A Multi-agent System Approach to Load-Balancing and Resource Allocation for Distributed Computing

Soumya Banerjee and Joshua P. Hecker

1 Introduction

Recent years have seen a trend in moving large computational tasks to collections of inexpensive, commercial off-the-shelf (COTS) computers that are geographically distributed. This has contributed significantly to the advancement of science by providing access to large-scale shared computing resources on which to solve computationally expensive problems. Some common examples are SETI@home [1] which runs tasks on millions of computers worldwide and Google MapReduce [9] which distributes calculation of web crawled metrics among thousands of computers. This move towards distributed computing has created a need for efficient task allocation and scheduling algorithms. Such algorithms should be very scalable since these systems typically have thousands to millions of computers. They should also be robust to single-point failures and be adaptive to task demand. Recent research on grid resource allocation has focused on volunteer resource allocation, agreement-based resource allocation, and economic resource allocation [17]. Multi-agent decentralized systems offer an exciting approach to distributed resource allocation. They have emergent global properties which arise from local interactions and have been previously used to model biological phenomena [2–7, 16, 19] and

S. Banerjee (✉)
The Broad Institute of MIT and Harvard, Cambridge, MA, USA

Complex Biological Systems Alliance, North Andover, MA, USA

Ronin Institute, Montclair, NJ, USA

University of Oxford, Mathematical Institute, Oxford, UK
e-mail: neel.soumya@gmail.com

J.P. Hecker
Department of Computer Science, University of New Mexico, Albuquerque, NM, USA
e-mail: jhecker@cs.unm.edu

solve real-world problems [11–15]. Here we use such a decentralized computing approach to allocate and schedule tasks on a grid. The remainder of the paper is organized as follows: Sect. 2 formalizes the problems and states the assumptions, Sect. 3 briefly reviews decentralized computing and the advantages it can afford to a distributed allocation problem, Sect. 4 introduces multi-agent systems, Sect. 5 introduces the simulator used for the experiments in this paper, Sect. 6 discusses the dRAP algorithm, Sect. 7 deals with analysis of the cost of searching through the global queue, Sect. 8 discusses some dRAP optimization techniques influenced by the immune system, Sect. 9 deals with experiments and results, Sect. 10 discusses related work in this area, and Sect. 11 presents concluding remarks and outlines future work.

2 Statement of Problem and Assumptions

Assume there is a queue Q of processes waiting to be allocated to processors. Each process is required to declare a priori its resource requirements viz. the number of threads into which it can be parallelized (TH_n) and the number of system resources it requires (the number of CPUs is assumed to be equal to the number of threads which can be run in parallel, $\mathrm{CPU}_{\mathrm{req}}$). Our system departs from traditional resource allocation techniques in that there is no centralized dispatcher. Instead, we dynamically organize a system of geographically distributed computers into clusters to service each process in Q. Over time, clusters of computers are dynamically created, dissociated, and created again in order to serve the resource requirements of the processes in Q. We define a *cluster* as a network of computers which together can completely service the resource requirements of a single process. Clusters of computers are created so as to be proximal to each other in order to reduce latency and communication costs.

We acknowledge the following assumptions in our system:

1. Distributed computers can communicate with each other.
2. There are advantages to computing with geographically proximal computers due to network latency and bandwidth limitations.
3. A new process P_1 that comes in the system will declare a priori the number of threads that it can be parallelized into and its resource requirements (e.g., the number of CPUs it will require, I/O devices required, amount of memory, etc.).
4. The approach will become viable in the asymptotic region of millions or billions of geographically dispersed computers, when there will be expected benefits from a decentralized computing approach that exploits geographical proximity and reduces latency costs, as opposed to a centralized monitor.

