Optimizing Distributed Test Generation

Harry Gros-Désormeaux
ironb972@gmail.com
Hacène Fouchal
Hacene.Fouchal@univ-reims.fr
Philippe Hunel
Philippe.Hunel@univ-ag.fr

Abstract: This paper presents two optimizations of the master-slave paradigm which provides more robustness. The former uses a mobile master which randomly jumps from node to node in order to dispatch the bandwidth load during the application execution. The latter aims to fully decentralize the paradigm like peer-to-peer systems. We present and experiment our new schemes on an application dedicated to test generation used in protocol engineering. We show that they can help for unknown dynamic environments where no information is given about nodes configuration.

Keywords : load balancing, master-slave, distributed networks, peer-to-peer

1 Introduction

The Master-Slave paradigm is a well-known concept used in distributed computing for decades. The master distributes works to several slaves for computing. Whenever a slave ends its computation, it sends back results to the master or to some repository. Actually, most of distributed computing environments leverage this concept to distribute the load across the network. Performances are very interesting for some types of application from which parallelism can be expressed as multiple independent tasks.

Numerous scientific applications use the master-slave paradigm and lack optimization for new dynamic environments as the peer-to-peer ones, Internet, etc. Such networks constantly evolve and decrease the relevance for using the master-slave paradigm according to load balancing aspects as well as robustness ones. Indeed, it is hard to compute efficient chunk size for works which have to be given to the slaves. Too small chunks will produce bandwidth overhead and too large ones provide computing overhead. On other hand, the fixed location of the master naturally becomes a soft spot. To overcome some of these downsides, we suggest to move the master in a randomly way on the network to enhance the robustness of the paradigm. Finally, a full decentralization is proposed through a total replication of the master behavior on all the network nodes.

The structure of this paper is as follows : section 2 recalls some relevant works done in the field of load balancing for the master-slave paradigm. Section 3 describes the two schemes that we have studied to cope with dynamic and heterogeneous environments.

Some preliminary experiments are exposed in section 4 where performance results are given. Finally, section 5 concludes and details future works to complete this study.

2 Related Work

Load Balancing for Master-Slave distribution is an important field in where a lot of works have been done. Bellow, we describe some known studies in this area. Heymann et al. [HSLL04] present an adaptive distribution strategy which optimize two criterions : efficiency, that is the optimal number of slaves set up according to the application speedup, and the make-span, by monitoring execution time for each distributed work over iterations (time splits).

An asymptotic optimal scheduling algorithm is given in [BLR03] for indivisible tasks following a linear cost model. The optimal number of slaves is solved with a linear programming theory and is used in the computation of the optimal number of tasks. Another complementary study, Almeida et al. [AGM06] derive the scheduling problem in a task allocation problem [IK88].

[Hag97] designs a sophisticated heuristic relevant for homogeneous environment that uses mean and standard deviation with the execution time of a processor to compute the batch element size. Despite the high complexity, this algorithm gives better results than others.

In some similar way, Cesar and al. [CMSL06] adjust in a regular way a ratio which gives the size of non-allocated elements from a batch work using predictive formulas.

Yang and al [YSF03] suggest a scheduling algorithm for loosely coupled applications by estimating the average load of a processor from its history. This algorithm is more efficient than ones given in [Dai02] which leverage the NWS service[Wol98, WSH99] to estimate processors load.

We suggest two schemes that enhance traditional ways to cope with load balancing in a Master-Slave environment and allow this class of application to become more decentralized. The former is an adaptation of the original paradigm in which the master moves. The latter emphasizes the actual trend used in decentralization processes : a load balancing emergent property results from local nodes interactions.

3 Incentives and Model

3.1 The master-slave paradigm and its drawbacks

The Master-Slave algorithm is well known in distributed programming and is commonly described as follows : one process performs an administrative role by allocating independent subtasks (the whole is the entire work to be done). In general, results are gathered on the master but can be stored on other repository (see Figure 1). More formally, a process p_m divides work W in some independent works w_n with $n \in \mathbb{N}^*$ which are sent to k

process p_k with $k \in \mathbb{N}^*$.

