Fernuniversität in Hagen
Fakultät für Mathematik und Informatik
Lehrgebiet Rechnerarchitektur

Bachelorarbeit im Fach Informatik

Tuning des Job Scheduling in heterogenen Systemen

01.05.2009 bis 03.08.2009

Jörg Lenhardt
Matrikelnummer: 6847811

Betreuer und Erstgutachter: Prof. Dr. W. Schiffmann
Zweitgutachter: Prof. Dr. J. Keller

Kurztext

Bei der Verteilung von Tasks in einer heterogenen Umgebung handelt es sich um ein Problem, dessen Lösung zum Beispiel für Grid Computing sehr wichtig ist [Ri07]. Das Finden optimaler Lösungen ist ein NP-vollständiges Problem. So werden Heuristiken eingesetzt, um in vertretbarer Zeit annehmbare Lösungen zu erhalten. Diese, durch die Heuristiken gefunden Lösungen können durch das Anwenden von Optimierungsstrategien oftmals noch weiter verbessert werden. In dieser Arbeit wird ein einfacher und schneller Optimierer vorgestellt, der eine Verbesserung der Laufzeiten verspricht.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

Abkürzungsverzeichnis

BLV	Base Line Vektor
bzw.	beziehungsweise
c	consistent, konsistent
cons	consistent, konsistent
CT	Completion Time
d. h.	das heißt
ET	Execution Time
et. al.	et alii, et aliae
ETC-Matrix	Expected Time to Compute Matrix
GA	Genetic Algorithm
GB	Giga Byte
GHz	Giga Hertz
GSA	Genetic Simulated Annealing
hM	hohe Maschinenheterogenität
hT	hohe Taskheterogenität
i	inconsistent, inkonsistent
i. A.	im Allgemeinen
IEEE	Institute of Electrical and Electronics Engineers
incons	inconsistent, inkonsistent
IV-Matrix	Init Value Matrix
LFS	Linux From Scratch
lM	geringe Machinenheterogenität
lT	geringe Taskheterogenität
MCT	Minimum Completion Time
MET	Minimum Execution Time
NP	Nichtdeterministisch Polynomiell
OLB	Opportunistic Load Balancing
p-cons	partially consistent, partiell-konsistent
pc	partially consistent, partiell-konsistent
SA	Simulated Annealing
SIMD	Single Instruction, Multiple Data
Trans.	Transactions on
usw.	und so weiter
vgl.	vergleiche
Z	Zeile
z. B.	zum Beispiel
z. T.	zum Teil

1. Einleitung

1.1. Motivation

Das Problem des Job Scheduling, gemeint ist die Verteilung von Tasks auf verschiedene Rechner oder Prozessoren, ist ein NP-vollständiges Problem. Das heißt, eine optimale Lösung ist sehr wahrscheinlich nicht in polynomialer Zeit berechenbar. Deshalb wird durch den Einsatz von verschiedenen Heuristiken versucht, Lösungen zu finden, die gute Ergebnisse in vertretbarer Zeit liefern.

In der Literatur wurden verschiedene Heuristiken vorgestellt, die sich des Problems des Job Scheduling annehmen und recht brauchbare Ergebnisse liefern. Es wird nach einem Ansatzpunkt gesucht, um diese Ergebnisses weiter zu optimieren, also die durch die Heuristik erreichten Ergebnisse nochmals zu verbessern und somit die Gesamtlaufzeit der durch das Job Scheduling verteilten Tasks zu minimieren.

1.2. Überblick

Grundsätzlich geht es bei dem in dieser Arbeit betrachteten Problems des Job Scheduling um die statische Verteilung von unabhängigen Tasks auf verschiedene Zielmaschinen. Statisch bedeutet in diesem Zusammenhang, dass die Verteilung der Tasks im Vorfeld der eigentlichen Ausführung stattfindet – das heißt, dass die Anzahl der Tasks, die Anzahl der zur Verfügung stehenden Maschinen und die Laufzeiten der Tasks auf den Maschinen im Vorfeld bekannt sind. Auch das nachträgliche Berechnen von bereits durchgeführten Metatasks kann sinnvoll sein, um zum Beispiel Optimierungsszenarios für zukünftige Läufe zu entwickeln.

Die hier betrachteten Tasks sind unabhängig voneinander. Dies bedeutet, dass kein Task von den Ergebnissen oder Zwischenergebnissen eines anderen Tasks abhängig ist und somit die Tasks vollständig unabhängig voneinander ausgeführt werden können.

In dieser Arbeit werden elf verschiedene Heuristiken betrachtet, die jeweils verschiedene Ansätze zur Taskverteilung verfolgen. Gemeinsam ist all diesen Heuristi-

ken, dass sie auf ihre Art und Weise die vorhandenen Tasks auf die zur Verfügung stehenden Maschinen verteilen. Dabei werden einfache Heuristiken betrachtet, welche die Tasks aufgrund ihrer Laufzeiten verteilen, aber auch genetische Modelle, die weit fortgeschrittene Techniken und Operationen verwenden, um die Taskverteilung durchzuführen.

1.3. Ziele

Durch den Einsatz von Optimierungsmethoden soll versucht werden, die Laufzeit der Metatasks nach dem Einsatz der Heurisiken nochmals zu verbessern. Dabei werden die Ergebnisse der Heuristiken, die erstellen Taskverteilungspläne, unabhängig von der verwendeten Heuristik betrachtet und der Versuch unternommen, die Ausführungszeit des Metatasks weiter zu verkürzen.

Der Grundgedanke ist, die Maschine mit der längsten Laufzeit zu identifizieren und Tasks von dieser Maschine auf eine andere zu verschieben und dadurch die Gesamtlaufzeit zu verringern. Dies wird solange durchgeführt, bis keine Verbesserung der Laufzeit mehr zustande gebracht wird.

Genau betrachtet handelt es sich bei dem Optimierungsvorgang um eine Suche nach einem lokalen Minimum. Es wird folglich im Rahmen einer lokalen Suche nach Lösungen in der Nachbarschaft der aktuellen Lösung gesucht, die nochmals besser sind. Die genaue Definition von "Nachbarschaft" stellt hier ein zu lösendes Problem dar [Ri07]. In dieser Umgebung werden Lösungen betrachtet, welche sich in der Taskzuordnung unterscheiden, d. h. in der Tasks anderen Maschinen zugeordnet werden.

2. Implementierung

Die Implementierung des Gesamtsystems umfasst drei Teile: Die ETC-Matrix als Basisdatenstruktur zur Darstellung der Tasklaufzeiten auf den verschiedenen Zielsystemen. Die Scheduler, welche die eigentlichen Heuristiken implementieren und der Tuner, der im Anschluss an die Scheduler-Läufe die Optimierung der erstellten GANTT-Diagramme[1], im weiteren Verlauf als Taskpläne bezeichnet, durchführt. Die ersten beiden Teile, die ETC-Matrix bzw. die Heuristiken, basieren auf [Tr01]. In den folgenden Abschnitten werden die drei Komponenten näher beschrieben und es wird auf die eigentliche Implementierung eingegangen. Abschnitt 2.1 befasst sich mit der ETC-Matrix, Abschnitt 2.2 mit den Heuristiken und dem Basisumfang des Schedulers. Abschnitt 2.3 umfasst den Hauptteil der Arbeit, den Tuner und dessen Implementierung. In Abschnitt 2.4 werden schließlich Strategien für eine erweiterte Implementierung des Tuners vorgestellt.

Bevor das eigentliche Thema begonnen wird, sollen an dieser Stelle einige Begrifflichkeiten geklärt werden, die im Verlauf des Textes noch vermehrt auftreten. Die Begriffe werden sowohl im englischen Original auftreten, meist im Bereich von Codebeispielen o. ä., sowie in übersetzter deutscher Variante.

Die *Laufzeit* oder *Ausführungszeit* (*Execution Time*, ET) eines Tasks t_i auf einer Maschine m_j ist die Rechenzeit des Tasks auf dieser Maschine. Es handelt sich um den entsprechenden Eintrag aus der ETC-Matrix ($ETC(t_i, m_j)$).

Die *Fertigstellungszeit einer Maschine*, (*machine availability time*, $mat(m_j)$), bezeichnet nach [Tr01] die minimale Zeit, die Maschine m_j benötigt, um die ihr zugeordneten Tasks abzuarbeiten.

Die *Fertigstellungszeit* (*completion time, CT*) für einen Task t_i auf einer Maschine m_j ($ct(t_i, m_j)$), entspricht nach [Tr01] der Verfügbarkeitszeit der Maschine m_j zuzüglich der Ausführungszeit des Tasks t_i auf dieser Maschine m_j. Es gilt $ct(t_i, m_j) = mat(m_j) + ETC(t_i, m_j)$.

Das Kriterium, um die Ergebnisse der verschiedenen Heuristiken (Scheduler) zu

[1] Benannt nach dem Unternehmensberater Henry L. Gantt (1861-1919), siehe auch *http://de.wikipedia.org/wiki/Gantt-Diagramm*

vergleichen, ist der Maximalwert von $ct(t_i, m_j)$ für $0 \leq i < \tau$ und $0 \leq j < \mu$. Der Maximalwert von $ct(t_i, m_j)$ ist die *Ausführungszeit des Metatasks* (*metatask execution time*). Dieser wird als *Makespan* bezeichnet.

2.1. Matrix und ETC-Matrix

Die grundlegende Datenstruktur zur Darstellung der Tasklaufzeiten auf den verschiedenen Maschinen ist nach [Tr01] die ETC-Matrix (*expected time to complete-Matrix*), siehe Abbildung 2.1. Es handelt sich hierbei um eine $\tau \times \mu$ Matrix, wobei τ der Anzahl der auszuführenden Tasks und μ der Anzahl der zur Verfügung stehenden Maschinen entspricht. Somit hat die Matrix τ Zeilen, jeweils eine für jeden Task t_i mit $0 \leq i < \tau$. Sie enthält μ Spalten, jeweils eine Spalte für jede Maschine m_j mit $0 \leq j < \mu$. Ein Eintrag $ETC(t_i, m_j)$ entspricht somit der Laufzeit des Tasks t_i auf der Maschine m_j. Die Laufzeit beinhaltet neben der eigentlichen Zeit für die Berechnungen auch die notwendige Zeit, um den Task auf die Zielmaschine zu verschieben und die benötigten Daten bereitzustellen [Tr01].

$$\begin{pmatrix} x_{0,0} & \cdots & x_{0,\mu-1} \\ \vdots & \ddots & \vdots \\ x_{\tau-1,0} & \cdots & x_{\tau-1,\mu-1} \end{pmatrix}$$

Abbildung 2.1.: Aufbau der ETC-Matrix

Die Generierung der ETC-Matrix wird nach [Tr01] wie folgt durchgeführt: In einem ersten Schritt wird ein $\tau \times 1$ *Base Line Vektor* (BLV) B erzeugt. Dabei wird dieser Vektor mit Zufallswerten befüllt, wobei Φ_b die Obergrenze der Werte darstellt. Jeder Eintrag im BLV ist eine Zufallszahl $x_b^i \in [1; \Phi_b[$. $B(i) = x_b^i$ für $0 \leq i < \tau$.

Im nächsten Schritt werden für jede Zeile der ETC-Matrix jeweils μ Zufallszahlen $x_r^{i,j}$ mit der Obergrenze Φ_r erzeugt. Diese Zufallszahlen sind die *Zeilenmultiplikatoren* $x_r^{i,j} \in [1; \Phi_r[$.

Jeder Eintrag in der ETC-Matrix wird dann als Produkt des Eintrages im BLV mit dem jeweiligen Zeilenmultiplikator generiert. Es gilt folgende Formel: $ETC(t_i, m_j) = B(i) \times x_r^{i,j}$ für $0 \leq i < \tau, 0 \leq j < \mu$. Der Wertebereich für jeden Eintrag in der ETC-Matrix liegt im Bereich von $[1; \Phi_b \times \Phi_r[$.

Programmtechnisch wurde für die Erzeugung der ETC-Matrix eine $\tau \times \mu + 1$ Hilfsmatrix, die *Init Value Matrix* (IV-Matrix), verwendet (siehe Abbildung 2.2a). Dabei entspricht der erste Spaltenvektor dem BLV (siehe Abbildung 2.2b) und die

Submatrix ab der zweiten Spalte den Zeilenmultiplikatoren (*Row Multiplier Sub Matrix*, Abbildung 2.2c).

$$\begin{pmatrix} x_{0,0} & x_{0,1} & \cdots & x_{0,\mu} \\ \vdots & \vdots & \ddots & \vdots \\ x_{\tau-1,0} & x_{\tau-1,1} & \cdots & x_{\tau-1,\mu} \end{pmatrix} \quad \begin{pmatrix} x_{0,0} \\ \vdots \\ x_{\tau-1,0} \end{pmatrix} \quad \begin{pmatrix} x_{0,1} & \cdots & x_{0,\mu} \\ \vdots & \ddots & \vdots \\ x_{\tau-1,1} & \cdots & x_{\tau-1,\mu} \end{pmatrix}$$

$a)$ $b)$ $c)$

Abbildung 2.2.: IV-Matrix, BLV und Row Multiplier Sub Matrix

Zur Darstellung verschiedener Umgebungen können in der ETC-Matrix bestimmte Parameter variiert werden. Zwei Parameter zur Anpassung der Werte der ETC-Matrix sind die *Taskheterogenität* bzw. die *Maschinenheterogenität* [Tr01]. Die Obergrenzen zur Erzeugung der Zufallszahlen Φ_b (Taskheterogenität) bzw. Φ_r (Maschinenheterogenität) werden durch diese beiden Werte festgelegt. Für hohe Taskheterogenität wird ein Wert von 3000, für geringe ein Wert von 100 als Obergrenze für Φ_b festgelegt [Tr01]. Analog dazu wird für hohe Maschinenheterogenität ein Wert von 1000, für geringe ein Wert von 10 als Obergrenze für Φ_r festgelegt [Tr01].

Als weiterer Parameter, der die Eigenschaften der ETC-Matrix beeinflusst, wird die Konsistenz eingeführt. Dabei werden drei Möglichkeiten unterschieden. Eine ETC-Matrix ist konsistent, wenn jede Maschine m_j den Task t_i immer schneller ausführt als Maschine m_{j+1} [Tr01]. Dies bedeutet, dass die Zeilen der ETC-Matrix aufsteigend sortiert sind. Daraus folgt, dass Maschine m_0 die schnellste und $m_{\mu-1}$ die langsamste Maschine ist. Ein Beispiel für eine konsistente Matrix ist in Abbildung 2.3 gegeben.

$$ETC_{cons} = \begin{pmatrix} 1.2342 & 1.4388 & 2.1838 & 2.7783 & 2.9377 & 3.8821 \\ 2.3433 & 2.4778 & 4.8382 & 4.8383 & 4.9992 & 5.9983 \\ 1.9333 & 2.0012 & 2.3884 & 3.8283 & 4.3838 & 6.3883 \\ 1.3844 & 2.8384 & 5.2883 & 5.4338 & 6.2883 & 7.2033 \end{pmatrix}$$

Abbildung 2.3.: Konsistente ETC-Matrix

Ist m_j für manche Tasks langsamer als m_{j+1} und für andere die schnellere Maschine, handelt es sich um eine inkonsistente Matrix [Tr01]. Es handelt sich um die unsortierte, zufallsgenerierte Matrix. Vgl. die Matrix in Abbildung 2.4.

Eine partiell-konsistente Matrix beinhaltet eine konsistente Submatrix vorgegebener Größe [Tr01]. Dabei werden die Zeilenelemente der Spalten $\{0, 2, 4, \dots\}$ der

$$ETC_{incons} = \begin{pmatrix} 2.1838 & 1.3833 & 1.8384 & 6.3838 & 7.3838 & 3.3833 \\ 7.3884 & 7.2488 & 2.3838 & 3.3883 & 3.3882 & 4.3883 \\ 4.3883 & 2.8383 & 1.3883 & 1.8838 & 1.2888 & 3.4884 \\ 3.2883 & 3.8844 & 5.1883 & 3.8219 & 3.3567 & 2.3819 \end{pmatrix}$$

Abbildung 2.4.: Inkonsistente ETC-Matrix

Reihe i extrahiert, sortiert und in der neuen Reihenfolge wieder in die Matrix eingefügt. Die Zeilenelemente $\{1, 3, 5, \dots\}$ bleiben unsortiert. Die Matrix in Abbildung 2.5 ist die partiell-konsistente Version von Abbildung 2.4. Die fett markierten Spalten entsprechen der konsistenten Submatrix.

$$ETC_{p-cons} = \begin{pmatrix} \mathbf{1.8384} & 1.3833 & \mathbf{2.1838} & 6.3838 & \mathbf{7.3838} & 3.3833 \\ \mathbf{2.3838} & 7.2488 & \mathbf{3.3882} & 3.3883 & \mathbf{7.3884} & 4.3883 \\ \mathbf{1.2888} & 2.8383 & \mathbf{1.3883} & 1.8838 & \mathbf{4.3883} & 3.4884 \\ \mathbf{3.2883} & 3.8844 & \mathbf{3.3567} & 3.8219 & \mathbf{5.1883} & 2.3819 \end{pmatrix}$$

Abbildung 2.5.: Partiell-konsistente ETC-Matrix

Es existieren zwölf verschiedene Permutationen aus Konsistenz, Task- und Maschinenheterogenität. Für die Tests wurde $\tau = 512$ und $\mu = 16$ gewählt (512 Tasks auf 16 Maschinen). Es wurden jeweils 100 Durchläufe pro Scheduler durchgeführt und das Ergebnis gemittelt ausgegeben.

Die Implementierung der ETC-Matrix erfolgte durch zwei Klassen. Die Basisklasse ist eine einfache Datenstruktur zur Darstellung einer Matrix von `long double` Werten. Die öffentliche Schnittstelle enthält Methoden zur Manipulation der Einträge sowie zur Auswahl einzelner Elemente oder Vektoren. Dabei besteht die eigentliche Matrix aus einem Vektor von Vektoren von `long double` Werten. Folgende öffentliche Methoden werden durch die Klasse `Matrix` bereitgestellt:

`Matrix(rows,cols,initVal)` Konstruktor zur Erzeugung einer $rows \times cols$ Matrix. Alle Werte werden mit `initVal` vorbelegt. Standardwert ist $0, 0$.

`Matrix(const Matrix& m)` Kopierkonstruktor zur Erzeugung identischer Matrix-Kopien.

`~Matrix()` Destruktor.

`init(rows,cols,initVal)` Erzeugung bzw. Neuerzeugung der Matrix. Es wird eine $rows \times cols$ Matrix mit Anfangswert `initVal` erzeugt.

`getRowCount()` Liefert die Anzahl der Zeilen zurück.

`getColumnCount()` Liefert die Anzahl der Spalten zurück.

`setRow(row,newRow)` Ersetzt die Reihe `row` mit der neuen Reihe `newRow`.

`setColumn(col,newCol)` Ersetzt die Spalte `col` mit der neuen Spalte `newCol`.

`getRow(row)` Liefert den Spaltenvektor der Spalte `row` zurück.

`getColumn(col)` Liefert den Zeilenvektor der Zeile `col` zurück.

`set(row,col,value)` Setzt den Eintrag (`row,col`) der Matrix auf den Wert `value`.

`get(row,col)` Liefert den Wert des Eintrags (`row,col`) der Matrix zurück.

`sortRow(row)` Sortiert die Zeile `row` aufsteigend.

`sortAllRows()` Sortiert alle Zeilen der Matrix.

`getMatrixAsString(spacing)` Liefert die Matrix als String zurück. Die Matrix wird um den Wert `spacing` eingerückt.

Von der Basisklasse `Matrix` ist wiederum die Klasse `ETCMatrix` als spezialisierte Variante abgeleitet. Die Klasse `ETCMatrix` stellt die Basisdatenstruktur zur Verwendung in den Schedulern im Sinne von [Tr01] dar. Die Klasse hat, zusätzlich zu den geerbten Methoden, weitere, speziell für die ETC-Matrix relevante Methoden. Folgende öffentliche Methoden werden durch die Klasse `ETCMatrix` bereitgestellt:

`ETCMatrix(rows,cols,th,mh,cons)` Konstruktor zur Erzeugung einer $rows \times cols$ ETC-Matrix. Die Parameter `th` bzw. `mh` legen die Task- bzw. Maschinenheterogenität fest. Über den Parameter `cons` wird die Konsistenz eingestellt.

`ETCMatrix(const ETCMatrix& m)` Kopierkonstruktor zur Erzeugung einer identischen Kopie der ETC-Matrix.

`~ETCMatrix()` Destruktor.

`init(rows,cols,th,mh,cons)` Erzeugung bzw. Neuerzeugung der ETC-Matrix. Es wird eine $rows \times cols$ ETC-Matrix mit Task- bzw. Maschinenheterogenität `th` bzw. `mh` und Konsistenz `cons` erzeugt.

`makeConsistent()` ETC-Matrix in eine konsistente Matrix überführen.

`makeInconsistent()` ETC-Matrix in eine inkonsistente Matrix überführen.

`makePartiallyConsistent()` ETC-Matrix in eine partiell-konsistente Matrix überführen.

`getTaskHeterogenity()` Liefert die Taskheterogenität zurück.

`getMachineHeterogenity()` Liefert die Maschinenheterogenität zurück.

`getConsistency()` Liefert die Konsistenz zurück.

`getInitValueMatrix()` Liefert die *Init Value Matrix* als Matrix-Objekt zurück.

`getSummaryAsString(spacing)` Liefert die ETC-Matrix mit den Einstellungsparametern als String zurück.

2.2. Vorstellung der Scheduler

Die Scheduler setzen sich aus zwei Teilen zusammen. Der erste Teil stellt die abstrakte Basisklasse `Scheduler` dar, welche die Grundfunktionalitäten für die eigentlichen Scheduler bereitstellt (siehe hierzu Abschnitt 2.2.1). Der zweite Teil besteht aus den von der Basisklasse abgeleiteten eigentlichen Schedulern. Diese implementieren jeweils eine eigene Heuristik und werden im Abschnitt 2.2.2 vorgestellt.