3 Decentralized Computing

The extreme size of the computing grid and an ever-increasing demand for computational power places exacting demands on any scheduling, allocation, and load-balancing algorithm. Here we argue that a decentralized computing paradigm presents an ideal solution to the bottlenecks and single-point failures inherent to a centralized monitor tasked with allocating resources and balancing loads in the grid:

1. The workload assigned to a centralized monitor increases as computers are added to the computing grid. A decentralized approach can alleviate the computing load on monitors. In this approach, each individual computer, or cluster of computers, will do some computation.
2. A centralized monitor makes the system susceptible to single-point failures. Distributing load-balancing and resource allocation tasks to individual computers will increase system robustness.
3. Individual computing nodes are naturally aware of their own workloads. As a result, the decentralized paradigm can achieve application-level resource management with significantly less communication overhead than a centralized monitor.
4. A decentralized system uses peer-to-peer networking to scale communication as the system grows, whereas a centralized monitor has to communicate with an increasing number of nodes.
5. A decentralized system is more robust to single node disruptions and failures, whether malicious or benign.
6. A decentralized system may be able to better respond to fluctuations in process requirements, e.g., in a scenario where the scheduler has to "forget" past process requirements and completely rebuild new clusters after servicing one process, i.e., there is no locality in process requirements.

4 Multi-agent Systems

Multi-agent systems use distributed agents to either model or solve a problem. An agent is an entity which matches some real-world object. It could be a biological cell, a virus particle, an ant, or in our case an individual computer. A computer program encodes simple rules or behaviors for interacting with other agents. The agents move about in space and interact with other agents in their neighborhood according to the encoded rules. Thus the behavior of low-level entities is specified and high-level behaviors evolve as simulation time progresses. Multi-agent systems emphasize local interactions based on first principles, and these interactions give rise to the complex high-level *emergent properties* of interest. Such systems have been used to model biological phenomenon such as the human immune system [16], as well as solve real-world problems like communication between distributed radar transmitters [13] and efficient resource collection in swarms of foraging robots [11, 12, 14, 15].

There is no centralized dispatcher to facilitate the formation and dissociation of clusters in the proposed dRAP algorithm. Instead, the algorithm relies on the *self-emergent* properties of a multi-agent system. A *multi-agent* or *agent-based system* is an architecture in which the global properties of the system emerge from local interactions.

The concept of a decentralized system presents a powerful counterpoint to the more common centralized control model often seen in business, government, and military organizations. Decentralization provides a number of important advantages over closed systems, such as robustness, adaptability, flexibility, innovation, and distributed intelligence. The key to this compelling architecture is the impressive ability of a decentralized system to react, mutate, or grow in response to challenging situations.

In any such decentralized system, the agent represents the base unit of computing power for the system. It behaves according to very simple rules. At each unit of model time (or *time step*), the agent senses its immediate local environment and takes actions based on its encoded rules. One rule might instruct the agent to divide if the number of neighbors is greater than 3, while another would cause it to die and be removed from the simulation if the number of neighbors is less than 2. These two examples are rules in the "Game of Life" [10], a paradigmatic system where complex patterns arise from local interactions and simple rules.

If we recast each agent's local sensing functionality as a peer-to-peer communication protocol with other nearby agents, then we can define a new set of rules for each agent that induces actions based on the state of these other, neighboring agents. Using this localized communication scheme, such rule-action pairs can be viewed as instructions for individual agents that produce *decentralized computation* across the system. There is no centralized monitor and yet this system is capable of performing complex computations. In fact, the computational power of such a system of distributed agents acting on simple rules has been proven to be Turing-complete [8].

We use such an agent-based system to dynamically create and dissociate clusters based on the resource requirements of each process. A snapshot of this system is shown in Figs. 1 and 2.

5 Software Platform

For this project we utilize the multi-agent simulation toolkit MASON [18]. MASON consists of a fairly small and portable set of Java library files that provide for design of both model (the "algorithm" component) and visualization (the "graphical user interface" component).

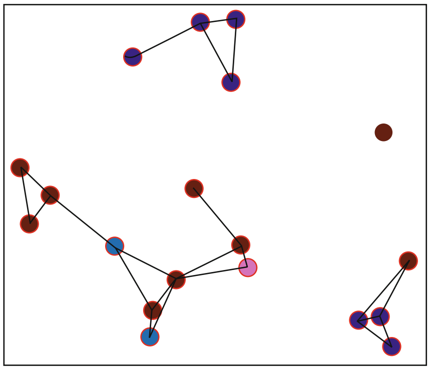
Fig. 1 Agents in large clusters; 1 free agent

