Task-Platzierung auf Many-Core-Prozessoren mit fehlerhaften Komponenten

Dissertation

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

Sebastian Schlingmann
schlingmann@informatik.uni-augsburg.de

2013
Fakultät für Angewandte Informatik der Universität Augsburg

1. Gutachter: Prof. Dr. Theo Ungerer
2. Gutachter: Prof. Dr. Bernhard Bauer

Tag der mündlichen Prüfung: 06.12.2013

Abstract

This thesis describes the development and evaluation of three algorithms for the placement of parallelized applications on future many-core processors. The algorithms are able to compensate for permanent hardware faults and thereby cause a graceful degradation rather than a program crash. The fault model considers permanently failed cores, routers and processor interconnect links.

The main goal of the placement algorithms is a working placement of applications, which consist of individual tasks communicating with each other, onto a many-core processor. The placement has to retain a high communication performance for each application. Since the algorithms are executed at the beginning of each program it is required that the runtime of the placement decision does not to grow excessively.

Many-core processors and applications are modelled as graphs which depict the communication structures, as well as the intensities of the communications. Both synthetic task graphs and task graphs derived from real world applications are considered.

This thesis develops the *Connectivity-Sensitive Algorithm* that generates static placements, which try to keep the distance between the tasks small. As a further refinement the *Fitting Algorithm* and the *Favorite Neighbor Algorithm* are presented. These two extensions to the basic algorithm are supposed to be beneficial in the placement of multiple task graphs and in the placement of task graphs with a strong imbalance of communication intensities.

Also methods to extend the static placements by dynamic processes are presented, which require changes to the hardware and the general placement strategy.

In a detailed evaluation chapter all aspects of the task placement on potentially faulty many-core processors are considered. Placement in both the error-free case, as well as placement assuming faulty hardware components is investigated. The effect of any error supported by the error model is first analyzed individually, before combinations of errors are considered. The results are compared and contrasted with those of a well known algorithm from the literature and two simple placement metrics.

Different network topologies and the bandwidth constraints of the on-chip network are also discussed.

It turns out that the described *Connectivity-Sensitive Algorithm* is in most cases able to generate the placements with the lowest communication overhead, and is also able to find valid placements in the presence of high error rates. The two extended algorithms are not able to generate better results.

Zusammenfassung

Die vorliegende Dissertation beschreibt die Entwicklung und Evaluierung dreier Algorithmen zur Platzierung von parallelisierten Applikationen auf zukünftigen Many-Core-Prozessoren. Die Algorithmen sind in der Lage, auftretende permanente Fehler der Hardware zu kompensieren und dadurch einen allmählichen Funktionsausfall anstatt eines Programmabsturzes herbeizuführen. Es werden permanent ausgefallene Kerne, Router und Interconnect-Verbindungen des Many-Cores betrachtet.

Das Hauptziel der Platzierungsalgorithmen besteht darin, eine funktionsfähige, jedoch auch performante Platzierung von Applikationen, welche aus einzelnen miteinander kommunizierenden Tasks bestehen, auf einen Many-Core-Prozessor durchzuführen. Da die Algorithmen jeweils zu Beginn eines jeden Programms ausgeführt werden, ist darauf zu achten, dass die Laufzeit der Platzierungsentscheidung nicht übermäßig wächst.

Many-Core-Prozessoren und Applikationen werden als Graphen dargestellt, die sowohl die Kommunikationsstrukturen als auch die Intensitäten der Kommunikationen abbilden. Es kommen sowohl synthetische Task-Graphen als auch Umsetzungen realer Applikationen zum Einsatz.

Die vorliegende Arbeit entwirft den *Konnektivitätssensitiven Algorithmus*, der statisch Platzierungen erzeugt, welche versuchen, die Entfernungen zwischen den Tasks gering zu halten. Als Verfeinerung werden zusätzlich der *Fitting Algorithmus* und der *Favorite Neighbour Algorithmus* entwickelt. Diese beiden Erweiterungen des Basisalgorithmus sollen bei der Platzierung mehrerer Task-Graphen und bei der Platzierung von Task-Graphen mit starkem Ungleichgewicht der Verbindungen Vorteile bringen. Die Ergänzung der statischen Platzierungsmethoden um dynamische Verfahren erfordert Änderungen an der Hardware und der Platzierungsstrategie. Es können aber Vorteile bei der Platzierungsgeschwindigkeit und der Qualität der Ergebnisse erzielt werden.

Ein ausführliches Evaluierungskapitel betrachtet alle Aspekte der Task-Platzierung auf potentiell fehlerhaften Many-Core-Prozessoren. Hierbei untersucht die Arbeit sowohl der fehlerfreie Fall als auch die Platzierung unter Annahme von fehlerhaften Hardwarekomponenten. Die Auswirkungen jeder Fehlerart des verwendeten Fehlermodells werden, vor der Betrachtung von Fehlerkombinationen, zuerst einzeln analysiert. Ein Vergleich mit den Ergebnissen eines bekannten Algorithmus aus der Literatur und zwei einfachen Platzierungsmetriken ist ebenfalls Teil der Evaluierung. Unterschiedliche Netzwerk-Topologien und die Belastung des On-Chip-Netzwerks werden auch behandelt.

Es stellt sich heraus, dass der entwickelte *Konnektivitätssensitive Algorithmus* in den meisten Fällen die Platzierung mit dem geringsten Kommunikationsoverhead erzeugen kann und zusätzlich auch bei hohen Fehlerraten gültige Abbildungen findet. Die beiden erweiterten Algorithmen schneiden schlechter ab.

Danksagung

Mein besonderer Dank gilt Professor Dr. Theo Ungerer für die intensive Betreuung und sein konstruktives Feedback zu meiner Arbeit. Er hat es geschafft eine gute Arbeitsatmosphäre an seinem Lehrstuhl aufzubauen, so dass ich mich sehr wohl gefühlt habe. Ohne seine Unterstützung wäre diese Arbeit nicht zustande gekommen. Für die Möglichkeit durch Konferenzreisen und die Mitarbeit an einem EU-Projekt an der internationalen Wissenschaftscommunity teilzuhaben bin ich ihm sehr dankbar.

Für Diskussionen zum Thema und oft darüber hinaus danke ich allen meinen Kollegen am Lehrstuhl für Systemnahe Informatik und Kommunikationssysteme. Es war sehr hilfreich andere Blickwinkel auf verschiedenste Themen zu sehen und diese in eigene Überlegungen einfließen zu lassen. Auch die gemeinsamen Aktivitäten außerhalb der Uni haben immer viel Spaß gemacht. Ein spezieller Gruß geht dabei an die Fast-Food-Friday-Gruppe und die Badminton-Mannschaft. Nicht zu vergessen ist Petra, ohne die der Lehrstuhl lange nicht so gut laufen würde.

Ich bedanke mich herzlich bei Julia dafür, dass sie mir das Leben etwas leichter gemacht hat während ich diese Arbeit verfasst habe.

Auch wenn es zu dem Zeitpunkt manchmal unangenehm war bedanke ich mich bei allen meinem Freunden, dass sie immer wieder nachgefragt haben wann dies Arbeit endlich fertig geschrieben ist. Speziell zu erwähnen sind in diesem Zusammenhang die Jungs von der Akademiker-Skifahrt.

Schlussendlich gebührt mein aufrichtiger Dank auch meiner gesamten Familie, deren Rückhalt mir immer ein Ansporn gewesen ist.

Inhaltsverzeichnis

1 Einleitung

Seit einiger Zeit zeichnet sich die Abkehr einer reinen Beschleunigung der Taktfrequenz von Mikroprozessoren ab. Da Moores Gesetz (Moore u. a. 1965), laut dem sich die Anzahl der Transistoren auf integrierten Schaltungen alle zwei Jahre verdoppelt, immer noch gilt, schnellere Taktfrequenzen allerdings nur noch mit für die Anwendungsfälle unrealistischen Energieanforderungen realisiert werden können, haben die Prozessorhersteller begonnen, mehrere CPU-Kerne auf einem Chip zu vereinen. Die sogenannten Multi-Core-Prozessoren erbringen einen Performancevorteil, da moderne Betriebssysteme viele Prozesse gleichzeitig ausführen. Der parallele Ablauf unabhängiger Programme auf unterschiedlichen CPU-Kernen löst somit das Problem des Flaschenhalses einer einzelnen Berechnungseinheit.

Gängige Hardware für Endbenutzer enthält momentan bis zu acht Kerne auf einem Prozessor. Eine weitere Steigerung der Kernanzahl kündigt sich ebenfalls an. So sind im Serverbereich bereits bis zu zwölf Kerne pro Chip erhältlich. Zusätzlich forschen die Prozessorhersteller an wesentlich größeren Systemen. Intels Single-Chip Cloud Computer (SCC) enthält 48 Kerne (Intel 2010) und der Forschungschip Polaris stellt sogar 80 Kerne (Held, Bautista und Koehl 2006) in einer 8x10 Gitter-Anordnung zur Verfügung. Als Ergebnis dieser Forschung und anderen Projekten wurde von Intel der Xeon Phi Coprozessor (Intel Corporation 2013) entwickelt, welcher mit dem Markt der General-Purpose-Berechnungen auf Grafik-Hardware konkurrieren soll. Er stellt in der höchsten Ausbaustufe 61 Kerne zur Verfügung, welche wiederum jeweils vier Hardware-Threads verarbeiten können. Die Firma Tilera stellt seit 2007 ebenfalls Prozessoren mit hohen Kernzahlen her. Der TilePro64 (Tilera 2008) und der TileGx (Tilera 2013) enthalten 64 und bis zu 72 Kerne, welche durch ein Maschennetzwerk auf dem Chip verbunden sind. Für den Bereich der eingebetteten Systeme entwickelt Kalray einen Many-Core-Prozessor mit bis zu 1024 Kernen, der sich durch eine niedrige Energieaufnahme auszeichnen soll. Derzeit ist allerdings nur die Variante mit 256 Kernen, der MPPA 256 (Kalray 2013), verfügbar.

Bei Prozessoren einer solchen Größe entstehen allerdings neue Probleme. Der Flaschenhals, der früher die einzelne CPU war, ist nun stärker auf die Speicherschnittstelle und die Kommunikation zwischen den einzelnen On-Chip-Komponenten verlagert. Die sogenannte Memory-Wall war zwar schon zu Zeiten der Einprozessorsysteme ein Problem, durch die gleichzeitige Abarbeitung mehrerer speicherhungriger Programme tritt allerdings eine noch höhere Belastung auf.

Bei der Konzeption von Many-Core-Prozessoren, also Prozessoren mit mehr als 16 Kernen, müssen daher neue Ansätze erdacht werden, wie dieses Problem abgeschwächt werden kann. Hier zeichnen sich derzeit nachrichtengekoppelte Prozessoren mit verteiltem Speicher als Lösungsmöglichkeit ab. Es wird eine komplette Netzwerkstruktur auf dem Prozessor nachgebildet, was eine einfache Skalierung ermöglicht und weniger po-

tentielle Flaschenhälse aufweist. Dies öffnet der Forschung den großen Bereich der On-Chip-Netzwerke, in dem viele Techniken aus traditionellen Kommunikationsnetzen auf die Bedürfnisse eines Prozessor-Interconnects angepasst werden. Die Forschung erstreckt sich über das Design der Router bis hin zu geeigneten Routingalgorithmen und Lastbalancierungsverfahren.

Um einen solchen Prozessor effizient zu programmieren, wird es allerdings ein Umdenken in der Softwareentwicklung und auf Ebene der Compiler und Betriebssysteme geben müssen. Die Voraussetzungen für die Ausführung eines Programms haben sich drastisch verändert. Da eine Ausnutzung der Parallelität in einem Programm sehr wichtig für die erzielte Performance ist kann davon ausgegangen werden, dass Software in viele zueinander parallel ausführbare Programmteile aufgespalten wird. Diese kommunizieren miteinander, um einen gemeinsamen Programmzweck zu erfüllen.

Die Aufgabe des Betriebssystemschedulers ist es nun, solche Systeme aus kommunizierenden Teilelementen möglichst günstig auf einem Many-Core-Prozessor auszuführen. Zu diesem Zweck ist eine Zuweisung der einzelnen Programmsegmente auf Hardwarekomponenten nötig. Die Platzierung der einzelnen Programmteile spielt hierbei für die resultierende Performance des Gesamtsystems eine große Rolle. Je kleiner Kommunikationspfade in einem System sind, desto weniger Verzögerung tritt auf. Zusätzlich kann Kommunikation über mehrere Zwischenstationen im Netz die Nachrichten anderer Programmteile beeinflussen. Dies erzeugt zusätzliche Verzögerungen. Eine Platzierung, die Kommunikationspfade zwischen zusammen arbeitenden Programmsegmenten optimiert, ist daher für den kompletten Many-Core-Prozessor vorteilhaft.

Bei sinkenden Strukturbreiten, welche für die Integration von immer mehr Hardwarekomponenten auf einem Prozessor nötig sind, tritt zusätzlich ein weiteres Problem auf. Es wird angenommen, dass durch die Minimierung der Strukturgrößen eine höhere Anfälligkeit für verschiedenartige Hardwarefehler besteht (Borkar 2005). Dies kann von gelegentlich auftretenden Fehlern bis hin zu permanent beschädigten Hardwarekomponenten reichen. Im Falle eines permanenten Defekts wäre es allerdings bei einem derartig komplexen System unvernünftig und nicht wirtschaftlich von einem kompletten Defekt des gesamten Many-Core-Prozessors auszugehen. Daher ist es sinnvoll Fehlertoleranztechniken in den Prozessor zu integrieren. Diese Techniken müssen in der Lage sein, auftretende Fehler zu erkennen und zu korrigieren oder zu umgehen. Folglich muss auch die Task-Platzierung des Betriebssystems fehlerhafte Prozessorelemente in Betracht ziehen. Damit entsteht ein verlässliches Gesamtsystem, dass aus potentiell unverlässlichen Komponenten besteht und im Fehlerfall in der Lage ist, einen allmählichen Funktionsausfall herbeizuführen ohne harte Abstürze in Kauf nehmen zu müssen.

Als Voraussetzung für diese Arbeit wird angenommen, dass ein nachrichtengekoppelter Many-Core-Prozessor für General-Purpose-Aufgaben bereitsteht, welcher in der Lage ist, auftretende permanente und transiente Fehler zu erkennen und diese Informationen den Task-Platzierungsmechanismen zur Verfügung zu stellen.

Der Inhalt dieser Arbeit trägt dazu bei, das Problem der Task-Platzierung auf einem nachrichtengekoppelten General-Purpose-Many-Core-Prozessor unter Berücksich-

tigung auftretender Fehler effizient zu lösen. Zu diesem Zweck werden Platzierungsalgorithmen entwickelt und detailliert evaluiert. Diese Algorithmen legen besonderen Wert auf eine Platzierung mit geringem Kommunikationsoverhead unter gleichzeitiger Vermeidung fehlerhafter Prozessorkomponenten. Zusätzlich wird eine prohibitiv lange Laufzeit der Platzierungsalgorithmen selbst vermieden.

Die wesentlichen Beiträge der vorliegenden Arbeit zur aktuellen Forschung sind:

- Entwicklung von drei Algorithmen zur Task-Platzierung auf Many-Core-Prozessoren
- Detailierte Evaluierung der entwickelten Algorithmen mit realen sowie mit synthetischen Applikationen
- Analyse des Einflusses von fehlerhaften Elementen in einem Many-Core-Prozessor auf Task-Platzierungsalgorithmen

Der weitere Teil der Arbeit gliedert sich wie folgt:

In Kapitel 2 werden die grundlegenden Komponenten von Many-Core-Prozessoren erläutert. Außerdem wird auf gängige Parallelprogrammierungs-Paradigmen und deren Eignung im Kontext eines Many-Cores eingegangen.

Kapitel 3 beschreibt das verwendete Systemmodell und Fehlermodell. Ebenfalls enthalten ist eine Beschreibung der für die Evaluierung betrachteten Applikationen.

Verwandte Arbeiten aus dem Forschungsfeld der Task-Platzierung werden in Kapitel 4 erläutert.

Kapitel 5 beschreibt im Detail den entwickelten Algorithmus zur Lösung von statischen Platzierungsproblemen auf Many-Cores mit fehlerhaften Elementen. Zusätzlich werden zwei Erweiterungen präsentiert, welche in einigen Situationen Verbesserungen versprechen.

In Kapitel 6 ist eine Übersicht über mögliche Erweiterungen auf das Problem einer dynamischen Anpassung von Platzierungen zur Laufzeit des Systems gegeben.

Eine umfangreiche Evaluierung der vorgestellten Algorithmen ist in Kapitel 7 zu finden. Es wird auf verschiedene Szenarien, sowohl mit auftretenden Fehlern als auch im fehlerfreien Fall, eingegangen. Zusätzlich wird ein Vergleich zu existierenden Algorithmen in der Literatur durchgeführt, um die Qualität der selbst entwickelten Algorithmen einschätzen zu können.

Abschlussbetrachtungen und ein Ausblick auf mögliche Erweiterungen dieser Forschung sind im Schlusskapitel zu finden.

Der Anhang enthält zusätzliche Evaluierungsergebnisse, die in Kapitel 7 keinen Platz gefunden haben, sowie die Abbildung einiger großer Task-Graphen der verwendeten Applikationen.

2 Grundlagen

2.1 Multi- und Many-Core-Prozessoren

In dieser Arbeit wird eine spezielle Grundarchitektur von Many-Core-Prozessoren angenommen. Es existieren viele konkurrierende Konzepte zur Integration einer großen Anzahl Kerne auf einem Chip.

Da jedes dieser Konzepte unterschiedliche Annahmen trifft, ist es wichtig, diese zu kennen und darauf einzugehen. Im folgenden Abschnitt werden die gängigen Designkonzepte vorgestellt und ihr Einfluss auf das Problemfeld der Task-Platzierung analysiert.

Designkonzepte lassen sich in drei Kategorien einteilen:

- *Kernarten*, die angeben, ob es nur gleichartige Kerne auf dem Chip gibt (Homogenität) oder ob und welche verschiedenartige Kerne existieren (Heterogenität).
- *Speicherorganisation*, die beschreibt, an welchen Stellen des Chips On-Chip-Speicher oder Speichercontroller für externe Speicher bereitstehen und welche Komponenten darauf zugreifen können.
- *Verbindungsnetze*, die dafür zuständig sind, Kerne untereinander und Kerne mit Speicher und Peripherie zu verbinden.

Kernarten

Zusätzlich zu den anderen Unterscheidungskriterien kann sich ein Prozessordesign auch sehr stark verändern, wenn die Art der verwendeten Kerne variiert wird.

Bei einem homogenen Prozessor muss einzig auf einen Instruktionssatz Rücksicht genommen werden und jeder Kern ist prinzipiell gleich gut geeignet, eine Aufgabe auszuführen. Die Position eines Kerns auf dem Prozessor kann allerdings trotzdem einen Einfluss auf die Performance einer darauf ausgeführten Applikation besitzen. Dies liegt darin begründet, dass für die Performance einer Programmausführung auch die Off-Chip-Anbindung von Relevanz ist. Benötigt ein Programmteil beispielsweise viele IO-Operationen, ist es sinnvoll, ihn auf einem Kern zu platzieren, der direkten Zugriff auf die benötigte Ressource besitzt. Allerdings können andere Kerne, unter Umständen mit reduzierter Performance, die selbe Aufgabe auch übernehmen.

Ein Prozessor mit heterogenen Kernen ist dagegen komplexer aufgebaut. Es müssen Kerne mit teilweise unterschiedlichen Fähigkeiten und Instruktionssätzen sinnvoll kombiniert werden. Durch diese Zusammensetzung kann auch nicht jeder Kern alle

Aufgaben übernehmen. Einige Aufgaben werden auch besser durch spezialisierte Kerne ausgeführt. Dies ermöglicht unter anderem auch eine wesentlich effizientere Ausführung und trägt dazu zur Energieeinsparung bei. In Mobilgeräten existieren bereits funktionierende Konzepte, die einen Energiesparmodus mit einfachen, aber langsamen Prozessoren vorsehen, bei erhöhtem Performancebedarf dann aber auf schnellere Prozessoren umschalten können (Greenhalgh 2011). Auch bei Grafikchips ist dieses Konzept bereits mit Hybridgrafikkarten umgesetzt (Temkine, Drapkin und Caruk 2006). Ein Prozessor mit heterogenen Kernen ist in der Platzierungsbetrachtung wesentlich komplexer.

Aktuelle High-Performance-Systeme verlassen sich stark auf heterogene Architekturen. Es werden Grafikbeschleuniger, sogenannte Graphics Processing Units (GPU), eingesetzt, um gewisse General-Purpose Aufgaben zu übernehmen (Yang u. a. 2011).

Diese Zusatzhardware arbeitet mit einfachen, auf Streaming-Aufgaben spezialisierten Architekturen und kann so wesentlich höhere Kernzahlen bei einem akzeptablen Energieverbrauch erreichen. Allerdings wird spezieller Code für die Ausführung auf GPUs benötigt und es kann auch nicht jede Aufgabe effizient auf einem Grafikbeschleuniger berechnet werden.

Aber auch in homogenen Architekturen muss manchmal mit heterogener Performance gerechnet werden. Durch dynamische Anpassungen der Taktrate der Kerne kann die Geschwindigkeit stark variieren. Diese wird hauptsächlich zum Stromsparen oder zum Steigern der Single-Core-Performance einzelner Kerne unter Einhaltung der Thermal-Design-Power eingesetzt (Charles u. a. 2009).

Der Rest der Arbeit nimmt einen homogenen Prozessor an, der hohe Performance durch eine gute Parallelisierung der ausgeführten Applikationen erreicht.

Speicherorganisation

Da Speicherzugriffe schon immer einer der limitierenden Faktoren der Performancesteigerung von Prozessoren waren, ist es auch nötig, bei der Untersuchung von Many-Core-Prozessoren verschiedene Modelle des Speicherzugriffes zu betrachten. Das bei handelsüblichen Multi-Core-Prozessoren verwendete Shared Memory Modell besitzt zwar Vorteile in der Programmierung, doch existieren Skalierungsprobleme, da der gemeinsame Speicher eine geteilte Ressource ist. Konkurrierende Zugriffe auf den Speicher verlangsamen die komplette Programmausführung. Um in einem solchen System die Korrektheit der Ergebnisse sicherzustellen, müssen vom Programmierer Synchronisationskonstrukte verwendet werden. Leider ist in aktueller Hardware zu beobachten, dass die Geschwindigkeit eines Locks mit der Anzahl der Kerne auf einem Prozessor stark abnimmt (Kägi, Burger und Goodman 1997).

Cache Hierarchien können diesen Effekt nicht kompensieren, da auch bei privaten Caches der Kerne auf Kohärenz geachtet werden muss. Auf längere Sicht muss hier also bei steigenden Kernzahlen eine Alternative gefunden werden.

Ein weiterer Lösungsansatz, um Wartezeiten bei Synchronisation zu verhindern, ist Transactional Memory. Das dahinter stehende Prinzip ist ein optimistischer Zugriff auf den Speicher, der ohne Locks auskommt. Zeigt sich beim Abschluss einer Transaktion, dass ein Konflikt zwischen mehreren laufenden Transaktionen auftreten würde,

wird eine der Transaktionen gültig gemacht, die anderen aber zurückgerollt. Diese müssen damit neu begonnen werden. Diese Vorgehensweise kann bei Programmen mit vielen Locks aber wenig Konfliktpotential die Performance stark steigern. Aktuelle Transactional Memory Systeme, wie das im Velox Projekt entstandene Deuce (Felber u. a. 2010), sind aber meist rein in Software realisiert. Eine Hardware-Implementierung des Transactional Memory Konzepts führte Intel mit der Haswell-Architektur ein (Intel Corporation 2012 Kapitel 8). Transactional Memory ist weiterhin problematisch, wenn viele konkurrierende Zugriffe stattfinden, da sehr oft Transaktionen abgebrochen werden (Perfumo u. a. 2008).

Bei Distributed Memory Systemen gibt es keinen gemeinsamen Speicher, der von allen Kernen genutzt werden kann. Jedes Speicherelement kann nur von einer Komponente gelesen und geschrieben werden. Solange diese Komponente nicht mehrfädig läuft, ist es bei Speicherzugriffen nicht nötig, auf Konflikte und Konsistenz zu achten. Zugriffe auf nicht-lokale Speicher müssen durch Kommunikation zwischen den Komponenten abgewickelt werden. Die einzelnen Kerne sind somit nachrichtengekoppelt.

Da die meisten aktuellen Programmiersprachen keine gute Unterstützung für Nachrichtenkopplung besitzen, ist dieses Paradigma für den Programmierer schwer zu verwenden. Der Zugriff auf verteilten Speicher wird durch Distributed Shared Memory vereinfacht, welches eine Zwischenschicht zwischen einer Distributed Memory Hardware und einem Shared Memory Programmiermodell darstellt. Viele parallele Programmiersprachen verwenden die Partitioned Global Address Space (PGAS) (Carlson u. a. 2003) Abstraktion, um verteilten Speicher zu nutzen. Die verteilte Natur des Speicherzugriffs wird maskiert, wodurch der Programmierer mit einem globalen, gemeinsamen Adressraum arbeiten kann. Intern kommt weiterhin Kommunikation über Nachrichten zur Anwendung, um auf entfernte Speicherbereiche zuzugreifen. Distributed Shared Memory erhöht allerdings die Komplexität des Gesamtsystems und kann zu unvorhersagbaren Speicherzugriffszeiten führen.

Die vorliegende Arbeit geht von einer Distributed Memory Architektur aus, da jeder Kern so seine Instruktionen und Daten lokal vorhält und Synchronisationen nicht betrachtet werden sollen.

Verbindungsnetze

Die gängige Methode, aktuelle Multi-Core-Prozessoren zu verbinden, sind ein oder mehrere Busse. Oftmals haben diese die Form eines Ringes, um Signallaufzeiten zu minimieren. Aktuelle Consumer-Prozessoren verwenden das QuickPath Interconnect (Ziakas u. a. 2010) oder HyperTransport (Conway und Hughes 2007).

Busse stellen eine erprobte Technologie dar, die allerdings auf Probleme stößt, wenn zu viele Teilnehmer angeschlossen sind. Zwar ist der Verkabelungsaufwand gering und die Netzlogik der Komponenten ist auf ein minimales Maß beschränkt, aber ein Bus ist ein geteiltes Medium, so dass nicht alle Komponenten zu jedem Zeitpunkt senden können. Wollen viele Komponenten gleichzeitig den Bus benutzen, entstehen Wartezeiten solange der Bus belegt ist. Dies steigert bei hoher Kommunikationsdichte die Laufzeiten der Nachrichten massiv und verringert so die Performance des kompletten Systems. Da das Kommunikationsvolumen bei Many-Core-Prozessoren stark ansteigen

wird, ist es ratsam, für solche Prozessoren andere Verbindungsnetze zu verwenden.

Eine Möglichkeit, das Problem der Wartezeiten zu verringern, ist die Kerne in Cluster aufzuteilen, die sowohl intern als auch extern an Busse angeschlossen sind. Die Wartezeiten an den einzelnen Bussen sind dadurch geringer. Dafür muss aber an mehreren Stellen gewartet werden, wenn sich die Kommunikation über mehrere Busse erstreckt.

Ideal für die Kommunikation wäre es, wenn alle Prozessorkomponenten direkt mit allen anderen verbunden wären. So könnte jede Komponente mit jeder anderen direkt und ohne Wartezeiten kommunizieren. Allerdings ist für eine solche Konfiguration der Hardwareaufwand unverhältnismäßig groß und das Verlegen der vielen Leitungen würde zu einem wesentlich komplexeren Layout führen. Viele Verbindungen lägen außerdem die meiste Zeit brach.

Aus Gesichtspunkten der Skalierbarkeit hat sich in der Literatur der Ansatz des Network on Chip (NoC) durchgesetzt (W.J. Dally und B. Towles 2001). Dieses Konzept sieht die Kommunikationsverbindungen auf dem Chip als vollwertiges Netzwerk an, das in einer definierten Topologie aufgebaut ist. Jede Komponente des Prozessors ist ein eigenständiger Netzteilnehmer, der über einen eigenen Router an der Kommunikation teilnimmt. Router sind untereinander, durch die Topologie definiert, direkt verbunden. Die gängigsten Topologien sind der 2D-Mesh und der 2D-Torus.

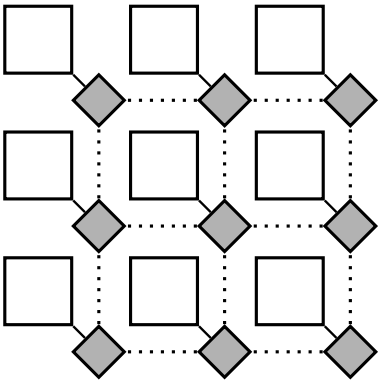


Abbildung 2.1: Eine 2D-Mesh Topologie im 3x3 NoC

Bei einem 2D-Mesh (Abbildung 2.1) ist jeder Router (grau markiert) zu den Nachbarn nach oben, unten, links und rechts verbunden. Zusätzlich benötigt ein Router eine Verbindung zur zugehörigen Prozessorkomponente.

Ein 2D-Torus (Abbildung 2.2) führt zusätzlich Verbindungen an den Rändern des Meshes ein, die vom rechten Rand des Meshes zum linken Rand und vom oberen Rand zum unteren Rand führen. Diese Verbindungen sind zwar länger und dadurch etwas langsamer, in vielen Fällen erfolgt die Kommunikation trotzdem schneller, da weniger Zwischenstationen verwendet werden müssen. Außerdem ist es möglich, die langen Verbindungen zu vermeiden, indem eine gefaltete Torus-Topologie eingesetzt wird (W. J. Dally und Seitz 1986). Hier werden, durch geschickte Anordnung der

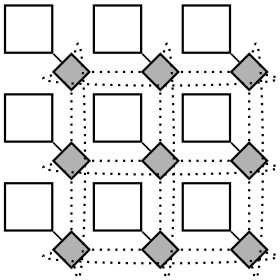


Abbildung 2.2: Eine 2D-Torus Topologie im 3x3 NoC

Router und Kerne, alle Verbindungen im Vergleich zum Mesh etwas länger, allerdings sind auch alle Verbindungen, bei gleichzeitiger Einhaltung der Torus-Eigenschaft der Randverbindungen, gleich lang (Abbildung 2.3; zur besseren Übersicht ohne Kerne).

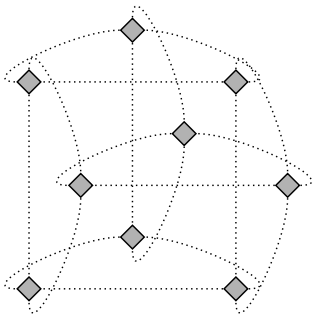


Abbildung 2.3: Eine gefaltete 2D-Torus Topologie im 3x3 NoC

Die NoC-Router sind alle gleich aufgebaut und beinhalten möglichst wenig Komplexität, um kurze Schaltzeiten zu erreichen. Sie besitzen im Allgemeinen Eingänge und Ausgänge in allen vier Himmelsrichtungen. Außerdem existiert sowohl ein Ein- als auch ein Ausgang zu der angeschlossenen Prozessorkomponente.

Die Kommunikation erfolgt paketorientiert. Jedes Datenpaket ist in einzelne Flow Control Units (Flits) aufgeteilt. Ein Flit entspricht einer routbaren Einheit im Network on Chip. Meist sind Flits nach ihrer Aufgabe gegliedert. Ein Head-Flit ist beispielsweise das erste Flit eines Pakets und beinhaltet Routinginformationen wie Quelladresse, Zieladresse und Länge der folgenden Daten. Datenflits befördern hingegen weitestgehend Nutzdaten. Der Schritt, den ein Flit von einem Router zum nächsten macht, wird

Hop genannt.

Je nach Switching-Strategie benötigen die Router zusätzlich noch Pufferspeicher an den Ein- oder Ausgängen. Diese Speicher sind möglichst klein, um Hardwarekomplexität zu vermeiden. Sie sollten aber passend zur Switching-Strategie und zum Routing-Algorithmus gewählt werden, um optimale Performance zu liefern. Ein beispielhafter NoC-Router ist in Abbildung 2.4 zu sehen, wobei die Bauart je nach NoC-Typ stark variieren kann.

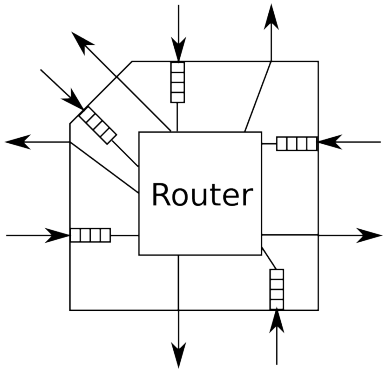


Abbildung 2.4: Ein beispielhafter Router für ein 2D-NoC

Die Auswahl eines geeigneten Routing-Algorithmus ist prinzipiell freigestellt. Die Literatur bietet hier viele Algorithmen mit unterschiedlichsten Anwendungsgebieten, Vorteilen und Nachteilen (Filch u. a. 2011, S. 135ff).

Wird eine möglichst gute Performance gewünscht, ist es wichtig, die Kommunikationswege kurz zu halten. Ein längerer Pfad durch das Netzwerk erhöht nicht nur die Latenz, sondern belegt unterwegs auch Pufferspeicher. Dies kann dazu führen, dass Verstopfungen im Netz auftreten, welche wiederum die Performance stark negativ beeinflussen. Auf einem kürzeren Kommunikationsweg sind die Daten schneller wieder aus dem Netz heraus und tragen daher nicht so stark zu Verstopfungen bei.

Am verbreitetsten ist aufgrund seines einfachen Aufbaus der XY-Algorithmus (Duato, Yalamanchili und Ni 2003, S. 146ff). Dieser gehört den Dimension-Order-Routing-Algorithmen an und sendet Pakete zuerst in X-Richtung bis die X-Koordinate mit der Zielkoordinate übereinstimmt und danach in Y-Richtung bis das Ziel erreicht ist. XY-Routing ist sehr einfach zu implementieren, hat aber gravierende Nachteile. Zum einen führt er zu ungleichmäßiger Auslastung des Netzes und damit zu Verstopfungen und langen Paketlaufzeiten. Zum anderen ist er nicht adaptiv - kann also bei Ausfall eines Routers oder einer Leitung auf dem vorgeschriebenen Weg sein Ziel nicht mehr erreichen.

Ein Beispiel für einen adaptiven Routing-Algorithmus ist der Minimal Adaptive Routing Algorithmus von William Dally (William Dally und Brian Towles 2003). Dieser Algorithmus verwendet einen Pfad mit kürzester Distanz, wobei er bei jedem Hop darüber entscheidet, welchen der möglichen nächsten Schritte er wählen soll. Dabei

wird der Netzzustand in Betracht gezogen. Umwege werden nicht gewählt.

Für die folgende Arbeit wird ein voll-adaptiver Routing-Algorithmus angenommen, da nur ein solcher in der Lage ist, defekte Netzelemente zu umgehen, ohne einen komplett neuen Kommunikationsplan aufzustellen.

2.2 Parallele Programmierung für Many-Core-Prozessoren

Bei der Betrachtung zukünftiger Many-Core-Prozessoren ist nicht nur der Aufbau der Hardware eine Problemstellung. Auch die Erstellung geeigneter Software erweist sich als schwer. Klar ist, dass nur Software, die einen hohen Parallelisierungsgrad erreicht die Vorteile eines Many-Core-Prozessors ausnutzen kann. Doch aktuelle Programmierweisen eignen sich nicht so gut für massive Parallelität. Der folgende Abschnitt gibt einen Überblick über die Eignung und die Schwächen aktueller Programmiermethoden und Parallelisierungsarten.

Parallelisierungsarten

In seinem Technical-Report „The Landscape of Parallel Computing Research: A View from Berkeley“ beschreibt Asanovic 13 parallele Programmkerne, die in aktueller Software auftreten können (Asanovic u. a. 2006). Hierzu untersuchte er, ausgehend von High-Performance-Software, auch andere Softwarearten wie Datenbanken, Computergrafik und Machine Learning. Die Programmkerne sind wie folgt benannt:

1. Dichte lineare Algebra
2. Dünnbesetzte lineare Algebra
3. Spektrale Methoden
4. N-Body Methoden
5. Strukturierte Gitter
6. Unstrukturierte Gitter
7. MapReduce
8. Kombinationslogik
9. Graphentraversierung
10. Dynamische Programmierung
11. Backtrack und Branch+Bound
12. Graphische Modelle
13. Endliche Automaten

Auf die Implementierungsdetails dieser Probleme wird nicht weiter eingegangen, da sich diese je nach verwendetem Programmierparadigma stark unterscheiden können.

Die eigentliche Aufteilung in Programmsegmente und die Festlegung der Kommunikationswege geschieht auf einer höheren Ebene.

Viele reale Probleme kommunizieren in einer ungleichmäßigen Struktur. An manchen Stellen lassen sich jedoch Muster feststellen. Ein sehr verbreitetes Muster ist Fork-Join. Es kann immer dort eingesetzt werden, wo Datenparallelität vorherrscht. Lässt sich der Wertebereich einer Aufgabe aufteilen und weitestgehend unabhängig

berechnen, so kann zu Beginn dieser Berechnung die Arbeit auf mehrere Threads aufgeteilt (Fork) und diese nach erledigter Arbeit wieder zusammengeführt werden (Join). Die Größe des Threadteams bestimmt entweder die Laufzeitumgebung, abhängig von der Problemgröße und der vorhandenen Hardware, oder sie wird vom Programmierer festgelegt.

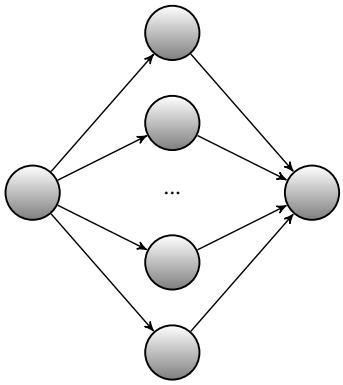


Abbildung 2.5: Ein Beispiel für einen Fork-Join-Task-Graphen

Da das Fork-Join-Muster so verbreitet ist und sich außerdem leicht parametrisieren lässt, kann es als synthetisches Benchmark herhalten. Es lassen sich Platzierungsalgorithmen mit Fork-Join-Lasten unterschiedlicher Größe unter verschiedenen Situationen ausprobieren. Dies liefert Rückschlüsse auf die Qualität eines Platzierungsalgorithmus bei diversen Größen der Threadteams.

Die in Abbildung 2.5 gewählte Repräsentation stellt eine leichte Abwandlung des Fork-Join-Modells dar. Normalerweise mündet der Join-Schritt wieder in den startenden Thread, welcher auch an der Arbeit teilnimmt. Diese Variante wurde gewählt, um interessantere und feiner gegliederte Applikationen zu erhalten. Es ist durchaus denkbar, diese Variante real zu implementieren, wenn genug Kerne zur Verfügung stehen. Die Kommunikationsbelastung eines einzelnen Kerns wird dabei ebenfalls reduziert.

Programmiermethodik

Imperative Programme lassen sich parallelisieren, indem sie in Ausführungsfäden (englisch: Threads) aufgeteilt werden. Ein Beispiel hierfür sind die POSIX-Threads, die eine plattformunabhängige Methode bereitstellen, um Threads zu starten und zu synchronisieren. Threads greifen auf den gemeinsamen Speicher zu. Der Programmierer muss dafür sorgen, dass Konsistenz und Korrektheit gewahrt bleiben.

Aufgrund dieser Komplexitäten bestehen von Hand in Threads aufgeteilte Programme meist aus wenigen großen Threads, die möglichst unabhängige Aufgaben erfüllen und möglichst wenig miteinander interagieren. Noch dazu ist der Threadwechsel auf einem Kern relativ teuer, was das schnelle Schedulen von vielen kleinen Threads behindert.

Ein derartig parallelisiertes Programm auf eine Distributed Memory Architektur abzubilden ist eine große Herausforderung und soll nicht Gegenstand dieser Arbeit sein.

Um dem Programmierer einen Teil der Parallelisierungsarbeit abzunehmen und ihn vor der Wiederholung bekannter Fehler zu schützen, sind Bibliotheken entstanden, die Threaderstellung und Synchronisation erleichtern.

Zwei bekannte Beispiele dieser Gattung sind OpenMP und die Threading Building Blocks (TBB). Beide Bibliotheken kapseln die Threaderstellung und vereinfachen die Aufteilung von Aufgabenbereichen auf dynamisch erstellte Threadpools. So kann die Arbeitslast auf die tatsächlichen Hardwaregegebenheiten angepasst und eine wesentlich feingranularere Aufteilung der Arbeitsaufgaben erreicht werden.

Sowohl OpenMP als auch TBB arbeiten, wie auch die POSIX-Threads, mit gemeinsamem Speicher. Allerdings gibt es einige Besonderheiten, die eine Aufteilung auf ein Distributed Memory Modell leichter ermöglichen. So können zu Beginn eines parallelen Bereiches Variablen als *shared* oder *private* markiert werden. Das erlaubt für Distributed Memory Systeme eine Analyse zu den Fragen, welche Speicherinhalte wohin kopiert werden können und welche über Kommunikation bereitgestellt werden müssen. Verfahren, um OpenMP Programme auf Message Passing zu übersetzen, existieren bereits (Basumallik, Min und Eigenmann 2007).

Ideal auf die Gegebenheiten einer Distributed Memory Architektur ist MPI in seinen verschiedenen Ausführungen vorbereitet. Ursprünglich wurde das Message Passing Interface (MPI) für Großrechner und Cluster entwickelt. Das Konzept des gemeinsamen Speichers gibt es dagegen nicht; gemeinsam genutzte Daten müssen über Kommunikation verteilt werden. Ein Nachteil ist allerdings das radikal andere Konzept der MPI-Programmierung. Zum einen ist es für Programmierer ungewohnt, alle benötigten Daten mit Sende-/Empfangsoperationen zu verteilen und zum anderen erleichtert MPI diese oft benötigten Operationen in keiner Weise.

Das Konzept des Message-Passing ist allerdings in andere Programmiersprachen, wie Erlang oder Go, eingeflossen und erweist sich dort als gut benutzbares und intuitives Paradigma.

Die Programmierung von Grafikbeschleunigern für allgemeine Berechnungsaufgaben wird mit Cuda oder OpenCL vorgenommen. Beide Bibliotheken ähneln sich in der Funktionsweise, wobei Cuda nur auf NVIDIA-Grafikbeschleunigern funktioniert, aber komfortabler zu programmieren ist, und OpenCL auf jeder Grafikhardware läuft, dafür aber wesentlich mehr Verwaltungscode benötigt.

In beiden Fällen müssen Quellcode und Daten in den Speicher der Grafikkarte übertragen werden, worauf diese dann lokal arbeitet. Damit verwenden Grafikbeschleuniger eigentlich schon eine Distributed Memory Architektur.

Im Rest der Arbeit wird davon ausgegangen, dass ein Programmierparadigma genutzt wird, welches in der Lage ist, eigenständige Tasks und deren Kommunikationsbeziehungen zu erzeugen.

Tasks beinhalten dabei sowohl Programmcode als auch alle benötigten Daten, welche nicht durch Kommunikation gewonnen werden. Eine Kommunikationsbeziehung beinhaltet genau zwei Kommunikationspartner, eine Kommunikationsrichtung und eine benötigte Bandbreite.

Die Gewinnung derartiger Task-Graphen (siehe Kapitel 3.3) ist auf vielfältige Weise möglich. Wird ein existierendes Programm zugrunde gelegt, dass mit Message Passing parallelisiert, ist lassen sich aus dem resultierenden Programmcode viele der benötigten Informationen bereits durch eine statische Codeanalyse extrahieren. In einigen Fällen ist Kommunikationen allerdings von Eingabewerten abhängig. Hier bietet es sich an, Informationen aus der Programmausführung zu gewinnen. Die MPI-Implementierung OpenMPI bietet zu diesem Zweck eine Tracing-Schnittstelle, durch welche alle relevanten Informationen mitprotokolliert werden. Somit lassen sich nach einer Programmausführung Kommunikationsreihenfolgen, Datenmengen und Kommunikationspartner bestimmen.

Derartig gewonnene Informationen gelten allerdings nur für die untersuchte Ausführung. Es ist durchaus möglich, dass bei einem nachfolgenden Programmlauf oder bereits bei einer anders bestückten Hardwareplattform stellenweise andere Ergebnisse erzielt werden. Aus diesem Grund ist die beste Repräsentation einer Applikation derzeit ein handgefertigter Task-Graph. Die Erstellung eines solchen Graphen bedeutet einen großen Zusatzaufwand für den Entwickler, da er nicht nur die Kommunikationspartner bestimmen muss, sondern auch die Datenmengen. Bei MPI-Programmen kann allerdings das Ergebnis einer statischen Analyse als Grundlage herangezogen werden, da viele der gewonnenen Informationen bereits zutreffend sind. Daten, die durch MPI-Tracing erzeugt wurden, sind in den meisten Fällen ungeeignet, da die dem Tracing-Lauf zugrunde liegende Hardwareplattform sehr spezifisch ist und die Ergebnisse somit nicht auf alle Zielplattformen anwendbar sind.

Die eigentliche Abbildung der Task-Graphen auf die Hardware findet zu Beginn der Ausführung des Programms durch speziell dafür entwickelte Algorithmen statt. Somit kann eine ideale Anpassung auf die Gegebenheiten des ausführenden Systems erfolgen.

3 Systemmodell und Fehlermodell

Die Algorithmen dieser Arbeit lösen das Task-Platzierungsproblem auf einem homogenen Many-Core-Prozessor mit einem Distributed Memory Modell, dessen Kerne mit einem Network-on-Chip verbunden sind. Um algorithmisch auf dieser Hardware arbeiten zu können, wurden verschiedene Annahmen getroffen und Abstraktionen eingeführt. Sowohl die Kerne und ihre Verbindungen als auch die Tasks einer Applikation werden als Graphen repräsentiert und im Folgenden Kern-Graph und Task-Graph genannt.

3.1 Fehlermodell und Fehlererkennung

Da bei einem Many-Core-Prozessor mit einer geringen Strukturbreite davon auszugehen ist, dass Fehler auftreten werden, sind diese auch bei der Task-Platzierung zu berücksichtigen. Die Task-Platzierung kann hierbei nur auf permanente und bekannte Fehler eingehen. Der Ort ihres Auftretens dient zur Kategorisierung der Fehler.

3.1.1 Fehlermodell

Fehler können in dem verwendeten Modell an verschiedenen Stellen auftreten.

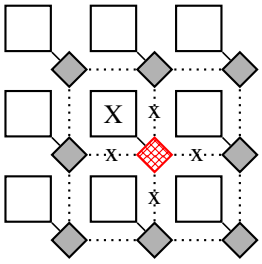


Abbildung 3.1: 2D-Mesh mit defektem Router

So ist es möglich, dass ein Kern komplett ausfällt. Das beinhaltet auch seinen Network-on-Chip Router, was somit auch alle Verbindungen zu diesem Kern nutzlos macht. Derselbe Effekt tritt auf, wenn nur der Router eines Kerns ausfällt (Abb. 3.1). Auch wenn der Kern in diesem Fall theoretisch noch funktionierte, kann er nicht mehr mit anderen Kernen kommunizieren und somit keinen sinnvollen Beitrag zum Gesamtsystem leisten. Der betroffene Kern ist in beiden Fällen kein gültiges Ziel für die Platzierung einer Task. Auch das Routing muss die betroffene Stelle umgehen, was Kommunikationswege eventuell verlängert.

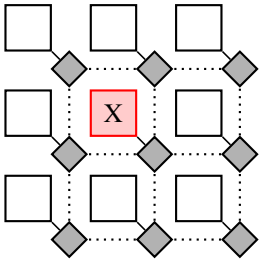


Abbildung 3.2: 2D-Mesh mit defektem Kern

Fällt allerdings nur ein Kern aus (Abb. 3.2), die Routingkomponente bleibt aber intakt, so können weiterhin Pakete durch die betroffene Stelle hindurch geleitet werden. Der Kern kann jedoch keine Tasks mehr übernehmen.

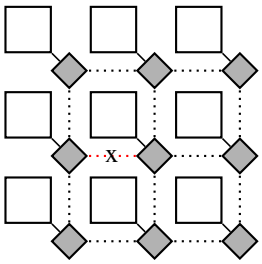


Abbildung 3.3: 2D-Mesh mit defekter Verbindung

Wenn ein Fehler nur in einer einzelnen Verbindung auftritt (Abb. 3.3), so muss dies in der Routingentscheidung zwar berücksichtigt werden, die angrenzenden Kerne sind aber weiterhin funktionsfähig und können somit Tasks ausführen.

Prinzipiell kann die Art der Fehler noch in die Klassen *permanent*, *transient* und *intermittierend* aufgeteilt werden. Als permanent fehlerhaft werden Komponenten bezeichnet, deren Defekte keine Aussicht auf Besserung haben. Die Ursachen der Fehler sind nicht reversibel.

Transiente Fehler treten im Allgemeinen durch Außeneinwirkung ein. Trifft beispielsweise ein Alpha-Teilchen unglücklich auf einen Prozessor, so ist es möglich, dass sich der Zustand der Komponente ändert, indem ein oder mehrere Bits umschalten. Aufgrund der geringen Wahrscheinlichkeit und der Zufälligkeit des Auftretens werden transiente Fehler in der Platzierungsentscheidung nicht als permanent vorhanden betrachtet.

Ein intermittierender Fehler tritt unter gewissen Umständen reproduzierbar auf. Im Allgemeinen sind diese Umstände von mehreren Faktoren abhängig und daher selten herbeizuführen. In der vorliegenden Arbeit werden intermittierende Fehler allerdings immer als permanent betrachtet. Dies liegt darin begründet, dass auch ein kleiner oder selten auftretender Fehler in einer Komponente in der Lage ist die Stabilität des Gesamtsystems zu gefährden. Es kann nicht mehr in jedem Fall angenommen werden,

dass ein betroffener Router, Kern oder Link korrekt funktioniert.

3.1.2 Fehlererkennung

Die Erkennung eines aufgetretenen Fehlers erfolgt durch einen Selbsttest beim Start des Systems. Hierzu werden alle Prozessoren, Router und Kommunikationswege einer systematischen Überprüfung unterzogen. Eine Kategorisierung jedes dabei erkannten Fehlverhaltens als permanenten Fehler führt zur Vermeidung der betroffenen Komponente.

Fehler, die während der Laufzeit neu auftreten, ob sie nun einmalig, sporadisch oder permanent sind, werden, sofern sie nicht maskierbar sind, durch bewährte Fehlererkennungstechniken lokalisiert.

