Hierarchical Self-Organizing Systems for Task-Allocation in Large Scaled Distributed Architectures

Dissertation zur Erlangung des Doktorgrades der Naturwissenschaften

vorgelegt beim Fachbereich Informatik und Mathematik
der Johann Wolfgang Goethe-Universität
in Frankfurt am Main

von
Andreas Lund
aus
Wiesbaden

Frankfurt 2019
(D30)

vom Fachbereich Informatik und Mathematik der
Johann Wolfgang Goethe-Universität als Dissertation
angenommen.

Dekan:	Prof. Dr. Lars Hedrich
Gutachter:	Prof. Dr. Uwe Brinkschulte
	Prof. Dr. Lars Hedrich
Datum der Disputation:	09.07.2019

You know you have a distributed system when the crash of a computer you've never heard of stops you from getting any work done.

-Leslie Lamport-

Abstract

This thesis deals with the subject of autonomous, decentralized task allocation in a large scaled multi-core network. The self-organization of such interconnected systems becomes more and more important for upcoming developments. It is to be expected that the complexity of those systems becomes hardly manageable to human users.

Self-organization is part of a research field of the *Organic Computing initiative*, which aims to find solutions for technical systems by imitating natural systems and their processes. Within this initiative, a system for task allocation in a small scaled multi-core network was already developed, researched and published. The system is called the *Artificial Hormone System (AHS)*, since it is inspired by the endocrine system of mammals. The *AHS* produces a high amount of communication load in case the multi-core network is of a bigger scale.

The contribution of this thesis is two new approaches, both based on the *AHS* in order to cope with large scaled architectures. The major idea of those two approaches is to introduce a hierarchy into the *AHS* in order to reduce the produced communication load. The first and more detailed researched approach is called the *Hierarchical Artificial Hormone System (HAHS)*, which orders the processing elements in clusters and builds an additional communication layer between them. The second approach is the *Recursive Artificial Hormone System (RAHS)*, which also clusters the system's processing elements and orders the clusters into a topological tree structure for communication.

Both approaches will be explained in this thesis by their principle structure as well as some optional methods. Furthermore, this thesis presents estimations for the worst-case timing behavior and the worst-case communication load of the *HAHS* and *RAHS*. At last, the evaluation results of both approaches, especially in comparison to the *AHS*, will be shown and discussed.

Organic Computing, Task-Allocation, Distributed Computing, Autonomous Computing, Decentralization, Self-Organization

Zusammenfassung

Zwei Entwicklungen verändern zunehmend die Möglichkeiten informationstechnischer Systeme, insbesondere die der eingebetteten Systeme: eine steigende Integrationsdichte und das „Internet of Things“.

Die steigende Integrationsdichte führt dazu, dass mehr und mehr Schaltungen auf einem einzigen Chip integriert werden können. Dadurch können komplexere „Systems-on-Chip“ mit mehreren Funktionalitäten vereint in einem Chip gefertigt werden. Doch birgt eine höhere Integrationsdichte auch eine höhere Gefahr von transienten sowie permanenten Fehlern und Ausfällen des Chips. Die Ursache für diese Fehler und Ausfälle ist zum einem die Elektromigration von Schaltungen und Leiterbahnen und zum anderen „Single Event“-Effekte in den Schaltungen, welche durch den Chip dringende Strahlungen ausgelöst werden.

Der Begriff „Internet of Things“ bezeichnet den zunehmenden Trend eingebettete Systeme mit einem internetfähigen Mikrocontroller oder Mikrochip auszustatten. Dies resultiert in großen, verteilten Systemen, die alle miteinander verbunden sind. Diese Systeme, die im Verbund zusammen arbeiten (zum Beispiel in einer „Smart Factory“ oder einer „Smart City“), haben unter Umständen eine komplexe Topologie mit mehreren Hierarchiebenen. Die Entwicklung und Wartung von Anwendungen für diese Systeme gestaltet sich aufgrund der Komplexität für Menschen schwierig.

Diese Entwicklungen erfordern neue Paradigmen in der Software- und Architekturentwicklung. Die *Organic Computing* Forschungsinitiative sucht nach Lösungen für die aus diesen neuen Entwicklungen resultierenden Problemen. Es wird dabei versucht Techniken und Methoden aus der Natur in technischen Systemen nachzuahmen. Das *Organic Computing* definiert dabei die sogenannten *Selbst-X*-Eigenschaften. Die wichtigsten dieser Eigenschaften sind die *Selbstkonfiguration*, die *Selbstoptimierung* und die *Selbstheilung*. Bei Umsetzung dieser Eigenschaften erlangt das System einen Grad der Selbstorganisation, welcher das System dazu befähigt, diese Eigenschaften autonom und ohne externen Einfluss durchzuführen. Auf diese Weise können Systeme entwickelt werden, die auch in unerwarteten Zuständen funktionsfähig bleiben und dabei die Komplexität der darunterliegenden Architektur gegenüber dem Benutzer verbergen.

Ein spezifisches Problem im Zusammenhang mit verbundenen Prozessorkernen ist die Taskverteilung in einem solchen System. Gegeben sei dabei eine Menge von Tasks, die auf eine Menge von heterogenen Prozessorkernen (auch *Processing Elements (PE)* genannt) verteilt werden müssen. Diese Taskverteilung soll während der Laufzeit anhand der aktuellen Eignung der PEs optimiert werden. Zusätzlich sollen beim Ausfall eines PEs die verlorenen gegangenen Tasks wieder neu verteilt werden. Die Heterogenität der PEs bedeutet in diesem Zusammenhang, dass die verschiedenen PEs gegebenfalls verschiedene Eignungen für die Tasks im System haben. Ein System, das diese Taskverteilung bewerkstelligt, ist das *Künstliche Hormonsystem (KHS)* (im Englischen: *Artifcial Hormon System (AHS)* genannt), welches von *Brinkschulte et al.* entwickelt und erforscht wurde. Das KHS benutzt verteilte Regelungsschleifen in jedem PE, welche durch Nachrichten, den Hormonen, miteinander kommunizieren. Drei verschiedene Typen von Hormonen existieren im KHS. Als Erstes die Eignungswerte (im Englischen: *Eager Values*), die die Eignung eines PEs zu einem Task angeben. Als Gegenspieler zu den Eignungswerten existieren Suppressoren, welche die Eignung eines Task auf einem PE verringern. Und als letzten Typ, die Acceleratoren, welche die Eignung von einzelnen PEs für eine Task verbessern. Pro Regelungszyklus werden die aktuellen Eignungen für alle Tasks auf dem PE anhand der empfangenen Hormone neu berechnet. Danach wählt jedes PE genau ein Task per Zyklus und überprüft, ob er für diese Task die höchste Eignung hat. Falls dem so ist, übernimmt das PE den Task und allokiert ihn. Daraufhin sendet er alle zuvor berechneten Eignungswerte, eventuelle Suppressoren und Acceleratoren an alle anderen PEs im System. Das KHS implementiert die Selbstkonfiguration, Selbstoptimierung und Selbstheilung für die Taskverteilung auf heterogenen Kernen. Zudem konnten für alle „Selbst-X“-Eigenschaften Zeitschranken nachgewiesen werden, sodass das KHS auch im Kontext von Echtzeitsystemen genutzt werden kann.

Ein Problem des KHS ist jedoch, dass die Skalierung in der Anzahl der PEs und Tasks schwierig ist. Die Kommunikationslast eines hochskalierten KHS bringt aktuelle Kommunikationsbusse an ihre Grenzen. Das Problem ist vor allem die maximale Kommunikationslast, die bei der initialen Taskverteilung nach dem Start des Systems, also der Selbstkonfiguration, erzeugt wird. So werden zum Anfang $n \cdot m$ Eignungswerte (n = Anzahl PEs; m = Anzahl Tasks)

in einem Zyklus ausgesendet. Dies führt bei 100 PEs und 50 Tasks zu einer benötigten Bandbreite von 1,7 MBytes/s.

Diese Dissertation stellt zwei neue Systeme vor, welche auf dem KHS basieren und gleichzeitig die maximale Kommunikationslast verringern. Die Idee des ersten neuen Systems, dem *Hierarchisch Künstlichen Hormonsystem (HKHS)* (im Englischen: *Hierarchical Artificial Hormone System (HAHS)*) ist die PEs in separate Cluster zu ordnen. In jedem Cluster wird exakt ein PE als sogenannter *Cluster Head* auserwählt. Alle Cluster Heads untereinander formen einen zweiten, übergeordneten Regelungszyklus. Dies stellt die einzige Kommunikation zwischen den Clustern dar. Alle Tasks werden nun zuerst in dem oberen Regelungszyklus (*inter-cluster* Zyklus) verhandelt und anschließend von dem Cluster Head in dem entsprechenden Cluster-Regelungszyklus (*intra-cluster* Zyklus) freigeschaltet. Dies führt dazu, dass die gesamte Taskmenge im oberen Regelungszyklus in Untermengen aufgeteilt wird. Der Vorteil dieses Beitrags ist, dass die anfängliche Kommunikationslast von $n \cdot m$ Hormonen auf $c \cdot m$ (wobei c = Anzahl der Cluster) Hormone reduziert werden.

Das HKHS kann in verschiedenen Varianten implementiert werden, die je nach Anwendungsszenario erforderlich sind. So stehen zwei Arten der Taskorganisation zur Verfügung. Zum einem kann man das Prinzip der Aufteilung der Tasks in Untermengen schon vordefinieren. Diese voreingestellten, statischen Taskuntermengen werden *Organe* genannt. Das dazu gehörig Konzept heißt *Organ Task* Konzept. In diesem Konzept werden nur noch die Untermengen als Ganzes im oberen Regelungszyklus anhand eines speziellen Organ Tasks, verhandelt. Gewinnt ein Cluster Head einen dieser Organ Tasks, so schaltet er alle Tasks des Organs gleichzeitig in seinem Cluster frei. Mit dieser Methode wird sowohl die Kommunikationslast nochmal deutlich reduziert als auch die Allokationszeit (die Zeit bis alle Tasks allokiert wurden) verkürzt. Das Gegenstück ist das sogenannte *Single Task* Konzept, in dem jeder Task sowohl im *inter-cluster* Zyklus als auch in den *intra-cluster* Zyklen einzeln verhandelt wird.

Neben dem Taskkonzept lässt sich das HKHS auch in seinem Verhalten hinsichtlich der Cluster und Cluster Heads einstellen. So können die Cluster Heads entweder vordefiniert werden und bleiben damit statisch ausgewählt oder sie können von dem System dynamisch ausgewählt und verwaltet werden. Zweiteres

lässt sich auch über das genutzte KHS in jedem Cluster implementieren. Dazu wird ein spezieller Task eingeführt, der pro Cluster einmal ausgeführt werden soll. Ein PE, das diesen Task gewinnt, übernimmt die Aufgabe des Cluster Heads für diesen Cluster. Mit den dynamischen Cluster Heads ist ein HKHS robuster, da Ausfälle von Cluster Head PEs kompensiert werden können, während dies bei statischen Cluster Heads zum Ausfall eines ganzen Clusters führen würde. Ähnlich kann auch mit der Zuteilung der PEs zu den Clustern verfahren werden. Entweder ist diese Zugehörigkeit statisch und vordefiniert oder wird dynamisch zur Laufzeit vom System selbst bestimmt. Eine dynamische Zuteilung kann durch das Versenden von sogenannten *Cluster Acceleratoren* durch die Cluster Heads umgesetzt werden. Diese Cluster Acceleratoren haben einen bestimmten Stärkewert, der sich an einer Eigenschaft der PEs orientiert. Ein jedes PE, das einen solchen Accelerator erhält, wird sich dem Cluster zuordnen, der ihm den höchsten Stärkewert gesendet hat. Eine solche Eigenschaft, die in dieser Dissertation benutzt wird, ist die räumliche Distanz vom Cluster Head; dass heißt der Stärkewert eines Cluster Accelerators nimmt mit der Entfernung zum Cluster Head ab. Dadurch bilden sich räumlich zusammenhängende Cluster.

Bei der Nutzung des HKHS ergibt sich eine Schwierigkeit mit den Eignungswerten in der oberen Reglungsschleife, der Schleife zwischen den Cluster Heads. Die Cluster Heads können nur durch den Einsatz von zusätzlicher Konmmunikation herausfinden, wie geeignet die PEs in ihrem Cluster für jede einzelne Task sind. Um dadurch nicht letztendlich eine höhere Kommunikationslast als das KHS zu erzeugen, werden in dieser Dissertation verschiedene Methoden zur Abbildung der Eignungswerte auf den oberen Regelungskreis vorgestellt. Alle Methoden beruhen dabei auf der Kommunikation zwischen den PEs, während sie versuchen diese möglichst stark zu reduzieren.

Das zweite System zur Lösung des Taskverteilungsproblems ist das *Rekursive Künstlichen Hormonsystem RKHS)* (im Englischen: *Recursive Artificial Hormone System (RAHS)*). Dieses System ist eine Weiterentwicklung des HKHS und benutzt mehrere hierarchische Ebenen in denen separate Cluster aus PEs angeordnet sind. Jeder Cluster hat genau ein Repräsentanten-PE in einem Cluster aus der nächsthöheren Ebene; diese sind vergleichbar mit den Cluster Heads aus dem HKHS. Wie beim HKHS startet die Taskverteilung in der obersten Ebene mit der gesamten Taskmenge. Die PEs allokieren die Tasks oder

reichen sie weiter in die nächste Ebene, falls sie ein Repräsentant sind. Ob eine solche PE den Task selber allokiert oder weiterreicht kann entweder statisch vordefiniert werden oder auch dynamisch entschieden werden. Eine dynamische Entscheidung könnte zum Beispiel durch Regelungsschleifen getroffen werden. Die verteilten Regelungsschleifen in den Clustern werden *horizontale Regelungszyklen* genannt, während die Kommunikation zwischen Repräsentant und einem Cluster der nächsttieferen Ebene beziehungsweise die zugehörige Regelungsschleifen *vertikale Regelungszyklen* genannt werden. Ähnlich wie beim HKHS ergibt sich auch im RKHS das Problem der Bestimmung der Eignungswerte in den höheren Ebenen. Hierfür stellt diese Dissertation eine Lösungsmethode vor, die die Eignungswerte zum Start des Systems sukzessive in die höheren Ebenen abbildet. Weiterhin stellt diese Dissertation eine Methode zur dynamischen Bestimmung ob ein Task von einer Repräsentanten-PE allokiert oder weitergereicht wird vor.

Für beide Systeme, das HKHS und das RKHS, werden in der vorliegende Dissertation Abschätzungen zum *Worst-Case*-Laufzeitverhalten und *Worst-Case*-Kommunikationsverhalten der drei *Selbst-X*-Eigenschaften (*-konfiguration, -optimierung, -heilung*) aufgestellt. Zudem wird der Einfluss der verschiedenen Konzepte und Varianten beider Systeme auf das Zeitverhalten und Kommunikationsverhalten analysiert und abgeschätzt.

Zu Evaluationszwecken wurden für das HKHS und RKHS Simulatoren entwickelt, in denen verschiedene Architekturkonfigurationen sowie verschiedene Methoden simuliert werden können. Während der Simulator für das HKHS auf dem bestehenden C++ Simulator für das KHS basiert, wurde für das RKHS, aufgrund seiner komplexen Hierarchiemöglichkeiten, ein komplett neuer Simulator in C++ entwickelt. Zudem wurde die bestehende C-Middleware des KHS um die Hierarchiemerkmale des HKHS erweitert, sodass dieses auch im Produktiveinsatz als Middleware zwischen mehreren Recheneinheiten genutzt werden kann.

Im Rahmen dieser Dissertation wurden beide Systeme analysiert und einige ihrer Konzepte, Varianten und Methoden evaluiert. Dabei konnte gezeigt werden, dass beide Systeme mit der maximalen Kommunikationslast unter dem Wert des KHS bleiben und dabei immer noch die *Selbst-X*-Eigenschaften des KHS gewährleisten. Zudem konnte gezeigt werden, dass ein Organ Task HKHS

schneller die vollständige Taskmenge allokiert als das KHS. Weiterhin wurde die optimale Anzahl an Organen in einem Organ Task HKHS in Abhängigkeit der Taskanzahl mittels Evaluationen bestimmt. Für das RKHS zeigte sich, dass mehrere Hierarchie Ebenen nicht nur zu einer Verlängerung der Allokationszeit führen, sondern durch sie auch die maximale Kommunikationslast steigt. Daher ist eine mehrstufiges RKHS nur dann sinnvoll, wenn die vorliegende Topologie dies erfordert.

Zuletzt stellt die vorliegende Dissertation vergleichbare Systeme und Lösungen für das Taskverteilungsproblem vor. Dabei liegt der Fokus auf den zahlreichen Methoden in *Multi-Agenten-Systemen*, welche ein ähnliches Funktionsprinzip wie das KHS (und damit auch das HKHS und RKHS) einsetzen.

Organic-Computing, Taskallokation, Verteiltes Rechnen, Autonomes Rechnen, Dezentralisierung, Selbst-Organisation

Contents

List of Figures

List of Tables

List of Terms

AHS	Artificial Hormone System.
CNP	Contract Net Protocol.
GUI	Graphical User Interface.
HAHS	Hierarchical Artificial Hormone System.
ILP	Integer Linear Program.
IoT	Internet of Things.
MAS	Multi-Agent System.
MCC	Meteorological Command and Control.
MFC	Microsoft Foundation Classes.
PE	Processing Element.
RAHS	Recursive Artificial Hormone System.
SCP	Set Covering Problem.
SMT	Satisfiability Modulo Theory.
SoC	System-on-Chip.
SPP	Set Partitioning Problem.

Chapter 1

Introduction

At present, two developments prevail in the field of computer science and change the interactions, opportunities and paradigms in many areas of our lives. Scientific research, industrial production, economic strategies and even the society itself will be changed by these developments.

The growing integration density for *integrated circuits* leads to a higher number of cores on a single die[1]. The consequences are more complex System-on-Chips (SoCs) which consist of many heterogeneous, independent cores working together in one device (e.g. a smartphone) [20].

The second development is the so-called Internet of Things (IoT). This term describes the ongoing trend to integrate micro-controllers into devices of everyday life, for instance thermostats, washing machines and cars. Those micro-controllers are connected to the internet and communicate with servers of the manufacturer and other devices fitted with an Internet-capable micro-controller.

In figure 1.1 processors from 1970 until today are plotted by means of five different metrics. Regarding the number of logical cores (black diamonds) in each processor, it is obvious that in the past few years (since approx. 2005) the amount grew fast; this trend is expected to continue.

Another interesting evident from the aforementioned figure is the still growing number of transistors on a single die as a result of rising integration density, i.e. the transistors take less and less space on a die. Such a trend has negative

[1] In this context a die is the block of semiconducting material on which a functional circuit is implemented

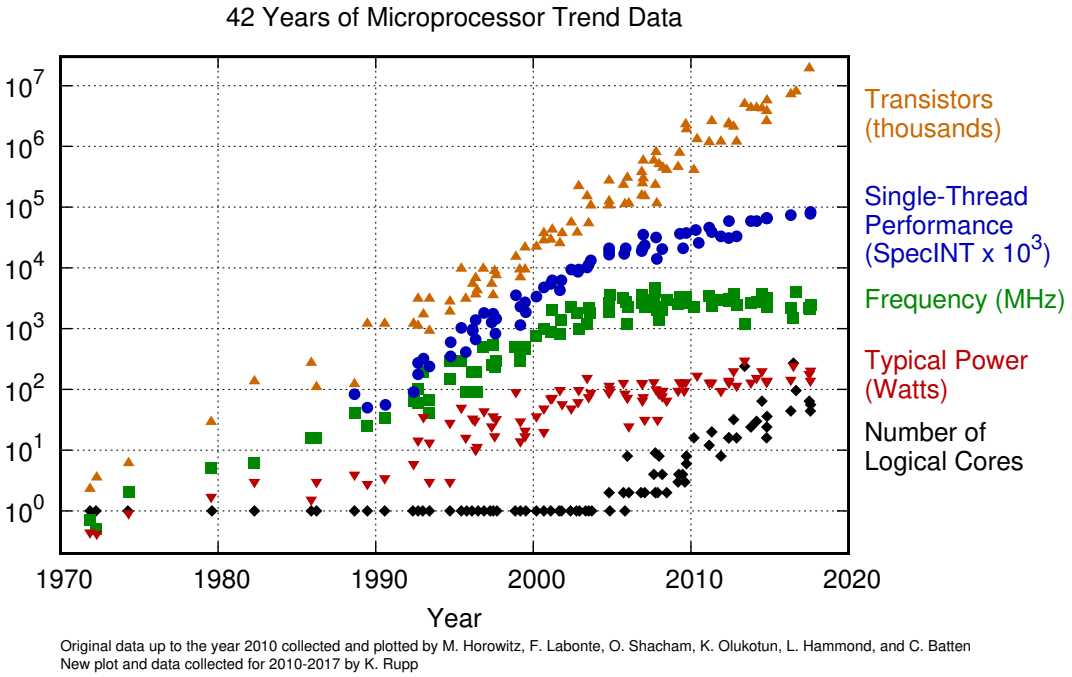


Figure 1.1: Transistor amount, single-thread performance, frequency, power and core amount of processors from 1970 until today [49]

side effects. The new smaller feature sizes lead to problems with chip reliability (e.g. electromigration). The consequences are distorted and broken conductor tracks. Due to smaller structure sizes and line widths, deformations cannot be compensated anymore [37].

Another related problem is that the constant reduction of feature size leads to the circuits becoming more and more sensitive to single event effects caused by radiation in the environment [29]. The radiation permeates the die and releases energy to the assemblies in the circuits. This can cause bit flips, short-circuits or the destruction of the underlying transistors.

The increasing integration density also influences the thermal dissipation of the micro-processors. The dissipation worsens with a reduced feature size [48] [50] and leads to hotspots in the chip. These thermal hotspots can cause damages to the circuits' assemblies.

The benefits of higher integration density and higher numbers of logical cores in a microprocessor raise the risk of failures in the chips. There could be either transient failures, which occur only temporarily and will disappear eventually (e.g. a bit flip), or permanent failures, that alter the chip in an irreversible

way. Broken conductor tracks are an example for such permanent failures. Anyway, both types of failures will influence the programs and software running on the microprocessors. The increased risk of failures can be countered by either hardened software or special hardware. In critical areas, e.g. real-time systems, redundancy is often implemented in order to deal with the failures (for instance in the Space Shuttle program [56]). Redundancy can solve the described problems but also raises the cost of the system. This is why intelligent solutions are necessary in order to cope with the continuing rise of logical core numbers and integration density. These solutions have to manage a certain number of failures in their underlying hardware without stopping or interfering in their actual tasks and purposes.

The second development, the IoT, connects many devices to the Internet which were not connected to another system before. The result is a large net of devices capable of sharing information and working together in order to fulfill complex tasks.

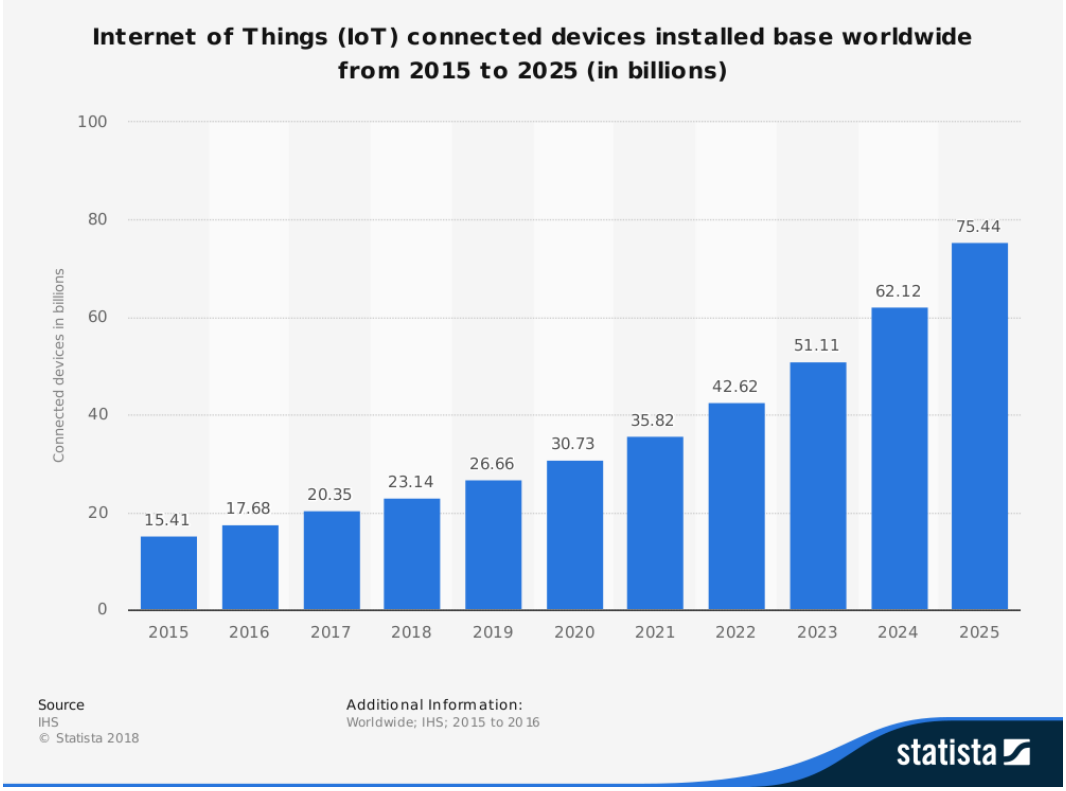


Figure 1.2: Number of worldwide IoT devices (in billions) from 2015 to 2025 (released in November 2016, years 2017 to 2025 are forecasts) [38]

It is expected that the number of IoT devices will continue to grow. In [38], the authors predict around 75.5 billion IoT devices for the year 2025 (see figure 1.2) which corresponds to a growth of nearly 500% within ten years. Additionally, the amount of internet traffic produced by machine-to-machine (M2M) communication will grow as well. The authors of [51] showed that the M2M traffic in cellular networks in the United States increased by more than 250% in the year 2011.

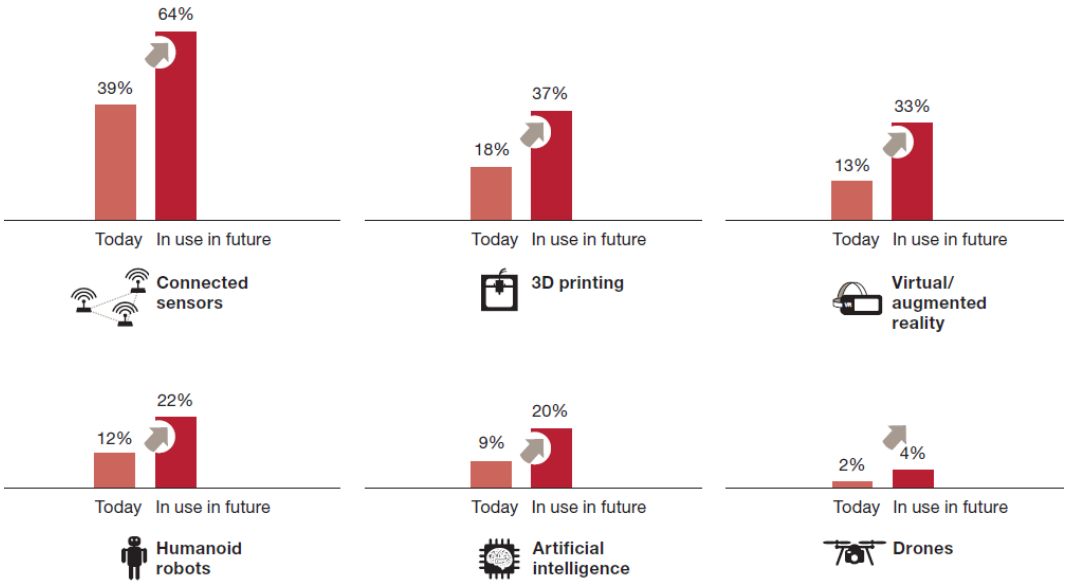


Figure 1.3: Results of a survey on already used and planned technology in factories (200 respondents in Germany, January 2017 - February 2017) [27]

Figure 1.3 shows the results of a survey of 200 respondents in Germany from 2017. They were questioned which technologies are in use or planned in their digital factory. The results show that the outstanding technology for factories are connected sensors, probably making them the most important technology in the future. Many sensors and devices in a factory will be connected in order to coordinate the production and improve the production process. These are the so-called “smart factories”, which will supersede the traditional factories.

The improvements in the field of parallel, ubiquitous and powerful computation will emerge into new applications, e.g. autonomous vehicles or “smart cities”. “Smart city” describes the architecture of collaboration of a city’s resources in order to provide good technical solutions for the citizens while simultaneously reducing the costs for public administration. In [67], an example

project for a smart city is given.

The interaction between many heterogeneous devices and its cores confronts us with new challenges in the fields of development, maintenance and expandability of such systems. Furthermore, the safety and security aspects of these highly distributed systems are critical. The challenge is the large complexity of systems consisting of hundreds and hundreds of cores working together, some being dependent and influencing each other. Additionally, solutions that cope with these challenges have to consider the dynamic aspects of such distributed infrastructures. Some of the devices in the network could be mobile devices which migrate between different domains of the system or even vanish completely from the network. Moreover, new devices can join the network and system along with new features and tasks.

In summary, it takes new paradigms and methods for future systems in order to handle such large, dynamic, heterogeneous and fault-prone distributed nets of processing cores.

Since 2002, the *Organic Computing Initiative* researches concepts for robust, scalable and fault-tolerant systems. Its primary aim is to develop solutions by implementing so-called *self-x properties*. These properties allow systems to become more autonomous and require less management and maintenance from human experts. In order to develop such systems, solutions from the nature are observed which solve comparable problems. Those techniques will then be mimicked and adapted to the technical problem.

Organic Computing, which is explained in detail in chapter 2, deals with many problems that cannot be handled by a single solution. This thesis focuses on one specific problem, namely the *task allocation in distributed multi-core processor networks*.

1.1 Task allocation in distributed multi-core processor networks

A large network of Processing Elements (PEs), which can be all kinds of computational cores (e.g. general purpose processors, signal processors, analog processors, timers, etc.) will also need to handle a large number of tasks. Otherwise, the majority of the PEs would be useless and could be taken out

of the system. In the networks described above, the number of tasks and PEs reaches a level of complexity that cannot be managed by human experts. A good distribution of all tasks among the PEs is not easily possible. A good task distribution is given when all tasks are assigned to PEs which are well suited for the corresponding task, e.g. an analog core processing an analog task or a special real-time core processing a time critical task. Thus, each PE has a certain suitability for each task in the system. In the following, the suitability will be discretized to a value called the *eager value*. The distribution of tasks in the system can be optimized by allocating each task to the best fit PE in the system thus resulting in a system performing in the best possible way.

The goal of the approaches presented in this thesis is to find such good distributions of tasks to PEs. It is not an option to use a central distribution unit having information about the system and its participants. As mentioned above, the reliability of single components cannot be guaranteed, so a central unit will introduce a single-point-of-failure. Redundancy of such a unit is expensive and, depending on the size of the network, several redundant units would be necessary. As mentioned before, the complexity of dynamic changes in such a large system cannot be managed by a single human user. Also, local changes that influence a PE's suitability for tasks might be hard to recognize for a central distribution unit. That is why this thesis concentrates on decentralized and autonomous solutions for the given problem of task distribution. A system which already deals with such kinds of problems is the Artificial Hormone System (AHS) [14]. The AHS was developed by *Brinkschulte et al.* and manages the task distribution in a network of connected heterogeneous PEs. It fulfills the requirements stated above, meaning that it is decentralized and autonomous. Further details of the AHS will be given in chapter 3.

Nevertheless, the AHS is only suitable for smaller scaled networks of PEs. Large scaled systems do not work properly since the amount of communication produced by the AHS is too high for most of the communication buses nowadays. Moreover, the AHS is not capable of handling a hierarchical topology of connected PEs. For this reason, this thesis deals with two new concepts, both based on the AHS, in order to face the challenges of large scaled and hierarchical networks of PEs. These two concepts are the Hierarchical Artificial Hormone System (HAHS) and the Recursive Artificial Hormone System (RAHS). The

HAHS is based upon the idea of running several separated AHS instances in parallel which partition the task set of the system among themselves. The separated AHSs are inter-connected by a superordinate AHS. The RAHS is a consequent extension of the HAHS supporting multiple levels of hierarchy: Several levels form a tree topology which can also be unbalanced. This system is predestined for hierarchical topologies which can be encountered in so-called "smart cities" or smart factories.

1.2 Goals and structure of the thesis

This thesis aims to develop and evaluate hierarchical self-organizing systems managing task-allocation in an autonomous and decentralized way. The base for this research is the AHS, which already provides this function for small-scaled systems. Large-scaled systems can be enhanced with autonomous task-allocation by adding hierarchical concepts to the AHS. Several concepts for hierarchical extensions will be presented, explained and evaluated.

The focus of the presented systems lies on the reduction of the system's data communication in comparison to the original AHS.

This thesis is structured as follows: At first, the basics are explained in chapters 2 and 3, focusing on *Organic Computing* and the AHS. The following two chapters 4 and 5 present the newly developed HAHS and RAHS with all of their concepts and methods. Chapter 6 analyzes the system properties of the HAHS and RAHS while chapter 7 briefly presents the developed software for the two systems. This software was used to evaluate the systems and their concepts. The evaluation results are presented and discussed in chapter 8. At last, the thesis compares the HAHS and RAHS to other systems in chapter 9 and concludes with a summary and outlook in chapter 10.

Chapter 2

Organic Computing

The description of *Organic Computing* used in this thesis relies on the description from [65]. Organic Computing is an approach to solve the problems arising from the increasing complexity in computerized systems, especially large nets of connected embedded systems. The complexity and the large number of connected cores make it more complicated to construct, maintain, use and develop for those systems. Organic Computing tries to overcome the gap between the complex technical system and the human designer, programmer, administrator and user.

Organic Computing is meant to enhance technical systems with "life-like" properties in order to construct those systems as robust, safe, flexible and trustworthy as possible [43]. The solutions are self-organizing systems adjusting themselves to changing environments [46].

The usual procedure of Organic Computing is to observe a mechanism in nature and transfer it to a technical system. For example, it can be observed how ants build a track to a food source by using pheromones [25]. This mechanism can be transferred to a set of robots working together.

As mentioned before, the benefits of Organic Computing for a system are increased flexibility, robustness as well as the ability to optimize itself all without any external control. In addition, the effort in designing a system decreases as not every potential system state has to be programmed [46].

The disadvantage is that those systems, like every learning system, can react in an unexpected way to unknown situations. Additionally, the high adaptivity could be used for a malicious influences on the system [46].

2.1 History of Organic Computing

The term Organic Computing appears for the first time in the context of a workshop in the year 2002. The result of this workshop was a position paper published by the German Computer Society (Gesellschaft für Informatik, GI) and the German Information-Technical Society (Informationstechnische Gesellschaft, ITG) [60]. In connection to the workshop, the Organic Computing initiative was founded in 2003. The initiative aims to research adaptive and self-organizing computers and computerized systems. Several projects included in priority programs were funded in the context of Organic Computing. Especially the priority program 1183, financed by the German Research Foundation (Deutsche Forschungsgesellschaft, DFG) led to a high research output from 2005 to 2011. The results from this priority program are summarized in [44].

2.2 Self-x properties

One of the most important principles of Organic Computing is the enhancement of systems with the so-called *self-x properties*. These properties describe functions and behaviors of the system which are usually achieved via external influence, e.g. by a user. A system implementing a *self-x property* will henceforth process the function or the behavior autonomously and without any external influence.

The *self-x properties* are:

- **self-configuration:** A system with the *self-configuration* property is able to find an initial start configuration for its components autonomously while taking the requirements and conditions of the system into account.

- **self-optimization:** The *self-optimization* property includes the ability to react to changing conditions. The system will react to changes in the environment by optimizing its current configuration.

- **self-healing:** When a part of a system fails, usually the functionality of this part will get lost for the system. A system with *self-healing* property is capable of coping with failure events like failing components. Without

any external interference it will reorganize its configuration such that the lost functionality will be restored if enough resources are still available.

- **self-protection:** In the context of attacking or exploiting technical systems the *self-protection* property enables the system to identify attacks. Moreover, the system can initiate countermeasures to stop the attack or to avert harm.

- **self-explanation:** A self-explaining system is able to reason about its decisions, which were made autonomously. This includes human-readable explanations of low-level mechanisms. Those are needed in applications observed and controlled by a human user. In these cases, those systems have to explain their reasoning for actions to the human user.

2.3 Emergence

An often used term in the context of self-organizing systems is *emergence*. While there exist many definitions for the term *emergence* [45], this thesis will focus on the so-called *quantitative emergence* which relies on the statistical definition of entropy by Shannon [52].

The entropy is a measure of structure for a system under observation. The statistical entropy value describes the amount of structure in the system. A high entropy value indicates less or no observable structures. A low entropy value, on the other hand, corresponds to a highly structured system. The entropy is calculated by choosing an attribute of the system and determining the probability of each possible state[1] ($z \in Z$) within this attribute.

$$H = -\sum_{z=0}^{Z} p_z \cdot \log_2 p_z \tag{2.1}$$

By means of the systems entropy the *quantitative emergence* can be calculated by ΔH of two states of the system under observation. For example, the emergence could be calculated by observing the change in entropy from the chronological beginning of the system to the chronological end of the system.

[1] in information theory, event is used instead of state (Shannon). However, state is used here since the states of the attributes will be observed

$$M_1 = H_{Start} - H_{End} \tag{2.2}$$

Since the start of the system is an arbitrary state, this definition of emergence is relative. Another more normalized emergence value can be calculated by subtracting the current entropy of the system by the maximum possible entropy, i.e. the state of maximal disorder (H_{Max}).

$$M_2 = H_{Max} - H \tag{2.3}$$

This definition of emergence can be used to calculate the degree of self-organization in organic computing systems.

2.3.1 Example for emergence

Two examples for calculating emergence in systems are presented in the following subsections.

2.3.1.1 Colored cells

The first example for emergence consists of a visualized 3x3 grid with a set of nine cells (C). Each cell in the grid has a blue intensity value (b) in the range from 0 to 255. Additionally the neighbor relationship of cells is given so that cell x has a set of neighboring cells ($N_x \subset C$). The neighbor relationship for this example will only concentrate on horizontal and vertical adjacent cells, not on diagonal adjacent cells. Over time, the blue intensity values of the cells will change by an unspecified mechanism. In order to measure the emergence of this system, an average distance measure for the cells is introduced (see equation 2.4).

$$d_x = \frac{\sum\limits_{i \in N_x} |b_x - b_i|}{|N_x|} \tag{2.4}$$

The example starts with an initial state (see figure 2.1). In this state, all cells have different color intensity values and the distances (d_x) are different, too. The range of the average distance values (d_x) have to be interpreted as the

states of the system when calculating the entropy of the system. The value of the average distance is between 0 and 255. Since all nine cells have a different average distance value, the entropy calculates to:

$$H_{initial} = -9 * (\frac{1}{9} * \log_2 \frac{1}{9}) \approx 3.1699 \tag{2.5}$$

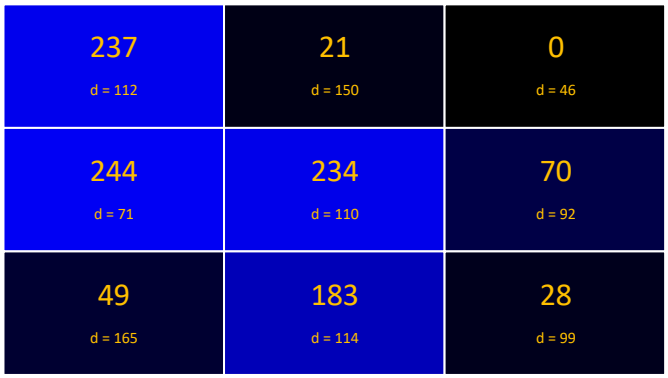


Figure 2.1: 3x3 cell grid initial state

The initial state corresponds to the maximum entropy since all distance values differ. Therefore, no structure is available and $H_{initial} = H_{max}$. Regarding equation 2.3 the emergence for the initial state of the example equals zero.

Assuming the system has ended (or paused) while all cells have exactly the same color (see figure 2.2), the average distances of all cells calculate to zero. Therefore, all cells are in the same state and the entropy calculates to:

$$H_1 = -1 * (\frac{9}{9} * \log_2 \frac{9}{9}) = 0 \tag{2.6}$$

This state equals a completely ordered system. The emergence for the system in this state calculates to $M_1 = H_{max} - H_1 \approx 3.1699$.

In figure 2.3 another state of the system is shown. This system consists of five cells having a blue value of 255 and four cells having a value of 127.

Even though the cells are divided in two groups, the system still reaches the entropy of $H_2 = 0$. The reason for the low entropy is that the average distances are all 128 and therefore all cells are in the same state. This again results in the high emergence as seen before.

