Es gibt daher keine Informationen über das gesamte System. Wenn man zum Beispiel einen Benchmark ausführt, um die Rechnerleistung zu testen, hat dieser gar nichts mit der späteren Leistung zu tun, da die anderen Rechner im Netz überhaupt nicht berücksichtigt werden. Aus diesem Grund müssen alle Entscheidungen, die das Programm trifft, nur auf lokalen Informationen basieren. Dazu gehört auch das nicht vorhanden sein einer globaler Uhr. Man kann nicht mehr einfach sagen „starte den Prozess um Mitternacht“, da der externe Rechner vielleicht in einer anderen Zeitzone steht. Alle Prozesse müssen daher extern synchronisiert werden. Ein wichtiger Punkt für die verteilten Systemen ist die Ausfallsicherheit. Bei einer lokalen Ausführung hat das Versagen des Rechners den sofortigen Abbruch der Applikation zu folge. Bei einem verteilten System ist es nicht der Fall. Normalweise können die Berechnungen auf einen anderen Rechner fortgesetzt werden. Um dieses Verhalten zu gewährleisten müssen die Systeme entsprechend konzipiert werden.

## 2.2. Hardware

Die Hardware für verteilte Systeme kann man in zwei Kategorien einteilen. Die Multiprozessor-Systeme verfügen über einen gemeinsamen Speicher. Bei diesen Systemen kann es sich um Mehrkern-CPUs handeln, oder Rechner mit mehreren Prozessoren. Da bei dieser Konstellation alle Prozessoren gleichzeitig auf den selben Speicherbereich zugreifen können, müssen die Zugriffe synchronisiert werden, um eine Speicherinkonsistenz zu vermeiden. Wobei man hier zwischen den Lese- und den Schreibzugriff unterscheiden

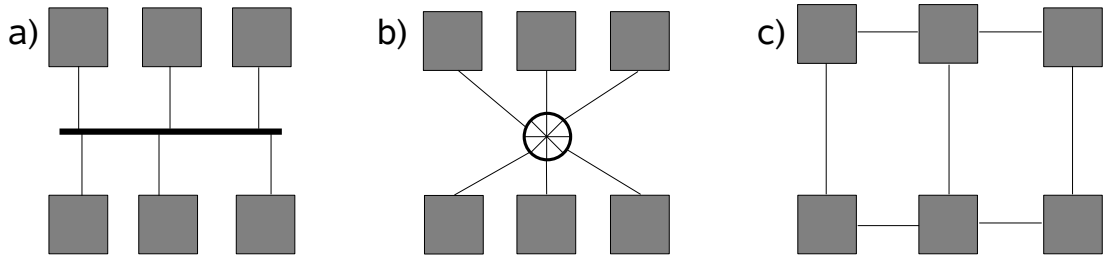



Abbildung 2.1.: Die 3 weit verbreiteten Netzwerktopologien. a) Die Bustopologie - eine Leitung für alle Rechner. b) Der Switch - jede Maschine hat eine eigene Anbindung bis zur Vermittlungsstelle. c) Routing - Der Pfad zwischen den Rechnern kann über andere Rechner führen

muss. Falls alle nur lesend auf die Daten zugreifen, ist die Synchronisation unnötig. Falls aber ein Schreibzugriff erfolgt, muss er exklusiv stattfinden. Alle anderen Aktivitäten auf diesen Daten müssen unterbunden werden. Viel komplizierter wird es, wenn jeder Prozessor einen lokalen Cache zur Verfügung hat. Dann werden die Zugriffe womöglich lokal bedient und nicht an den gemeinsamen Speicher weitergeleitet. So kann es passieren, dass manche CPUs auf veralteten Daten arbeiten. Hier muss ein Benachrichtigungssystem implementiert werden, damit die veränderten Daten im Cache als ungültig markiert werden und bei dem nächsten Lesezugriff verworfen und neu aus dem Speicher eingelesen werden.
Bei den Multicomputer-Systemen hat jeder Rechner einen privaten Speicher. Andere Rechner können nicht ohne einen großen Aufwand auf diesen Speicher zugreifen. Bei diesen Systemen muss normalerweise ein Kommunikationsprotokoll definiert werden, damit die Aufgabenverteilung an einzelne Systeme funktioniert. Sollen die Rechner auf den gleichen Daten arbeiten, kann es auch hier zu Inkonsistenzen kommen. Auch die Schreibzugriffe finden jetzt lokal statt. Es muss also ein Mechanismus geschaffen werden, um alle Teilnehmer von den Veränderungen zu informieren.
Während die Multiprozessor-Systeme normalerweise aus homogenen Einheiten bestehen, ist es bei den Multicomputer-Systemen nicht mehr der Fall. Sehr oft kommt es vor, dass die einzelnen Rechner im Netz sich in der Leistung oder Konfiguration ziemlich stark unterscheiden. Eine wichtige Rolle bei der Leistung spielt auch die Vernetzung der Rechner untereinander. Die Abbildung 2.1 zeigt die 3 häufigen Topologien für die Rechnervernetzung. Bei einer busbasierten Vernetzung hängen alle Computer an der selben Leitung. Dass heißt, es kann jeweils nur ein Gerät zur gleichen Zeit senden und diese Sendung wird von allen Rechnern im Netz empfangen. Jeder Rechner muss dann für ihn bestimmte Nachrichten rausfiltern und den Rest verwerfen. Diese Art der Vernetzung wird zum Beispiel bei den meisten LANs verwendet. Bei verstärkter Kommunikation wird aber das Netz zum Engpass und verhindert die weitere Skalierbarkeit von Anwendungen. Abhilfe schafft nur die Installation eines schnelleren Netzwerks oder der Umstieg auf eine andere Vernetzungsart. Alternativ kann man eine schalterbasierte Vernetzung wählen.

Bei dieser Art der Vernetzung ist jeder Rechner mit einem zentralen Switch verbunden. Es können dadurch gleichzeitig mehrere Rechner senden und empfangen, ohne dass die Kommunikation von den Aktivitäten anderer Rechner beeinträchtigt wird. Das einzige Problem tritt auf, falls mehrere Sender zur gleichen Zeit an den gleichen Empfänger senden wollen. Dann muss einer der Sender die Kommunikation wiederholen, wenn die Leitung wieder frei ist. Diese Variante der Vernetzung ist kostenintensiver, da außer der Leitungen auch der zentrale Switch angeschafft werden muss. Auch hat ein Switch nur eine begrenzte Anzahl an Ports, an die man die Rechner anschließen kann. Sollen noch weitere Rechner angeschlossen werden, muss ein weitere Switch in das System eingebunden werden. So sind die Rechner dann nicht mehr direkt verbunden. Die Verbindung zwischen den Rechnern wird dann durch das „Routing“ erreicht. Bei dieser Variante gibt man den Vorsatz auf, jeden Rechner direkt mit einem anderen verbinden zu wollen. Eine Nachricht wird nicht direkt an den Zielrechner verschickt, sondern durchläuft mehrere Stationen. Nach diesem Prinzip funktioniert das heutige Internet. Da es normalerweise zwischen dem Sender und dem Empfänger mehrere Verbindungsmöglichkeiten gibt, ist der Weg, den eine Nachricht zurücklegt in der Regel nicht fest, und wird für jede Übertragung neu bestimmt. Dabei entscheidet jeder Knoten autonom, in welche Richtung die Weiterleitung erfolgen soll. Natürlich können diese Entscheidungen suboptimal sein. Manche Streckenabschnitte können viel langsamer sein als andere, oder kurzzeitig komplett ausfallen. Dadurch steigt die Zeit, die eine Nachricht braucht, um den Empfänger zu erreichen, denn die Nachricht muss vielleicht sogar mehrmals versandt werden. Dem Nachteil der längeren Latenzzeit zwischen dem Versenden und dem Empfangen einer Nachricht stehen mehrere Vorteile gegenüber. Ein solches Netz ist ziemlich ausfallsicher. Da es mehrere Wege von der Quelle zum Ziel gibt, sorgt der Ausfall von einem Knoten nicht für den Zusammenbruch der Kommunikation. Außerdem ist das Hinzufügen von neuen Rechnern zu diesem Netz sehr einfach.

## 2.3. Cluster und Grids

Wenn die miteinander verbundenen Rechner in einer räumlich lokalen Umgebung stehen, nennt man sie einen Rechnercluster. Normalerweise sind alle Rechner in einem Cluster identisch und haben die gleiche Software installiert. Auf diese Weise ist es leicht möglich die Transparenzkriterien zu erfüllen. Auch befinden sich diese Rechner in dem Administrationsbereich einer Person, so dass die Wartung normalerweise ziemlich einfach ist. Um die Arbeit mit den Rechnerclustern zu erleichtern wurde 1998 das Beowolf System vorgestellt[2]. Dieses System ist eine unixbasierte Clusterlösung. Den Hauptteil stellte ein modifiziertes Linux-Betriebssystem dar, dass Bibliotheken und Erweitrungen enthält, um ohne einen großen Aufwand die verteilte Umgebung nutzen zu können.
Die Ursprungliche Clusterlösung sah eine Vernetzung von identischen Maschinen vor, doch die Einschränkung wurde mit der Zeit aufgehoben. Inzwischen können unterschiedliche Rechner ohne Nachteile miteinander verbunden werden.
Mit der immer weiteren Verbreitung des Internets wurde auch die räumliche Bindung

[2] http://www.beowulf.org/

der einzelnen Maschinen aufgehoben. Man fing an, die einzelnen Cluster miteinander zu verbinden, um so noch mehr Rechnerleistung und eine stärkere Parallelität zu erhalten.
Ende der 90er Jahre wurde von Foster und Kesselman eine neue Sicht auf das Netz vorgestellt. [9] Es kam die Vision auf, die Dienstleistungen eines verteilten Systems genau so einfach beziehen zu können, wie Strom aus der Steckdose. In Ahnlehnung an das Stromnetz (energy grid) wurde die neue Sicht auf das Netz ein „Grid“ getauft. Diese Sicht wurde später noch weiter präzisiert. [10] Die Idee war eine universelle Schnittstelle zu schaffen, so dass jeder Netzteilnehmer entweder seine Dienstleistung anbieten oder eine Dienstleistung in Anspruch nehmen konnte. Dieses System sollte offen sein und so den Menschen auf der ganzen Welt ermöglichen sich daran zu beteiligen. Auch sollte ein gemeinsamer Protokoll geschaffen werden, um mehrere Grids miteinander verbinden zu können.
Die Dienstleistungen dieses Netzes können ganz unterschiedlich sein. Ein Rechengrid könnte für verteilte Berechnungen benutzt werden. In einem Speichergrid könnte man die eigenen Daten bei Bedarf auf externen Rechnern ablegen und müsste so keine freien lokalen Speicherkapazitäten vorhalten. Ein Informationsgrid könnte man wie eine sehr große verteilte Datenbank benutzen und so schnell und einfach auf das Wissen anderer zugreifen.
Leider gab es bei der Ausführung dieser Vision viele Probleme. Da die Maschinen verschiedenen Institutionen (oder Firmen) angehören, werden sie auch von verschiedenen Menschen administriert. So gibt es keine einfache Möglichkeit der Kontrolle. Man müsste den anderen Teilnehmern vertrauen. Man könnte also nichts über die Qualität des Angebots sagen. Es gab auch keine Garantien, dass ein bestimmten Cluster nicht abgeschaltet wird, und so die Daten da drauf verloren gehen. Das ganze System müsste stark redundant ausgelegt sein. Auch für die Sicherheit der Daten bei der Auslagerung und die Korrektheit der Informationen beim Datenbankabruf kann es keine Garantie geben. Des weiteren können bei der Benutzung des Grids Kosten entstehen, die an die Betreiber der einzelnen Cluster abgeführt werden müssten.
Durch die vielen Probleme wurde die ursprüngliche Idee des Grids bis heute nicht realisiert. Der Begriff selber wurde immer weiter verwässert, so dass er inzwischen für fast alle verteilten Anwendungen steht, die räumlich nicht gebunden sind. Es wird zum Beispiel sehr oft das Projekt **SETI@home**[3] als ein Grid-Projekt bezeichnet. Dieses Projekt sollte bei der Suche nach außerirdischem Leben helfen. Allerdings erfüllt dieses Projekt viele Anforderungen des Grids nicht. Es wurde nur eine bestimmte Dienstleistung angeboten, und die Software wurde exakt darauf abgestimmt. Auch gab es nur einen Nutznießer. Es konnte kein weiterer Teilnehmer die Rechenleistung in Anspruch nehmen. Allerdings wurde gerade durch dieses Projekt der Begriff Grid stark geprägt. Inzwischen wurde die Software auch so weit erweitert, dass auch andere Projekte mit dieser ausgeführt werden können.[4] Aber auch mit diesen Erweiterungen ist man noch weit von der ursprünglichen Grid-Idee entfernt.
Um die Definition des Grids zu präzisieren wurde 2006 eine Umfrage bei den Forschern

[3] http://setiathome.berkeley.edu/
[4] http://boinc.berkeley.edu/

und Mitarbeitern von Computerfirmen gestartet. [30] Dabei kamen einige unterschiedliche Ansichten zum Vorschein. Manche konzentrierten sich auf die Hardwarebasis, die anderen auf die organisatorische Sicht und die dahinter liegende Motivation. Allerdings blieb der Konsens bestehen, dass man noch weit davon entfernt ist, die hochgesteckten Ziele des Grids zu realisieren.

## 2.4. Kommunikationsmechanismen

Bei der Kommunikation zwischen den Rechnern können verschiedene Mechanismen eingesetzt werden. Die Kommunikationsmechanismen können sich auf zwei Arten unterschieden. Ein Unterschied ist das Kommunikationsmuster. Bei mitteilungsorientierten Kommunikation wird eine Nachricht an den Empfänger gesendet, aber keine Antwort erwartet. Als Beispiel kann man einen Rechner, der alle interessierten Nutzer in bestimmten Abständen über aktuelle Nachrichten informiert, nennen. Bei der auftragsorientierten Kommunikation wartet der Sender auf eine Reaktion von dem Empfänger. Das kann zum Beispiel ein Mailclient sein, der eine Bestätigung abwartet, dass die versendete E-Mail auch wirklich angekommen ist.

Der zweite Unterschied in der Kommunikation ist die Synchronisation. Bei einer synchronen Kommunikation wartet der Sender so lange, bis die Nachricht bei dem Empfänger angekommen ist. Ein Beispiel für die synchrone Kommunikation ist der RPC (Remote Procedure Call). Hierbei wird eine Funktion auf dem fremden Rechner mit den übergebenen Parametern aufgerufen, und es wird so lange gewartet, bis dieser Rechner die Berechnungen der übergebenen Funktion abgeschlossen hat und ein Ergebnis zurückschickt. Danach wird auf dem lokalem Rechner weiter gearbeitet.

In dem asynchronen Fall ist der Sender nach dem Versenden wieder frei und kann weitere Aktivitäten ausführen. Man spricht von „Remote Service Invokation“. So können Aufgaben auf fremden Rechnern angestoßen werden, von denen man keine Ergebnisse erwartet. In diesem Fall kann es aber passieren, dass der Sender schneller Nachrichten verschickt, als der Empfänger diese verarbeiten kann. Dadurch sind der Sender und der Empfänger nicht mehr zeitlich an einander gekoppelt. In diesem Fall muss aber auch ein Puffer implementiert werden, der die Nachrichten für die Versendung zwischenspeichert und nacheinander an den Empfänger ausliefert.

Bei jeder dieser Kommunikationsarten muss sichergestellt werden, dass die Nachrichten auch den Empfänger erreichen. Bei der synchronen Übertragung kann man dies durch Bestätigungsnachrichten erreichen. Bei dem asynchronen Fall wird es schwieriger. Da man keine Antwort von dem Zielsystem erwartet, kann auch keine Bestätigung empfangen werden. In diesem Fall muss das darunter liegende Kommunikationsprotokoll eine Zustellung garantieren. Es muss aber auch die Möglichkeit berücksichtigt werden, dass die Nachricht, zum Beispiel durch den Ausfall des Empfängers oder den Abbruch der Netzverbindung, nicht ausgeliefert werden kann. In diesem Fall müssen geeignete Maßnahmen für den weiteren reibungslosen Verlauf getroffen werden.

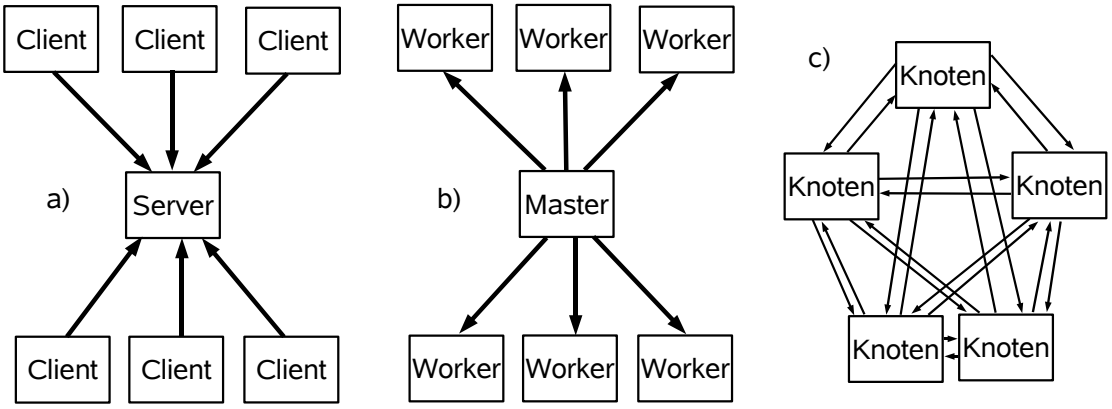



Abbildung 2.2.: Drei Modelle für die Rechnerorganisation. a) Das klassische Client/Server Modell. b) Das Master/Worker Modell c) Ein P2P Netz

## 2.5. Rollen der Rechner im Netz

Wie schon erwähnt dient die Vernetzung von Systemen dazu, Nutzer und Ressourcen zu verbinden. Dabei nehmen die einzelnen Netzteilnehmer unterschiedliche Rollen an. Die Abbildung 2.2 zeigt die möglichen Konstellationen. Sehr weit verbreitet ist das klassische Client/Server Modell. Bei diesem Szenario stellt ein fremder Rechner (der Server) eine Dienstleistung zur Verfügung, die ein Dienstnutzer (der Client) in Anspruch nehmen möchte. Dabei sind es meistens viele Clients, die diese Anfragen stellen, und ein Server, der für die Bearbeitung sorgt und das Ergebnis zurück an den Client versendet. Die Aufgaben der einzelnen Rechner sind klar verteilt. Der Server, der einen zentralen Anlaufpunkt bietet, lässt sich ziemlich leicht administrieren. So sind normalerweise höhere Sicherheitsstandards möglich und die korrekte Bearbeitung der Anfragen lässt sich garantieren. Allerdings hat die Zentralität auch Nachteile. Um die Anfragen schnell genug beantworten zu können, braucht der Server viel Leistung. Reicht diese nicht aus, so lässt sich der Server nicht ohne weiteres erweitern. Dieses System lässt sich also schlecht skalieren. Das gleiche gilt für die Netzwerkanbindung. Sie muss so dimensioniert sein, dass auch bei einer ungewöhnlich hohen Anfrageanzahl noch ein normaler Betrieb gewährleistet werden kann. Durch die Zentralität entsteht auch noch ein anderes Problem. Sollte der Server ausfallen, ist die Bereitstellung der Dienstleistung nicht mehr möglich.

Ungefähr das gleiche Konzept verfolgt das sogenannte Master/Worker Modell. Bei diesem Modell gibt es einen Client, der als Master fungiert. Dieser Master delegiert seine Arbeit an die einzelnen Server, die jetzt Worker heißen. Im Unterschied zum klassischen Modell gibt es normalerweise nur einen Master, der von vielen Workern bedient wird. Der zweite große Unterschied ist die Eigenschaft der Anfrage. In dem klassischen Model wird die gesamte Anfrage an einen Rechner gesandt. Bei dem Master/Worker Paradigma wird die Aufgabe normalerweise in viele Teilaufgaben zerlegt. Jeder der Worker kriegt

dann eine der Teilaufgaben zur Ausführung. Durch diese Aufteilung ist ein Ausfall eines Workers nicht mehr schlimm, die Arbeit kann normalerweise an andere Rechner delegiert werden. Auch lässt sich in diesem System leicht durch den Anschluss weiterer Rechner die Leistung steigern. Das einzige Problem ist die Netzanbindung des Masters. Da er gleichzeitig mit allen Workern kommunizieren muss, könnte die Bandbreite ein Problem werden. Bei diesem Modell ist der Ausfall des Masters genau so schlimm, wie der Ausfall des Servers im klassischen Modell. Kann die Arbeit nicht verteilt werden, gibt es auch keine Ergebnisse.

Das dritte Modell ist ein Peer to Peer Netz. Dieses Modell wird oft benutzt, wenn es keine klare Hierarchie im System geben soll. Jeder Knoten kann gleichzeitig die Rolle eines Servers und eines Clients annehmen. Dadurch ist dieses System sehr ausfallsicher. Es muss schon ein großer Teil der Knoten ausfallen, um die Systemstabilität zu beeinträchtigen. Auch kann das Netz gut skaliert werden. Wird weitere Leistung benötigt, kann ein zusätzlicher Rechner angeschlossen werden. Der große Nachteil ist der stark erhöhte Kommunikationsaufwand. Da es keinen zentralen Ansprechpartner mehr gibt, muss sich jeder Knoten mit mehreren anderen Maschinen verbinden. Wie man leicht aus dem Bild sehen kann, kann dies schnell zu großen Ansprüchen an die Netzanbindung führen. Deshalb wird bei den meisten Peer to Peer Systemen ein Routing eingesetzt, um die Zahl der gleichzeitigen Verbindungen zu reduzieren.
Jedes dieser Modelle hat eigene Einsatzbereiche. Deshalb kann man auch nicht sagen, welches von ihnen das bessere ist. Es kommt sehr stark auf die Anwendung an. Bei meiner Arbeit hat es sich angeboten das Master/Worker Modell zu verwenden, da es für die verteilte Ausführung von Aufgaben konzipiert wurde.

## 2.6. Software für verteilte Systeme

Es gibt sehr viel Software auf der Basis von verteilten Systemen. Manche Lösungen richten sich an die Endbenutzer, die anderen an die Entwickler. Hier sollen beispielhaft einige Anwendungen für die verschiedenen Einsatzgebiete vorgestellt werden.

**JXTA**
Als erstes soll ein Protokoll und dessen Implementierung in Java für die Kommunikation zwischen den einzelnen Netzteilnehmern vorgestellt werden. JXTA (Abkürzung für Juxtapose)[5] wird seit 2001 von Sun entwickelt und soll einen Kommunikationsprotokoll für die einzelnen Netzteilnehmer definieren. Insgesamt wurden in JXTA 6 Protokolle definiert. Sie regeln die Entdeckung anderer Nutzer, die Organisation in Nutzergruppen, die Entdeckung und Bereitstellung von Ressourcen und schließlich die Kommunikation zwischen den einzelnen Benutzern. Implementiert wurden diese Protokolle in Java SE, JAVA ME, C/C++ und C#, wobei für letzteres große Teile der C++ Implementierung verwendet wurden.

[5] https://jxta.dev.java.net/

Das Ziel der Entwicklung war es, sehr unterschiedliche Endgeräte, die sich in unterschiedlichen Netzen befinden können und über unterschiedliche Kommunikationskanäle verbunden sind, miteinander kommunizieren zu lassen. Zum Beispiel können über JXTA normale PCs, PDAs, Drucker und Mobiltelefone miteinander verbunden werden. Dabei braucht der Nutzer sich keine Gedanken über die Verbindung zu machen, den Transport der Nachrichten übernimmt das System. Um diese Funktionalität zu ereichen, wird über das Netz eine neue Transportschicht gelegt und die gesamte Kommunikation über diese abgewickelt. Die darunter liegende Netzarchitektur spielt so keine weitere Rolle.
Das JXTA Netzwerk teilt alle Nutzer in 2 Kategorien, die Edge Peers und die Super Peers. Die Edge Peers sind normale Nutzer, die sich mit dem System verbinden und dessen Dienste in Anspruch nehmen. Im Gegenteil dazu stehen die Super Peers. Sie übernehmen die Verwaltung des Systems. Rendezvous Peers (RDV) stellen den Eintrittspunkt in das System dar. Sie sind miteinander Verbunden und sorgen für die Weiterleitung der Nachrichten im Netz. Jeder Nutzer muss sich mit einem RDV verbinden, und ist damit in dem Netz registriert. Um sich bekannt zu machen, muss der Nutzer dann ein sogenanntes „Advertisement“ verbreiten, dass Informationen über die bereitgestellten Dienste und die aktuelle Adresse enthält. Ab jetzt kann jeder anderer Nutzer mit ihm kommunizieren. Für die Kommunikation zwischen den Teilnehmern werden sogenannte „Pipes“ benutzt. Mit den Unicast Pipes kann eine Verbindung zwischen zwei Nutzern hergestellt werden. Die Propagate Pipe hingegen sendet die Informationen an viele Nutzer und kann für Multicasts und Broadcasts verwendet werden. Zu den Pipes ist anzumerken, dass sie keine IP des Gegenübers brauchen und zwischen beliebigen Teilnehmern hergestellt werden können, falls die Nutzer eingehende Verbindungen zulassen. Ist es nicht der Fall, können die Daten über einen Relay Super Peer verschickt werden. Der nicht erreichbare Nutzer verbindet sich mit diesem Peer und kriegt von ihm alle Nachrichten weitergeleitet. Dieses System ermöglicht sogar eine Verbindung, falls beide Nutzer nicht erreichbar sind und einen Relay Peer benutzen müssen.
Obwohl JXTA vor allem in heterogenen Systemen einige Vorteile bietet, konnte es sich bis jetzt nicht durchsetzen. Sun spricht zwar von 17000 Benutzern, es gibt aber kaum Software, die auf dieses System aufsetzt. Die große Hemmnis ist wohl ein Rendezvous Peer, der auf jeden Fall vorhanden sein muss. Da vor allem zum Einsatz in stark verteilten Systemen mehrere RDVs eingesetzt werden müssen, erfolgt die Entdeckung anderer Nutzer sehr langsam. Auch wirkt sich das Routing der Nachrichten über die RDV negativ auf die Kommunikationsgeschwindigkeit aus. Man kann aber gespannt sein, ob mit dem großen Aufkommen von Java-fähigen Handies, diese doch noch einen Durchbruch schafft.

### CORBA

CORBA (Common Object Request Brocker Architecture) wurde als Spezifikation von der OMG[6] (Object Management Group) Anfang der 90er Jahre rausgebracht. Diese Spezifikationen definieren die Schnittstellen für RPC zwischen den Rechnern. Allerdings wurde nicht der Prozedurale, sondern der objektorientierte Ansatz gewählt, so dass sich

[6] http://www.omg.org/

diese Spezifikationen vor allem in objektorientierten Sprachen einsetzen lassen. Dabei setzt man auf einer höheren Schicht an als JXTA. Die gesamte Netzwerkschicht wird vom System übernommen, das als Middleware fungiert, und ist so für den Entwickler transparent. Mitte der 90 Jahre gab es die ersten Implementierungen dieser Spezifikationen. Inzwischen gibt es CORBA Implementierungen in Java von Sun als ein Teil der Java-API und C++ durch zum Beispiel MICO.[26]
Möchte ein Benutzer die Ressourcen der externen Rechner in Anspruch nehmen, so muss er zuerst selber die Methoden definieren, die er aufrufen möchte. Es müssen vor allem der Methodenname, die zu übergebenden Parameter und Rückgabewerte angegeben werden. Auch mögliche Exceptions können spezifiziert werden. Insgesamt lassen sich diese Definitionen gut mit einem Java-Interface vergleichen. Diese Definitionen müssen in der IDL (interface definition language) geschrieben sein. Danach werden sie in die Zielsprache kompiliert. Dabei können die Sprachen auf dem Server und auf dem Client unterschiedlich sein. Die Sprachbindung ist damit also aufgehoben. Jetzt müssen auf beiden Seiten die entsprechenden Methoden des Objekts implementiert werden. Danach stehen diese Methoden zur Verfügung und können verwendet werden.
Um externe Rechner ansprechen zu können muss auf dem Client zuerst CORBA initialisiert werden. Danach müssen die Objekte, die extern ausgeführt werden sollen an CORBA gebunden werden. Jetzt werden die entsprechenden Methoden bei einem Aufruf extern ausgeführt. Dabei ist der Aufruf einer lokalen und einer externen Methode gleich. Das System kümmert sich um die Ausführung und liefert, wenn definiert, die Ergebnisse zurück. Intern wird der entsprechende Methodenaufruf samt den Parametern serialisiert und an den ORB (Object Request Brocker) weitergeleitet. Der ORB kümmert sich um die gesamte Kommunikationen. Er sucht die in Frage kommenden Server. Verbindet sich mit ihnen und leitet die auszuführende Methode weiter. Auf der Serverseite wird das Objekt deserialisiert und ausgeführt. Nach der Fertigstellung werden die Ergebnisse wieder an den Client übertragen. Dieser kann das die Ausführung des Programms wie gewohnt fortsetzen.
CORBA ist ein schönen Beispiel für ein verteiltes System, da man hier die erforderliche Transparenz geschaffen hat. Der Benutzer braucht sich nicht um die Interna der Ausführung zu kümmern. Er hat eine einfache Sicht auf das System. Obwohl CORBA eine Zeit lang erfolgreich war, tritt sie in letzter Zeit immer mehr in den Hindergrund. Vor allem mit dem Aufkommen von Webservices wurde dieses System immer mehr vernachlässigt.

**Web Services und SOAP**
Mit der immer stärkeren Verbreitung von XML wurde vom W3C die Spezifikation SOAP (Simple Object Access Protocol) geschaffen[7], um Objekte mit Hilfe von XML zu kodieren und diese dann übers Netz zu übertragen. Später wurde SOAP als Abkürzung aufgegeben und wurde eigenständig. Für den Transport der Objekte über das Netz wurden die Protokolle HTTP und SMTP vorgesehen. Diese Objekte lassen sich dann zum Beispiel

[7] http://www.w3.org/TR/soap/

für RPC-Dienste nutzen. Als Grundlage dazu wählte man Web Server und erschaffte das Web Services Framework, das aus vielen verschiedenen Spezifikationen besteht. Ähnlich CORBA müssen auch hier erst die Schnittstellen spezifiziert werden, was mittels der WSDL (Web Services Description Language) in einer XML-Configdatei geschieht. Zur Laufzeit muss dann ein geeigneter Server mittels einem „Service Brocker“ gefunden werden. Auf dem gefundenen Rechner läuft normalerweise ein Webserver, der die XML-Objekte empfängt und zur Ausführung an das angeschlossene System liefert. Nach der Ausführung wird das Ergebnis wieder an den Client zurückgeliefert.

Als großen Vorteil dieser Lösung wird angegeben, dass man sich nicht um Firewalls kümmern muss, da die Webserver normalerweise einen freien Zugang zum Netz haben und man die Anfragen über HTTP tunnelt. Als Nachteil erweist sich das XML. [16] Da die Objekte überwiegend binäre Daten enthalten, müssen sie bei der Serialisierung in lesbaren Text umgewandelt werden, was die Größe der Nachrichten stark aufbläht. Inzwischen gibt es immer mehr kritische Stimmen, die den Web Services wegen ihrer Komplexität das gleiche Schicksal wie CORBA vorhersagen.

**Condor®**

Nach den verschiedenen Spezifikationen und Entwicklungsumgebungen soll noch ein Projekt vorgestellt werden, dass eine komplette Umgebung für parallele Tastausführung bietet. Hier wurde der Schwerpunkt nicht auf einzelnen Objekte gelegt, man kann diesem System komplette Prozesse zur parallelen Ausführung in Auftrag stellen.

Das Condor Projekt wurde 1988 an der Wisconsin-Madison Universität gestartet.[19] Ziel des Projektes war es, unbenutzte Workstations für die Ausführung von Tasks zu nutzen. Untersuchungen haben ergeben, dass die Nutzer ihre Ressourcen nur zur 30% auslasten. Die restlichen 70% lagen brach. Jeder User im Condor-Netz kann seine Jobs zur Ausführung an das System übertragen. Dieses sucht nach einem passenden Rechner und sorgt für die Ausführung. Der Aufraggeber kann bestimmte Eigenschaften für die Ausführung spezifizieren und die Priorität des Jobs festlegen. Neben den Rechnereigenschaften ist für die Zuordnung vor allem entscheidend, dass der Rechner zu dem Zeitpunkt nicht von einem lokalen Nutzer verwendet wird. Darum werden die Eingaben über Tastatur und Maus überwacht und nur freie Rechner für die Rechenaufgaben verwendet. Kommt der Nutzer zurück, kann das System den momentanen Zustand des Jobs speichern und die Ausführung stoppen oder auf einen anderen Rechner übertragen. Sollte der Task auf externen Dateien zugreifen, werden diese auch auf den Rechner übertragen. Nachdem der Task fertig ist, wird der Nutzer über den Abschluss unterrichtet. Am Anfang war dieses System nur in den lokalen Rechnerclustern einsetzbar, doch mit dem Aufkommen des Grid-Computing wurde Condor-G[13] entwickelt, eine Version, die mehrere Cluster gleichzeitig ansprechen kann, und so auch außerhalb des lokalen Netzes nach passenden Maschinen sucht. Zur Zeit sollen über 800 Cluster weltweit die Condor Software für die Jobausführung benutzen.

Es ist leicht, simple Aufgaben mit Condor auszuführen. Um jedoch die weitergehenden Eigenschaften, wie die Migration, zu nutzen, muss der Task normalerweise neu kompi-

liert werden. Auch kann es passieren, dass einige der dynamisch gelinkten Bibliotheken auf dem Zielrechner nicht verfügbar sind. Es wird daher empfohlen, alle Bibliotheken statisch zu linken, um den Task von dem Ausführungsort unabhängig zu machen. Da es bei den Tasks meistens um kompilierte Programme handelt, muss der Zielrechner natürlich das gleiche Betriebssystem, und die gleiche Rechenarchitektur haben. Das gilt auch für die Rechnerauswahl bei einer Migration. Das schränkt das System auf homogene Umgebungen ein.

Da verteilte Anwendungen in letzter Zeit immer mehr gefragt sind, gibt es eine sehr große Anzahl weiterer Lösungen und Spezifikationen. Es gibt aber nichts, was sich stark durchsetzen konnte. Sehr viele Projekte laufen nebeneinander und werden für eigene Insellösungen verwendet. So muss man sich zwischen einer Vielzahl von Systemen entscheiden, die auf dem ersten Blick keinen großen Vorteile gegenüber anderen Systemen aufweisen. Als einen letzten Schritt kann man auch die existierenden Lösungen verwerfen, und eine eigene entwickeln.

# 3. Data Mining und RapidMiner

Nach dem im vorherigen Kapitel die verteilten Systeme vorgestellt wurden, wird hier ein sinnvolles Einsatzgebiet für diese Systeme vorgestellt. Zuerst werden einige allgemeine Sachen über die Data Mining Techniken vorgestellt, dann konkret auf RapidMiner eingegangen, ein Data Mining Werkzeug, dass für diese Diplomarbeit benutzt wurde.

In der heutigen Gesellschaft werden immer mehr Daten produziert. Manche fallen automatisch an. Andere müssen aufwendig erhoben werden. Leider sind diese Daten in ihrer Rohform nutzlos. Der Nutzen entsteht erst durch die Gewinnung von Informationen aus diesen Daten. Den Vorgang der „Gewinnung von implizierten, vorher unbekannten und möglicherweise nützlichen Informationen aus den Rohdaten durch nichttriviale Methoden nennt man Data Mining“[12]. Genau so wie die Daten, können die daraus gewonnen Informationen ganz unterschiedlicher Natur sein. Auch der Nutzen dieser Informationen ist sehr stark von der Aufgabenstellung abhängig.
Die Anwendungsgebiete für Data Mining sind sehr unterschiedlich. Es gibt kaum einen Bereich in unserer Umgebung, der sich nicht für Datengewinnung eignet. Besonders leicht kann man die Daten im Internet erheben. Jeder Webserver protokolliert zum Beispiel einzelne Webseitenzugriffe. Sogar hier fallen sehr viele Daten an. Man kriegt die IP-Adresse, die Kennung des verwendeten Browsers und einen Referer (Die Seite, von der man die aktuelle Seite aufruft). Schon aus diesen Daten lassen sich viele Informationen gewinnen. Durch Einsatz von Javascript und Cookies kann die Anzahl der erhobenen Daten noch weit vergrößert werden. Man kann damit sogar die Anzahl und die Häufigkeit der Besuche von jedem Nutzer erfahren. Aus all diesen Daten kann man Informationen über das Verhalten und die Interessen der Besucher gewinnen und die Seite entsprechend anpassen. Vor allem kann die Werbung an die Besucher angepasst werden.
Schon in diesem einfachen Beispiel kommen kommerzielle Interessen durch. Auch sonst kommen sehr viele Beispiele aus der Wirtschaft, die Informationen über ihre Kunden sammelt, um das Angebot oder den Werbungseinsatz zu optimieren. Ein ganz einfaches Verfahren ist es, die Kunden an der Kasse ihre Postleitzahl angeben zu lassen. Durch die Verknüpfung mit den gekauften Gegenständen läst sich das Aussehen der Werbung für einzelne Ortschaften optimieren. Etwas aufwendiger ist das Verteilen von Kundenkarten, so dass man die kompletten Einkäufe zeitlich zuordnen kann, und so das Angebot an das vielleicht geänderte Verhalten anpassen kann. Aber auch allein der Inhalt des Warenkorbs kann viele Informationen für bessere Platzierung der einzelnen Güter im Kaufhaus liefern.
Es gibt aber auch andere Einsatzgebiete. Die Anzahl von unerwünschten Mails (Spam) nimmt immer weiter zu. Auch Data Mining Verfahren können bei der automatischen Erkennung solcher Mails helfen. Durch die Analyse des Mailheaders und der Suche nach

bestimmten Schlagwörtern, kann man Regeln aufstellen, mit denen man die eingehende Post in erwünschte und unerwünschte klassifizieren kann.
Leider hat das Data Mining in großen Teilen der Gesellschaft einen negativen Ruf. Da sehr oft persönliche Daten zur Analyse verwendet werden ist die Angst vor dem Missbrauch sehr groß. Auch können Fehler in der Auswertung zu sehr ernsten Konsequenzen für die betroffenen Menschen führen.

## 3.1. Generelle Data Mining Verfahren

Wenn man aus Daten irgendwelche Informationen gewinnen möchte, muss man zuerst definieren, wie diese Daten vorliegen. Da sehr oft Datenbanken für die Speicherung verwendet werden, gibt es eine Anzahl von einzelnen Einträgen, die „Beispiele“ genannt werden. Jedes Beispiel hat dabei mehrere Merkmale, die ihn charakterisieren. Man kann sich jedes Beispiel als einen Vektor vom Merkmalen vorstellen, so dass eine Beispielmenge sich als eine zweidimensionale Matrix darstellen lässt. In dem obigen Fall mit den Webseitenlogs wäre jeder Zugriff ein Beispiel. Die Merkmale wären der Zeitpunkt des Zugriffs und die Browserkennung/Referer als Text. Leider ist einfacher Text in diesem Fall schlecht für eine Analyse geeignet. Da es nur eine begrenzte Anzahl an Browsern gibt, könnte man hier zum Beispiel für jeden Typ ein Merkmal anlegen und das richtige Feld auf „wahr“ und den Rest auf „falsch“ setzen. Im Endeffekt sollte man eine Tabelle mit numerischen und/oder nominalen Werten haben.
Auf diesen Daten sollen jetzt bestimme Verfahren angewendet werden, um Informationen zu erhalten. Doch welche Art von Informationen will man haben? Normalerweise will man entweder Voraussagen für zukünftige Daten treffen, oder den aktuellen Daten eine Struktur geben. Nachfolgend sollen drei wichtige Verfahren für die Datenanalyse vorgestellt werden: Es sind die Klassifikation, die Regression und das Clustering. [14]

### Die Klassifikation

Bei der Klassifikation möchte man jedes Beispiel einer Klasse aus einer bestimmten Klassenmenge zuordnen. Ein Teil der Beispiele wird dabei per Hand klassifiziert und dient als eine Beispielmenge für den Lernalgorithmus. Auf dieser Menge sucht der Lerner eine Reihe von Regeln, die die Zuordnung des Beispiels zu der Klasse am besten beschreiben. Sind diese Regeln gefunden, so können weitere Beispiele maschinell klassifiziert werden, und benötigen keine manuelle Entscheidung mehr. Die Lernverfahren, die eine manuell vorverarbeitete Beispielmenge brauchen gehören zu dem „überwachten Lernen“. Das Ergebnis der Klassifikation hängt dabei sehr stark von der Qualität der abgeleiteten Regeln ab. Oft ist es nicht möglich eine Klasse ganz genau gegenüber der Anderen abzugrenzen, so dass es später zu Fehlentscheidungen kommen kann. Auch kann die Anzahl der Beispiele für eine genaue Regelbildung zu gering sein, so dass der Lerner auf den vorhandenen Daten zwar eine perfekte Klassifikation ermöglicht, auf den Fremddaten dann aber Fehler aufweist. Auf dem Bild 3.1 kann man die Einteilung einer Beispielmenge in

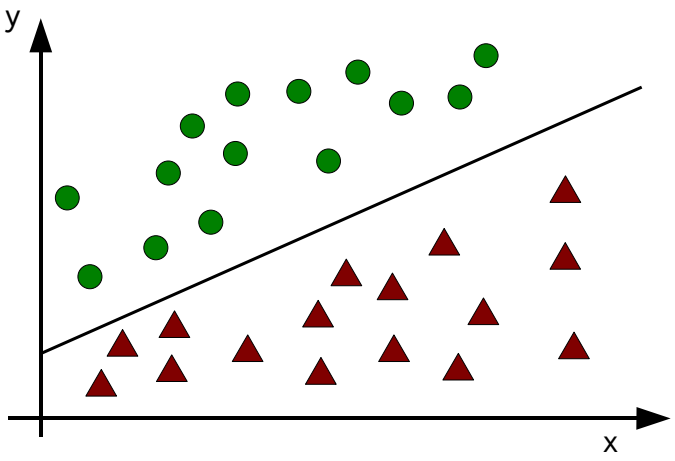



Abbildung 3.1.: Beispiel einer Klassifikation in zwei Klassen. Durch die Linie in werden die beiden Klassen eindeutig definiert.

zwei Klassen sehen. Dabei können die beiden Klassen durch eine lineare Funktion getrennt werden. Zu Beachten ist, dass die Klassen schon vorher feststanden, die Aufgabe bestand darin, eine Regel zu finden, die diese beiden Klassen voneinander abtrennt, so dass zukünftige Beispiele sofort einsortiert werden können.

**Die Regression**

Ein anderes Verfahren ist die Regression. Hier geht man davon aus, dass alle Beispiele einer bestimmten Funktion folgen. Die lineare Regression zum Beispiel hat eine lineare Funktion als Basis und wurde Anfang des 19ten Jahrhunderts vorgestellt. Ein numerisches Merkmal wird ausgewählt und repräsentiert dabei den aktuellen Funktionswert des Beispiels. Bei der Regression wird versucht, aus einer gewichteten Kombination der anderen Merkmale das ausgewählte Merkmal vorauszusagen. Natürlich ist sehr schwer, oder sogar unmöglich eine Funktion zu finden, die für alle Beispiele gilt. Meistens liegen Ungenauigkeiten bei der Datenerhebung vor, so dass sich kleine Fehler nicht vermeiden lassen. Es kann also eine Differenz zwischen dem erfassten Wert und dem tatsächlichen Wert geben. Die Regression versucht diesen Fehler für alle Beispiele zu minimieren. Dabei braucht man bei der Abstandsberechnung nicht unbedingt eine einfache Differenz der beiden Merkmale zu nehmen. Meistens wird die Abstandsberechnung angepasst. Man kann den Abstand zum Beispiel quadrieren, um weiter entfernte Ergebnisse stärker zu bestraffen, oder sehr kleine Abweichungen werden auf null gesetzt, um eine Toleranz gegenüber leichtem Rauschen bei der Datengewinnung zu erlauben. Die Regression eignet sich sehr gut, um Zusammenhänge zwischen Variablen zu beweisen. Wenn es gelingt, eine Funktion zu finden, die die Datenmenge abbilden kann, müssen die Variablen miteinander in Verbindung stehen. Außerdem kann die Regression benutzt werden, um Aussagen

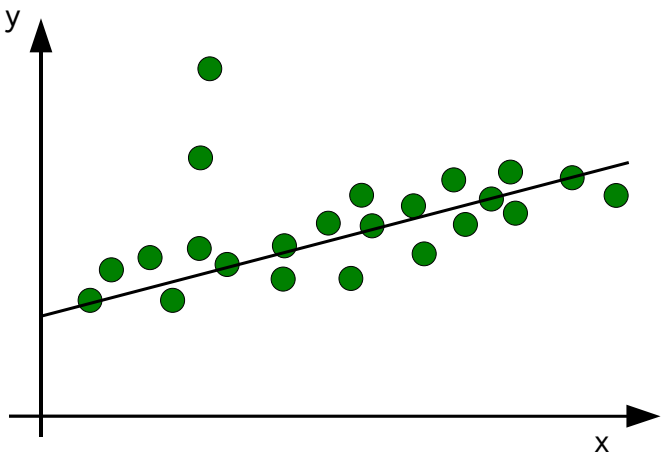



Abbildung 3.2.: Beispiel einer linearen Regression. Man versucht eine Funktion zu finden, die alle Datenpunkte möglichst genau beschreibt.

über die möglichen Entwicklungen in der Zukunft zu treffen. Das Bild 3.2 zeigt eine lineare Regression. Die Beispiele haben jeweils zwei Merkmale. Dabei wird der $x$ Wert zu dem $y$ Wert in Verhältnis gesetzt Auf dieser Basis wird dann eine lineare Funktion konstruiert. Die $x$-Achse könnte dabei eine Zeitachse repräsentieren. Die $y$-Achse zu diesem Zeitpunkt gemessenen Wert. Wenn man eine geeignete Funktion gefunden hat, könnte man also Prognosen für die Zukunft erstellen. Dieses Beispiel mit nur zwei Merkmalen ist allerdings sehr vereinfacht. Normalerweise besitzen die Daten weit mehr Attribute. Wie man sieht müssen auch nicht alle Punkte auf der Linie liegen. Es gibt sogar zwei Ausreißer. Regressionsverfahren sollten gegenüber solchen Daten tolerant sein.

### Das Clustering

Das dritte wichtige Verfahren gehört in die Klasse des unüberwachten Lernens. Bei dem unüberwachten Lernen werden die einzelnen Beispiele nicht vorher per Hand klassifiziert und enthalten daher keine Zusatzinformationen. Die Clusteringalgorithmen versuchen nun, die Daten in mehrere Teilmengen zu gruppieren. Für die Zuordnung zu einer bestimmten Menge wird die „Ähnlichkeit“ zwischen den Beispielen als Vergleich genommen. Um die „Ähnlichkeit“ zu bestimmen, findet normalerweise ein Vergleich auf der Merkmalsebene statt. Die einzelnen Merkmale werden zwischen den Beispielen verglichen. Daraus kann man eine Distanz zwischen den beiden Beispielen erhalten. Beispiele, die eine kleine Distanz zueinander haben, werden dann zu der gleichen Teilmenge zugeordnet. Clusteringverfahren, die auf einer Distanzmetrik basiert sind, gehören zu dem distanz-based Clustering. Zur Distanzberechnung können verschiedene Distanzmetriken benutzt werden. Bei der Manhatten Distanz zweier Punkte bildet man die Differenz ein-

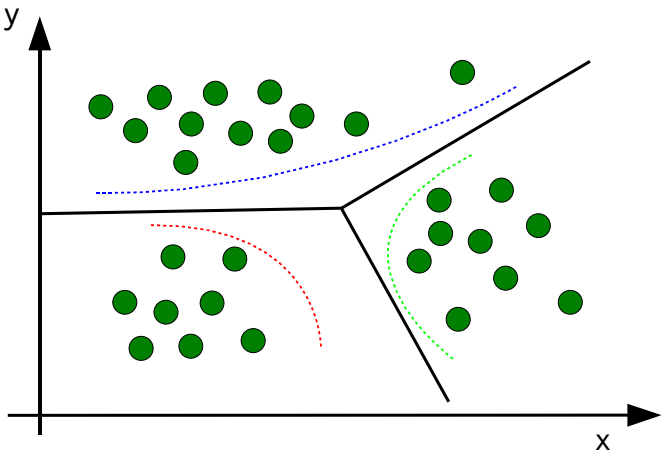



Abbildung 3.3.: Ergebnis eines Clusterers

zelner Attribute und addiert sie. Bei der Euklidischen Distanz werden die Differenzen quadriert, dann daraus die Summe gebildet und anschließend die Wurzel gezogen. Eine Allgemeine Formel für Distanzen bildet die Minkowski Metrik:

$$Distanz_p(x_i, x_j) = (\sum_{k=1}^{d} |x_{i,k} - x_{j,k}|^p)^{1/p} \tag{3.1}$$

Die einzelnen Attribute der zwei $d$-dimensionalen Punkte $x_i$ und $x_j$ werden von einander subtrahiert. Dann mit $p$ potenziert und aus der Summe die $p$-te Wurzel gezogen. Bei $p = 1$ entspricht diese Formel der Manhatten Distanz, für $p = 2$ der Euklidischen. Wenn man diese Metriken beim Data Mining verwenden möchte, muss man einige Punkte beachten. Die einzelnen Merkmale der Beispiele können in sehr verschiedenen Intervallen liegen. Ein Attribut könnte zum Beispiel die Werte von 0 bis 1 einnehmen, ein anderer zwischen 0 und 1000 liegen. Natürlich wird das zweite Attribut viel ausschlaggebender für die Distanz sein als das Erste. Es gibt zwei Möglichkeiten mit solchen Werten umzugehen. Man kann die Merkmale normieren, so dass sie alle nur in einem bestimmten Wertebereich liegen, und ihre Differenzen so den gleichen Einfluss auch die Distanz haben. Man kann jedes Attribut auch gewichten, um so einige Daten als besonders wichtig zu markieren.
Ein anderes Problem ist die Berechnung einer Differenz zwischen zwei nominalen Attributen. Auch hier gibt es mehrere Lösungen, die von der Beschaffenheit der Attribute abhängen. Man kann zum Beispiel für zwei gleiche Attribute die Distanz auf null setzen, und für verschiedene die maximale Distanz der numerischen Attribute verwenden. Lassen sich die nominalen Werte ordnen, so kann man auch mehrere Stufen der Entfernung verwenden.
Die Abbildung 3.3 zeigt das Ergebnis eines Clusteringvorgangs. Man hat sich entschieden die Daten in drei Teilmengen zu unterteilen. Die Linien trennen die einzelnen Cluster

von einander. Zu beachten ist, dass die Zugehörigkeit zu einer Gruppe, im Gegensatz zu der Klassifikation, erst nach der Ausführung des Verfahrens fest steht.