Eine klassische Fehlererkennungsmethode ist die Mehrfachausführung (Lyons und Vanderkulk 1962). Die einzelnen Programmteile werden gleichzeitig oder zeitlich versetzt auf mehreren Kernen ausgeführt. Der anschließende Vergleich der Ergebnisse führt zur Feststellung von Unterschieden. Abweichende Resultate bedingen eine Wiederholung der Ausführung oder, im Falle einer Parallelausführung mit mehr als zwei Teilnehmern, die Wahl eines korrekten Ergebnisses. Eine Verringerung des Hardwareoverheads wird durch das DIVA-System (Austin 1999) erzielt. Es führt einfache Kontrollkerne ein, die jede Operation der Prozessorpipeline auf Korrektheit überprüfen. Fehlerhafte Zustände lösen Ausnahmen aus. Defekte in der Netzwerkinfrastruktur können durch Mehrfachausführung nur sehr schwer erkannt werden. Zusätzlich stellt sich das Problem, wie zu verfahren ist, wenn Nachrichten durch Netzwerkdefekte nicht ihr Ziel erreichen. Außerdem wird eine große Menge an zusätzlicher Hardware benötigt, was die Leistungsfähigkeit des Prozessors stark einschränkt (Hentschke u. a. 2002). Mögliche Fehler im überprüfenden Kern können das System der Mehrfachausführung unzuverlässig machen (Wakerly 1976).

Das EU-Projekt Teraflux entwickelt eine Fault Detection Unit (FDU) für Many-Core-Prozessoren (Garbade, Weis, Sebastian Schlingmann u. a. 2013), (Garbade, Weis, Fechner u. a. 2013). In einem Network on Chip werden eine oder mehrere solcher Einheiten platziert, welche anschließend durch Heartbeat-Nachrichten kontinuierlich den Zustand des Prozessors überprüfen. FDUs kontrollieren außerdem gegenseitig ihre Funktionsweise. Durch geschickten Nachrichtenversand können durch die Heartbeats, welche als kleine Nachrichten mit Kontrollinformationen realisiert sind, defekte Kerne, defekte Netzwerkinfrastruktur und Überlastsituationen erkannt werden.

Mit etwas mehr Komplexität in jeder Hardwarekomponente lassen sich Fehlererkennungstechniken auch direkt in die einzelnen Teile eines Prozessors integrieren (McCluskey 1990). Eine solche Absicherung der Hardware erlaubt eine konkrete Lokalisierung eines Fehlers, erfordert jedoch zusätzliche Mechanismen, um diese Informationen der Task-Platzierungseinheit zukommen zu lassen.

Zusammengenommen ist eine Fehlererkennungstechnik zu wählen, welche die drei benötigten Fehlerarten feststellen und der Task-Platzierungseinheit mitteilen kann. Welche konkret verwendet wird, spielt für die Task-Platzierung keine Rolle und ist eher von den übrigen Anforderungen des Systems abhängig.

Betreffen erkannte Fehler einen Prozessorteil, der gerade in Benutzung ist, führen sie zu einem Programmabsturz. Sind die betroffenen Komponenten momentan nicht in Benutzung, fließt der Defekt nur in die nächste Platzierungsbetrachtung mit ein.

3.2 Kern-Graphen

Die Kerne und ihre Verbindungen sind als ein ungerichteter Graph zu verstehen, bei dem die Kanten Verbindungen zwischen Kernen und die Knoten die Kerne selbst repräsentieren. Kantengewichte existieren in diesem Modell nicht, womit für alle Verbindungen ein identischer Durchsatz angenommen wird. Router sind im verwendeten Kern-Graphen nicht getrennt vom Kern modelliert.

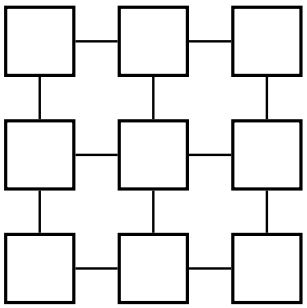


Abbildung 3.4: Ein Kern-Graph der eine 3x3 Mesh-Struktur darstellt

Das verwendete Modell geht von einfachen Kernen aus. Es wurde angenommen, dass Multitasking nicht möglich ist, was bedeutet, dass zu jedem Zeitpunkt immer nur eine Task auf einem Kern laufen darf. Erst nachdem diese Task beendet ist, kann eine weitere Task auf dem selben Kern gestartet werden. Jeder Kern besitzt einen privaten Speicher für Code und Daten, auf den nur er selbst zugreifen kann. Alle nötigen Informationen für die Ausführung einer Task auf einem Kern müssen entweder im lokalen Speicher des Kerns liegen oder bei Bedarf durch Kommunikation zu anderen Kernen beziehungsweise zum Off-Chip-Speicher beschafft werden. Zusätzlich ist zu beachten, dass alle Kerne am Rand des Graphen nach außen zu Off-Chip-Speicher und IO-Controller verbunden sind. Kerne, die sich nicht am Rand befinden, können nur über den Umweg eines Randkerns nach außen kommunizieren. Diese Eigenschaft erzeugt eine sinnvolle und intuitive Abstraktion von Off-Chip Speicherhierarchien.

Da Defekte in allen Komponenten des Prozessors auftreten können, ist es möglich, dass der Kern-Graph Lücken enthält. Daher interessiert zusätzlich zu den Dimensionen des Graphen auch die Funktionsfähigkeit aller angeschlossenen Komponenten. Die Erzeugung des Graphen kann bereits beim Systemstart erfolgen, muss daher also nicht bei jedem Platzierungslauf durchgeführt werden.

Synonym zu dem Begriff Kern-Graph wird in dieser Arbeit auch der englische Ausdruck Core-Graph verwendet.

3.3 Task-Graphen

Die auszuführenden Applikationen wurden in untereinander verbundene Tasks aufgeteilt, die auf dem Kern-Graphen zu platzieren sind. Im Gegensatz zu den Kernen sind die Tasks als gerichteter Graph modelliert, da eine Flussrichtung der Daten durch den Programmablauf vorgegeben ist. Die Tasks selbst werden als Knoten abgebildet und die Kommunikation zwischen einzelnen Tasks ist durch die Kanten repräsentiert. Die Richtung der Kanten entspricht dem Fluss der Daten im Programm. Kanten können gewichtet sein, was den Umfang der Kommunikation abbildet. Dies ermöglicht es, Datentransfers unterschiedlicher Größe im Modell genauer zu unterscheiden.

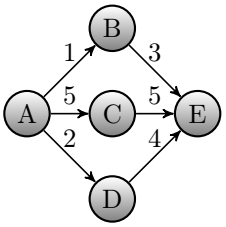


Abbildung 3.5: Ein Task-Graph einer Applikation, die nach dem Fork-Join-Modell parallelisiert ist

In dieser Arbeit werden sowohl synthetische Task-Graphen als auch Task-Graphen, die aus realen Applikationen gewonnen wurden, untersucht.

Im Bereich der synthetischen Task-Graphen ist vor allem das Fork-Join-Modell sehr verbreitet (siehe auch Abbildung 3.5). Fork-Join-Programme laufen bis zu einem bestimmten Punkt sequentiell. Dort wird der Programmablauf in mehrere parallele Teile aufgespalten (Fork). Haben diese Teile ihre Berechnungen abgeschlossen, vereinigen (Join) sie sich wieder, um sequentiell weiterzulaufen. Ein prominentes Beispiel für Fork-Join-Programme sind mit OpenMP parallelisierte for-Schleifen. Der große Vorteil synthetischer Fork-Join Task-Graphen ist die beliebige Skalierbarkeit. Wird eine größere Anzahl an Tasks benötigt, kann die Fork-Operation einfach in mehr Tasks aufspalten. Somit ist es möglich, flexible Anpassungen auf die verfügbaren Kerne eines Kern-Graphen vorzunehmen. Die Gewichtungen der Kommunikationswege einer Fork-Join-Applikation sind, anders als in der Abbildung 3.5, in der Praxis meist gleichmäßig.

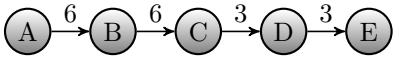


Abbildung 3.6: Ein Task-Graph einer Applikation, die nach dem Pipeline-Modell parallelisiert ist

Ein weiterer, aber auch weit weniger interessanter synthetischer Task-Graph ist die Pipeline (siehe Abbildung 3.6). Hier sind einfach mehrere Tasks in Reihe geschaltet. Jede Task übernimmt dabei eine Teilaufgabe einer aufwändigen Berechnung. Gewich-

tungen der Kommunikationswege sind in der Praxis meist gleichmäßig. Durch Kompression oder Expansion der Daten kann allerdings auch eine Variation auftreten. Für die Task-Platzierung ist die alleinige Platzierung von Pipeline-Task-Graphen wenig interessant da sie nicht verzweigen und jeder Knoten nur ein bis zwei Nachbarn besitzt. Aus diesem Grunde wurden synthetische Pipeline-Task-Graphen in der Evaluierung der Algorithmen nicht weiter betrachtet.

3.4 Verwendete Benchmark-Task-Graphen

Zusätzlich zu den synthetischem Task-Graphen ist es interessant, auch solche zu betrachten, die aus realen Programmen gewonnen wurden. Sie stellen die Abbildung real existerender Parallelprogrammierungsprobleme dar und bilden somit einen besseren Maßstab der erreichbaren Qualität einer Platzierung. Für die vorliegende Arbeit wurden aus einer Vielzahl existierender Task-Graphen einige charakteristische für die Evaluierungen in Kapitel 7 ausgewählt. Die Aufgabe der Erstellung der Task-Graphen liegt weitestgehend beim Programmierer, wobei denkbar ist dies, mit geeigneter Software-Unterstützung, zumindest teilweise zu automatisieren.

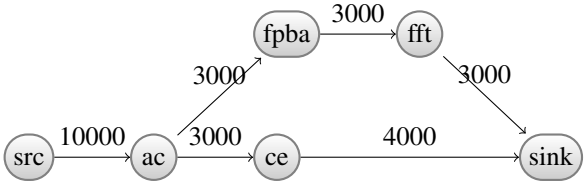


Abbildung 3.7: Telekommunikationsanwendung

Die Telekommunikationsanwendung (Abb. 3.7) beschreibt einen Teil des Innenlebens eines DSL-Modems. Sie wurde aus der Embedded System Synthesis Benchmark Suite (E3S) (Dick 2013) gewonnen, die wiederum auf den EEMBC Benchmarks (Embedded Microprocessor Benchmark Consortium 2013) aufbaut. Sie besteht aus sechs Tasks. *Src* und *Sink* sind der Einstiegs- und der Ausstiegspunkt des Algorithmus und damit für die Einleitung und Entnahme der anfallenden Daten zuständig. Die Task *ac* führt eine Autokorrelation durch, was die Menge der eingehenden Daten reduziert. Anschließend wird das gewonnene Signal zum einen durch eine Faltung kodiert (*ce*) und zum anderen durch eine Bit-Allokation (*fpba*) auf eine Fast-Fourier-Transformation (*fft*) vorbereitet. Nach der FFT werden beide Datenströme in der Task *sink* zusammengeführt. Die Bit-Allokation dient dazu, ein DSL-Signal in mehrere Kanäle aufzuteilen. Diese werden dann mit der Fast-Fourier-Transformation analysiert.

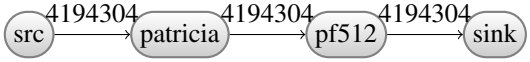


Abbildung 3.8: Netzwerkrouter

Die Netzwerk-Anwendung (Abb. 3.8) stellt ein vereinfachtes Modell eines Netzwerkrouters dar. Sie ist als Pipeline mit vier Tasks modelliert und ebenfalls aus E3S (Dick 2013) entnommen. Die Aufgaben von *src* und *sink* sind bereits bekannt. Die *patricia*-Task führt eine Routingentscheidung basierend auf einer Routingtabelle mit einem Patricia-Trie aus. Anschließend verändert die *pf512*-Task den Paketheader, um widerzuspiegeln, dass ein weiterer Hop im Netzwerk genommen wurde. Da die Datenmenge gleich bleibt, sind alle Verbindungen identisch gewichtet.

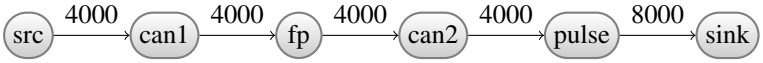


Abbildung 3.9: Automobil: CAN-Bus Anwendung

Ebenfalls als Pipeline ist die erste betrachtete Automobil-Anwendung (Abb. 3.9) realisiert. Sie beinhaltet sechs Tasks, die mehrfach über einen CAN-Bus kommunizieren, Berechnungen durchführen und einen Aktuator benutzen. Entnommen wurde dieser Task-Graph aus der E3S(Dick 2013). Die Tasks *can1* und *can2* führen die Kommunikation über den CAN-Bus durch. Die *fp*-Task führt eine komplexe Berechnung aus, deren Ergebnis nach dem Transfer über CAN von der Aktuatortask *pulse* entgegengenommen und ausgeführt werden. Das Datenvolumen bleibt fast über die komplette Kommunikation identisch. Nur der Aktuator sendet mit doppeltem Volumen.

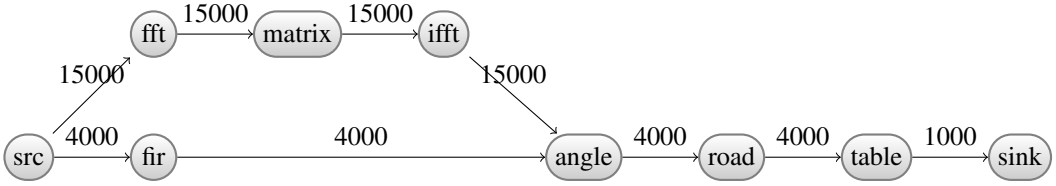


Abbildung 3.10: Automobil: Berechnung

In Abbildung 3.10 ist eine weitere Anwendung aus dem Automobilbereich zu sehen, die ebenfalls ein Teil der E3S(Dick 2013) ist. Sie besteht aus neun Tasks. Diesmal kommuniziert die Quelle *src* direkt mit zwei parallel arbeitenden Tasks und spaltet somit die Ausführung in zwei Pfade. Der obere Pfad besteht aus einer FFT, einer Task mit Matrixoperationen auf den resultierenden Daten und einer IFFT, um die modifizierten Werte zurück zu transformieren. Der andere Ausführungspfad besteht aus der Task *fir*, welche einen Filter mit endlicher Impulsantwort (finite impulse response) darstellt. Beide Ausführungspfade münden in der *angle*-Task, die über den Winkel der Kurbelwelle die aktuelle Geschwindigkeit feststellt. Anschließend berechnet die *road*-Task mit diesen Werten die aktuelle Straßengeschwindigkeit, woraufhin die *table*-Task einige der erstellen Datenpunkte in einer Tabelle ablegt, die von verschiedenen Systemen des Fahrzeugs verwendet werden können. Der untere Zweig des Task-Graphen sendet bis auf den letzten Schritt immer mit konstanter Bandbreite, wohingegen der FFT Pfad eine höhere Datenrate benötigt. Die *angle*-Task verwendet diese Daten daraufhin.

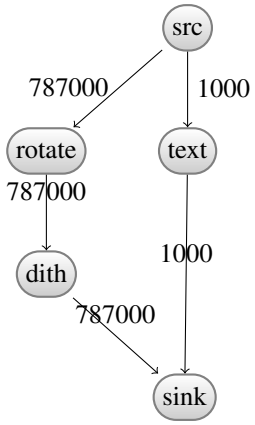


Abbildung 3.11: Büroautomatisierung

Die Büroautomatisierungsanwendung in Abbildung 3.11 ist ein Teil der E3S(Dick 2013). Sie bildet eine Komponente einer Druckauftragsverarbeitung ab. Die benutzten fünf Tasks sind in zwei Ausführungspfade, zum einen die Text-Verarbeitung und zum anderen die Bild-Verarbeitung, aufgeteilt. Wie zu erwarten, sind die Datenraten des Bild-Pfades wesentlich höher als die des Text-Pfades. Desweiteren ist der Bild-Pfad in die Tasks *rotate* und *dith* aufgeteilt, welche respektive für Bildrotation um 90° und das Dithering der Bilder zuständig sind.

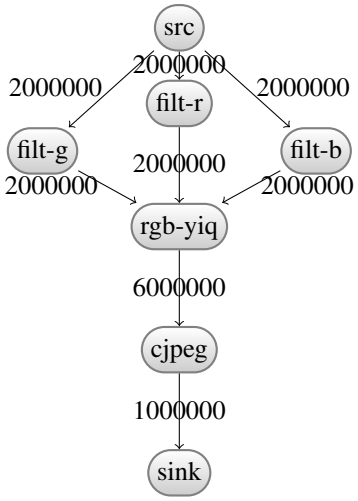


Abbildung 3.12: Bildfilter

Die Bildfilterapplikation aus Abbildung 3.12 bildet schematisch die Bildaufnahme-

einheit einer Videokamera dar. Sie ist aus den E3S Benchmarks(Dick 2013) entnommen und besteht aus sieben Tasks. Direkt nach der Quelltask werden die drei Farbkomponenten des RGB-Farbraumes parallel gefiltert. Jede der drei verwendeten Tasks erhält einen umfangreichen Datenstrom und gibt diesen gefiltert weiter an die *rgb-yiq*-Task. Diese führt eine Umwandlung des Farbraumes von RGB auf YIQ durch, was zur Vorbereitung auf die anschließende JPEG-Kompression in der Task *cjpeg* dient. Es ist zu sehen, dass der Datenstrom nach der Kompression wesentlich geringer ist, wohingegen die Bandbreite zwischen *rgb-yiq* und *cjpeg* sehr hoch ist.

Aus Gründen der Übersichtlichkeit ist der umfangreiche Task-Graph des MPEG4-Decoders im Anhang in Abbildung 9.36 dargestellt. Er ist eine Anpassung einer Implementierung, die in (Tol und Jaspers 2001) entwickelt wurde. Die expliziten Quell- und Zieltasks wurden weggelassen, so dass 15 Tasks im Graphen übrig bleiben. Die Applikation arbeitet auf einem SDRAM aus dem mit zwei Tasks gelesen wird. Zwei weitere Tasks schreiben den dekodierten Datenstrom wieder in das SDRAM zurück. Zusätzlich steht SRAM zur Verfügung, welches für die Rasterung verwendet wird. Der Task-Graph beinhaltet zwei allgemeine Rechentasks, welche *risc*-Task und *med CPU*-Task genannt werden, sowie eine Audio-Signalverarbeitung in der *adsp*-Task. Zur Bilddekompression dient die *idct*-Task, welche in der Lage ist, eine inverse diskrete Cosinus Transformation durchzuführen. Die *bab*-Task verarbeitet die Binary Alpha Blocks des MPEG-Codecs. Die Tasks *au* und *vu* stellen respektive die Audioausgabe sowie die Videoausgabe dar. Die Kommunikationsbandbreite ist im allgemeinen recht hoch, allerdings verwendet der Audio-Datenpfad (die Kommunikation über *au* und *adsp*) ein geringeres Datenvolumen. Dies macht die Platzierung des Task-Graphen interessant, da es den Platzierungsalgorithmus durch die Segmentierung der Kantengewichte vor Herausforderungen stellt. Von den bisher betrachteten Task-Graphen weist der MPEG4-Decoder den höchsten Grad der Parallelität auf.

Auch aus dem Bereich des MPEG4-Videocodecs kommt der Video Object Plane (VOP) Decoder Task-Graph. Dieser ist in Abbildung 3.13 zu finden. Die Applikation wird, wie schon der MPEG4-Decoder, in dem Paper von Tol und Jasters (Tol und Jaspers 2001) entworfen. Außerdem verwendet die Veröffentlichung des NMAP-Algorithmus (Murali und De Micheli 2004) den VOP-Decoder zur Evaluierung. Ein MPEG4-Decoder arbeitet intern mit mehreren überlagerten Ebenen, die einzelne bewegliche Segmente eines Videos repräsentieren. Jede dieser Ebenen ist eine Video Object Plane. Daher ist die Performance des VOP-Decoders ein integraler Bestandteil des MPEG4-Decodierungsprozesses. Die Applikation besteht aus 19 Tasks. Explizite Quell- und Zieltasks wurden nicht abgebildet.

Der Decoder besteht aus zwei Task-Gruppen. Die erste, welche im Task-Graphen die obere Hälfte einnimmt, besitzt einen Content Based Arithmetic Decoder in der *CAD*-Task, der auf den Eingabedaten arbeitet. Aus den resultierenden Informationen wird der Binary Alpha Block (BAB) errechnet und in einen lokalen Speicher geschrieben. Zusätzlich wird ein neuer Kontext aus dem neuen BAB und den alten BABs berechnet. Die Task *ref BAB mem* enthält die aktuellen Informationen über Binary Alpha Blöcke und liefert diese an andere Tasks, die damit arbeiten müssen. Die zweite Task-Gruppe arbeitet ebenfalls auf dem Eingabedatenstrom. Einer Pipeline-Struktur führt zuerst eine Lauflängenkodierung durch und tastet die Daten invers ab. Anschließend findet

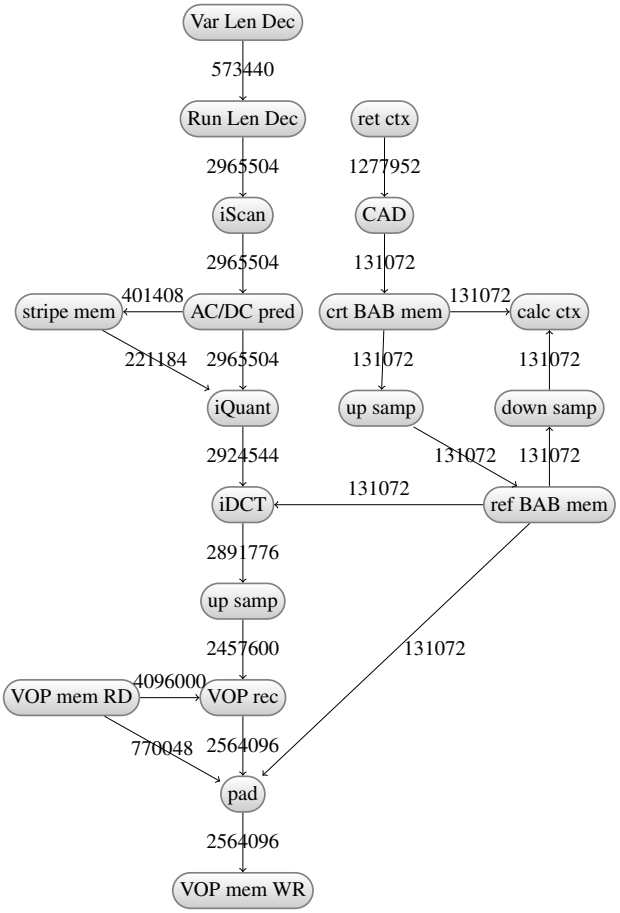


Abbildung 3.13: Video Object Plane Decoder von MPEG-4

in der Task *AC/DC pred* eine Vorhersage der AC/DC Koeffizienten statt, welche in einen lokalen Speicher geschrieben und anschließend invers quantifiziert werden. In *iDCT* durchlaufen die Daten zusätzlich eine inverse Diskrete Kosinus Transformation. Abschließend werden mit den resultierenden Daten der beiden Task-Gruppen die Video Object Planes rekonstruiert, wobei darauf geachtet wird, Bewegungen der Objekte im Verhältnis zu alten VOP-Daten aus der Task *VOP mem RD* auszugleichen. Das Ergebnis wird in der Task *VOP mem WR* in den Ausgabendatenstrom geschrieben.

Datenströme sind in dem vorliegenden Task-Graphen innerhalb der beiden Gruppen recht einheitlich. Die BAB-Berechnung verwendet verhältnismäßig wenig Daten, wohingegen die andere Gruppe den Großteil der Datenströme verarbeitet. Innerhalb beider Gruppen sind die Datenvolumina größtenteils gleichbleibend.

Der letzte betrachtete Task-Graph ist eine umfangreiche Multimedia-Applikation,

welche im Anhang in Abbildung 9.37 zu sehen ist. Sie besteht aus 31 Tasks und enthält jeweils einen Decoder und einen Encoder für das Audioformat MP3 und das Videoformat H.263. Der Graph ist eine Abwandlung einer Applikation, die zur Evaluierung eines Branch-and-Bound Mapping-Algorithmus (Jingcao und Marculescu 2005) verwendet wurde. Die einzige Änderung ist die Vereinigung gemeinsam genutzter Einheiten auf eine Task. Damit sind die einzelnen Tasks stärker miteinander verwoben, was die scharfe Trennung der einzelnen Decoder und Encoder erschwert. Der MP3-Codec verwendet die *MDCT*- und *IMDCT*-Tasks, um eine modifizierte Diskrete Kosinus Transformation durchzuführen, wohingegen die *DCT*- und *IDCT*-Tasks Teil der Videokodierung sind. Die verschiedenen FS-Tasks lesen oder schreiben Daten. Ein Zwischenpuffer kümmert sich in der *Buffer*-Task und den *Sync*-Tasks darum, dass Audio- und Videosignal nicht aus dem Takt geraten. Die Applikation verarbeitet große Datenströme, die teilweise eine Summe aus Audio- und Videosignalen darstellen. Es ist festzustellen, dass außerdem viele unterschiedliche Kommunikationsbandbreiten genutzt werden, was eine interessante Herausforderung für Platzierungsentscheidungen darstellt.

Die Aufteilung eines Programms in Task-Graphen geschieht nicht zur Laufzeit, sondern zur Compilezeit. Die Erstellung des Graphen muss, mangels geeigneter Tool-Unterstützung, derzeit von Hand vorgenommen werden. Die benötigten Informationen stehen zu diesem Zeitpunkt schon komplett zur Verfügung. Jede Task benötigt einen Code- und einen Datenteil sowie die angeschlossenen Kommunikationspartner. Für eine ideale Platzierung ist außerdem noch ein Kantengewicht der Kommunikationspfade nötig, das entweder als Empfehlung vom Programmierer erzeugt wird oder durch Analyse des Quellcodes berechnet wurde. Verbindungsinformationen können innerhalb der Task als einfache Liste realisiert sein.

Die vorliegenden Task-Graphen sind aus den jeweils angegebenen Quellen entnommen, in denen entweder direkt fertige Task-Graphen zur Verfügung stehen oder genug Informationen beinhaltet sind, um das benötigte Task-Graph-Format zu erzeugen.

In dieser Arbeit werden Task-Platzierungsverfahren für zukünftige, große Many-Core-Prozessoren untersucht. Die vorliegenden Task-Graphen umfassen allerdings nur eine beschränkte Anzahl an Tasks. Die vorgestellten Task-Graphen wurden aufgrund ihrer unterschiedlichen Kommunikationsmuster und ihrer Praxisrelevanz ausgewählt. Eine Einzelplatzierung macht allerdings wegen der Anzahl an Tasks nur bei den größeren Task-Graphen Sinn. Daher wurden für die Auswertung Pakete aus unterschiedlichen Applikationen geschnürt, um eine hohe Auslastung der Kern-Graphen zu erreichen. Verbreitete Benchmark-Anwendungen aus dem High-Performance-Bereich wurden aufgrund der gleichmäßigen Kommunikationsmuster und der langlebigen großen Tasks nicht weiter betrachtet.

4 Verwandte Arbeiten

Der **NMAP-Algorithmus** wurde entworfen, um Task-Graphen möglichst kommunikationseffizient auf Core-Graphen abzubilden (Murali und De Micheli 2004). Dazu wendet er ein heuristisches Verfahren an, dass in drei Phasen vorgeht. In der ersten Phase wird eine Initialplatzierung berechnet. Die zweite Phase führt eine Reihe von kürzeste-Wege-Berechnungen durch, die in der dritten Phase verwendet werden, um die erste Platzierung iterativ zu verbessern.

Phase eins platziert als Initialisierung die Task mit dem höchsten Kommunikationsbedarf auf einen Kern mit maximaler Anzahl von Nachbarn. Nun wird jeweils eine weitere Task zur Platzierung ausgewählt, die von den verbliebenen Tasks am meisten mit den bereits platzierten kommuniziert. Sind alle Tasks auf den Core-Graph abgebildet, ist die erste Phase abgeschlossen.

Phase zwei erstellt für jedes Paar von Tasks, die miteinander kommunizieren, einen Quadranten im Core-Graphen und berechnet in diesem den kürzesten Weg von Quelle zu Ziel. Auf dem berechneten Weg wird die verwendete Bandbreite reserviert.

Phase drei tauscht alle Kerne paarweise gegeneinander aus und führt jeweils noch einmal Phase zwei durch, um zu prüfen, ob so eine bessere Platzierung erzeugt wird.

Bei der Platzierung mehrerer Multimedia Applikationen erzielt NMAP signifikant geringere Bandbreiten- und Kommunikationskosten als konkurrierende Platzierungsverfahren.

Die Komplexität des NMAP-Algorithmus ist mit $\mathcal{O}(|U|^3 E \log |F|)$ angegeben wobei $|U|$ die Anzahl der Kerne, E die Anzahl der Verbindungen im Task-Graphen und $|F|$ die Anzahl der Kanten im Core-Graphen bezeichnet.

Durch die wiederholten Berechnungen der kürzesten Wege und die vielen Iterationen in Phase drei ist die reale Laufzeit des NMAP-Algorithmus bei wachsenden Größen der Task-Graphen und der Kern-Graphen sehr hoch. Zusätzlich werden beschädigte Elemente des Prozessors bei der Platzierungsentscheidung nicht in Betracht gezogen.

Aufgrund seiner guten Platzierungsergebnisse dient der NMAP-Algorithmus in Kapitel 7 als Maßstab, um die neu entwickelten Algorithmen mit dem aktuellen Stand der Technik zu vergleichen. Zu diesem Zweck wurde er in dem verwendeten Simulator anhand seiner Spezifikation neu implementiert.

Der **PMAP-Algorithmus** wurde mit dem Ziel entwickelt, Task-Graphen auf Core-Graphen beliebiger Topologien abzubilden (Koziris u. a. 2000). Er arbeitet in zwei Phasen, wobei die erste Phase eine Wurzel-Task platziert und die zweite Phase die restlichen Tasks abbildet.

Phase eins entfernt zuerst so lange Kanten aus dem Task-Graph, bis jede Task genau so viele Nachbarn besitzt, wie der Kern mit den meisten Nachbarn im Kern-Graph. Es werden dabei immer die Kanten mit der geringsten Gewichtung entfernt. Anschließend

werden sowohl Tasks als auch Kerne nach der Anzahl und Gewichtung ihrer Nachbarn sortiert und die kostspieligste Task dem am besten geeigneten Kern zugewiesen. Alle Nachbarn dieser Task werden zusätzlich auf benachbarte Kerne platziert.

Mit diesen Platzierungen beginnt Phase zwei, welche als Vorbereitung zuerst die vorher entfernten Kanten des Task-Graphen wiederherstellt. Aus der Menge der noch freien Cores wird der ausgewählt, der die meisten bereits belegte Nachbarn besitzt. Auf diesen Core wird nun eine Task platziert, die nicht mehr als i Schritte von allen Tasks auf den Nachbarn des gewählten Cores entfernt ist. Existiert keine solche Task, wird i, welches initial auf eins gesetzt ist, inkrementiert. Dieses Verfahren wird so lange wiederholt bis alle Tasks platziert sind.

Eine Betrachtung der algorithmischen Komplexität führen die Autoren nicht durch. Daher ist es nicht ohne Weiteres möglich, Aussagen über die Laufzeit des Algorithmus zu treffen. Sein Aufbau impliziert aber eine vergleichsweise kurze Laufzeit, da keine zeitaufwändigen Iterationsschritte benötigt werden. Der Einfluss defekter Prozessorelemente wird bei der Platzierung nicht in Betracht gezogen, kann jedoch durch die Struktur des Kern-Graphen modelliert werden.

Ein weiterer Ansatz, das Task-Platzierungsproblem zu lösen, ist die Verwendung eines **künstlichen Hormonsystems** (Brinkschulte, Pacher und Von Renteln 2007), (Brinkschulte, Von Renteln und Pacher 2008), (Pacher und Brinkschulte 2010). Hierbei werden drei Hormone verwendet, die eine Platzierungsentscheidung beeinflussen können. Der *Eager-Value* legt fest, wie gut eine Task auf einem Prozessor ausgeführt werden kann. Ist dieser Wert höher, so ist es wahrscheinlicher, dass eine Task auf diesem Prozessor platziert wird. Das *Suppressor-Hormon* versucht, die Ausführung einer Task auf einem Prozessor zu verhindern. Es kann dazu verwendet werden, überlastete Prozessoren unattraktiver zu machen oder doppelte Platzierung einer Task zu verhindern. Das *Accelerator-Hormon* begünstigt die Ausführung einer Task auf einem Prozessor und kann dazu verwendet werden, kommunizierende Tasks in einer Umgebung zusammenzufassen.

Das künstliche Hormonsystem geht von einem komplexen Many-Core oder einem verteilten System aus. Es können gleichzeitig mehrere Tasks auf einem Processing Element (PE) ausgeführt werden, so lange die verfügbaren Ressourcen ausreichen. Heterogene Core-Graphen sind ebenfalls möglich. Das Hormonsystem erfüllt einige der Selbst-X Eigenschaften, die im Forschungsbereich des Autonomic Computing (Kephart und Chess 2003), beziehungsweise Organic Computing vorgestellt wurden.

Für jede Task auf jedem PE wird eine Regelschleife durchgeführt, die so lange läuft, bis eine Entscheidung zur Platzierung oder Nichtplatzierung getroffen ist. Durch Zeitschranken in dieser Schleife lässt sich die Laufzeit des Algorithmus nach oben beschränken, was allerdings die Qualität der Platzierung beeinflussen kann. Durch diese Struktur wird auch eine fortlaufende Optimierung einer Platzierung zur Laufzeit einer Applikation ermöglicht.

Die Behandlung fehlerhafter Komponenten kann durch geeignete Verwendung des Suppressor-Hormons durchgeführt werden. Kerne mit defekten Komponenten erzeugen damit eine größere Menge des Suppressor-Hormons. Bei kleineren Defekten, wie beispielsweise unterbrochenen Verbindungen, kann eine geringere Menge des Hormons

ausgeschüttet werden.

Das Hormonverfahren ist problematisch für einen Many-Core-Prozessor, da die Notwendigkeit besteht, eine hohe Anzahl an Hormonwerten auf dem Prozessor zu verwalten. Berechnungen und Optimierungen müssen somit auf einem Kern mit viel lokalem Speicher und Rechenkapazität durchgeführt werden.

Der **Branch-and-Bound-Algorithmus** (Jingcao und Marculescu 2003), welcher von Jingcao und Marculescu vorgeschlagen wurde, erzeugt eine Platzierung von IP-Cores in NoCs, was mit der Platzierung von Tasks auf Cores vergleichbar ist. Besonderes Augenmerk liegt in dieser Arbeit auf einer möglichst energieeffizienten Kommunikation. Da die verwendete Energieformel von der Manhattan-Distanz zweier kommunizierender Kerne dominiert wird, ist die Zielsetzung eine möglichst geringe Entfernung zwischen Kommunikationspartnern.

Die Abbildung der IP-Cores auf Positionen im NoC wird als Baum repräsentiert. Dies ermöglicht eine effiziente Navigation des Suchraumes, da schon beim Konstruieren und Erweitern der Baumstruktur teure und ungültige Äste entfernt werden. Jedes Baumelement bildet den Zustand einer teilweisen oder vollständigen Abbildung aller IP-Cores im NoC ab. Im Wurzelknoten sind somit noch keine IP-Cores abgebildet, wohingegen jedes Blatt des Baumes eine vollständige Platzierung darstellt. Knoten innerhalb des Baumes sind unvollständige Platzierungen.

Der Algorithmus expandiert den Baum schrittweise von der Wurzel beginnend, während er in jedem Schritt prüft, ob die neu hinzugefügten Knoten gültig und kosteneffizient sind. Die Gültigkeit hängt davon ab, ob das Interconnect die Bandbreitenanforderungen der Platzierung erfüllen kann. Energiekosten werden jeweils innerhalb der bereits platzierten IP-Cores berechnet. Knoten, die eine der beiden Bedingungen verletzen, werden samt all ihrer Kinder entfernt.

Der vorgeschlagene Algorithmus hat eine wesentlich bessere Laufzeit als das in der Publikation verglichene Verfahren mit simulierter Abkühlung, benötigt aber trotzdem für Systeme mit 10x10 Cores mehrere Minuten für die Platzierungsberechnung. Somit ist der Algorithmus ungeeignet für eine Platzierungsberechnung beim Programmstart. Defekte Prozessorelemente werden nicht in Betracht gezogen. In vielen Fällen sind fehlerhafte Elemente aber durch eine Anpassung der NoC-Struktur abbildbar.

Das Problem des Task-Mappings kann auch durch Metaheuristiken wie **genetische Algorithmen** oder **simulierte Abkühlung (simulated annealing)** gelöst werden. Diese Ansätze wurden in (Ascia, Catania und Palesi 2005), (Ascia, Catania und Palesi 2006) und (Jingcao und Marculescu 2003) betrachtet. Die Grundalgorithmen sind vom Platzierungsproblem unabhängig und müssen nur an einigen Stellen, wie beispielsweise der Bewertung der Ergebnisse, an die eigentliche Problemstellung angepasst werden. Bei geeigneter Anwendung vermeiden die Algorithmen lokale Maxima der Platzierungsqualität und liefern bei einer ausreichend hohen Iterationsanzahl sehr gute oder sogar optimale Platzierungsergebnisse. Allerdings ist die Komplexität dieser Algorithmen sehr groß und die Bestimmung einer guten Platzierung dauert damit im Allgemeinen zu lange, um sie beim Programmstart durchzuführen. Metaheuristiken eigenen sich infolge dessen eher für Offline-Platzierungsbestimmungen, wo sie eine gute Alternative

zu Brute-Force-Berechnungen einer optimalen Platzierung darstellen.

Es ist allerdings möglich, eine angemessene Laufzeit durch starke Anpassung einer Metaheuristik auf das Problemfeld der Task-Platzierung zu erreichen. Dies wird beispielsweise durch den **Optimized Simulated Annealing Algorithmus (OSA)** (Radu und Vintan 2011) belegt. OSA beschränkt zum einen den Suchraum sehr stark indem er die untere und obere Schranke der Temperaturverteilung festlegt. Zusätzlich führt eine Anpassung der Akzeptanzfunktion dazu, dass eine gleich gute neue Bewertung nur mit einer 50%-igen Wahrscheinlichkeit akzeptiert wird. Dadurch ist der endgültige Systemzustand schon wesentlich früher erreichbar. Außerdem passt OSA den Task-Tauschmechanismus derart an, dass nicht mehr zufällig zwei Tasks zum Vertauschen ausgewählt werden, sondern wählt beide Tauschpartner aufgrund ihrer Kommunikation aus, was im Endeffekt dazu führt, dass intensiv kommunizierende Tasks voneinander angezogen werden.

Durch diese problemspezifischen Anpassungen wird erreicht, dass der Algorithmus eine vergleichbar gute Laufzeit erzielt, seine Ergebnisse sich jedoch nicht stark von generalisierten simulated annealing Methoden unterscheiden. Der Algorithmus behandelt keine fehlerhaften Prozessorkomponenten.

Im Bereich der Supercomputer, die aus mehreren verbundenen Maschinen bestehen, lassen sich Parallelen zur Task-Platzierungsproblematik auf Many-Core-Prozessoren ziehen. Klassischerweise erfüllen hier Management-Softwarepakete die Zuteilung von Rechenlast zur physischen Maschine. So ermöglicht das **Extreme Cloud Administration Toolkit (xCAT)** (IBM 2012b) von IBM das Verwalten und Administrieren vieler verbundener Rechner. Task-Platzierung kann der Tivoli Workload Scheduler LoadLeveler (IBM 2012a) erledigen. Die Verteilung der Tasks geschieht nach Prioritäten und Auslastung des Systems. Außerdem wird die Nutzung beobachtet, um nachträglich Nutzungszeiten abrechnen zu können. Die zur Platzierung genutzten Algorithmen sind nicht veröffentlicht. xCat-Cluster können allerdings auch von anderen Task-Platzierungs-Servern genutzt werden.

Eine Alternative ist das im Quelltext frei verfügbare **Simple Linux Utility for Resource Management (SLURM)** (Lawrence Livermore National Laboratory 2012), (Georgiou 2010). SLURM verwendet einen einfachen Best-Fit-Algorithmus (Pascual, Navaridas und Miguel-Alonso 2009), basierend auf dem Hilbert-Kurven-Scheduling (Drozdowski 2009). Hierbei wird ein mehrdimensionales Task-Scheduling-Problem in ein eindimensionales Problem umgewandelt. Gleichzeitig behalten kommunizierende Tasks eine Nachbarschaftsbeziehung. Das Scheduling-Verfahren ist austauschbar und kann somit auf spezielle Gegebenheiten angepasst werden (Yoo, Jette und Grondona 2003).

Der **Maui Cluster Scheduler** (Adaptive Computing 2013a) und seine kommerzielle Variante die Moab Cluster Suite (Adaptive Computing 2013b) werden von der Firma Adaptive Computing entwickelt. Sie stellen Scheduling-Komponenten für verschiedene Cluster-Management-Systeme zur Verfügung. SLURM und xCAT können auf diese Komponenten zurückgreifen, um eine ausgefeiltere Platzierungsstrategie zu nutzen.

Zu diesem Zweck werden viele Parameter beachtet, welche unter anderem Fairness-Bedingungen, Prioritäten und Lokalität der zu platzierenden Tasks beinhalten. Auch spezielle Hardwareanforderungen lassen sich spezifizieren. Moab erweitert Maui, indem es feingranularere Einstellungsmöglichkeiten bietet und das Nutzerinterface erweitert. Intern arbeiten Maui mit *Jobs*, *Knoten*, *Reservierungen*, *QOS-Strukturen* und *Policies*. Jobs und Knoten sind jeweils Task-Graphen und Kernen gleichzusetzen. Für jede Task wird zuerst festgestellt, welche Zielknoten aufgrund der Anforderungen zur Verfügung stehen. Anschließend wird der geeignetste Kandidat durch die *Node Allocation Policy* ausgewählt und die Platzierung durchgeführt. Es stehen verschiedene, global oder pro Job konfigurierbare *Node Allocation Policies* bereit. Diese reichen von einer einfachen Selektion des ersten Knotens in der Liste über blockweise Platzierung bis hin zu ressourcebasierten Auswahlverfahren, welche auf minimale CPU-Auslastung, gute Lastbalancierung oder höchste Geschwindigkeit der ausgewählten Knoten achten.

Die Algorithmen für die Belegung von Rechenclustern sind zwar denen zur Task-Platzierung ähnlich, sie müssen jedoch einige Nebenziele erfüllen, die auf Many-Core-Prozessoren unwichtig sind. Abrechnung des Ressourcenverbrauchs ist beispielsweise in einem Many-Core nicht nötig. Außerdem ist auf Rechenclustern die initiale Platzierung sehr wichtig, da die meisten Aufgaben sehr langlebig sind. Somit lohnt es sich, mehr Zeit und Aufwand in die Platzierungsberechnung zu stecken. Zusätzlich betrachten die Platzierungen auf Rechenclustern auch die Heterogenität der Hardware. Die Task-Platzierungs-Applikationen besitzen die Möglichkeit, Aufgaben für spezielle Hardwaregegebenheiten, wie beispielsweise eine bestimmte GPU, vorzusehen und nur geeignete Rechenknoten auszuwählen. Fehlertoleranz ist im Cluster-Bereich ein wichtiges Thema. Die Situation ist allerdings etwas entspannter als bei Many-Core-Prozessoren, da fehlerhafte Komponenten einfach ersetzt werden können. Die Toleranz von permanenten Defekten ist daher nicht gefragt.

	Laufzeit	**Einsatzgebiet**	**Fehlertoleranz**	**optimale Platzierung**
NMAP	+	On-Chip	-	-
PMAP	++	On-Chip	-	- -
Hormon	0	distributed systems	+	+
branch & bound	-	IP-Cores im NoC	-	++
Abkühlung	- -	IP-Cores im NoC	0	++
Genetisch	- -	IP-Cores im NoC	0	++
OSA	0	IP-Cores im NoC	0	++
xCAT	++	Cluster	0	0
SLURM	++	Cluster	0	0
Maui	++	Cluster	0	0

Tabelle 4.1: Gegenüberstellung der vorgestellten Platzierungsverfahren

Die in diesem Kapitel beschriebenen Veröffentlichungen lassen sich in mehrere Kategorien der Task-Platzierung einordnen. In Tabelle 4.1 werden sie nach Laufzeit, Einsatzgebiet, Fehlertoleranz und Erreichbarkeit einer optimalen Platzierung verglichen. Als Maßstab für die in dieser Arbeit entwickelten Algorithmen wurde NMAP herangezogen, da er bessere Platzierungsergebnisse als PMAP erzeugt und für On-Chip-Netzwerke gedacht ist. Gänzlich disqualifiziert sind alle Algorithmen mit einer schlechten Laufzeit, da die Platzierungsentscheidung bei jedem Programmstart durchgeführt werden soll. Platzierungsverfahren aus dem Bereich der Rechencluster scheiden als Vergleichsgröße ebenfalls aus, da sie mit einem anderen Modell arbeiten und auf andere Problemfelder ausgerichtet sind.

5 Statische Task-Platzierung

Die statische Task-Platzierung verwendet eine globale Sicht auf den Prozessor, um eine möglichst vorteilhafte Konfiguration einzelner Tasks eines Programmes zu erzeugen. Das Optimierungsziel ist hierbei, kommunizierende Tasks in möglichst kurzer Entfernung zueinander zu platzieren. Alle Tasks eines Task-Graphen werden direkt zu Beginn der Programmausführung auf Kerne abgebildet. Die globale Sicht ist sinnvoll, da auch potentiell defekte Elemente eines Prozessors in die Kalkulation mit einbezogen werden müssen, um eine Platzierung nicht stark negativ zu beeinflussen.

Wenn also eine statische Platzierung verwendet wird, bestimmt der Platzierungsalgorithmus beim Start einer Applikation auf dem betrachteten Many-Core-System einen geeigneten Kern für jede Task des Task-Graphen der Applikation, damit die benötigten Daten und Instruktionen an die richtige Stelle transferiert werden können. Damit ist eine optimale Ausführung gewährleistet. Dies vermeidet Verzögerungen, bei denen der Programmablauf die Zuteilung weiterer benötigter Tasks auf Kerne abwarten muss. Außerdem steht so das Kommunikationsprofil schon beim Start der Anwendung fest, was bei Routingentscheidungen in Betracht gezogen werden kann. Zusätzlich ist es beim Programmstart möglich, bereits bekannte Defekte zu kompensieren. Sind einzelne Komponenten des Chips nicht nutzbar, kann ein statischer Platzierungsalgorithmus eine Applikation so positionieren, dass die Ausführung nicht oder nur gering negativ beeinflusst wird.

Durch die Ausführung bei jedem Programmstart scheiden Platzierungsalgorithmen mit langer Laufzeit und aufwändigen Berechnungen aus. Eine Platzierungsentscheidung muss möglichst schnell erfolgen, soll aber nicht viele Kompromisse bei der Qualität des Resultats eingehen müssen.

Die Scheduling-Verfahren der derzeit verbreitetsten Betriebssysteme verfolgen andere Ziele. Sie lösen das Problem, eine hohe Anzahl von Prozessen auf eine beschränkte Anzahl von Prozessorkernen abzubilden. Dabei ist ein Zeitmultiplexing wichtiger als das Platzmultiplexing. Tasks sind unterbrechbar und werden zu vom Scheduler bestimmten Zeiten ausgeführt beziehungsweise unterbrochen. Fast alle verbreiteten Betriebssysteme setzen derzeit beim Scheduling auf eine *Multilevel Feedback Queue* (Kleinrock und Muntz 1972). Die Ausnahmen bilden AIX, wo unter verschiedenen FIFO und Round Robin Implementierungen gewählt werden kann, und neuere Linux Systeme, die eine Variante des aus der Netzwerkwelt bekannten Fair Queueings nutzen.

Die hier vorgestellte statische Task-Platzierung trifft hingegen die Annahme, dass mehr Prozessorkerne als Tasks zur Verfügung stehen. Einzelne Tasks sind außerdem nicht unterbrechbar, was die Ausführung auf den Kernen vereinfacht, da Tasks nicht ausgelagert werden müssen oder eine Warteschlange von Tasks vorgehalten werden muss.

Bei der statischen Platzierung findet im Gegensatz zur dynamischen keine Replat-

zierung oder Anpassung der bestehenden Platzierung nach dem Start des Programmes statt. Für Ausfälle zur Laufzeit eines Programmes bietet sie also keine ausreichende Lösung. Im Folgenden wird der *Konnektivitätssensitive Algorithmus* als einfache Heuristik zur Task-Platzierung und die *Fitting* und *Favorite Neighbour Algorithmen* als dessen Erweiterung motiviert und vorgestellt.

5.1 Konnektivitätssensitiver Algorithmus

Das Ziel des bereits veröffentlichten *Konnektivitätssensitiven Algorithmus* (S. Schlingmann u. a. 2011) ist es, Kommunikationswege kurz zu halten. Da die Kommunikation zwischen Kernen auch bei Verwendung eines Network on Chip einen Flaschenhals darstellt, hilft das Vermeiden von unnötigen Latenzen durch die Reduzierung der Länge der Kommunikationswege die Performance des Gesamtsystems zu verbessern. Kürzere Kommunikationswege verringern auch die Wahrscheinlichkeit einer Verstopfung im Verbindungsnetz, da Daten im Schnitt weniger lang im Netz verweilen. Ein wichtiger Faktor für die Platzierung ist deshalb die Kommunikation der Tasks mit ihren Nachbarn im Task-Graph. Hierbei ist sowohl die Anzahl der Kommunikationspartner als auch die Menge der versendeten Daten interessant. Optimal wäre eine Platzierung, bei der jede Nachbartask, mit der eine Task kommunizieren muss, auf einem direkt benachbarten Kern platziert wurde. Die Kommunikationsdistanz ist in diesem Fall ein Hop pro Verbindung. Ist eine solche Platzierung nicht möglich, muss priorisiert werden, welche Tasks auf direkt benachbarten Kernen zu platzieren sind. Für den Kommunikationsoverhead ist es zielführend, Tasks, die in höherem Volumen oder höherer Frequenz miteinander kommunizieren, auf benachbarten Kernen zu platzieren. Als Maß für die Qualität einer betrachteten Platzierung wird der Bezeichner Δ_{Com} verwendet. Δ_{Com} beschreibt den gewichteten Unterschied der Kommunikationsdistanzen zwischen einem unplatzierten und einem platzierten Task-Graphen:

$$\Delta_{Com} = Com_{Placed} - Com_{Task}$$

Com_{Placed} und Com_{Task} sind dabei bei einem Task Graphen $T(V_T, E_T)$ mit der Menge der Tasks V_T und der Menge der Verbindungstupel E_T wie folgt definiert:

$$Com_{Placed} = \sum dist(a,n) * weight(a,n)\ \forall a \in V_T\ \forall n \mid (a,n) \in E_T$$
$$Com_{Task} = \sum weight(a,n)\ \forall a \in V_T\ \forall n \mid (a,n) \in E_T$$

Die Funktion $dist(a,b)$ liefert hierbei die Anzahl der Hops im Core-Graph zwischen den platzierten Tasks a und b. $weight(a,b)$ gibt die Gewichtung der Verbindung zwischen den Tasks a und b im Task-Graph zurück.