2.2.1. Beschreibung der Basisklasse

Die Klasse `Scheduler` stellt Grundfunktionalitäten für die von dieser Klasse abgeleiteten eigentlichen Scheduler bereit. Insbesondere Methoden, um Tasks bestimmten Maschinen zuzuordnen, zu verschieben oder aus dem Taskplan zu löschen. Diese Funktionalitäten stehen allerdings nur den abgeleiteten Klassen zur Verfügung. Die öffentliche Schnittstelle bietet eine Reihe von Methoden an, um Scheduler zu erzeugen und auszuführen, allgemeine und spezielle Informationen zu den Taskplänen zu erhalten, den Taskplan, die ETC-Matrix und erweiterte Informationen als String zurückzuliefern. Zudem existieren Methoden, um Merkmale bei Bedarf zu- oder abzuschalten. Aktuell werden folgende Merkmale unterstützt:

`DISABLE_ADDTASK_EXTINFO` Wenn dieses Merkmal aktiviert wird, werden keine erweiterten Informationen bereitgestellt, die das Hinzufügen von Tasks in den Taskplan betreffen.

`ENABLE_TUNE_RATIOCHECK` Standardmäßig wird beim Tuning nur die Verbesserung der Makespan berücksichtigt. Anhand dieser Information wird entschieden, welcher Task verschoben werden soll. Ist der Ratio-Check aktiviert, wird bei gleicher Verbesserung der Makespan als zusätzliches Indiz der Quotient aus Tasklaufzeit auf der Zielmaschine und Tasklaufzeit auf der Quellmaschine geprüft. Siehe hierzu auch Abschnitt 2.3.

In der Klasse ist zudem die Hauptmethode `schedule()` der abgeleiteten Scheduler als abstrakte Methode deklariert. Diese Methode ist der Einstiegspunkt für jeden abgeleiteten Scheduler und wird in der entsprechenden Klasse implementiert. Durch den Aufruf der Methode `schedule()` wird eine entsprechende Heuristik ausgeführt, welche die Tasks der ETC-Matrix in den Taskplan einsortiert.

In Abbildung 2.6 ist ein solcher Taskplan ersichtlich, in den die Tasks aus der Quellmatrix, dargestellt in Tabelle 2.1, zufällig einsortiert wurden. Auf der vertikalen Achse sind die Maschinen, welche die Tasks ausführen, auf der horizontalen Achse die Tasklaufzeiten der einzelnen Tasks angeordnet. So wurden Tasks T1 und T5 der Maschine M1, Task T3 der Maschine M2, Tasks T6 und T8 der Maschine M3 sowie Tasks T2, T4 und T7 der Maschine M4 zugeordnet. Maschine M1 hat eine *Fertigstellungszeit* von $4,5650$, M2 von $2,8383$, M3 von $4,5766$ und M4 von $10,5938$. Somit hat Maschine M4 ganz offensichtlich die längste Fertigstellungszeit und diese Maschine legt die Makespan für den gesamten Metatask fest. Die Verteilung der Tasks erfolgte in diesem Beispiel, wie oben erwähnt, zufällig und stellt dementsprechend eine äußerst schlechte Lösung dar. Andere Heuristiken würden hier deutlich bessere Ergebnisse liefern.

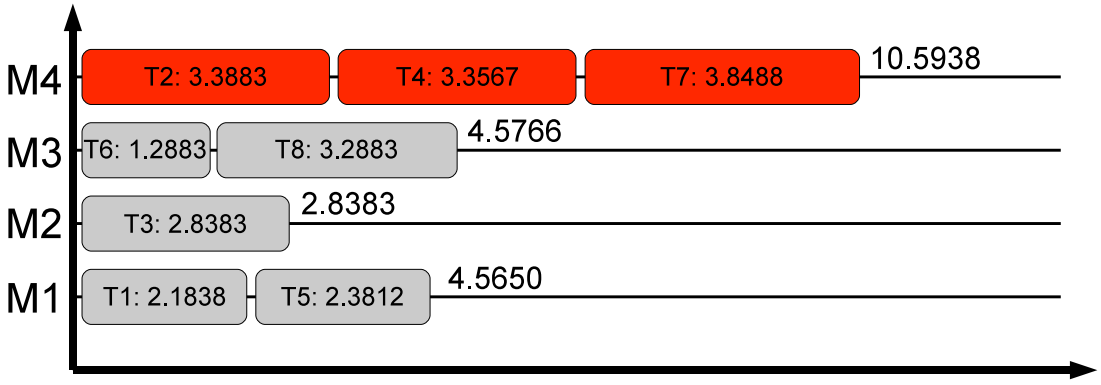

Abbildung 2.6.: Taskplan Beispiel

	Maschinen			
	2.1838	1.3833	1.8384	6.3838
T	7.3884	7.2488	2.3838	**3.3883**
a	4.3883	**2.8383**	1.8838	1.2888
s	3.2883	5.1883	3.8219	**3.3567**
k	**2.3812**	3.2383	8.3882	1.2382
s	2.3881	3.4882	**1.2883**	4.3884
	7.3884	1.2838	3.2883	**3.8488**
	5.3488	3.2883	**3.2883**	3.4532

Tabelle 2.1.: Quellmatrix zum Taskplan

Die öffentlichen Methoden, die durch die Klasse `Scheduler` bereitgestellt werden, sind:

`Scheduler(etc)` Konstruktor zu Erzeugung eines Scheduler-Objekts. Als Übergabeparameter wird die zu verwendende ETC-Matrix übergeben.

`~Scheduler()` Destruktor

`init(etc)` Initialisierungsmethode, die auch vom Konstruktor aufgerufen wird. Hiermit kann das Scheduler-Objekt mit einer anderen ETC-Matrix neu initialisiert werden.

`getTaskCountOnMachine(machine)` Liefert die Anzahl der Tasks zurück, die auf der Maschine `machine` ausgeführt werden. Im Beispiel aus Abbildung 2.6 würde diese Methode für den Übergabeparameter 4 den Wert 3 zurückliefern.

`getTaskNumber(entry,machine)` Diese Methode liefert die tatsächliche Tasknummer des Eintrags `entry` der Maschine `machine` aus dem Taskplan zurück. Im Falle des Beispiels aus Abbildung 2.6 würde diese Methode für die Übergabeparameter $(3, 4)$, entspricht Eintrag 3 auf Maschine 4, den Wert 7, also Tasknummer 7, zurückliefern.

`getExecTime(machine)` Liefert die Fertigstellungszeit der Maschine `machine` zurück. Für das Beispiel aus Abbildung 2.6 würde diese Methode für den Übergabeparameter 2 den Wert $2,8383$ zurückliefern.

`getExecTimes()` Liefert einen Vektor der Fertigstellungszeiten aller Maschinen im Taskplan zurück. Würde im Falle des Beispiels aus Abbildung 2.6 den Vektor $(4,5650; 2,8383; 4,5766; 10,5938)$ zurückliefern.

`getMakespan()` Liefert die Makespan, also die Gesamtlaufzeit, zurück. Liefert bei dem Beispiel aus Abbildung 2.6 den Wert $10,5938$ zurück.

`getTaskCount()` Liefert die Anzahl der Tasks zurück, die auf den Taskplan verteilt werden sollen. Entspricht der Methode `getRowCount()` der Klasse `Matrix`. Im Beispiel aus Abbildung 2.6 wird hier der Wert 8 zurückgegeben.

`getMachineCount()` Liefert die Anzahl der Maschinen zurück, auf die die Tasks verteilt werden sollen. Entspricht der Methode `getColumnCount()` der Klasse `Matrix`. Im Falle des Beispiels aus Abbildung 2.6 wird der Wert 4 zurückgegeben.

`schedule()` Abstrakte Methode, die von den abgeleiteten Schedulern implementiert wird.

`tune()` Die Methode, die den eigentlichen Tuner implementiert. Details hierzu sind im Abschnitt 2.3 zu finden.

`getScheduleAsString(spacing)` Gibt den Taskplan als String zurück. `spacing` gibt vor, um wie viele Zeichen die Ausgabe eingerückt werden soll.

`getETCMatrixAsString(spacing)` Gibt die ETC-Matrix als String aus. Entspricht der Funktion `getMatrixAsString()` aus der Klasse `Matrix`.

`getExtSchedInfo()` Gibt erweiterte Informationen zum Scheduling-Ablauf als String zurück.

`getExtTuneInfo()` Gibt erweiterte Informationen zum Tuning-Ablauf als String zurück.

`enable(feature)` Schaltet das Merkmal `feature` an.

`disable(feature)` Schaltet das Merkmal `feature` aus.

`clearFeatures()` Schaltet alle Merkmale ab.

`getFeatures()` Gibt einen Bitvektor (`unsigned int`) der Merkmale zurück. Ein gesetztes Bit entspricht einem aktivierten, ein 0-Bit einem deaktivierten Merkmal.

`isEnabled(feature)` Gibt `true` zurück, wenn das Merkmal `feature` aktiviert ist, ansonsten `false`.

2.2.2. Beschreibung der verschiedenen Scheduler

Die öffentliche Schnittstelle eines abgeleiteten Schedulers unterscheidet sich nicht von der der Basisklasse. Der Konstruktor sowie die aus der Basisklasse geerbte abstrakte Methode `schedule()` werden implementiert. Gegebenenfalls wird die Initialisierungsmethode überschrieben, soweit dies notwendig sein sollte. Die Methode `schedule()` stellt den Einstiegspunkt für die eigentlichen Scheduler dar. Es wurden 13 Scheduler entwickelt, die jeweils eine bestimmte Heuristik zur Taskverteilung implementieren. Diese werden im Folgenden näher erläutert.

SimpleScheduler: Dieser Scheduler verteilt die Tasks der Reihe nach jeweils auf die nächste Maschine. Ist die letzte Maschine erreicht, wird der Folgetask wieder der ersten Maschine zugeordnet. Dies wird solange ausgeführt, bis alle Tasks verteilt sind. Dieser Scheduler ist nur ein Testmodul, um die Grundfunktionalität zu überprüfen. Die Ergebnisse sind, wie dies bei dieser "Heuristik" zu erwarten war, nicht besonders gut. Dieser Scheduler wird in der Analyse nicht weiter berücksichtigt.

RandomScheduler: Dieser Scheduler verteilt die Tasks zufällig auf die Maschinen, bis alle Tasks verteilt sind. Auch dieser Scheduler ist ein Testmodul und wird in der Analyse nicht weiter berücksichtigt.

OLBScheduler: Der OLB (*Opportunistic Load Balancing*) Scheduler ordnet jeden Task in der Reihenfolge, in der er in der ETC-Matrix auftaucht, der Maschine zu, die am Wahrscheinlichsten als nächstes verfügbar ist [Tr01]. Dabei wird die Laufzeit des Tasks auf der Maschine nicht berücksichtigt. Das Ziel ist es, die Maschinen soweit als möglich auszulasten [Tr01]. Der Vorteil von OLB ist die Tatsache, dass es sich um eine sehr einfache Heuristik handelt, allerdings werden nur sehr schlechte, d. h. sehr lange Makespans, erreicht [Tr01].

Listing 2.1: Pseudocode OLB Scheduler

```
1 for(task=0;task<TASKCOUNT;task++) {
2   nextMachineAvail=0;
3   for(machine=1;machine<MACHINECOUNT;machine++)
4     if(avail(machine)<avail(nextMachineAvail)
5       nextMachineAvail=machine;
6   addTask(task,nextMachineAvail);
7 }
```

In den Zeilen 1-7 des Listings 2.1 werden alle Tasks durchgegangen. In Zeilen 3-5 werden sämtliche Maschinen durchlaufen und die Variable `nextMachineAvail` auf die Maschine gesetzt, die den Task am ehesten ausführen kann. Schließlich wird in Zeile 6 der Task für die entsprechende Maschine in den Taskplan übernommen.

METScheduler: Der MET (*Minimum Execution Time*) Scheduler ordnet jeden Task in der Reihenfolge, in der er in der ETC-Matrix auftaucht, der Maschine zu, die den Task am schnellsten ausführen kann, ohne die Verfügbarkeit der Maschine zu berücksichtigen [Tr01], siehe auch Pseudocode in Listing 2.2. Das Ziel ist es, jeden Task der für diesen besten Maschine zuzuordnen [Tr01]. Dies kann dazu führen, dass die Maschinen deutlich unausgeglichen mit Arbeiten ausgelastet sind. Nach [Tr01] ist diese Heuristik für heterogene Umgebungen, die durch eine konsistente ETC-Matrix gekennzeichnet sind, wenig geeignet. In diesem Fall sind alle Tasks der ersten Maschine zugeordnet, während den anderen Maschinen keine Tasks zur Bearbeitung zugeordnet werden.

Listing 2.2: Pseudocode MET Scheduler

```
1 for(task=0;task<TASKCOUNT;task++) {
2   ShortestET=0;
3   for(machine=1;machine<MACHINECOUNT;machine++) {
4     if(execTime(task,machine)<execTime(task,ShortestET)){
5       ShortestET=machine;
6     }
7   }
8   addTask(task,ShortestET);
9 }
```

MCTScheduler: Der MCT (*Minimum Completion Time*) Scheduler, siehe Pseudocode 2.3, ordnet jeden Task in der Reihenfolge, in der er in der ETC-Matrix auftaucht, der Maschine zu, die für diesen Task die wahrscheinlich minimalste Fertigstellungszeit hat [Tr01]. Dies führt dazu, dass Tasks unter Umständen auch Maschinen zugeordnet werden, die nicht die schnellste Ausführungszeit für diesen Task haben [Tr01]. Die Idee hinter MCT ist nach [Tr01] die Vorteile von OLB und MET zu kombinieren, während die Nachteile von OLB bzw. MET vermieden werden.

Listing 2.3: Pseudocode MCT Scheduler

```
for(task=0;task<TASKCOUNT;task++) {
  MinCT=0;
  for(machine=1;machine<MACHINECOUNT;machine++)
    if(CT(task,machine)<CT(task,MinCT))
      MinCT=machine;
  addTask(task,MinCT);
}
```

MinMinScheduler: Der Min-Min Scheduler, siehe Listing 2.4, beginnt mit einer Menge U aller noch nicht verteilten Tasks [Tr01]. In jedem Durchlauf wird die Menge der minimalen Fertigstellungszeiten $M = \{min_{0 \leq j < \mu}(ct(t_i, m_j))\}$ gebildet. Aus dieser Menge wird der Task mit der geringsten Fertigstellungszeit ausgewählt und, der entsprechenden Maschine zugeordnet, in den Taskplan übernommen [Tr01]. Im Anschluss daran wird der Task aus der Menge U entfernt und der Vorgang von vorn begonnen, bis alle Tasks aus U in den Taskplan übernommen worden sind [Tr01].

Listing 2.4: Pseudocode Min-Min Scheduler

```
while(unmapped tasks not empty) {
  create set of minimal ct;
  add task with overall minimum ct to schedule;
  remove this task from unmapped tasks;
}
```

Min-Min basiert auf MCT, betrachtet aber im Gegensatz zu diesem nicht nur einen Task pro Runde, sondern alle noch nicht verteilten Tasks [Tr01]. Die Tasks werden dem Taskplan so zugefügt, dass sich die Verfügbarkeit der Maschine am wenigsten ändert [Tr01].

MaxMinScheduler: Die Max-Min Heuristik, siehe Listing 2.5, ist der Min-Min sehr ähnlich. Auch hier wird mit der Menge U der noch nicht verteilten Tasks begonnen. Die Menge M der minimalen Fertigstellungszeiten wird gebildet, aber im Gegensatz zu Min-Min wird der Eintrag mit der maximalen Fertigstellungszeit aus M ausgewählt und, der entsprechenden Maschine zugeordnet, in den Taskplan übernommen [Tr01]. Dann wird der Task aus U entfernt und der Vorgang wiederholt, bis alle Tasks zugeordnet wurden und die Menge U leer ist.

Listing 2.5: Pseudocode Max-Min Scheduler

```
1 while(unmapped tasks not empty) {
2   create set of minimal ct;
3   add task with overall maximum ct to schedule;
4   remove this task from unmapped tasks;
5 }
```

Max-Min versucht die Nachteile von langläufigen Tasks dadurch zu umgehen, dass diese möglichst frühzeitig verteilt werden [Tr01].

DuplexScheduler: Der Duplex Scheduler, siehe auch Listing 2.6, ist eine Kombination aus Min-Min und Max-Min [Tr01]. Es werden beide Heuristiken ausgeführt und das bessere Ergebnis als Taskplan übernommen [Tr01]. Duplex eignet sich nach [Tr01] daher dafür, die für die jeweiligen Bedingungen bessere Heuristik auszuwählen – zu dem Preis, der etwa doppelten Laufzeit des Schedulers.

Listing 2.6: Pseudocode Duplex Scheduler

```
1 perform(minmin);
2 perform(maxmin);
3 if(minmin better)
4   use minmin schedule;
5 else
6   use maxmin schedule;
```

GAScheduler: Auch die Implementierung dieses Schedulers ist an die Arbeiten von [Tr01] angelehnt. Der GA-Scheduler (*Genetic Algorithm*) beginnt mit einer Population von 200 Chromosomen, wobei jedes Chromosom durch einen $\tau \times 1$ Vektor dargestellt wird [Tr01]. Dabei entspricht die Position innerhalb des Vektors der Tasknummer und der Eintrag selbst der Maschinenzuordnung. Es gilt $cr_i = m_j$ mit $0 \leq j < \mu$ und $0 \leq i < \tau$. Der grundlegende Ablauf ist in Abbildung 2.7 ersichtlich, Codelisting 2.7 stellt einen stark vereinfachten Pseudocode dar.

In der **Generierungsphase**, Zeile 1 im Listing 2.7, werden jeweils 200 Chromosomen erstellt. Dabei werden zwei verschiedene Arten der Generierung durchgeführt: Einmal werden alle 200 Chromosomen aus Zufallswerten (also zufälligen Task-Maschine-Zuordnungen) erstellt, ein anderes Mal wird ein Chromosom aus dem Ergebnis eines Min-Min Taskplans gewonnen, während die restlichen 199 Chromosomen zufällig erzeugt werden. Das zweite Vorgehen wird mit dem englischen Be-

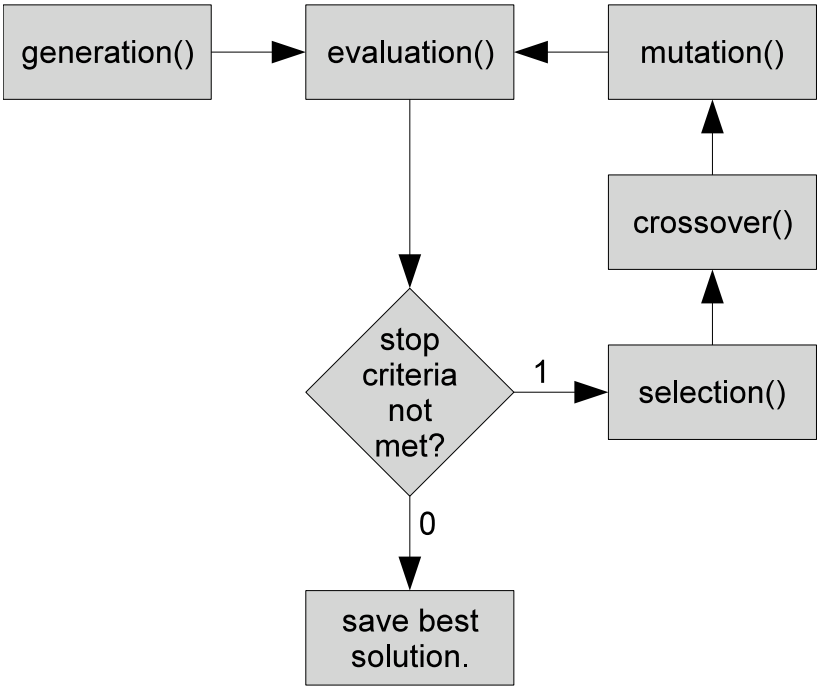

Abbildung 2.7.: Ablaufplan GA Scheduler

griff *seeding* bezeichnet. Für die Analyse wurden beide Varianten jeweils viermal durchlaufen und das beste Ergebnis aus den insgesamt acht Läufen als Endergebnis übernommen [Tr01].

Listing 2.7: Pseudocode GA Scheduler nach [Tr01]

```
1 generation();
2 evaluation();
3 while(stopping criteria not met) {
4   selection();
5   crossover();
6   mutation();
7   evaluation();
8 }
9 save_best_solution;
```

Nach [Tr01] existiert für jedes Chromosom ein *fitness value*, welcher der Makespan für dieses Chromosom entspricht. Er ist umso besser, je niedriger der Wert ist. Die Berechnung der *fitness values* erfolgt erstmals nach der Generierung der Population in dem Schritt der **Evaluierung**, im Listing 2.7 in Zeile 2. Im Anschluss an Generierung und erstmaliger Evaluierung beginnt die Hauptschleife des Schedulers, im

Codelisting 2.7 in den Zeilen 3-8. Die Schleife wird ausgeführt, solange keines der folgenden drei Haltkriterien erfüllt ist:

1. insgesamt 1000 Schleifendurchläufe
2. keine Änderung im Elitechromosom für mehr als 150 Schleifendurchläufe
3. alle Chromosomen konvergieren gegen das gleiche Mapping

Der erste Schritt in der Hauptschleife ist die **Auswahl-** bzw. **Selektionsphase**, im Listing 2.7 in Zeile 4. Innerhalb dieser Phase werden bestimmte Chromosomen für die nächste Runde, teilweise auch mehrfach, übernommen, andere werden verworfen. Grundsätzlich haben bessere Mappings eine höhere Chance, für die nächste Runde übernommen zu werden, als schlechtere Mappings. Dabei wird eine vereinfachte, an [Sr94] angelehnte, Roulette-basierte Auswahl getroffen. Jedem der 200 Chromosomen wird ein Bereich auf dem (gedachten) Rouletterad zugeteilt. Je besser die Makespan ist, desto größer ist der zugeordnete Bereich und umso größer ist die Wahrscheinlichkeit, dass dieses Chromosom ausgewählt wird. Ist das Rouletterad erzeugt worden, wird eine Zufallszahl im Bereich des Rouletterades gezogen. Das Chromosom, welches den Bereich abdeckt, in dem die Zufallszahl liegt, wird dann in das neue Mapping übernommen. Dieser Ablauf ist nochmals im Listing 2.8 dargestellt:

In Phase 1 (Zeilen 3 bis 7) wird für jedes Chromosom der Roulette-Basis-Wert als Quotient aus dem *fitness value* des Elitechromosoms und dem des aktuellen Chromosoms berechnet – d. h. Chromosomen mit sehr hoher Makespan erhalten kleinere Wert, solche mit kleiner Makespan entsprechend größere Werte im Bereich von $]0;1]$. Diese Werte werden in dem Vektor `r` gespeichert sowie zu dem Skalar `sum` addiert.