## 3.2. Konkrete Data Mining Verfahren

Nach dem einige Vorgehensweisen allgemein besprochen wurden, wird jetzt auf die einzelnen Verfahren im Detail eingegangen. Diese Verfahren wurden für die spätere Parallelisierung ausgewählt.

### 3.2.1. Feature Optimization

Viele Aufgaben haben die Eigenschaft, dass sie an sehr vielen Merkmalen pro Beispiel verfügen. Jedes zusätzliche Merkmal verlängert die Bearbeitungszeit. Es gibt auch das Problem, dass manche Merkmale (fast) identisch sein können. Andere sind wiederum nutzlos und helfen bei der Bearbeitung nicht. Manche Lernalgorithmen werden bei der Bewertung diesen Attributen zu viel Gewicht beimessen und das Ergebnis verfälschen. Um bessere Ergebnisse zu bekommen, ist es oft vom Nutzen, eine Vorverarbeitung auf den Daten durchzuführen und die Merkmale an die entsprechende Aufgabe anzupassen. Dabei kann man in zwei Richtungen vorgehen. Bei der Feature Selection werden die Merkmale, die sich für die Analyse als schädlich erweisen einfach entfernt. Bei Feature Construction werden nach einem bestimmten Schema neue Merkmale aus den alten gebildet.
Das ganze Verfahren besteht aus mehreren Schritten. Zuerst wird eine Menge von Merkmalkombinationen ausgewählt. In dem nächstem Schritt wird auf jedem Element dieser Menge, also für jede Merkmalkombination das entsprechende Data Mining Verfahren angewendet. Die Leistung des Algorithmus wird dann überprüft. Aus der Menge der Kombinationen wird dann nach bestimmten Kriterien eine neue Menge gebildet und das ganze Verfahren so oft wiederholt, bis sich keine Verbessung der Analyseergebnisse mehr zeigt. Für die Wahl und die Veränderung der Merkmalmenge gibt es mehrere Verfahren. Drei von ihnen werden hier kurz vorgestellt.

#### Forward Selection

Bei der Forward Selection wird eine $n$-elementige Merkmalmenge in $n$ 1-elementige disjunkte Mengen zerlegt. Jetzt wird die Analyse $n$ mal durchgeführt. Ein Mal auf jeder Menge. So kann man messen, welches Merkmal den größten Einfluss auf ein gutes Ergebnis hat. Aus dieser Menge werden jetzt die $p$ besten Ergebnisse ausgewählt. Die Anzahl $p$ kann dabei frei festgelegt werden. Als nächstes wird die Anzahl der Merkmale in jeder Menge erhöht. Es wird jeweils einer der anderen $n-$ Merkmale zu der Menge hinzugefügt, so dass es aus einer alten 1-elementige Menge $n-1$ neue 2-elementige Mengen entstehen. Jetzt kann wieder mit der Analyse fortgefahren werden. So entsteht eine Schleife. In jedem Durchgang wird ein neues Merkmal in die Menge hinzugefügt und getestet. Das Verfahren wird beendet, wenn sich eine Zeit lang keine Verbesserung der Ergebnisse

einstellt, oder alle Attribute benutzt wurden. Am Ende wird die Merkmalmenge mit den besten Ergebnissen ausgegeben.

#### Backward Elemination

Dieses Verfahren ist der Forward Selection recht ähnlich. Allerdings startet man hier nicht mit einer leeren Beispielmenge, sondern mit einer vollständigen. Aus dieser Menge wird jeweils ein Merkmal entfernt, so dass es $n$ neue Mengen mit jeweils $n-1$ Merkmalen entstehen. Jetzt wird die Leistung jeder dieser Mengen gemessen. So kann man feststellen, welche Merkmale den kleinsten Anteil an dem Ergebnis haben. Es werden wieder die $p$ besten Mengen ausgewählt, und ein weiteres der $n-1$ Elemente aus jeder Menge entfernt, so dass aus jeder $n$-elementigen Menge jetzt $n-1$ Mengen mit jeweils $n-2$ Elementen entstehen. Die Analyse wird mit den neuen Mengen in einer Schleife wiederholt, bis jede Menge nur noch ein Element enthält, oder die Ergebnisse sich nicht mehr verbessern. Dieses Verfahren hat den Nachteil eines großen Speicherverbrauchs. In den ersten Schritten, werden viele fast vollständige Teilmengen der Daten generiert. Allerdings enthalten die, von diesem Verfahren generierte Merkmalmengen, meistens mehr Merkmale als der Forward Selection Ansatz.

#### Evolutionärer Ansatz

Die beiden ersten Verfahren probieren nur eine kleine Teilmenge der Merkmalkombinationen aus. Es ist also überhaupt nicht klar, ob die beste Menge überhaupt getestet wird. Allerdings arbeiten sie systematisch. Ein anderer Ansatz ist die evolutionäre Auswahl der Attribute. Hier wird man Anfang eine oder mehrere zufällige Attributmengen ausgewählt, und analysiert. In jedem Schritt verändert man diese Mengen, in dem man Merkmale hinzufügt oder entfernt. Diese Veränderungen können zufällig oder nach bestimmten Kriterien geschehen. Die Suche wird abgebrochen, wenn eine maximale Anzahl von Veränderungen stattgefunden hat, oder sich seit einiger Zeit keine Verbessung der Ergebnisse erreicht wurde. Dieser Ansatz testet auch nur einen kleinen Teil der gesamten möglichen Kombinationen und hat eine Zufallskomponente drin. Es kann also auch hier nicht garantiert werden, dass die perfekte Lösung gefunden wird.

### 3.2.2. K-Means

Nach dem die Vorverarbeitung der Daten abgeschlossen ist kann das maschinelle Lernen beginnen. Als ein Beispiel für ein Lernverfahren soll hier ein Clustering Algorithmus vorgestellt werden.
Der K-Means Algorithmus [20] wurde von James B. MacQueen im Jahr 1967 vorgestellt. Der Algorithmus selber ist einfach, er garantiert aber keine guten Ergebnisse. Als Eingabe benötigt das Verfahren einen Wert $k$, der für die Anzahl der Gruppen (Cluster) steht, in denen die Daten einsortiert werden. In der Initialisierungsphase werden zufällig in den Datenraum $k$ Punkte gesetzt.
Diese Punkte repräsentieren die Zentroiden der Cluster. Jetzt wird die Distanz zwischen jedem Datenpunkt und den einzelnen Zentroiden berechnet und die Daten in die Cluster

mit der kleinsten Distanz eingeordnet. Am Ende sind alle Daten den $k$ Clustern zugeordnet. Im nächstem Schritt müssen die Zentroiden neu berechnet werden. Dabei wird der Mittelwert der Merkmale aller dem Cluster zugeordneten Beispiele genommen. Jetzt können wieder die Distanzen zwischen den Daten und den neuen Zentroiden berechnet werden und die Daten entsprechend neu zugeordnet werden, wobei hoffentlich andere Cluster entstehen, da das Zentrum des Clusters sich verschoben hat. Es entsteht eine Schleife, die erst dann beendet wird, wenn die Zentren bei der Neuberechnung sich nicht mehr bewegen. Dabei wurde die folgende Zielfunktion minimiert:

$$Gesamtdistanz = \sum_{j=1}^{k} \sum_{i=1}^{n} |x_i^{(j)} - c_j|^2 \tag{3.2}$$

Wobei $x_i^{(j)}$ alle Beispiele sind, die zum Cluster $j$ gehören und $c_j$ der Zentroid dieses Clusters ist und der $x_i^{(j)} - c_j$ Ausdruck eine Distanzmetrik zwischen dem Beispiel und dem Zentrum des Custers darstellt. Diese Zielfunktion minimiert die quadratische Distanz aller Daten zu den entsprechenden Clusterzentren. Die Qualität der Lösung hängt dabei sehr stark von den anfangs gewählten Startpunkten für die Zentroiden. Man findet zwar ein Optimum, es kann sich aber um ein lokales und kein globales Optimum handeln. Es kann sogar passieren, dass während der Ausführung ein Cluster ganz leer bleibt. Darum wird der Algorithmus normalerweise mehrmals hintereinander mit verschiedenen Startpunkten ausgeführt, mit der Hoffnung, dass wenigstens eine Ausführung das globale Optimum findet. Das beste Ergebnis wird dann übernommen.

### 3.2.3. Parameter Optimierung

Der Erfolg des Lernens hängt nicht zuletzt von den gewählten Parametern des Lerners ab. Schon bei K-Means kann man die Anzahl der Cluster, die Startpunkte und die Anzahl der Ausführungen einstellen. Bei anderen Verfahren sind die Einstellungsmöglichkeiten noch umfangreichlicher. Doch ist es nicht immer ganz klar, welche Parameterkombination die besten Ergebnisse liefert. Eine Möglichkeit ist es, den Algorithmus mit vielen möglichen Parametern aufzurufen, und dann das beste Ergebnis zu behalten. Natürlich steigert es die Ausführungszeit enorm, da pro Parameterkombination ein Gesamtlauf des Experiments erforderlich ist. Allerdings kann dadurch die Qualität der Ergebnisse oft verbessert, und vor allem Information über die besseren Parameterwerte für die Zukunft gewonnen werden.
Wie wählt man die Parameter zum Optimierung? Sind die zu optimierenden Parameter unabhängig von einander, kann jeder Parameter einzeln optimiert werden. Bei 3 Parametern mit je 10 verschiedenen Werten erfordert es insgesamt 30 Durchläufe. Doch was, wenn die Parameter voneinander abhängig sind? Dann muss für jede Kombination ein Durchlauf gestartet werden. Beim obigen Beispiel wären es 1000 Durchläufe, die durchgeführt werden müssen. Auch kann es passieren, dass man zwar den Bereich, in dem die Werte liegen sollte, kennt, aber keine konkreten Werte angeben kann, ohne Gefahr zu laufen, dass das Optimum dazwischen liegt. Hier kann ein evolutionärer Ansatz helfen. Bei diesem Ansatz gibt man nur den Wertebereich an, in dem die Parameter liegen

könnte. Der Algorithmus führt dann die Suche selbständig durch. Dabei das Ergebnis eines Suchvorgangs evaluiert, und die nächste Suche entsprechend angepasst.

### 3.2.4. Kreuzvalidierung

Ein Problem beim überwachten Lernen ist die Unwissenheit, wie gut die ermittelte Regelmenge/Gewichtung auf unbekannten Daten funktioniert. Es kann sehr gut sein, dass die Regeln zwar perfekt auf die vorhandenen Beispiele passen, aber schlecht auf anderen Daten sind. Das ist normalerweise auf eine Überanpassung an eine bestimmte Datenmenge zurückzuführen. Doch wie kann man diese Überanpassung feststellen, und wie geht man dagegen vor? Eine Lösung wäre es eine zweite Datenmenge bereitzuhalten, die gewählten Regeln darauf anzuwenden, und dann die Performanz zu ermittelt. Diese Daten müssen aber auch vorher manuell klassifiziert werden und stehen nicht immer zur Verfügung. Eine viel verbreiterte Methode ist es, die Originaldaten in zwei Teilmengen zu splitten, in ein Trainingsset und ein Testset. Auf der ersten Menge wird dann gelernt und die zweite dient der Überprüfung. Jetzt weiß man zwar, wie gut der Algorithmus auf fremden Daten abschneidet, allerdings können wir die Überanpassung nicht verhindern, und haben sie vielleicht sogar noch durch die kleinere Trainingsmenge noch verstärkt. Hier kommt die Kreuzvalidierung ins Spiel. Bei diesem Verfahren werden die Daten in viele kleine Teilmenge unterteilt. Eine der Teilmengen wird zur Prüfung eingesetzt, auf den anderen wird gelernt. Man führt das Lernen mehrmals aus, und jedes mal wird eine andere Teilmenge für die Prüfung verwendet. Wenn man die Daten zum Beispiel in 10 Teilmengen unterteilt hat, wird der Algorithmus 10 mal ausgeführt, und man spricht von einer 10fachen Kreuzvalidierung. Am Ende wird ein Mittelwert über alle Validierungen gebildet, so dass man eine Vorstellung hat, wie gut oder schlecht das verwendete Verfahren auf unbekannten Daten abschneidet. Es bleibt noch die Frage, in wie viele Teilmengen man die Daten unterteilen soll. Ein Extrem dabei ist die „Leave one out“ Kreuzvalidierung, bei der die Testmenge jedes mal aus nur einem Element besteht. Die Anzahl der Validierungen ist dann gleich der Anzahl der Beispiele. Eine andere Methode ist es, die Kreuzvalidierung mehrmals auszuführen. Da die Daten dabei jedes mal anders aufgeteilt werden, kann es auch zur Steigerung der Genaugkeit der Vorhersage genutzt werden.

## 3.3. RapidMiner

Das größte Problem des Data Minings ist ein extremer Zeitaufwand, der für die Durchführung der Algorithmen auf den Daten benötigt wird. Oft ist es auch nicht ersichtlich, welcher Weg am besten zum Ziel führt, auf welchen Daten man genau arbeiten soll oder welcher Algorithmus eingesetzt werden soll. Normalerweise müssen mehrere Methoden ausprobiert werden, um nachher das beste Verfahren zu nehmen. Um Zeit zu sparen, probiert man die entsprechenden Verfahren erst auf einer kleineren Testmenge aus, bevor man die kompletten Daten analysiert. Dabei ist es wichtig, die einzelnen Experimente schnell erstellen und verändern zu können. Um den Vorgang des Ausprobierens zu vereinfachen wird seit 2004 ein Tool namens RapidMiner (früher Yale - Yet Another Learning

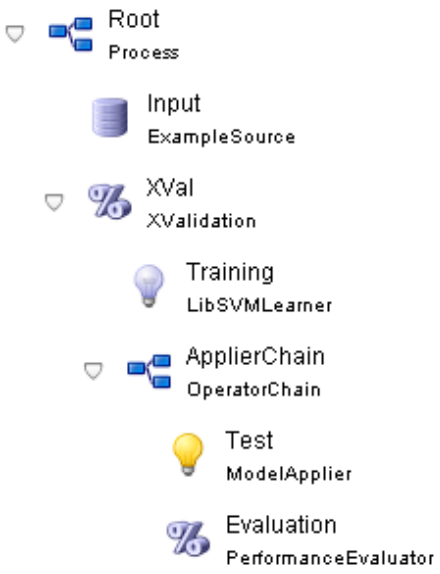



Abbildung 3.4.: Operatorbaum des RapidMiners. Die einzelnen Operatoren sind hierarchisch aufgebaut und können mehrere innere Operatoren haben.

Enviroment)[22] entwickelt. Das Ziel der Entwicklung ist es, ein flexibles Framework für Data Mining zu schaffen. Man sollte sehr schnell die einzelnen Experimente verändern und neu erstellen, geeignete Eingabedaten auswählen und die erstellten Experimente ohne großen Aufwand ausführen können. Der komplette Vorgang des Data Mining wird von dem Tool als ein „Process“ dargestellt. Die Gesamtaufgabe wird dann in mehrere Teilaufgaben unterteilt. Für jede Teilaufgabe ist ein eigener „Operator“ zuständig. So entstehen eine Liste von Operatoren. Da ein Operator aber weitere Unteroperatoren enthalten kann, die kleinere Teilaufgaben übernehmen bot sich die Darstellung als ein Operatorbaum an. Das Bild 3.4 zeigt ein Beispiel für ein Experiment. Es ist klar eine Hierarchie erkennbar. Der oberste Root-Operator steht für das ganze Experiment. Darunter sind die Operatoren für die Teilaufgaben, wobei die Kreuzvalidierung noch weitere Unteroperatoren enthält, die normalerweise in einer Schleife ausgeführt werden. Dabei ist jeder Operator gekapselt. Er braucht eine bestimmte Eingabe und produziert eine bestimmte Ausgabe. Ansonsten handeln die meisten Operatoren autonom. Dies ermöglicht einen schnellen Austausch der einzelnen Operatoren durch äquivalente Bausteine. Sie müssen nur für die gleiche Ein- und Ausgabe definiert sein. Auch das Einfügen und das Entfernen der einzelnen Teile ist sehr leicht.

Dabei läuft jedes Experiment mehr oder weniger gleich ab. Als ersten Schritt müssen die Daten geladen werden. Diese Daten können sich in einer Datei befinden, aus einer Datenbank kommen oder durch einen Operator generiert werden. Jetzt können auf diesen Daten verschiedene Operationen durchgeführt werden. Normalerweise wird zuerst eine

Vorverarbeitung auf den Daten durchgeführt. Entweder wird die Beispielmenge verkleinert, oder einzelnen Merkmale aus den Beispielen entfernt. Jetzt kommt die eigentliche Analyse. Es werden spezielle Operatoren, die sogenannten „Lerner“ auf die Daten angewendet, und liefern irgendwelche Ergebnisse. In dem Nachbearbeitungsschritt können diese Ergebnisse an bestimmte Vorgaben angepasst werden. Im letzten Schritt können die Daten gespeichert werden. Daneben wird eine visuelle Ausgabe erzeugt, die in einer Listen-, Tabellen- oder Diagramform sein kann.
Die Auswahl an verschiedenen Operatoren ist enorm. Insgesamt stellt RapidMiner über 400 Operatoren für verschiedene Aufgaben bereit. Außerdem lassen sich die Operatoren von WEKA[34] (Waikato Environment for Knowledge Analysis) sehr leicht integrieren. Es lassen sich auch sehr einfach eigene Operatoren schreiben und als ein Plugin in den RapidMiner hinzufügen.

# 4. Parallel Task Processing

Durch die Entwicklung von verteilten Systemen gibt es immer mehr Möglichkeiten Aufgaben auf mehreren Rechner parallel laufen zu lassen. Dabei stellt sich immer öfter die Frage, nach welchem Muster die Verteilung vor sich gehen soll, um die Ausführung möglichst effizient zu gestalten. „Den Prozess, endliche Ressourcen den Aufgaben zuzuordnen nennt man Scheduling.“ [24] Bei der Verteilung verfolgt man normalerweise ein bestimmtes Ziel. Dabei ist das Schedulingproblem älter als die verteilten Systeme oder Rechner im allgemeinen. Den Ursprung hat das Scheduling in der Industrialisierung, als man angefangen hat, durch Fliessbände und andere Mechanismen große Aufgaben in kleinere Tasks zu zerlegen und an verschiedene Arbeiter zu delegieren. Ein Beispiel aus der Fertigung soll das Problem verdeutlichen.
In einem Betrieb befinden sich mehrere Pressen, die für verschiedene Aufgaben genutzt werden können. Auf jeder Presse können mehrere unterschiedliche Werkzeuge installiert werden, wobei der Aufbau und der Abbau einen gewissen zeitlichen Aufwand erfordert. Außerdem kann nicht jeder Mitarbeiter an jeder Maschine arbeiten, da für einige Maschinen eine besondere Fortbildung erforderlich ist. Jetzt sollen in diesem System eine Reihe von Aufträgen ausgeführt werden. Jeder Auftrag besteht aus mehreren Arbeitsschritten, die auf verschiedenen Maschinen ausgeführt werden müssen, wobei die einzelnen Arbeitschritte eine unterschiedliche Dauer haben. Das bedeutet, dass die halbfertigen Produkte irgendwo zwischengelagert werden müssen.
Das ganze Problem kann beliebig kompliziert werden. Die Maschinen können kaputtgehen, dann muss das Werkzeug auf eine andere Maschine übertragen werden, oder die Arbeit muss während der Ausfallzeit ruhen. Das Werkzeug selbst kann beschädigt werden, so dass es mehrere Stunden dauern kann, bis die Produktion wieder aufgenommen werden kann. In der Zwischenzeit müssen die Arbeiter an andere Maschinen zugewiesen werden. Dabei versucht man auch gleichzeitig mehrere Ziele zu erreichen. Man möchte die Aufträge rechtzeitig erfüllen, dabei keine Mitarbeiter ohne Arbeit lassen und die Maschinen so gleichmäßig wie möglich auslasten.
An dieser Stelle wird auf die Einzelheiten des Schedulings eingegangen. Zuerst werden einige Notationen und Definitionen vorgestellt. Anschließend das Problem des Schedulings auf die heutige Rechnerinfrastruktur übertragen und einige ausgewählte Algorithmen vorgestellt, die zu Lösung der Aufgabe verwendet werden können.

## 4.1. Scheduling in der Theorie

Viele Wissenschafter haben in der Vergangenheit versucht die Ablaufplanung zu perfektionieren. Es wurden Verfahren entwickelt, um die Zuordnung zumindest teilweise

zu automatisieren. Mit dem Aufkommen des Computers hat man versucht die „perfekte“ Zuordnung zu erstellen und festgestellt, dass man sehr schnell an die Grenzen des Machbaren stößt.
In dem allgemeinen Modell hat man eine Gesamtaufgabe (sogenannten „Job“), die aus $n$ Teilaufgaben besteht, die Tasks genannt werden. Diese Tasks möchte man an $m$ verschiede Maschinen zuordnen und dabei eine Funktion maximieren oder minimieren. Das Schedulingproblem ist also ein Optimierungsproblem und wird normalerweise als ein Tripel $\alpha|\beta|\gamma$ dargestellt. In dem $\alpha$ Feld sind die Informationen über die Maschinenumgebung gespeichert. $\beta$ beschreibt die Anzahl und die Eigenschaften der Tasks, außerdem sind hier die Restriktionen bezüglich der Ausführung angegeben. In dem letzten Feld ist schließlich die Zielfunktion angegeben, die optimiert werden soll.
In dem $\alpha$ Feld können viele verschiedene Maschineneigenschaften definiert werden. In unserer Umgebung sind viele von diesen nicht wichtig, darum wird hier nur ein kleiner Ausschnitt der möglichen Umgebungen vorgestellt.

(1) Es steht nur eine Maschine zur Verfügung. Das Scheduling von einfachen Aufgaben ist hier normalerweise trivial. Deshalb haben die Tasks hier weitere Einschränkungen in der Ausführungsreihenfolge oder haben variable Ankunftszeiten.

($P_m$) Die Umgebung hat $m$ Maschinen zur Verfügung, die alle eine identische Geschwindigkeit haben. Es macht keinen Unterschied, welchen Task man auf welcher Maschine ausführen lässt, die Laufzeit ist identisch. Diese Umgebung kann einen Rechner mit $m$ Prozessoren, oder einen Rechnercluster mit $m$ identischen Maschinen repräsentieren.

($Q_m$) Es stehen wieder $m$ Maschinen zur Verfügung. Allerdings haben die einzelnen Maschinen unterschiedliche Geschwindigkeit, die mit $v_i$ angegeben ist. Damit kann ein Rechnercluster mit unterschiedlichen Rechnern repräsentiert werden.

($R_m$) Hier sind $m$ unabhängige Maschinen vorhanden. Sie unterscheiden sich nicht nur in ihrer Geschwindigkeit, sondern auch in anderen Eigenschaften. In Folge dessen können sie die verschiedenen Tasks mit unterschiedlicher Geschwindigkeit bearbeiten. Mit $v_{ij}$ wird angegeben, wie schnell die Maschine $m_i$ den Task $n_j$ ausführen kann. Der Geschwindigkeitsunterschied kann durch unterschiedliche Rechnerstruktur kommen. Manche Prozessoren sind zum Beispiel für die Ausführung von Gleitkommaoperationen optimiert. Diese Optimierung nützt aber nichts, wenn ein Tasks nur auf Ganzzahlen arbeitet. Andere Prozessoren können einen größeren Cache Speicher haben, und so Zugriffe auf den langsamen RAM minimieren. Wenn die zu bearbeitende Datenmenge aber klein ist, bringt der zusätzliche Cache keinen Vorteil.

Jeder Task $n_j$, der zugeordnet werden soll, hat verschiedene Eigenschaften, die im Folgenden definiert werden:

**Definition 4.1** *Sei $n_j$ ein Task, der auf Maschine $m_i$ ausgeführt wurde. Mit der Laufzeit $p_{ij}$ (Processing Time) bezeichnet man die Zeitspanne, die zwischen dem Start und dem Ende der Ausführung des Tasks $j$ auf der Maschine $i$ liegt. Der Index der Maschine kann*

*weggelassen werden, falls alle Maschinen die gleiche Zeit für die Ausführung benötigen.*

**Definition 4.2** *Sei $n_j$ ein Task, der ausgeführt werden soll. Die Ankunftszeit $r_j$ (Release Date) bezeichnet dann den Zeitpunkt, zu dem dieser Task im System ankommt und für die Ausführung bereitsteht. Ist dieser Wert bei allen Tasks 0, so stehen alle zum Beginn der Ausführung zur Verfügung.*

**Definition 4.3** *Sei $n_j$ ein Task, der Fälligkeitszeitpunkt $d_j$ (Due Date) gibt an, wann der Task spätestens mit der Ausführung fertig sein soll. Das Überschreiten dieses Wertes zieht normalerweise starke Konsequenzen nach sich. Wenn diese Zeit unbedingt erfüllt werden muss, wird sie als 'Deadline' genannt.*

**Definition 4.4** *Sei $n_j$ ein Task, dann bezeichnet das Gewicht $w_j$ (Weight) die Wichtigkeit des Tasks. Je größer die Zahl, desto schneller sollte man mit der Ausführung beginnen, desto größer ist auch der Vorteil einer frühen Fertigstellung oder die Strafe beim Verpassen der Deadline.*

Neben den Taskeigenschaften können auch weitere Einschränkungen in diesem Feld stehen. Die meisten von ihnen sind aber nur für Spezialfälle interessant. Für unsere Umgebung ist nur noch der Eintrag (*brkdwn*) von Nutzen. Er gibt an, dass Maschinen während der Ausführung ausfallen können. Wobei man hier zwischen dem geplanten Herunterfahren und einem unvorhergesehen Fehler unterscheiden kann. In dem letztem Feld steht schließlich die Zielfunktion, die optimiert werden soll. Bevor diese Zielfunktionen vorgestellt werden können, müssen erst die zum Einsatz kommende Variablen vorgestellt werden.

**Definition 4.5** *Sei $n_j$ ein Task, dessen Ausführung abgeschlossen wurde, dann gibt $C_j$ den Zeitpunkt der Fertigstellung an.*

**Definition 4.6** *Sei $n_j$ ein Task mit der Due Date $d_j$. Die Latenz $L_j$ dieses Tasks ist definiert als $L_j = C_j - d_j$. Ist der Task vor der Deadline fertig, ist dieser Wert negativ.*

**Definition 4.7** *Sei $L_j$ die Latenz eines Tasks, dann ist die Tardiness $T_j$ definiert als $T_j = \max(C_j - d_j, 0) = \max(L_j, 0)$. Im Unterschied zu der Latenz, kann dieser Wert nicht negativ sein.*

Aus diesen Werten können verschiedene Zielfunktionen gebildet werden. wie zum Beispiel:

- **Maximale Latenzzeit** ($L_{max}$)
  Man versucht die maximale Latenzzeit zu minimieren. Mit dieser Funktion wird die größte Verletzung der Deadline gemessen.
- **Gesamte gewichtete Verspätung** ($\sum w_j T_j$)
  Diese Funktion summiert die Deadlineübertretungen, die mit der Wichtigkeit der Tasks gewichtet werden. Hier wird vor allem versucht wenigstens die wichtigsten Tasks vor der Deadline fertig zu stellen.

- **Makespan** $C_{max} = \max(C_1, \ldots, C_n)$
  Der Makespan gibt den Zeitpunkt an, an dem der letzte Task fertiggestellt wird. Es gilt diesen Wert zu minimieren, um dadurch eine möglichst kurze Ausführungsdauer für den gesamten Job zu erhalten. Da der Makespan nur die Zeit des letzten fertiggestellten Tasks berücksichtigt, muss man dafür sorgen dass der Unterschied zwischen den Fertigstellungszeiten der letzten $m$ Tasks möglichst gleich sind. Dadurch wird der Leerlauf der einzelnen Maschinen minimiert. Im Optimalfall beendet jede Maschine die Ausführung zur gleichen Zeit, so dass kein Leerlauf entsteht und der absolut minimale Makespan erreicht wird.

Aus diesen Angaben können viele verschiedene Schedulingprobleme beschrieben werden. Nachfolgend werden 3 Beispiele präsentiert, wobei auf Angabe der konkreten Werte für die einzelnen Variablen verzichtet wurde.

**Beispiel 1** $1|p_1 \ldots p_n; d_1 \ldots d_n; w_1 \ldots w_n| \sum w_j T_j$
*Es sollen n Tasks auf einer Maschine ausgeführt werden. Als variierende Eigenschaften haben diese Tasks die Laufzeit, das Gewicht und die Deadline. Die Aufgabe besteht darin, eine Permutation der Ausführungsreihenfolgen zu finden, so dass die gewichtete Deadline-Überschreitung minimiert wird.*

**Beispiel 2** $P_m|p_1 \ldots p_n|C_{max}$
*Hier ist ein einfacher Rechnercluster mit m identischen Maschinen vorhanden. Es sollen n Tasks, so auf diese Maschinen verteilt werden, dass der Makespan minimiert wird. Dies ist eine Standardaufgabe bei Schedulingproblemen.*

**Beispiel 3** $Q_m|p_1 \ldots p_n; brkdwn|C_{max}$
*Eine Erweiterung des vorherigen Beispiels. Hier haben die Maschinen unterschiedliche Geschwindigkeiten und können dazu noch abstürzen oder abgeschaltet werden. Dies repräsentiert eine dynamische instabile Umgebung.*

Leider sind viele der Schedulingprobleme NP-hart. Sogar die relativ einfache Aufgabe $P_m|p_j|C_{max}$ ist in NP. Das ist das klassische Makespan Problem, bei dem es gilt $n$ Tasks mit verschiedenen Laufzeiten auf $m$ identischen Maschinen zu verteilen und dabei die Gesamtlaufzeit zu minimieren. Der Beweis, dass Makespan NP-hart ist, ist durch eine Reduktion auf Bin Packing möglich.[3]

**Bin Packing**
Es sei eine Menge $n$ von Objekten gegeben. Jeder dieser Objekte hat ein Gewicht $p_j$. Außerdem sind $m$ Kisten gegeben, die eine Gewichtsschranke $b$ besitzen. Die Aufgabe besteht darin, die kleinste Gewichtsschranke $b$ zu finden, so dass eine Zerlegung der $n$ Objekte in $m$ passende Teilmengen möglich ist, so dass jeweils eine Teilmenge in eine Kiste passt.

Die Parallelen zu dem Makespan Problem sind ofensichtlich. Die $m$ Kisten entsprechen den Maschinen. Die Gewichtsschranke $b$ ist unser Makespan, der minimiert werden soll. Leider sind die heute bekannten Algorithmen für NP-Harte Probleme in der Praxis

nicht effektiv einsetzbar, da sie eine exponentielle Laufzeit zu der Größe der Eingabe haben. Darum hat man begonnen nach Alternativen zu suchen. Dabei hat man die Eigenschaften der Tasks und die Rechnerumgebung als gegeben genommen, und versucht die Ausführungszeiten für diese Umgebung zu minimieren.
Parallel dazu wurde auch die Forschung in den Szenarien betrieben, in denen die nicht Werte der Tasks und der Umgebung bekannt sind. Da man allerdings nicht mit Unbekanntem arbeiten kann, wurden in viele Fällen Wahrscheinlichkeiten für die fehlenden Werte eingeführt. Als weitere Eigenschaften kamen dann der Erwartungswert und Wahrscheinlichkeitsverteilung dieser Variablen hinzu. Aber auch diese Werte sind eher theoretischer Natur, und können in einer normalen Rechnerumgebung sehr schwer erfasst werden.

## 4.2. Scheduling in der Praxis

Leider kann die Theorie (wie so oft) nur sehr schwer in der Praxis umgesetzt werden. In der Theorie geht man davon aus, dass es für ein Problem das perfekte Scheduling existiert. Man muss es nur noch finden. Man kann sich einen Ablaufplan einfach als eine Permutation der Gesamttaskmenge anschauen. Zusammen mit der Angabe der Anzahl der Tasks auf jeder Maschine kann ein eindeutiger Ablaufplan erstellt werden. Man muss also nur noch eine Permutation der Tasks und die Variabelenbelegung für die Maschinenauslastung finden, die den Makespan minimiert. Es ist also ein Suchproblem mit festen Parametern. In der Praxis sind die Parameter sehr häufig unbekannt, so dass man keine definierte Umgebung für eine Suche hat. Dadurch kann man auch die Güte einer Taskverteilung gegenüber anderen Lösungen nicht bewerten, was eine essenzielle Voraussetzung für das Funktionieren von Suchalgorithmen ist. Die einzige Möglichkeit der Bewertung ist es, die Aufgabe auszuführen und den Makespan zu messen. Allerdings ist dann die Suche sinnlos, da sie viel länger braucht als eine Ausführung mit zufälliger Zuordnung. Darum sind viele Algorithmen, die auf der Suche basieren nur in ganz speziellen Umgebungen in der Praxis einsetzbar.
Die schwierigste Aufgabe in der Praxis ist es, korrekte Parameter für den Schedulingalgorithmus zu erhalten. Das fängt schon bei der Tasklaufzeit an. In der Industrie, wo die Tasks oft wiederholt werden und einen eindeutige Kennung haben, ist es ohne Probleme möglich, ein mal die Dauer der Ausführung zu messen und diesen Wert dann für das spätere Scheduling zu verwenden. Beim Data Mining kann dies nicht gemacht werden, da hier normalerweise jedes mal eine andere Eingabe verwendet wird. Es werden sehr selten zwei absolut identische Jobs ausgeführt. Sogar wenn man die Zeit misst, kann ein einfacher bedingter Sprung, der von der Art der Eingabe abhängt, die Laufzeit stark verändern. Deshalb ist es auch nicht möglich die Laufzeit mit „10.000 Gleitkommaoperationen, 500 Ganzzahloperationen und 2000 Vergleiche“ anzugeben. Aus diesem Grund fragen viele Algorithmen einfach den Benutzer nach der Laufzeit. Es hat sich aber herausgestellt, dass die Benutzerangaben häufig sehr weit daneben lagen. [6] Als ein Extremfall wird angegeben, dass ein Task, der mehrere Stunden für die Ausführung brauchen sollte, innerhalb weniger Sekunden fertig war. Ein Grund für die falschen Angaben ist die

Eigenschaft vieler Systeme die Tasks, die die angegebene Laufzeit überschritten haben, einfach abzubrechen. [23][8] Dadurch konnte ein mal erstellte Ablaufplan auf jeden Fall eingehalten werden, zwang die Benutzer aber sehr vorsichtige Angaben über die Laufzeit zu machen, was die Qualität des erstellten Ablaufplans verschlechterte. Durch die, zu früh fertig gewordenen Tasks, entstanden viele kleine Löcher, die nur schwer zu füllen waren. Ein weiteres Problem bei der Laufzeitabschätzung ist die Beobachtung, dass ein Task einen Rechner nie zu 100% für sich allein hat. Daneben gibt es mehrere Prozesse des Betriebssystems, die nebenläufig ausgeführt werden. Auch die eingesetzte Programmiersprache spielt eine Rolle. Bei Java zum Beispiel, kann der Garbage Collector, der für die Löschung, der nicht mehr gebrauchter Objekte, zuständig ist und den belegten Speicher wieder freigibt, unterschiedlich viel Zeit beanspruchen. Das alles führt dazu, dass sogar auf dem gleichem System ohne externe Einflusse die Laufzeit identischer Tasks variieren kann. Allerdings wird die exakte Laufzeit gar nicht benötigt, meistens reicht es zu wissen, wie sich die Laufzeit des aktuellen Tasks im Verhältnis zu anderen Tasks steht.
Ähnliches gilt für die Geschwindigkeit der Rechner. Es gibt zwar Versuche die zukünftige Leistungsfähigkeit der Rechner vorherzusagen, [36] [35] aber eine grobe Einschätzung der Leistungsfähigkeit kriegt man schon durch das Ausführen eines Benchmarks. Dabei kann man zum Beispiel messen, wie lange der Rechner für eine bestimmte Aufgabe braucht, oder wie oft bestimmte Berechnungen in einer festen Zeitspanne ausgeführt werden. Leider erhält man dadurch die Ergebnisse, die nur für den Benchmark stimmen. Jeder Aufgabe setzt sich aus verschiedenen Operationen zusammen, die unterschiedliche Zeit zur Ausführung brauchen. Einen großen Unterschied macht es zum Beispiel, ob man mit Gleitkomma- oder Ganzzahlen rechnet. Stimmen die Verhältnisse zwischen dem Benchmark und dem späteren Task nicht, ergibt es ein falsches Bild von dem Rechner. Noch schwieriger wird es die Ausführungsgeschwindigkeit von einem Task auf einem bestimmten Rechner anzugeben. Da man schon die beiden Einzelwerte nicht zuverlässig ermitteln kann, ist eine korrekte Angabe dieses Wertes fast unmöglich.
Nachdem man die Parameter erhalten hat, stellt sich sofort das nächste Problem. Alle Informationen über die Umgebung sind Momentaufnahmen. Vor allem die Leistung der Rechner kann sich durch Last von anderen Nutzern verändern. Auch kann ein Rechner komplett ausfallen. Die meisten Algorithmen berücksichtigen diese Änderungen nicht, andere setzen voraus, dass man zumindest die Wahrscheinlichkeit für den Ausfall und die Verteilung der Last kennt. Auch diese beiden Werte kann man in einem offenen System nicht kriegen. Es kann sich zum Beispiel jeder Zeit ein anderer Nutzer „remote“ auf einen Rechner einlogen. Auch aus der Tageszeit können keine Schlüsse gezogen werden, da es genug Menschen gibt, die auch Nachts arbeiten, oder sich aus einer anderen Zeitzone auf den Rechner einloggen. Man muss also jeder Zeit mit dem Wegfall eines Rechners oder den Anstieg der Ausführungszeiten einzelner Tasks rechnen.
Auch benutzen die meisten Algorithmen ein sehr vereinfachtes Modell der Umgebung. Dies stellt im Normalfall keine Probleme dar, da die weggelassenen Parameter entweder nicht ins Gewicht fallen oder nur sehr schwer einzuschätzen sind.

# 5. Schedulingalgorithmen

Die Schedulingalgorithmen kann man in mehrere Klassen einordnen. Die Abbildung 5.1 zeigt die einzelnen Klassen. Auf der obersten Ebene kann eine Unterscheidung zwischen den statischen und den dynamischen Verfahren gemacht werden. Die dynamischen Verfahren können weiter gegliedert werden. Der Hauptunterschied ist dann der Einsatz der Migration.
Die einfachste Klasse ist die Klasse der statischen Algorithmen. Die Parameter für diese Verfahren werden schon zur Programmier- und Kompilierzeit festgelegt. Die Verteilung selbst findet nur zum Beginn der Ausführung statt. Danach kann diese Zuordnung nicht mehr geändert werden. Dadurch ergeben sich einige Nachteile. Die Information über die Umgebung und die auszuführende Arbeit muss zum Beginn der Ausführung vorliegen und korrekt sein. Auch darf sie sich während der Ausführungszeit nicht ändern, da es sonst zur großen Abweichungen zwischen der errechneten und der tatsächlichen Ausführungszeit kommen kann.
Obwohl die Algorithmen unflexibel sind, können sie ohne Probleme in statischen (dedizierten) Umgebungen eingesetzt werden. Vor allem in eingebetteten und Echtzeit-Systemen ist die Umgebung klar definiert. Wenn man sich allerdings in einer dynamischen Umgebung befindet, wo es keine Garantien für die Ausführungszeit der Tasks und die Leistungsfähigkeit der Rechner geben kann, versagt der statische Ansatz sehr schnell.
Darum wurden sogenannte „dynamische“ Algorithmen entwickelt, um die Flexibilität zu erhöhen. Im Gegensatz zu den statischen Verfahren wird bei dem dynamischen Ansatz die Zuordnung der Tasks während der Laufzeit durchgeführt. Dadurch kann die Verteilung dynamisch auf die Umgebung angepasst werden. Zum Beginn der Ausführung wird nur ein Teil der Tasks zugeordnet. Das weitere Verhalten hängt sehr stark von den Eigenschaften des Systems ab. Dadurch ist es viel einfacher sich an eine verändern-

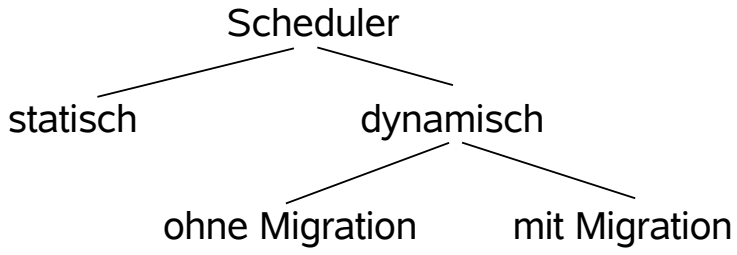



Abbildung 5.1.: Einteilung der Schedulingverfahren in Klassen

de Umgebung anzupassen. Auch die Genauigkeit der Information zum Startpunkt der Ausführung ist nicht mehr so wichtig, da man Fehler, die zum Beginn gemacht wurden sehr oft im Verlauf der weiteren Ausführung korrigieren kann. Die Größe der initialen Zuordnung spielt eine wichtige Rolle und ist immer ein TradeOff. Werden zu viele Tasks verteilt, können, die vielleicht bei der Zuordnung gemachten Fehler, nicht mehr korrigiert werden. Ist die Zuordnung zu klein, sind die einzelnen Arbeiter sehr schnell mit den Aufgaben fertig und befinden sich im Leerlauf, bevor die nächste Aufgabenserie eintrifft. Um falsche Entscheidungen zu revidieren, kann die sogenannte „Migration“ eingesetzt werden. Dabei wird eine schon zugeordnete Aufgabe von einem Rechner zurückgenommen und zu einem anderen Rechner delegiert. Das verkompliziert die Ablaufplanung, da jetzt weitere Informationen von den externen Rechnern benötigt werden. Es muss vor allem der aktuelle Fortschritt des Tasks bekannt sein, um die möglichen Vorteile der Migration richtig einschätzen zu können.

## 5.1. Einfache Statische Algorithmen

Zuerst sollen hier einige einfache statische Algorithmen vorgestellt werden. Wie alle statischen Algorithmen, verteilen sie die komplette Arbeit zum Beginn der Programmausführung und sind danach untätig. Normalerweise gehen diese Algorithmen von einer statischen Rechneranzahl aus. Das Verhalten bei einem Rechnerausfall ist daher nicht definiert.

### 5.1.1. Fixed Scheduling ($FIXED$)

Der Fixed Scheduling Algorithmus weist jedem Rechner die gleiche Anzahl an Tasks zu. Für die Verteilung nimmt dieses Verfahren nur zwei Werte. Die Anzahl der Maschine $m$ und die Anzahl der zu verteilenden Tasks $n$. Es wird ein Quotient $Q = n/m$ berechnet, der angibt, wie viele von den $n$ Tasks an jede Maschine übertragen werden sollen. Jeder Worker kriegt somit den gleichen Anteil an den Tasks. Die Implementierung dieses Verfahrens kann mit zwei Listen geschehen. Man geht die Liste der Arbeiter durch und fügt bei jedem einen Task hinzu bis keine offenen Tasks mehr da sind. Falls die Liste zu Ende ist, wird sie wieder von vorne durchgearbeitet. Dieser Algorithmus benötigt keine Vorverarbeitung und jeder Schritt kann in konstanter Zeit durchgeführt werden. Die Gesamtlaufzeit beträgt somit $O(n)$. Auch braucht er keinen zusätzlichen Speicher, da die Worker- und die Taskliste bereits vorhanden sind und zu der Eingabe zählen.

### 5.1.2. Random Scheduling ($RANDOM$)

Dieses Verfahren wählt den Rechner für jeden freien Task zufällig aus. Dabei werden überhaupt keine Informationen über die Aufgaben oder die Rechnerumgebung benötigt. Dabei wird jeder Rechner mit der gleichen Wahrscheinlichkeit für die Taskausführung ausgewählt. Bei $m$ Rechnern ist dann die Wahrscheinlichkeit eines Tasks auf einem bestimmten Rechner ausgeführt zu werden $1/m$.

Die Zuordnungen, die dieser Algorithmus erstellt, sollten mit einer ausreichenden Anzahl an Tasks die gleiche Anzahl an Aufgaben an jeden Rechner zuordnen und damit der Verteilung des $FIXED$ Verfahrens sehr ähnlich sein. Allerdings kann hier durch die Zufallskomponente für keine Rechnerumgebung eine ausgewogene Verteilung garantiert werden. Im Extremfall können alle Tasks an den langsamsten Rechner zugeordnet werden, und die Laufzeit gegenüber der lokalen Ausführung sogar vergrößern.

Dieser Algorithmus wurde nur ausgewählt, um einen Vergleich für die mögliche Worst Case Laufzeit der Algorithmen zu haben. Für den normalen Einsatz ist dieses Verfahren durch den kompletten Zufall unbrauchbar.

Da für die Berechnung der Zufallszahlen normalerweise eine Folge von Funktionswerten verwendet wird, ist jeder Schedulingschritt in einer konstanten Zeit möglich. Da ansonsten keine Berechnungen ausgeführt werden, beträgt die Gesamtlaufzeit dieses Algorithmus $O(n)$. Ähnlich verhält es sich mit dem Speicherverbrauch. Außer der Eingabe wird kein zusätzlicher Speicher benötigt.

### 5.1.3. Fixed mit Gewichtung ($FIXED - W$)

In den ersten beiden Algorithmen wurden keine Informationen aus der Umgebung verwendet. An eine Information kommt man relativ einfach heran. Es ist die Leistungsfähigkeit der einzelnen Rechner im Netz. Diese kann durch ein Benchmarkprogramm, das auf dem Zielrechner läuft, festgestellt werden. Diese Daten kann man verwenden um die Rechner beim Zuordnen der Tasks zu gewichten und das Scheduling anhand dieser Gewichte durchzuführen.

Nachdem man die Gewichte $v_i$ von den Maschinen bekommen hat, bildet man die Summe um die Gesamtrechenkapazität $v_{ges}$ zu erhalten. Die Formel für Verteilung ist dann wie folgt:

$$n_j = \frac{n * v_i}{v_{ges}} = \frac{n * v_i}{\sum_{i=1}^{m} v_i} \tag{5.1}$$

Die Leistung des Arbeiters wird mit der Taskanzahl multipliziert und die Gesamtleistung geteilt. So entspricht die Anzahl der Tasks, die an jeden Rechner übermittelt wird, seiner Leistung. Der Erfolg dieses Algorithmus hängt sehr stark von der Genauigkeit des Leistungsindikators. Wenn das Verhältnis nicht stimmt, kommt es sogar in statischen Umgebungen sehr schnell zu unausgewogener Verteilung, und somit zu einer deutlich höheren Laufzeit der Gesamtaufgabe.

Auch dieser Algorithmus zeichnet sich durch eine kurze Laufzeit aus. Die Gesamtleistung am Anfang wird in der Zeit $O(m)$ berechnet. Die Zuordnung eines Tasks zu einer Maschine kann in einer konstanten Zeit durchgeführt werden. Somit beträgt die Gesamtlaufzeit dieses Verfahrens $O(m+n)$. Der zusätzliche Speicherverbrauch kommt durch die Aufbewahrung der Benchmarkdaten für jeden Rechner, so dass ein zusätzlicher Bedarf von $O(m)$ entsteht.

