Available online at www.sciencedirect.com

SCIENCE DIRECT®

Future Generation Computer Systems 22 (2006) 676–687

FGCS FUTURE GENERATION COMPUTER SYSTEMS

www.elsevier.com/locate/fgcs

Task distribution with a random overlay network

Ladislau Bölöni*, Damla Turgut, Dan C. Marinescu

School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, FL 32816, United States

Received 1 September 2005; received in revised form 8 November 2005; accepted 10 November 2005
Available online 24 January 2006

Abstract

We consider a model where commodity service providers are offering commodity computational services to a set of customers. We provide a solution for the efficient distribution of tasks by forwarding the service requests on an overlay network comprised on random cycles. We introduce algorithms for the creation, maintenance and repair of the overlay network. We discuss two algorithms, random wandering and weighted stochastic forwarding, for the allocation of the tasks to providers. Both approaches are highly scalable because the algorithms use only limited local information.

As we are designing our approach for use in a commercial setting, there is a requirement that the tasks, being a source of profits, be allocated fairly to the providers. We investigate the fairness of the algorithms and show that adding a random pre-walk can improve the fairness.

Through a simulation study we show that the approach provides efficient task allocation on networks loaded up to 95% of their capacity.
Published by Elsevier B.V.

Keywords: Grid computing; Commodity computers; Routing

1. Introduction and motivation

Grand challenge computing was traditionally considered the domain of expensive, specialized hardware. In recent years, however, a number of "public computing" initiatives have exploited the abundance of commodity resources for solving highly parallel applications. Examples are SETI@Home [22], Folding@Home [19], the cryptographic challenges sponsored by RSA Laboratories [21] or the Mersenne prime search [20]. The Berkeley Open Infrastructure for Network Computing (BOINC, [1,18]) proposes to provide a framework more general than the SETI@Home project, which can be shared by a number of projects following this pattern of interaction.

Public computing is concerned with *commodity tasks* with moderate processor and memory requirements. In addition, public computing introduces certain simplifying assumptions, which typically do not hold in other settings: there is a single client for all tasks, there are no hard deadlines, and as there are no financial transactions involved, the accounting, fairness and security issues are of secondary importance. For instance, in the SETI@Home system, computers are "rewarded" for executing a task, but they are not penalized for accepting a task for execution and then not executing it. Public computing relies on a "gift economy" of donated computing services. This model is only suitable for computational problems for which there is a perception of widespread benefits for the society.

We note that many high performance computing workflows contain both specialized and commodity tasks.[1] The optimal solution for the specialized tasks is to send and queue them at the appropriate specialized providers, for instance through a system such as Condor [14]. The commodity components in these workflows have a granularity similar to the subtasks of the SETI@Home or Mersenne prime search, but they do not share the other simplifying assumptions of those approaches. In particular, there are a large number of providers as well as consumers, the providers do not provide their services for free, the consumers are interested in the fast execution of the tasks, the tasks are not identical nor equivalent and there can be dependencies between the tasks.

* Corresponding author. Tel.: +1 407 823 2320; fax: +1 407 823 5835.
E-mail addresses: lboloni@cpe.ucf.edu (L. Bölöni), turgut@cpe.ucf.edu (D. Turgut), dcm@cs.ucf.edu (D.C. Marinescu).

[1] For instance, the authors' experience with computational virology workflows shows that approximately 75% of the workflow contains image processing tasks on moderate datasets which can be written as commodity tasks [7].

0167-739X/$ - see front matter Published by Elsevier B.V.
doi:10.1016/j.future.2005.11.004

In this paper we propose and study an architecture which leads to the efficient execution of commodity tasks. The architecture strongly relies on the equivalence properties of commodity resources, but it takes into account the requirements of a commercial system. The problem we are trying to solve is that of efficient and fair task allocation. We need to pair the providers of the resources with the consumers who have tasks to be executed. We strive for an efficient utilization of the resources and minimization of the overhead. In addition we also need to consider the fairness towards the customers (such that every customer has the same chance of its task being executed) and towards the provider (such that every provider will be allocated similar amounts of profit-generating tasks).

The approach we are proposing is based on forwarding the task requests on a overlay network formed by n randomized cycles (the n-Cycle overlay). We introduce algorithms for the creation, maintenance and repair of the overlay network. We discuss two algorithms, random wandering and weighted stochastic forwarding, for the allocation of the tasks to providers. Both approaches are highly scalable because the algorithms use only limited local information.

The remainder of this paper is organized as follows. We introduce the n-Cycle overlay network, algorithms for building and maintaining it, and two distributed algorithms for task allocation in Section 2. Simulation results are presented in Section 3. We overview related work in Section 4 and conclude in Section 5.

2. The *n*-Cycle task distribution algorithm

We consider a market of computations and borrow concepts such as *commodity task* and *commodity provider* from economic market models. In this new market *providers* p carry out *tasks* t on behalf of clients c and receive a fee $cost(p, t)$ for their services.

Let us consider a set of tasks $T = \{t_1, t_2 \dots t_k \dots\}$. A set of providers $P = \{p_1, p_2, \dots\}$ are *commodity providers* for the tasks T if (a) any provider p_i can execute any task t_k and (b) the performance differences between the providers in respect to any task t_k are negligible. This means that if a client c wants to execute the task t_k, it has no reason to pick any of the providers over the other. Naturally, this leads to uniform pricing across the providers:

$$\forall p_i, p_j, t_k \quad cost(p_i, t_k) = cost(p_j, t_k).$$

A set of T consists of commodity tasks, if the revenue each task $t_k \in T$ brings to a provider is proportional with its execution time.[2] Thus, for a provider the tasks are indistinguishable, the provider has no reason to choose one task over another. This leads to a pricing model in which the value of execution time is constant across the providers

$$\forall t_k \, cost(p_j, t_k) = C \cdot exectime(t_k).$$

[2] See the definition of a commodity good: the value of a commodity is proportional with its quantity.

Note that we can have commodity providers, but differentiated tasks. We call a computational market a *commodity market* if it consists exclusively of the commodity providers and tasks.

This approach is similar to the one of network-centric applications such as SETI@Home or Folding@Home. The differences between our approach and one of these applications are: (i) the financial aspect, and (ii) the presence of multiple clients.

In general, shorter tasks are more likely to be considered as commodity tasks, as the differences in their performance are hidden by the overhead. On a commodity market, the completion time of a task is determined completely by its time to entering into execution. Thus it is critical that for a new task t, a provider which can take it into execution is found in an efficient way. Once the provider is found, the remainder is an archetypal remote execution problem, for which there are a variety of mature solutions.

This paper concentrates on the task allocation problem on a commodity market. We had to make several design decisions: centralized versus distributed algorithms, unicast or multicast communication. A centralized approach for task allocation is feasible for small and stable networks and for tasks with relatively long execution time. Yet, a centralized approach is not feasible for a large network of providers, which join and leave the network dynamically and for relatively short tasks. The approach we propose is inherently distributed, it does not use global information, and does not involve single-failure points.

We choose to use unicast communication even though popular resource allocation methods, such as the expanding circle search, are based upon multicast. Let us consider the way in which expanding circle search operates. The request is replicated and sent first to the immediate neighbors; if no appropriate provider is found, the request is sent out to a larger circle, and so on. This approach does a good job to minimize the time to find a provider, but has significant drawbacks. It requires the provider to temporally reserve the resource if a request is received. This reservation needs to be maintained until the client explicitly notifies the provider of its decision, or until a timeout occurs. This leads to long processing time. In contrast, with a unicast approach there is a single task request in the network; a positive response from a provider guarantees that the provider will be chosen for the execution of the task. The client is not motivated to seek multiple offers, given that in a commodity network all providers are indistinguishable. Of course, without the parallel communication implied by the multicast model, the task allocation usually takes longer. Our experiments show that the n-Cycle algorithms enjoy very good allocation time performance and prevents the waste of computational capacity by eliminating the time when the resource is reserved, but not used.