In Phase 2 (Zeilen 13-19) wird für jedes Chromosom die Bereichsabdeckung aus dem Produkt des in Phase 1 berechneten Basiswerts mit dem Quotienten aus 100000 und der aus Phase 1 berechneten Wertsumme berechnet. Dadurch werden die Chromosomen im nun zweiten Schritt in einen Bereich von $[0;100000[$ angeordnet. Ab dem zweiten Chromosom werden diese um die Summe der Werte der Vorgängerchromosome verschoben.

In Phase 3 (Zeilen 23 bis 30) werden schließlich 199 Zufallszahlen erzeugt und die dadurch ausgewählten Chromosomen in die neue Zielpopulation kopiert.

Listing 2.8: Pseudocode GA Scheduler Roulette

```
1  // Phase 1: Umgekehrte Verteilung in Abhaengigkeit
2  //          des besten Chromosoms.
3  sum = 0.0;
4  foreach chromosom c do
5    r[c] = fitnessValueBestChromosom / c.fitnessValue;
6    sum += r[c];
7  end foreach
8
9  // Phase 2: Erweiterung des Roulettes auf 100000
10 //          Einheiten (Summe der Roulettewerte
11 //          entspricht danach 100000). Anordnung
12 //          der Chromosomen im Roulette
13 foreach chromosom c do
14   // Bereich berechnen
15   r[c] *= ( 100000.0 / roulettesum );
16   // Anordnen
17   if( c > 0 )
18     r[c] += r[c-1];
19 end foreach
20
21 // Phase 3: Rien ne va plus. Zufallszahl ziehen
22 //          und Chromosom auswaehlen
23 for i = 0 to CHROMOSOMCOUNT - 1 do
24   rnd = random(0..100000);
25   foreach chromosom c do
26     if rnd in r[c]
27       copyChromsomToNewPopulation(c)
28       break foreach
29   end foreach
30 end for
```

Innerhalb der Auswahlphase wird sichergestellt, dass das Elitechromosom mit der aktuell besten Makespan, auf jeden Fall in die neue Population übernommen wird, vgl. hierzu auch [Ru94]. Als erstes Chromosom wird das Chromosom mit der besten Makespan und im Anschluss werden 199 Chromosomen anhand der oben

beschriebenen Roulette-Auswahltechnik in die neue Population übernommen. Somit ist sichergestellt, dass die Population das Elitechromosom enthält und immer aus 200 Chromosomen besteht.

Der zweite Schritt in der Hauptschleife ist das **Kreuzen**, dargestellt im Listing 2.7 in Zeile 5. Dazu werden zufällig zwei Chromosome ausgewählt, sowie ein zufälliger Punkt innerhalb dieser [Tr01]. Nun wird mit einer Wahrscheinlichkeit von 60 % eine Kreuzung durchgeführt, d. h. ab dieser Position werden die Maschinenzuordnungen beider Chromosome vertauscht [Tr01].

Der dritte Schritt ist die **Mutation**, dargestellt im Listing 2.7 in Zeile 6. Dazu wird zufällig ein Chromosom und eine Position innerhalb des Chromosoms ausgewählt, sowie eine zufällige neue Maschinenzuordnung erzeugt [Tr01]. Mit einer Wahrscheinlichkeit von 40 % wird diese Mutation übernommen.

Der letzte Schritt in der Schleife ist die **Evaluation**, in Listing 2.7 in Zeile 7, in der erneut die *fitness values* berechnet werden.

Ist das Haltkriterium erfüllt, wird die Hauptschleife verlassen und das beste Chromosom der Population als Taskplan übernommen. Damit ist diese Heuristik beendet.

SAScheduler: Der SA (*Simulated Annealing*) Scheduler verwendet nur ein Mapping, welches die gleiche Form wie ein Chromosom des GA Schedulers hat [Tr01]. SA verwendet eine Prozedur, die auch schlechtere Ergebnisse zulässt, um den Lösungsraum besser abzudecken. Dabei basiert die Heuristik auf einer Systemtemperatur, die sich jede Runde um einen vorgegebenen Wert verringert (*Abkühlungsrate* bzw. im englischen Original mit *cooling rate* bezeichnet) [Tr01]. Die initiale Systemtemperatur ist die Makespan des anfänglichen Mappings. Durch die Abkühlung wird es für schlechte Ergebnisse Runde für Runde schwieriger übernommen zu werden. Der grundsätzliche Ablauf ist im Codelisting 2.9 ersichtlich.

Es werden insgesamt 16 Hauptrunden (Listing 2.9, Zeilen 1-12) durchlaufen, dabei jeweils acht mit einer Abkühlungsrate von 80 % und acht mit 90 %. Jeweils die Hälfte beider Varianten wird zufällig, die andere Hälfte durch das Ergebnis eines Min-Min Scheduler-Laufes erzeugt (*seeding*). Dieser Vorgang der **Generierung** erfolgt im Listing 2.9 in Zeile 2. Dann folgt die eigentliche Bearbeitung des aktuellen Mappings in einer weiteren Schleife, siehe Codelisting 2.9 Zeilen 3-11. Es wird eine **Mutation** des Mappings – ähnlich der Mutation bei GA – in Zeile 4 des Listings 2.9 durchgeführt.

Listing 2.9: Pseudocode SA Scheduler

```
1  for(round=0;round<16;round++) {
2    generation();
3    while(stopping criteria not met) {
4      mutation();
5      if(mutation suceeded)
6        iterations_with_same_makespan=0;
7        systemTemperature *= coolingRate;
8      else {
9        iterations_with_same_makespen++;
10     }
11   }
12 }
13 generate_schedule_with_bestMapping();
```

Die Entscheidung, ob das Mapping übernommen wird, richtet sich nach [Co96] und hängt von der Makespan des neuen Mappings ab. Ist diese geringer als die bisherige, wird das neue Mapping übernommen. Ist sie schlechter, wird eine Zufallszahl $z \in [0; 1[$ erzeugt. Zudem wird eine Zahl

$$y = \frac{1}{1 + e^{\frac{\text{old makespan} - \text{new makespan}}{\text{system temperature}}}}$$

gebildet. Ist nun $z > y$ wird das schlechtere Mapping übernommen, ansonsten verworfen und das alte Mapping beibehalten [Tr01].

Bei Lösungen mit ähnlichen Makespans oder hoher Systemtemperatur geht $y \rightarrow 0.5$, woraus folgt, dass schlechtere Lösungen mit einer Wahrscheinlichkeit von etwa 50 % übernommen werden [Tr01]. Sind die Makespans sehr unterschiedlich oder die Systemtemperatur sehr niedrig, d. h. $y \rightarrow 1$, werden schlechte Lösungen viel wahrscheinlicher verworfen [Tr01].

Wenn die Mutation geglückt ist, wird die Variable, welche die Anzahl der Iterationen mit der gleichen Makespan enthält auf 0 zurückgesetzt (siehe Listing 2.9 Zeile 6). Zudem wird in Zeile 7 in Listing 2.9 der Wert der Systemtemperatur auf die Abkühlungsrate (80 % oder 90 %, je nach Hauptrunde) reduziert. Für den Fall, dass die Mutation nicht durchgeführt wurde bzw. keine Änderung am Mapping verursacht hat (dies ist der Fall, wenn die neue zufällig gewählte Maschine gleich der bisherigen war), wird die Variable, welche die Anzahl der Iterationen mit der gleichen

Makespan enthält, um eins erhöht (siehe Listing 2.9 Zeile 9).

Jede Runde stoppt, wenn entweder 200 Iterationen mit dem gleichen Mapping erreicht wurden oder die Systemtemperatur gegen 0 geht (wird bei 10^{-200} angenommen) [Tr01].

Nach den 16 Hauptdurchläufen wird das beste Mapping als Taskplan übernommen, im Codelisting 2.9 in Zeile 13 dargestellt.

GSAScheduler: Nach [Tr01] handelt es sich bei *Genetic Simulated Annealing* um eine Kombination aus GA und SA. Grundsätzlich verläuft GSA wie der GA-Algorithmus. Allerdings wird für den Auswahlprozess (*selection*) der SA Kühlplan mit der Systemtemperatur verwendet und ein vereinfachter SA-Entscheidungsprozess, um neue Chromosomen anzunehmen oder zu verwerfen.

Nach [Sh96] wird die initiale Systemtemperatur auf die durchschnittliche Makespan der initialen Population gesetzt und während jeder Iteration auf 90 % gesenkt. Die Implementierung der genetischen Operationen *crossover* und *mutation* entspricht im Wesentlichen den Operationen des GA-Algorithmus. Allerdings werden Operationen für alle Chromosomen der Population durchgeführt und die Ergebnisse jeweils in einem Vektor gespeichert. Die Entscheidung, ob die Änderung übernommen wird, erfolgt dann in der *Selection*-Phase.

Innerhalb der *Selection*-Phase wird für jedes Chromosom das Ergebnis des Crossovers angewendet. Das Ergebnis, die neue Makespan, wird gespeichert. Anschließend wird auf das Original-Chromosom das Ergebnis der Mutation angewendet und die neue Makespan gesichert. Dann werden die Ergebnisse mittels *Voting* überprüft. Dabei wird in einem ersten Schritt das Ergebnis der Kreuzung mit der Original-Makespan verglichen. Ist die Crossover-Makespan kleiner oder gleich der Original-Makespan zuzüglich der Systemtemperatur, wird für das gekreuzte Chromosom ein Voting durchgeführt, andernfalls für das Original-Chromosom. Selbiges wird für das mutierte Chromosom durchgeführt. Abschließend werden noch die Makespans des gekreuzten Chromosoms mit dem des mutierten verglichen. Ist die Crossover-Makespan kleiner oder gleich der Mutation-Makespan wird für das gekreuzte Chromosom ein Voting durchgeführt, andernfalls für das mutierte. Dieses Vorgehen hält sich grob an [Ch98].

Wurde zweimal für eines der Chromosomen (Original, Crossover oder Mutation) ein Voting durchgeführt, wird das entsprechende Chromosom übernommen. Wurde für jedes Chromosom einmal ein Voting durchgeführt, bleibt das Original-Chromosom erhalten [Ch98].

Nach [Tr01] wird der Algorithmus gestoppt, wenn sich das Elitechromosom in den letzten 150 Iterationen nicht geändert hat oder insgesamt 1000 Iterationen erreicht wurden.

TabuScheduler: Der Tabu Scheduler sucht den Lösungsraum ab und speichert dabei Informationen über die Bereiche, die schon durchsucht worden sind, um dort nicht nochmals einen Suchlauf zu starten [Tr01]. Eine Lösung im Sinne von Tabu hat die gleiche Darstellung wie ein Chromosom des GA-Schedulers. Dabei handelt es sich beim ersten Mapping um eine zufallsgenerierte Lösung.

Listing 2.10: Pseudocode Tabu Scheduler [Tr01]

```
1  LOOP:
2     for(int ti=0;ti<getTaskCount();ti++) {
3        for(int mi=0;mi<getMachineCount();mi++) {
4           for(int tj=ti;tj<getTaskCount();tj++) {
5              for(int mj=0;mj<getMachineCount();mj++) {
6                 if(ti==tj) {
7                    mapping[tj]=mj;
8                 } else {
9                    mapping[ti]=mi;
10                   mapping[tj]=mj;
11                }
12                if(new solution is better) {
13                  replace old solution with new solution;
14                  successful_hops++;
15                  goto LOOP;
16                }
17                if(successful_hops==hoplimit)
18                  goto END:
19             }
20          }
21       }
22    }
23 END:
```

Der grundlegende Ablauf des Tabu Schedulers ist wie folgt: Das aktuelle Mapping wird im Rahmen einer Short-Hop-Prozedur manipuliert. Ziel ist es, das nächstlie-

gende lokale Minimum der Lösung zu finden [Tr01]. Dabei wird für jedes Paar von Tasks jedes mögliche Paar von Maschinenzuordnungen geprüft, während der Rest der Tasks/Maschinen-Zuordnungen unverändert bleibt.

In Listing 2.10 ist der vereinfachte Pseudocode der Short-Hop-Prozedur abgebildet. Die Implementierung erfolgte nach den Vorgaben in [Tr01]. Es existieren insgesamt vier ineinander verschachtelte Schleifen, die der Reihe nach für jedes Taskpaar jedes mögliche Maschinenzuordnungspaar durchlaufen, solange die Anzahl der erfolgreichen Short-Hops `successful_hops` noch nicht das Hop-Limit erreicht hat (Zeilen 17-18).

In jedem Durchlauf werden die Maschinenzuordnungen neu gesetzt. In den Zeilen 8-11 wird der Standardfall, in dem die beiden Tasks t_i und t_j verschieden sind, abgehandelt. In diesem Fall werden diesen Tasks die neuen Maschinenzuordnungen m_i bzw. m_j zugeteilt. Für den Fall, dass es sich um den gleichen Task handelt ($t_i = t_j$), wird für diesen Task jede mögliche Maschinenzuordnung betrachtet (Zeilen 6-7). Falls sich die Lösung durch die Neuzuordnung verbessert hat und die Makespan geringer geworden ist, ersetzt diese neue Lösung das momentane Mapping und die Anzahl der erfolgreichen Short-Hops wird um eins erhöht. Die Suche beginnt ab diesem Zeitpunkt mit dem geänderten Mapping vom Anfang der Short-Hop-Prozedur.

Die Short-Hop-Prozedur bricht, wie oben erwähnt, ab, wenn die Anzahl der geglückten Short-Hops das Hop-Limit erreicht oder alle paarweisen Neuzuordnungen ohne Verbesserung überprüft worden sind.

Die Lösung des Durchlaufs wird in eine Liste, die *Tabu List*, übernommen. Diese Liste dient dazu, die schon überprüften Regionen des Lösungsraums zu speichern, um eine weitere Suche in diesem Bereich zu vermeiden. Im Anschluss daran wird ein neues Mapping im Rahmen der *Long-Hop*-Prozedur erzeugt. Dabei muss sich die Lösung von allen bisherigen Lösungen in mindestens der Hälfte der Maschinenzuordnungen unterscheiden. Ein erfolgreicher Long-Hop erhöht die Anzahl der *erfolgreichen Long-Hops* um eins. Mit dem neuen Mapping beginnt die Short-Hop-Prozedur erneut.

Die gesamte Heuristik stoppt, wenn die Summe aus erfolgreichen Short- und Long-Hops das Hop-Limit erreicht. In diesem Fall wird aus der Tabu List die beste Lösung ermittelt und als Taskplan übernommen.

Da die Ausführungszeit der Heuristik stark von der Konsistenz der Quellmatrix abhängt, erfolgt die Festlegung des Hop-Limits in Abhängigkeit von der Konsistenz [Tr01]. Jeder Task wird in der Short-Hop-Prozedur für eine Neuzuordnung auf die Maschinen in der Reihenfolge m_0, m_1, usw. geprüft. Im Falle von konsistenten

Matrizen sind dies die schnellsten Maschinen für die entsprechenden Tasks. Daraus folgt, dass folgende Permutationen der Maschinenzuordnungen erfolglos bleiben, da keine Verbesserung erreicht wird [Tr01]. Dies ist der Fall, da die Prozedur sequentiell von der besten Maschine aus sucht und dementsprechend eine größere Anzahl von nicht geglückten Short Hops bei konsistenten ETC-Matrizen erfolgt. Damit erhöht sich für diese Fälle die Ausführungszeit. Das Hop-Limit wurde für konsistente auf 1000, für partiell-konsistente auf 2000 und für inkonsistente Eingabematrizen auf 2500 gesetzt [Tr01].

AStarScheduler: Die A* Technik basiert auf einem μ-ären Baum [Tr01]. Der Wurzelknoten repräsentiert dabei eine Null-Lösung, in der noch kein Task irgendeiner Maschine zugeordnet wurde [Tr01]. Während der Baum wächst, wachsen in den Knoten partielle Lösungen heran, die einen schon existierenden Teil der Task-Maschine-Zuordnungen repräsentieren. Jeder Kindknoten ordnet einen weiteren Task einer bestimmten Maschine zu. Dabei generiert der Vaterknoten für den nächsten, in der partiellen Lösung noch nicht zugeordneten Task t_a, μ Kindknoten – einen für jede mögliche Zuordnung des Tasks auf eine der möglichen Maschinen. Im Anschluss daran wird der Vaterknoten, der seine Arbeit damit erledigt hat, inaktiv. Der so entstehende Baum muss ab einer gewissen Größe bereinigt werden, um Ausführungszeit und Speicherbedarf realistisch zu halten. So wird die Anzahl der maximal aktiven Knoten auf 256 begrenzt. Ohne diese Bereinigung würde A* eine umfassende Suche durchführen und eine optimale Lösung finden (Speicher und Zeit im Überfluss müssten dazu vorhanden sein) [Tr01].

Jedem Knoten n des Baumes ist eine Kostenfunktion $f(n)$ zugeordnet. Diese stellt die geschätzte Untergrenze der Makespan dar und beinhaltet neben der Makespan der Teillösung auch die geschätzte Laufzeit, wenn die restlichen Tasks verteilt werden. Dabei entspricht die Funktion $g(n)$ der Makespan der partiellen Lösung, d. h. $g(n)$ ist das Maximum der Fertigstellungszeiten der einzelnen Maschinen $max_{0 \leq j < \mu} mat(m_j)$, basierend auf der bisherigen Zuordnung der Teillösung. Die Funktion $h(n)$ stellt den geschätzten Unterschied zwischen der Makespan der partiellen Lösung des Knotens n und der Makespan der vollständigen Lösung, welche die Teillösung enthält, dar. Somit ist die Kostenfunktion für Knoten n nach [Tr01] wie folgt definiert:

$$f(n) = g(n) + h(n)$$

Die Funktion $h(n)$ wird nach [Tr01] durch zwei Funktionen $h_1(n)$ und $h_2(n)$ repräs-

sentiert, die jeweils einen Ansatz zur Darstellung der Untergrenze der weiteren Berechnungszeit darstellen. Sei $M = \{min_{0 \leq j < \mu}(ct(t_i, m_j)),\ \text{für alle } t_i \in U\}$. $mmct(n)$ sei das maximale Element aus M, also die *Maximal-Minimale Fertigstellungszeit*. Somit repräsentiert $mmct(n)$ die beste Makespan, die unter der unrealistischen Annahme, dass jeder Task aus U konfliktfrei der entsprechenden Maschine aus M zugeordnet werden kann, erreicht werden kann [Tr01]. Somit ist $h_1(n)$ nach [Tr01] wie folgt definiert:

$$h_1(n) = max(0, mmct(n) - g(n))$$

$sdma(n)$ ist die Summe der Unterschiede zwischen $g(n)$ und der Fertigstellungszeiten der einzelnen Maschinen nach der Ausführung der diesen zugeordneten Tasks der Teillösung. $sdma(n)$ repräsentiert die *übrigen* Zeiten, die noch verwendet werden können, ohne die Makespan der aktuellen Lösung zu verändern [Tr01].

$$sdma(n) = \sum_{j=0}^{m-1} (g(n) - mat(m_j))$$

$smet(n)$ entspricht der Summe der Tasks in U mit der minimal erwarteten Ausführungszeit.

$$smet(n) = \sum_{t_i \in U} (min_{0 \leq j < \mu}(ETC(t_i, m_j)))$$

Insgesamt stellt sich damit die Funktion $h_2(n)$ nach [Tr01] wie folgt dar:

$$h_2(n) = max\left(0, \frac{smet(n) - sdma(n)}{\mu}\right)$$

Dabei entspricht nach [Tr01] der Quotient $\frac{smet(n) - sdma(n)}{\mu}$ der minimalen Vergrößerung der Makespan, wenn die Tasks aus U ideal auf die Maschinen verteilt werden können. Insgesamt ergibt sich für $h(n)$ nach [Tr01] folgendes:

$$h(n) = max(h_1(n), h_2(n))$$

$h(n)$ stellt damit die Untergrenze der geschätzten Ausführungszeit dar, welche die verbleibenden Tasks aus U, ausgehend von der Teillösung der Knotens n, besitzen [Tr01].

Nachdem der Wurzelknoten μ Kindknoten für den Task t_0 generiert hat, erstellt der Knoten mit der geringsten geschätzten Fertigstellungszeit $f(n)$ wieder μ Kindknoten usw., bis die festgelegte Maximalzahl der Knoten erreicht wurde. Ab diesem

Zeitpunkt wird der Baum immer bereinigt, sobald ein Knoten hinzugefügt wird. Dafür wird der Knoten mit dem größten Wert aus $f(n)$ aus dem Lösungsraum entfernt.

2.3. Vorstellung des Tuners

Der Tuner dient dazu, die Ergebnisse der Schedule-Läufe der verschiedenen Heuristiken im Nachhinein zu optimieren. Dabei werden je nach verwendeter Heuristik üblicherweise Verbesserungen im Bereich von 0 % bis 10 % erreicht. Im Abschnitt 3.2 wird darauf näher eingegangen.