### 5.1.4. Zufällig mit Gewichtung ($RANDOM - W$)

Dieser Algorithmus erweitert die „normale“ zufällige Zuordnung mit den Gewichten der einzelnen Arbeiter. Das Ziel ist es, die Verteilung zwar zufallsbasiert ablaufen zu lassen, die Wahrscheinlichkeit für die einzelnen Rechner sollte aber proportional ihrer Leistung sein. Das heißt, die Wahrscheinlichkeit für einen Task an einem Rechner mit der Leistung $v_i$ zugewiesen zu werden, $F(n_j) = v_i/v_{ges}$ betragen soll. Das Problem hier ist die Zuordnung zwischen der Zufallszahl und einem Rechner. Wenn die Gesamtleistung $b_{ges}$ bekannt ist, kann man die ermittelten Zufallszahlen auf die Werte zwischen 1 und $b_{ges}$ beschränken. Den richtigen Rechner zu finden, ist dann in linearer Zeit in Verhältnis zu der Maschinenanzahl möglich. Durch die Sortierung der Rechner nach ihrer Leistungsfähigkeit und/oder das Abspeichern der Leistungsfähigkeit in einem Baum kann man die Zeit für die Rechnerauswahl auf $O(\log m)$ drücken. Da sich die Anzahl der Maschinen normalerweise in einem einstelligen oder niedrigem zweistelligen Bereich befindet lohnt sich der erhöhte Initialisierungsaufwand jedoch nicht.
So braucht dieses Verfahren eine Vorverarbeitungszeit von $O(m)$ für die Berechnung der Gesamtleistung, $O(m)$ für einen Schedulingschritt. Die Gesamtlaufzeit beträgt dann $O((n + 1) * m)$. An zusätzlichem Speicher wird eine Liste mit den Gewichten der $m$ Maschinen benötigt.

## 5.2. Heuristiken

Da das Problem einen minimalen Makespan für ein Schedulingproblem zu finden NP-hart ist, wurde versucht Algorithmen zu entwickeln, die trotz ihrer polynomiellen Laufzeit eine bestmögliche Approximation errechnen können. Diese Art von Algorithmen verlässt sich darauf, dass alle Umgebungsparameter bekannt und auch korrekt sind. Anhand dieser Parameter wird versucht eine Zuordnung zu berechnen, die bei einer sich nicht veränderbaren Umgebung eine gute Laufzeit garantiert.
Die heuristischen Verfahren lassen sich sehr gut in der Güterfertigung einsetzten, da die gleichen Tasks dort sehr oft wiederholt werden, und es so möglich ist für jeden Task die Ausführungszeit zu messen. Auch bleibt die Anzahl und die Leistung der Maschinen gleich, so dass man davon ausgehen kann, dass einen erstellte Zuordnung für eine längere Zeit gültig bleibt.
In einer Computerumgebung hingegen ist es sehr schwierig oder sogar unmöglich die korrekten Parameter für diese Heuristiken zu kriegen. Auch gehen die meisten Heuristiken davon aus, dass eine Maschine für manche Tasks besser geeignet ist, als für andere und man diese Unterschiede in Zahlen ausdrücken kann.
Dies ist in unserer Umgebung nicht der Fall. Wir gehen davon aus, dass wenn auf einer Maschine $m_i$ gilt $p_{ia} > p_{ib}$ , es keine andere Maschine $m_k$ gibt, mit $p_{ka} < p_{kb}$. Dies ermöglicht uns eine Vereinfachung der Algorithmen. So wird im Originalverfahren eine $n * m$ Matrix vorausgesetzt, in der für alle Maschinen/Task Kombinationen die entsprechende Laufzeit eingetragen ist. Diese Matrix wird nicht mehr benötigt, da die tatsächliche Laufzeit aus der erwarteten Tasklaufzeit und der Leistungsfähigkeit der Maschine berechnet werden kann.

Es gibt viele Forscher, die sich mit den einzelnen Heuristiken auseinander gesetzt und deren Leistung verglichen haben. [4][2][21] Alle diese Vergleiche setzen die Parameter für die Umgebung und die Tasks als gegeben voraus. Deshalb können auch zum Beispiel evolutionäre Verfahren für das Scheduling eingesetzt werden. Das ist in unserer Umgebung nicht der Fall. Für die Implementierung wurden nur solche Verfahren ausgewählt, die auch ohne Probleme eingesetzt werden können.

Ein mögliches Verfahren ist es, die Tasks zu der Maschine zuzuordnen, die die kürzeste Ausführungszeit für diesen Task hat. So wird garantiert, dass dieser Task in der möglichst kürzesten Zeit ausgeführt wird. Leider kann es dadurch zu sehr großen Unausgewogenheit zwischen den einzelnen Maschinen führen, wenn alle Tasks auf der gleichen Maschine die schnellste Ausführungszeit haben. Alle anderen Maschinen bleiben dann leer. Im Folgenden sollen 3 weitere Verfahren vorgestellt werden, die eine bessere Verteilung ermöglichen.

### 5.2.1. MCT Heuristik ($MCT$)

Die Minimum Completion Time (MTC) Heuristik versucht die Tasks auf die Maschinen zuzuweisen, auf denen diese Tasks voraussichtlich am frühesten fertiggestellt werden. Dass kann dazu führen, dass manche Tasks zu Maschinen zugeordnet werden, die nicht die schnellste Ausführungszeit für sie haben. Allerdings ist die Last über alle Rechner mehr oder weniger gleich verteilt. Die Reihenfolge der Zuordnung spielt bei diesem Verfahren keine Rolle. Die Tasks werden in der gleichen Reihenfolge, in der sie erstellt wurden auf die Rechner verteilt.
Um gute Ergebnisse mit diesem Verfahren zu ereichen, müssen Informationen über die Last auf dem Worker durch andere Master vorhanden sein. Ansonsten kann der tatsächliche Zeitpunkt der Fertigstellung des Tasks nicht berechnet werden.
Dieser Algorithmus braucht keine Vorverarbeitungszeit. In jedem Schedulingschritt muss die Fertigstellungszeit des Tasks für jede Maschine kalkuliert und dann das beste Ergebnis ausgewählt werden. Dies geschieht in der Zeit $O(m)$. Speichern muss man wie immer die Leistungsfähigkeit der Rechner, aber auch die aktuelle Last. Außerdem braucht man die erwartete Laufzeit der Tasks, was zu einem Gesamtspeicherverbrauch von $2 * m + n$ führt.

### 5.2.2. Min-Min Heuristik ($MINMIN$)

Die Min-Min Heuristik geht einen Schritt weiter, als die MCT Heuristik. Pro Schedulingschritt wird nicht nur ein Task, sondern alle noch offenen Tasks angeguckt. Zuerst wird für jeden Task die Maschine gefunden, die diesen Task am frühesten fertig stellen kann. Dann wird aus dieser Paarenmenge das Paar genommen, dass die Last einer Maschine am wenigsten verändert, dass also die kleinste erwartete Laufzeit hat. Der ausgewählte Task wird aus der Taskmenge herausgenommen und an die entsprechende Maschine zugeordnet. Es geht weiter mit den verbleibenden Tasks. Die naive Implementierung dieser Heuristik ist rechenintensiv, da in jedem Schritt jede Task/Maschine Kombination ge-

prüft werden muss. Dies gilt aber nur für den Fall, dass jede Maschine nicht für jeden Task gleich gut geeignet ist. In unserem Fall kann nur ein Task mit der minimalen erwarteten Laufzeit ausgewählt werden, da dieser Task auf allen Maschinen die minimale Ausführungszeit hat. Es reicht also die Tasks nach ihrer erwarteten Laufzeit aufsteigend zu sortieren, und dann mit der MTC Heuristik weiterzumachen. Für die Sortierung von $n$ Elementen braucht man eine Zeit von $O(n \log n)$. Ansonsten ist die Laufzeit und der Speicherverbrauch mit der MTC Heuristik identisch.
Diese Heuristik soll erreichen, dass so viele Ergebnisse so schnell wie möglich zur Verfügung stehen. Die größeren Tasks bewahrt man sich bis zuletzt auf. Leider verschafft uns diese Eigenschaft bei unseren Experimenten keinen Vorteil. Die Eingabegröße für die Tasks hängt nicht mit der Laufzeit zusammen, also kann sich der Leerlauf einer Maschine zwischen der Ausführung zweier Tasks nur vergrößern. Auch die ersten Ergebnisse nutzen nicht viel, da die Nachbearbeitung der Ergebnisse beim Master sehr schnell ist. Außerdem hat die MinMin Heuristik einen unangenehmen Nebeneffekt. Die langsamen Rechner kriegen weniger Arbeit, als die schnellen, und sind die meiste Zeit untätig. Dieses Verhalten kann man an einem sehr einfachen Beispiel zeigen. Wir nehmen an, dass 3 Tasks mit den Laufzeiten 3,6,9 verteilt werden sollen. Zur Verfügung stehen 2 Maschinen $m_1$ mit einer Geschwindigkeit $v_1$ von 3 und $m_2$ mit einer Geschwindigkeit $v_2$ von 1. Den Ablauf kann man in der Tabelle unten sehen. Die ersten beiden Spalten mit den Maschinen geben die voraussichtliche Last auf den Maschinen nach dem Schedulingschritt an, falls die jeweilige Maschine gewählt wurde. Die Spalte in der Mitte enthält die gewählte Maschine. Die beiden hinteren Spalten zeigen die tatsächliche Last nach den Schedulingschritt an.

| Schritt | Task | $m_1$ | $m_2$ | Wahl | $m_1^{neu}$ | $m_2^{neu}$ |
|---|---|---|---|---|---|---|
| Schritt 1 | $n_1$ | 1 | 3 | $m_1$ | 1 | 0 |
| Schriit 2 | $n_2$ | 3 | 6 | $m_1$ | 3 | 0 |
| Schritt 3 | $n_3$ | 6 | 9 | $m_1$ | 6 | 0 |

Wie man sieht, beträgt die Last der ersten Maschine 6. Die zweite Maschine bleibt leer. Bei einer perfekten Zuordnung würde $m_2$ den Task $n_1$ kriegen und eine Last von 3 haben. Auf $m_1$ würde die Last mit den Tasks $n_2$ und $n_3$ den Wert von 5 betragen. Dieser Effekt verhindert effektiv, dass lange Tasks an den langsamen Rechnern ausgeführt werden. Dies kann den Makespan verbessern, wenn die Tasklaufzeiten unbekannt sind oder geschätzt werden.

### 5.2.3. Min-Max Heuristik ($MINMAX$)

Die Min-Max Heuristik ist der Min-Min Heuristik sehr ähnlich. Auch hier werden alle Task/Maschinen Kombinationen angeguckt und die Maschine mit der frühesten Fertigstellungszeit genommen. Aus diesen Paaren wird aber, nicht wie Min-Min der kleinste, sondern der größte Task ausgewählt und der entsprechenden Maschine zugeordnet. Anschließend macht man mit den restlichen Tasks weiter. Auch die Implementierung dieses

Verfahrens ist dem Min-Min Scheduling sehr ähnlich. Man muss bei der initialen Sortierung der Tasks die Reihenfolge nicht aufsteigend, sondern absteigend wählen, so dass der Task mit der größten erwarteten Laufzeit vorne steht.
Das Ziel dieses Verfahrens ist es, am Anfang die langen Tasks an die schnellen Rechner zuzuweisen. Danach die kürzeren an die langsamen Rechner, die noch leer, beziehungsweise sehr kleine Last haben. Dadurch könnten aber den langsamen Rechnern viele sehr kurze Tasks zugeordnet werden. Wenn die Angaben zu der erwarteten Tasklaufzeit ungenau oder falsch sind, kann es sich negativ auf den Makespan auswirken. Die Anforderungen an die Rechenzeit und der Speicherverbrauch ist mit der Min-Min Heuristik identisch.

## 5.3. Batch Allocation Lösungen

Batch Allocation Lösungen sind eine Reihe von dynamischen Algorithmen, die in den 80er Jahren aufkamen, als man angefangen hat bessere Lösungen für das Scheduling der Berechnungen auf Mehrprozessorsystemen zu suchen. Entwickelt wurden diese Verfahren hauptsächlich, um lange Schleifen für die Parallelverarbeitung auf mehrere Prozessoren zu verteilen. Die parallelen Rechensysteme dieser Zeit hatten eine „shared-memory“ Architektur, hatten also einen gemeinsamen Arbeitspeicher, auf den alle Prozessoren zugreifen konnten. Die statischen Ansätze waren nicht zufriedenstellend, da man die Laufzeit der einzelnen Schleifendurchläufe nicht vorhersagen konnte. Würde man die Tasks schon vor Beginn der Ausführung auf die einzelnen Prozessoren verteilen, könnte sich ein sehr großer Unterschied in den Rechenzeiten der Prozessoren ergeben. Also fing man an, die Menge der Aufgaben zur Laufzeit in kleinere Teilmengen zu unterteilen und dynamisch den einzelnen Prozessoren zuzuordnen. Allerdings war es eigentlich keine „Zuordnung“. Wenn ein Prozess seinen Teil der Schleife abgearbeitet hat, hat er sich aus dem Speicher den nächsten Teil geholt. Durch die dynamische Verteilung entstand aber ein Problem: mehrere Prozesse konnten zur gleichen Zeit auf die gemeinsamen Variablen (vor allem dem Zähler für die abgearbeiteten Schleifendurchläufe) zugreifen. Diese Zugriffe mussten synchronisiert werden, so dass es für jeden Prozessor eine Zeitspanne gab, in der er auf die Ressourcen wartete. Es wurden Algorithmen entwickelt, die versuchten diesen Overhead zu minimieren. Sie alle versuchen anstatt nur einen Task, sofort einen größeren Block an einen Prozessor zuzuweisen, und so den Synchronisationsoverhead zu minimieren. Allerdings stieg mit der Blockgröße auch die Gefahr, die Arbeit ungleichmäßig auf die einzelnen Prozessoren zu verteilen. Und so wurde von vielen Wissenschaftlern ein TradeOff zwischen dem Overhead und der Ungleichverteilung gesucht.
Das von mir verwendete System, arbeitet zwar nicht nach den Shared Memory Prinzip, es gibt aber auch hier eine Zeit, in der die Arbeiter auf die Zuweisung von neuen Tasks warten. Dabei ist die Situation analog.
Jedes Mal wenn ein Task fertig ist, muss das Ergebnis an den Master geschickt und ein neuer Task angefordert werden. So entstehen Pausen nach der Fertigstellung von jedem Task. Es lohnt sich auch hier die Tasks nicht einzeln, sondern in Gruppen zu übertragen. So kann der Rechner sofort mit dem nächsten Task weitermachen, ohne das Ende der Übertragung abzuwarten. Aber auch hier entsteht die Gefahr durch eine zu

ungleichmäßige Verteilung an die einzelnen Worker.

### 5.3.1. Die Gewichtung

Die Originalalgorithmen waren darauf ausgelegt auf Mehrprozessormaschinen mit identischen Prozessoren oder Rechnerclustern mit identischen Rechnern zu laufen. In heutigen verteilten Systemen ist die Rechenleistung der einzelnen Maschinen oft sehr unterschiedlich. Um die Algorithmen auch in heterogenen Umgebungen benutzen zu können, muss die Blockgröße an die Rechenleistung bei der Verteilung angepasst werden. Als Indikator für die Rechenkapazität nehmen wir hierbei die Ergebnisse des Benchmarks. Ziel des Gewichtungsfaktors ist es, den überdurchschnittlich schnellen Rechnern mehr Aufgaben, und denn langsamen Rechnern weniger Aufgaben zukommen zu lassen. Es wird hier, der bei [29] vorgestellte Gewichtungsvektor verwendet.
Um die Gewichtung zu bekommen holt man sich die Leistungsinformationen $v_i$ jedes einzelnen Rechners. Daraus wird die Gesamtleistung $v_{ges} = \sum_{i=1}^{m} v_i$ und der Leistungsanteil $R_i = v_i / v_{ges}$ jeder einzelnen Maschine berechnet. Diese Gewichtung ist noch nicht vollständig. Sie gibt zwar den Leistungsanteil an der Gesamtleistung an, da wir sie aber nur auf eine Teilmenge von Tasks anwenden wollen, wäre die Blockgröße viel zu klein. Also wird sie noch mit der Maschinenanzahl multipliziert. Die Gesamtformel für die Gewichtung lautet dann:

$$GF_i = \frac{m * v_i}{\sum_{i=1}^{m} v_i} \tag{5.2}$$

Diese Gewichtung wird bei jeder Berechnung der Blockgröße verwendet und modifiziert diese entsprechend. Dabei ist sie für überdurchschnittlich schnelle Rechner größer und für unterdurchschnittlich langsame Rechner kleiner als eins. Da die Anzahl der Tasks immer eine ganze Zahl sein muss, kann mal leider nicht verhindern, dass Ungenauigkeiten bei der Rundung auftreten, was bei Aufgaben mit sehr wenigen zu verteilenden Tasks zu einem Ungleichgewicht in der Rechnerlast führen kann. Auch ist die Mindestblockgröße auf eins festgelegt und wird entsprechend aufgerundet. Es kann also nicht sein, dass ein langsamer Rechner durch die geringe Gewichtung keinen Task zugeordnet bekommt.

### 5.3.2. Work Queue ($WQ$)

Die Work Queue (auch Self Scheduling genannt [32]) ist das einfachste dynamische Verfahren für die Zuordnung der Tasks an die Worker. Hier wird die Bockgröße konstant auf 1 gesetzt. Weder die Anzahl der Rechner, noch ihre Leistung wird zu Berechnung herangezogen. Das heißt, man braucht auch keine Gewichtung, die die Blockgröße anpasst. Wenn es einen Arbeiter ohne Arbeit gibt, wird genau ein Task an den „Arbeitslosen“ übermittelt. Wenn es keine Tasks mehr zu verteilen gibt, wartet man auf die restlichen Ergebnisse und beendet die Berechnungen. Dieses Verfahren ist sehr flexibel, was Schwankungen in der Rechnerleistung angeht. Der Unterschied im Makespan beträgt maximal die Laufzeit des längsten Tasks auf dem langsamsten Rechner. Leider sind die Leerlaufzeiten durch die Kommunikation sehr groß, da nach jedem Task eine Pause für das Senden der Ergebnisse und das Empfangen von neuen Daten eingelegt werden muss.

Die Laufzeit dieses Verfahrens ist linear. Weder die Blockgröße, noch die Gewichtung muss berechnet werden, und das Zuweisen der einzelnen Tasks an die Rechner geschieht in konstanter Zeit. An zusätzlichem Speicher wird auch nichts benötigt.

### 5.3.3. Fixed Size Chunking und Guided Self Scheduling

Zwei Verfahren sollen hier nur am Rande erwähnt werden. Der Fixed Size Chunking Algorithmus [18] kommt der Funktionalität des Self Scheduling Algorithmus sehr nah. Auch er wählt eine konstante Blockgröße, allerdings größer eins. Es wird versucht anhand der Anzahl der Worker und der Anzahl der Tasks eine „perfekte“ Blockgröße für die Ausführung zu berechnen. Diese liegt zwischen der $n/m$ Größe des Fixed Scheduling Verfahrens und der minimalen Größe von 1 der Work Queue und wird während der gesamten Ausführungszeit beibehalten. Dadurch ist dieser Algorithmus für dynamische Umgebungen ungeeignet, da er einen größeren Block Tasks an einen langsamen Rechner am Ende der Ausführung zuweisen kann. Auch kann er die möglichen Schwankungen in der Laufzeit der einzelnen Tasks nicht ausgleichen, da die Blockgröße fest ist. Obwohl dieser Ansatz einige Nachteile hat, wird in diese Richtung immer noch geforscht, um die perfekte Blockgröße zu finden. [37] Allerdings wird dabei nicht auf eine dynamische Umgebung eingegangen.

Die Lösung des Problems besteht in einer variablen Blockgröße. Zum Beginn der Ausführung wählt man die Blockgröße ziemlich hoch und lässt sie bis zu einem gewissen Mindestwert sinken. Diesen Mindestwert behält man bis zum Ende der Ausführung bei. Dieses Vorgehen reduziert den Kommunikationsoverhead, beziehungsweise die Latenzzeit zum Beginn der Ausführung. Auch die ungleiche Lastverteilung zwischen den einzelnen Rechnern wird durch die kleine Blockgröße am Ende minimiert.

Ein Algorithmus, der von der abnehmenden Blockgröße gebrauch macht, ist Guided Self Scheduling. [25] Vorgestellt wurde dieses Verfahren von Polychronopoulos und Kuck im Jahr 1987. Die Blockgröße bewegt sich zwischen den beiden Extremen $n/m$ und 1. Das Verfahren funktioniert ziemlich einfach. Man initialisiert die aktuelle Iteration $i$ mit 1 und eine Variable $R_i = n$ mit der Anzahl der auszuführenden Tasks. Dann wird die Blockgröße mit $x_i = \lceil R_i/m \rceil$ berechnet. Danach muss das $R$ für die nächste Iteration durch $R_{i+1} = R_i - x_i$ angepasst werden. Das neue $R_i$ ist also die Anzahl der noch nicht zugeordneten Tasks. Das ganze wiederholt man so lange, bis es keine Tasks mehr zu verteilen gibt. Wenn man zum Beispiel 100 Tasks auf 5 Rechner verteilen will, beträgt die Blockgröße der ersten 5 Iteration 20, 16, 13, 10 und 9. Das entspricht einer exponentiell fallenden Funktion und die Blockgröße der letzten $m - 1$ Iterationen beträgt 1.

Dieses Verfahren bringt zwei Probleme mit sich. Die initiale Blockgröße von $n/m$ ist zu groß. Falls die Tasks eine fallende Ausführungszeit haben, wird die erste Maschine mit den Berechnungen zuletzt fertig. Das zweite Problem liegt in der zu schnell fallenden Blockgröße. Der initiale Block für die letzte Maschine ist gerade mal halb so groß, wie der Block für die erste Maschine. Das erhöht unnötig die Anzahl der Iterationen und sorgt für zusätzlichen Overhead. Dieser Overhead wird durch die letzen Iterationen mit der Blockgröße von 1 noch verstärkt.

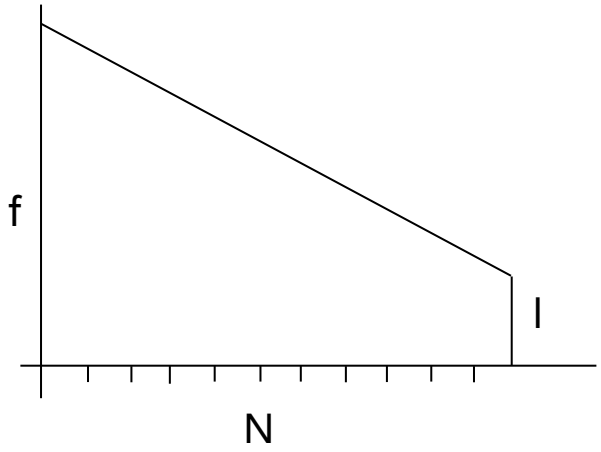



Abbildung 5.2.: Entwicklung der Blockgröße bei TSS. $f$ und $l$ sind die Start und die Endblockgrößen. $N$ ist die Anzahl der Schedulingschritte. Wenn man das Bild um 90 Grad dreht nach links dreht, wird die Trapezform offensichtlich.

### 5.3.4. Trapezoidal Self Scheduling ($TSS$)

Der Trapezoidal Self Scheduling Algorithmus [33] ist dem Guided Self Scheduling ähnlich. Auch hier verfolgt man die Strategie der abnehmenden Blockgrößen. Allerdings verwendet man keine exponentielle Funktion für die Reduzierung der Blockgröße, sondern eine lineare. Die initiale Blockgröße wird vom Benutzer vorgegeben. Die Berechnung des Gefälles der Blockgrößenfunktion wird vor Beginn des Schedulings durchgeführt. Da es sich um eine lineare Funktion handelt ist deren Steigung eine negative Konstante. Also muss die Blockgröße während jedes Schedulingschrittes um diese Konstante vermindert werden. Für seine Ausführung braucht der Trapezoid Self Scheduling zwei Parameter. Der Wert $f$ legt die initiale Blockgröße und der Parameter $l$ die Endblockgröße fest. Auf dem Bild 5.2 sieht man die Funktion der Blockgröße Die X-Achse gibt die Anzahl der Schedulingschritte an, also die Anzahl der Blocke, die verteilt werden. Die Y-Achse stellt die Größe der einzelnen Blocke dar. Wenn an das Bild um 90 grad dreht, ergibt das ganze einen Trapez mit den Längen $f$ und $l$ für die beiden parallelen Seiten und der Anzahl der Schedulingschritte $N$ als die Höhe des Trapezes. Die Gesamtanzahl der Tasks ist bekannt. Also kennen wir die Fläche des Trapezes. Daraus kann man die benötigte Blockanzahl für die Parameter $f$ und $l$ mit Hilfe der Flächenformel für Trapeze berechnen. Um die Fläche zu erhalten muss man den Mittelwert der beiden parallelen Strecken (unsere Parameter $f$ und $l$) berechnen, und dann mit der Höhe (unsere Anzahl der Blöcke $N$) multiplizieren. Wir erhalten die folgenden Formeln:

$$n = \frac{f + l}{2} * N \tag{5.3}$$

$$N = \lceil \frac{2 * n}{f + l} \rceil \tag{5.4}$$

Durch die Umformung nach der Blockanzahl erhalten wir die Formel 5.4. Da wir jetzt die Blockanzahl kennen, können wir zusammen mit den Werten für die Start- und Endblockgröße die Steigung unserer Schedulingfunktion berechnen. Also den Wert, um den die Blockgröße pro Schedulingschritt abnimmt. Die genaue Formel lautet:

$$\Delta BG = \frac{f - l}{N - 1} \tag{5.5}$$

Damit ist die Initialisierung des Algorithmus abgeschlossen. In jedem Schedulingschritt muss nur noch die Blockgröße $BG$ ausgehend von der initialen Blockgröße $f$ um den konstanten Wert $\Delta BG$ verändert werden.
Eine wichtige Rolle spiel die Wahl von $f$ und $l$. Der optimale Wert ist immer von den Eigenschaften der Eingabedaten abhängig und kann zur schlechten Ergebnissen bei falschen Einschätzungen führen. Bei der Vorstellung des Verfahrens schlugen Tzen und Ni daher vor, eine konservative Strategie zu fahren, die bei beliebigen Eingabedaten vernünftige Ergebnisse produziert.
Die Art der Eingabe kann man in vier verschiedene Kategorien einteilen. Der ideale Eingabetyp hat die gleiche Laufzeit für jeden Task, so dass der Fixed Size Chunking Algorithmus die besten Ergebnisse liefern wird. Leider kann man diese Blockgröße nicht benutzen, da die wenigsten Eingaben aus diesem Typ sind. Der schwierigste Typ hat eine stark variierende und daher unbekannte Laufzeit für jeden Task. Die Startblockgröße sollte daher nicht zu groß gewählt werden. Die Endblockgröße sollte auf jeden Fall 1 betragen, um die Schwankungen in der Lastverteilung zu minimieren. Der einfache Self Scheduling Algorithmus wäre hier am besten geeignet, kann aber wegen der Anfangsblockgröße von 1 nicht eingesetzt werden. Die Tasks mit einer steigenden Laufzeit werden zum Ende kritisch. Wählt man eine zu große Endblockgröße kommt es zu einer großen Ungleichheit in der Lastverteilung und zur Vergrößerung der Ausführung. Es hat sich bewährt die Endgröße auf den Wert von 1 zu setzen um den Makespan zu minimieren. Bleiben noch die Tasks mit einer sinkenden Laufzeit. Hier ist es fatal, anfangs eine zu große Blockgröße zu nehmen, da die nachfolgenden Tasks diese Laufzeit nicht ausgleichen können und viel früher enden als der erste Block. Hier war der Vorschlag die initiale Blockgröße auf $n/2m$ zu setzen, und damit auf die Hälfte der von dem Guided Self Scheduling Algorithmus vorgeschlagenen Wertes.
Kommen wir zur Laufzeit. Die Gewichte können in der Initialisierungsphase in linearer Zeit zu den Anzahl der Maschinen berechnet werden. Die Anfangsberechnung für die Blockgröße geschieht in konstanter Zeit. Jeder Schedulingschritt kann in konstanter Zeit durchgeführt werden, da man nur die Blockgröße um eine Konstante vermindern muss. Die Gesamtlaufzeit beträgt daher $O(n + m)$. Auch an zusätzlichem Speicher braucht dieser Algorithmus nur Platz für die Gewichtung der Rechner.

### 5.3.5. Factoring ($FAC$)

Beide oben vorgestellte Algorithmen mit abnehmender Blockgröße haben den gleichen Nachteil. Die Größe der Blocks nimmt ständig ab, so dass sogar bei der ersten Block-

zuordnung der letzte Rechner eine deutlich kleinere Anzahl an Tasks bekommt als der Erste. Die Factoring Strategie [15] hat diesen Nachteil nicht. Hier werden alle Schedulingschritte in Runden eingeteilt. Die Blockgröße bleibt innerhalb einer Runde konstant. So kriegt jeder Rechner in einer Runde die gleiche Anzahl an Tasks. Die Dauer einer Runde ist mit der Rechneranzahl identisch. Pro Runde wird insgesamt die Hälfte der offenen Tasks zugeordnet, deshalb nimmt auch hier die Blockgröße zwischen den Runden exponentiell ab. In der ersten Runde kriegt jeder Rechner $n/2m$ Tasks. Die Anfangsgröße ist damit mit dem vorgeschlagenen Startwert für den Trapezoidal Self Scheduling Algorithmus identisch. Die Berechnung der Blockgröße lässt sich iterativ oder, wie es Shao [29] vorschlägt, absolut berechnen. Die Berechnung der Rundennummer ist einfach. Man teilt die aktuelle Iteration durch die Anzahl der Rechner $Runde = \lceil Iter/m \rceil$. Bei der Iterativer Methode, halbiert man mit jeder neuen Runde die Blockgröße. Für die absolute Methode lautet die Formel:

$$BG = \frac{n}{m * 2^{Runde}} \tag{5.6}$$

Sollte die Blockgröße unter 1 fallen, wird sie auf 1 gesetzt.
Dieser Algorithmus hat die gleiche Laufzeit, wie das Trapezoidal Self Scheduling Verfahren. Auch der Speicherverbrauch ist gleich.

### 5.3.6. Der Vergleich der Blockgröße

Alle Verfahren mit der variablen Blockgröße verfolgen das gleiche Ziel. Sie starten mit einem großen Block, um den Transferoverhead zu minimieren. Danach wird die Größe minimiert, um die Lastverteilung auf allen Rechner möglichst gut auszugleichen. In der unteren Tabelle sind die Blockgrößen aufgelistet für $n = 100$ und $m = 5$. Die Grafische Entwicklung kann man in der Abbildung 5.3 nachschauen. Auch hier beschreibt die X-Achse die Blockanzahl, und die Y-Achse die Blockgröße.

| | |
|---|---|
| Fixed | 20 20 20 20 20 |
| GSS | 20 16 13 10 9 7 5 4 4 3 2 2 1 1 1 1 |
| TSS | 10 9 9 8 8 7 7 6 6 5 5 4 4 3 3 2 2 1 1 |
| FAC | 10 10 10 10 10 5 5 5 5 2 2 2 2 1 1 1 1 1 |
| SelfS | 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 |

Auf Anhieb lässt sich nicht sagen, welches der Verfahren besser ist. Alles hängt von der Eigenschaften der Eingabe und an Anzahl der Rechner im Netz.

## 5.4. Replikation

Zum Schluss soll noch die letzte Klasse der Schedulingalgorithmen vorgestellt werden: dynamische Verfahren, die Migration einsetzen. Die Schwierigkeit der Durchführung der Migration hängt sehr stark von dem Zustand, in dem sich der betroffene Task in dem Augenblick befindet.

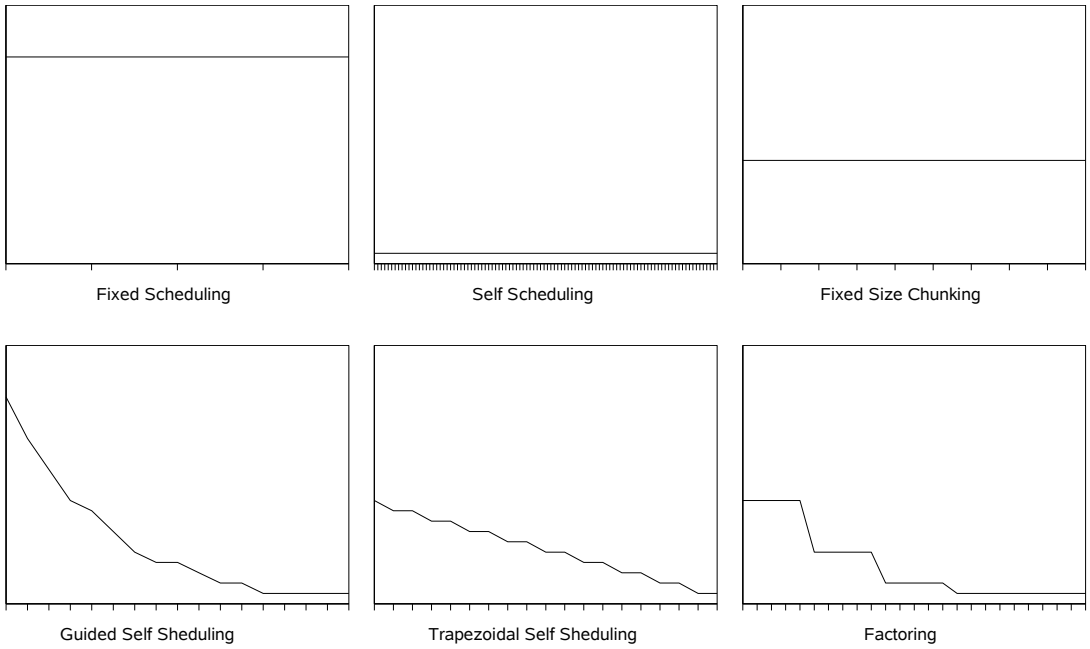



Abbildung 5.3.: Übersicht über die Entwicklung der Blockgröße der einzelnen Batch Allocation Lösungen. Auf der y-Achse ist die Blockgröße eingetragen. Die x-Achse stellt die Anzahl der Schedulingschritte dar. Die Blockgröße ist eine konstante oder fallende Funktion. Es wurden insgesamt 100 Tasks auf 5 Maschinen verteilt. Zu beachten ist, dass ein Bild die gesamten Schedulingschritte darstellt. Bei dem Self Scheduling Verfahren sind es etliche mehr, als bei dem Fixed Size Chunking

Falls der zu migrierende Task noch auf die Ausführung gewartet hat, ist die Migration nicht schwierig. Dem aktuellen Rechner wird einfach befohlen diese Aufgabe aus der Warteliste zu streichen. Danach wird die Aufgabe zu einem anderen Rechner übertragen. Eine große Rolle spielt es, ob der Scheduler noch eine Kopie der zu migrierenden Aufgabe hat. Wenn nicht, muss durch einen Mechanismus die Kommunikation zwischen den ausführenden Rechnern ermöglicht werden, was nicht in allen Rechnerumgebungen möglich oder sinnvoll ist. Wenn zum Beispiel einzelne Rechner hinter einer Firewall sind, ist die Verbindung zwischen ihnen nicht möglich. Als letzten Ausweg, kann die Aufgabe zurück an den Scheduler übertragen werden, um dann an die neue Maschine weitergeleitet zu werden.

Viel schwieriger wird die Migration, wenn mit der Ausführung der Aufgabe bereits begonnen wurde. Um diese Art von Migration zu ermöglichen, müssen in die einzelnen Tasks sogenannte „Breakpoints“ eingefügt werden. Das sind Stellen im Programm, an denen die Ausführung unterbrochen werden kann. Gibt es diese Stellen nicht, muss die

Berechnung von vorne durchgeführt werden. Die, bis dahin erzielten Ergebnisse, gehen dabei verloren, und man kann mit diesem Task nach dem gleichen Prinzip, wie vor der Ausführung verfahren. Wenn es aber die Haltepunkte gibt, werden die bis dahin berechneten Zwischenergebnisse und die aktuelle Position in der Ausführung gespeichert. Diese Daten müssen zusammen mit den (vielleicht inzwischen veränderten) Eingabedaten an den neuen Arbeiter übermittelt werden. Dieser muss dann anhand der übermittelten Daten die Position in der Ausführung feststellen und die Zwischenergebnisse wieder zurückschreiben. Danach kann die Ausführung fortgesetzt werden.
Die Migration hat auch Nachteile. Die gleichen Daten müssen mehr als ein mal zwischen den Rechnern übertragen werden, was zu einem größeren Kommunikationsoverhead führt und die Bandbreite des Kommunikationskanals für andere Teilnehmer mindert. Ein Geschwindigkeitsvorteil durch die Migration kann auch nicht garantiert werden. Es kostet Zeit, die Daten zum Transport vorzubereiten, sie dann zu serialisieren und zu übertragen, was in Abhängigkeit von der Bandbreite und der Leistungsfähigkeit des Rechners eine längere Zeit beanspruchen kann. Auch dauert es eine gewisse Zeit, bis der Zielrechner die ankommenden Daten verarbeitet hat und mit der Ausführung weitermachen kann. Das kann den Geschwindigkeitsvorteil des schnelleren Rechners zunichte machen, und die Gesamtausführung sogar verlangsamen.
Wie man sieht ist die Migration in vielen Fällen ziemlich aufwendig zu realisieren. Es gibt aber ein Verfahren, dass einen ähnlichen Effekt hat. Man kann die sogenannte „Replikation“ durchführen. Man schickt dabei den gleichen Task an mehrere Rechner, wartet auf das erste Ergebnis und sendet ein Signal an andere Rechner, dass sie mit der Ausführung aufhören können. Dabei kann es passieren, das man die Ausführung nicht unterbrechen kann und somit die gleiche Arbeit mehrmals durchführt. Damit entzieht man den anderen Teilnehmern des Netzes die Rechenzeit. Auch die sequenzielle Ausführung mehrerer Jobs des gleichen Masters hintereinander kann erheblich verlangsamt werden, da manche Rechner nach dem Abschluss des Jobs immer noch an den alten Tasks weiter rechnen.

### 5.4.1. Simple Work Queue Replication ($S - WQR$)

Das Simple Work Queue Replication Verfahren wurde von da Silva [7] vorgestellt. Als Basis für diesen Algorithmus dient das einfache Self Scheduling Verfahren. Man überträgt jedes mal einen Task an den gerade freien Rechner. Die Änderung, die dieser Algorithmus vorstellt passiert am Schluss, wenn alle Tasks zugeordnet wurden, und man nur noch auf das Ende deren Ausführung wartet. Wenn der erste Rechner mit der Ausführung fertig ist und sich somit im Leerlauf befindet, startet der Replikationsteil des Verfahrens. Aus der Liste, der noch nicht abgeschlossenen Tasks, wird zufällig einer ausgewählt und an den freien Rechner übertragen. Dadurch führen jetzt zwei Rechner parallel die gleiche Arbeit aus. Man muss nur auf den schnelleren warten. Dieses Vorgehen wiederholt man mit jedem freiem Rechner, bis keine offenen Tasks mehr zur Verfügung stehen.
Dieses Verfahren versucht den Einfluss, den das Zuordnen von einem langen Task zu einen langsamen Rechner hat, zu minimieren. Mit jedem fertigen Task steigt die Wahrscheinlichkeit, dass der lange Tasks ausgewählt wird, und auf eine (hoffentlich) schnellere Maschine kommt. Ist nur noch ein Task übrig, so wird er auf allen anderen Maschinen

ausgeführt, so dass automatisch die schnellste Maschine dabei ist. Leider kann man mit diesem Verfahren keine Verbesserung garantieren. Durch eine ungünstige Wahl von dem zu replizierenden Task kann es passieren, dass das Original und die Kopie zur gleichen Zeit fertig werden. Es wird zwar die doppelte Anzahl an Berechnungen ausgeführt, der Makespan ändert sich aber nicht.
Das Kopieren der Tasks findet statt, wenn es noch höchstens $m-1$ nicht abgeschlossene Tasks gibt, so dass man eine freie Maschine zur Verfügung hat. Mit jeder Fertigstellung eines Tasks erhöht sich die Zahl der freien Maschinen um eins. Der letzte noch offene Task wird auf $m-1$ Maschinen kopiert. Insgesamt finden schlimmstenfalls $1+2+\ldots+m-1 = \frac{(m-1)*m}{2}$ (Geometrische Reihe von 1 bis $m-1$) Replikationen statt.
Der große Vorteil von diesem Algorithmus ist, dass er in jeder Umgebung eingesetzt werden kann. Er braucht keine Informationen über die Leistungsfähigkeit der Rechner, die Laufzeit der Tasks und die Aktivitäten anderer Maschinen im Netz. Das alles wird durch den erhöhten Rechenaufwand erkauft. Ob der Einsatz dieses Verfahrens sinnvoll ist, hängt vor allem von der Anzahl der Rechner im Verhältnis zu der Anzahl der Tasks ab. In ungünstigen Fällen kann sich der Rechenaufwand mehr als verdoppeln.
Für die Ausführung wird keine Vorverarbeitung benötig. Jeder Schedulingschritt kann in der Zeit $O(1)$ durchgeführt werden. Allerdings müssen hier mehr als $n$ Tasks zugeordnet werden. An zusätzlichem Speicher ist nichts nötig.

### 5.4.2. Replication with TimeOuts ($TO - WQR$)

Als letztes soll hier noch die Weiterentwicklung des Simple Work Queue Replication Verfahrens vorgestellt werden. Der erste Nachteil bei dem Verfahren war, dass die Tasks durch das Self Scheduling auch zu Beginn der Ausführung nur einzeln zugeordnet wurden. Dieser Teil soll durch einen besseren Algorithmus ersetz werden, der nicht einzelne Tasks, sondern komplette Blöcke überträgt. Die beiden Algorithmen mit exponentiell fallenden Blockgrößen haben sich als ungünstig erwiesen, da viele Iterationen am Ende die Blockgröße 1 haben. Dies ist nicht mehr nötig, da der Ausgleich der Last der einzelnen Maschinen am Ende von der Vervielfältigung übernommen wird. Also wurde für die Implementierung das Trapezoidal Self Scheduling Verfahren gewählt.
Der zweite Nachteil bestand darin, dass sehr viele Replikationen durchgeführt wurden. Das neue Verfahren versucht, die Vervielfältigungen zu reduzieren. Es gibt mehrere Situationen, in denen das Kopieren sinnvoll ist. Nach dem alle Tasks verteilt worden sind, passiert es oft, dass ein Rechner frei ist, ein anderer aber noch mehr als einen Tasks in der Ausführungsschlange hat. In diesem Fall wird sogar keine Replikation, sondern eine echte Migration durchgeführt. Der Task wird aus der Warteschlange des überlasteten Rechners entfernt und zu dem freien Rechner übertragen. Es entsteht kein Overhead bei den Berechnungen. Nachdem sichergestellt worden ist, dass jeder Rechner höchstens einen Task hat, werden Maschinen gesucht, von denen noch überhaupt kein Ergebnis zurückgekommen ist. Das ist ein Hinweis auf einen sehr langsamen oder abgestürzten Rechner. Die Tasks von diesen Maschinen werden vervielfältigt. Danach sind die Rechner an der Reihe, die für die Ausführung zu lange brauchen. Während der gesamten Ausführungszeit wird die Dauer, die benötigt wird, um einen Task abzuschließen für

jeden Rechner gespeichert. Stellt der Algorithmus fest, dass diese Dauer um 50% der bisherigen Zeiten überschritten wurde, findet eine Replikation statt. Das bedeutet aber auch, dass dieser Algorithmus verhindert, dass ein Task, der am Ende an eine langsame Maschine zugewiesen wurde, repliziert wird, falls alle Tasks die gleiche Laufzeit haben. Allerdings ist die gleiche Laufzeit eher ein Idealzustand. Wie schon in dem Abschnitt über das Trapezoidal Self Scheduling Verfahren beschrieben gibt es 4 Typen von Jobs. Die mit den konstanten Laufzeiten sorgen für keine großen Unterschiede in der Last. Die Jobs mit sinkenden Laufzeiten machen auch kein Problem, da die letzten Tasks nur einen kleinen Beitrag zu dem Makespan leisten. Entweder sind die langen Tasks schon fertig, dann gibt es keine Probleme, oder ein langsamer Rechner hat einen besonders langen Task erwischt. Dann hat er zum Ende noch keine Ergebnisse geliefert, und der Task wird vervielfältigt. Bei Tasks mit zufälligen oder steigenden Laufzeiten hilft die Aufzeichnung der früheren Laufzeit. Liegt die jetzige Laufzeit weit über dem Durchschnitt, wird eine Vervielfältigung durchgeführt.
Bleibt noch das Problem, dass ein Task an zu viele Rechner verschickt wird. Bei dem letzten Task kann die Anzahl der Replikationen $m - 1$ betragen. Um dies zu verhindern wurde für jeden Task ein Zähler mit der Anzahl der Replikationen eingeführt und dafür gesorgt, dass ein Task nicht auf mehr als drei Rechnern gleichzeitig ausgeführt wird. Auch wird nach der ersten Replikation die Wahrscheinlichkeit für die zweite Replikation verringert, so dass andere Tasks vorher dran kommen. Durch all diese Maßnahmen soll die Anzahl der unnötigen Replikationen minimiert werden. Allerdings kann dadurch auch der Makespan ansteigen und sich schlimmstenfalls an die Zeiten von Self Scheduling angleichen. Es muss auch geprüft werden, ob der Einsatz von Trapezoidal Scheduling sich in diesem Fall gelohnt hat.
Es ist nicht einfach die Laufzeit dieses Verfahrens abzuschätzen. An Vorverarbeitung braucht man durch das Trapezoidal Scheduling die Zeit $O(m)$ um die Rechnergewichte zu kriegen. Jeder Schedulingschritt am Anfang kann in konstanter Zeit durchgeführt werden. Bei jeder Replikation müssen 3 Sachen geprüft werden, Ob es Rechner mit mehr als einem Task gibt, ob es Rechner ohne Ergebnisse gibt, und ob der Timeout für die Vervielfältigung erreicht wurde. Das kann in der Zeit $3m$ überprüft werden. Die Höchstanzahl an Replikationen beträgt $2m$, da jeder Task nur zwei mal repliziert werden kann. Allerdings kommen hier noch die Migrationen hinzu, um das mögliche Ungleichgewicht durch das Trapezoidal Scheduling auszugleichen. Dabei können aber höchstens $n - m$ Tasks neu verteilt werden. Auch beim Speicherverbrauch ist dieser Algorithmus der Anspruchsvollste. Man muss die Rechnergewichte, den Zeitpunkt für die letzte Taskübermittlung und die durchschnittliche Tasklaufzeit für jeden Rechner abspeichern, außerdem kommt für jeden Task eine Liste mit der Anzahl an Replikationen. Der Gesamtspeicherverbrauch beträgt somit $O(3m + n)$.

## 5.5. Kurzübersicht über die Algorithmen

In der Tabelle 5.1 ist eine Übersicht über das Laufzeitverhalten der Algorithmen zusammengestellt. In der ersten Spalte sind die Werte für die Initialisierung. Das ist die

| Name | Laufzeit | | | Speicher- |
|---|---|---|---|---|
| | init | step | gesamt | verbrauch |
| FIXED | — | $O(1)$ | $O(n)$ | $O(1)$ |
| RANDOM | — | $O(1)$ | $O(n)$ | $O(1)$ |
| FIXED-W | $O(m)$ | $O(1)$ | $O(n+m)$ | $O(m)$ |
| RANDOM-W | $O(m)$ | $O(m)$ | $O((n+1)*m)$ | $O(m)$ |
| MCT | — | $O(m)$ | $O(m*n)$ | $O(2m+n)$ |
| MINMAX | $O(n \log n)$ | $O(m)$ | $O(n \log n + n*m)$ | $O(2m+n)$ |
| MINMIN | $O(n \log n)$ | $O(m)$ | $O(n \log n + n*m)$ | $O(2m+n)$ |
| WQ | — | $O(1)$ | $O(n)$ | $O(1)$ |
| TSS | $O(m)$ | $O(1)$ | $O(m+n)$ | $O(m)$ |
| FAC | $O(m)$ | $O(1)$ | $O(m+n)$ | $O(m)$ |
| S-WQR | — | $O(1)$ | $O(n+m^2)$ | $O(1)$ |
| TO-WQR | $O(m)$ | $O(3m)$ | $O(m+3m*(2n+m)$ | $O(3m+n)$ |

Tabelle 5.1.: Übersicht über die Laufzeit und Speicherverbrauch der einzelnen Algorithmen. Es ist jeweils die Laufzeit für die Initialisierung, die Zuordnung von einem Task und die gesamte Laufzeit angegeben.