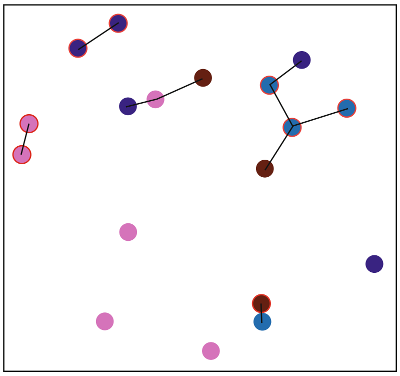
Fig. 2 Agents in several smaller clusters

The agent, the base component of computation in MASON (as in any multi-agent system), is coded in the familiar object-oriented programming format: the class "Agent" that contains all generalized methods and parameters needed for the object "agent" that is simply an instantiation of the Agent class. Following this format, each instantiated agent may contain a unique set of parameters, thereby allowing for minor variation in the replicated objects.

Agents are allowed to make decisions (and even communicate with one another) in a randomized batch lock-step. That is, the MASON scheduler moves through the (randomized) queue of all agents at each time step of the simulation. Scheduling of agents continues as long as the simulation itself is running, although the user may interrupt at any point by pausing or stopping the model.

MASON in particular was selected because of its all-in-one toolkit approach, making multi-agent simulation much easier than if done from scratch, as well as the authors' familiarity and experience with the MASON system.

6 dRAP Algorithm

The distributed resource allocation protocol (dRAP) is described below and some intended optimizations are suggested for future work. An agent in our system is simply a computer. Each agent has a vector containing the time remaining to finish executing its current process (time_{rem}) and the number of CPUs in its current cluster ($\text{CPU}_{\text{cluster}}$). Each agent (or node) is guaranteed to be in exactly 1 of 4 modes (or states) during the simulation:

Mode 1: An agent/node that is currently not part of a cluster and has no task assigned to it

1. The agent scans the queue Q, considers the resource requirements CPU_{req} of unallocated tasks, and takes on the task which minimizes the equation $|\text{CPU}_{\text{req}} - 1|$.

Mode 2: An agent/node that is currently not part of a cluster and has a task assigned to it

1. The agent continues executing the task and updates its information vector (time_{rem}, $\text{CPU}_{\text{cluster}}$).
2. If the task requirements are not completely satisfied (i.e., if $\text{CPU}_{\text{req}} > 1$), the agent will query its neighbors and attempt to form a cluster such that $\text{CPU}_{\text{req}} = \text{CPU}_{\text{cluster}}$.
3. When the agent finishes executing the task, it returns to **Mode 1**.

Mode 3: An agent/node that is currently part of a cluster and has no task assigned to it

1. The agent scans the queue Q, considers the unallocated tasks, and takes on the task which minimizes the equation $|\text{CPU}_{\text{req}} - \text{CPU}_{\text{cluster}}|$.

Mode 4: An agent/node that is currently part of a cluster and has a task assigned to it

1. The agent continues executing the task and updates its information vector (time_{rem}, $\text{CPU}_{\text{cluster}}$).
2. When the task completes, the agent dissociates from the cluster and returns to **Mode 1**.

A key feature of our algorithm is that nodes query their neighbors (other nodes that are close to them physically) in order to form clusters. This has the effect of reducing latency and communication costs. One optimization to consider would be to delay cluster dissociation in **Mode 4**. This would lead to *learning* or *memory* in the system where the scheduler would be able to remember past process requirements.

7 Analysis of Queue Cost

The dRAP algorithm requires a traversal through the global task queue in **Mode 1** and **Mode 3**. The algorithmic complexity is given by $\sum (n-i)m = O(n^2 m)$ where $m =$ the number of tasks in the global task queue, and $n =$ the average number of clusters. At a given timestep, the worst case can be approximated as $O(nm)$.

8 Optimizations Inspired by the Immune System

The immune system is able to find rare spatially localized pathogens and eliminate them in a timely manner [5, 6]. Similar to how in our system clusters of computers find processes, the immune system uses specialized cells to find pathogens in anatomical regions called lymph nodes. In previous work we showed how a sub-modular arrangement of lymph nodes could lead to fast elimination of pathogens in the immune system and also faster search for solutions in immune inspired distributed systems of computers [5, 6, 19]. Let an artificial lymph node be composed of a number of clusters and a process queue. Also let there be a number of such artificial lymph nodes that have the capability of communicating with each other. An "artificial lymph node" is supposed to be a computer in charge of a number of clusters. This computer will store the process queue and also will have some memory and CPU to communicate with other "lymph nodes."