In the n-Cycle architecture, the task requests are forwarded on an overlay network, according to a forwarding policy. The forwarding stops when a free provider is found. The number of hops until the task is allocated is the principal performance criteria of the system. The two components of the architecture are (a) the creation and maintenance of the n-Cycle overlay and (b) the forwarding policy together with the maintenance

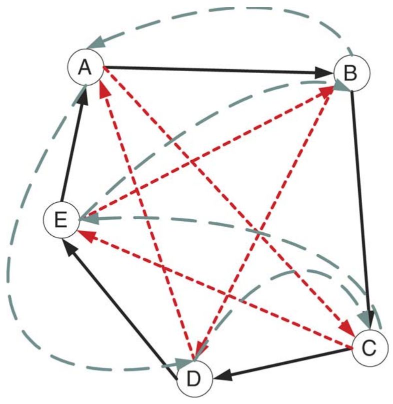

Fig. 1. A 3-Cycle overlay network on a network of 5 nodes. The first cycle is ABCDE (shown with continuous line), the second cycle is ACEBD (dotted line) and the third cycle is ADCEB (dashed line).

of the forwarding data. Our goal is to design efficient and fully distributed algorithms for both steps, which maximize the performance of the forwarding.

2.1. *Creation and maintenance of the overlay network*

The n-Cycle algorithm [4] creates an overlay network of directional links. For any link A $\rightarrow$ B, we forward the task from A to B, while status information travels from B to A. The links of the overlay network form n separate cycles connecting all the nodes in the grid. Such a cycle can be represented by a permutation of the nodes of the network, with all the nodes in the permutation being connected to the next, and the last one connected to the first node. The randomness of the cycles is an important part of the algorithm. For any n-Cycle overlay network, every individual node has n nodes "upstream" and n nodes "downstream". A node forwards tasks to nodes downstream and receives status updates from them. Similarly, the node receives tasks from the nodes upstream and forwards status updates to them. Fig. 1 shows a 3-Cycle forwarding mesh on a grid of 5 nodes.

Our design decisions are driven by the following considerations:

- To guarantee that an available provider can be found, independently of the insertion point of the request, we use *cycles*.
- To prevent re-forwarding of the request in the same direction, the cycles are **directional**.
- To increase the number of providers that can be reached from an entry point we allow **multiple cycles**.
- The individual cycles are **random** in relation to each other, to prevent *locality*; the requests are inserted at a point to be distributed as widely as possible in the network.

2.1.1. *Centralized network maintenance algorithms*

To maintain the overlay network, we need algorithms to create a network from a set of nodes, and to add or remove nodes from an existing network. A centralized implementation, with a global view of the network is straightforward.

If we have a list of N nodes, an n-Cycle network can be *created from scratch* by generating $n - 1$ random permutations of the nodes. The first cycle will follow the natural order of the list, the remaining $n - 1$ cycles will follow the order of the nodes in the permutations. For all cases, we close the cycle by connecting the last node to the first node.

Adding a new node to the existing overlay network can be accomplished by making n "cuts" at random locations in the existing cycles and "splicing" the node in the cycles at these locations. These random locations can be chosen by generating n random numbers between 1 and N.

Finally we can remove a node from the n-Cycle network, by "tying together" its uplink and downlink nodes on each of the individual cycles.

For all these algorithms, an additional consideration should be to assure that the n downstream nodes from any node are distinct. For the creation algorithms, we can check if this condition is satisfied at the addition of every new cycle, and either create a new permutation or fix the existing one by changing the position of the offending node. For the addition algorithm, if a certain cut leads to a duplicate downstream node, we create a new random cut, until all the downstream nodes are distinct. The problem is most difficult for the case of the removal of a node. If the removal of a node leads to its upstream nodes to have duplicate downstream nodes, we can fix this only by changing the position of the offending downstream node in the given cycle. Nevertheless, as $n \ll N$, the number of these occurrences are small and they can be ignored for large networks.

2.1.2. *Decentralized network maintenance algorithms*

As pointed out earlier, centralized algorithms are impractical for very large networks consisting of several million nodes. At the setup time it is reasonable to expect global knowledge about the set of nodes which want to participate in the mesh. However, overlay networks would normally start with a smaller number of nodes and will be extended over time; so most nodes will be added to the overlay using the dynamic addition algorithm. The algorithms for dynamically adding and removing nodes from the mesh are invoked many times during the lifecycle of the network, thus it is desirable that they operate without global information about the mesh.

The algorithm for removing the node requires only the upstream and downstream neighbors of the node to be removed—information which is readily available to the node. The centralized algorithm for adding a node requires global information about the mesh to determine the set of n random nodes ("cuts") where the new node will be inserted in the n cycles. To maintain the randomness of the network, the cuts have to be randomly chosen from the *complete set of nodes* W, with a uniform probability. Let us assume that the cuts are chosen only from a set $W' \subset W$. Then, if we add a set of nodes $E = \{e_1, e_2 \ldots e_n\}$ the resulting network assumes an hourglass shape, with W' being the bottleneck. The task forwarding algorithm will still work, but it will be unbalanced.

In the following, we present two fully distributed algorithms for the insertion of a new node into the n-Cycle network.

The first algorithm guarantees that the node is connected through cuts which are selected from the full set of nodes W, but its complexity is linear in the size of the network. The second algorithm offers only a statistical expectation, but its complexity is logarithmic. Both algorithms exploit the properties of the n-Cycle network. For both algorithms we assume that we have an estimate of the size of the network N. Although this is global information, it is a much weaker requirement than having a snapshot of the network; moreover, overestimates are acceptable.

The first algorithm relies on forwarding a message on a single cycle of the n-Cycle network. As the cycle contains all the nodes, we can pick the random cuts by randomly selecting n numbers between 0 and N, and reaching the nodes by sending a message from node to node, in the direction of the uplinks.

The number of messages needed to add a single node is $O(N)$. This makes the number of messages for building a network through repeated additions $O(\frac{N(N-1)}{2})$. This process is naturally parallelized over all the nodes; in average, every node needs to forward one message whenever a new node is added to the network. However, the time for adding a single node is still unacceptably high.

We now introduce a new algorithm which relies on the randomness in the existing n-Cycle network to build future randomness. The approach, described in Algorithm 2, performs a random walk of k steps by randomly selecting one of the n downlink nodes—basically jumping to a random cycle in each step.