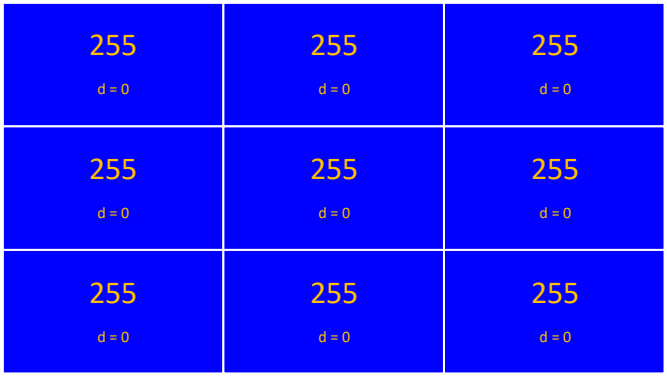


Figure 2.2: 3x3 cell grid first state

Figure 2.3: 3x3 cell grid second state

2.3.1.2 Traffic jam

Another example for emergence is a traffic jam. Figure 2.4 shows a circular course with vehicles on it. In [58] and [3] could be demonstrated that the vehicles in this system behave like interacting particles in a non-equilibrium system. Even without a bottleneck the system changes from a free flow state (without any jams) to a jamming state when the vehicle density exceeds a threshold. This is explained with small fluctuations in the movement of the vehicles. As long as the density is low enough, those fluctuations can be compensated. If the density rises over a certain threshold, the fluctuations can no longer be compensated anymore. Thus they accumulate, until a traffic jam

is formed.

This is a perfect example for emergence since it demonstrates that actions from autonomous individuals on the micro-level will eventually form to structures on the macro-level.

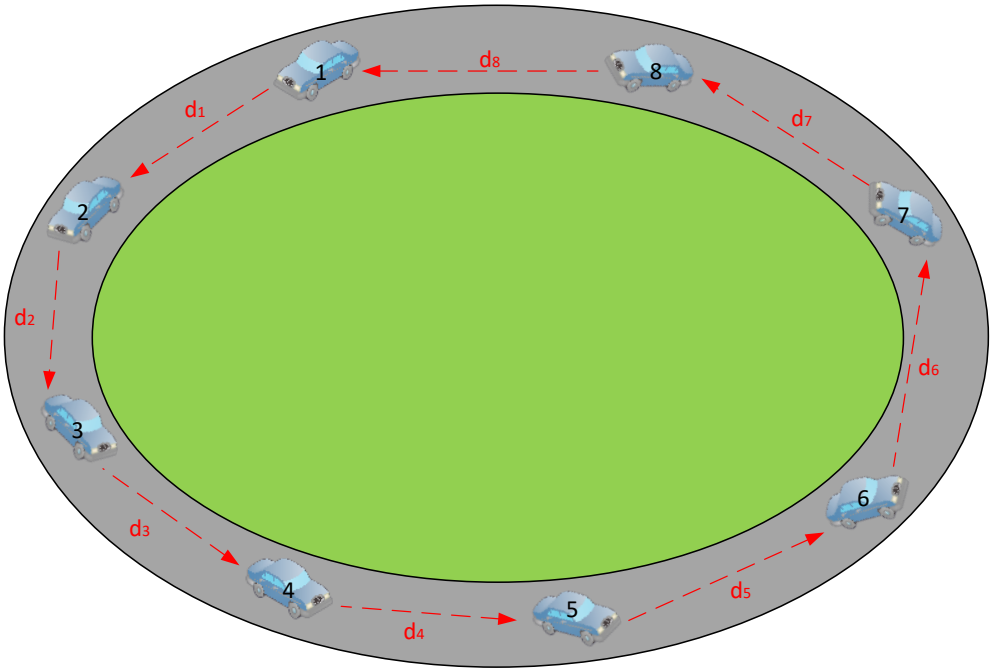


Figure 2.4: A circular course with cars. Each car has a distance (d_x) to the car in front of it

For calculating the emergence of this example, the distances between the cars have to be observed. For each car, the distance to the car driving in front of it is measured. The distances of each car to the next car will be classified according to table 2.1. Hence, each car will be assigned to a distance class.

Let's first regard a free flow situation with 8 vehicles on the track (see figure 2.4). The distances between the vehicles are:

$$d_1 = 39m \quad d_2 = 209m \quad d_3 = 229m \quad d_4 = 222m$$
$$d_5 = 257m \quad d_6 = 129m \quad d_7 = 157m \quad d_8 = 81m$$

The outcome is an almost equal distribution of distance classes (see table 2.2). It is observable that six distance classes only have one car assigned. One distance class ($D_{Class} = 7$) includes two cars.

D_{Class}	distance interval
1	$d_x \leq 50m$
2	$50m < d_x \leq 95m$
3	$95m < d_x \leq 135m$
4	$135m < d_x \leq 170m$
5	$170m < d_x \leq 200m$
6	$200m < d_x \leq 225m$
7	$225m < d_x \leq 245m$
8	$245m < d_x \leq 260m$
9	$260m < d_x \leq 270m$

Table 2.1: Distance classification

D_{Class}	1	2	3	4	5	6	7	8	9
Ex.1: #Vehicles p. class	1	1	1	1	0	1	2	1	0

Table 2.2: Distance distribution of the first scenario

Therefore, the entropy is calculated with six times a one eighth probability plus a two eight probability:

$$H_1 = -(6 * \frac{1}{8} * \log_2(\frac{1}{8}) + \frac{2}{8} * \log_2(\frac{2}{8})) = 2.75 \tag{2.7}$$

For calculating the emergence H_1 has to be subtracted of $H_{1_{max}}$, which in turn is the case when all cars are in different distance classes:

$$H_{1_{max}} = -(8 * \frac{1}{8} * \log_2(\frac{1}{8}) = 3 \tag{2.8}$$

$$M_1 = H_{1_{max}} - H_1 = 0.25 \tag{2.9}$$

In the second example, eight more vehicles are introduced to the track. This results in a higher vehicle density which eventually leads to a traffic jam. This

D_{Class}	1	2	3	4	5	6	7	8	9
Ex.2: #Vehicles p. class	6	3	3	2	0	0	0	1	1

Table 2.3: Distance distribution of the second scenario

is also visible in the class distribution (see table 2.3). A lot of vehicles now have a distance of 50 meters or less.

Similar to above, the entropy calculates by interpreting the number of cars in the distance classes as probabilities for those classes:

$$\begin{aligned} H_2 = &- (\frac{6}{16} * \log_2(\frac{6}{16}) + 2 * \frac{3}{16} * \log_2(\frac{3}{16}) \\ &+ \frac{2}{16} * \log_2(\frac{2}{16}) + 2 * \frac{1}{16} * \log_2(\frac{1}{16})) \\ &\approx 2.31 \end{aligned} \tag{2.10}$$

As before, the emergence will be calculated by subtracting the calculated entropy from the maximum entropy. For the maximum entropy the theoretical case of all cars assigned to different classes is assumed. In order to calculate this the distance classification (see table 2.1) has to be expanded to at least 16 classes.

$$H_{2_{max}} = -(16 * \frac{1}{16} * \log_2(\frac{1}{16}) = 4 \tag{2.11}$$

$$M_2 = H_{2_{max}} - H_2 = 1.69 \tag{2.12}$$

Even though the two entropy values of the scenarios are not directly comparable due to the different amount of cars, the emergence values can be compared. For this reason, we calculate the percentage of structure in the system by dividing the calculated emergence through the maximum emergence ($S_x = \frac{M_x}{M_{x_{max}}}$).

$$S_1 = \frac{0.25}{3} = 0.0833 \hat{=} 8.33\% \tag{2.13}$$

$$S_2 = \frac{1.69}{4} = 0.4225 \hat{=} 42.25\% \tag{2.14}$$

The calculated values for the structure (see equations 2.13 and 2.14) show that the system in the second scenario is more structured due to the traffic jam.

Chapter 3

The Artificial Hormone System

Today's distributed systems grow more and more complex. They often consist of a large number of heterogeneous PEs. Commonly, a middleware layer is introduced for those systems in order to manage the cooperation of tasks and to hide the distribution to the application. Such a middleware is complex and hard to manage. Therefore, self-organizing techniques (see chapter 2) can help building such systems. Finding an initial task allocation is one of the jobs of the middleware. Furthermore, it should adapt or optimize the configuration to changes in the environment and internal states, heal itself in case of failures and finally protect itself against malicious attacks.

The AHS implements such a middleware and organizes the task allocation amongst a set of PEs. No central decision-making unit exists in the AHS. Each PE decides upon basic rules and information received from other PEs. As a consequence, the AHS is *completely decentralized*. Additionally, no external organization can influence or affect the task-allocation, which means that the AHS is *self-organizing*. The system was named *artificial hormone system* since it is inspired by the endocrine system of higher mammals. Hence, the first section in this chapter will summarize the properties of the endocrine system. Afterward, the structure of the AHS, as well as its *hormone loop* will be explained in detail. Furthermore, the developed extensions of the AHS, the system's communication load, and time constraints will be presented in this chapter. The time constraints also prove that the system can be used in *real-time* environments.

3.1 The biological endocrine system

The AHS is inspired by the endocrine system of higher mammals. Accordingly, this section briefly explains the functionality of the biological system. The endocrine system serves as a global communication system in the body which coordinates actions between the cells of the body [30]. For this reason, the endocrine system emits hormones in the blood stream. This realizes global communication. For local communication, cells can also produce hormones and sent them via the tissue to neighboring cells. The hormones are produced by endocrine glands and work as messages for the cells. Often the hormones realize regulation cycles between organs: certain hormones favor the production of particular substances and others inhibit this production. A cell receives a hormone via a molecule on the cell membrane, the so-called hormone receptor, which only binds to one specific hormone. Not all cells have hormone receptors for all hormones, in that way not all cells in the body will be effected when a hormone is emitted. The concentration of the hormones determines the reaction of the cell, too. Sometimes a certain threshold of concentration is needed to trigger a reaction of the cell.

In summary, the endocrine system works in a decentralized way. Cells work together on tasks in an independent and autonomous way. The coordination of the tasks is managed by the hormones and the cells influence each other by emitting hormones as well. No cell is in charge of coordinating tasks. By this, the endocrine system is robust against failing cells.

3.2 Formal definitions

In this section, the basic formal definitions of the AHS will be defined. An AHS consists of n PEs applying for m tasks in the system. For *self-optimization*, the PEs will offer tasks after a defined period of cycles. The period is denoted by w. Messages sent between the PEs are called hormones and are used for the *self-organization* of the task-allocation among the PEs. It exists a difference between sent hormones of a PE γ from a task i and received hormones which were sent by a PE γ for a task i. Sent hormones will be denoted with subscripted indices, for example $E_{\gamma i}$, while received hormones will have superscripted indices, e.g.

$E^{\gamma i}$.

3.3 Structure of the AHS

Adapting the endocrine system to a technical system is complex because of the limited resources in the technical system. Therefore, the principles of the endocrine system were simplified in the AHS to keep the main properties of *decentralized control of cell activity* and *fault tolerance*. The following abstractions were made:

- A *cell* is represented by a *PE*. The *PE*, in turn, can be represented by, e.g. a processing core, an analog core or a graphics processing unit.
- A *cell activity* is represented by a *processing task*.
- *Hormones* are represented by *messages* sent via the network between the PEs and tasks.

All PEs constantly send out hormones to all other PEs (broadcast) or to neighboring PEs (multicast) including information for which task the PE applies.

Additionally, PEs, that decided to allocate a task, will inform the other PEs about the allocation by broadcasting a hormone, too. Those hormones effect the above-mentioned hormones by a negative feedback loop. This is similar to the endocrine system. Hormones from tasks in the AHS only effect those PEs which are interested in the task.

A failure of a PE in the AHS can be compensated as long as there still exist PEs which are able to allocate the tasks of the failed PE. Therefore, the AHS realizes the *self-healing* property and shares this ability with the endocrine system. With the absence of negative hormones from the task which was allocated on the failed PE, the re-allocation will be regulated autonomously.

A difference between the endocrine system and the AHS is the intensity of hormones. In the endocrine system, the intensity of an effect induced by a hormone is determined by the quantity of the hormone in the blood stream. In the AHS the intensity cannot be determined by the quantity since the bandwidth of the communication network is limited. Therefore, the intensity

Subtype	Symbol	Description
Local eager value	$E_{i\gamma}$	The *local eager value* represents the predefined suitability of the PE γ to the task i
Modified eager value	$Em_{i\gamma}$	This value indicates the current suitability of PE γ to task i, after the received accelerators and suppressors were added respectively subtracted.

Table 3.1: Eager value types

is determined by the quality of the hormone. This means, only one hormone will be sent periodically to all PEs. The hormone consists of a value, which indicates the intensity of the hormone in the system.

In order to implement the negative feedback loop, the AHS needs different hormones, which can influence each other.

The first kind of hormones are the so-called **eager values**. These hormones indicate the ability of a PE to process a specific task at the exact point in time. The PE sends out those *eager values* for each task the PE is interested in. The value or intensity of the *eager value* determines the suitability. The higher the *eager value*, the more suitable it is for this task. There exist exactly two subtypes of eager values (see table 3.1).

As stated before, the AHS, like the endocrine system, works with negative feedback loops. Therefore, an antagonistic hormone to the *eager values*, the so-called **suppressors**, is implemented. Those *suppressors* effect the suitability of the receiving PEs to a specific task. Thus, the received *suppressor* will be subtracted from the suitability value of the PE, which is the *eager value* of the task. *Suppressors* can have their origin from several events or conditions (see table 3.2). Most commonly they will be used to limit the instances of tasks in the system. This is achieved by the *acquisition suppressors*. Another important suppressor is the *load suppressor*, which prevents a PE from allocating more tasks than it can process.

The last type of hormones are the **accelerators**. Those hormones will be added to the suitability of a PE for a task. The *accelerators* favor the allocation of a task to a PE. Similar to the suppressors there are some subtypes of accelerators (see table 3.3). The *stay accelerator* will be used in the context

Subtype	Symbol	Description
Acquisition suppressor	$Sa_{i\gamma}$	The *acquisition suppressor* is sent to all other PEs as soon as the task i is allocated.
Load suppressor	$Sl_{i\gamma}$	The *load suppressor* will also be sent as soon as the PE allocates task i. In contrary to the acquisition suppressor, this hormone will only be sent to the PE itself. It represents the load produced by the task.
Monitoring suppressor	$Sm_{i\gamma}$	This hormone is also sent only locally, and can be used to represent temperature, energy level or similar influences on the PE.

Table 3.2: Suppressor types

of self-optimization. It will be emitted from a PE when a task is offered for optimization: Periodically (period w), a PE allocating a task will stop sending the *acquisition suppressor* for this task for a defined amount of hormone cycles (d). In this time, the other PEs get the chance to allocate the task if they have become more suitable. The *stay accelerator* will only be sent to the original PE itself and not to other PEs in the system. It increases the modified eager value by a small amount x. Other PEs applying for this task have to have an *eager value* which is at least $x + 1$ greater than the eager value of the offering PE. Accordingly, the *stay accelerator* represents the costs of a task migration in the process of a self-optimization. To move a task to a new PE, the *eager value* of the PE has to exceed the *eager value* of the old PE plus the *stay accelerator*. The *organ accelerator* will be multicasted to neighboring PEs and favor allocations of related tasks in the neighborhood. Thereby, the communication distance of the related tasks will be reduced.

An example overview of the hormones and their different effects on the PE is given in figure 3.1.

Subtype	Symbol	Description
Stay accelerator	$As_{i\gamma}$	The *stay accelerator* will be emitted locally, when a PE offers a task. This prevents that tasks migrate too easily in their offer phase.
Organ accelerator	$Ao_{i\gamma}$	In most applications tasks will work together and have to communicate with each other. To prevent long communication distances, the *organ accelerator* can be used to allocate those tasks closely together. It will be send to all neighboring PEs and affect their suitability for the related tasks.
Monitoring accelerator	$Am_{i\gamma}$	This hormone is the counterpart of the monitoring suppressor. It represents influences on the PE which can favor the allocation of tasks on the PE.

Table 3.3: Accelerator types

3.4 The hormone loop

As stated in the section before, each PE sends out hormones periodically. This is regulated by the *hormone loop*, which is a control loop running on each PE (see figure 3.2). The *hormone loop* consists of three phases:

- In the **receiving stage**, the PE receives all hormones from all other PEs in the system. The PE accumulates these received hormones for each task separated by the three hormone types (*eager values, suppressors, accelerators*).

- The second stage is the **computation and decision stage**. In this stage the *modified eager value* for each task will be calculated (see table 3.1). This is done by subtracting all received suppressors for this task and all received global suppressors from the initial eager value of the task. Afterward, all received accelerators for the task and all received global accelerators will be added. The result is the *modified eager value*. This calculation is made for every task the PE γ is able to allocate. The decision part will only be applied to one task per cycle. Therefore, one task will be chosen from a sequential order. Then, the calculated *modified*

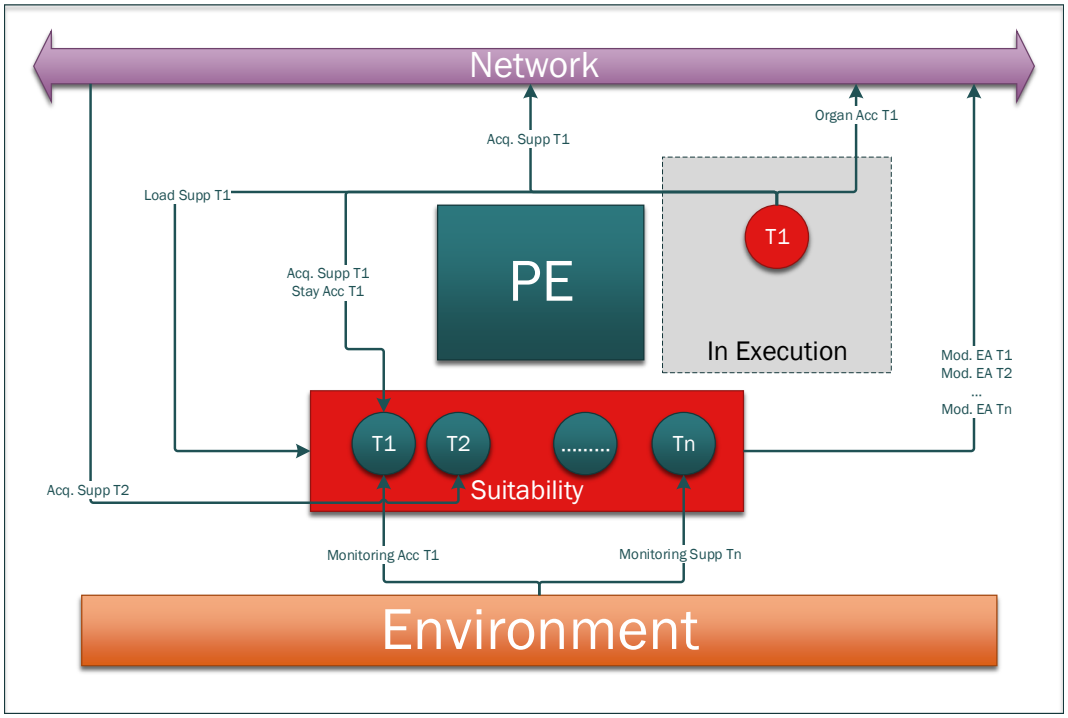


Figure 3.1: The effects of the different hormones on a PE (example)

eager value of this task will be compared to all received *eager values* of all other PEs. If the own *modified eager value* of PE γ is the highest of all compared *eager values*, PE γ knows it is currently most suitable for this task. Thus, it decides to allocate the task. All PEs operate on the same hormones, this means they share the same knowledge about the system, and all PEs will result in the same decision about a task. Hence, all other PEs which also chose the same task for decision will come to the result that PE γ is the most suitable for the task and will not allocate it[1].

- The last stage in the hormone loop is the **sending stage**. In this stage the PE emits the hormones into the system, e.g. the *modified eager values* which were calculated in the stage before. *Suppressors* and *accelerators* will also be sent out in this stage. In contrast to the decision part of the stage before, this stage will be proceeded for every task in which the PE is interested.

As soon as the PE completes all three stages, the hormone loop is completed

[1]In order to avoid race hazards between the PEs, the hormone loop period must be at least twice the communication time between the PEs [14]

and the next cycle of the hormone loop starts. This hormone loop forms a distributed decentralized control loop.

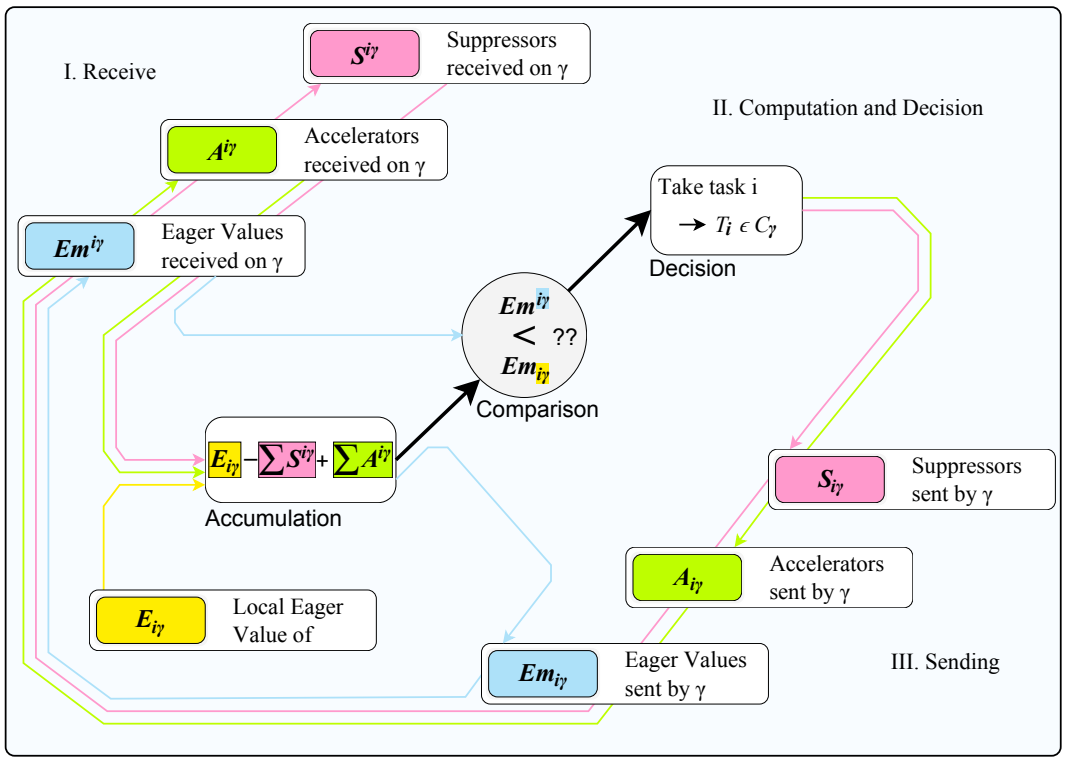


Figure 3.2: The hormone loop of the AHS [9]

3.5 Extensions of the AHS

While researching the AHS several extensions were developed in order to improve certain aspects of the AHS.

3.5.1 Priority decision

A first extension of the AHS, the *priority decision* decreases the upper bound of the worst-case timing in the task allocation. Hence, the PEs actively observe the *eager values* of their tasks. The PEs will decide on a task whose *eager value* has increased instead of deciding on the next sequential task from the task list. This approach also takes place if the task is currently offered. It reduces the upper bound for the worst-case timing of the task allocation from

m^2 to $2m-1$ hormone cycles, whereby m is the number of tasks in the system. Further information to this extension can be found in [14].

3.5.2 Aggressive task allocation

Another extension reducing the worst-case timing for the task-allocation, is *the aggressive task allocation strategy* [13]. It is a further improvement to the priority decision (section 3.5.1). In contrast to the original AHS, every PE tries to find a task to allocate in every hormone cycle. Hereby, the PE iterates through its own task list to find a task for which it is the most suitable PE. Like in the original system, the PE will still only decide on one task in each hormone cycle. The worst-case timing for the task-allocation is reduced to m hormone cycles by this strategy, whereby m is the number of tasks in the system.

3.5.3 Virtual accelerators

Another technique to save communication load is the *virtual accelerator* [11]. In order to reduce the communication time between related tasks, the AHS tries to allocate them to adjacent PEs. This is managed by sending out accelerators to all neighbor PEs for all related tasks as soon as a task is allocated at a PE. However, the accelerator hormones produce additional communication. To avoid this additional communication overhead, *virtual accelerators* can be used. Due to the fact that the PE already sends out the acquisition suppressor, the receiving neighbor PEs can deduce those accelerators for all related tasks by themselves. Upon receiving the *acquisition suppressor* for a task the PE looks up which tasks are related to this task and if the origin PE is in its neighborhood. In order to substitute an accelerator by a *virtual accelerator* the PE needs to know the strength of the accelerator for the task. In fact, this value is initially distributed to the PEs with the task definition. Therefore, no change has to be made to the system to meet this requirement.

3.5.4 Lightweight AHS

In its original implementation, the AHS needs an own list for received hormones for every task (n) on every PE (m) and every hormone type (h). Those lists

have to be as big as the amount of PEs in the system itself. This results in $n \cdot m \cdot h$ lists with a size of m. It is imaginable that in bigger scaled scenarios this occupies a lot of memory space. Therefore, a **lightweight** variant of the AHS was developed in [61]. This variant sums up all received accelerators and suppressors for each task and saves only the recent and current sums. It also only stores the highest received eager value for each task of the recent and the current hormone cycle.

Unfortunately, this variant comes with a problem. Due to asynchronous operation of the PEs, it is possible that hormones are not received at all or received twice by a PE within a hormone cycle. This is compensated through the saving of all received hormones in lists. The *lightweight* variant dispenses this additional saving of all received hormones. Therefore, the variant needs a mechanism to ensure that all PEs work synchronously. This is achieved by autonomously adjusting the hormone cycles of the PEs. In the original implementation, after the decision phase has ended, all PEs wait for the same amount of time before starting with the sending phase. In the *lightweight* variant, the PE which first finishes the waiting time will immediately start with the sending phase. All other PEs that receive those hormones will start immediately with their sending phase, too, even though their waiting time might not be finished. In this way, the AHS synchronizes itself in every hormone cycle.

3.6 Time constraints of the AHS

As mentioned before, some worst-case timing behaviors could be estimated for the AHS and also could be improved by extensions. This section summarizes and separates them for the different self-x properties. Those estimations show that the AHS in principle is capable of being used in a real-time environment.

The first worst-case timing estimation is for the *self-configuration*. During the development of the AHS, this worst-case timing could be improved by the *Priority Decision* (see section 3.5.1) and the *Aggressive Task Allocation* (see section 3.5.2). With those two extensions the worst-case timing of the self-configuration(t_{SC}) calculates to equation 3.1.

$$WCT_{SC} = m \tag{3.1}$$

The second self-x property to investigate is the *self-optimization*. When a PE optimizes its suitability to a task and becomes the best PE for the task, it will send its updated eager values in the next self-optimization cycle of the AHS. Assuming, an allocated task will be offered for optimization in a defined interval w, the task then will be migrated to the more suitable PE after w cycles at the latest. Provided that every task is optimized at the same time, the worst-case time for self-optimization (t_{SO}) is aligned on the migration of the task whose offer phase ended in the cycle before. This concludes to a worst-case time:

$$WCT_{SO} = w + m \tag{3.2}$$

The last self-x property is the *self-healing*. This property takes action when a PE in the system fails. It will allocate the tasks which were lost due to the PE failure. The missing *acquisition suppressors* of the failing PE will no longer suppress the eager values in the remaining PEs. Therefore, the PE with highest eager value of the remaining PEs will allocate the missing tasks immediately after the last *acquisition suppressor* expired (expiration time of hormones: e). The worst-case is the case of a failing PE which allocated all tasks. When this PE fails all tasks of the system will become free for allocation. This equals the self-configuration scenario and therefore the *self-healing* has a worst-case timing (t_{SH}) of:

$$WCT_{SH} = m + e \tag{3.3}$$

3.7 Communication load of the AHS

The communication load produced by the hormones of the AHS scales with the amount of PEs and tasks in the system. Increasing one of those, leads to a drastically increased communication load, especially at the start of the

system. In the *self-configuration* phase, all PEs will send out their eager values for all available tasks. At the start of the system, all tasks will still be available. Therefore $n \cdot m$ eager values (n = PEs, m = tasks) will flood the communication network. Each time a PE allocates a task and sends out an *acquisition suppressor*, n eager values will vanish in the next hormone cycle. Section 3.6 shows that in worst-case only one task will be allocated by the system in each hormone cycle. This concludes to equation 3.4 for the numbers of eager values (EAs) over the self-configuration time, starting for $t_{SC} = 0$ to $t_{SC} = m$ when the system is fully configured ($t_{SC} \in \{0, m\}$).

$$|Em(t_{SC})| = n * (m - t_{SC}) \tag{3.4}$$

For a full communication load analysis, the *acquisition suppressors* (S_A) have to be considered, too. They rise with the number of allocated tasks in the system (see equation 3.5).

$$|Sa(t_{SC})| = t_{SC} \tag{3.5}$$

Combining these two values leads to an equation for calculating the communication load (C) of an AHS during any hormone cycle in the self-configuration (see equation 3.6).

$$C_{SC}(t_{SC}) = n \cdot (m - t_{SC}) + t_{SC} \tag{3.6}$$

The worst-case communication load occurs at $t_{SC} = 0$, the immediate start of the system, when no task is allocated yet (equation 3.7).

$$WCC_{SC} = n \cdot m \tag{3.7}$$

In figure 3.3 the self-configuration communication load for several symmetrical ($n = m$) AHS configurations is visualized. It is visible that the critical point in terms of communication load of the self-configuration is the start of the system. It also shows that scaling of a system becomes problematic since the communication load grows with $n \cdot m$.

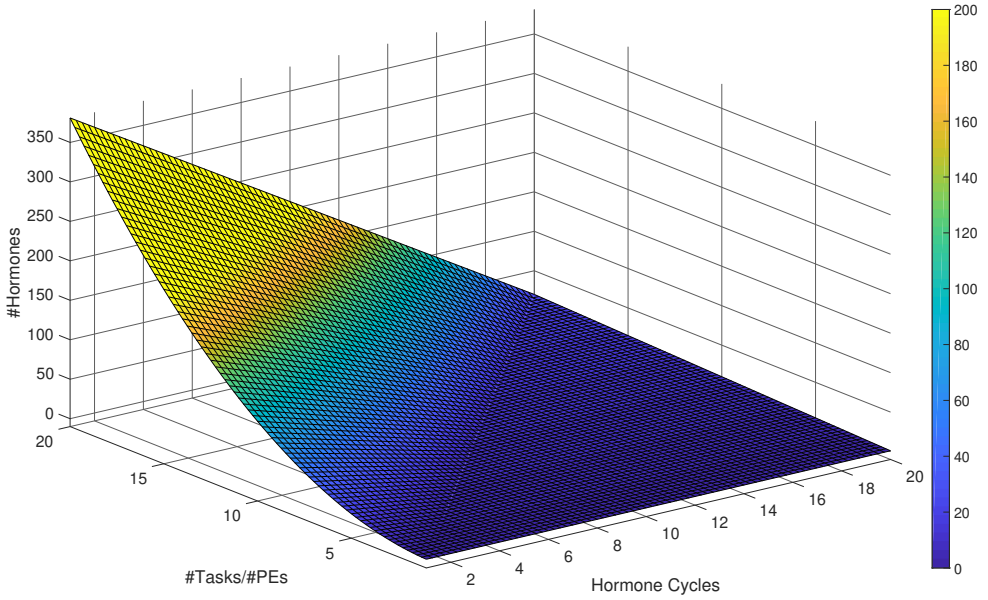


Figure 3.3: Communication load (by means of hormone amount) of different symmetric AHS configurations over self-configuration time

Including the *self-optimization* cycles, the critical communication phase still stays the same (see figure 3.4). The *self-optimization* cycles produce much less communication load in comparison to the *self-configuration* phase.

For calculating the needed bandwidth in a real-world implementation, the size of the eager values respectively suppressors (S_{EA} and S_{Supp}) in bits and the duration of the hormone cycle (D_H) in seconds have to be known. From implementation we know the typical hormone size of 50 bits and the typical hormone cycle duration of 100ms.

Table 3.4 shows the needed bandwidth of several PEs and task combinations in MBytes per second. The bandwidth needed for the configuration of 150 PEs and 150 tasks is already too high for a 10 MBit/s Ethernet. Moreover, the calculated bandwidth is only for the hormone communication. Any payload produced by the tasks will raise the needed bandwidth, too.

In [47] the needed bandwidth for an AHS with related tasks[2] was calculated. In the example the AHS has only 100 PEs and 50 tasks and results in a needed

[2]The related tasks (tasks which work together on one job and communicate together) will produce additional accelerators. Accelerators have a size of 66 Bit.

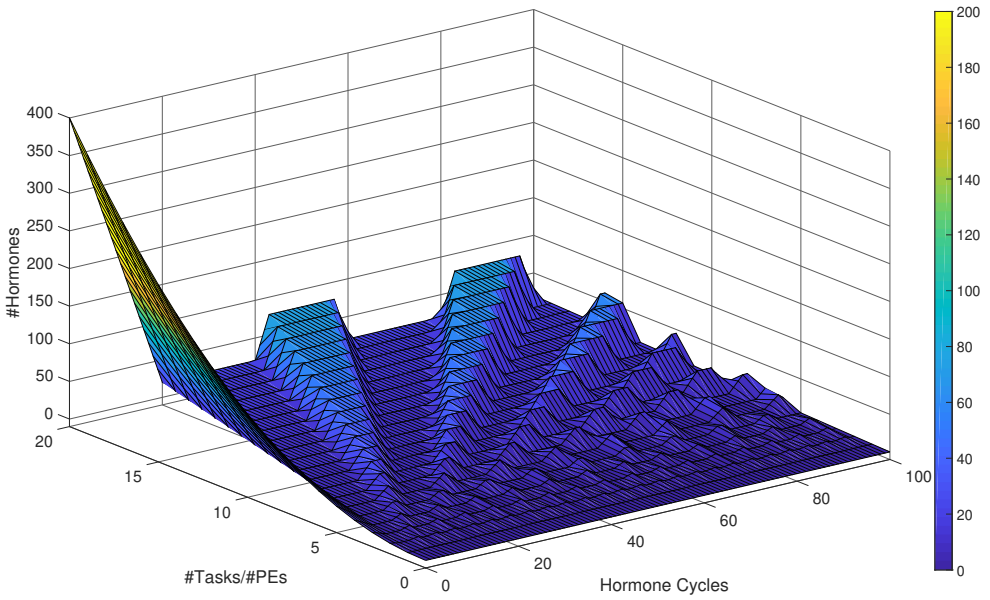


Figure 3.4: Communication load (by means of the numbers of hormones) of different symmetric AHS configurations over the entire time (including self-optimization cycles)

bandwidth of 1.7 MBytes/s.

These calculations show that the AHS works well in small scaled scenarios, but leads to high communication loads when scaled in large dimensions (in terms of PE and/or task amount). Especially, the broadcasted eager value amount in the self-configuration phase is a problem. In order to support large scaled scenarios, techniques for reducing this amount have to be found.

PEs\Tasks	**50**	**100**	**150**	**200**	**250**
100	312.5	625	937.5	125	1562.5
150	468.75	937.5	1406.25	1875	2343.75
200	625	1250	1875	2500	3125
250	781.25	1562.5	2343.75	3125	3906.25

Table 3.4: Needed bandwidths (in KBytes/s)

Chapter 4

The Hierarchical Artificial Hormone System

As presented in chapter 3, the AHS is used to handle the task allocation among a set of heterogeneous PEs. In large scaled scenarios, the AHS communication load grows rapidly. Especially, the broadcasted eager values in the *self-configuration* phase flood the network. For reducing this high communication load, the AHS was extended towards the HAHS. The *eager values* in the *self-configuration* phase can be reduced by introducing a hierarchical level above the original AHS and grouping PEs together to clusters. While the main focus in the development of the HAHS lays on the reduction of the communication load, the flexibility and adaptivity of the AHS should also be kept. That is why, different concepts in flexibility as well as different methods for determining *eager values* in the hierarchical upper level were researched and developed.

This chapter explains the hierarchical extension of the AHS and presents the different concepts of the HAHS.

4.1 From the AHS towards the HAHS

The HAHS is the two level extension of the AHS. The goal of the HAHS is to reduce the communication load in higher scaled application scenarios. In order to reach that goal, the number of PEs sending eager values for each task is reduced. Especially, in the communication-rich self-configuration phase, a

reduction of communication can be helpful. Therefore, the PEs will be grouped into disjoint clusters. Each cluster is separated from the other clusters, thus no PE of a cluster can communicate with a PE from another cluster (see figure 4.1).

With regard to the goal of reducing the amount of sent eager values, the HAHS builds disjoint task-subsets. Each task-subset represents the task set for the separated AHS in each cluster. Thus, for each task only a small group of PEs send its *eager values*. Assuming the HAHS consists of n PEs split into l equal sized clusters, the complete task set (with m tasks in it) will then be divided into equal sized task-subsets. The size of a cluster is $\frac{n}{l}$ and the size of a task-subset[1] is $\frac{m}{l}$. Every cluster will broadcast $\frac{m}{l} \cdot \frac{n}{l}$ *eager values* at the beginning of the *self-configuration*. The overall number of *eager values* in the beginning of the *self-configuration* phase calculates to $l \cdot \frac{m}{l} \cdot \frac{n}{l}$. So the number of broadcasted *eager values* decreases by a factor of l in comparison to the AHS ($n \cdot m$; see section 3.7).

To build and deploy the task-subsets among the clusters in a decentralized manner, a second communication level between the clusters has to be established. For this purpose, each cluster has exactly one special PE representing the cluster, the so-called **cluster head**. The cluster heads will negotiate via the second communication level to create the task-subsets which will be deployed to the clusters.

4.2 The hormone loop

Since the HAHS is based on the AHS, the hormone loop of the AHS (see section 3.4) is utilized in the HAHS. The AHS hormone loop (see figure 3.2) will be left almost unchanged as hormone loop for the *intra-cluster* cycle. The only difference is that tasks have to be unlocked for allocation before *eager values* will be sent from the PEs to the task.

For the cluster heads, a second hormone loop for the *inter-cluster* hormones have to be implemented. Again, the hormone loop of the AHS is used for this.

[1] Of course the resulting number of clusters and task-subset sizes can be non-natural numbers. In this case one cluster/task-subset will receive one PE/task more than the others. This will still be assumed as "equally distributed".

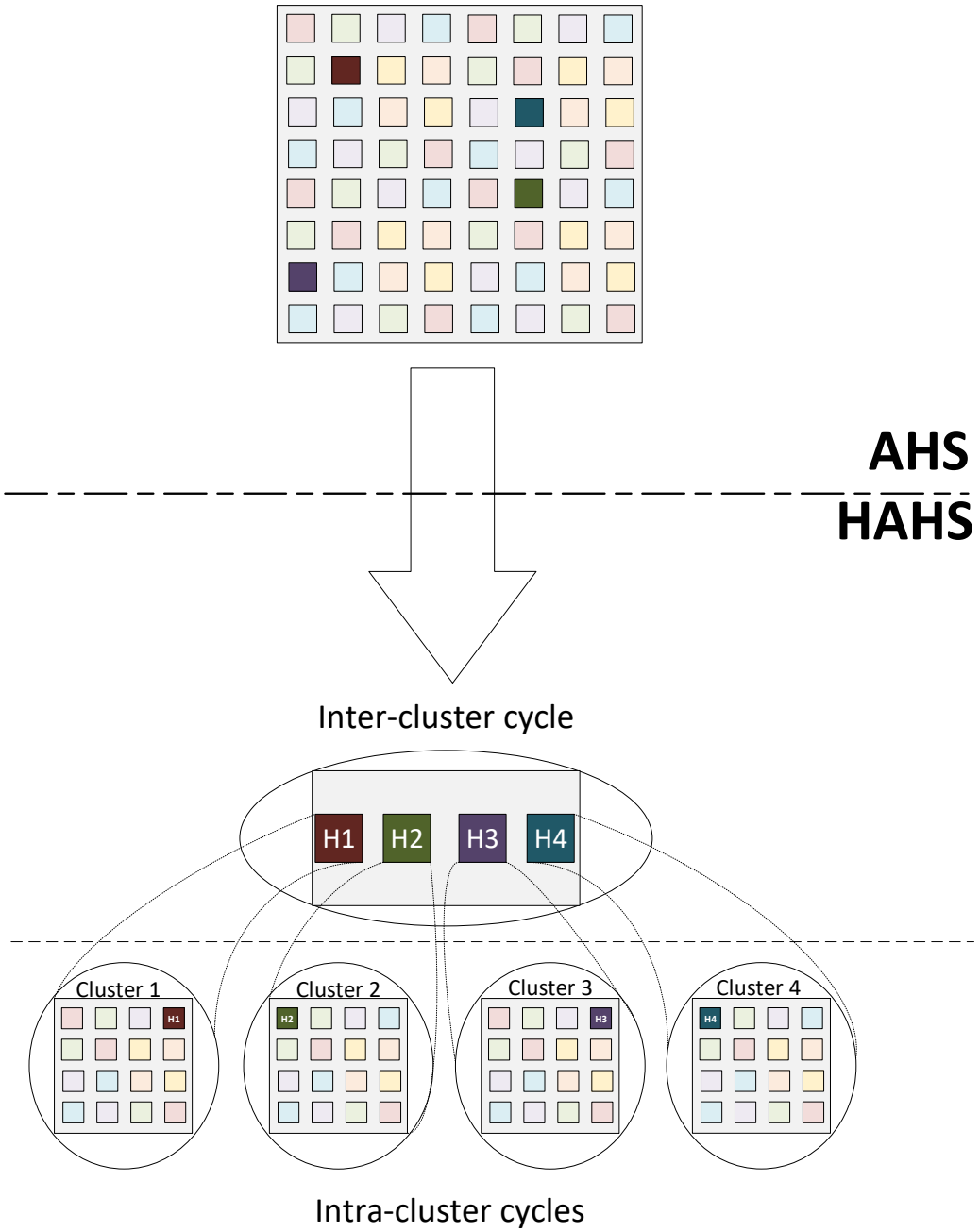


Figure 4.1: The hierarchical extension of the AHS *(top)* to the HAHS *(bottom)*: The set of PEs will be split into disjoint clusters. Each cluster gets exactly one cluster head *(marked with H_x)* which represents the cluster in the *inter-cluster* cycle.