Für die initiale Bewertung eines Task-Graphen wird sowohl die Anzahl der Kommunikationspartner als auch das Kommunikationsvolumen verwendet. Nach der Bewertungsphase wird der Rest des Algorithmus 1 ausgeführt bis alle Tasks einem Kern zugewiesen sind.

Eingabe : Task-Graph, Kern-Graph
Ausgabe : Zuordnung Task auf Kern

Bewerte Tasks $T(V_T, E_T)$
Bewerte Kerne $C(V_C, E_C)$

Wähle Task t mit höchster Bewertung
Platziere Task t auf Kern c mit höchster Bewertung

Platziere alle Nachbarn t_N von t in Platzierungsliste P als Tupel (t_N, c)

for *jedes Element* (t', c') *in* P **do**
  if t' *ist noch nicht platziert* **then**
    Platziere t' auf c_{new} mit minimalem Abstand zu c'
    Platziere Nachbarn von t' in P als Tupel (t_N, c_{new})
  end
end

Algorithmus 1 : *Konnektivitätssensitiver Algorithmus* (pseudo code)

Die Bewertung einer einzelnen Task a wird nach der Formel

$$\sum weight(a, n)\ \forall n \mid (a, n) \in E_T$$

berechnet, wobei E_T die Menge der Verbindungen im Task-Graph bezeichnet und $weight(a, b)$ das Gewicht der Verbindung von Task a zu Task b im Task-Graph zurückgibt. Bei einem ungewichteten Task-Graphen liefert $weight(a, b)$ immer den Wert eins zurück. Die Bewertung eines Kerns im Kern-Graphen ist die Anzahl seiner ausgehenden Kanten. Ist eine zum Kern verbundene Leitung defekt oder ist der Kern und Netzwerkrouter, zu dem diese Leitung führt, ausgefallen, so wird die Leitung bei der Bewertung nicht gezählt. Das liegt darin begründet, dass der durch diese Leitung angeschlossene Kern nicht zur Platzierung oder zumindest nicht zur Platzierung eines direkten Nachbarn verwendet werden kann.

Nachdem die Bewertungsphase abgeschlossen ist, wird als erster Schritt die Task mit der höchsten Bewertung auf dem Kern mit der höchsten Bewertung platziert. Existieren mehrere Tasks oder Kerne mit identischen höchsten Bewertungen wird einer per Zufall bestimmt, da die Wahl aus dieser Gruppe für den weiteren Verlauf des Algorithmus egal ist. Nach Platzierung der initialen Task t, werden alle Nachbarn von t in die Platzierungsliste eingefügt. Die Platzierungsliste enthält Tupel von Tasks und Kernen und repräsentiert die Menge der Tasks, die in der nächsten Iteration des Algorithmus platziert werden können. Der mit einer Task verknüpfte Kern in einem Tupel ist als Positionsempfehlung zu verstehen. Die Task des Tupels sollte möglichst nah bei diesem Kern platziert werden. Der angegebene Kern kann allerdings selbst nicht Platzierungsziel sein, da er laut Algorithmus schon mit einer Task belegt ist.

Der Hauptteil des Algorithmus iteriert über die Elemente der Platzierungsliste bis diese geleert ist. In jeder Iteration wird ein Element (t', c') aus der Platzierungsliste P entfernt und, falls t' noch nicht platziert wurde, einem freien Kern c_{new} zugewiesen. Der

Kern wird mit minimalem Abstand zu c' gewählt. Die Abstandsberechnung betrachtet nur funktionsfähige Verbindungen. Ist t' erfolgreich auf c_{new} platziert, werden noch alle Nachbartasks t_N von t' als Tupel (t_N, c_{new}) in die Platzierungsliste eingefügt. Optional kann an dieser Stelle geprüft werden, ob t_N bereits platziert ist, allerdings ist dies nicht unbedingt nötig, um die Terminierung des Algorithmus sicherzustellen. Der Performancegewinn bei einer derartigen Optimierung fällt nicht ins Gewicht, da diese Überprüfung im Algorithmus sowieso zu Beginn jeder Iteration durchgeführt wird. Die Platzierungsliste fungiert als FIFO-Puffer, begünstigt also keine neuen Kandidaten und führt auch keine Entfernung von Duplikaten durch.

Um eine Task in der Nähe einer bereits platzierten Task zu positionieren, wird eine Breitensuche verwendet, die einen freien und funktionsfähigen Kern zu finden soll, der eine minimale Distanz zum Kern der platzierten Task besitzt.

Ist die Platzierungsliste leer, sind alle Tasks auf geeigneten Kernen platziert und der Algorithmus terminiert.

Der Algorithmus kann in $\mathcal{O}(|V_C|^2 + |V_C||E_C|)$ ausgeführt werden. Der zeitaufwändige Teil der Berechnung besteht in der Platzierungsschleife und der Platzierung mit minimalem Abstand. Die Tasks in V_T sind bereits als bewertete, sortierte Liste in der Eingabe enthalten, da diese Bewertung offline möglich ist. Dasselbe gilt anfangs für die Kerne in V_C. Wurde bereits ein Task-Graph platziert, ist die Bewertung einiger Elemente in V_C unterschiedlich und muss somit wiederholt werden, was einen Aufwand von $\mathcal{O}(|V_C| \log |V_C|)$ erzeugen würde. Dieser Aufwand kann allerdings schon direkt nach der Platzierung des vorhergehenden Task-Graphen erfolgen. Die Laufzeit dieser Operationen findet daher im Offline-Bereich statt.

Die Schleife, die P abarbeitet hat im schlimmsten Fall, einem vollständig verbundenen Task-Graphen, $(|V_T| - 1)^2$ Durchläufe. Werden Duplikate bereits beim Einfügen der Nachbarn in die Platzierungsliste erkannt, verringert sich die Anzahl der Schleifendurchläufe auf $|V_T| - 1$, doch erzeugt dies einen Zusatzaufwand von $|V_T| - 1$ bei der Erkennung der Duplikate, was den Gesamtaufwand wieder auf $(|V_T| - 1)^2$ anhebt.

Für die Platzierung mit minimalem Abstand kann eine Breitensuche verwendet werden, die eine Laufzeit von $\mathcal{O}(|V_C| + |E_C|)$ erreicht. Diese wird pro Task einmal, also $|V_T| - 1$ mal ausgeführt, was zu einer akkumulierten Laufzeit von $|V_C|^2 + |V_C||E_C|$ für die Abstandsberechnungen führt, da im schlimmsten Fall $|V_T| = |V_C|$. gilt.

Diese Betrachtungen erlauben es, die Gesamtlaufzeit des Algorithmus mit $\mathcal{O}(2|V_C|^2 + |V_C||E_C|)$, beziehungsweise $\mathcal{O}(|V_C|^2 + |V_C||E_C|)$ anzugeben.

Ein Beispiel für die Platzierung von Tasks auf Kerne wird in Abbildung 5.1 gegeben. Ein Fork-Join-Task-Graph mit drei Arbeitertasks wird auf ein Maschennetz der Größe 3x3 abgebildet. Nach der Bewertungsphase wird Task A auf dem zentralen Kern platziert, da diese Task eine der beiden mit der höchsten Anzahl an ausgehenden Kanten ist und der zentrale Kern die größte Anzahl an Nachbarn besitzt. Gewichtungen von Verbindungen werden in diesem Beispiel zum einfacheren Verständnis vernachlässigt. Zu diesem Zeitpunkt könnte der Algorithmus durchaus auch Task E anstatt Task A platzieren, da sie dieselbe Anzahl an Nachbarn im Task-Graph hat. Anschließend werden die Tasks B, C und D mit dem zentralen Kern als Referenzpunkt in die Platzierungsliste P eingefügt. Wird jetzt beispielsweise die Task B als nächstes zur Platzierung ausgewählt, so wird Task E auch in die Platzierungsliste eingesetzt. Als

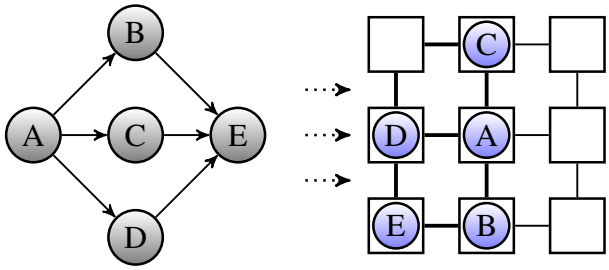


Abbildung 5.1: Ein Beispielergebnis des *Konnektivitätssensitiven Algorithmus*

Referenzpunkt für Task E wird der Kern gewählt, auf dem Task B platziert wurde. Das bedeutet, dass Task E nahe zu Task B platziert werden wird, die Distanz zu ihren weiteren Kommunikationspartnern aber keine Rolle spielt.

Defekte Elemente des Prozessors werden vom *Konnektivitätssensitiven Algorithmus* bei der Bewertung der Kerne und bei der Platzierung mit minimalem Abstand betrachtet. In der Bewertungsphase werden defekte Links im Core-Graph von der Bewertung eines Cores abgezogen:

$$
\begin{aligned}
Bewertung_{Core} &= |N| - |N_{unerreichbar}| \\
N &= \{n \mid (core, n) \in E_C\} \\
N_{unerreichbar} &= \{n \mid (core, n) \in E_C \wedge \\
&\qquad (core\_defekt(n) \vee link\_defekt(core, n) \vee router\_defekt(n))\}
\end{aligned}
$$

Die Funktionen $core\_defekt()$, $link\_defekt()$ und $router\_defekt()$ liefern Wahr zurück, falls der angegebene Core, Link oder Router defekt ist.

Die Abstandsberechnung betrachtet nur den Abstand, der über funktionierende Links und Router erreichbar ist. Hier wird eine Breitensuche über funktionierende Verbindungen durchgeführt, was sicherstellt, dass defekte Elemente nicht verwendet werden.

5.2 *Fitting Algorithmus*

Bei der Konzeption des *Konnektivitätssensitiven Algorithmus* wurde nicht beachtet, dass eine Task unter Umständen auf einem Kern platziert wird, der mehr Verbindungen zur Verfügung hat als eigentlich verwendet werden. Dadurch reserviert der Algorithmus in einigen Situationen mehr Ressourcen als nötig. Kerne mit vielen Verbindungen könnten gewinnbringender für Tasks mit vielen Verbindungen eingesetzt werden. An diesem Punkt setzt der *Fitting Algorithmus* an, der eine Erweiterung des *Konnektivitätssensitiven Algorithmus* darstellt.

Der *Fitting Algorithmus* bewertet ebenfalls zuerst die Kerne nach dem selben Verfahren wie der *Konnektivitätssensitive Algorithmus*. Analog werden auch die Tasks

Eingabe : Task-Graph, Kern-Graph
Ausgabe : Abbildung Task auf Kern

Bewerte Tasks $T(V_T, E_T)$
Bewerte Kerne $C(V_C, E_C)$

Wähle Task t mit höchster Bewertung
Platziere Task t auf Kern c mit passender Bewertung

Platziere Nachbar t_N von t in Platzierungsliste P als Tupel (t_N, c)

for *jedes Element* (t', c') *in* P **do**
  if t' *ist noch nicht platziert* **then**
    Platziere t' auf c_{new} mit minimalem Abstand zu c' und passender Bewertung
    Platziere Nachbarn von t' in P als Tupel (t_N, c_{new})
  end
end

Algorithmus 2 : *Fitting Algorithmus* (pseudo code)

bewertet, allerdings wird zusätzlich zur Bewertung pro Task auch die ungewichtete Anzahl der Verbindungen zu Nachbartasks gespeichert. Als Initialknoten wird die Task mit der höchsten Bewertung gewählt und auf einen passenden Kern platziert. Die Eignung eines Kerns c wird hierbei durch den Vergleich der Anzahl seiner benutzbaren Nachbarn $|N_K|$ ($N_K = \{n \mid (c, n) \in E_C\}$) mit der Anzahl der Nachbarn $|N_T|$ ($N_T = \{n \mid (t, n) \in E_T\}$) der zu platzierenden Task t bestimmt. Ist ein Kern mit $|N_K| = |N_T|$ verfügbar, so wird dieser als Platzierungsziel gewählt. Ist dies nicht der Fall, wird ein Kern mit $|N_K| > |N_T|$ verwendet. Kann auch diese Bedingung nicht erfüllt werden, so entscheidet sich der Algorithmus für einen Kern mit $|N_K| < |N_T|$. In allen Fällen wählt der Algorithmus den Kern aus, dessen $|N_K|$ die geringste Differenz zu $|N_T|$ aufweist.

Der Rest des *Fitting Algorithmus* läuft weitestgehend analog zum *Konnektivitätssensitiven Algorithmus* ab. Der einzige Unterschied liegt in der Platzierung, die nun zuerst die Menge der freien Knoten mit minimalem Abstand zum Referenzknoten c' bestimmt und danach die Bewertungskriterien des *Fitting Algorithmus* anwendet. Damit liegt die Bewertung des gewählten Knoten so nahe wie möglich an der Anzahl der Nachbarn der zu platzierenden Task, die Distanz ist aber weiterhin höher gewichtet als ein passender Kern.

Die Terminierung des Algorithmus erfolgt wie zuvor sobald die Platzierungsliste leer ist.

Die algorithmische Komplexität liegt in $\mathcal{O}(|V_C|^2 + |V_C||E_C|)$. Die komplexen Teile der Platzierungsberechnung sind wie beim *Konnektivitätssensitiven Algorithmus* die Abarbeitung des Platzierungsschleife und die Platzierung mit minimalem Abstand.

Task-Graph und Core-Graph sind bereits offline vorsortiert. Im Fall einer Platzierung mehrerer Task-Graphen muss der Core-Graph gegebenenfalls mit einem Aufwand von $\mathcal{O}(|V_C| \log |V_C|)$ neu sortiert werden. Dies kann direkt nach der Platzierung eines

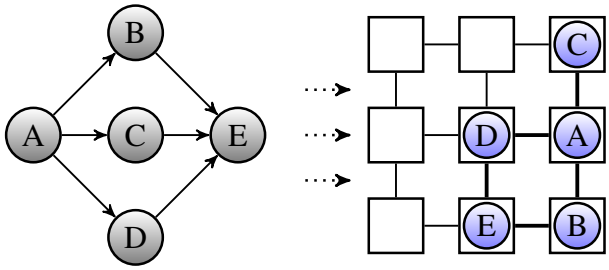


Abbildung 5.2: Ein Beispielergebnis des *Fitting Algorithmus*

vorhergehenden Task-Graphen geschehen.

Die Abarbeitung der Platzierungsliste P benötigt im schlimmsten Fall $(|V_T| - 1)^2$ Iterationen. Eine genauere Betrachtung der Iterationsanzahl findet sich in der Betrachtung der Laufzeit des *Konnektivitätssensitiven Algorithmus*.

Die Platzierung mit minimalem Abstand kann wie zuvor mit einer Breitensuche durchgeführt werden. Allerdings muss die Breitensuche so lange laufen, bis alle Kerne mit minimalem Abstand gefunden wurden, um dann die Anzahl der Nachbarn vergleichen zu können. Beim *Konnektivitätssensitiven Algorithmus* konnte die Breitensuche terminieren, sobald ein geeigneter Kandidat gefunden wurde. Dies ist hier nur möglich, wenn ein Kern mit der genau passenden Anzahl an nutzbaren Nachbarn gefunden wird. Somit kann also eine Breitensuche nach einem Kern mit minimalem Abstand plus eins angenommen werden, was aber nichts an der maximalen Laufzeit von $\mathcal{O}(|V_C| + |E_C|)$ ändert.

Damit wird also eine akkumulierte Laufzeit von $\mathcal{O}(|V_T||V_C|+|V_T||E_C|)$ erreicht, was äquivalent zu $\mathcal{O}(|V_C|^2 + |V_C||E_C|)$ ist, da im aufwändigsten Fall $|V_T| = |V_C|$ gilt. Der *Fitting Algorithmus* besitzt also die selben Laufzeitschranken wie sein Vorgänger.

Auch der *Fitting Algorithmus* kann mit fehlerhaften Elementen im Many-Core umgehen. Defekte Kerne können selbstverständlich keine Tasks enthalten und werden somit nicht betrachtet. Analog zum *Konnektivitätssensitiven Algorithmus* werden bei der Bewertung der Kerne defekte oder unbenutzbare Links im Core-Graph von der Bewertung eines Cores abgezogen:

$$
\begin{aligned}
Bewertung_{Core} &= |N| - |N_{unerreichbar}| \\
N &= \{n \mid (core, n) \in E_C\} \\
N_{unerreichbar} &= \{n \mid (core, n) \in E_C \wedge \\
&\qquad (core\_defekt(n) \vee link\_defekt(core, n) \vee router\_defekt(n))\}
\end{aligned}
$$

Die Funktionen $core\_defekt()$, $link\_defekt()$ und $router\_defekt()$ liefern wie schon zuvor Wahr zurück, falls der angegebene Core, Link oder Router defekt ist.

Bei der Abstandsberechnung wird wieder nur der Abstand zu Cores betrachtet, die über funktionierende Links und Router erreichbar sind. Die verwendete Breitensuche darf also nur funktionierende Verbindungen und Router benutzen.

In der Abbildung 5.2 wird im gleichen Szenario, das auch für den *Konnektivitätssensitven Algorithmus* gewählt wurde, der *Fitting Algorithmus* angewendet. Kandidaten für die Initialplatzierung sind also wiederum Task A und Task E. Wird Task A gewählt, so bieten sich im Kern-Graph vier Kandidaten für eine Platzierung. Diesmal ist die Mitte ungeeignet, da Task A nur drei Verbindungen besitzt und daher Kerne präferiert, die ebenfalls nur drei Nachbarn besitzen. Ist Task A platziert, werden die Tasks B, C und D in die Platzierungsliste eingefügt. Diese Tasks präferieren die Positionen direkt über und unter Task A, da diese Kerne zwei Nachbarn haben, was der Anzahl der Nachbarn der Tasks entspricht. Task D wird in die Mitte des Kern-Graphen gesetzt, da kein weiterer Kern mit Distanz gleich eins zu Task A existiert und der Algorithmus die kürzere Distanz höher bewertet als die eventuelle Beeinflussung der Platzierung ausstehender Tasks. Im dargestellten Beispiel wurde B als zweites platziert und hat damit Task E in die Platzierungsliste eingefügt. E wird mit minimaler Distanz zu B platziert und in diesem Fall sogar auf einen Kern mit passender Anzahl an Nachbarn.

Das Beispiel zeigt den Effekt des *Fitting Algorithmus* recht anschaulich. Platzierte Tasks landen eher am Rand des Kern-Graphen und nehmen keine zentrale Position ein. Dadurch steht dem Algorithmus weniger Platz zur Verfügung, um weitere Tasks zu platzieren, da der Task-Graph nicht in alle Richtungen wachsen kann. Ein positiver Effekt der Randplatzierung ist allerdings, dass mehr zusammenhängender Platz im Kern-Graphen zur Platzierung weiterer Task-Graphen zur Verfügung steht. Sollen also mehrere Programme, repräsentiert durch zugehörige Task-Graphen, gleichzeitig auf dem betrachteten Many-Core-Prozessor ausgeführt werden, verspricht der *Fitting Algorithmus* eine bessere Durchschnittsperformance anstatt den erstplatzierten Task-Graphen stark zu bevorzugen.

5.3 *Favorite Neighbour Algorithmus*

In einigen Fällen kann die naive Gleichbehandlung von Verbindungen im Task-Graph, wie es der *Konnektivitätssensitiven Algorithmus* oder der *Fitting Algorithmus* durchführt, zu unvorteilhafter Platzierung der Tasks führen. Die Abarbeitung der zu platzierenden Tasks wird ohne besondere Präferenz durchgeführt und das Gewicht der Verbindungen fließt einzig in die initiale Platzierungsentscheidung ein.

Der *Favorite Neighbour Algorithmus* wurde entwickelt, genau dieses Problem zu lösen. Zusätzlich zu der Bewertung der Tasks, wie sie auch im *Fitting Algorithmus* vorgenommen wird, bestimmt der *Favorite Neighbour Algorithmus* zu jeder Task die favorisierte Nachbartask. Es wird genau die Nachbartask favorisiert, die über die Verbindung mit der höchsten Gewichtung erreichbar ist. Der Algorithmus stellt nun sicher, dass die favorisierte Nachbartask, wenn irgend möglich, mit Abstand eins zu der favorisierenden Task platziert wird.

Initial werden analog zum *Fitting Algorithmus* die Tasks und Kerne bewertet. Zu jeder Task wird außerdem eine favorisierte Nachbartask bestimmt. Da anfangs noch keine Task platziert ist, wird als erstes die Task mit der höchsten Bewertung auf einen Kern mit passender Bewertung platziert. Diese erste Platzierung wird also nach den selben Kriterien wie beim *Fitting Algorithmus* vorgenommen. Alle noch nicht platzier-

Eingabe : Task-Graph, Kern-Graph
Ausgabe : Abbildung Task auf Kern

Bewerte Tasks $T(V_T, E_T)$
Bestimme den Favorite Neighbour jedes Tasks
Bewerte Kerne $C(V_C, E_C)$

Wähle Task t mit höchster Bewertung
Platziere Task t auf Kern c mit passender Bewertung

Platziere übrige Tasks t' in Platzierungsliste P als Tupel (t', t'_{FAV})

while P *nicht leer* **do**
  if P *enthält Element* (t', t'_{FAV}) *mit* t'_{FAV} *schon platziert* **then**
    Platziere t' mit minimalem Abstand zu t'_{FAV} auf einen Kern mit passender Bewertung
  else
    Wähle unplatzierten Nachbar t' von schon platzierter Task t_P mit höchster Bewertung
    Platziere t' auf einem Kern mit minimalem Abstand zu t_P und passender Bewertung
  end
  Entferne (t', t'_{FAV}) aus P
end

Algorithmus 3 : *Favorite Neighbour Algorithmus* (pseudo code)

ten Tasks werden nun als Tupel (t', t'_{FAV}) gemeinsam mit ihren favorisierten Nachbartasks in die Platzierungsliste eingefügt. So lange die Platzierungsliste noch Tupel enthält werden nun Elemente daraus platziert. Hierbei können zwei Fälle unterschieden werden. Im ersten, günstigeren Fall ist bereits eine Task platziert, die der favorisierte Nachbar einer noch nicht platzierten Task ist. Ist dies der Fall, so wird dieser Nachbar im Kern-Graphen so nah wie möglich bei der bereits existierenden Task platziert.

Im zweiten Fall wird, ähnlich zum *Konnektivitätssensitiven Algorithmus*, eine unplatzierte Task nahe zu einer im Task-Graphen benachbarten und bereits platzierten Task gesetzt. Hierbei wird darauf geachtet, dass aus der Menge der unplatzierten Tasks die ausgewählt wird, die am höchsten gewichtet ist. Der Algorithmus terminiert sobald die Platzierungsliste leer ist.

Der *Favorite Neighbour Algorithmus* besitzt eine algorithmische Komplexität von $\mathcal{O}(|V_C|^2 + |V_C||E_C|)$. Für die Aufwandsbestimmung sind die Schleife und ihr Inhalt interessant.

Die Sortierung und Bewertung von Task-Graph und Core-Graph geschieht, wie bei den anderen Algorithmen auch, offline. Wurde bereits ein Task-Graph platziert, muss die Bewertung des Core-Graphen mit einem Aufwand von $\mathcal{O}(|V_C| \log |V_C|)$ neu sortiert werden. Die kann aber bereits vor der Platzierung des neuen Task-Graphen geschehen.

Die Platzierungsliste enthält zu Beginn der Schleife $|V_T| - 1$ Elemente und wird mit jedem Schleifendurchlauf um ein Element kleiner. Die if-Abfrage benötigt $|V_T|$ Schritte,

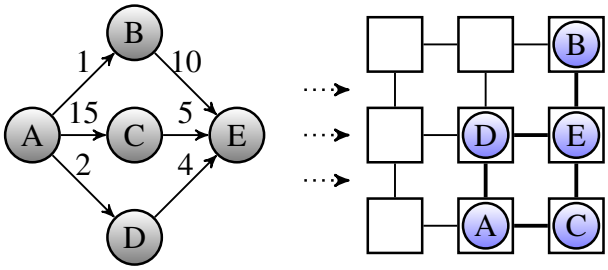


Abbildung 5.3: Ein Beispielergebnis des *Favorite Neighbour Algorithmus*

da zwei Listen durchwandert werden müssen, die gesamt $|V_T|$ Elemente besitzen.

Im if-Fall wird eine Platzierung mit minimalem Abstand vorgenommen. Diese benötigt wie in den vorhergehenden Algorithmen eine Breitensuche mit einer Laufzeit von $\mathcal{O}(|V_C| + |E_C|)$.

Im else-Fall wird ebenfalls eine Platzierung mit minimalem Abstand durchgeführt, doch vorher muss eine zu platzierende Task ausgewählt werden. Dies benötigt wie die if-Bedingung maximal $|V_T|$ Schritte, da sowohl die Liste der platzierten Tasks als auch die der unplatzierten durchlaufen werden muss. Dies erhöht die Laufzeit des else-Falls auf $\mathcal{O}(|V_T| + |V_C| + |E_C|)$.

Für die akkumulierte Obergrenze wird also der else-Fall betrachtet, was zu einer Gesamtlaufzeit von $\mathcal{O}(|V_T|^2 + 2|V_T||V_C| + |V_T||E_C|)$ führt. Im ungünstigsten Fall gilt außerdem $|V_T| = |V_C|$. Dies ändert die Laufzeitabschätzung zu $\mathcal{O}(|V_C|^2 + |V_C||E_C|)$.

Die Behandlung defekter Komponenten des Many-Cores verläuft analog zum *Konnektivitätssensitiven Algorithmus*. Defekte Kerne sind nicht in der Lage, Tasks auszuführen, und sind somit nicht Teil des Core-Graphen. Bei der Bewertung des Kern-Graphen werden defekte oder zu defekten Routern führende Links von der Bewertung des Cores abgezogen:

$$
\begin{aligned}
Bewertung_{Core} &= |N| - |N_{unerreichbar}| \\
N &= \{n \mid (core, n) \in E_C\} \\
N_{unerreichbar} &= \{n \mid (core, n) \in E_C \wedge \\
&\qquad (core\_defekt(n) \vee link\_defekt(core, n) \vee router\_defekt(n))\}
\end{aligned}
$$

Die Funktionen $core\_defekt()$, $link\_defekt()$ und $router\_defekt()$ liefern wie schon zuvor Wahr zurück, falls der angegebene Core, Link oder Router defekt sind.

Die Breitensuche in der Abstandsberechnung der Platzierungsschleife verwendet nur Links und Router, die nicht defekt sind.

In Abbildung 5.3 wird eine beispielhafte Anwendung des *Favorite Neighbour Algorithmus* dargestellt. Durch die initiale Bewertung der Tasks wird C als Task mit der höchsten Gewichtung ermittelt und damit zuerst platziert. Mit nur zwei Nachbarn wird C in ein Ecke mit zwei angrenzenden freien Kernen gesetzt. Alle anderen Tasks werden an die Platzierungsliste angehängt. A besitzt C als favorisierten Nachbarn und wird

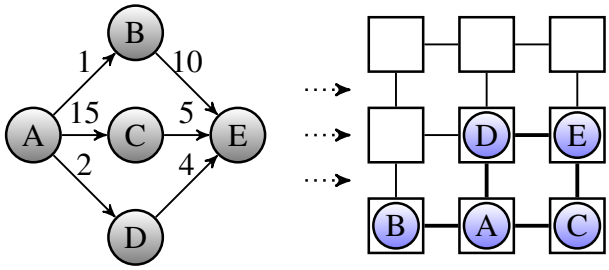


Abbildung 5.4: Vergleichsplatzierung des *Fitting Algorithmus*

daher als nächstes platziert. Anschließend verbleiben keine Tasks in der Platzierungsliste, die schon platzierte Favoriten besitzen. Aufgrund der Nachbarschaftsbeziehungen im ursprünglichen Task-Graphen ist es möglich, jeden der Tasks in der Platzierungsliste als nächstes zu setzen. Die Task mit der höchsten Bewertung ist in diesem Fall E, welches benachbart zu C platziert wird. Nun verbleiben B und D in der Platzierungsliste, die beide E als ihren favorisierten Nachbarn gewählt haben. Beide werden nacheinander benachbart zu E platziert und der Algorithmus terminiert.

Würde für den selben Task-Graphen der *Fitting Algorithmus* ausgeführt, ergäbe sich eine deutlich unvorteilhaftere Platzierung, wie in Abbildung 5.4 zu sehen ist. Zu Beginn platziert auch der *Fitting Algorithmus* die Task mit der höchsten Bewertung. Dies ist ebenfalls C, da die Bewertungsmethodik identisch ist. Für dieses Beispiel wird angenommen, das der Algorithmus C nun ebenfalls in die untere rechte Ecke des Core-Graphen platziert. Die Platzierungsliste ergänzt sich um A und E. Beide sind mit Abstand eins zu C auf einen Kern mit drei Nachbarn zu platzieren. Wird A zuerst behandelt, erweitert sich die Platzierungsliste um B und D mit der Absicht, nahe zu A gesetzt zu werden. Die zwei übrigen, verfügbaren Plätze mit Abstand eins bieten Platz für B und D. Anschließend terminiert der Algorithmus.

Bei der Analyse der beiden Platzierungen in Abbildung 5.3 und 5.4 ist festzustellen, dass die platzierten Tasks bis auf eine Ausnahme an der selben Stelle liegen. Task B wurde mit dem *Fitting Algorithmus* in der unteren linken statt der oberen rechten Ecke platziert. Bei einer Betrachtung der Kommunikationsströme von B zu seinen Kommunikationspartnern A und E kann festgestellt werden, dass B mit einer Intensität von eins zu A und mit einer Intensität von 15 zu E kommuniziert. In Abbildung 5.3 benötigt die Kommunikation zu A drei Hops und die Kommunikation zu E nur einen. In Abbildung 5.4 ist es genau umgekehrt, was das gesamte Kommunikationsaufkommen um 18 erhöht.

Die Δ_{Com} Bewertungen der beiden Platzierungen bestätigen diesen Wert:

$$\textit{Favorite Neighbour Algorithmus} : \Delta_{Com} = 39 - 37 = 2$$
$$\textit{Fitting Algorithmus} : \Delta_{Com} = 57 - 37 = 20$$

Der *Konnektivitätssensitive Algorithmus* besitzt das selbe Problem mit dem Unter-

schied, dass C zu Beginn in der Mitte platziert wird.

Bei der Anwendung des *Favorite Neighbour Algorithmus* bilden sich Ballungsgebiete, in denen viel kommuniziert wird. Durch die bevorzugte Platzierung von Verbindungen im Task-Graph, die hohe Bandbreite benötigen, befinden sich Links mit niedrigem Kommunikationsaufkommen eher in den Randgebieten der Platzierung. Damit gibt es weniger Konflikte mit etwaigen Datenströmen anderer Applikationen, die möglicherweise durch Randgebiete des gerade platzierten Task-Graphen fließen.

5.4 Platzierung mehrerer Task-Graphen

Software, die einen Many-Core-Prozessor vollständig auslasten kann, ist nur in sehr wenigen Anwendungsfällen möglich. Da es aber nicht sinnvoll ist, einen groß dimensionierten Prozessor zu bauen und diesen nicht vollständig nutzen zu können, ist es wichtig zu untersuchen, wie die Algorithmen auf die Situation reagieren, dass mehrere durch Task-Graphen repräsentierte Applikationen gleichzeitig oder nacheinander auf dem Prozessor platziert und ausgeführt werden.

Die bisher vorgestellten Platzierungsalgorithmen können mit einer solchen Situation problemlos umgehen. Da das verwendete Modell es nicht vorsieht, mehrere Tasks gleichzeitig auf einen Kern zu platzieren, dürfen Kerne, die bereits mit einer Task beschäftigt sind, nicht mehr bei weiteren Platzierungen berücksichtigt werden. Zu diesem Zweck können sie einfach innerhalb der Platzierungsalgorithmen als defekt markiert werden, da alle vorgestellten Algorithmen mit defekten Kernen umgehen können. Alternativ können sie aus dem Core-Graph entfernt werden, was aber Probleme bereitet, da die angeschlossenen Router und Interconnect-Verbindungen auch von dem neuen Task-Graphen mit genutzt werden können.

Diese Mehrfachnutzung geteilter Prozessorelemente von unabhängigen Applikationen führt dazu, dass sich diese Applikationen gegenseitig behindern, was wiederum zu reduzierter Performance und einem anderen Zeitverhalten führt. Ist ungestörte Ausführung der einzelnen Applikationen nötig oder erwünscht, ist es sinnvoller, den Algorithmus so zu benutzen, dass er annimmt, der Router eines von einer anderen Applikation belegten Kerns sei defekt. Somit würde die Platzierung nicht auf diesen Router und dessen Verbindungen zurückgreifen.

Für die generelle Übersicht über den Zustand des Many-Core-Prozessors ist es allerdings nicht zielführend, Kerne und Router als defekt zu markieren, wenn diese in Wirklichkeit funktionstüchtig, aber beschäftigt sind. Daher empfiehlt es sich im Modell des Core-Graph für die Mehrfachplatzierung einen zusätzlichen Zustand der Kerne, Router und Interconnect-Verbindungen einzuführen. All diese Elemente können somit *defekt* oder exklusiv *belegt* sein.

Zusätzlich müssen die oben genutzten Hilfsfunktionen $core\_defekt()$, $link\_defekt()$ und $router\_defekt()$ durch solche ersetzt werden, die den aktuellen Platzierungsmodus des Algorithmus in Betracht ziehen. $core\_unbenutzbar()$, $link\_unbenutzbar()$ und $router\_unbenutzbar()$ liefern im exklusiven Modus *Wahr* zurück, wenn das betroffene Element entweder defekt oder exklusiv belegt ist. Im nicht exklusiven Platzierungsmodus wird nur *Wahr* zurückgegeben, wenn das entsprechende Prozessorelement

defekt ist.

Wenn Router und Links von einer Platzierung genutzt werden, müssen sie nicht zwangsläufig exklusiv belegt sein. Cores hingegen sind immer belegt, sobald eine Task auf ihnen platziert wurde, da sie unfähig sind, mehrere Tasks gleichzeitig auszuführen.

Die Unterscheidung, ob eine Platzierung Ressourcen exklusiv belegt oder nicht, führt zu zwei Betriebsarten der Mehrfachplatzierung. Diese können vom Programmautor vorgegeben oder vom Betriebssystem durchgesetzt werden. Wird also der Task-Graph für ein Programm erstellt und in den Programmcode eingebettet, kann zusätzlich die Eigenschaft der exklusiven Belegung der Prozessorelemente vorgegeben werden. Die Platzierungsalgorithmen erfüllen dann die Vorgaben und erlauben keine geteilten Interconnect-Verbindungen und Router.

6 Dynamische Task-Platzierung

Im Gegensatz zur statischen Platzierung, die nur einmal zu Beginn einer Programmausführung abläuft, wird die dynamische Platzierung kontinuierlich während der Laufzeit einer Applikation durchgeführt. Damit sind Anpassungen auf veränderte Situationen und weitere Optimierungen möglich. Eine rein dynamische Platzierung ist nicht möglich, da die dynamischen Verfahren immer von bereits platzierten Tasks ausgehen. Es sind zwei dynamische Verfahren, die auch in Kombination genutzt werden können, denkbar: Anpassung einer bestehenden Platzierung sowie komplette Replatzierung von Applikationen.

6.1 Voraussetzungen

Um überhaupt eine dynamische Platzierung vornehmen zu können, müssen Voraussetzungen erfüllt sein, die eine statische Platzierung nicht benötigt. Der Systemzustand muss überwacht, die bestehenden Platzierungen müssen optimiert und eine Migration von Tasks muss ermöglicht werden.

Überwachung

Vor der Optimierung einer bestehenden Platzierung steht die Aufdeckung eines Optimierungsbedarfs. Hierzu zählen auftretende Fehler in der Hardware, ungünstige Kommunikationswege und Performance-Veränderungen auf dem Chip. Diese geänderten Voraussetzungen können unter anderem durch das Starten von neuen Applikationen verursacht werden, wenn dadurch mehrere Task-Graphen die Kommunikationskanäle gemeinsam benutzen.

Ungünstige Kommunikationswege werden im Allgemeinen durch den statischen Platzierungsalgorithmus erzeugt, der nicht für alle Tasks und ihre Kommunikationspartner eine Platzierung mit optimaler Distanz finden kann. In vielen Fällen ist dieses Problem durch die Struktur des Netzwerks bedingt und kann nicht behoben werden, aber manchmal lässt sich durch Optimierungstechniken eine bessere Platzierung erzeugen. Zusätzlich erhebt der statische Platzierungsalgorithmus nicht den Anspruch, eine optimale Platzierung zu finden. Vielmehr versucht er seine eigene Laufzeit gering zu halten, während er eine überdurchschnittliche Platzierung erzeugt.

Fehler in der Hardware werden durch bewährte Fehlererkennungstechniken gefunden. Auf diese wird in der vorliegenden Arbeit nicht weiter eingegangen. Ist durch den Fehler ein Router oder ein Kern betroffen, der direkt in einem platzierten Task-Graphen enthalten ist, stürzt die Applikation ab. Durch Snapshot-Techniken oder Mehrfachausführung kann dieses Problem gemildert werden, indem in einem solchen

Fehlerfall zu einem vorherigen Snapshot zurückgerollt oder das Ergebnis der Ausführung auf einem anderen Kern verwendet wird.

Ist durch den Fehler nur eine Verbindung betroffen, kann diese in der Regel umgangen werden. Das erledigt ein adaptiver Routing-Algorithmus automatisch, allerdings werden dadurch einige Wege länger. Dies beeinflusst den Kennwert Δ_{COM} negativ. In einem solchen Fall sollten Optimierungsalgorithmen versuchen, die Kommunikationswege durch Migration von Tasks wieder zu verkürzen.

Ein weiterer Indikator für die Notwendigkeit eines Optimierungslaufs kann eine lokale Kommunikationsanalyse sein. Hierzu können auf den Kernen lokale Wartezeiten analysiert werden, in denen die Task auf Nachrichten von anderen Tasks wartet. Steigen diese Wartezeiten stark an oder nehmen sie einen Großteil der Laufzeit einer Task ein, so ist es unter Umständen möglich, die Wartezeiten durch eine veränderte Platzierung zu reduzieren. In vielen Fällen ist allerdings auch das Kommunikationsprofil der Applikation verantwortlich.

Kommuniziert eine Task viel mit einer anderen Task sowohl in der Frequenz als auch in der Größe der Datenpakete, ist es sinnvoll zu versuchen, die Distanz dieser beiden Tasks zu reduzieren. Intensive Kommunikation erzeugt somit eine Art Anziehungskraft zwischen teilnehmenden Tasks. Die Kommunikationsmengen werden zwar bereits bei der statischen Platzierung betrachtet, allerdings ist es denkbar, dass eine realistische Einschätzung über das Kommunikationsvolumen erst zur Laufzeit möglich ist.

Bei allen Arten der Überwachung ist darauf zu achten, dass temporäre Effekte nicht zu stark ins Gewicht fallen. Tritt ein Zustand, welcher eine Anpassung der Platzierung sinnvoll machen würde, nur zeitlich beschränkt auf, muss eine Abwägung zwischen Kosten und Nutzen getroffen werden. Jegliche Anpassung einer Platzierung kostet Zeit und Ressourcen. Es ist daher sinnvoll mit Schwellwerten zu arbeiten, die eine vorschnelle Migrationsanweisung verhindern können. Damit kann auch ein Oszillieren vermieden werden.

Optimierung

Die Optimierung einer Task-Platzierung kann in zwei verschiedene Kategorien eingeteilt werden. Zum einen kann eine ständige Optimierung durchgeführt werden, die während der kompletten Laufzeit einer Applikation im Hintergrund durchgeführt wird und dabei verbesserte Kommunikationsbedingungen durch Anpassungen der Platzierung bewirkt. Zum anderen ist eine ereignisgesteuerte Optimierung möglich, die an definierten Zeitpunkten angestoßen wird. Die Festlegung dieser Zeitpunkte geschieht entweder durch definierte Zeitintervalle oder durch bestimmte Ereignisse.

Ziele der Optimierung sind eine verbesserte Platzierung, die durch eine effizientere Kommunikation charakterisiert wird, oder eine Lastbalancierung, um auf eventuelle Überlastungen vorab zu reagieren.

Bei einer ständigen Hintergrundoptimierung arbeiten die Platzierungsmechanismen, wenn sie nicht gerade einen neuen Task-Graphen platzieren, an der Verbesserung der bestehenden Abbildungen. Der Vorteil dieser Methode ist, dass auch länger andauernde Algorithmen, wie beispielsweise Ameisenalgorithmen oder komplexe Hormonsystem-Algorithmen, zum Einsatz kommen können. Allerdings muss die Möglichkeit bestehen,

einen Optimierungslauf zu jedem Zeitpunkt unterbrechen zu können. Wird eine neue Applikation gestartet, wäre es ungünstig, auf eine Optimierung warten zu müssen.

Neue Informationen, die über den Systemzustand gesammelt werden, fließen zu jeder Zeit in die Optimierung ein und beeinflussen die erzeugten Ergebnisse. Gleichzeitig kann der Optimierungsalgorithmus, wenn er es für nötig hält, Migrationen anstoßen.

Die ereignisgesteuerte Optimierung wird durch bestimmte Ereignisse, wie das Verstreichen einer definierten Zeitspanne, die Entdeckung eines Hardwaredefekts oder des Endes eines laufenden Task-Graphen ausgelöst. Sofern ein auslösendes Ereignis existiert, kann die Optimierungsstrategie auf ein konkretes Ziel angepasst werden. Beispielsweise können Fehler im Interconnect kompensiert werden, da die Fehlerstelle bekannt ist und somit alle Tasks, die über diese Komponente kommunizieren, zielgerichtet neu analysiert werden können.

Ein weiterer Vorteil der ereignisgesteuerten Optimierung ist die Tatsache, dass die Platzierungseinheit nicht immer laufen muss. Ist gerade kein Platzierungslauf im Gange und außerdem kein auslösendes Ereignis aufgetreten, kann sie in einen Energiesparmodus fahren.

Migration

Wurde eine bessere Platzierung einer oder mehrerer Tasks gefunden, muss eine Migration dieser Tasks stattfinden. Während eine Applikation läuft, kann dies zu Komplikationen führen. Daher besteht der erste Schritt darin, den kompletten Task-Graphen, zu dem diese Task gehört, anzuhalten. Hierzu muss gegebenenfalls ein geeigneter Zeitpunkt abgewartet werden, um laufende Operationen zu Ende führen zu können.

Ist der Stop-Zustand erreicht, versendet jede zu migrierende Task ihren kompletten Speicherinhalt auf ihren Ziel-Kern. Dies ist problemlos möglich, da Tasks nur mit Informationen aus ihrem lokalen Speicher arbeiten und dieser bei allen Kernen gleich groß ist. Zur Verringerung des Datenaufkommens können einfache Kompressionsalgorithmen (Laufzeitkodierung, etc.) herangezogen werden. Es bleibt allerdings unnötig, weil das unterliegende Kommunikationsnetz zum Zeitpunkt der Migration nicht genutzt wird, da die betroffene Applikation angehalten wurde.

Zusätzlich zum Speicherinhalt muss auch noch der Rest des Task-Kontextes übertragen werden. Dies umfasst die Registerinhalte und den Program Counter.

Alle Kommunikationspartner der migrierten Tasks müssen anschließend noch über die neue Position in Kenntnis gesetzt werden. Hier reicht es aus, die direkten Nachbarn im Task-Graph zu benachrichtigen, da dies die einzigen Tasks sind, die direkte Kommunikationsbeziehungen zu den migrierten Tasks besitzen.

Die Kerne, auf denen die verschobenen Tasks vorher liefen, werden zu diesem Zeitpunkt als frei markiert. Bei Bedarf kann deren Speicher zusätzlich noch geleert werden. Der Ablauf der Migration einer Task von Kern A nach Kern B ist Abbildung 6.1 zu entnehmen.

Damit ist die Migration abgeschlossen und die Ausführung der Applikation kann fortgesetzt werden.

Dadurch, dass die Migration einen kompletten Task-Graphen anhalten muss, besteht ein großer Einfluss auf die Gesamtperformance. Als Optimierung der Migrationsmetho-

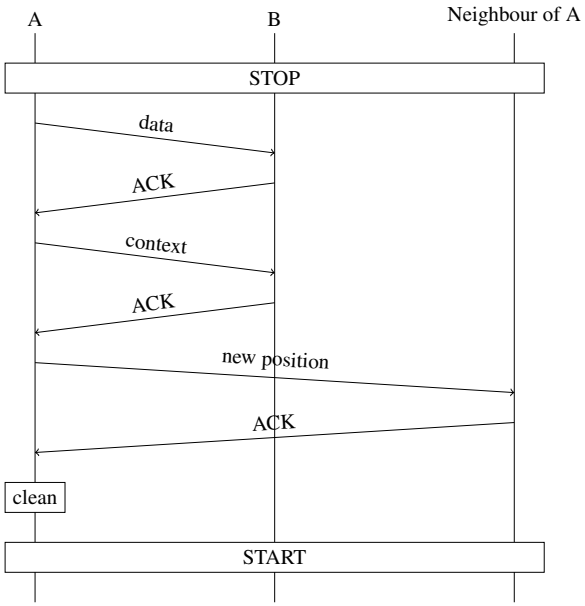


Abbildung 6.1: Migration von A nach B mit angehaltenem Task-Graph

de ist es daher sinnvoll, nicht den kompletten Task-Graphen anzuhalten. Dies erfordert allerdings eine komplexere Vorgehensweise welche in Abbildung 6.2 dargestellt ist.

In einem ersten Schritt wird der Zielkern der Migration reserviert und mit unveränderlichen Daten, wie beispielsweise dem Programmcode, befüllt. Allen Kommunikationspartnern des Quellkerns wird der Beginn und das Ziel der Migration angekündigt. Anschließend sendet der Kern, auf dem die zu migrierende Task liegt, der Reihe nach seinen Programmkontext, seine veränderlichen Daten im internen Speicher und alle Daten, die während der Migration über das Interconnect ankommen, an den Zielkern weiter. Der Quellkern führt währenddessen keinen Programmcode mehr aus, sondern widmet sich voll dem Datenverkehr. Ab dem Zeitpunkt, an dem der Zielkern alle Daten besitzt, kann dieser die Ausführung wieder aufnehmen. Dies signalisiert er dem Quellkern durch eine Nachricht über das Interconnect. Dieser benachrichtigt nun alle seine alten Nachbarn, dass die Migration der Task abgeschlossen ist. Er leitet allerdings weiterhin alle eingehenden Nachrichten an den Zielknoten weiter bis jeder seiner alten Nachbarn den Erhalt der neuen Positionsinformationen bestätigt hat. Zu diesem Zeitpunkt signalisiert der Quellkern dem Zielkern das Ende der Migration und kann sich beenden und seine Ressourcen freigeben.

Eine Voraussetzung für das Funktionieren dieser Migrationsart ist, dass Netzwerkpakete Quellinformationen enthalten, damit der Zielkern der Migration den ursprünglichen Absender einer weitergeleiteten Kommunikation feststellen kann. Zusätzlich werden die Netzwerkpuffer stärker belastet und sollten daher etwas größer dimensioniert werden.

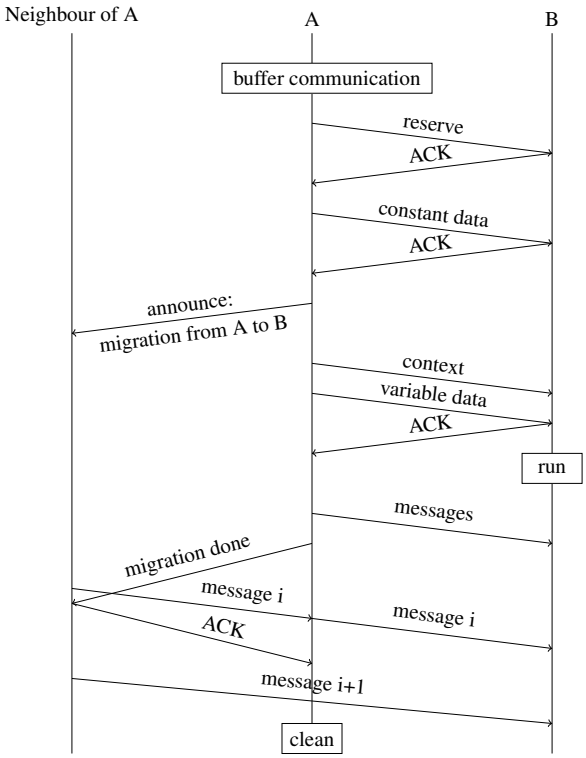


Abbildung 6.2: Migration von A nach B zur Laufzeit

Mit dieser Methode können alle zu migrierenden Tasks jeweils einzeln der Reihe nach migriert werden, ohne das Komplettsystem anhalten zu müssen. Die Migration an sich erstreckt sich allerdings über einen etwas längeren Zeitraum und verursacht währenddessen eine höhere Netzwerklast.

6.2 Anpassung

Die Anpassung der Platzierung einzelner Tasks ermöglicht es, eine bereits existierende Task-Platzierung weiter zu verbessern. Einzelne Tasks werden hierbei auf andere Kerne verschoben, um den globalen Δ_{COM}-Wert zu verbessern und damit eine effizientere Kommunikation zu ermöglichen.

Die einfachste Möglichkeit einer Anpassung besteht darin, jeweils zwei Tasks zu vertauschen, um auszuprobieren, ob ein geringerer Kommunikationsoverhead entsteht. Der NMAP-Algorithmus verwendet diese Methode nach der Initialplatzierung, um sein Ergebnis weiter zu verbessern. Diese Vorgehensweise ist allerdings problematisch, da eventuelle freie Kerne nicht als Migrationsziel für Tasks in Betracht gezogen werden.

Zusätzlich ist davon auszugehen, dass viele Vertauschungen nicht den gewünschten Effekt einer Verbesserung nach sich ziehen, da einfach systematisch durchprobiert wird.

Weniger zufällig ist eine Anpassung, die reale Kommunikation der Tasks in Betracht zieht. Für den Δ_{COM}-Wert ist es im Allgemeinen positiv, wenn Tasks näher zueinander wandern und damit die Hop-Zahl für ihre Kommunikation reduzieren. Ist also ein Kern frei, der näher am Kommunikationspartner liegt, ist es vorteilhaft, eine Task zu migrieren. Allerdings müssen bei jeder Task die Kommunikationen zu allen Nachbartasks und die verwendeten Bandbreiten in Betracht gezogen werden. Es kann durchaus vorkommen, dass eine Migration, welche eine Kommunikationsverbindung positiv beeinflusst bei einer anders gearteten Kommunikation eine Verschlechterung erzielt.