We are interested in making the system sub-modular so that we can minimize the total time to find a cluster. There is a tradeoff between the local cost and the global cost; the local cost is $O(n^2)$ and the global cost $O(N/n)$. The total cost of traversing through the queue in a lymph node and the cost of communicating with other lymph nodes can be summed up as:

$$t_{\text{total}} = t_{\text{local}} + t_{\text{global}} \tag{1}$$

$$t_{\text{total}} = O(n^2) + O(N/n) \tag{2}$$

where n is the number of clusters in a single lymph node and N is the total number of clusters in the complete system. We assume that the global cost of finding another cluster in another lymph node that can service some process requirement is proportional to the number of lymph nodes (where N/n is the number of lymph nodes in the system).

Minimizing the total time cost, we get $2n - N/n^2 = 0$

$$n = O(N^{1/3}) \tag{3}$$

This implies that in larger systems (more computers, more clusters, and more lymph nodes), the number of clusters within a single lymph node should grow larger but only sub-linearly in the number of total clusters in the system. This would balance local costs of queue traversal and global costs of finding another lymph node with another cluster that can service the process. The key point here is that the number of clusters in a lymph node should scale sub-linearly with the size of the whole system, i.e., if a system of networked artificial lymph nodes were to grow a 1000 times bigger (1000 times more clusters), then the number of clusters within a lymph node need only increase by a factor of 10. Such sub-modular systems inspired by the immune system have been proposed previously for mobile ad hoc networks, control of mobile robots, intrusion detection systems, and peer-to-peer networks [5, 6, 19].

More generally, if the local and global communication costs scale with exponents α and β, we have

$$t_{\text{total}} = O(n^{\alpha}) + O(N^{\gamma}/n^{\beta}) \tag{4}$$

Minimizing the expression with respect to N, we get

$$n = O\left(N^{\frac{\gamma}{\alpha+\beta}}\right) \tag{5}$$

1. If $\gamma < \alpha + \beta$ we have sub-linear scaling.
2. If $\gamma > \alpha + \beta$ we have super-linear scaling.
3. If $\gamma = \alpha + \beta$ we have linear scaling.
4. If $\gamma/(\alpha + \beta) = 0$ we have no scaling (constant).
5. If $\gamma/(\alpha + \beta) < 0$ we have negative scaling.

9 Experiments

We conduct several experiments that compare our dRAP algorithm to a null model, i.e., a first-in first-out (FIFO) scheduling system. Additionally, we measure the effective computational complexity of queue traversals and examine the scaling properties of our system by varying the number of nodes and measuring the effect on performance. We define two timing metrics on which our system performance will be judged: T_{complete} is the time required to complete all tasks in the queue, and T_{wait} is the average wait time for a task added to the queue. Unless otherwise noted, system parameters are defined as such: number of nodes $=$ 100, number of tasks $=$ 1000, tasks are randomly selected from a normal distribution s.t. CPU_{req} varies from 1 to 5, with initial time_{rem} varying from 25 to 125 in increments of 25. That is, a task t_i with $\text{CPU}_{\text{req}} = 1$ has an initial $\text{time}_{\text{rem}} = 25$, and a task t_j with $\text{CPU}_{\text{req}} = 5$ has an initial $\text{time}_{\text{rem}} = 125$. All averages are computed across ten trials.

Table 1 Average timing comparison of dRAP and FIFO scheduling algorithms with 95 % confidence intervals

	T_{complete}	T_{wait}
dRAP	845.60 (861.94, 829.26)	342.54 (349.30,335.78)
FIFO	1071.20 (1088.99, 1053.41)	475.31 (485.79,464.82)

Table 2 Average cluster utilization of dRAP and FIFO scheduling algorithms with 95 % confidence intervals

	μ_{cluster}
dRAP	100 %
FIFO	56 % (54 %,58 %)

9.1 Comparison to Null Model

Here we present three separate experiments which compare the dRAP and FIFO algorithms. The first is a simple timing comparison that looks at T_{complete} and T_{wait} for each case. Values are presented in Table 1 (including 95 % confidence intervals).