Again, it will stay almost unchanged, with a slight difference: when a task is won in the *inter-cluster* cycle, a special *notification hormone* is broadcasted

to the *intra-cluster* cycle. The hormone causes the unlock of the task in the *intra-cluster* cycle. The interaction of both loops is visualized in figure 4.2.

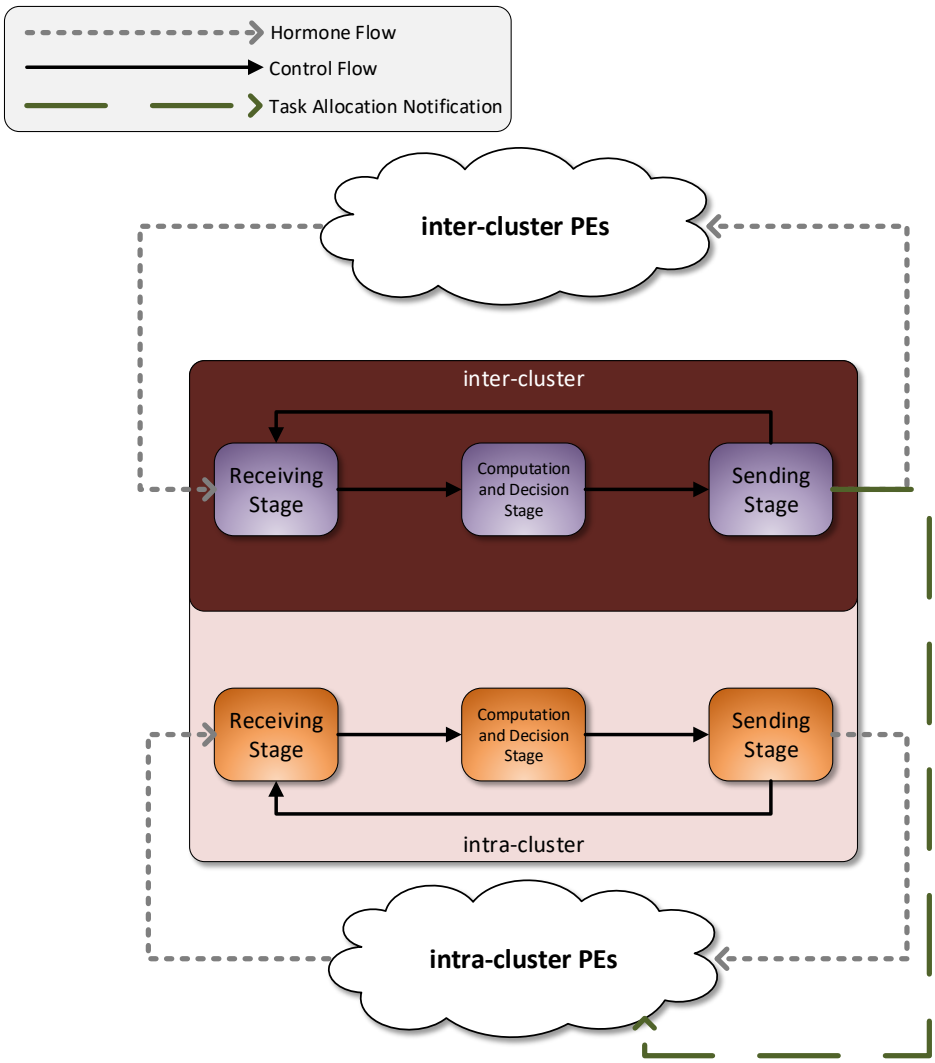


Figure 4.2: The hormone loop of the HAHS for a cluster head PE. It consists of two separated AHS hormone loops: one loop for the *inter-cluster* cycle and the other for the *intra-cluster* cycle. In the basic concept, they are only connected by a *notification hormone* which unlocks a task in the *intra-cluster* cycle.

This figure describes the basic principle of the two separated loops and their interaction. Details vary according to the chosen task concepts, cluster concepts and other different methods. These details are explained and visualized in the corresponding sections.

4.3 Cluster heads

The cluster heads represent a cluster. Each cluster has to have exactly one cluster head. All cluster heads together form the *inter-cluster* cycle. In this cycle, the task-subsets will be negotiated. Those task-subsets then will be activated in the clusters (*intra-cluster* cycles).

It is important that all cluster heads are interconnected. In case of a full-meshed or bus network this requirement is trivial. In case of separated networks for the clusters, a network connection between the cluster heads has to be ensured.

How to select a cluster head from the cluster members will be discussed in section 4.6.

4.4 Task set concepts

As stated before the overall task set is divided into several subsets which then will be distributed to the clusters. There exist two different methods in order to determine the task subsets.

4.4.1 Organ Task Concept

The first method is the so-called *organ task* concept. In this concept, the tasks will be grouped into subsets by the designer of the system. In this context, the subsets are called *organs*. The grouping of the tasks can be performed by their functionality, e.g. grouping similar tasks or communicating tasks. This grouping is predefined and cannot be altered at runtime. For each *organ* exists a representative task, the so-called *organ task*. This task is handled in the *inter-cluster* cycle.

The cluster heads apply for the *organ tasks*. If a cluster head allocates an *organ task*, it notifies its cluster members to start their *intra-cluster* cycle for all tasks in the corresponding *organ*. The advantage of this method is a lower overhead since only the *organ tasks* are negotiated in the *inter-cluster* cycle. This leads to less sent *eager values* and *suppressors* in the cycle. The second advantage of this concept is the reduced allocation time. As stated before, only the *organ tasks* have to be distributed amongst the cluster heads. This

can be done faster than distributing all tasks separately. The disadvantage of this concept are the static organs. They cannot be altered during run-time and therefore the system cannot react to dynamic changes in the suitability of the PEs to the single tasks in the *organs*. If, for example, a PE in a cluster fails and the affected task cannot be allocated anywhere else in the cluster, the *cluster head* must react by reducing its suitability for the entire *organ*. By doing this, the *organ* will probably be migrated to another cluster even though most of the tasks were functional in the original cluster.

In this concept, the size of the organs is crucial and must be chosen wisely by the designer. The smaller the organ size the longer the allocation will take and the higher the communication load will be. Designing bigger organ sizes can lead to systems in which one PE failure leads to a functionality failure, for example when an organ can neither be fully processed in the original cluster nor migrated to another cluster. This would result in an incomplete task distribution, even though the complete set of PEs could process all tasks.

4.4.2 Single Task Concept

The opposite of the *organ task* concept is the *single task* concept. In this concept, every task is treated separately in the *inter-cluster* cycle, too. This means that every task will first be assigned by a *cluster head* to its cluster and afterward allocated by a PE in the cluster. That corresponds to an *organ task* concept with only one task per organ. Of course, this takes more time than the *organ task* concept and the communication load increases as well. In return, this concept can react more dynamically to changes in the clusters than the *organ task* concept (e.g. when a single PE fails, its tasks can migrate to another PE in the same cluster or to a PE in another cluster without affecting the remaining task allocation).

4.5 Formal definition

In this section the formal definitions for the HAHS used in this thesis will be stated.

The HAHS system consists of a set of PEs:

$$P = \{p_1, ..., p_n\}\ n \in \mathbb{N},$$

a set of clusters:

$$C = \{c_1, ..., c_l\}\ l \in \mathbb{N};\ l \leq n,$$

a set of tasks:

$$T = \{t_1, ..., t_m\}\ m \in \mathbb{N},$$

and dependent on the concept a set of organs:

$$O = \{o_1, ..., o_k\}\ k \in \mathbb{N};\ k \leq m$$

Each organ itself consists of tasks of the task set:

$$o_i \subset T \text{ with } 1 \leq i \leq k$$

and

$$T = \dot{\bigcup_{o_i \in O}} o_i$$

Each PE is part of exactly one cluster:

$$\lambda : P \to C$$

$$\forall c \in C\ \exists p \in P : c = \lambda(p)$$

The set of PEs of one cluster is defined as:

$$P_{C_i} = \{\forall p_j \in P : \lambda(p_j) = c_i; 1 \leq j \leq n\}$$

Each cluster has exactly one PE, which is the so-called cluster head

$$\exists! p_{CH_i} \in P_{C_i}$$

Additionally, the set of PEs belonging to the same cluster as PE p_i is described as:

$$CM_{p_i} = \{\forall p_j \in P; \lambda(p_i) = \lambda(p_j)\}$$

The set of allocated tasks on PE γ is defined as Ξ:

$$\Xi_\gamma \subseteq T$$

Hormones are denoted in the following form: *eager values* are abbreviated with EA, *suppressors* with S and *accelerators* with A. Additionally, the *modified eager value* will be denoted as mEA and the *eager values* in the *inter-cluster* cycle, the so-called *cluster eager values*, as CEA.

In order to guarantee the self-optimization property the AHS implements an offer mechanism. PEs allocating tasks will stop sending *suppressors* for those tasks after w hormone cycles, where w is the so-called *offer period*. After d hormone cycles (the so-called *retake delay*) the PE will send *suppressors* again, if no other PE has won the task meanwhile.

4.6 Cluster set concepts

The HAHS introduces two new main aspects on the one hand the clustering of PEs and on the other hand the cluster heads. Along with those aspects the questions arises which PEs form a cluster and which PEs will become cluster heads. Both questions can be handled with two different approaches: a static, predetermined approach or a dynamic, flexible approach. In comparison to the static approaches, the dynamic approaches tend to be more resilient regarding failures in the system, but they are also more complex.

The different approaches will be explained in the following sub-chapters 4.6.1, 4.6.2, 4.6.3, 4.6.4.

4.6.1 static-static

The first variant of the HAHS is the so-called *static-static* HAHS. In this variant, both the clusters and the cluster heads are static. This means they are predefined by the designer of the system and cannot be altered by the system itself. With this approach, single-points-of-failure are introduced in the clusters. If the cluster head of a cluster fails, the entire cluster will fail because all other cluster members will no longer have a connection to the *inter-cluster* cycle. Without this connection the cluster cannot take part in the negotiation of the task subsets. The other clusters will have to take over the tasks that were originally allocated by the failed cluster.

4.6.2 static-dynamic

The next variant of the HAHS is the *static-dynamic* HAHS. In this case the clusters are still *static* and predefined, but the cluster heads are dynamic. This means the cluster heads in each cluster will be determined by the system during run-time. It also means the system can reconfigure the cluster head ability during run-time. This comes in handy when a cluster head PE fails. In this case, the system is able to react to the failure by determining a new cluster head for the cluster. This approach prevents the single-point-of-failure for clusters which appears in the static-static configuration.

The dynamic determination of cluster heads can be achieved in many different ways. The simplest solution is to determine a fall-back cluster head. This fall-back cluster head will take over the cluster head function of the cluster as soon as the first predefined cluster head fails and does not respond to messages anymore. This solution is only dynamic to a certain degree. As soon as the fall-back cluster head fails, the whole cluster becomes useless for the system. Of course this could be coped by introducing a second fall-back cluster head, but a ranking of the fall-back cluster heads has to be implemented as well. The ranking prevents that both fall-back cluster heads simultaneously try to take over the cluster head function of the cluster. If this ranking is static it will not consider any environmental influences, e.g. the processing load of the PE. For example, if the second fall-back cluster head would be more suitable to take over the cluster head function because it does not process any task at the

moment, regardless of this, the first one will take over the job as cluster head anyway.

A better solution is a fully dynamic mechanism to elect the cluster head. This would also allow a reaction to failures in a more flexible way. Such a mechanism can be designed by utilizing the AHS. As stated before, the AHS has the properties of self-optimization and self-healing. Each cluster runs its own separated AHS that is utilized for the cluster head determination. A cluster head voting mechanism can be designed by introducing a special task in each of those isolated cluster AHSs. All PEs add this special task to their task set. The PE allocating this special task becomes the cluster head of the cluster. If the PE fails or becomes overloaded such that it cannot process the upper control loop any more, the AHS ensures that a substitute cluster head will be determined (see figure 4.3). The AHS properties of *self-configuration, self-healing and self-optimization* will be applied to the cluster head determination process.

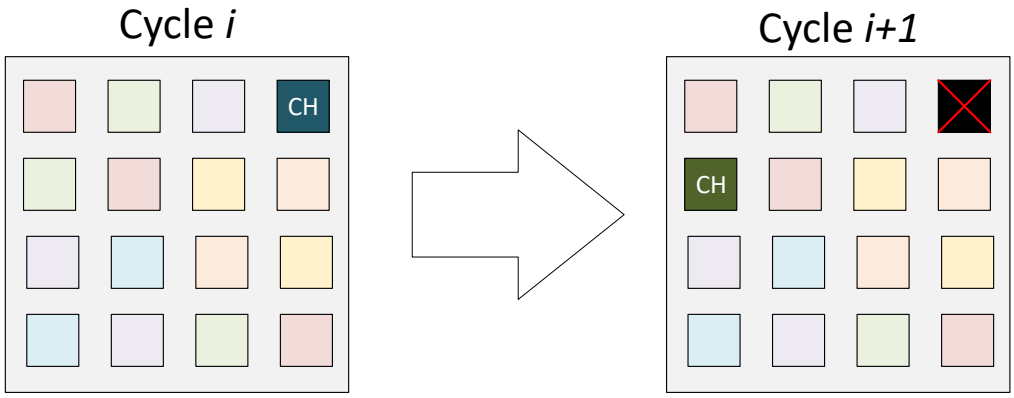


Figure 4.3: Self-healing of the dynamic cluster head. The cluster head task *CH* will be reallocated to the next best PE after the cluster head permanently failed. Therefore, a new cluster head emerges and the cluster remains functional.

4.6.3 dynamic-static

An advanced variant is the dynamic-static HAHS. In this variant the cluster heads will be predefined from the designer of the system again. However, the clusters will be *dynamic*. That means that all PE, except the predefined cluster

heads, have no information about their cluster membership. Each cluster head belongs to another cluster and cannot change this membership. When the system starts, the PEs form autonomously as many clusters as cluster heads exist. So, the system autonomously finds an initial configuration of clusters. Cluster suitability might depend on spatial or functional properties. Furthermore, the system is capable of changing the cluster configuration during run-time. It can react to changes in the environment or failures of PEs. Still this variant has a cluster single-point-of-failure: the cluster heads are predefined. In case one of those fails the cluster will be lost. However, instead of losing all PEs of this cluster, like in the *static-static* variant, those PEs can change their cluster membership dynamically and therefore re-join the system.

The *dynamic* cluster aspect in the context of the HAHS can be implemented by introducing a new hormone. This is the so-called *cluster accelerator*. The cluster heads will emit the *cluster accelerator*. Depending on the chosen membership method, the value of the *cluster accelerator* is different at the receiving PEs. The most obvious method for the membership is the spatial distance. Other methods (e.g. based on functionality) are also conceivable but will not be covered in this thesis In case of a spatial distance, the value of the *cluster accelerator* decreases with the distance of its origin (e.g. Manhattan Distance). A PE assigns itself to the cluster with the highest *cluster accelerator* it received[2]. The *cluster accelerators* are emitted constantly by the cluster heads, in order to ensure that the PEs stays in its cluster, as long as the cluster head is alive.

[2]In case it receives two or more cluster accelerators with the same value it uses a second criterion (e.g. the id of the sender) to make a decision

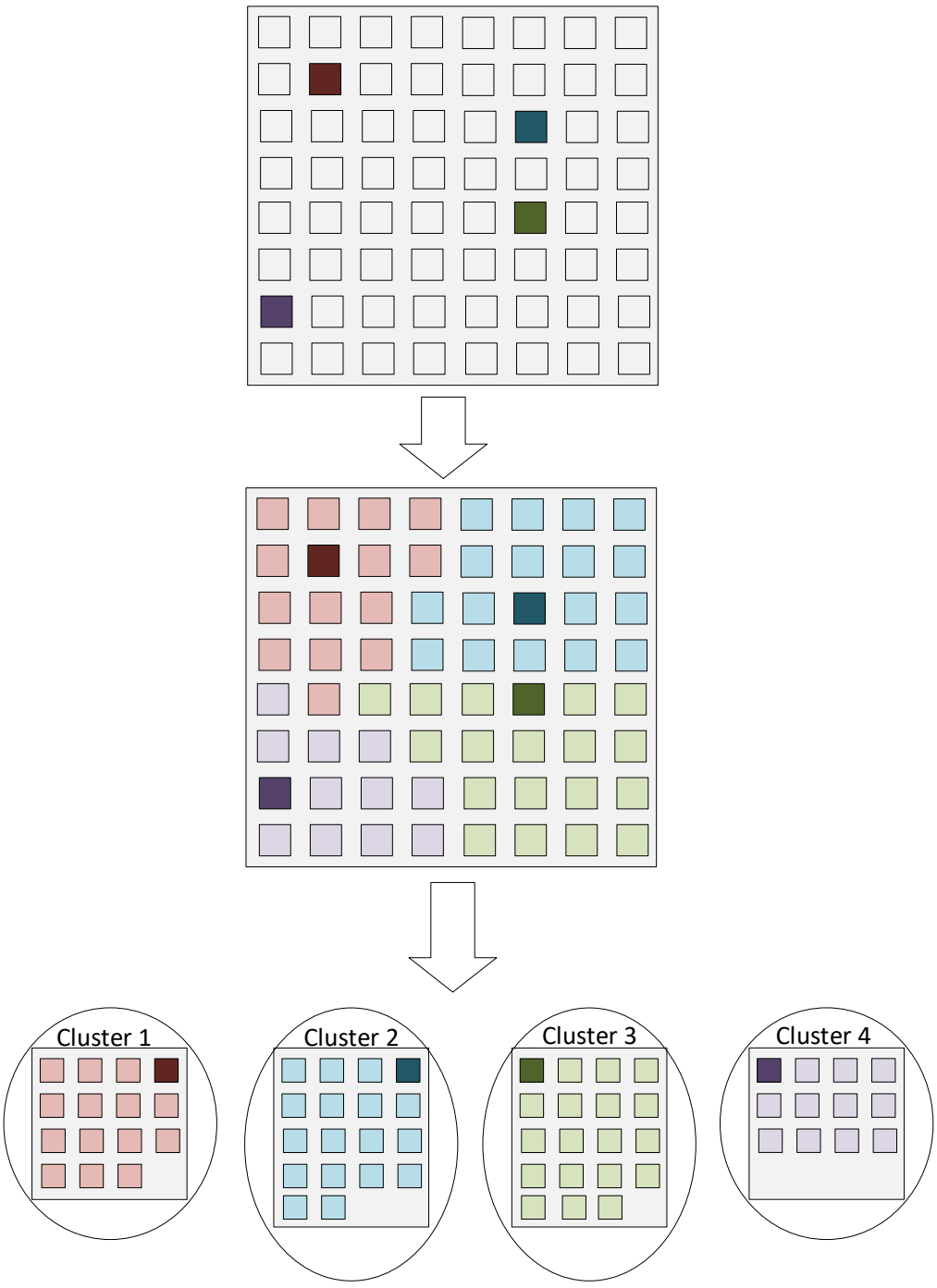


Figure 4.4: Example of the clustering in the *dynamic-static* concept: The cluster heads broadcast their cluster accelerator and the PEs assign themselves to the cluster head with the highest received cluster accelerator (in this example based on spatial distance).

4.6.4 dynamic-dynamic

The last concept is the most flexible one. In this concept, both the clusters and the cluster heads are *dynamic* and autonomously chosen by the system. This is a composition of the two *dynamic* aspects of the previous variants. To achieve this complex *dynamic* behavior, two different approaches are conceivable.

One way is to start with the autonomous election of cluster heads and then proceed with the method described in section 4.6.3. The election of cluster heads in a complete cluster-less environment can be achieved similarly as described in section 4.6.2. A certain amount of cluster head tasks is introduced into the system. When the system starts, the AHS starts distributing those cluster head tasks. Allocating the cluster head tasks unlocks the sending of the *cluster accelerator* in order to build the clusters for the cluster heads.

The second way is to use the reverse order. This means that at first the clusters and then the cluster heads are determined. For determining the clusters, a cluster analysis is needed. Two main approaches of a cluster analysis exist: the *agglomerative* and *divisive* cluster analysis [26]. The *agglomerative* cluster analysis starts with every PE being its own cluster. So at the beginning the system consists of as many clusters as PEs. During run-time the PEs form bigger clusters by migrating to neighboring clusters. This procedure is repeated until all clusters reach a certain minimum of PEs as cluster members. The *divisive* cluster analysis, on the other hand, starts with all PEs belonging to the same cluster. When the cluster is started, it splits up into more and more clusters until each of those new clusters reaches the amount of required PEs. As soon as the clusters are determined, the cluster heads can be elected as described in section 4.6.2.

4.6.4.1 Regulating unbalanced clusters

When using a *dynamic cluster* concept in conjunction with the *organ task* concept (see section 4.4.1), a problem can occur depending on the specific configuration: unbalanced clusters with evenly sized organs can lead to missing tasks in the allocation even though the resources (the PEs) are sufficient.

This "under allocation" arises from the static, equally sized organs combined with the dynamic, unequally sized clusters. Let's assume the PEs can only

allocate a certain number of tasks, e.g. only one or two. Then it is possible that there are clusters having not enough PEs to fully allocate all tasks from an organ. Due to the *organ task* concept, the cluster head of such a cluster still wins an organ and tries allocating all its tasks. This results in missing tasks in the system.

To compensate this different strategies have been developed, to assign more PEs to a undersized cluster, as soon as this problem is recognized[3].

Regulating the cluster accelerator by a PID controller The first strategy implements a *PID controller* [15] for every cluster head. This controller is responsible for adjusting the cluster accelerator of the cluster head in case the cluster is too small. The number of allowed tasks from the organ (the organ size) is used as *setpoint* of the controller and the amount of received acquisition suppressors is used as the *process variable*. The difference of *setpoint* and *process variable* is the *error* the controller tries to minimize. The error is then used in the proportional, integral and derivative terms which together calculate the *control variable*. The calculated *control variable* amplifies the cluster accelerator. A crucial point in this approach is the period of the control loop. The period should be long enough to let the change of the cluster accelerator take effect on the system and its allocation. On the other side, the period should be as short as possible in order to reach a fully functional state as soon as possible. It also has to be considered that the amplification of the cluster accelerator of one cluster effects the size of the clusters in its neighborhood. This might include clusters which are suitable sized for their organs. Those clusters then lose PEs to the cluster which amplified its cluster accelerator and therefore might not allocate all their tasks anymore. Subsequently, those clusters then start to use their PID controller in order to amplify their cluster accelerator, too. This can result in a chain reaction which either eventually finds a working state for all clusters or never stops to oscillate. This method was evaluated and the results are presented in section 8.1.1.4.

[3] A cluster head can recognize this problem by comparing the number of allowed tasks in its cluster to the amount of received acquisition suppressors.

Regulating the clusters with a special hormone Another developed strategy introduces a new special hormone, called *stress hormone*, for those cluster heads which recognize missing tasks from their organ. This hormone is broadcasted into the whole network, such that every PE receives it. Upon receiving a *stress hormone*, a PE without any allocated tasks and not being the cluster head of a cluster stores an amplification value for the cluster accelerator of the sender of the *stress hormone*. Thereupon, the sender might acquire more PEs for its cluster in the next hormone cycle. The amplification value can depend on the amount of missing tasks of the sender, such that clusters missing many tasks have a higher chance of acquiring free PEs. The disadvantage of this method, besides the additional communication, is the resulting spatial fragmentation of those clusters. This might lead to slower inter-task communication of tasks belonging to the same *organ* (depending on the physical topology of the system). This method was also evaluated and its results can be found in section 8.1.1.4.

Worsening the cluster head ability Another idea in this context, which is only applicable in the full dynamic-dynamic concept, is to adjust the cluster head which recognizes the missing tasks in its cluster. So instead of amplifying the cluster accelerator the suitability of the cluster head PE to the cluster head task is reduced. This might lead to a migration of the cluster head task to another PE, which in turn forms a complete different cluster due to its spatial position. Similar to the first presented strategy, this relocation of the cluster head might influence clusters which are perfectly sized, probably causing clusters not having enough PEs in order to allocate all tasks anymore. Again the result is a chain reaction, which either stops eventually in an optimal state for every cluster or it keeps worsening the cluster head suitabilities of the PEs. The latter can cause the effect of vanishing cluster heads. If enough PEs are weakened in their cluster head suitability, there might not exist enough PEs for the desired amount of cluster heads. This results in fewer but bigger clusters, which might remedy the problem of missing task allocations. On the other hand, it also can cause a chain reaction in which all PEs might worsen their cluster head suitability. The result is one remaining cluster head allocating all organs in one big cluster. This state would be like the original AHS with

additional overhead and is, therefore, not desirable. Due to its disadvantages, which are similar to the ones from the first strategy, this method was not evaluated.

Splitting organs Another conceivable and dynamic method to cope with the problem of missing allocations due to small clusters is the splitting of organs. As soon as a cluster head recognizes unallocated tasks, it detaches the missing tasks from the organ. Afterward, it informs all PEs about the changed task set. Upon receiving this information, the cluster heads start applying for the detached tasks in the *inter-cluster* cycle independently. Those detached tasks are treated as in the *single task* concept. The disadvantage of this solution is that tasks belonging originally to one organ get separated to different clusters. This might lead to a slower inter-task communication between those tasks. Additionally, the communication load rises due to the increased number of tasks in the *inter-cluster* cycle. Furthermore, the cluster heads necessarily do not know if their cluster is able to additionally allocate one or several tasks. Therefore, they either have to apply on spec and release the task again if the cluster cannot allocate the task, or they have to use a method of determining the cluster eager values for the detached tasks (see section 4.7), resulting in further communication. A similar method which splits all organs after the self-configuration is proposed in chapter 10 and currently part of research. Therefore, an evaluation of the method was waived.

4.7 Determining cluster eager values

Since the *eager values* are stored locally at the PEs, the *cluster heads* cannot access all *eager values* in their cluster without further communication. This means that the *cluster head* has a lack of information concerning the suitability of its cluster for the tasks. This in turn means that a sensible distribution of the tasks on the *inter-cluster* cycle is hardly possible.

That is why a mechanism is needed in order to depict the suitability of a cluster in the *inter-cluster* cycle. In a *static* cluster environment (see 4.6.1 and 4.6.2) each *cluster head* or *cluster head candidate* could retrieve a set of pre-defined *cluster eager values*. This is the simplest solution and is used for

the *organ task* concept[4]. A feedback loop between the suitability of a cluster and its cluster head is missing with this method. Also, a PE failure in a cluster does not change the predefined *cluster eager value* (see figure 4.5), which can be a problem for the complete task allocation.

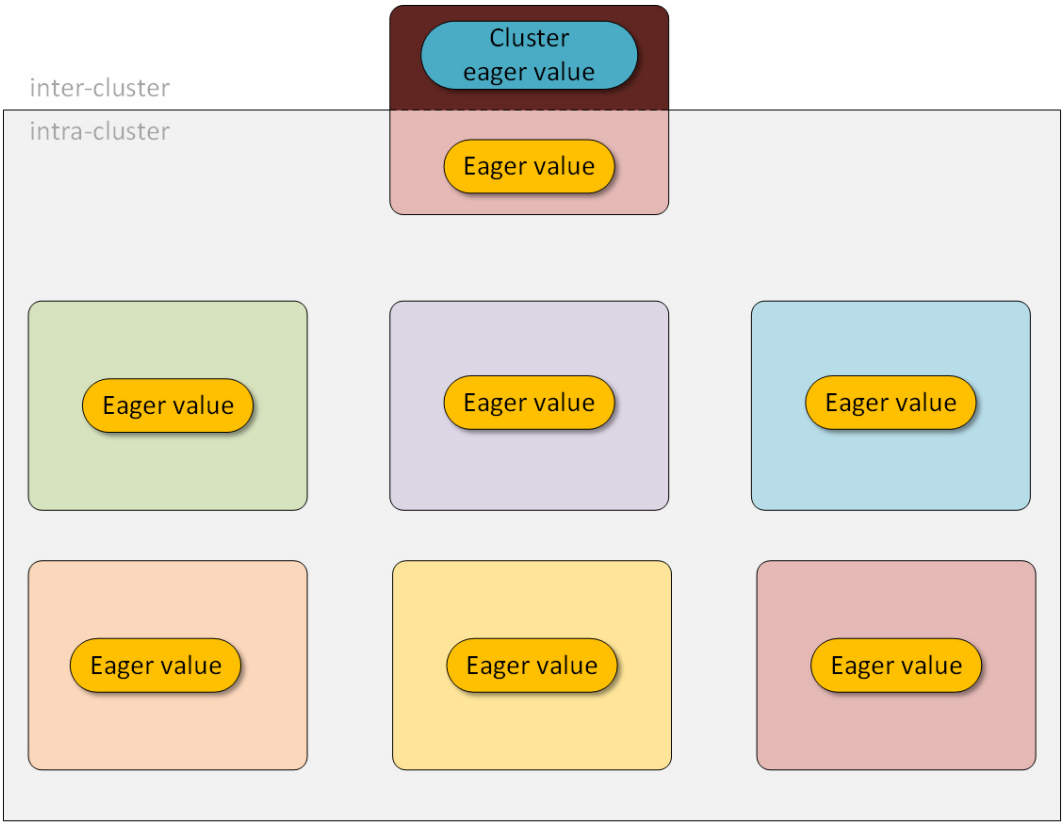


Figure 4.5: Predefined cluster eager values. The cluster eager value is not related to any of the eager values from the cluster members and will not be influenced by changes of those.

4.7.1 Recalculation of cluster eager values

One method to solve this issue is the periodic recalculation of the *cluster eager values* (CEA) by means of the eager values of the cluster children. For this reason, all PEs in the clusters send periodically all their eager values to the cluster head, even for those tasks which are currently not active in the cluster. The *cluster head* then sums up the received eager values for each task and

[4]a dynamic method to generate the *eager values* of the organs would be also conceivable, but is not be covered in this thesis

multiplies it by a factor (τ) indicating the current load of the cluster (see figure 4.6 and equation 4.1)[5]. This represents the current *cluster eager value* for the task. In order to reflect the dynamic suitability, this value has to be recalculated periodically. The periodic resending of all *eager values* ensures that dynamic changes of the suitability in the *intra-cluster* cycles are depicted to the *inter-cluster* cycle. An evaluation of the recalculation method can be found in section 8.1.1.3.

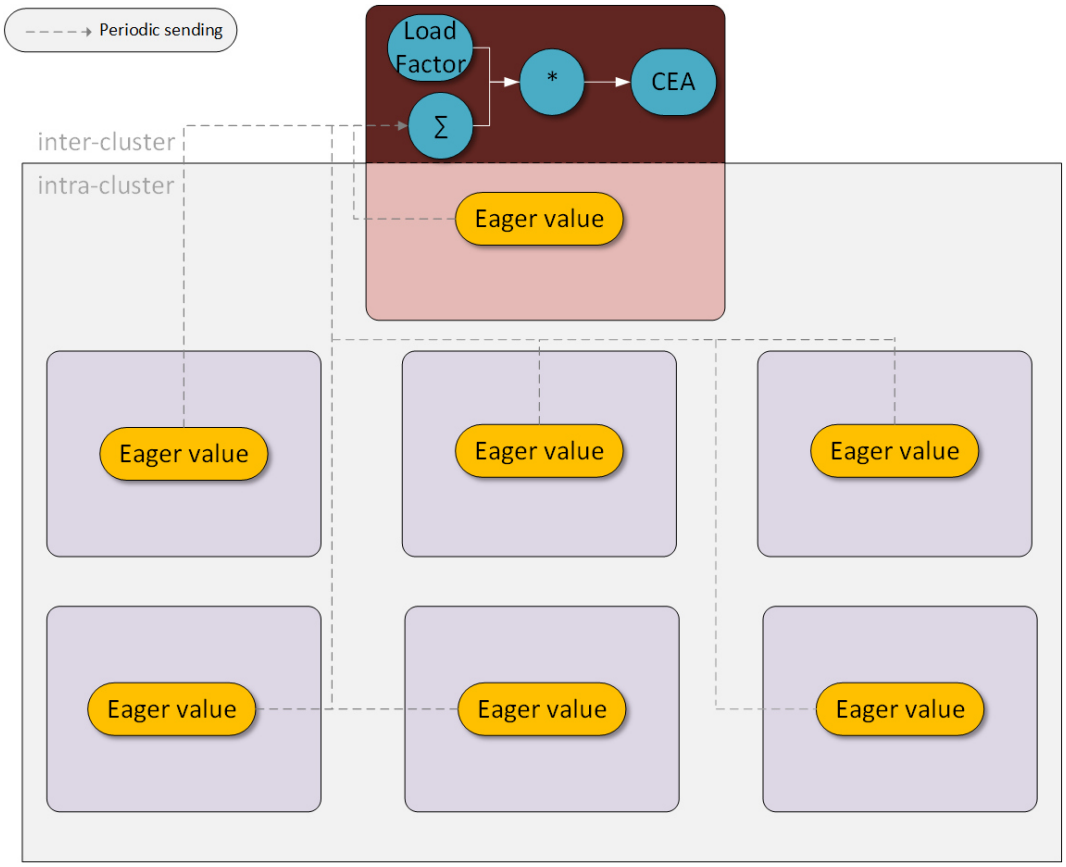


Figure 4.6: Cluster eager values (CEAs) are recalculated periodically.

$$CEA_{i\gamma} = \sum_{\delta \in CM_\gamma} (EA_{i\delta}) \cdot \tau \tag{4.1}$$

[5]The load factor can be replaced by any other calculated factor, which fits the application scenario.

4.7.2 Mimic best PE

A second method for depicting the suitability of the cluster children to the *inter-cluster* cycle is that the *cluster head* copies the best *eager value* for all tasks of all cluster children. Therefore, the PEs send all hormone sets (these are the information about the *eager value*, the *suppressors* and *accelerators*) of all their tasks to the cluster head in the beginning. The cluster head chooses the best hormone set (by comparing the *eager values*) for each task and uses those hormones for the task in the *inter-cluster* cycle. In this way for each task the best PE is represented in the *inter-cluster* cycle. This procedure can also be repeated periodically, to depict dynamic changes in the cluster children.

4.7.3 Magnitude of eager vectors

In this method, each PE interprets its *eager values* for the task as one vector, the eager vector. Instead of sending an *eager value* for each task in each cycle, the PEs send the magnitude of their eager vector to their cluster heads in each cycle. This results in only one hormone per PE instead of m hormones per PE (m = number of tasks in the system). The cluster head uses the received magnitudes and calculates, based on those (e.g. the average, the mean or Euclidean distance) a cluster eager value for all tasks. This method was also evaluated and the results are presented in section 8.1.1.3.

4.7.4 Greatest Hormone

The disadvantages of the methods 4.7.1 and 4.7.2 are the periodic recalculation and sending of all their eager values. Because of the fact that a cluster child PE does not know which PE is the *cluster head* of its cluster[6], the *eager values* have to be broadcasted through the *intra-cluster* network. This leads to a short but high communication load in the whole system. Instead of the PEs sending *eager values* for all tasks to their cluster heads, in this method they only send the *eager value* of the task with the highest *eager value*. The cluster head only stores the maximum eager value it received for each task. Thus, the cluster head receives an incomplete picture of the suitability of its cluster for all tasks. This means it only knows the maximum *eager value* for some tasks in

[6] At least not without producing additional communication load, which should be avoided.

the cluster. For the remaining tasks, it "guesses" their *eager values* by means of calculating a heuristic value (see figure 4.7). The heuristic value is calculated by summing up all received maximum *eager values* and dividing the sum by the number of received *eager values* (see equation 4.2). Therefore, it calculates the mean of the maximum *eager values*.

$$h_{i\gamma} = \frac{\sum_{\delta \in CM_\gamma} (E_{max_\delta})}{|CM_\gamma|} \tag{4.2}$$

Further details of the process of winning a task by a heuristic value are explained in the appendix A.1.

4.8 Comparison of the concepts

In this section, the different concepts will be compared according to their communication load, flexibility and the self-x properties.

4.8.1 Task set concepts

Comparing the two task set concepts, it is clear that the *single task* concept is just one possible configuration of an *organ task* concept (where each organ only has one task). Nevertheless, there are significant differences. While the *organ task concept* also reduces the number of tasks to apply on the *inter-cluster* level, the *single task concept* only reduces the number of PEs which apply on each task in the *intra-cluster* cycle. This of course depends on the number of organs in the system. It also effects the task allocation time, i.e. the time it takes until all tasks are allocated. The *organ task concept* is faster due to the time saving in the *inter-cluster* level. The big disadvantage of the *organ task* concept compared to the *single task* concept is the reduced flexibility in terms of *self-optimization* and *self-healing*. Only complete organs can be migrated, due to an optimization or a failure. The *single task* concept is more flexible and tasks can migrate to another cluster without affecting the remaining task distribution. Therefore, the *self-optimization* and *self-healing* can be faster in the *single task* concept.

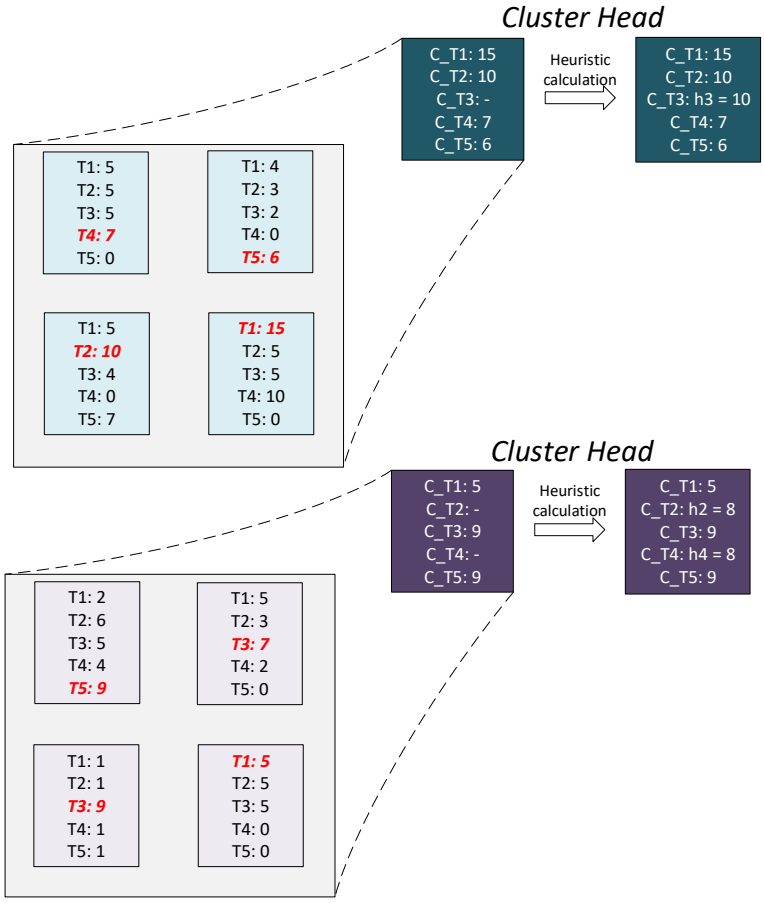


Figure 4.7: Calculating the cluster eager values with the *greatest hormone* method. In the first cluster a heuristic value of 10 will be used for task 3. The second cluster uses heuristic values for the tasks 2 and 4. All other cluster eager values are based on eager values of the cluster members. The maximum eager values of each PE are displayed in red.