Daher ist vor der Migration eine lokale Veränderung des Δ_{COM}-Wertes zu bestimmen. Zu diesem Zweck wird für jede Verbindung der zu migrierenden Task im Task-Graphen die Hop-Veränderung im Kern-Graphen bestimmt. Diese kann sowohl positiv als auch negativ sein. Anschließend wird jede Hop-Veränderung mit der Gewichtung der betrachteten Verbindung im Task-Graphen multipliziert und die Summe über alle resultierenden Produkte gebildet. Ist diese Summe positiv, so ist durch die Migration eine Verbesserung der Platzierung zu erwarten.

Als Migrationskandidaten werden bei dieser Optimierungsmethode alle Kerne betrachtet, die eine gleiche oder kürzere Entfernung zum Zielkern besitzen. Es ist nicht sinnvoll, Tasks nur so lange in Einzelschritten in eine Richtung zu bewegen bis sich keine Verbesserung des Kommunikationsoverheads mehr zeigt. Eine solche Vorgehensweise hätte zur Folge, dass lokale Minima nicht als solche erkannt werden könnten.

Die Bestimmung der lokalen Veränderung einer Task-Migration wird ein wenig komplizierter, wenn dadurch eine andere Task ihren Kern verlassen muss. Ist die verdrängte Task ein Teil des Task-Graphen, dem auch die zu migrierende Task angehört, lassen sich die lokalen Veränderungen von Δ_{COM} einfach addieren. Wird jedoch eine Task eines anderen Task-Graphen bewegt, muss eine Abwägung getroffen werden. Werden beide Applikationen durch den Positionstausch der betrachteten Tasks bezüglich ihres Kommunikationsoverheads besser oder verschlechtern sich beide, so ist die Entscheidung, die Migration durchzuführen oder zu verwerfen, trivial. In dem Fall, dass ein Task-Graph sich durch den Tausch verbessert, der andere dadurch aber schlechter wird, können die Unterschiede zwar miteinander verglichen werden. Jedoch ist, falls keine Priorität unter den Applikationen auf dem Prozessor existiert, die Entscheidung, ob eine Migration durchzuführen ist, subjektiv und sollte demnach nicht automatisch erfolgen.

Ein weiterer Anwendungsfall für die selektive Anpassung einer Task-Platzierung ist eine räumliche Zusammenfassung in Vorbereitung auf die Platzierung weiterer Task-Graphen. Hierbei wird hauptsächlich auf eine kompakte Form des platzierten Task-Graphen Wert gelegt. Dabei müssen gegebenenfalls Verschlechterungen der Δ_{COM}-Werte in Kauf genommen werden. Der Vorteil einer solchen Anpassung ist jedoch die bessere Ausnutzung des Platzes auf einem Many-Core-Prozessor, was Platzierungsalgorithmen unterstützt und die gemeinsame Nutzung von Leitungen durch mehrere Applikationen reduziert.

6.3 Replatzierung

Die Replatzierung kompletter Task-Graphen ermöglicht die Nutzung der statischen Task-Platzierungs-Algorithmen. Dies ist vorteilhaft, da die Algorithmen in ihrem Kontext gut analysiert sind und daher zielgerichtet eingesetzt werden können. Allerdings ist eine weitere Ausführung mit denselben Ausgangsdaten nicht sinnvoll. Daher müssen der Platzierungsentscheidung veränderte Randbedingungen zugrunde gelegt werden.

Ein veränderter Kommunikationsbedarf zwischen zwei Tasks beeinflusst beispielsweise den Task-Graphen, wohingegen neu ausgefallene Komponenten im Prozessor den Kern-Graphen verändern können. Auch zusätzlich auf dem Many-Core ausgeführte Applikationen beeinflussen die Kommunikation bereits laufender Task-Graphen. In jedem der genannten Fälle kann eine erneute Platzierung für die Performance des Gesamtsystems sinnvoll sein.

Um Unterbrechungen der Programmausführung minimal zu halten, kann der Platzierungsalgorithmus im Hintergrund ablaufen. Als Eingabedaten bekommt er den Gesamtzustand des Systems zu einem gewissen Zeitpunkt. Nach Abschluss der Berechnung wird verglichen, ob der neue Δ_{COM}-Wert eine signifikante Verbesserung gegenüber dem der aktuellen Platzierung darstellt. Nur wenn dies der Fall ist, wird die Migration der betroffenen Tasks angestoßen.

Der Migrationsaufwand ist bei einer kompletten Neuplatzierung wesentlich höher als bei der Anpassung der Platzierung einzelner Tasks, da die Anzahl der zu verschiebenden Tasks im Allgemeinen wesentlich größer ist.

Ein Vorteil bietet allerdings die Möglichkeit, besser auf eine drastisch veränderte Ausgangssituation zu reagieren. Werden beispielsweise durch die Ausführung einer weiteren Applikation auf einmal viele Kerne zusätzlich belegt oder wird durch den Ausfall eines Links oder Routers ein größerer Bereich des Kern-Graphen abgeschnitten, führt die Replatzierung schneller zu einem stabilen und performanten System.

Zusätzlich besteht die Möglichkeit, den verwendeten Platzierungsalgorithmus auf die aktuelle Situation anzupassen. Somit kann ein Algorithmus gewählt werden, der besser für den neuen Füllstand des Core-Graphen oder für die neue Struktur des Task-Graphen geeignet ist.

6.4 Mehrstufige Platzierung

Eine vielversprechende Variante der dynamischen Task-Platzierung ist die Möglichkeit, eine mehrstufige Platzierung einzusetzen. So kann für die Initialplatzierung eines Task-Graphen ein Algorithmus mit einer geringen Komplexität eingesetzt und anschließend durch Platzierungsanpassungen oder Replatzierungen zur Laufzeit der Applikation die Performance verbessert werden.

Durch diese Vorgehensweise wird die Applikation schneller gestartet, entwickelt danach über ihre Laufzeit eine bessere Performance. So wird für die Initialplatzierung eine einfache Metrik, wie beispielsweise die Sequential-Metrik aus Kapitel 7, welche in $\mathcal{O}(n)$ durchlaufen werden kann, eingesetzt. Die Applikation wird anschließend bereits ausgeführt, während im Hintergrund die Platzierungsberechnung eines komplexeren

Algorithmus läuft.

Es ist auch möglich, eine verbesserungswürdige, zufällige Initialplatzierung durch kontinuierliche Anpassung einzelner Task-Positionen zu einer guten Performance zu verhelfen.

Bei einer mehrstufigen Platzierung wird die Laufzeit der Platzierungsalgorithmen eher zweitrangig. So lange eine schnelle Methode für die erste Platzierung gewählt wird, kann die darauf folgende Platzierungsmethode sich Zeit für komplexere Berechnungen und damit für eine bessere Qualität der Zielplatzierung nehmen.

Dynamische Methoden zur Platzierung bilden somit eine gute Ergänzung zu den statischen Methoden der Task-Platzierung. Sie bieten dem Gesamtsystem das Potential zu schnellerer Reaktion, trotz Anforderungen nach guter Performance.

Dynamische Platzierungsmethoden verlangen allerdings auch nach zusätzlicher Komplexität im Many-Core-Prozessor. Die Migration einzelner Tasks muss von der Hardware und der Firmware der Kerne unterstützt werden. Der Grundsatz, möglichst einfache Kerne zu verwenden, wird dadurch abgeschwächt. Daher ist bei der Entwicklung von Many-Core-Prozessoren abzuwägen, ob eine dynamische Task-Platzierung für den gewünschten Anwendungsfall genug Vorteile bietet, um eine komplexere Hardware zu rechtfertigen.

7 Evaluierung

Um festzustellen, wie sich die vorgestellten Algorithmen unter verschiedenen Bedingungen verhalten, wurden diese Situationen auf einem selbst entwickelten Simulator evaluiert. Die Simulationsumgebung führt statische Platzierungen durch und bewertet die Ergebnisse. Es wird keine Simulation der eigentlichen Programmausführung vorgenommen. Vielmehr wird ein Kommunikationsprofil einer Applikation erstellt, welches das reale Verhalten mit ausreichender Genauigkeit abstrahiert.

Als Maß für die Qualität einer Platzierung wird Δ_{COM} verwendet. Wie schon in Abschnitt 5.1 erwähnt beschreibt Δ_{COM} den gewichteten Unterschied der Kommunikationsdistanzen zwischen einem unplatzierten und einem platzierten Task-Graphen. Der Wert der Kommunikationsdistanzen im unplatzierten Task-Graphen (COM_{Orig}) ist die Summe der Gewichte aller Verbindungen. In einem platzierten Task-Graphen berechnet sich die Kommunikationsdistanz (COM_{Placed}) durch die Summe der Produkte der kürzesten Distanzen zwischen kommunizierenden Tasks und der Gewichtung der Kommunikation zwischen diesen Tasks. Hierbei werden nur Distanzen über funktionierende Verbindungen und Router betrachtet. Δ_{COM} errechnet sich einfach als Differenz zwischen COM_{Orig} und COM_{Placed} ($\Delta_{COM} = COM_{Placed} - COM_{Orig}$). Diese Differenz ist immer größer oder gleich Null, da COM_{Orig} den Optimalwert darstellt, weil alle Tasks direkt mit ihren Kommunikationspartnern verbunden sind und eine Platzierung von mehreren Tasks auf einem gemeinsamen Kern im verwendeten Modell nicht möglich ist.

Zusätzlich bestimmt die Evaluierung die maximale Bandbreite, die bei einer Platzierung auf einer einzelnen Interconnect-Verbindung benötigt wird. Dies ist sinnvoll, um die Dimensionierung des verwendeten Interconnects abschätzen zu können.

Wenn nicht anders erwähnt wurden die Platzierungsalgorithmen auf einem Many-Core-Modell mit 64 homogenen Prozessorkernen, die mit einem Maschennetzwerk der Dimension 8x8 verbunden sind, evaluiert. Um das Verhalten möglichst vieler Anwendungsprogramme nachbilden zu können, wurde eine Kombination aus synthetischen Task-Graphen unterschiedlicher Größe und realen Task-Graphen gewählt.

Da die Algorithmen besonderen Wert darauf legen, auch unter Annahme von fehlerhaften Elementen gute Ergebnisse zu liefern, wurden auch Evaluierungen mit einem festgesetzten Prozentsatz von zufällig ausfallenden Komponenten durchgeführt. Die Art der Defekte und Komponenten ist hierbei durch das bereits in 3.1.1 beschriebene Fehlermodell festgelegt.

Um durch den Zufall erzeugte verfälschte Ergebnisse zu vermeiden, ist in allen Evaluierungsergebnissen der Mittelwert aus mehreren Evaluierungsläufen dargestellt.

7.1 Fehlerfreie Hardware

Idealerweise ist die Hardware, auf die die Algorithmen angewendet werden, frei von fehlerhaften Komponenten. Dies kann sicherlich nicht über die komplette Lebenszeit des Many-Core angenommen werden, jedoch müssen die vorgeschlagenen Algorithmen natürlich auch hier gute Ergebnisse liefern.

Analyse des *Konnektivitätssensitiven Algorithmus*

Um den *Konnektivitätssensitiven Algorithmus* einschätzen zu können, wird er am Anfang mit zwei einfachen Platzierungsmetriken, Random und Sequential, verglichen. Die Random-Metrik stellt eine zufällige Platzierung dar, bei der jede Task einem zufälligen, freien, funktionierenden und erreichbaren Kern zugeordnet wird. Sequential platziert Tasks der Reihe nach auf Kerne. Hierbei beginnt die Metrik auf einem 8x8 Mesh mit dem Kern an Position (0,0), dann (0,1) und so fort bis (0,7) erreicht ist. Danach wird mit (1,0) fortgefahren. Bei dieser Platzierung gelten weiterhin die Kriterien, dass nur freie, funktionierende und erreichbare Kerne verwendet werden. Ein Vergleich mit den einfachen Metriken liefert einen Einblick, ob die Weiterentwicklung des Algorithmus sinnvoll ist.

Die Evaluierung mit synthetischen Task-Graphen hilft die Grenzbereiche der Auslastung eines Core-Graphen zu untersuchen. Es ist problemlos möglich, die Anzahl der zu platzierenden Tasks genau auf die Kapazität eines Many-Core-Prozessors anzupassen, und es existiert keine Bindung an die semantischen Zusammenhänge innerhalb einer realen Programmstruktur. Zusätzlich hat jede Verbindung im Task-Graphen die gleiche Gewichtung.

In Diagramm 7.1 sind Ergebnisse für Fork-Join Task-Graphen verschiedener Größen auf einem 8x8 Kern-Graphen ohne Defekte abgebildet. Auf der X-Achse ist die Anzahl der Tasks angezeichnet. Durch die strukturbedingten Einschränkungen der Fork-Join Task-Graphen beginnen die Ergebnisse ab vier Tasks und enden bei 64, da hier die Kapazitätsgrenze des Kern-Graphen erreicht ist. Die Y-Achse stellt die erreichten Durchschnittswerte für Δ_{COM} dar. Ein kleinerer Wert repräsentiert eine bessere Platzierung, da Δ_{COM} und damit der Kommunikationsoverhead geringer ist.

In Diagramm 7.1 ist erkennbar, dass der *Konnektivitätssensitive Algorithmus* nach der Platzierung den geringsten Kommunikationsoverhead produziert. In der dargestellten Auswertung ist der *Konnektivitätssensitive Algorithmus* in jedem Fall besser als die zwei einfachen Vergleichsmetriken. Die Durchschnittswerte der Random-Metrik steigen nahezu linear. Bei der sequentiellen Metrik bewegen sich die Werte bis zu einer Grenze von etwa 40 Tasks in Wellenbewegungen unterhalb der Durchschnittswerte der Random-Metrik, aber deutlich über den Werten des *Konnektivitätssensitiven Algorithmus*. Nach der 40 Task Grenze erzielt die Sequential-Metrik die schlechtesten Ergebnisse. Die Wellen haben eine Frequenz von acht Tasks, was durch die Dimension von 8x8 des untersuchten Kerngraphen zustande kommt. Da die erste platzierte Task bei einem Fork-Join Task-Graphen zu fast allen anderen Tasks verbunden ist, steigt die Kommunikationsdistanz bis das Ende einer Zeile erreicht ist und fällt danach kurzzeitig ab, weil neu platzierte Tasks durch den Zeilenwechsel näher an der Initial-Task

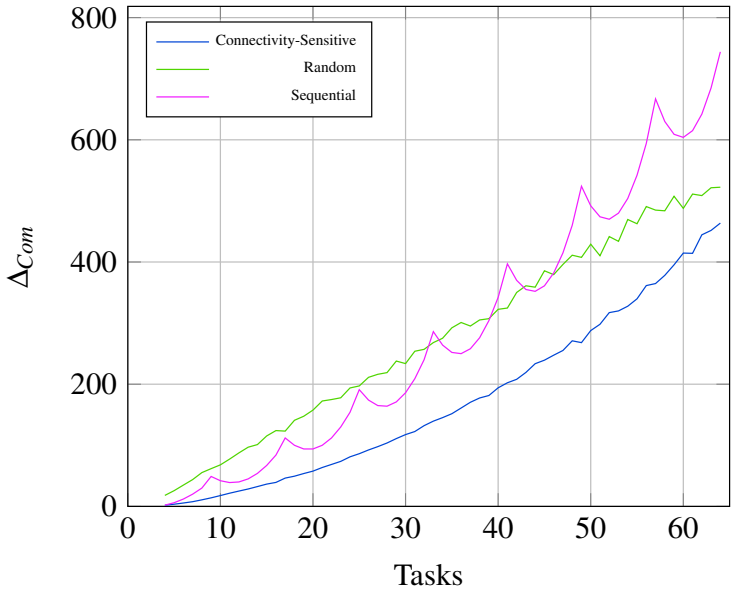


Abbildung 7.1: Fehlerfreier 8x8 Kern-Graph; Fork-Join Task-Graphen

liegen. Die sequentielle Metrik wird bei großen Task-Graphen und damit bei einem stärkeren Füllstand des Kerngraphen immer schlechter und ist deshalb am schlechtesten geeignet, um eine performante Platzierung bei hohem Füllstand zu erreichen.

Die Durchschnittswerte von Δ_{COM} alleine reichen allerdings nicht aus, um eine qualifizierte Aussage über die Fähigkeiten eines Platzierungsalgorithmus zu treffen. Da bei einigen Entscheidungen der Zufall im Spiel ist, wurden in Diagramm 7.2 darüber hinaus die Bereiche angezeichnet, in denen die Δ_{COM} Werte variieren.

Zusätzlich zu der Kurve der Durchschnittswerte sind die erreichten Maximal- und Minimalwerte durch senkrechte Linien angezeichnet. Der Mittelpunkt zwischen Minimum und Maximum ist auf dieser Linie als x markiert. Mit dieser Darstellung kann primär ein Eindruck gewonnen werden, welche Platzierungsstrategie den geringsten Kommunikationsoverhead erzeugt. Desweiteren ist anschaulich zu sehen, welche Variationen die Ergebnisse innerhalb der einzelnen Simulationsläufe durchmachen. Aus Gründen der Vorhersagbarkeit ist es günstiger, wenn diese Variation gegen Null geht. An Stellen, an denen die x-Markierung von der Ergebniskurve abweicht, wird die Tendenz der Ergebnisse sichtbar. Der errechnete Durchschnittswert liegt dort nicht in der Mitte des Intervalls zwischen dem minimalen und dem maximalen gemessenen Wert.

Das Intervall zwischen minimalen und maximalen gemessenen Werten ist beim *Konnektivitätssensitiven Algorithmus* nicht sehr groß, was auf konsistent gute Platzierungen hindeutet.

Wie zu erwarten weist die Random-Metrik sehr große Intervalle der Ergebniswerte auf; die Durchschnittswerte steigen nahezu linear. An einigen Stellen sind die Mini-

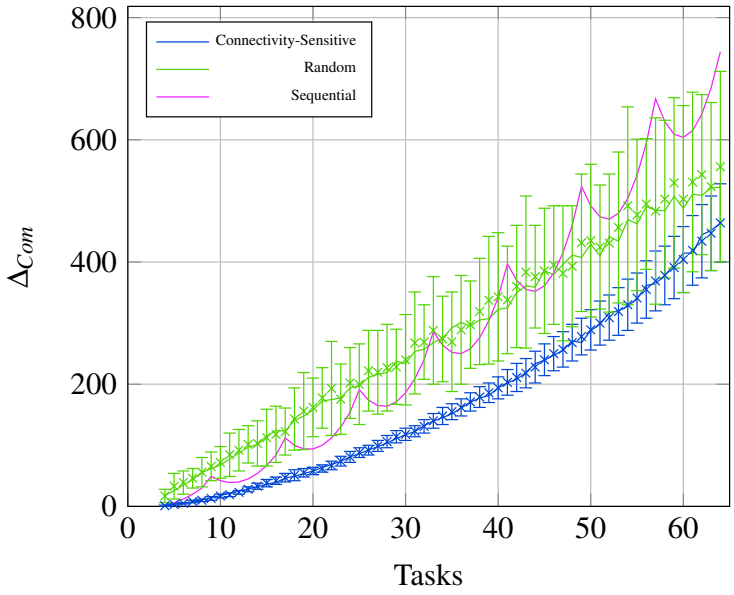
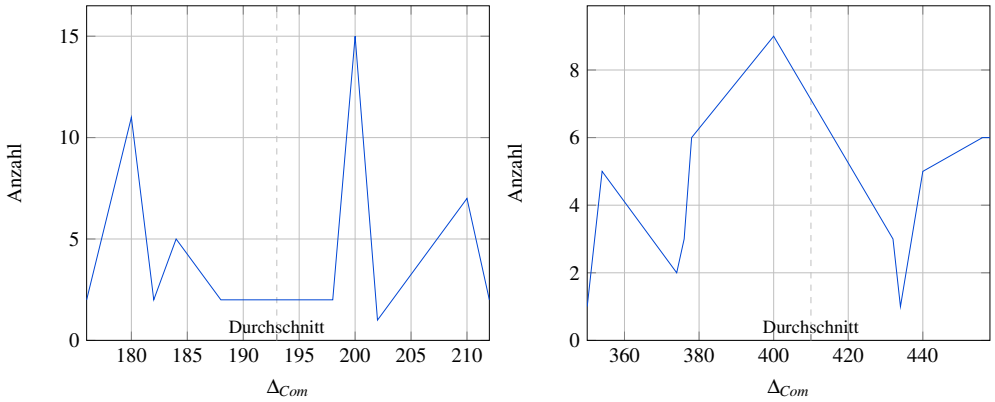


Abbildung 7.2: Fehlerfreier 8x8 Kern-Graph; Fork-Join Task-Graphen

malwerte der Random- besser als die Durchschnittswerte der *Konnektivitätssensitiven* Platzierung, was jedoch nicht zu bedeuten hat, dass für diese Task-Größen die Random-Platzierung zu bevorzugen ist. Da bei einer zufälligen Platzierung schlecht vorausgesagt werden kann, ob der Minimalwert oder sogar der Maximalwert für Δ_{COM} errechnet wird tritt es häufiger auf, dass in diesem Fall die Random-Platzierung schlechter ist als die algorithmische.

Der Graph der sequentiellen Metrik beinhaltet keine Minimum-/Maximum-Intervalle, weil die Metrik keine Zufallselemente besitzt. Da das Intervall mit Größe Null immer direkt auf der Kurve liegen würde, sind die Markierungen aus Gründen der Übersichtlichkeit ausgespart.

Weiterhin ist für die Bewertung der Algorithmen noch interessant, wie die einzelnen Messwerte innerhalb des Minimum-/Maximum-Intervalls verteilt sind. Das Diagramm 7.3 zeigt dies anhand ausgewählter Messungen für den *Konnektivitätssensitiven Algorithmus* und in Diagramm 7.4 sind diese Messwerte für die Random-Metrik zu sehen. In den Diagrammen ist jeweils der Durchschnittswert, also der Kurvenpunkt aus Diagramm 7.2, als gestrichelte senkrechte Linie angezeichnet. Bei beiden betrachteten Algorithmen ist festzustellen, dass die Variation der Δ_{COM}-Werte sich über den kompletten Bereich erstreckt. Nur bei hohem Füllungsgrad des Kern-Graphen (Diag. 7.3b) ist unter Verwendung des *Konnektivitätssensitiven Algorithmus* eine Häufung um den Durchschnittswert festzustellen. Die Random-Metrik weist kaum eine Häufung auf, was durch die Zufälligkeit begründbar ist. Wird die Iterationsanzahl erhöht, so ist weiterhin ein ähnliches Bild zu beobachten. Daher ist die gewählte Anzahl der Iterationen



(a) 40 Tasks

(b) 60 Tasks

Abbildung 7.3: Häufigkeit der Messergebnisse (*Konnektivitätssensitiver Algorithmus*)

ausreichend. Bei den folgenden Analysen wurden jeweils 50 Iterationen durchgeführt, die Ergebnisse aber zusätzlich mit einem Simulationslauf mit 200 Iterationen verifiziert (siehe Anhang 9.10 und 9.11). Wenn nicht anders erwähnt, ergaben beide Läufe vergleichbare Resultate.

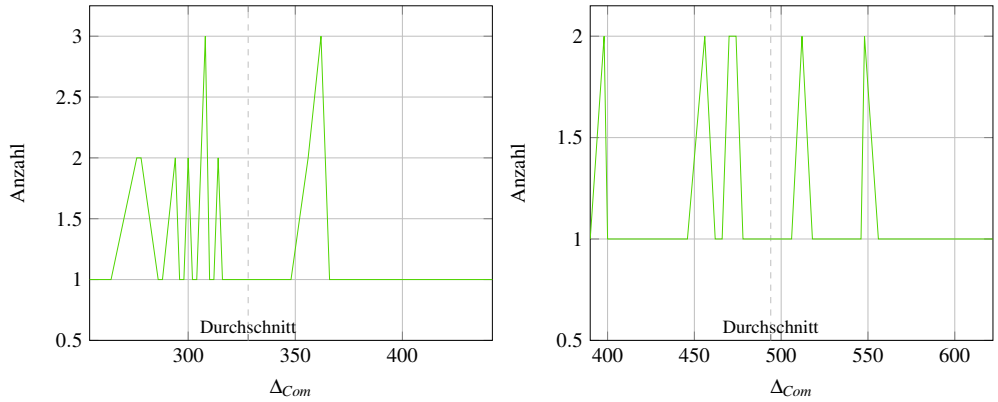
Die bisher analysierten Fork-Join Task-Graphen sind synthetische Lasten, die eine reale Applikation nur zu einem gewissen Grad abbilden können. Aus diesem Grund wurden auch Traces und Modelle von existierenden Programmen als Task-Graph dargestellt und evaluiert.

Zunächst wird ein Task-Graph eines VOP-Decoders (siehe Abbildung 3.13) betrachtet. Er besteht aus 19 Tasks, die mit gewichteten Kanten kommunizieren. Da die Gewichtung in die Δ_{COM} Berechnung mit eingeht, werden hier absolut betrachtet wesentlich größere Werte für das durchschnittliche Δ_{COM} erreicht. Die Ergebnisse sind in Tabelle 7.1 zu sehen. Weil die analysierte Applikation eine feste Größe hat, konnten keine Variationen bezüglich der Größe des Task-Graphen vorgenommen werden.

Algorithmus	**ØΔ_{COM}**	**min(Δ_{COM})**	**max(Δ_{COM})**
Sequential	46.088.192		
Random	134.129.418	85.393.408	182.812.672
Connectivity-Sensitive	1.961.656	1.564.672	5.095.424

Tabelle 7.1: VOP-Task-Graph; Fehlerfreier 8x8 Kern-Graph

Bei Verwendung der Random-Metrik sind sehr starke Schwankungen zu beobachten.



(a) 40 Tasks (b) 60 Tasks

Abbildung 7.4: Häufigkeit der Messergebnisse (Random-Metrik)

Die sequentielle Platzierung liefert durch ihren vollständigen Determinismus immer die selben Ergebnisse. Es ist offensichtlich, dass der *Konnektivitätssensitive Algorithmus* die beiden einfachen Metriken um Längen übertrifft. Die starken Schwankungen zwischen Minimum und Maximum sind dadurch zu erklären, dass die ungünstig platzierte Tasks durch die hohen Kantengewichte einen viel stärkeren negativen Einfluss besitzen. Der VOP-Task-Graph besitzt ein durchschnittliches Kantengewicht von 1.394.874, wodurch die Differenz zwischen Minimum und Maximum beim *Konnektivitätssensitiven Algorithmus* circa zweieinhalb zusätzlichen Hops entspricht. Daher ist das Ergebnis des *Konnektivitätssensitiven Algorithmus* immer noch als gut zu bewerten.

Allerdings bleiben bei einem Core-Graphen der Größe 8x8 nach der Platzierung eines Task-Graphen mit nur 19 Tasks viele Kerne ungenutzt. Außerdem ist es in der Realität des General-Purpose-Computing recht ungewöhnlich, nur eine Applikation auszuführen. Daher wurde die Evaluierung ebenfalls mit zwei Applikationspaketen durchgeführt.

Das erste besteht aus einem MPEG4-Decoder (Abb. 9.36), einer Netzwerkanwendung (Abb. 3.8), einer Telekommunikationsanwendung (Abb. 3.7), einer Büro-Automatisierungsanwendung (Abb. 3.11), einer Consumer-Anwendung zur Bildbearbeitung (Abb. 3.12) und zwei Automobilanwendungen (Abb. 3.9 und Abb. 3.10). Insgesamt umfassen die sieben Task-Graphen 52 Tasks. Das zweite Applikationspaket beinhaltet nur zwei Task-Graphen. Zum einen den bereits bekannten VOP-Decoder (Abb. 3.13) und zum anderen eine umfangreiche Multimedia-Applikation (Abb. 9.37). Gemeinsam umfassen diese Task-Graphen 50 Tasks.

Mit diesen Paketen ist es außerdem möglich zu überprüfen, wie gut die Algorithmen mehrere Task-Graphen zur gleichzeitigen Ausführung nacheinander auf einem Kern-Graphen platzieren.

Auch bei einem gut gefüllten Kern-Graphen ist der *Konnektivitätssensitive Algo-*

Algorithmus	**ØΔ_{COM}**	**min(Δ_{COM})**	**max(Δ_{COM})**
Sequential	311.073.200		
Random	400.020.153	256.990.208	549.563.992
Connectivity-Sensitive	47.723.231	31.972.448	89.765.288

Tabelle 7.2: Applikationspaket 1; Fehlerfreier 8x8 Kern-Graph

rithmus sowohl Sequential als auch Random weit überlegen. Die relativ große Spanne zwischen Minimum und Maximum ist genau so wie beim VOP-Task-Graphen zu begründen. Trotz allem ist der durchschnittliche Δ_{COM}-Wert gegenüber dem des VOP-Decoders gestiegen. Dies war allerdings anzunehmen, da wesentlich mehr Tasks platziert werden und später platzierten Applikationen nur ein stark eingeschränkter Spielraum für die Platzierung zur Verfügung steht.

Des Weiteren ist die Sequential-Metrik immer noch besser als eine durchschnittliche Random-Platzierung, jedoch ist der verhältnismäßige Unterschied lange nicht mehr so groß.

Algorithmus	**ØΔ_{COM}**	**min(Δ_{COM})**	**max(Δ_{COM})**
Sequential	1.024.807.936		
Random	2.513.174.773	1.881.850.880	3.619.514.368
Connectivity-Sensitive	28.782.284	3.098.624	80.718.848

Tabelle 7.3: Applikationspaket 2; Fehlerfreier 8x8 Kern-Graph

Bei Applikationspaket 2 ist der Vorsprung des *Konnektivitätssensitiven Algorithmus* noch einmal wesentlich deutlicher. Dies liegt daran, dass nur zwei große Task-Graphen platziert werden. Solange die Platzierung des ersten bereits viel Lokalität ausnutzt und dadurch eine kompakte Platzierung erzeugt bleibt noch eine genügend große Menge von zusammenhängenden Kernen übrig, um einen zweite Applikation zu platzieren. Weder die Sequential- noch die Random-Metrik können dies garantieren.

Analyse der Erweiterungen des Algorithmus

Gegen die einfachen Metriken kann sich der *Konnektivitätssensitive Algorithmus* auf fehlerfreier Hardware gut behaupten und rechtfertigt den zusätzlichen Rechenaufwand, der für die Ausführung des Algorithmus anfällt. Im nächsten Schritt werden die Ergebnisse der modifizierten Algorithmen *Fitting* und *Favorite Neighbour* mit denen des

ursprünglichen Algorithmus verglichen. Zusätzlich wird der NMAP-Algorithmus aus der Literatur herangezogen, um einen Vergleich mit dem State-of-the-Art herzustellen. Abbildung 7.5 zeigt die Ergebnisse der Simulation. Eine Variante des Diagramms mit eingezeichneten Minimum-/Maximum-Abständen ist im Anhang als Abbildung 9.3 zu finden. Die Diagramme besitzen dieselben Eigenschaften wie schon für Abbildung 7.2 beschrieben.

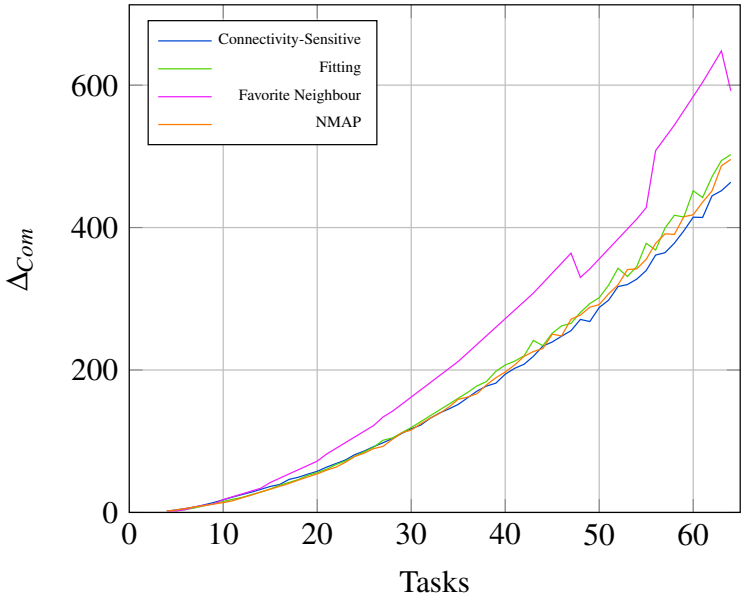


Abbildung 7.5: Fehlerfreier 8x8 Kern-Graph; Fork-Join Task-Graphen

Der *Fitting Algorithmus* bewegt sich sehr nahe an den Werten des *Konnektivitäts-sensitiven*, ist aber bei mehr als 30 Tasks leicht schlechter. Auch das Intervall zwischen maximalem und minimalem Wert ist größer. Der schlechtere Durchschnittswert für Δ_{COM} ist damit zu begründen, dass der *Fitting Algorithmus* durch die Beschränkung der Anzahl der Nachbarn auf die exakt benötigte Menge eine ungünstigere Startposition einnimmt. So ist die Platzierung der Knoten anfangs zwar besser, lässt aber nicht so viele Platzierungsmöglichkeiten für zukünftige Knoten offen. Allerdings geht der *Fitting Algorithmus* dadurch vorsichtiger mit den Ressourcen des Kern-Graphen um, was eine spätere Platzierung zusätzlicher Task-Graphen erleichtern kann. Die größeren Intervalle zwischen Minimal- und Maximalwert der Ergebnisse des *Fitting Algorithmus* sind darin begründet, dass sich der Task-Graph eher am Rand des Kern-Graphen ausbreitet, weil hier Kerne zu finden sind, die weniger als vier Nachbarn besitzen und damit laut Algorithmus besser dazu geeignet sind, Tasks mit zwei Nachbarn aufzunehmen. Dies ist bei Fork-Join Task-Graphen besonders relevant, da die meisten Tasks genau zwei Nachbarn besitzen.

Die Kurve des *Favorite Neighbour Algorithmus* weist, wie auch schon die der sequen-

tiellen Metrik, keine Variationen zwischen Minimum- und Maximumwerten für Δ_{COM} auf. Die Werte bewegen sich ab der 20 Task Grenze oberhalb der Maximumwerte des *Fitting Algorithmus*. Damit ist der *Favorite Neighbour Algorithmus* in diesem Vergleich der schlechteste Kandidat.

Zusätzlich gibt es noch zwei interessante Punkte bei den Task-Graph-Größen 48 und 64. Hier gibt es jeweils einen starken Abfall des Δ_{COM}-Werts. Um zu überprüfen, ob dies auf einem Effekt der Dimensionen des Kern-Graphen beruht, wurde die selbe Simulation mit einem Kern-Graphen der Größe 10x10 durchgeführt (siehe Abbildung 9.1 im Anhang). Dort ist zu erkennen, dass die starken Schwankungen nicht von der Dimension des Kern-Graphen abhängen; es ist also der Struktur des Task-Graphen zuzuschreiben. Wie bereits in Kapitel 3 beschrieben, besitzt der Fork-Join Task-Graph einen Quellknoten, der mit einer bestimmten Anzahl Arbeitertasks kommuniziert (fork). Diese kommunizieren wiederum mit einem Zielknoten (join). Durch die Verwendung eines kombinierten Quell- und Zielknoten sind die Schwankungen der *Favorite Neighbour* Platzierung gänzlich verschwunden (siehe Abbildung 9.2 im Anhang). Es ist außerdem noch anzumerken, dass der *Favorite Neighbour Algorithmus* bei ungewichteten Task-Graphen, wie sie in dieser Auswertung verwendet wurden, nicht sein volles Potential ausspielen kann. Unter der impliziten Annahme, dass jede Verbindung mit eins gewichtet wird, lassen sich keine Vorteile durch eine Berücksichtigung stark frequentierter Verbindungen gewinnen.

Der NMAP-Algorithmus bewegt sich auf demselben Niveau wie der *Konnektivitätssensitive* und der *Fitting Algorithmus*. Es ist eine Tendenz zu erkennen, wobei die Ergebnisse von NMAP zwischen denen der beiden anderen Algorithmen liegen. Bei der Betrachtung der Minimum-/Maximum-Intervalle ist zu erkennen, dass die Platzierungen des NMAP-Algorithmus eine wesentlich höhere Variation aufweisen als die der anderen Algorithmen. Für eine Platzierung in einem möglichst engen Δ_{COM} Zielbereich ist NMAP daher nicht so gut geeignet.

Bei der Analyse realer Applikationen stellt sich das Ergebnis ähnlich dar. Die Resultate der Platzierung eines einzelnen VOP-Decoder Task-Graphen sind in Tabelle 7.4 zu sehen.

Algorithmus	**ØΔ_{COM}**	**min(Δ_{COM})**	**max(Δ_{COM})**
Connectivity-Sensitive	1.961.656	1.564.672	5.095.424
Fitting	1.826.816		
Favorite Neighbour	4.571.136		
NMAP	2.332.590	1.384.448	8.380.416

Tabelle 7.4: VOP-Task-Graph; Fehlerfreier 8x8 Kern-Graph

Bei der VOP-Applikation erzielen der *Konnektivitätssensitive Algorithmus*, der *Fit-*

ting Algorithmus und der NMAP-Algorithmus recht ähnliche Durchschnittsergebnisse. Der *Favorite Neighbour Algorithmus* ist wesentlich schlechter. Wie schon bei den synthetischen Task-Graphen kann beim *Favorite Neighbour Algorithmus* keine Variation zwischen minimal und maximal erreichten Werten für Δ_{COM} unterschieden werden. Außerdem liefert auch der *Fitting Algorithmus* keine unterschiedlichen Werte in den verschiedenen Iterationen. Dies ist darin begründet, dass der einzige Zufallsfaktor darin besteht, welche Task als nächster Platzierungskandidat gewählt wird. Da diese Auswahl, für den Fall, dass mehrere Kandidaten mit gleich guter Bewertung zur Asuwahl stehen allerdings nur durch Zufall bestimmt wird, und die Bewertungen der Tasks im platzierten Task-Graphen alle sehr unterschiedlich sind, tritt im fehlerfreien Kern-Graphen keine Variation auf.

Im Durchschnitt liefert der *Fitting Algorithmus* somit konstant das beste Ergebnis, dicht gefolgt vom *Konnektivitätssensitiven Algorithmus*, der in einigen Fällen, sogar noch bessere Platzierungen erzeugen kann. Der Unterschied zwischen diesen beiden Platzierungsverfahren lässt sich bei Betrachtung der Kantengewichte vernachlässigen. Ebenfalls auf einem ähnlichen Niveau liegen die Platzierungen des NMAP-Algorithmus, der allerdings wie schon bei der Analyse der synthetischen Fork-Join Task-Graphen eine wesentlich höhere Spanne zwischen dem minimal und dem maximal erreichten Wert für Δ_{COM} aufweist.

Der *Favorite Neighbour Algorithmus* ist mit seinem konstanten Δ_{COM}-Wert leicht besser als das schlechteste Ergebnis des *Konnektivitätssensitiven Algorithmus*. Mit den Durchschnittswerten der beiden anderen Algorithmen kann er aber nicht mithalten. Dies ist damit zu begründen, dass der VOP-Task-Graph viele Datenströme mit gleicher Bandbreite beinhaltet. Somit ist der positive Effekt der Priorisierung stark kommunizierender Nachbarn nicht so deutlich zu erkennen.

Im Vergleich mit den bereits betrachteten Random- und Sequential-Metriken sind allerdings alle hier aufgeführten Algorithmen wesentlich besser.

Um einen höheren Füllstand des Kern-Graphen zu untersuchen, wurden die Algorithmen auch zur Platzierung der beiden zuvor definierten Applikationspakete herangezogen. Die Simulationsergebnisse sind in den Tabellen 7.5 und 7.6 abzulesen.

Algorithmus	**ØΔ_{COM}**	**min(Δ_{COM})**	**max(Δ_{COM})**
Connectivity-Sensitive	47.723.231	31.972.448	89.765.288
Fitting	42.682.752		
Favorite Neighbour	103.372.672		
NMAP	26.281.527	19.922.976	57.696.888

Tabelle 7.5: Applikationspaket 1; Fehlerfreier 8x8 Kern-Graph

Bei der Platzierung der sieben Task-Graphen mit insgesamt 52 Tasks ist dieselbe

Rangfolge der Algorithmen zu beobachten, wie bereits bei der Platzierung des einzelnen VOP-Decoder Task-Graphen. Der *Fitting Algorithmus* erzeugt, dicht gefolgt vom *Konnektivitätssensitiven Algorithmus*, den geringsten Kommunikationsoverhead. Mit der *Favorite Neighbour* Platzierung wird mehr als der doppelte Wert für Δ_{COM} erreicht.

Den Vorsprung erreicht der *Fitting Algorithmus*, da er gut für die Platzierung von mehreren Task-Graphen auf einen gemeinsamen Kern-Graph konzipiert ist. Ein Durchlauf des Algorithmus sorgt dafür, dass Task-Graphen in Randbereichen des freien Kern-Graphen platziert werden. Somit steht mehr zusammenhängender Platz für die kommenden Task-Graphen zur Verfügung. Der *Konnektivitätssensitive Algorithmus* kann trotzdem recht gute Ergebnisse erreichen, da einige der sieben platzierten Task-Graphen wenig komplex und damit nicht schwer zu platzieren sind. NMAP ist besonders gut geeignet, um das Applikationspaket 1 zu platzieren, was aber durch eine höhere Laufzeit des Algorithmus erkauft wird. Für kleine, unkomplizierte Task-Graphen liefert der NMAP-Algorithmus sehr gute Ergebnisse, was das gute Abschneiden in dieser Tabelle erklärt. Obwohl der *Favorite Neighbour Algorithmus* eine bessere Platzierung als die Random- oder die Sequential-Metrik erzeugt, kann er beim Applikationspaket 1 nicht mit den anderen beiden Algorithmen mithalten. Er erzeugt einen mehr als doppelt so großen Wert für Δ_{COM} wie der *Konnektivitätssensitive Algorithmus*. Dies spricht gegen die Eignung des *Favorite Neighbour Algorithmus*, um viele unterschiedliche Task-Graphen auf einem Kern-Graph zu platzieren. Die Favorisierung einzelner Task-Nachbarn erzeugt eine weniger kompakte Platzierung, was jeden nachfolgenden Task-Graphen behindert.

Die Ergebnisse zeigen allerdings für das Applikationspaket 2 ein anderes Bild. Entgegen der Erfahrungen aus den vorhergehenden Auswertungen ist hier der *Fitting Algorithmus* an die letzte Stelle zu setzen. Der Grund besteht darin, dass zwei umfangreiche Task-Graphen platziert werden. In dieser Konstellation kann der *Konnektivitätssensitive Algorithmus* im Durchschnitt eine bessere Platzierung erreichen, da er möglichst einen überdimensionierten Kern auswählt, um Raum für Verbesserungen zu lassen.

Algorithmus	**ØΔ_{COM}**	**min(Δ_{COM})**	**max(Δ_{COM})**
Connectivity-Sensitive	28.782.284	3.098.624	80.718.848
Fitting	87.665.664		
Favorite Neighbour	73.089.024		
NMAP	36.555.345	3.147.776	128.797.696

Tabelle 7.6: Applikationspaket 2; Fehlerfreier 8x8 Kern-Graph

Bei der Platzierung des ersten Task-Graphen (VOP-Decoder) liegen der *Fitting* und der *Konnektivitätssensitive Algorithmus* noch beinahe gleichauf (siehe Tabelle 7.4).

Der zweite Task-Graph ist danach aber mit dem *Konnektivitätssensitiven Algorithmus* wesentlich besser zu platzieren. Tabelle 9.1 im Anhang zeigt die Ergebnisse der einzelnen Platzierung des zweiten Task-Graphen des Applikationspakets 2. Hier wird offensichtlich, dass die Multimedia-Applikation mit dem *Fitting Algorithmus* schlecht zu platzieren ist. Dies liegt daran, dass die Multimedia-Applikation überwiegend aus Tasks mit zwei Nachbarn besteht. Dadurch werden solche Tasks bevorzugt am Rand des Kern-Graphen platziert. Muss nun aber eine Task mit mehreren Nachbarn platziert werden, ist der Abstand zu den übrigen bereits platzierten Tasks im Schnitt wesentlich höher.

Die *Favorite Neighbour Platzierung* lässt sich gut auf die Multimedia-Applikation anwenden, wodurch sie in der Gesamtwertung vom Applikationspaket 2 auch leicht besser als der *Fitting Algorithmus* abschneidet.

Der NMAP-Algorithmus zeigt, dass er im Schnitt sehr wenig durch die gleichzeitige Abbildung der beiden Task-Graphen verliert. Sowohl bei der einzelnen Platzierung des VOP-Decoders als auch bei der einzelnen Platzierung der Multimedia-Applikation liegt er über dem Durchschnittswert des *Konnektivitätssensitiven Algorithmus*. Bei gleichzeitiger Platzierung steigt der durchschnittliche Δ_{COM}-Wert allerdings nicht sehr stark. Es bleibt trotzdem zu beachten, dass das Intervall zwischen Minimum und Maximum sehr groß ist.

Weiterhin sind die Platzierungen aller hier betrachteten Algorithmen effizienter als die der Random- oder Sequential-Platzierung.

7.2 Fehlerbehaftete Hardware

Bisher wurden Task-Graphen nur auf fehlerfrei funktionierende Kern-Graphen platziert. Eine wichtige Anforderung bei der Entwicklung der Algorithmen war allerdings eine gute Platzierung unter Berücksichtigung von fehlerhaften Elementen im Kern-Graphen. Untersuchungen werden mit dem in Kapitel 3.1.1 beschriebenen Fehlermodell durchgeführt. Hierbei wird eine Fehlerquote zugrunde gelegt. Die Fehlerquote gibt den Prozentsatz der Hardwareelemente an, die ausgefallen sind. Welche Elemente konkret ausfallen, wird zufällig bestimmt. Es wird auch kontrolliert, ob komplette Bereiche des Kern-Graphen durch eine Konstellation von Defekten abgetrennt werden. Abgeschnittene Kerne, Router und Leitungen werden anschließend nicht mehr für die Platzierung berücksichtigt. Um eine Einordnung der erreichbaren Performance zu bekommen, steht zu Beginn wieder der Vergleich des *Konnektivitätssensitiven Algorithmus* mit den einfachen Metriken Random und Sequential. Anschließend werden die Ergebnisse des *Fitting Algorithmus* und des *Favorite Neighbour Algorithmus* mit denen des *Konnektivitätssensitiven* verglichen.

7.2.1 Fehlerhafte Router

Ist ein Router fehlerhaft stehen auch der zugehörige Kern und die anliegenden Leitungen nicht zur Verfügung. Dies stellt einen schwerwiegenden Fehler dar, weil aufgrund dessen auch noch funktionierende Elemente in ihrer Funktion beeinträchtigt werden.

In Abbildung 7.6 sind die Ergebnisse einer Simulation mit einer Fehlerquote von 10% dargestellt.

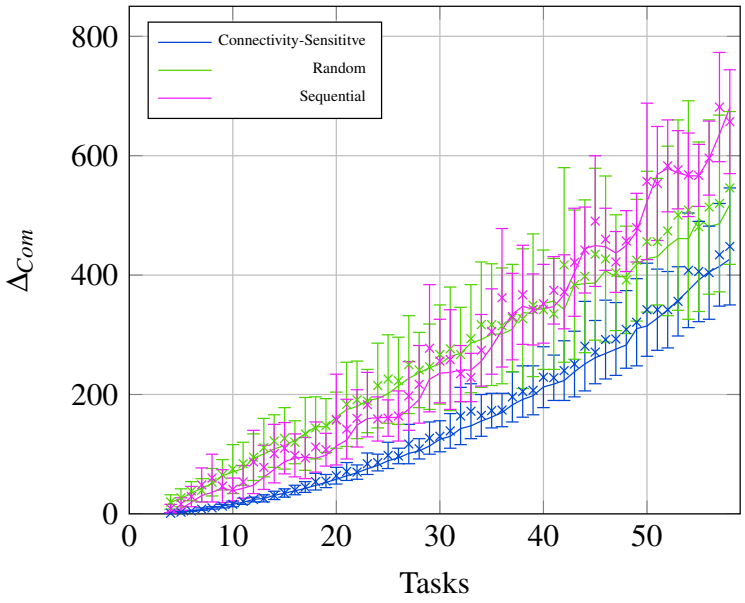


Abbildung 7.6: 8x8 Kern-Graph mit 10% ausgefallenen Routern; Fork-Join Task-Graphen

Die Kurve der sequentiellen Metrik erinnert an das Wellenmuster aus der Evaluierung ohne fehlerhafte Elemente. Die Ergebnisse sind wie gehabt weit von der Performance der Platzierung mit dem *Konnektivitätssensitiven Algorithmus* entfernt. Vereinzelt können große Abweichungen der Minimal- und Maximalwerte beobachtet werden.

Random ist bis zu einer Task-Graph-Größe von 40 die schlechteste Metrik. Bei größeren Applikationen erzeugt die Random-Metrik jedoch bessere Ergebnisse als die sequentielle Platzierung. Wie bei einer zufälligen Platzierung zu erwarten, sind die Minimum-/Maximum-Intervalle recht groß, weshalb die Qualität einer Random-Platzierung stark variieren kann.

Der *Konnektivitätssensitive Algorithmus* kann sich wiederum gegen die einfachen Metriken behaupten und erzeugt konstant bessere Ergebnisse als die beiden Vergleichsalgorithmen. Oft liegt die Kurve des *Konnektivitätssensitiven Algorithmus* sogar unter den Minima der Vergleichsmetriken. Gegenüber der Platzierung im fehlerfreien Kern-Graphen sind die Δ_{COM}-Werte des *Konnektivitätssensitiven Algorithmus* nur leicht angestiegen, wie Abbildung 7.7 zu entnehmen ist. Geringe Fehlerzahlen werden folglich gut ausgeglichen.

Auswertungsergebnisse für Fehlerquoten von 20%, 30% und 40% sind im Anhang in den Abbildungen 9.4, 9.5 und 9.6 zu finden. Der Trend bleibt weiter bestehen. Die Random-Metrik erzeugt ab einer gewissen Größe des Task-Graphen bessere Ergebnisse

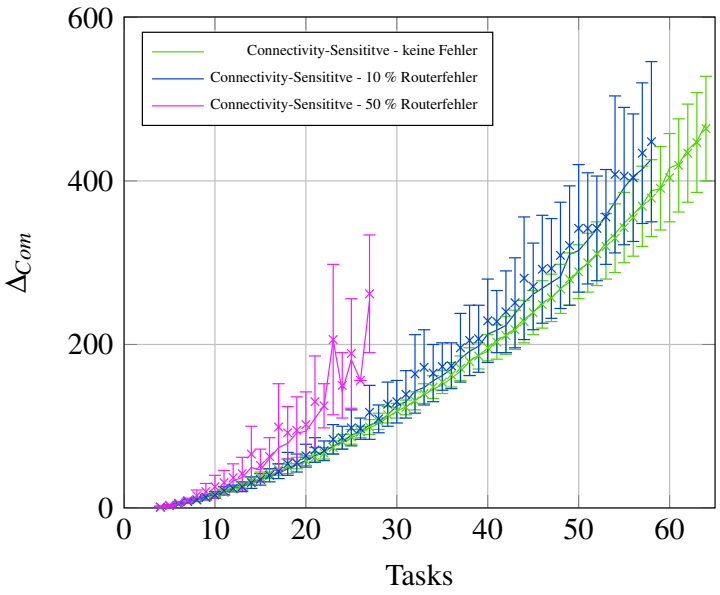


Abbildung 7.7: Vergleich: keine Fehler zu 10% und 50% ausgefallene Router

als eine sequentielle Platzierung. Beide bleiben mit leichten Schwankungen oberhalb der Ergebnisse des *Konnektivitätssensitiven Algorithmus*. Die Intervalle zwischen dem minimalen und dem maximalen Ergebnis des *Konnektivitätssensitiven Algorithmus* steigen allerdings mit wachsender Fehlerquote auch stark an, was natürlich zusätzlich durch die Konstellation der fehlerhaften Elemente beeinflusst wird.