We observe an approximate 20 % reduction in T_{complete} and an approximate 25 % reduction in T_{wait} when comparing dRAP to FIFO.

Our second experiment comparing dRAP and FIFO involves average cluster utilization. Because dRAP assigns tasks s.t. $\text{CPU}_{\text{cluster}} == \text{CPU}_{\text{req}}$, this ensures that all nodes in the cluster will be utilized. However, the FIFO scheduling system hands out tasks to the first available cluster, meaning it allows for the possibility that $\text{CPU}_{\text{cluster}} > \text{CPU}_{\text{req}}$. For example, a task with $\text{CPU}_{\text{req}} = 2$ that is assigned to a cluster with $\text{CPU}_{\text{cluster}} = 5$ will leave three unused nodes. Thus, we present an analysis of cluster utilization using the metric in Eq. (6):

$$\mu_{\text{cluster}} = \frac{\text{CPU}_{\text{req}}}{\text{CPU}_{\text{cluster}}} \tag{6}$$

If $\text{CPU}_{\text{cluster}} \leq \text{CPU}_{\text{req}}$, we simply set $\mu_{\text{cluster}} = 1$. Values are presented as percentages in Table 2 (note that dRAP's μ_{cluster} is always 100 % by definition).

Finally, our third experiment is designed to measure global node utilization over the time of the simulation. Here we simply document the number of nodes that do computation on a given timestep and normalize by the total number of nodes in the system. Results are displayed in Fig. 3 (taken from a single simulation run).

We observe that the dRAP algorithm utilizes approximately 90–95 % of the nodes for the majority of the simulation, while FIFO utilizes approximately 70–75 %.

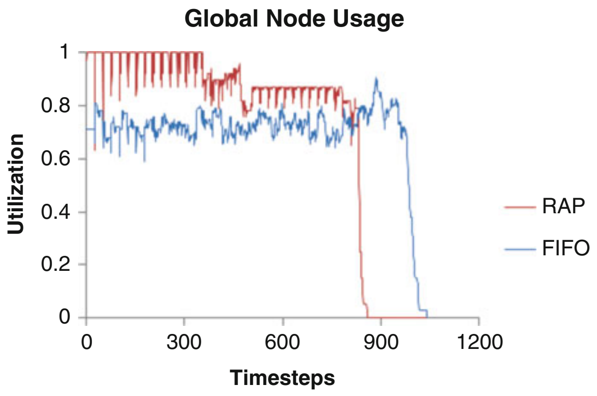

Fig. 3 Global utilization of nodes throughout simulation

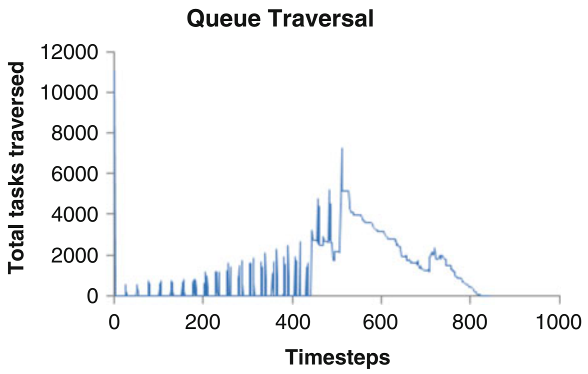

Fig. 4 Total task traversals by all clusters per timestep

9.2 Effective Complexity

For this experiment, we estimate the "effective" computational complexity of the dRAP algorithm. That is, in comparison to the qualitative $O(nm)$ worst case runtime per timestep, we are interested in how much of the task queue must be traversed in order to properly fit the $\text{CPU}_{\text{cluster}} == \text{CPU}_{\text{req}}$ requirement. Total tasks traversed per timestep from one selected simulation run are presented in Fig. 4. Note that the initial traversal (timestep "0"), although difficult to see, is approximately 11,000.

"Worse case" here, as addressed above, is $O(nm)$, or 100,000 tasks traversed per timestep if n = number of clusters = number of nodes = 100 and m = number of tasks = 1000. From this plot (plus additional runs not included here), we can conclude that effective computational complexity is no more than approximately 10 % of the worst case runtime $O(nm)$.

