A-FAST: Autonomous Flow Approach to Scheduling Tasks

Sagnik Nandy Larry Carter Jeanne Ferrante

Department of Computer Science and Engineering
University of California at San Diego
{snandy, carter, ferrante}@cs.ucsd.edu *

Abstract. This paper investigates the problem of *autonomously* allocating a large number of independent, equal sized tasks on a distributed heterogeneous grid-like platform, using only local information. We propose A-FAST (Autonomous Flow Approach to Scheduling Tasks), an *efficient, scalable, dynamic and generic* (imposing no restrictions on the topology) protocol for this purpose. Motivated by the idea of pressure guiding the flow in fluid networks, A-FAST only uses parameters available *locally* to a node to guide scheduling decisions. Simulations show that the protocol performs well over a variety of networks, averaging more than 99.5% of the optimal performance and outperforms related techniques like *RID* (Receiver Initiated Diffusion). We also show how a modified use of local information can improve the performance of an unreliable system. Preliminary results from implementing A-FAST on a small but real-life distributed system show the performance of our protocol to be near the maximum throughput of the system. Such a protocol has the potential to aid the efficient deployment of large, data intensive applications on very large or dynamically changing heterogeneous peer-to-peer computing platforms.

Key Words : Heterogeneous computing, peer-to-peer computing, network flows, scheduling.

1 Introduction

The advent of collaborative computing efforts like SETI@home project [25], GIMP [21] and Entropia [9] has given rise to a range of applications where a large set of tasks can be distributed across a grid-like platform and solved concurrently. These applications form the driving motivation of our work, which aims to schedule a large number of *independent, equal-sized* tasks, *online* across a *dynamic* and *heterogeneous* computing platform. We seek a scheduling strategy with the following properties:

– **Autonomous** - Uses minimal (or no) global information. In particular it should not require network-wide information.

* This work was supported in part by NSF grant ACI-0234233

- **Generic** - Applies to all kinds of networks, regardless of topology.
- **Efficient** - Results in high overall throughput.
- **Scalable** - Applies to networks of very large size.
- **Dynamic** - Adjusts to systems where, due to contention or other reasons, the bandwidths and computation speeds change over time.
- **Practical** - Is easy to implement in real-life scenarios.

The autonomic behavior of fluid networks, using pressure as a guiding force, forms the key inspiration for our work. One can imagine the nodes in a grid as fluid reservoirs, and the links as pipes connecting these reservoirs. Tasks are analogous to the circulating fluid in this scenario. In case of the fluids, *pressure* helps in bringing the system to a steady state without the use of any centralized control. We propose a similar approach where nodes autonomously measures their own pressure. This pressure is then used to decide when to move a task to a neighboring node, eliminating the need for centralized control over scheduling. A-FAST shares similarities with well-known techniques like Cycle Stealing [5] and RID [22], but differs from these techniques by taking both computation and communication into account, which makes it better suited for a wider range of networks. We show how several important scheduling-related issues, including fairness, throughput and reliability, can be easily incorporated in our approach. Initial simulations show that the protocol achieves more than **99.5%** of the maximum throughput over a range of networks, while preserving the above-mentioned properties.

The rest of the paper is organized as follows - Section 2 discusses the related work in this area and Section 3 describes the protocol in detail. In Section 4 we present experimental results showing performance of the protocol under various conditions. We conclude in Section 5 with a summary of our findings and suggest future research directions.

2 Related Work

Scheduling independent tasks across heterogeneous sets of resources is a well known problem. We differ from many of these approaches [14, 1, 6, 24, 10, 17, 19, 12, 29] in that we are developing an *autonomous* scheduling strategy that does not require centralized control or knowledge for scheduling.

Several research efforts have formulated the problem of scheduling tasks across heterogeneous systems as a max-flow problem [7], [30]. However, the most popular max-flow algorithms, including Ford-Fulkerson [15] and Edmonds-Karp [8] use global information to make network-wide decisions. Golberg's algorithm [11] is closer to being autonomous but still requires a notion of *height* that depends on the total number of nodes in the network. In [26], the authors provide a parallel solution to the max-flow problem. However, their approach uses a notion of *timesteps* across the network. This involves network-wide synchronization and is difficult to achieve in large networks. Moreover, all these techniques were designed specifically for static systems. In practice, system properties, such

as node speed, bandwidth, network topology, change over time, making these techniques unsuitable.