Die Grundfunktionalität des Tuners ist äußerst einfach. Es wird für den zu optimierenden Taskplan die Maschine mit der größten Fertigstellungszeit m_{source} sowie die mit der geringsten m_{dest} ausgewählt. Jeder Task t_i wird von m_{source} testweise auf m_{dest} verschoben. Es wird der Task ausgewählt, der die kürzeste Makespan nach dem Verschieben mit sich bringt. Wird die Makespan des Taskplans durch die Verschiebung dieses Tasks verbessert, wird der Task tatsächlich verschoben und der Vorgang beginnt erneut. Wird keine Verbesserung erreicht, bricht die Prozedur ab. Vergleiche hierzu auch Codelisting B.1 in Anhang B.

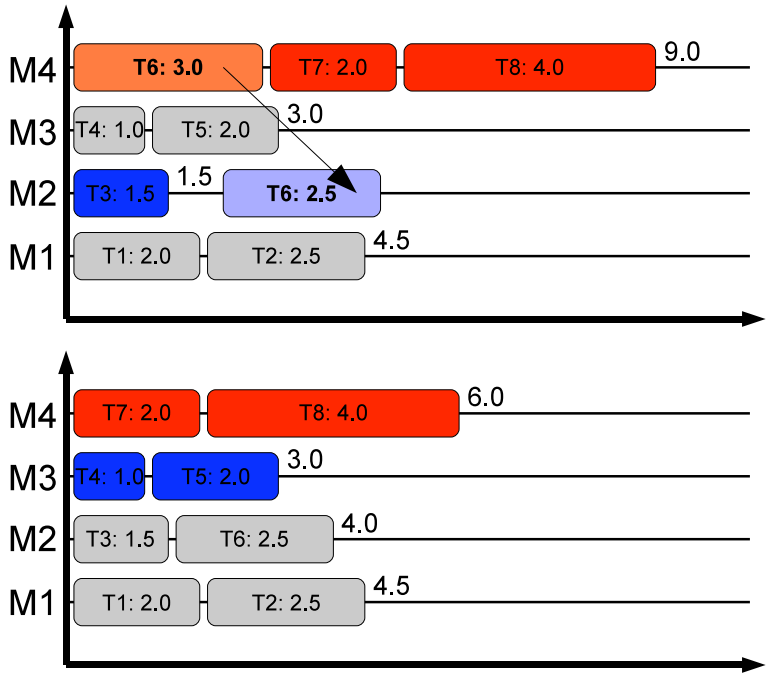

Abbildung 2.8.: Tuner Taskverschiebung

In Abbildung 2.8 ist die Verschiebung eines Tasks beispielhaft dargestellt. Tabelle

2.2 ist die zugehörige ETC-Matrix. In diesem Beispiel wird Maschine M4 als die Maschine mit der größten Fertigstellungszeit 9,0, sowie M2 mit der geringsten 1,5 identifiziert. Bei einer testweisen Verschiebung der Tasks T6, T7 und T8 von Maschine M4 zu M2 ergeben sich folgende neue Makespans: T6: 6,0, T7: 7,0, T8: 6,5. Somit erreicht die Verschiebung des Tasks T6 von Maschine M4 auf M2 die beste Makespan für diesen Durchlauf. Die Makespan hat sich auch gegenüber der bisherigen Makespan von 9,0 verbessert und folglich wird der Task tatsächlich verschoben. In der nächsten Runde wäre wieder Maschine M4 die mit der größten, nun aber Maschine M3 die mit der geringsten Fertigstellungszeit (siehe 2.8, unterer Taskplan).

	M1	M2	M3	M4
T1	2.0	2.5	1.5	2.0
T2	2.5	2.5	3.0	3.5
T3	1.5	1.5	2.0	3.0
T4	1.5	1.5	1.0	3.0
T5	2.0	2.5	2.0	3.0
T6	2.0	***2.5***	3.5	***3.0***
T7	2.0	**6.0**	3.0	**2.0**
T8	3.5	**5.0**	4.5	**4.0**

Tabelle 2.2.: ETC-Matrix zur Tuner Taskverschiebung

Für den Fall, dass mehrere Tasks bei der Verschiebung das beste Ergebnis liefern, würde die oben angeführte Grundvariante den ersten als Lösung annehmen und verschieben. Dies muss nicht die optimale Lösung sein. Für diese Fälle wurde neben der minimalen Makespan für die Taskverschiebung noch die **Ratio** der Taskverschiebung eingeführt. Die Ratio für einen Task t_i, der von Maschine m_{source} nach m_{dest} verschoben wird, ist wie folgt definiert:

$$ratio = \frac{ETC(t_i, m_{dest})}{ETC(t_i, m_{source})}$$

Existiert im Verlauf der Taskverschiebung ein Task, der die gleiche Makespan erreicht wie der aktuell gewählte Task, wird die Ratio verglichen und die Verschiebung mit der geringeren Ratio wird als bester Task für die Verschiebung festgelegt. Siehe auch hierzu Codelisting B.1 im Anhang B.

Das Problem mit gleichen Ergebnissen bei der Taskverschiebung tritt insbesondere dann auf, wenn durch die Verschiebung die Fertigstellungszeit der momentan langläufigsten Maschine unter die der bisherigen zweitlangläufigsten fällt, aber die

kürzeste diese durch die Verschiebung nicht überschreitet. Trifft dieser Umstand für mehrere Tasks zu, führt dies zu mehreren Tasks, die die gleiche verbesserte Makespan, welche der die Fertigstellungszeit der zweitlangläufigsten Maschine, erreichen. In diesem Fall ist auch keine größere Verbesserung zu erreichen – zumindest nicht für diese Runde.

Für das Beispiel aus Abbildung 2.8 würde sich eine solche Konstellation ergeben, wenn Task T8 auf der Zielmaschine M2 eine Laufzeit von 4,5 anstatt 5,0 hätte. In diesem Fall würde sowohl die Verschiebung von T6 eine Makespan von 6,0 ergeben als auch eine Verschiebung von T8. Im Falle von T6 bei einer Verschiebung auf Maschine M2, im Falle von T8 auf Maschine M4. Vergleiche hierzu Abbildung 2.9, die beide Verschiebemöglichkeiten unter Berücksichtigung des abgeänderten Matrix-Eintrags darstellt:

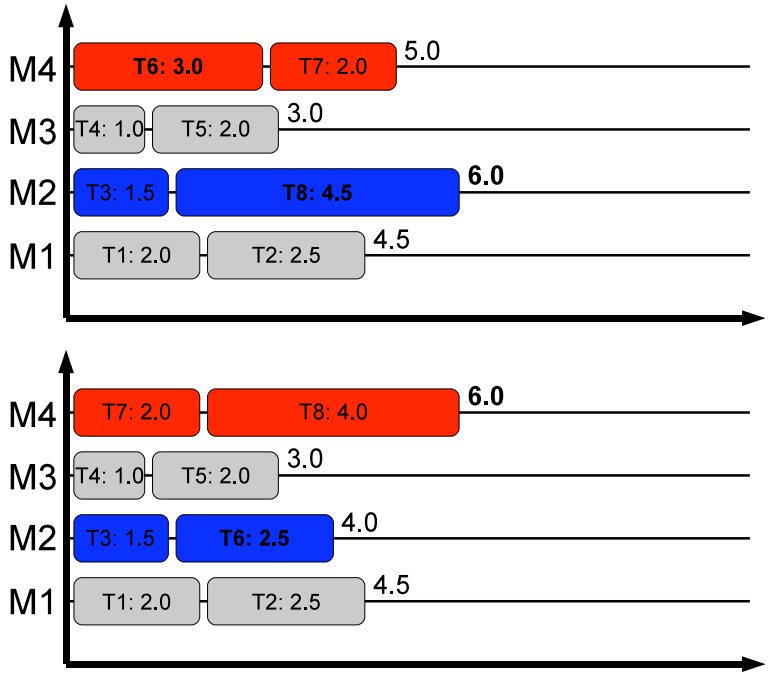

Abbildung 2.9.: Task Verschiebung mit gleicher Makespan

Nun ergibt die Ratio für Task T6 und T8 folgende Ergebnisse:

$$ratio_{T6} = \frac{ETC(T6, M2)}{ETC(T6, M4)} = \frac{2.5}{3.0} \approx 0.83$$

$$ratio_{T8} = \frac{ETC(T8, M2)}{ETC(T8, M4)} = \frac{4.5}{4.0} \approx 1.23$$

In diesem Fall wird Task T6 als der bessere Kandidat für eine Verschiebung ausgewählt, da die Ratio geringer ist als die für Task T8.

2.4. Erweiterte Optimierungsmöglichkeiten

Der einfache Tuning Algorithmus, der im vorangegangenen Abschnitt 2.3 vorgestellt wurde, bietet noch Potenzial für Verbesserungen. So werden in [Ri07] die folgenden zwei Strategien vorgeschlagen:

Bei der ersten Idee wird die Maschine betrachtet, welche die längste Laufzeit hat, die wiederum der Makespan des Metatasks entspricht. Für den Fall, dass mehrere Maschinen die längste Laufzeit haben, d. h. diese haben die gleiche Fertigstellungszeit, werden alle Maschinen in die Betrachtung mit einbezogen. Nun wird jeweils einer der Tasks dieser Maschine jeweils auf eine Maschine kürzerer Fertigstellungszeit verschoben – dies wird für alle Tasks durchgeführt. Dadurch werden $op_1 = \iota \times (\mu - 1)$ Lösungen in der direkten Nachbarschaft erzeugt [Ri07], wobei ι der Anzahl der Tasks entspricht, die der Maschine mit der längsten Laufzeit zugeordnet wurden.

Im zweiten Fall werden die Tasks nicht verschoben, sondern mit den Tasks der Maschinen mit kürzerer Laufzeit getauscht. Dadurch ändert sich die Anzahl der Tasks, die jeder Maschine zugeordnet wurden, bei dieser Optimierungsmethode nicht. Wenn ι die Anzahl der Tasks ist, die der langsamsten Maschine zugeordnet wurden, ergeben sich somit insgesamt $op_2 = \iota \times (\tau - \iota)$ Lösungen, die sich in der Nachbarschaft der durch die verwendete Heuristik gefundenen Lösung befinden [Ri07].

Insgesamt ergeben sich nach [Ri07] somit $op_1 + op_2$ mögliche Lösungen. Die Beste wird ausgewählt. Falls sich eine Verbesserung gegenüber der bisherigen Makespan ergeben hat, ersetzt die gewählte Lösung die momentane und der Algorithmus wird erneut ausgeführt. Wird keine Verbesserung erzielt, terminiert der Algorithmus.

Durch diese Optimierungsstrategie können nochmals bessere Ergebnisse erzielt werden. So kann die Min-Min Lösung auf diesem Weg so optimiert werden, dass diese bessere Ergebnisse als GA liefert. Mit der einfachen, in Abschnitt 2.3 vorgestellten, Tuninglösung kann Min-Min gerade mit GA gleichziehen. Die besseren Lösungen werden mit einer erhöhten Laufzeit des Tuners erkauft, da dieser deutlich mehr Zwischenlösungen betrachten muss. Trotz alledem ist die Laufzeit des Tuners auch in dieser Variante sehr gering, verglichen zu den Laufzeiten der komplexeren Heuristiken wie z. B. GA oder A*.

3. Analyse der Ergebnisse

Die im vorherigen Kapitel vorgestellten Heuristiken bzw. der vorgestellte Tuner wurden so implementiert, dass das fertige Programm mit einer Reihe von Argumenten aufgerufen werden kann. Über diese Argumente können Task- bzw. Maschinenanzahl sowie die jeweilige Heterogenität festgelegt werden. Zudem können die Konsistenz, die zu verwendende Heuristik und die Anzahl der Durchgänge, in der die gewählte Heuristik ausgeführt werden soll, übergeben werden.

Für die Analyse wurde jede Heuristik 100 mal je Heterogenitäts-/Konsistenz-Kombination ausgeführt. Insgesamt wurden 12 Kombinationen jeweils 100 mal durchlaufen. Die Taskanzahl wurde auf 512, die Maschinenanzahl auf 16 festgelegt.

Als Ergebnis wurden die Durchschnittswerte, der Minimal- sowie der Maximalwert der Makespan der ausgeführten Heuristik sowie der Durchschnittswert, Minimal- bzw. Maximalwert der Makespan nach der Optimierung ausgegeben. Zudem wurde die Makespan zu jedem einzelnen Lauf in beiden Varianten festgehalten. In einer Logdatei wurden detaillierte, z. T. heuristikspezifische Informationen hinterlegt.

Die Laufzeiten der Heuristiken stellten dabei im Allgemeinen den zeitintensivsten Teil der Berechnung dar, während der Optimierer selbst nur einen zu vernachlässigenden Anteil der Gesamtrechenzeit beanspruchte. Die Laufzeit des Tuners ist im Wesentlichen von der Verteilung der Tasks auf die verschiedenen Zielmaschinen abhängig. Ein Extremfall ist z. B. die MET Heuristik bei konsistenter Eingabematrix. Dort beanspruchte der Tuner etwa 75 % der Gesamtlaufzeit des Programms. Im Falle von ausgeglicheneren Verteilungen, wie dies z. B. bei Min-Min der Fall war, lag die Laufzeit des Tuners bei weniger 0,001s und damit weit unter einem Prozent der Gesamtlaufzeit der Heuristik.

Im Anhang C ist für jede Konsistenz-, Task- und Maschinenheterogenitätskombination ein Balkendiagramm zu finden. In diesen sind durchschnittlichen Makespans der durch die Heuristiken und des Tuners berechneten Metatasks abgebildet.

Das Programm wurde auf einem Zweiprozessor Intel Xeon System mit 2,8 GHz je Prozessor ausgeführt. Aufgrund der Implementierung wurde durch das Programm allerdings nur einer der Prozessoren für die Berechnung verwendet – Nebenläufigkei-

ten wurden nicht umgesetzt. Das System war mit 3 GB Arbeitsspeicher ausgestattet. Als Betriebssystem kam ein LFS 6.3 mit Linux Kernel 2.6.22.5 zum Einsatz.

Für die einzelnen Heurisiken ergaben sich in Abhängigkeit der gewählten Konsistenz die in Tabelle 3.1 und 3.2 aufgeführten Laufzeiten. Dabei steht hT für hohe Taskheterogenität, lT für geringe sowie hM für hohe Maschinenheterogenität bzw. lM für geringe. Die Werte ergaben sich als Durchschnitt aus fünf Läufen der jeweiligen Heuristik. Für A* wurde jeweils ein Lauf durchgeführt und auf volle 10 Sekunden abgerundet. Größere Schwankungen traten nur bei Tabu auf.

	Konsistent				Inkonsistent			
	hT/hM	hT/lM	lT/hM	lT/lM	hT/hM	hT/lM	lT/hM	lT/lM
OLB	0,028s	0,028s	0,028s	0,028s	0,040s	0,028s	0,040s	0,028s
MET	0,070s	0,088s	0,072s	0,088s	0,028s	0,028s	0,028s	0,028s
MCT	0,028s	0,028s	0,028s	0,028s	0,028s	0,028s	0,028s	0,028s
MinMin	0,164s	0,164s	0,164s	0,164s	0,164s	0,164s	0,164s	0,164s
MaxMin	0,164s	0,164s	0,164s	0,164s	0,164s	0,164s	0,164s	0,164s
Duplex	0,300s	0,300s	0,300s	0,300s	0,300s	0,300s	0,300s	0,300s
GA	20s	20s	20s	20s	20s	19s	21s	19s
SA	1,3s	1,0s	1,1s	1,1s	1,3s	1,1s	1,3s	1,0s
GSA	55s	55s	55s	55s	55s	55s	55s	55s
Tabu	5-10s	25-45s	3-15s	25-50s	5-60s	45-90s	15-60s	30-50s
A*	14:40m	13:50m	13:10m	13:10m	20:50m	13:50m	16:50m	20:10m

Tabelle 3.1.: Laufzeiten der einzelnen Heuristiken (konsistent/inkonsistent)

	P-konsistent			
	hT/hM	hT/lM	lT/hM	lT/lM
OLB	0,032s	0,028s	0,028s	0,028s
MET	0,036s	0,040s	0,040s	0,040s
MCT	0,028s	0,028s	0,028s	0,028s
MinMin	0,164s	0,164s	0,164s	0,164s
MaxMin	0,164s	0,164s	0,164s	0,164s
Duplex	0,300s	0,300s	0,300s	0,300s
GA	21s	20s	21s	18s
SA	1,1s	1,0s	1,2s	1,0s
GSA	55s	55s	55s	55s
Tabu	5-45s	15-60s	5-50s	80-85s
A*	17:50m	15:50m	15:10m	14:30m

Tabelle 3.2.: Laufzeiten der einzelnen Heuristiken (partiell-konsistent)

In Abschnitt 3.1 erfolgt die Analyse der Ergebnisse der einzelnen Heuristiken, während sich Abschnitt 3.2 detailliert mit der Analyse der Ergebnisse der optimierten Variante näher befasst.

3.1. Grundlegende Ergebnisse der Heuristiken

Die Ergebnisse der Scheduling-Läufe entsprachen im Weitesten den Ergebnissen, die in [Tr01] erreicht wurden. Abweichungen der durchschnittlichen Makespans ergaben sich geringfügig bei SA und Tabu, wobei die Verhältnisse bzw. Einordnung im Bezug zu den anderen Heuristiken weitestgehend erhalten blieb. Da die GSA Heuristik nicht nach [Tr01] implementiert wurde, sondern nach den Vorgaben aus [Ch98] bzw. [Sh96], weichen die Ergebnisse dieser Heuristik teilweise stark von den Ergebnissen aus [Tr01] ab. Insgesamt konnten mit GSA bessere Makespans erreicht werden. Im Folgenden werden die Ergebnisse der Scheduling-Läufe näher beleuchtet. Im Anhang C sind die Diagramme zu den verschiedenen Durchläufen zu finden.

Bei gleicher Maschinenheterogenität erreichten die Heuristiken im Allgemeinen jeweils die gleiche Reihenfolge bei der erreichten Makespan. Teilweise waren die Plätze bei Duplex bzw. Min-Min vertauscht. Allerdings wurde in allen Fällen bei Duplex die Min-Min Lösung ausgewählt. In der folgenden Abbildung 3.1 ist die sich daraus ergebende Reihung, von niedriger bis zu hoher Makespan abgebildet.

	1	2	3	4	5	6	7	8	9	10	11
c-low	GA	Duplex	GSA	Min-Min	A*	SA	Tabu	MCT	Max-Min	OLB	MET
c-high	GA	Duplex	Min-Min	GSA	A*	MCT	Tabu	SA	Max-Min	OLB	MET
i-low	GA	A*	Min-Min	Duplex	GSA	MCT	MET	SA	Tabu	Max-Min	OLB
i-high	GA	A*	Min-Min	Duplex	MCT	GSA	MET	SA	Max-Min	Tabu	OLB
pc-low	GA	Min-Min	Duplex	GSA	A*	MCT	SA	Tabu	Max-Min	OLB	MET
pc-high	GA	Duplex	Min-Min	GSA	A*	MCT	SA	Tabu / Max-Min	Tabu / Max-Min	OLB	MET

Abbildung 3.1.: Reihung der Heuristiken

In Abbildung 3.1 beziehen sich die Beschriftungen der Spalten auf den erreichten Platz. In den Beschriftungen der Zeilen steht **c** für *konsistent*, **i** für *inkonsistent* und **pc** für *partiell-konsistent*; **low** steht für *geringe* und **high** für *hohe Maschinenheterogenität*.

Bei partiell konsistenter Matrix und hoher Maschinenheterogenität war die Reihenfolge bei Tabu/Max-Min vertauscht. Bei niedriger Taskheterogenität lag Tabu vor Max-Min, umgekehrt bei hoher Taskheterogenität. In beiden Fällen lag die Makespan allerdings sehr nahe beieinander. Die Unterschiede lagen unter 5 %.

OLB hatte im konsistenten sowie partiell-konsistenten Fall jeweils das zweitschlechteste Ergebnis und landete somit auf Platz zehn. Bei inkonsistenten ETC-Matrizen erreichte OLB das schlechteste Ergebnis (Platz elf). Die Verteilung der Ergebnisse, also die erreichten Makespans, bewegten sich bei ±13 % Abweichung zum Durchschnittswert im normalen Bereich. Im Falle von konsistenten Matrizen und geringer Maschinenheterogenität lag OLB mit Max-Min etwa gleich auf, wobei Max-Min geringfügig schneller war. In allen anderen Fällen war Max-Min deutlich schneller.

Bei OLB werden die ersten μ Tasks jeweils einer noch freien Maschine zugeordnet, was nach [Tr01] bei konsistenten Matrizen dazu führt, dass einige sehr schlechte initiale Zuordnungen entstehen. So werden hier die Tasks $\mu - 2$ und $\mu - 1$ ihrer zweitschlechtesten bzw. schlechtesten Maschine zugeordnet [Tr01]. Im weiteren Verlauf macht sich die Tatsache negativ bemerkbar, dass die Tasklaufzeiten auf den einzelnen Maschinen nicht berücksichtigt werden, was abermals dazu führt, dass Tasks Maschinen zugeordnet werden, die diese sehr langsam ausführen [Tr01]. Dieser Punkt führt laut [Tr01] auch dazu, dass es bei inkonsistenten Matrizen zu den festgestellten schlechten Ergebnissen kommt. Im Gegensatz hierzu werden bei konsistenten Matrizen zumindest die Tasks auf Maschine m_0 am schnellsten, auf m_1 am zweitschnellsten usw. ausgeführt [Tr01]. Dies ist nicht der einzige Grund, dass OLB in den konsistenten und partiell-konsistenten Fällen einen leicht besseren Platz zehn erreichen konnte. OLB profitiert hier zudem davon, dass MET in gerade diesen Fällen aufgrund der bei dieser Heuristik gewählten Strategie deutlich schlechter abschneiden muss.

Bei konsistenten und partiell-konsistenten Matrizen hatte **MET** immer das schlechteste Ergebnis, im Falle von inkonsistenten Matrizen erreichte MET mit Platz sieben eine Platzierung im unteren Mittelfeld. Die Verteilung der Ergebnisse waren im Gegensatz zu OLB sehr unterschiedlich. Während die Ergebnisse in den konsistenten Fällen innerhalb von ±16 % zum Durchschnittswert lagen, gab es in den inkonsistenten und partiell-konsistenten Fällen Verteilungen, die bis zu 30 % Abweichung hatten.