Algorithm 1 Adding a node A into the n-Cycle overlay network using a random walk on a cycle

```
When node A to be inserted into overlay network W
    generate n random numbers $c_i \in \{0 \ldots N\}$
    sort them in increasing order
    create a message $M = \{0, \{c_1, c_2 \ldots c_n\}, A\}$
    send it to downlink 0
When node B receives a message $M = \{i, \{c_k \ldots c_m\}, A\}$
    If $i == c_k$
        make B's $k$-th uplink the $k$-th uplink of A
        make A B's $k$-th uplink
        If $k < n$
            create new message
                $M' = \{i + 1, \{c_{k+1} \ldots c_m\}, A\}$
            send it to downlink 0
    Else
        create new message $M' = \{i + 1, \{c_k \ldots c_m\}, A\}$
        send it to downlink 0
```

Let us now consider the probability that a node n is not on the list of the nodes reachable by $k = \lceil \log_n(N) \rceil + s$. We assume that there is an even probability for any node to be at the end of any k step path, this probability is $1/N$, with the probability for a node not to be at the end of the path being $1 - 1/N$. There are n^k such paths, so the probability of the node not terminating any of them is $(1-1/N)^{n^k} = (1-1/N)^{n^{\log_n(N)+s}} = (1-1/N)^{Nn^s}$. However, $\lim_{N\to\infty}(1-1/N)^N = 1/\text{e}$, therefore, the probability that a given node will not be reachable in $\lceil \log_n(N) \rceil + s$ steps is approximately $(\frac{1}{\text{e}})^{n^s}$ for large N.

For instance, probability for a node not be reachable in a 10,000 node 5-Cycle network are 0.3679 for $s = 0$, 0.0067 for $s = 1$, 1.3887×10^{-11} for $s = 2$ and 5.1656×10^{-55} for $s = 3$. Thus, we conclude that performing $\lceil \log_n(N) \rceil + 3$ steps offers statistically sufficient guarantees that the new node will be inserted at a random position chosen from the entire set of existing nodes.

Algorithm 2 Adding a node into the n-Cycle overlay network using a random walk with cycle jumping

```
When node A to be inserted into overlay network W
    For $i = 0$ to $n$
        generate $k = \lceil \log(N) \rceil + 3$ random numbers
        $c_m \in \{0 \ldots n - 1\}$
        create a message $M = \{i, \{c_1 \ldots c_k\}, A\}$
        send it to downlink $c_1$
When node B receives a message
    $M = \{i, \{c_j \ldots c_k\}, A\}$
    If $j == k$
        make B's $i$-th uplink the $i$-th uplink of A
        make A B's $i$-th uplink
    Else
        create new message $M' = \{i, \{c_{j+1} \ldots c_k\}, A\}$
        send it to downlink $c_j$
```

An additional requirement would be that every node have the same probability of being picked, independently from the starting point. One way to measure this probability is to count the number of paths leading from the insertion point to the given random point. Due to the structure of the network, the probabilities are not uniform, but empirical evidence (see the results in Sections 3.4 and 3.5) shows that the probabilities tend to equalize with the increasing number of hops. An analytic study of this problem is outside the scope of this paper.

For this algorithm, the time complexity is much smaller, $O(n\log_n(N))$, which leads to the complexity of the complete network building $O(nN\log_n(N))$.

2.1.3. Robustness and recovery from failures

In any large distributed system, node failures are a fact of life. If a node fails unexpectedly, it will not be able to correctly remove itself from the n-Cycle network. This disrupts the structure of the overlay network. In the following, we investigate the consequences of such a disruption, as well as preventive and repair measures.

The robustness of an n-Cycle network has two aspects: (a) the connectivity of the network and (b) the integrity of the cycles. We will see that the n-Cycle network is structurally very robust from the point of view of connectivity, but relatively vulnerable from the point of view of the integrity of the cycles.

Let us first consider what is happening in the case of a node failure. The n upstream nodes will lose one of their downstream nodes, remaining with only $n - 1$ nodes to forward tasks to. Similarly, the n downstream nodes will remain with only $n - 1$ nodes to receive tasks from. Assuming that the connections between these nodes are based on a guaranteed delivery transport layer, the loss of the connection will be discovered at the latest at the first attempted transmission. In addition to this, all the n cycles are now interrupted. We note that the loss of a node does not affect in a significant way the behavior or performance of the network. Nevertheless, as the failures are additive, the overlay network needs to be protected from continuing deterioration and eventually, repaired.

A simple way to add some level of redundancy in the n-Cycle network is by adding a *bypass*, maintaining in the node information links to the second, third and so on nodes in the n cycles both in the upstream and in the downstream direction. Normally a node needs to maintain information about $2n$ nodes (n upstream and n downstream). With a single bypass, this number will become $4n$, while with a double bypass, $6n$. This is still a relatively modest number of nodes to maintain (10, 20 and 30 nodes for a 5-cycle network).

Let us now consider how the connectivity of the network is affected by failures in the nodes. In order for a single node to become disconnected from the n-Cycle network, a set of 10, 20 or 30 exactly selected nodes need to fail simultaneously (before the node has a chance to repair any connection). As these nodes are randomly scattered over the network, there is no single failure (such a disconnected subnetwork) which can trigger such a simultaneous failure of nodes. Even if it happens, the disconnection of a single node is a minor problem compared with the failure of 30 nodes. If this happens, however, the node can simply reinsert itself into the network.

A more major problem would happen if an n-Cycle network would separate into two networks of approximately equal size. However, this is made impossible by the structure of the network. Let us consider the hypothetical case when a network on N nodes is split into two networks of equal size A and B by a series of failures. To have this, we need all the connections to remain inside the two disjoint networks. The random structure of the network guarantees that for any node, any given upstream or downstream node has an equal probability to be in the subset A and B. The probability of a node to have at least one of its upstream or downstream nodes in the opposite subset is the very high $1 - 2^{-30}$. All these types of nodes need to fail to prevent connectivity between A and B. In conclusion, short of a major failure simultaneously shutting most of the network, the n-Cycle network will maintain connectivity.

Let us now consider the repair of the n-Cycle network. While the loss of connectivity in the network is statistically improbable and requires a massive number of failed nodes, the integrity of the cycles can be lost with only a small number of failed nodes. In a network without a bypass, the failure of a single node can lead to a disconnected cycle. If the nodes maintain a single or double bypass, the failure of one or two nodes can be quickly and simply corrected, by simply skipping the failed node. The failure of 2 or 3 nodes, for single or double bypasses, however, still leads to the disconnection of the cycle. The simultaneous failure of 3 nodes is a very rare occurrence, thus we can afford to use relatively expensive algorithms.

In the following, we discuss an algorithm for repairing a cycle. Let us consider the case when a series of failures lead to the situation that cycle i is disconnected in k different locations. This means that we will have k end-nodes (which are missing the downstream node on the cycle i) and k start-nodes (which are missing the upstream node on cycle i). Our goal is to connect every start-node to an end-node, not necessarily in the order in which they were in the original network. This can be achieved with a *double expanding cycle search* starting from both the start and the stop node. The disconnected node will send to its remaining $2n - 1$ connections a start- or end-point advertisement message indicating its disconnected cycle i and its end or start status. If the opposite cycle was not found, in the next cycle the message will be sent to all nodes two hops away and so on. Whenever a node receives both types of messages (start and end) for the same cycle, it notifies the two nodes which reconnect the cycle.

Let us analyze the computational complexity of this algorithm. Let us consider that we have N nodes and both sides had broadcasted to K nodes. The probability that any given node will have a start/end advertisement is $p_{\text{start}} = p_{\text{end}} = K/N$. The probability that there is a node which has both advertisements is:

$$p_{\text{match}} = 1 - \left(1 - \frac{K^2}{N^2}\right)^N .$$

This probability becomes very close to 1 for relatively small K's compared to N. For instance, for $K = 90$ and $N = 10{,}000$, we get $p_{\text{match}} = 0.9997063092544672$. However, $K = 90$ represents only two steps, each from the disconnected start-node and end-node (as these nodes have only 9 active neighbors). Thus, using this algorithm, the cycles can be repaired quickly and with limited cost in terms of messages or network bandwidth.