Zeit, die benötigt wird, um Informationen über die Umgebung zu verarbeiten. Die zweite Spalte gibt die Kosten an, die benötigt werden um einen Task zuzuordnen. In der dritten Spalte ist die Gesamtlaufzeit des Verfahrens angegeben. Da jedes mal mindesten $n$ Tasks verteilt werden müssen, gibt es auch mindesten $n$ Zuordnungen. Die Laufzeit ist also zu der Anzahl der Tasks mindestens linear. Wie man sieht sind die Laufzeiten ziemlich gering. Die Anzahl der Maschinen $m$ befindet sich normalerweise in einem einstelligen oder niedrigen zweistelligen Bereich. Ein Schedulingschritt dauert normalerweise keine Millisekunde. Die Intervalle zwischen den einzelnen Schedulingschritten betragen die Laufzeit eines Tasks durch die Anzahl der Maschinen. Der Scheduler befindet sich also bei allen Verfahren die meiste Zeit in einem Leerlauf. Auch der Speicherverbrauch ist linear in der Anzahl der Tasks und/oder Maschinen, und somit sehr gering. All diese Algorithmen können ohne Probleme auf langsamen Maschinen mit wenig RAM laufen. Den größten Anteil am Speicherverbrauch haben die Tasks selber. In der Tabelle 5.2 sind die Informationen aufgelistet, die die einzelnen Verfahren zur Ausführung brauchen. Die Anzahl der Tasks und Anzahl der Rechner ist aus der Eingabe ersichtlich. Den Rest muss erst verschafft werden. Die Leistungsfähigkeit und die Last muss bei den Rechnern angefragt werden. Die Tasklaufzeiten muss der „Auftraggeber“ mitliefern. Die Algorithmen sind auf die Korrektheit der Daten angewiesen. Je größer die Abweichung von dem tatsächlichen Wert, desto schlechter sind auch die Schedulingergebnisse. Natürlich kann man nie Fehler ausschließen. Deshalb sollten vor allem die Heuristiken, die sehr viel Informationen benötigen, fehleranfällig sein. Interessant ist auch, dass bis auf die Heuristiken, kein anderer Algorithmus die Tasklaufzeiten benötigt. Das spricht entweder für die totale

| Name | Rechner Anzahl | Rechner Leistung | Task Anzahl | Task Laufzeit | externe Last |
|---|---|---|---|---|---|
| FIXED | x | | x | | |
| RANDOM | x | | | | |
| FIXED-W | x | x | x | | |
| RANDOM-W | x | x | | | |
| MCT | x | x | x | x | x |
| MINMAX | x | x | x | x | x |
| MINMIN | x | x | x | x | x |
| WQ | | | | | |
| TSS | x | x | x | | |
| FAC | x | x | x | | |
| S-WQR | | | | | |
| TO-WQR | x | x | x | | |

Tabelle 5.2.: Übersicht über die benötigten Informationen, die als Eingabe für die einzelnen Algorithmen benötigt werden. Die Anzahl der Rechner und der Tasks ist automatisch vorhanden. Der Rest muss ermittelt oder geschätzt werden.

Unwichtigkeit dieser Angabe, oder für den großen Aufwand hier korrekte Werte zu liefern. Da die Last der Rechner direkt von den Tasklaufzeit abhängt ist auch dieser Wert sehr Fehleranfällig. Bei der empirischen Untersuchung wird es interessant zu sehen sein, wie die „primitiven Algorithmen“ Self Scheduling und Simple Work Queue Replication, die gar keine Information benötigen, sich gegenüber den fortschrittlichen Verfahren schlagen.

# 6. Distributed RapidMiner

Um die verschiedenen Schedulingalgorithmen zu testen, musste zuerst eine geeignete Testumgebung geschaffen werden. Es bot sich an, das RapidMiner Tool für die parallele Ausführung zu erweitern. Bei der Parallelisierung bot RapidMiner einige Vorteile. Dieses Tool ist Open Source und steht unter der GPL Lizenz. Durch den Zugang zum Code konnte die Entwicklung sehr schnell gestartet werden. RapidMiner ist in Java geschrieben, so dass die Entwicklung plattformunabhängig gestaltet werden konnte. Und vor allem bietet RapidMiner eine gute Schnittstelle für Erweiterungen an, so dass sich die Verteilungsplattform leicht in das bestehende System integrieren ließ.
Um die Experimente durchführen zu können, mussten die einzelnen Abläufe in den Operatoren parallelisiert werden. Am Anfang der Überlegungen gab es insgesamt zwei Wege, die für die Parallelisierung des RipidMiners in Frage kamen. Die erste Möglichkeit war es, eine eigenständige Applikation zu schaffen, die dann mit dem RapidMiner kommuniziert und für die Verteilung sorgt. Allerdings müsste man bei diesem Weg trotzdem den RapidMiner erweitern und die Module für die Kommunikation hinzufügen. Nach der Analyse des RapidMiners wurde der zweite Weg favorisiert. Die gesamte Anwendung sollte als ein Teil des RapidMiners laufen, und als ein Plugin für diesen zur Verfügung stehen. Die Kommunikation zwischen dem Plugin und dem RapidMiner sollte über die Operatoren geschehen. Dies bot sich an, da der RapidMiner gerade für die einfache Integration von neuen Operatoren gute Schnittstellen bietet.
Das Ziel war es, eine Erweiterung zu schaffen, die leicht eingesetzt werden kann, keinen großen Konfigurationsaufwand erfordert und in möglichst vielen Umgebungen funktioniert. Die komplette Funktionalität sollte in den Operatoren des Experimentes untergebracht werden. So müsste keine Anpassung an den Originalquellen des RapidMiners vorgenommen werden. Ein vorhandenes Experiment sollte man so ohne großen Aufwand durch das Hinzufügen und/oder den Austausch von Operatoren in die verteilte Version überführen können. Ein parallelisierter Operator sollte seine Aufgabe in viele Teilaufgaben zerlegen können und diese dann an das Plugin liefern. Später sollten die fertigen Ergebnisse zurück an den Operator geschickt werden, die er dann auswerten und zu einem Gesamtergebnis zusammenfügen kann.
Die Funktionalität des Plugins sollte in mehreren Modulen untergebracht werden. Diese Module sollten zu Beginn des Experiments gestartet werden, so dass alle Operatoren einen Zugriff darauf haben. Nach der Beendigung der Ausführung werden diese Module dann wieder gestoppt und aus dem Speicher entfernt. Die Verteilungskomponente existiert also nur während der Laufzeit eines Experiments. Den Gesamtaufbau des Plugins kann man in der Abbildung 6.1 sehen. Die Funktionalität teilt sich in zwei Bereiche. Durch die neu erstellten Operatoren ist die Integration in den RapidMiner möglich. Bei der Ausführung des Distributed Scheduler Operators wird das DRM-Modul erzeugt. Da-

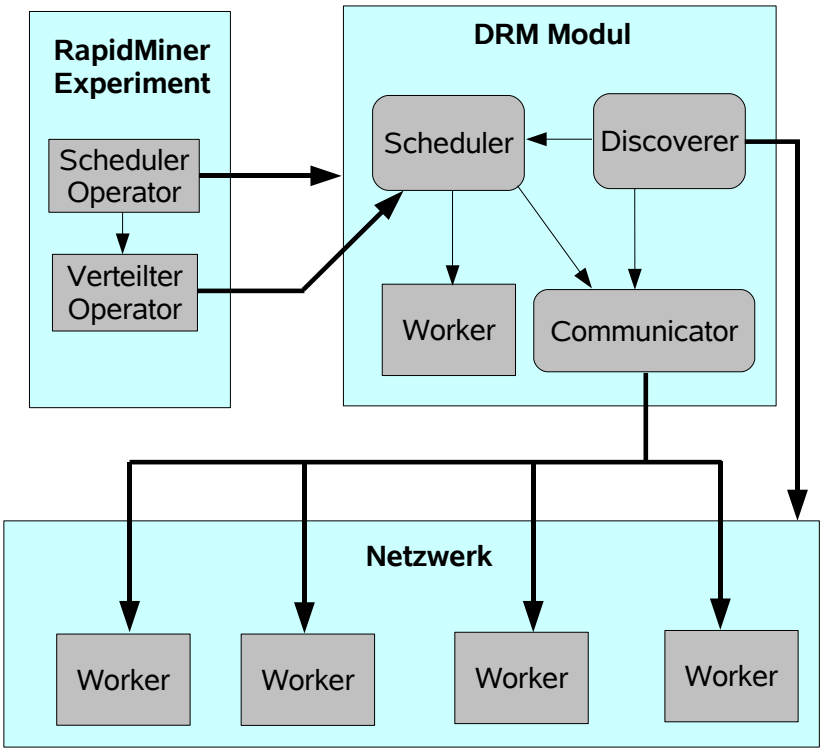



Abbildung 6.1.: Aufbau des DRM Moduls.

bei können auch einige Einstellungen an dem Verhalten vorgenommen werden. Die nachfolgenden Operatoren können dann direkt die einzelnen Teile des Plugins ansprechen. Auf der anderen Seite ist das Plugin selbst. Es besteht aus drei großen Komponenten. Dem Scheduler, dem Discoverer und dem Communicator.

## 6.1. Der Scheduler

Der Scheduler fungiert als Schnittstelle zwischen dem RapidMiner Experiment und den externen Rechnern im Netz. Seine Aufgabe besteht darin, die Tasks von dem aktuellen Operator entgegenzunehmen und sie an geeignete Worker weiterzuleiten. Dabei wendet der Scheduler eine der Zuordnungsmethoden, die der Benutzer in den Optionen ausgewählt hat. Nach der Verteilung wartet der Scheduler auf die Ergebnisse und sendet diese dem Operator zurück. Dies ist das Herzstück des Plugins auf der Masterseite. Um eine bestimmte Zuordnung von Tasks zu Workern zu gewährleisten muss der Scheduler Informationen über die vorhanden Worker und aktuellen Tasks speichern. Zu den weiteren Aufgaben diese Moduls gehört auch das Starten des lokalen Workers und die Versorgung von diesem mit Arbeit.

## 6.2. Der Discoverer

Der Discoverer ist das erste Modul, das beim Starten des Plugins zum Einsatz kommt. Dessen Aufgabe ist es geeignete Rechner im Netz zu finden, die als Worker für das Experiment fungieren können. Die so gefundenen Rechner werden dann weiter an den Communicator weitergeleitet, der für die restliche Kommunikation sorgt.
Es gibt insgesamt drei verschiedene Möglichkeiten die verfügbaren Rechner zu finden. Falls eine Workerliste von dem Benutzer angegeben wurde, arbeitet der Discoverer diese Liste Schritt für Schritt ab und versucht die Rechner auf dieser Liste zu erreichen. Die zweite Methode ist die Verwendung eines zentralen Servers, falls dieser in den Optionen angegeben wurde. Von diesem Rechner wird eine Liste mit den aktuell aktiven Rechnern geliefert, die dann genau so wie einen normale Workerliste behandelt wird.
Die ersten beiden Verfahren sind statisch. Einmal ausgeführt ist die Entdeckung der Rechner abgeschlossen. Neu hinzugekommene Rechner können so nicht berücksichtigt werden. Hier hilft die dritte Methode. Bei dieser Methode beginnt der Discoverer das lokale Netz aktiv und passiv zu überwachen. Er lauscht auf ankommende Bekanntmachungen anderer Rechner im Netzwerk. Wird eine Bekanntmachung eines Arbeiters festgestellt, wird der Versuch eines Verbindungsaufbaus gestartet. Außerdem versendet der Entdecker alle zwei Minuten einen Ping, um sich selber im Netz bekannt zu machen. Natürlich kann die Überwachung des Netzes nur in einem LAN funktionieren. Sie ermöglicht es aber während der Laufzeit des Experiments neue Rechner für die Ausführung zu finden.

## 6.3. Der Communicator

Wie der Name schon sagt übernimmt der Communicator die gesamte Kommunikation mit den anderen Rechnern. Seine Hauptaufgabe besteht darin, die Nachrichten von dem Scheduler weiter an die einzelnen Rechner zu versenden und die ankommenden Nachrichten für die Weiterverarbeitung weiterzuleiten. Für die Serialisierung der Nachrichten wird X-Stream[1] verwendet, der die Daten in das XML-Format wandelt. Danach wird der Inhalt mit dem ZIP-Verfahren komprimiert. Das Versenden selber geschieht nach dem FIFO-Prinzip. Die Nachrichten kommen in eine Schlange und werden nach und nach abgearbeitet. Die einzige Priorität dabei ist die Ankunftszeit. Neben der Datenübertragung überwacht der Communicator auch die Verbindung zu dem Worker. Sollte sie getrennt werden, unterrichtet er den Scheduler über den Ausfall. Auch die Verbindungsversuche von außen werden von diesem Modul behandelt. Bei einer erfolgreichen Verbindung, wird diese an den Scheduler gemeldet und die Kommunikation mit dem Rechner gestartet.

## 6.4. Der Worker

Neben den Funktionen auf der Masterseite mussten auch die Worker implementiert werden. Sie laufen als eigenständige Applikationen auf den externen Rechnern. Nach ihrem

[1] http://xstream.codehaus.org/

Start warten sie auf eine Verbindung von dem Master. Ist ein Master verbunden, werden von ihn die Teilaufgaben angenommen und ausgeführt, dabei wird auch auf die Funktionalität des RapidMiners zurückgegriffen, dass als eine Bibliothek vorhanden ist. Die genaue Funktionalität des Workers kann in dem Anhang A angeschaut werden.

## 6.5. Eigenschaften des Systems

Bevor das DRM Plugin implementiert wurde, musste man sich auf eine Technologie festlegen. In dem ersten Kapitel wurden mehrere Ansätze, Protokolle und Middleware Systeme vorgestellt. Leider sind die meisten Implementierungen sehr weit gefasst, um ein möglichst großes Einsatzgebiet abzudecken. So entsteht bei allen Anwendungen viel überflüssiger Ballast. Meistens werden auch die Anforderungen nicht zu 100% erfüllt, was zu Schwierigkeiten beim Einsatz führt, so dass man gezwungen wird, Workarounds für die Probleme zu suchen. Deswegen wurden alle externen Lösungen verworfen, und die Schnittstellen für die Rechnerkommunikation selber geschrieben. So konnte man von Anfang an Einfluss auf die Entwicklung nehmen und nur die Sachen implementieren, die man wirklich braucht. Für die Implementierung stellt Java zwei verschiedene Ansätze zur Verfügung. Entweder über die Sockets, oder über die Java-Channels, die ein Teil von dem Java „New IO“ Packet sind. Die Channel Lösung wurde gewählt, weil sie es ermöglicht, die gesamte Kommunikation in einem Thread laufen zu lassen. Es wurde angenommen, dass hierdurch die Komplexität der Anwendung gesenkt werden kann. Leider ist die Channel-Schnittstelle komplex, was Schwierigkeiten bei der Implementierung bereitete, die allerdings mit der Zeit beseitigt werden konnten.
Die Implementierung lehnt sich an des „Remote Service Invocation“ Prinzip an. Der Benutzer (hier: der Operator) gibt die Tasks zum Ausführen ab. Danach ist er frei und kann weitere Berechnungen anstellen. Die Ergebnisse der Tasks werden gesammelt, und können von ihm jeder Zeit abgeholt werden. Im Gegensatz zu RPC, wo nur eine Methode zur gleichen Zeit aufgerufen wird, gibt es hier eine komplette Liste mit Aufträgen, die abgearbeitet werden müssen. Die Bearbeitung soll parallel auf mehreren Rechnern stattfinden. Am Ende gibt es mehrere Ergebnisse, die aber nicht zur gleichen Zeit ankommen. Hier wäre der RPC Ansatz von Nachteil, da er keine Bearbeitung der Teilergebnisse zulässt.
Bei der Implementierung wurde darauf geachtet, die Anforderungen an die Transparenz möglichst gut zu erfüllen. Wenn man sich die Liste anschaut, stellt man fest, dass die aufgeführten Transparenzen zum großen Teil erfüllt wurden. Die Ressource ist in diesem Fall die Rechenzeit, die man leihen möchte:
**Zugriffstransparenz:** Wenn der Benutzer in den Optionen einen lokalen Worker gewünscht hat, wird dieser automatisch in die Verarbeitung eingebunden. Für den Nutzer ergibt sich sonst kein Unterschied in dem Zugriff auf den lokalen Worker, er übergibt wie gewohnt einfach eine Liste mit den Tasks.
**Ortstransparenz:** Jeder Task kann an jedem Rechner ausgeführt werden. Die genaue Verteilung ist für den Nutzer normalerweise gar nicht sichtbar. Auch kann die Ausführung gleichzeitig an mehreren Rechnern stattfinden, ohne dass es für das Ender-

gebnis einen Unterschied macht. Alles was der Nutzer am Ende kriegt, ist eine Liste mit den Ergebnissen.
**Leistungstransparenz:** Diese Transparenz wurde leider nicht ganz erfüllt. Der Nutzer muss sich aktiv um die Entdeckung der externen Rechner kümmern, entweder durch die Bereitstellung einer Liste mit den möglichen Workern, oder die Angabe des Servers mit der Workerliste. Nur in einer LAN Umgebung kann das Auffinden der anderen Rechner des Systems automatisch geschehen.
**Ausfalltransparenz:** Das System wurde so implementiert, dass der Auftrag ausgeführt werden kann, solange noch mindestens ein Worker aktiv ist. Wenn also der lokale Worker aktiviert ist, kann die Netzverbindung komplett wegbrechen, ohne dass der Auftrag abgebrochen werden muss. Ansonsten beeinflusst der Ausfall eines Rechners die Ausführung sehr gering. Die Aufgabe wird einfach an einen anderen Worker versendet.
**Sprachtransparenz:** Da RapidMiner und damit auch das RDM-Plugin in Java geschrieben sind, können sie ohne Anpassungen auf einer Vielzahl von Maschinen eingesetzt werden.

## 6.6. Parallelisierte Operatoren

Die normalen Operatoren des RapidMiners sind nicht für die parallele Ausführung ausgelegt, da sie ihre Aufgaben sequentiell ausführen. Sogar wenn sie diese Aufgaben an das DRM-Plugin weiterreichen könnten, würde es keinen Geschwindigkeitsvorteil gegenüber der lokalen Ausführung bringen. Die Operatoren müssen also umgeschrieben werden. Insgesamt müssen zwei Veränderungen an jedem zu parallelisierenden Operator durchgeführt werden. Als erstes muss die Gesamtaufgabe, die dieser Operator ausführen soll, in mehrere kleine Teilaufgaben zerlegt werden. Diese Teilaufgaben müssen unabhängig voneinander ausgeführt werden können, dürfen also keine Ergebnisse der anderen Teilaufgaben für eigene Ausführung brauchen. Diese Teilaufgaben müssen jeweils in ein Task gekapselt werden, der die komplette Information, die zur Ausführung nötig ist, enthält. Die zweite Änderung besteht darin, die Aufgaben nicht selber auszuführen, sondern sie einfach an den Scheduler weiterzuleiten und später die Ergebnisse einzusammeln.

Um das Plugin zu testen wurde eine Reihe von Operatoren parallelisiert. Dabei wurde versucht die Operatoren aus möglichst vielen Einsatzgebieten zu nehmen, aber auch auf die einfache Parallelisierung geachtet. In den nächsten Abschnitten wird auf die einzelnen Verfahren und die damit verbundenen Operatoren näher eingegangen.

### 6.6.1. Kreuzvalidierung

Für die Kreuzvalidierung stellt RapidMiner den XValidation Operator zur Verfügung. Dieser Operator ist wahrscheinlich der am häufigsten benutzte Operator. Die parallelisierte Version zerteilt zuerst die Eingabe in mehrere Teilmengen. Jeder Task kriegt dann ein TrainingsSet und ein TestSet. Daneben werden auch die inneren Operatoren für die Lernaufgaben übertragen. Die Anzahl der Tasks entspricht dabei die Anzahl der Validationen. Bei einer „leave one out“ Validierung ist die Taskanzahl entsprechend hoch. Bei

einer normaler 10fachen Validierung ziemlich niedrig. Der Lernvorgang wird dann parallel ausgeführt. Am Ende werden die Ergebnisse wieder eingesammelt und der Mittelwert der Ergebnisse berechnet.

### 6.6.2. K-Means

Um diesen Operator zu parallelisieren wurde ein einfacher Weg gewählt. Die einzelnen Läufe mit verschiedenen Startpunkten werden komplett unabhängig von einander ausgeführt. Da bietet es sich an die verschiedenen Startpunkte zentral zu generieren und dann die einzelnen Läufe mit einem Task pro Lauf parallel auszuführen. Dabei bleibt die Taskanzahl gering und die einzelnen Tasks sind sehr grobkörnig. Am Ende werden dann die einzelnen Ergebnisse gesammelt und das Beste von ihnen ausgewählt. Bei dieser Variante muss die Eingabe nicht in mehrere Teilmengen zerlegt werden und kann jedes mal komplett übertragen werden. Es werden auch keine inneren Operatoren benötigt, denn auf der Workerseite kann der normale K-Means Vorgang ausgeführt werden.

### 6.6.3. Parameter Optimierung

Für die Parameteroptimierung bietet der RapidMiner unter anderem auch den GridParameterOptimization Operator an.
Dieser Operator nimmt eine Liste mit den zu prüfenden Parametern als Eingabe. Zu jedem Parameter werden dabei die möglichen Werte, die eingesetzt werden sollen angegeben. Es wird für jedes Parameter/Wert Tupel ein eigenes Experiment ausgeführt. Die Ergebnisse am Ende evaluiert und die besten Belegungen für die Parameter ausgegeben. Jedes Experiment läuft dabei unabhängig von den anderen und kann parallel ausgeführt werden. Auf dieser Basis wurde der DistributedGridParameterOptimization Operator geschaffen. Jedes Experiment wird hier zu einem eigenen Task. Die Taskanzahl schwankt hierbei mit der Anzahl der zu optimierenden Parameter-/Werte. Nach der Taskausführung wird das beste Ergebnis wieder zentral ausgewählt und weitergegeben.

### 6.6.4. FeatureSelection und GeneticFeatureSelection

Die Verfahren zur Merkmalsauswahl arbeiten alle nach dem gleichen Muster. In jedem Schritt wird eine Population erstellt und evaluiert. Die Erstellung selbst ist sehr schnell. Die Evaluierung verbraucht viel Zeit. Dabei kann jedes Individuum dieser Population unabhängig von den anderen überprüft werden. Hier setzt die Parallelisierung an. Nachdem eine Population erstellt wurde, wird für jedes Individuum ein eigener Task erstellt und ausgeführt. Nach der Evaluation wird zentral wieder eine Population erstellt und die nächste Evaluierung ausgeführt. Nach dem letzten Durchgang der Schleife wird das beste Ergebnis genommen und ausgegeben.
Da es mehrere Evaluationsphasen gibt, gibt es auch mehrere Phasen der Taskerstellung. Die kompletten Tasks können also nicht zu Beginn der Ausführung erstellt werden. Auch variiert die Anzahl der Tasks von Phase zu Phase. Allerdings müssen die kompletten

Ergebnisse einer Phase vorhanden sein, bevor eine neue Population berechnet werden kann. Damit unterscheidet sich dieser Operator nicht stark von den anderen parallelisierten Operatoren und kann einfach als eine Hintereinaderausführung von mehreren Parallelisierungen angesehen werden.

### 6.6.5. ExampleSet Iteration

Der ExampleSetIteration Operator nimmt mehrere Beispielmengen als Eingabe und führt die Operatoren, die er als Kinder hat, auf ihnen aus. Ein Einsatzbereich ist es, eine Beispielmenge nach bestimmten Kriterien in verschiedene Teilmengen zu splitten und dann auf jeder dieser Teilmengen eine Analyse durchzuführen.
Dieser Operator führt selber keine Data Mining Operation aus. Alle Ergebnisse werden einfach weitergeleitet. Um die Ergebnisse zu verwerten muss später ein weiterer Operator ausgeführt werden, der die Ergebnisse weiterverarbeitet und sie entsprechend ausgibt.
Für die Parallelisierung wird für jede Beispielmenge aus der Eingabe ein eigener Task generiert. Die Anzahl der Tasks schwankt dabei mit der Anzahl den zur bearbeitenden Beispielmengen. So können fein- und grobkörnige Tasks generiert werden. Allerdings werden die Ergebnisse dieser Tasks nicht bearbeitet, sondern einfach weitergeleitet.

### 6.6.6. Iterating OperatorChain

Dieser Operator wird dazu benutzt innere Operatoren mehrmals hintereinander auszuführen. Die Anzahl der Iterationen kann dabei als Parameter übergeben werden. Jede dieser Ausführungen kann in einem Task parallel ausgeführt werden, und so erstellt der neue Operator für jede Iteration einen neuen Task. Allerdings muss man dabei vorsichtig sein. Wenn eine spätere Iteration Ergebnisse einer früheren Iteration braucht, klappt die Parallelisierung nicht. Die parallele Version dieses Operators kann vor allem dazu benutzt werden, Operatoren mit verschienen Zufallszahlen auszuführen. Wie auch der ExampleSetIterator bearbeitet dieser Operator die Ergebnisse nicht. Es sollte also ein Operator folgen, der die entsprechende Nachbearbeitung der Ergebnisse vornimmt.

# 7. Empirische Untersuchungen

Nach der Auswahl und der Vorstellung der geeigneten Algorithmen, müssen sie jetzt auf ihre Tauglichkeit in einen Praxistest untersucht werden. Dabei muss eine geeignete Testumgebung gefunden und die entsprechenden Experimente für die Tests zusammengestellt werden.
Für die Leistungsprüfung stehen zwei Methoden zur Verfügung. Eine der Möglichkeiten ist die Simulation der Ausführung. Die Vorteile bestehen darin, dass man ein vorhandenes Framework nutzen kann, dass einem viel Implementierungsarbeit ersparen sollte. Auch das Erstellen und Verarbeiten von Testergebnissen wird von dem System übernommen. Außerdem können sehr leicht verschiedene Einsatzgebiete erschaffen und mit verschieden Parametern getestet werden.
Der große Nachteil der Simulation ist der fehlende Bezug zu der realen Umgebung. So entstehen viele Dinge, auf die man achten muss, wie die Auswahl der richtigen Modelle. [11] Die Parameter für die Simulation werden von dem Nutzer festgelegt und müssen nicht unbedingt die Realität wiederspiegeln. Wie schon im Kapitel 4.2 angemerkt, kann es sehr große Schwierigkeiten bei der Beschaffung von genauen Daten geben. Also müsste man bei der Simulation für sehr viele Werte Zufallsvariablen einsetzen, was das Ergebnis nutzlos machen könnte.
Wenn man für die Ergebnisse reale Werte braucht, kommt man selten um die tatsächliche Ausführung der Verfahren in einem Arbeitssystem herum. Wenn die Ausführung in einer authentischen Umgebung stattfindet, können auch die Ergebnisse viel einfacher erklärt und vor allem für weitere Vorhersagen genutzt werden. Natürlich ist es schwer ganz ohne Simulation auszukommen. Manche Ereignisse oder Parameterkombinationen sind selten, so dass die Wahrscheinlichen für das Eintreten des Ereignisses erhöhen muss, oder sogar zu einer bestimmten Zeit das Ereignis selbst initiiert werden muss. Auch müssen die Versuche geeignet vereinfacht werden, so dass zwar die Ergebnisse immer noch aussagekräftig sind, die Versuchsdurchführung aber nicht zu aufwendig und zeitintensiv wird.
Da das Ziel in einer Implementierung eines Plugins für RapidMiner bestand, bietet sich die reale Ausführung mit diesem Plugin an. Allerdings müssen einige Erweiterungen an dem Programm durchgeführt werden. Das loggen von Daten musste nachimplementiert werden, und einige Ereignisse um andere Umgebungen zu simulieren mussten hinzugefügt werden.

## 7.1. Die Experimente

Die einzelnen zur Verarbeitung stehenden Jobs können in zwei Punkten unterschieden werden. Einmal ist es die Anzahl der Tasks, in die sie zerlegt wurden. Je feinkörniger die

Zerlegung, desto größer der Kommunikationsoverhead im Vergleich zu der Rechenzeit für einzelnen Tasks. Die Optimierung des Makespans durch eine feinkörnige Jobzerlegung wird einfacher, da die einzelnen Tasks nicht so stark ins Gewicht fallen und die vielleicht am Anfang gemachten Fehler später noch korrigiert werden können. Eine grobkörnige Zerlegung lässt sich schwieriger handhaben. Meistens hat man nicht genug Tasks um Fehler auszugleichen. Der Einfluss der einzelnen Tasks auf den Makespan ist ungleich größer.
Das zweite wichtige Merkmal ist die Laufzeit. Hierbei können entweder alle Tasks die gleiche Laufzeit haben, wenn zum Beispiel die selbe Aufgabe mehrmals ausgeführt werden soll. Oder die Laufzeit der einzelnen Tasks kann sehr stark variieren, wenn man zum Beispiel mit dem ExampleSetIterator die gleiche Prozedur auf verschiedene ExampleSets anwendet. Das sind die beiden Extremen. Die Laufzeit der Tasks liegt meistens irgendwo dazwischen, wobei man anmerken muss, dass sogar zwei identische Aufgaben durch Störungen von außen wahrscheinlich verschiedene Laufzeiten haben.
Es wurden insgesamt vier Experimente erstellt, die verschiedene Merkmale der einzelnen Schedulingalgorithmen zum Ausdruck bringen sollen. Zwei mit einer feinen Jobzerlegung und zwei mit grobkörnigen Tasks. Diese Experimente werden im Folgenden genauer beschrieben. Die genaue Laufzeit der einzelnen Tasks dieser Experimente kann in den Diagrammen 7.1 abgelesen werden. Die XML-Dateien, aus denen die Experimente bestehen können im Anhang B gefunden werden.

**Feinkörnig, identische Laufzeit (FEIN-80)**
Für dieses Experiment wurde der IteratingOperatorChain Operator verwendet. Im inneren dieses Operators wurde eine zehnfache Kreuzvalidierung erzeugt, die 80 mal wiederholt wird. Da jede Wiederholung identisch ist, sollte auch die Laufzeit der einzelnen Tasks identisch sein. Durch die große Anzahl an Wiederholungen fallen die einzelnen Tasks nicht ins Gewicht. Da die Eingabe für die Tasks durch einen Generator erst auf dem Worker erzeugt wird, sind die Tasks selber sehr klein und erzeugen fast keinen Kommunikationsoverhead. Dieses Experiment stellt den Idealfall für einen Scheduler dar, da er die Tasklaufzeit nicht berücksichtigen muss.
In einer realen Umgebung ist die Art von Experimenten aber eher selten anzutreffen. Meistens gibt es zumindest leichte Variationen in den Parametern oder der Eingabe, was extreme Folgen für die Laufzeit haben kann. Trotzdem wurde diese Experiment für die Tests ausgewählt, um den einfachsten Fall für die Scheduler als Benchmark zu haben.
Wenn man sich die Laufzeiten der einzelnen Tasks anschaut, haben nicht alle Tasks die gleiche Laufzeit. Die Laufzeit der ersten Tasks ist um einiges höher. Das liegt wahrscheinlich daran, dass die JVM erst die Ressourcen bereitstellen muss und die entsprechenden Klassen von der Festplatte nachlädt. Für spätere Tasks sind sie dann bereits im Speicher vorhanden. Auch sieht man eine leichte Erhöhung in der Mitte der Ausführung. Dies kann viele Ursachen haben, vielleicht ist zum Beispiel zur dieser Zeit der Java Garbage Collector angesprungen, der für die Freigabe des nicht mehr benötigten Speichers zuständig ist. Da die Laufzeit ansonsten gleich ist, wurde sie bei allen Tasks mit 500 angegeben.

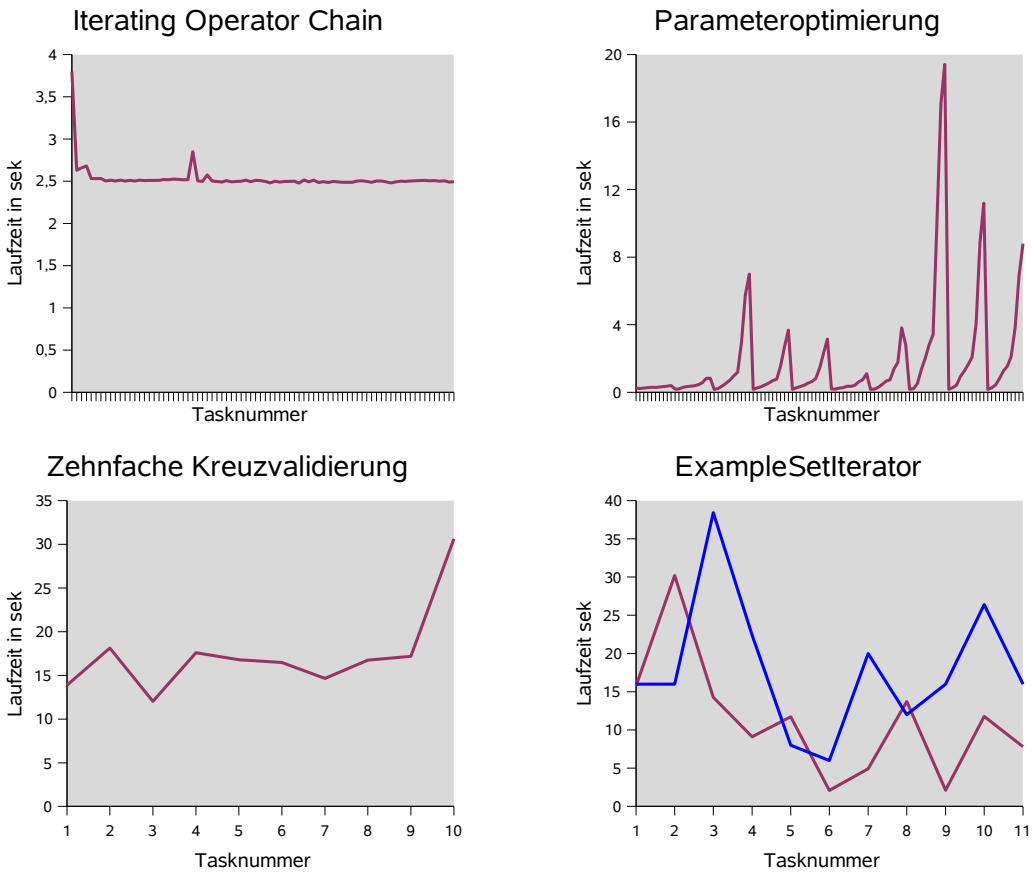



Abbildung 7.1.: Tasklaufzeiten der einzelnen Jobs. Die blaue Linie bei dem ExampleSet-Iterator ist die vorher geschätzte Laufzeit. Die roten Linien entsprechen den tatsächlichen Laufzeiten.

**Feinkörnig, unbekannte Laufzeit (FEIN-100)**
Für den zweiten Versuch wurde der ParameterOpimization Operator benutzt. Es sollten die Parameter für eine Support Vector Machine optimiert werden. Insgesamt wurden 100 Parameterkombinationen zum testen Ausgewählt. Es ergaben sich also 100 Tasks, die die gleichen Operatoren auf den gleichen Eingabedaten ausgeführt haben. Allerdings führten verschiedene Parameter zu einer großen Variation in der Laufzeit, die sich nicht vorhersagen ließ. Da die Laufzeiten der meisten Tasks unter einer Sekunde liegen, fällt hier der Overhead für die Kommunikation stark ins Gewicht. Als Tasklaufzeiten wurden für diese Tasks der Wert 1000 genommen. Dadurch sollte überprüft werden wie stark das Ergebnis die heuristischen Schedulingalgorithmen von der Exaktheit der Tasklaufzeiten abhängt. Um bei diesem Test gut abzuschneiden, darf sich das Schedulingverfahren nicht auf die Tasklaufzeit verlassen und muss vor allem versuchen zum Ende des Experiments

die Last der einzelnen Rechner möglichst gut auszugleichen. Alle statischen Algorithmen sollten hierbei versagen, da sie nach der Anfangsphase keine Änderungen an der Zuordnung mehr vornehmen.

**Grobkörnig, ähnliche Laufzeit (GROB-10)**
Die ersten beiden Experimente haben eine relativ große Anzahl an Tasks, die es ermöglichen später noch steuernd in die Ausführung einzugreifen. Also wurde auch ein Experiment erstellt, bei dem das Verhältnis zwischen den ausführenden Maschinen und den Tasks sehr gering ist. Für das erste Experiment wurde eine zehnfache Kreuzvalidierung verwendet. Die Laufzeit einzelner Validierungsschritte sollte mehr oder weniger gleich sein, da sie alle auf den gleichen Operatoren basieren, und die Änderungen in den Daten der einzelnen Tasks gerade mal 10% betragen. Das sollte zwar die Gleichheit der Tasks nicht garantieren, aber doch mehr oder weniger zu einer ähnlichen Laufzeit für einzelne Tasks führen. Als Tasklaufzeiten wurden jedes Mal zufällig die Werte zwischen 500 und 600 generiert. So dürfte jeder Lauf mit den Schedulingverfahren, die die Laufzeiten berücksichtigen anders sein.

Nach den ersten Tests hat sich herausgestellt, dass die Ähnlichkeit der Tasklaufzeiten nicht garantiert werden konnte. Wie man auf dem Bild sehen kann sind die ersten 9 Validierungen mehr oder weniger gleich, die letzte Validierung dauert aber mehr als doppelt so lange. Damit ist es ein „worst case“ Szenario für jeden Scheduler. Da nur 10 Tasks zur Verfügung stehen, wirkt sich eine falsche Platzierung extrem auf den Makespan aus, da sie nicht mehr kompensiert werden kann. Sogar wenn man eine Kompensierung versucht, wird der letzte Task durch seine sehr lange Laufzeit die Lastverteilung wieder kaputtmachen. Es ist überhaupt nicht klar, ob es einen Scheduler gibt, der mit diesem Experiment überhaupt gute Ergebnisse liefert, oder ob die Größe des Makespans vom Zufall abhängt.

**Grobkörnig, variierende Laufzeit (GROB-11)**
Das zweite Experiment mit einer grobkörnigen Struktur. Hier wurde der ExampleSetIterator verwendet. Zuerst wurden elf ExampleSets unterschiedlicher Größe generiert. Auf jedem der ExampleSets wurde eine Kreuzvalidierung durchgeführt. Es entstanden also 11 unterschiedliche Tasks. Da in jedem Task die gleiche Arbeit ausgeführt wird, unterscheiden sie sich nur in der Eingabe - also der Größe des ExampleSets. Dadurch ist die Laufzeit der Tasks zwar unterschiedlich, kann aber trotz dem geschätzt werden. Für die Tasklaufzeiten wurden die Anzahl des Beispiel im ExampleSet mit der Anzahl der Merkmale pro Beispiel multipliziert. Da die erzeugten Tasks zwischen 50 und 1000 Beispiele mit 3 bis 10 Attributen hatten, bewegten sich die erwarteten Tasklaufzeiten zwischen 400 und 4800. Auf dem Bild 7.1 zeigt die rote Linie den tatsächlichen Verlauf der Tasklaufzeiten, die blaue den Berechneten. Auch hier fällt es auf, dass die beiden Werte von einender Abweichen. die Größe der Eingabe hatte also bei diesem Versuch einen kleinen Einfluss auf die Ausführungsdauer.

| Name | Prozessor | Ram | OS | Benchmark |
|---|---|---|---|---|
| kiepe | 1x UltraSparc IIe 650MHz | 1024 MB | SunOS 5.9 | 194 |
| kieme | 2x UltraSparc III+ 1015MHz | 2048 MB | SunOS 5.10 | 599 |
| kiefer | 2x UltraSparc-IIIi 1280 MHz | 2048 MB | SunOS 5.9 | 788 |
| kiez | 2x UltraSparc-IIIi 1280 MHz | 2048 MB | SunOS 5.9 | 789 |
| kiew | 2x UltraSparc-IIIi 1280 MHz | 2048 MB | SunOS 5.10 | 792 |
| kino | 2x UltraSparc-IIIi 1280 MHz | 2048 MB | SunOS 5.10 | 791 |
| ls8olc00 | Dual Opteron 248 2,2 GHz | 16384 MB | 64bit Linux | 3008 |
| ls8olc01 | Dual Opteron 248 2,2 GHz | 12228 MB | 64bit Linux | 3019 |
| ls8olc02 | Dual Opteron 248 2,2 GHz | 8192 MB | 64bit Linux | 3005 |
| kimme | Athlon 2x MP 2100+ | 2048 MB | 32bit Linux | 754 |
| king | Athlon 2x MP 2100+ | 2048 MB | 32bit Linux | 782 |
| kippe | Athlon 64 X2 4600+ | 1024 MB | 32bit Linux | 966 |
| kilo | Athlon 64 X2 4600+ | 1024 MB | 32bit Linux | 970 |
| kibosh | P4 3GHz | 1024 MB | 32bit Linux | 235 |

Tabelle 7.1.: Liste der verfügbaren Rechner

## 7.2. Die Rechnerumgebung

Die Rechnerumgebung hat einen sehr großen Einfluss auf die Leistungsfähigkeit der Schedulingverfahren. Manche Algorithmen sind für spezielle Umgebungen angepasst worden, und sollten in diesen gute Ergebnisse liefern. Es ist aber auch interessant zu sehen, ob sie ohne einen großen Leistungsverlust in anderen Umgebungen eingesetzt werden könnten und sich somit auch für allgemeine Aufgaben eignen. Die Details der einzelnen Maschinen können in der Tabelle 7.1 nachgeschaut werden. Zu jedem Rechnernamen ist der Prozessor, die Größe des Arbeitsspeichers und das Betriebsystem aufgeführt. Außerdem ist in der letzten Spalte das gemittelte Ergebnis der zehnfachen Ausführung des Benchmarks eingetragen. Für die Experimente wurden vier verschiedene Umgebungen ausgewählt, die sich nah an die real eingesetzte verteilte Rechnersysteme anlehnen.

**Homogene Rechnerumgebung**
Für die ersten Test wurde eine klassische Rechnerumgebung ausgewählt. Als Worker wurden jeweils 3 der 4 baugleiche Solarisrechner eingesetzt. Diese Rechner haben, bis auf das Betriebssystem eine identische Konfiguration und sollten die Tasks gleich schnell ausführen können. Da all diese Rechner über zwei Prozessoren verfügen, sollten die Schwankungen der Rechenleistung, die durch andere Systemprozesse verursacht werden, sehr gering sein. Diese Maschinenumgebung entspricht dem $P_m|p_j|C_{max}$ Schedulingproblem aus der Theorie.
Diese Umgebung soll einen einfachen Rechnercluster mit dedizierten identischen Maschinen repräsentieren. Die Fähigkeiten der einzelnen Rechner sind gleich, und so sollten sie auf den gleichen Tasks eine sehr ähnliche Laufzeit erzeugen. Das ist die perfekte Umgebung für Schedulingverfahren, da sie die Rechenleistung komplett aus ihren Be-

rechnungen streichen können.

**Heterogene Rechnerumgebung**
Die zweite Umgebung sollte aus einer Rechnerfarm, der Maschinen unterschiedlicher Leistung angehören, bestehen. Neben den Rechnern aus der ersten Umgebung wurden zusätzlich die beiden Solarismaschinen ***kieme*** und ***kiepe*** hinzugefügt. Außerdem kamen noch 2 der schnellen Opteron Rechner hinzu. Wie man an den Benchmarkdaten aus der Tabelle 7.1 sieht, ist die Leistung der einzelnen Rechnergruppen sehr unterschiedlich. Vor allem ***kiepe*** ist sehr langsam und hat nur einen Prozessor, was bedeutet, dass die Systemprozesse die Ausführung der Experimente stören können. Ansonsten sollten die Benchmarkwerte konstant bleiben. Einmal ermittelt, können sie für die gesamte Dauer des Experiments verwendet werden. Diese Umgebung kann mit $Q_m|p_j, v_i|C_{max}$ charakterisiert werden und stellt einen dedizierten Rechnercluster mit verschiedenen Maschinen dar. Zwar ist es eine Standardkonfiguration, aber Algorithmen, die von gleichartigen Rechnern ausgehen, werden mit dieser Umgebung große Schwierigkeiten haben.

**Variierende Last**
Die ersten beiden Umgebungen stellten einen dedizierten Cluster dar. Das heißt, die einzige Aufgabe dieser Rechner besteht darin, die Aufgaben, die ihnen der Master senden, auszuführen. Viel öfter ist es aber der Fall, dass ein Rechner nicht seine komplette Leistung für eine Aufgabe verwenden kann. Entweder ist gleichzeitig ein lokaler Nutzer angemeldet, oder es gibt andere Verbindungen von außen, die auch CPU-Leistung benötigen. Leider war es mir nicht möglich eine solche Umgebung zu erzeugen, also musste sie simuliert werden. Die Leistung der Rechner musste über die Zeit schwanken und diese Schwankung musste zufällig sein, da sie von außen verursacht ist. Um dies zu erreichen, wurden die Laufzeiten der Tasks variiert. Nach dem Ende der Ausführung wird der Rechner für 0 bis 30 Prozent der Tasklaufzeit in den Schlaf versetzt. Dadurch ist die ungefähre Leistung der Rechner zwar bekannt, nützt aber nicht viel, da sie sehr ungenau ist. Algorithmen, die sich auf die Rechenleistung verlassen, sollten hier schlechtere Ergebnisse liefern.

**Instabile Umgebung**
Die letzte Umgebung soll eine dynamische Umgebung simulieren, in der die Rechner ausfallen können und neue Rechner dazu kommen. Um dies zu erreichen wird eine Wahrscheinlichkeit für einen Ausfall des Rechners während der Ausführung eingeführt. Diese Wahrscheinlichkeit liegt bei 5% und wird alle 7 Sekunden geprüft. Sollte ein Rechner nicht mehr antworten, so muss der Master die entsprechenden Tasks an andere Arbeiter verteilen. Nach einer gewissen Pause, wird der Arbeiter wieder gestartet und steht damit wieder dem Master zur Verfügung. Allerdings gehen dabei alle zuvor an diesen Arbeiter geschickten Daten verloren. Durch diese Implementierung werden die Verbindungsabbrüche und Rechnerabstürze simuliert. Alle Algorithmen, die mehr als einen Task auf ein Mal an einen Arbeiter verschicken, werden bei dieser Umgebung einen größeren

Transferoverhead erzeugen. Da die meisten Verfahren nicht für eine solche Umgebung ausgelegt sind, mussten sie erweitert werden. Falls ein Rechner ausfällt, werden alle an ihn geschickten Tasks wieder in die Liste der offenen Tasks hinzugefügt und neu zugeordnet. Das bricht vor allem der Vorsatz von statischen Verfahren, die eigentlich ihre Arbeit mit der Zuordnung am Anfang der Ausführung erledigt haben. Würde man diese Änderung nicht vornehmen, würden die Verfahren ewig auf die Antwort des Workers warten.

## 7.3. Erfasste Messwerte

Bei der Ausführung der Versuche können viele verschiedenen Informationen aufgezeichnet werden. Für die Auswertung der Algorithmuseigenschaften wurden 4 Werte ausgewählt. Das sind:

**Der Makespan**
Wie schon im Kapitel 4 beschrieben bezeichnet der Makespan die Zeit, die für die Fertigstellung der gesamten Aufgabe benötigt wird. Das ist der (einzige) wichtige Wert für den Benutzer, und auch der einzige nach außen sichtbare Messwert, der aufgezeichnet wurde. Die Erfassung dieses Wertes ist einfach. Es wird einfach die Zeitdifferenz zwischen dem Beginn und dem Ende des Experiment ermittelt.

Aufgezeichnet wurde hier der Mittelwert über alle Experimente, um die durchschnittliche Leistung jedes Verfahrens zu testen. Außerdem wurde der minimale und der maximale erreichte Wert angegeben, um die beste und die schlechteste erreichte Zeit zu haben. An diesen Zahlen kann man gut ablesen, wie zuverlässig der Durchschnittswert erreicht wird. Bei manchen Experimentserien wurde auch die Standardabweichung angegeben. Dabei war die Entwicklung der Abweichung bei Veränderung der einzelnen Experimentparameter interessant.

**Auslastung der Arbeiter**
Während der Experimentlaufzeit sind die einzelnen Arbeiter nicht immer mit den Aufgaben versorgt. Am Anfang braucht der Master eine gewisse Zeit, um die Tasks zu erstellen und eine passende Zuordnung der Tasks an die Maschinen zu berechnen. Da bei meinem Plugin die Entdeckung der Arbeiter nach dem Start des Experiments stattfindet, startet der Scheduler erst nach einer Sekunde Laufzeit, um den Workern im Netz Zeit für einen Verbindungsaufbau zu geben, da diese sonst besondern bei den statischen Algorithmen nicht für das Scheduling berücksichtigt werden. Auch zwischen den Taskausführungen entstehen Leerlaufzeiten. In diesen Zeiten muss das Ergebnis an den Master gesendet und ein neuer Task empfangen werden. Für nähere Informationen siehe Kapitel 5. Der letzte große Block der Leerlaufzeit kommt am Ende des Experimentes. Meistens ist es unmöglich, die Arbeit auf die Rechner so zu verteilen, dass sie exakt zum gleichen Zeitpunkt mit der Ausführung fertig werden. Also müssen die Rechner, die früher fertig geworden sind auf den Rest warten. Ist kein anderer Master vorhanden, der sie mit Aufgaben versorgen kann, entsteht eine Wartezeit, während der die Ressourcen der Worker unbenutzt sind.