4.8.2 Cluster set concepts

The cluster set concepts differ in two dimensions, the cluster dimension and the cluster head dimension. On the one hand, the less static and predefined the dimensions are, the more flexible the system will become. On the other hand, this flexibility comes with a price, the production of additional communication load between the PEs. Therefore, the degree of flexibility and adaptivity has to be chosen wisely according to the underlying network bandwidth. A more static and predefined system might reduce the communication load a

lot but also might contradict the idea of *organic computing*, by introducing single-point-of-failures and centralized structures.

4.8.3 Cluster eager values

The different methods for determining cluster eager values also differ in their production of communication and their flexibility/adaptivity. Hence, the recalculation method (see section 4.7.1) with a small period can react fast to changes in the suitability but also produces a lot of communication overhead. A higher period reduces the communication output but also extends the reaction time.

The *mimic* method (see section 4.7.2) only produces communication overhead in the beginning, but cannot depict changes from the *intra-cluster* cycle to the *inter-cluster* cycle.

In contrast, the *magnitude* method (see section 4.7.3), has a constant communication flow between the suitability in the *intra-cluster* cycle and the *inter-cluster* cycle. It also reduces the number of sent hormones by only sending one special *eager value* every cycle instead sending one for every task every cycle.

A step further from this method is the *greatest hormone* method which reduces the communication even more, but has to extrapolate some *cluster eager values* due to the reduced communication. This might increase the reaction time because tasks could be allocated or migrated to sub-optimal PEs at first.

4.8.4 Summary

In a nutshell the concepts differ a lot in the way they implement a HAHS. The combination of the different techniques depends on the exact application for the HAHS and the technical requirements. Figure 4.8 shows the different methods and concepts on a two-dimensional system. This should serve as reference point to estimate which method and technique is conceivable for implementing a HAHS and which flexibility and communication load it creates.

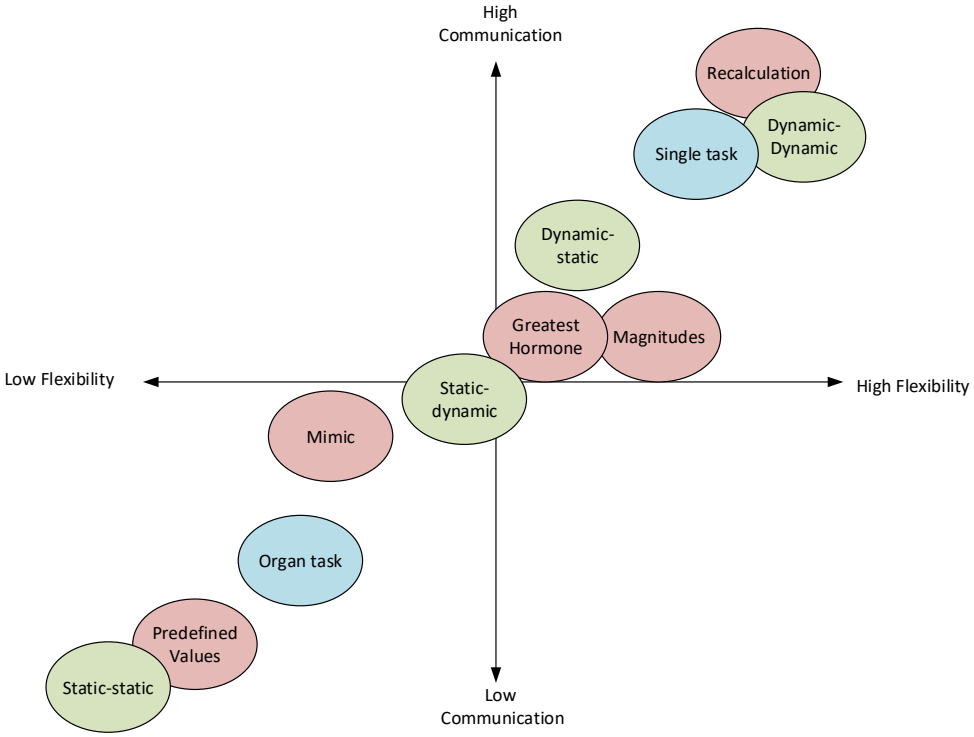


Figure 4.8: Comparison of the different concepts according to flexibility and communication overhead

Chapter 5

The Recursive Artificial Hormone System

The RAHS is also based upon the AHS and is an advanced development of the HAHS. The idea of the RAHS is similar to the idea of the HAHS. While the HAHS uses exactly two levels, the RAHS arranges several isolated AHSs in a hierarchical tree structure with several levels. The isolated AHSs (clusters) are connected vertically to a processing element from a cluster of a higher level. This processing element is the so-called representative in the higher level (*upper representative*). The PEs in the topmost hierarchical level do not have a representative since they already are in the highest level. The topology of such a RAHS can be seen as a tree. The AHS in the topmost level represents the root of the tree, while all AHSs in the lowest level represent the leaves of the tree. Figure 5.1 shows an example topology of connected city blocks with several hierarchical levels. The tree topology is unbalanced and consists of many different devices, which can include several PEs. The RAHS builds an isolated AHS cluster in every node and determines a representative for this cluster in the next higher level. This representative needs a physical communication to the PEs in the cluster of the next higher level. Therefore, network routers are suited for this special job.

Similar to the HAHS the complete task set will be split into task-subsets and distributed to the AHSs in the next lower hierarchical level (see figure 5.2).

Each task-allocating PE can either decide to allocate the task or to pass it to the next lower hierarchical level, if it is a representative. In the RAHS

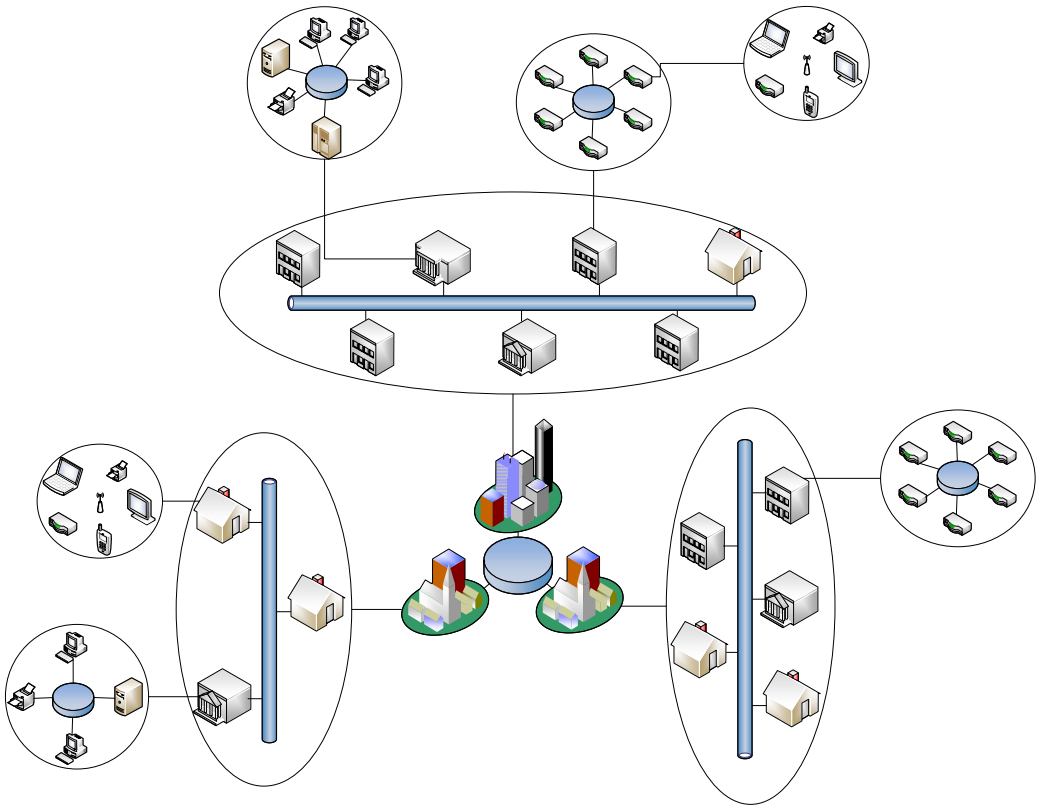


Figure 5.1: Example of a topology for the RAHS [39]

concept two kinds of hormone flows exist. First, the *horizontal hormones (horizontal eager values, horizontal suppressors, horizontal accelerators)*, are all hormones which are interchanged between PEs in one of the separated AHS. The communication between the hierarchical levels will also be provided by hormones. These are called *vertical hormones (vertical eager values, vertical suppressors, vertical accelerators)*.

Regarding the concept of the RAHS, the HAHS can be interpreted as a RAHS with only two hierarchical levels.

The advantage of the RAHS is its flexible adaption to the application and the topology of the system. If a given topology is large and tree-like, the RAHS reduces the communication load in every node and between the nodes.

One possible adaption and implementation of a RAHS is presented and evaluated in [39]. The presented adaption focuses solely on binary-trees. Additionally, an allocation is allowed on the lowest hierarchical level only. This means all PEs in higher levels pass the tasks down to the next lower level.

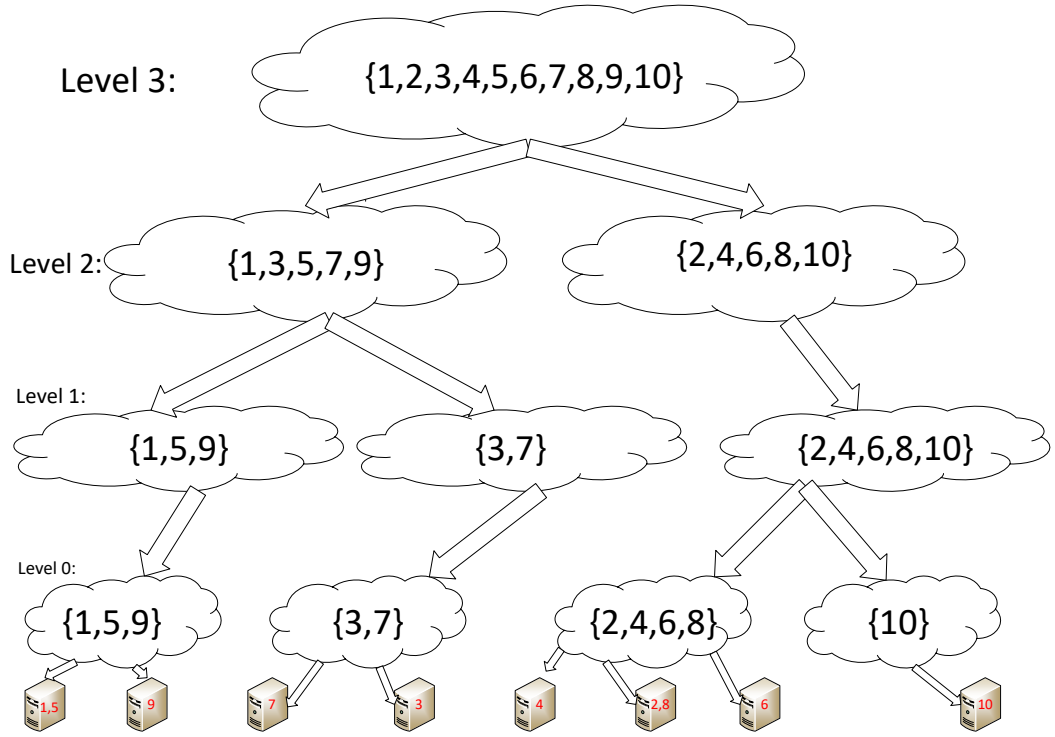


Figure 5.2: Example of the task splitting in the RAHS [40]

5.1 Formal definition

Additionally to the formal definitions of the HAHS (see section 4.5), the RAHS has a set of levels:

$$L = \{l_1, ..., l_g\}\ g \in \mathbb{N}$$

each cluster $c_i \in C$ belongs to exactly one level:

$$\mu : C \to L$$
$$\forall l \in L\ \exists c \in C : l = \mu(c)$$

Furthermore, the RAHS ha a set of PEs belonging to the lowest level. Those are the *leaf* PEs:

$$LP = \{\forall p_i \in P | p_i \in c_x \wedge \mu(c_x) = |L|\}$$

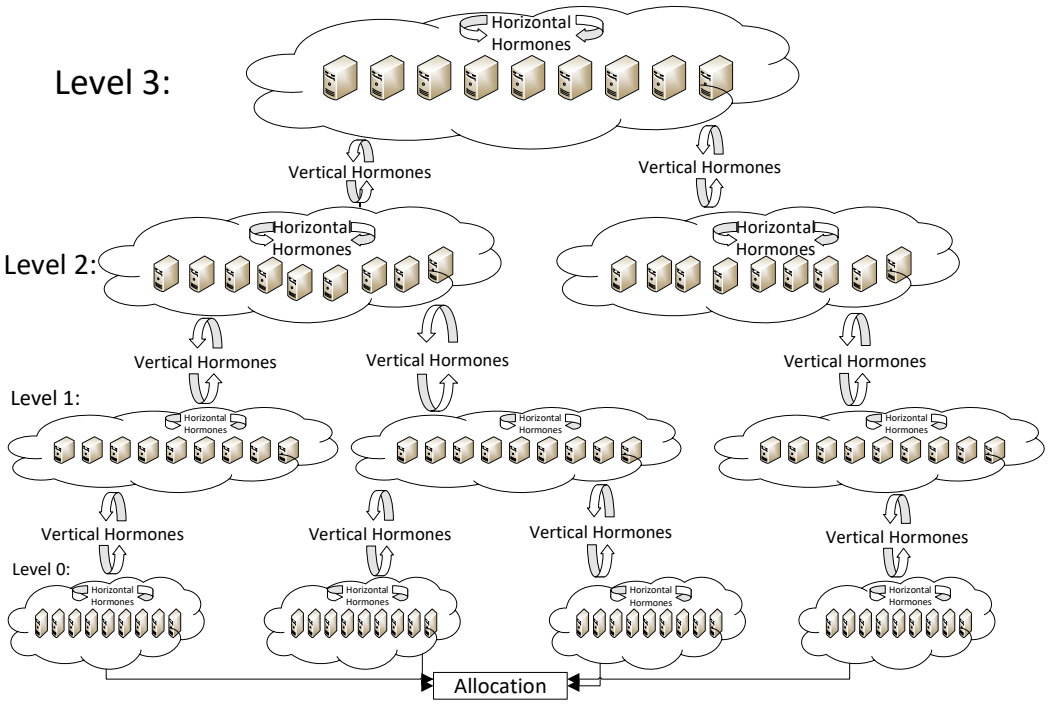


Figure 5.3: The horizontal and vertical hormone loops of the RAHS [40]

5.2 Pass or allocate

As mentioned before, different approaches are conceivable in the RAHS concept concerning the level of task allocation. The PEs could either pass the tasks always to the lowest level (see [39]) or dynamically decide if they allocate the task or pass it to their underlying cluster. Besides the already mentioned strategy to pass all tasks to the leaves, other simple solutions are conceivable like giving each task an allocation level. If a PE belongs to the allocation level of the task (or a lower level), it allocates the task. All other PEs winning the task passes it to the next lower level. Other strategies include more dynamic approaches. A strategy could introduce a hormone feedback loop between the representatives and their underlying AHS. The deciding PE can compare its own *horizontal eager value* to the *horizontal eager values* from the PEs of the underlying AHS.

5.3 Determining the eager values of the lower level

Similar to the HAHS, the RAHS also has the problem of depicting the suitability of the leaf PEs to the node and root PEs. Therefore, techniques for transporting these information through the levels are needed, too. As before, these techniques should reduce the amount of communication load in the AHSs and between the levels. At the same time, it is important that they depict the suitability of all PEs to the higher levels.

5.3.1 Periodic update

One technique to depict the suitability of the PEs is a periodic update. Every x ($x > 1$) hormone cycles the PEs sends their current *modified eager value* to their representative in the next higher level. The receiving representative can for example take the average *eager value* or the highest *eager value* as *vertical eager value*. It compares the value to its own *horizontal eager value* and sends the higher value to its own representative. This technique is repeated from the lowest level up to the highest level. As soon as a task gets unlocked in a cluster, the PEs applies for the task the own *horizontal eager value* or the *vertical eager value* from below, which one is higher. If the PE wins the task in its cluster, it either starts allocating the task, in case the sent *eager value* was its own, or passes it to its underlying cluster if the eager value was the *vertical eager value*. This is an accurate method for depicting the suitability among the hierarchical levels, but only if the periodic interval is not too big. A big interval leads to delayed reactions. Small intervals, on the other hand, produce a lot of communication load. Therefore, this method has similar problems like the HAHS with the *recalculation method* (see section 4.7.1).

5.3.2 Max eager value

The *max eager value* method is similar to the *greatest hormone* method of the HAHS (see section 4.7.4). It also relies on the maximum *eager values* of each PE. The algorithm presented in [39] works as follows (see figure 5.4 for a flow chart):

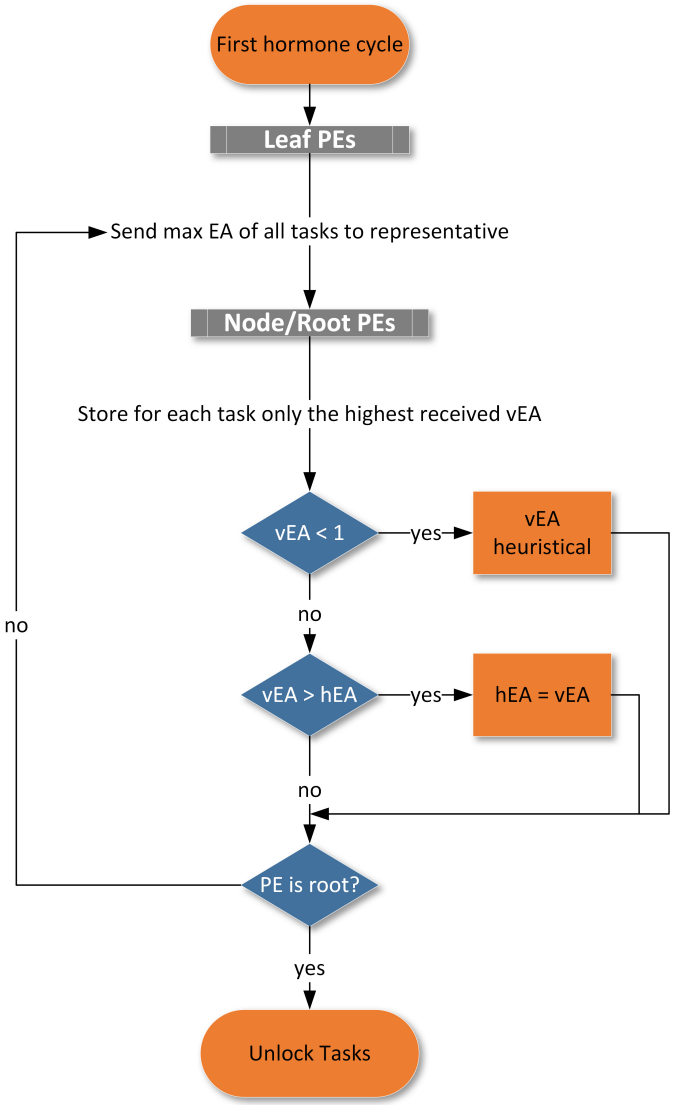


Figure 5.4: Flow chart of the first part of the *max eager value* algorithm

1. In the first hormone cycle, right after the start of the system, all PEs in the lowest level find their maximum *eager value* of all tasks and send the value as well as the task id to their representative PE in the next higher level. These are the *vertical eager values*.

2. The representative PEs only save the highest received *vertical eager value* for each task.

3. Then, they update their *horizontal eager value* for each task as follows:

   (a) If the PE received a *vertical eager value* for this task, the *horizontal*

eager value will be set to the value of the *vertical eager value*

(b) If no vertical eager value was received for this task, all received *vertical eager values* are summed up and divided by the number of possible tasks. This is the heuristic value for the task on this PE.

4. Following, the PE sends the maximum value of all *horizontal eager values* as *vertical eager value* to its own representative in the next higher level.

5. This is repeated until the PEs in the root cluster calculated their *horizontal eager values*, whereupon the sending of the *eager values* starts in the cluster and therefore the allocation of tasks starts as well.

This approach generates the initial eager values in each level. As soon as the allocation mechanism, the separated AHSs to be specific, starts, the second phase of the approach comes into effect(see figure 5.5 for a flow chart):

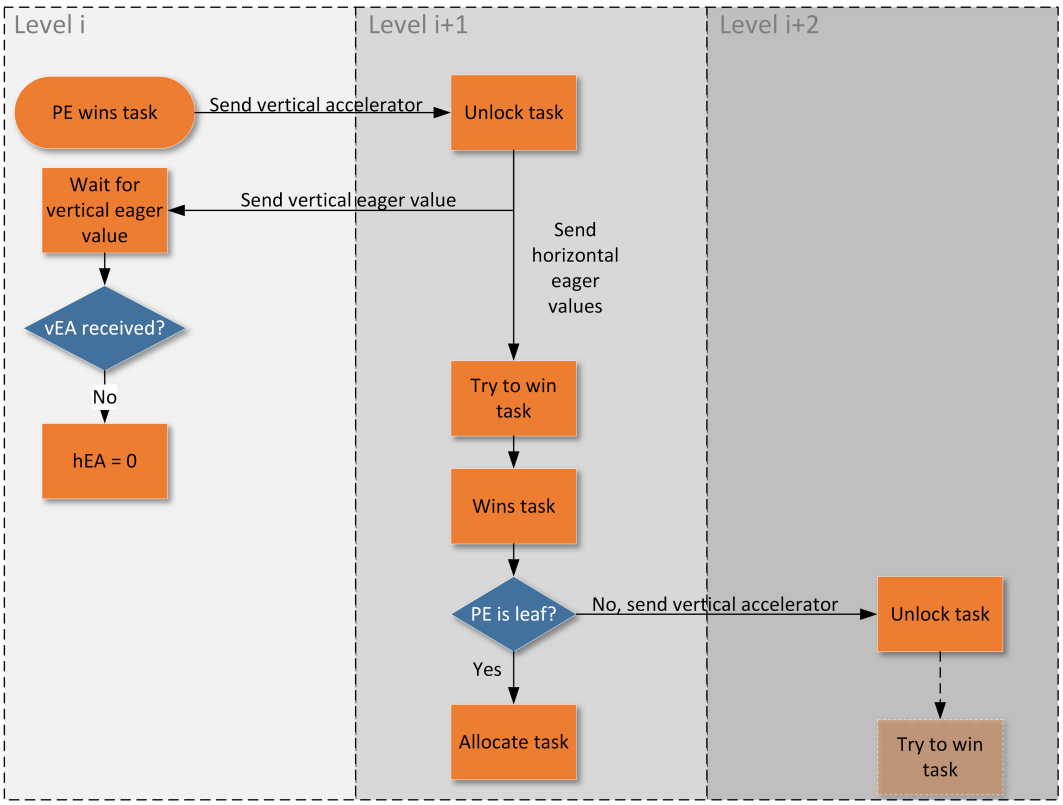


Figure 5.5: Flow chart of the second part of the *max eager value* algorithm

1. When a PE wins a task and passes it down, it sends out a *vertical accelerator* to its underlying cluster in the next lower level. This *vertical accelerator* acts as activation hormone for the task.

2. When receiving the *vertical accelerator*, the PE starts sending out the *horizontal eager value* for this task to all other PEs in the cluster. It also sends it to the representative PE (in the form of a *vertical eager value*) which sent the *vertical accelerator*.

3. The PE in the cluster which wins the task sends out the task suppressor to all other PEs in the cluster and to the representative as *vertical suppressor*.

4. The representative saves the maximum of the received *vertical eager values* as new *horizontal eager value* for the task. If no *vertical eager value* was received after a certain time, the representative assumes that the task is not allocatable in the underlying cluster at the moment. Consequently, it sets its *horizontal eager value* for this task to 0.

5. The representative also uses the received *vertical suppressor* as indication that the task was successfully allocated and the *horizontal eager value* does not have to be adjusted at the moment.

With this method, a task which was falsely passed down to a cluster, which is not able to allocate the task (anymore), is reallocated in the higher level. An evaluation of this method can be found in section 8.2.1.

5.3.3 Least eager value method

Until now, all tasks were allocated in the lowest level, the leaf level. In order to use the full potential of an unbalanced, heterogeneous tree of PEs, a strategy for deciding to allocate or pass the task is necessary.

5.3.3.1 Allocation strategy

In order to find an adequate strategy for this purpose, it is necessary to know when a PE should allocate the task. The answer is that it should allocate the task if it is more suitable than all other PEs in the entire system. Unfortunately, a PE in the RAHS can have the information if or if not it is the

most suitable in the entire system, only by producing so much communication that using a conventional AHS would be more reasonable. Therefore a simple mechanism relying on the information a PE can have without producing additional communication was developed.

If the PE is the most suitable in its cluster for the specific task is the first information. Retrieving the information is simple due to the separated AHS running in each cluster. The second information needed is, if the PE has an eager value, which is at least as high as the required eager value. This indicates if it is currently suitable enough to execute the task. For this reason, each task gets a system-wide least eager value, which indicates how high the eager value has to be for allocation. If the eager value of a PE is less than this requirement, it passes the task to its underlying cluster. In case the PE is a leaf of the tree, it allocates the task, nevertheless its *eager value* is less to the least eager value of the task. If the task was passed, the representative PE passing the task, will record the sent out eager values in the lower cluster. In case one of the recorded eager values from below is higher than its own, it will copy that as *horizontal eager value*. In case the sent out eager values from the underlying cluster are all lower than its own eager value, the PE will further record the eager values until the next optimization cycle starts. If until then no eager value from the underlying cluster is higher than its own, it withdraws the task again and allocates the task itself, even though its *eager value* is not equal or higher to the least eager value of the task.

5.3.3.2 Self-Healing

The self-healing in such a complex system like the RAHS is important. The system is widely distributed and autonomous. Therefore, failing PEs are not unlikely. The self-healing inside each cluster stays of course untouched and works as in the original AHS. This means that a failed PE is compensated by the other PEs in the same cluster if possible. If the failure cannot be compensated cluster-internally the representative PE in the next higher level is responsible for the *self-healing* of the lost tasks. The representative recognizes the lost tasks through the vanished *eager values* from the lower level. It reacts by revoking the pass and allocating the task even though its *eager value* is less than the least required *eager value* of the task. If the representatives *eager value* is 0 it

does not allocate the task but also stop sending *eager values horizontally*. This either leads to a re-migration of the task to another PE in the same cluster or to an intervention of the clusters representative of the next higher level in case no other PE of the cluster is able to migrate the task. This procedure is possibly repeated until the task ends up at the topmost level. From there the task migrates to another branch of the tree. The *self-healing* ability of this method was also evaluated and the results are presented in section 8.2.2.

Chapter 6

System properties

6.1 The properties of the HAHS

The HAHS is responsible for the decentralized allocation of tasks between the two hierarchical levels. The concept of the HAHS has been presented in chapter 4. The worst-case timings and the communication estimations of the HAHS in terms of the *self-configuration, self-optimization* and *self-healing* are presented in the following sub-chapters.

6.1.1 General

First, the general system and its properties are investigated. Since the HAHS consists of two hierarchical levels, the analysis has to differ between the *inter-cluster* cycle and the *intra-cluster* cycles. Due to the fact that each cycle, whether the *inter-cluster* cycle or the *intra-cluster* cycles, implements a full AHS, all properties are inherited from the original AHS.

Timing If one of the AHS cycles is responsible for x tasks, then the worst-case timing behavior for the *self-configuration*, *self-optimization* and *self-healing* is x hormone cycles (see section 3.6)[1]. Assuming that the *inter-cluster* cycle is responsible for y tasks and the *intra-cluster* cycle i is responsible for x_i tasks, whereby: $i \in \{1...l\}$ and $\sum_{i \in \{1...l\}} x_i = y$. Then, the first estimation for the worst-case time of the entire HAHS is the sum of the worst-case time of the

[1]For the *self-optimization and self-healing* an offset has to be added

inter-cluster cycle and the worst-case time of the *intra-cluster* cycle with the most tasks:

$$WCT_{general_{first}} = y + \max_{i \in \{1...l\}} x_i \tag{6.1}$$

Transferred to the formal definitions from section 4.5 with $T_i \subseteq T$:

$$\begin{aligned} WCT_{general_{first}} &= m + max_{i \in \{1...l\}} |T_i| \\ WCT_{general_{first}} &= m + m = 2m \end{aligned} \tag{6.2}$$

Only the maximum worst-case of the *intra-cluster* cycles is chosen because all *intra-cluster* AHSs work in parallel. Since it is possible that all tasks are distributed to only one cluster, the maximum worst-case for the *intra-cluster* cycle can be m.

However, this estimation can be refined: not only the *intra-cluster* cycles work in parallel, but also the *inter-cluster* cycle works in parallel to the *intra-cluster* cycles. After the first task decision in the *inter-cluster* cycle is completed, the cluster head will send the notification to all PEs in its cluster. The notification hormone unlocks the sending of *eager values* for the task in the next hormone cycle. And the task will be allocated in the following hormone cycle. Consequently, while the *inter-cluster* cycle allocates x tasks, the *intra-cluster* cycles will have allocated $x - 2$ tasks simultaneously. This can be subtracted from the first worst-case estimation:

$$WCT_{general_{second}} = 2m - (m - 2) = m + 2 \tag{6.3}$$

So the final worst-case timing for all three *self-x properties* mentioned before is $m + 2$ for the general HAHS.

Communication The worst-case communication load of the HAHS also origins from the AHS, since it is built of several isolated AHSs. In section 3.7, the worst-case communication load for the AHS was stated to $n \cdot m$. In regards of the *inter-cluster* cycle, this transforms to:

$$WCC_{general_{inter}} = |C| \cdot m \leq l \cdot m \tag{6.4}$$

The n is replaced with l due to the fact that in the *inter-cluster* cycle exist as many participating PEs (*cluster heads*) as clusters exist in the system.

For an *intra-cluster* cycle i the worst-case communication load derives to:

$$WCC_{general_{intra}} = |P_{C_i}| \cdot m = n \cdot m \tag{6.5}$$

The worst-case communication load can be $n \cdot m$ for one *intra-cluster* cycle under the condition that there is only one cluster in the system.

The following sections will discuss the changes and additional overhead of this worst-case analysis when using the different concepts and methods of the HAHS.

6.1.2 Task concepts

In this section, the two different concepts for task organization of the HAHS are analyzed, starting with the *organ task* concept succeeded by the *single task* concept.

6.1.2.1 Organ task concept

The *organ task concept* builds subsets of tasks and calls them *organs*. The *organs* are distributed to the clusters in the *inter-cluster* cycle. Then the tasks of the *organs* are allocated by a PE in the assigned cluster.

Timing At first the timing behavior of an *organ task concept* HAHS is analyzed in detail. For the analysis, the three *self-x properties* are handled separately, starting with the *self-configuration*.

Self-configuration For the analysis of the *organ task* concept the first estimation of the worst-case of the general HAHS (see equation 6.2) is used. Beginning with the *inter-cluster* cycle, it is observable that exactly as many tasks in the *inter-cluster* cycle have to be distributed as *organs* exist in the system. Therefore, the worst-case of the *inter-cluster* cycle is $|O| = k$ hormone cycles. For the *intra-cluster* worst-case timing, it must be considered that after a task is activated in the *inter-cluster* cycle, the cluster head sends an activation hormone to all PEs in its cluster. After receiving the activation hormone, the PEs starts sending *eager values* for the tasks in the corresponding

organ. The effect of the *inter-cluster* and *intra-cluster* cycle working in parallel nearly vanishes. When the last *organ* i is allocated in the *inter-cluster* cycle, two hormone cycles later the assigned cluster starts allocating the $|o_i|$ tasks from the *organ*. Therefore, the worst-case timing of the *self-configuration* is:

$$WCT_{SC_{OrganTask}} = k + \max_{i \in \{1...k\}} |o_i| + 2 \tag{6.6}$$

Self-optimization The *self-optimization* in the *organ task* concept depends on the *offer period* (w). This defines the interval for offering an allocated task to the other PEs. In case a more suitable PE exists in the same cluster, the system needs as many additional hormone cycles as optimizable tasks exist in the corresponding organ. This results in a worst-case self-optimization timing for one task of $w+|o_i|$ for organ i. If a PE from another cluster is more suitable, then the whole organ has to migrate to the cluster with the designated PE. For one *organ* the time for the *self-optimization* is $w + 1 + |o_i|$. Regarding the worst-case, in which all organs have to migrate to other clusters, the worst-case time for the *self-optimization* calculates to:

$$WCT_{SO_{OrganTask}} = w + k + \max_{i \in \{1...k\}} |o_i| + 2 = w + WCT_{SC_{OrganTask}} \tag{6.7}$$

Self-healing If a PE fails in the *organ task* concept, two possible scenarios are conceivable. The first one is that the failure can be compensated in the cluster itself. This means that the tasks from the assigned *organ* of the cluster will be reordered, such that the failure can be compensated. In fact, the tasks from the failed PE will be migrated to a PE in the same cluster. In this case, the self-healing will take $|o_i|$ cycles in worst-case.

The second scenario is the more interesting one. In this scenario the failure cannot be compensated cluster-internally. This results in a necessary organ migration since the tasks of an *organ* cannot be migrated to another cluster without migrating the whole organ to the other cluster. In this case, the organ itself has to be migrated in the *inter-cluster* cycle and then all its belonging tasks have to be allocated in the new cluster in the *intra-cluster* cycle. For the worst-case scenario, it has to be considered that this happens to all organs at the same time, for this reason the worst-case self-healing timing calculates to:

$$WCT_{SH_{OrganTask}} = k + \max_{i \in \{1...k\}} |o_i| + 2 = WCT_{SC_{OrganTask}} \tag{6.8}$$

Communication The next aspect to analyze is the additional communication load produced by an *organ task* HAHS in comparison to the general HAHS (see section 6.1.1). Again, the worst-case is regarded.

In the beginning, the worst-case communication loads of the *inter-cluster* cycle and the *intra-cluster* cycles are estimated. The *inter-cluster* cycle behaves almost like an AHS in terms of communication load. The worst-case is also the start of the system, when all *cluster heads* send out *eager values* for the organ tasks. This ends in a worst-case communication load of:

$$WCC_{OrganTask_{Inter}} = |C| \cdot |O| = l \cdot k \tag{6.9}$$

The worst-case of one *intra-cluster* cycle is reached when all its assigned tasks are not allocated yet and all *organs* are assigned to the corresponding cluster i. Then, all PEs of the cluster send *eager values* for all assigned tasks:

$$WCC_{OrganTask_{Intra}} = |P_{C_i}| \cdot m \tag{6.10}$$

For the entire system, the worst-case is reached when all organs are assigned to the same cluster with the most PEs assigned. Additionally, the suppressors of the *inter-cluster* cycle have to be considered in the calculation, one for each *organ* (k). Therefore, the overall worst-case estimation derives to:

$$WCC_{OrganTask} = \max_{i \in \{1...l\}} |P_{C_i}| \cdot m + k \tag{6.11}$$

6.1.2.2 Single task concept

The second task concept is the *single task concept*, in which the tasks are assigned separately to the clusters in the *inter-cluster* cycle. Consequently, this concept behaves similar as the general concept from section 6.1.1.

Timing At first the worst-case timings of the *single task* concept are analyzed, again split by the three *self-x properties*.

Self-configuration The worst-case timing on the *inter-cluster* level evaluates to m cycles from the original AHS. Two additional cycles are needed for the allocation in the second hierarchical level. This is explained in section 6.1.1.

$$WCT_{SC_{SingleTask}} = m + 2 \tag{6.12}$$

For comparison with the *organ task* concept, an example with 100 tasks ($m = 100$) is used. Consequently, the $WCT_{SC_{SingleTask}}$ is 102 cycles. For the $WCT_{SC_{OrganTask}}$ further information about the organ sizes are needed. Let's assume there exist 5 organs ($|O| = 5$) and all organs have an equal size ($\max_{i \in \{1...|O|\}} |m_{O_i}| = 20$). Then the $WCT_{SC_{OrganTask}}$ is 27 cycles. Equal sized organs are not a necessary requirement for the *organ task* HAHS. That is why inequality among the organ sizes has to be considered for the worst-case estimation. Assuming the most unequal scenario for the example above, one organ would have the size of 96, while the other four organs would have a size of 1. This leads to $WCT_{SC_{OrganTask}}$ of 103 cycles, which is even higher than the worst-case time of the *single task* concept. The equation for this scenario simplifies to:

$$WC_{SC_{OrganTask_{Inequal}}} = k + (m - (k - 1)) + 2 = m + 3 > WC_{SC_{SingleTask}} \tag{6.13}$$

Self-optimization The *self-optimization* in the *single task* concept bases on the same principle as in the *organ task* concept. It depends on the *offer period* (w), too. The worst-case is again the case in which all tasks have to be migrated to new PEs. The difference is that no organs have to be migrated. Therefore, the migration overhead for a single task is less than with the *organ task* concept and calculates to $w + 2$. For the whole system, the worst-case timing for the *self-optimization* calculates to:

$$WCT_{SO_{SingleTask}} = w + m + 2 = w + WCT_{SC_{SingleTask}} \tag{6.14}$$

Self-healing In contrast to the *organ task* concept, the *single task* concept can react more flexibly to failures since there exists no task grouping. The two

described scenarios from above are applicable for the *single task* concept as well.

In the first scenario, all failures are compensated cluster-internally, which ends in a worst-case self-healing timing of $\sum_{\gamma \in C_j} |\Xi_\gamma|$ hormone cycles.

The second scenario, when a task has to be migrated to another cluster, is more complex and the worst-case timing is worse. The lost tasks have to be migrated to another cluster in the *inter-cluster* cycle and afterward migrated to a PE in the new cluster. Therefore, the worst-case self-healing timing behavior for the *single task concept* derives to:

$$WCT_{SH_{SingleTask}} = m + 2 \tag{6.15}$$

Communication The worst-case communication load of the *single task* concept does not differ from the worst-cases stated in section 6.1.1, since it behaves in exactly the same way. The worst-case in the *inter-cluster* cycle appears when all *cluster heads* broadcast *eager values* for all tasks:

$$WCC_{SingleTask_{Inter}} = l \cdot m \tag{6.16}$$

The worst-case for one *intra-cluster* cycle happens when all tasks are assigned to one cluster:

$$WCC_{SingleTask_{Intra}} = |P_{C_i}| \cdot m \tag{6.17}$$

The worst-case for the entire system can be derived from this. When all tasks are assigned to the same cluster, which is simultaneously the biggest cluster, the most communication load is produced. Additionally, the number of *suppressors* in the *inter-cluster* cycle has to be added:

$$WCC_{SingleTask} = |P_{C_i}| \cdot m + m \tag{6.18}$$

When comparing the worst-case communication of the two task set concepts, it is observable that the *organ task* concept produces less communication. This

is because the worst-case communication in the *intra-cluster* cycles is the same, but the worst-case communication in the *inter-cluster* cycle for the *organ task* concept depends on k instead of m for the *single task* concept. From the definition (see section 4.5) it is known that $k \leq m$. So the *organ task* concept at its worst produces the same amount of communication as the *single task* concept. Regarding this one scenario in which k equals m, then there must exist as many organs as tasks in the system. Since each organ must consist of at least one task and all organs are disjoint to each other, each organ will consist of exactly one task. This scenario equals exactly the *single task* concept. This shows that the *single task* concept is a special case of the *organ task* concept. The *organ task* concept produces always less or equal communication than a *single task* concept of the same scale.

6.1.3 Cluster set concepts

The next set of concepts developed are the *cluster set concepts*. In contrary to the section before, not the entire worst-case timing behavior and communication load is examined. Instead, the additional worst-case overhead of the introduced methods are analyzed.

6.1.3.1 static-static

The *static-static* cluster set concept does not introduce any special method into the HAHS. Therefore all estimations of the worst-case timing and communication load from before hold.

6.1.3.2 static-dynamic

In the *static-dynamic* variant the cluster heads are dynamic. This means that the cluster heads are elected in the clusters. The election is realized by allocating a special task in each cluster, indicating which PE is the current cluster head.

Timing In a first step, the worst-case timing of the dynamic cluster head election is examined, again separated by the three *self-x properties*.

Self-configuration The *self-configuration* for the dynamic cluster head method means the process of initially determining a cluster head. As mentioned before, the method is implemented by a special task handled separately from the other tasks by the AHS loop. Therefore, this can be interpreted as AHS with a single task. The original AHS has a worst-case time of m cycles for the *self-configuration*. Consequently, the worst-case *self-configuration* time for the dynamic cluster head method is:

$$WCT_{SC_{dynCH}} = 1 \tag{6.19}$$

This means that after one hormone cycle the clusters have elected a cluster head and can start the usual process for allocation of the tasks. The process is delayed by this one hormone cycle.