Die schlechten Ergebnisse im konsistenten und partiell-konsistenten Bereich sind direkt auf die Verteilungsstrategie dieser Heuristik zurückzuführen. Im konsistenten Fall werden alle Tasks der Maschine m_0 zugeordnet, während den anderen Maschinen überhaupt keine Tasks zur Abarbeitung zugeordnet werden. Im Falle von partiell-konsistenten Matrizen werden immerhin noch etwa 50% der Tasks der Maschine m_0 zugeordnet [Tr01]. Die deutlich besseren Ergebnisse im inkonsistenten Fall können auf die Tatsache zurückgeführt werden, dass die Maschinen, die bestimmte Tasks am schnellsten ausführen können, besser in der ETC-Matrix verteilt sind und somit im Umkehrschluss auch die Tasks besser auf die Maschinen verteilt werden [Tr01].

MCT erreichte immer einen guten Platz im Mittelfeld (Platz sechs und einmal Platz fünf). Eine Ausnahme bildet der konsistente Fall mit geringer Maschinenheterogenität. Hier erreichte MCT nur Platz acht. Allerdings liegen hier die Konkurrenten MCT, SA, Tabu und A* sehr nahe beieinander. Die Abstände liegen unter 2 %, im Falle von Tabu sogar unter 1 %. Die Verteilung der Makespans um den Durchschnittswert liegen bei etwa ±15 %.

Die Heuristik erreicht im Vergleich zu den beiden anderen einfachen Algorithmen bessere Ergebnisse, kann aber nicht mit aufwändigeren Algorithmen wie Min-Min mithalten. MCT iteriert einmal durch die ETC-Matrix und kann nach [Tr01] daher nur sehr eingeschränkte Zuordnungsentscheidungen treffen. Die Tasks werden in der Reihenfolge, in der sie in der Matrix auftauchen, in den Taskplan einsortiert. Intelligente globale Entscheidungen, wie diese bei Min-Min oder A* möglich sind, kann MCT nicht treffen [Tr01]. Bei inkonsistenten und partiell-konsistenten Matrizen konnte MCT ausnutzen, dass die Maschinen, welche die Tasks am schnellsten ausführen, besser verteilt waren. Daher stieg die Wahrscheinlichkeit, dass ein Task einer Maschine zugeordnet wurde, die diesen schneller ausführen konnte [Tr01]. Daher konnte MCT in diesen Fällen zwar kaum eine Verbesserung in der Reihung erreichen, der Abstand zu den Hauptkonkurrenten SA und Tabu konnte aber insbesondere bei inkonsistenten Fällen vergrößert werden.

Min-Min erreichte stets gute Plätze im vorderen Bereich. Da Duplex immer die Min-Min Lösung ergab, teilten sich die beiden Min-Min Lösungen immer zwei Positionen in der Reihung. Bei konsistenten sowie partiell-konsistenten Eingabematrizen erreichte Min-Min den zweiten und dritten Platz, außer bei konsistent und niedriger Maschinenheterogenität. Hierbei belegte Min-Min die Plätze zwei und vier. In den Fällen der inkonsistenten Matrizen schob sich A* zwischen GA und Min-Min und

es konnten daher nur die Plätze drei bzw. vier erreicht werden.

Insgesamt handelt es sich durchwegs um sehr gute Ergebnisse, die Min-Min trotz seiner, im Vergleich zu den komplexeren Heuristiken, niedrigen Laufzeit, erreichen konnte. So ist der Abstand vom Spitzenreiter GA immer unter 10 %. Die Verteilung der Ergebnisse im Vergleich zur Durchschnitts-Makespan bewegten sich zwischen etwa -15 % und +20 %.

Mit dieser Heuristik war es möglich zur Verteilung der Tasks auf die einzelnen Maschinen globale intelligente Entscheidungen zu treffen und dadurch die Fertigstellungszeit zu minimieren [Tr01]. Die Maschinen konnten sehr gut ausgelastet werden. Im Vergleich zu Max-Min erreichte Min-Min deutlich bessere Ergebnisse.

Max-Min konnte im Gegensatz zu Min-Min bei weitem keine so guten Ergebnisse erreichen und rangierte immer im letzten Drittel auf Platz neun, bei inkonsistenter ETC-Matrix und niedriger Maschinenheterogenität sogar auf Platz zehn. Im Falle von partiell-konsistenten ETC-Matrizen und hoher Maschinenheterogenität abwechselnd auf Platz acht und neun. Die Abweichungen vom Durchschnittswert bewegten sich zwischen etwa ±15 %.

Laut [Tr01] ändert sich die Verfügbarkeitszeit der Maschinen durch die Zuordnung von Tasks i. A. stärker als dies bei Min-Min der Fall ist. Wie bei Min-Min wird der gegebene Task der Maschine zugeordnet, welche für diesen die beste Fertigstellungszeit verspricht [Tr01]. Allerdings wird bei Min-Min der Task eher der Maschine zugeordnet, die diesen auch am schnellsten ausführen kann, während dies bei Max-Min i. A. nicht der Fall ist [Tr01].

Nach [Tr01] ist die Wahrscheinlichkeit höher, dass zwischen den Maschinen größere Unterschiede bei den Verfügbarkeitszeiten entstehen, was zu einem *load balancing*-Effekt führt. Dies führt wiederum dazu, dass Tasks eher Maschinen zugeordnet werden, welche nicht die beste Ausführungszeit für diesen Task versprechen [Tr01]. In allen Konsistenzklassen erreichte Max-Min schlechte Ergebnisse. Im Falle von konsistenten Matrizen war Max-Min um den Faktor 1,25 bis 1,45 langsamer als Min-Min, im Falle von inkonsistenten Matrizen um den Faktor 1,80 bis 2,05 und bei partiell-konsistenten um den Faktor 1,65 bis 1,95.

Duplex wählte immer die Min-Min Lösung aus. Es gab keinen Fall, in der Max-Min ein besseres Ergebnis liefern konnte. Dementsprechend verhielt sich Duplex genau wie die oben beschriebene Min-Min Heuristik und teilte sich mit dieser auch immer zwei Plätze.

GA erreichte durchwegs über alle Konsistenzklassen hinweg das beste Ergebnis. Die Abweichungen vom Durchschnittswert lagen zwischen etwa -15 % und +25 %.

In allen Fällen wurde die Population ausgewählt, die mit dem Min-Min Lösung als erstes Chromosom initialisiert worden war. Durch die genetischen Operation *crossover* und *mutation* konnte das Min-Min Ergebnis nochmals um bis zu 10 % verbessert werden. Durch genetischen Operationen verfügt GA nach [Tr01] über gute Suchfähigkeiten und kann daher bessere Ergebnisse liefern. Die Ausführungszeit war grundsätzlich unabhängig von der gewählten Heterogenitätsklasse.

Die von **SA** erbrachten Ergebnisse lagen im unteren Mittelfeld. So erreichte diese Heuristik im konsistenten Fall Platz sechs bzw. Platz acht, im inkonsistenten jeweils Platz acht und im partiell-konsistenten Platz sieben.

An die Leistungsfähigkeit von GA konnte SA damit in diesem heterogenen Umfeld nicht anknüpfen und überzeugte daher nicht. Die Operationen *crossover* und *mutation* von GA sind denen von SA in diesem Problembereich deutlich überlegen [Tr01].

In nur zwei von 1200 Fällen wurde das zufallsinitialisierte Mapping gewählt, jeweils bei hoher Task- und Maschinenheterogenität und 80 % bzw. 90 % Abkühlungsrate. In fast allen Fällen setzte sich das Mapping mit der 80 %-igen Abkühlungsrate durch, nur in 19 von 1200 Fällen die 90 %-ige.

GSA erreichte bei konsistenten Matrizen einen guten dritten bzw. vierten Platz, im inkonsistenten mit Platz fünf bzw. sechs das Mittelfeld und jeweils mit Platz vier im partiell-konsistenten einen guten Platz im vorderen Mittelfeld. Im konsistenten Fall lag GSA etwa gleichauf mit Min-Min bzw. Duplex.

Wie weiter oben schon beschrieben wurde für GSA eine andere Implementierung durchgeführt, als dies bei [Tr01] der Fall war. Insgesamt konnten bessere Ergebnisse, allerdings nicht die Leistungen von GA erzielt werden. In allen Fällen wurde die Population ausgewählt, die mit der Min-Min Lösung initialisiert worden war. Während eines Durchlaufs vergrößerte sich die Makespan deutlich, um sich dann im weiteren Verlauf während des Absinkens der Systemtemperatur wieder zu verringern und sich der Min-Min Lösung anzunähern.

Tabu konnte in den konsistenten Fällen jeweils Platz sieben erreichen, in den inkonsistenten Platz neun bei niedriger Maschinenheterogenität und Platz zehn bei hoher Maschinenheterogenität, in den partiell-konsistenten jeweils Platz acht bei

niedriger Maschinenheterogenität und Platz acht und neun bei hoher Maschinenheterogenität. Somit bewegte sich diese Heuristik je nach Heterogenitätsklasse im Mittelfeld bis unterem Drittel.

Die Laufzeiten schwankten deutlich – insbesondere bei niedriger Maschinenheterogenität dauerte ein Durchlauf teilweise sehr lange. Im konsistenten Fall wurden aufgrund der Implementierung der Short-Hop-Prozedur die meisten geglückten Short-Hops am Anfang gefunden, während zum Ende hin eine große Anzahl nicht geglückter Short-Hops die Folge war [Tr01]. Da das Abbruchkriterium die Anzahl der gesamten Hops war, folgt nach [Tr01] daraus, dass aufgrund der geringeren Anzahl von Short-Hops eine größere Anzahl von Long-Hops durchgeführt wurde und damit ein größerer Bereich des Lösungsraums durchsucht werden konnte. Daher rühren auch die besseren Ergebnisse bei konsistenter und partiell-konsistenter Matrix.

Im Gegensatz dazu wurde im inkonsistenten Fall ein kleinerer Bereich des Lösungsraums abgesucht und es wurden nur schlechtere Lösungen gefunden [Tr01]. Nach [Tr01] führte die sequentielle Prozedur zur Erzeugung der Short-Hops vereint mit der inkonsistenten Struktur der Eingabe-Matrix dazu, das mehr geglückte Short-Hops durchgeführt wurden, als nicht geglückte. Es gab also deutlich mehr marginale Verbesserungen der Zwischenergebnisse und daher wurde das Hop-Limit, trotz Erhöhung, eher durch geglückte Short-Hops erreicht [Tr01]. Die Ratio aus geglücktem Short-Hops zu geglückten Long-Hops vergrößerte sich verglichen mit dem konsistenten bzw. partiell-konsistenten Fall [Tr01].

A* hatte von allen Heuristiken die mit Abstand größte Laufzeit. So benötigt dieser Algorithmus für eine 512×16 Matrix etwa zwischen 13 und 20 Minuten. Trotz dieser langen Laufzeiten konnte A* in den Fällen konsistenter und partiell-konsistenter Eingabematrizen nur einen Platz fünf im Mittelfeld erreichten. Im Falle von inkonsistenten Matrizen war das Ergebnis besser und A* konnte jeweils Platz zwei vor Min-Min und hinter GA belegen. Die Abweichungen vom Durchschnitt lagen bei etwa ± 15 %.

Nach [Tr01] verhindern die Funktionen $h_1(n)$ und $h_2(n)$ bessere Ergebnisse in konsistenten bzw. partiell-konsistenten Fällen. Im Gegensatz dazu erreichten die genannten Funktionen im inkonsistenten Fall akkuratere Ergebnisse, da die Maschinen mit der schnellsten Laufzeit für bestimmte Tasks besser verteilt sind [Tr01].

3.2. Analyse der Tuning Ergebnisse

In fast allen Fällen konnte durch den Einsatz des Tuners zumindest eine leichte Verbesserung der Makespan erreicht werden. Allerdings gab es auch Heuristiken, bei denen sich fast keines der erreichten Ergebnisse verbessern ließ. Interessanterweise hing es wenig davon ab, wie gut die berechnete Makespan der Heuristik war. So konnten die schlechten Ergebnisse von Max-Min kaum verbessert werden, während die schon relativ guten Ergebnisse von Min-Min nochmals um ein ganzes Stück nach vorn verschoben werden konnten. So zog Min-Min nach dem Tuning sogar mit den Ergebnissen von GA gleich.

Wie bei den Läufen der Heuristik an sich festgestellt wurde, hat die Wahl der Taskheterogenität, bei gleicher Maschinenheterogenität und Konsistenz, kaum einen Einfluss auf das Verbesserungspotenzial. Deshalb sind die Ergebnisse wieder zusammengefasst worden. Wenn folglich von einer 5 %-igen Verbesserung bei hoher Maschinenheterogenität und inkonsistenten Matrizen die Rede ist, sind beide Fälle der Taskheterogenität mit einbezogen. In Fällen, bei denen sich doch geringfügige Änderungen je nach Taskheterogenität ergaben, wird explizit darauf hingewiesen.

In Abbildung 3.2 ist ersichtlich, wie sich die Reihung der Heuristiken nach dem Lauf des Tuners geändert hat.

	1	2	3	4	5	6	7	8	9	10	11
c-low	GA (0)	Min-Min (+2)	Duplex (-1)	GSA (-1)	A* (0)	SA (0)	MCT (+1)	Tabu (-1)	MET (+2)	OLB (0)	Max-Min (-2)
c-high	GA (0)	Min-Min (+1)	Duplex (-1)	GSA (0)	A* (0)	MET (+5)	MCT(-1) SA(+1)	MCT(-2) SA(0)	Tabu (-2)	Max-Min (-1)	OLB (-1)
i-low	Min-Min (+2)	Duplex (+2)	GA (-2)	A* (-2)	MET (+2)	GSA (-1)	MCT (-1)	SA (0)	Tabu (0)	Max-Min (0)	OLB (0)
i-high	Min-Min (+2)	Duplex (+2)	GA (-2)	A* (-2)	MET (+2)	MCT (-1)	GSA (-1)	SA (0)	Max-Min (0)	Tabu (0)	OLB (0)
pc-low	Min-Min (+1)	GA (-1)	Duplex (0)	GSA (0)	A* (0)	MCT (0)	SA (0)	MET (+3)	Tabu (-1)	Max-Min (-1)	OLB (-1)
pc-high	Min-Min (+2)	Duplex (0)	GA (-2)	GSA (0)	A* (0)	MCT (0)	MET (+4)	SA (-1)	Max-Min Tabu(-1)	Max-Min Tabu(-1)	OLB (-1)

Abbildung 3.2.: Reihung der Heuristiken nach dem Tuning

In Abbildung 3.2 beziehen sich die Beschriftungen der Spalten auf den erreichten Platz. In den Beschriftungen der Zeilen steht **c** für *konsistent*, **i** für *inkonsistent* und **pc** für *partiell-konsistent*; **low** steht für *geringe* und **high** für *hohe Maschinenheterogenität*. Die Zahl in den Klammern bei der jeweiligen Heuristik gibt die Verschiebung der Heuristik in der Reihung nach dem Tuning an. Ein Pluszeichen

entspricht dabei einer Verbesserung, ein Minuszeichen einer Verschlechterung um den angegeben Wert in der Reihung.

In den Fällen, in denen Duplex und Min-Min je Taskheterogenität vertauschte Plätze hatten, wurde Min-Min vorgezogen. Die Ergebnisse von GA und Min-Min (und damit auch Duplex) lagen nach der Optimierung etwa gleich auf. Deshalb gibt es auf den ersten drei Plätzen teilweise Unterschiede in der Reihenfolge bei diesen Heuristiken. Im Falle von konsistenten Eingabematrizen und hoher Maschinenheterogenität tauschten SA und MCT je nach Taskheterogenität die Plätze. Die Ergebnisse lagen jedoch in diesem Fall sehr nahe beieinander. Gleiches gilt für Tabu und Max-Min bei hoher Maschinenheterogenität und partiell-konsistenten ETC-Matrizen.

Im weiteren Verlauf werden einige Taskpläne gezeigt, die aus zwei ETC-Matrizen erzeugt wurden. Diese sind am Ende dieses Kapitels als Tabelle 3.3 und 3.4 zu finden. Dabei handelt es sich bei Tabelle 3.3 um eine konsistente und bei Tabelle 3.4 um eine partiell-konsistente ETC-Matrix.

Bei den Taskplänen wurden aus Gründen der Übersichtlichkeit die Nachkommastellen nicht berücksichtigt. Dementsprechend kann auch das Endergebnis der Gesamtlaufzeiten der Tasks auf bestimmten Maschinen vom tatsächlichen Wert abweichen. Diese Abweichungen sind allerdings gering und haben keinen Einfluss auf die Umstände, die durch die Taskpläne erklärt und visualisiert werden sollen.

OLB erreichte nach dem Tuning bei konsistenter ETC-Matrix und niedriger Maschinenheterogenität Platz zehn vor Max-Min in der Reihung, wobei beide Heuristiken in diesem Fall sehr nahe beieinander lagen. In allen anderen Fällen bildete diese Heuristik mit Platz elf und großem Abstand zu den anderen das Schlusslicht.

Bei konsistenten und partiell-konsistenten Eingabematrizen konnte OLB die Makespan auf etwa 94 % (geringe Maschinenheterogenität) bzw. etwa 90 % (hohe Maschinenheterogenität) reduzieren. Im Falle von inkonsistenten Eingabematrizen war eine Reduzierung auf etwa 91,5 % (geringe Maschinenheterogenität) und etwa 65 % (hohe Maschinenheterogenität) möglich.

OLB verteilt die Tasks auf die nächste freie Maschine, ohne die Laufzeit des Tasks zu berücksichtigen. Dies führt zum Einen zu den grundsätzlich schlechten Ergebnissen von OLB, aber auch dazu, dass nach dem Lauf der Heuristik während der Optimierung viele Kandidaten gefunden werden, die auf eine andere Maschine verschoben werden können, um somit die Makespan zu reduzieren.

Festzustellen ist, dass die Maschinenheterogenität einen direkten Einfluss auf das Verbesserungspotenzial hat. So ist bei geringer Heterogenität eine geringere Verbesserung möglich, als dies bei einer hohen der Fall wäre. Dies ist darauf zurückzuführen, dass sich bei hoher Maschinenheterogenität ungünstige Zuordnungen auch deutlich stärker auswirken und damit umgekehrt die Möglichkeit bieten, auch größere Verbesserungen durch das Tuning zu erzielen.

Bei inkonsistenten Matrizen waren im Vergleich zu den anderen beiden Fällen deutlich größere Verbesserungen möglich. Auch dies ist direkt auf die Verteilungsstrategie von OLB zurückzuführen: Da OLB die Laufzeit des Tasks auf der Zielmaschine nicht berücksichtigt, ist die Wahrscheinlichkeit, dass kein Task seiner besten/zweitbesten usw. Maschine zugeordnet wird, höher als in den Fällen konsistenter bzw. partiell-konsistenter Eingabematrizen – hier sind zumindest alle Tasks der Maschine m_0 ihrer besten Maschine zugeordnet [Tr01]. In den konsistenten Fällen konnten je Durchgang etwa 20 Tasks während der Optimierung verschoben werden, bei inkonsistenter Matrix im Falle niedriger Maschinenheterogenität etwa 100 und bei hoher sogar um die 650 Tasks.

MET konnte durch das Tuning bei konsistenter ETC-Matrix und niedriger Maschinenheterogenität von Platz elf auf Platz neun vorrücken, bei hoher Maschinenheterogenität sogar auf Platz sechs. Bei partiell-konsistenter Eingabematrix in beiden Fällen der Maschinenheterogenität von Platz elf auf Platz acht, bei inkonsistenter von Platz sieben auf Platz fünf.

Die mit Abstand größten Verbesserungen konnten bei konsistenten und partiell-konsistenten Eingabematrizen erreicht werden. So konnte die Makespan bei konsistenter Matrix auf etwa 17 % (niedrige Maschinenheterogenität) bzw. etwa 22,5 % (hohe Maschinenheterogenität) abgesenkt werden. Im Falle von partiell-konsistenten Matrizen konnten etwa 23,5 % bzw. 29,5 % erreicht werden. Bei inkonsistenten Eingabematrizen war auch eine recht hohe Verbesserung auf etwa 79 % bzw. 76 % je nach Maschinenheterogenität möglich.

MET verteilt die Tasks ausschließlich nach ihrer Laufzeit auf der Zielmaschine. Dementsprechend sind bei konsistenter Eingabematrix alle Tasks der Maschine m_0 zugeordnet, siehe Abbildung 3.3, bei partiell-konsistenter etwa die Hälfte der Tasks, siehe Abbildung 3.4. Dies führt auf der einen Seite zu extrem schlechten Ergebnissen der Heuristik, auf der anderen Seite bietet es natürlich die Möglichkeit, durch den Tuner sehr gute Verbesserungen zu erzielen.

Die Taskpläne in Abbildung 3.3 und 3.4 sind durch echte Durchläufe der Heuris-

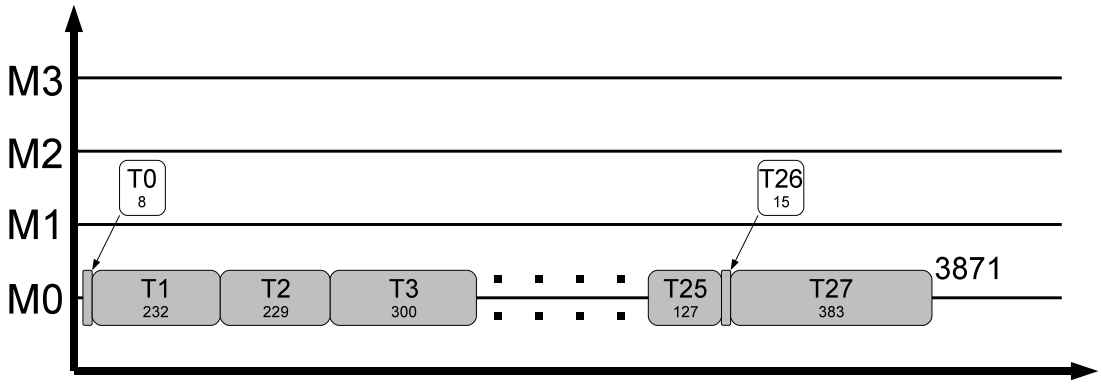

Abbildung 3.3.: Taskplan MET mit konsistenter Matrix aus Tabelle 3.3

tiken entstanden. So wurde bei dem oberen Taskplan die ETC-Matrix aus Tabelle 3.3 und bei unteren die aus Tabelle 3.4 verwendet. In den Taskplänen wurden aus Gründen der Übersichtlichkeit die Nachkommastellen der Tasklaufzeiten entfernt. Die große Zahl hinter dem T entspricht der Tasknummer, die kleine darunter der Laufzeit des Tasks.