Auch bei dem Leerlauf wurde dem minimale der durchschnittliche und der maximale Wert aufgezeichnet. Allerdings war die Berechnung dieser Werte etwas aufwendiger. Da es bei jedem Experiment mehrere Worker gab, müssten erst bei jedem Experiment die minimalen, durchschnittlichen und maximalen Leerlaufzeiten der Worker ermittelt werden. Im zweiten Schritt wurden diese 3 Zahlen aus jedem Experiment über alle Experimentausführungen gemittelt. So gibt jede dieser Zahlen den durchschnittlichen Wert, der bei den Experimenten erreicht wurde. Dabei ist zu Beachten, dass der minimale Wert nicht unter eine Sekunde sinken kann, da alle Verfahren erst nach einer Sekunde aktiv wurden, um den Workern Zeit zu lassen, sich mit dem Master zu verbinden.

**Transferoverhead**
Bevor ein Task auf einem externen Rechner ausgeführt werden kann, müssen die entsprechenden Daten übermittelt werden. Manche Algorithmen verschicken mehr als eine Aufgabe auf ein Mal, oder verschicken sie gleich an mehrere Rechner. Kommt es zu Verbindungsabbrüchen oder reagiert der Arbeiter aus irgendwelchen Gründen nicht mehr, müssen diese Daten erneut an andere Rechner übertragen werden. Es werden also mehr Übertragungen durchgeführt, als Aufgaben da sind. Der gemessene Wert setzt einfach die Anzahl der Tasks mit der Anzahl der Übertragungen ins Verhältnis und gibt an, wie ökonomisch der Master mit der Bandbreite umgeht. Um diesen Wert zu erfassen, wurden alle erfolgreichen Aufgabentransfer an jeden Arbeiter während des Experiments zusammengezählt. Wenn es also während der Übertragung zu Problemen kann, oder das Experiment vor der kompletten Übertragung endete taucht diese Zahl hier nicht auf.

**Rechenoverhead**
Dieser Messwert ist dem Transferoverhead sehr ähnlich. Allerdings bezieht er nur die Aufgaben mit ein, die auch tatsächlich auf dem Zielrechner gestartet wurden. Wichtig ist dieser Wert vor allem für Algorithmen, die Replikation oder Migration einsetzen. Dies kann zwar den Makespan senken, da man nur das erste Ergebnis nimmt und die nachfolgenden einfach verwirft, erhöht aber auch die globale Rechnerbelastung. Bei der Ausführung von einzelnen Experimenten fällt die Mehrbelastung nicht ins Gewicht. Wenn allerdings mehrere Master auf die selben Worker zugreifen, oder eine ganze Serie von Experimenten hintereinander ausgeführt werden soll, kann der Makespan wegen der vielen überflüssigen Berechnungen sogar erhöht werden. Es muss auch beachtet werden, das eine Replikation den Leerlauf des Rechners senkt, da anstatt des Wartens die neue Aufgabe ausgeführt wird.
Für die Angabe des Transfer- und Ausführungsoverheads wurde das gemittelte Ergebnis über die gesamte Versuchsreihe genommen. Dabei wurde zuerst die Anzahl der Gesamtübertragungen/Ausführungen berechnet. Von diesen Wert wurde die Anzahl der Tasks abgezogen, so dass das Endergebnis tatsächlich nur den Overhead repräsentiert. Dabei sind diese Werte absolut. Bei einem grobkörnigen Job ergibt sich bei den gleichen Werten ein ganz anderes Verhältnis als bei einem feinkörnigen Job.

## 7.4. Erste Versuchsserie - ein Master

Für die erste Versuchsserie wurde ein Rechner als Master ausgewählt und von diesem aus die einzelnen Versuche durchgeführt. Dabei fungierte eine der AMD Linux Maschinen als Master. Welcher der Linuxrechner als Master fungierte hat sich als unwichtig herausgestellt. Die Maschinen unterscheiden sich in ihrer Geschwindigkeit nicht stark. Auch werden keine rechenintensiven Berechnungen auf dem Master angestellt. Er befindet sich die meiste Zeit im Leerlauf. Die Verarbeitung der einzelnen Ergebnisse passiert während der Bearbeitung der Tasks auf den anderen Rechnern, so dass diese Zeit nicht in den Makespan eingeht. So liegt der Einfluss des Masters auf die Größe des Makespans in einem sehr geringen Bereich.
Da es bei dieser Versuchsserie nur einen Master im Netz gab, wurden die Tasks auf den Workern sofort ausgeführt. So entstand keine Wartezeit durch andere Master. Damit waren die äußeren Einflüsse stark beschränkt. Allerdings ist es nicht einfach eine exakt gleiche Umgebung für alle Experimente bereitzustellen. Die Ergebnisse schwanken daher leicht. Auch durch die lange Dauer einer Testserie konnten leichte Veränderungen an der Rechnerlast nicht ausgeschlossen werden.
Um den Einfluss von außen auf die Ergebnisse zuminimieren wurde jeder Versuch 20 mal ausgeführt. Da es insgesamt 12 Schedulingverfahren mit 4 verschiednen Experimenten gab wurde pro Versuchreihe 960 Versuche durchgeführt. Die kompletten Ergebnisse der ersten Versuchsserie können in den Tabellen C.1 bis C.16 angeschaut werden

### 7.4.1. Homogene Umgebung

Um einen Benchmark für die zukünftigen Versuche zu haben, wurde als erstes die einfachste Testreihe durchgeführt. Dieser Test fand in einer homogenen Umgebung statt. Als Worker wurden jeweils 3 der 4 baugleichen Solarisrechner ausgewählt. Da die Rechner identisch waren, spielte die Wahl des Rechners bei der Zuordnung keine Rolle. Nur die Anzahl der zugeordneten Tasks war ausschlaggebend für das Ergebnis. In der Tabelle C.3 kann man die Ergebnisse des einfachsten Versuchs sehen. Hier haben sogar die einzelnen Tasks die gleiche Laufzeit. Uns so schneiden alle Verfahren gleich gut ab. Das Ergebnis liegt konstant um die 62 Sekunden. Nur die beiden Algorithmen, die diese Tasks zufällig an die Worker verteilen, haben ein um 20% schlechteres Ergebnis. Allerdings war es zu erwarten, da jede Abweichung von der optimalen Zuordnung den Makespan erhöht. Auch der Leerlauf ist sehr niedrig. Da bei diesen Tasks ein ExampleSetGenerator benutzt wurde, und so die Eingabe für das Experiment erst auf dem Worker generiert wurde, ist die Taskgröße sehr klein, so dass die Latenzzeiten durch das Senden nicht ins Gewicht fallen. Zwischen dem Versenden des Ergebnisses und dem Empfangen einer neuen Aufgabe vergingen hier jedes Mal ca. 30 ms. Bei 80 Tasks, die auf 3 Rechner verteilt werden sind das im Durchschnitt 27 Tasks oder maximal 810 ms Leerlauf pro Rechner. Hinzu kommt noch die eine Sekunde Initialisierung zu Beginn der Ausführung. Aber auch hier zeichnen sich die beiden Zufallsalgorithmen durch schlechtere Zahlen aus, da durch die ungleichmäßige Verteilung vor allem zum Ende der Ausführung ein großer Leerlauf

entsteht.
Der Transferoverhead bei den Replikationen ist ebenfalls sehr niedrig. Die Replikation fand bei den einfachen Verfahren im Schnitt 2 mal statt. Bei der TimeOut Variante nur ein mal. Das liegt zum einen an der niedrigen Anzahl der Rechner. Wenn der erste Rechner frei wird, sind insgesamt nur noch höchstens 2 Tasks am laufen, was die Anzahl der Replikationen sehr beschränkt. Zum anderen die spielt die Homogenität eine Rolle. Die Ausführungszeit der Tasks ist konstant, so dass der TimeOut nicht eintritt.
Bei den anderen 3 Experimenten zeigen sich die ersten Unterschiede in den einzelnen Algorithmen. Interessant ist das Ergebnis des feinkörnigen Jobs mit unbekannter Tasklaufzeit. Die Ergebnisse dieses Tests sind in der Tabelle C.4 zusammengefasst. Obwohl hier die Laufzeit der Tasks unbekannt war, und sehr stark schwankte, liegen die Ergebnisse der einzelnen Verfahren sehr nah bei einander. Auch die Heuristiken liefern hier gute Ergebnisse, obwohl die tatsächliche Tasklaufzeit stark von der angegeben Tasklaufzeit abgewichen ist. Das liegt an der Verteilung der Tasklaufzeiten innerhalb des Jobs. Die meisten Algorithmen verteilen die Tasks nach dem Round Robin Prinzip an die einzelnen Arbeiter. So wurde die Last hier mehr oder weniger gleichmäßig an mehrere Worker verteilt. Wenn man von dem Round Robin Prinzip zu der FIFO Methode übergeht werden die Ergebnisse viel schlechter. Das sieht man sehr gut an dem $FIXED - W$ Algorithmus, der zuerst einem Worker komplett mit Arbeit versorgt, bevor er zum nächsten übergeht. So ergibt sich die doppelt so lange Laufzeit gegenüber dem einfachen $FIXED$ Algorithmus. Da die Tasks mit der extrem langen Laufzeit zum Ende der Ausführung auftauchen, wurde der Grossteil von ihnen an dem gleichen Rechner zugeordnet, was den große Unterschied im Makespan erklärt. Zwar arbeiten auch die dynamischen Scheduler $TSS$ und $FAC$ nach diesem Prinzip. Hier hilft die abnehmende Blockgröße (zum Schluss die Größe eins) die Unausgewogenheit der Last zu minimieren, und so schneiden sie nicht viel schlechter aus die anderen Verfahren ab. Der $WQ$ Algorithmus schneidet hier auch etwas schlechter ab. Die größere Leerlaufzeit resultiert aus der Notwenigkeit vor der Taskausführung jedes mal zuerst den Task zu deserialisieren. Die statischen Algorithmen haben dieses Problem nicht. Sie können diese Operationen währen der Ausführung anderer Tasks vornehmen. Da es sich hierbei um Mehrprozessormaschinen handelt, können diese Berechnungen auf dem zweitem Prozessor durchgeführt werden und beeinträchtigen somit die Dauer der Taskausführung nicht.
Bei der 10fachen Kreuzvalidierung sind die Ergebnisse wieder sehr ähnlich. Wenn man sich die Entwicklung der Tasklaufzeiten anschaut, wird auch schnell klar warum. Der letzte Task hat fast die doppelte Laufzeit als der Rest. Wenn man die Tasks auf die Rechner verteilt, kriegt jeder Rechner 3 Tasks. Es bleibt also nur noch der letzte Task übrig, und er bestimmt dann auch den Makespan. Ein Rechner führt diesen Task aus, die anderen beiden befinden sich im Leerlauf. Um so liegt der maximale Leerlauf bei den meisten Verfahren um die 30 Sekunden. Auch die Replikation hilft hier nicht. Da alle Rechner gleich schnell sind, kann das spätere nochmalige Starten des letzten Task das Ergebnis nicht verbessern.

| Feinkörnig, identische Laufzeit | | | | |
|---|---|---|---|---|
| Worker | 1 | 2 | 3 | 4 |
| Makespan | 180.9 | 94.2 | 63.3 | 48.6 |
| Speedup | — | 1.92 | 2.85 | 3.73 |
| Effezienz | — | 0.96 | 0.95 | 0.93 |

| Grobkörnig, ähnliche Laufzeit | | | | |
|---|---|---|---|---|
| Worker | 1 | 2 | 3 | 4 |
| Makespan | 152.9 | 103.5 | 78.3 | 65.1 |
| Speedup | — | 1.48 | 1.95 | 2.35 |
| Effezienz | — | 0.74 | 0.65 | 0.59 |

| Feinkörnig, unbekannte Laufzeit | | | | |
|---|---|---|---|---|
| Worker | 1 | 2 | 3 | 4 |
| Makespan | 175.1 | 99.5 | 69.5 | 54.3 |
| Speedup | — | 1.76 | 2.55 | 3.23 |
| Effezienz | — | 0.88 | 0.84 | 0.81 |

| Grobkörnig, unterschiedliche Laufzeit | | | | |
|---|---|---|---|---|
| Worker | 1 | 2 | 3 | 4 |
| Makespan | 120.4 | 65.9 | 44.5 | 35.8 |
| Speedup | — | 1.83 | 2.71 | 3.36 |
| Effezienz | — | 0.91 | 0.9 | 0.84 |

Tabelle 7.2.: Effizienz der parallelen Ausführung der Tasks

**Speedup und Effizienz**

Diese Versuchsreihe eignet sich hervorragend um den Speedup der parallelen Ausführung gegenüber der Sequenziellen zu messen. Unter Speedup versteht man beim verteiltem Rechner den Geschwindigkeitsgewinn, den man bei dem Verteilen der Berechnung auf mehrere Prozessoren erhält. Die Allgemeine Formel lautet:

$$S_p = \frac{T_1}{T_p} \tag{7.1}$$

Wobei $T_1$ die Ausführungszeit des Jobs auf einem Prozessor und $T_p$ auf $p$ Prozessoren ist. Aus dem Speedup lässt sich auch die Effizienz $E_p$ der parallelen Ausführung berechnen, die sich als

$$E_p = \frac{T_p}{p} \tag{7.2}$$

definiert. Da sich alle Algorithmen in der homogenen Umgebung mehr oder weniger gleich verhalten, wurde der einfache Work Queue Algorithmus für diesen Test genommen. Es wurden jeweils die Experimente 10 mal ausgeführt und die Ergebnisse gemittelt. Der sequentielle Task mit nur einem Prozessor wurde mit Hilfe der normalen RapidMiner GUI gemacht. Hier entfällt also der Overhead für den Transport der Tasks und die Serialisierung.

Die Tabelle 7.2 zeigt die Ergebnisse. Diese Werte spiegeln die Ergebnisse der vorher ausgeführten Experimente wieder. Bei den feinkörnigen Jobs fällt die Effizienz nur sehr langsam ab, was für eine gute Parallelisierbarkeit spricht. Allerdings liegt die Effizient bei dem Experiment mit den unbekannten Laufzeiten mit 2 Rechnern 8 Prozentpunkte unter dem Job mit gleichen Tasklaufzeiten. Hier sieht man deutlich den Overhead durch den Transport der Nachrichten und der Serialisierung. Bei den Grobkörnigen Tasks erreicht die Kreuzvalidierung die neidigsten Werte, da hier gleich zwei Faktoren negativ auf die Parallelisierung auswirken.

Wie man sieht erreicht keiner der Jobs eine 100%ige Effizienz. Es gibt immer noch einen fixen Anteil, der auf den Master ausgeführt werden muss. Dieser Anteil fällt bei mehr

Prozessoren (also kürzerer Ausführungszeit) immer mehr ins Gewicht. Außerdem steigt auch der Aufwand für die Verwaltung der extra Worker und die Zuordnung der Tasks. Deshalb fällt die Effizienz der parallelen Ausführung immer weiter ab.
Nachdem man die Ergebnisse in einer perfekten Umgebung gesehen hat, ist die nächste Frage, ob es in jeder Situation besser ist, wenn mehr Rechner für die Ausführung zur Verfügung stehen. Das heißt, ob sich der Speedup mit der Erhöhung der Rechneranzahl auch erhöht, oder vielleicht doch in bestimmten Situationen gleich bleibt, oder sogar fällt.
Natürlich ist es leicht triviale Beispiele für solche Situationen zu finden. Man muss zum Beispiel nur mehr Rechner nehmen, als es Tasks gibt, und schon haben einige nichts mehr zu tun. Der Speedup bleibt gleich, die Effizienz fällt. Es soll aber versucht werden, weniger extreme Beispiele für solche Situationen zu finden. Es soll also weniger Rechner als Tasks geben, so dass theoretisch alle Rechner mit den Berechnungen beschäftigt werden können.
Bei den nächsten Versuchen wurde mit Veränderungen der Rechnerumgebung versucht, Situationen zu finden in denen manche Verfahren versagen, um so die Einsatzmöglichkeiten für diese Algorithmen abzuschätzen.

### 7.4.2. Heterogene Umgebung

In der zweiten Versuchsreihe wurden zusätzliche Rechner zu der vorherigen Umgebung hinzugefügt. Neben den sehr schnellen Linuxrechnern ***ls8olc00*** und ***ls8olc01*** auch der sehr langsame Rechner ***kiepe***. Jetzt gab es deutliche Unterschiede in der Ausführungszeit zwischen den einzelnen Verfahrensarten. Vor allem der $FIXED$ Algorithmus schneidet ganz schlecht ab. Da er an jeden Rechner die gleiche Anzahl von Tasks zuordnet, werden die langsamen Rechner viel zu stark belastet. Um befinden sich die schnellen Rechner die meiste Zeit im Leerlauf.
Der $RANDOM$ Algorithmus liefert ähnliche Ergebnisse. Auf lange Sicht sollte dieser Algorithmus die gleiche Anzahl der Task an jedem Rechner verteilen und die gleiche Verteilung wie $FIXED$ erreichen. Aber anders als der homogenen Umgebung führt eine Abweichung nicht unbedingt zu einer Verschlechterung des Ergebnisses. Und so schwankt die Ausführungszeit bei diesem Verfahren sehr start, und kommt auf keiner Weise an die besseren Verfahren heran.
Bei den Grobkörnigen Jobs schneiden vor allem die Heuristiken überraschen gut ab. Von allen Verfahren liefern sie den besten Makespan, obwohl die angegebenen Tasklaufzeiten in keime Bezug zu den tatsächlichen Tasklaufzeiten stehen. Sie sind sogar besser als die normalen dynamischen Algorithmen. Das liegt daran, dass sie bei der Verteilung der Tasks, nicht die aktuelle Last, sondern die Zukünftige Last auf einem Rechner zur Berechnungen verwenden. Es wird festgestellt, dass die Zuordnung von einem Task zu einem sehr langsamen Rechner wie ***kiepe***, einen sehr großen Einfluss auf die zukünftige Last hat, so wird dieser Rechner vernachlässigt. Die dynamischen Verfahren hingegen sehen nur dass ein Rechner sich gerade im Leerlauf befindet, und weisen ihm den nächsten Arbeitsblock zu, was den Makespan sehr stark erhöhen kann. Der $S-WQR$ Algorithmus kann diese Entscheidung ohne Probleme rückgängig machen, da er den entsprechenden

Task einfach an mehrere Maschinen verteilt. Bei dem $TO - WQR$ gelingt dies nicht immer, da er vor dem Replizieren den Timeout abwarten muss. Allerdings wird diese Leistungssteigerung teuer erkauft. Bei dem $S - QWR$ Verfahren kann die zusätzliche Ausführungsoverhead bei 70% der Tasks liegen.
Es stellt sich hier die Frage, wie wichtig die korrekten Tasklaufzeiten für die Heuristiken sind. Diese Verfahren sollten bei den Experimenten eigentlich versagen, da sie falsche Informationen über die Eingabe hatten mit Abweichungen von mehreren 100% von dem tatsächlichen Wert. Und doch schnitten sie sehr gut ab. Es wird immer mehr ersichtlich, dass das gute Abscheiden der Heuristiken auf das Vernachlässigen der langsamen Rechner zurück zu führen ist. So spielen dann die Schwankungen der Tasklaufzeiten eine viel kleinen Rolle, da sie auf den schnellen Rechnern immer noch innerhalb einer sehr kurzen Zeit ausgeführt werden können. Dafür spricht auch der große Leerlauf einiger Rechner bei diesen Verfahren. Sogar bei den feinkörnigen Jobs waren bei einer Gesamtlaufzeit von 28 Sekunden manche Rechner ca. 23 Sekunden nicht ausgelastet. Dabei handelte es sich die meiste Zeit um den langsamsten Rechner ***kiepe***. Da seine Leistung nicht mal 7% der Leistung der schnellen ***ls8olc*** Rechner beträgt, ist auch sein Beitrag zu der Gesamtleistung sehr gering. Andersrum ist es zum Beispiel bei den $TCC$ und $FAC$. Hier sind die maximalen Leerlaufzeiten zwar kleiner, aber der Makespan größer. Durch die Mindestblockgröße von 1 haben die langsamen Rechner zum Ende der Jobausführung meistens immer noch was zu tun, und stellen oft die Bremse im System dar. In diesem Fall entsteht der Leerlauf auf den schnellen Rechnern, was viel mehr Leistung verschwendet, und so den Makespan erhöht.
Bei den Jobs feinkörnigen Tasklaufzeiten können die dynamischen Algorithmen aufholen. Jetzt liegen sie mit den Heuristiken gleich. Der Grund, ist die geringe Laufzeit der einzelnen Tasks, so dass die Zuordnung von einem Task an einem langsamen Rechner kein großes Ungleichgewicht mehr erzeugt.
Ein Beispiel für die Unterschiedliche Vorgehensweise der Scheduler kann man in der Abbildung 7.2 sehen. Da nur 10 Tasks zur Verfügung stehen, lässt die Heuristik die beiden langsamsten Maschinen komplett außen vor. Das kann der TSS Algorithmus nicht machen, da er mindestens einen einen Task an jede Maschine zuordnet. Die Replikation führt insgesamt 19 Tasks aus. Nach dem der Job fertig ist, sind immer noch 6 Tasks offen. Obwohl das Experiment abschlossen ist, und die kompletten Ergebnisse bereits vorliegen, werden die restlichen Tasks trotzdem ausgeführt. Auf dem Rechner ***kiepe*** ist kein einziger Task fertig geworden, was für die Korrektheit der Entscheidung der Heuristik spricht, diesen Rechner frei zu lassen. In der letzten Abbildung ist das Ergebnis der zufälligen Verteilung. Man sieht sofort die Unausgewogenheit. Der schnellste Rechner hat keinen einzigen Task zugeteilt bekommen, der langsamste dafür 3.

### 7.4.3. Externe Last

Die ersten beiden Umgebungen stellten einen dedizierten Cluster dar. In der nächsten Versuchsreihe wird eine externe Komponente eingeführt. Es gibt jetzt andere Benutzer, die die gleichen Rechner zur selben Zeit benötigen, und so auf ihnen eine extra Last produzieren. Anfänglich sollte das Experiment mit einer Schwankung von höchstens


Abbildung 7.2.: Ergebnis des Schedulings einer Kreuzvalidierung. Die grünen Balken sind abgeschlossene Tasks. Bei der Replikation sind auch noch offene Tasks sichtbar.

30% in der Laufzeit ausgeführt werden. Doch war dieser Wert zu klein, um irgendwelche Aussagen zu machen, darum wurde noch ein zweiter Versuch durchgeführt, der eine Verlängerung der Ausführungszeit um maximal 100% erlaubt.

Da die Laufzeiten der einzelnen Tasks schwanken, schwanken damit auch die Ausführungszeiten der kompletten Jobs. Deswegen ist die erwartete Laufzeit bei jeder Durchführung eines Versuchs anders. Allerdings sollten sich auf lange Sicht die Schwankungen ausgleichen. Auf längere Sicht sollte sich eine um 15% (bzw 50%) höhere Ausführungszeit für die Jobs einstellen. Bei den feinkörnigen Tasks ist die Wahrscheinlichkeit hoch, dass alle Jobs eine ähnliche Laufzeit haben, was über 80 bzw. 100 Tasks gemittelt wird. Bei den grobkörnigen Tasks ist es nicht der Fall. Bei nur 10 Tasks ist die Wahrscheinlichkeit sehr hoch, dass es ziemlich große Schwankungen in der Ausführungszeit der Jobs geben wird. Eine einfache Angabe des Makespans ist in diesem Fall nicht sinnvoll. Deswegen wurde in den Ergebnissen auf die Angabe des minimalen und des maximalen Makespans ver-

zichtet. Statt dessen wurde die Standardabweichung berechnet, um zu sehen wie stark die Schwankungen sich auf die Leistung der Algorithmen auswirken. Ähnliches gilt auch für die Leerlaufzeiten. Hier wurde nur die durchschnittliche Zeit angegeben, um zu sehen wie viele Kapazitäten noch frei verbleiben. Die Angabe des minimalen Wertes ist hier nicht interessant, da sich die Transportkosten nicht ändern. Der maximale Wert sagt auch nicht viel aus, da er zum Beispiel von Heuristiken beabsichtigt werden kann.

Die Ergebnisse sind in den Tabellen C.9 bis C.12 zusammengefasst. In jeder Tabelle sind die beiden Experimente im direkten Vergleich zu sehen. Eigentlich waren diese Tests dafür gedacht, den Vorteil der dynamischen Algorithmen gegenüber den Statischen zu zeigen. Allerdings schneiden hier wieder die Heuristiken sehr gut ab. Besondern bei den groben Jobs ist der Makespan und die Standardabweichung sehr niedrig. Dies erklärt sich wieder aus der Vernachlässigung der langsamen Rechner. Durch die Schwankungen der Leistung hat der Rechner ***kiepe*** einen besonders hohen Einfluss auf den Makespan. Und so brauchen die Self Scheduling Algorithmen 100% bzw. 60% länger für die Ausführung. Nur die dynamischen Verfahren, die auf die Replikation setzen können hier mithalten und bei 100%er Leistungsschwankung sogar die Heuristiken überholen. Durch die Replikationen kann fast immer der schnellste Rechner für die Ausführung gefunden werden. Die Schwankungen spielen daher keine große Rolle.

Bei den feinkörnigen Tasklaufzeiten fiel der Unterschied nicht so groß aus. Hier schneiden die besseren Verfahren alle gut ab.

Bei der Durchführung dieser Versuchsreihe hat sich auch die Möglichkeit geboten die Experimente in einer Umgebung mit echten anderen Benutzern durchzuführen. Die beiden schnellen OLC-Rechner wurden zu gleichen Zeit von anderen Nutzern zu 100% ausgelastet, so dass die Ausführungszeit der Tasks auf diesen Rechnern stark schwankte. Da dies die beiden schnellsten Rechner sind, hatten die Schwankungen einen großen Einfluss auf den Makespan. Das Problem war, dass die Berechnung des Benchmarks, die nur sehr kurz andauern, nur zum Teil betroffen war. In folge dessen sahen die beiden Rechner nach außen schneller aus, als sie es tatsächlich waren. Dass hat dazu geführt, dass zu viele Tasks an diese Rechner verschickt wurden. Während bei den Grobkörnigen Jobs keine großen Unterschiede feststellbar waren, waren die Ergebnisse des Experimentes mit den Feinkörnigen Tasks mit identischer Tasklaufzeit interessant. Hier versagten die Heuristiken. Durch die standardmäßige Vernachlässigung der langsamen Rechner, waren wie immer überproportional viele Tasks an die OLC-Rechner zugewiesen. Gegenüber den dynamischen Verfahren lag die Ausführungszeit der Heuristiken um 20% höher. Die Standardabweichung war mit 8 Sekunden auch deutlich höher als die 3 Sekunden bei dem Work Queue Verfahren. Er stellt sich die Frage, warum die Standardabweichung so groß ausfällt. Obwohl die Rechner belastet waren, sollten sie doch für jeden Nutzer eine konstante Leistung liefen, so dass die Makespanwerte sehr ähnlich aussehen sollten. Insgesamt sorgten 3 Benutzer auf einem Rechner für Last. Da aber die Rechner jeweils zwei Prozessoren hatten, musste das Betriebssystem drei Prozesse auf zwei Prozessoren zuordnen, was wohl die Schwankungen in Leistung erklärt. Entweder hat man einen Prozessor für sich allein gekriegt, und hatte somit die volle Leistung, oder musste ihn mit

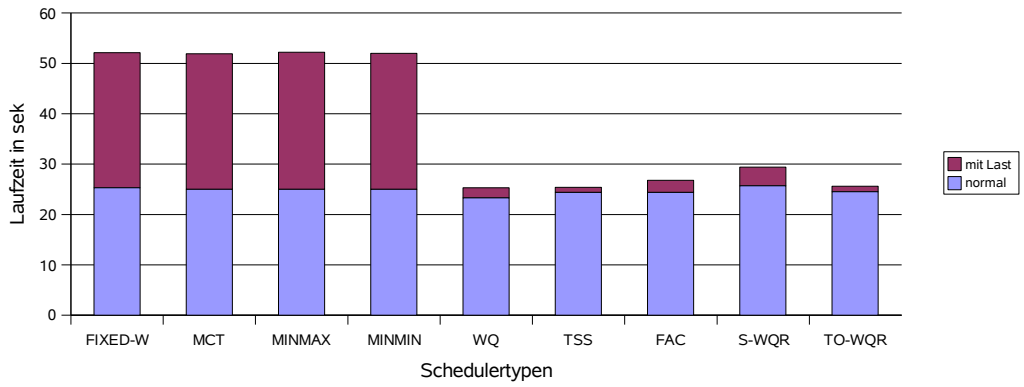



Abbildung 7.3.: Veränderung der Laufzeit der einzelnen Schedulingverfahren bei der Drosselung eines Rechners nach dem Beginn des Experimentes.

einem anderen Nutzer teilen, was einen 50%igen Leistungsabfall mit sich brachte. Durch die statische Natur der Heuristiken konnten diese Schwankungen nicht aufgefangen werden, was die schlechtere Leistung erklärt.

Da die bisher durchgeführten Experimente keine besonderen Nachteile der statischen Algorithmen gezeigt hatten, wurde noch ein Test extra für diesen Zweck durchgeführt. Er fand in der normalen heterogenen Umgebung statt. Allerdings wurde die Leistung des ***ls8olc00*** Rechnern direkt nach dem start des Experiments auf die Hälfte reduziert. So gehen die statischen Algorithmen von den falschen Werten für die Leistung dieses Rechnern. Die Ergebnisse können in den Tabellen C.41 bis C.44 angeschaut werden. Die statischen Algorithmen schneiden hier deutlich schlechter ab, als die dynamischen. Besonders stark in der Unterschied bei dem Experiment mit dem 80 identischen Tasks zu sehen. Der Vergleich ist in der Abbildung 7.3 dargestellt. Die rote Balken stellen die normale Laufzeit dar. Die blauen Balken zeigen die Erhöhung der Laufzeit bei der Verlangsamung des Rechners. Damit ist klar, dass die Angabe der korrekten Leistung der Rechner einen sehr wichtigen Faktor für die statischen Algorithmen darstellt.

### 7.4.4. Instabile Umgebung

Mit einer letzten Versuchsreihe sollte überprüft werden, wie sich die Scheduler in einer instabilen Umgebung verhalten. Bei der Ausführung dieses Experiments gab es einige Schwierigkeiten, so dass man die Ergebnisse nicht einfach zusammenfassen kann.
Da man mit Wahrscheinlichkeiten für den Ausfall gearbeitet hat, konnte man die tatsächliche Anzahl der Ausfälle nicht kontrollieren. Zwar war die Wahrscheinlichkeit sehr hoch, dass während eines Versuchs mindesten ein Rechner ausfällt, es gab aber keine Garantie für die Mindest- und Höchstahnzahl an Ausfällen. Und so gab es Experimente, bei denen

kein Rechner ausfallen ist, bei anderen aber fielen mehrere Rechner aus, manche sogar mehrmals, was einen verheerenden Einfluss auf die Laufzeit gehabt hat. Insgesamt kann man die Art der Ausfälle in vier Kategorien einteilen.
Sehr einfach ist ein Ausfall zu handhaben, wenn er am Ende des Experimentes stattfindet. Falls der Rechner bereits im Leerlauf war, hat dieser Ausfall überhaupt keine Auswirkungen. Die Arbeit kann wie gewohnt beendet werden. Nur die Algorithmen, die mit Replikationen arbeiten, müssen diesen Ausfall besonders behandeln und möglicherweise kompensieren.
Solle der Ausfall während der Ausführung eines Tasks passieren, hat der Schedulingalgorithmus einen großen Einfluss auf die Auswirkungen. Bei den Algorithmen, die mehr als einen Task an die Worker am Stück verschicken ist der Zeitpunkt des Ausfalls wichtig. Je früher der Ausfall passiert, desto mehr Tasks müssen neu zugeordnet werden, desto größer ist der Transportoverhead. Dabei wirkt sich jede neue Zuordnung immer stärker aus. Durch den Ausfall sinkt die Gesamtrechenleistung. Dadurch erhöht sich der Makespan und die Anzahl der Tasks pro Rechner. Durch den erhöhte Laufzeit ist die Wahrscheinlichkeit für einen zweiten Ausfall höher. Es müssen auch mehr Tasks neu verteilt werden.
Der Rechner kann auch schon vor Beginn des Experiments ausfallen. Dann steht er er zur Beginn des Experiments nicht zur Verfügung. Sollte der Fehler während des Experiments wieder behoben werden, und dieser Rechner geht wieder online haben die dynamischen Scheduler einen klaren Vorteil. Da sie nicht alle Tasks sofort zuordnen in die Wahrscheinlichkeit hoch, dass es noch eine Menge nicht verteilter Tasks gibt. Diese können nun an den neuen Rechner versendet werden. Die statischen Scheduler haben diese Möglichkeit nicht. Da sie alle Tasks zu Beginn der Laufzeit zuordnen ist die einzige Möglichkeit neue freie Tasks zu erhalten der Ausfall eines Rechners. Sie können also von einem freien Rechner nur im Fall eines neuen Ausfalls profitieren.
Durch die feinkörnigen Tasks in dem Experiment hat der Ausfall eines Rechners bei dynamischen Verfahren kaum Auswirkungen auf den Makespan. Der Transferoverhead ist in der Regel ziemlich gering und nach der Reparatur wird der Rechner sofort wieder mit Tasks versorgt. Interessanterweise kann ein Ausfall den Makespan sogar verbessern. Dies passier zum Ende des Experiments, falls ein langsamer Rechner am Ende den Makespan bestimmt. Durch seinen Ausfall müssen die Tasks an andere, meist schnellere Rechner verteilt werden, was die Ausführung des Gesamtjobs sogar beschleunigt. Aber auch dann kommt ein statisches Schedulingverfahren, nicht an die Werte der dynamischen Algorithmen heran.

## 7.5. Zweite Versuchsserie - mehrere Master

Es kommt nicht selten vor, dass in einer Umgebung gleich mehrere Nutzer die entsprechenden Rechenkapazitäten nutzen wollen. In diesem Fall konkurrieren mehrere Master um die gleichen Worker. In kleinen geschlossenen Umgebungen ist es möglich eine zentrale Stelle zu schaffen, die sich um das Scheduling kümmert. Bei größeren Umgebungen und vor allem in Grids ist dies nicht mehr möglich. In diesem Fall lässt jeder Master einen

eigenen lokalen Scheduler laufen, der keine vollständige Information über die Umgebung und die Aktivitäten anderer Master hat. In diesem Fall wird das Scheduling zu einem zweistufigen Problem. Zuerst muss der Master die Tasks an die Worker zuweisen. Danach muss der Worker entscheiden, von welchem Master er die Tasks zuerst bearbeitet.
Dabei kann der Worker auf zwei verschiedene Weisen die ankommenden Task verarbeiten. Die einfache Methode ist es alle, Tasks in eine gemeinsame Schlange zu packen. Dann wird jeweils der aktuelle Task von dem Kopf der Schlange genommen und ausgeführt. Dies entspricht dem sogenannten FIFO (First In First Out) -Prinzip. Ein Task, egal aus welcher Quelle, der früher angekommen ist, wird auch früher verarbeitet.
Die zweite Variante ist es, für jeden Master eine eigene Schlange zu verwalten. Dann wird jeweils von dem Kopf jeder Schlange ein Task genommen und ausgeführt. Diese Prinzip wird das Round Robin Prinzip genannt und stellt sicher, dass in einem Zeitraum die gleiche Taskanzahl jedes Masters ausgeführt wird. Allerdings erfordert diese Verfahren einen größeren Verwaltungsoverhead, da es jetzt mehrere Schlangen gibt, die auch dynamisch erstellt und aufgelöst werden müssen.

Für die zweite Testreihe wurden drei Maschinen als Master bestimmt. Auf diesen Rechnern wurden dann parallel zu gleichen Zeit die Jobs gestartet. Jedes Experiment wurde auf jeder Master 8 mal ausgeführt, was einer Gesamtanzahl von 24 Versuchen pro Experiment entspricht. Da die Master nie alle gleichzeitig mit ihren Experimenten anfangen, mussten die ersten und letzten Versuche gefiltert werden, da sie eine erheblich kürzere Laufzeit aufweisen. Es bleiben aber noch ca. 20 Versuche übrig, so dass die Anzahl der Durchführungen mit der ersten Testreihe identisch ist. Jeder Master hat eine Pause von 8 Sekunden zwischen den Experimenten, so dass es bei einer Mindestlaufzeit von 20 Sekunden pro Job immer genug Arbeit für die Worker gab.
Es wurde auch angenommen, dass alle Master und alle Worker im Netz den gleichen Schedulingalgorithmus benutzen. Allerdings konnten die Master zur gleichen Zeit verschiedene Experimente durchführen, so dass die Anzahl der zu verteilenden Tasks für jeden Master unterschiedlich war.
Es ergaben sich einige Änderungen bei der Auswertung der Ergebnisse. Die Angabe von den drei Werten für den Leerlauf ist nicht mehr sinnvoll. Durch mehrere Master sollte der Worker immer beschäftigt sein, sogar wenn ein oder mehrere Master sich gerade in einer Kommunikationsphase befinden. Darum wurde der Leerlaufwert für die Kommunikation weggenommen. Es wurde nur noch der Wert für den durchschnittlichen aller Worker Leerlauf während der Ausführung übernommen. Die Größe dieses Leerlaufs ist ein Indikator für die Lastverteilung. Ist die Verteilung der Tasks sehr unausgewogen, hat ein Rechner irgendwann alle Tasks der drei Master abgearbeitet und befindet sich im Leerlauf, obwohl alle Master noch freie Tasks zur Verfügung haben.

### 7.5.1. Homogene Umgebung

Als erstes wurde wie immer der Test in einer homogenen Umgebung durchgeführt. Da es in dieser Umgebung drei Master und drei Worker gibt, müsste die durchschnittliche

Laufzeit eines Jobs der Laufzeit auf einer Maschine entsprechen.
Die Ergebnisse unterscheiden sich ziemlich stark zwischen der FIFO und der Round Robin Strategie. Die FIFO Strategie begünstigt die Scheduler mit möglichst wenigen Schedulingschritten. Bei den Jobs mit feinkörnigen Tasks sollten vor allem die statischen Algorithmen davon profitieren. Auf der anderen Seite ist der Work Queue Algorithmus, der immer nur einen Task zuordnet, sehr benachteiligt. Bei dem Job mit 100 Tasks braucht der Work Queue Algorithmus im Schnitt 237 Sekunden pro Experiment. Das ist 35% mehr als die Ausführung lokal auf einem der Solaris Rechner. Etwas besser schneiden die dynamischen Verfahren ab, die sofort einen größeren Taskblock zuordnen. Das vermindert die Anzahl der Schedulingschritte und sorgt so für einen höheren Anteil der Last auf dem Worker im Vergleich zu anderen Mastern.
Allgemein fällt bei der FIFO-Strategie eine sehr starke Schwankung der einzelnen Ergebnisse auf, was sich über alle Algorithmen hinzieht und in der Umgebung mit nur einem Master nicht vorhanden war. Da der Leerlauf der einzelnen Worker dabei fast auf dem Nullpunkt liegt, sind die Rechner gleichmäßig ausgelastet, und die Laufzeitschwankungen kommen tatsächlich von den unterschiedlichen Eigenschaften der einzelnen Jobs.
Die Round Robin Strategie ist vor allem bei den Jobs mit möglichst wenig Tasks erfolgreich. Wenn zum Beispiel ein Job mit 10 und ein Job mit 100 Tasks parallel laufen, wird mit der Fertigstellung des ersten Jobs gerade mal 10 Tasks des zweiten Jobs fertig, was gerade mal 10% der Gesamttaskanzahl entspricht. Es ist also weniger wichtig, welcher Scheduler gerade im Einsatz ist. Es kommt vor allem auf die Eigenschaften der Jobs der anderen Master an. Es macht keinen Unterschied, wie viele Tasks zu gleichen Zeit auf dem Worker in der Warteschlange sind, die Wartezeit für jeden Tasks bleibt gleich. Auffällig ist das schlechte Abschneiden der Heuristiken bei den grobkörnigen Jobs. Allerdings muss man sich die Ergebnisse aller 4 Experimente anschauen, da sie parallel ausgeführt wurden. Durch das bessere Abschneiden bei den feinkörnigen Jobs, müssen die grobkörnigen schlechter ausfallen, damit die Gesamtlaufzeit gleich bleibt. Allerdings kann das nicht die einzige Erklärung sein. Denn die Heuristiken weisen auch einen erhöhen Leerlauf auf. Diese kann nur entstehen, wenn ein Rechner alle Tasks aller Master abgearbeitet hat, aber keiner dieser Master den Job komplett hat. Diese Situation ist für eine homogene Umgebung ungewöhnlich, da die Tasks eher gleichmäßig an die Worker verteilt werden. Die Erklärung hier ist die Inkompatibilität der Angaben zu den Tasklaufzeiten untereinander, da sie von verschiedenen Benutzern stammen können und ganz unterschiedliche Einschätzungen enthalten.

### 7.5.2. Heterogene Umgebung

In der nächsten Versuchsreihe wurde wieder die heterogene Umgebung getestet. Dabei bietet sich wieder das bekannte Bild. Bei den grobkörnigen Jobs dominieren wieder die Heuristiken und die Replikationen. Allerdings steigen auch Leerlaufzeiten der Worker für alle Schedulingalgorithmen an, was für eine ungleichmäßige Verteilung der Tasks spricht. Die Restlichen Algorithmen sind um so mehr von den langsamen Rechnern betroffen. Da jetzt mehrere Master ihre Tasks auf dem Rechner ausführen möchten, erhöht sich auch dem entsprechend die Wartezeit in der Schlange. Damit wird der langsame Rechner zum

Flaschenhals für alle Algorithmen, die ihre Tasks auf ihn ausführen. Die Replikationen können hier gegensteuern und den Makespan bei den grobkörnigen Tasks auf die gleiche Stufe mit den Heuristiken stellen.

### 7.5.3. Externe Last

Bei dieser Testserie wird deutlich, dass die Anzahl der Tasks pro Experiment in einer Multi-User Umgebung einen viel höheren Einfluss auf die Leistung der Algorithmen hat, als die Schwankung der Rechenleistung. Bei Jobs mit feinkörnigen Tasks versagt der $S-WQR$ Algorithmus völlig, der Makespan ist hier um mehr als 50% höher als bei den Heuristiken. Bei den Grobkörnigen Tasks liefert er aber die besten Ergebnisse, und kann sogar die Heuristiken übertreffen.

### 7.5.4. Instabile Umgebung

Wenn sich die Anzahl der zur Verfügung stehenden Rechner ändern kann, wirken viele Faktoren auf die Gesamtlaufzeit der Jobs ein. Natürlich ist die Rechenzeit von dem aktuell ausgeführten Task auf dem ausgefallenen Rechner verloren. Da jetzt aber 3 Master im Netz sind, werden die Tasks der beiden anderen nicht ausgeführt und stehen nur in der Warteschlange. Die insgesamt verlorene Rechnerzeit ist im vergleich zu einer Umgebung mit nur einem Master viel geringer. Die Ausfalldauer von 10 Sekunden wurde von der ersten Testrehe übernommen. Allerdings verdreifachen sich die Tasklaufzeiten bei den jetzigen Versuchen. So spielt auch die aktuelle Ausfalldauer keine große Rolle für die Gesamtausführung. Dafür ist die aber die Ausfallwahrscheinlichkeit für einen Rechner während eines Experiments wegen der höheren Joblaufzeiten auch höher. Im Durchschnitt fielen zum Beispiel bei dem $FIXED$ Scheduler 6 Rechner bei dem feinkörnigen Experiment mit identischen Tasklaufzeiten aus. Aber die durchschnittliche Laufzeit pro Job sank von 170 auf 122 Sekunden ab. Dieser Algorithmus ist der große Nutznießer von Ausfällen, da sich dadurch die Chance bietet, die Tasks auf die schnellen Rechner neu zu verteilen. Das gleiche gilt auch für den $RANDOM$ Scheduler. Die durchschnittliche Laufzeit der restlichen Algorithmen erhöht sich um gerade mal 2 bis 5 Sekunden.

## 7.6. Weitere Testreihen

Neben der normalen Tests wurde noch eine Reihe weiterer Tests durchgeführt, die zum Testen der Umgebung dienten oder einzelnen Aspekte des Schedulings veränderten.

### 7.6.1. Einfluss der Datenübertragung auf die Rechenleistung

In unserem vereinfachten Modell nehmen wir an, das ein Rechner mehrere Handlungen gleichzeitig ausführen kann, ohne dass diese sich gegenseitig stören. Moderne Betriebssysteme ermöglichen es, mehrere Prozesse parallel auf einer CPU laufen zu lassen. Ein Prozess könnte also gleichzeitig an einem Task rechnen und ein anderer über das Netz weitere Daten empfangen. Da die Datenübertragung nicht Rechenintensiv sein sollte,

sollte der Einfluss auf die Ausführung des Tasks minimal sein. Auch der Einsatz von DMA(Direct Memory Access) begünstigt diese Annahme. Beim DMA werden Teile des Speichers ohne den Umweg über die CPU direkt an die einzelnen Rechnerkomponenten weitergeleitet. Man kann also den Speicherinhalt direkt an die Grafikkarte schicken oder die ankommenden Daten von der Netzwerkkarte direkt in den Hauptspeicher schreiben. Interessanterweise haben Untersuchungen von Kreaseck et al gezeigt, dass die parallele Datenübertragung einen großen Einfluss auf die Rechenleistung eines Rechners haben kann. [17] Bei den Versuchen haben sie den Einfluss der Datenübertragung getestet. Verwendet wurden verschiedene Linux und Solarisrechner. Auch wurden die Testprogramme in Java und C geschrieben, um den Einfluss der Programmiersprache auf den Test zu minimieren. In allen Fällen war ein Einbruch der Leistung festzustellen. Interessant ist es, dass das Senden von Daten einen größeren Einfluss hatte, als das Empfangen. Auch spielte die Anzahl der Sender/Empfänger keine große Rolle. Der ausschlaggebende Faktor war die Übertragungsgeschwindigkeit. In einigen Extremfällen lag der Einbruch bei 50% der Rechenleistung. Das heißt, die Rechenzeit würde sich in diesen Fällen verdoppeln.
Da dieses Ergebnis doch sehr überraschend ist, und nicht im Einklang mit den „allgemeinen“ Informationen steht, habe ich auch eine Reihe von Tests durchgeführt, um die Ergebnisse zu überprüfen. Zum Testen wurden die Rechner ***kiepe*** und ***kieme*** ausgewählt. Beides sind Solarisrechner, doch sie unterscheiden sich sehr stark in ihrer Rechenleistung. Nähere Informationen zu diesen Maschinen können aus der Tabelle 7.1 entnommen werden. Als Job für die Berechnungen wurde eine zehnfache Kreuzvalidierung verwendet und die Dauer der einzelnen Validationsschritte festgehalten. Auf jedem Rechner wurden jeweils zwei Tests durchgeführt, ein mal ohne die Kommunikation, und ein mal mit. Jeder Test wurde 10 mal hintereinander wiederholt und der Mittelwert der Ergebnisse gespeichert.
Die Ergebnisse sind in der Tabelle 7.3 dargestellt. Jeweils in den ersten beiden Spalten ist die Dauer der Berechnung einer Validierung. Dahinter die Zunahme der Rechendauer in Prozent bei der aktiven Datenübertragung. Bei ***kiepe*** ist der Leistungsverlust enorm. Die Aufgabe brauche eine mehr als die doppelte Zeit für die Fertigstellung. Bei ***kieme*** dagegen ist die Steigung der Rechenzeit mit knapp 20% eher gering. Allerdings besitzt ***kieme*** 2 Prozessoren, so dass der zweite Prozess möglicherweise auf der zweiten CPU laufen konnte. Dann ist der Leistungsverlust auf diese Maschine auch enorm. Die Datenübertragung fand während der gesamten Berechnungen statt. Dabei wurden von ***kiepe*** ca. 690 MB und von ***kieme*** (wegen der kürzeren Ausführungsdauer) ca. 190 MB übertragen.
Welchen Einfluss hat diese Feststellung auf die einzelnen Schedulingverfahren? Zum Einen muss man die übertragene Datenmenge im Verhältnis zu der Rechenzeit setzen. Die Datenmenge, auf welcher der Test gearbeitet hat, betrug ungefähr 120 KByte. (80Kbyte nach dem die Daten für die Übertragung komprimiert wurden) Im Hintergrund wurden pro Test bei ***kiepe*** ca. 70 MB übertragen. Die Wahrscheinlichkeiten für ein solches Verhältnis zwischen der Rechenzeit und der Datenmenge ist in einer realen Umgebung sehr gering. Die Testrechner waren in einem „unbelasteten“ LAN. In einer Master/Worker Umgebung findet die Kommunikation gleichzeitig zwischen vielen Rechnern statt, was die übertragende Datenmenge pro Rechner einschränkt. Die einzi-

| Kiepe | | | Kieme | | |
|---|---|---|---|---|---|
| Idle | Transfer | Zunahme | Idle | Transfer | Zunahme |
| 1883 | 3799 | 101.6% | 703 | 841 | 19.5% |
| 1562 | 3664 | 134.6% | 627 | 732 | 16.8% |
| 1681 | 3468 | 106.3% | 599 | 708 | 18.3% |
| 1556 | 3665 | 135.5% | 608 | 719 | 18.2% |
| 1736 | 3805 | 119.2% | 604 | 703 | 16.3% |
| 1772 | 3560 | 103.1% | 615 | 705 | 14.6% |
| 1613 | 3699 | 129.4% | 606 | 696 | 14.9% |
| 1714 | 3642 | 112.5% | 601 | 700 | 16.4% |
| 1738 | 3784 | 118.5% | 594 | 703 | 18.3% |
| 1597 | 3655 | 128.8% | 604 | 711 | 17.7% |
| 16846 | 36780 | 118.0% | 6161 | 7218 | 17.0% |

Tabelle 7.3.: Ergebnisse des Übertragungstests. Die erste Spalte zeigt die Laufzeit ohne Datentransfer, die zweite Spalte mit. In der dritten Spalte ist die prozentualle Zunahme der Laufzeit. Die Zeilen geben die Laufzeit der Validierungen in Sekunden an. In der letzten Zeile sind die Zahlen für das gesamte Experiment.

ge Maschine, die mit voller Bandbreite senden kann, ist der Master, allerdings führt er gleichzeitig keine aufwendigen Berechnungen durch. Damit ist der Einfluss auf die Berechnungen zwar vorhanden, aber relativ gering so dass er normalerweise vernachlässigt werden kann. Es können aber auch Techniken wie Caching der Daten benutzt werden, die zu übertragende Datenmenge zu reduzieren.