2.2. Distributing tasks on the overlay network

Once the overlay network is built, we can use it to distribute tasks to the providers. There are two main classes of forwarding algorithms: *stateless algorithms* do not use any information about the load of the individual nodes, relying exclusively on the structure of the overlay network and rules for forwarding. In contrast, *stateful algorithms* maintain knowledge about the current state of the network, and use this information in the task forwarding. Naturally, we expect better performance from the stateful algorithms, but this performance improvement comes at the cost of maintaining the load information. The challenge is to balance the cost and benefits of the routing information. We discuss two forwarding algorithms: the stateless *random wandering* algorithm and the stateful *weighted stochastic forwarding* algorithm.

2.2.1. Random walk (RW)

The *Random walk* (RW) task forwarding algorithm is based on the following rules: if current host is free, accept the incoming task. If not, then forward with equal probability the task to any downstream node. The number of hops until the task is accepted depends on the average load of the network l, calculated as the number of busy nodes over the total number of nodes N. In a first approximation, for any number of hops h, the probability that a node will be allocated in less than h hops is $(1 - l)^h$. Although this approach leads to satisfactory average values as long as the load is not getting close to 100%, the maximum values can be (potentially) indefinitely long.

2.2.2. Weighted stochastic forwarding

The *weighted stochastic forwarding* (WSF) uses information collected from downstream nodes in the forwarding decision. Every node maintains its weight w which represents the desirability of the node as a forwarding target for a task. The weight w is composed in equal parts from (a) the ability of the node to receive a task for execution and (b) the weights of the nodes downstream from the node. The new candidate weight is computed as follows:

$$w' = \frac{w_{\text{self}}}{2} + \frac{\sum_{k=1}^{n} w_k}{2n}. \tag{1}$$

The candidate weight w' will replace the weight w only if $|w - w'| > 1/2n$. If the weight is updated, it is propagated to the upstream nodes using *weight update messages*. This heuristic limits the propagation of the weight update messages. The acceptance or the termination of a task triggers a change of 0.5 in the weight of the current node, but only $1/2n$ in the n immediate upstream nodes, and $(1/2n)^k$ in nodes k steps away in the upstream direction. The effect of a single task allocation or termination is normally limited to n weight update messages to the immediate upstream nodes (although multiple downstream allocations can trigger further messages through the addition of received weights).

At any given node, a task is either taken into execution (if the node is free), or forwarded to one of the downstream nodes with a probability proportional with their weights. A time-to-live (TTL) based approach limits the number of hops a task can be forwarded, the TTL being decreased at every hop. When TTL reaches 0, the request is sent back to the originating client as rejected. The complete approach is presented in Algorithm 3.

Algorithm 3 Weighted stochastic forwarding

```
Initially
    $w_{\text{self}} = 1$
    $w_i = 1, \ \forall i \in \{1 \dots n\}$
When task t received by node N
    If $w_{\text{self}} == 1$
        take t into execution
        $w_{\text{self}} = 0$
        calculate new weight $w'$
        If $|w - w'| > \frac{1}{2n}$
            $w = w'$
            send the new weight $w = w'$ to all upstream nodes
    Else
        If TTL of the request is 0
            send the task t back to the originator as rejected
        Else
            decrease the TTL of the request with 1
            forward t to downstream node i with probability $\frac{w_i}{\sum_{k=1}^{n} w_k}$
When execution of task t is terminated at node N
    $w_{\text{self}} = 1$
    calculate new weight $w'$
    send the new weight $w = w'$ to all upstream nodes
When weight $w'_i$ received from $i$-th downstream node
    $w_i = w'_i$
    calculate new weight $w'$
    If $|w - w'| > \frac{1}{2n}$
        $w = w'$
        send the new weight $w$ to all upstream nodes
```

2.3. The fairness of the task distribution algorithms

The architecture presented in this paper is based on the voluntary cooperation of resource providers and customers. This cooperation can be assured only if the algorithm is viewed as "fair" by the participants.[3] In this section, we consider the issues of fairness for this algorithm, and propose modifications which increase its fairness at the cost of slight reductions in efficiency.

There are two, largely independent viewpoints towards the fairness of the task allocation algorithm. From the point of view of the customers, fairness means that the servicing of a task does not depend on its customer. From a point of view of providers, fairness means that every available provider has an equal chance to service a given task. To assure the cooperation of customers and service providers the algorithm should treat both groups fairly.

2.3.1. Fair treatment of customers

A fair treatment of customers means that every task of every customer is treated equally: every task should have the same chance of being accepted or rejected. A less important factor is the number of hops the task needs to travel until it is accepted for processing. As long as the number of hops is small (on the order of 10–20), this factor is of little importance.

The task distribution algorithms do not label the requests based on the customer; therefore, the only possible source of unfairness towards the customers is the distribution of the insertion points.

For networks with a single insertion point, every task is treated identically, and its acceptance/rejection and hops until allocated depend only on the current state of the network. The same considerations apply to the case where the tasks are inserted at a random point in the network, provided that the insertion point is uniformly distributed and independent of the previous tasks.

Unfair treatment of customers can happen if there is a small number of fixed insertion points, with different rates of incoming tasks. Intuitively, every insertion point "fills up" the grid nodes in its vicinity. Tasks which arrive at insertion points which have a higher rate of incoming tasks have to travel farther until an available provider is found. The question is whether for networks with a very high load this can make the difference between a request being accepted or rejected. Let us recall that after $\lceil \log_n(N) \rceil + 3$ hops, the location of the packet is effectively random. In practice, however, the TTL of the request is set to values significantly larger than $\lceil \log_n(N) \rceil + 3$. For instance in our experiments for a 5-Cycle network of 10,000 nodes with $\lceil \log_5(10{,}000) \rceil + 3 = 9$ we used a TTL of 200 hops. Therefore, the choice of the insertion point is basically irrelevant after the first 9 hops.

[3] We note that it is outside the scope of this paper to propose methods of detecting participants who feign cooperation, or cheat through other methods.

Table 1
Simulation parameters

Input parameters	
Number of grid nodes	10,000
Overlay network	5-Cycle
Task arrival	Poisson-distributed arrival, mean 10 . . . 200 tasks/s
Task servicing	Normally distributed, mean 60 s/task
Simulation time	5000 s
Pre-walk hops	0 and 9
Output parameters (Measurements)	
Hops per task	Number of hops a task is forwarded until it finds a host for execution (avg, max)
Average load	Ratio of busy vs. total nodes
Discarded tasks	Number of tasks which were discarded

We conclude that even in the most unfavorable conditions, every task has the same chance of being allocated, with only minor differences in the number of hops until allocated for the case of a small number of insertion points with highly asymmetric loads.

2.3.2. *Fair treatment of providers*

The fair treatment of the providers means that each provider has the same chance of being allocated a task. In practice, it is easier to measure the number of tasks allocated to a provider during a period of time. Since every allocated task provides some revenue, a selfish provider wishes to have as many tasks allocated as possible.

Both the random wandering and weighted stochastic forwarding are greedy regarding the allocation of tasks—if a request reaches an available host it will be immediately allocated. This leads to an unbalanced (and, according to our definition, unfair) load distribution. The nodes in the vicinity of the insertion point will be almost always fully loaded, while nodes farther from the insertion point will be idle. Paradoxically, this effect is more pronounced for lighter loads. If the number of tasks is sufficiently small, the entire load can be handled by the neighbors of the insertion point, while the rest of the network would not receive any task.[4]

It follows that the algorithms ensure provider fairness only if the insertion points are selected at random. It is *not* enough for a consumer to choose an insertion point randomly, and later send all its tasks through that point; the random selection needs to be repeated for every task. This would require every consumer to have global information about the network.