In der Auswertung für eine Fehlerquote von 50% in Abbildung 7.8 kann festgestellt werden, dass die Random-Metrik an einigen Stellen keine nutzbaren Ergebnisse mehr liefert und die sequentielle Metrik ab einer Task-Graph Größe von 25 ebenfalls nicht zuverlässig arbeitet. Der *Konnektivitätssensitive Algorithmus* hingegen produziert weiterhin gültige Platzierungen. Auch wenn die Werte von Δ_{COM} weit über denen für Auswertungen ohne Fehler oder mit weniger Fehlern liegen (siehe Abbildung 7.7) und auch die Intervalle stark wachsen, ist trotzdem eine Platzierung und damit eine Programmausführung möglich. In der Praxis stellt ein derartig stark beschädigter Chip eher ein Risiko dar.

Abbildung 7.9 stellt die Evaluierungsergebnisse des VOP-Task-Graphen in Kern-Graphen mit fehlerhaften Routern dar. Auf der Y-Achse ist wie in den vorherigen Diagrammen der Kommunikationsoverhead Δ_{COM} angetragen. Die X-Achse stellt diesmal die Fehlerquote in Prozent dar. Am schlechtesten schneidet in diesem Fall die Random-Metrik ab. Die Ergebnisse sind weit über denen der sequentiellen Metrik zu finden. Das schlechte Abschneiden und die hohe Variation der Ergebnisse sind darauf zurückzuführen, dass ein recht kleiner Task-Graph platziert wird, dafür jedoch viel Platz zur Verfügung steht. Da Random keinerlei Zusammenhang garantiert, kommen ho-

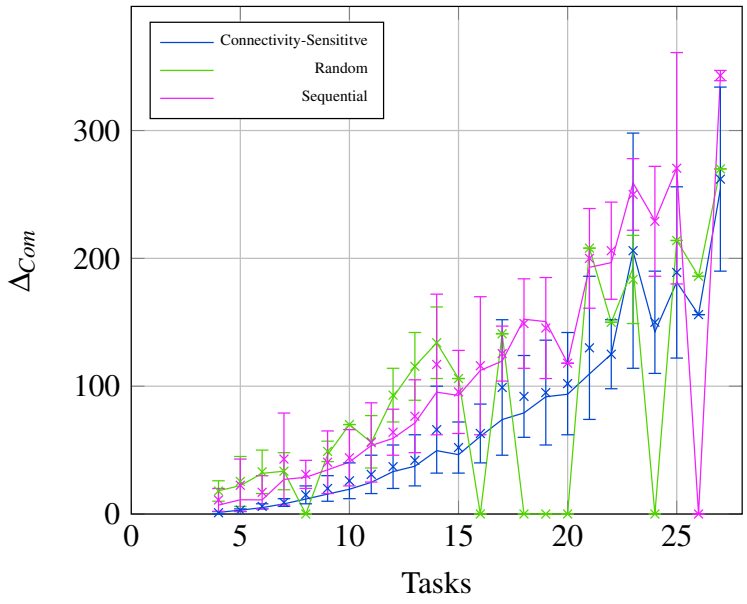


Abbildung 7.8: 8x8 Kern-Graph mit 50% ausgefallenen Routern; Fork-Join Task-Graphen

he Abstände und damit ein erhöhtes Kommunikationsaufkommen zustande. Bei der Sequential-Metrik steigt Δ_{COM} nur moderat an, kann aber nicht die Güte des *Konnektivitätssensitiven Algorithmus* erreichen. Dieser erzeugt klar die besseren Platzierungen mit einem wesentlich kleineren Kommunikationsoverhead. Trotzdem steigt der Wert von Δ_{COM} zwischen dem fehlerfreien Fall und der Variante mit 50% defekten Routern um circa 440%. Was allerdings durch den Maßstab der Abbildung nicht so stark auffällt.

Eine dichtere Belegung des Kern-Graphen wird anhand der beiden bereits definierten Applikationspakete untersucht. Wie in Abbildung 7.10 zu sehen ist, kann weiterhin eine Empfehlung für den *Konnektivitätssensitiven Algorithmus* ausgesprochen werden. Random ist immer noch die schlechteste Wahl. Die Ergebnisse bewegen sich moderat ansteigend oberhalb der anderen Kurven. Eine interessante Entwicklung ist bei der sequentiellen Metrik zu erkennen. Diese erzielt im fehlerfreien Fall merklich schlechtere Ergebnisse.

Die Qualitätssteigerungen der zehnprozentigen Fehlerquote gegenüber dem fehlerfreien Fall liegt darin begründet, dass das Applikationspaket aus vielen kleinen Task-Graphen zusammengesetzt ist, die theoretisch unabhängig voneinander platziert werden können. Für Task-Graphen wird auch getrennt der Δ_{COM}-Wert berechnet. Lässt sich einer dieser Graphen nicht platzieren, was bedeutet, dass die platzierten Tasks nicht miteinander kommunizieren können, kann für diesen Graphen auch kein Δ_{COM} berechnet werden. Die scheinbar bessere Platzierung bedeutet daher, das im Fehlerfall

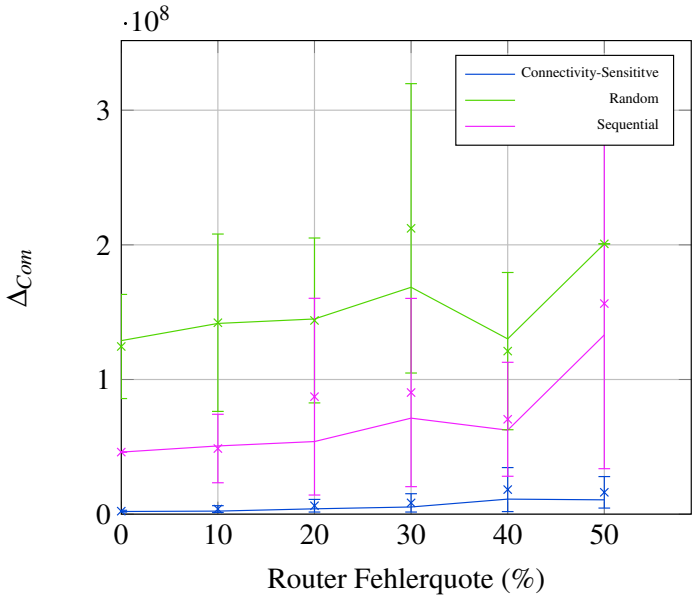


Abbildung 7.9: 8x8 Kern-Graph mit ausgefallenen Routern; VOP-Task-Graph

nur ein Teil der Task-Graphen platziert werden konnten. Dies erklärt auch die hohe Spanne zwischen Minimum und Maximum der Platzierungsmetriken. Selten wird eine Platzierung gefunden, was einen höheren Δ_{COM}-Durchschnitt erzeugt. Meistens wird jedoch kein gültiges Ergebnis erzielt. Die Auswertungsergebnisse haben gezeigt, dass der *Konnektivitätssensitive Algorithmus* bis zu 20% Fehlerquote immer eine Platzierung findet, wenn genug zusammenhängender Platz im Kern-Graph verfügbar ist. Die Δ_{COM} Steigerung von 0% auf 20% Fehlerquote beträgt lediglich 27%.

Jenseits der 20% Fehlerquote ist keine Platzierung mehr möglich, da nicht genügend funktionierende Kerne zur Verfügung stehen, um das komplette Paket zu platzieren.

Abbildung 7.11 beleuchtet die Ergebnisse der Platzierungen von Applikationspaket 2. Es zeigt sich ein bekanntes Bild, in dem die Random-Metrik weit abgeschlagen über der sequentiellen Platzierung rangiert. Diese hält sich wiederum oberhalb des *Konnektivitätssensitiven Algorithmus*. Die riesigen Minimum-/Maximum-Intervalle der Metriken bei einer Fehlerquote von 20% sind darauf zurückzuführen, dass hier nur zwei große Applikationen platziert werden. Ist eine der beiden aus den oben genannten Gründen nicht mehr platzierbar oder wird ungünstig platziert, hat dies einen massiven Einfluss auf die Qualität des Gesamtergebnisses. Bei 20% Fehlerquote ist der Δ_{COM}-Wert des *Konnektivitätssensitiven Algorithmus* um 217% gestiegen.

Für synthetische Fork-Join Applikationen ist der *Konnektivitätssensitive Algorithmus* also auch bei ausgefallenen Routern deutlich besser als die einfachen Metriken Random und Sequential. Seine Überlegenheit zeigt sich auch bei den untersuchten Applikationen und Applikationspaketen.

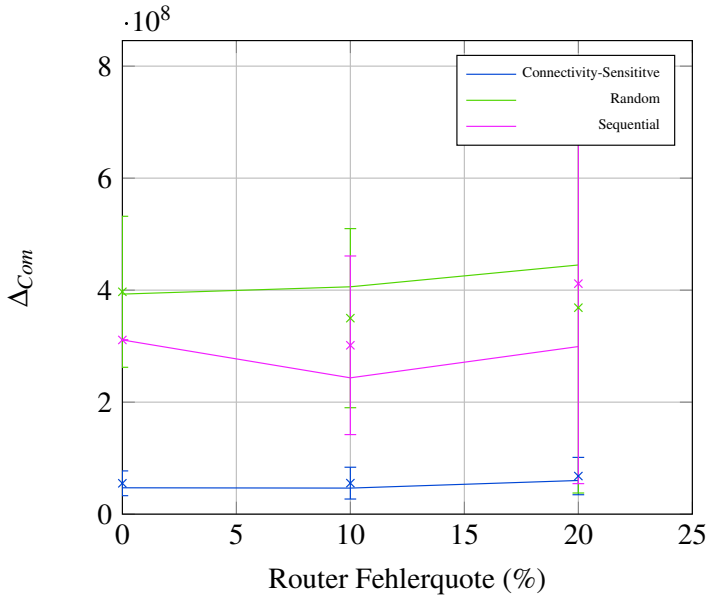


Abbildung 7.10: 8x8 Kern-Graph mit ausgefallenen Routern; Applikationspaket 1

Abbildung 7.12 vergleicht nun den *Konnektivitätssensitiven Algorithmus* mit dem *Fitting Algorithmus*, dem *Favorite Neighbour Algorithmus* und dem NMAP-Algorithmus unter Annahme einer Fehlerquote von 10%. Für die bessere Übersicht ist die Variante mit eingezeichneten Minimum-/Maximum-Abständen im Anhang als Abbildung 9.12 zu finden.

Das schlechteste Ergebnis wird vom *Favorite Neighbour Algorithmus* erzielt. Der durchschnittliche Kommunikationsoverhead liegt weit über dem der anderen Algorithmen. Es ist immer noch ein leichtes Zackenmuster festzustellen, welches schon im fehlerfreien Fall vorhanden war. Die Position der Zacken ist ebenfalls identisch.

Der *Fitting*, der *Konnektivitätssensitive* und der NMAP-Algorithmus weisen sehr ähnliche Ergebnisse auf. Allgemein lässt sich aussagen, dass die NMAP-Platzierungen in der Mitte der anderen beiden Algorithmen liegen. *Fitting* stellt die Schranke nach oben dar und die *Konnektivitätssensitive* Platzierung die Schranke nach unten. Das Gesamtbild der Auswertung ist dem fehlerfreien Fall sehr ähnlich. Es gelten deshalb dieselben Beurteilungen der Platzierungsqualität.

Auswertungsergebnisse für Fehlerquoten von 20%, 30% und 40% sind im Anhang in den Abbildungen 9.7, 9.8 und 9.9 zu finden.

Auch mit höheren Fehlerquoten ist festzustellen, dass der *Favorite Neighbour Algorithmus* schlechtere Ergebnisse als die drei anderen Algorithmen liefert.

Die Qualität der Platzierungen des *Konnektivitätssensitiven Algorithmus* und des *Fitting Algorithmus* bleiben weitestgehend gleichauf und nähern sich sogar bei höheren Fehlerquoten einander an. NMAP kann auch weiterhin mit den beiden besseren

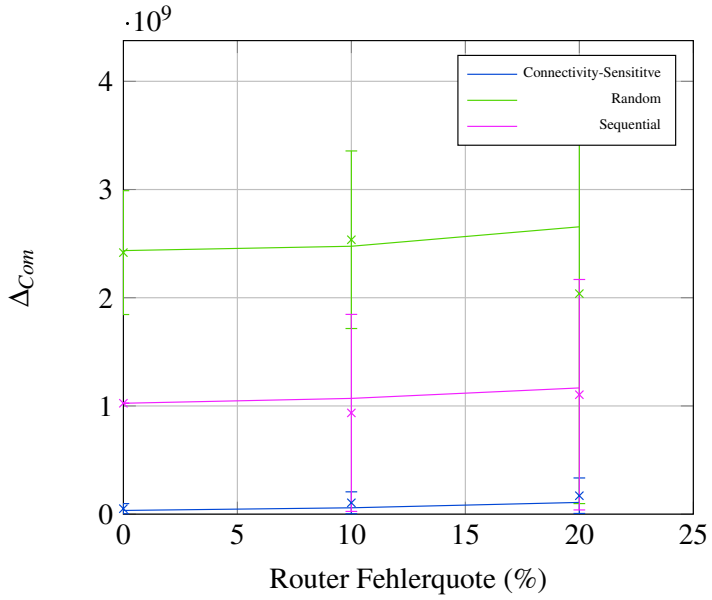


Abbildung 7.11: 8x8 Kern-Graph mit ausgefallenen Routern; Applikationspaket 2

Algorithmen mithalten. Daher sind NMAP, *Fitting* und *Connectivity-Sensitive* für Fork-Join-Applikationen mit fehlerhaften Routern gleichwertig zu betrachten. Die hohe Variation in den Δ_{COM} Intervallen deutet darauf hin, dass stellenweise sehr hohe Werte für Δ_{COM} erzielt werden. Der Durchschnitt liegt aber meist im unteren Bereich des Intervalls. Also erzeugen die Algorithmen gute Platzierungen, können aber schlechte Werte hervorbringen, wenn die Verteilung der fehlerhaften Elemente im Kern-Graphen unvorteilhaft ist. Die unvorteilhafte Verteilung der defekten Router ist auch der Grund für sichtbare Schwankungen bei hohen Fehlerquoten und starker Netzbelegung (beispielsweise in Abbildung 9.9).

In Abbildung 7.13 sind die Ergebnisse für eine Fehlerquote von 50% aufgetragen. Alle Algorithmen finden gültige Platzierungen bis zu einer Task-Graphen-Größe von 29 Tasks. Allerdings sind bei einer derartig hohen Fehlerquote viele theoretisch funktionierende Kerne nicht mehr erreichbar, da alle Kommunikationswege zu ihnen unterbrochen sind.

Das Bild der vorherigen Auswertungen setzt sich fort, indem der *Favorite Neighbour Algorithmus* weiterhin das Schlusslicht bildet, und der *Konnektivitätssensitive Algorithmus* ähnliche Ergebnisse zum *Fitting Algorithmus* aufweist. Beide Algorithmen erzeugen starke Schwankungen ab 21 zu platzierenden Tasks. Hier ist eine gute Platzierung nicht mehr zu garantieren. Die Durchschnittswerte für Δ_{COM} halten sich aber erfreulicherweise in der unteren Hälfte des Minimum-/Maximum-Intervalls.

NMAP hat schon bei kleinen Größen des Task-Graphen Probleme, gültige Platzierungen zu finden. Bei einer derartig hohen Fehlerquote ist dieser Algorithmus offen-

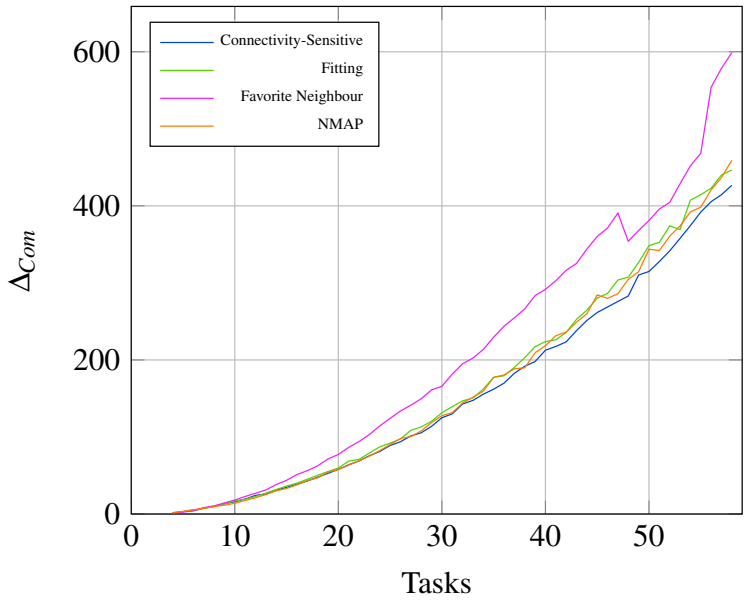


Abbildung 7.12: 8x8 Kern-Graph mit 10% ausgefallenen Routern; Fork-Join Task-Graphen

sichtlich für eine verlässliche Platzierung nicht geeignet.

Abbildung 7.14 zeigt die Platzierungsergebnisse des VOP-Decoder Task-Graphen mit defekten Routern. Bei diesem Diagramm ist auf der X-Achse wieder die prozentuale Fehlerquote angetragen.

Auf den ersten Blick fällt auf, dass der *Favorite Neighbour Algorithmus* auch hier keine optimalen Ergebnisse liefert. Die durchschnittlichen Werte verlaufen konstant über denen der anderen Algorithmen. Seine Maximumwerte bilden zusätzlich einen sehr großen Korridor, was dazu führt, dass die Platzierungen unter Umständen schlechtere und unvorhersagbare Werte produzieren. Wie bereits bei den synthetischen Lasten bewegen sich die übrigen drei Algorithmen ungefähr im selben Bereich. Dabei ist der *Konnektivitätssensitive Algorithmus* meist etwas besser als der *Fitting Algorithmus*. Doch NMAP platziert noch effizienter. Vor allem die Minimum-/Maximum-Intervalle sind bei NMAP deutlich enger. Daher wird eine Platzierung mit genauer vorhersagbarer Qualität erzeugt.

Bei Platzierungsentscheidungen mit einer größeren Anzahl von Tasks, wie beispielsweise bei Applikationspaket 1 (Abbildung 7.15), ist zu beobachten, dass *Favorite Neighbour* und NMAP wieder die schlechteste beziehungsweise beste Position einnehmen.

Beide bewegen sich konstant jeweils über beziehungsweise unter den anderen Algorithmen. Der *Fitting Algorithmus* platziert nur bei fehlerfreien Kern-Graphen besser als der *Konnektivitätssensitive*. Für alle höheren Fehlerquoten produziert er schlechte-

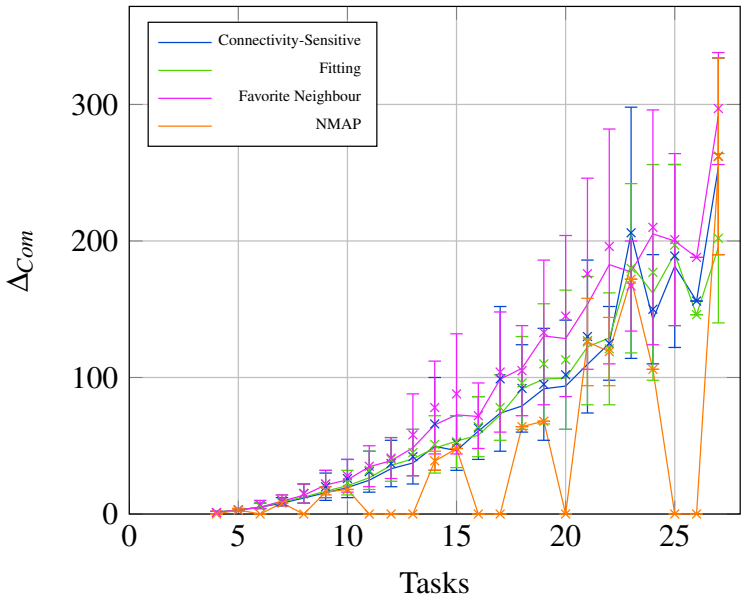


Abbildung 7.13: 8x8 Kern-Graph mit 50% ausgefallenen Routern; Fork-Join Task-Graphen

re Ergebnisse. Die Minimum-/Maximum-Intervalle des *Konnektivitätssensitiven Algorithmus* sind bei dieser Platzierung wesentlich enger, wohingegen die des NMAP-Algorithmus breiter werden. Der *Fitting Algorithmus* weist immer noch eine sehr hohe Varianz auf. Dies wird durch die Platzierung auf Kerne mit einer möglichst zur Task identischen Nachbarzahl verursacht. Existieren fehlerhafte Router, verwendet der Algorithmus so mit höherer Wahrscheinlichkeit Kerne, die in Sackgassen liegen, da diese nur eine begrenzte Nachbaranzahl besitzen. In vielen Fällen ist dies für die weitere Platzierung nachteilig.

Etwas knapper ist die Entscheidung bei Applikationspaket 2, wo nur wenige, aber dafür große Task-Graphen platziert werden. Abbildung 7.16 besteht zum großen Teil aus den Minimum-/Maximum-Intervallen des *Fitting Algorithmus*.

Diese große Variation kommt durch denselben Effekt zustande, der schon für die schlechten Ergebnisse des *Fitting Algorithmus* bei Applikationspaket 1 gesorgt hat. Verstärkt wird er in diesem Fall dadurch, dass durch wenige schlecht platzierte Tasks in einem umfangreichen Task-Graphen sehr schnell die Δ_{COM}-Werte ansteigen. Ist in einem fehlerhaften Kern-Graphen erst einmal einer der beiden großen Task-Graphen platziert, wächst zusätzlich noch die Wahrscheinlichkeit, mit dem zweiten Task-Graphen an einer ungünstigen Stelle zu beginnen. Der Ansatz des *Konnektivitätssensitiven Algorithmus* ist hier offensichtlich besser geeignet. Bei 20%iger Fehlerquote ist er im Schnitt sogar leicht besser als NMAP. Einzig bei einer Fehlerquote von 10% ist NMAP den anderen Platzierungsmethoden weit überlegen.

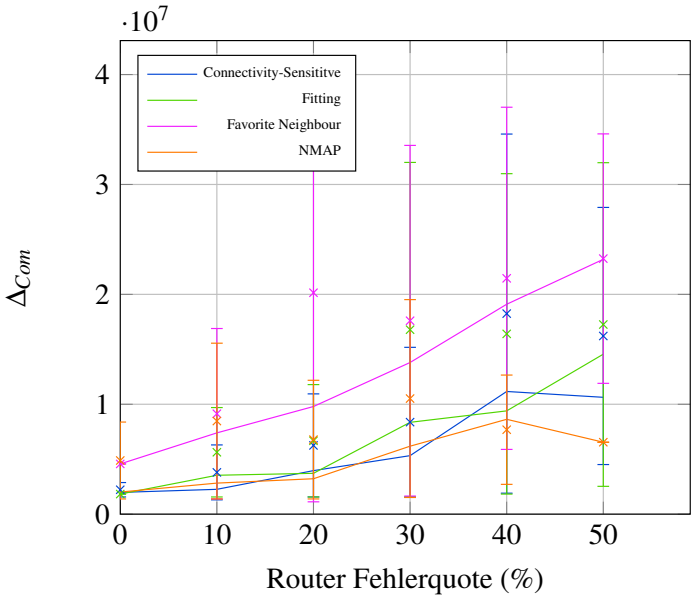


Abbildung 7.14: 8x8 Kern-Graph mit ausgefallenen Routern; VOP-Task-Graph

Zusammen betrachtet zeigen die Ergebnisse der Messungen in Kern-Graphen mit fehlerhaften Routern, dass in den meisten Fällen der einfache *Konnektivitätssensitive Algorithmus* die besten Platzierungen vornimmt. Seine Erweiterungen erweisen sich bei fehlerhaften Routern nicht als sinnvoll, um konstant gute Ergebnisse zu erreichen. Die NMAP Platzierungen sind bei der Abbildung des VOP-Task-Graphen und der beiden Applikationspakete sehr gut. Allerdings zeigt sich bei Fork-Join-Lasten und einer erhöhten Anzahl defekter Router, dass Platzierungen nicht mehr zuverlässig erzeugt werden. Somit ist keine generelle Empfehlung für NMAP auszusprechen, wenn Fehler in der Hardware kompensiert werden sollen.

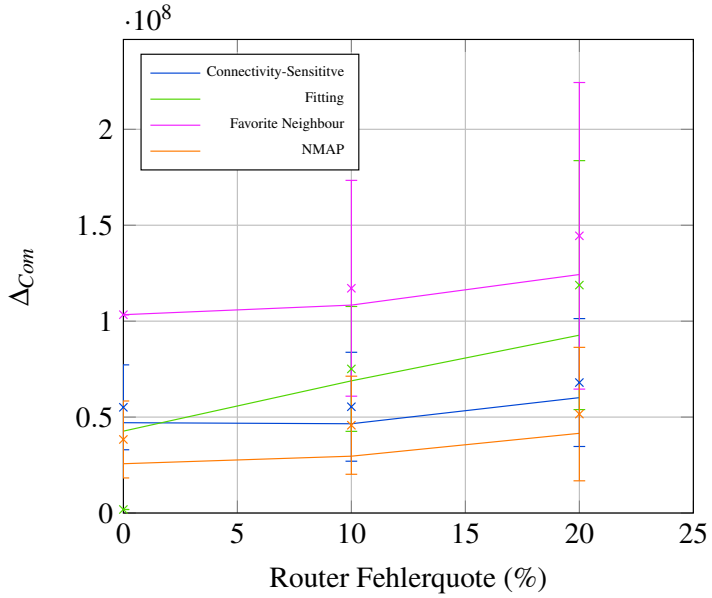


Abbildung 7.15: 8x8 Kern-Graph mit ausgefallenen Routern; Applikationspaket 1

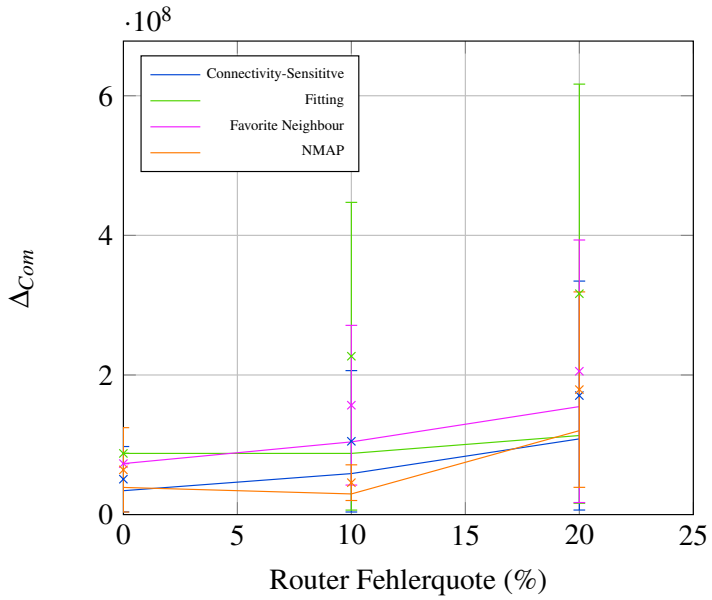


Abbildung 7.16: 8x8 Kern-Graph mit ausgefallenen Routern; Applikationspaket 2

7.2.2 Fehlerhafte Verbindungen

Ist eine Verbindung zwischen zwei Routern fehlerhaft, so kann sie nicht mehr für die Kommunikation verwendet werden. Ein angeschlossener Router funktioniert allerdings auch weiterhin, solange nicht alle seine Verbindungen ausgefallen sind. Mit angrenzenden Kernen verhält es sich genauso. Damit fallen Link-Fehler nicht so schwer ins Gewicht wie Router-Fehler. In Abbildung 7.17 sind die Ergebnisse einer Simulation mit einer Link-Fehlerquote von 10% dargestellt.

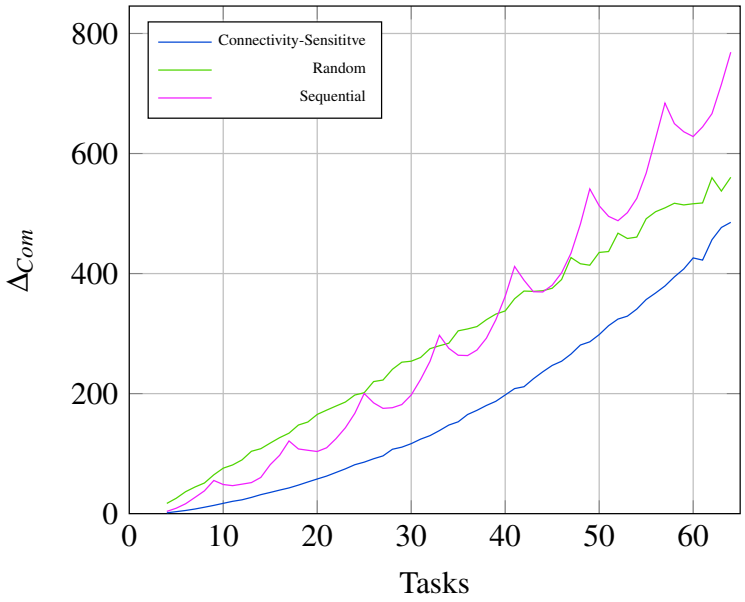


Abbildung 7.17: 8x8 Kern-Graph mit 10% ausgefallenen Verbindungen; Fork-Join Task-Graphen

Das Diagramm sieht dem Ergebnis einer Platzierung ohne Fehler sehr ähnlich. Die Kurven sind leicht nach oben verschoben. Für den *Konnektivitätssensitiven Algorithmus* ergibt sich lediglich ein zusätzlicher Kommunikationsoverhead von 5%. Für die Random-Metrik liegt er bei 7%. Die Δ_{COM}-Werte der sequentiellen Platzierung steigen nur um 3%. Die Form und der Verlauf der Kurven haben sich nicht verändert, allerdings sind die Minimum-/Maximum-Intervalle etwas gewachsen. Die sequentielle Metrik weist nun auch Variationen auf, die im fehlerfreien Fall nicht zu sehen waren. Dies liegt an den teils erhöhten Kommunikationskosten durch Umwege, die von defekten Links ausgelöst wurden. Eine Fehlerquote von 10% unbrauchbaren Links fällt also offensichtlich kaum ins Gewicht. Die Link-Fehler werden erst kritisch, wenn die Fehlerquote drastisch steigt. Im Anhang sind Diagramme für steigende Fehlerraten abgebildet (9.14, 9.15 und 9.16). Auffällig ist, dass bei defekten Verbindungen bis zu einer hohen Fehlerquote noch jeder Kern im Kern-Graph erreichbar bleibt. Somit können

auch größere Task-Graphen weiterhin platziert werden. Die Variation der Ergebnisse steigt mit wachsender Fehleranzahl weiter an bis bei 40% defekten Verbindungen einige der größeren Task-Graphen nicht mehr platziert werden können.

Wie bereits in den vorherigen Fällen bleibt der *Konnektivitätssensitive Algorithmus* auch bei fehlerhaften Verbindungen die beste Wahl für eine effiziente Abbildung der Tasks.

Im Extremfall einer Link-Fehlerquote von 50% (siehe Abbildung 7.18) ist die Random-Metrik kaum mehr zu gebrauchen. Bereits ab 15 Tasks sind nur noch vereinzelt Platzierungen möglich. Die sequentielle Metrik kann immerhin bis zu 30 Tasks platzieren.

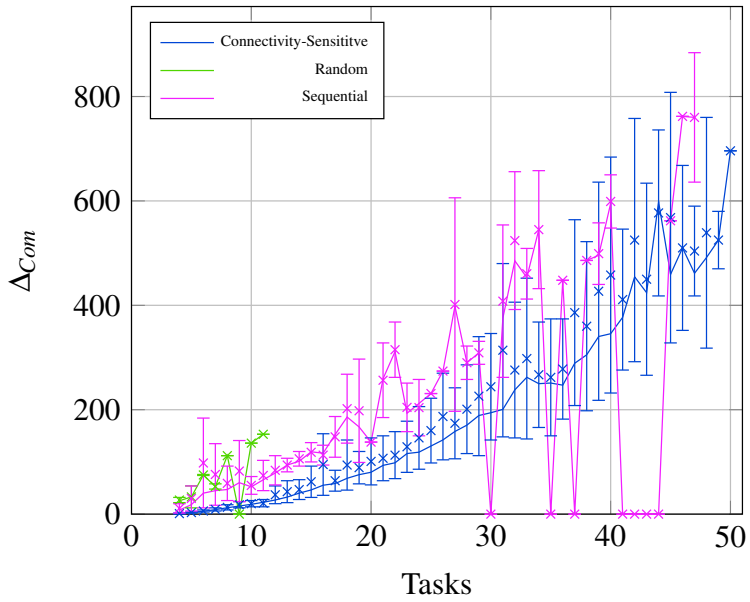


Abbildung 7.18: 8x8 Kern-Graph mit 50% ausgefallenen Verbindungen; Fork-Join Task-Graphen

Der *Konnektivitätssensitive Algorithmus* hingegen platziert mit einigen Schwankungen bis zu 50 Tasks auf den beschädigten Kern-Graphen. Damit ist er bei synthetischen Task-Graphen im Falle defekter Verbindungen den beiden Vergleichsmetriken überlegen.

Bei der Platzierung des VOP-Decoder Task-Graphen ist der *Konnektivitätssensitive Algorithmus* ebenfalls überlegen. In Abbildung 7.19 steigt sein Δ_{COM}-Wert mit steigender Fehlerquote um 348%.

Die Random-Metrik und die Sequential-Metrik liegen allerdings bei jeder Fehlerwahrscheinlichkeit oberhalb der erzielten Ergebnisse. Dies geht sogar so weit, dass die Minimim-/Maximum-Intervalle des *Konnektivitätssensitiven Algorithmus* in jedem Fall unter denen der Vergleichsmetriken liegen. Damit ist der Algorithmus nicht nur im

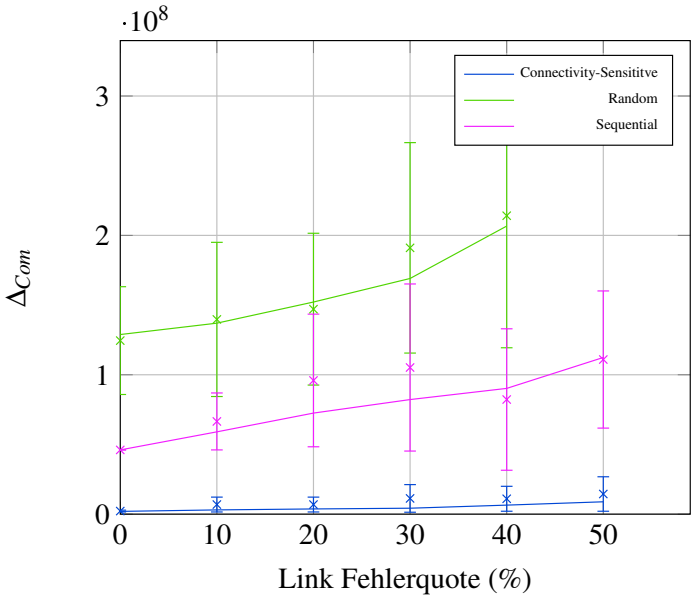


Abbildung 7.19: 8x8 Kern-Graph mit ausgefallenen Verbindungen; VOP-Task-Graph

Schnitt, sondern absolut besser. Die sequentielle Metrik ist in jedem Fall effizienter als die zufällige Platzierung. Dies ist darauf zurückzuführen, dass bei der Platzierung des 19-teiligen VOP-Decoder Task-Graphen viele zufällige Konstellationen ohne starken Zusammenhang und daraufhin mit längeren Kommunikationswegen erzeugt werden.

Wird eine größere Menge Tasks platziert (siehe Abbildung 7.20), ist ein prozentualer Anstieg von 92% bei den Δ_{COM}-Werten des *Konnektivitätssensitiven Algorithmus* zu beobachten.

Durch die höhere Auslastung des Kern-Graphen kommt es bei größeren Fehlerquoten zu Problemen bei der korrekten Platzierung der Task-Graphen. Dies wirkt sich auf die Sequental-Metrik und die Random-Metrik in der Weise aus, dass ab einer Fehlerwahrscheinlichkeit von 30% die Δ_{COM} Ergebnisse besser werden. Hier konnten nicht alle Task-Graphen des Applikationspakets gültig platziert werden. Implizit ist dieses Verhalten schon vorher zu sehen, da die Minimum-/Maximum-Intervalle von Sequential und Random ab 20% Linkfehlern sehr groß sind. Offensichtlich sind hier bereits einige wenige ungültige Platzierungen vorhanden. Der *Konnektivitätssensitive Algorithmus* führt in jedem untersuchten Fall eine vollständige Abbildung aller Task-Graphen durch.

Applikationspaket 2 bereitet dem Algorithmus etwas mehr Probleme. Obwohl immer noch alle Abbildungen durchgeführt werden können, steigen die Δ_{COM}-Werte, vom fehlerfreien Fall zu einer 50 %-igen Ausfallwahrscheinlichkeit, im Schnitt um 753%. Positiv zu bemerken ist allerdings, dass die Vergleichsmetriken, wie in Abbildung 7.21 zu sehen, deutlich schlechter abschneiden.

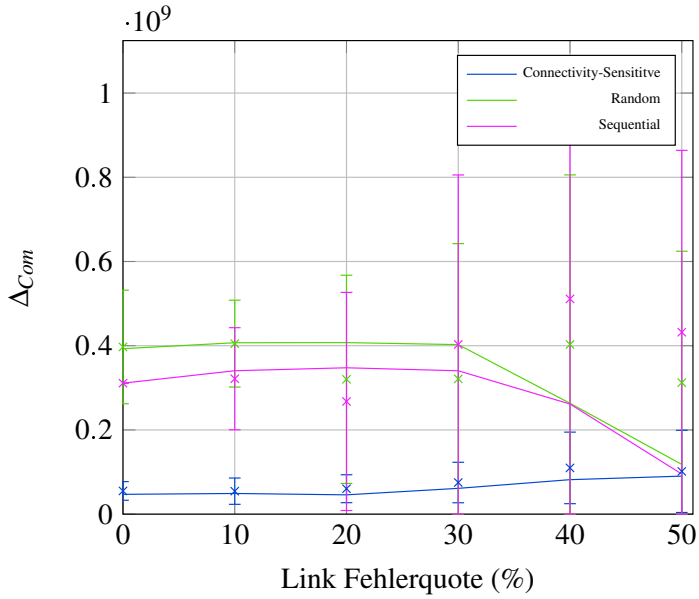


Abbildung 7.20: 8x8 Kern-Graph mit ausgefallenen Verbindungen; Applikationspaket 1

Bei 50% ausgefallenen Verbindungen ist die Random-Metrik nicht mehr in der Lage, eine gültige Platzierung zu finden, und bei 40% schafft sie es meist nur, einen der beiden Task-Graphen unterzubringen. Die Variationen sind außerdem viel zu groß, um eine verlässliche Abbildung zu gewährleisten. Auch die Sequential-Metrik wird bei größeren Fehlerquoten unzuverlässig. Bereits bei 30% wird meistens nur einer der beiden Task-Graphen platziert. Interessant ist die Fehlerquote von 50%, bei der ein besseres Gesamtergebnis als beim *Konnektivitätssensitiven Algorithmus* erzielt wird, wohlgemerkt nur für einen anstatt zwei Task-Graphen. Die Schranken an dieser Stelle deuten darauf hin, dass sehr oft keine gültige Platzierung erzeugt werden kann, vereinzelt jedoch einer der Task-Graphen abgebildet werden konnte. Im Mittel ist hier mit der Sequential-Metrik keine zuverlässige Platzierung möglich.

Auch bei defekten Verbindungen ist der *Konnektivitätssensitive Algorithmus* daher den beiden einfachen Metriken Random und Sequential vorzuziehen. Wie er sich unter diesen Umständen gegenüber seinen Erweiterungen und dem NMAP-Algorithmus behaupten kann, ist in Abbildung 7.22 zu sehen.

Der generelle Verlauf der Kurven sieht dem fehlerfreien Fall sehr ähnlich. Bei der Betrachtung der Minimum-/Maximum-Intervalle (Abb. 9.12) fällt allerdings auf, dass sich diese etwas vergrößert haben und außerdem wesentlich ungleichmäßiger ausfallen. Auch der *Favorite Neighbour Algorithmus* besitzt in diesem Fall solche Intervalle. Durch zufällig ausfallende Verbindungen wird also mehr Unruhe in die Platzierungsergebnisse gebracht.

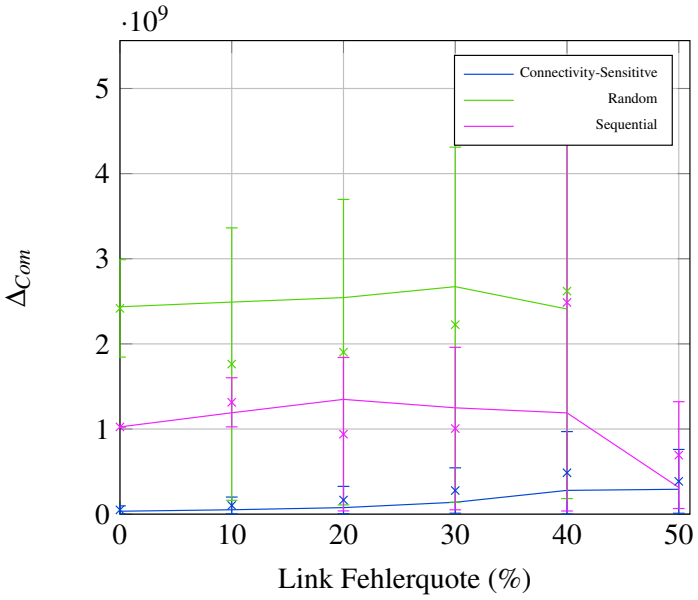


Abbildung 7.21: 8x8 Kern-Graph mit ausgefallenen Verbindungen; Applikationspaket 2

Bei der Fehlerquote von 10% erzeugt der *Konnektivitätssensitive Algorithmus* die besten Platzierungen. NMAP und *Fitting* produzieren in etwa vergleichbare Resultate. Der *Favorite Neighbour Algorithmus* zeigt wieder seine charakteristischen Zacken, kann allerdings keine konkurrenzfähigen Platzierungen erzeugen.

Bei Erhöhung der Quote von fehlerhaften Verbindungen (9.18, 9.19, 9.20 und 7.23) ist zu beobachten, dass sich die Zacke des *Favorite Neighbour Algorithmus* immer weniger ausprägt als im fehlerfreien Fall. Gleichzeitig wachsen die Abstände zwischen Minimum und Maximum aller Algorithmen. Am wenigsten ist dieses Verhalten bei NMAP zu beobachten. Ab 40% ausgefallenen Links beginnen alle Platzierungsverfahren aufgrund der Gegebenheiten des Verbindungsnetzes stark zu schwanken. Bei hoher Auslastung des Kern-Graphen sind alle Algorithmen in etwa gleichwertig. Vorher ist *Favorite Neighbour* klar im Nachteil.

Bei einer Fehlerquote von 50% ist der NMAP-Algorithmus praktisch unbenutzbar. Er erzeugt nur noch vereinzelt gültige Platzierungen. Die anderen Algorithmen arbeiten weiterhin zuverlässig. Aufgrund der allgemeinen Platzierungsqualität ist eine Empfehlung für den *Konnektivitätssensitiven Algorithmus* auszusprechen. Er produziert, wenn auch nur mit einem geringen Abstand, die Platzierungen mit dem geringsten Kommunikationsoverhead.

Für synthetische Fork-Join Task-Graphen ist der *Konnektivitätssensitive Algorithmus* daher auch eine gute Wahl, wenn ausgefallene Verbindungen kompensiert werden müssen.

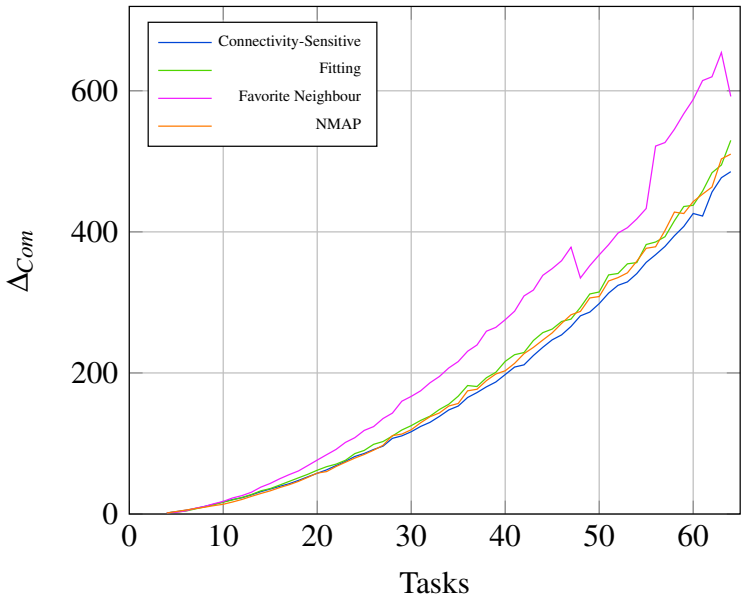


Abbildung 7.22: 8x8 Kern-Graph mit 10% ausgefallenen Verbindungen; Fork-Join Task-Graphen

Dieses Empfehlung lässt sich auch für die Platzierung des VOP-Decoder Task-Graphen aussprechen. Wie in Abbildung 7.24 zu sehen ist, kann der *Konnektivitätssensitive Algorithmus* abermals die besten Ergebnisse liefern.

Bei der Untersuchung der synthetischen Task-Graphen hat sich schon angekündigt, dass der NMAP-Algorithmus nicht sehr gut mit vielen ausgefallenen Verbindungen zurecht kommt. In diesem Fall ist zu sehen, dass NMAP kein gültiges Ergebnis erzeugen kann. Damit ist er für Situationen mit vielen Fehlern ungeeignet. Bis zu einer Fehlerquote von 40% kann er allerdings mit einem geringeren Minimim-/Maximum-Intervall aufwarten und somit vorhersagbare Platzierungsqualität bieten. Der *Fitting Algorithmus* rangiert nur leicht oberhalb des *Konnektivitätssensitiven* und weist somit eine vergleichbare Qualität auf. *Favorite Neighbour* bildet wieder das Schlusslicht. Zusätzlich sind seine Intervalle zu groß, um eine konsistente Platzierung zu gewährleisten.

Bei einem von Applikationspaket 1 ausgefüllten Kern-Graphen sieht NMAP zunächst wie der klare Gewinner aus. Doch für eine 50%ige Fehlerquote kann wieder keine gültige Platzierung gefunden werden. Alle anderen Algorithmen finden bei einer derartig hohen Fehlerquote Platzierungen, wenn auch mit einem großen Minimim-/Maximum-Intervall.

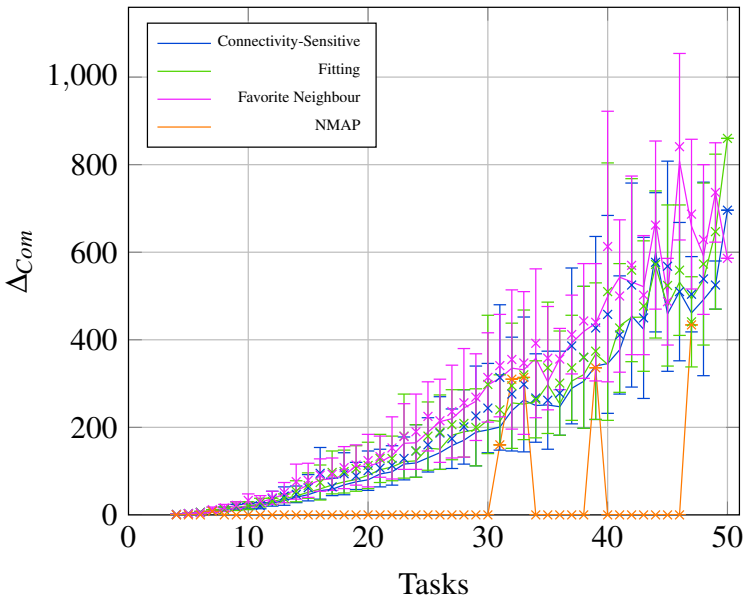


Abbildung 7.23: 8x8 Kern-Graph mit 50% ausgefallenen Verbindungen; Fork-Join Task-Graphen

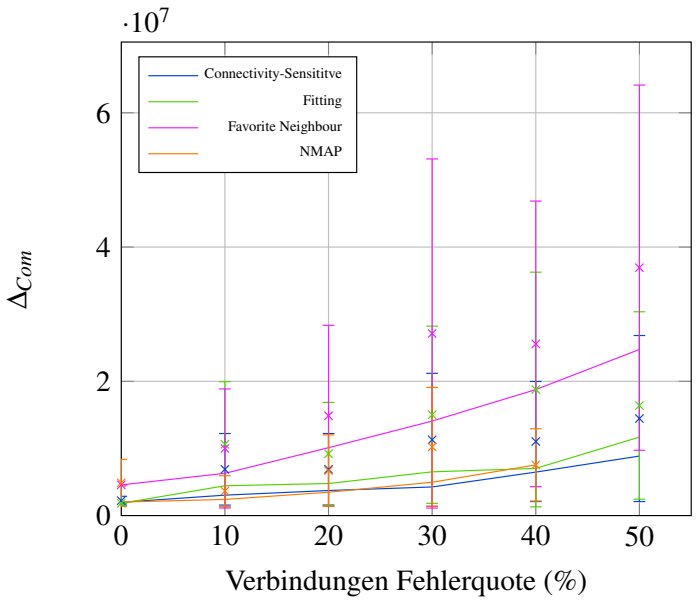


Abbildung 7.24: 8x8 Kern-Graph mit ausgefallenen Verbindungen; VOP-Task-Graph

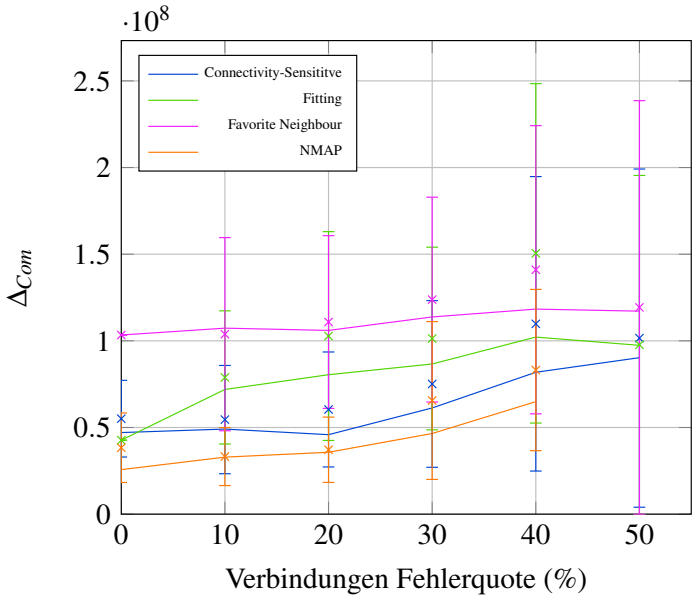


Abbildung 7.25: 8x8 Kern-Graph mit ausgefallenen Verbindungen; Applikationspaket 1

Im fehlerfreien Fall kann *Fitting* den *Konnektivitätssensitiven Algorithmus* noch übertreffen. Sobald allerdings defekte Verbindungen vorliegen, ist die Situation umgekehrt. Dies liegt an der Arbeitsweise des *Fitting Algorithmus*, der in Ecken und damit unter Umständen in Sackgassen beginnt. Damit schafft er sich eine ungünstige Ausgangssituation. Der *Favorite Neighbour Algorithmus* ist in allen Fällen schlechter als die anderen Algorithmen.