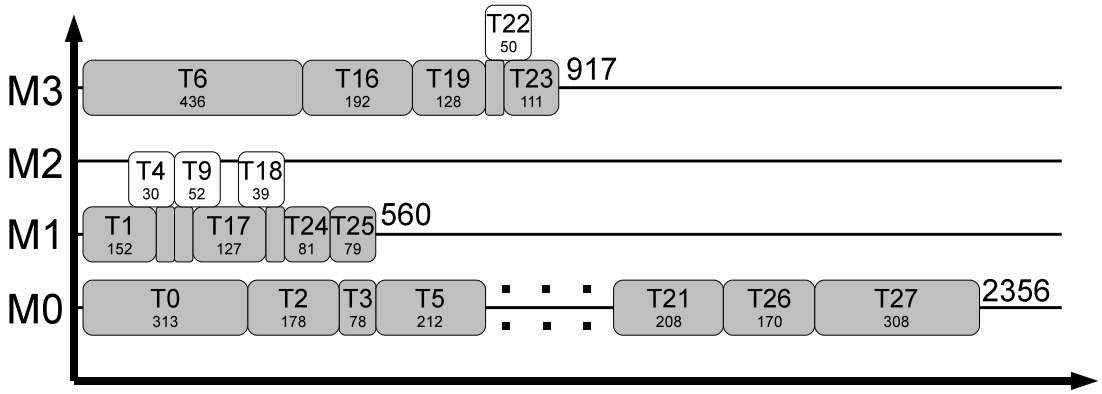

Abbildung 3.4.: Taskplan MET mit partiell-konsistenter Matrix aus Tabelle 3.4

Durch Einsatz des Tuners konnten die Ergebnisse, die bei beiden oben angeführten Taskplänen erreicht werden konnten, deutlich verbessert werden. So erreichte die konsistente Variante, Abbildung 3.3, eine Makespan von 3871,76 nach dem Lauf der Heuristik und 1380,11 nach dem Einsatz des Schedulers, also eine Verbesserung um etwa 65 %. Bei der partiell-konsistenten Variante, Abbildung 3.4, wurde durch den Einsatz der Heuristik eine Makespan von 2356,29 erreicht, nach dem Tuning 1375,96. Dies entsprach einer Verbesserung um etwa 40 %. Im Falle von konsistenter Eingabematrix waren alle Maschinen außer m_0 nicht mit Tasks belegt, im Falle

partiell-konsistenter Eingabematrizen alle geraden Maschinen außer m_0. Hier konnte nun der Tuner eine Reihe Tasks von der langläufigen Quellmaschine m_0 auf die freien Maschinen verschieben.

Deutlich bessere, aber auch nicht wirklich überzeugende Ergebnisse erreicht die Tuning-Heuristik bei inkonsistenter Eingabematrix. Auch hier macht sich bemerkbar, dass MET die Fertigstellungszeiten der Maschinen nicht berücksichtigt. Dadurch waren auch in diesem Fall deutliche Verbesserungen durch den Einsatz des Tuners möglich.

Insgesamt wurden im Schnitt bei konsistenter Eingabematrix und geringer Maschinenheterogenität etwa 300 Tasks, bei hoher Maschinenheterogenität etwa 185 Tasks verschoben. Wie erwartet, konnten bei partiell-konsistenter Matrix etwa die Hälfte der Anzahl der Taskverschiebungen erreicht werden, nämlich etwa 140 bzw. 90 Tasks je nach Heterogenität. In allen inkonsistenten Fällen konnte etwa 35 Tasks durch den Tuner verschoben werden.

MCT, **Min-Min** und **Max-Min** verteilen die Tasks anhand der Fertigstellungszeit, wobei jede der Heuristiken eine andere Strategie verfolgt. MCT verteilt die Tasks einfach in der Reihenfolge, in der diese in der ETC-Matrix auftauchen jeweils auf die Maschine, die für diesen Task im Augenblick der Verteilung die beste Fertigstellungszeit verspricht. Min-Min und Max-Min bilden dagegen jeweils für die noch zu verteilenden Tasks die Menge der minimalen Fertigstellungszeiten M (siehe hierzu auch Abschnitt 2.2.2). Min-Min wählt aus dieser Menge je Durchlauf den Task mit der kürzesten Fertigstellungszeit aus, während Max-Min den mit der längsten auswählt und ordnet diesen der entsprechenden Maschine zu.

Im Falle der hier betrachteten heterogenen Umgebungen liefert Min-Min das für diese drei Heuristiken beste Ergebnis, Max-Min das schlechteste und MCT bewegt sich zwischen den beiden. MCT ordnete sich nachvollziehbar in die Mitte der beiden Heuristiken ein. Während Min-Min jeweils die kürzeste und Max-Min jeweils längste Fertigstellungszeit aus M bevorzugt, wählt MCT die Tasks wie oben schon angesprochen in der Reihenfolge aus, wie diese in der ETC-Matrix auftauchen. Dadurch ist die Wahrscheinlichkeit größer, dass MCT eher durchschnittliche Werte für die Taskverteilung ermittelt, als die extremen, die durch Min-Min bzw. Max-Min bei der Verteilung erreicht werden.

Auch bei dem Verbesserungspotenzial durch den Einsatz des Tuners wird die Reihung von MCT, Min-Min und Max-Min bestätigt. Während bei Min-Min große Verbesserungen bis zu 10 % möglich sind, konnten bei Max-Min kaum Verbesse-

rungen durch das Tuning erreicht werden. MCT ordnet sich auch in diesem Bezug wieder zwischen Min-Min und Max-Min ein.

Min-Min, Duplex und GA lagen nach dem Tuning etwa gleich auf. Da bei Duplex immer die Min-Min Lösung gewählt wurde, handelte es sich um einen zweiten Min-Min Lauf – deshalb wird im Weiteren nicht zwischen Duplex und Min-Min unterschieden. Die Ergebnisse zwischen GA und Min-Min lagen sehr dicht beieinander und belegten die Plätze eins bis drei, siehe Abbildung 3.2. Es ergaben sich nach dem Tuning folgende Konstellationen: (1) GA belegte Platz eins, Duplex und Min-Min Platz zwei und drei, (2) Duplex und Min-Min belegten Platz eins und zwei und GA Platz drei und (3) Duplex und Min-Min schlossen GA mit den Plätzen eins und drei ein.

Min-Min konnte bei inkonsistenten Eingabematrizen in allen Fällen mit geringem Abstand vor GA auf die Plätze eins und zwei vorrücken (Fall 2). Auch bei partiell-konsistenter Matrix und hoher Maschinenheterogenität konnte Min-Min nach vorn auf die ersten beiden Plätze rutschen, wie auch bei konsistenter Eingabematrix und jeweils hoher Taskheterogenität. Nur im Fall von konsistenter Eingabematrix, niedriger Maschinen- und Taskheterogenität konnte GA seinen ersten Platz behaupten (Fall 1) – in den drei verbleibenden Fällen war GA von den beiden Min-Min Lösungen eingerahmt (Fall 3).

Bei konsistenter Eingabematrix konnte Min-Min die Makespan auf etwa 95 % reduzieren, bei inkonsistenter und partiell-konsistenter auf etwa 94 % (geringe Maschinenheterogenität) bzw. etwa 90 % (hohe Maschinenheterogenität). Damit erreichte Min-Min von den drei ähnlichen Algorithmen neben dem schon guten Grundergebnis der eigentlich Heuristik auch bei dem Tuning die besten Ergebnisse.

Min-Min verteilt die noch verbleibenden Tasks nach Fertigstellungszeiten und nimmt dabei jede Runde jeweils den Kandidaten, der die kürzeste Fertigstellungszeit für diese Runde verspricht. Somit baut Min-Min den Taskplan im Allgemeinen von kurzen zu immer längeren Tasks auf. Dies führt zu den guten Ergebnissen bei der Makespan auf der einen Seite, aber auch dazu, dass die Ausführungszeiten der Maschinen zum Ende der Verteilung hin weit weniger ausgeglichen sind, als das z. B. bei Max-Min der Fall ist. Hier ist ein guter Ansatzpunkt für den Optimierer, der dadurch eher die Möglichkeit hatte, Kandidaten für eine Verschiebung zu identifizieren.

Der Tuner konnte in den konsistenten Fällen etwa 24-27 Tasks verschieben, bei den inkonsistenten 20-22 und bei den partiell-konsistenten 21-23.

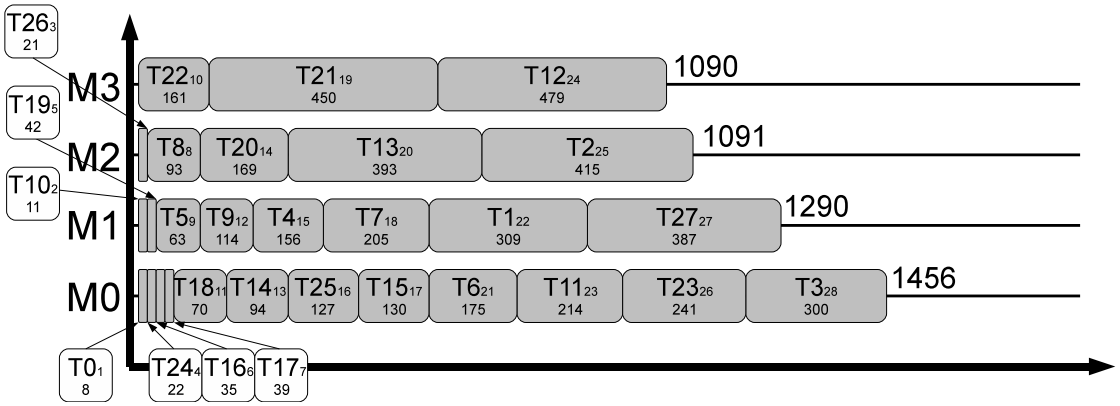

Abbildung 3.5.: Taskplan Min-Min mit konsistenter Matrix aus Tabelle 3.3

In Abbildung 3.5 ist ein beispielhafter Taskplan dargestellt, der durch Anwendung der Min-Min Heuristik auf die konsistente ETC-Matrix aus Tabelle 3.3 entstanden ist. Die große Zahl hinter dem T in der ersten Zeile der Tasks gibt die Tasknummer an, die kleine dahinter Reihenfolge der Einordnung in den Taskplan. Die untere Zahl entspricht der Ausführungszeit des Tasks.

Gut erkennbar ist einerseits, dass die Makespan geringer ist, als die von den folgenden Taskplänen, die Max-Min und MCT bei Anwendung auf die gleiche Quellmatrix erreichen konnten. Zudem ist andererseits erkennbar, dass die Verteilung der Tasks trotz der guten Makespan sehr unausgeglichen ist. So ist die Laufzeit der am stärksten ausgelasteten Maschine m_0 um annähernd die Hälfte länger als die der am wenigsten ausgelasteten m_3. Somit kann der Tuner bei diesem Taskplan nochmals deutliche Verbesserung durch seinen Einsatz erreichen. Die Laufzeit nach dem Lauf der Heuristik betrug 1462,78, nach dem Tuning 1318,59 – dies entspricht einer Verbesserung um etwa 10 %. Auch die Verteilung von kurzer zu langer Ausführungszeit der Tasks ist gut zu erkennen.

Max-Min rutschte bei konsistenter Eingabematrix jeweils von Platz neun auf Platz elf (niedrige Maschinenheterogenität) bzw. Platz zehn (hohe Maschinenheterogenität) ab. Bei inkonsistenter Eingabematrix konnte Max-Min jeweils Platz zehn bzw. neun halten. Bei partiell-konsistenter Matrix verlor Max-Min einen Platz und erreichte jeweils Platz zehn, bis auf den Fall hoher Maschinenheterogenität und geringer Taskheterogenität, in dem Max-Min nach dem Tuning Platz neun erreichte.

In allen Fällen konnten die Ergebnisse von Max-Min durch das Tuning kaum verbessert werden. In den konsistenten Fällen konnte die Makespan überhaupt nicht verbessert werden, in den partiell-konsistenten bzw. inkonsistenten war eine Verbes-

serung um maximal 0,02 % möglich. Bei den meisten Durchläufen konnte maximal ein Task verschoben und dadurch höchstens eine geringe Verbesserung der Makespan erreicht werden.

Im Gegensatz zu Min-Min wählt Max-Min aus der Menge der Tasks mit der geringsten Fertigstellungszeit jeweils den Tasks mit der längsten Fertigstellungszeit aus und teilte diesen der entsprechenden Maschine zu. Dadurch wird in den betrachteten heterogenen Umgebungen das weniger gute Ergebnis erzielt. Nach dem Durchlauf der Heuristik sind die Maschinen bei annähernd gleicher Ausführungszeit nahe bei der Makespan des Metatasks gleichmäßig ausgelastet. Hierdurch hat der Tuner kaum einen Ansatzpunkt, um eine Verbesserung durch eine Taskverschiebung zu erreichen.

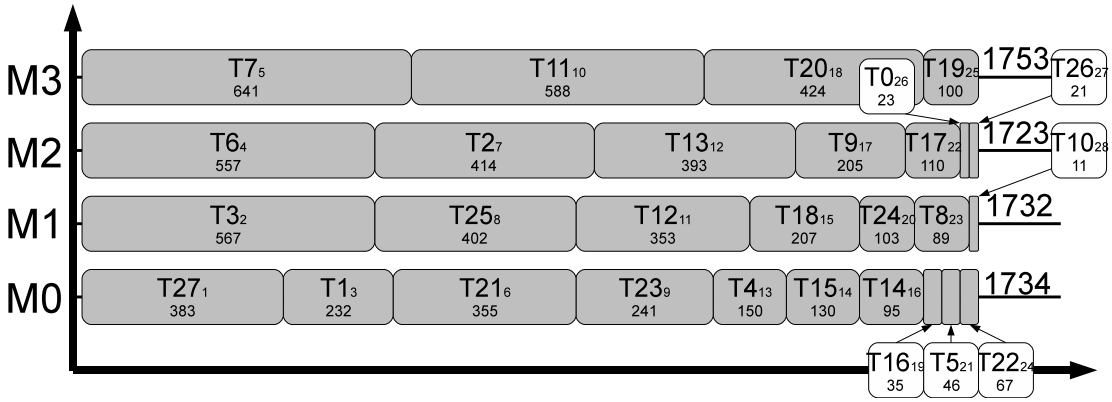

Abbildung 3.6.: Taskplan Max-Min mit konsistenter Matrix aus Tabelle 3.3

Der Taskplan in Abbildung 3.6 ist durch die Anwendung der Max-Min Heuristik auf die ETC-Matrix aus Tabelle 3.3 entstanden. Die große Nummer hinter dem T entspricht den Tasknummern, die kleine dahinter der Reihenfolge der Einsortierung in den Taskplan. Die kleine Nummer darunter entspricht der Tasklaufzeit. Es ist gut zu erkennen, wie Max-Min die Tasks von großer Laufzeit hin zu kleinerer in dem Plan verteilt hat. Dies führte einerseits dazu, dass der Taskplan am Ende sehr ausgeglichen ist und somit die Laufzeiten der einzelnen Maschinen sehr nahe beieinander liegen. Andererseits folgte daraus, dass die Laufzeiten sehr hoch sind und damit auch die Makespan.

Für den Einsatz des Tuners gab es aufgrund der Taskverteilung in diesem Fall keinen Ansatzpunkt. Die erreichte Makespan von 1754,25 konnte nicht verbessert werden.

MCT ist die dritte Heuristik, welche die Taskverschiebung anhand der Fertigstellungszeit durchführt. Anders als bei Max-Min und Min-Min werden hierbei allerdings die Tasks in der Reihenfolge verteilt, in der diese in der ETC-Matrix auftauchen (also mehr oder weniger zufällig). Dadurch erreicht MCT keinen der beiden Extremwerte von Min-Min oder Max-Min und pendelt sich sowohl in den grundlegenden Ergebnissen als auch im Verbesserungspotenzial zwischen den beiden anderen Heuristiken ein.

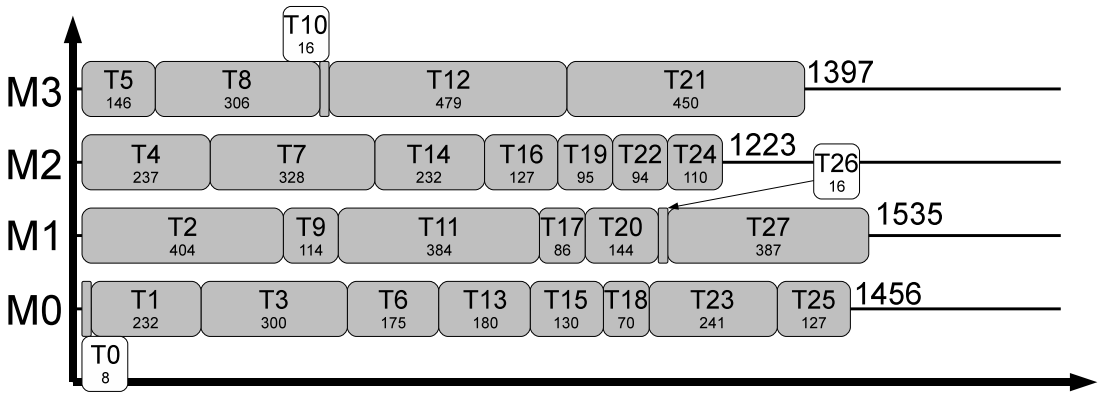

Abbildung 3.7.: Taskplan MCT mit konsistenter Matrix aus Tabelle 3.3

In Abbildung 3.7 ist der Taskplan abgebildet, der durch den Einsatz von MCT auf die ETC-Matrix aus Tabelle 3.3 entstand. Die große Zahl hinter dem T gibt die Tasknummer an, die kleine darunter die Laufzeit des Tasks. Die Reihenfolge der Einordnung in den Taskplan entspricht den Tasknummern.

Im Vergleich zu den Taskplänen, die durch den Einsatz der Min-Min- (Abbildung 3.5) bzw. Max-Min-Heuristik (Abbildung 3.6) entstanden sind, ist zu erkennen, dass sich MCT in allen Aspekten zwischen den beiden Heuristiken einordnet. So führt die Verteilung der Tasks nicht zu einer solchen Ausgeglichenheit, wie dies bei Max-Min der Fall ist, aber auch nicht zu einem solchen Ungleichgewicht, das Min-Min erreichte. Die Verteilung der Tasks ist nicht strikt von kurz nach lang oder umgekehrt, sondern durcheinander. Das Verbesserungspotenzial durch den Einsatz des Tuners lag mit etwa 5 % zwischen den Ergebnissen von Max-Min und Min-Min. So konnte MCT die Laufzeit durch den Tunereinsatz von 1541,06 auf 1459,42 verbessern.

MCT konnte die Makespan in den Testläufen bei konsistenter Eingabematrix auf etwa 98,5 % verbessern, und erreichte bei geringer Maschinenheterogenität Platz sieben (vorher acht), rutschte bei hoher Maschinenheterogenität allerdings von Platz

sechs auf Platz sieben (niedrige Taskheterogenität) bzw. Platz acht (hohe Taskheterogenität) ab.

Bei inkonsistenter Eingabematrix und geringer Maschinenheterogenität konnte das Tuning die Makespan auf etwa 96 % reduzieren, trotzdem erreichte MCT hier nur Platz sieben (vorher Platz sechs). Bei hoher Maschinenheterogenität wurde die Makespan um etwa 7 % verbessert, allerdings rutschte MCT auch hier um einen Platz auf sechs ab. In allen Fällen von partiell-konsistenter Eingabematrix konnte MCT durch das Tuning um etwa 2,5 bis 3,0 % verbessert werden. MCT hielt hier in allen Fällen sowohl vor als auch nach der Optimierung Platz sechs.

Im Falle von **Duplex** wurde immer die Min-Min Lösung ausgewählt. Deshalb treffen die oben beschriebenen Ergebnisse von Min-Min eins zu eins auf Duplex zu.

GA rutschte nach dem Tuning bei konsistenter ETC-Matrix, niedriger Maschinen- und Taskheterogenität auf Platz zwei ab, bei hoher Taskheterogenität auf Platz drei. Bei hoher Maschinenheterogenität landete die Heuristik auf Platz zwei bzw. drei. Im Falle von inkonsistenten Eingabematrizen rutschte GA in allen Fällen auf den dritten Platz ab, bei partiell-konsistenten auf Platz zwei (geringe Maschinenheterogenität) bzw. Platz drei (hohe Maschinenheterogenität).

In allen betrachteten Fällen wurden die Plätze unter den Min-Min-Lösungen und GA verteilt. In den Fällen, bei denen GA seinen ersten Platz abgab nahm eine der Min-Min-Lösungen (Min-Min selbst oder Duplex) diesen ein.

GA konnte durch das Tuning in den konsistenten Fällen rund 1 % Verbesserung erreichen, bei partiell-konsistenten ETC-Matrizen etwa 1,5 % und bei inkonsistenten ungefähr 2 %. Insgesamt war das Verbesserungspotenzial nicht sehr hoch und GA erreichte ungefähr die gleichen Ergebnisse wie Min-Min, benötigte dafür allerdings deutlich längere Berechnungszeiten. Die Abstände zwischen den Ergebnissen von Min-Min und GA lagen nach dem Tuning unter 2 %, in den meisten Fällen sogar deutlich unter einem Prozent.