By exploiting the built-in randomness of the n-Cycle network, we can make a small modification in the forwarding algorithms which will ensure fairness at the cost of a small decrease in performance. We require every task description packet to perform a random pre-walk of length m before the task can be taken into execution. The pre-walk number m will be part of the task description packet. As long as $m > 0$, the packet will be forwarded randomly according to the random wandering algorithm, but it will not be allocated to any node even if the current node is free. At every forwarding, the pre-walk number will be decreased by 1. The node at which the packets arrive with $m = 0$ will be called *effective insertion point*. From then on, the weighted stochastic algorithm will be followed.

With the same reasoning as in Section 2.1.2, we can prove that a value of $m = \lceil \log_n(N) \rceil + 3$ leads with a statistical certainty to a random effective insertion point. We study the fairness properties of the presented algorithms with and without pre-walk in Section 3.3.

We note that participating in pre-walk is not in the immediate interest of the providers. Some providers might cheat, by accepting a task for execution, before the required pre-walk tests. The potential solutions to this problem (such as by the use of signatures) is outside the scope of this paper. We note that many networking protocols (such as the TCP rate control) rely on the cooperation of the participants without deploying any cryptographic assurance, despite the fact that there is a possibility of the nodes cheating, for instance by not reducing their rate in the case of congestion.

3. Simulation studies

We have used the YAES [3,24] simulation framework to simulate the behavior of the algorithm. Table 1 illustrates the input and output parameters of the simulation as specified in the YAES configuration files.

3.1. *Performance study of the forwarding algorithms*

First we present the results of a performance study where a gradually increasing number of tasks are inserted in the network through a single insertion point. The tasks are forwarded using the random wandering and weighted stochastic forwarding algorithms. We measure the average and maximum number of hops required for a task to be assigned to an available node. To prevent the tasks to wander indefinitely in the network, we assign to every request a time-to-live (TTL) value, which in our experiments was set to 200 hops. Once the TTL of a packet expired, the receiving node will forward it back to the client, instead of forwarding it to one of the downstream nodes. Thus the client receives prompt notification that the task was not allocated and can take appropriate action. We also measure the load of the network (the ratio of the busy nodes to the

[4] The number of providers which will be allocated tasks can be determined by considering the arrival rate and distribution of the tasks, the servicing rate of N nodes and a simple queuing theoretic model.

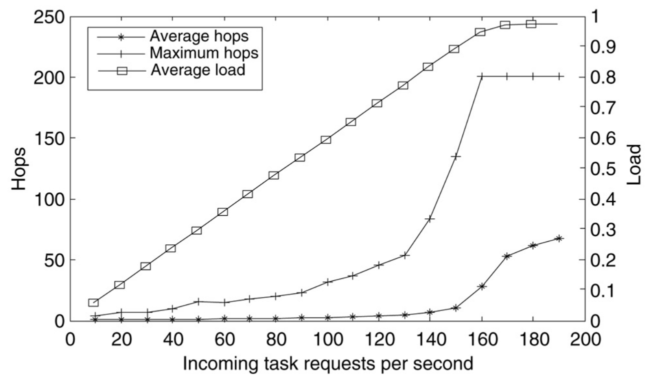

Fig. 2. Number of hops (average and maximum) and network load vs. the incoming number of tasks per second, using random wandering on the n-Cycle overlay network.

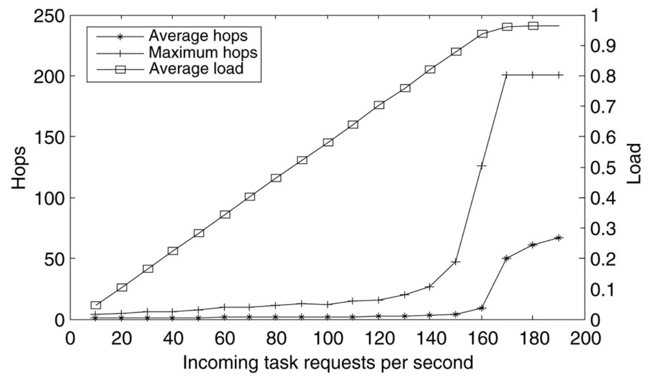

Fig. 3. Number of hops (average and maximum) and network load vs. the incoming number of tasks per second, using the weighted stochastic forwarding algorithm.

total number of nodes). We show the graphs for the number of hops needed for a task to be allocated on the same plot with the graphs for the load for a better representation of the relationship between these quantities. The flattening of the load graph indicates the level where requests start to get discarded.

The results for the random wandering algorithm are presented in Fig. 2. For light loads, this algorithm shows very good results (due to the randomizing nature of the n-Cycle network). However, for greater loads, the maximum number of hops start to increase.

The results for the weighted stochastic forwarding algorithm are presented in Fig. 3. We note that both the average and maximum number of hops stay virtually constant at a very low number (under 10 hops), up to loads approaching 95%. For instance, at a load of 90% the maximum will be as low as 10 hops, versus about 75 hops for the RW algorithm. At that moment the number of hops increases dramatically as the algorithm struggles to find free nodes in an overwhelmingly busy network.

The relatively constant number of nodes for moderate loads is explained by the single insertion point. The nodes closer to the insertion point will be filled in relatively quickly, so the majority of tasks need to "hop over" the busy nodes in this area. A good approximation of the size of this constant value is $\log_N(|W|)$ which in our case is $\log_5(10{,}000)$, approximately 5.7. If we choose a random insertion point, the diagram has a similar shape, but with an average number of hops for lightly loaded networks much smaller (about 1–2 hops).

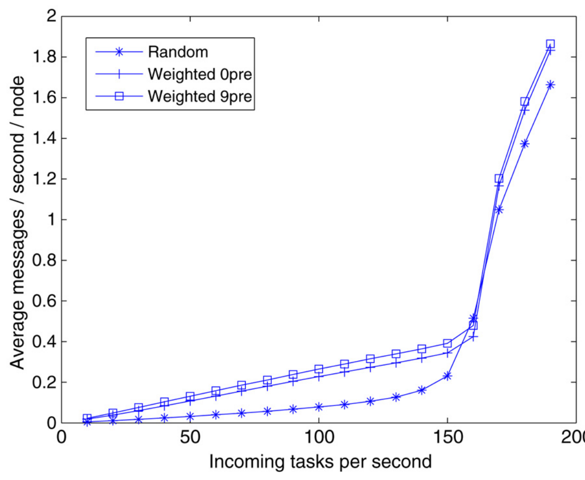

Fig. 4. The cost of task allocation expressed as the number of messages per second which need to be processed by the algorithms as a function of the incoming tasks.

3.2. *The cost of the forwarding algorithms*

The use of an overlay network for task allocation comes with the cost of the messages which need to be forwarded by the individual nodes. These messages are task requests, and, in the case of weighted stochastic forwarding, also messages for the maintenance of the status information. In a series of experiments, we measured the number of messages forwarded by every node for the RW, WSF and WSF with pre-walk cases (Fig. 4). As expected, the cost of the WSF algorithm is higher due to the existence of status messages. Pre-walk also adds to the number of messages which needs to be processed by the system. However, in general, the average number of messages processed by the nodes is quite low. Even for very high loads, it stays at a level of about 1 message every 2 s. This load can be easily handled by any node of the grid.