Wird das Applikationspaket 2 platziert, ist der NMAP-Algorithmus nicht mehr so gut wie bei Applikationspaket 1. Es stellt sich heraus, dass er sogar in den meisten Fällen das schlechteste Ergebnis erzeugt. Außerdem kann er auch im Fall der aktuell betrachteten Evaluierung keine Platzierung bei einer Fehlerquote von 50% ausgefallenen Verbindungen bestimmen.

Der *Konnektivitätssensitive Algorithmus* produziert wie bei den vorherigen Applikationen gute Ergebnisse, wird aber ab einer Fehlerquote von 40% vom *Fitting Algorithmus* überflügelt. Bei einer Fehlerquote von 50% ist sogar der *Favorite Neighbour Algorithmus*, der bisher immer schlecht abgeschnitten hatte, der beste. Wie am großen Intervall zwischen dessen minimalem und maximalem Δ_{COM}-Wert zu sehen, platziert Favorite Neighbour allerdings nur einen der beiden Task-Graphen. Abhängig von der Konstellation der Linkfehler können auch der *Fitting* und der *Konnektivitätssensitive Algorithmus* ungültige Platzierungen produzieren. In den meisten Situationen werden allerdings funktionsfähige Abbildungen erzeugt.

Bei der Analyse der Simulationsergebnisse unter Annahme von defekten Verbindungen hat sich in den meisten Fällen der *Konnektivitätssensitive Algorithmus* als der

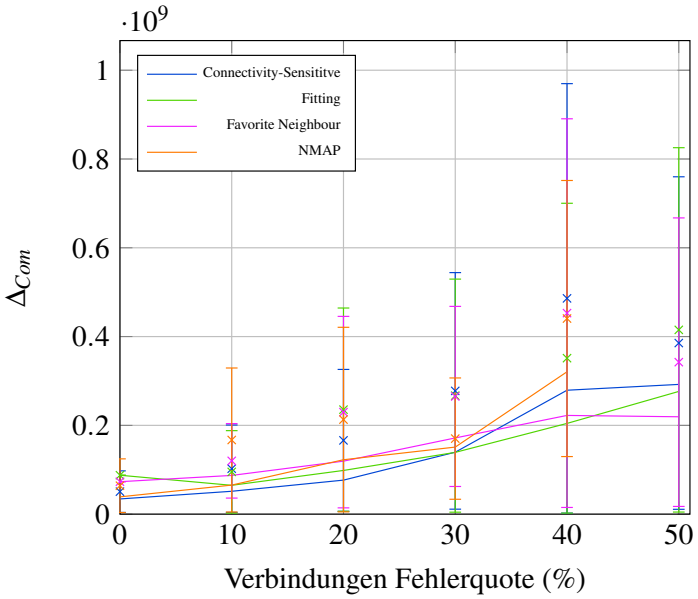


Abbildung 7.26: 8x8 Kern-Graph mit ausgefallenen Verbindungen; Applikationspaket 2

beste herausgestellt. *Fitting* bietet nur in sehr wenigen Situationen eine Verbesserung. Der *Favorite Neighbour Algorithmus* liegt meist weit von einem guten Ergebnis entfernt und ist damit nicht zu empfehlen, um fehlerhafte Verbindungen im Interconnect zu behandeln. NMAP produziert oft gute Ergebnisse, hat aber, wie bereits im Fall defekter Router, bei höheren Fehlerquoten Probleme überhaupt eine gültige Platzierung zu erzeugen.

7.2.3 Fehlerhafte Kerne

Durch einen defekten Kern wird der angrenzende Router nicht beeinflusst. Der Kern kann lediglich keine Tasks mehr ausführen. Die Kommunikation anderer Kerne über seine Position im Kern-Graphen ist weiterhin uneingeschränkt möglich. Der Ausfall eines Kerns ist damit nicht so gravierend wie der Ausfall eines Routers, da die Kommunikation nicht beeinflusst wird.

Bei einer 10 prozentigen Ausfallwahrscheinlichkeit der Kerne schlägt sich dieser Vorteil noch nicht so deutlich nieder (siehe Abbildung 7.27).

Die Kurven verlaufen weitestgehend analog zu denen der Analyse eines gleichstarken Ausfalles der Router. Bei defekten Routern werden leicht höhere Δ_{COM}-Werte erzeugt. Auch ist die Variation der minimal und maximal erzeugten Werte stärker ausgeprägt (siehe Anhang Abb. 9.21). Die Qualität der Platzierungsverfahren stellt sich wie erwartet dar, indem der *Konnektivitätssensitive Algorithmus* die günstigste

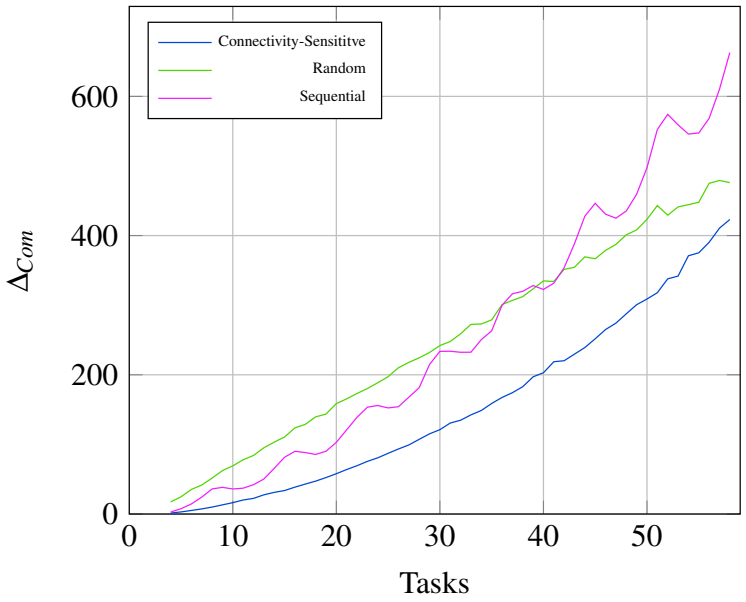


Abbildung 7.27: 8x8 Kern-Graph mit 10% ausgefallenen Kernen; Fork-Join Task-Graphen

Platzierung erzeugt und die einfachen Metriken weit hinter sich lässt.

Dieses Muster setzt sich auch bei höheren Ausfallquoten fort (siehe Anhang Abbildungen 9.22, 9.23 und 9.24). Allerdings sind auch bei hohen Ausfallwahrscheinlichkeiten keine starken Schwankungen oder unmögliche Platzierungen zu erkennen, da ausgefallene Kerne keinerlei Einfluss auf das Netzwerk ausüben. Die Kommunikationswege können somit zwar länger werden, ungünstige Segmentierungen des Netzes werden jedoch vermieden.

Auch bei einer Ausfallwahrscheinlichkeit von 50% (Abb. 7.28) erzeugen alle Verfahren noch eine gültige Platzierung.

Bei geringen Taskanzahlen ist das Random-Verfahren bei allen Fehlergrößen besser als die sequentielle Metrik. Bei einem hohen Füllstand des Kern-Graphen kehrt sich dieses Verhältnis allerdings um. Keine der beiden einfachen Metriken kann die Qualität einer Platzierung mit dem *Konnektivitätssensitiven Algorithmus* erreichen.

Bei der Betrachtung der entstandenen Platzierung eines einzelnen VOP-Decoder Task-Graphen unter Annahme von fehlerhaften Kernen (Abb. 7.29) ergibt sich in etwa das erwartete Bild. Interessant ist allerdings, dass die sequentielle Metrik bis zu einem gewissen Punkt unter Annahme von Fehlern leicht bessere Platzierungen erzeugen kann als im fehlerfreien Fall. Dies ist damit zu begründen, dass durch die vollständige Ausnutzung der Zeilenbreite im fehlerfreien Fall größere Kommunikationsdistanzen entstehen als durch unterbrochene Zeilen im Fehlerfall.

Der *Konnektivitätssensitive Algorithmus* besitzt einen sehr großen Vorsprung. Seine

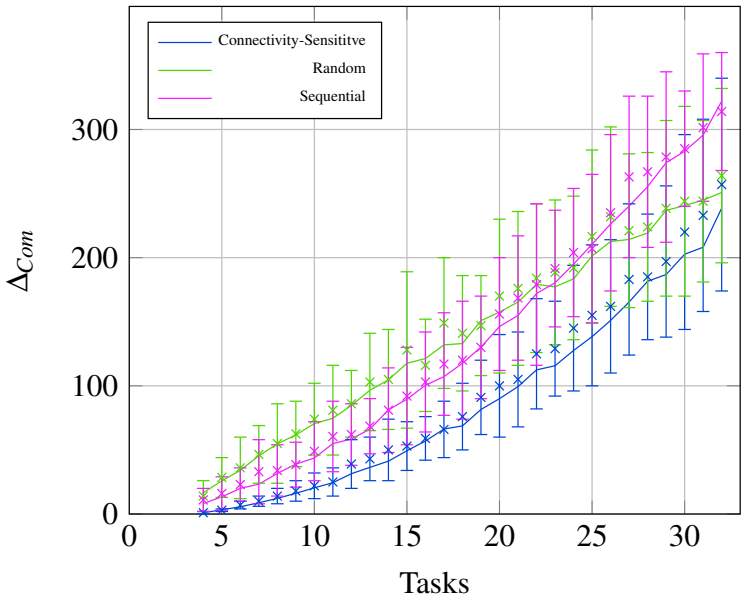


Abbildung 7.28: 8x8 Kern-Graph mit 50% ausgefallenen Kernen; Fork-Join Task-Graphen

Maximalwerte liegen noch unterhalb der Minimalwerte der anderen Metriken.

Wenn der Kern-Graph besser gefüllt ist, lässt sich dieses Verhalten weiterhin beobachten. Bei Applikationspaket 1 ist der *Konnektivitätssensitive Algorithmus* immer noch zu bevorzugen (siehe Abb. 7.30).

Die Random-Metrik ist im Schnitt beinahe über die verschiedenen Fehleranzahlen konstant, weist aber eine hohe Variation auf. Interessanterweise werden die durchschnittlichen Ergebnisse der sequentiellen Metrik mit zunehmender Fehlerwahrscheinlichkeit immer besser, während die Minimum- und Maximum-Werte in etwas gleich bleiben. Wie schon bei der Analyse des VOP-Decoder Task-Graphen ist dies auf verringerte Abstände durch die unvollständige Ausnutzung der Zeilen zu begründen. Gerade bei stark verzweigenden Task-Graphen (wie beispielsweise dem VOP-Decoder) oder bei kleinen Task-Graphen, die gerade über das Ende einer Zeile hinausgehen, ist dieser Umstand von Vorteil. Die sequentielle Platzierungsmetrik ist damit in jedem Fall besser als eine zufällige Platzierung.

Applikationspaket 2 liefert ein ähnliches Bild (Abb. 7.31). Auch hier ist eine sequentielle Platzierung ist wesentlich besser als eine zufällige.

Bei der Platzierung der zwei großen Applikationen kann sich die sequentielle Metrik bei höheren Fehlerquoten allerdings nicht so viel verbessern wie bei Applikationspaket 1. Werden die Kommunikationswege bei einem der beiden Task-Graphen beeinträchtigt, kann dies für den anderen Task-Graphen von Vorteil sein. Der *Konnektivitätssensitive Algorithmus* kann die beiden einfachen Platzierungsmetriken deutlich hinter

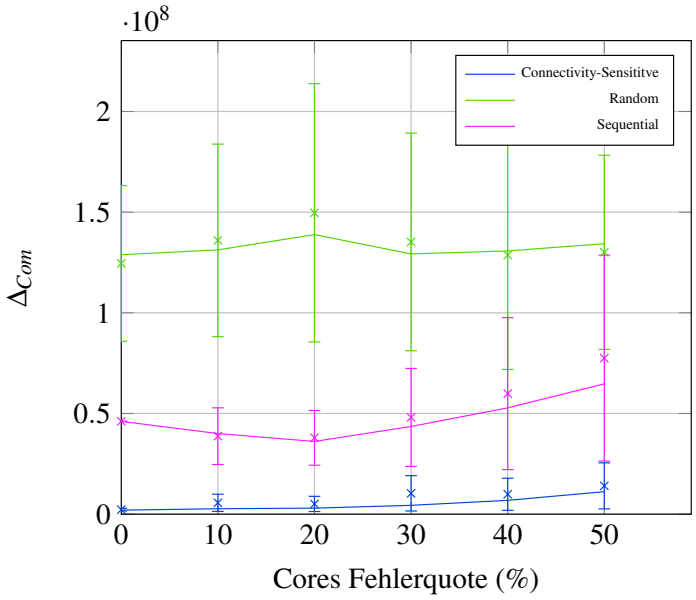


Abbildung 7.29: 8x8 Kern-Graph mit ausgefallenen Kerne; VOP-Task-Graph

sich lassen.

Im Vergleich mit den einfachen Metriken Sequential und Random ist der *Konnektivitätssensitive Algorithmus* unter Annahme von defekten Kernen am besten für eine kommunikationsoptimierte Platzierung geeignet.

Die Unterschiede zu den erweiterten Algorithmen und NMAP bei synthetischen Task-Graphen sind in Abbildung 7.32 zu sehen.

Bei der geringen Fehlerquote von 10% ist ein Kurvenverlauf zu beobachten, der dem der Routerausfälle ähnlich ist. Defekte Router erzeugen allerdings leicht höhere Δ_{COM}-Werte. Die übrigen charakteristischen Eigenschaften der Kurven sind erhalten geblieben. *Favorite Neighbour* weist immer noch den Sprung bei Task-Graph-Größen von 48 auf. Der *Konnektivitätssensitive Algorithmus* liefert bessere Platzierungen als *Fitting* und NMAP. Die Minimum-/Maximum-Variationen sind allerdings bei defekten Kernen weniger gravierend als bei defekten Routern (siehe Abbildung 9.25 im Anhang). Dies liegt daran, dass die Kommunikationswege durch einen Kern-Ausfall nicht beeinträchtigt werden.

Tritt eine höhere Anzahl von Kern-Fehlern auf, bleibt die Tendenz der Kurven bestehen (siehe Abbildungen 9.26, 9.27 und 9.28 im Anhang). *Favorite Neighbour* bleibt der schlechteste Algorithmus. Die besten Ergebnisse erzielt der *Konnektivitätssensitive Algorithmus*. Allerdings bewegen sich *Fitting* und NMAP sehr nahe an dessen Werten. Somit ist jeder der drei Algorithmen unter Annahme von defekten Kernen gut für die Platzierung geeignet.

Bei einer astronomischen Fehlerquote von 50% finden alle Algorithmen immer noch

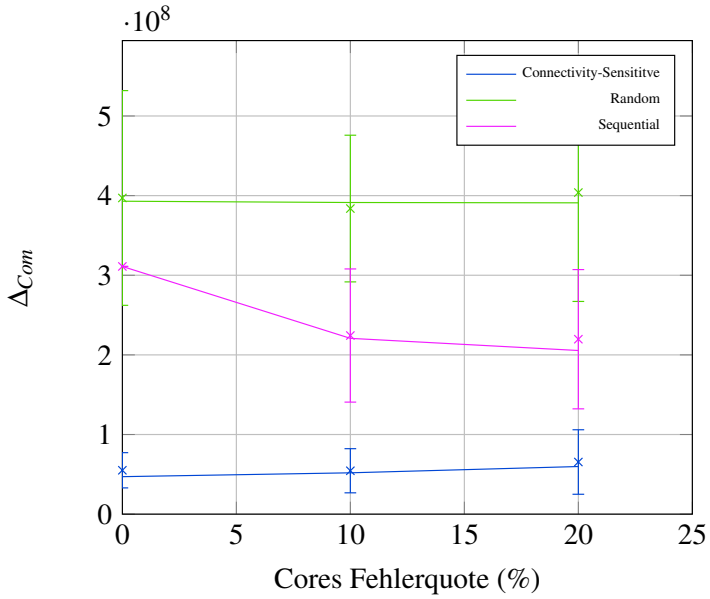


Abbildung 7.30: 8x8 Kern-Graph mit ausgefallenen Kerne; Applikationspaket 1

eine gültige Platzierung (Abb. 7.33), da die Kommunikationswege nicht beeinträchtigt werden. Im Fall der defekten Router hat hier der NMAP-Algorithmus schon früh starke Probleme, gültige Platzierungen zu erzeugen.

Interessanter ist allerdings, ob die Algorithmen auch bei realistischen Applikationen gute Platzierungen erzeugen. Die Abbildung eines einzelnen VOP-Decoder Task-Graphen auf einen Kern-Graphen mit defekten Kernen stellt wiederum den *Favorite Neighbour Algorithmus* als den schlechtesten Kandidaten dar (siehe Abb. 7.34).

Die anderen drei Algorithmen bewegen sich auf einem ähnlichen Qualitätsniveau. Bei hohen Fehlerraten ist diesmal allerdings der NMAP-Algorithmus am besten für eine Platzierung mit effizienter Kommunikation geeignet. Von den drei besseren Algorithmen ist der *Konnektivitätssensitive* bei mehr als 40% ausgefallener Kerne am schlechtesten. Allerdings muss auch beachtet werden, dass die Intervalle der Minimum-/Maximumwerte bei allen Algorithmen sehr groß sind. Hier gibt es keinen Kandidaten, der bei hohen Fehlerwahrscheinlichkeiten eine konstant gute Qualität der Platzierungen produzieren kann.

Bei Applikationspaket 1 kann NMAP wie bei der Einzelplatzierung des VOP-Decoders wieder die besten Ergebnisse erzielen. Der Verlauf der NMAP-Kurve ist bei niedrigeren Δ_{COM}-Werten beinahe parallel zu der des *Konnektivitätssensitiven Algorithmus*. Die Variabilität ist bei NMAP etwas geringer. *Fitting* ist im fehlerfreien Fall leicht besser als der *Konnektivitätssensitive Algorithmus*, wird danach allerdings schnell schlechter. Der *Favorite Neighbour* ist, wie bereits in den vorherigen Untersuchungen, am schlechtesten.

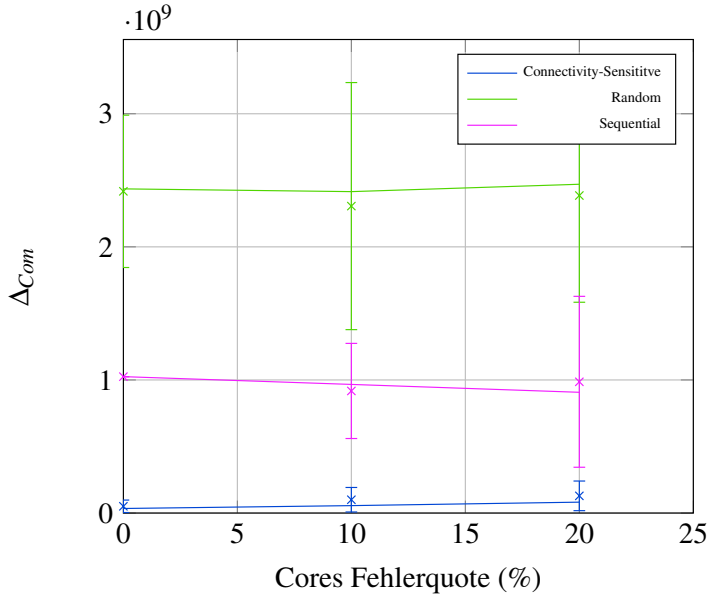


Abbildung 7.31: 8x8 Kern-Graph mit ausgefallenen Kerne; Applikationspaket 2

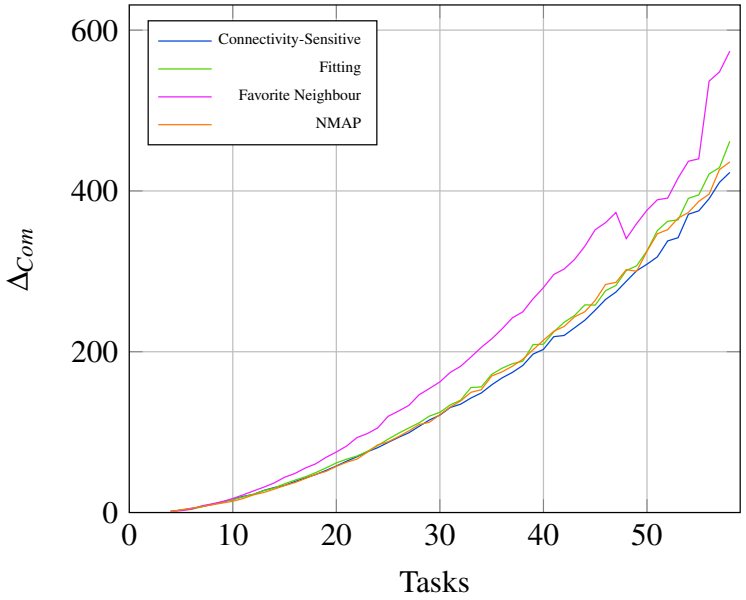


Abbildung 7.32: 8x8 Kern-Graph mit 10% ausgefallenen Kerne; Fork-Join Task-Graphen

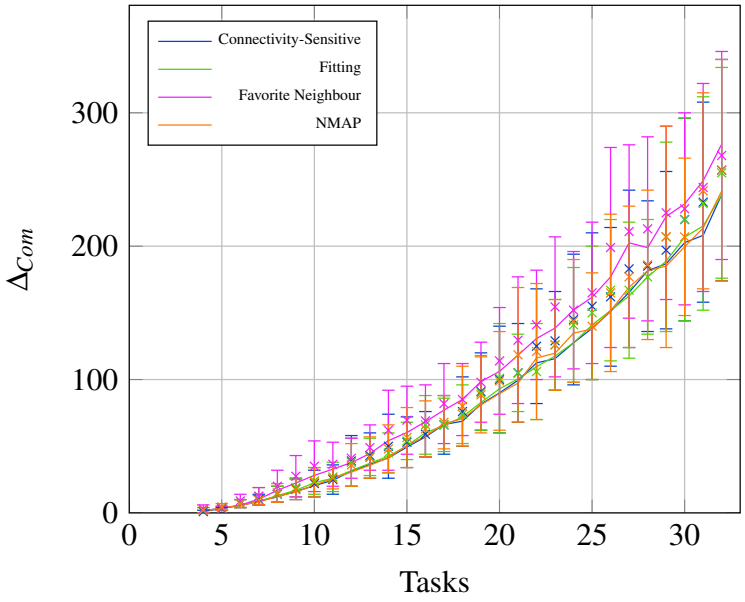


Abbildung 7.33: 8x8 Kern-Graph mit 50% ausgefallenen Kerne; Fork-Join Task-Graphen

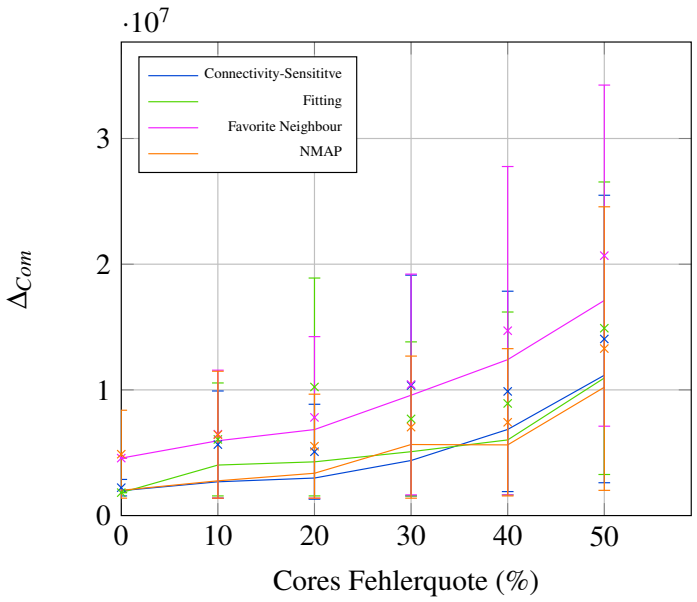


Abbildung 7.34: 8x8 Kern-Graph mit ausgefallenen Kerne; VOP-Task-Graph

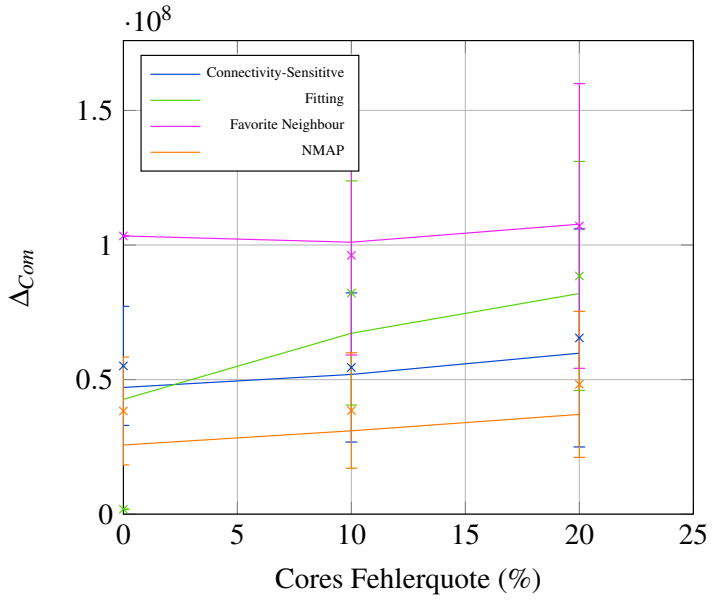


Abbildung 7.35: 8x8 Kern-Graph mit ausgefallenen Kerne; Applikationspaket 1

Bei der Platzierung von Applikationspaket 2 stellen sich die Ergebnisse etwas anders dar (Abb. 7.36).

Zuerst einmal fallen die großen Minimum-/Maximum-Intervalle bei relativ geringen Durchschnittswerten auf. Der Grund ist darin zu sehen, dass in diesem Applikationspaket zwei große und stark verbundene Task-Graphen platziert werden. Hier kann es schnell dazu kommen, dass einzelne Tasks ungünstig platziert werden. Dies wird vor allem durch die Position der fehlerhaften Kerne beeinflusst.

Die besten Platzierungen erzeugt der *Konnektivitätssensitive Algorithmus*. NMAP kann diesmal keine besseren Ergebnisse produzieren. Die schlechtesten Werte kommen vom *Favorite Neighbour Algorithmus*, der allerdings im fehlerfreien Fall besser als der *Fitting Algorithmus* abschneidet.

Bei der Behandlung des Platzierungsproblems auf Kern-Graphen mit defekten Kernen ist abschließend zu bemerken, dass der *Konnektivitätssensitive Algorithmus* in allen Fällen den einfachen Metriken Random und Sequential überlegen ist. Bei synthetischen Task-Graphen kann er auch bessere Platzierungen als seine Erweiterungen *Fitting* und *Favorite Neighbour* und als der NMAP-Algorithmus erzeugen. Dies gilt ebenfalls für die Platzierung eines einzelnen VOP-Decoders sowie des Applikationspakets 2. Bei Applikationspaket 1 ist NMAP der beste Algorithmus. In allen Fällen ist *Favorite Neighbour* schlechter als die Vergleichsalgorithmen. *Fitting* erzielt durchschnittliche Ergebnisse, kann aber in keinem Fall Alleinstellungsmerkmale aufweisen.

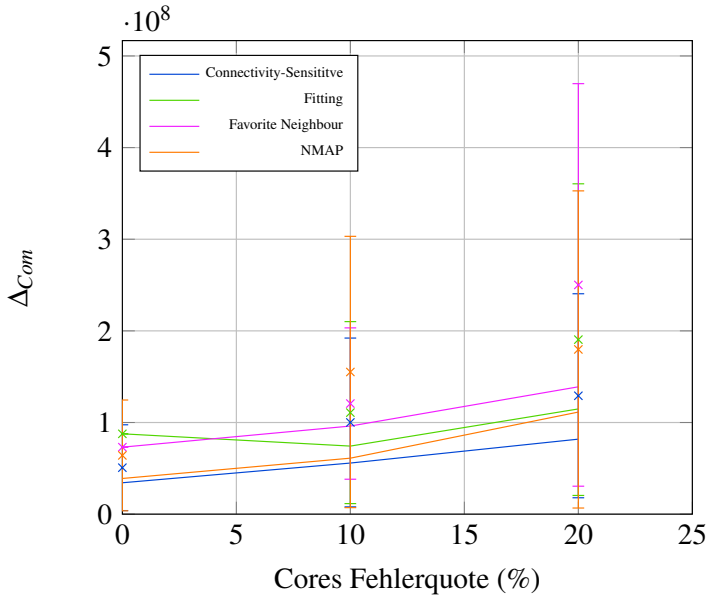


Abbildung 7.36: 8x8 Kern-Graph mit ausgefallenen Kerne; Applikationspaket 2

7.2.4 Kombination der Fehlerarten, Algorithmen und Task-Graphen

Bisher wurden die verschiedenen Fehlerarten isoliert betrachtet. Realistisch gesehen sollten allerdings Kombinationen verschiedener Fehler auftreten.

Die Untersuchungen haben jedoch ergeben, dass die isolierte Betrachtung bereits ausreicht, um gültige Aussagen über das Verhalten der Algorithmen zu treffen. Bei der Kombination von Router- und Linkfehlern wird jeweils das Kommunikationsnetz beschädigt. Routerfehler beeinträchtigen zusätzlich die Anzahl der verfügbaren Kerne, was allerdings bei Linkfehlern auch auftritt, wenn alle an einen Router angrenzenden Links defekt sind oder etwa ganze Bereiche des Kern-Graphen abgeschnitten wurden. Somit können die Effekte von Routerfehlern denen von gehäuften Linkfehlern gleichgesetzt werden.

Defekte Kerne haben hingegen keinen Einfluss auf das Kommunikationsnetz. Allerdings muss ein leichtes Ansteigen der Kommunikationsdistanzen betrachtet werden, da stellenweise längere Wege in Kauf genommen werden müssen, um unbenutzbare Kerne zu umgehen. Daher besitzt die Kombination von fehlerhaften Kernen mit defekten Verbindungen oder Routern weitestgehend die Eigenschaften der fehlerhaften Elemente der Kommunikationsinfrastruktur unter Berücksichtigung eines zusätzlichen leichten Anstiegs der Δ_{COM}-Werte durch die defekten Kerne selbst.

Es ist denkbar, dass in einiges Fällen eine Kombination von unterschiedlichen Algorithmen sinnvoll sein kann. Hierzu muss erst eine Kategorisierung der Gesamtsituation

einer Platzierung vorgenommen werden. Anschließend wird ein passender Algorithmus für die derzeitige Situation ausgewählt. Relevante Kategorien sind die Größe eines zu platzierenden Task-Graphen, der aktuelle Füllstand des Kern-Graphen und die Ausgewogenheit der Gewichtungen der Verbindungen eines Task-Graphen. Auch die allgemeine Verbindungsstruktur eines Task-Graphen kann starken Einfluss auf eine Platzierung haben, was sehr gut bei der Benutzung des *Favorite Neighbour Algorithmus* mit synthetischen Fork-Join-Task-Graphen zu sehen ist.

Bei der Kombination verschiedener Algorithmen bleibt auch zu bedenken, dass die erzielten Vorteile den Aufwand der Analyse der Grundsituation aufwiegen müssen.

Mit geringem Aufwand ist festzustellen, wie groß ein zu platzierender Task-Graph ist. Idealerweise ist die Applikation bereits offline mit diesen Informationen ausgestattet worden, welche bei der Platzierung direkt verwendet werden können. In Abbildung 7.37 ist die Platzierungsqualität bei verschieden großen, einzelnen Fork-Join Task-Graphen zu sehen. Unter den Balken ist jeweils der beste Algorithmus für diese Größe des Task-Graphen aufgeführt. Es wurde ein fehlerfreies Netz angenommen.

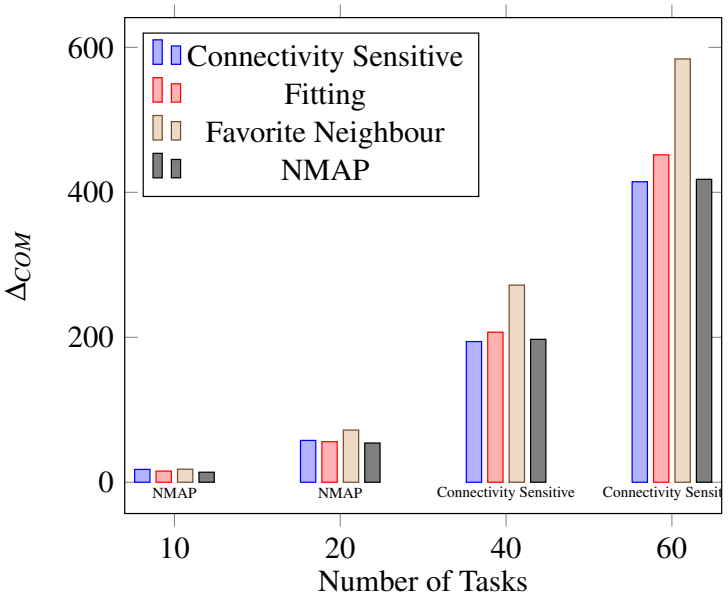


Abbildung 7.37: Platzierung verschieden großer Fork-Join Task-Graphen

Der *Konnektivitätssensitive Algorithmus* ist besser für größere Applikationen geeignet. Bei kleineren Task-Graphen liefert NMAP überlegene Werte. Unter Annahme von fehlerhaften Elementen verschiebt sich dieses Verhältnis zu Gunsten des *Konnektivitätssensitiven Algorithmus*, womit dieser auch bei kleineren Task-Graphen besser als NMAP wird. Somit ist unter Annahme von fehlerhaften Komponenten sowohl bei großen als auch bei kleinen Task-Graphen der *Konnektivitätssensitive Algorithmus* zu empfehlen. Existieren keine oder nur sehr wenige Fehler ist dem NMAP-Algorithmus

oder dem *Fitting Algorithmus* Vorrang zu geben.

Die Qualität der Platzierungsverfahren im Verhältnis zum derzeitigen Füllstand des Kern-Graphen ist sehr einfach an den Ergebnissen der Auswertung unter Annahme von fehlerhaften Kernen abzulesen. Ein defekter Kern verhält sich für die Platzierungsverfahren identisch zu einem belegten Kern. Das Kommunikationsnetz ist weiterhin unbeeinflusst, der betroffene Kern steht allerdings nicht für eine weitere Platzierung zur Verfügung. Den Füllstand des Kern-Graphen festzustellen ist ohne Aufwand möglich, da bekannt ist, welche Kerne bereits belegt sind. Aus den vorangegangenen Auswertungen ist der Schluss zu ziehen, dass der *Konnektivitätssensitive Algorithmus* in einem vollen Netz am besten geeignet ist, um eine Platzierung vorzunehmen. Bei einer geringen Belegung ist NMAP brauchbarer. Allerdings ist hier auch eine Abhängigkeit von der Größe des Task-Graphen festzustellen. Die alleinige Betrachtung des Füllstandes, um einen optimalen Algorithmus auszuwählen, ist daher nicht möglich.

Da der *Favorite Neighbour Algorithmus* speziell für den Fall eines Task-Graphen mit stark unterschiedlichen Kantengewichten entwickelt wurde, ist der Einfluss einer unausgewogenen Gewichtung ebenfalls interessant. Eine aussagekräftige Metrik zu finden, die besagt, wie hier die Verteilung aussieht, ist allerdings nicht ganz einfach. Es wäre denkbar, hier die Varianz der Kantengewichte heranzuziehen, jedoch muss dabei ein Schwellwert festgelegt werden, ab dem ein Task-Graph als unausgewogen gilt. Es ist möglich, diese Metrik offline zu berechnen, da Kantengewichte des Task-Graphen schon vor der Ausführung feststehen.

In einem kleinen Beispiel ist allerdings kein nennenswerter Effekt eines unbalancierten Task-Graphen zu erkennen. In den Abbildungen 9.31 und 9.32 im Anhang ist ein Vergleich zwischen der normalen und einer ausbalancierten Variante des Consumer Task-Graphen zu sehen. Es ist festzustellen, dass unter Annahme von Fehlern die balancierte Variante besser zu platzieren ist. Der Umstand, dass in einem der beiden Task-Graphen ein Ungleichgewicht der Kantenbewertungen vorliegt, hat allerdings keinen nennenswerten Effekt auf die Qualität der *Favorite Neighbour* Platzierung. Diese zieht ihre günstige Bewertung aus der Struktur des Graphen und nicht aus seinen Kantengewichten.

Ein für den *Favorite Neighbour Algorithmus* sehr ungünstiges Ergebnis zeigt die Auswertung der Platzierung eines unbalancierten Fork-Join Task-Graphen. Bei diesem wurden zwei Verbindungen mit höheren Gewichtungen ausgestattet. Im fehlerfreien Fall sind speziell beim *Favorite Neighbour Algorithmus* mehrere starke Schwankungen des Δ_{COM}-Wertes zu beobachten (sieh Abbildung 7.38).

Eine auffällige Steigerung ist bei den Task-Graph-Größen sechs und sieben zu betrachten. Desweiteren treten ab einer Größe von 48 Tasks starke Schwankungen der Δ_{COM}-Werte auf. Diese Phänomene werden zum Teil durch die schlechten Fähigkeiten des *Favorite Neighbour Algorithmus* bei der Platzierung von Fork-Join Task-Graphen hervorgerufen. Die starken Änderungen der Werte liegen jedoch an ungünstiger Positionierung der schwergewichtigen Verbindungen.

Das beschriebene Verhalten setzt sich auch bei der Betrachtung fehlerhafter Kern-Graphen fort (siehe Abb. 9.33 im Anhang). Alle übrigen Algorithmen verlaufen in etwa im selben Verhältnis wie im balancierten Fall. Es kann daher gefolgert werden, dass die Balancierung des Task-Graphen zwar einen Einfluss auf die Qualität einer Platzie-

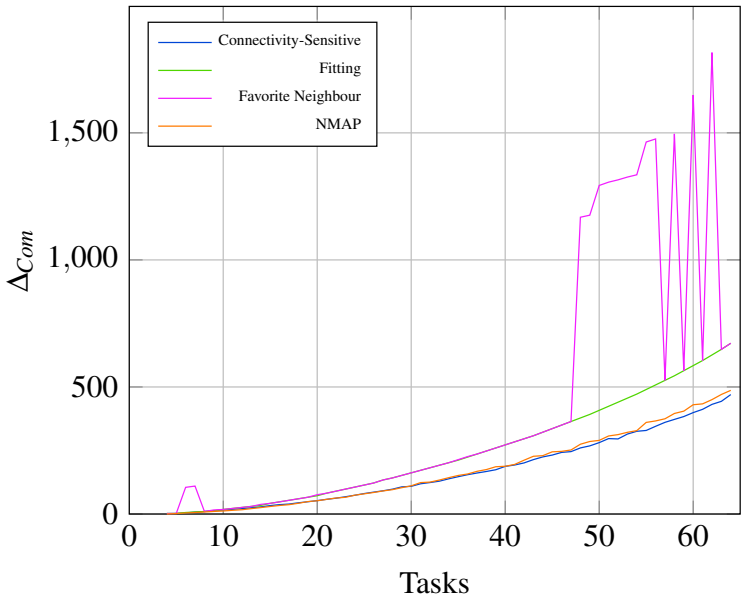


Abbildung 7.38: 8x8 Kern-Graph ohne Fehler; unbalancierte Fork-Join Task-Graphen

rung ausüben kann, drastische Veränderungen allerdings nur den *Favorite Neighbour Algorithmus* betreffen.

Schlussendlich besitzt die Verbindungsstruktur des Task-Graphen auch einen großen Einfluss auf die mögliche Qualität einer Platzierung. Ein rein linearer Task-Graph, wie beispielsweise die Automotive-Anwendung in Abbildung 3.9, lässt sich optimal mit der sequentiellen Metrik positionieren. Stärker verzweigte Task-Graphen sind hingegen nicht so gut sequentiell abbildbar.

Die Charakterisierung der Task-Graph-Struktur ist nicht einfach. Es muss darauf geachtet werden, welche Informationen für einen Algorithmus bei der Platzierung relevant sind. Bei den betrachteten Algorithmen sind dies die Anzahl der Tasks, die Anzahl der Kanten und die Gewichtungen. Die Einflüsse von Taskanzahl und ungleichmäßiger Gewichtung wurden bereits besprochen. Interessant ist daher noch die Anzahl der Verbindungen im Task-Graph, vor allem in Bezug auf die Anzahl der Tasks. Diese Information lässt sich auch einfach als Metrik benutzen. Für die Evaluierung mit verschiedenartigen Task-Graph-Strukturen wurden die Task-Graphen 3.9, 3.12, 3.13 und 9.36 ausgewählt. Der erste Task-Graph ist linear und besitzt daher mehr Tasks als Verbindungen. Der zweite besteht aus einer grundsätzlich linearen Struktur mit einem Fork-Join Abschnitt. Die beiden restlichen Task-Graphen sind stark verzweigt. In Abbildung 7.39 sind diese Platzierungen von links nach rechts aufgelistet.

Alle Algorithmen können den linearen Task-Graphen mit einem Δ_{COM}-Wert von Null platzieren. Treten keine Verzweigungen auf, ist eine Platzierung folglich trivial. Bei dem weitestgehend linearen Task-Graphen mit einem Fork-Join-Abschnitt erzielen

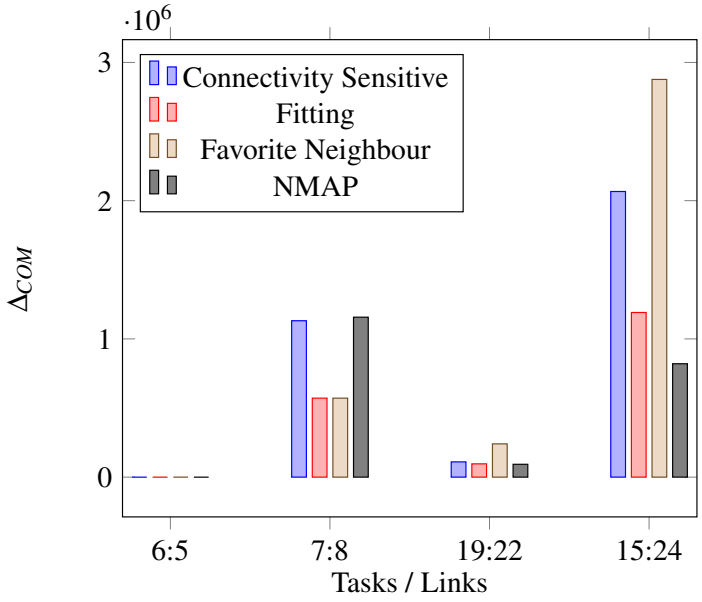


Abbildung 7.39: Platzierung von Task-Graphen mit unterschiedlichem Verzweigungsgrad

der *Favorite Neighbour* und der *Fitting Algorithmus* einen weiten Vorsprung vor den beiden anderen. Bei einem geringen Verzweigungsgrad sind diese Algorithmen zu bevorzugen. Die beiden stark verzweigten Task-Graphen werden am besten von NMAP und dem *Fitting Algorithmus* platziert. Das Schlusslicht bildet dabei jeweils der *Favorite Neighbour Algorithmus*.

Diese Simulationsergebnisse spiegeln die Situation ohne Fehler wider. Kommen defekte Elemente hinzu, ergibt sich ein leicht anderes Bild (siehe Abb. 9.29 im Anhang). Der *Konnektivitätssensitive Algorithmus* schneidet im Verhältnis zu den anderen Algorithmen besser ab. Es muss allerdings eine recht hohe Anzahl Fehler erreicht werden, damit dieser zum besten Kandidaten wird.

Somit ist festzustellen, dass bei geringem Verzweigungsgrad der Task-Graphen *Fitting* und *Favorite Neighbour* im fehlerfreien Fall gute Platzierungen erzeugen. Bei hohem Verzweigungsgrad ist NMAP am besten geeignet. Mit zunehmender Fehlerwahrscheinlichkeit zeigt sich die Qualität des *Konnektivitätssensitive Algorithmus* im Verhältnis immer mehr.

Bei der Auswahl eines Algorithmus für die Platzierungsentscheidung ist besonders auf die Struktur des Task-Graphen zu achten. Unausgewogene Task-Graphen sollten zusätzlich nicht mit dem *Favorite Neighbour Algorithmus* platziert werden. Andere Faktoren spielen kaum eine Rolle.

Sind mehrere Task-Graphen gleichzeitig zu platzieren, stellt sich die Frage, in welcher Reihenfolge das optimale Ergebnis erzielt wird. Größere Task-Graphen benötigen

mehr zusammenhängenden Platz und sind damit günstiger zu Beginn zu platzieren. Applikationspaket 1 enthält mehrere Task-Graphen unterschiedlicher Größe. Durch Variation der Reihenfolge ist eine unterschiedliche Platzierungsqualität zu beobachten. In Abbildung 7.40 wurden die Applikationen aus dem Paket in vier verschiedenen Reihenfolgen platziert. Zuerst in absteigender, danach in aufsteigender Reihenfolge, anschließend abwechselnd große und kleine Task-Graphen.

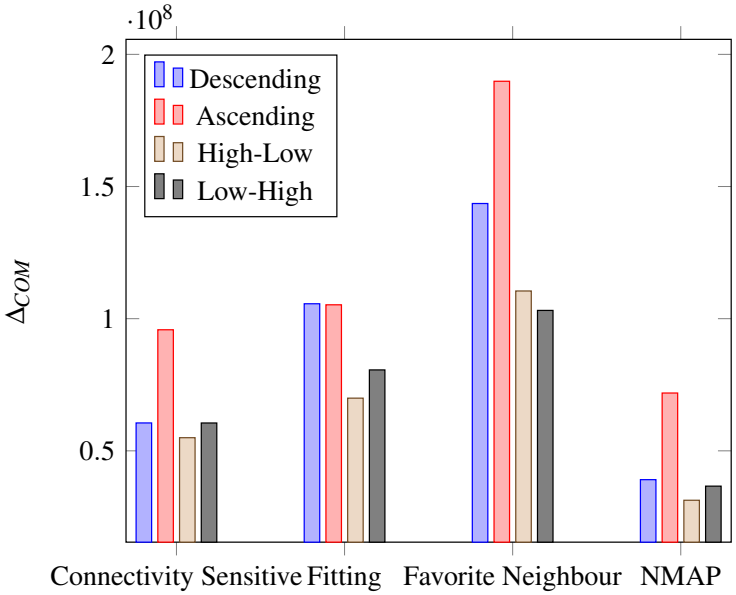


Abbildung 7.40: Platzierung von Applikationspaket 1 in unterschiedlicher Reihenfolge

Es ist zu sehen, dass die Platzierungsreihenfolgen, bei denen der größte Task-Graph zuerst platziert wird, generell besser abschneiden als die, bei denen der große Task-Graph ganz am Schluss platziert wird. Nur der *Fitting Algorithmus* erzielt hier nahezu identische Ergebnisse. Eine abwechselnde Platzierung von großen und kleinen Task-Graphen ist jedoch in den meisten Fällen vorzuziehen. Dieses Verhalten macht Sinn, da eventuelle Lücken, die durch die Platzierung großer Task-Graphen entstehen können, durch kleinere Applikationen aufgefüllt werden. Bei zusätzlicher Betrachtung defekter Komponenten verstärkt sich dieses Verhalten sogar noch (siehe Abb. 9.30 im Anhang). Es ist daher zu empfehlen, bei gleichzeitiger Platzierung mehrerer Applikationen zuerst die mit dem größten Task-Graphen abzubilden und anschließend abwechselnd Applikationen mit einer geringen und einer hohen Anzahl von Tasks.

7.2.5 Datendurchsatz

Um die Belastung des Verbindungsnetzes durch die einzelnen Algorithmen einordnen zu können, ist Δ_{COM} nicht in jedem Fall die geeignete Kennzahl. Zusätzlich ist es

wichtig im Auge zu behalten, wie viel Bandbreite die Verbindungen und die Router maximal bewältigen können müssen. Zu diesem Zweck wurde bei jeder Simulation der maximale Bandbreitenbedarf einer Verbindung im Bezug auf die vorgenommene Platzierung ermittelt. Die Kommunikation verwendet hierzu jeweils einen der kürzesten Wege. Auf Leitungen, die von mehreren Kommunikationen benutzt werden, bildet die Bandbreitenberechnung die Summe der benötigten Bandbreiten. Durch intelligente Routingalgorithmen mit Lastbalancierungsfunktionen könnte die maximal verwendete Bandbreite noch weiter reduziert werden.

Durch eine Analyse der Bandbreitenanforderungen aller Simulationen stellt sich heraus, dass die Algorithmen in fast allen Szenarien dasselbe Verhalten im Bezug auf die verbrauchte Bandbreite zeigen. Der *Konnektivitätssensitive Algorithmus* benötigt bei Fork-Join Task-Graphen pro Verbindung mehr Durchsatz als die sequentielle Platzierung, welche wiederum mehr Bandbreite als die Random-Metrik in Anspruch nimmt. Beispielhaft ist dies in Abbildung 7.41 zu sehen. Auf der Y-Achse ist diesmal nicht Δ_{COM}, sondern die benötigte Bandbreite in Byte/s abgebildet.