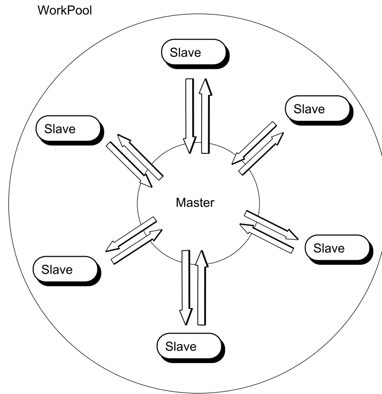

Abbildung 1: The Master-Slave Paradigm

As emphasized by Crutchfield and Mitchell [CM94], centralized applications exhibit three major drawbacks with respect to :

- the speed: a central coordinator can be a bottleneck to fast information processing,
- robustness: central failure destroys system structure,
- balanced resource allocation: the main controller draws a lot of resources.

These problems can arise on all Master-Slave schemes which limit the use of this paradigm on dynamic systems. For sake of reusability, we suggest some mechanisms to overcome these drawbacks.

3.2 The mobile Master-Slave paradigm

The mobile Master-Slave paradigm consists of permanent move of the master across the network. The paradigm can be formalized as above :

Definition 3.1 (Mobile Master)**.** The *mobile master* is an administrative process p_{i_k} hosted by some resource H_k which must achieve coordination and distributes the work $W = w_1, \ldots, w_j$ over a set of processes $p = p_1, p_2, \ldots, p_n$ respectively hosted by $H_i(1 \leq i \leq n)$.

By moving the coordinator during the application execution, the resource allocation is shared along the whole process over the nodes of the network. Furthermore, bandwidth allocation changes continously during the application execution and de facto, inherently shares in a balanced way the bandwidth between all computational resources.

Definition 3.2. We call trajectory $\mathcal{T}$ of the mobile master, the ordered set of hosts which possessed the administrative process.

New masters are elected randomly. The current master randomly selects the new master after a fixed period of time (time frame). Each computational resources keeps in a list of master locations. To ensure a minimal fault tolerance, soft state can maintain the system consistency : if for some reasons, the current master is not reachable, the previous one is reached in order to re-elect a new one. To enable such functionality, our computational resources store the master trajectory during our application execution.

Although our paradigm does not use the full peer-to-peer paradigm, it turns out to make the system more balanced. In fact, all system agents share the same capabilities. An example of such a system can be seen in Figure 2.

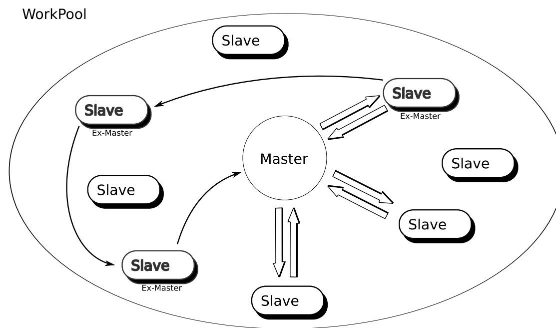

Abbildung 2: The Mobile Master Paradigm

Unfortunately, in such case, our scheme is not efficient. Moving too much data can struggle the overall performance of the system. For this reason all data relevant to the master computations are replicated on each slave. Thereby, just work index can be used to determine the new data which has to be processed by a slaver. So, very small information need to be moved whenever a new master is elected. Then results have to be stored on any repository on the the master.

Proposition 3.1. *If the whole work $W = w_1, \ldots, w_j$ is already on each host then the work index $i \in [1..j]$ is sufficient to characterize the task to be handled by a slave.*

Definition 3.3. We call *trace* of the mobile master the set of works $w_i (1 \leq i \leq j)$ which have already been distributed and processed.

Corollary 3.1. *If propositon 3.1 holds, then the trace of the mobile master can only be its work indexes.*

The model presented above is rather simple. More complex mechanisms can be implemented to leverage the load balancing techniques described previously in our related work section (see section 2). These schemes often keep information computed from scratch about nodes in the network to achieve their goal. In this case, the masters can create their

own subnetwork by binding each other and update[1] themselves on a regular way. All information is then centralized in this *Master Network*. So, such a scheme virtually allows almost all the load balancing scheme described in section 2 to be used with our mobile master paradigm without too much transformations.

Proposition 3.2. *The more the masters we have, the more the application gains in robustness.*

Beweis. More formally, let p be the probability for a node in the network to be flawed. N is the cardinal of the network and thus, the probability for a node to be the master is $\frac{n}{N}$ where n is the total number of different masters used during the application. A master is unreachable with probability $p\frac{n}{N}$. Finally, the probability that a node fails to find the master is $(p\frac{n}{N})^k$ with $1 \leq k \leq n$, the actual number of ex-masters. We notice that this probability lowers whenever the master moves inducing a more robust application to each move. □

3.3 Full Decentralization

Our previous scheme uses mobility to overcome problems which arise in centralized applications whereas our new one emphasizes ubiquity. To fully decentralize the Master-Slave paradigm, we propose to replicate the master process on each nodes in the network.