9.3 Scaling

For our last experiment, we are interested in collecting information on the scaling ability of our algorithm. Our goal in this test is to increase the number of nodes (in intervals of 50), while also maintaining an equal number of neighbors for each node. That is, we ensure that the neighborhood size parameter defined in the simulation scales inversely with the number of nodes s.t. a given node has approximately the same number of neighbors regardless of the total nodes in the system. Results are presented for our two timing metrics: T_{complete} scaling in Fig. 5 and T_{wait} scaling in Fig. 6. Data in both figures are $\log_2$-transformed in order to correlate doubling of nodes with halving of the timing metrics.

We note a near-perfect scaling for both timing metrics, as shown in the fitted power law equations inset into each plot (Figs. 5 and 6). Note that the T_{wait} exponent above 1 is most likely a result of inexact tuning of the neighborhood size with increasing nodes, and this issue will rectified in future work.

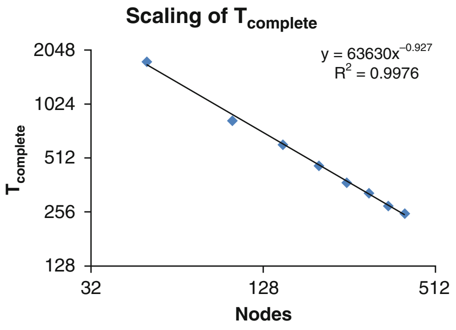

Fig. 5 Scaling of T_{complete}

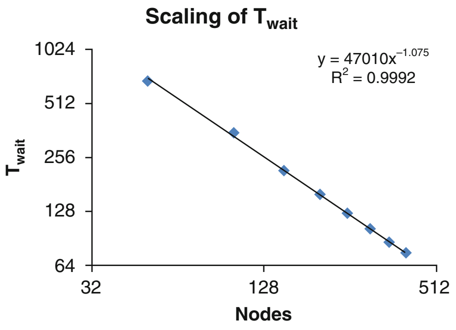

Fig. 6 Scaling of T_{wait}

10 Related Work

Resource allocation for grid computing is an active area of research. For example, SLURM [20] is a configurable Linux utility for cluster allocation that uses static allocation of nodes to clusters, called partitions, in contrast to the dynamic cluster allocation presented in this paper. LSF [21] is another proprietary cluster management facility; however, details of its allocation algorithm are not publicly available.

11 Conclusions and Future Work

In this paper we have presented an algorithm for allocating, scheduling, and load-balancing processes on a massively distributed system. This is very relevant to current research in operating systems, especially with a trend of moving computation tasks onto inexpensive, distributed hardware. The proposed decentralized algorithm draws inspiration from biology, adaptively creating, and dissociating clusters from nodes to match task demand. Decentralization enables scalability, robustness, alleviation of computing load on monitor, better response and adaptability to process queue fluctuations, and learning about process requirements. The dRAP algorithm outperforms a FIFO scheduling algorithm on time to complete all tasks, average waiting time, and CPU utilization. The scheduling is also shown to be robust to a malicious adversary that might permute the order of the tasks such that high demand tasks would be queued first followed by low demand tasks. A key feature of our procedure is that nodes communicate with neighboring computers in order to dynamically form clusters. Hence our algorithm also holds promise in areas where it is advantageous to communicate with immediate neighbors due to network latency, e.g., Google MapReduce uses a locality optimization to reduce latency due to network communication [9]. The comparison of this algorithm to other scheduling algorithms like Shortest Remaining Time First (SRTF) on other metrics like response time, as well as collection of data on the exact distribution of process demand in a queue in a real-world scenario, will be the subject of future investigation.

12 Scientific Validation

This paper has been unanimously validated in a collaborative review mode with the following reviewers:

- Cyrille Bertelle, from Université du Havre (France)
- Pierre Collet, from Strasbourg University
- Carlos Jaime Barrios Hernandez.

Acknowledgements We thank Dr. Dorian Arnold for fruitful discussions.