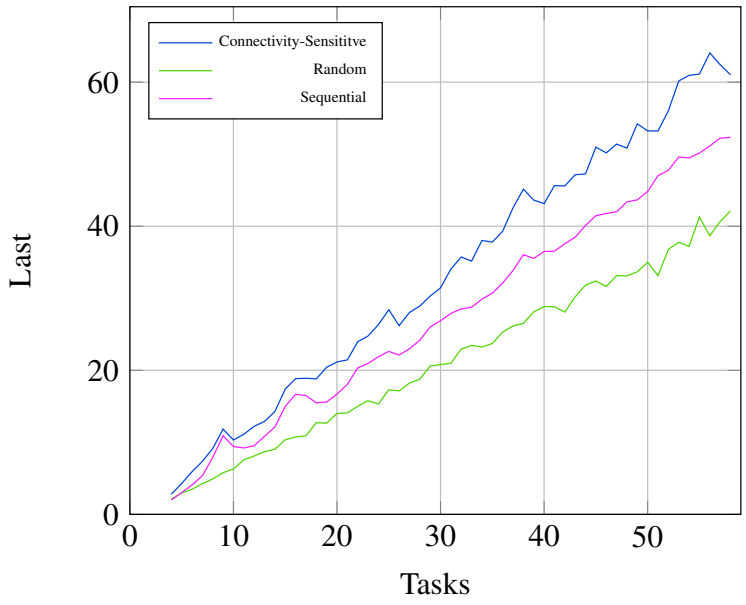


Abbildung 7.41: Fork-Join Task-Graph; 10% defekte Router; maximale Last

Das Verhalten der Graphen ist recht eingängig, da die Metriken einzelne Tasks viel weiter über den Kern-Graphen verteilen wohingegen der *Konnektivitätssensitive Algorithmus* zu einer lokal konzentrierten Platzierung führt. Interessanterweise benötigt eine sequentielle Abbildung bei den Applikationspaketen weniger Bandbreite als die Random-Metrik (siehe 9.34 im Anhang). Die Erklärung dafür ist, dass die einzelnen Applikationen des Paketes nicht miteinander kommunizieren. Somit erzeugt die Platzierung in einer Reihe weniger Kommunikationswege, über die Nachrichten mehrerer

Applikationen fließen. Dies kann die Random-Metrik nicht garantieren. Auch bei der Betrachtung von verschiedenartigen Fehlerarten im Kern-Graph verändert sich die Reihenfolge der Platzierungsmethoden bezüglich des maximalen Bandbreitenverbrauchs nicht.

Im Vergleich mit seinen Erweiterungen schneidet der *Konnektivitätssensitive Algorithmus* gut ab, wie Abbildung 7.42 zu entnehmen ist.

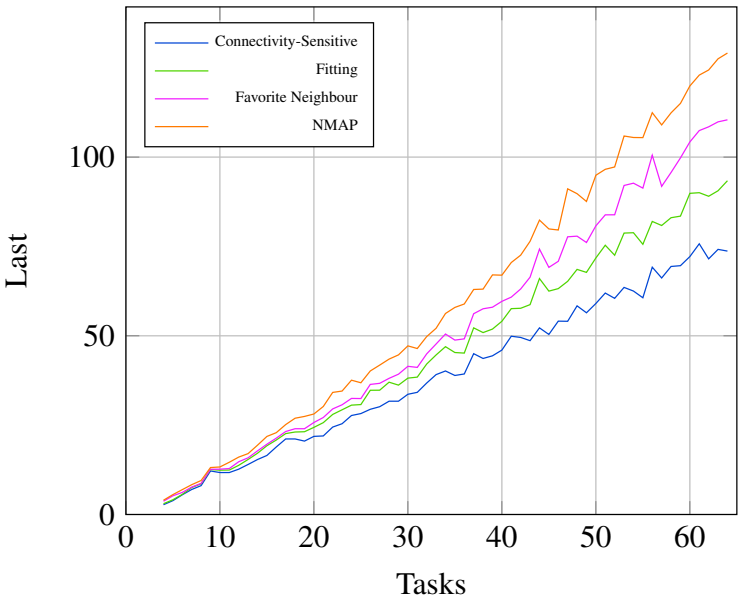


Abbildung 7.42: Fork-Join Task-Graph; 10% defekte Verbindungen; maximale Last

In allen betrachteten Simulationen ist festzustellen, dass der *Konnektivitätssensitive Algorithmus* am sparsamsten mit der Bandbreite einzelner Verbindungen umgeht. Das Schlusslicht bildet immer NMAP. In vielen Situationen ist der *Fitting Algorithmus* bei geringeren Fehlerzahlen nahezu gleichwertig zu den Ergebnissen des *Konnektivitätssensitiven Algorithmus* (siehe beispielsweise Abbildung 9.35 im Anhang). Der *Favorite Neighbour Algorithmus* ordnet sich zwischen dem *Fitting* und dem NMAP-Algorithmus ein.

Diese Beobachtungen legen nahe, dass der *Konnektivitätssensitive Algorithmus* am besten geeignet ist, um die Bandbreite einzelner Verbindungen gering zu halten. Zwar sind die Sequential- und die Random-Metrik oft schonender für die Bandbreite, allerdings ist die Qualität der Platzierung geringer, was zu längeren Kommunikationswegen führt.

7.2.6 Kern-Graphen mit Torus-Topologie

In vielen Situationen hängt die Qualität einer Platzierung stark von der zugrunde liegenden Topologie ab. Daher ist es sinnvoll, einen Vergleich der gewöhnlichen Mesh-Topologie mit einer anderen anzustellen. Zu diesem Zweck wurde hier die Torus-Topologie gewählt, die aus einem Mesh besteht, dessen Randknoten zu den Konten auf der jeweils gegenüberliegenden Seite verbunden sind. Dadurch lassen sich Kommunikationswege halbieren, da entweder die eine oder die andere Richtung gewählt werden kann. Es ist anzunehmen, dass dadurch Δ_{COM}-Werte stark reduziert werden können. Bei dieser Evaluierung wurde ein gefaltetes Torus-Netzwerk angenommen, weswegen die äußeren Verbindungen die gleiche Latenz und damit das gleiche Bewertungsgewichtung besitzen.

Im folgenden finden sich einige ausgewählte Evaluierungsergebnisse, die charakteristische Eigenschaften der Torus-Topologie im Vergleich zum herkömmlichen Mesh-Netzwerk aufzeigen.

Für die sequentielle Metrik bringt die Torus-Topologie starke Vorteile, da das Wellenmuster, dass durch den Zeilenwechsel verursacht wurde, stark abgeschwächt wird. In Abbildung 7.43 sind die Evaluierungsergebnisse in einem Torus-Netzwerk ohne fehlerhafte Elemente zu sehen.

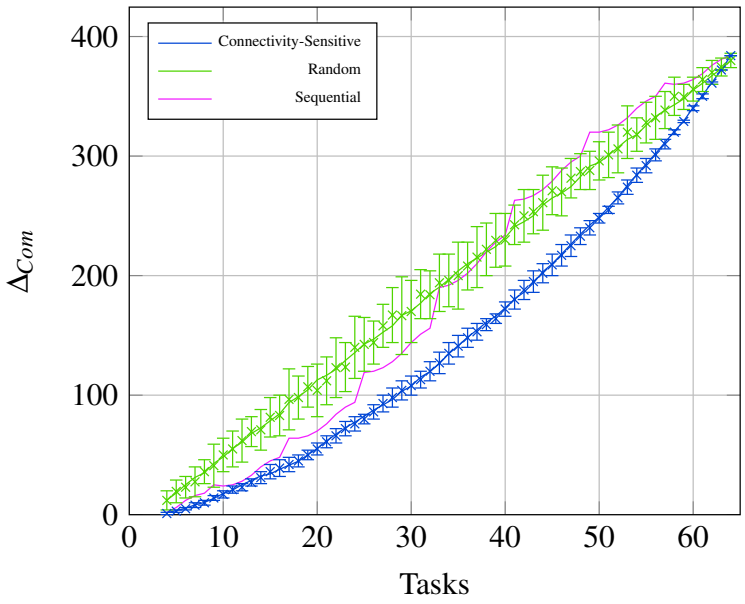


Abbildung 7.43: 8x8 Torus-Kern-Graph ohne Fehler; Fork-Join Task-Graphen

Die sequentielle Metrik weist hier nur noch ein schwaches Wellenmuster auf. Der Ausschlag ist allerdings nicht mehr so gravierend, da die Abstände allgemein kürzer werden. Auch das Gesamtergebnis der Sequential-Metrik reicht näher an die Qualität

der zufälligen Platzierung heran. Der *Konnektivitätssensitive Algorithmus* kann allerdings immer noch die besseren Ergebnisse liefern. Im Vergleich zum Mesh-Netzwerk sind die Δ_{COM}-Werte erwartungsgemäß gesunken.

Die Betrachtung von fehlerhaften Elementen in Torus-Netzwerken lässt erkennen, dass Verbindungs- und Routerfehler einen wesentlich geringeren Einfluss auf die Platzierungsqualität haben. Vor allem Situationen, in denen ab einer gewissen Fehlerrate keine sinnvolle Platzierung mehr möglich war, bleiben viel länger benutzbar. Als typisches Beispiel dient hier die Auswertung der Platzierung eines VOP-Decoder Task-Graphen mit defekter Routerhardware (Abb. 7.44).

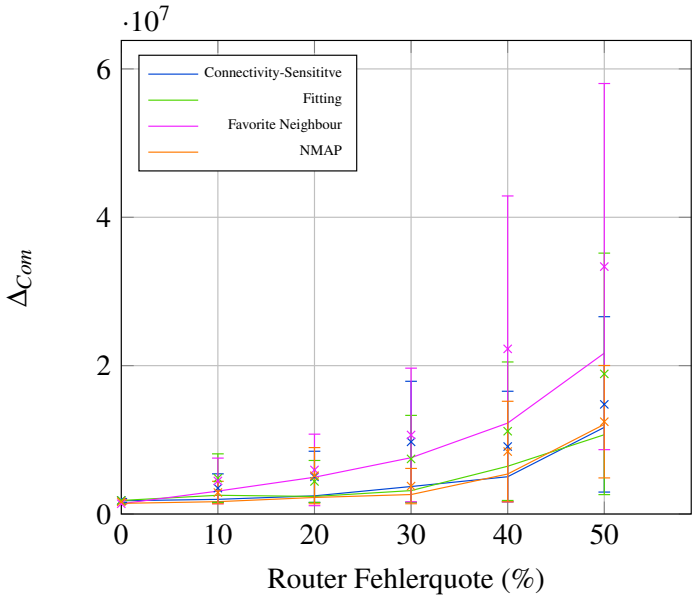


Abbildung 7.44: 8x8 Torus-Kern-Graph mit ausgefallenen Kerne; VOP-Task-Graph

Besonders werden durch eine Torus-Topologie der *Fitting Algorithmus* und der NMAP-Algorithmus begünstigt. Bei gehäuften Fehlern im Verbindungsnetz erzielen diese wesentlich bessere Ergebnisse und sind damit eher zu empfehlen als der *Konnektivitätssensitive Algorithmus*. Beim *Fitting Algorithmus* liegt das bessere Ergebnis an der geringeren Anzahl der Sackgassen. Wurde bei einem Mesh-Netzwerk eine Task an den Rand platziert, gab es nur begrenzte Wachstumsmöglichkeiten. Dies bereitet bei größeren Task-Graphen oftmals Probleme, weshalb der *Fitting Algorithmus* dort auch meist schlechtere Platzierungen als der *Konnektivitätssensitive Algorithmus* erzeugt. In einem Torus-Netz platziert der *Fitting Algorithmus* Tasks oftmals in der Nähe bereits platzierter anderer Tasks. Da dies unter Anderem auch Tasks aus dem gerade zu platzierenden Task-Graphen sind, wird eine kompaktere Platzierung und damit auch ein besserer Δ_{COM}-Wert erzielt. *Fitting* profitiert somit von der Torus-Topologie.

Diese Vorteile gelten vor allem beim Auftreten von Defekten, die Verbindungsstruk-

turen beeinflussen. Auswertungen bei fehlerhaften Kernen hingegen zeigen kaum ein verändertes Verhalten. Allerdings werden auch hier geringere Δ_{COM}-Werte erreicht.

Abschließend ist festzustellen, dass die Torus-Topologie durch die durchschnittlich geringeren Kommunikationsdistanzen Platzierungen mit niedrigeren Δ_{COM}-Werten erzielt. Das grundsätzliche Verhalten der verschiedenen Platzierungsalgorithmen bleibt weitestgehend identisch, allerdings wird der *Fitting Algorithmus* bei defekten Verbindungen oder Routern besser als der *Konnektivitätssensitive Algorithmus*. Auch NMAP erzeugt bei hohen Fehleranzahlen länger gültige Platzierungen, deren Qualität sich allerdings analog zu Mesh-Netzen verhält.

Die Erkenntnisse der Mesh-Netze lassen sich daher in den meisten Fällen übertragen.

7.3 Bewertung der Ergebnisse

Zusammenfassend ist zu sehen, dass der *Konnektivitätssensitive Algorithmus* das Platzierungsproblem in den betrachteten Szenarien am besten löst. Zusätzlich muss weniger Rechenzeit als beim NMAP-Algorithmus aufgewendet werden, was für den betrachteten Anwendungsfall essentiell ist.

Leider konnten die Erweiterungen des Algorithmus (*Fitting* und *Favorite Neighbour*) die in sie gesetzten Erwartungen nicht erfüllen. *Favorite Neighbour* produziert in fast allen Fällen die schlechteste Platzierung. Daher ist davon abzusehen, diesen Algorithmus einzusetzen.

Fitting wird bei Benutzung einer Torus-Topologie für den Kern-Graphen besser. Für Torus-Netze ist der *Fitting Algorithmus* daher gleichwertig zum *Konnektivitätssensitiven Algorithmus*. In einigen Fällen können bei gleichem Aufwand sogar bessere Ergebnisse erzielt werden.

	Konnektivitätssensitiv	*Fitting*	*Favorite Neighbour*	*NMAP*	*Sequential*	*Random*
Fehlerfrei	++	+	-	+	–	–
Fehlerhafte Router	++	+	-	0	–	–
Fehlerhafte Links	++	+	-	0	–	–
Fehlerhafte Kerne	++	+	-	++	–	-
Datendurchsatz	+	0	-	–	+	++
Torus	++	++	0	++	0	-
Laufzeit	+	+	+	-	++	++
Fehlerresistenz	++	+	-	–	0	–

Tabelle 7.7: Bewertung der Platzierungsverfahren

Der NMAP-Algorithmus, der in diesem Kapitel als Vergleichsmetrik diente, kann im Allgemeinen gute Platzierungen produzieren. Er benötigt dafür allerdings mehr Rechenaufwand. Außerdem können mit NMAP in Szenarien mit hohen Fehlerzahlen oft keine gültigen Platzierungen gefunden werden.

Die einfachen Vergleichsmetriken Random und Sequential benötigen zwar kaum Aufwand für die Platzierungsentscheidung, produzieren danach allerdings auch wesentlich schlechtere Δ_{COM}-Werte als die komplizierteren Algorithmen. Vor allem in fehlerhaften Kern-Graphen sind diese beiden Metriken ungeeignet.

Eine oberflächliche Zusammenfassung der Ergebnisse ist in Tabelle 7.7 zu sehen. Die Bewertungen sind nicht als absolute Angaben zu verstehen, da die erzielten Ergebnisse der Algorithmen in der Praxis von vielen Randbedingungen abhängen. Jedoch bieten die Angaben in der Tabelle einen guten Einstiegspunkt um konkrete Ergebnisse im Text nachzuvollziehen.

8 Schluss

Nach einer Zusammenfassung des Inhalts dieser Arbeit mit eine Betrachtung der erzielten Ergebnisse ist ein abschließender Abschnitt den Erweiterungsmöglichkeiten der vorliegenden Forschung gewidmet.

8.1 Zusammenfassung und Abschlussbetrachtung

Die vorliegende Arbeit beschreibt und analysiert Algorithmen zur Platzierung von Applikationen, bestehend aus durch Kommunikationsbeziehungen verbundenen Tasks, auf Many-Core-Prozessoren. Die betrachteten Algorithmen legen besonderen Wert auf eine kurze Laufzeit und eine hohe Toleranz gegenüber fehlerhaften Komponenten des Prozessors.

Kapitel 2 behandelt die notwendigen Grundlagen für die Platzierungsalgorithmen. Dies beinhaltet eine Klassifizierung der verschiedenen Hardwarekomponenten eines Many-Core-Prozessors, welche auf ihre Eignung für das vorliegende Problem geprüft werden. Es kommt zu der Schlussfolgerung, dass sich eine Many-Core-Architektur mit homogenen Prozessoren, verteiltem Speicher und einem Network on Chip am besten für die Aufgabenstellung eignet. Das Network on Chip sollte mit adaptiven Routern ausgestattet sein, um auf auftretende Fehler reagieren zu können. Zusätzlich werden Arten der Parallelisierung und relevante Methoden der Parallelprogrammierung beleuchtet. Die Programmiermethodik, welche am besten zu der gewählten Hardwarearchitektur passt, ist das Message-Passing-Paradigma.

Im dritten Kapitel wird das Fehlermodell und das Systemmodell der vorliegenden Arbeit erläutert. Zusätzlich beinhaltet es eine Beschreibung der Applikationen, welche in Kapitel 7 zur Evaluierung zum Einsatz kommen. Das Fehlermodell geht davon aus, dass Fehler durch geeignete Hardwaremechanismen erkannt und der Platzierungseinheit gemeldet werden. Dabei gehen drei unterschiedliche Fehlerorte in die Betrachtung ein. Auftretende Fehler in einem Kern, einem Router oder in einer Verbindung gelten hierbei als permanenter Defekt. Es werden außerdem gängige Methoden der Fehlererkennung beschrieben, welche in der Industrie und der Wissenschaft derzeit üblich sind. Das Systemmodell besteht aus den Abstraktionen der Task-Graphen und der Kern-Graphen. Diese Graphen ermöglichen es den Platzierungsalgorithmen, mit denen für sie relevanten Informationen zu arbeiten. Task-Graphen bestehen aus Knoten und gerichteten, gewichteten Kanten wohingegen Kern-Graphen ungerichtete und ungewichtete Kanten besitzen. Es werden außerdem die verwendeten Benchmark-Applikationen, welche aus verschiedenen Benchmark-Sammlungen und Veröffentlichungen zusammengestellt sind, beschrieben. Dies gibt einen Einblick in die Charakteristiken der einzelnen

Graphen und ermöglicht somit eine bessere Bewertung der erzielten Evaluierungsergebnisse.

Das 4. Kapitel beinhaltet eine Übersicht über verwandte Arbeiten aus der Literatur. Es werden sowohl Algorithmen betrachtet, die Task-Platzierung auf Prozessorclustern durchführen, als auch Platzierungsmethoden für traditionelle Supercomputer und Rechencluster. Dieses Kapitel beschreibt auch den NMAP-Algorithmus im Detail, welcher im Evaluierungskapitel als Maßstab für den aktuellen Stand der Technik dient.

In Kapitel 5 wird der *Konnektivitätssensitive Algorithmus* für die statische Platzierung von Task-Graphen vorgestellt. Sein Ziel ist es, unter Betrachtung minimaler Kommunikationskosten, die Abbildung eines Task-Graphen auf einen Kern-Graphen vorzunehmen. Es wird sowohl auf die Funktionsweise des Algorithmus als auch auf sein Laufzeitverhalten eingegangen. Anschließend kommen zwei Erweiterungen des Algorithmus zur Diskussion, welche in speziellen Situationen bessere Ergebnisse liefern sollen. Der *Fitting Algorithmus* achtet bei der Platzierung auf eine genau passende Anzahl von Nachbarn. Damit erzeugt er in der Theorie kompaktere Platzierungen als der *Konnektivitätssensitive Algorithmus*, was bei der Ausführung mehrerer Applikationen auf einem Prozessor zu besseren Ergebnissen führen soll. Sein Laufzeitverhalten ist auch im Vergleich zu dem des *Konnektivitätssensitiven Algorithmus* identisch. Desweiteren wird der *Favorite Neighbour Algorithmus* vorgestellt, welcher speziell auf hoch gewichtete Kommunikationsbeziehungen eingeht. Seine Laufzeit ist ebenfalls identisch zu der des *Konnektivitätssensitiven* Platzierungsverfahrens und er soll bei Task-Graphen mit stark unterschiedlicher Gewichtung der Kommunikationsverbindungen bessere Ergebnisse erzielen.

Kapitel 6 beschäftigt sich mit der dynamischen Task-Platzierung. Dies bezeichnet eine Veränderung der Position einer oder mehrerer Tasks zur Laufzeit der Applikation. Zuerst wird auf die nötigen Voraussetzungen eingegangen, da Mechanismen für Überwachung, Optimierung und Migration zur Unterstützung der dynamischen Platzierungsverfahren existieren müssen. Anschließend wird die Anpassung der Position einzelner Tasks und die Replatzierung von Task-Graphen zur Laufzeit beschrieben. Das System der mehrstufigen Platzierung, bei der zuerst statisch mit einem einfachen und schnellen Algorithmus platziert und anschließend durch dynamische Optimierung das Ergebnis verbessert wird, ist am Ende des Kapitels beschrieben.

Kapitel 7 widmet sich der ausgiebigen Evaluierung der Platzierungsverfahren. Es werden sowohl synthetische Task-Graphen als auch solche, die aus realen Applikationen gewonnen wurden, unter verschiedenen Bedingungen betrachtet. Für Kern-Graphen wurde sowohl die Mesh- als auch die Torus-Topologie eingesetzt. Das Kapitel beginnt mit einem Vergleich des *Konnektivitätssensitiven Algorithmus* mit zwei einfachen Vergleichsmetriken im fehlerfreien Fall. Anschließend werden diese Ergebnisse in Relation zu denen der anderen Algorithmen und des NMAP-Algorithmus, als Vertreter der bereits existierenden Verfahren, gestellt. Den Hauptteil des Kapitels nimmt die Analyse des Verhaltens der Algorithmen in Situationen ein, in welchen fehlerhafte Komponenten im Kern-Graph in verschiedenen Ausprägungen auftreten. Vor den Schlussfolgerungen bezüglich ihrer Kombination ist hier jede Fehlerart des verwendeten Fehlermodells einzeln betrachtet. Es wird außerdem auf den zu erwartenden maximalen Datendurchsatz eingegangen, was dabei hilft, das Network on Chip

zu dimensionieren. Die Auswertung aller Ergebnisse führt zu dem Schluss, dass der *Konnektivitätssensitive Algorithmus* die besten Platzierungen produziert. Seine beiden Erweiterungen, der *Fitting Algorithmus* und der *Favorite Neighbour Algorithmus*, ergeben keine Verbesserung. Der NMAP-Algorithmus erzeugt bei höherem Rechenaufwand gute Platzierungen, ist allerdings nicht geeignet, auf Kern-Graphen mit hohen Fehlerzahlen zu arbeiten.

Der *Konnektivitätssensitive Algorithmus* stellt abschließend betrachtet einen guten Kompromiss zwischen fehlertoleranter Platzierungsqualität und Komplexität dar. Eine wichtige Eigenschaft ist seine Fähigkeit, auch bei hohen Fehlerwahrscheinlichkeiten gültige und den Umständen entsprechend performante Task-Platzierungen zu erzeugen. Die Auswertungsergebnisse haben gezeigt, dass das verwendete Platzierungsverfahren einen starken Einfluss auf die Performance der ausgeführten Applikation hat. Daher sollte bei der Auswahl eines Platzierungsverfahrens genau abgewägt werden, welche Architektur und Anwendungen zum Einsatz kommen, um abgestimmte Platzierungsalgorithmen einsetzen zu können.

8.2 Ideen für zukünftige Forschung

Die Ergebnisse dieser Arbeit bieten Ansatzpunkte für mehrere Erweiterungen.

Eine mögliche Erweiterung der entwickelten Algorithmen könnte die Unterstützung von heterogenen Many-Core-Prozessoren sein. Dies beinhaltet sowohl die Behandlung unterschiedlicher Fähigkeiten der einzelnen Kerne eines Kern-Graphen als auch die Betrachtung von speziellen Anforderungen der auszuführenden Tasks. Ein Beispiel dafür stellen aktuelle Supercomputer dar, die teilweise mit GPU-Beschleunigung arbeiten, diese aber nicht auf allen Rechenknoten zur Verfügung stellen. Ebenfalls eine Art von Heterogenität bietet ein Many-Core-Design, welches nur von Kernen am Rand des Prozessors einen Zugriff auf externen Speicher und Eingabe-/Ausgabe-Geräte zulässt. Die Platzierungsentscheidung muss einen Abgleich zwischen Anforderungen und Fähigkeiten durchführen. Es können auch weiche Anforderungen enthalten sein, welche bei Nichterfüllung nur eine Verschlechterung der Performance darstellen würden.

Durch eine weitere Anpassung der Algorithmen ließe sich eine fehlerresistente Platzierung erzeugen. Die große Anzahl an Kernen in einem Many-Core-Prozessor ermöglicht eine direkt vom Algorithmus erzwungene Mehrfachplatzierung der Task-Graphen. Dabei wäre zu beachten, dass die beiden Task-Graph-Kopien in unterschiedlichen Gegenden des Prozessors platziert werden, um den möglichen Ausfall von Komponenten an einer Stelle ausgleichen zu können.

Ein weiteres Forschungsthema ist die Erweiterung der Analyse der Effekte dynamischer Platzierungen. Dies kann sogar so weit gehen, eine zentrale Platzierungseinheit gänzlich zu entfernen. Dezentrale Platzierungsentscheidungen würden die Skalierbarkeit des Many-Core-Prozessors weiter verbessern, da ein bedeutender Flaschenhals nicht mehr vorhanden ist. Dies wirft allerdings viele weitere Fragen auf, die sich damit beschäftigen, wie komplex die einzelnen Kerne dadurch zu gestalten sind, wie Daten

und Programmcode im Prozessor zu verteilen ist und wie das Wissen über defekte Komponenten sinnvoll kommuniziert wird.

Die automatisierte Gewinnung der Task-Graphen ist eine bisher ungelöste Herausforderung für die Softwareentwicklung. Derzeit müssen Task-Graphen von einem Programmierer händisch spezifiziert werden, was umständlich und fehleranfällig ist. Eine bessere Tool-Unterstützung und eine Integration in Programmiermodelle für Many-Core-Prozessoren wäre ein möglicher Lösungsansatz. Die nachrichtengekoppelte Programmierung bildet Kommunikationen und Kommunikationspartner bereits explizit ab. Speicherkopplung erfordert einen höheren Aufwand. Die benötigten Informationen lassen sich in diesem Fall durch statische Analyse und Annotationen des Quellcodes gewinnen. Zusätzlich muss in jedem Fall die Gewichtung der Kommunikation bestimmt werden. Dies bietet viele Herausforderungen, die die Frage nach der Gewinnung der Task-Graphen zu einem interessanten Forschungsthema machen.

9 Anhang

zusätzliche Evaluationsergebnisse und Task-Graphen

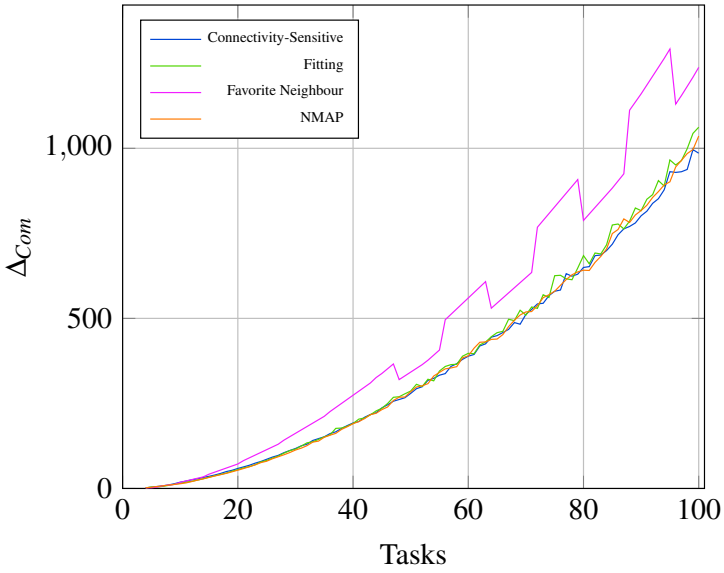


Abbildung 9.1: Fehlerfreier 10x10 Kern-Graph; Fork-Join Task-Graphen

Algorithmus	**ØΔ_{COM}**	**min(Δ_{COM})**	**max(Δ_{COM})**
Connectivity-Sensitive	18.365.194	201.728	79.141.888
Fitting	50.400.256		
Favorite Neighbour	28.082.176		
NMAP	30.176.542	403.456	110.026.752

Tabelle 9.1: Multimedia Applikation; Fehlerfreier 8x8 Kern-Graph

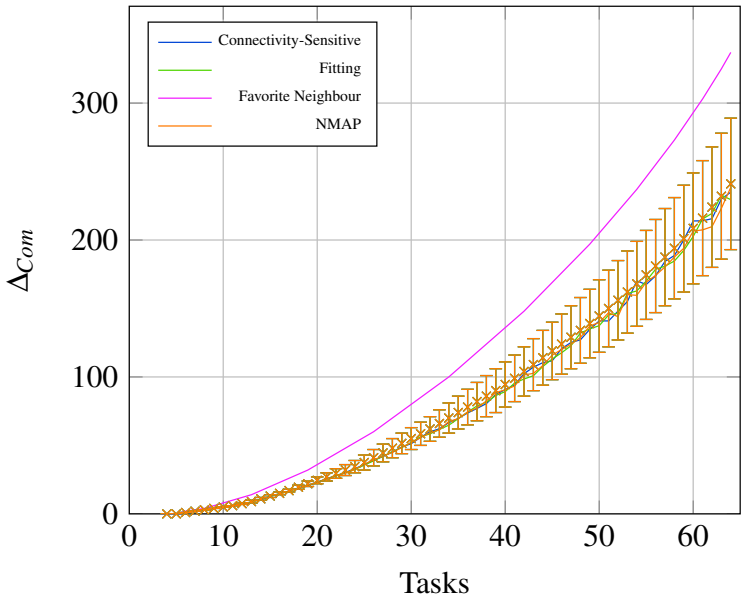


Abbildung 9.2: Fehlerfreier 8x8 Kern-Graph; einseitige Fork-Join Task-Graphen

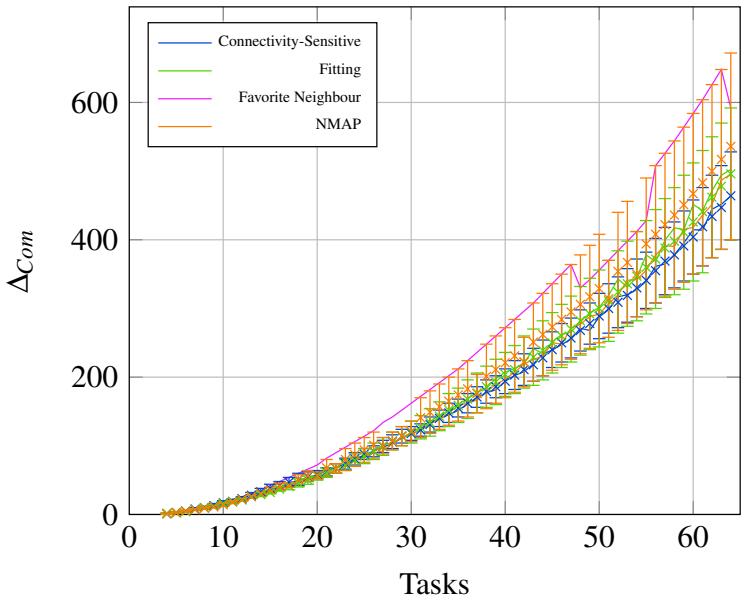


Abbildung 9.3: Fehlerfreier 8x8 Kern-Graph; Fork-Join Task-Graphen

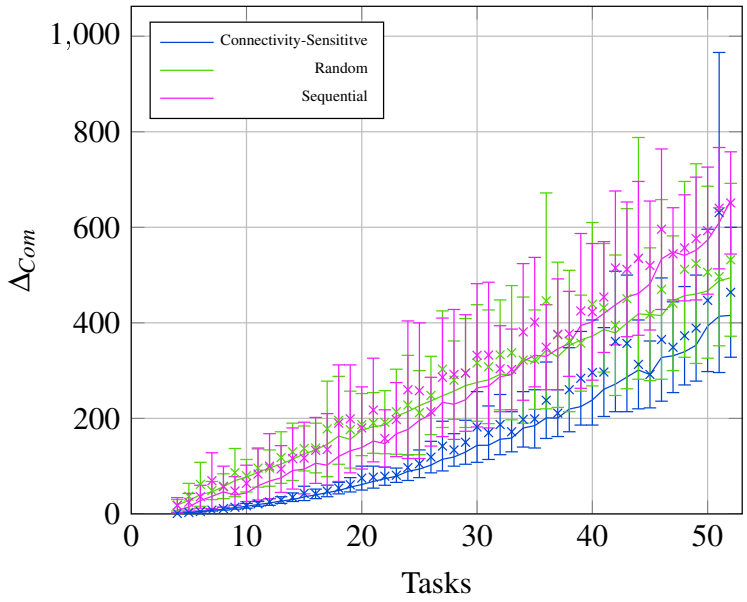


Abbildung 9.4: 8x8 Kern-Graph mit 20% ausgefallenen Routern; Fork-Join Task-Graphen

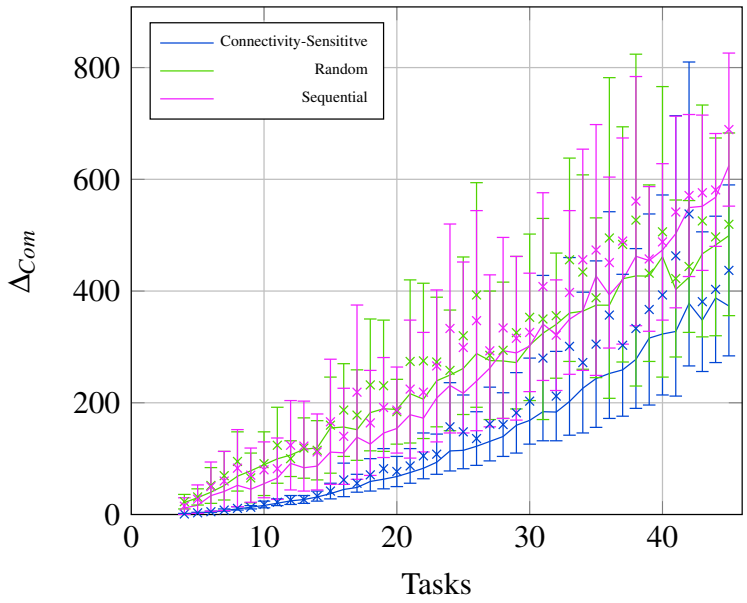


Abbildung 9.5: 8x8 Kern-Graph mit 30% ausgefallenen Routern; Fork-Join Task-Graphen

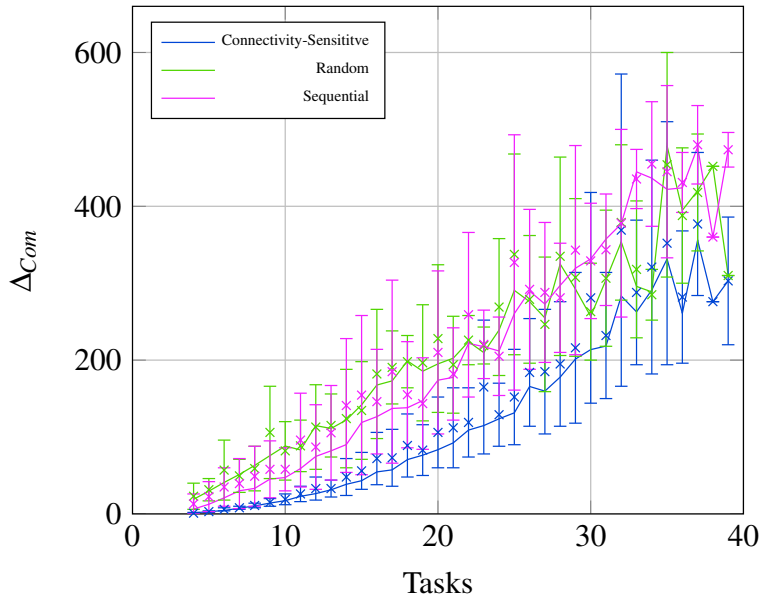


Abbildung 9.6: 8x8 Kern-Graph mit 40% ausgefallenen Routern; Fork-Join Task-Graphen

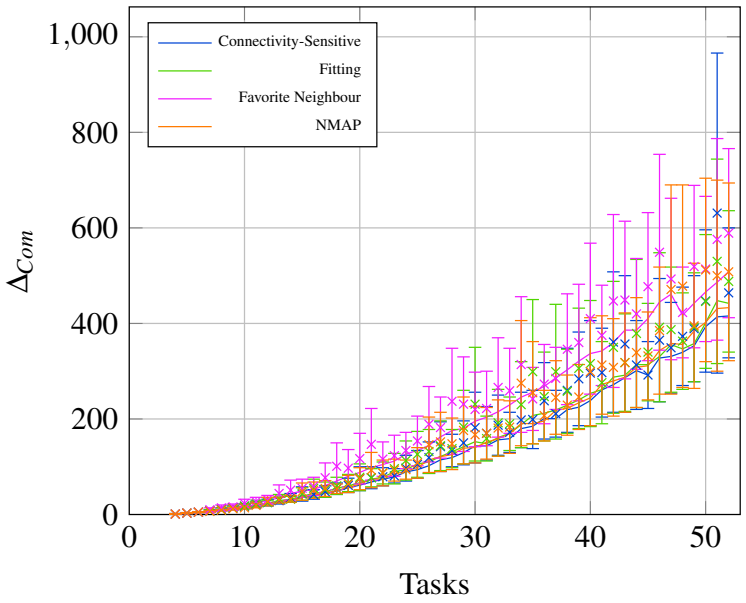


Abbildung 9.7: 8x8 Kern-Graph mit 20% ausgefallenen Routern; Fork-Join Task-Graphen

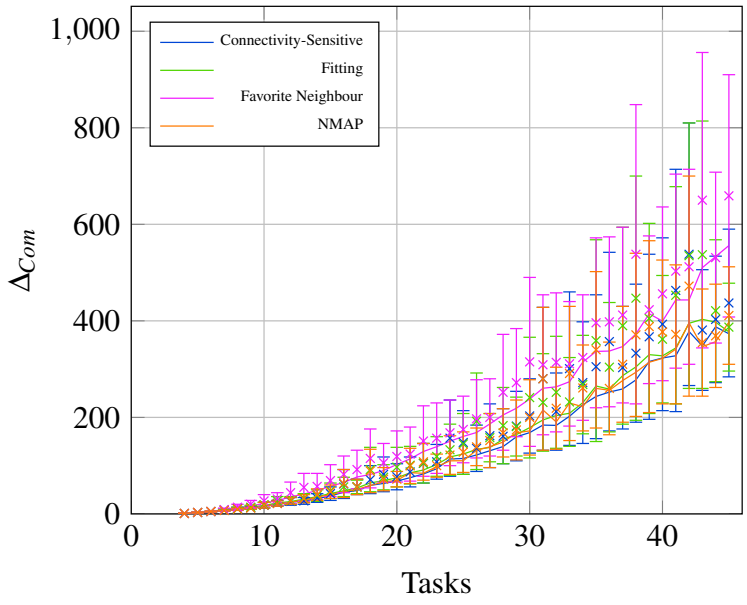


Abbildung 9.8: 8x8 Kern-Graph mit 30% ausgefallenen Routern; Fork-Join Task-Graphen

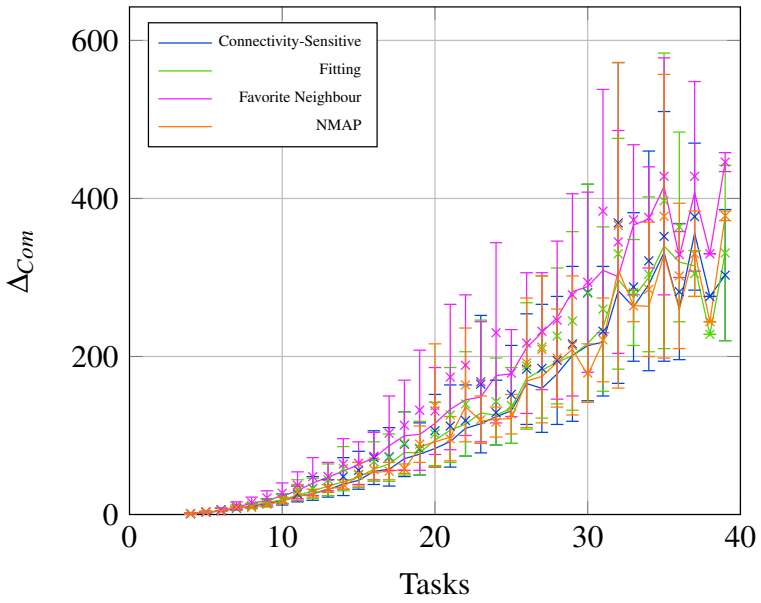
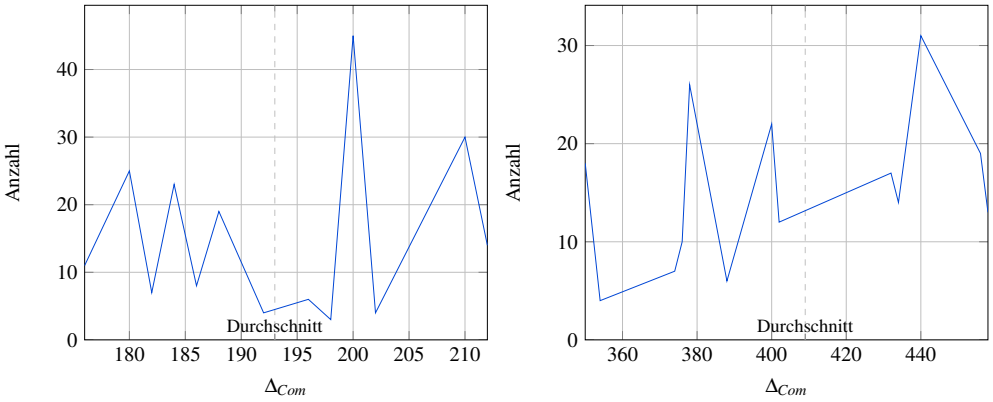
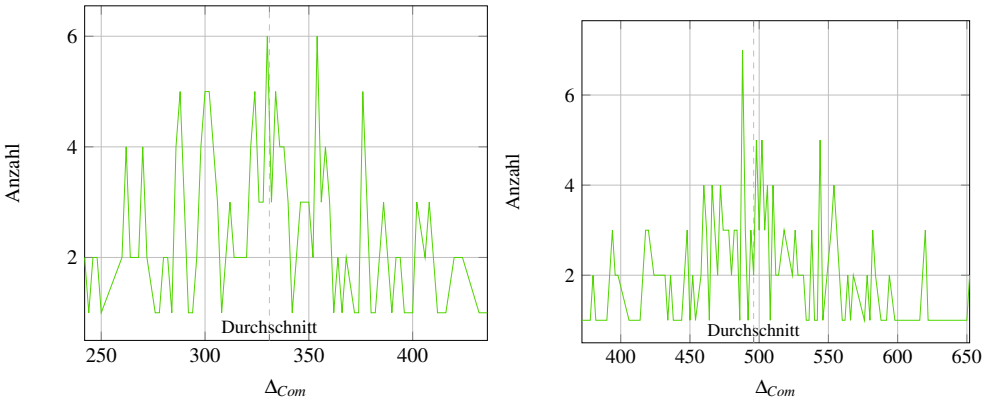


Abbildung 9.9: 8x8 Kern-Graph mit 40% ausgefallenen Routern; Fork-Join Task-Graphen



(a) 40 Tasks (b) 60 Tasks

Abbildung 9.10: Häufigkeit der Messergebnisse (*Konnektivitätssensitiver Algorithmus*); 200 Iterationen



(a) 40 Tasks (b) 60 Tasks

Abbildung 9.11: Häufigkeit der Messergebnisse (Random-Metrik); 200 Iterationen

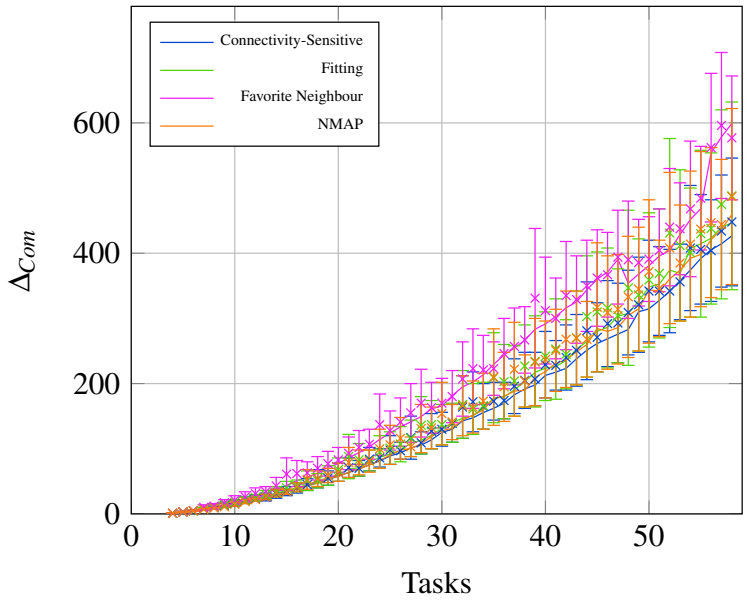


Abbildung 9.12: 8x8 Kern-Graph mit 10% ausgefallenen Routern; Fork-Join Task-Graphen

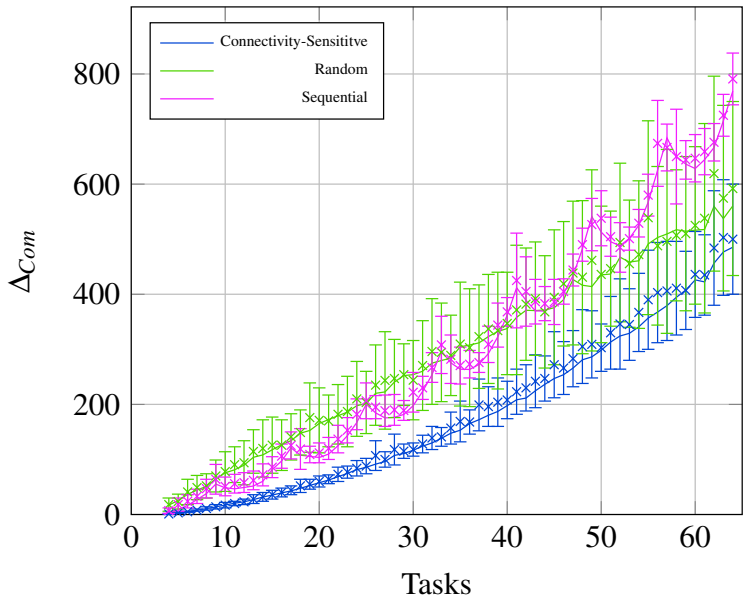


Abbildung 9.13: 8x8 Kern-Graph mit 10% ausgefallenen Verbindungen; Fork-Join Task-Graphen

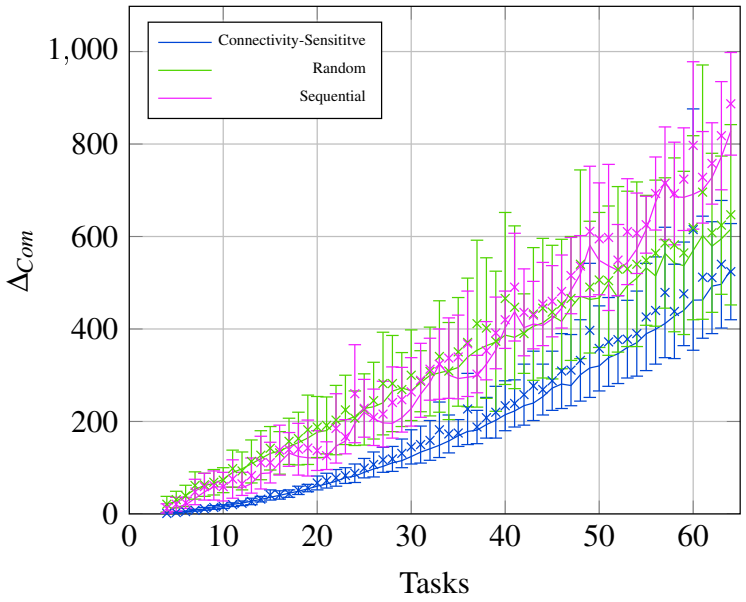


Abbildung 9.14: 8x8 Kern-Graph mit 20% ausgefallenen Verbindungen; Fork-Join Task-Graphen

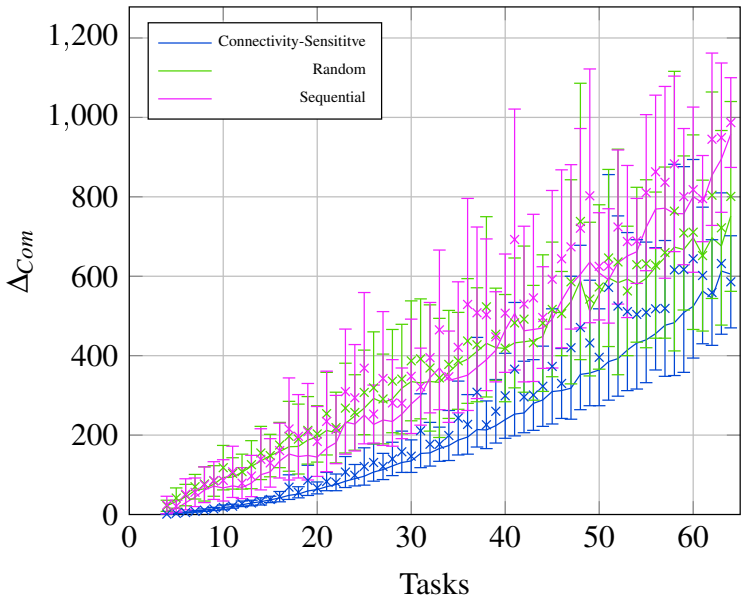


Abbildung 9.15: 8x8 Kern-Graph mit 30% ausgefallenen Verbindungen; Fork-Join Task-Graphen