Definition 3.4 (Decentralized Masters)**.** Decentralized masters are computational resources H_k which bear an administrative process p_k and which have their own batch of works W_k.

Full peer-to-peer applications are the ones for which each peer is a decentralized master. So, load balancing becomes a more challenging task because each slave becomes its own master and because the data has to be initially at some node, it has to be spread over the network by load balancing mechanism. Bahi and al.[BCV05] have proved that synchronous distributed load balancing can be used on dynamic networks. We propose to use the GAE algorithm to distribute works of the batch pool between all network nodes.

3.3.1 The GAE algorithm

The GAE algorithm derives from GDE algorithms used in [HLM$^+$90]. These algorithms find an edge coloration of the overlaying communication graph $G(V, E)$, where V is the vertices set and E, the edge set, and uses edge color as dimension to balance the load. The graph $G(V, E)$ is transformed in a k-color graph $G_k(V, E_k)$ where E_k is a set of 3-tuples $\{i, j; c\}$ with (i, j) an edge of E and c, the color of the edge (i, j) $(0 \leq c \leq k - 1)$.
At time t, each node balances along one dimension with its neighbor its load

$$w_i^{(t+1)} = \begin{cases} w_i^{(t)} + \lambda(w_j^{(t)} - w_i^{(t)}) & \text{if } \exists j \text{ such that } \{(i,j);c\} \in E_k \\ w_i^{(t)} & \text{otherwise} \end{cases} \quad (1)$$

[1] Note that this update can be full or partial according to our objectives

The parameter λ is empirically fixed according to the network topology.

In order to use this algorithm in our decentralized scheme, load has to be considered as the size of the batch and nodes swap a part of this batch between each other to balance the load in the network. This algorithm does not take into account the fault tolerance aspect. However, this problem can be solved through batch elements replication.

We give here our interpretation of this algorithm that joins the current mainstream for massive distributed computations, that is *Emergence and Self-Organization* [Joh06, SGK06, WH04]. It is said that emergent properties are necessary to compute at massive scale when central coordinators (servers) cannot cope with the high load they encounter. These properties are obtained through local interactions between all agents which give at the macro-level the desired properties for the global system. For example, ant-based algorithms leverage this paradigm as well as many other complex systems that can be found in life. Wolfram[Wol02], through cellular automata, studies this principle and notably defines a class of these systems corresponding to complex ones (class IV) that can even simulate life. The GAE algorithm can be seen as local interactions from each node which balances its load locally. As a consequence, the distributed application turns out to be well balanced w.r.t the load.

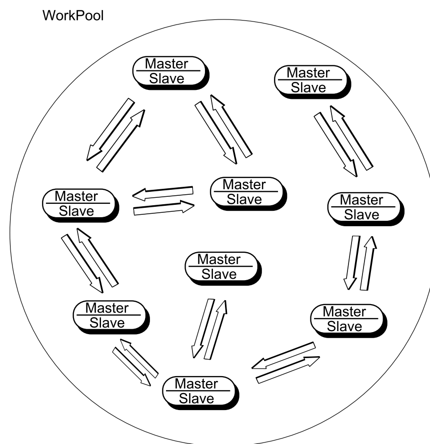

Abbildung 3: Full Decentralized Master-Slave Paradigm

4 Experimentation

We experiment our decentralization scheme on a distributed application used for test sequence generation [GDFH06]. This application computes UIO (Unique Input Output) sequences[SD88] used in conformance testing in a distributed manner. To test a system, its specification have to be described by a deterministic automaton. Computed from the automaton, unique sequences of transitions which characterize controllable states (states

on which the user can act) are drawn on the system implementation in order to detect incorrect behaviors. The distributed test generation application leverages the master-slave paradigm : the master distributes to computational resources: automaton and states for which unique test sequences have to be found when they exist.

Implementation is done through the ProActive API [BBC+06] which leverages the power of the active object model. Active objects can move freely accross the network and possess their own execution context. They can be accessed by any other process or object owned by the application. Furthermore, several mechanisms (pointer redirection, address update, etc.) ensure active object consistency whenever they move on the network.