Self-optimization The dynamic cluster head also implements the *self-optimization* property, which is reasonable since it uses the mechanism of the AHS. The *suppressor* of the cluster head task vanishes in periodic intervals (w), such that the other PEs have the chance of allocating the cluster head task in case they became more suitable meanwhile. The worst-case timing for the *self-optimization* then is:

$$WCT_{SO_{dynCH}} = w + 1 \tag{6.20}$$

Self-healing The *self-healing* of a cluster head is finished in one cycle. The suppressors vanishes immediately, and the next best PE allocates the cluster head task in the next hormone cycle.

$$WCT_{SH_{DynCH}} = 1 \tag{6.21}$$

Communication The dynamic cluster head method also produces additional communication. The additional communication load is limited to the *intra-cluster* cycle since the cluster head task is handled internally in the separated

intra-cluster cycles. For estimating the worst-case communication load, *self-configuration* can be considered. Nevertheless, this is also applicable for *self-optimization* and the *self-healing*. Let's assume, that all PEs of a cluster are able to become the cluster head. This means that each PE has an *eager value* for the cluster head task. When the system starts, all PEs send out their eager value. This is the worst-case in terms of communication load. Regarding only one cluster i the amount of communication derives to $|P_{C_i}|$. For the complete system, the load of the separated clusters have to be summed up:

$$WCC_{DynCH} = \sum_{i \in \{1...l\}} |P_{C_i}| = n \tag{6.22}$$

Therefore, the worst-case communication load of this method equals the amount of PEs in the system.

6.1.3.3 dynamic-static

The *dynamic-static* concept has predefined static cluster heads, while all other PEs determine dynamically their cluster membership during run-time. For this reason, the cluster heads send out special hormones to all other PEs. Those calculate their cluster membership according to the received hormones.

Timing Again, the timing behavior in terms of worst-case estimations is investigated. As before, the three *self-x properties* will be investigated separately.

Self-configuration The sending of the *cluster accelerators* at the start of the system and the following decision upon the membership consumes one additional hormone cycle:

$$WCT_{SC_{dynCluster}} = 1 \tag{6.23}$$

Likewise with the dynamic cluster head method, all following processes, e.g. the allocation of the tasks, get delayed for this one hormone cycle.

Self-optimization In contrary to the dynamic cluster head method, the dynamic cluster method does not rely on a feedback loop with suppressors. The *self-optimization* is, therefore, faster. A change in the *cluster accelerator* of the cluster head is emitted immediately since the cluster accelerators are broadcasted every hormone cycle anyway. This results in a *self-optimization* worst-case time of:

$$WCT_{SO_{dynCluster}} = 1 \tag{6.24}$$

Self-healing The same reasoning can be used for the *self-healing* property. A cluster head which fails stops immediately sending its *cluster accelerator*. All PEs belonging to the now vanished cluster, reassign themselves to the next best cluster head in the next hormone cycle.

$$WCT_{SH_{DynCluster}} = 1 \tag{6.25}$$

Communication This method also produces additional communication in order to gain flexibility.

In this concept, the differentiation between the *inter-cluster* and the *intra-cluster* cycles is more difficult due to the fact that the clusters are created during the self-configuration time, i.e. before the allocation of tasks begins. For this reason, a separation of the two cycles does not make sense.

In this method all cluster heads in the system constantly send out their *cluster accelerators* in order to form the clusters. This results in l *cluster accelerators*:

$$WCC_{SC_{DynCluster_{Inter}}} = |C| = l \tag{6.26}$$

6.1.3.4 dynamic-dynamic

The last cluster set concept is the *dynamic-dynamic* concept. It combines dynamic clusters with dynamic cluster heads and uses the two methods presented in the *static-dynamic* and *dynamic-static* concepts. There are two ways to combine those concepts: In the first approach (A), the cluster heads are

elected after the start, and then the clusters are determined using the method presented in the *dynamic-static* concept. Another approach (B) is to use a cluster analysis based on the *cluster accelerators* as described in section 4.6.4 and afterward using the method described for the *static-dynamic* concept in order to determine the cluster heads of the clusters. The cluster analysis is only used for the *self-configuration*. The *self-optimization* and *self-healing* works as described in the *dynamic-static* concept.

Timing This concept is examined for its worst-case timing behavior, beginning with the *self-configuration*.

Self-configuration For the *self-configuration* the two approaches mentioned before have to be investigated separately.

Approach (A): In this approach the cluster heads are elected at first. This consumes one additional hormone cycle. Afterward, the dynamic cluster method takes place, as described in the *dynamic-static* concept. Therefore, the *self-configuration* worst-case time is:

$$WCT_{SC_{DynamicDynamic(A)}} = WCT_{SC_{dynCluster}} + WCT_{SC_{dynCH}} = 2 \tag{6.27}$$

Approach (B): Let's regard an *agglomerative* cluster analysis in which it is guaranteed that in each step at least one PE joins each of the final clusters. Then, the clusters are built after $\left\lceil \frac{n}{l} \right\rceil$ steps in the worst-case. For a *divisive* cluster analysis (which builds each step at least one final cluster) the worst-case estimation derives to l. So the worst-case for this approach is:

$$WCT_{SC_{DynamicDynamic(B)}} = \max(\left\lceil \frac{n}{l} \right\rceil, l) + WCT_{SC_{dynCH}} = \max(\left\lceil \frac{n}{l} \right\rceil, l) + 1 \tag{6.28}$$

Self-optimization For the *self-optimization*, the worst-case timings of the two methods have to be combined again:

$$WCT_{SO_{DynamicDynamic}} = WCT_{SO_{dynCH}} + WCT_{SO_{dynCluster}} = w + 2 \tag{6.29}$$

Self-healing The *self-healing* of a *dynamic-dynamic* HAHS is, like in the *self-optimization*, a concatenation of the two *dynamic* methods from the *static-dynamic* and *dynamic-static* concepts. The worst-case timing for the *self-healing* in this concept is:

$$WCT_{SH_{DynamicDynamic}} = WCT_{SH_{DynCH}} + WCT_{SH_{DynCluster}} = 2 \tag{6.30}$$

Communication The worst-case communication load for this concept is the combination of the communication loads of the two methods used in this scenario. This results in:

$$WCC_{DynamicDynamic} = WCC_{DynCH} + WCC_{DynCluster} = n + l \tag{6.31}$$

6.1.4 CEA Methods

As described in section 4.7, different methods were developed in order to determine the *cluster eager values*. Those methods have an impact on the worst-case timing and the communication load, too. As in the cluster set concepts, only the worst-case timing and the communication load of the special mechanisms introduced by the CEA methods are analyzed.

6.1.4.1 Predefined cluster values

For the sake of completeness the *predefined cluster values* are mentioned here, too. This method does not have any influence in terms of worst-case timing or communication load since no special mechanism is introduced. All *cluster eager values* are static and determined by the designer.

6.1.4.2 Recalculation

In the *recalculation* of cluster eager values (as well as in the *mimic best* method), the cluster member PEs send eager values for all tasks in periodic intervals (r). The cluster heads use those eager values to update their cluster eager values.

Timing The following paragraphs analyze the worst-case timing of this method by means of the three different *self-x properties*: *self-configuration*, *self-optimization* and *self-healing*.

Self-configuration Ideally, the first recalculation takes place at the start of the system. The cluster heads receive those hormones in the first cycle. They calculate their cluster eager values and send them in the second cycle. That is why this method needs an additional cycle. All succeeding processes are delayed by this one hormone cycle.

$$WCT_{SC_{CEARecalc}} = 1 \tag{6.32}$$

Self-optimization Since the *recalculation* updates the cluster eager values only in certain periodic intervals (r), the worst-case is when a change occurs exactly after such a recalculation. When the recalculation process takes place, the new value of the allocating PE is transmitted to the cluster head. Therefore, the worst-case timing of the self-optimization for this method is:

$$WCT_{SO_{CEARecalc}} = r \tag{6.33}$$

Self-Healing Equal to the *self-optimization*, the *self-healing* of the *cluster eager value* generation in the *recalculation* method is dependent from the periodic interval r. The change in the *cluster eager value* caused by a failing PE is taken into account after r hormone cycles latest.

$$WCT_{SH_{CEARecalc}} = r \tag{6.34}$$

Communication The *recalculation* causes all PEs to broadcast all their eager values in periodic intervals. The worst-case communication load of this method is:

$$WCC_{CEARecalc} = n \cdot m \tag{6.35}$$

This worst-case communication load is produced every r hormone cycles.

6.1.4.3 Greatest hormone

In this method, the *cluster eager values* are calculated throughout an initial sending of the highest eager value from each PE to its cluster head. The cluster head uses those values as its *cluster eager value*. For the tasks it did not receive any *eager values*, it calculates a *cluster eager value* heuristically based on the received *eager values* of the other tasks. The cluster head constantly updates those values by listening to the *eager values* sent in the *intra-cluster* cycle.

Timing The worst-case timing behavior of the *greatest hormone* method is again analyzed for the three *self-x properties*.

Self-Configuration As mentioned before, all PEs initially send their highest *eager value* to their cluster head. Then the cluster head takes those values as *cluster eager values* or calculate heuristic values. The process of sending the maximum eager values needs one hormone cycle:

$$WCT_{SC_{GreatestHormone}} = 1 \tag{6.36}$$

Self-optimization In case that the suitability of a PE for a task changes the cluster head either receives an updated eager value in the next hormone cycle if the task is not allocated yet, or when the task is offered again. Consequently, the worst-case timing of the *greatest hormone* method with respect to the *self-optimization* is:

$$WCT_{SO_{GreatestHormones}} = w \tag{6.37}$$

Self-healing When a PE fails in the HAHS using the *greatest hormone method*, the *intra-cluster* suppressors for its allocated tasks vanish. The remaining PEs in the cluster send out *eager values* for those tasks again. This leads to an update of the *cluster eager values* for those tasks in the cluster head. Then the tasks are migrated to other PEs in the same cluster or might be migrated to other clusters in the next optimization cycle. However, it might happen that no remaining PE can allocate the lost tasks from the failed PE. In this case,

the cluster head updates its *cluster eager values* for the lost tasks in the next optimization cycle. Therefore, the worst-case timing for the *self-healing* of the greatest hormone method is:

$$WCT_{SH_{GreatestHormone}} = w \tag{6.38}$$

Communication In this method, the PEs only sends the eager values for the task with the highest *eager value*[2]. The worst-case communication load for this method builds up when all PEs send those maximum *eager values* at the beginning. Each PE sends exactly one. Therefore, the worst-case communication load is:

$$WCC_{Greatest} = n \tag{6.39}$$

6.1.4.4 Magnitude of eager vectors

In this method, the cluster eager values are calculated by the magnitudes of the eager vectors which are sent by each cluster member to the cluster head. The magnitudes are sent every cycle and processed for the cluster eager values in the following cycle.

Timing As before, the first aspect for analyzing is the worst-case timing behavior of this method.

Self-Configuration In the first hormone cycle, all PEs send out their *magnitudes*. In the second hormone cycle, the cluster heads choose their *cluster eager values* and broadcast it afterward. Therefore, the worst-case timing for the *self-configuration* is:

$$WCT_{SC_{Magnitudes}} = 1 \tag{6.40}$$

[2]if two tasks have the highest value, the eager value of the one with the lower id is sent

Self-Optimization In this method, the cluster eager values rely on the magnitudes of the eager vectors of the PEs. A change in the environment that causes a change in the suitability takes effect at latest when the task is offered. Hence, the worst-case timing is exactly the offer period:

$$WCT_{SO_{Magnitudes}} = w \tag{6.41}$$

Self-Healing The *magnitude of eager vectors method* interprets all *eager values* as a vector and calculates its magnitude. A PE failure leads either to a *cluster-internal* migration or a change in the *cluster eager value*. The change depends on the chosen calculation method of the *cluster eager value*, e.g. by which operation the magnitudes will influence the *cluster eager value*. A migration will take place only in the next optimization cycle (w). Therefore, the worst-case timing for the *self-healing* calculates to:

$$WCT_{SH_{Magnitude}} = w \tag{6.42}$$

Communication The *magnitude of eager vectors* method behaves similar to the *greatest hormone* method in terms of communication load. However, instead of sending each cycle only the eager value of the task with the highest eager value, it sends the magnitude of all eager values. Therefore, the communication load is the same as with the *greatest hormone* method.

$$WCC_{Magnitudes} = WCC_{Greatest} = n \tag{6.43}$$

6.2 The properties of the RAHS

This section will analyze the timing behavior and the communication load of the RAHS, the advanced development of the HAHS.

6.2.1 Timing behavior

For analyzing the timing behavior of the general RAHS, a worst-case estimation for an 1-level RAHS is given first. An 1-level RAHS is a system with no

hierarchical aspect. Therefore, the 1-level RAHS equals an AHS. From the AHS it is known that the worst-case timing scales with the amount of tasks m in the system. Hence, the worst-case timing for the *self-configuration* and *self-healing* for this RAHS is:

$$WCT_{SC/SH_{1level}} = m \tag{6.44}$$

Similarly, the *self-optimization* estimates to:

$$WCT_{SO_{1level}} = w + m \tag{6.45}$$

The next step is a 2-level RAHS. Now a hierarchical super-level, consisting of one cluster ($c_1 = 1$), is introduced. The PEs in the lower level split up in to c_0 clusters. All tasks are distributed to the PEs in the upper level (1) and then either passed downward or allocated directly to the PE in the upper level. For the upper level this ends up into m hormone cycles as worst-case timing. Each cluster in the lower level ends up allocating m tasks as absolute worst-case. Therefore, the worst-case is also m hormone cycles in the lower level. Combining both levels, this sums up to $2m$ hormone cycles. However, equally to the HAHS the upper and lower level do not work sequentially but partly parallel. Therefore, the worst-case is identical to the worst-case of the HAHS:

$$WCT_{SC/SH_{2level}} = m + 2 \tag{6.46}$$

Accordingly, the *self-optimization* worst-case timing calculates to:

$$WCT_{SO_{2level}} = w + m + 2 \tag{6.47}$$

The next further step is to add a new lowest level of clusters. For the worst-case timing this means that after the last task has been distributed in the second level ($= WCT_{2level}$), this task still has to be distributed in the third level. The sending of the activation hormones and the allocation itself costs additional 2 hormone cycles. Consequently, the worst-case timing sums to:

$$WCT_{SC/SH_{3level}} = WCT_{SC/SH_{2level}} + 2 = m + 4 \tag{6.48}$$

and:

$$WCT_{SO_{3level}} = WCT_{SO_{2level}} + 2 = w + m + 4 \tag{6.49}$$

So with each additional level two hormone cycles sum up on the worst-case timing. The general worst-case timing for the RAHS derives to:

$$WCT_{SC/SH} = m + (2 \cdot (l - 1)) \tag{6.50}$$

and:

$$WCT_{SO} = w + m + (2 \cdot (l - 1)) \tag{6.51}$$

6.2.2 Communication load

The second aspect under investigation is the communication load produced by the RAHS. Especially, the worst-case communication load is interesting for designing a system. Again, the analysis starts with a 1-level RAHS.

As shown before a 1-level RAHS is the same as the original AHS. Therefore, the worst-case communication load is the same, too:

$$WCC_{1level} = n \cdot m \tag{6.52}$$

Regarding x-level RAHSs the communication load has to be analyzed for each level separately. This is oriented on the communication load of the HAHS. The number of PEs of the biggest cluster in the respective level multiplies with the number of tasks in the entire system:

$$WCC_{xlevel} = \sum_{g \in L} \max_{i \in \{\forall c : \mu(c) = g\}} (|P_i|) \cdot m \tag{6.53}$$

Additionally, the vertical communication has to be considered for RAHSs with more than one level. The vertical communication is limited to the vertical activation hormones. This means that in the worst-case each level, except the last level, sends m activation hormones. This results in a hormone amount of:

$$WCC_{xlevel} = (|L| - 1) \cdot m + \sum_{g \in L} \max_{i \in \{\forall c : \mu(c) = g\}} (|P_i|) \cdot m \tag{6.54}$$

6.2.3 Methods of the RAHS

Two methods for the RAHS were presented in section 5.3.2 and 5.3.3. Both methods are investigated in this section, regarding their impact on timing behavior and communication load.

6.2.3.1 Max eager value

In the first method the maximum *eager values* of each leaf PE are forwarded to the root PEs. Each node PE which receives the maximum *eager values* of the cluster members from the underlying cluster forwards only the maximum of the received *eager values*.

Timing behavior Due to the initial forwarding of the maximum *eager values* the worst-case allocation time will be increased by the number of levels.

$$WCT_{SC_{MAXEA}} = |L| \tag{6.55}$$

This only holds if all allocations based upon heuristic values (decisions for those tasks whose *eager values* were not forwarded) lead to an allocation in the leaf PEs. In case the underlying PEs cannot allocate the task, the upper representative revokes its allocation after a certain waiting time (v). If no other PE in the same cluster can allocate the task, an additional time of h cycles (the decay time of the vertical hormones) is needed to revoke the pass in the upper level. The overall worst-case estimation for this concept was stated and explained in [39] and derives to:

$$WCT_{SC_{MAXEA}} = |L| + m + (2^{|L|-1} - 1) \cdot v + (2^{|L|-1} - |L|) \cdot h + (2^{|L|-1} - 1) \tag{6.56}$$

Communication load The additional worst-case communication load produced by this concept occurs in the first cycle, when all leaf PEs send out their maximum eager value to their upper representative. Therefore, the maximum worst-case communication load produced by this method is the number of leaf PEs in the system.

$$WCC_{MAXEA} = |LP| \tag{6.57}$$

6.2.3.2 Least eager value

In this method, the tasks are allocated by those PEs having an eager value which is at least as high as the required *eager value* of the task. This means, the tasks can already be allocated by a PE in a non-leaf level.

Timing behavior The timing behavior of this method depends on the number of tasks which at first are passed down the half-tree in which they cannot be allocated. Those tasks have to be remigrated in worst-case to the other half-tree of the system. For the worst-case it has to be assumed that this happens to the last task which will be decided on in the first level. This task will be passed down one half-tree and afterward remigrate to the other half. This adds up to

$$WCT_{SC_{LEAST}} = m + |L| + |C| \cdot (w + 1) \tag{6.58}$$

hormone cycles (w is the self-optimization period).

Communication load This method does not produce additional communication load.

Chapter 7

Systems Implementation

This chapter presents the implementations of the HAHS and RAHS which were used to evaluate the concepts presented in this thesis (see chapter 8). The aim of this chapter is not to give a full review of the software. However, a short overview is given. All presented software is included on the CD, appended to this thesis. The AHS was firstly developed and evaluated in a simulator. Afterward, the system was developed as a middleware for Windows and Linux operating systems. This procedure was repeated for the HAHS. For the RAHS, only a simulator was developed since the concept of the system is still evolving.

7.1 Simulator

This section presents the simulator developments for the HAHS and RAHS.

7.1.1 HAHS

As mentioned before, a simulator was developed for the AHS. The simulator was used to develop and evaluate the system itself as well as its extension components.

Due to the broad correspondence in the concepts of the AHS and the HAHS, the AHS simulator was extended to the HAHS simulator. The simulator is written in C++ and uses the Microsoft Foundation Classes (MFC) library [42] in order to build a Graphical User Interface (GUI) (see figure 7.1 for a screenshot).

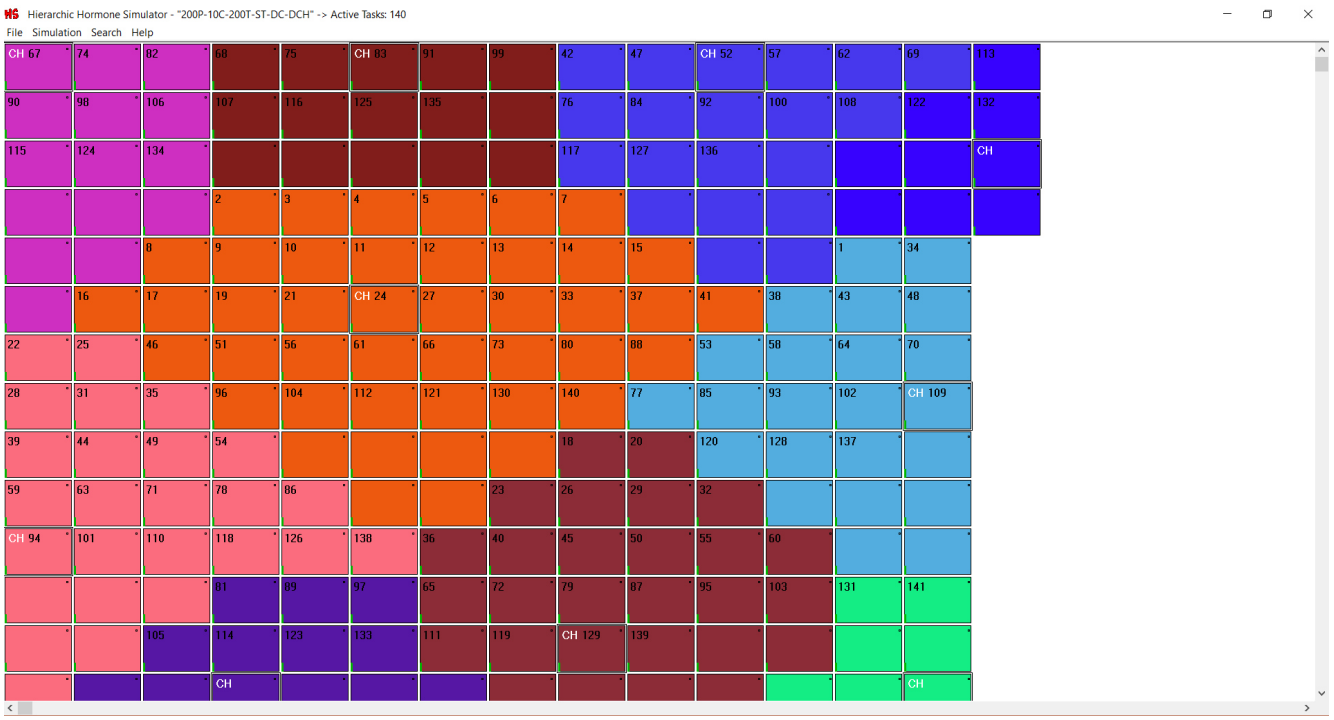


Figure 7.1: Screenshot of the HAHS simulator. Double framed rectangles are cluster heads. The cluster memberships is expressed through the color of the rectangles.

The main class of the simulator is the class representing a PE (*CProcessingElement*, see figure 7.2). Each PE has two lists for organizing its tasks. One list is for its *intra-cluster* cycle (*taskList*) and a second one (*clusterTaskList*) exists in case the PE is a cluster head. Each task list consists of the tasks with their corresponding hormone values and sub-lists for receiving hormones from other PEs. Hence, the received hormones which were sent for a task are saved in the corresponding sub-list for the task. Furthermore, the PE has lists for receiving hormones. Indeed, those lists only save *global* hormones (e.g. *load suppressors*).

In case the *organ task* concept (see section 4.4.1) is used, each PE additionally has a list of organs. The organs, in turn, represent a list of tasks. As soon as the organ is won by the PE in the *inter-cluster* cycle, it activates all tasks in the organ object.

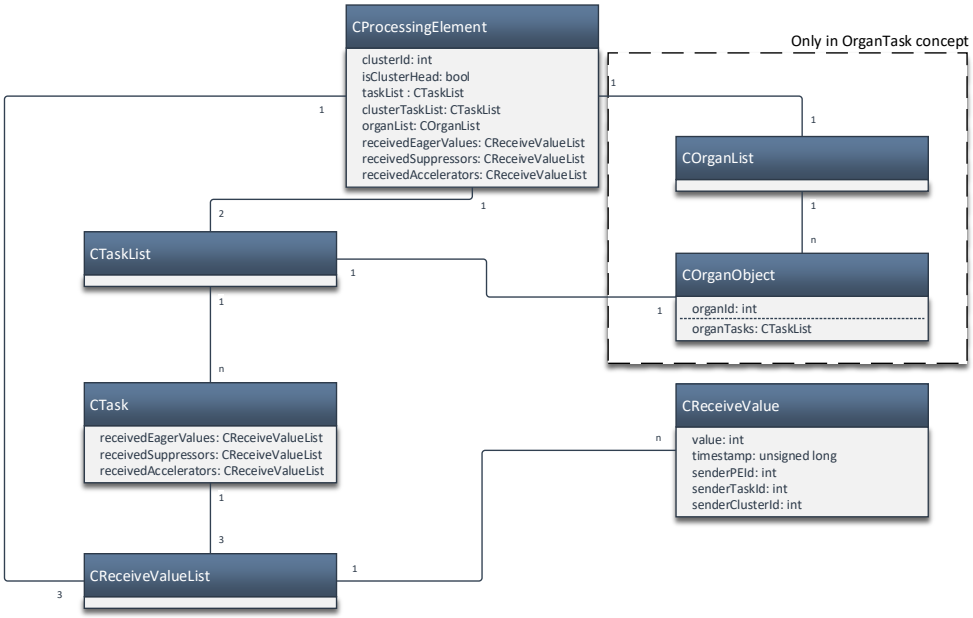


Figure 7.2: A simplified class diagramm of the HAHS simulator

7.1.2 RAHS

Similar to the HAHS, a simulator for the RAHS has been developed. However, in contrast to the HAHS the RAHS simulator is a complete new development and not an extension of the AHS simulator. A further extension of the AHS simulator into the direction of the RAHS would not have been feasible. The concept of levels did not fit well into the structure of the AHS simulator.

The new RAHS simulator is written in C++, too. However, this time the QT5 Framework [59] is used in order to create a GUI (see figure 7.3 for a screenshot).

Similar as before, one of the main classes in this object-oriented software is the PE itself (*CProcessingElement*, see figure 7.4). One of the main differences is the PE class not having any task lists. Those are now associated to the level instances of the PE. The level instances also have lists for storing received global hormones. Therefore, a level instance represents more or less a PE from the HAHS simulator.

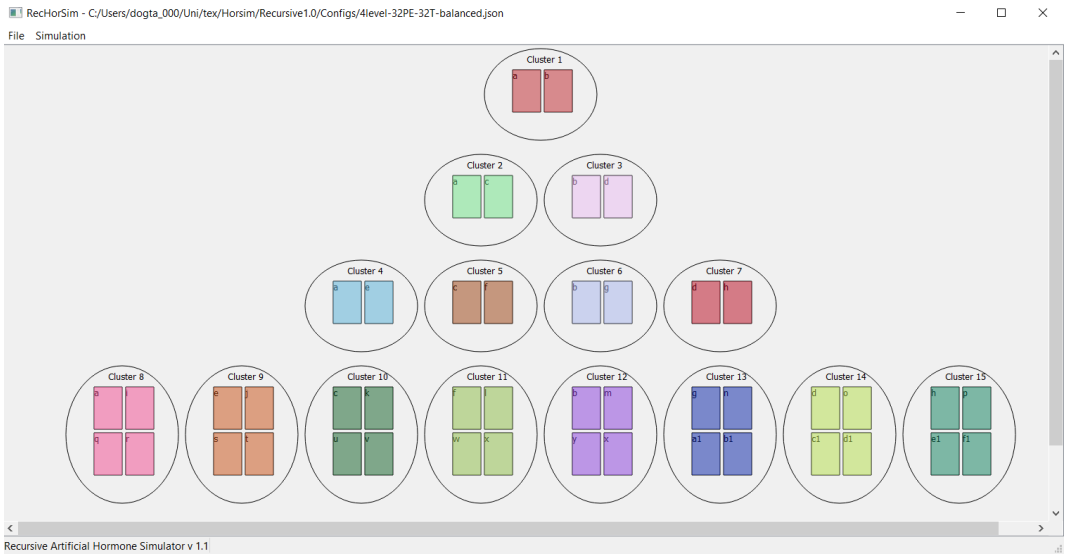


Figure 7.3: Screenshot of the RAHS simulator

7.2 Middleware

The middleware of the AHS was built for creating applications which can take advantage of the robust, decentralized and self-organizing concept. For instance, the middleware was used in a composite with the Artificial DNA [10] to build a robust, fault-tolerant self-balancing two-wheel vehicle (comparable to a segway) [12].

This middleware was taken as a base for a new middleware implementing the HAHS. The intended purpose of the middleware is to process high scaled multi-task applications on a network of connected computers. The user and administrator do not have to coordinate the allocation of the tasks among the system. This will be managed autonomously and failures will be compensated without interfering with the computation.

For the RAHS, no middleware was implemented. This is due to the fact that the final concept of the system is still evolving. Additional research based on simulations has to be done before developing a new middleware.

7.2.1 HAHS

As mentioned before, the AHS middleware was extended to the HAHS middleware. It is written as a static linked library in C and can be compiled for

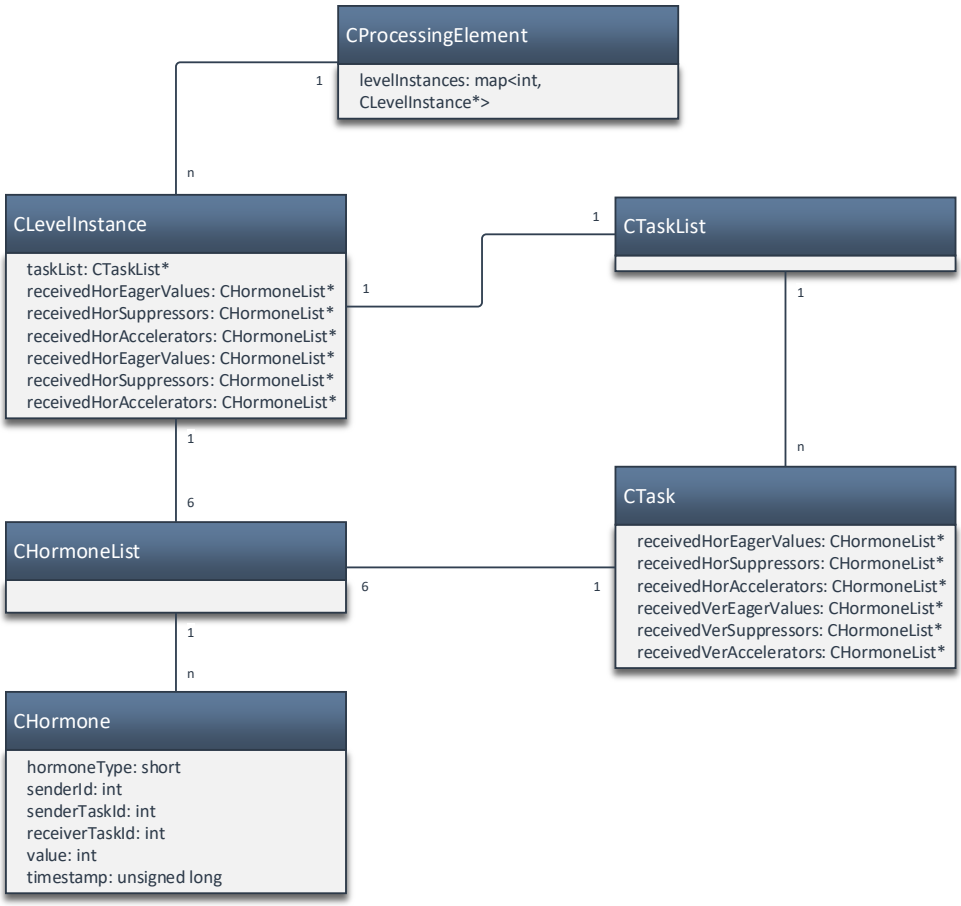


Figure 7.4: A simplified class diagramm of the RAHS simulator

Windows and Linux operating systems.

In order to use the middleware, a program needs to bind the static library. It instantiates itself as PE. The system then will handle the task allocation autonomously and the instantiated PE starts allocated tasks independently (see figure 7.5). The middleware is built upon modules (*HAHSBasicCommunication* for instance), which can be easily replaced in order to support other communication interfaces or operating systems. Several independent PEs can run simultaneously on the same computer, as long as each PE program uses its own HAHS library.

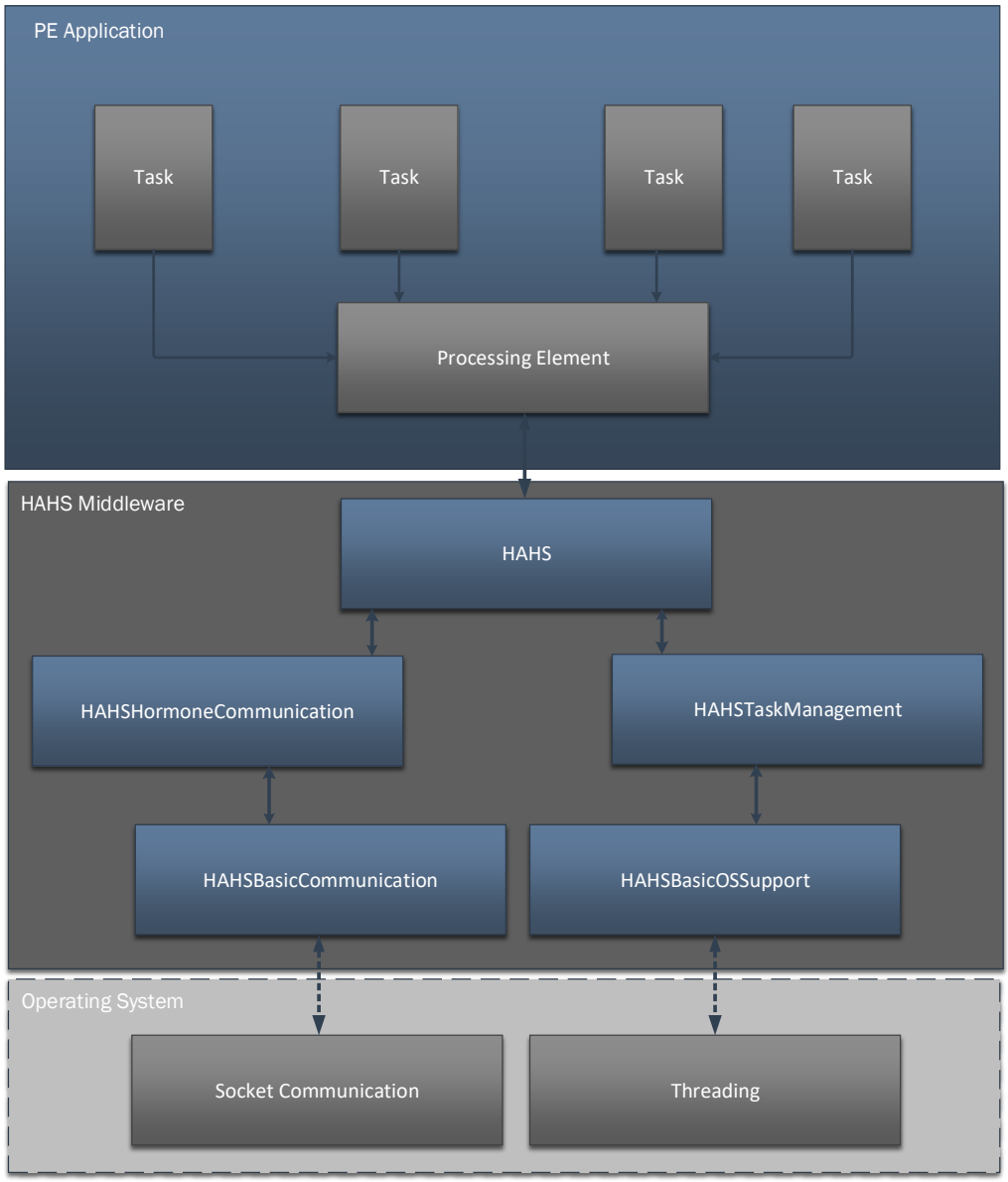


Figure 7.5: A simplified module diagram of the HAHS middleware and a single PE

Chapter 8

Evaluation

This chapter shows the evaluation of the two systems presented in this thesis. Due to the vast number of combinations of the different methods and scenario configurations in each system, not all of them were evaluated, instead the most suitable of those combinations were chosen.

In order to evaluate the two systems, criteria for the system's performance and stability have to be defined. An evaluation of the time needed for the *self-configuration* is performed to evaluate the performance. This also shows if and when the system finds a stable state for the given scenario. A second performance criterion is the communication load produced by the system during *self-configuration*. The important part of this evaluation is the comparison with the communication load of the original AHS. The reduction of this communication load is a major motivation for the development of the HAHS and RAHS. The focus lays on the comparison to the original AHS, but this evaluation will also compare the communication load of the different methods and concepts to each other. At last, the evaluation of this thesis will also examine at the emergence (see chapter 2.3) of the system after the *self-configuration*, *self-healing* and *self-optimization*.

A complete overview of all evaluations can be seen in table 8.1.

8.1 HAHS

For the evaluation of the HAHS, three different main scenarios were prepared and evaluated in the HAHS simulator (see chapter 7.1.1). These scenarios

System	Properties/Concepts/Variants			Section
HAHS	Self-Configuration	Timing behavior	Organ task	8.1.1.1
			Single task	
		Communication load	Organ task	8.1.1.2
			Single task	
		Emergence	PE Emergence	8.1.1.3
			Cluster Emergence	
		Cluster Regulation		8.1.1.4
	Self-Healing	Timing behavior	Organ task	8.1.2
RAHS	Max eager value			8.2.1
	Least eager value			8.2.2

Table 8.1: Overview of the evaluations

consist of 1000 PEs and a total of 1000 tasks. A high scale of PEs and tasks was chosen in order to evaluate the scalability of the system. Of course the number of clusters matters in a scenario and influences the performance. That is why, three different cluster amounts for this scenario were evaluated: 10, 50 and 100 clusters. All clusters were evenly sized, meaning each cluster had the same amount of PEs. Additionally, all PEs were equal in terms of suitability for the tasks. This means the scenarios are a net of homogenous PEs, even though the system can handle heterogeneous PEs. A heterogeneous net of PEs would produce less communication since not all PEs would apply for all tasks and, therefore, not all would send eager values for all tasks. A system handling this amount of homogeneous PEs is, therefore, also able to handle the same amount of heterogeneous PEs.

8.1.1 Self-configuration

At first, the *self-configuration* is examined. The *self-configuration* is the phase of the system after the start. It finishes when an initial working state is found. For the HAHS this state is reached as soon as all tasks are allocated somewhere.

8.1.1.1 Timing behavior

The first essential aspect under evaluation is the proper functioning of the system, meaning that all tasks are allocated and processed by the system. For this reason, the number of allocated tasks in the system over the time is observed. Simultaneously, this is utilized for evaluating the timing behavior.

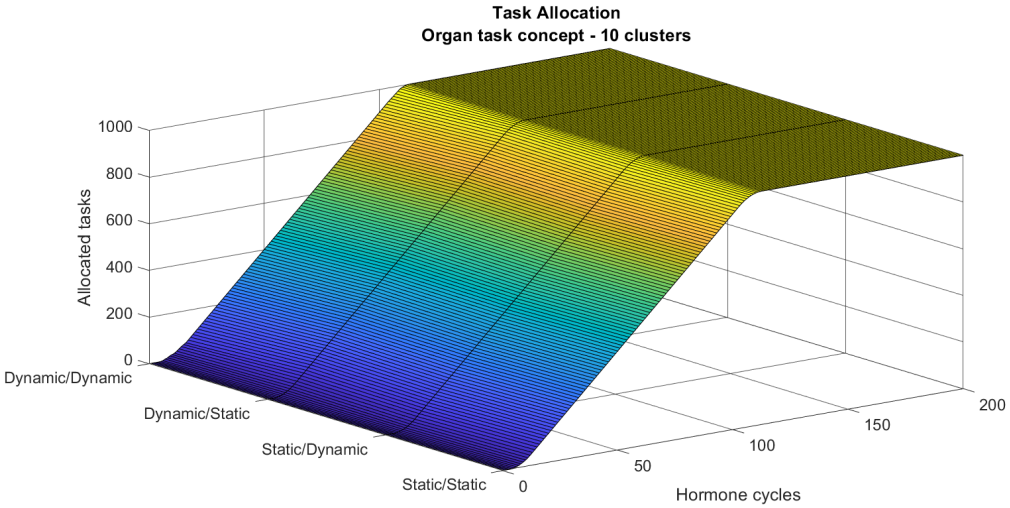
Organ task concept The *organ task concept* combines the tasks in disjoint task groups, called *organs*. The PEs will apply for the organ in the *inter-cluster cycle*. Therefore, it is expected to be faster in terms of allocation time than the original AHS. The scenarios were configured to have exactly the same number of *organs* as clusters exist in the system. Hence, the 10 clusters scenario has fewer *organs* than the 50 clusters scenario.

The four different combinations of cluster and cluster head concepts (*static clusters and static cluster heads, static clusters and dynamic cluster heads, dynamic clusters and static cluster heads, dynamic clusters and dynamic cluster heads*) in the 10 clusters scenario show the same behavior (see figure 8.1). The number of allocated tasks increases over the time linearly until it reaches 1000, which is the number of tasks in the system (see figure 8.1a). Plotted to 2D (see figure 8.1b) it is visible that the dynamic cluster heads versions start a little later in allocating the tasks. The reason for that is the allocation of the cluster head which has to be finished first, before the application of the other tasks can start.