### 7.6.2. Caching der Eingabe

Wenn man sich die Kommunikation zwischen dem Master und den Workern anschaut, wird man schnell feststellen, dass die Tasknachrichten einen sehr großen Teil des Traffics ausmachen. Bei jeder Tasknachricht muss die Eingabe, was normalerweise ein ExampleSet ist, übertragen werden.
Allerdings arbeiten viele Experimente auf den gleichen Eingabedaten, was bedeutet, dass die mehrmalige Übertragung des ExampleSets überflüssig ist. Die Operatoren, die auf den gleichen Eingabedaten arbeiten, sind die Kreuzvalidierung, Parameteroptimierung und die IteratingOperatorChain. Bei diesen Operatoren ist es sinnvoll das ExampleSet nur ein mal zu übertragen. In den restlichen Nachrichten wird dann nur eine Referenz darauf angegeben. Der Worker verwaltet einen Zwischenspeicher für die Eingabe und benutzt diese bei der Ausführung von Tasks, anstatt sie immer wieder vom Master zu beziehen. Die implementierte Erweiterung sieht folgendermaßen aus. Der Master ordnet jeder Eingabe eine ID zu und verschickt diese, anstatt den konkreten Daten. Der Worker

überprüft, ob er die erforderlichen Daten bereits gespeichert hat. Falls nicht, werden diese bei dem Master angefordert und die Tasks solange als wartend eingestuft. Kommt die Antwort auf die Ressourcenanfrage, werden diese Daten in den Cache eingefügt und an jeden wartenden Task weitergeleitet. Natürlich bringt dieses Verfahren am Anfang eine Wartezeit mit sich, da bei jeder neuen Eingabe der Master kontaktiert werden muss. Es ergeben sich aber 3 Vorteile. Der offensichtliche Vorteil ist die Reduzierung des Netzwerktraffics. Da eine ResourceRequest-Nachricht sehr klein ist und die Eingabe nur ein mal übertragen wird, senkt es das Übertragungsvolumen bei den feinkörnigen Jobs auf rund 10% der Ursprunggröße. Bei grobkörnigen Jobs, wie die Kreuzvalidierung, kann immerhin eine Senkung auf 50% erreicht werden, wenn jeder Worker mindestens zwei Tasks zugeordnet bekommt. Der zweite Vorteil ist der geringere Speicherverbrauch bei den Workern. Die gleiche Eingabe ist insgesamt nur ein mal im Speicher vorhanden, und wird von allen Tasks verwendet. Der dritte Vorteil ist die Deserialisierung. Die Nachrichten sind für die Übertragung komprimiert worden. Also müssen sie erst entpackt werden. Dann muss mit X-Stream aus dem XML wieder ein Nachrichtenobjekt erstellt werden, was bei größeren Eingabedaten eine längere Zeit (mehrere Sekunden) beanspruchen kann. In dieser Zeit muss der Rechner die Taskausführung verlangsamen, was eine negativen Einfluss auf die Tasklaufzeit hat und zur Erhöhung des Makespan führen kann.

Um den genauen Einfluss des Cachings auf die Ausführungszeit zu erfahren wurde wieder eine Reihe von Experimenten durchgeführt.

Die GridParameterOptimization war ein Experiment, was eine sehr kurze Ausführungszeit im Verhältnis zu der Eingabedaten hatte. Hier wurde wieder der Effizientstest durchgeführt.

| Feinkörnig, unbekannte Laufzeit mit Caching | | | | |
|---|---|---|---|---|
| Worker | 1 | 2 | 3 | 4 |
| Makespan | 175.1 | 96.0 | 66.8 | 51.7 |
| Speedup | — | 1.82 | 2.62 | 3.39 |
| Effezienz | — | 0.91 | 0.87 | 0.85 |
| Zunahme | — | 0.3 | 0.03 | 0.04 |

Zwar ist die Effizienzsteigerung 3 bis 4 Prozentpunkten gering, sie ist dennoch sichtbar. Das Caching ist hier eindeutig von Vorteil. Daneben wurden noch die Versuche in einer heterogenen Umgebung mit einem und 3 Mastern durchgeführt. Leider zeigten sich hier keine positiven Ergebnisse. Die Unterschiede zu der normalen Ausführung lagen oft in dem Bereich einiger Promille. Bei dem Experiment, der für jeden Task eine eigene Eingabe hat, wurde sogar eine leichte Verschlechterung der Laufzeit festgestellt, da hier kein Caching stattfinden kann und für jeden Task die Eingaben einzeln übertragen werden müssen.

Es bleibt also nur ein sicheres Vorteil des Caching in manchen Situationen, und das ist der geringere Speicherverbrauch auf dem Worker.

### 7.6.3. CaseBase Approach

Vor allem die Heuristiken brauchen eine korrekte Angabe der Tasklaufzeiten. Auch andere Verfahren können so angepasst werden, dass sie die Laufzeiten in den Berechnungen verwenden. Da diese Angaben, falls sie von dem Nutzer stammen, mit einer sehr hohen Wahrscheinlichkeit falsch sind, und sich nicht vor der Taskausführung berechnen lassen, wurde der Case-Base Ansatz versucht. [1] Die Idee besteht darin, Informationen während der Ausführung zu speichern, diese in einer Datenbank abzulegen und für spätere Ausführungen zu verwenden. Dazu wurde eine Erweiterung für das Plugin geschrieben, die Taskinformation zusammen mit der tatsächlichen Laufzeit speichert. Zu den Informationen gehören die Größe des ExampleSets für den Input (Anzahl der Reihen und Spalten), die Anzahl der Operatoren, die ausgeführt werden müssen, dazu der Name des Workers, der diesen Task ausführen sollte, und die Leistungsfähigkeit des Rechners, der für die Ausführung benutzt wurden. Es wurde gehofft, dass durch diese Informationen ein Task genau beschrieben werden konnte und mit ähnlichen Tasks eine ähnliche Laufzeit aufweist.
Leider führte diese Vorgehensweise nicht zum Erfolg. Wie man bei der Vorstellung der einzigen Experimente sehen konnte, variiert sogar die Laufzeit von identischen Tasks auf der gleichen Maschine. Extrem sind die Schwankungen vor allem bei der Parameter Optimierung. Das verändern eines Parameters führte zu sehr starken Veränderungen in der Laufzeit. Das Abspeichern aller Parameter für alle möglichen inneren Operatoren in der Datenbank ist sehr aufwendig und nicht praktikabel. Auch kann es zu extremen Schwankungen in der Laufzeit bei identischen Tasks kommen, die zwar die gleiche Eingabegröße haben, die Werte dieser Eingabe(zum Beispiel ein ExampleSet mit vielen Nullen) sich unterscheiden. Sehr gut konnte man es bei der Kreuzvalidierung beobachten, wo der letzte Validierungsschritt doppelt so lange braucht, als der Rest.
Nach diesen Fehlversuchen wurde die Idee, durch den Case-Base Einsatz die Performanz zu steigen, aufgegeben. Es ist nach wie vor ein ungelöstes Problem, die Laufzeit der Tasks genau vorherzusagen, ohne vorher das Experiment ausgeführt zu haben.

### 7.6.4. Zentraler Schedulingserver

Der bisherige Schedulingansatz hat jedem Master die volle Freiheit bei der Zuordnung der Tasks an die einzelnen Worker gelassen. Dieser Ansatz wiederspricht der klassischen Methode den kompletten Schedulingvorgang einer zentralen Instanz zu überlassen.[28] Bei dieser Methode sammelt eine zentrale Stelle alle Tasks ein und sorgt für die Verteilung. Der Unterschied in den beiden Ansätzen ist die unterschiedliche Sicht auf die Worker im Netz. Während ein Master nur sich und die eigenen Tasks wahrnimmt, kann eine zentrale Stelle die Tasks aller Master sehen und diese entsprechend verteilen. Um diese beiden Ansätze vergleichen zu können, wurde ein zentraler Scheduler implementiert. Die Aufgabe der einzelnen Master beschränkt sich bei dieser Variante nur auf das Versenden aller eigenen Tasks an die zentrale Instanz und das Empfangen der Ergebnisse. Der eigentliche Schedulingalgorithmus wird auf dem zentralem Server ausgeführt. Die normalen Schedulingverfahren können in dieser Umgebung nicht benutzt werden, da sie

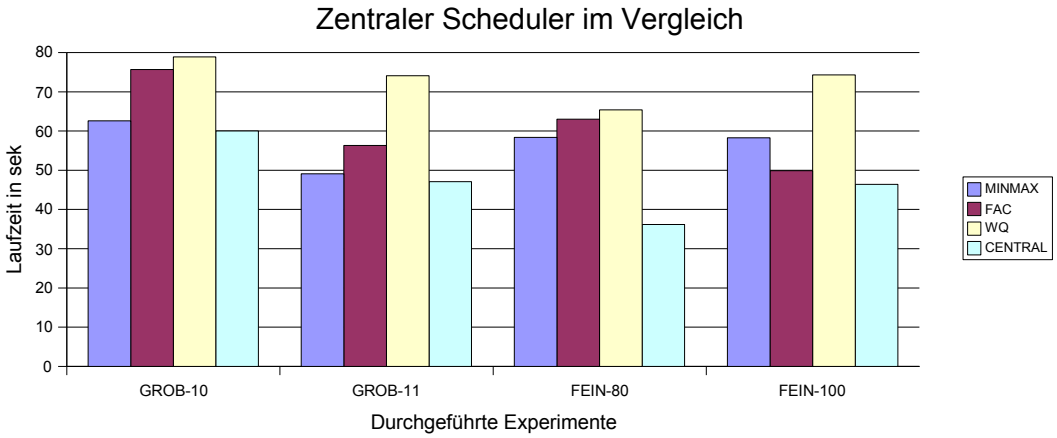



Abbildung 7.4.: Zentraler Scheduler im Vergleich zu anderen Algorithmen

für das Schedulingproblem entworfen wurden, bei dem für alle Tasks $n_j$ gilt: $r_j = 0$. Sie setzen voraus, dass alle Tasks zum Beginn der Ausführung zur Verfügung stehen. In der neuen Umgebung kommen die Tasks aber dynamisch nach dem Start der Ausführung bei dem Scheduler an. So ist für manche Tasks das $r_j$ größer 0. Für die Ausführung braucht man also ein Verfahren, dass die Anzahl der zu verteilenden Tasks nicht benötigt. Laut Tabelle 5.2 kommen die Algorithmen $RANDOM$, $WQ$ und $S - WQR$ in Frage. Nach bisherigen Versuchen ist $RANDOM$ unbrauchbar. Eine Replikation braucht man auch nicht anzuwenden, da die freien Maschinen sofort für die Bearbeitung der Tasks anderer Master verwendet werden können. Es bleibt also nur noch Self Scheduling übrig. Dieses Verfahren ist zwar sehr einfach, ist aber sehr gut geeignet um den Makespan zu minimieren. Die kompletten Ergebnisse der Tests können in den Tabellen C.21 bis C.24 angeschaut werden. Die Abbildung 7.4 zeigt außerdem den Vergleich zu 4 anderen ausgewählten Schedulern, die in ihrer Kategorie am besten abschneiden. Wie man sieht ist die Ausführungszeit niedriger, als bei dem Rest. Bei den grobkörnigen Jobs ist der zentrale Scheduler den Heuristiken ebenbürtig. Bei den Feinkörnigen klar überlegen. Dies lässt sich sehr leicht erklären. In der verteilten Variante sieht jeder Master nur seine eigenen Tasks. Also werden auch die Rechner, die gerade Aufgaben von den anderen Mastern ausführen als frei betrachtet und die lokalen Aufgaben dahin geschickt. In diesem Fall werden vor allem an den langsamen Rechner ***kiepe*** mehr Aufgaben zugeordnet, als für das normale Scheduling nötig. Bei der zentralen Variante wird der Rechner nur benutzt, wenn er wirklich frei ist. Die Tasks der anderen Master werden dann entsprechend auf die anderen Worker verteilt. Es entstehen also keine Leerlaufzeiten auf den einzelnen Workern, solange noch Arbeit verfügbar ist. Jeder Worker wird vollständig ausgelastet. Das entspricht dem Vorgehen der Replikation, allerdings wird hier keine Leistung verschwendet. So kann man von einem optimalen Scheduling sprechen. Auch in den Umgebungen mit externer Last und Workerausfall leistet der Scheduler gute Arbeit. Durch das dynamische Scheduling sind alle Rechner ausgelastet. Es kommt also zu

keinem Ungleichgewicht. Auch die Auswirkungen des Ausfalls sind sehr gering. Es ist immer nur ein Task betroffen. Er wird wieder vorne in die Schlange der offenen Tasks eingefügt und sofort an den nächsten freien Rechner gesendet. So kann die Verzögerung minimal gehalten werden.

Doch der zentrale Scheduler hat auch Nachteile. Die Tasks gehen nicht direkt zu dem Worker, sondern müssen zuerst zu dem Scheduler übertragen werden. Wenn dabei auch die Eingabedaten übertragen werden, verdoppelt sich effektiv die Netzauslastung. Eine Lösung ist es, nur die Metadaten über dem Task zu übertragen und vor allem die Eingabe lokal zu lassen. Wird der Task zugeordnet, so sendet der Scheduler die Adresse der Workers an den Master. Der Master überträgt den Task direkt an den Worker und kriegt von diesem auch sofort das Ergebnis. Danach muss der Worker den Scheduler noch benachrichtigen, dass er wieder frei ist. Die Lösung erfordert aber einen komplizierten Kommunikationsprotokoll. Es müssen neue Nachrichtentypen und TaskObjekte definiert werden. Außerdem muss man dann zwischen den Verbindungen zu dem Scheduler, dem Worker und dem Master unterscheiden. Da dieser Ansatz große Änderungen in der internen Struktur des Plugins benötigen würde, wurde er verworfen. Alle Tasks gehen zu dem Scheduler und werden von dieser verteilt. Er empfängt auch die Ergebnisnachrichten und schickt sie weiter an die Master. Um den Netzwerktraffic zu minimieren, wurde auf das Caching zurückgegriffen. Der Scheduler holt die Eingabedaten von den Mastern und speichert zwischen. Wenn der Worker die Eingabedaten benötigt, werden sie von dem Scheduler geliefert. Durch dieses Konzept wird der Netzwerktraffic reduziert und die eine komplette Trennung zwischen dem Master und den Workern erreicht. Diese Lösung kann dann auch bei Systemen mit einer Firewall eingesetzt werden.

Natürlich wäre es schön die Vorteile des zentrales Schedulers auch bei der Standardkonfiguration zu haben. Um dies zu erreichen, muss man verlässlich wissen ob der Worker, zu dem man die Tasks schicken möchte, auch wirklich frei ist. Natürlich kann der Master eine entsprechende Anfrage senden. Doch wie oft soll es passieren? Jede Nachricht erhöht den Transferoverhead. Was passiert, wenn der Worker beschäftigt ist? Die Anfrage sollte nach einer Zeitspanne wiederholt werden, doch dies ist nicht praktikabel. Der Worker könnte sofort nach einer Anfrage frei werden. Die Rechenleistung bis zur nächsten Anfrage wäre dann verschenkt. Die Benachrichtigung könnte auch workerseitig angestoßen werden. Wenn der Worker fertig ist, wird eine entsprechende Nachricht an alle Master versendet. Allerdings führt dieses Vorgehen nicht zum Ziel. Jeder Master würde einen neuen Task versenden und es entsteht die gleiche Situation, wie bei der verteilten Variante. Der Worker könnte auch die Benachrichtigung nur an einen Master verschicken. Die Master würden dafür nach und nach aus der Liste der verbundenen Rechner ausgewählt. Das entspricht ungefähr dem Round Robin verhalten. Hier kann es aber passieren, dass der Master zur Zeit keinen Task zum Scheduling bereit hat und nur noch auf das Ende der Ausführung wartet. In diesem Fall müsste der Master eine Benachrichtigung an den Worker senden, und dieser müsste den nächsten Master für die Ausführung auswählen. Während dieser Kommunikation bleibt der Worker aber ohne Arbeit. All diese Ansätze scheinen unpraktikabel. Die einzige Abhilfe scheint die Replikation zu sein oder eine sorgfältige Auswahl der Hardware für die Rechentasks.

## 7.7. Fazit der Evaluierung

Nach der Durchführung der Experimente kann man die Algorithmen in mehrere Gruppen einordnen. Die Einordnung gleicht der Reihenfolge, in der die Algorithmen vorgestellt wurden. Die Unterschiede innerhalb dieser Gruppen sind eher minimal und können vernachlässigt werden. Die Unterschiede zwischen den Gruppen sind schon bedeutender und sollten bei der Auswahl eines geeignetes Schedulingalgorithmus berücksichtigt werden. Außerdem spielt die Anzahl der gleichzeitigen Master im Netz eine sehr große Rolle. Bei mehreren Mastern hängt die Geschwindigkeit der Jobausführung vor allem von den Eigenschaften der Tasks ab.

Die erste Gruppe ist sind die einfachen statischen Algorithmen und die zufällige Verteilung. Diese Algorithmen haben sich als ungeeignet für die meisten Umgebungen erwiesen. Sogar die Varianten, die mit Gewichten arbeiten sind den Heuristiken unterlegen.

Die zweite Gruppe sind die Heuristiken. Obwohl sie zu den statischen Algorithmen gehören, schneiden sie in den meisten Umgebungen sehr gut ab. Allerdings ist das gute Abschneiden eher auf die Vernachlässigung der langsamen Rechner zurückzuführen. Die Tasklaufzeit spielt dann nur noch eine untergeordnete Rolle. Durch die fehlende korrekte Angaben über die Tasklaufzeiten sind auch die Unterschiede zwischen den drei Heuristiken nicht mehr gegeben, da sie nur in der Vorsortierung der Tasks bestehen.

Die dynamischen Algorithmen bilden die dritte Gruppe. Sie liefern gute Ergebnisse vor allem bei feinkörniger Taskzerlegung. Hier können die Verfahren im nachhinein noch steuernd eingreifen. Auch die variierende Rechnerleistung stellt kein großes Problem dar und kann am Ende kompensiert werden.

Die letzte Gruppe bilden die Verfahren, die auf der Replikation basieren. Sie können in jeder Situation eingesetzt werden und liefern stets gute Ergebnisse. Dabei wird die Vernachlässigung der langsamen Rechner der Heuristiken mit der Möglichkeit auf die Änderungen der Umgehung der dynamischen Verfahren kombiniert. Auch grobkörnige Jobs stellen kein Problem dar. Die Leistung wird aber durch die hohe Verschwendung von Ressourcen erkauft. Die überflüssig ausgeführten Berechnungen können dabei die Rechenanforderung der Tasks übersteigen.

Bei mehreren Mastern im System, spielt die Anzahl der Task pro Job und das Verhalten des Workers die wichtigste Rolle. Haben alle Jobs eine ähnliche Anzahl der Tasks, können die Tasks gleichberechtigt ausgeführt werden. Wenn die Anzahl dagegen stark schwankt, bevorzugt die FIFO-Strategie die Verfahren mit möglichst wenigen Schedulingschritten, das Round Robin Prinzip die Jobs mit möglichst kleiner Taskanzahl.
Insgesamt kann man kein Verfahren als einen Sieger küren. Die meisten haben mehrere Gebiete, in denen sie versagen. Der passende Scheduler muss daher von Umgebung zu Umgebung und von Job zu Job neu ausgewählt werden.

# 8. Zusammenfassung und Ausblick

Das Ziel dieser Diplomarbeit war es, verschiedene Ansätze zum Verteilen der Data Mining Aufgaben zu untersuchen. Um dieses Ziel zu erreichen, mussten verschiedene Forschungsgebiete miteinander kombiniert werden. Als Grundlage dienten die verteilten Systeme. Man musste zuerst eine verteilte Umgebung schaffen, auf der man aufbauen konnte. Der zweite Bereich war das Data Mining. Es mussten verschiedene Verfahren angeschaut und passende für die Verteilung ausgewählt werden. Zur Hilfe wurde das Data Mining Tool RapidMiner genommen, dass zahlreiche Data Maning Verfahren bietet und sich leicht erweitern lässt. Der wichtigste Teil war schließlich das Scheduling, das einen großen Einfluss auf die Effizienz der parallelen Ausführung hat.

Um die Tests durchführen zu können, musste zuerst ein Unterbau geschaffen werden. So wurde auf der Grundlage des Master/Worker Paradigmas ein verteiltes Systems implementiert. Die Masterseite wurde dabei durch ein Plugin für den RapidMiner realisiert. Außerdem wurde auch workerseitig eine entsprechende Software implementiert, die er ermögliche Teilaufgaben des Data Mining Vorgangs auf diesen Rechnern auszuführen.

Nachdem die verteilte Umgebung geschaffen worden war, mussten die Data Mining Verfahren auf ihre Parallelisierbarkeit untersucht werden. Nach der Identifizierung geeigneter Verfahren wurden diese parallelisiert und in das Distributed RapidMiner Plugin eingegliedert.

Danach kam der Kern der Arbeit. Es mussten verschiedene Schedulingverfahren angeschaut und für die Implementierung ausgewählt werden. Für die Tests wurden insgesamt 13 Verfahren ausgewählt. Neben den einfachen statischen Verfahren und Heuristiken, kamen auch dynamische Lösungen mit und ohne Replikation zum Einsatz. Auch wurde der zentrale Scheduler eingesetzt, um seine Leistung gegenüber dem verteilten Scheduling zu testen.

Nach der Implementierung begannen die Tests. Er wurde eine Reihe von Experimenten ausgewählt und in verschiedenen Rechnerumgebungen getestet. Die erste Testreihe wurde mit nur einem Master im System, die zweite mit drei gleichzeitigen Mastern durchgeführt. In der ersten Testreihe war es nicht schwierig klare Ergebnisse zu bekommen und diese auf die Eigenschaften des Schedulers zurückzuführen. Dabei konnte keine klare Überlegenheit eines Verfahrens festgestellt werden. Es gab immer Situationen, in denen einzelne Verfahren gegenüber anderen unterlagen.

In den Tests mit mehreren Mastern war die Sache nicht so offensichtlich. Die Eigenschaften der Scheduler traten in den Hintergrund. Der Erfolg der parallelen Ausführung hing jetzt viel stärker von der Beschaffenheit der Data Mining Aufgaben und den Aktivitäten der anderen Nutzer im Netz ab. Das größte Problem war die fehlende Information der einzelnen Master über ihre Umgebung, so dass die Schedulingentscheidungen nur auf Grund von lokalen Informationen getroffen werden mussten. Eine Abhilfe schaffte hier der Einsatz eines zentralen Schedulers. Dadurch konnte die Leistung in manchen Fällen enorm erhöht werden. Doch auch bei diesem Verfahren zeigten sich einige Nachteile, wie die stärkere Netzwerkbelastung und ein hoher Speicherverbrauch.

Insgesamt wurden die im Vorfeld dieser Arbeit gestellten Ziele erreicht. Das Plugin für den RapidMiner wurde fertiggestellt, auch die Schedulingverfahren beschrieben und getestet. Allerdings fielen die Testergebnisse nicht immer entsprechend den Erwartungen aus und boten einige Überraschungen.

### Ausblick

Viele der RapidMiner Operatoren haben weitere innere Operatoren, die meistens in einer Schleife ausgeführt werden. Die Parallelisierung solcher geschachtelter Operatoren ist zu dem jetzigen Moment nicht möglich. Dies würde eine neue Vorgehensweite beim Scheduling anfordern. Es würde noch ein zusätzlicher Schritt hinzukommen: das Auswählen von geeigneten Rechnern für den Task. Dieser Rechner würde dann auch das Scheduling der inneren Operatoren übernehmen. So entsteht ein zweistufiges System und eine mehrstufige Hierarchie. Eine Möglichkeit wäre es einen „SuperMaster“ einzuführen [5], doch wäre es auch interessant zu sehen, wie sich die Scheduler ohne eine zentrale Instanz verhalten und welche Eigenschaften das verteilte System mit sich bringen muss, um erfolgreich zu sein.

# A. Das DRM Plugin

In diesem Anhang wird eine Gebrauchsanleitung für das Distributed RapidMiner Modul gegeben. Weiter hinten wird dann auf die einzelnen Java-Klassen eingegangen, die verändert werden müssen, um dieses Modul zu verändern oder zu erweitern. Dabei wird sowohl auf das Schreiben neuer verteilter Operatoren eingegangen, als auch auf das Schreiben neuer Schedulingverfahren und das ändern des Verhaltens auf der Worker-Seite.

## A.1. Die neuen Operatoren

Damit das Plugin gestartet wird, muss in das Experiment der entsprechende Operator eingebunden werden. An erster Stelle sollte der DistributedScheduler Operator stehen, der das ganze System startet. Danach können andere Operatoren benutzt werden, die auf verteilte Funktionen zugreifen.
Bevor man mit der parallelen Ausführung beginnen kann, müssen zuerst die geeigneten Rechner in der Umgebung gefunden werden. Dafür sorgt der DistributedScheduler Operator. Es sollte so früh wie möglich in einem Experiment gestartet werden und übernimmt komplett die gesamte mit der Verteilung der Tasks benötigte Arbeit. Vor der Verwendung muss dieser Operator entsprechend konfiguriert werden. Die Standardeinstellungen sind für den Betrieb in einem LAN geeignet. Ansonsten stehen folgende Konfigurationsmöglichkeiten zur Auswahl:

**local_worker** Legt die Anzahl der Worker fest, die auf der lokalen Maschine gestartet werden sollen. Da das Scheduling keine große Prozessorlast erzeugt, hat der Rechner, auf dem das Experiment gestartet wird, noch freie Kapazitäten zur Verfügung. Wenn man sie nicht anderweitig nutzen möchte, kann man einen Teil der Arbeit auch lokal ausführen. In dem Fall wird der lokale Arbeiter genau so wie die Externen behandelt. Für den Fall, dass die Maschine mehrere Prozessoren oder Prozessorkerne besitzt, kann auch mehr als ein Worker angegeben werden, damit der Prozess auch lokal parallelisiert wird. Dabei sollte auf die Speicherauslastung geachtet werden, da jeder Worker auf einer Kopie der Eingaben arbeitet, so dass der Speicherverbrauch bei mehreren lokalen Workern schnell explodiert.

**worker_list** Hier kann man eine Datei angeben, in der sich eine Liste mit den bekannten Arbeitern im Netz befindet. Nach dem Start wird der Scheduler versuchen, sich mit jedem auf der Liste stehenden Rechner zu verbinden und sie als Arbeiter für das aktuelle Experiment zu benutzen. Jede Zeile der Datei sollte einen neuen Rechner enthalten. Die Einträge sollten das Format *RechnerName:PortNummer* haben. Die Rechner in dieser Liste sollten frei zugänglich sein. Der Port mit der *PortNummer* muss für die

Verbindungen von außen offen sein. Die *PortNummer* ist dabei von jedem Worker frei konfigurierbar. Der Verbindungsversuch geschieht dabei nur ein mal. Sollte der Rechner während des Experimentes ausfallen, steht er für die restlichen Berechnungen nicht mehr zur Verfügung.

**UDP_Discovery** Wenn diese Option aktiviert ist, startet der Scheduler eine automatische Suche nach anderen Rechnern. Für diese Suche wird die Broadcast Eigenschaft der LAN Netzwerke verwendet. Dadurch ist es möglich eine Nachricht gleichzeitig an alle aktiven Rechner im Netz zu senden. Eine Liste mit den Arbeitern ist dann nicht mehr nötig. Da die UDP Suche aber nur innerhalb einer LAN Umgebung funktioniert, kann die Workerliste trotzdem angegeben werden, um zusätzliche Rechner außerhalb des lokalen Netzwerkes zu ereichen. Auch für diese Art der Rechnerentdeckung gilt: Der Rechner muss frei erreichter sein, und eingehende Verbindungen aus dem Netz akzeptieren. Nach der initialen Suche, wird diese in periodischen Abständen wiederholt, so dass neu hinzugekommene Rechner entdeckt und für die Verteilung benutzt werden können. Man kann also nach dem Starten des Experimentes zusätzliche Rechenleistung zur Verfügung stellen.

**central_server** Hier kann ein Rechner angegeben werden, der als ein zentraler Server fungieren soll. Es gibt zwei Einsatzmöglichkeiten für den Server. Er kann als ein zentraler Scheduler verwendet werden. Dann fungiert er als einziger Worker, und alle Tasks werden an ihn versandt. Für nähere Informationen siehe Kapitel 7.6.4. Die zweite Einsatzmöglichkeit ist die Hilfe bei der Suche nach anderen Arbeitern. Falls ein zentraler Server vorhanden ist, werden sich alle Arbeiter im Netz mit diesem verbinden. Der Master kann, indem er bei dieser Option die Adresse des Servers einträgt, eine Liste aller aktiven Rechner bekommen. Danach geschieht ein Versuch des Verbindungsaufbaus mit jedem Rechner auf dieser Liste. Auch wird zusätzlich Information über neuen Master an alle Worker geschickt, so dass sie ihrerseits auch einen Verbindungsaufbau versuchen. Eine freie Erreichbarkeit der Worker ist hierbei nicht nötig. Nur eine der Seiten (Master oder Worker) sollte von außen erreichbar sein. Für alle Verbindungsversuche gilt aber, wenn beide Seiten keine externen Verbindungen zulassen, kann Master den Rechner nicht als Worker benutzen.

**data_cache** Falls diese Option aktiviert ist, werden die Eingabedaten für die einzelnen Tasks Standardmäßig nicht übertragen. Ein externer Worker muss diese Daten zuerst anfordern. Nach der Anforderung werden sie bei ihm zwischengespeichert. Bei bestimmten Aufgaben kann diese Option den Netzwerktraffic und den Aufwand der Deserialisierung merklich reduzieren. Für nähere Informationen siehe Kapitel 7.6.2

**scheduler_Type** Diese Einstellung legt fest, welcher Scheduler für die aktuelle Aufgabe verwendet werden soll. Im Moment sind 13 verschiedene Schedulingverfahren implementiert. Für die Auswahl an verschiedenen Schedulingalgorithmen siehe Kapitel 5. Die Option hat einen sehr großen Einfluss auf die Gesamtlaufzeit der Aufgabe. Der am besten geeignete Scheduler hängt sehr stark von der Umgebung und der Art der Eingabe ab, und sollte dem entsprechend gewählt werden.

**Die angepassten Operatoren**
Nachdem der Scheduler gestartet wurde, können die anderen verteilten Operatoren normal benutzt werden. Es ist keine weitere Konfiguration mehr nötig. Allerdings muss darauf geachtet werden, dass keine zwei verteilten Operatoren verschachtelt benutzt werden. Dass hätte zur Folge, dass einer von diesen Operatoren an den Worker geschickt wird, die standardmäßig keinen Scheduler einsetzen. Das hätte den Abbruch der Ausführung zur Folge.

## A.2. Aufbau des Plugins

Das Plugin selbst besteht aus drei großen Komponenten. Die Operatoren werden auf dem lokalen Rechner als Teil des Masters eingesetzt. Daneben gibt es noch den Worker und den zentralen Server, die normalerweise auf den externen Rechnern eingesetzt werden. Dabei ist der Zentrale Server optional und kann in geschlossenen Umgebungen weggelassen werden.

### A.2.1. Der Master

Der Master wird gestartet, wenn der DistributedScheduler Operator in RapidMiner ausgeführt wird. Der Master besteht aus mehreren Modulen. Jedes Modul ist dabei als ein Singleton implementiert und arbeitet in seinem eigenen Thread. Dadurch ist es von den anderen Modulen unabhängig. Die Kommunikation zwischen den Modulen geschieht über festgelegte Schnittstellen. So kann die Zeit, die für die Ausführung von bestimmten Aufgaben notwendig ist, dynamisch vergeben werden. Allerdings stellt sich das Problem der Synchronisation der einzelnen Module untereinander. Die einzelnen Module sind:

**Der Scheduler** nimmt die Tasks der Operatoren entgegen und sorgt für ihre Verteilung. Speichert dabei die Information über die Worker.

**Der Discoverer** Sorgt für die Entdeckung der neuen Rechner im Netz. Arbeitet dabei vollkommen unabhängig von den anderen Modulen.

**Der Communicator** Sorgt für den Austausch von Nachrichten zwischen den einzelnen Rechnern. Leitet diese an den Message Processor weiter. überwacht den Verbindungsstatisch zu den Workern und unterrichtet den Scheduler über die Ausfälle.

**Der Message Processor** nimmt die einzelnen Nachrichten von dem Communicator entgegen und sorgt für ihre Weiterverarbeitung. Dabei werden die entsprechenden Methoden der Scheduler-Klasse verwendet. Die Auslagerung des Message Processors in einen eigenen Thread soll dafür sorgen, dass der Datentransfer während der Nachrichtendekodierung und Verarbeitung nicht unterbrochen wird. Die Verarbeitung geschieht in drei Schritten. Zuerst muss die Nachricht aus dem ZIP-Format entpackt werden. Dann wird mit Hilfe des X-Stream die Nachricht aus dem XML erstellt. Der Typ der Nachricht entscheidet dann über die Weiterverarbeitung durch die entsprechenden Methoden in dem Scheduler Modul.

**Data Dipository** speichert alle IOObjekte, die als Eingabe für die Tasks benutzt werden, falls die entsprechende Option im DistributedScheduler Operator aktiviert wurde. Auf Anfrage werden diese Daten an die einzelnen Worker verschickt.

**Bechmark** führt in regelmäßigen Abständen einige Berechnungen durch, um die Leistungsfähigkeit des Rechners gegenüber den Workern im Netz zu vergleichen und so über die Anzahl der Tasks, die an den lokalen Worker zugeordnet werden sollen, zu entscheiden. Ist der lokale Worker deaktiviert, wird auch der Benchmark nicht gestartet. Als Berechnungen fungieren die Multiplikation und die Division von Gleitkommazahlen. Natürlich geben dise Werte keine zuverlässigen Information über die Rechner, sie sind aber geeignet um die ungefähre Einschätzung der Leistung gegenüber anderen Rechnern im Netz zu erhalten.

### A.2.2. Der Worker

Der zweite Teil des Plugins läuft auf den externen Rechnern im Netz. Der sogenannte Worker nimmt die Tasks von den Mastern entgegen und sorgt für ihre Verarbeitung. Die Konfiguration des Workers geschieht über eine Textdatei. Wird keinen Konfigurationsdatei gefunden, wird eine neue mit den Standardwerten angelegt. Folgende Einträge können in der Datei stehen:

**yalePort** Gibt den Port an, auf dem der Worker auf die eingehenden Verbindungen von den Mastern warten soll. Diese Portnummer wird auch bei der UDPDiscovery und bei der Verbindung zu dem Zentralen Server mitgeteilt.

**UDPDiscovery** Legt fest, ob die Verbreitung der Workerinformation auch über den UDP Multicast erfolgen soll.

**UDPPort** Gibt an, welcher Port für den UDP Multicast verwendet werden soll. Dieser Port muss für alle Rechner im Netz identisch sein.

**BufferSize** Gibt die Blockgröße an, die man versucht bei der Kommunikation mit dem Master zu lesen oder zu schreiben. Je schneller die Verbindung, desto größer kann dieser Wert genommen werden. Die Standardeinstellung ist 64KB.

**Number of workers** gibt die Anzahl der Arbeiter an, die auf diesem Rechner laufen sollen. Es gelten die gleichen Einschränkungen, wie mit dem lokalen Worker auf dem Masterrechner.

**ServerAddress** Wird eine Serveradresse angegeben, versucht der Worker beim Start sich zu diesem Server zu verbinden und wartet auf die Information über die Master.

**UseFIFO** Ist dieser Wert gesetzt wird die FIFO Implementierung der Workers verwendet, ansonsten wird nach dem Round Robin Prinzip verfahren.

Der Rest der Einstellungen in dieser Datei dient dem Debug- und Testzwecken und ist für die normale Benutzung nicht von Bedeutung.
Von dem Aufbau her, ist der Worker dem Master sehr ähnlich. Das Programm setzt sich auch aus drei Hauptmodulen zusammen. Dem Discoverer, dem Communicator und dem

Worker. Die Funktion des Communicators ist dabei mit der Mastervariante identisch. Der Discoverer hat etwas andere Aufgaben. Er wird nur gestartet, falls in den Optionen ein zentraler Server oder die UDPDiscovery aktiviert wurden. Dann verbreitet er seine Verbindungsinformationen über das Netz. Ansonsten ist der Rechner passiv und wartet einfach darauf, dass ein Master sich mit ihm verbindet. Der Worker selber verwaltet alle ankommenden Tasks und führt sie aus. Falls die Ausführung erfolgreich war, wird das Ergebnis zurück an den Master geschickt. Ansonsten wird eine Fehlernachricht erstellt und ebenfalls an den Master gesendet. Wenn sich mehr als ein Master zu dem Worker verbunden hat, stellt sich die Frage in welcher Reihenfolge die Tasks ausgeführt werden sollen. Hier wurden zwei Verfahren implementiert und können in den Optionen ausgewählt werden. Bei dem FIFO (First In First Out) Verfahren gibt es nur eine globale Schlange für alle Tasks. Sie werden gemäss ihrer Ankunftszeit ausgeführt. Der älteste Task zuerst. Anders geht das Round Robin Verfahren vor. Hier gibt es eine Schlange für jeden Master. Der Worker geht die Liste mit den Mastern durch, und führt dabei nur einen Task pro Master aus. So wird gewährleistet, dass die Anzahl der ausgeführten Tasks pro Master gleich ist.

### A.2.3. Der Server

Die dritte Komponente des Plugins ist der zentrale Server. Er funktioniert als eine Art Worker Datenbank. Der Nutzen von dem Server liegt in zwei Bereichen. In einer unbekannten Umgebung ist es für einen Master oft sehr schwer eine aktualisierte Workerliste zu pflegen. Falls ein bekannter Server existiert, braucht der Master nur seine aktuelle Adresse. Nach der Verbindung kriegt er so die Daten aller aktiven Worker im Netz. Ein anderes Problem könnte sein, dass ein Worker oder Master hinter einer Firewall oder einem falsch konfiguriertem Router sind. In diesem Fall kann ein Rechner zwar Verbindungen nach außen herstellen, aber keine Verbindungen von außen akzeptieren. Das heißt, sogar wenn der Master eine aktuelle Workerliste hat, wird er einen Worker hinter einer Firewall nicht erreichen können. Hier kommt der Server ins Spiel. Er besitzt eine permanente Verbindung zu jeden bekannten Worker. Verbindet sich ein neuer Master, so werden seine Daten an alle Worker weitergeleitet, so dass diese von sich aus eine Verbindung initiieren können. Damit dieses System funktioniert, muss entweder der Master oder der Worker externe Verbindungen akzeptieren. Diese Adresse des Servers muss auch jedem Worker bekannt sein, damit er sich zu diesem verbinden kann.

Die Implementierung des Servers ist viel einfacher, als der beiden anderen Teile. Da er passiv ist, ist der Entdecker nicht nötig. Auch finden keine zeitintensiven Berechnungen statt. So reicht für das Funktionieren nur ein Modul: Der Communicator. Er wartet auf die Verbindungen von Mastern und Workern. Alle verbundenen Worker werden in einer Liste verwaltet. Diese Liste wird regelmäßig überprüft und von den nicht mehr erreichbaren Rechnern bereinigt. Sollte sich ein Master verbinden, so kriegt er die aktuelle Liste. Außerdem wird eine Nachricht mit den Masterdaten an jeden Worker in der Liste geschickt.

### A.2.4. Der Schedulingserver

Wie schon in 7.6.4 beschrieben besteht die Möglichkeit einen externen Server für die Schedulingaufgabe zu verwenden. Dieser Server muss einen Teil der Funktionalität von dem Master und dem Worker bereitstellen. Für den Master fungiert er als Worker und kann über die entsprechenden Optionen im DistributedScheduler Operator angegeben werden. Für die Worker fungiert er als Master. In der jetzigen Version ist nur die Entdeckung der Worker im LAN implementiert, lässt sich aber leicht für die beiden anderen Methoden erweitern. Dabei schalten die Master ihre UDP Discovery in den Optionen aus. Jetzt ist der zentrale Scheduler der einzige sichtbare Master im Netz und wird von den Workern entsprechend behandelt.

## A.3. Eigene Operatoren schreiben

Zwar kommt das Plugin mit einigen Operatoren, doch diese können ohne großen Aufwand durch neue erweitert werden. Falls man einen geeigneten Operator für die parallele Ausführung identifiziert hat, ist die Parallelisierung nicht schwer. Die Ausführung des Experiments läuft dabei in mehreren Schritten ab.

1. Das Experiment wird gestartet. Der DistributedScheduler Operator wird ausgeführt, der das Netzwerk nach Rechnern absucht, und sich mit ihnen verbindet.
2. Der parallelisierte Operator wird aufgerufen. Er erstellt eine Liste mit Tasks und sendet diese für die Bearbeitung an den Scheduler.
3. Der Scheduler ordnet die erhaltenen Tasks nach der ausgewählten Methode den einzelnen Workern zu.
4. Die Rechner im Netz führen die Berechnungen aus und schicken die Ergebnisse zurück an den Scheduler.
5. Der Scheduler leitet die Ergebnisse weiter an den parallelisierten Operator.
6. Der Operator wertet die Ergebnisse aus.
7. Nachdem alle Tasks abgeschlossen sind, signalisiert der Aufraggeber das Ende der Berechnungen an den Scheduler.
8. Der Scheduler räumt den Cache auf und sendet entsprechende Nachrichten an die einzelnen Rechner.
9. Das Experiment wird beendet. Die Verbindungen mit den einzelnen Workern werden getrennt.

Die erste Aufgabe des Operators ist es also die Berechnungen in einzelne Tasks zu zerlegen und daraus dann TaskObjekte zu erstellen, die dann zu dem Scheduler weitergeleitet werden. In dem TaskObjekt müssen folgende Informationen über den Task erhalten sein:

**TaskID** Falls die Reihenfolge in der die Ergebnisse zurückkommen eine Rolle spielt, sollte jeder Tasks eine ID kriegen. Wenn die Reihenfolge nicht von Bedeutung ist (wie zum Beispiel bei einer Kreuzvalidierung), kann die ID einfach weggelassen werden und wird vom Scheduler automatisch erstellt. Diese TaskID ist mit der ID der Ergebnisse identisch und kann verwendet werden um Tasks den Ergebnissen zuzuordnen.

**WorkerOperator** Es muss spezifiziert werden, welche Klasse auf dem Zielrechner die Bearbeitung des Tasks übernimmt. Zur Bearbeitung können die Klassen aus der network.worker.operator Package benutzt oder eigene geschrieben werden. Die Angabe muss den kompletten Paketpfad und den Klassennamen enthalten, so dass die entsprechende Klasse von Java gefunden werden kann. Die meisten Operatoren sind dabei für Spezialfälle bestimmt. Für die einfache Hintereinanderausführung von Operatoren kann die OperatorChain Worker-Klasse benutzt werden. Der WorkerOperator ist eine der beiden Pflichtinformationen, die an den Scheduler übergeben werden müssen.

**Aufraggeber** Die zweite Pflichtinformation. Hier wird angegeben, an welche Klasse der Scheduler die Ergebnisse der Berechnungen schicken soll. Diese Empfängerklasse muss das MasterOperator Interface implementieren.

**Liste der Parameter** Es kann eine Liste der Parameter für die Ausführung übergeben werden. Die Liste ist eine MAP mit Parametername/ Parameterwert Paaren. Beide Werte müssen in dem String-Format vorliegen. Die Werte werden von dem Worker ausgewertet und bei den Berechnungen miteinbezogen.

**Die Eingabe** ist ein IOObject auf dem die Berechnungen ausgeführt werden sollen. Normalerweise ein ExampleSet. Bei der Generierung der Eingabe, sollte man beachten, dass man die Eingabe des vorherigen Tasks nicht überschreibt, also wenn nötig eine Kopie erstellen. Die Eingabe kann zwar auch weggelassen werden, da aber die meisten Operatoren eine Eingabe benötigen, sollte ein entsprechender Generator dem Worker übergeben werden.

**Liste der Operatoren** Es kann eine Liste mit Operatoren übergeben werden, die am Zielrechner ausgeführt werden sollen. Es sind alle Operatoren und auch OperatorChains erlaubt, sollte der RapidMiner auf dem Workerrechner sie kennen. Dabei sollte auf die richtige Einfügereihenfolge der Operatoren in die Liste geachtet werden, da es die einzige Identifikationsmöglichkeit für den Worker ist.

Nachdem die Tasks zusammengestellt worden sind, müssen sie an der Scheduler weitergeleitet werden. Dafür stehen in der Schedulerklasse zwei Methoden zur Verfügung. Die Methode schedule(TaskObject task) kann benutzt werden um einzelne Tasks zu verteilen. Wenn man eine Liste der Tasks hat, kann auch die Methode schedule(List taskList) benutzt werden. Diese Methode ist vorzuziehen, da viele Algorithmen die genaue Anzahl der Tasks wissen müssen, bevor sie mit der Zuordnung anfangen.

Sind alle Tasks an dem Scheduler übergeben worden, bleibt es nur noch auf die Ergebnisse zu warten. Diese Ergebnisse werden an den vorher spezifizierten MasterOperator zurückgeliefert. Diese Klasse muss die addResult Methode implementieren, über welche die Ergebnisse dann übermittelt werden. Jedes Ergebnisobjekt enthält dabei die ID des

Tasks, den Verweis auf den ursprünglichen Task und das Resultat der Berechnungen auf dem externen Rechner als einen IOContainer. Sehr oft kommen nach einer gewissen Wartezeit gleich mehrere Ergebnisse an, weil mehrere gleichschnelle Rechner gleichzeitig fertig geworden sind. Deshalb sollte man die Ergebnisse zwischenspeichern und in einem eigenen Thread mit des Auswertung beginnen, um den Rest den Programms nicht zu blockieren.
Der Operator sollte sich die Anzahl der erwarteten und der bereits verarbeiteten Ergebnisse merken, um das Warten zu beenden, nachdem alle Ergebnisse eingetroffen sind. Wenn die Verarbeitung aller gegenwärtigen Tasks abgeschlossen ist, kann der Auftraggeber ein Signal an den Scheduler senden, das signalisiert, dass die zwischengespeicherten Daten nicht mehr benötigt werden, und gelöscht werden können. Dies geschieht durch den Aufruf der computationFinished() Methode des Schedulers. Dieser Aufruf ist dann nützlich, wenn ein Operator seine Tasks in „Wellen“ aussendet, die alle auf den gleichen Daten arbeiten. Zwischen jeder Welle kann man die gespeicherten Eingabedaten löschen, um Speicherplatz zu sparen. Durch diese Methode wird der Cache-Speicher bei dem Master und den Workern geleert. Außerdem werden alle noch nicht gestarteten Tasks entfernt. Sehr nützlich ist diese Methode bei der Feature Selection, um nach dem Ende jeder Generation den Speicher wieder freizumachen. Der Aufruf der Methode ist aber keine Pflicht. Das Aufräumen geschieht nach dem Ende des Experiments automatisch.

Wenn man auf der Workerseite nicht die Standardklasse für die Bearbeitung verwenden möchte, kann man auch hier eine eigene Klasse verwenden. Diese Klasse muss das Interface WorkerOperator implementieren und folgende Methoden besitzen:
*setParameters(Map parameters)* Diese Methode wird vor der Ausführung aufgerufen. Die vom Master stammenden Parameter für die Ausführung können hier analysiert und gespeichert werden.
*apply(IOContainer input, List operatorList)* In dieser Methode findet die Ausführung des Tasks statt. Als Eingabe muss man einen IOContainer vom RapidMiner akzeptieren und eine Liste mit Operatoren, die ausgeführt werden sollen. Diese Eingaben können auch leer sein. Nach dem Ende der Berechnungen sollte ein IOContainer als Ergebnis zurückgegeben werden. Falls ein Fehler auftritt, wird dieser an den Master weitergeleitet und die Ausführung des Tasks abgebrochen.