4.1 ProActive

The ProActive framework provides the active object model used to implement our applications. An active object is a distributed object bound to its own thread and to a list of possibly remote methods calls to be processed. Indeed, each remote call to an active object is asynchronous, that is, the caller continues its execution flow until the end or some barrier mechanism. The request is put in a list of pending method calls in order to be processed later (by default, in a FIFO manner) or consumed according to a user-defined strategy. At the caller, each remote procedure result is bound to an abstraction, a *future*, which empowers asynchronicity in the application. Accessing futures only puts synchronization bareer the program if the remote procedure results are not yet provided by the active object. This synchronization mechanism is also called *wait-by-necessity*.

4.1.1 The Mobile Master

Our implementation of the mobile master for test generation use three active objects :

- the *master*, the process which distributes controllable states to the computational resources,
- the *slave*, the computational resource which computes unique sequence from the received controllable states,
- and the *storage*, which keeps the unique test sequences found at the slaves.

As described previously in section 3.2, the mobile master periodically moves on each slave using the ProActive mobility primitives. We fixed empirically each move to be done after 10 seconds. Moreover, each slave stores the master trajectory, that is, it keeps each master address during the test generation. If for some reasons, the master cannot be reached, slaves contact the previous slave which hosted the master in order to respawn the administrative process. We fixed a quorum of 70% of slaves to contact the ex-master before it can respawn the master. This let suppose that we assume a connected network without more than 30% of unreachable nodes.

4.1.2 The Decentralized peer-to-peer application

In this implementation of the distributed test generation, each peer inherits the master capabilities. This yields to an active object *peer* for each computational resource on the network. As seen previously, an active object *storage* is used to store unique sequences found by each slave. We used a third active object, the *time server*, to synchronize the load balancing by color. This object is also used to synchronize our greedy coloration algorithm : each node of the network runs algorithm 1 at its turn.

Algorithm 1: Distributed Edge Coloration

```
Data: Set of neighbors V of node i
foreach neighbor j do
    if edge (i, j) is not colored then
        Take the first available color c;
        /* Check if c is not already used by the neighbor j,
           and update it if needed                              */
        c = UpdateColor(i,j,c);
        Color (i, j) with d.
```

Function UpdateColor(*node i, neighbor j, color c*)

```
if c is not available at node j then
    Update c with the first available color following c;
Color (i, j) with c;
return c
```

The parameter λ used in equation 1 is empirically set to 0.5 and each node updates its load with its neighbor every five seconds.

4.2 Performance Evaluation

Experiments were conducted on 10 personal computers (INTEL P4 2.4 GHZ - 128 Mo RAM) with an OS-based Linux (FEDORA CORE 6) linked together with a switch 100 Mbps. Written in JAVA 1.5, our software uses JDK 1.5.0 as well as ProActive 3.2.

To compare our schemes, we generate several random automata whose specifications largely encompass real ones. Twenty 5000-states automata were used during these preliminary experiments for each scheme. Figure 4 shows performance comparisons between the two methods: the mobile and the decentralized one.

The mobile master outperforms the decentralized method on our samples. First, it is due to our testbed conditions. Indeed, our decentralized method is designed to cope with dynamic