3.3. *Fairness measures*

In a separate series of measurements, we measure the fairness towards the providers as a function of the number of tasks arriving to the network. The measurements are performed by counting the number of tasks executed by every host. These values are then sorted and four values picked at the minimum, maximum, 5% and 95% percentiles. The reason for plotting the intermediate values is to filter out providers having special positions in the network. For example, for a single insertion point network, the insertion point has a special situation, given that all incoming tasks are passing through it. We performed all the measurements using WSF.

Fig. 5 shows the results of the measurements for the case of a single insertion point. As expected, the maximum value shows that the insertion point will achieve 100% load. The bottom 5%

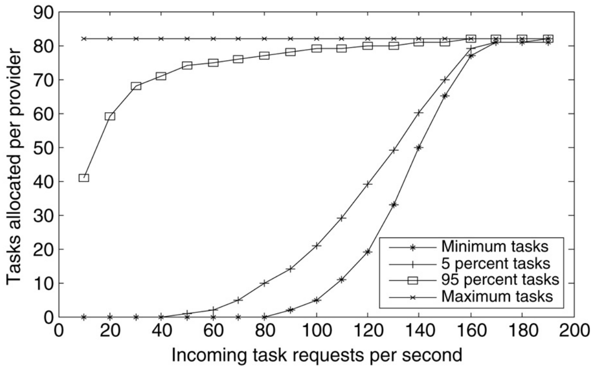

Fig. 5. Fairness in terms of tasks allocated to providers. Single insertion point, no pre-walk, weighted stochastic forwarding.

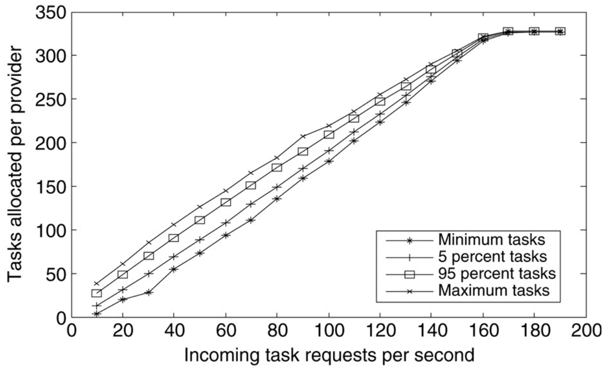

Fig. 6. Fairness in terms of tasks allocated to providers. Random insertion point, no pre-walk, weighted stochastic forwarding.

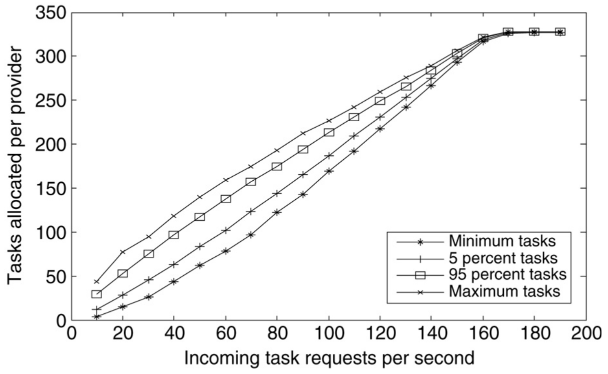

Fig. 7. Fairness in terms of tasks allocated to providers. Single insertion point, 9-hop pre-walk, weighted stochastic forwarding.

has no tasks allocated for task arrival rates as high as 40 tasks/s. The gap between the four measurements is higher at low loads, and lower at high loads when even providers far from the entry point will be allocated tasks. This measurement validates our prediction that for a single insertion point the allocation method leads to unbalanced and unfair distribution of tasks.

Fig. 6 presents the measurements for the case of a random insertion point. Again, the simulation results match the prediction; the number of tasks executed by the nodes are in a relatively narrow range, without standout values. As an observation, the reason of the spread in the values is due to the inherent randomness (Poisson arrival, random insertion point, normally distributed execution time) and the limited timeframe of the simulation. Simulated over longer timeframes, these values are converging to a single line. This is basically the ideal fairness, but as we stated before, it requires global information about the network to prepare a proper random insertion point.

Fig. 7 shows the measurements for a single insertion point and the task distribution algorithm including a 9 hop pre-walk. The value of 9 is the empirically obtained value of $\lceil \log_n(|W|) \rceil + 3$ for $|W| = 10{,}000$ and $n = 5$. We should note the resemblance of the diagram to Fig. 6. We conclude that a pre-walk with sufficient number of hops achieves the same results as the random insertion point approach, while still requiring only local information.

3.4. *Comparison of the three methods of creating an n-Cycle network*

We proposed three methods for the creation of the n-Cycle overlay: through a centralized approach, through the traversal of a single cycle and through multi-cycle jumping. While the first two methods are equivalent in terms of the randomness of the network created, the multi-cycle jumping offers only a statistical probability that the newly inserted node is in a random position uniformly chosen from the complete network.

In this series of experiments, we compare the n-Cycle networks created by the three approaches, as well as the cost of creation. Fig. 8(a) compares the cost of creating a network using the single cycle traversal versus the multi-cycle jumping approach. As expected, the cost of single cycle traversal increases quadratically with the number of nodes, and it is significantly larger than the $N \cdot \log(N)$ cost of the multi-cycle jumping. The centralized creation method is not comparable in these terms, as it requires a global view of the network.

Fig. 8(b) compares the performance of the n-Cycle networks created with the three methods on a network of 10,000 nodes using the weighted stochastic forwarding algorithm. We can conclude that the performance is virtually the same for all the three creation methods, with a very slight disadvantage of the multi-cycle jumping method, visible only at high loads. These results establish the multi-cycle jumping method as being the preferred method of creating and maintaining an n-Cycle network; it has an order of magnitude better creation performance and virtually indistinguishable forwarding performance compared with more expensive methods.

3.5. *The effect of correlation between the cycles*

The n-Cycle algorithms rely on the randomness of the network to provide an efficient and fair distribution of the tasks. Correlated sequences in the cycles, or local clusters of interconnections, lead to a degradation of the performance. In this series of experiments, we measure the performance of the network (in terms of average hops needed to allocate a task) on n-Cycle networks with various levels of correlation. If the cycles are created from independently created permutations, there is, of course no correlation between them. However, if

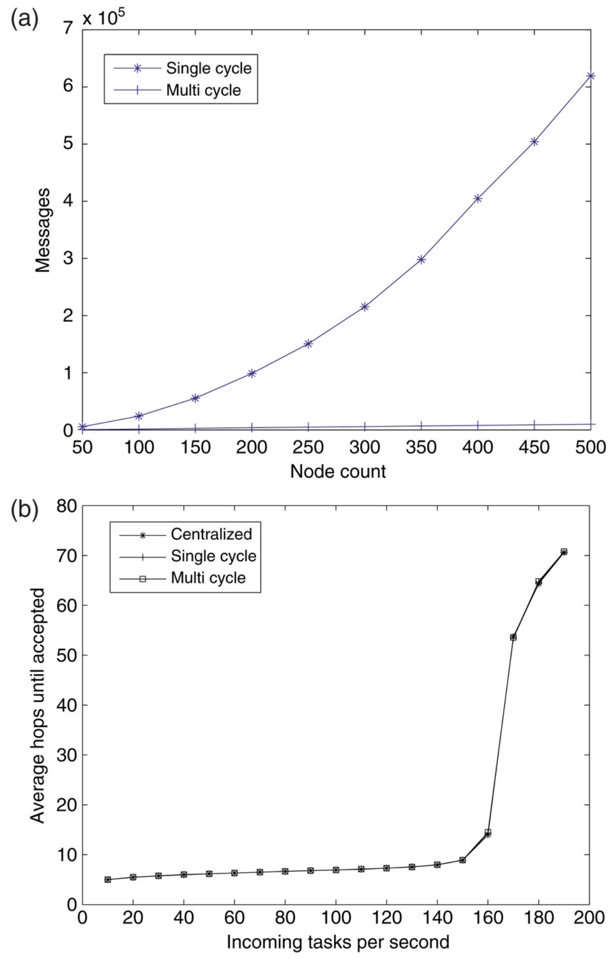

Fig. 8. (a) The number of messages required to create a network using random walk on a single cycle vs. random walk with cycle jumping as a function of the node count. (b) The number of hops needed to allocate a task using WSF for the three different creation methods.