## A.4. Eigene Schedulingverfahren schreiben

Obwohl das Plugin bereits zahlreiche verschiedene Schedulingverfahren zur Verfügung stellt, können auch weitere eingefügt werden. Um ein neues Verfahren zu implementieren, muss man entweder den bestehenden DistributedScheduler Operator in RapidMiner überschreiben, oder -und das ist der bessere Weg- einen neuen Operator schreiben, der von DistributedScheduler erbt. Möchte ein Benutzer den neuen Algorithmus verwenden, so bindet er den neuen Operator anstelle der Standardimplementierung in sein Experiment ein.

Beim Schreiben des neuen DistributedSchedulers sind die gleichen Regeln, wie für die Erstellung von normalen Operatoren zu beachten. Das heißt, es müssen die Methoden *getInputClasses*, *getOutputClasses*, und *getParametertypes* vorhanden sein, wobei man hier auf die vorhandene Implementierung der Oberklasse zugreifen kann. Außerdem muss der Operator in einer xml-Datei angegeben werden, und diese Datei dem RapidMiner bekannt sein.
Um zusätzlich das eigene Schedulingverfahren einzufügen müssen zwei Dinge getan werden. Das Array SchedulerTypes enthält die Namen der bereits implementierten Verfahren. Diese Liste muss mit dem neuen Verfahren ergänzt werden. Dazu legt man ein neues Array an, und kopiert die alten Werte hinein. Anschließend fügt man seine neuen Verfahren am Ende der Liste an. Schließlich muss das neue Array der Variable SCHEDULERTYPES hinzugewiesen werden. Jetzt taucht das neue Verfahren in der GUI auf, und kann von dem Benutzer ausgewählt werden.
Abschließend muss noch die Methode *createSchedulerThread()* überschrieben werden. In dieser Methode sorgt man dafür, dass das neue Verfahren auch initialisiert wird. Das erreicht man durch den Aufruf des Konstruktors des neuen Schedulers und gibt anschließend das neu generierte Objekt zurück.
Jetzt fehlt nur noch die Implementierung des Verfahrens selber. Die neue Klasse sollte vom Scheduler erben, um auf alle bereits implementierten Methoden zugreifen zu können. Der Hauptteil der neuen Klasse sollte aus einer Endlosschleife bestehen, die in einem eigenen Thread auf eingehende Tasks wartet und diese dann auf einzelne Worker verteilt.

### Die wichtigsten Datenstrukturen und Variablen

Bei der Ausführung kann das neue Verfahren auf viele nützliche Variablen und Datenstrukturen der Oberklasse zugreifen. Hier sind die wichtigsten im Überblick:
**tasks** Diese MAP bildet die ID Nummer auf einen Task ab. In ihr sind alle noch nicht fertigen Tasks gespeichert und werden mit dem Eintreffen der Ergebnisse entfernt.
**openTasks** In dieser Liste sind alle Tasks enthalten, die noch keinen Arbeiter zugewiesen bekommen haben. Nach der Zuweisung werden sie von dieser Liste entfernt. Normalerweise sollte der Schedulingthread erst aktiv werden, wenn diese Liste nicht leer ist, es also nicht zugeordnete Tasks gibt. Ausnahmen sind Verfahren, die mit der Replikation oder Migration arbeiten.
**workerMap** Diese MAP bildet Netzwerkadressen auf die einzelnen Arbeiter ab. Sie sollte dafür benutzt werden, geeignete Empfänger für die Taskausführung zu finden. Die Größe dieser Datenstruktur kann sich während der Laufzeit ändern, falls neue Worker hinzukommen, oder die Verbindung zu einem bestehenden Worker unterbrochen wird. Auch der lokale Worker ist hier gespeichert. Falls er existiert kann er unter dem Schlüssel 'null' gefunden werden. Dadurch kann er, wenn nötig, von den externen Workern unterschieden werden. Ansonsten ist sein verhalten gleich. Auch hier können die Tasks mit der *sendTask* Methode zugeordnet werden.
**stopped** Ist dieser boolean Wert auf wahr gesetzt, wurde das Experiment aus irgendeinem Grund unterbrochen. Der Scheduler sollte die run() Methode verlassen und die Ausführung beenden.

**scheduledTasks** In dieser Variable ist die Anzahl der bereits zugeordneten Tasks gespeichert.

Neben den Variablen der Schedulerklasse stehen auch Informationen der anderen Objekte zur Verfügung.

**Das Task Objekt**
Der Task selbst ist für den Scheduler eher unwichtig. Die einzige Information die man aus diesem Objekt benutzen könnte ist die erwartete Laufzeit des Tasks. Sie ist allerdings, wie schon vorher besprochen, sehr ungenau und sollte nur mit Vorsicht verwendet werden.

**Der Worker**
Der Worker liefert alle Information, die für das Verteilen von Tasks notwendig sind. Die einzelnen Informationen sind:
**isFree()** Liefert wahr, wenn dieser Master zur Zeit keine Tasks an diesen Worker zugeordnet hat. Falls keine anderen Master im Netz existieren, könnte der Worker mit dem nächsten Task sofort beginnen.
**scheduledLocalTasks()** Liefert die Anzahl der Tasks, die von diesem Master an diesen Arbeiter gesendet wurden. Sogar wenn diese Liste leer ist, kann die isFree() Methode falsch liefern, wenn der Worker zum Beispiel abgebrochene Tasks noch immer ausführt. Für die Feststellung der Verfügbarkeit des Worker sollte daher die isFree() Methode verwendet werden.
**finishedTasks()**Liefert die Anzahl der bisher fertigbearbeiteten Tasks zurück. Kann benutzt werden, um die relative Geschwindigkeit zu den anderen Arbeiten zu berechnen, oder die entsprechende Ausgabe in der GUI zu aktualisieren.
**getLastComputingTime()** Liefert die Zeit zurück, die nötig war um den aktuellen Task zu bearbeiten. Kann benutzt werden, um Aussagen für die Zukunft zu treffen.
**getWorkerSpeed()** In periodischen Abständen führt jeder Arbeiter einen Benchmark aus. Das Ergebnis kann hier abgefragt werden. Je höher dieser Wert, desto Leistungsfähiger der Rechner. Und das Netz nicht zu belasten, erfolgt die Übertragung der Benchmarkergebnisse nicht automatisch, sondern muss mit der Methode requestStatus() beim Scheduler angestoßen werden.
**getNumberOfAssignedTasks()** Liefert die Gesamtzahl der Tasks, die bei diesem Arbeiter auf die Ausführung warten. Das schließt die lokalen und die Tasks der anderen Master im Netz mit ein.
**getExpectedRuntime()**Liefert die erwartet Ausführungszeit für alle wartenden Tasks beim Arbeiter. Dieser Wert enthält die Laufzeit der Tasks aller Master (einschließlich des Lokalen) im Netz. Da die erwartete Laufzeit der Tasks sehr ungenau ist und die Angabe der gleichen Zeit zwischen den Mastern stark schwanken kann, ist dieser Wert mit großer Vorsicht zu gebrauchen.
**usesFifo()**Liefert wahr, falls der Arbeiter das FIFO(First In First Out) Verfahren zur Taskverarbeitung benutzt. Die Alternative ist das RoundRobin Prinzip.
**getName()** Liefert den Namen dieses Arbeiters zurück. Als Name wird dabei der Hostname des Rechners verwendet, beziehungsweise LocalWorker für den lokalen Arbeiter.

Da der Name sich zwischen den einzelnen Experimenten nicht ändert, kann er zur Arbeiteridentifikation benutzt werden.

**Das Schedulingprozedere**
Wurde ein Task zum Scheduling ausgewählt muss dieser zuerst aus der Liste der offenen Tasks gelöscht werden. Danach wird aus der Arbeiterliste ein passender Arbeiter für die Ausführung gesucht. Ist der Arbeiter gefunden, so kann der Task mit der sendTask() Methode an den entfernten Rechner übertragen werden. Den Rest übernimmt das System. Auch die Antwort wird automatisch an den Operator weitergeleitet.

**Veränderung der Kommunikation**
Die Kommunikation zwischen dem Master und dem Worker wird durch einzelne Nachrichten realisiert. Die Reaktion auf diese Nachrichten kann der Scheduler beeinflussen. Dazu muss man die entsprechenden Methoden der Scheduler-Kalsse überschreiben. Folgende Methoden können verändert werden:
**processInputRequestMessage** Diese Methode wird ausgeführt, wenn einem Arbeiter die Eingabe für den Task fehlt. Die Eingabe wird dann aus der Data Dipository geholt und zurück an den Arbeiter geschickt.
**processResultMessage** Hier wird festgelegt, was mit den Ergebnissen passiert, die von anderen Rechnern kommen. Standardmäßig wird das Ergebnis einfach weiter an der Auftraggeber weitergeleitet. Für manche Schedulingverfahren ist zum Beispiel aber auch der konkrete Zeitpunkt der Nachrichtankunft wichtig.
**processErrorMessage** Legt das Verhalten fest, falls eine Fehlermeldung von einem Arbeiter kommt. Standardmäßig wird der Task wieder zu der Liste der offenen Tasks hinzugefügt. Der Master könnte aber auch das Programm beenden oder den Arbeiter als unsicher markieren beziehungsweise von der Arbeiterliste entfernen.
**processReportMessage** Wertet Information über den Arbeiter aus und speichert sie in dem Worker-Objekt.

Normalerweise müssen diese Methoden nicht verändert werden. Für manche Scheduler ist es aber wichtig, Zugriff auf zusätzliche Informationen zu haben, oder ein bestimmtes Programmverhalten zu erzwingen. Neben der Kommunikation kann auch das Verhalten beim hinzufügen oder entfernen von Workers festgelegt werden. Dies geschieht durch die addWorker und removeWorker Methoden des Schedulers.

Für weitere Informationen sollte auch die Dokumentation in den einzelnen Klassen verwendet werden.

# B. Experimente im XML-Format

In diesem Teil des Anhangs sind die ausgeführten Experimente in XML-Format aufgelistet. Dieses Format wird vom RapidMiner benötigt um die Tests einlesen zu können. Insgesamt wurden 4 verschiedene Experimente erstellt. Nähere Informationen über die einzelnen Experimente können in dem Abschnitt 7.1 nachgelesen werden.

```
<?xml version="1.0" encoding="UTF-8"?>
<process version="4.0">
<operator name="Root" class="Process">
<operator name="Scheduler" class="Scheduler">
<parameter key="scheduler_type" value="fixed"/>
</operator>
<operator name="ExampleSetGenerator" class="ExampleSetGenerator">
<parameter key="number_examples" value="600"/>
<parameter key="target_function" value="sinus frequency"/>
</operator>
<operator name="DistributedXValidator" class="DistributedXValidator">
<parameter key="local_random_seed" value="100"/>
<parameter key="sampling_type" value="shuffled sampling"/>
<operator name="Training" class="LibSVMLearner">
<parameter key="C" value="100.0"/>
<list key="class_weights">
</list>
<parameter key="kernel_type" value="linear"/>
<parameter key="svm_type" value="epsilon-SVR"/>
</operator>
<operator name="ApplierChain" class="OperatorChain">
<operator name="Test" class="ModelApplier">
<list key="application_parameters">
</list>
</operator>
<operator name="RegressionPerformance" class="RegressionPerformance">
<parameter key="root_mean_squared_error" value="true"/>
</operator>
</operator>
</operator>
</operator>
</process>
```

Abbildung B.1.: GROB-10: Grobkörnig, ähnliche Laufzeit

```
<?xml version="1.0" encoding="UTF-8"?>
<process version="4.0">
<operator name="Root" class="Process">
<operator name="Scheduler" class="Scheduler">
<parameter key="scheduler_type" value="fixed"/>
</operator>
<operator name="ExampleSetGenerator" class="ExampleSetGenerator">
<parameter key="local_random_seed" value="5"/>
<parameter key="number_examples" value="400"/>
<parameter key="target_function" value="polynomial"/>
</operator>
<operator name="ExampleSetGenerator (2)" class="ExampleSetGenerator">
<parameter key="local_random_seed" value="10"/>
<parameter key="number_examples" value="400"/>
<parameter key="target_function" value="non linear"/>
</operator>
<operator name="ExampleSetGenerator (3)" class="ExampleSetGenerator">
<parameter key="local_random_seed" value="15"/>
<parameter key="number_examples" value="800"/>
<parameter key="number_of_attributes" value="6"/>
<parameter key="target_function" value="sum"/>
</operator>
<operator name="ExampleSetGenerator (4)" class="ExampleSetGenerator">
<parameter key="local_random_seed" value="20"/>
<parameter key="number_examples" value="400"/>
<parameter key="number_of_attributes" value="7"/>
<parameter key="target_function" value="polynomial"/>
</operator>
<operator name="ExampleSetGenerator (5)" class="ExampleSetGenerator">
<parameter key="local_random_seed" value="25"/>
<parameter key="number_examples" value="200"/>
<parameter key="target_function" value="polynomial"/>
</operator>
<operator name="ExampleSetGenerator (6)" class="ExampleSetGenerator">
<parameter key="local_random_seed" value="30"/>
<parameter key="number_examples" value="150"/>
<parameter key="target_function" value="polynomial"/>
</operator>
<operator name="ExampleSetGenerator (7)" class="ExampleSetGenerator">
<parameter key="local_random_seed" value="35"/>
<parameter key="number_examples" value="500"/>
<parameter key="target_function" value="polynomial"/>
</operator>
<operator name="ExampleSetGenerator (8)" class="ExampleSetGenerator">
<parameter key="local_random_seed" value="40"/>
<parameter key="number_examples" value="300"/>
<parameter key="target_function" value="non linear"/>
</operator>
<operator name="ExampleSetGenerator (9)" class="ExampleSetGenerator">
<parameter key="local_random_seed" value="45"/>
<parameter key="number_examples" value="200"/>
<parameter key="number_of_attributes" value="10"/>
<parameter key="target_function" value="polynomial"/>
</operator>
<operator name="ExampleSetGenerator (10)" class="ExampleSetGenerator">
<parameter key="local_random_seed" value="50"/>
<parameter key="number_examples" value="1100"/>
<parameter key="number_of_attributes" value="3"/>
<parameter key="target_function" value="polynomial"/>
</operator>
<operator name="ExampleSetGenerator (11)" class="ExampleSetGenerator">
<parameter key="local_random_seed" value="45"/>
<parameter key="number_examples" value="200"/>
<parameter key="number_of_attributes" value="10"/>
<parameter key="target_function" value="polynomial"/>
</operator>
<operator name="DistributedESetIterator" class="DistributedESetIterator">
<operator name="XValidation" class="XValidation">
<parameter key="local_random_seed" value="300"/>
<parameter key="sampling_type" value="shuffled sampling"/>
<operator name="LibSVMLearner" class="LibSVMLearner">
<parameter key="C" value="80.0"/>
<list key="class_weights">
</list>
<parameter key="kernel_type" value="linear"/>
<parameter key="svm_type" value="epsilon-SVR"/>
</operator>
<operator name="OperatorChain" class="OperatorChain">
<operator name="ModelApplier" class="ModelApplier">
<list key="application_parameters">
</list>
</operator>
<operator name="Performance" class="Performance">
</operator>
</operator>
</operator>
</operator>
</operator>
</process>
```

Abbildung B.2.: GRB-11: Grobkörnig, unterschiedliche Laufzeit

```
<?xml version="1.0" encoding="UTF-8"?>
<process version="4.0">
<operator name="Root" class="Process">
<operator name="Scheduler" class="Scheduler">
<parameter key="scheduler_type" value="fixed"/>
</operator>
<operator name="Input" class="ExampleSource">
<parameter key="attributes" value="yale/sample/data/polynomial.aml"/>
</operator>
<operator name="DistGridParameterOptimization" class="DistributedGridParameterOptimization">
<list key="parameters">
<parameter key="Training.C" value="0,20,50,100,150,200,250,500,750,1000"/>
<parameter key="Training.degree" value="1,2,3,4,5"/>
<parameter key="Training.shrinking" value="true, false"/>
</list>
<operator name="Validation" class="XValidation">
<parameter key="local_random_seed" value="100"/>
<parameter key="sampling_type" value="shuffled sampling"/>
<operator name="Training" class="LibSVMLearner">
<parameter key="C" value="1000.0"/>
<list key="class_weights">
</list>
<parameter key="degree" value="5"/>
<parameter key="epsilon" value="0.01"/>
<parameter key="kernel_type" value="poly"/>
<parameter key="shrinking" value="false"/>
<parameter key="svm_type" value="epsilon-SVR"/>
</operator>
<operator name="ApplierChain" class="OperatorChain">
<operator name="Test" class="ModelApplier">
<list key="application_parameters">
</list>
</operator>
<operator name="RegressionPerformance" class="RegressionPerformance">
<parameter key="root_mean_squared_error" value="true"/>
</operator>
</operator>
</operator>
</operator>
</operator>
</process>
```

Abbildung B.3.: FEIN-100: Feinkörnig, unbekannte Laufzeit der Tasks

```
<?xml version="1.0" encoding="UTF-8"?>
<process version="4.0">
<operator name="Root" class="Process">
<operator name="Scheduler" class="Scheduler">
<parameter key="scheduler_type" value="fixed"/>
</operator>
<operator name="DistributedIteratingOperatorChain" class="DistributedIteratingOperatorChain">
<parameter key="iterations" value="80"/>
<parameter key="local_operators" value="0"/>
<operator name="ExampleSetGenerator" class="ExampleSetGenerator">
<parameter key="local_random_seed" value="50"/>
<parameter key="number_examples" value="200"/>
<parameter key="number_of_attributes" value="4"/>
<parameter key="target_function" value="polynomial"/>
</operator>
<operator name="XVal" class="XValidation">
<parameter key="local_random_seed" value="100"/>
<parameter key="sampling_type" value="shuffled sampling"/>
<operator name="Training" class="LibSVMLearner">
<parameter key="C" value="80.0"/>
<list key="class_weights">
</list>
<parameter key="kernel_type" value="linear"/>
<parameter key="svm_type" value="epsilon-SVR"/>
</operator>
<operator name="ApplierChain" class="OperatorChain">
<operator name="Test" class="ModelApplier">
<list key="application_parameters">
</list>
</operator>
<operator name="RegressionPerformance" class="RegressionPerformance">
<parameter key="root_mean_squared_error" value="true"/>
</operator>
</operator>
</operator>
</operator>
</operator>
</process>
```

Abbildung B.4.: FEIN-80: Feinkörnig, identische Laufzeit

# C. Ergebnisse der Testreihen

In diesem Anhang sind die gesamten Ergebnisse der Experimente gesammelt. Pro Versuchsreihe wurde eine eigene Tabelle erstellt. Die Werte in den Tabellen sind die gemittelten Ergebnisse der Ausführung. Die Ausführungszeit ist dabei in Sekunden angegeben. Genaue Angaben für die Zusammensetzung der Werte und Auswahl der Spalten können in dem Abschnitt 7.3 nachgelesen werden.
In den Tabellen C.1 bis C.16 befinden sich die Ergebnisse der ersten Testreihe mit einem Master. Dabei Sind die Ergebnisse einer Umgebung jeweils in einer Tabelle zusammengefasst.
Die Ergebnisse der zweiten Testreihe können ab der Tabelle C.17 angesehen werden. Hier gibt es für Experimente mit externer Last mehrere Tabellen pro Experiment, da man zwischen FIFO und Round Robin auf der Workerseite unterscheiden muss.
Schließlich geben die letzen 4 Tabellen C.41 bis C.44 die Ergebnisse des Experiments mit einem nach dem Start gedrosselten Rechner wieder.

| Algo | Makespan | | | Leerlauf | | |
|---|---|---|---|---|---|---|
| — | min | average | max | min | average | max |
| FIXED | 78.3 | 78.9 | 79.6 | 1.4 | 20.4 | 32.9 |
| FIXED-W | 76.3 | 79.8 | 84.9 | 1.8 | 21.3 | 33.0 |
| RANDOM | 65.4 | 85.0 | 132.7 | 1.5 | 26.6 | 49.6 |
| RANDOM-W | 66.0 | 87.9 | 129.4 | 1.7 | 29.3 | 55.3 |
| MCT | 76.7 | 80.4 | 85.6 | 2.1 | 21.8 | 34.3 |
| MINMAX | 64.7 | 70.9 | 82.7 | 2.0 | 12.4 | 23.6 |
| MINMIN | 65.0 | 73.1 | 85.1 | 1.9 | 14.6 | 27.4 |
| WQ | 78.1 | 78.6 | 79.3 | 2.9 | 20.1 | 30.9 |
| TSS | 77.6 | 78.2 | 78.8 | 2.6 | 19.9 | 29.4 |
| FAC | 77.7 | 78.5 | 79.3 | 2.9 | 20.1 | 30.9 |
| S-WQR | 77.4 | 78.6 | 84.5 | 2.7 | 2.9 | 3.3 |
| TO-WQR | 79.8 | 80.5 | 81.2 | 2.4 | 22.5 | 33.2 |

Tabelle C.1.: GROB-10 - ein Master - homogene Umgebung. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | Makespan | | | Leerlauf | | |
|---|---|---|---|---|---|---|
| — | min | average | max | min | average | max |
| FIXED | 64.5 | 65.2 | 65.9 | 1.6 | 24.2 | 46.4 |
| FIXED-W | 59.9 | 68.2 | 73.2 | 1.3 | 27.4 | 45.2 |
| RANDOM | 44.1 | 62.7 | 95.7 | 1.4 | 21.8 | 42.4 |
| RANDOM-W | 43.9 | 59.5 | 77.7 | 1.5 | 18.6 | 35.2 |
| MCT | 47.7 | 49.4 | 56.2 | 2.1 | 8.5 | 18.6 |
| MINMAX | 58.2 | 59.2 | 60.2 | 1.9 | 18.4 | 36.0 |
| MINMIN | 55.5 | 56.1 | 57.3 | 1.5 | 15.3 | 23.0 |
| WQ | 44.1 | 44.8 | 46.8 | 2.3 | 4.0 | 5.1 |
| TSS | 45.4 | 45.9 | 46.6 | 1.2 | 4.9 | 8.3 |
| FAC | 44.2 | 44.5 | 44.9 | 2.0 | 3.7 | 4.8 |
| S-WQR | 43.7 | 44.1 | 46.4 | 1.8 | 2.3 | 2.7 |
| TO-WQR | 45.2 | 45.6 | 46.7 | 1.2 | 3.1 | 5.6 |

Tabelle C.2.: GROB-11 - ein Master - homogene Umgebung. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | Makespan | | | Leerlauf | | |
|---|---|---|---|---|---|---|
| — | min | average | max | min | average | max |
| FIXED | 61.8 | 62.3 | 62.9 | 1.0 | 2.1 | 3.6 |
| FIXED-W | 61.9 | 62.8 | 64.8 | 1.1 | 2.7 | 4.5 |
| RANDOM | 64.0 | 73.3 | 85.3 | 1.0 | 13.0 | 24.5 |
| RANDOM-W | 64.2 | 72.1 | 92.1 | 1.0 | 11.8 | 22.0 |
| MCT | 62.0 | 62.5 | 64.4 | 1.1 | 2.3 | 4.0 |
| MINMAX | 61.8 | 62.4 | 62.9 | 1.1 | 2.2 | 3.7 |
| MINMIN | 61.9 | 62.5 | 64.1 | 1.1 | 2.3 | 4.0 |
| WQ | 62.2 | 62.5 | 62.9 | 1.6 | 2.3 | 3.4 |
| TSS | 61.8 | 62.3 | 62.8 | 1.2 | 2.0 | 3.2 |
| FAC | 61.8 | 62.2 | 62.7 | 1.3 | 1.9 | 2.9 |
| S-WQR | 63.0 | 63.2 | 63.4 | 2.7 | 2.8 | 2.9 |
| TO-WQR | 61.4 | 61.7 | 62.8 | 1.2 | 2.0 | 3.2 |

Tabelle C.3.: FEIN-80 - ein Master - homogene Umgebung. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | Makespan | | | Leerlauf | | |
|---|---|---|---|---|---|---|
| — | min | average | max | min | average | max |
| FIXED | 62.1 | 62.5 | 63.1 | 1.2 | 2.8 | 3.7 |
| FIXED-W | 112.5 | 120.0 | 129.5 | 5.4 | 60.2 | 91.1 |
| RANDOM | 65.4 | 79.1 | 93.0 | 1.4 | 19.3 | 35.3 |
| RANDOM-W | 64.7 | 82.5 | 102.1 | 1.3 | 22.8 | 41.2 |
| MCT | 63.8 | 65.9 | 72.6 | 4.1 | 6.2 | 8.1 |
| MINMAX | 64.1 | 65.6 | 69.0 | 4.3 | 5.9 | 7.0 |
| MINMIN | 64.2 | 65.6 | 68.4 | 4.3 | 5.8 | 7.1 |
| WQ | 68.6 | 69.1 | 69.6 | 5.6 | 9.6 | 13.3 |
| TSS | 64.7 | 68.5 | 74.0 | 2.7 | 8.8 | 15.3 |
| FAC | 74.7 | 75.0 | 75.6 | 2.6 | 15.3 | 24.2 |
| S-WQR | 68.2 | 68.5 | 69.0 | 4.3 | 5.3 | 6.1 |
| TO-WQR | 63.0 | 69.7 | 72.8 | 2.7 | 4.0 | 5.7 |

Tabelle C.4.: FEIN-100 - ein Master - homogene Umgebung. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | Makespan | | | Leerlauf | | |
|---|---|---|---|---|---|---|
| — | min | average | max | min | average | max |
| FIXED | 46.6 | 66.4 | 102.7 | 2.0 | 39.2 | 58.4 |
| FIXED-W | 32.9 | 33.3 | 33.9 | 2.7 | 19.3 | 33.3 |
| RANDOM | 35.3 | 88.7 | 227.9 | 1.9 | 61.1 | 87.3 |
| RANDOM-W | 27.9 | 46.5 | 67.4 | 1.7 | 30.2 | 46.5 |
| MCT | 31.0 | 33.3 | 36.3 | 1.9 | 18.9 | 33.3 |
| MINMAX | 28.1 | 32.0 | 36.1 | 1.8 | 17.3 | 32.0 |
| MINMIN | 31.4 | 33.8 | 36.9 | 1.8 | 20.4 | 33.8 |
| WQ | 34.3 | 43.6 | 56.2 | 2.2 | 22.0 | 30.2 |
| TSS | 34.0 | 46.9 | 53.6 | 2.3 | 24.7 | 33.8 |
| FAC | 39.8 | 47.6 | 54.8 | 2.4 | 25.6 | 34.4 |
| S-WQR | 26.5 | 32.0 | 41.2 | 1.6 | 2.3 | 2.9 |
| TO-WQR | 34.5 | 43.8 | 46.6 | 2.0 | 20.0 | 29.4 |

Tabelle C.5.: GROB-10 - ein Master - heterogene Umgebung. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | Makespan | | | Leerlauf | | |
|---|---|---|---|---|---|---|
| — | min | average | max | min | average | max |
| FIXED | 29.0 | 56.0 | 98.1 | 1.5 | 37.2 | 53.7 |
| FIXED-W | 25.9 | 28.6 | 30.1 | 1.1 | 19.8 | 28.6 |
| RANDOM | 32.4 | 70.8 | 163.9 | 1.6 | 50.5 | 70.8 |
| RANDOM-W | 18.6 | 36.5 | 92.7 | 1.5 | 25.0 | 36.5 |
| MCT | 21.8 | 24.4 | 25.1 | 1.7 | 16.0 | 24.4 |
| MINMAX | 24.6 | 25.0 | 25.5 | 1.8 | 14.2 | 25.0 |
| MINMIN | 29.2 | 29.7 | 30.4 | 1.5 | 22.9 | 29.7 |
| WQ | 31.4 | 38.8 | 91.0 | 1.7 | 21.7 | 31.3 |
| TSS | 18.4 | 35.0 | 91.5 | 1.6 | 18.6 | 28.8 |
| FAC | 18.6 | 41.1 | 91.2 | 1.8 | 24.4 | 34.9 |
| S-WQR | 18.4 | 24.3 | 29.9 | 1.3 | 1.9 | 2.7 |
| TO-WQR | 22.1 | 29.4 | 36.9 | 1.4 | 10.8 | 17.0 |

Tabelle C.6.: GROB-11 - ein Master - heterogene Umgebung. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | Makespan | | | Leerlauf | | |
|---|---|---|---|---|---|---|
| — | min | average | max | min | average | max |
| FIXED | 67.5 | 71.3 | 75.1 | 1.0 | 43.0 | 61.3 |
| FIXED-W | 24.9 | 25.3 | 25.8 | 1.1 | 9.4 | 19.4 |
| RANDOM | 43.6 | 76.0 | 116.3 | 1.0 | 47.2 | 67.5 |
| RANDOM-W | 23.5 | 27.8 | 32.7 | 1.0 | 11.5 | 22.7 |
| MCT | 24.5 | 25.0 | 25.5 | 1.1 | 8.7 | 19.0 |
| MINMAX | 24.4 | 25.0 | 26.0 | 1.1 | 8.9 | 18.9 |
| MINMIN | 24.5 | 25.0 | 25.7 | 1.1 | 8.8 | 18.9 |
| WQ | 21.6 | 23.3 | 26.2 | 1.2 | 3.4 | 4.7 |
| TSS | 21.5 | 24.3 | 25.8 | 1.2 | 4.3 | 5.8 |
| FAC | 21.6 | 24.4 | 26.3 | 1.2 | 4.4 | 6.0 |
| S-WQR | 21.6 | 22.0 | 23.0 | 1.0 | 1.4 | 1.6 |
| TO-WQR | 21.6 | 24.5 | 26.3 | 1.3 | 4.6 | 6.3 |

Tabelle C.7.: FEIN-80 - ein Master - heterogene Umgebung. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | Makespan | | | Leerlauf | | |
|---|---|---|---|---|---|---|
| — | min | average | max | min | average | max |
| FIXED | 44.0 | 69.6 | 108.4 | 1.5 | 42.4 | 62.3 |
| FIXED-W | 31.7 | 43.5 | 69.4 | 4.8 | 28.8 | 41.9 |
| RANDOM | 39.5 | 74.1 | 126.3 | 1.6 | 46.4 | 68.6 |
| RANDOM-W | 24.2 | 35.2 | 57.2 | 2.0 | 19.2 | 31.9 |
| MCT | 26.1 | 30.9 | 38.1 | 4.0 | 14.6 | 29.0 |
| MINMAX | 25.6 | 27.8 | 29.2 | 4.1 | 14.8 | 26.3 |
| MINMIN | 25.2 | 30.0 | 38.5 | 3.6 | 13.7 | 28.8 |
| WQ | 26.7 | 32.9 | 53.7 | 2.6 | 13.1 | 18.9 |
| TSS | 23.9 | 34.8 | 58.1 | 3.0 | 16.2 | 22.4 |
| FAC | 23.4 | 30.5 | 47.2 | 3.7 | 11.3 | 16.7 |
| S-WQR | 23.7 | 26.4 | 29.4 | 2.3 | 3.0 | 3.8 |
| TO-WQR | 22.0 | 25.5 | 30.2 | 2.2 | 4.3 | 7.5 |

Tabelle C.8.: FEIN-100 - ein Master - heterogene Umgebung. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | 30% Last | | | 100% Last | | |
|---|---|---|---|---|---|---|
| — — | average Makespan | Standard-abweichung | average Leerlauf | average Makespan | Standard-abweichung | average Leerlauf |
| FIXED | 88.3 | 32.5 | 55.4 | 110.4 | 41.9 | 67.9 |
| FIXED-W | 40.1 | 6.3 | 23.1 | 49.1 | 7.2 | 28.0 |
| RANDOM | 95.2 | 53.9 | 63.1 | 132.4 | 51.3 | 90.1 |
| RANDOM-W | 55.0 | 15.4 | 35.7 | 67.1 | 15.8 | 43.9 |
| MCT | 38.5 | 4.1 | 20.9 | 52.0 | 6.1 | 29.7 |
| MINMAX | 35.3 | 4.0 | 17.1 | 50.6 | 8.6 | 28.0 |
| MINMIN | 38.8 | 3.8 | 21.8 | 51.5 | 5.7 | 31.7 |
| WQ | 55.1 | 6.5 | 28.8 | 66.8 | 12.7 | 33.6 |
| TSS | 57.0 | 8.3 | 31.0 | 70.6 | 18.5 | 36.5 |
| FAC | 53.6 | 7.8 | 27.5 | 73.7 | 16.0 | 39.1 |
| S-WQR | 38.4 | 4.2 | 4.2 | 48.7 | 4.7 | 2.0 |
| TO-WQR | 47.1 | 5.7 | 20.4 | 59.8 | 8.2 | 25.1 |

Tabelle C.9.: GROB-10 - ein Master - heterogene Umgebung - externe Last auf den Rechnern. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | 30% Last | | | 100% Last | | |
|---|---|---|---|---|---|---|
| — — | average Makespan | Standard-abweichung | average Leerlauf | average Makespan | Standard-abweichung | average Leerlauf |
| FIXED | 73.5 | 30.2 | 50.1 | 97.6 | 38.7 | 67.2 |
| FIXED-W | 32.8 | 4.8 | 21.8 | 43.1 | 4.8 | 29.9 |
| RANDOM | 70.6 | 26.5 | 48.2 | 84.7 | 50.7 | 57.9 |
| RANDOM-W | 39.6 | 12.6 | 26.3 | 57.0 | 17.5 | 39.4 |
| MCT | 24.2 | 2.4 | 13.0 | 36.2 | 4.3 | 23.7 |
| MINMAX | 26.1 | 2.4 | 13.3 | 35.2 | 3.5 | 19.6 |
| MINMIN | 32.3 | 3.9 | 23.0 | 44.3 | 3.1 | 34.3 |
| WQ | 58.4 | 28.7 | 37.1 | 70.3 | 39.4 | 43.3 |
| TSS | 47.2 | 15.9 | 27.8 | 69.1 | 39.7 | 43.3 |
| FAC | 55.6 | 25.9 | 34.9 | 57.9 | 25.8 | 32.7 |
| S-WQR | 28.3 | 4.6 | 3.4 | 34.6 | 6.1 | 1.9 |
| TO-WQR | 30.4 | 4.0 | 9.5 | 35.9 | 4.7 | 8.3 |

Tabelle C.10.: GROB-11 - ein Master - heterogene Umgebung - externe Last auf den Rechnern. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | 30% Last | | | 100% Last | | |
|---|---|---|---|---|---|---|
| — — | average Makespan | Standard-abweichung | average Leerlauf | average Makespan | Standard-abweichung | average Leerlauf |
| FIXED | 86.8 | 4.8 | 53.5 | 109.9 | 11.4 | 63.7 |
| FIXED-W | 28.1 | 1.2 | 8.6 | 37.1 | 1.5 | 12.8 |
| RANDOM | 86.3 | 26.5 | 53.2 | 98.5 | 24.3 | 56.8 |
| RANDOM-W | 31.5 | 2.6 | 11.7 | 44.7 | 8.6 | 19.7 |
| MCT | 28.0 | 1.2 | 8.5 | 39.8 | 8.6 | 14.4 |
| MINMAX | 28.4 | 1.0 | 9.1 | 39.2 | 6.1 | 13.7 |
| MINMIN | 28.1 | 1.1 | 8.7 | 42.9 | 13.5 | 16.5 |
| WQ | 26.8 | 2.4 | 3.7 | 35.0 | 2.9 | 4.7 |
| TSS | 26.2 | 2.0 | 3.2 | 34.5 | 1.9 | 4.7 |
| FAC | 26.3 | 2.1 | 3.4 | 34.4 | 2.2 | 4.9 |
| S-WQR | 26.1 | 1.1 | 2.8 | 33.5 | 1.4 | 1.4 |
| TO-WQR | 28.0 | 2.1 | 5.0 | 35.2 | 3.6 | 4.5 |

Tabelle C.11.: FEIN-80 - ein Master - heterogene Umgebung - externe Last auf den Rechnern. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | 30% Last | | | 100% Last | | |
|---|---|---|---|---|---|---|
| — — | average Makespan | Standard-abweichung | average Leerlauf | average Makespan | Standard-abweichung | average Leerlauf |
| FIXED | 81.7 | 20.4 | 49.9 | 111.9 | 25.1 | 69.5 |
| FIXED-W | 51.6 | 11.8 | 32.5 | 59.0 | 13.6 | 36.2 |
| RANDOM | 106.2 | 44.8 | 72.1 | 118.2 | 45.8 | 77.2 |
| RANDOM-W | 40.4 | 11.4 | 21.3 | 52.1 | 11.1 | 29.1 |
| MCT | 41.5 | 10.7 | 23.4 | 47.6 | 6.3 | 24.2 |
| MINMAX | 33.5 | 3.2 | 17.6 | 46.1 | 9.2 | 24.7 |
| MINMIN | 43.3 | 11.9 | 24.4 | 46.6 | 6.9 | 23.4 |
| WQ | 35.7 | 8.8 | 13.3 | 56.4 | 17.4 | 26.0 |
| TSS | 36.3 | 10.5 | 15.2 | 49.4 | 17.6 | 21.3 |
| FAC | 41.2 | 10.4 | 18.5 | 53.3 | 17.2 | 22.9 |
| S-WQR | 31.3 | 2.1 | 4.8 | 38.8 | 3.2 | 2.9 |
| TO-WQR | 30.4 | 3.4 | 6.2 | 39.5 | 5.6 | 6.1 |

Tabelle C.12.: FEIN-100 - ein Master - heterogene Umgebung - externe Last auf den Rechnern. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | Makespan | | | Transfer- | Rechnen- | Rechner- |
|---|---|---|---|---|---|---|
| | min | average | max | overhead | overhead | ausfälle |
| FIXED | 26.1 | 76.4 | 191.1 | 1.5 | 1.2 | 3.1 |
| FIXED-W | 33.4 | 46.1 | 82.6 | 1.9 | 1.1 | 2.5 |
| RANDOM | 50.7 | 89.1 | 139.9 | 2.3 | 1.3 | 4.2 |
| RANDOM-W | 33.5 | 62.7 | 132.2 | 2.2 | 1.4 | 3.7 |
| MCT | 30.9 | 41.5 | 68.8 | 1.4 | 0.8 | 2.2 |
| MINMAX | 29.2 | 38.1 | 51.2 | 1.1 | 0.9 | 1.9 |
| MINMIN | 28.4 | 40.5 | 63.6 | 1.8 | 1.0 | 2.3 |
| WQ | 40.3 | 56.7 | 81.4 | 1.3 | 1.3 | 2.9 |
| TSS | 35.1 | 52.3 | 102.2 | 2.0 | 1.7 | 2.7 |
| FAC | 26.9 | 49.8 | 99.6 | 1.6 | 1.6 | 2.8 |
| S-WQR | 28.0 | 35.4 | 49.6 | 8.3 | 8.1 | 1.9 |
| TO-WQR | 29.3 | 50.5 | 87.0 | 3.1 | 2.5 | 2.7 |

Tabelle C.13.: GROB-10 - ein Master - heterogene Umgebung, Ausfall der Worker möglich. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden. Der Overhead und die Ausfälle sind in der absoluten Taskanzahl angegeben.

| Algo | Makespan | | | Transfer- | Rechnen- | Rechner- |
|---|---|---|---|---|---|---|
| | min | average | max | overhead | overhead | ausfälle |
| FIXED | 29.6 | 61.1 | 123.2 | 1.5 | 0.9 | 3.1 |
| FIXED-W | 22.8 | 45.7 | 233.1 | 1.0 | 0.7 | 2.3 |
| RANDOM | 42.3 | 84.1 | 268.0 | 2.3 | 1.1 | 3.7 |
| RANDOM-W | 23.5 | 37.6 | 66.7 | 1.6 | 0.8 | 2.3 |
| MCT | 20.9 | 26.8 | 40.5 | 1.1 | 0.4 | 1.6 |
| MINMAX | 20.6 | 33.3 | 101.0 | 1.7 | 1.1 | 2.4 |
| MINMIN | 28.8 | 33.3 | 47.5 | 1.7 | 0.7 | 1.6 |
| WQ | 24.9 | 46.9 | 92.2 | 0.8 | 0.8 | 2.4 |
| TSS | 18.6 | 46.2 | 91.8 | 1.3 | 1.3 | 2.3 |
| FAC | 19.0 | 45.1 | 95.5 | 1.0 | 0.9 | 1.8 |
| S-WQR | 18.8 | 26.3 | 31.7 | 8.8 | 7.9 | 1.5 |
| TO-WQR | 22.6 | 28.5 | 38.3 | 4.5 | 4.3 | 1.8 |

Tabelle C.14.: GROB-11 - ein Master - heterogene Umgebung, Ausfall der Worker möglich. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden. Der Overhead und die Ausfälle sind in der absoluten Taskanzahl angegeben.

| Algo | Makespan | | | Transfer- | Rechnen- | Rechner- |
|---|---|---|---|---|---|---|
| | min | average | max | overhead | overhead | ausfälle |
| FIXED | 33.0 | 73.4 | 151.9 | 16.9 | 2.0 | 3.6 |
| FIXED-W | 24.9 | 29.2 | 39.3 | 7.6 | 0.9 | 1.6 |
| RANDOM | 32.9 | 60.8 | 99.0 | 14.4 | 1.9 | 3.7 |
| RANDOM-W | 24.2 | 31.7 | 39.1 | 5.9 | 0.9 | 1.8 |
| MCT | 24.9 | 31.3 | 56.6 | 6.7 | 0.8 | 1.5 |
| MINMAX | 25.3 | 29.8 | 45.0 | 11.6 | 1.2 | 1.9 |
| MINMIN | 25.4 | 29.2 | 36.5 | 6.3 | 0.9 | 1.6 |
| WQ | 22.5 | 35.1 | 52.9 | 1.9 | 1.9 | 2.2 |
| TSS | 21.7 | 26.7 | 33.4 | 3.6 | 1.5 | 1.7 |
| FAC | 22.4 | 26.3 | 29.2 | 5.8 | 1.5 | 1.7 |
| S-WQR | 21.9 | 23.6 | 25.8 | 8.6 | 8.6 | 1.4 |
| TO-WQR | 21.8 | 26.1 | 34.6 | 3.5 | 1.6 | 1.7 |

Tabelle C.15.: FEIN-80 - ein Master - heterogene Umgebung - Ausfall der Worker möglich. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden. Der Overhead und die Ausfälle sind in der absoluten Taskanzahl angegeben.

| Algo | Makespan | | | Transfer- | Rechnen- | Rechner- |
|---|---|---|---|---|---|---|
| | min | average | max | overhead | overhead | ausfälle |
| FIXED | 41.2 | 70.8 | 114.6 | 7.5 | 2.8 | 3.0 |
| FIXED-W | 29.8 | 43.1 | 81.0 | 2.2 | 0.6 | 1.9 |
| RANDOM | 39.6 | 78.6 | 154.0 | 7.6 | 2.3 | 3.2 |
| RANDOM-W | 25.4 | 36.6 | 62.1 | 4.9 | 1.2 | 1.5 |
| MCT | 31.7 | 41.0 | 61.2 | 10.4 | 2.2 | 3.0 |
| MINMAX | 26.8 | 43.1 | 136.2 | 14.1 | 1.0 | 2.3 |
| MINMIN | 25.1 | 42.4 | 103.3 | 4.3 | 1.0 | 1.9 |
| WQ | 29.1 | 42.1 | 75.7 | 1.9 | 1.9 | 2.9 |
| TSS | 25.3 | 36.3 | 53.3 | 2.7 | 1.3 | 2.3 |
| FAC | 28.0 | 39.8 | 64.0 | 3.5 | 1.5 | 2.1 |
| S-WQR | 25.5 | 29.7 | 42.7 | 10.2 | 10.2 | 1.9 |
| TO-WQR | 22.9 | 27.9 | 38.7 | 10.3 | 5.8 | 1.5 |

Tabelle C.16.: FEIN-100 - ein Master - heterogene Umgebung, Ausfall der Worker möglich. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden. Der Overhead und die Ausfälle sind in der absoluten Taskanzahl angegeben.

| Algo | FIFO Worker | | | | Round Robin Worker | | | |
|---|---|---|---|---|---|---|---|---|
| | Makespan | | | Leer- | Makespan | | | Leer- |
| | min | average | max | lauf | min | average | max | lauf |
| FIXED | 138.6 | 166.7 | 190.2 | 0.7 | 80.7 | 106.2 | 172.5 | 3.3 |
| FIXED-W | 107.1 | 178.4 | 266.5 | 4.1 | 81.7 | 100.7 | 130.8 | 6.8 |
| RANDOM | 93.9 | 178.0 | 245.8 | 14.1 | 85.1 | 122.5 | 259.7 | 5.1 |
| RANDOM-W | 140.6 | 182.2 | 226.4 | 3.7 | 75.8 | 119.2 | 169.5 | 6.7 |
| MCT | 90.4 | 188.6 | 315.1 | 6.6 | 104.1 | 163.0 | 242.9 | 6.3 |
| MINMAX | 95.7 | 167.2 | 296.4 | 2.1 | 92.8 | 159.9 | 284.7 | 8.3 |
| MINMIN | 98.4 | 192.3 | 331.3 | 5.1 | 89.5 | 154.3 | 267.4 | 2.9 |
| WQ | 84.2 | 103.1 | 154.9 | 0.2 | 83.7 | 102.3 | 160.9 | 0.2 |
| TSS | 103.1 | 151.6 | 183.7 | 0.7 | 85.1 | 109.7 | 155.8 | 0.1 |
| FAC | 93.9 | 138.9 | 190.8 | 0.7 | 83.7 | 107.7 | 162.0 | 0.3 |
| S-WQR | 83.3 | 103.5 | 148.2 | 0.0 | 85.8 | 117.4 | 166.4 | 0.0 |
| TO-WQR | 100.6 | 161.9 | 194.3 | 0.1 | 90.5 | 117.6 | 173.1 | 0.1 |

Tabelle C.17.: GROB-10 - mehrere Master - homogene Umgebung. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | FIFO Worker | | | | Round Robin Worker | | | |
|---|---|---|---|---|---|---|---|---|
| | Makespan | | | Leer- | Makespan | | | Leer- |
| | min | average | max | lauf | min | average | max | lauf |
| FIXED | 85.2 | 146.0 | 182.9 | 0.4 | 64.5 | 87.9 | 135.3 | 4.4 |
| FIXED-W | 69.6 | 153.8 | 247.8 | 1.1 | 60.9 | 89.0 | 157.7 | 7.6 |
| RANDOM | 114.9 | 170.4 | 209.1 | 2.9 | 60.3 | 114.6 | 274.5 | 8.4 |
| RANDOM-W | 118.0 | 165.3 | 215.7 | 2.5 | 57.9 | 95.0 | 180.1 | 3.9 |
| MCT | 74.8 | 157.1 | 260.1 | 2.9 | 74.8 | 122.3 | 183.9 | 4.1 |
| MINMAX | 77.2 | 165.4 | 270.1 | 2.6 | 78.3 | 156.2 | 285.8 | 19.4 |
| MINMIN | 74.3 | 150.3 | 293.0 | 2.0 | 72.1 | 142.8 | 267.3 | 8.7 |
| WQ | 54.3 | 75.3 | 128.6 | 0.2 | 49.0 | 72.3 | 116.1 | 0.5 |
| TSS | 100.2 | 130.1 | 171.1 | 0.6 | 52.5 | 77.6 | 128.4 | 0.1 |
| FAC | 71.7 | 125.2 | 153.5 | 0.4 | 54.8 | 84.0 | 139.3 | 0.3 |
| S-WQR | 53.2 | 76.5 | 128.8 | 0.0 | 54.1 | 89.5 | 140.5 | 0.0 |
| TO-WQR | 84.4 | 121.3 | 146.4 | 0.0 | 53.7 | 73.2 | 117.9 | 0.0 |

Tabelle C.18.: GROB-11 - mehrere Master - homogene Umgebung. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | FIFO Worker | | | | Round Robin Worker | | | |
|---|---|---|---|---|---|---|---|---|
| | Makespan | | | Leer- | Makespan | | | Leer- |
| | min | average | max | lauf | min | average | max | lauf |
| FIXED | 71.9 | 159.0 | 191.9 | 1.0 | 153.6 | 230.3 | 355.4 | 3.8 |
| FIXED-W | 123.1 | 174.0 | 240.6 | 2.9 | 158.1 | 237.7 | 320.4 | 7.4 |
| RANDOM | 131.7 | 171.3 | 206.5 | 4.6 | 128.1 | 225.4 | 371.1 | 9.2 |
| RANDOM-W | 149.3 | 176.1 | 211.9 | 3.8 | 149.7 | 239.2 | 363.3 | 5.5 |
| MCT | 92.2 | 149.0 | 257.0 | 0.9 | 87.5 | 184.1 | 293.5 | 2.2 |
| MINMAX | 98.9 | 171.3 | 292.0 | 0.4 | 91.5 | 202.2 | 405.0 | 2.6 |
| MINMIN | 97.7 | 162.9 | 270.8 | 0.5 | 78.4 | 206.5 | 350.4 | 2.3 |
| WQ | 121.9 | 211.8 | 277.9 | 0.2 | 145.7 | 231.3 | 352.4 | 0.2 |
| TSS | 162.1 | 186.1 | 214.9 | 1.8 | 124.2 | 211.9 | 323.1 | 0.1 |
| FAC | 147.0 | 178.7 | 215.1 | 1.4 | 129.1 | 210.2 | 306.6 | 0.4 |
| S-WQR | 138.0 | 231.4 | 356.7 | 0.1 | 171.6 | 207.3 | 278.7 | 0.0 |
| TO-WQR | 149.4 | 180.8 | 215.1 | 0.2 | 132.4 | 222.2 | 316.9 | 0.1 |