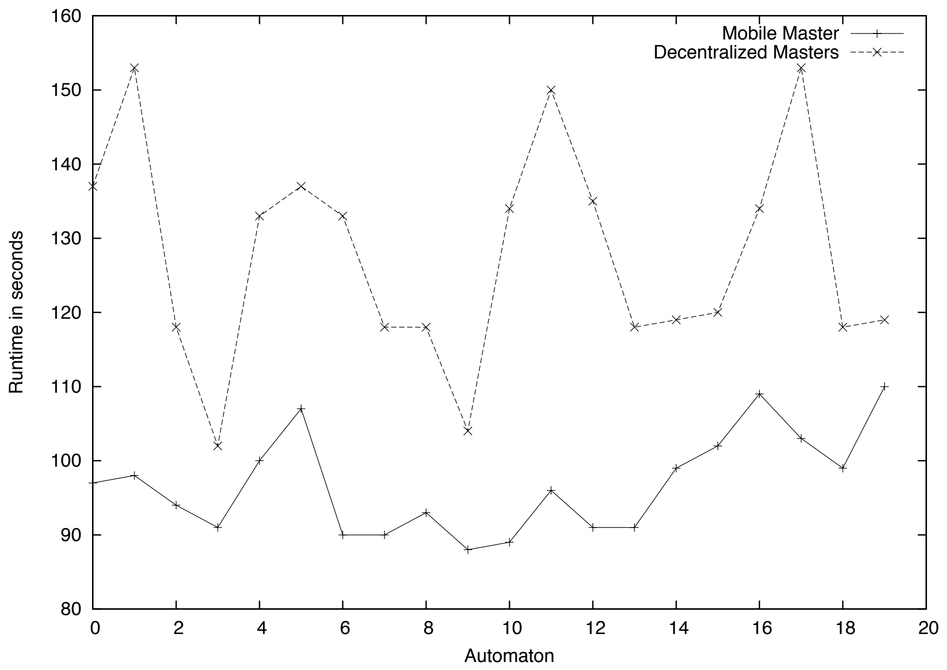

Abbildung 4: Mobile Master vs Decentralized Masters

large scale networks whereas our network environments is small, homogeneous and fast. Further, it suffers at starting from the edge coloration overhead which increases its runtime. On the other side, very low communication overhead is generated for the mobile master scheme since only work indexes (states number) are exchanged between the master and its slave. As a consequence, this scheme tends to be optimal more especially when distributed works are small. Last but not least, the distributed masters update their load every five seconds along a color and de facto, our application converges more slowly toward optimal resource allocation since works are centralized initially. Finally, some limitations hold such as our computer memory, which cannot get very large automata like millions states ones. In these conditions, the optimal number of computational resources can be maintained low and benefits the mobile master scheme. Figure 5 gives the speedup of the Mobile master application speedup (mean : 4.8) as well as the Decentralized masters one (mean : 3.7).

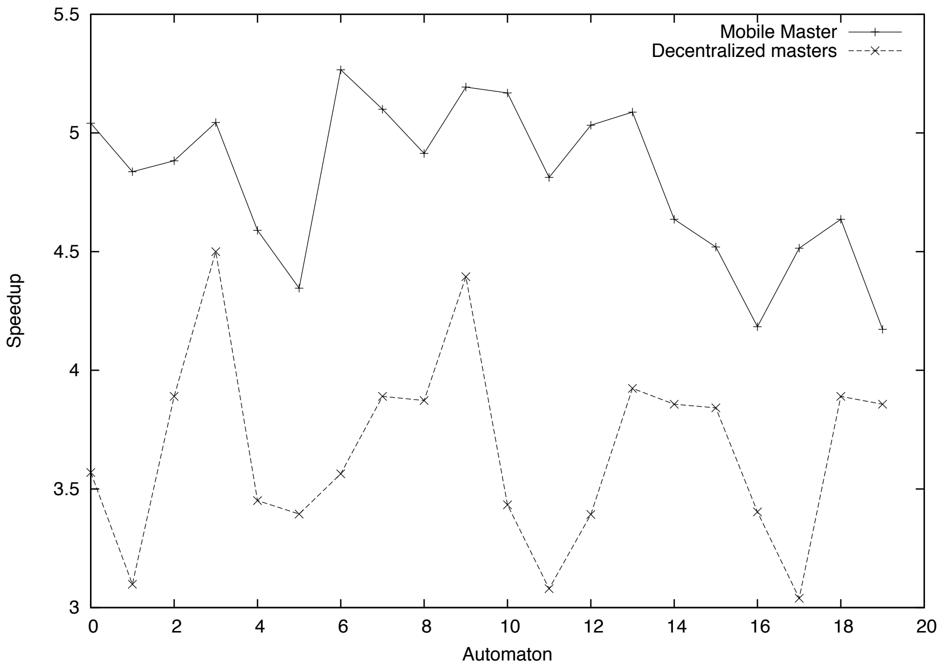

Abbildung 5: Speedup for Mobile Master and Decentralized masters

Although our decentralized masters scheme seems to be not efficient in our experiments, it is very likely it gains advantage in dynamic large scale networks. We simulate our two schemes on a parallel machine (quadri-XEON 3.6 ghz - 4Go RAM). We set the number of computational resources to 40. Unfortunately, communications exchange use the native interface memory of the system, so bandwidth is larger than real network ones. The mobile master scheme never succeeds to run properly. Investigations are on the way to know whether our implementation have memory leaks or crashes because a lot of stress are put on the system.

5 Conclusion

We suggested in this paper two schemes that enhance the master-slave paradigm. This paradigm tends to be inappropriate for large scale dynamic networks. Indeed, centralized applications are be proved to not be efficient for these kind of environments. Our first scheme uses mobility to overcome robustness downsides known for the master-slave paradigm in dynamic networks. The latter uses ubiquity and follows the actual mainstream which consists of solving problems through emergent properties on large scale networks. Our schemes were implemented on a distributed test generation application and then compared on a small homogeneous environment. This preliminary study reveals that the master-slave paradigm, enhanced with our schemes, can still be used in our actual networks without a lot of modifications and opens the way to automatic transformations for this old parallel method.