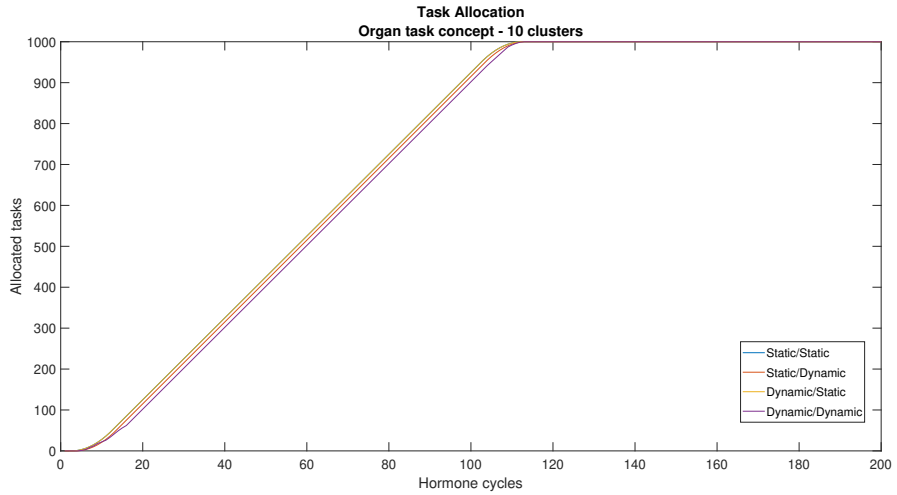
A similar observation can be made regarding the task allocation of the 50 clusters scenarios with the *organ task concept* (see figure 8.2).

In comparison to the 10 clusters scenario from before, this scenario reaches the full task-allocation earlier (see figure 8.2b). While it took 100-120 hormone cycles (timestamps) in the 10 clusters scenarios (see figure 8.1b), the 50 clusters scenarios only needed 60 - 80 hormone cycles. This behavior is related to the cluster respectively *organ* number. The more *organs* exist, the less tasks are in each *organ*. When allocating an *organ*, all its tasks will be unlocked for the cluster, resulting in a worst-case allocation time of exactly the number of tasks in the *organ* (see 3.6). Having few, but bigger clusters/*organs* the allocation time increases and, therefore, the overall *self-configuration* time increases, too.

The last *organ task* scenarios examined have 100 clusters/*organs* (see figure



(a) Surface plot of the allocated tasks

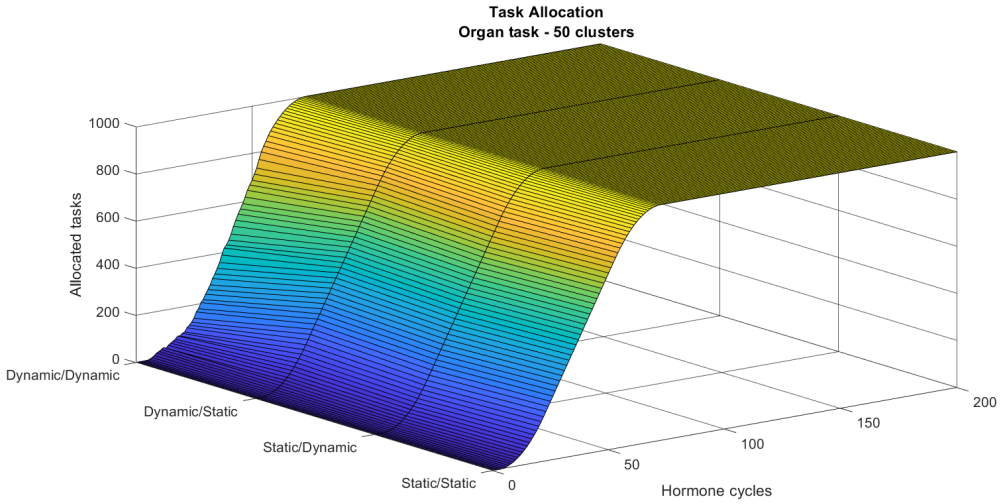


(b) Line plot of the allocated tasks

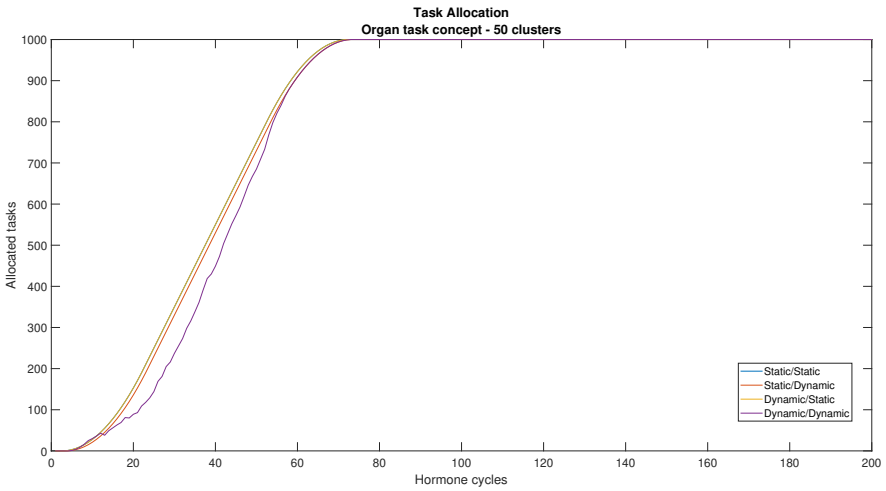
Figure 8.1: Allocated Tasks over the time of the 10 clusters, organ task scenario.

8.3). Again, the scenarios show the same behavior among the four different concept combinations as before. Besides that, it also shows that these scenarios needed 100 - 120 hormone cycles until they reached the full task-allocation, like the 10 clusters scenarios, too. Even though these scenarios have more and smaller cluster/*organs* the time needed increased again in comparison to the 50 clusters scenarios.

The explanation for this increase, despite the smaller sized *organs*, is the increased number of organs. For each additional *organ*, the worst-case allocation



(a) Surface plot of the allocated tasks

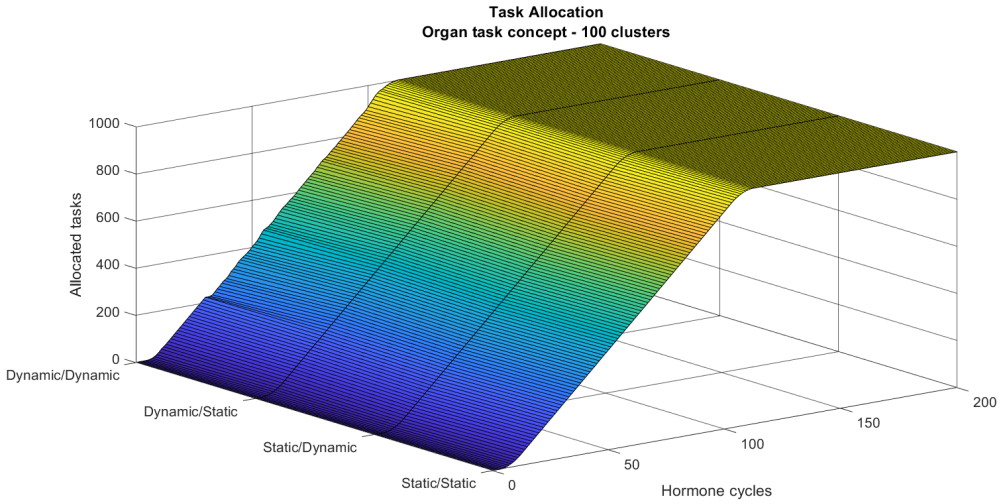


(b) Line plot of the allocated tasks

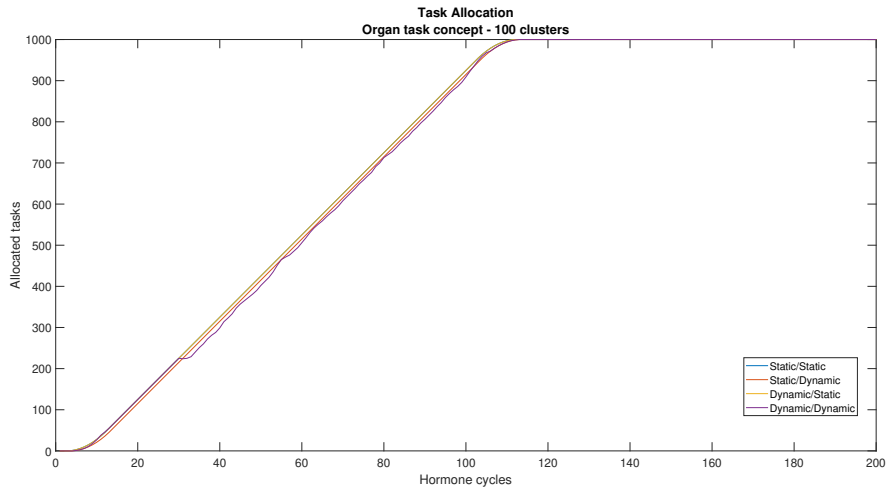
Figure 8.2: Allocated Tasks over the time of the 50 clusters, organ task scenario.

time in the *inter-cluster* cycle is increased by one.

In order to decrease the worst-case *self-configuration* time, the right number of *organs* has to be chosen. For this reason, the relationship of the *inter-cluster* and *intra-cluster* cycle has to be observed. For the *inter-cluster cycle*, the worst-case timing is exactly the number of organs ($k = |O|$). For the *intra-cluster cycles*, the worst-case timing is the number of tasks per organ ($\frac{|T|}{k}$). Additionally, one cycle has to be considered for the unlocking of the tasks in the *intra-cluster* cycle. In case a dynamic cluster head method (DCH) is used,



(a) Surface plot of the allocated tasks



(b) Line plot of the allocated tasks

Figure 8.3: Allocated Tasks over the time of the 100 clusters, organ task scenario.

a second additional cycle is needed for this method. The overall worst-case is therefore

$$f(k)_{SCH} = \left\lceil k + \frac{|T|}{k} + 1 \right\rceil \tag{8.1}$$
$$f(k)_{DCH} = \left\lceil k + \frac{|T|}{k} + 2 \right\rceil$$

Figure 8.4 shows the worst-case times for a 1000 tasks *organ task* HAHS

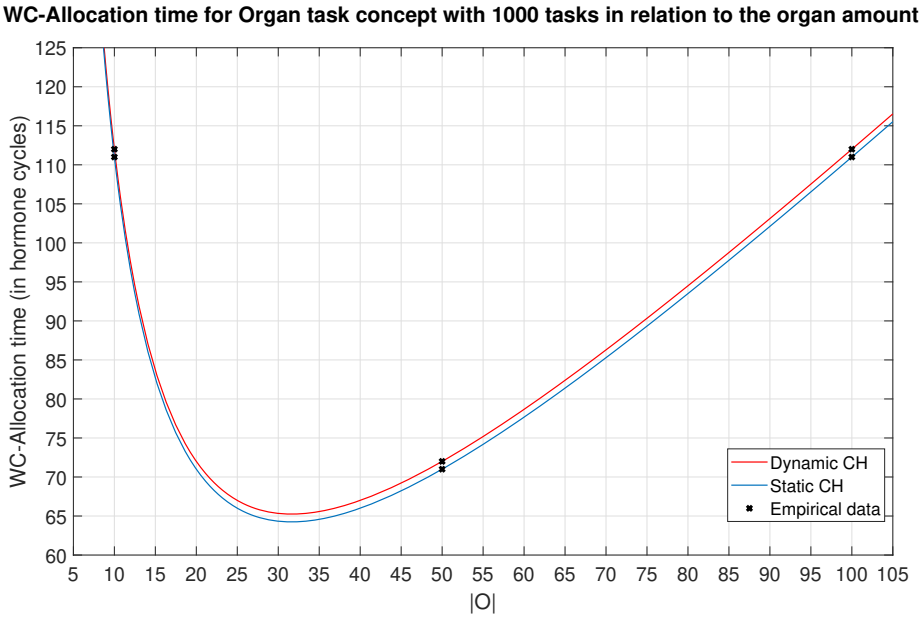


Figure 8.4: The worst-case time for the self-configuration needed by an organ task HAHS with 1000 tasks in dependance of the organ number

with different *organ* numbers. In order to determine the best number of organs the global minimum of the function $f(k)$ has to be found [1].

For the present scenario, the minimum of the function calculates to:

$$
\begin{aligned}
f(k) &= k + \frac{1000}{k} && (8.2)\\
f(k)' &= 1 - \frac{1000}{k^2} = 0 && \text{for } k > 0\\
\Rightarrow k &= 10 \cdot 10^{\frac{1}{2}}\\
k &= 31.6228
\end{aligned}
$$

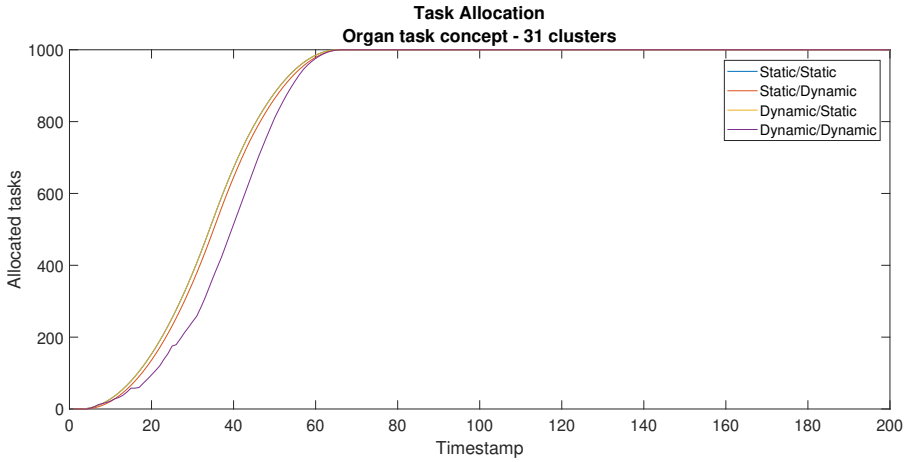
So the perfect *organ* number for these scenarios, is between 31 and 32. Generalized, the perfect *organ* number calculates to:

$$k = \sqrt{|T|} \tag{8.3}$$

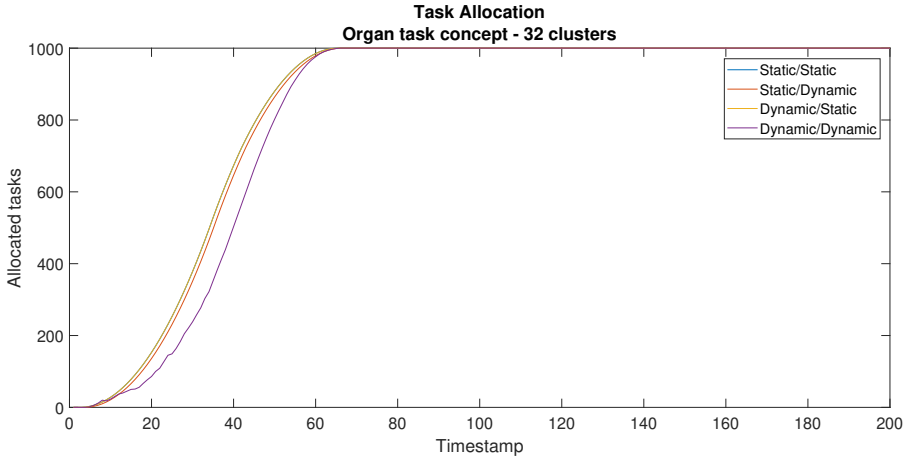
To verify this, two additional scenarios were created, one with 31 clusters and one with 32 clusters.

Both scenarios (see figures 8.5a and 8.5b) behave as predicted in terms of the task allocation time. The scenario with 31 clusters (see figure 8.5a)

[1] This is done by differentiating the function and finding the roots of the differentiated function. Afterward, the roots have to be checked if they are minimum or maximum.



(a) Allocated Tasks over the time of the 31 clusters, organ task scenario as line plot.



(b) Allocated tasks over the time of the 32 clusters, organ task scenario as line plot.

Figure 8.5: Allocated tasks over the time of the 31 and 32 clusters, organ task scenario.

needs $\lceil 64.258 \rceil = 65$ hormone cycles for the static versions and $\lceil 65.258 \rceil = 66$ hormone cycles for the dynamic versions according to equation 8.1. The simulated scenarios finished the *self-configuration* at exactly these time points.

The 32 clusters scenario (see figure 8.5b) needs $\lceil 64.25 \rceil = 65$ respectively $\lceil 65.25 \rceil = 66$ hormone cycles according to the formula. The empirical simulations results confirm this.

In comparison to a similar scaled AHS, the results show a speed-up in the allocation time when using an *organ task* HAHS. The worst-case allocation time of the AHS is m hormone cycles (the amount of tasks in the system, see section3.6), while the *organ task* HAHS only needs $\sqrt{m}$ hormone cycles in

worst-case.

Single task concept In contrast to the *organ task* concept, the *single task* concept treats every task independently in the *inter-cluster* cycle. This results in a more flexible handling of the task set. Problems as described in section 4.6.4.1 are completely avoided. Nevertheless, dealing with the complete task set in the first hierarchical level extends the allocation time in this level. In comparison to the *organ task* concept, the allocation time is extended by the factor of $\frac{m}{l}$.

As described in section 4.7, the *single task* concept needs a method to depict the suitability from the cluster children to their cluster head. Two methods were evaluated in order to demonstrate their influences on the task allocation and the communication load. The first method is the *Recalculation of cluster eager values* (see section 4.7.1), in which all cluster children send all their current eager values for all tasks in periodic time intervals to their cluster head. This method is abbreviated in the following with: *CEASUM*. The other method evaluated is the *Magnitude of eager vectors* (see section 4.7.3). In this method all cluster children constantly build the magnitude value of the eager value vector and send this value to their cluster head in each cycle. This method is abbreviated in the following with: *EAGER*.

For the evaluation, a scenario with 50 clusters and again 1000 PEs and 1000 tasks was used. All four cluster concepts (see section 4.6) were evaluated, consisting either of *static* or *dynamic* cluster heads and either of *static* or *dynamic* clusters. In combination with the cluster eager value methods this results in 8 different evaluations for the scenario.

static-static The evaluations of the first two runs with *static* clusters and *static* cluster heads (see figure 8.6) show that both *cluster eager value* methods have the same behavior in the task allocation. Both reach the full allocation after 1003 hormone cycles, as analyzed in chapter 6 (1002 hormone cycles for the *single task* concept and one additional cycle for the *cluster eager value* method).

static-dynamic When letting *static* clusters elect their cluster's head (*dynamic* cluster heads), the task allocation will take one additional cycle for

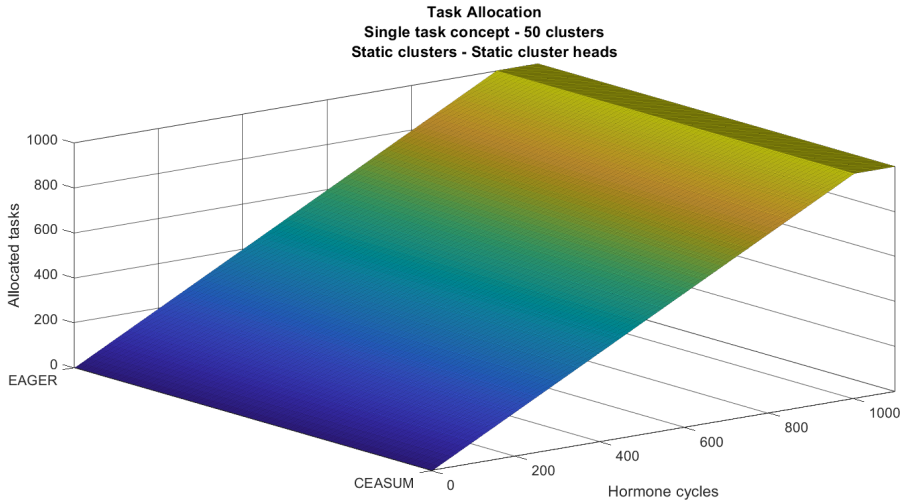


Figure 8.6: Allocated tasks over the time. Scenario with 50 clusters, single task concept, static clusters and static cluster heads.

the election (see figure 8.7). This is confirmed by the evaluation of the variant with the *EAGER* method. This combination takes exactly 1004 cycles to finish *self-configuration*. For the other method the estimation does not hold, this variant needs 1052 hormone cycles. The reason is the recalculation period of the *CEASUM* method. The period was set to 50 hormone cycles, meaning every 50 cycles all PEs send their eager values to the cluster head. This happens for the first time in the first hormone cycle. In the first cycle, the cluster heads are not elected yet. Therefore, none of them has the necessary information to build *cluster eager values*. That is why, they will have to wait until the next recalculation period at hormone cycle 50. Afterward, in the 51st cycle, the allocation starts and needs 1001 cycles as estimated in chapter 6.

dynamic-static The *dynamic-static single task* HAHS shows for both methods exactly the same task allocation behavior as the *static-dynamic* variant (see figure 8.7). In the *static-dynamic* variant the *CEASUM* method does not start the allocation until the first recalculation period is reached. In contrast, the *dynamic-static* variant, could start the allocation right at the beginning. At this moment, the eager values are sent out for the first time. The waiting cycles until the first recalculation interval passed (in the scenario at hormone cycle number 50) would be avoided. However, letting the cluster heads calculate their

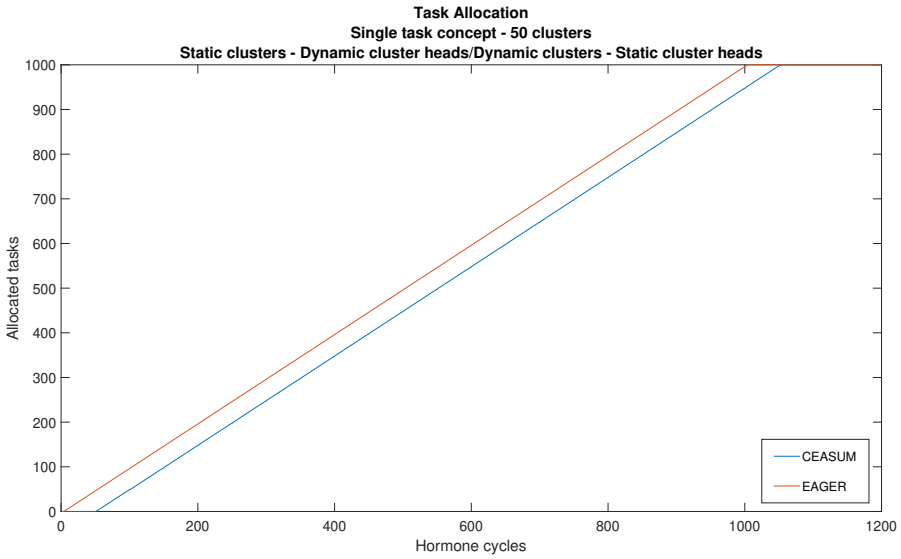


Figure 8.7: Allocated tasks over the time. Scenarios with 50 clusters, single task concept, static clusters and dynamic cluster heads respectively dynamic clusters and static cluster heads.

cluster eager values right at the start would result in misrepresented *cluster eager values* since the clusters are not built at this moment. The only sources available to the cluster heads at this moment are their own *eager values*. This can lead to non-optimal allocations among the clusters, which then have to be optimized as soon as all clusters are built and the next recalculation period is reached.

dynamic-dynamic The most flexible HAHS still has a similar task allocation behavior as the two variants before. To demonstrate the difference, the evaluation of the 100 cluster configuration was chosen (see figure 8.8).

It can be seen that both methods are not monotonically increasing the task allocations in the first 100 to 150 hormone cycles like before. Due to the sequential cluster head task allocation, it takes as many hormone cycles as clusters are in the system until all cluster heads are present. Simultaneously, those cluster heads which already exist start to build their cluster. These two mechanisms combined result in a constantly changing cluster environment (clusters tend to start big and become smaller with time). These constantly changing clusters influence the *cluster eager values* of their cluster heads. This

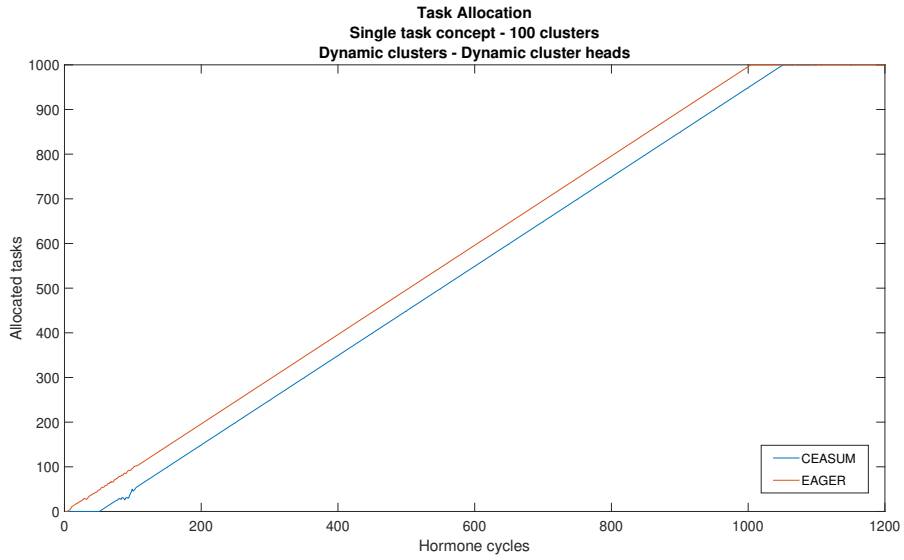


Figure 8.8: Allocated tasks over the time. Scenario with 100 clusters, single task concept, dynamic clusters and dynamic cluster heads.

means they start with high *cluster eager values*, allocating many tasks, then their cluster might shrink. Finally, they have to pass some of the tasks to other cluster heads.

Compared to the *organ task* concept, the *single task concept* takes longer until the *self-configuration* is finished. All scenarios needed at least 1001 hormone cycles which is also slightly worse than the worst-case allocation time of the AHS.

8.1.1.2 Communication load

The main reason for the development and research of the HAHS is the high communication load of the AHS. The sent hormones in each cycle of the simulations were counted in order to show that the communication load can be reduced using the HAHS. Additionally, the counted hormones were used to compare the different concepts and methods.

Organ task concept For the *organ task* concept the 32 cluster scenario was chosen for comparison with the AHS (see figure 8.9).

The figure shows that all four variants of the *organ task* concept perform much better than the AHS in terms of communication load. While the AHS

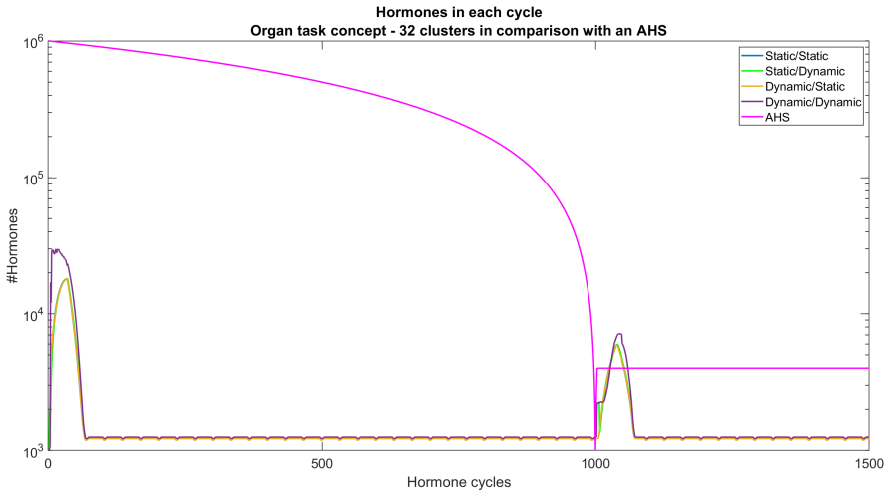


Figure 8.9: Hormones over the time. Scenario with 32 clusters, organ task concept compared with an AHS.

starts with sending 1 million hormones and then slowly commutes around 4000 hormones per cycle, none of the four HAHS variants exceeds 30000 hormones in any hormone cycle. After the self-configuration is done (around 100 hormone cycles), the HAHSs commute to a level of around 1189-1253 hormones per cycle, interrupted by periodic peaks up to 6000-7100 hormones per cycle. The peaks are produced by the self-optimization techniques implemented, trying to optimize the allocation every 1000 hormone cycles. The optimization is performed with the same period in the AHS, too. Since the AHS does not finish the *self-configuration* until hormone cycle 1000, it will immediately continue with the first *self-optimization* phase.

Figure 8.10 shows the start phase of the four different HAHSs in detail again. Noticeable in this figure is that the *static-static*, *static-dynamic* and the *dynamic-static organ task* HAHS have more or less the same communication load in this phase, while the *dynamic-dynamic* variant produces more load. The observation is reasonable since the *dynamic-dynamic organ task* HAHS coordinates its clusters and its cluster heads autonomously. The coordination needs additional communication. Furthermore, the first allocated cluster heads build big clusters since the cluster head tasks will be allocated sequentially. Those big clusters produce more load because the task allocation also already started and operates in parallel to the allocation of the cluster head task.

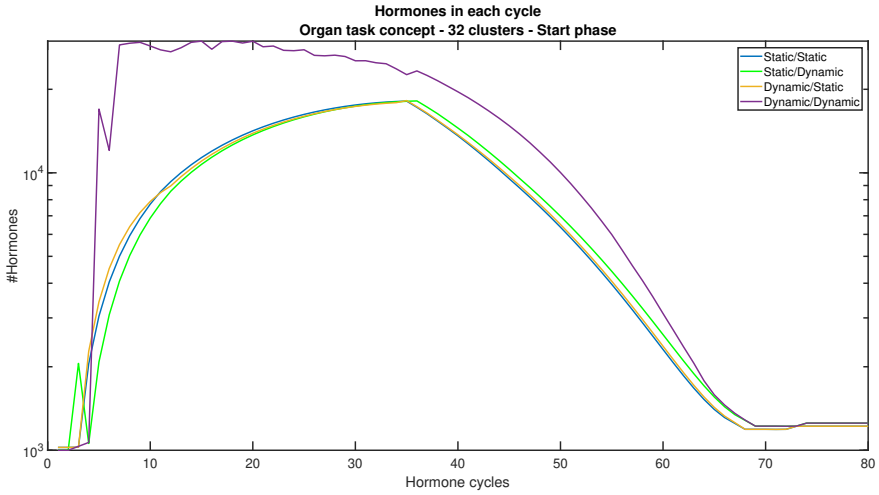


Figure 8.10: Hormones over the time during the start phase. Scenario with 32 clusters, organ task concept.

The big clusters will shrink gradually since new cluster heads spawn in their neighborhood and build their own cluster around them.

When examining the hormone number of the *dynamic-dynamic* variant with different cluster numbers (see figure 8.11), it can be seen that the different numbers of clusters have different impacts on the communication load.

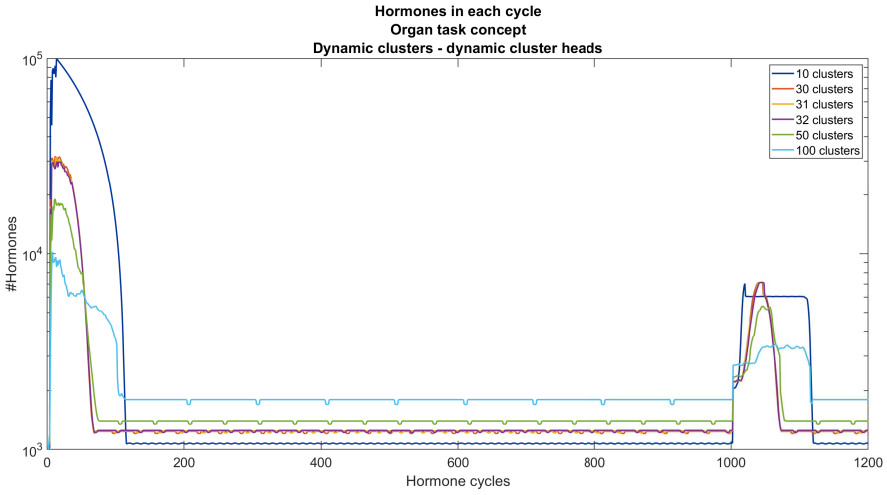


Figure 8.11: Hormones over the time. Scenario with the organ task concept, dynamic clusters and dynamic cluster heads.

Especially the duration and the communication load in the start phase differs. For example, the 10 and 100 cluster scenario have a longer high

load phase in comparison to the other configurations. The longer phase is induced by the longer task allocation time needed by both scenarios (see section 8.1.1.1). However, the 100 clusters configuration does not send out nearly as much hormones as the 10 clusters scenario and also never reaches the communication load of the other configurations. So, only regarding the maximum communication load per cycle, it seems to be better to choose a configuration with a high number of clusters.

For further examination, it has to be analyzed which kind of hormone has the highest impact on the overall communication load.

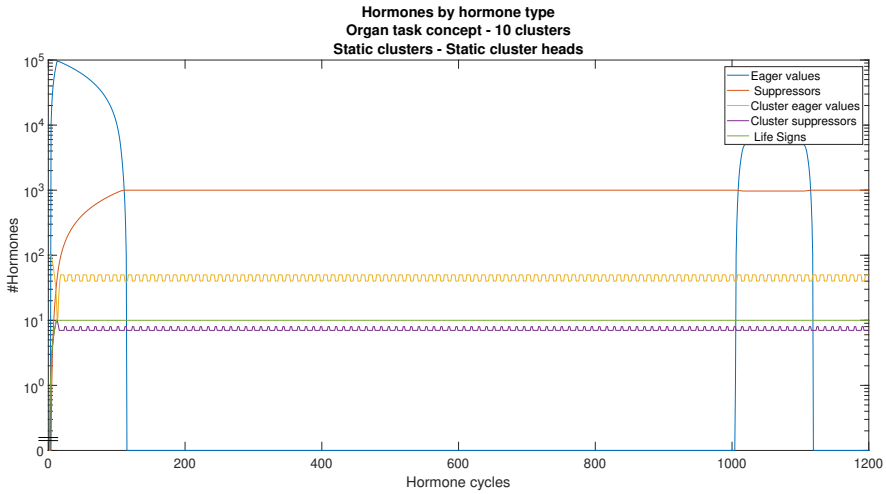


Figure 8.12: Hormones over the time, split by type. Scenario with the organ task concept, static clusters and static cluster heads and 10 clusters.

Figure 8.12 shows the number of hormones, separated by their type, for the *static-static* variant with a 10 cluster scenario. Apparently, the *eager values* have the highest impact on the total communication load. They are especially responsible for the maximum communication load per cycle in the system.

Figure 8.13 shows the same distribution for the 100 clusters scenario. This time, the *eager values* are not responsible for the high communication load, but the *organ eager values*. These are the *eager values* sent in the *inter-cluster* network for the organs.

Apparently, the cluster size (and hence the organ number) has effects on the number of *eager values* and on the number of *organ eager values*. In order to find the best organ number/cluster size to reduce the maximum communication

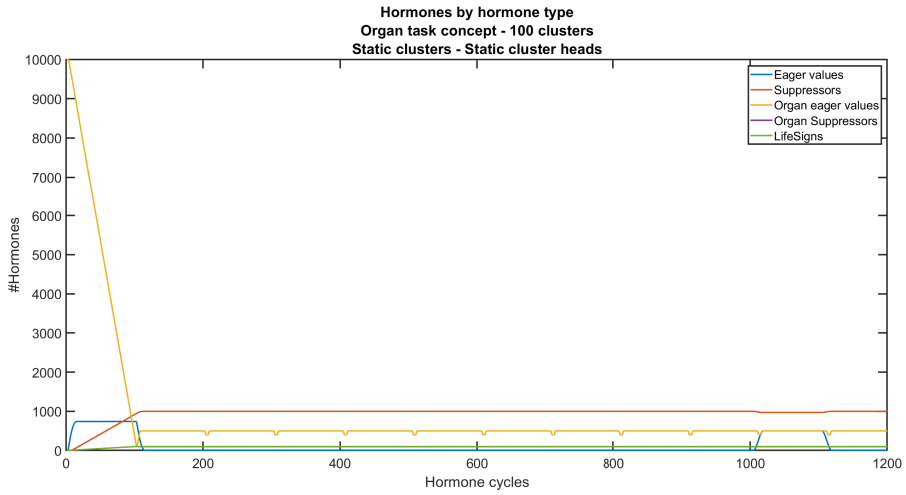


Figure 8.13: Hormones over the time, split by type. Scenario with the organ task concept, static clusters and static cluster heads and 100 clusters.

load per cycle, at first the maximum number of *organ eager* values and *eager values* have to be found. Since both numbers depend from the organ number and from the hormone cycle, equations for both numbers can be deducted. For the *organ eager values*, this equation is simple since it is related to the number of *eager values* in the AHS (see equation 3.4). The number can be calculated by multiplying the number of cluster heads (which corresponds to the number of clusters ($|C|$) and hence the number of organs ($|O|$)) with the number of not already allocated *organ tasks* (in every cycle t at least one organ task will be allocated, therefore: $\max(0, |O| - t)$). This results in equation 8.4.

$$|OEA(t)| = |C| \cdot \max(0, |O| - t) \tag{8.4}$$

The equation for the number of *eager values* turns out to be much more complex. The reason for that is the sequential unlocking of *organs* among the clusters and the following parallel allocation of the tasks in the organs. The equation 8.5 will be explained in the appendix (see section A.2).

$$\epsilon(t, k, l) = \gamma(t, k, l) + \delta(t, k, l) \tag{8.5}$$

Figure 8.14 shows the results of the equation for the first 200 hormone cycles and 1-100 organ numbers. The number is higher for small organ numbers and

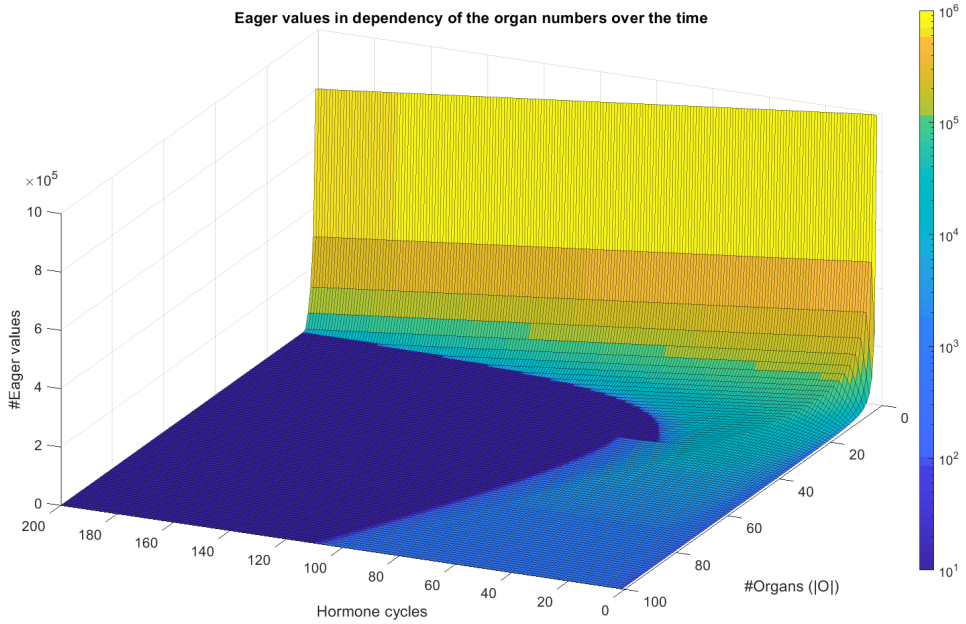


Figure 8.14: Eager value number in dependency of the organ number and the hormone cycles

falls rapidly when raising this number.

According to equation 8.4, small organ numbers produce less communication on the *inter-cluster* network, while more organs produce more *organ eager values*. Hence, adjusting the organ number of the system leads to opposing numbers of *organ eager values* and *eager values*. In order to find the right size to reduce the communication load, the maximum of the *organ eager values* and the cluster internal *eager values* for every organ number has to be determined and compared. For organ numbers of 1 - 100 this is depicted in figure 8.15. The least communication load is produced with an organ number of 58. This is the tradeoff value from where the *organ eager value* load starts to exceed the *eager value* load.

When designing an organ task HAHS, the communication load has to be considered but also the worst-case allocation time should be minimized (discussed in section 8.1.1.1).