References

1. Anderson D, Cobb J, Korpela E, Lebofsky M, Werthimer D (2002) SETI@ home: an experiment in public-resource computing. Commun ACM 45(11):61
2. Banerjee S (2009) An immune system inspired approach to automated program verification. arXiv preprint arXiv:0905.2649. http://arxiv.org/abs/0905.2649
3. Banerjee S (2013) Scaling in the immune system. Ph.D. thesis, University of New, Mexico
4. Banerjee S, Moses M (2009) A hybrid agent based and differential equation model of body size effects on pathogen replication and immune system response. In: Timmis J (ed) The 8th international conference on artificial immune systems (ICARIS). Lecture notes in computer science. Springer, Berlin, pp 14–18. http://www.springerlink.com/content/b786g874642q2j37/
5. Banerjee S, Moses M (2010) Modular radar: an immune system inspired search and response strategy for distributed systems. In: Artificial immune systems. Springer, Berlin, pp 116–129
6. Banerjee S, Moses M (2010) Scale invariance of immune system response rates and times: perspectives on immune system architecture and implications for artificial immune systems. Swarm Intell 4(4):301–318. http://www.springerlink.com/content/w67714j72448633l/
7. Banerjee S, Levin D, Moses M, Koster F, Forrest S (2011) The value of inflammatory signals in adaptive immune responses. In: Artificial immune systems. Springer, Berlin, pp 1–14. http://www.springerlink.com/index/U634HJ83W62W5383.pdf
8. Cook M (2004) Universality in elementary cellular automata. Complex Syst 15(1):1–40
9. Dean J, Sanjay G (2008) MapReduce: simplified data processing on large clusters. Commun ACM 51(1):107–113
10. Gardner M (1970) Mathematical games: the fantastic combinations of John Conway's new solitaire game 'Life'. Sci Am 223(4):120–123
11. Hecker JP, Moses ME (2013) An evolutionary approach for robust adaptation of robot behavior to sensor error. In: Proceedings of the 15th annual conference companion on genetic and evolutionary computation (GECCO '13 companion). ACM, New York, NY, pp 1437–1444. http://doi.acm.org/10.1145/2464576.2482724
12. Hecker JP, Moses ME (2015) Beyond pheromones: evolving error-tolerant, flexible, and scalable ant-inspired robot swarms. Swarm Intell 9(1):43–70
13. Hecker J, Wu A, Herweg J, Sciortino J Jr (2008) Team-based resource allocation using a decentralized social decision-making paradigm. In: Proceedings of SPIE, vol 6964, p 696409
14. Hecker JP, Letendre K, Stolleis K, Washington D, Moses ME (2012) Formica ex machina: ant swarm foraging from physical to virtual and back again. In: Swarm intelligence: 8th international conference, ANTS 2012. Springer, Berlin, pp 252–259
15. Hecker JP, Stolleis K, Swenson B, Letendre K, Moses ME (2013) Evolving error tolerance in biologically-inspired iAnt robots. In: Proceedings of the twelfth European conference on the synthesis and simulation of living systems (Advances in artificial life, ECAL 2013). MIT Press, Cambridge, MA, pp 1025–1032
16. Jacob C, Litorco J, Lee L (2004) Immunity through swarms: agent-based simulations of the human immune system. In: Artificial immune systems. Lecture notes in computer science. Springer, Berlin, pp 400–412
17. Krawczyk S, Bubendorfer K (2008) Grid resource allocation: allocation mechanisms and utilisation patterns. In: Proceedings of the sixth Australasian workshop on grid computing and e-research-Volume 82. Australian Computer Society Inc, Darlinghurst, pp 73–81
18. Luke S, Cioffi-Revilla C, Panait L, Sullivan K (2004) Mason: a new multi-agent simulation toolkit. In: Proceedings of the 2004 SwarmFest workshop, vol 8

19. Moses M, Banerjee S (2011) Biologically inspired design principles for scalable, robust, adaptive, decentralized search and automated response (radar). In: 2011 IEEE symposium on artificial life (ALIFE), pp 30–37
20. Yoo A, Jette M, Grondona M (2003) SLURM: simple linux utility for resource management. In: Job scheduling strategies for parallel processing. Lecture notes in computer science. Springer, Berlin, pp 44–60
21. Zhou S (1992) LSF: load sharing in large-scale heterogeneous distributed systems. In: Proceedings of workshop on cluster computing, pp 1995–1996