Our two schemes must be compared more thoroughly to validate our assumptions. They have to be deployed not only on a larger and more dynamic environments but also must be applied on others applications which leverage the master-slave paradigm. Some work is in progress to provide a generic platform which ease automatic transformations of applications leveraging the master-slave paradigm.

Literatur

[AGM06] Francisco Almeida, Daniel Gonzalez und Luz Marina Moreno. The master-slave paradigm on heterogeneous systems: A dynamic programming approach for the optimal mapping. *Journal of Systems Architecture*, 52(2):105–116, Februar 2006.

[BBC+06] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel und Romain Quilici. *Grid Computing: Software Environments and Tools*, Kapitel Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[BCV05] Jacques Bahi, Raphael Couturier und Flavien Vernier. Synchronous Distributed Load Balancing On Dynamic Networks. *Journal of Parallel and Distributed Computing*, 65(11):1397–1405, November 2005.

[BLR03] O. Beaumont, A. Legrand und Y. Robert. Optimal algorithms for scheduling divisible workloads on heterogeneous systems, 2003.

[CM94] James P. Crutchfield und Melanie Mitchell. The Evolution Of Emergent Computation. Bericht 94-03-012, Santa Fe Institute, 1994.

[CMSL06] E. Cesar, A. Moreno, J. Sorribes und E. Luque. Modeling Master/Worker applications for automatic performance tuning. *Parallel Computing*, In Press, Corrected Proof:–, 2006.

[Dai02] H. Dail. A Modular Framework for Adaptive Scheduling in Grid Application Development Environments, 2002.

[GDFH06] H. Gros-Desormeaux, H. Fouchal und P. Hunel. Testing Timed Systems Using a Distributed Environment. In *Sixth IEEE International Symposium and School on Advance Distributed Systems*, 2006.

[Hag97] Torben Hagerup. Allocating Independent Tasks to Parallel Processors: An Experimental Study. *J. Parallel Distrib. Comput*, 47(2):185–197, 1997.

[HLM+90] Hosseini, Litow, Malkawi, McPherson und Vairavan. Analysis of a Graph Coloring Based Distributed Load Balancing Algorithm. *JPDC: Journal of Parallel and Distributed Computing*, 10, 1990.

[HSLL04] E. Heymann, M. A. Senar, E. Luque und M. Livny. Efficient resource management applied to master-worker applications. *Journal of Parallel and Distributed Computing*, 64(6):767–773, Juni 2004.

[IK88] T. Ibaraki und N. Katoh. *Resource Allocation Problems: Algorithmic Approaches*. MIT Press, Cambridge, MA, 1988.

[Joh06] Christopher W. Johnson. What are emergent properties and how do they affect the engineering of complex systems? *Reliability Engineering & System Safety*, In Press, Corrected Proof:–, 2006.

[SD88] K. Sabnani und A. Dahbura. A Protocol Test Generation Procedure. 15:285–297, 1988.

[SGK06] Giovanna Di Marzo Serugendo, Marie-Pierre Gleizes und Anthony Karageorgos. Self-Organisation and Emergence in Multi-Agent Systems: An Overview. In *Informatica, An International Journal of Computing and Informatics, ISSN 0350-5596*, Jgg. 30, Seiten 45–54, Ljublana, Slovenia, janvier 2006. The Slovene Society Informatika.

[WH04] Tom De Wolf und Tom Holvoet. Emergence Versus Self-Organisation: Different Concepts but Promising When Combined. In Sven Brueckner, Giovanna Di Marzo Serugendo, Anthony Karageorgos und Radhika Nagpal, Hrsg., *Engineering Self-Organising Systems*, Jgg. 3464 of *Lecture Notes in Computer Science*, Seiten 1–15. Springer, 2004.

[Wol98] Richard Wolski. Dynamically forecasting network performance using the Network Weather Service. *Cluster Computing*, 1(1):119–132, 1998.

[Wol02] Stephen Wolfram. *A New Kind of Science*. Wolfram Media Inc., 2002.

[WSH99] Rich Wolski, Neil T. Spring und Jim Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.

[YSF03] Lingyun Yang, Jennifer M. Schopf und Ian Foster. Conservative Scheduling: Using Predicted Variance to Improve Scheduling Decisions in Dynamic Environments. In *SC'2003 Conference CD*, Phoenix, AZ, November 2003. IEEE/ACM SIGARCH. ANL.