Figure 8.16 shows the two properties, communication load and worst-case allocation time, in dependency of the organ numbers for the presented HAHS scenario. A pareto front can be extracted from the scatter plot in order to

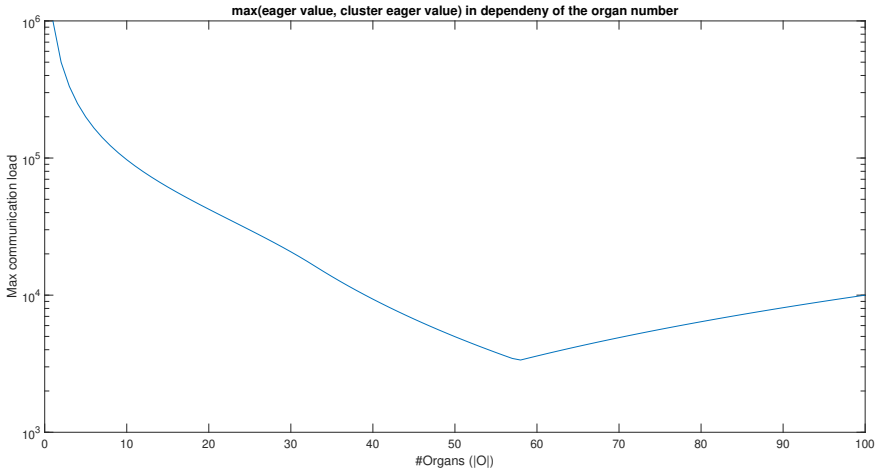


Figure 8.15: Maximum eager value number in dependency of the organ number

determine optimal organ numbers (see figure 8.17). The pareto front shows that organ numbers of 36 to 58 are optimal for an organ task HAHS with 1000 PEs and 1000 tasks.

Single task concept For the single task concept, the communication load compared to the AHS was the major evaluation goal. A 50 cluster scenario of the *single task* concept was chosen for that comparison (see figure 8.18). Furthermore, the focus was set on the two *cluster eager value* methods. For this reason, only the *static-static* variant will be evaluated here.

As expected the two HAHSs begin with fewer hormones per cycle than the AHS. The *EAGER* method will produce fewer hormones until the *self-configuration* is done, then a little more hormones will be sent out, due to the additional *inter-cluster* hormones and the *magnitude of eager values* hormones from the method. In contrast, the *CEASUM* method creates load peaks when recalculating its *cluster eager values*. At this moment all PEs send *eager values* for all tasks in the system to their cluster head. Those peaks reach the load level of the AHS at the beginning, but repeat periodically, while the AHS constantly reduces its load, until it reaches *self-configuration*. To decide which of those two systems is more stressful for a network, the communication loads have to be normalized to fit to the typical property of a network: the bandwidth. The bandwidth indicates how many data per second can be transmitted by a network. For this reason, the assumption was made that each hormone

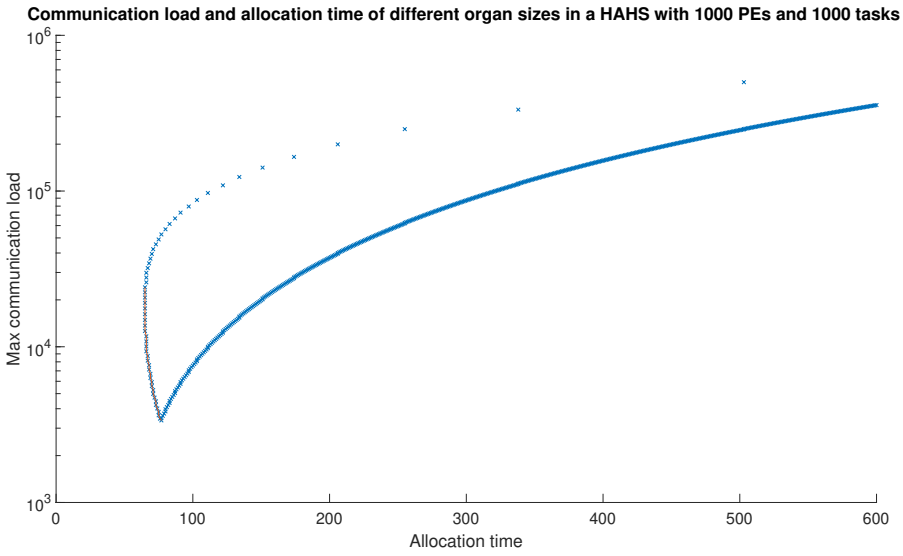


Figure 8.16: Pareto optima for an organ task HAHS with 1000 PEs and 1000 tasks

cycle needs exactly 100ms to finish. The resulting hormone rates (number of hormones send by the system in each second) are depicted in figure 8.19.

The results show that in terms of the network bandwidth, the HAHS performs better than the AHS, even with the communication rich *CEASUM* method.

8.1.1.3 Emergence

As mentioned in section 2.3, emergence is an often used measure when it comes to evaluate *organic computing systems*. It can also be used in the context of the HAHS to demonstrate the different behavior and influences of the concepts/methods. Assuming each PE (or cluster) is an attribute of the system, and the number of tasks allocated on a PE (or cluster) are the states of the attributes. The resulting emergence rate shows if the tasks are distributed equally or not among the PEs (or clusters). This can be used to evaluate different *cluster eager value* methods in the *single task* concept. The calculation of the emergence rate is explained in section 2.3.

Figure 8.20 shows the task per PE emergence rate over the time for a single task, (*dynamic clusters, dynamic cluster heads*, 100 clusters HAHS) with both *cluster eager value* methods. Both methods start with the maximal emergence the system can reach which is reasonable since both start with no tasks allocated

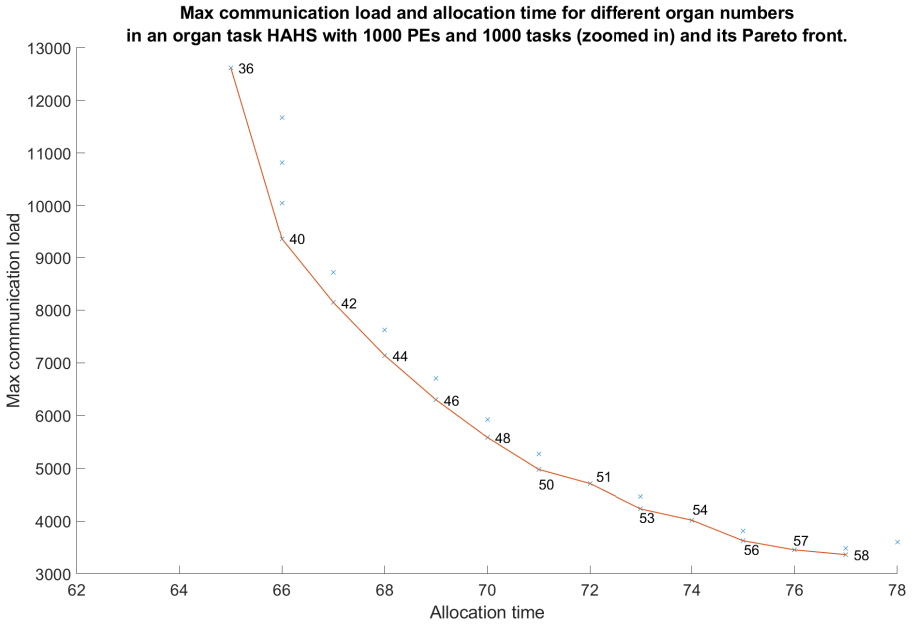


Figure 8.17: Pareto optima for an organ task HAHS with 1000 PEs and 1000 tasks (zoomed to the pareto front (red line))

at all. Therefore, all PEs are in the same state. The *CEASUM* stays on this level until the 50th hormone cycle has passed. Then, the recalculation takes effect for the first time, and the system starts with the allocation. In both methods the emergence rate falls rapidly as soon as the allocation starts. The fall is reasonable since the tasks get distributed sequentially to the PEs, such that some already start allocating, while others not have unlocked any tasks yet. Furthermore, since these are the results of the *dynamic-dynamic* run the clusters will build themselves sequentially, too. This might result in a PE already having tasks allocated then, changing the cluster and therefore losing its allocated tasks again. All this results in the PEs having many different states. Therefore, the overall emergence subsides. The difference between the methods is as follows: while the emergence of the *CEASUM* method rises again after around 550 hormone cycles, the emergence of the *EAGER* method nearly stays on the same level. This means that the *CEASUM* method reaches a higher level of structure in its system, an almost equal distribution among the PEs. In turn, the *EAGER* method has a more unequal distribution, meaning more PEs have different states.

For a complete evaluation, the emergence of the tasks per cluster is inves-

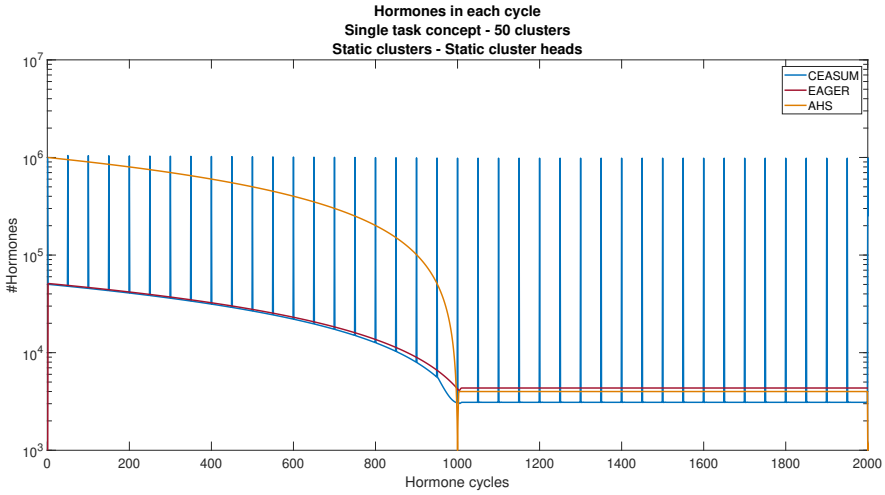


Figure 8.18: Hormones over the time. Scenario with 50 clusters, single task concept, static clusters and static cluster heads compared with the AHS.

tigated (see figure 8.21). It is interesting that this result shows a contrary emergence behavior for the two methods than before. The *CEASUM* method forfeits more emergence and stays on a lower level than the *EAGER* method.

As already mentioned before (see section 8.1.1.1), the *dynamic-dynamic* HAHS tends to create differently sized clusters, meaning that they are not having all the same number of PEs in their clusters. Therefore, a system trying to find an equal distribution of tasks among all PEs has to consider this when distributing the tasks to the different sized clusters. That is why the emergence of the *CEASUM* method for the clusters is low while the method reaches the maximal emergence for the PEs.

Concluding from these results: the *CEASUM* method depicts the cluster ability for tasks to the *inter-cluster* cycle better than the *EAGER* method. Of course, this has the price of a higher communication load (see section 8.1.1.2).

8.1.1.4 Cluster regulation

As mentioned in section 4.6.4.1, using the *organ task concept* in combination with a *dynamic cluster* and *dynamic cluster head* concept can cause the problem of under-allocation. Four different strategies to cope with this situation were presented and two of them were evaluated. For this reason, two additional configurations of an *organ task* HAHS with *dynamic clusters* and *dynamic*

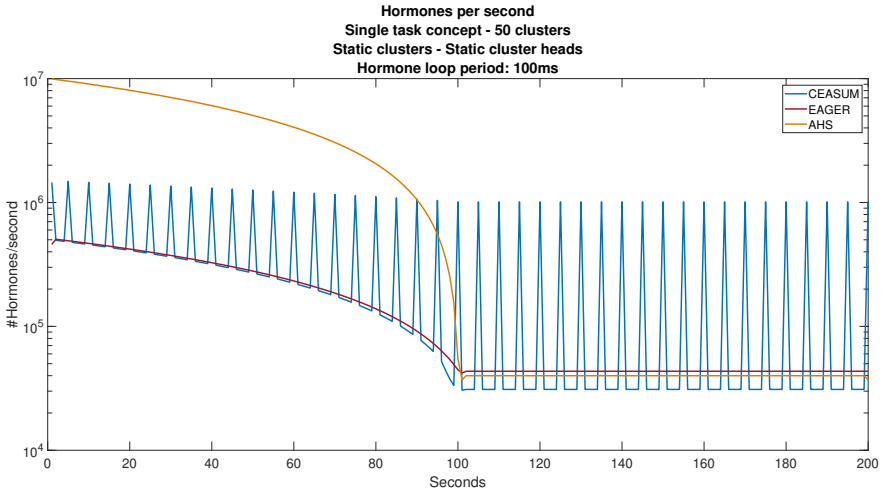


Figure 8.19: Hormones per second. Scenario with 50 clusters, single task concept, static clusters and static cluster heads compared with the AHS.

cluster heads were prepared. The first one consisted of PEs able to allocate only two tasks at maximum. The second one is even more restrictive, allowing only one allocated task per PE.

Figure 8.22 shows the allocated tasks in each hormone cycle of the first prepared configuration (the one in which two tasks per PE are allowed) with no cluster regulation, with PID controllers trying to regulate the *cluster accelerators*, and with the stress hormone method. With no cluster regulation, the problem becomes obvious: eight tasks will not be allocated by the system, even though the resources are available.

In both regulation strategies the cluster heads periodically check how many of their assigned tasks are allocated in their cluster. In case they experience a discrepancy, they act.

In the *PID controller* version, the cluster head amplifies its *cluster accelerator* by means of a PID controller. It takes a few cycles until a change in the number of allocated tasks is visible. Of course, this depends on the chosen P-,I- and D-coefficients. Around hormone cycle 1000, the number of allocated tasks climbs up for the first time. This means that a cluster which was too small, raised its cluster accelerator in order to to gain more PEs for itself. In the next cycle this happens again: the cluster which amplified its *cluster eager value* gained PEs from one or several clusters in the neighborhood. However, those

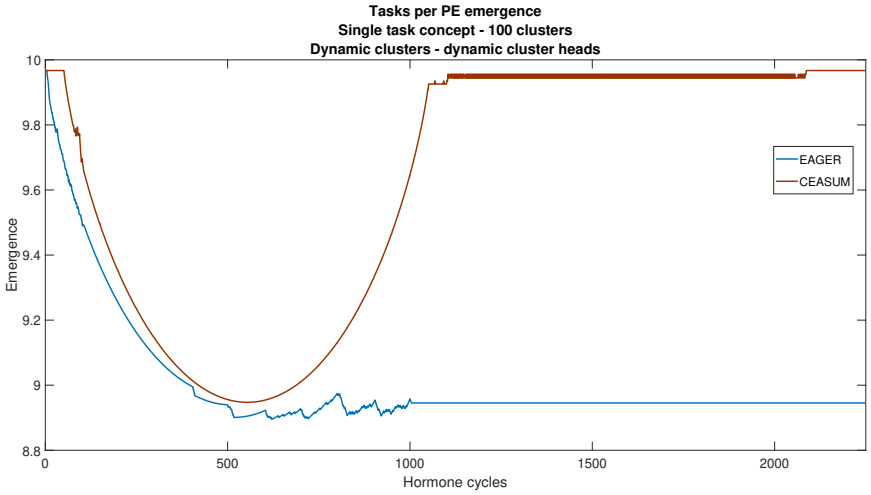


Figure 8.20: Emergence of the task allocation among the PEs in a 100 clusters, single task, dynamic cluster, dynamic cluster head HAHS.

neighbored clusters now lack enough PEs in order to fully allocate their organ. That is why, a short drop is visible in the graph. After several further cycles each cluster finally reached a size which is sufficient to fully allocate the organ.

In the *stress hormone* version, the cluster head which discovered a discrepancy immediately broadcasts a stress hormone. Those PEs which have not allocated any task yet, will immediately amplify the received cluster accelerator of the stress hormone sender. Therefore, they assign themselves to the next cluster head which sends a stress hormone. This method is fast, but leads to unconnected, fragmented clusters.

In the second configuration each PE can only allocate one PE a maximum. This aggravates the problem.

In this configuration, the unregulated version reaches only an allocated task number of around 890, this is worse than before. Still, the *stress hormone* version manages to fix the under-allocation in the first period. The impact of the *PID controller* version is completely different. Apparently, the number of allocated tasks oscillates around 890. It seems that the clusters with too less PEs raise their cluster accelerator and take PEs from neighboring clusters which needed those PEs. This is similar to before, but this time no stable state between all clusters is found. The cluster heads constantly raise their *cluster accelerator* as a reaction to the raise of neighboring clusters. Therefore,

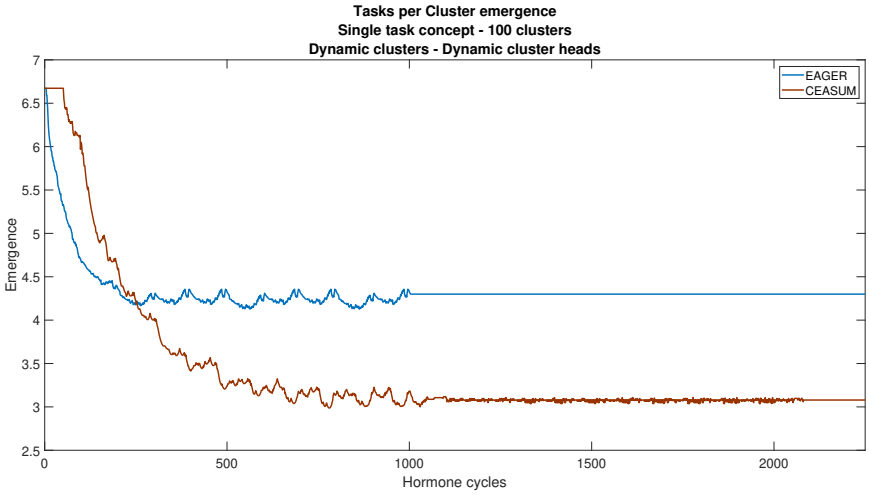


Figure 8.21: Emergence of the task allocation among the Clusters in a 100 clusters, single task, dynamic cluster, dynamic cluster head HAHS.

this strategy is not adequate in this situation, at least not with the evaluated coefficients for the PID controller.

8.1.2 Self-healing

Another important *self-x property* in an *organic computing* system is *self-healing*. The *self-healing* enables the system to regain the functionality in case some of its parts fail. The *self-healing* in the AHS compensates PE failures by migrating the tasks from the failed PEs to PEs still running. The *self-healing* in the HAHS depends heavily on the chosen concepts and methods. Furthermore, the HAHS has to differ between failures of cluster heads and failures of cluster children. Especially the second case is complicated if the tasks of the failed PE cannot be compensated cluster-internally. The failure then has to be depicted to the *inter-cluster* cycle. In order to evaluate this scenario, a special simulation configuration of the *static-static* HAHS with 10 clusters was prepared. In this configuration, 76 PEs of the first cluster fail at hormone cycle 2000. The remaining PEs are not able to compensate this loss by allocating all the lost tasks. Therefore, the lost tasks have to migrate to other clusters.

The *organ task* HAHS does not provide a mechanism for depiction of the *intra-cluster* cycles to the *inter-cluster* cycles. Therefore, the number of allocated tasks will sink at hormone cycle 2000 (see blue plot in figure 8.24), then rise

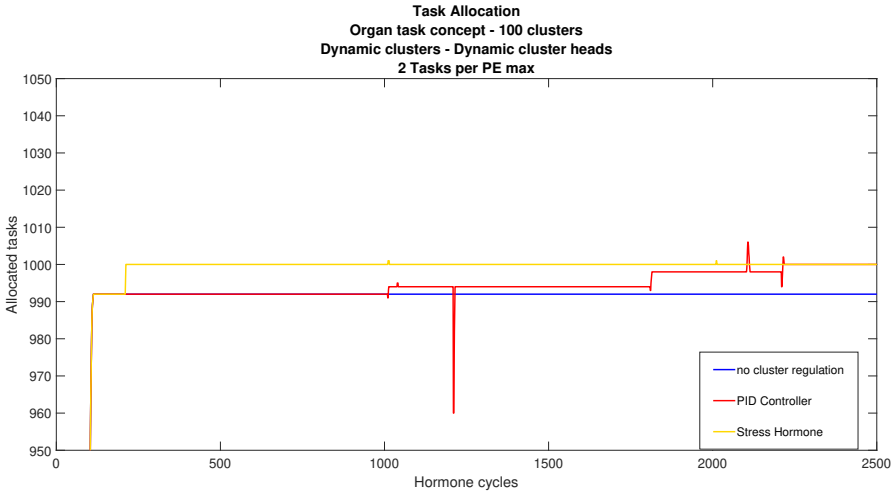


Figure 8.22: Allocated tasks of an organ task, dynamic-dynamic HAHS in which each PE can allocate up to two tasks

again because some tasks will be compensated cluster-internally, but never reach the full functionality (1000 allocated tasks) anymore. In contrast, the same configuration as *single task* concept with the *CEASUM* method (yellow plot in figure 8.24) recovers the same number of tasks cluster-internally and then migrates the remaining lost tasks to other clusters as soon as those tasks will be offered in the *inter-cluster* cycle.

8.2 RAHS

The RAHS is a further development of the HAHS with more than one possible hierarchical level. The evaluation of the RAHS also concentrates on the task allocation and the communication load. The evaluation of two different approaches of the RAHS is presented in this section. The evaluations of the first concept (see section 8.2.1) were already published in [39].

8.2.1 Max eager value approach

The first approach examined is the *max eager value* approach (see section 5.3.2). In this approach initial eager values in the upper levels will be generated by the maximum eager values of the PEs in the underlying clusters. Therefore, it takes some additional hormone cycles at the beginning until the first allocation

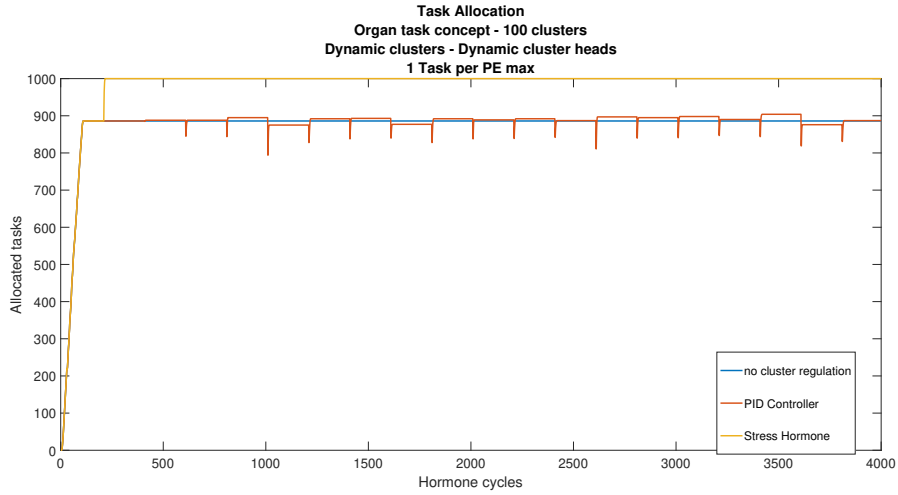


Figure 8.23: Allocated tasks of an organ task, dynamic-dynamic HAHS in which each PE can allocate up to one task

in the uppermost level starts. For the evaluation, scenarios with 1000 PEs and 1000 tasks were prepared. The scenarios differ in the number of levels. All scenarios have a binary-tree topology.

8.2.1.1 Timing behavior

At first, the timing behavior of the RAHS in terms of the task allocation is evaluated.

Figure 8.25 shows the task allocations of eight different RAHSs. The configurations differ in their number of levels (2 to 9 levels), while everything else is the same. As expected, the time for the *self-configuration* rises with the number of levels. This is not surprising since this approach requires that all tasks have to be passed to the last level before they are allocated. Therefore, a raise in the number of levels leads to a raise of the *self-configuration* time.

8.2.1.2 Communication load

For evaluating the communication load, the topology of the RAHS has to be considered. Since the RAHS will be likely implemented on a hierarchical topology, measuring the communication of the complete system would not make sense. Therefore, the communication load of each PE in the RAHS is measured at each hormone cycle. For comparison, the highest communication

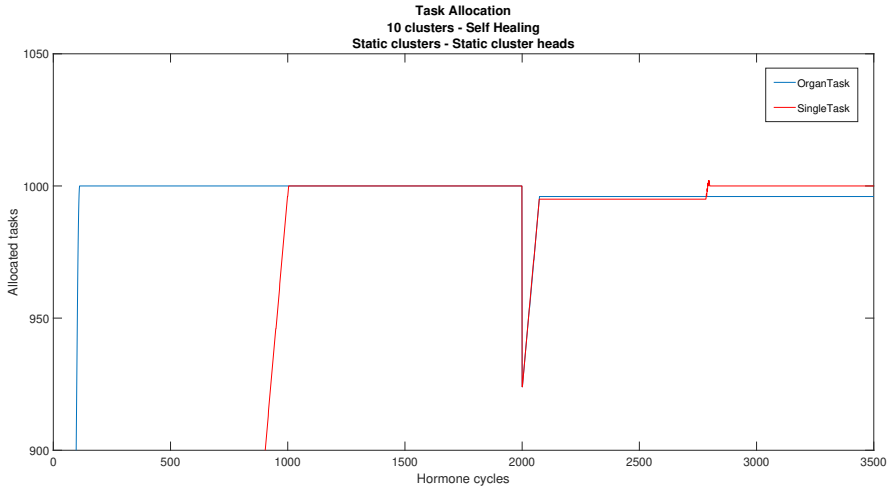


Figure 8.24: Allocated tasks of a 10 clusters, static cluster and static cluster head HAHS. 76 PEs of cluster 1 fail at hormone cycle 2000.

load per PE in each hormone cycle is chosen.

Figure 8.26 shows the results from the different configurations. The results are rather surprising. One would expect that with a raise in the number of levels the maximum communication load falls, because even less PEs apply for the same task at the same time. Actually, this effect is counteractivated by the fact that each non-leaf cluster consists of exactly two PEs in a binary tree topology and each PE can decide on exactly one task at each hormone cycle. The consequence is that in the underlying clusters at most one task is activated for application at one hormone cycle. This means a reduction of the leaf cluster size does not reduce the communication load for the PE with the highest communication load. On the contrary, the additional vertical communication and the higher chances for passes which have to be revoked lead to a higher communication load.

8.2.2 Least eager value approach

The second approach removes the restrictions of the *max eager value* approach. On the one hand, the *least eager value* approach supports task allocations on PEs not belonging to the lowest level. On the other hand, a *self-healing* mechanism exists in this approach. For the evaluation a scenario of a non-binary tree with 3 levels was created and simulated.

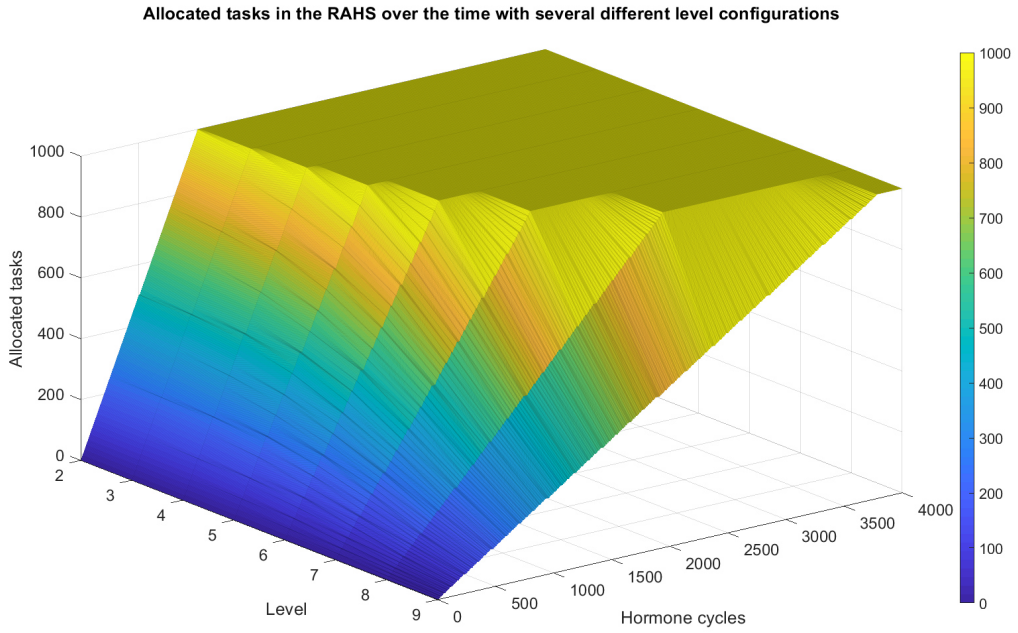


Figure 8.25: Allocated tasks in the RAHS with different level configurations

8.2.2.1 Self-configuration

At first the *self-configuration* of this approach is investigated. The timing behavior and the communication load will be compared to the *max eager value* RAHS and the AHS.

Figure 8.27 shows the allocated tasks of all three systems on a logarithmic time scale. It can be observed that the *least eager value* RAHS is the system with the longest allocation time, while the other two need similar time to finish the *self-configuration*. Due to the fact that the *least eager value* approach is also able to allocate tasks in upper levels, its task allocation is faster at the beginning. The long period of no allocation in the approach is caused by the withdrawing of tasks from lower levels to upper levels (see section 5.3.3).

Additional to the different allocation times, the three systems also differ in their communication load (see figure 8.28). The evaluation shows that both approaches for the RAHS have a lower maximum communication load than the AHS. The second observation from the figure is that the *max eager value* approach has a nearly stable communication load while the *least eager value* approach produces more communication in the beginning. Eventually, its

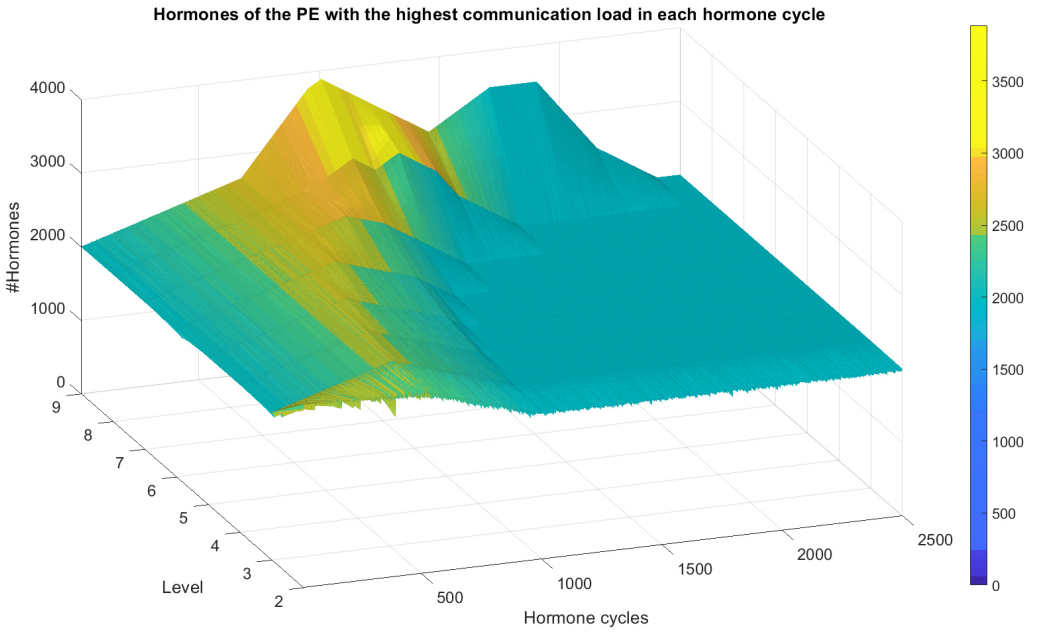


Figure 8.26: Communication load in the RAHS with different level configurations

communication load falls under the communication load of the *max eager value* approach and nearly reaches the amount of communication from the AHS. The reason for that is the ability of the *least eager value* approach to allocate tasks in upper levels, which reduces vertical and horizontal hormones in lower levels.

8.2.2.2 Self-healing

The *least eager value* approach enhances the RAHS with the *self-healing* property. For evaluating the property three different configurations for the system were prepared.

In figure 8.29 the number of tasks over time for the three configurations of this scenario are visible. In the first scenario, at hormone cycle 3500, an arbitrary PE of the topmost level was shut down and recovered at hormone cycle 7000. In the second scenario an arbitrary PE of the middle level was chosen to fail. Finally, in the third scenario, one of the lowest level PEs was chosen. The effect on the task distribution can be seen marked by the two red circles. The smaller circle shows the effect after the shutdown: a small fall in the number of tasks is observable, but the system quickly regains full

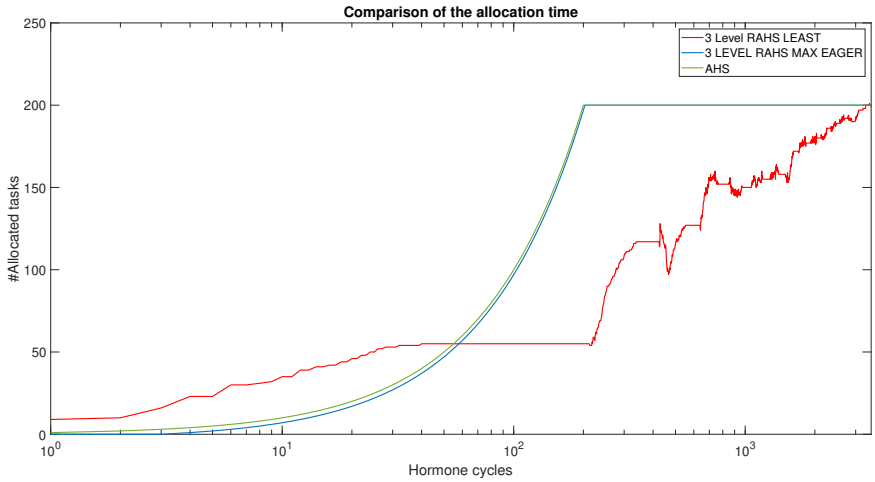


Figure 8.27: Allocation time of the least eager value RAHS in comparison to the Max eager value RAHS and the AHS

functionality. In contrast to that, the effect after recovering the failed PE is more severe. This is observable in the large circle, which shows a huge fall in the number of allocated tasks and a long time until all tasks were allocated again. This is explainable through the recovering of the PE. Afterward, it will win a lot of tasks in the first level and will pass many of them down to its underlying cluster. There, the tasks might get allocated or passed again.

The difference between the three configurations is the deflection in the allocated tasks, which is much lower when the failing PE is in a lower level. A failing PE in the lowest level has hardly any effect on the allocated tasks. This result is reasonable since the failure of a PE in the higher levels entails the migration of tasks from a whole sub-tree, while in the lowest level only the tasks from the PE in question have to be migrated.

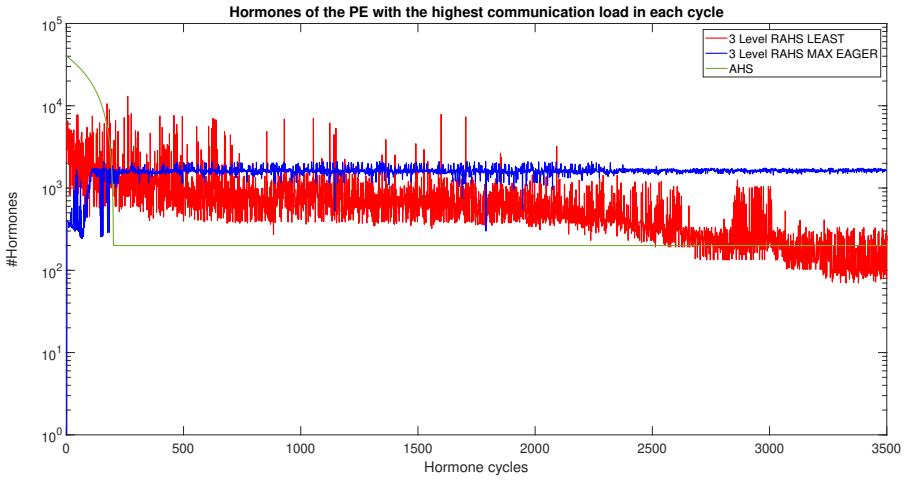


Figure 8.28: Communication load of the least eager value RAHS in comparison to the Max eager value RAHS and the AHS

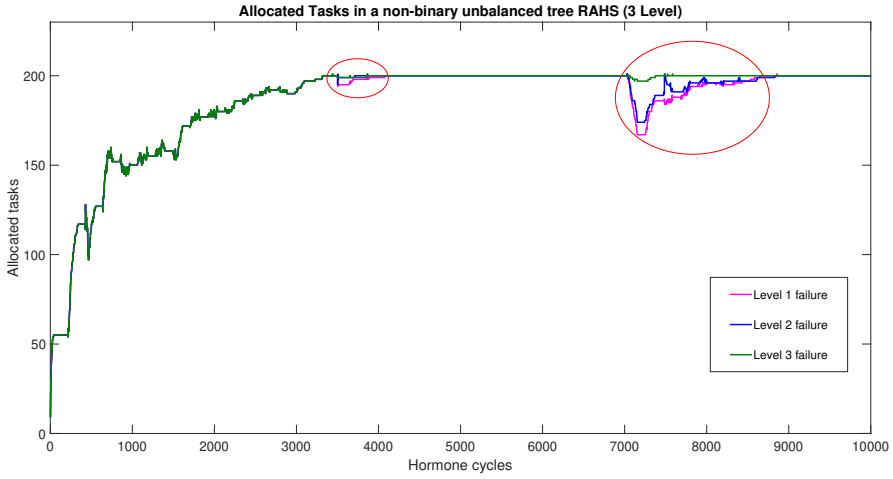


Figure 8.29: Self-healing of the RAHS demonstrated by the number of allocated tasks. PE failure at hormone cycle 3500, recover at 7000.

Chapter 9

Comparison and applications

This chapter presents other systems and concepts for a distributed task allocation, as well as applications for those. Additionally, some of them will be compared to the HAHS respectively RAHS.

9.1 Off-line methods

The problem of mapping tasks to a multicore system is a well known problem in the scientific community. Often, the task mapping is solved off-line at design-time of the system. The resulting task mapping is static and will not change during run-time, unless several configurations were prepared.

In [33] and [19] the problem of mapping tasks of applications to multiple processing nodes is solved off-line. The tasks of an application are structured as a directed graph. This means that some tasks have preceding tasks from which they depend. Additionally, tasks can be recurring. In interaction with real-time requirements for the applications/tasks, finding an optimal mapping becomes a NP-complete problem[28]. The authors solved the mapping by formulating it as a Satisfiability Modulo Theory (SMT) problem. Afterward, they used Z3, a SMT solver, to find the mapping. Those two methods differ to the presented HAHS and RAHS by not considering hierarchical topologies and heterogeneous processing nodes.

[4] deals with the task mapping problem for heterogeneous processing nodes. Here, different kinds of processing nodes may have different execution times for the same task. The author states that finding an optimal mapping is

NP-hard and proved this by transforming the 3-Partition problem to the presented problem. Moreover, the author formulated the problem to an Integer Linear Program (ILP) and used an approximation algorithm for calculating the solution.

The authors from [36] and [66] used genetic respectively evolutionary algorithms in order to find an optimal task mapping for heterogeneous processing nodes. While [36] tries to minimize the overall delay of the system, [66] focuses on the reduction of resource consumption by the system.

Finally, heuristics are used to solve the task mapping problem, too. [8] presents eleven different heuristics and compares them to each other. The authors show that genetic algorithms perform best in comparison to all other heuristics.

In contrast to all those off-line methods, the AHS, HAHS and RAHS are on-line, dynamic methods which solve the task-mapping at run-time. The advantage of dynamic methods is their flexibility towards unforeseen system states. If the system enters a state which was not covered at design-time, an off-line method cannot react with a suitable task mapping. An on-line method in the same situation will react towards the changed state with a new task mapping. The advantage of off-line methods is their possibility to find the optimal task mapping for the given states. This is due to the fact that off-line methods have enough time to calculate the optimal solution at design-time while on-line methods need to deliver a working solution fast at run-time. Therefore, a comparison between the presented HAHS/RAHS and off-line methods in terms of allocation time and communication load is not reasonable.

9.2 On-line methods

In contrast to the off-line methods, dynamic on-line methods have only limited time to generate the task mapping. Therefore, those task mappings tend to be the sub-optimal mappings for the given state of the system. In the following subsections, several different on-line methods are presented, from which all of them focus on different aspects of the problem or have a different approach in solving the problem.

9.2.1 Multi-Agent Systems

When it comes to dynamic distributed task-allocation, researchers often use the same principle of designing the allocating entities (e.g. PEs or robots) as independent and autonomous agents. This results in a so-called Multi-Agent System (MAS). A MAS is based on multiple agents which act autonomously based on rules with their environment [64]. MASs can be used to solve many problems, in which a centralized solution is not applicable. In this chapter, only MASs solving the problem of an autonomous task allocation onto entities will be regarded. MASs solving this problem are involved in many applications, like the coordination of tasks between independent robots or the distribution of resources to workers. In the following sections, some MAS and their core functionality will be presented.