they are created using the cycle jumping algorithm, and the number of jumps is too low, the next hops will be selected from the same neighborhood—leading to an undesired locality in the structure of the network. In Section 2.1.2 we showed that using $\log_5(|W|) + 3$ hops, where $|W|$ is the size of the network, leads to a statistical certainty that no undesired locality will appear in the network. To illustrate this effect, on Fig. 9 we show the shape of the n-Cycle network, for a network of $N = 50$ nodes, created with the cycle jumping method with $\log_5(N) - 1$, $\log_5(N)$ and $\log_5(N) + 1$ hops, respectively. As we see, if the number of hops is too low, the links tend to be more dense around the perimeter of the cycle, which shows that they are mostly local links.

To study the effect of correlation on the performance of the task allocation, we use an artificially weakened version of the cycle-jumping network creation method. Instead of the analytically determined $\log_n(N) + 3$ hops, we create networks with $\log_n(N)$, $\log_n(N) - 1$ and $\log_n(N) - 2$ hops respectively. The results in Fig. 10 show that there is relatively little performance degradation at the $\log_n(N) - 1$ level, but performance suffers significantly at $\log_n(N) - 2$. Task allocations which take 6–7 hops on a random network would take 60–140 hops if correlations are present.

3.6. *The influence of the order of the network*

In this series of experiments, we study the influence of the order of the network in the performance of the forwarding algorithms. Evidently, the higher the order of the network, the higher performance we expect, due to the additional connections which can be exploited. The higher order however comes with a price. The messages needed for the creation of the network, and the number of update messages for the weighted stochastic forwarding, are linearly increasing with the order of the network. There is no performance penalty with the increasing order for the random wandering allocation algorithm.

We measure the average number of hops to allocation for networks ranging from 2-Cycle to 25-Cycle. Fig. 11 plots these values for a relatively highly loaded grid. The measurements were taken with the weighted stochastic forwarding algorithm. With 170 tasks per second on a network of 10,000 nodes, the load is just below the full capacity of the grid, which amplifies the performance differences. Even in these cases, we find that the increase of the order beyond 5-Cycle led to insignificant increase in performance. The increase of order from 2-Cycle to 5-Cycle increased the performance with approximately 25%. In conclusion, the 5-Cycle network represents a good compromise in most cases.

4. Related work

This paper proposes an architecture where the commodity tasks are allocated on a grid by forwarding the requests on a

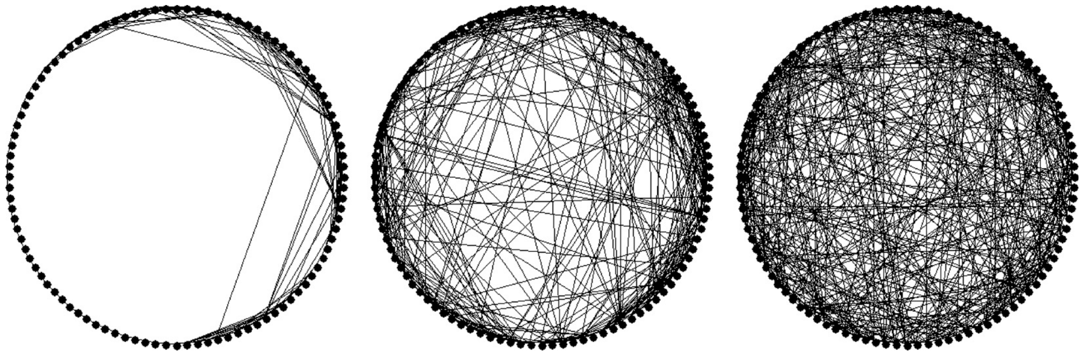

Fig. 9. Visualizing an n-Cycle network of $N = 50$ nodes created with the cycle jumping method with $\log_5(N) - 1$, $\log_5(N)$ and $\log_5(N) + 1$ hops, respectively.

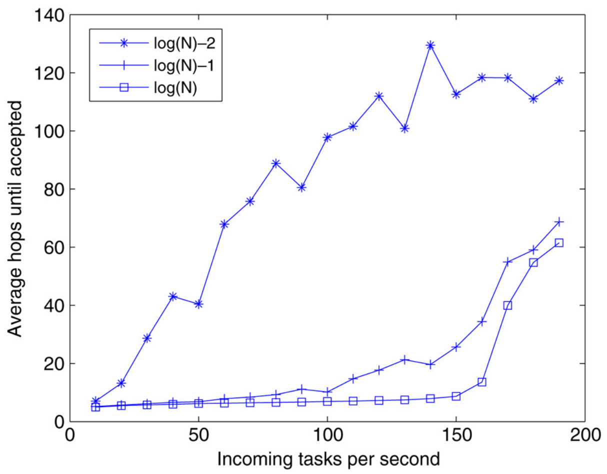

Fig. 10. The effect of the correlations between the cycles of the n-Cycle overlay on the performance of the forwarding algorithm. We compare networks created with the cycle-jumping method with $\log_n(N)$, $\log_n(N) - 1$ and $\log_n(N) - 2$ hops. The smaller the number of hops during creation, the stronger the correlation between the cycles.

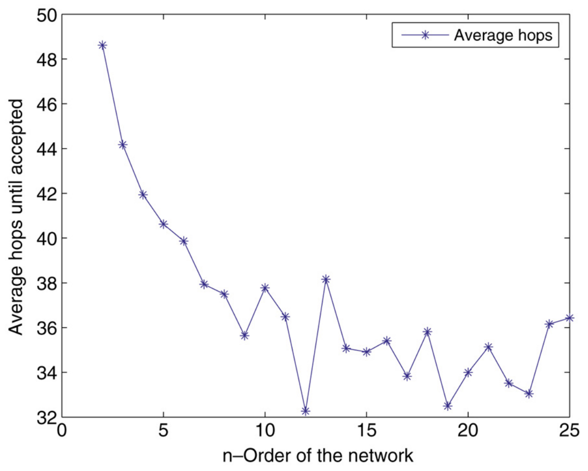

Fig. 11. The effect of the order of the network on the performance of the forwarding algorithms for a heavily loaded network.

overlay network. Similar designs are proposed in [2,6,9]. The Wire Speed Grid Project [23] proposes an architecture in which the task allocation is performed hardware accelerated on the network routers. [11] proposes a model for the allocation of cooperating tasks on networks of workstations.

Alternatively, we can see our work as the problem to discover available resources on the grid. A super-peer model for resource discovery is proposed in [8]. A QoS aware task scheduling model is discussed in [15].