Tabelle C.19.: FEIN-80 - mehrere Master - homogene Umgebung. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | FIFO Worker | | | | Round Robin Worker | | | |
|---|---|---|---|---|---|---|---|---|
| | Makespan | | | Leer- | Makespan | | | Leer- |
| | min | average | max | lauf | min | average | max | lauf |
| FIXED | 138.0 | 167.9 | 194.8 | 0.4 | 121.4 | 248.5 | 425.0 | 2.8 |
| FIXED-W | 116.0 | 184.0 | 300.5 | 8.3 | 195.5 | 298.1 | 427.6 | 10.9 |
| RANDOM | 115.4 | 166.0 | 236.4 | 1.4 | 168.9 | 242.0 | 411.4 | 9.2 |
| RANDOM-W | 95.5 | 169.6 | 228.0 | 3.7 | 126.7 | 238.2 | 397.8 | 5.8 |
| MCT | 114.6 | 184.7 | 263.9 | 1.3 | 119.2 | 202.8 | 369.0 | 3.5 |
| MINMAX | 100.1 | 162.7 | 227.3 | 0.3 | 116.8 | 211.6 | 334.3 | 6.8 |
| MINMIN | 99.9 | 178.8 | 271.4 | 1.5 | 91.4 | 181.9 | 280.8 | 5.1 |
| WQ | 151.1 | 237.2 | 359.8 | 0.4 | 150.2 | 227.9 | 326.6 | 0.3 |
| TSS | 140.9 | 173.0 | 198.3 | 0.9 | 107.6 | 236.2 | 326.2 | 0.8 |
| FAC | 139.8 | 199.6 | 260.0 | 0.8 | 162.9 | 226.1 | 357.1 | 0.2 |
| S-WQR | 179.4 | 301.7 | 424.7 | 0.1 | 234.0 | 296.8 | 403.6 | 0.1 |
| TO-WQR | 138.7 | 190.7 | 233.1 | 0.2 | 189.9 | 259.2 | 351.6 | 0.1 |

Tabelle C.20.: FEIN-100 - mehrere Master - homogene Umgebung. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | FIFO Worker | | | | Round Robin Worker | | | |
|---|---|---|---|---|---|---|---|---|
| | Makespan | | | Leer- | Makespan | | | Leer- |
| | min | average | max | lauf | min | average | max | lauf |
| FIXED | 115.2 | 186.2 | 238.9 | 36.6 | 55.3 | 117.5 | 248.0 | 26.3 |
| FIXED-W | 46.5 | 60.3 | 74.4 | 10.8 | 32.9 | 45.2 | 76.4 | 12.0 |
| RANDOM | 54.0 | 210.0 | 302.5 | 51.1 | 37.7 | 123.5 | 244.3 | 33.5 |
| RANDOM-W | 39.5 | 69.6 | 102.6 | 15.1 | 36.6 | 58.1 | 109.6 | 10.8 |
| MCT | 29.8 | 56.4 | 82.5 | 8.0 | 32.1 | 51.9 | 90.3 | 4.9 |
| MINMAX | 33.6 | 62.6 | 103.7 | 8.9 | 27.1 | 50.5 | 100.1 | 6.4 |
| MINMIN | 32.5 | 55.8 | 89.5 | 7.0 | 32.9 | 53.7 | 102.9 | 6.2 |
| WQ | 36.3 | 78.9 | 137.4 | 8.3 | 35.2 | 80.8 | 174.0 | 6.7 |
| TSS | 38.7 | 93.0 | 159.3 | 14.2 | 42.9 | 82.6 | 137.0 | 9.1 |
| FAC | 46.5 | 75.7 | 112.4 | 9.7 | 41.1 | 72.7 | 119.9 | 6.2 |
| S-WQR | 31.0 | 46.2 | 66.1 | 0.1 | 34.1 | 53.5 | 75.0 | 0.1 |
| TO-WQR | 41.6 | 53.5 | 69.3 | 0.7 | 37.9 | 51.6 | 70.6 | 1.4 |
| CENTRAL | 33.3 | 60.0 | 129.5 | — | — | — | — | — |

Tabelle C.21.: GROB-10 - mehrere Master - heterogene Umgebung. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | FIFO Worker | | | | Round Robin Worker | | | |
|---|---|---|---|---|---|---|---|---|
| | Makespan | | | Leer- | Makespan | | | Leer- |
| | min | average | max | lauf | min | average | max | lauf |
| FIXED | 76.0 | 155.7 | 233.6 | 21.7 | 33.0 | 99.8 | 242.9 | 30.9 |
| FIXED-W | 38.9 | 53.0 | 80.5 | 9.8 | 22.2 | 35.9 | 45.4 | 11.4 |
| RANDOM | 21.4 | 138.5 | 329.9 | 32.7 | 43.4 | 98.6 | 212.4 | 23.6 |
| RANDOM-W | 39.9 | 58.6 | 79.7 | 11.9 | 24.7 | 46.7 | 83.8 | 10.1 |
| MCT | 22.7 | 60.0 | 121.6 | 11.6 | 23.8 | 60.3 | 160.8 | 9.9 |
| MINMAX | 19.4 | 49.1 | 127.8 | 9.1 | 21.5 | 54.0 | 170.8 | 5.0 |
| MINMIN | 23.3 | 57.6 | 127.5 | 10.7 | 23.8 | 51.4 | 158.0 | 7.9 |
| WQ | 32.0 | 74.1 | 134.4 | 18.3 | 26.5 | 72.7 | 165.8 | 9.6 |
| TSS | 32.2 | 67.2 | 119.8 | 9.9 | 23.2 | 66.0 | 119.3 | 5.2 |
| FAC | 37.6 | 56.3 | 110.5 | 4.1 | 35.6 | 69.2 | 116.3 | 5.5 |
| S-WQR | 27.5 | 38.3 | 54.2 | 0.0 | 25.7 | 39.2 | 64.4 | 0.0 |
| TO-WQR | 31.9 | 43.8 | 58.4 | 0.3 | 30.9 | 36.2 | 44.1 | 0.1 |
| CENTRAL | 23.7 | 47.1 | 73.9 | — | — | — | — | — |

Tabelle C.22.: GROB-11 - mehrere Master - heterogene Umgebung. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | FIFO Worker | | | | Round Robin Worker | | | |
|---|---|---|---|---|---|---|---|---|
| | Makespan | | | Leer- | Makespan | | | Leer- |
| | min | average | max | lauf | min | average | max | lauf |
| FIXED | 102.8 | 170.2 | 229.5 | 36.5 | 128.2 | 249.7 | 384.5 | 28.9 |
| FIXED-W | 42.2 | 68.3 | 85.0 | 9.8 | 52.1 | 82.1 | 131.9 | 9.5 |
| RANDOM | 97.0 | 199.6 | 306.5 | 54.3 | 53.2 | 227.5 | 436.8 | 22.2 |
| RANDOM-W | 30.6 | 63.4 | 97.0 | 10.4 | 45.8 | 75.3 | 169.7 | 10.3 |
| MCT | 27.8 | 56.2 | 102.3 | 8.5 | 30.9 | 58.0 | 83.2 | 6.4 |
| MINMAX | 27.6 | 58.4 | 143.1 | 4.5 | 45.6 | 64.1 | 86.7 | 6.9 |
| MINMIN | 31.1 | 56.3 | 119.7 | 4.7 | 28.7 | 56.6 | 99.8 | 5.8 |
| WQ | 36.4 | 65.4 | 129.6 | 15.4 | 25.3 | 53.4 | 84.4 | 4.4 |
| TSS | 28.6 | 69.3 | 124.4 | 14.9 | 33.0 | 60.8 | 96.2 | 4.1 |
| FAC | 40.2 | 63.0 | 97.3 | 6.7 | 25.9 | 54.3 | 104.5 | 5.0 |
| S-WQR | 39.3 | 85.2 | 122.1 | 0.1 | 48.1 | 83.0 | 118.4 | 0.0 |
| TO-WQR | 44.7 | 60.0 | 73.7 | 0.4 | 48.8 | 71.1 | 99.4 | 0.1 |
| CENTRAL | 23.4 | 36.2 | 53.0 | — | — | — | — | — |

Tabelle C.23.: FEIN-80 - mehrere Master - heterogene Umgebung. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | FIFO Worker | | | | Round Robin Worker | | | |
|---|---|---|---|---|---|---|---|---|
| | Makespan | | | Leer- | Makespan | | | Leer- |
| | min | average | max | lauf | min | average | max | lauf |
| FIXED | 150.6 | 205.0 | 268.9 | 40.7 | 163.8 | 277.4 | 412.8 | 31.9 |
| FIXED-W | 38.6 | 60.9 | 89.1 | 14.2 | 47.4 | 88.0 | 135.9 | 10.2 |
| RANDOM | 108.8 | 202.0 | 274.9 | 33.8 | 106.5 | 280.8 | 552.6 | 33.2 |
| RANDOM-W | 32.9 | 54.3 | 88.3 | 7.2 | 36.5 | 81.8 | 152.6 | 10.4 |
| MCT | 38.0 | 60.5 | 112.3 | 4.9 | 42.3 | 74.5 | 136.8 | 9.6 |
| MINMAX | 31.1 | 58.3 | 94.7 | 6.7 | 41.6 | 70.2 | 109.0 | 8.8 |
| MINMIN | 29.1 | 60.0 | 112.5 | 6.1 | 32.7 | 67.2 | 122.4 | 7.0 |
| WQ | 26.9 | 74.3 | 166.1 | 10.2 | 26.1 | 59.1 | 133.5 | 7.2 |
| TSS | 33.9 | 60.8 | 119.7 | 9.7 | 32.7 | 63.9 | 110.0 | 4.8 |
| FAC | 35.2 | 49.6 | 76.8 | 5.7 | 33.8 | 66.0 | 115.5 | 8.7 |
| S-WQR | 68.8 | 96.6 | 163.9 | 0.3 | 61.0 | 102.6 | 149.3 | 0.2 |
| TO-WQR | 42.8 | 63.6 | 83.2 | 0.8 | 41.1 | 71.3 | 107.6 | 0.5 |
| CENTRAL | 24.6 | 46.4 | 69.8 | — | — | — | — | — |

Tabelle C.24.: FEIN-100 - mehrere Master - heterogene Umgebung. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | 30% Last | | | 100% Last | | |
|---|---|---|---|---|---|---|
| — | average | Standard- | average | average | Standard- | average |
| — | Makespan | abweichung | Leerlauf | Makespan | abweichung | Leerlauf |
| FIXED | 241.4 | 59.8 | 45.5 | 289.0 | 67.1 | 61.6 |
| FIXED-W | 76.9 | 14.7 | 14.3 | 97.0 | 8.2 | 18.8 |
| RANDOM | 242.1 | 107.2 | 65.7 | 262.4 | 99.1 | 69.8 |
| RANDOM-W | 78.7 | 15.9 | 12.1 | 107.2 | 22.9 | 15.4 |
| MCT | 66.0 | 19.4 | 7.0 | 97.0 | 45.9 | 15.7 |
| MINMAX | 89.9 | 42.3 | 12.6 | 107.7 | 51.1 | 14.2 |
| MINMIN | 68.6 | 16.7 | 7.7 | 96.3 | 32.0 | 11.3 |
| WQ | 98.7 | 32.8 | 12.5 | 122.4 | 48.3 | 19.6 |
| TSS | 96.8 | 35.9 | 15.1 | 109.4 | 29.6 | 16.7 |
| FAC | 99.9 | 37.8 | 11.7 | 118.9 | 36.5 | 21.9 |
| S-WQR | 53.7 | 10.0 | 0.1 | 66.1 | 15.6 | 0.1 |
| TO-WQR | 64.8 | 9.5 | 0.7 | 85.4 | 11.8 | 1.2 |
| CENTRAL | 67.3 | 25.2 | — | 99.8 | 34.3 | — |

Tabelle C.25.: GROB-10 - mehrere Master - heterogene Umgebung - externe Last auf den Rechnern - FIFO Worker. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | 30% Last | | | 100% Last | | |
|---|---|---|---|---|---|---|
| — | average | Standard- | average | average | Standard- | average |
| — | Makespan | abweichung | Leerlauf | Makespan | abweichung | Leerlauf |
| FIXED | 220.3 | 34.7 | 45.2 | 278.5 | 64.9 | 45.4 |
| FIXED-W | 73.3 | 15.5 | 19.5 | 92.1 | 16.4 | 18.9 |
| RANDOM | 217.2 | 100.4 | 44.7 | 170.7 | 93.8 | 32.5 |
| RANDOM-W | 63.0 | 12.8 | 13.4 | 82.8 | 22.2 | 11.7 |
| MCT | 67.0 | 38.3 | 9.9 | 91.0 | 76.4 | 11.4 |
| MINMAX | 69.9 | 35.1 | 11.2 | 64.9 | 24.4 | 8.1 |
| MINMIN | 70.2 | 26.7 | 11.2 | 94.3 | 59.0 | 19.1 |
| WQ | 81.5 | 30.0 | 10.3 | 121.4 | 37.8 | 14.6 |
| TSS | 86.7 | 39.2 | 14.2 | 83.3 | 32.9 | 14.8 |
| FAC | 92.2 | 44.0 | 17.6 | 109.5 | 49.7 | 9.1 |
| S-WQR | 44.4 | 13.2 | 0.1 | 60.4 | 15.8 | 0.0 |
| TO-WQR | 53.0 | 9.1 | 0.4 | 64.7 | 12.0 | 0.3 |
| CENTRAL | 65.2 | 26.9 | — | 71.0 | 17.4 | — |

Tabelle C.26.: GROB-11 - mehrere Master - heterogene Umgebung - externe Last auf den Rechnern - FIFO Worker. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | 30% Last | | | 100% Last | | |
|---|---|---|---|---|---|---|
| — | average | Standard- | average | average | Standard- | average |
| — | Makespan | abweichung | Leerlauf | Makespan | abweichung | Leerlauf |
| FIXED | 229.9 | 38.4 | 55.8 | 318.7 | 50.4 | 70.5 |
| FIXED-W | 80.8 | 13.6 | 15.4 | 94.6 | 8.8 | 18.2 |
| RANDOM | 208.1 | 59.1 | 43.9 | 298.2 | 57.6 | 52.1 |
| RANDOM-W | 64.9 | 14.0 | 7.0 | 88.8 | 23.9 | 10.8 |
| MCT | 62.0 | 14.4 | 3.8 | 77.6 | 40.8 | 7.3 |
| MINMAX | 66.5 | 20.3 | 8.5 | 79.8 | 20.2 | 2.5 |
| MINMIN | 60.8 | 14.6 | 6.1 | 83.3 | 30.6 | 10.6 |
| WQ | 83.4 | 23.6 | 13.2 | 99.7 | 34.4 | 16.6 |
| TSS | 75.4 | 18.1 | 13.7 | 107.9 | 43.7 | 16.4 |
| FAC | 78.0 | 22.7 | 13.5 | 100.4 | 38.3 | 15.6 |
| S-WQR | 85.7 | 22.0 | 0.1 | 128.8 | 33.1 | 0.1 |
| TO-WQR | 71.4 | 10.6 | 0.4 | 91.8 | 12.3 | 0.5 |
| CENTRAL | 40.0 | 8.3 | — | 54.1 | 14.1 | — |

Tabelle C.27.: FEIN-80 - mehrere Master - heterogene Umgebung - externe Last auf den Rechnern - FIFO Worker. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | 30% Last | | | 100% Last | | |
|---|---|---|---|---|---|---|
| — | average | Standard- | average | average | Standard- | average |
| — | Makespan | abweichung | Leerlauf | Makespan | abweichung | Leerlauf |
| FIXED | 230.3 | 39.3 | 47.1 | 308.8 | 58.7 | 75.0 |
| FIXED-W | 68.8 | 17.1 | 9.6 | 84.3 | 14.3 | 16.1 |
| RANDOM | 262.0 | 71.4 | 54.4 | 276.7 | 83.0 | 53.7 |
| RANDOM-W | 65.5 | 14.7 | 9.7 | 79.9 | 20.8 | 11.1 |
| MCT | 60.9 | 18.9 | 4.6 | 88.6 | 39.1 | 6.0 |
| MINMAX | 68.3 | 23.8 | 8.0 | 91.2 | 38.1 | 7.3 |
| MINMIN | 70.4 | 18.7 | 7.1 | 89.6 | 38.0 | 6.1 |
| WQ | 70.1 | 19.2 | 13.1 | 91.9 | 29.7 | 10.5 |
| TSS | 77.7 | 25.4 | 15.4 | 91.6 | 35.6 | 7.8 |
| FAC | 76.1 | 30.3 | 10.3 | 100.2 | 33.3 | 13.9 |
| S-WQR | 118.1 | 26.4 | 0.2 | 141.2 | 28.5 | 0.1 |
| TO-WQR | 72.8 | 11.8 | 0.4 | 97.8 | 17.3 | 0.9 |
| CENTRAL | 56.5 | 18.5 | — | 71.2 | 16.5 | — |

Tabelle C.28.: FEIN-100 - mehrere Master - heterogene Umgebung - externe Last auf den Rechnern - FIFO Worker. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | 30% Last | | | 100% Last | | |
|---|---|---|---|---|---|---|
| — — | average Makespan | Standard-abweichung | average Leerlauf | average Makespan | Standard-abweichung | average Leerlauf |
| FIXED | 128.5 | 48.8 | 61.8 | 289.0 | 67.1 | 61.6 |
| FIXED-W | 58.6 | 15.8 | 20.2 | 98.8 | 7.1 | 17.0 |
| RANDOM | 157.7 | 93.6 | 64.7 | 262.4 | 99.1 | 69.8 |
| RANDOM-W | 64.1 | 12.0 | 12.4 | 107.1 | 22.4 | 14.8 |
| MCT | 74.0 | 36.3 | 8.9 | 97.0 | 45.9 | 15.7 |
| MINMAX | 69.0 | 24.8 | 9.7 | 107.7 | 51.1 | 14.2 |
| MINMIN | 64.4 | 28.1 | 7.3 | 96.3 | 32.0 | 11.3 |
| WQ | 81.2 | 26.8 | 7.8 | 124.2 | 48.6 | 17.9 |
| TSS | 98.0 | 36.6 | 15.0 | 110.8 | 29.5 | 17.8 |
| FAC | 85.2 | 36.7 | 12.1 | 120.3 | 36.7 | 19.9 |
| S-WQR | 51.9 | 9.0 | 0.0 | 66.7 | 15.7 | 0.0 |
| TO-WQR | 56.8 | 11.6 | 0.8 | 85.4 | 11.8 | 1.2 |

Tabelle C.29.: GROB-10 - mehrere Master - heterogene Umgebung - externe Last auf den Rechnern - Round Robin Worker. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | 30% Last | | | 100% Last | | |
|---|---|---|---|---|---|---|
| — — | average Makespan | Standard-abweichung | average Leerlauf | average Makespan | Standard-abweichung | average Leerlauf |
| FIXED | 122.2 | 43.5 | 56.6 | 278.5 | 64.9 | 45.4 |
| FIXED-W | 48.8 | 10.4 | 13.7 | 92.1 | 16.4 | 18.9 |
| RANDOM | 145.3 | 99.8 | 63.0 | 172.6 | 95.5 | 31.8 |
| RANDOM-W | 52.4 | 14.2 | 7.9 | 82.8 | 22.2 | 11.7 |
| MCT | 56.0 | 35.5 | 7.9 | 91.0 | 76.4 | 11.4 |
| MINMAX | 70.8 | 39.7 | 16.2 | 63.6 | 24.6 | 8.1 |
| MINMIN | 67.2 | 31.6 | 9.8 | 94.3 | 59.0 | 19.1 |
| WQ | 83.9 | 40.0 | 12.8 | 121.4 | 37.8 | 14.6 |
| TSS | 73.9 | 40.8 | 12.4 | 84.2 | 33.4 | 15.5 |
| FAC | 74.5 | 34.6 | 12.3 | 111.0 | 50.5 | 9.6 |
| S-WQR | 43.4 | 10.2 | 0.0 | 59.1 | 15.6 | 0.0 |
| TO-WQR | 46.4 | 11.1 | 0.7 | 65.3 | 11.9 | 0.3 |

Tabelle C.30.: GROB-11 - mehrere Master - heterogene Umgebung - externe Last auf den Rechnern - Round Robin Worker. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | 30% Last | | | 100% Last | | |
|---|---|---|---|---|---|---|
| — — | average Makespan | Standard-abweichung | average Leerlauf | average Makespan | Standard-abweichung | average Leerlauf |
| FIXED | 328.1 | 94.3 | 36.4 | 318.7 | 50.4 | 70.5 |
| FIXED-W | 83.1 | 21.1 | 20.3 | 94.6 | 8.8 | 18.2 |
| RANDOM | 325.3 | 118.8 | 47.5 | 298.2 | 57.6 | 52.1 |
| RANDOM-W | 91.9 | 24.6 | 16.5 | 88.8 | 23.9 | 10.8 |
| MCT | 73.0 | 21.9 | 9.4 | 77.6 | 40.8 | 7.3 |
| MINMAX | 85.1 | 27.4 | 10.0 | 79.8 | 20.2 | 2.5 |
| MINMIN | 72.2 | 15.9 | 10.1 | 82.8 | 29.9 | 10.1 |
| WQ | 79.4 | 25.7 | 11.2 | 99.7 | 34.4 | 16.6 |
| TSS | 76.9 | 30.8 | 13.1 | 107.9 | 43.7 | 16.4 |
| FAC | 71.6 | 22.3 | 8.7 | 101.2 | 38.9 | 16.5 |
| S-WQR | 98.1 | 24.0 | 0.0 | 128.8 | 33.1 | 0.1 |
| TO-WQR | 70.4 | 17.5 | 0.2 | 91.8 | 12.3 | 0.5 |

Tabelle C.31.: FEIN-80 - mehrere Master - heterogene Umgebung - externe Last auf den Rechnern - Round Robin Worker. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | 30% Last | | | 100% Last | | |
|---|---|---|---|---|---|---|
| — — | average Makespan | Standard-abweichung | average Leerlauf | average Makespan | Standard-abweichung | average Leerlauf |
| FIXED | 357.8 | 87.7 | 58.5 | 308.8 | 58.7 | 75.0 |
| FIXED-W | 113.8 | 25.3 | 25.5 | 84.3 | 14.3 | 16.1 |
| RANDOM | 333.0 | 151.0 | 77.7 | 276.7 | 83.0 | 53.7 |
| RANDOM-W | 91.2 | 29.6 | 14.0 | 79.9 | 20.8 | 11.1 |
| MCT | 74.5 | 17.9 | 9.5 | 88.6 | 39.1 | 6.0 |
| MINMAX | 76.6 | 22.7 | 16.4 | 89.5 | 38.0 | 7.1 |
| MINMIN | 81.3 | 27.9 | 11.4 | 90.3 | 38.7 | 6.2 |
| WQ | 66.7 | 23.2 | 9.9 | 91.9 | 29.7 | 10.5 |
| TSS | 84.3 | 46.0 | 8.8 | 87.3 | 33.3 | 8.1 |
| FAC | 64.1 | 17.9 | 7.8 | 101.0 | 33.9 | 14.6 |
| S-WQR | 115.3 | 27.9 | 0.2 | 143.1 | 29.2 | 0.1 |
| TO-WQR | 91.1 | 20.9 | 0.5 | 97.8 | 17.3 | 0.9 |

Tabelle C.32.: FEIN-100 - mehrere Master - heterogene Umgebung - externe Last auf den Rechnern - Round Robin Worker. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | Makespan | | | Transfer- | Rechnen- | Rechner- |
|---|---|---|---|---|---|---|
| | min | average | max | overhead | overhead | ausfälle |
| FIXED | 55.7 | 116.8 | 203.7 | 4.4 | 1.2 | 5.3 |
| FIXED-W | 65.3 | 109.8 | 211.2 | 5.1 | 1.3 | 5.9 |
| RANDOM | 49.7 | 150.4 | 329.0 | 3.1 | 1.2 | 7.1 |
| RANDOM-W | 47.1 | 81.8 | 156.4 | 1.8 | 0.5 | 3.4 |
| MCT | 38.8 | 73.5 | 173.0 | 1.8 | 0.8 | 3.0 |
| MINMAX | 34.7 | 73.2 | 208.0 | 2.0 | 0.7 | 3.5 |
| MINMIN | 30.6 | 73.9 | 154.7 | 2.6 | 1.0 | 3.3 |
| WQ | 41.2 | 74.3 | 184.3 | 1.1 | 0.8 | 3.3 |
| TSS | 47.5 | 77.2 | 158.6 | 1.4 | 0.7 | 3.3 |
| FAC | 50.4 | 84.2 | 176.6 | 1.6 | 0.8 | 3.7 |
| S-WQR | 37.6 | 51.3 | 80.6 | 11.3 | 6.7 | 2.7 |
| TO-WQR | 45.7 | 72.5 | 133.4 | 6.3 | 2.9 | 3.6 |
| CENTRAL | 36.0 | 72.7 | 147.1 | — | — | — |

Tabelle C.33.: GROB-10 - mehrere Master - heterogene Umgebung, Ausfall der Worker möglich - FIFO Worker. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden. Der Overhead und die Ausfälle sind in der absoluten Taskanzahl angegeben.

| Algo | Makespan | | | Transfer- | Rechnen- | Rechner- |
|---|---|---|---|---|---|---|
| | min | average | max | overhead | overhead | ausfälle |
| FIXED | 54.5 | 133.0 | 217.0 | 4.4 | 1.3 | 7.9 |
| FIXED-W | 31.6 | 66.6 | 120.4 | 2.1 | 0.7 | 3.1 |
| RANDOM | 40.5 | 146.7 | 267.9 | 3.5 | 1.1 | 6.7 |
| RANDOM-W | 38.6 | 71.6 | 92.9 | 3.1 | 0.9 | 4.1 |
| MCT | 24.2 | 62.4 | 127.2 | 3.3 | 0.7 | 3.4 |
| MINMAX | 20.3 | 74.1 | 200.6 | 3.4 | 1.1 | 3.5 |
| MINMIN | 32.8 | 70.3 | 134.6 | 2.1 | 0.6 | 3.9 |
| WQ | 47.2 | 76.9 | 130.6 | 1.3 | 0.9 | 3.1 |
| TSS | 43.4 | 77.3 | 152.9 | 2.2 | 0.9 | 4.2 |
| FAC | 35.4 | 77.0 | 130.7 | 1.3 | 0.7 | 3.9 |
| S-WQR | 27.3 | 41.6 | 57.4 | 11.1 | 5.1 | 1.9 |
| TO-WQR | 24.9 | 47.8 | 99.7 | 6.2 | 2.6 | 2.6 |
| CENTRAL | 27.8 | 51.5 | 124.0 | — | — | — |

Tabelle C.34.: GROB-11 - mehrere Master - heterogene Umgebung, Ausfall der Worker möglich - FIFO Worker. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden. Der Overhead und die Ausfälle sind in der absoluten Taskanzahl angegeben.

| Algo | Makespan | | | Transfer- | Rechnen- | Rechner- |
|---|---|---|---|---|---|---|
| | min | average | max | overhead | overhead | ausfälle |
| FIXED | 48.8 | 127.6 | 217.5 | 24.0 | 1.3 | 6.0 |
| FIXED-W | 29.9 | 74.8 | 181.1 | 19.1 | 1.1 | 4.1 |
| RANDOM | 55.7 | 155.5 | 285.2 | 13.8 | 1.2 | 5.8 |
| RANDOM-W | 41.2 | 77.1 | 127.3 | 20.3 | 1.3 | 3.7 |
| MCT | 41.9 | 69.7 | 139.4 | 18.6 | 1.4 | 4.1 |
| MINMAX | 37.4 | 69.3 | 135.7 | 12.3 | 1.4 | 3.5 |
| MINMIN | 30.9 | 63.1 | 114.2 | 17.2 | 1.0 | 3.1 |
| WQ | 35.7 | 98.2 | 321.1 | 1.8 | 1.0 | 4.2 |
| TSS | 29.7 | 71.6 | 104.1 | 3.7 | 1.0 | 3.1 |
| FAC | 40.4 | 67.2 | 95.0 | 7.0 | 1.0 | 2.9 |
| S-WQR | 55.5 | 88.0 | 150.7 | 13.8 | 4.2 | 3.9 |
| TO-WQR | 39.8 | 61.8 | 108.6 | 15.0 | 2.4 | 3.0 |
| CENTRAL | 29.2 | 50.6 | 88.5 | — | — | — |

Tabelle C.35.: FEIN-80 - mehrere Master - heterogene Umgebung, Ausfall der Worker möglich - FIFO Worker. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden. Der Overhead und die Ausfälle sind in der absoluten Taskanzahl angegeben.

| Algo | Makespan | | | Transfer- | Rechnen- | Rechner- |
|---|---|---|---|---|---|---|
| | min | average | max | overhead | overhead | ausfälle |
| FIXED | 60.3 | 152.1 | 237.3 | 29.2 | 2.5 | 8.3 |
| FIXED-W | 39.0 | 75.5 | 144.1 | 17.8 | 1.0 | 4.1 |
| RANDOM | 59.1 | 167.6 | 315.5 | 19.4 | 1.2 | 7.0 |
| RANDOM-W | 40.6 | 81.0 | 177.4 | 12.2 | 1.3 | 3.6 |
| MCT | 29.6 | 73.0 | 155.8 | 14.5 | 1.3 | 3.0 |
| MINMAX | 31.3 | 62.1 | 129.8 | 17.8 | 0.8 | 3.0 |
| MINMIN | 36.3 | 71.8 | 159.0 | 20.9 | 1.0 | 3.6 |
| WQ | 44.4 | 80.4 | 149.0 | 2.4 | 1.2 | 4.1 |
| TSS | 35.0 | 68.0 | 121.5 | 6.3 | 1.0 | 3.1 |
| FAC | 38.1 | 68.4 | 129.0 | 7.7 | 1.5 | 3.3 |
| S-WQR | 72.9 | 102.3 | 178.0 | 14.7 | 4.6 | 4.5 |
| TO-WQR | 39.6 | 70.7 | 105.8 | 15.6 | 2.6 | 4.1 |
| CENTRAL | 24.0 | 39.0 | 60.2 | — | — | — |

Tabelle C.36.: FEIN-100 - mehrere Master - heterogene Umgebung, Ausfall der Worker möglich - FIFO Worker. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden. Der Overhead und die Ausfälle sind in der absoluten Taskanzahl angegeben.

| Algo | Makespan | | | Transfer- | Rechnen- | Rechner- |
|---|---|---|---|---|---|---|
| | min | average | max | overhead | overhead | ausfälle |
| FIXED | 44.2 | 127.4 | 259.5 | 2.7 | 1.5 | 5.9 |
| FIXED-W | 35.7 | 56.7 | 87.5 | 2.5 | 1.1 | 2.3 |
| RANDOM | 41.2 | 108.5 | 285.5 | 1.9 | 1.0 | 3.9 |
| RANDOM-W | 35.1 | 68.0 | 157.0 | 2.0 | 1.0 | 3.4 |
| MCT | 38.7 | 67.8 | 137.5 | 2.3 | 1.0 | 3.6 |
| MINMAX | 34.0 | 66.6 | 119.8 | 2.5 | 1.2 | 3.6 |
| MINMIN | 35.5 | 74.9 | 179.5 | 2.8 | 1.0 | 3.8 |
| WQ | 43.3 | 78.5 | 176.7 | 1.3 | 1.1 | 3.1 |
| TSS | 42.7 | 90.3 | 183.2 | 1.7 | 1.0 | 5.4 |
| FAC | 33.3 | 80.6 | 123.1 | 1.9 | 1.4 | 3.5 |
| S-WQR | 36.8 | 50.4 | 68.2 | 10.7 | 7.0 | 2.2 |
| TO-WQR | 41.9 | 66.8 | 123.8 | 5.5 | 2.9 | 3.6 |

Tabelle C.37.: GROB-10 - mehrere Master - heterogene Umgebung, Ausfall der Worker möglich - Round Robin Worker. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden. Der Overhead und die Ausfälle sind in der absoluten Taskanzahl angegeben.

| Algo | Makespan | | | Transfer- | Rechnen- | Rechner- |
|---|---|---|---|---|---|---|
| | min | average | max | overhead | overhead | ausfälle |
| FIXED | 39.8 | 96.9 | 342.1 | 2.9 | 1.0 | 5.0 |
| FIXED-W | 29.7 | 43.2 | 68.9 | 1.3 | 0.7 | 2.3 |
| RANDOM | 34.5 | 106.9 | 191.2 | 2.8 | 1.0 | 5.0 |
| RANDOM-W | 24.3 | 49.0 | 92.8 | 1.6 | 0.7 | 2.6 |
| MCT | 25.4 | 66.1 | 160.4 | 2.7 | 1.0 | 2.9 |
| MINMAX | 20.4 | 61.8 | 166.2 | 1.9 | 0.7 | 2.8 |
| MINMIN | 30.4 | 72.5 | 137.7 | 2.2 | 0.8 | 3.7 |
| WQ | 28.2 | 79.7 | 157.6 | 1.5 | 1.0 | 3.5 |
| TSS | 44.0 | 87.6 | 173.0 | 1.9 | 1.1 | 5.1 |
| FAC | 24.1 | 81.5 | 187.1 | 0.9 | 0.6 | 3.5 |
| S-WQR | 22.6 | 38.4 | 54.7 | 10.7 | 6.0 | 2.2 |
| TO-WQR | 26.4 | 41.2 | 94.5 | 6.6 | 2.7 | 2.2 |

Tabelle C.38.: GROB-11 - mehrere Master - heterogene Umgebung, Ausfall der Worker möglich - Round Robin Worker. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden. Der Overhead und die Ausfälle sind in der absoluten Taskanzahl angegeben.

| Algo | Makespan | | | Transfer- | Rechnen- | Rechner- |
|---|---|---|---|---|---|---|
| | min | average | max | overhead | overhead | ausfälle |
| FIXED | 80.4 | 172.8 | 354.6 | 21.0 | 1.1 | 7.5 |
| FIXED-W | 47.8 | 99.5 | 174.1 | 18.6 | 1.0 | 5.0 |
| RANDOM | 68.8 | 148.6 | 341.9 | 22.4 | 1.3 | 6.6 |
| RANDOM-W | 57.4 | 85.5 | 159.4 | 11.5 | 0.7 | 4.3 |
| MCT | 29.3 | 73.3 | 156.2 | 21.5 | 1.3 | 3.5 |
| MINMAX | 36.4 | 73.9 | 145.9 | 14.2 | 0.7 | 3.6 |
| MINMIN | 39.5 | 68.1 | 109.3 | 17.0 | 1.0 | 3.3 |
| WQ | 27.6 | 91.0 | 228.2 | 2.6 | 1.4 | 5.1 |
| TSS | 33.2 | 79.8 | 246.6 | 5.6 | 0.6 | 4.1 |
| FAC | 38.7 | 77.7 | 301.1 | 6.3 | 1.0 | 4.0 |
| S-WQR | 48.0 | 83.2 | 120.7 | 14.3 | 4.0 | 4.0 |
| TO-WQR | 34.2 | 73.0 | 101.2 | 14.0 | 2.5 | 3.9 |

Tabelle C.39.: FEIN-80 - mehrere Master - heterogene Umgebung, Ausfall der Worker möglich - Round Robin Worker. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden. Der Overhead und die Ausfälle sind in der absoluten Taskanzahl angegeben.

| Algo | Makespan | | | Transfer- | Rechnen- | Rechner- |
|---|---|---|---|---|---|---|
| | min | average | max | overhead | overhead | ausfälle |
| FIXED | 51.4 | 145.9 | 231.9 | 17.5 | 1.5 | 6.0 |
| FIXED-W | 70.1 | 106.4 | 153.6 | 16.6 | 0.4 | 4.6 |
| RANDOM | 86.3 | 166.6 | 303.9 | 24.5 | 1.8 | 7.9 |
| RANDOM-W | 41.4 | 96.5 | 146.8 | 14.7 | 1.0 | 4.4 |
| MCT | 28.0 | 68.3 | 134.3 | 15.2 | 0.7 | 3.5 |
| MINMAX | 35.7 | 71.6 | 109.7 | 17.3 | 0.9 | 3.8 |
| MINMIN | 30.4 | 70.9 | 99.7 | 18.0 | 1.3 | 3.9 |
| WQ | 34.8 | 82.4 | 192.9 | 2.6 | 1.0 | 4.9 |
| TSS | 33.9 | 62.0 | 105.5 | 3.6 | 1.0 | 2.6 |
| FAC | 29.6 | 70.4 | 173.2 | 4.7 | 1.0 | 2.8 |
| S-WQR | 64.9 | 109.5 | 159.1 | 15.5 | 3.9 | 4.7 |
| TO-WQR | 37.0 | 63.9 | 93.1 | 12.7 | 3.2 | 3.1 |

Tabelle C.40.: FEIN-100 - mehrere Master - heterogene Umgebung, Ausfall der Worker möglich - Round Robin Worker. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden. Der Overhead und die Ausfälle sind in der absoluten Taskanzahl angegeben.

| Algo | Makespan | | | Leerlauf | | |
|---|---|---|---|---|---|---|
| — | min | average | max | min | average | max |
| FIXED | 44.6 | 64.0 | 118.3 | 2.0 | 35.4 | 54.4 |
| FIXED-W | 50.0 | 55.6 | 61.0 | 1.5 | 37.0 | 55.6 |
| RANDOM | 47.1 | 91.2 | 176.1 | 1.9 | 58.7 | 86.9 |
| RANDOM-W | 40.2 | 59.9 | 87.4 | 1.5 | 40.0 | 59.9 |
| MCT | 37.8 | 61.9 | 76.7 | 1.6 | 42.7 | 61.9 |
| MINMAX | 39.5 | 59.6 | 74.7 | 1.7 | 40.5 | 59.6 |
| MINMIN | 37.5 | 60.7 | 78.4 | 1.8 | 41.5 | 60.7 |
| WQ | 34.3 | 45.3 | 53.8 | 2.0 | 21.0 | 30.2 |
| TSS | 35.2 | 47.3 | 53.8 | 2.3 | 22.5 | 33.3 |
| FAC | 34.9 | 45.2 | 48.6 | 2.3 | 20.9 | 30.8 |
| S-WQR | 31.8 | 36.5 | 47.1 | 1.5 | 2.3 | 3.2 |
| TO-WQR | 35.2 | 45.4 | 49.0 | 2.1 | 18.9 | 29.3 |

Tabelle C.41.: GROB-10 - ein Master - heterogene Umgebung - ein verlangsamter Worker nach dem Start der Ausführung. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | Makespan | | | Leerlauf | | |
|---|---|---|---|---|---|---|
| — | min | average | max | min | average | max |
| FIXED | 33.9 | 47.9 | 97.1 | 1.5 | 27.9 | 45.5 |
| FIXED-W | 27.3 | 37.4 | 62.3 | 1.2 | 25.7 | 37.4 |
| RANDOM | 32.5 | 58.6 | 148.7 | 1.6 | 38.6 | 58.1 |
| RANDOM-W | 28.8 | 44.5 | 76.6 | 1.4 | 30.7 | 44.5 |
| MCT | 36.4 | 41.7 | 51.3 | 1.5 | 29.4 | 41.7 |
| MINMAX | 23.7 | 34.7 | 50.5 | 1.5 | 21.6 | 34.7 |
| MINMIN | 41.2 | 47.4 | 61.9 | 1.3 | 36.5 | 47.4 |
| WQ | 23.0 | 41.7 | 90.3 | 1.6 | 23.4 | 32.4 |
| TSS | 32.1 | 45.1 | 90.0 | 1.6 | 26.3 | 37.4 |
| FAC | 18.7 | 38.9 | 89.8 | 1.6 | 21.2 | 30.4 |
| S-WQR | 22.3 | 29.1 | 32.3 | 1.3 | 2.0 | 2.6 |
| TO-WQR | 25.0 | 34.9 | 45.2 | 1.8 | 12.1 | 20.7 |

Tabelle C.42.: GROB-11 - ein Master - heterogene Umgebung - ein verlangsamter Worker nach dem Start der Ausführung. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | Makespan | | | Leerlauf | | |
|---|---|---|---|---|---|---|
| — | min | average | max | min | average | max |
| FIXED | 66.0 | 69.4 | 73.9 | 1.1 | 39.6 | 59.4 |
| FIXED-W | 49.3 | 52.1 | 55.2 | 1.1 | 31.7 | 46.1 |
| RANDOM | 37.3 | 70.6 | 84.6 | 1.1 | 40.8 | 60.2 |
| RANDOM-W | 35.4 | 50.7 | 62.7 | 1.0 | 30.3 | 45.9 |
| MCT | 46.9 | 51.9 | 54.3 | 1.1 | 31.7 | 46.0 |
| MINMAX | 50.3 | 52.2 | 55.0 | 1.1 | 31.9 | 46.2 |
| MINMIN | 47.2 | 52.0 | 62.5 | 1.1 | 31.5 | 45.8 |
| WQ | 25.1 | 25.3 | 25.6 | 1.2 | 2.3 | 3.0 |
| TSS | 25.1 | 25.4 | 27.7 | 1.2 | 2.4 | 3.2 |
| FAC | 25.2 | 26.8 | 40.1 | 1.2 | 3.6 | 5.0 |
| S-WQR | 24.7 | 25.7 | 26.9 | 0.9 | 1.5 | 1.7 |
| TO-WQR | 24.6 | 25.6 | 26.7 | 1.3 | 2.6 | 3.6 |

Tabelle C.43.: FEIN-80 - ein Master - heterogene Umgebung - ein verlangsamter Worker nach dem Start der Ausführung. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

| Algo | Makespan | | | Leerlauf | | |
|---|---|---|---|---|---|---|
| — | min | average | max | min | average | max |
| FIXED | 45.2 | 80.7 | 109.2 | 1.4 | 50.4 | 73.2 |
| FIXED-W | 33.6 | 56.1 | 79.0 | 4.4 | 37.6 | 54.2 |
| RANDOM | 33.9 | 73.8 | 151.2 | 1.6 | 44.8 | 67.5 |
| RANDOM-W | 30.7 | 44.3 | 63.1 | 1.7 | 25.6 | 39.4 |
| MCT | 36.9 | 44.9 | 53.4 | 2.2 | 26.7 | 41.0 |
| MINMAX | 39.9 | 47.4 | 57.3 | 2.7 | 30.4 | 45.7 |
| MINMIN | 41.2 | 47.5 | 55.9 | 3.0 | 29.4 | 43.3 |
| WQ | 28.3 | 41.1 | 56.3 | 3.3 | 18.1 | 24.6 |
| TSS | 25.4 | 35.6 | 52.6 | 2.7 | 14.1 | 20.2 |
| FAC | 27.0 | 36.5 | 60.4 | 3.5 | 14.3 | 20.0 |
| S-WQR | 27.0 | 30.5 | 34.5 | 2.6 | 3.4 | 4.4 |
| TO-WQR | 27.3 | 32.7 | 43.1 | 2.4 | 5.7 | 10.5 |

Tabelle C.44.: FEIN-100 - ein Master - heterogene Umgebung - ein verlangsamter Worker nach dem Start der Ausführung. Die Ergebnisse sind die Laufzeiten der Tests in Sekunden.

# Literaturverzeichnis


[1] Agnar Aamodt and Enric Plaza. Case-based reasoning : Foundational issues, methodological variations, and system approaches. *AI Commun.*, 7(1):39–59, 1994.

[2] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Lasislau L. Bölöni, Muthucumara Maheswaran, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, Bin Yao, Debra Hensgen, and Richard F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *J. Parallel Distrib. Comput.*, 61(6):810–837, 2001.

[3] James Bruno, Edward G. Coffman Jr., and Ravi Sethi. Scheduling independent tasks to reduce mean finishing time. *Commun. ACM*, 17(7):382–387, 1974.

[4] Henri Casanova, Dmitrii Zagorodnov, Francine Berman, and Arnaud Legrand. Heuristics for scheduling parameter sweep applications in grid environments. In *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*, page 349, Washington, DC, USA, 2000. IEEE Computer Society.

[5] Anthony T. Chronopoulos, Satish Penmatsa, and Ning Yu. Scalable loop self-scheduling schemes for heterogeneous clusters. In *CLUSTER '02: Proceedings of the IEEE International Conference on Cluster Computing*, pages 353–359, Washington, DC, USA, 2002. IEEE Computer Society.

[6] Jennifer Hardy Allan Snavely Cynthia Lee, Yael Schartzman. Are user runtime estimates inherently inaccurate? In *10th Workshop on Job Scheduling Strategies for Parallel Processing*, 2004.

[7] Daniel P. da Silva, Walfredo Cirne, and Francisco V. Brasileiro. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In *Euro-Par*, pages 169–180, 2003.

[8] Dror G. Feitelson, Dan Tsafrir, and Yoav Etsion. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Trans. Parallel Distrib. Syst.*, 18(6):789–803, 2007.

[9] Ian Foster and Carl Kesselman. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, November 1998.

[10] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2150, 2001.

[11] Eitan Frachtenberg and Dror G. Feitelson. Pitfalls in parallel job scheduling evaluation. In *JSSPP*, pages 257–282, 2005.

[12] William J. Frawley, Gregory Piatetsky-Shapiro, and Christopher J. Matheus. Knowledge discovery in databases - an overview. *Ai Magazine*, 13:57–70, 1992.

[13] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steve Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5:237–246, 2002.

[14] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The Elements of Statistical Learning*. Springer, August 2001.

[15] Susan F. Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: a method for scheduling parallel loops. *Commun. ACM*, 35(8):90–101, 1992.

[16] SeongKi Kim and Sang-Yong Han. Performance comparison of dcom, corba and web service. In *PDPTA*, pages 106–112, 2006.

[17] Barbara Kreaseck, Henri Casanova, Larry Carter, Jeanne Ferrante, and Sagnik Nandy. Interference aware scheduling. *International Journal of High Performance Computing Applications*, 20(1):45–59, February 2006.

[18] Clyde P. Kruskal and Alan Weiss. Allocating independent subtasks on parallel processors. *IEEE Trans. Softw. Eng.*, 11(10):1001–1016, 1985.

[19] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[20] James Macqueen. Some methods of classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathemtical Statistics and Probability*, pages 281–297, 1967.

[21] Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra Hensgen, and Richard F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *HCW '99: Proceedings of the Eighth Heterogeneous Computing Workshop*, pages 30–44, Washington, DC, USA, 1999. IEEE Computer Society.

[22] Ingo Mierswa, Michael Wurst, Ralf Klinkenberg, Martin Scholz, and Timm Euler. Yale (now: RapidMiner): rapid prototyping for complex data mining tasks. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 935–940, New York, NY, USA, 2006. ACM.

[23] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Trans. Parallel Distrib. Syst.*, 12(6):529–543, 2001.

[24] Michael Pinedo. *Scheduling - Theory, Algorithms, and Systems Second Edition.* Prentice Hall, 2002.

[25] Constantine D. Polychronopoulos and David J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. Comput.*, 36(12):1425–1439, 1987.

[26] Arno Puder and Kay Römer. *MiddleWare für verteilte Systeme.* dpunkt-Verl., 2001.

[27] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM Press.

[28] Hemant G. Rotithor. Taxonomy of dynamic task scheduling schemes in distributed computing systems. *IEE Proceedings - Computers and Digital Techniques*, 141(1):1–10, 1994.

[29] Gary Shao. *Adaptive Scheduling of Master/Worker Applications on Distributed Computational Resources.* PhD thesis, University of California at San Diego, May 2001.

[30] Heinz Stockinger. Defining the grid: a snapshot on the current view. *J. Supercomput.*, 42(1):3–17, 2007.

[31] Andrew S. Tanenbaum and Maarten van Steen. *Verteilte Systeme.* Pearson Studium, 2003.

[32] Peiyi Tang and Pen-Chung Yew. Processor self-scheduling for multiple-nested parallel loops. In *ICPP*, pages 528–535, 1986.

[33] Ten H. Tzen and Lionel M. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, 1993.

[34] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition.* Morgan Kaufmann, June 2005.

[35] Rich Wolski, Neil T. Spring, and Jim Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.

[36] Lingyun Yang, Jennifer M. Schopf, and Ian Foster. Conservative scheduling: Using predicted variance to improve scheduling decisions in dynamic environments. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 31, Washington, DC, USA, 2003. IEEE Computer Society.

[37] Yang Yang and Henri Casanova. Umr: A multi-round algorithm for scheduling divisible workloads. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 24.2, Washington, DC, USA, 2003. IEEE Computer Society.