9.2.1.1 The contract net protocol

A widely used method to implement the task-allocation is using an auction-based negotiation between the agents, called the Contract Net Protocol (CNP) [57]. The CNP works auction-based, meaning that two kinds of agents exist in the system, the *managers* announcing tasks/jobs and the *contractors* bidding on the announced tasks. The *contractor* with the best bid on a task/job receives it for execution and will send the result to the *manager*. A *contractor* receiving a task/job can also become a *manager* and announce the task again. The method to determine *managers*, *contractors* and which *manager* announces which task, is heavily dependent from the exact implementation of the CNP.

In [7] the authors implemented an adapted CNP in their protocol for multi-robot cooperation. This protocol (called the M+ protocol) is composed of three components responsible for different system properties. In one of those, the CNP is used to distribute the incoming tasks to the robots in the system (the tasks are generated by a higher level, called the *mission layer*, which is centralized). The other two components care about the fail-safety of the agent and the task execution and synchronization of the task states. In contrast to the two presented layered AHSs in this thesis, this system does not implement a hierarchy. In case of many tasks (m tasks) spawning at the same time, all n

agents in the system bid on those tasks[1]. This results in $n \cdot m$ messages for the *self-configuration*, which is equivalent to the message load of the original AHS. A difference to the AHS is that all tasks are allocated simultaneously and no *self-optimization* mechanism is implemented. This may result in a sub-optimal task-allocation in which powerful agents overbid all other agents, but are not able to execute all tasks they won.

9.2.1.2 Consensus-based Algorithms

Similar to the CNP algorithms, the *consensus-based* algorithms use the market principle of bidding for tasks. In [21] such a *consensus-based* algorithm is used to distribute tasks among a group of healthcare robots. The specialty about the presented algorithm is that it starts the execution of tasks before all tasks are allocated. This is why it is called *consensus-based parallel auction and execution*. As an agent/robot advances with the progression of its current task, it places higher bids for the not allocated tasks. In this way a certain load balancing is implemented implicitly. Furthermore, in comparison to the CNPs, the *consensus-based* algorithms do not differ between two different roles. Every agent/robot has a list of still unallocated tasks on which it bids. This concept resembles the AHS, even though neither a *self-optimization* nor a *self-healing* are implemented. In comparison to the HAHS and the RAHS, this concept does not work with hierarchy. Additionally, it is worse in terms of communication load which is as high as the communication load of the AHS.

9.2.1.3 Hierarchical topology

In [5] the authors also used a hierarchical approach for MASs. Likewise to the *cluster heads* concept of the HAHS and the *upper representatives* concept in the RAHS, the presented architecture sorts the agents in two groups. It consists of *specialists* and *supervisors*. While the *specialists* are ordered into groups, the *supervisors* are responsible for the coordination among the *specialists*. For the coordination among the *supervisors*, a special *global supervisor* is necessary. This is equal to a central control unit and therefore not completely decentralized like the HAHS or RAHS.

[1] One agent does not bid on a task since it is the manager of the task. Instead it sends an announce message to all agents

The authors of [1] also use a hierarchical approach, called *ADAM*, in order to implement a dynamic task mapping. The tiles of a *network on chip (NoC)* will be ordered into virtual clusters. Similar to the *dynamic clusters* HAHS, the virtual clusters are not predefined and not static. Each cluster receives a special lightweight task, called the *cluster agent (CA)*. The CAs are responsible for the task mapping inside their cluster. Additionally, there exists a *global agent (GA)* for the entire *NoC*. This GA, which is also a lightweight task, is responsible to map applications and their tasks to the virtual clusters. In case there exists no suitable virtual cluster for an application it creates one or reorganizes the existing ones. Decisions by the *global agent* are based on the energy consumption and requirement of the PE respectively application. The decisions in the cluster depend on a cost function based on spatial, communication and resource requirements. This proposed approach is very similar to the HAHS. The difference is the implementation of managers. While the HAHS implements two control loop levels in which every participant is equal, this approach adds one special manager agent to each level.

9.2.1.4 Coalitions

Similar to the clusters in the HAHS and RAHS, [54] and [18] use coalitions of agents to form groups which work together on a set of tasks/jobs. Determining these coalitions autonomously is a key feature of both researches. In [54] the authors use algorithms for the Set Partitioning Problem (SPP) and Set Covering Problem (SCP) in order to form optimal coalitions of agents. Similar to the HAHS, a value for each coalition and each task is calculated. This *coalitional value* is the joint utility of all members of the coalition. Likewise to the *recalculation of cluster eager values* method (see section 4.7.1), those *coalitional values* will be recalculated, too. This becomes necessary since the utility values vary over the time and over the progressing allocation. Despite having coalitions of agents, this approach is not really a hierarchical architecture. All agents operate collectively on the same level to find the coalitions and to allocate the tasks.

In contrast, the authors of [18] used a coalition principle with a hierarchical aspect. In this work several robots with different spatial distances and different resources should form coalitions for the task allocation. Robots possessing key

resources will be selected as so-called *leader* of a coalition, while the others will be named *followers*. This reminds of the *dynamic clusters* and *static cluster head* concept of the HAHS. A big difference between the HAHS and the concept in [18] is the decentralized aspect. While the HAHS finds the clusters in an autonomous and decentralized manner, the presented concept needs a central server organizing the coalition formation.

9.2.1.5 Probabilistic method

The authors of [22] used also a biological example to solve the task allocation problem among a group of robots. The model used, which is called *response threshold model*, defines for every task in every robot a certain threshold. The threshold is used to calculate a probability of a response to a stimulus. The stimulus of a task can be a quantitative value of the task. This model is inspired by the reaction of bees to an experiment consisting of two food sources with different sugar concentrations. The authors then use a Linear Reward-Inaction algorithm [53] to let the robots learn their thresholds. This is a stochastic reinforcement algorithm combined in a learning automaton. The obvious difference to the presented HAHS and RAHS is the usage of probabilities, which impedes the estimations of worst-case allocation time, worst-case communication load and the allocation at all. Furthermore, no hierarchical separation was considered in order to lower the communication produced.

9.2.1.6 Tree-structure topology

The RAHS distributes the tasks of the system over a tree of connected clusters consisting of PEs. The idea of utilizing a tree topology is also used in [17], which adapted the idea from [35]. The authors use so-called *mobile agents* in a grid computing environment [6]. Those agents are able to clone themselves to other workstations in the grid and work on sub-tasks on the clone. The clone is able to clone itself again to another workstation in order to divide the task subset in even smaller subsets. The distribution of the sub-tasks is dependent from the bandwidth between the nodes. In contrast to [35], the tree topology is not static, but dynamic and can change during the execution of the tasks. A node in the second hierarchical level can be pulled up to the first

level due to its good performance in terms of its bandwidth. This goes beyond the current status of the RAHS and mixes it with the idea of *dynamic clusters* of the HAHS.

9.2.2 Heuristics

Not only multi-agent systems are used as an on-line solution. In [23] the authors used an *integer linear program* to find the initial task mapping for a *network-on-chip (NoC)*. Additionally to this the static, off-line part, they implemented several heuristics in order to react to failures in the *NoC*. One of those on-line self-healing mechanisms relies on the center of gravity method, which calculates the new coordinates on the *NoC* for a task depending on its communication with other tasks. The other heuristics are based on [31], which proposes three scheduling heuristics for nonidentical multiprocessors (NMS). The evaluation of all heuristics showed that a combination of one of the NMS heuristics and the center of gravity performed best. While this work provides an on-line method only for the *self-healing* the HAHS and RAHS use on-line techniques for the *self-configuration*, *self-optimization* and *self-healing*.

In [16] several heuristics for dynamic task mapping are compared. The focus of the evaluated heuristics lies on the mitigation of congestion in a *NoC*. All heuristics were implemented by a centralized manager processor. The authors showed that the *Path Load* heuristic performed best. The *Path Load* heuristic tries to either reduce the maximum channel occupation or the overall average channel occupation in the system. Additionally, *Path Load* considers only the communication path of the task to map, which means that it includes the position of communicating tasks in its decision.

9.2.3 Centralized solutions

An obvious solution for designing an on-line task mapping algorithm is to establish an omniscient manager. This manager will be in charge for mapping the tasks to the PEs at any time. Examples for this are given in [32] and [55]. The authors implement a special *manager processor* which decides the task mapping based on heuristic consisting of the manhattan distance and the nearest neighbor property.

In [63] the authors also use a *global manager* in a *NoC* in order to minimize the overall cost of all tasks. The authors define the cost of a task mapped to a PE by means of the sum of the worst-case start time, a distance metric and a neighborhood metric.

9.2.4 Decentralized solutions

An interesting solution to a related problem is presented in [62]. Here, the authors developed a *self-embedding*, decentralized algorithm for mapping the tasks of an application to a set of PEs. The application consists of a directed acyclic graph of tasks. The root task will be placed by a *seed point selection*, which can be for example the center region of a cluster build by *k-means*. Afterward, the mapped task will start mapping its successor tasks to the most suitable neighbor in a 1-hop region. Each mapped task will also continue doing this with its successors, which ends in a decentralized and parallel task mapping of the application.

9.3 Application examples

An important question for the research in the field of distributed, autonomous and hierarchical task allocation systems is: what are the applications for those? Some application examples were already presented above, like the multi-robot system for healthcare facilities [21] or the bandwidth-centric tree topology for grid computing [17] (which in turn can be used for many large applications needing a high number of computational resources, like the SETI@Home project [2]).

Another application example is a network of distributed sensors, which can either be all in the same level or ordered to a hierarchical topology. The latter is used in [34], which considers a network built from several Meteorological Command and Control (MCC) systems distributed spatially. Also, many *NetRad* radar systems were distributed in the network. Those are short-range, adaptive radars for weather detection in low elevations. These radars are clustered based on the location. In each cluster exactly one MCC will be present. The MCC correlates with the *cluster head* of the HAHS. The MCCs

will negotiate over the tasks in order to scan the space completely without redundant scanning.

Also interesting is the work in [24] in which a hierarchical MAS is implemented for on-chip systems. The hierarchical MAS is responsible for the thermal management of the chip. Therefore, the MAS will reactively and proactively migrate tasks to different areas of the chip in order to avoid chip failures due to overheating. Similar to the research of AHS, HAHS and RAHS this work also considers the real-time aspects of its proposed model, which most of the other MAS works presented in this chapter did not. The difference to the AHS and its advanced developments is the focus on only one chip. It is, therefore, application-specific. In contrast, the systems presented in this thesis can be adapted to many different application environments.

Another conceivable application, besides the already mentioned, is a network of wireless connected clusters, which also can be spatial mobile. Experiments with the HAHS middleware and a network of three clusters consisting of three *Raspberry Pis 3* were conducted in [41]. Besides some minor problems with the lossy wireless communication, the experiment shows that the system works in this application scenario.

Chapter 10

Conclusion

In this thesis, two main concepts for distributed and autonomous task allocation in large scaled, distributed many-core architectures are presented. Two ongoing developments in the field of computer science are presented in chapter 1: The growing integration density of integrated circuits and the Internet of Things. Both developments lead to many connected processing cores that are working together in one network. Chapter 3 presents the AHS, a general solution for autonomous task allocation that is based on bio-inspired control loops. The AHS was developed in the context of the *organic computing* research, which defines the *self-x* properties. The properties are briefly described in chapter 2.

It is difficult to distribute tasks autonomously and decentralized in a large scaled network. For smaller scaled architectures the AHS performs well in terms of the *self-x* properties. However, for larger scales the produced communication congests the network for payload traffic.

For this reason this thesis presents two solutions, namely the HAHS and the RAHS. Both rely on a hierarchy to cope with the communication problem. However, both are different in their application use case.

The HAHS implements a two-level hierarchy and organizes the PEs of the system in separated clusters, each containing one special PE, the *cluster head*. The system's entire task set is split into several subsets, either by the system designer (*organ task concept*) or during run-time by the *cluster heads* themselves (*single task concept*). The amount of sent hormones can be lowered drastically by only letting a fraction of the available PEs decide on the entire task set. Especially, during the system start the communication load is lower than in

the AHS. The PEs in the separated clusters only negotiate on the task subset distributed to their cluster. Besides the two already mentioned task concept variants, several other properties of the HAHS can be chosen: The HAHS can be instructed to choose autonomously the *cluster heads* by itself (*dynamic cluster heads*) and even the cluster membership of the PEs in the system (*dynamic clusters*). Furthermore, in case the single task concept is chosen, the *cluster heads* need a method to determine how suitable the cluster is for a certain task. All these methods have in common that they do not touch the system's autonomy and the decentralization. The task allocation results as well as the entire behavior of the system vary considerably under the different concepts and methods. A more flexible HAHS, meaning that the system has more decision space, produces more communication and the task allocation takes longer to complete. In contrast, a more static system does not adapt well to changes in the environment, for example when encountering PE failures. Chapter 4 explains the HAHS in detail and its properties are shown in chapter 6.

Contrarily to the HAHS, the RAHS is not limited to two hierarchical level. It is designed to work with n ($n \in \mathbb{N}_{>0}$) levels of PEs clusters. It is assumed that the underlying topology is static and known to the system. Therefore, the focus in its development laid on methods of determining the suitability of the underlying PEs for tasks. These methods should favor a good and balanced task allocation. Two potential methods for solving this problem are presented in chapter 5 and the overall analysis of the RAHS' properties is given in chapter 6.

Chapter 7 presents prototype implementations of both systems. Both have their own simulator for evaluation and demonstration purposes. The HAHS is also implemented as a middleware with the already existing *cluster eager value* method. The simulator and the middleware of the HAHS are further developments of the ones developed for the AHS, while a completely new simulator has been developed for the RAHS.

The simulators were used to evaluate the two systems with their concepts and methods. The evaluation presented in chapter 8 consists of measurements of the allocation time as well as the produced communication load, especially the maximum load. Those two aspects were compared to the original AHS as well as other concepts and methods of the system.

In chapter 9, related work from other researchers was presented and discussed. This chapter focuses on different MASs having many similarities to the AHS, HAHS and RAHS.

This thesis has shown that the two presented systems solve the problem of an autonomous and decentralized task allocation in a large scaled multi-core architecture. Depending on the chosen concept and method, the systems take more or less time to complete the *self-configuration*. The system's maximum communication load will vary with the chosen concept, but will be drastically lower than the maximum communication load produced by the AHS. Therefore, the goal of finding an AHS with a lower communication load is achieved. Besides evaluating both systems as substitutes for the AHS in large scaled architectures, some other findings are of interest.

When it comes to strict bandwidth limitations, an *organ task* HAHS should be preferred over a *single task* HAHS. Due to the allocation speedup in the *inter-cluster* cycle of an *organ task* concept it also should be preferred when the time required for allocation is critical. In case the PEs' underlying hardware is unreliable or tends to change its suitability quickly, a *single task* concept is more reasonable. The *single task* HAHS handles optimizations and failures with a finer granularity and can therefore react to those changes with less overhead. It is able to optimize and heal situations in which the *organ task* concept fails. Another evaluation result is the connection between the number of organs of the *organ task* concept and the system's performance. It shows how to determine a Pareto optimal number of organs in order to keep the allocation time and the maximum communication load as low as possible.

For the RAHS, it could be shown that it performs better in terms of maximum communication load per PE than the AHS. An interesting effect observed is the rise of the maximum communication load with the number of levels in the system. This observation was rather unexpected, but can be explained by the basic operational principle of each PE in the system which was taken from the AHS. An expected insight was the fact that a failure in a higher level causes more migrations and therefore a longer downtime than a failure in a lower level. Taking both findings into consideration, it seems more reasonable to keep the number of levels as low as possible. Still, in some application scenarios, like the initially motivated connected *smart city*, a hierarchical topology is given. This

topology consists of many *IoT* devices connected by routers. Here, the RAHS can match the scenario's network topology and thus avoid sending unnecessary messages through the routers.

As described above, the most suitable approach depends strongly on the application scenario. Many aspects of the scenario have to be considered, meaning the research on the topic of hierarchical AHSs is not complete yet. Different application scenarios may need different approaches to reach their full potential, but also the already existing approaches may be improved. Thus, the question arises which areas are promising candidates for future research. An evaluation of a *single task* HAHS in which PEs are allowed to make more than one decision per cycle would certainly be interesting. Such an approach would probably lead to an allocation speedup but might also lead to a suboptimal allocation among the clusters which needs to be optimized later. A further idea is a combination of the *organ task* concept and the *single task* concept. This idea involves the designer grouping the tasks into organs and the HAHS to distribute them to the clusters. When the *self-configuration* is completed, the groups will be dissolved and the system becomes a fully *single task* concept HAHS. This approach combines the fast and communication-less allocation of the *organ task* concept with the *single tasks* concept's flexibility regarding optimizations and failures.

For the RAHS, further research on additional methods to generate reliable suitability values is desirable. Additionally, a flexible dynamic method to determine the allocation level of a task in dependency of the suitability and load of the underlying cluster could be researched. Furthermore, a development into a self-building system could be considered, this means the system builds and even reconfigures its tree topology at run-time, based on network topology or functional resources of the PEs or any other criteria. Such a self-building approach probably involves additional hormone communication and would have an impact on allocation time and maximum communication load, similar to the *dynamic-dynamic* HAHS.

Appendix A

A.1 Allocation by a heuristic value in the *greatest hormone* method

This section briefly explains and visualizes the process of allocating a task in the HAHS, in case heuristic values are used (see section 4.7.4).

When a cluster head wins a task for which it only has a heuristic value, it starts the allocation by unlocking the task in its cluster (see figure A.1). As soon as all cluster children sent out their *eager values* for the unlocked task, the cluster head replaces the used heuristic cluster eager value by the greatest received eager value (see figure A.2).

This allows a reduced communication load for generating the *cluster eager values* and simultaneously maintains the dynamic reaction to suitability changes in the cluster children.

The disadvantage of this method is the potential mispredictions in the *cluster eager values*, e.g. the heuristic value for a task is higher than the real suitability in the cluster, thus the task will be allocated in a non-optimal cluster. This can be corrected by self-optimization phases (see figure A.3)) but depending on the configuration it is also possible that it will not be corrected during run-time.

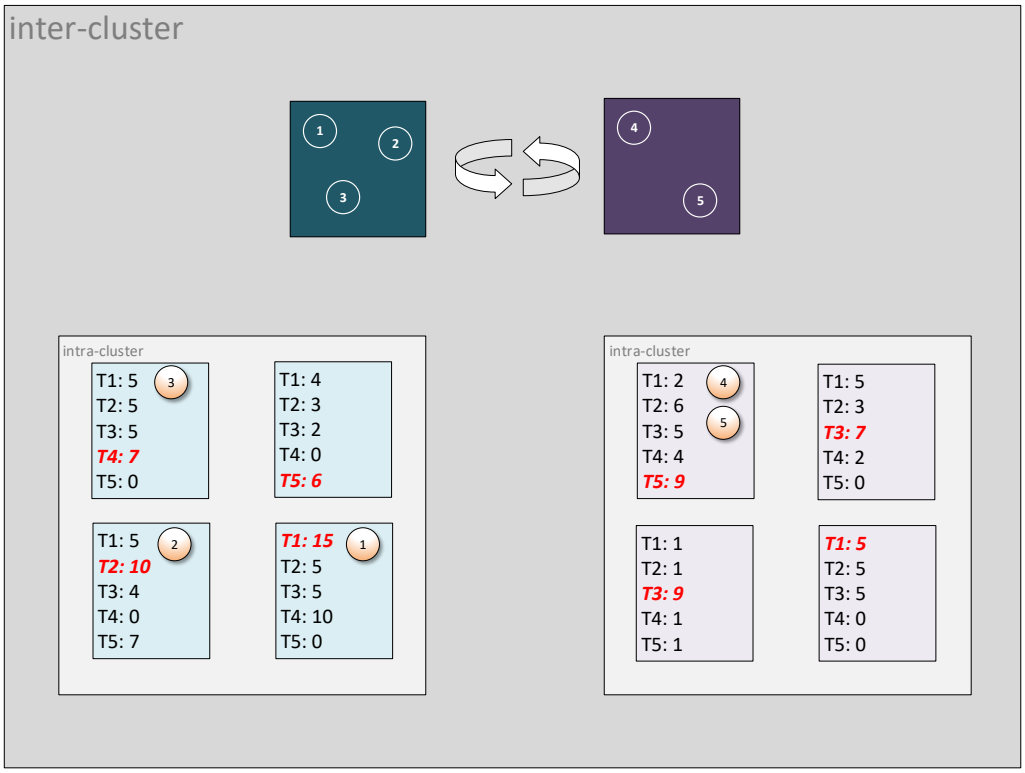


Figure A.1: The task distribution after the self-configuration phase

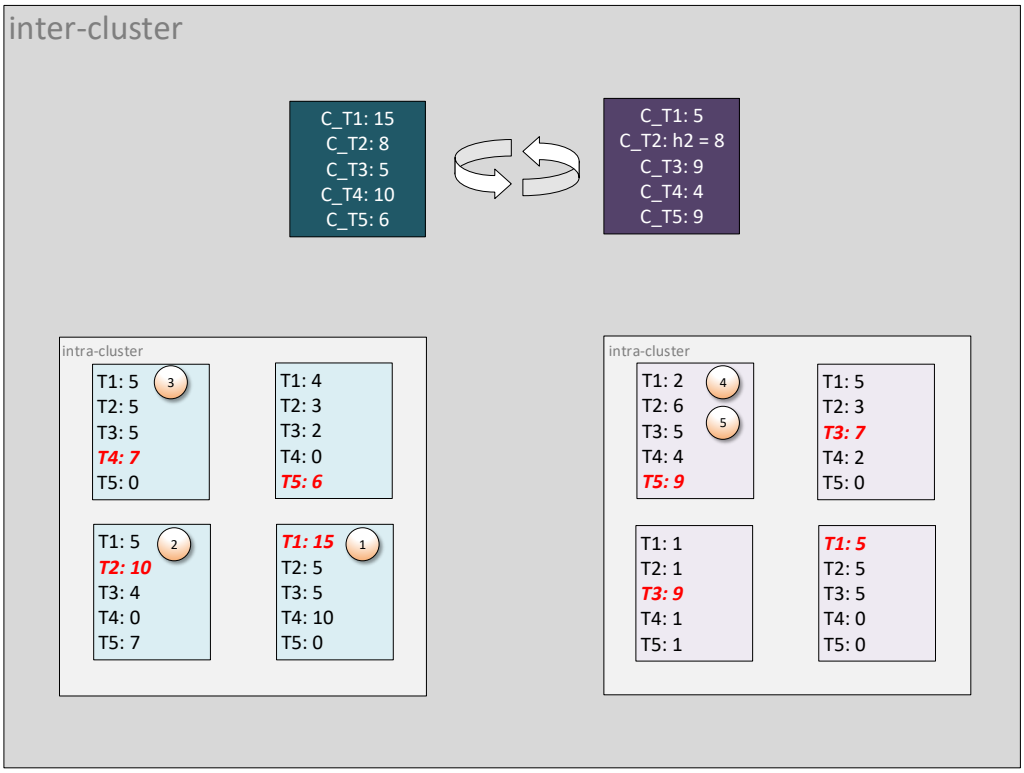


Figure A.2: The updated cluster eager values after the self-configuration phase

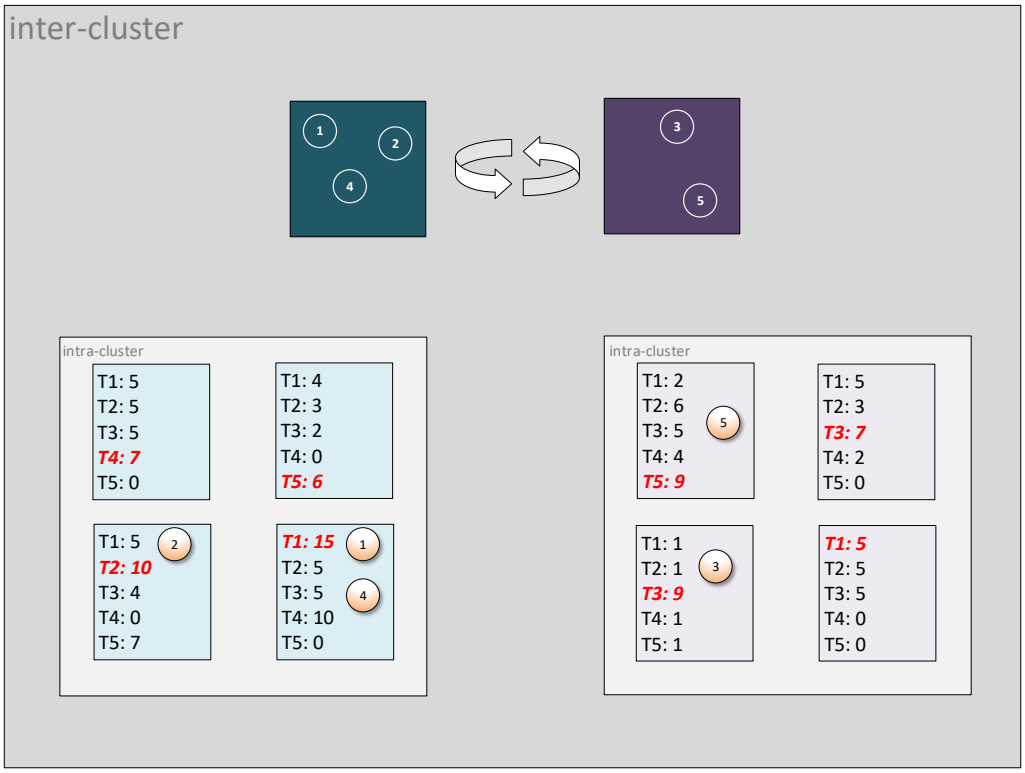


Figure A.3: The resulting task distribution after the first self-optimization phase

A.2 Equation to calculate the number of eager values in the organ task HAHS

In order to estimate the communication load of the *organ task* HAHS the maximum *eager value* emission has to be found. This turns out to be complicated, due to the sequential allocation of *organs* in the *inter-cluster* cycle and the following parallel allocation of tasks across the clusters. For this reason, an equation has to be found which estimates the worst-case *eager value* number in dependency of the hormone cycle (t). In order to find this equation, it has to be estimated how many tasks are unlocked already at the timestamp t. This is achieved by subtracting the timestamp t from the organ number k. The result is the number of not unlocked organs[1]. This number has to be multiplied by the organ size ($\frac{|T|}{k}$) and then subtracted from the total number of tasks in the system (1000 in this example). The result reflects the number of tasks which are already unlocked. From this number, the number of already allocated tasks has to be subtracted. The already allocated tasks are generally calculated by summing up 1 to t, but only as long as t does not exceed the organ number (k) or the tasks per organ ($\frac{|T|}{k}$). Those two points in time correspond to the state that all organs are allocated respectively the first organ allocated all its tasks. In case one of those points in time are exceeded, the growth of allocated tasks turns from exponential to linear. The corresponding sum (1 to $t-k$ or 1 to $t-\frac{1000}{k}$) has to be subtracted from the sum 1 to t. Function $\beta(t,k)$ (see equation A.1) calculates this number of allocated tasks for a given timestamp t and organ number k.

[1]Of course, this holds only for $t \leq k$ otherwise the result is 0

$$\alpha(t,k) = \begin{cases} \alpha(t,t-1) + \alpha(t, \frac{1000}{t-1}), & t > 0 \;\&\&\; t > \frac{1000}{k} \\ \sum_{i=1}^{t-k} i, & t > 0 \\ \sum_{i=1}^{t-\frac{1000}{k}} i, & t > \frac{1000}{k} \\ 0, & else \end{cases}$$

$$\beta(t,k) = \begin{cases} \sum_{i=1}^{t} i - \alpha(t,k), & t \leq o + \frac{1000}{k} \\ 0, & else \end{cases} \tag{A.1}$$

The number of tasks which are unlocked and not already allocated calculates to: $1000 - (\min(0,(k-t)) \cdot \frac{1000}{k}) - \beta(t-2,k)$. Every of those *free* tasks will be assigned to exactly one cluster. Therefore, each task will be applied by the number of PEs per cluster ($\frac{1000}{l}$). The number of *eager values* of the *free* tasks results in:

$$\gamma(t,k,l) = (1000 - (\min(0,(k-t)) \cdot \frac{1000}{k}) - \beta(t-2,k)) \cdot \frac{1000}{l} \tag{A.2}$$

Indeed, the estimation is not done yet. As a matter of fact, not only the *free* tasks produce *eager values*. Also, the tasks which were allocated in the cycle before, still produce some *eager values*. When a PE allocates a task at hormone cycle t, all other PEs in the same cluster receive the suppressor for the task in hormone cycle $t+1$ and will update their *eager value* to 0 and broadcast the 0-hormone. For calculating the number of those *0-eager values*, the number of allocated tasks in the cycle $t-2$ has to be calculated. Equation A.3 calculates this number and multiplies it with the number of PEs in each cluster subtracted by one (the one PE which allocated the task).

$$\delta(t,k,l) = (\beta(t-2,k) - \beta(t-3,k)) \cdot (\frac{1000}{l} - 1) \tag{A.3}$$

Finally, the total number of *eager values* is the sum of the *eager values* from the *free* tasks and δ:

$$\epsilon(t,k,l) = \gamma(t,k,l) + \delta(t,k,l) \tag{A.4}$$

Bibliography

[1] M. Al Faruque, R. Krist, and J. Henkel. Adam: Run-time agent-based distributed application mapping for on-chip communication. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 760–765, New York, NY, USA, 2008. ACM.

[2] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[3] M. Bando, K. Hasebe, A. Nakayama, A. Shibata, and Y. Sugiyama. Dynamical model of traffic congestion and numerical simulation. *Phys. Rev. E*, 51:1035–1042, Feb 1995.

[4] S. Baruah. Task partitioning upon heterogeneous multiprocessor platforms. In *Proceedings of the IEEE Real-Time Systems and Embedded Technology and Applications Symposium*, pages 536–543, 2004.

[5] N. Bensaid and P. Mathieu. A hybrid and hierarchical multi-agent architecture model. *Proceedings of PAAM 97*, pages 145–155, 1997.

[6] F. Berman, G. Fox, T. Hey, and A. J.G. Hey. *Grid computing: making the global infrastructure a reality*, volume 2. John Wiley and sons, 2003.

[7] S. C. Botelho and R. Alami. M+: a scheme for multi-robot cooperation through negotiated task allocation and achievement. In *IEEE international conference on robotics and automation*, pages 1234–1239, 1999.

[8] T. Braun, H. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. Reuther, J. Robertson, M. Theys, B. Yao, D. Hensgen, and R. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto

heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810 – 837, 2001.

[9] U. Brinkschulte. Reducing the communication overhead of an artificial hormone system for task allocation by a task window. In *Workshop on Embedded Self-Organizing Systems (ESOS 2013) co-located to the 10th International Conference on Autonomic Computing (ICAC 2013)*, June 2013.

[10] U. Brinkschulte. An artificial DNA for self-descripting and self-building embedded real-time systems. *Concurrency and Computation: Practice and Experience*, 28(14):3711–3729, 2014.

[11] U. Brinkschulte. Introducing virtual accelerators to decrease the communication overhead of an artificial hormone system for task allocation. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2014 IEEE 17th International Symposium on*, pages 117–124. IEEE, 2014.

[12] U. Brinkschulte. Prototypic implementation and evaluation of an artificial DNA for self-descripting and self-building embedded systems. *EURASIP Journal on Embedded Systems*, 2017(1):23, Feb 2017.

[13] U. Brinkschulte and M. Pacher. An agressive strategy for an artificial hormone system to minimize the task allocation time. In *2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, pages 188–195, April 2012.

[14] U. Brinkschulte, M. Pacher, and A. von Renteln. An artificial hormone system for self-organizing real-time task allocation in organic middleware. In *Organic Computing*, chapter 12, pages 261–283. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[15] U. Brinkschulte and T. Ungerer. *Mikrocontroller und Mikroprozessoren*. Springer-Verlag, 2010.

[16] E. Carvalho and F. Moraes. Congestion-aware task mapping in heterogeneous mpsocs. In *2008 International Symposium on System-on-Chip*, pages 1–4, Nov 2008.

[17] A. J. Chakravarti, G. Baumgartner, and M. Lauria. The organic grid: self-organizing computation on a peer-to-peer network. *IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans*, 35(3):373–384, 2005.

[18] J. Chen and D. Sun. Resource constrained multirobot task allocation based on leader–follower coalition methodology. *The International Journal of Robotics Research*, 30(12):1423–1434, 2011.

[19] Z. Cheng, H. Zhang, Y. Tan, and Y. Lim. Smt-based scheduling for multiprocessor real-time systems. In *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*, pages 1–7, June 2016.

[20] M. Coppola, B. Falsafi, J. Goodacre, and G. Kornaros. From embedded multi-core socs to scale-out processors. In *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 947–951. EDA Consortium, 2013.

[21] G. P. Das, T. M. McGinnity, S. A. Coleman, and L. Behera. A distributed task allocation algorithm for a multi-robot system in healthcare facilities. *Journal of Intelligent & Robotic Systems*, 80(1):33–58, 2015.

[22] J. de Lope, D. Maravall, and Y. Quiñonez. Response threshold models and stochastic learning automata for self-coordination of heterogeneous multi-task distribution in multi-robot systems. *Robotics and Autonomous Systems*, 61(7):714–720, 2013.

[23] O. Derin, D. Kabakci, and L. Fiorin. Online task remapping strategies for fault-tolerant network-on-chip multiprocessors. In *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '11, pages 129–136, New York, NY, USA, 2011. ACM.

[24] T. Ebi, H. Rauchfuss, A. Herkersdorf, and J. Henkel. Agent-based thermal management using real-time i/o communication relocation for 3d many-cores. In *International Workshop on Power and Timing Modeling, Optimization and Simulation*, pages 112–121. Springer, 2011.

[25] S. Edenhofer, S. Tomforde, D. Fischer, J. Hähner, F. Menzel, and S. Von Mammen. Decentralised trust-management inspired by ant pheromones. *International Journal of Mobile Network Design and Innovation*, 7(1):46–55, 2017.

[26] G. Gan, C. Ma, and J. Wu. *Data clustering: theory, algorithms, and applications*, volume 20. Siam, 2007.

[27] R. Geissbauer, S. Schrauf, P. Berttram, and Cheraghi F. Digital factories 2020: Shaping the future of manufacturing, April 2017.

[28] T. Hoefler and M. Snir. Generic topology mapping strategies for large-scale parallel architectures. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 75–84, New York, NY, USA, 2011. ACM.

[29] G. Hubert, L. Artola, and D. Regis. Impact of scaling on the soft error sensitivity of bulk, fdsoi and finfet technologies due to atmospheric radiation. *Integration, the VLSI Journal*, 50:39 – 47, 2015.

[30] J. Huppelsberg and K. Walter. *Kurzlehrbuch Physiologie*. Georg Thieme Verlag, 2013.

[31] O. Ibarra and C. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *J. ACM*, 24(2):280–289, April 1977.

[32] B. Kamel, A K. Singh, A. Benyamina, A. Kumar, and P. Boulet. Heuristics for dynamic task and communications mapping in noc-based heterogeneous mpsocs. *The Mediterranean Journal of Computers and Networks*, 9, 10 2013.

[33] A. Kovalov, E. Lobe, A. Gerndt, and D. Lüdtke. Task-node mapping in an arbitrary computer network using smt solver. In N. Polikarpova and S. Schneider, editors, *Integrated Formal Methods*, pages 177–191, Cham, 2017. Springer International Publishing.

[34] M. Krainin, B. An, and V. Lesser. An application of automated negotiation to distributed task allocation. In *Proceedings of the 2007 IEEE/WIC/ACM international conference on intelligent agent technology*, pages 138–145. IEEE Computer Society, 2007.

[35] B. Kreaseck, L. Carter, H. Casanova, and J. Ferrante. Autonomous protocols for bandwidth-centric scheduling of independent-task applications. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 10–pp. IEEE, 2003.

[36] T. Lei and S. Kumar. A two-step genetic algorithm for mapping task graphs to a network on chip architecture. In *Euromicro Symposium on Digital System Design, 2003. Proceedings.*, pages 180–187, Sep. 2003.

[37] J. Lienig and M. Thiele. The pressing need for electromigration-aware physical design. In *Proceedings of the 2018 International Symposium on Physical Design,ISPD 2018, Monterey, CA, USA, March 25-28, 2018*, pages 144–151, 2018.

[38] S. Lucero. Iot platforms: enabling the internet of things. *White paper*, 2016.

[39] A. Lund and U. Brinkschulte. Task-allocation in a large scaled hierarchical many-core topology. In *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, June 2018.

[40] A. Lund, M. Pacher, and U. Brinkschulte. Towards a recursive approach for an artificial hormone system. In *Seventh IEEE Workshop on Self-Organizing Real-Time Systems (SORT 2016)*, May 2016.

[41] A. Lund, M. Pacher, and U. Brinkschulte. Task-allocation in a hierarchical network topology by means of an organic middleware. In *ARCS 2017; 30th International Conference on Architecture of Computing Systems*, pages 1–8, April 2017.

[42] Microsoft. MFC desktop applications, 8 2018.

[43] C. Müller-Schloer. Organic computing: On the feasibility of controlled emergence. In *Proceedings of the 2Nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '04, pages 2–5. ACM, 2004.

[44] C. Müller-Schloer, H. Schmeck, and T. Ungerer. *Organic computing- A paradigm shift for complex systems*. Springer Science & Business Media, 2011.

[45] C. Müller-Schloer and B. Sick. Controlled emergence and self-organization. In *Organic Computing*, chapter 4, pages 81–103. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[46] C. Müller-Schloer, C. von der Malsburg, and R. P. Würtz. Organic computing. *Informatik Spektrum*, 27(4):332–336, 2004.

[47] M. Pacher. Two-level extensions of an artificial hormone system. *Concurrency and Computation: Practice and Experience*, 28(14):3730–3750, 2016.

[48] E. Pop, S. Sinha, and K. E. Goodson. Heat generation and transport in nanometer-scale transistors. *Proceedings of the IEEE*, 94(8):1587–1601, Aug 2006.

[49] K. Rupp. 42 years of microprocessor trend data, 9 2018.

[50] P. K. Schelling, L. Shi, and K. E. Goodson. Managing heat for electronics. *Materials Today*, 8(6):30 – 35, 2005.

[51] M. Z. Shafiq, L. Ji, A. X. Liu, J. Pang, and J. Wang. A first look at cellular machine-to-machine traffic: Large scale measurement and characterization. *SIGMETRICS Perform. Eval. Rev.*, 40(1):65–76, June 2012.

[52] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, July 1948.

[53] I. J. Shapiro and K. S. Narendra. Use of stochastic automata for parameter self-optimization with multimodal performance criteria. *IEEE Transactions on Systems Science and Cybernetics*, 5(4):352–360, Oct 1969.

[54] O. Shehory and S. Kraus. Methods for task allocation via agent coalition formation. *Artificial intelligence*, 101(1):165–200, 1998.

[55] A. Singh, T. Srikanthan, A. Kumar, and W. Jigang. Communication-aware heuristics for run-time task mapping on noc-based mpsoc platforms.

Journal of Systems Architecture, 56(7):242 – 255, 2010. Special Issue on HW/SW Co-Design: Systems and Networks on Chip.

[56] J. R. Sklaroff. Redundancy management technique for space shuttle computers. *IBM Journal of Research and Development*, 20(1):20–28, Jan 1976.

[57] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, C-29(12):1104–1113, Dec 1980.

[58] Y. Sugiyama, M. Fukui, M. Kikuchi, K. Hasebe, A. Nakayama, K. Nishinari, S.i Tadaki, and S. Yukawa. Traffic jams without bottlenecksexperimental evidence for the physical mechanism of the formation of a jam. *New journal of physics*, 10(3):033001, 2008.

[59] The QT Company. QT|Cross-platform software development for embedded & desktop, 8 2018.

[60] VDE/ITG. VDE/ITG/GI-positionspapier organic computing: Computer und systemarchitektur im jahr 2010. *GI, ITG, VDE*, pages 1–7, 2003.

[61] A. von Renteln, U. Brinkschulte, and M. Pacher. Introducing a simplified implementation of the ahs organic middleware. In *Proceedings of the 2011 workshop on Organic computing*, pages 51–58. ACM, 2011.

[62] A. Weichslgartner, S. Wildermann, and J. Teich. Dynamic decentralized mapping of tree-structured applications on noc architectures. In *Proceedings of the Fifth ACM/IEEE International Symposium*, pages 201–208, May 2011.

[63] S. Wildermann, T. Ziermann, and J. Teich. Run time mapping of adaptive applications onto homogeneous noc-based reconfigurable architectures. In *2009 International Conference on Field-Programmable Technology*, pages 514–517, Dec 2009.

[64] M. Wooldridge. *An introduction to multiagent systems*. John Wiley & Sons, 2009.

[65] R. P Würtz. *Organic computing*. Springer Science & Business Media, 2008.

[66] H. Yang and S. Ha. Pipelined data parallel task mapping/scheduling technique for mpsoc. In *2009 Design, Automation Test in Europe Conference Exhibition*, pages 69–74, April 2009.

[67] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi. Internet of things for smart cities. *IEEE Internet of Things journal*, 1(1):22–32, 2014.