Durch das Tuning konnten bei GA im Gegensatz zu Tabu oder Max-Min jedes Ergebnis verbessert werden. Das heißt der durchschnittliche Wert der Verbesserung liegt zwar sehr nahe zu den Durchschnittswerten der anderen beiden Heuristiken, allerdings gab es bei GA immer einen Ansatzpunkt für Verbesserungen.

SA konnte im Falle von konsistenter Eingabematrix und geringer Maschinenheterogenität Platz sechs halten. Bei hoher Maschinenheterogenität verbesserte sich SA

von Platz acht auf sieben (geringe Taskheterogenität) bzw. hielt Platz acht (hohe Taskheterogenität). Bei inkonsistenten Eingabematrizen blieb die Heuristik in allen Konsistenzfällen auf Platz acht. Bei partiell-konsistenten Matrizen und geringer Maschinenheterogenität konnte Platz sieben gehalten werden, bei hoher Maschinenheterogenität rutschte der Algorithmus von Platz sieben auf acht ab.

In allen Fällen konnte eine Verbesserung von etwa 1 % – im Vergleich zum Durchschnittswert nach dem Scheduling – erreicht werden. Etwa 5 % Verbesserung wurde im Falle von konsistenter Eingabematrix und hoher Maschinenheterogenität erreicht.

Insgesamt konnte der Tuner in der aktuellen Implementierung keine überzeugenden Verbesserungen bei dieser Heuristik liefern. SA blieb auf den schon schlechten hinteren Plätzen und es konnten nur vereinzelt geringfügige Verbesserungen erzielt werden.

GSA fiel bei konsistenter Matrix und geringer Taskheterogenität von Platz drei auf Platz vier ab, bei hoher konnte Platz vier gehalten werden. In den inkonsistenten Fällen verlor GSA jeweils einen Platz und rutschte bei geringer Maschinenheterogenität auf Platz sechs ab, bei hoher auf sieben. In allen Fällen von partiell-konsistenten Eingabematrizen konnte GSA Platz vier halten.

Das Ergebnis von GSA konnte durch das Tuning nur geringfügig im Bereich von 0,5 % bis 1,0 % verbessert werden. Somit eignet sich der verwendete Tuner wenig für die, durch GSA erbrachten, Ergebnisse.

GSA erreichte in der hier verwendeten Implementierung grundsätzlich schon recht gute Ergebnisse. Bei konsistenter Matrix wurden annähernd die Ergebnisse erreicht, die auch Min-Min erzielen konnte. Gleiches gilt für partiell-konsistente Matrizen. In den inkonsistenten Fällen war GSA grundsätzlich weiter von den Ergebnissen Min-Mins entfernt und konnte hier dementsprechend auch nach der Optimierung weniger gute Ergebnisse liefern. Das Verbesserungspotenzial durch die Optimierung war allerdings deutlich geringer als bei Min-Min. Dementsprechend wurde der Abstand zu Min-Min nach dem Einsatz des Tuners deutlich vergrößert bzw. im Vergleich zu GA gehalten.

Tabu verlor bei konsistenter Eingabematrix und geringer Maschinenheterogenität einen Platz und erreichte damit Rang acht, bei hoher Maschinenheterogenität sogar zwei Plätze und damit Platz neun. In den inkonsistenten Fällen konnte Tabu Platz neun (geringe Maschinenheterogenität) bzw. Platz zehn (hohe Maschinenheterogenität) halten. Bei partiell-konsistenter Matrix rutschte die Heuristik jeweils um einen

Platz ab und erreichte Platz neun (geringe Maschinenheterogenität) bzw. Platz neun und zehn (hohe Maschinenheterogenität, geringe bzw. hohe Taskheterogenität).

Verbesserungen waren bei Tabu kaum zu erreichen. So bewegten sich die durchschnittlichen Verbesserungen bei etwa 0,1 %. Einige der Schedulerläufe konnten durch den Tuner gar nicht verbessert werden, bei einem Großteil waren nur geringfügige Verringerungen der Makespan möglich. Somit konnte das schon anfänglich schlechtere Ergebnis dieser Heuristik mittels des Tunings kaum verbessert werden.

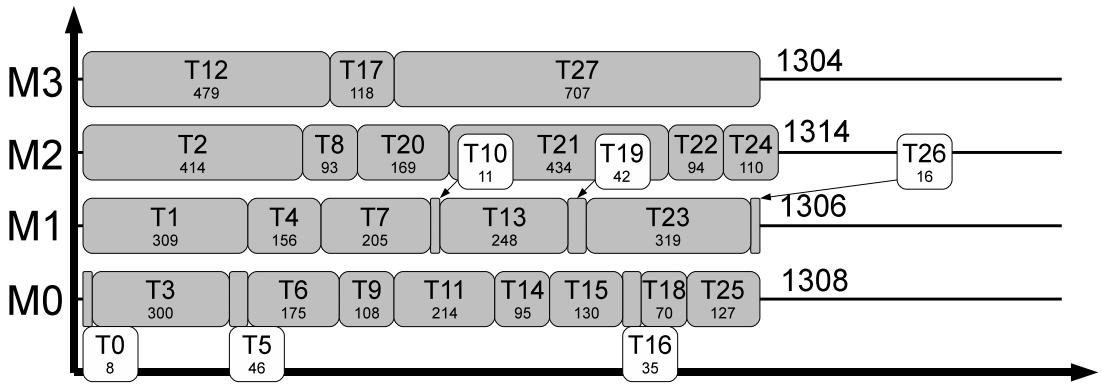

Abbildung 3.8.: Taskplan Tabu mit konsistenter Matrix aus Tabelle 3.3

Tabu führt mittels seiner Strategie schon ein ähnliches, wenn auch komplexeres, Vorgehen wie der Tuner durch. So werden Maschinenzuordnungen geändert (sprich die Tasks verschoben). Wenn durch die Verschiebung eine Verkürzung der Makespan erreicht wird, wird dieses Ergebnis weiter verwendet. Dadurch wird eine zumindest ähnliche Strategie wie beim Tuner angewendet, was nach dem Lauf von Tabu dazu führt, dass der Tuner wenig oder sogar gar nichts verbessern kann. Siehe hierzu auch die Verteilung der Tasks, wie in Abbildung 3.8 dargestellt. Hier wurde die Tabu Heuristik auf die ETC-Matrix aus Tabelle 3.3 angewandt. Die große Zahl hinter dem T entspricht der Tasknummer, die kleine darunter der Laufzeit auf dieser Maschine.

A* konnte bei konsistenten und partiell-konsistenten Matrizen den erreichten fünften Platz halten. Bei inkonsistenter Eingabematrix fiel A* von dem erreichten guten zweiten Platz auf Platz vier ab. Hier konnte sich A* nicht gegen die guten Tuning-Ergebnisse, die Min-Min erreichen konnte, durchsetzen.

Insgesamt war das Optimierungspotenzial bei A*, ähnlich wie bei GA, nicht sonderlich hoch. So konnte durch das Tuning im Falle von konsistenten und partiell-konsistenten Eingabematrizen eine Verbesserung von etwa einem Prozentpunkt, im

Falle von inkonsistenten Eingabematrizen um etwa zwei Prozentpunkte erreicht werden.

Insgesamt konnten die Ergebnisse der aufwändigeren Algorithmen wie GA, A* oder Tabu, durch das Tuning weit weniger verbessert werden, als die Ergebnisse der einfacheren Algorithmen, wie z. B. Min-Min oder MCT. Es entsteht der Eindruck, dass die aufwändigeren Algorithmen in ihrem Zielbereich weit intensiver lokale Minima ausnutzen bzw. finden konnten, als die einfachen. Somit konnte der hier verwendete Tuner kaum Verbesserungen erzielen. Interessanterweise fand der einfache Algorithmus Min-Min schon sehr gute Ergebnisse, teilweise weit besser, als die Ergebnisse der aufwändigen Heuristiken und konnte zudem noch in einem weit größeren Umfang verbessert werden.

		Maschinen			
		0	1	2	3
	0	8.63	11.13	23.34	25.02
	1	232.92	309.96	674.10	734.23
	2	229.14	404.78	414.62	575.73
	3	300.31	567.60	721.58	768.89
	4	150.33	156.22	237.15	507.80
	5	46.90	63.34	122.43	146.57
	6	175.97	406.64	557.67	667.31
	7	169.06	205.02	328.46	641.06
	8	70.42	89.65	93.94	306.13
	9	108.58	114.16	205.23	374.30
	10	8.66	11.53	16.41	16.76
	11	214.93	384.55	431.20	588.71
T	12	240.13	353.48	464.72	479.18
a	13	180.63	248.77	393.73	493.10
s	14	95.65	156.61	232.13	499.83
k	15	130.39	251.22	331.36	531.24
s	16	35.22	124.40	127.45	143.39
	17	39.90	88.72	110.30	118.96
	18	70.57	207.73	217.19	381.60
	19	34.79	42.68	95.45	100.21
	20	116.42	144.65	169.98	424.25
	21	355.20	430.24	434.56	450.43
	22	67.11	82.36	94.13	161.84
	23	241.35	319.77	470.18	582.34
	24	22.58	103.93	110.72	141.64
	25	127.25	402.55	402.73	412.03
	26	15.49	16.76	21.65	30.45
	27	383.11	387.41	621.86	707.84

Tabelle 3.3.: ETC-Matrix 28×4, konsistent

		Maschinen			
		0	1	2	3
	0	313.23	456.75	352.67	361.13
	1	164.06	152.35	437.46	763.94
	2	178.49	284.22	557.64	429.34
	3	78.59	135.67	374.19	257.21
	4	110.44	30.67	185.26	148.06
	5	212.88	456.84	470.71	570.78
	6	838.45	910.91	864.44	436.19
	7	101.83	404.08	622.80	578.81
	8	68.31	263.38	367.33	259.24
	9	150.27	52.82	253.65	177.04
	10	95.53	282.44	227.29	866.22
	11	229.46	956.27	347.07	864.07
T	12	133.01	830.33	700.71	604.41
a	13	65.90	133.75	140.61	128.80
s	14	81.73	442.77	417.48	419.84
k	15	90.76	664.27	624.50	678.74
s	16	339.27	233.75	393.08	192.96
	17	188.51	127.58	292.54	193.41
	18	40.33	39.24	195.60	180.84
	19	187.55	324.63	200.27	128.66
	20	19.09	29.06	29.63	44.36
	21	208.42	874.21	688.62	521.50
	22	176.01	229.68	234.42	50.01
	23	143.32	141.23	183.67	111.00
	24	457.84	81.98	598.66	471.98
	25	175.78	79.83	315.83	238.55
	26	170.69	469.47	709.28	616.57
	27	308.28	816.85	414.91	761.55

Tabelle 3.4.: ETC-Matrix 28×4, partiell-konsistent

4. Fazit

Es hat sich gezeigt, dass mittels eines schon relativ einfach gestalteten Optimierers deutliche Leistungsverbesserungen zu erzielen waren. Zwar konnte die GA-Heuristik, welche die besten Grundergebnisse lieferte, nur um ein bis zwei Prozentpunkte verbessert werden, allerdings konnten dafür bei Min-Min deutlich bessere Ergebnisse erzielt werden. So zog durch den Einsatz des einfachen Tuners die deutlich schnellere Min-Min Heuristik mit GA gleich. Die Möglichkeiten, die ein erweiterter Optimierer bietet sind nochmals größer. So erreicht eine optimierte Min-Min Heuristik bessere Ergebnisse als GA und das in einem Bruchteil der notwendigen Zeit [Ri07].

Weitere Verfeinerungen des Tuningprozesses können noch weitere Verbesserungen nach sich ziehen und stellen sicher ein interessantes Thema dar, das weiter verfolgt werden sollte.

A. OO Modell

Das folgende objektorientierte Modell beschreibt die grundlegende Struktur, die für die Implementierung des Tuners, der Scheduler und der Basis-Datenstrukturen gewählt wurde. Es existieren zwei voneinander unabhängige Klassenhierarchien. Abbildung A.1 stellt die relativ einfache Hierarchie der Matrix- bzw. ETC-Matrix-Datenstruktur dar. Siehe hierzu auch Abschnitt 2.1.

In Abbildung A.2 wird die zweite, komplexere Hierarchie des abstrakten Basisschedulers sowie die davon abgeleiteten vollständigen Scheduler beschrieben. Dabei wird jeweils in den abgeleiteten Schedulern die entsprechende Heuristik implementiert (siehe Abschnitt 2.2), während allgemeine Funktionalitäten, wie z. B. der Tuner (siehe Abschnitt 2.3), in der Basisklasse umgesetzt werden.

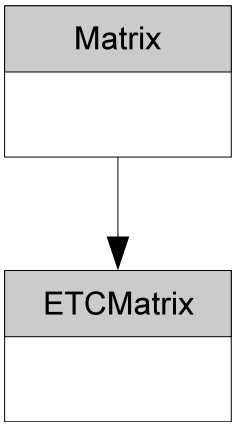

Abbildung A.1.: Klassenhierarchie der Matrix bzw. ETC-Matrix

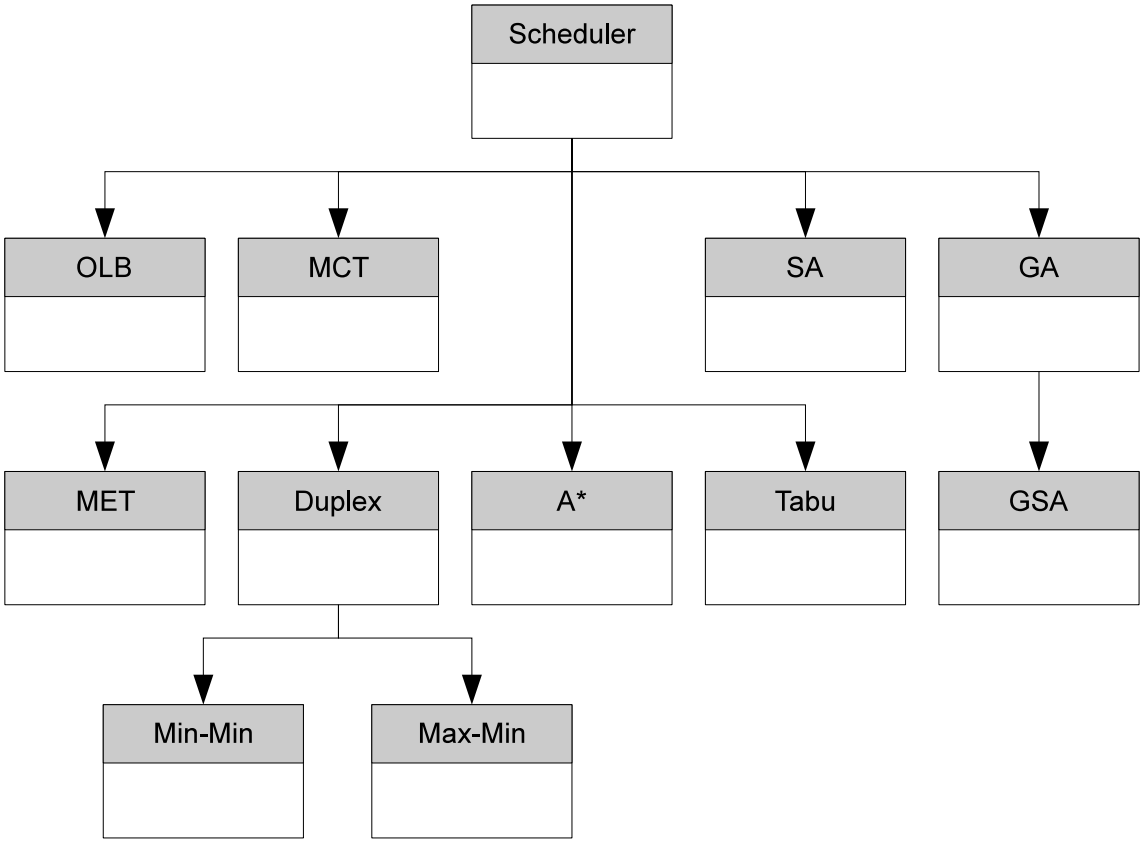

Abbildung A.2.: Klassenhierarchie der Scheduler

Neben den oben beschrieben Klassenhierarchien existieren noch einige, sehr einfache Funktionen, die nicht innerhalb eines Klassenmodells implementiert wurden. Ein Beispiel ist die Funktion `void randomize()`, welche den Zufallszahlengenerator initialisiert.

B. Vereinfachter Tuner Code

Listing B.1: Vereinfachter Tuner Source Code

```
1  void Scheduler::tune()
2  {
3    bool success=true;
4    vector<long double> execTimes;
5    vector<long double> ratio;
6    int BestMakespan;
7    int ShortestET;
8    int LongestET;
9
10   while(success) {
11     success=false;
12     execTimes.clear();
13     ShortestET=identifyMachineWithShortestET();
14     LongestET=identifyMachineWithLongestET();
15
16     for(int entry=0;
17       entry<this->getTaskCountOnMachine(LongestET);
18       entry++) {
19       execTimes.push_back(getVirtualMakespan(entry,
20          LongestET,ShortestET));
21       ratio.push_back(this->matrix->get(entry,
22                           ShortestET) /
23                         this->matrix->get(entry,
24                           LongestET));
25       }
26
27     BestMakespan=0;
```

```
28      for(int entry=1;entry<execTimes.size();entry++) {
29        if(execTimes[entry]<execTimes[BestMakespan])
30          BestMakespan=entry;
31        if(execTimes[entry]==execTimes[BestMakespan]&&
32           ratio[entry]<ratio[BestMakespan])
33          BestMakespan=entry;
34      }
35
36      if(execTimes[BestMakespan]<this->getMakespan()) {
37        this->moveTask(BestMakespan,LongestET,ShortestET);
38        success=true;
39      }
40    }
41  }
```

Das oben angeführte Listing B.1 stellt eine vereinfachte Version des Tuners dar. Der Code wurde auf die wesentlichen Teile reduziert, weniger wichtige Teile wurden entfernt oder durch Pseudofunktionsaufrufe ersetzt. Im Folgenden werden die wichtigen Eigenschaften des Tuners kurz beschrieben. Für eine detaillierte Beschreibung des Tuners siehe Kapitel 2.3.

Z. 3-8 Variablen, die im Tuner verwendet werden. `success` wird immer dann auf `true` gesetzt, wenn im aktuellen Durchlauf eine Verbesserung der Makespan erreicht wurde. `execTimes` enthält die Ausführungszeiten der einzelnen Maschinen bei der aktuellen Taskbelegung. `ratio` ist ein Vektor, der den Quotienten der Tasklaufzeiten auf der Zielmaschine und der Quellmaschine enthält ($ratio = \frac{ET_{(t_i,m_{dest})}}{ET_{(t_i,m_{source})}}$). `BestMakespan` enthält für den aktuellen Lauf die Taskverschiebung, die die beste Makespan mit sich bringt. `LongestET` bzw. `ShortestET` enthalten jeweils die Maschinennummern, die der Maschine mit der längsten bzw. kürzesten Ausführungszeit für diesen Durchlauf entsprechen.

Z. 10-40 Die Hauptschleife des Tuners. Diese wird solange ausgeführt, bis keine Verbesserung im Taskplan mehr erreicht werden kann, also die Variable `success` auf `false` gesetzt ist.

Z. 11-14 Initialisierungen für die aktuelle Runde. `success` wird auf false gesetzt, der Vektor `execTimes` wird geleert. Zudem werden die beiden Variablen, die die Verweise auf die Maschine mit der längsten und kürzesten Laufzeit auf die jeweiligen Maschinen gesetzt.

Z. 16-25 In dieser Schleife werden sämtliche Tasks auf der Quellmaschine, also der Maschine mit der längsten Ausführungszeit, durchlaufen und die Makespans berechnet, wenn diese Tasks auf der Zielmaschine (Maschine mit der kürzesten Laufzeit) ausgeführt werden würden. Die Ergebnisse werden im Vektor `execTimes` gespeichert. Zudem wird für jeden Task der Quotient aus Laufzeit auf der Zielmaschine und Laufzeit auf der Quellmaschine gebildet und im Vektor `ratio` gespeichert.

Z. 27-34 In dieser Schleife wird der Eintrag mit der besten Makespan aus dem Vektor `execTimes` ermittelt. Wird ein Eintrag gefunden, der die gleiche Laufzeit wie der aktuell beste Eintrag hat, wird zudem die Ratio geprüft und der Eintrag gewählt, der die bessere (kleinere) Ratio besitzt. Am Ende der Schleife beinhaltet die Variable `BestMakespan` die Eintragsnummer, die die beste Makespan hat.

Z. 36-39 Hier wird nun überprüft, ob die Makespan der besten Verschiebung auch eine Verbesserung der Makespan gegenüber dem bisherigen Taskplan erbracht hat. Ist dies der Fall, wird der Task endgültig auf die Zielmaschine verschoben und die Variable `success` auf `true` gesetzt. Wird keine Verbesserung erreicht, ist mit diesem Durchlauf der Tuner beendet.

C. Scheduling-Ergebnisse

In diesem Abschnitt sind die Diagramme zu sämtlichen Scheduling-Durchläufen aufgeführt. Je Konsistenzklasse (konsistent, inkonsistent und partiell-konsistent) gibt es vier Diagramme für jede Task-/Maschinen-Heterogenitätskombination. Abbildung C.1 zeigt ein solches Diagramm.