The algorithms presented in this paper have their closest relatives in the class of distributed algorithms which create and exploit an additional graph structure, built on top of the existing, fully connected internet (often called *overlay network*). One of the most important classes of overlay networks are Distributed Hash Tables (DHT). These networks store pieces of data with their associated unique key. Every key and the associated data is mapped to a certain host, which is normally not known to the user. Data can be inserted and retrieved from a DHT without knowledge on where it will actually be stored—in fact, it is possible that the location of data will change as hosts join and leave the DHT. A number of DHT architectures were proposed such as CAN [10], Chord [13], Pastry [12], and Tapestry [16]. For most of these networks, every node maintains $O(\log(N))$ neighbors and a message can be routed in $O(\log(N))$ hops.

The properties of a DHT allow us to use it as the basis for a resource discovery and allocation framework. An example of this is the Self-Organizing Flock of Condors project [5] which is augmenting the Condor program with a DHT based on the Pastry overlay.

The CCOF (Cluster Computing On The Fly) project [17] implements a system in which idle cycles are harvested from a collection of computers. The system employs community based overlay networks, which allow hosts to dynamically join and leave. For the actual resource allocation step, a variety of search algorithms were implemented and measured, the most complex being Advertisement Based Search and Rendezvous Point Search.

The n-Cycle differs in many ways from the approaches outlined in this section. Its overlay network is not based upon Distributed Hash Tables. It does not use multicast communication. The algorithm requires exclusively local information both for the actual forwarding and the maintenance of the overlay network. The grid model considers commodity tasks and commodity resource providers and under this assumption it is more efficient to queue the tasks on the consumer side than on the provider side. This differentiates n-Cycle from models based on provider side queuing, which are more suitable for specialized resources, e.g., Condor.

The system assumes some sort of payment services. When a host plays both the role of a consumer and a provider there is no guarantee that the credits and debits of a host will cancel out in time. The "long term fairness" is not an inherent property of our algorithm, and the fairness of the task allocation must be ensured with explicit techniques.

5. Conclusions and future work

In this paper, we introduce two algorithms for task allocation on a commodity grid. Our analysis based upon simulation studies shows that the algorithms are: (a) scalable and produce very good results even for very large networks of several million nodes, and (b) efficient.

Our future work covers extensions and a more extensive analysis of the algorithms. An important extension is to design algorithms for heterogeneous tasks and heterogeneous services. We also plan to develop congestion control algorithms and to continue our studies of fairness in a more complex setting.

References

[1] D.P. Anderson, Public computing: Reconnecting people to science, in: Proceedings of the Conference on Shared Knowledge and the Web, November 2003.

[2] B. Liljeqvist, L. Bengtsson, Grid computing distribution using network processors, in: Proc. of the 14th IASTED Parallel and Distributed Computing Conference, November 2002.

[3] L. Bölöni, D. Turgut, YAES—a modular simulator for mobile networks. In: Proceedings of the 8-th ACM/IEEE International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems MSWIM 2005, October 2005, pp. 169–173.

[4] L. Bölöni, D. Turgut, D.C. Marinescu, *n*-Cycle: a set of algorithms for task distribution on a commodity grid, in: IEEE International Symposium on Cluster Computing and the Grid, CCGrid 2005, May 2005.

[5] A.R. Butt, R. Zhang, Y.C. Hu, A self-organizing flock of Condors, in: Proceedings of IEEE/ACM Supercomputing 2003, Phoenix, AZ, November 2003.

[6] A. Iamnitchi, I. Foster, On fully decentralized resource discovery in grid environments, in: Proceedings of the International Workshop on Grid Computing, Denver, CO, November 2001.

[7] D.C. Marinescu, Y. Ji, A computational framework for the 3d structure determination of viruses with unknown symmetry, Journal of Parallel and Distributed Computing 63 (7–8) (2003) 738–758.

[8] C. Mastroianni, D. Taliab, O. Vertab, A super-peer model for resource discovery services in large-scale grids, Future Generation Computer Systems 21 (2005) 1235–1248.

[9] C. Pairot, P. García, A.F.G. Skarmeta, R. Mondéjara, Towards new load-balancing schemes for structured peer-to-peer grids, Future Generation Computer Systems 21 (2005) 125–133.

[10] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Schenker, A scalable content-addressable network, in: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM'01, 2001, pp. 161–172.

[11] C. Rehn, Dynamic mapping of cooperating tasks to nodes in a distributed system, Future Generation Computer Systems 22 (2006) 35–45.

[12] A. Rowstron, P. Druschel, Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems, Lecture Notes in Computer Science 2218 (2001) 329–350.

[13] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, H. Balakrishnan, Chord: A scalable peer-to-peer lookup service for internet applications, in: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM'01, 2001, pp. 149–160.

[14] D. Thain, T. Tannenbaum, M. Livny, Condor and the grid, in: F. Berman, G. Fox, T. Hey (Eds.), Grid Computing: Making the Global Infrastructure a Reality, John Wiley & Sons Inc., December 2002.

[15] C. Weng, X. Lu, Heuristic scheduling for bag-of-tasks applications in combination with QoS in the computational grid, Future Generation Computer Systems 21 (2005) 271–280.

[16] B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph, J.D. Kubiatowicz, Tapestry: A global-scale overlay for rapid service deployment, IEEE Journal on Selected Areas in Communications 22 (1) (2004).

[17] D. Zhou, V. Lo, Cluster computing on the fly: Resource discovery in a cycle sharing peer to peer system, in: Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid, 2004, pp. 66–73.

[18] Berkeley Open Infrastructure for Network Computing. URL http://boinc.berkeley.edu/.

[19] Folding@Home project. URL http://www.stanford.edu/group/pandegroup/folding/.

[20] Mersenne Prime search. URL http://www.mersenne.org/prime.htm.

[21] RSA Challenge. URL http://www.rsasecurity.com/rsalabs/challenges/.

[22] SETI@Home project. URL http://setiathome.ssl.berkeley.edu/.

[23] The Wire Speed Grid project. URL http://www.ce.chalmers.se/staff/labe/WireSpeedGridProject.htm.

[24] YAES: Yet Another Extensible Simulator. URL http://netmoc.cpe.ucf.edu/Yaes/Yaes.html.

Ladislau Bölöni is an assistant professor at the School of Electrical Engineering and Computer Science at University of Central Florida. He received his Ph.D. degree from the Computer Sciences Department of Purdue University in May 2000. He is a senior member of IEEE, member of the ACM, AAAI and the Upsilon Pi Epsilon honorary society. His research interests include autonomous agents, grid computing and wireless networking.

Damla Turgut is an assistant professor at the School of Electrical Engineering and Computer Science at University of Central Florida. She received her Ph.D. degree from the Department of Computer Science and Engineering, University of Texas at Arlington. Her current research interests include wireless networking and mobile computing, distributed computing, software engineering, object-oriented and mobile databases.

Dan C. Marinescu is Professor of Computer Science at the School of Electrical Engineering and Computer Science at University of Central Florida. He is also an adjoint professor at Tsinghua University in Beijing. From 1984 until August 2001 he was a Professor of Computer Science and (by courtesy) of Electrical and Computer Engineering at Purdue University. Before coming to Purdue, Dr. Marinescu was an associate professor of EECS at the Polytechnic Institute in Bucharest and a senior researcher at the Institute for Atomic Physics of the Romanian Academy of Science, the Joint Nuclear Research Institute at Dubna, and G.S.I. Darmstadt. He was a visiting professor at: IBM T.J. Watson Research Center, Yorktown Heights, New York (1985); Institute of Information Sciences, Beijing (1992); Scalable Systems Division of Intel Corporation (1993); Deutsche Telecom (1996); and INRIA Paris (1998, 2000). His research interests cover parallel and distributed systems, Petri Nets, scientific computing, and quantum computing and quantum information theory.