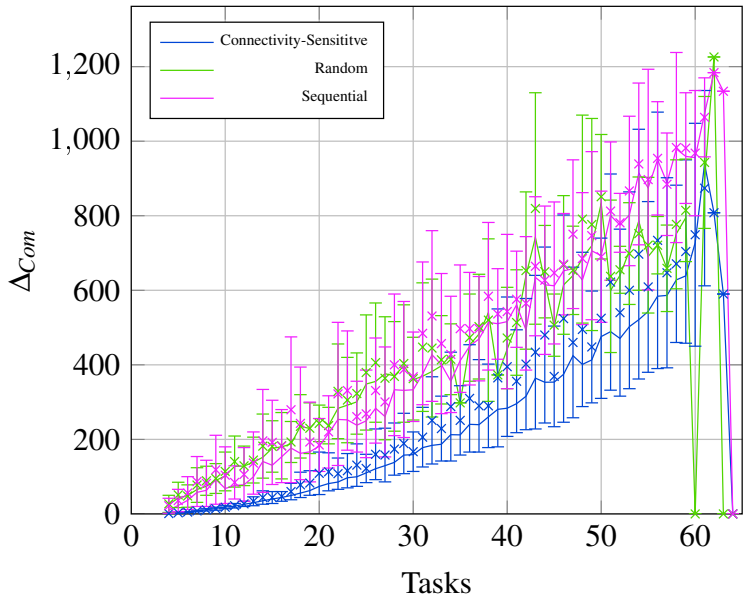


Abbildung 9.16: 8x8 Kern-Graph mit 40% ausgefallenen Verbindungen; Fork-Join Task-Graphen

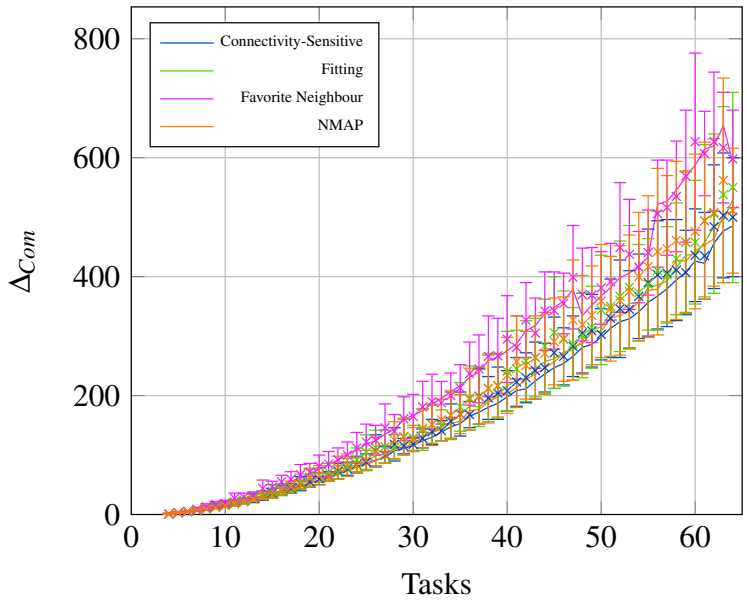


Abbildung 9.17: 8x8 Kern-Graph mit 10% ausgefallenen Verbindungen; Fork-Join Task-Graphen

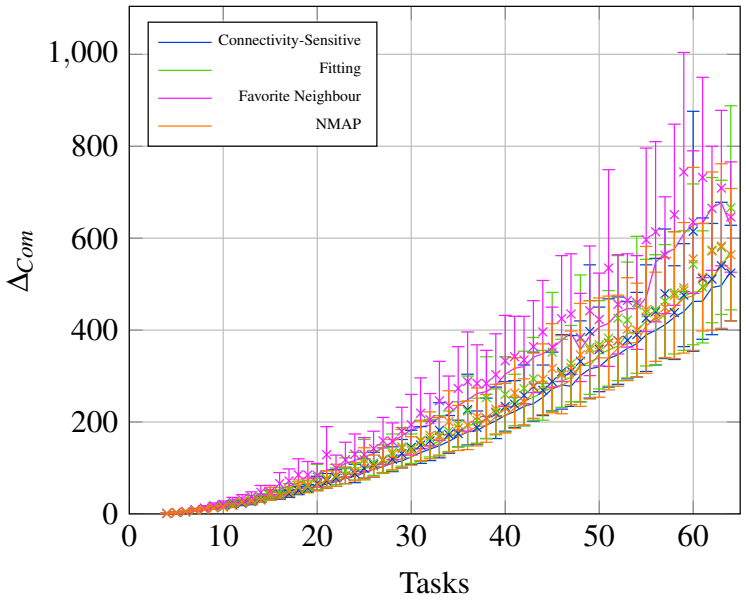


Abbildung 9.18: 8x8 Kern-Graph mit 20% ausgefallenen Verbindungen; Fork-Join Task-Graphen

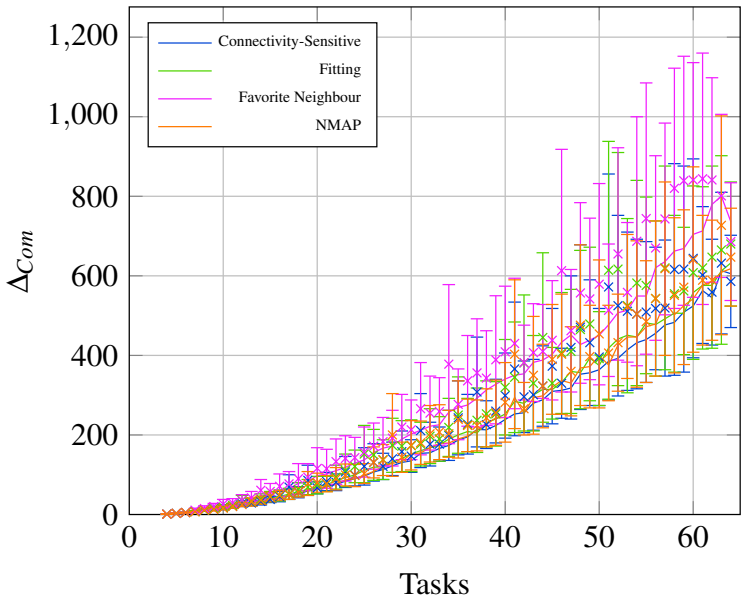


Abbildung 9.19: 8x8 Kern-Graph mit 30% ausgefallenen Verbindungen; Fork-Join Task-Graphen

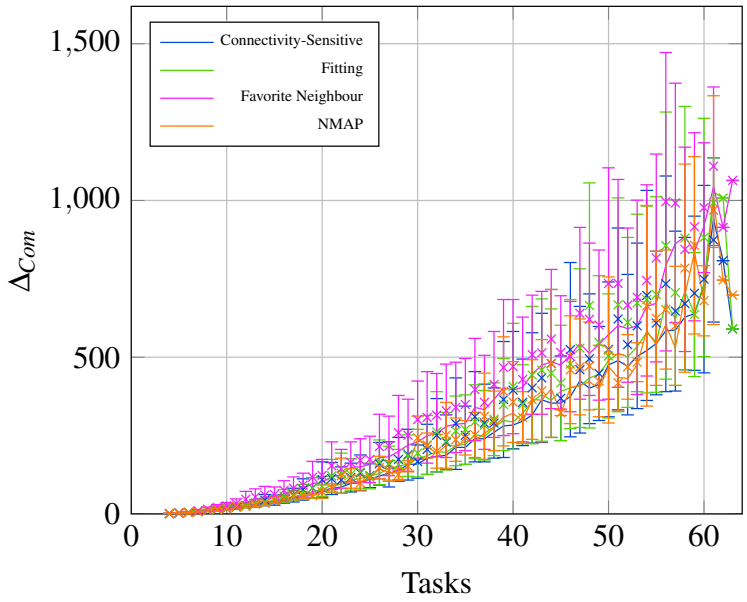


Abbildung 9.20: 8x8 Kern-Graph mit 40% ausgefallenen Verbindungen; Fork-Join Task-Graphen

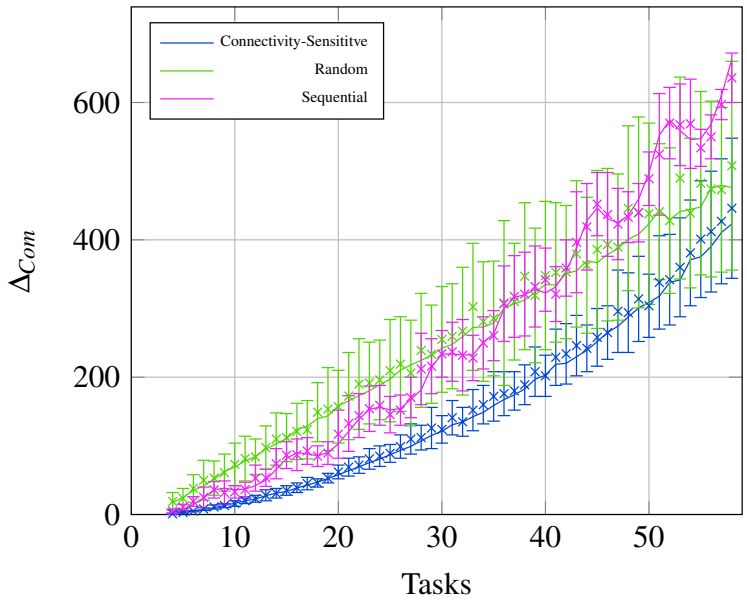


Abbildung 9.21: 8x8 Kern-Graph mit 10% ausgefallenen Kernen; Fork-Join Task-Graphen

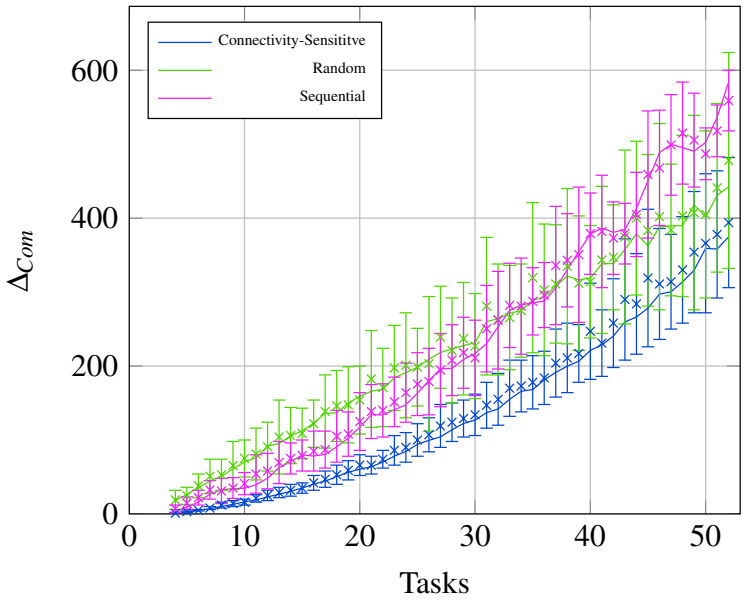


Abbildung 9.22: 8x8 Kern-Graph mit 20% ausgefallenen Kernen; Fork-Join Task-Graphen

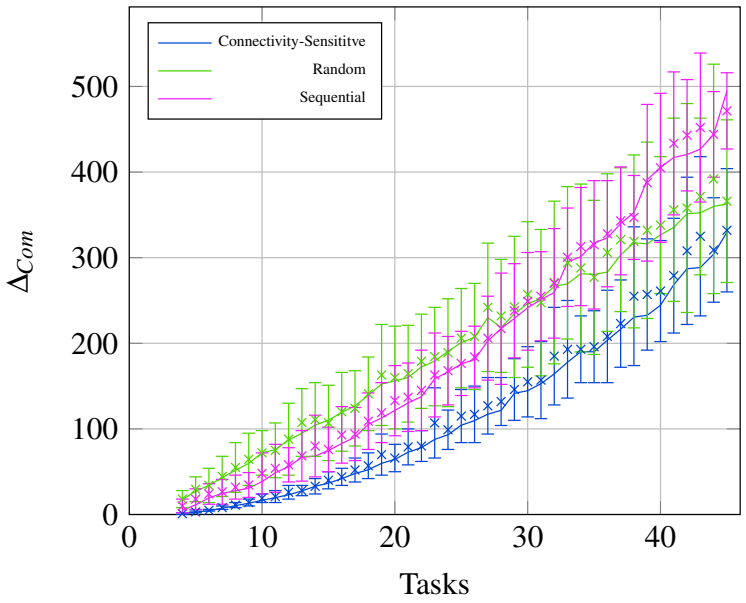


Abbildung 9.23: 8x8 Kern-Graph mit 30% ausgefallenen Kernen; Fork-Join Task-Graphen

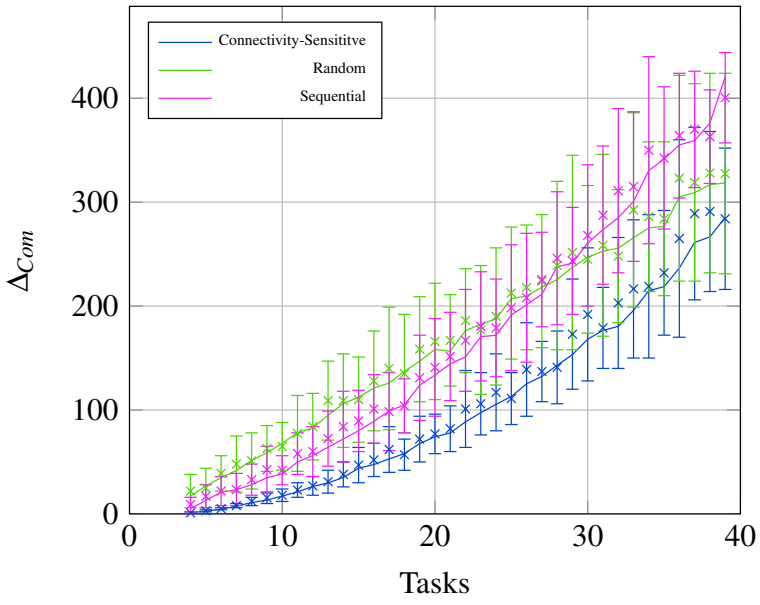


Abbildung 9.24: 8x8 Kern-Graph mit 40% ausgefallenen Kernen; Fork-Join Task-Graphen

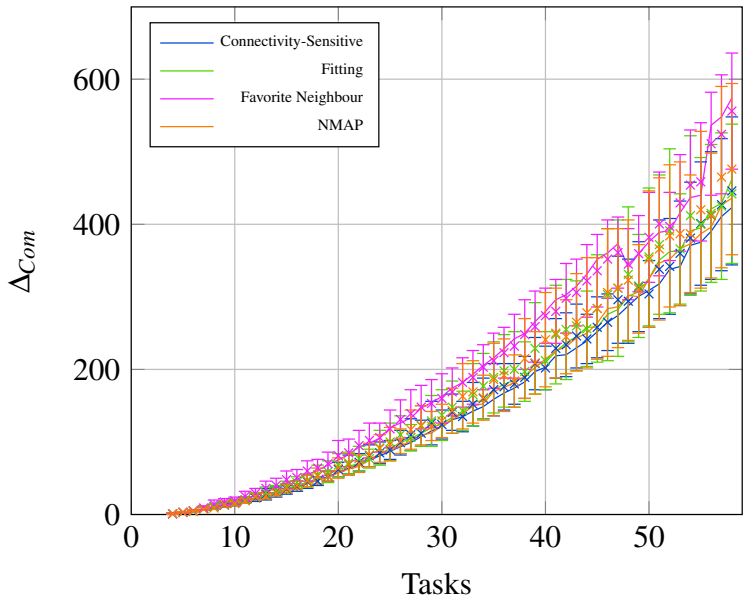


Abbildung 9.25: 8x8 Kern-Graph mit 10% ausgefallenen Kerne; Fork-Join Task-Graphen

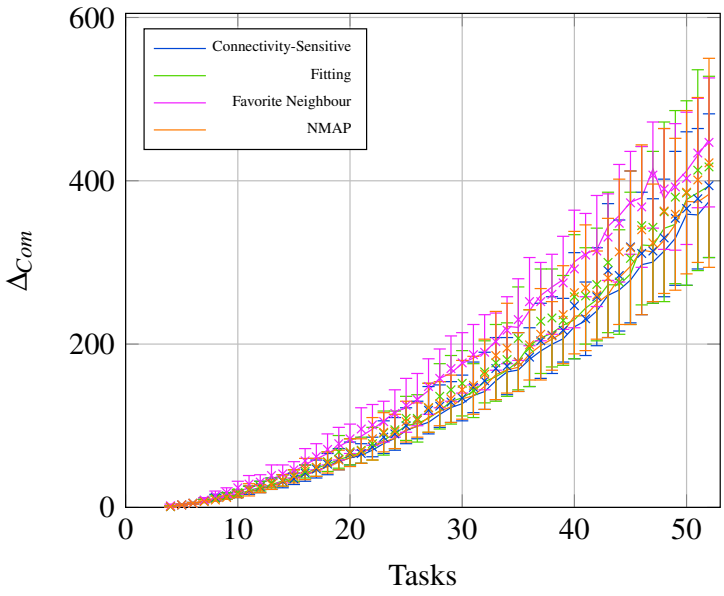


Abbildung 9.26: 8x8 Kern-Graph mit 20% ausgefallenen Kerne; Fork-Join Task-Graphen

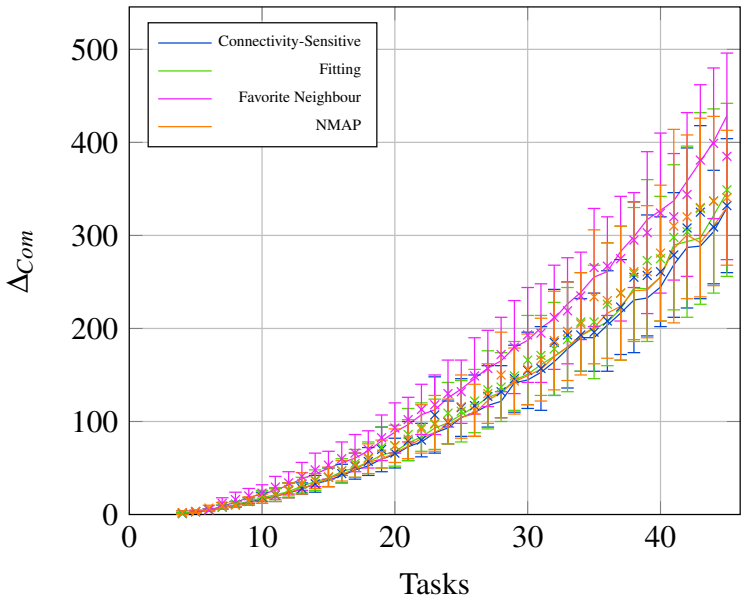


Abbildung 9.27: 8x8 Kern-Graph mit 30% ausgefallenen Kerne; Fork-Join Task-Graphen

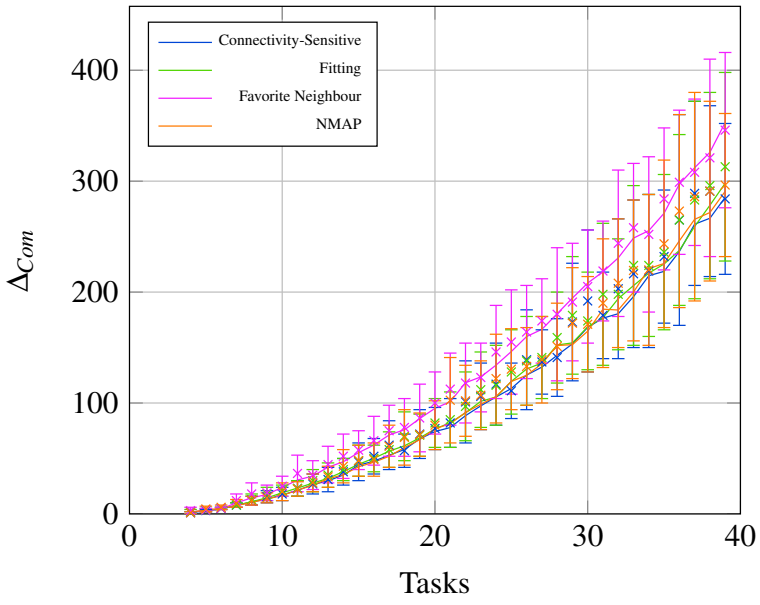


Abbildung 9.28: 8x8 Kern-Graph mit 40% ausgefallenen Kerne; Fork-Join Task-Graphen

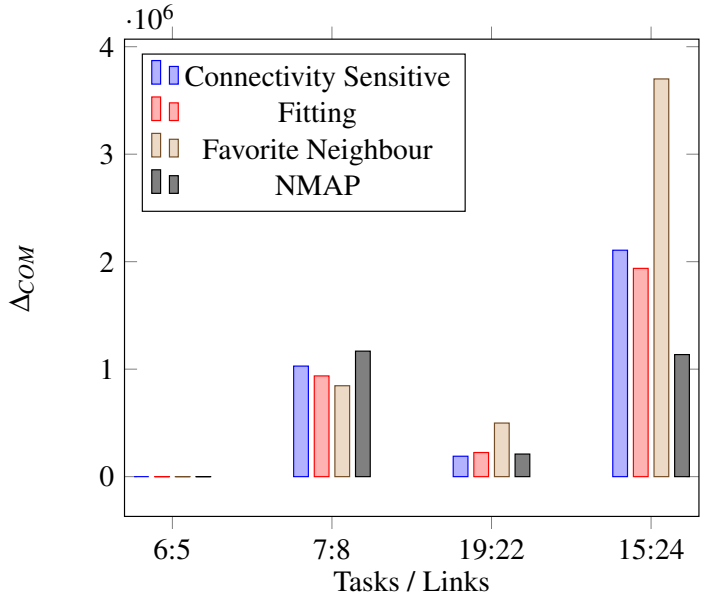


Abbildung 9.29: Platzierung von Task-Graphen mit unterschiedlichem Verzweigungsgrad

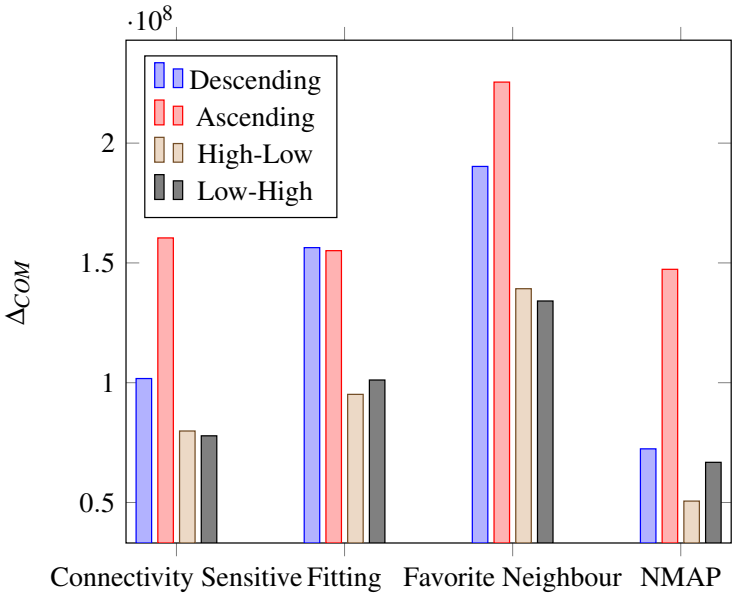


Abbildung 9.30: Platzierung von Applikationspaket 1 in unterschiedlicher Reihenfolge; defekte Router

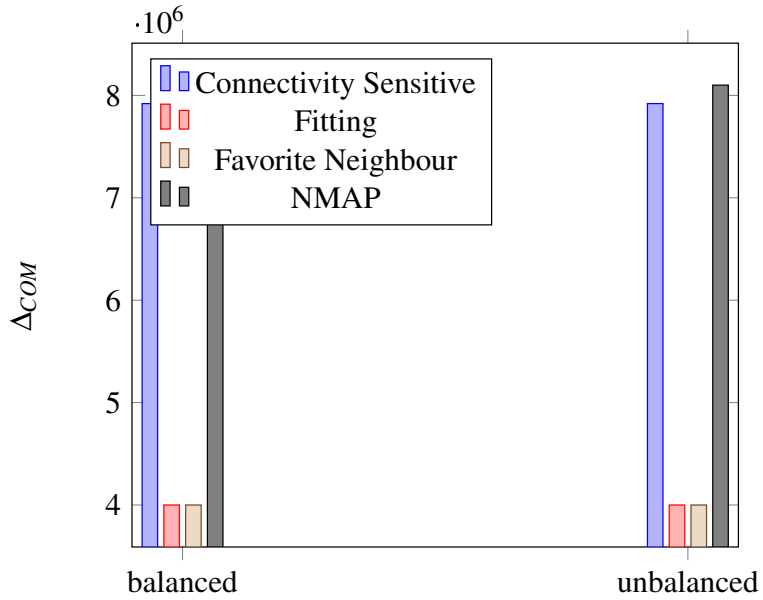


Abbildung 9.31: balancierte und unbalancierte Variante des Consumer Task-Graphen

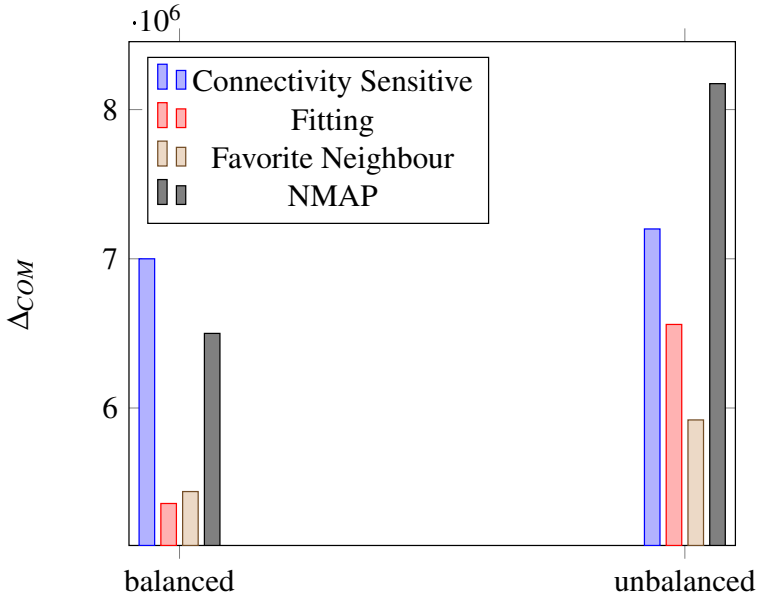


Abbildung 9.32: balancierte und unbalancierte Variante des Consumer Task-Graphen; 20% defekte Router

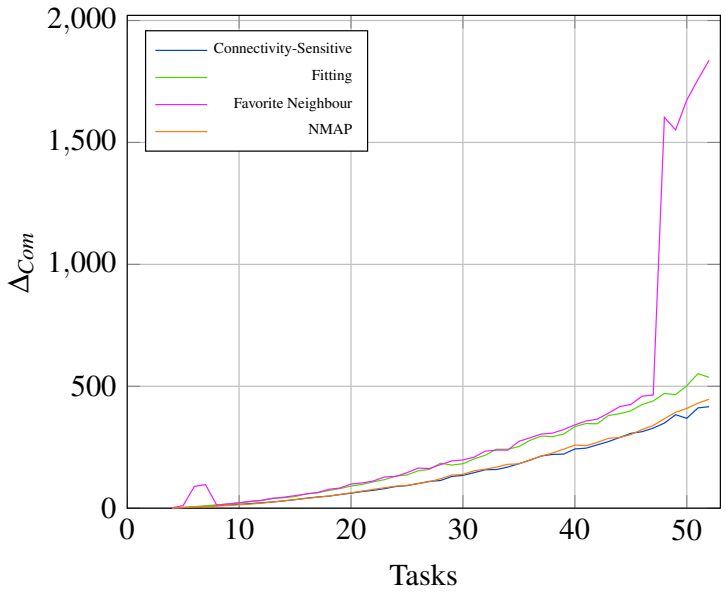


Abbildung 9.33: 8x8 Kern-Graph mit 20% ausgefallenen Routern; unbalancierte Fork-Join Task-Graphen

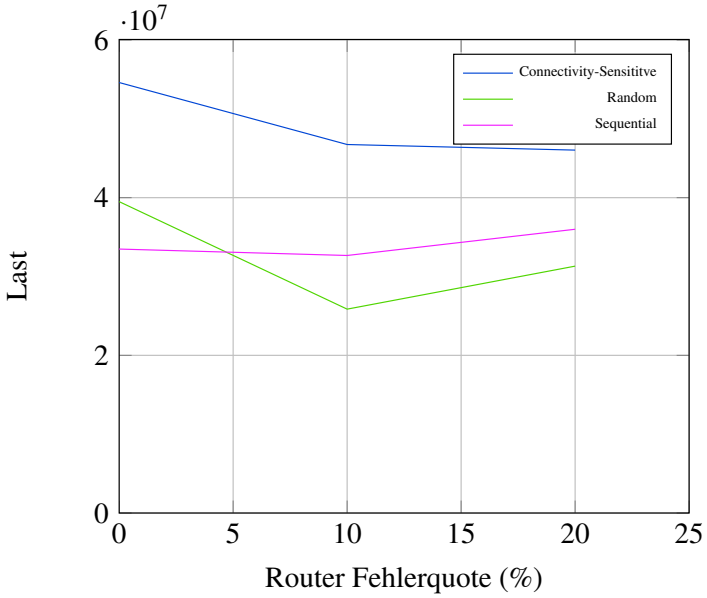


Abbildung 9.34: Applikationspaket 1; maximale Last

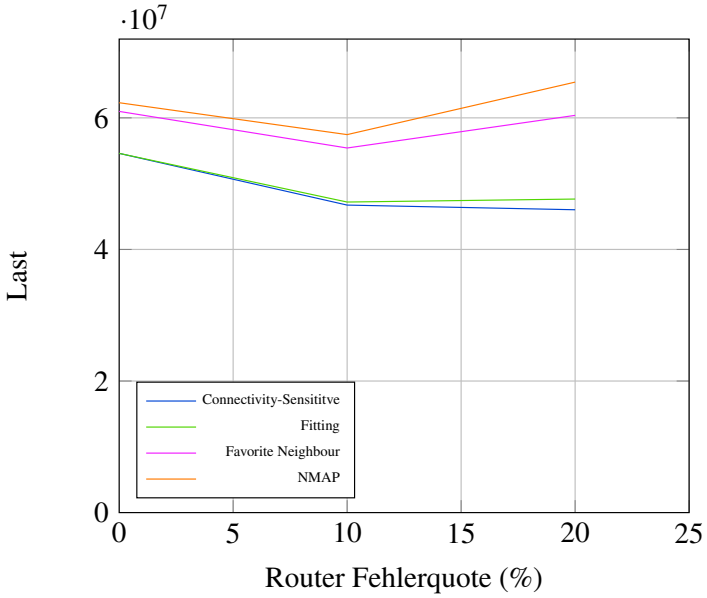


Abbildung 9.35: Applikationspaket 1; maximale Last

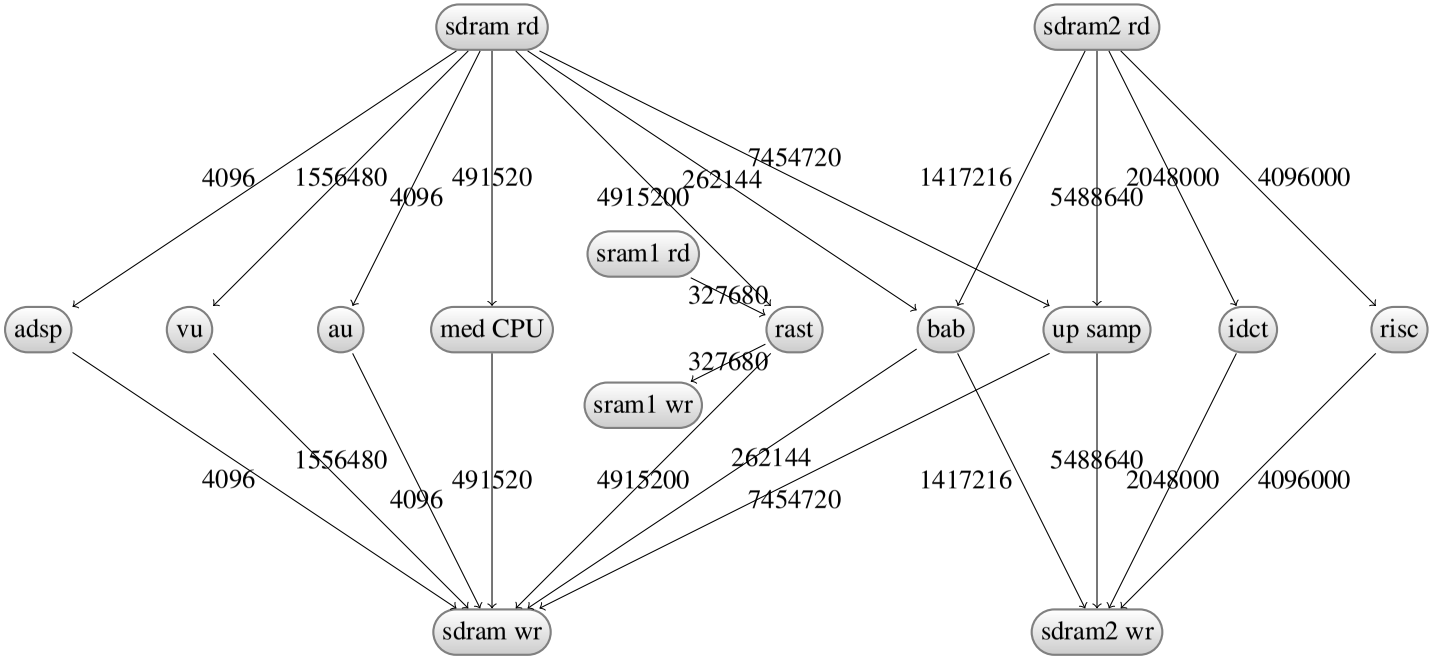


Abbildung 9.36: MPEG-4 Decoder

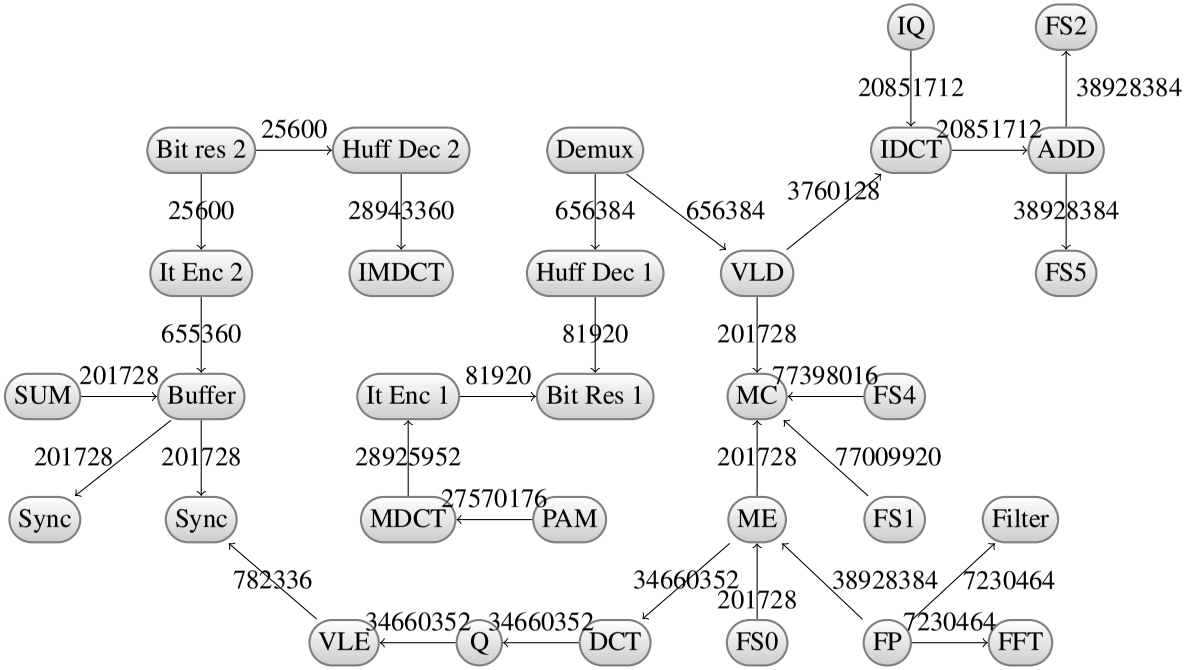


Abbildung 9.37: Multimedia Anwendung

Literatur

Adaptive Computing (2013a). *Maui Cluster Scheduler Administrator's Guide*. URL: http://docs.adaptivecomputing.com/maui/index.php.

– (2013b). *Moab Cluster Suite*. URL: http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-basic-edition/.

Asanovic, Krste u. a. (2006). *The Landscape of Parallel Computing Research: A View from Berkeley*. Techn. Ber. UCB/EECS-2006-183. EECS Department, University of California, Berkeley. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

Ascia, G., V. Catania und M. Palesi (2005). „Mapping cores on network-on-chip". In: *International Journal of Computational Intelligence Research* 1.1-2, S. 109–126.

– (2006). „A multi-objective genetic approach to mapping problem on Network-on-Chip". In: *Journal of Universal Computer Science* 12.4, S. 370–394.

Austin, T.M. (1999). „DIVA: a reliable substrate for deep submicron microarchitecture design". In: *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, S. 196–207. DOI: 10.1109/MICRO.1999.809458.

Basumallik, A., S.-J. Min und R. Eigenmann (2007). „Programming Distributed Memory Sytems Using OpenMP". In: *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, S. 1–8. DOI: 10.1109/IPDPS.2007.370397.

Borkar, Shekhar (2005). „Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation". In: *IEEE Micro* 25.6, S. 10–16. ISSN: 0272-1732.

Brinkschulte, U., M. Pacher und A. Von Renteln (2007). „Towards an artificial hormone system for self-organizing real-time task allocation". In: *Software Technologies for Embedded and Ubiquitous Systems*, S. 339–347.

Brinkschulte, U., A. Von Renteln und M. Pacher (2008). „Measuring the quality of an artificial hormone system based task mapping". In: *Proceedings of the 2nd International Conference on Autonomic Computing and Communication Systems*. ICST (Institute for Computer Sciences, Social-Informatics und Telecommunications Engineering), S. 32.

Carlson, Bill u. a. (2003). „Programming in the partitioned global address space model". In: *Tutorial at Supercomputing*.

Charles, J. u. a. (2009). „Evaluation of the Intel Core i7 Turbo Boost feature“. In: *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, S. 188–197. DOI: 10.1109/IISWC.2009.5306782.

Conway, P. und B. Hughes (2007). „The AMD Opteron Northbridge Architecture“. In: *Micro, IEEE* 27.2, S. 10–21. ISSN: 0272-1732. DOI: 10.1109/MM.2007.43.

Dally, William J. und Charles L. Seitz (1986). „The torus routing chip“. English. In: *Distributed Computing* 1.4, S. 187–196. ISSN: 0178-2770. DOI: 10.1007/BF01660031. URL: http://dx.doi.org/10.1007/BF01660031.

Dally, William und Brian Towles (2003). *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN: 0122007514.

Dally, W.J. und B. Towles (2001). „Route packets, not wires: on-chip interconnection networks“. In: *Design Automation Conference, 2001. Proceedings*, S. 684–689. DOI: 10.1109/DAC.2001.156225.

Dick, Robert (2013). *Embedded System Synthesis Benchmark Suite (E3S)*. URL: http://ziyang.eecs.umich.edu/~dickrp/e3s/.

Drozdowski, Maciej (2009). „Scheduling for Parallel Processing“. In: Springer London Ltd, S. 166. ISBN: 1-84882-309-6.

Duato, José, Sudhakar Yalamanchili und Lionel Ni (2003). *Interconnection Networks*. Morgan Kaufmann.

Embedded Microprocessor Benchmark Consortium (2013). *EEMBC Benchmarks*. URL: http://eembc.org.

Felber, P. u. a. (2010). „The Velox Transactional Memory Stack“. In: *Micro, IEEE* 30.5, S. 76–87. ISSN: 0272-1732. DOI: 10.1109/MM.2010.80.

Filch, José u. a. (2011). *Designing Network On-Chip Architectures on the Nanoscale Era*. Hrsg. von José Filch und Davide Bertozzi. Chapman & Hall/CRC.

Garbade, Arne, Sebastian Weis, Bernhard Fechner u. a. (2013). „Fault Localization in NoCs Exploiting Periodic Heartbeat Messages in a Many-Core Environment“. In: *Proceedings of the 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum (CASS 2013)*. IEEE, S. 791–795.

Garbade, Arne, Sebastian Weis, Sebastian Schlingmann u. a. (2013). „Impact of Message-Based Fault Detectors on a Network on Chip“. In: *Belfast Proceedings of the 21th International Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2013)*. IEEE, S. 470–477.

Georgiou, Y. (2010). „Contributions for Resource and Job Management in High Performance Computing“. In:

Greenhalgh, Peter (2011). *big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7*. Whitepaper. ARM. URL: http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf.

Held, Jim, Jerry Bautista und Sean Koehl (2006). *From a few Cores to many: A Tera-Scale Computing Research Overview*. Whitepaper. Intel. URL: http://software.intel.com/sites/default/files/m/1/f/9/terascale_overview_paper.pdf.

Hentschke, R. u. a. (2002). „Analyzing area and performance penalty of protecting different digital modules with Hamming code and triple modular redundancy“. In: *Integrated Circuits and Systems Design, 2002. Proceedings. 15th Symposium on*, S. 95–100. DOI: 10.1109/SBCCI.2002.1137643.

IBM (2012a). *IBM LoadLeveler*.
URL: http://www-03.ibm.com/systems/software/loadleveler/.

– (2012b). *IBM xCAT*.
URL: http://www-03.ibm.com/systems/software/xcat/index.html.

Intel (2010). *SCC External Architecture (EAS) - Revision 0.94*.

Intel Corporation (2012).
Intel Architecture Instruction Set Extensions Programming Reference. 319433-012. Intel Corporation. URL: http://download-software.intel.com/sites/default/files/m/3/2/1/0/b/41417-319433-012.pdf.

– (2013). *The Intel Xeon Phi Product Family - Highly-Parallel Processing for Unparalelled Discovery*. Whitepaper. Intel. URL: http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/high-performance-xeon-phi-coprocessor-brief.pdf.

Jingcao, Hu und R. Marculescu (2003). „Energy-aware mapping for tile-based NoC architectures under performance constraints“. In: *Design Automation Conference, 2003. Proceedings of the ASP-DAC 2003. Asia and South Pacific*, S. 233–239. DOI: 10.1109/ASPDAC.2003.1195022.

– (2005). „Energy- and performance-aware mapping for regular NoC architectures“. In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 24.4, S. 551–562. ISSN: 0278-0070. DOI: 10.1109/TCAD.2005.844106.

Kägi, Alain, Doug Burger und James R. Goodman (1997).
„Efficient synchronization: let them eat QOLB“. In: *Proceedings of the 24th annual international symposium on Computer architecture*. ISCA '97.
Denver, Colorado, USA: ACM, S. 170–180. ISBN: 0-89791-901-7.
DOI: 10.1145/264107.264166.
URL: http://doi.acm.org/10.1145/264107.264166.

Kalray (2013). *Kalray MPPA 256 Processor*.
URL: http://www.kalray.eu/products/mppa-manycore/mppa-256/.

Kephart, J.O. und D.M. Chess (2003). „The Vision of Autonomic Computing“.
In: *IEEE Computer* 36.1, S. 41–50. ISSN: 0018-9162.
DOI: 10.1109/MC.2003.1160055.

Kleinrock, L. und R. R. Muntz (1972). „Processor Sharing Queueing Models of Mixed Scheduling Disciplines for Time Shared System“.
In: *J. ACM* 19.3, S. 464–482. ISSN: 0004-5411. DOI: 10.1145/321707.321717.
URL: http://doi.acm.org/10.1145/321707.321717.

Koziris, N. u. a. (2000). „An efficient algorithm for the physical mapping of clustered task graphs onto multiprocessor architectures“. In: *Parallel and Distributed Processing, 2000. Proceedings. 8th Euromicro Workshop on*, S. 406–413.
DOI: 10.1109/EMPDP.2000.823437.

Lawrence Livermore National Laboratory (2012). *SLURM*.
URL: https://computing.llnl.gov/linux/slurm/.

Lyons, R.E. und W. Vanderkulk (1962).
„The Use of Triple-Modular Redundancy to Improve Computer Reliability".
In: *IBM Journal of Research and Development* 6.2, S. 200–209. ISSN: 0018-8646.
DOI: 10.1147/rd.62.0200.

McCluskey, E.J. (1990). „Design techniques for testable embedded error checkers".
In: *Computer* 23.7, S. 84–88. ISSN: 0018-9162. DOI: 10.1109/2.56855.

Moore, Gordon E u. a. (1965). *Cramming more components onto integrated circuits*.

Murali, S. und G. De Micheli (2004).
„Bandwidth-constrained mapping of cores onto NoC architectures". In: *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*.
Bd. 2, 896–901 Vol.2. DOI: 10.1109/DATE.2004.1269002.

Pacher, M. und U. Brinkschulte (2010). „Real-Time Distribution of Time-Dependant Tasks in Heterogeneous Environments".
In: *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2010 13th IEEE International Symposium on*, S. 21–28.
DOI: 10.1109/ISORCW.2010.9.

Pascual, J., J. Navaridas und J. Miguel-Alonso (2009).
„Effects of topology-aware allocation policies on scheduling performance".
In: *Job Scheduling Strategies for Parallel Processing*. Springer, S. 138–156.

Perfumo, Cristian u. a. (2008). „The limits of software transactional memory (STM): dissecting Haskell STM applications on a many-core environment".
In: *Proceedings of the 5th conference on Computing frontiers*. CF '08.
Ischia, Italy: ACM, S. 67–78. ISBN: 978-1-60558-077-7.
DOI: 10.1145/1366230.1366241.
URL: http://doi.acm.org/10.1145/1366230.1366241.

Radu, C. und L. Vintan (2011).
„Optimized Simulated Annealing for Network-on-Chip Application Mapping".
In: *Proceedings of the 18th International Conference on Control Systems and Computer Science (CSCS-18)*. Bd. 1, S. 452–459.

Schlingmann, S. u. a. (2011). „Connectivity-Sensitive Algorithm for Task Placement on a Many-Core Considering Faulty Regions".
In: *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, S. 417–422. DOI: 10.1109/PDP.2011.58.

Temkine, Grigori, Oleg Drapkin und Gordon Caruk (2006). „Methods and apparatus for processing graphics data using multiple processing circuits". English.
US2006282604 (A1). URL: http://worldwide.espacenet.com/publicationDetails/biblio?CC=US&NR=2006282604&KC=&FT=E&locale=en_EP.

Tilera (2008). *TilePro64 Processor*. Whitepaper. Tilera.
URL: http://www.tilera.com/sites/default/files/productbriefs/TILEPro64_Processor_PB019_v4.pdf.

– (2013). *Tile-Gx8072 Processor*. Whitepaper. Tilera.
URL: http://www.tilera.com/sites/default/files/productbriefs/TILE-Gx8072_PB041-02.pdf.

Tol, Erik B. van der und Egbert G. Jaspers (2001).
„Mapping of MPEG-4 decoding on a flexible architecture platform“. In: S. 1–13.
DOI: 10.1117/12.451067. URL: +%20http://dx.doi.org/10.1117/12.451067.

Wakerly, J.F. (1976).
„Microcomputer reliability improvement using triple-modular redundancy“.
In: *Proceedings of the IEEE* 64.6, S. 889–895. ISSN: 0018-9219.
DOI: 10.1109/PROC.1976.10239.

Yang, Xue-Jun u. a. (2011).
„The TianHe-1A Supercomputer: Its Hardware and Software“. English.
In: *Journal of Computer Science and Technology* 26.3, S. 344–351. ISSN: 1000-9000.
DOI: 10.1007/s02011-011-1137-8.
URL: http://dx.doi.org/10.1007/s02011-011-1137-8.

Yoo, AndyB., MorrisA. Jette und Mark Grondona (2003).
„SLURM: Simple Linux Utility for Resource Management“.
In: *Job Scheduling Strategies for Parallel Processing*.
Hrsg. von Dror Feitelson, Larry Rudolph und Uwe Schwiegelshohn. Bd. 2862.
Lecture Notes in Computer Science. Springer Berlin Heidelberg, S. 44–60.
ISBN: 978-3-540-20405-3. DOI: 10.1007/10968987_3.
URL: http://dx.doi.org/10.1007/10968987_3.

Ziakas, D. u. a. (2010). „Intel QuickPath Interconnect Architectural Features Supporting Scalable System Architectures“. In: *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, S. 1–6.
DOI: 10.1109/HOTI.2010.24.

Abbildungsverzeichnis

Tabellenverzeichnis

Sebastian Schlingmann

Universität Augsburg
Institut für Informatik
Lehrstuhl für Systemnahe Informatik und Kommunikationssysteme
Universitätsstr. 6a, 86135 Augsburg

Tel:	+49 (821) 598-2379
Fax:	+49 (821) 598-2359
eMail:	schlingmann@informatik.uni-augsburg.de
Web:	www.informatik.uni-augsburg.de

Persönliches

Geboren am 12.05.1981 in Mainz

Ausbildung

2007 Diplom im Fach Angewandte Informatik an der Universität Augsburg

Beruf

2008 - 2013 Wissenschaftlicher Mitarbeiter am Lehrstuhl für Systemnahe Informatik und Kommunikationssysteme, Universität Augsburg

Veröffentlichungen

Impact of Message-Based Fault Detectors on a Network on Chip - 2013
Arne Garbade, Sebastian Weis, Sebastian Schlingmann, Bernhard Fechner, Theo Ungerer
Proceedings of the 21th International Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2013), pages 470-477

Connectivity-sensitive Algorithm for Task Placement on a Many-core Considering Faulty Regions - 2011
Sebastian Schlingmann, Arne Garbade, Sebastian Weis, Theo Ungerer
Proceedings of the 19th International Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2011)

weitere Veröffentlichungen

Towards Fault Detection Units as an Autonomous Fault Detection Approach for Future Many-Cores - **2011**
Sebastian Weis, Arne Gabarde, Sebastian Schlingmann, Theo Ungerer
Proceedings of the 1st Workshop on Software-Controlled, Adaptive Fault-Tolerance in Microprocessors (SCAFT 2011) at the 24th International Conference on Architecture of Computing Systems (ARCS 2011), pages 20-23

MANJAC - Ein Many-Core-Emulator auf Multi-FPGA-Basis - **2011**
Christian Bradatsch, Sebastian Schlingmann, Theo Ungerer, Sascha Uhrig
Mitteilungen - Gesellschaft für Informatik e. V., Parallel-Algorithmen und Rechnerstrukturen, ISSN 0177 - 0454, 2011, pages 48-57

OC Techniques Applied to Solve Reliability Problems in Future 1000-core Processors - **2011**
Arne Garbarde, Sebastian Weis, Sebastian Schlingmann, Theo Ungerer
Organic Computing — A Paradigm Shift for Complex Systems, pages 575 - 577

Self-optimized Routing in a Network on-a-Chip - **2008**
Wolfgang Trumler, Sebastian Schlingmann, Theo Ungerer, Jun Ho Bahn, Nader Bagherzadeh
IFIP 20th World Computer Congress, Second IFIP TC 10 International Conference on Biologically-Inspired Collaborative Computing, September 8-9 2008, Milano, Italy