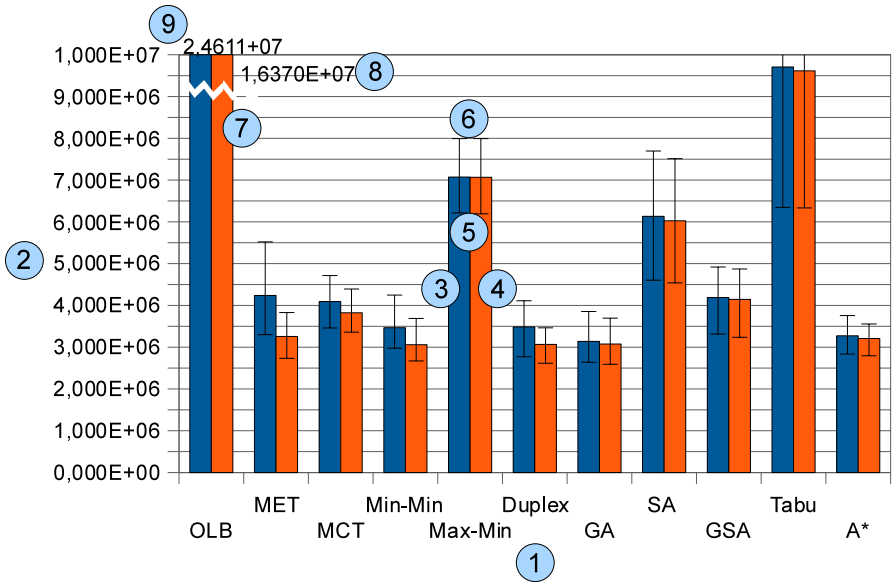

Abbildung C.1.: Diagrammbeschreibung

Auf der x-Achse sind die Heuristiken angeordnet (1), auf der y-Achse die Laufzeit (2). Je Heuristik zeigen zwei Balken die Laufzeit, also die Makespan an. Der linke, blaue Balken entspricht der durchschnittlichen Makespan der Heuristik (3), der rechte, rote Balken der durchschnittlichen Laufzeit der Heuristik nach der Optimierung (4). Die kleinen Balken zeigen die Minimal- (5) bzw. Maximal-Makespan (6) an.

Manche Ergebnisse weiche stark von den anderen Ergebnissen ab. Daher werden diese nicht 1:1 in dem Diagramm angezeigt, sondern werden *abgeschnitten* (7). Die tatsächlich erreichten Werte werden nahestehend des Balkens angezeigt, hier (8) und (9). Nummer (8) gibt dabei die Makespan des rechten, roten Balkens an, Nummer (9) die des linken, blauen.

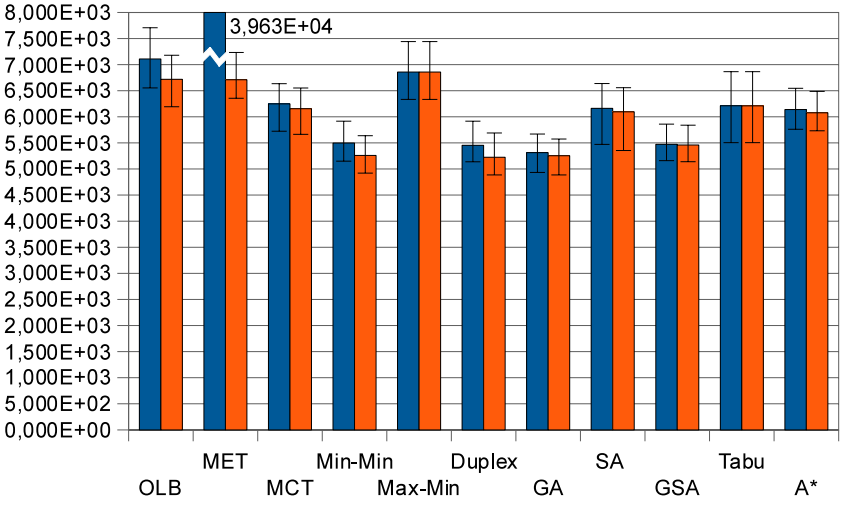

Abbildung C.2.: Low Task, Low Machine, Consistent, 512x16

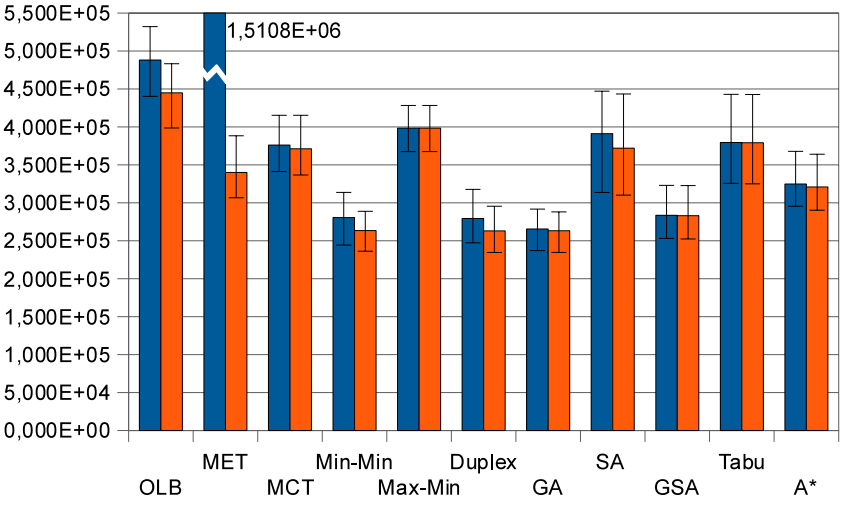

Abbildung C.3.: Low Task, High Machine, Consistent, 512x16

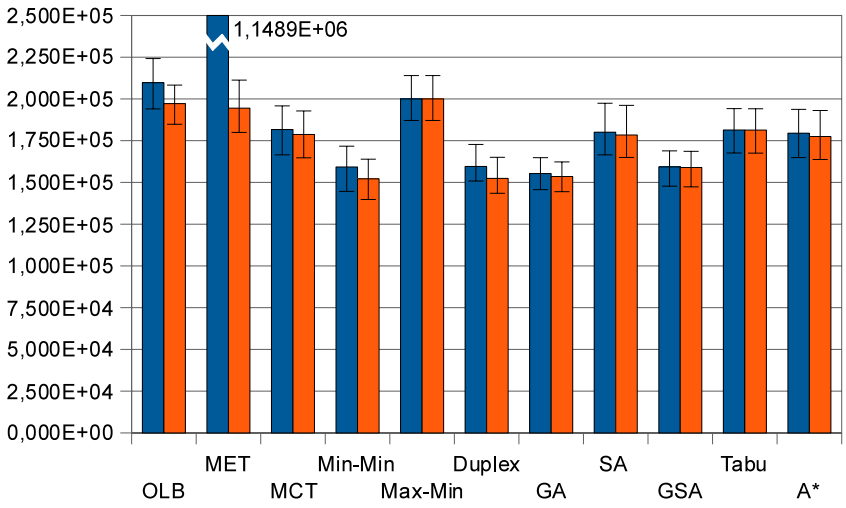

Abbildung C.4.: High Task, Low Machine, Consistent, 512x16

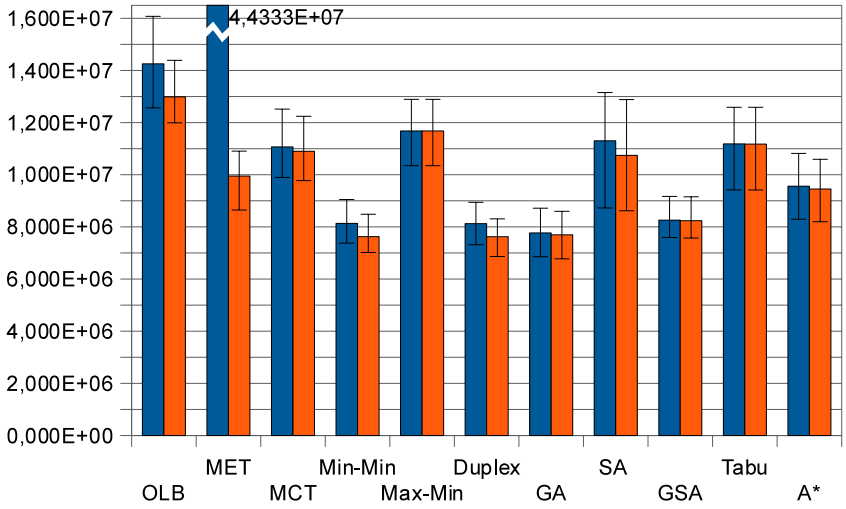

Abbildung C.5.: High Task, High Machine, Consistent, 512x16

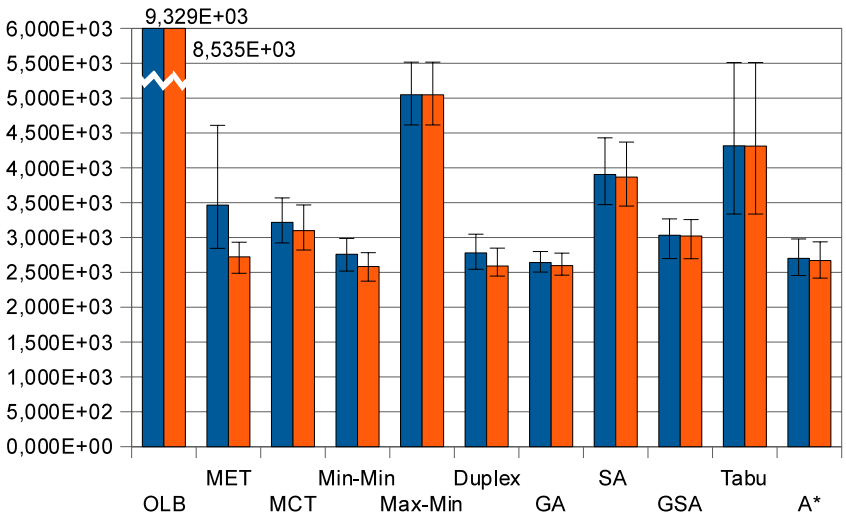

Abbildung C.6.: Low Task, Low Machine, Inconsistent, 512x16

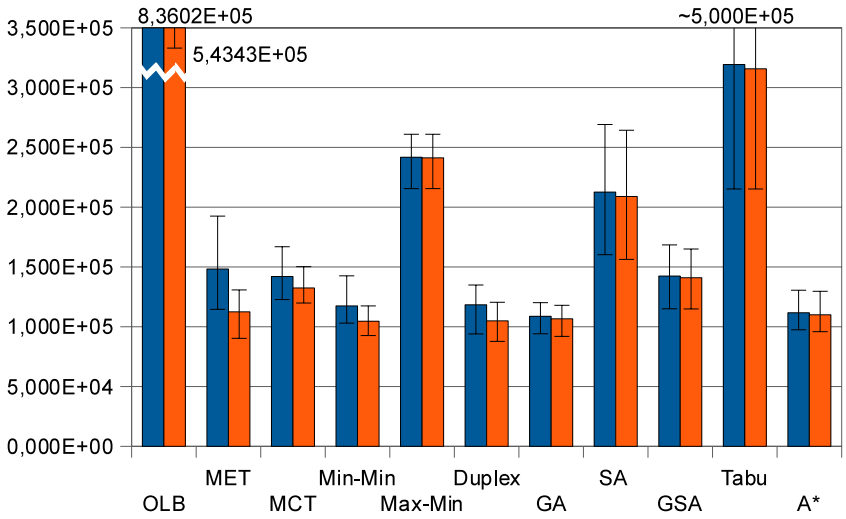

Abbildung C.7.: Low Task, High Machine, Inconsistent, 512x16

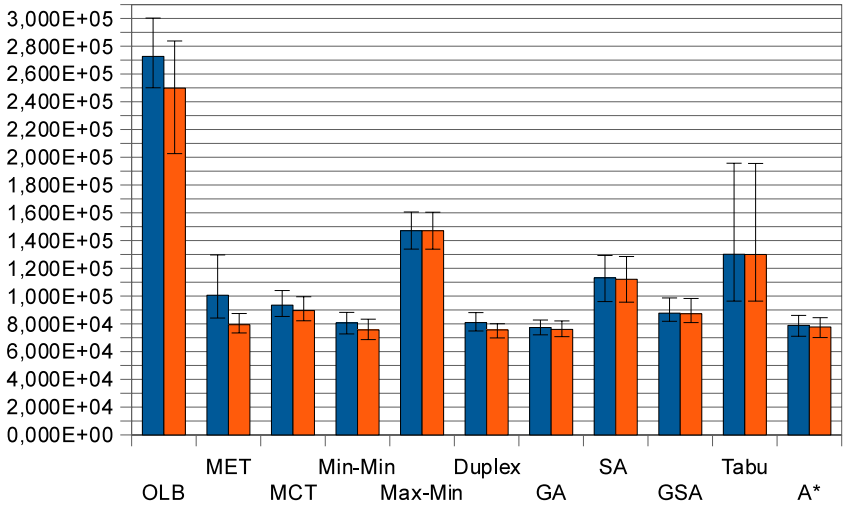

Abbildung C.8.: High Task, Low Machine, Inconsistent, 512x16

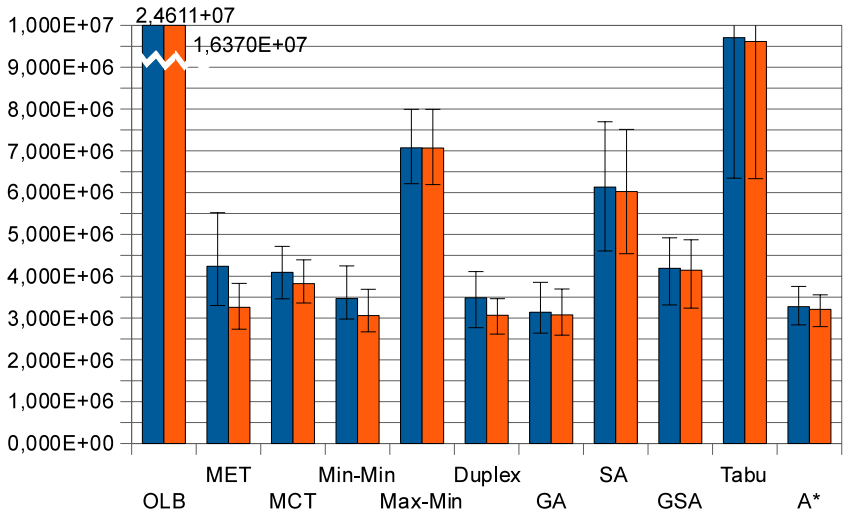

Abbildung C.9.: High Task, High Machine, Inconsistent, 512x16

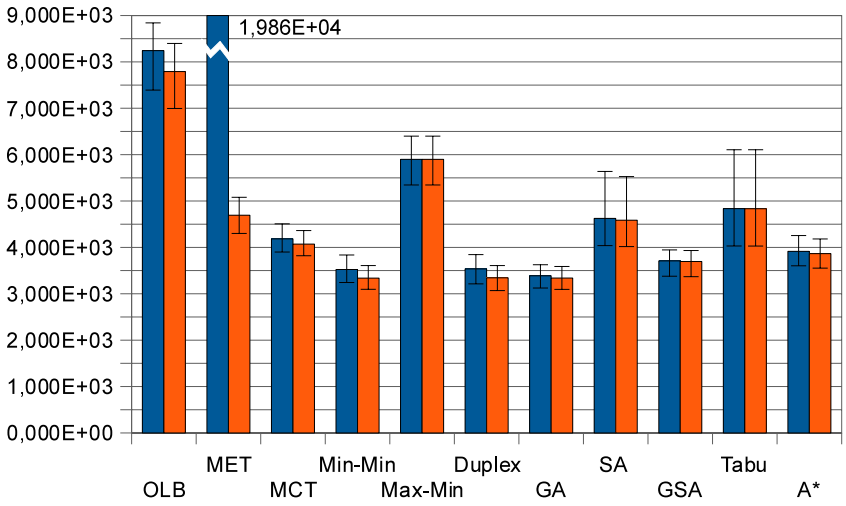

Abbildung C.10.: Low Task, Low Machine, Partially Consistent, 512x16

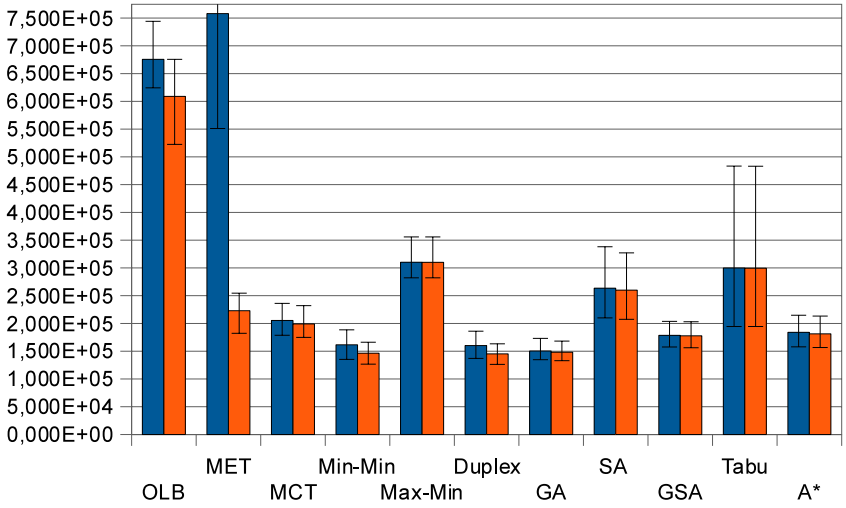

Abbildung C.11.: Low Task, High Machine, Partially Consistent, 512x16

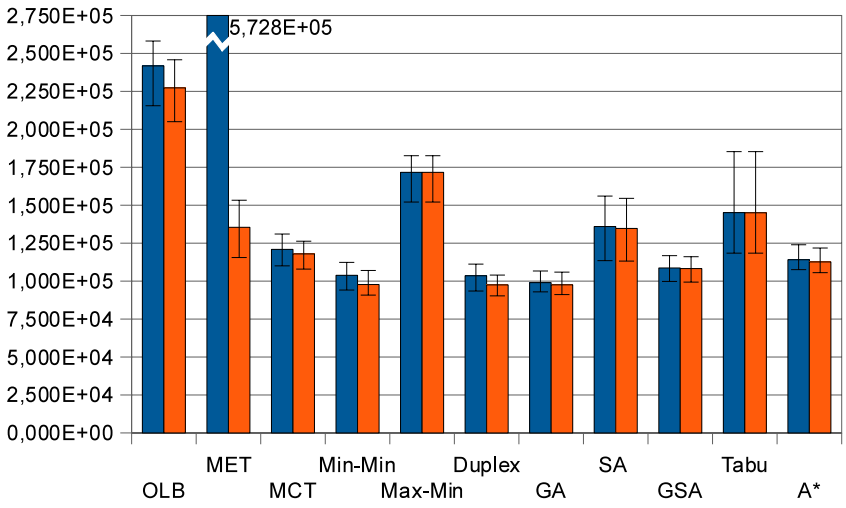

Abbildung C.12.: High Task, Low Machine, Partially Consistent, 512x16

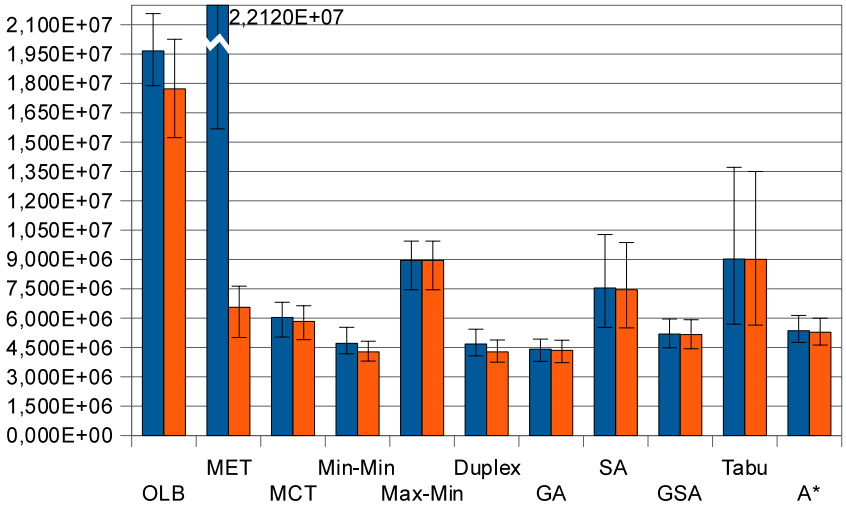

Abbildung C.13.: High Task, High Machine, Partially Consistent, 512x16

Literaturverzeichnis

[Tr01] Tracy D. Braun, Howard Jay Siegel, Noah Beck et. al., *A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems*, Journal of Parallel and Distributed Computing 61 (810-837), 2001

[Ru94] Günter Rudolph, *Convergence Analysis of Canonical Genetic Algorithms*, IEEE Trans. Neural Networks 5, 1 (96-101), 1994

[Sr94] M. Srinivas, Lalit M. Patnaik, *Genetic Algorithms: A Survey*, IEEE Comput. 27, 6 (17-26), 1994

[Co96] M. Coli, P.Palazzari, *Real time pipelined system design through simulated annealing*, J. Systems Architecture 42, 6-7 (465-475), 1996

[Ch98] Hao Chen, Nicholas S. Flann, Daniel W. Watson, *Parallel Genetic Simulated Annealing: A Massively Parallel SIMD Algorithm*, IEEE Transactions on Parallel and Distributed Systems, Vol. 9, No. 2 (126-136), 1998

[Sh96] Pankaj Shroff, Daniel W. Watson, Nicholas S. Flann, Richard F. Freund, *Genetic Simulated Annealing for Scheduling Data-Dependent Tasks in Heterogeneous Environments*, 5th IEEE Heterogeneous Computing Workshop (98-104), 1996

[Ri07] Graham Ritchie, John Levine, *A fast, effective local search for scheduling independent jobs in heterogeneous computing environments*, Journal of Parallel and Distributed Computing, Vol. 67, Issue 6, 2007

Erklärung

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, sind in jedem einzelnen Fall unter Angabe der Quelle als Entlehnung (Zitat) kenntlich gemacht. Gleiches gilt für beigefügte Skizzen und Darstellungen.

Unterschrift