A-FAST shares similarities with the *RID* (Receiver Initiated Diffusion) [22, 13] and other similar *gradient-based approaches* [18, 27]. In these approaches nodes use some notion of gradient to balance their workload among their neighbors. However, they make their scheduling decisions completely based on the load at a node without taking its communication into account. A-FAST adopts a diffusion-like approach similar to these techniques, but requests tasks based on the supply rate of a node. This ensures that more tasks are received from nodes connected by faster link-speeds. This makes the protocol applicable to both *computation and communication* dominated systems. Moreover, in A-FAST all communication decisions for a pair of nodes is done independently of their remaining neighbors, reducing the synchronization requirements among nodes. We also show later in the paper how A-FAST manages to capture other system properties like reliability in its notion of pressure.

In [5, 2, 28] variants of the *Cycle Stealing* technique addresses a similar problem as ours. In Cycle Stealing, a node that has exhausted all its work randomly asks its neighbors for additional work. While this approach is autonomous and works well for computation intensive applications, it requires the nodes to be arranged in a hierarchical fashion to avoid unnecessary transfer of tasks. Moreover, Cycle Stealing does not take communication time into account and does not differentiate between nodes connected by different connection speeds.

We only consider applications where there are far more tasks to be executed than nodes in the system, so throughput is more important than makespan, latency or response time. In our previous work [4], [16], we presented an autonomous algorithm that, when the network is a *tree*, achieves the optimum throughput for a static network. Our experiments showed that the protocol reacts quickly to changes in the network as well. However, it may not be desirable to impose a tree-structure on large networks. In [3], it is proven that the problem of finding the best tree from a given network is NP-complete, and even if one could find the best tree, there are networks for which the performance of the optimal tree is unboundedly worse than the whole network's performance. Thus, finding an autonomous solution for a generic network is still open.

3 The A-FAST Task Scheduling Protocol

We begin with a formal description of the problem. We are given a labeled, directed graph *G = (N, E, P, C)* representing the network. $N = \{0, 1, ..., n-1\}$ is the set of computing resources. Each node $i \epsilon N$) has a computing speed $P(i)$ ($P : N \rightarrow R^+$), denoting the number of tasks the node can complete in a unit time. $E = \{(i, j) : i, j \epsilon N\}$ is the set of links connecting the various nodes in this graph, and $C(i, j)$ ($C : N \times N \rightarrow R^+$) denotes the number of tasks that can be sent from node i to node j in a unit time. All tasks are of equal size (both

computationally and communication-wise)[2] and initially reside in the source node 0. We assume that node 0 has a large number of tasks, so we can ignore the start-up and wind-down times of the protocol. The graph G is dynamic in nature, i.e. *(N, E, P, C)* can evolve during execution. Nodes and edges can be added and deleted from N and E (except for node 0, which is always present) and $P(i)$ and $C(i, j)$ can also change. Our objective is to maximize the number of tasks completed per unit time.

The A-FAST protocol assumes that some number of incoming tasks can be buffered in a node. Nodes begin by advertising a quantity we will call their *pressure* (p) to their neighbors, requesting them for tasks. On receiving a request, a node compares the requester's p to its own to decide whether the request should be serviced. Such an approach allows us to do away with the need for a centralized scheduler, and instead make all scheduling decisions locally based on differences in pressure. If a node does not service a request, it informs the requestor of its decision. On being serviced by a neighbor, a node requests another task from the same neighbor. However, if its request is denied, it waits for a set length of time before making another request. Nodes thus periodically query their neighbors, requesting further tasks. To process a task, a node takes a task from its buffer. If the buffer is empty the node waits till it receives a task.

We now give two case studies that show how A-FAST can autonomously achieve two different system requirements — high throughput and improved reliability — by making suitable definition of pressure.

3.1 Task Scheduling in Dynamic Heterogeneous Systems

For each edge $(j, i) \epsilon E$, we assume there is an *incoming buffer* $IB_{j,i}$ on node i that holds the most recent response (either a denial or a task) sent by j to i. Additionally, each node i has a *task buffer* TB_i that has a capacity of m_i "*slots*", where each slot can hold one task. These slots can be in one of the following states:

- **S1**: the slot is "empty".
- **S2**: a task is being transferred into the slot from one of the IB_{ji}s.
- **S3**: the task in the slot is getting executed by N_i.
- **S4**: the task in the slot is being sent to another node N_k i.e. it is being transferred from TB_i into IB_{ik}.
- **S5**: the slot holds a task and is currently not in any of the above states.

Task buffers can have multiple slots in states **S1**, **S2**, **S4** and **S5**, but for simplicity we will allow only one task at a time to be in state **S3**. We say "TB_i is full" when the number of slots e_i in state **S1** is zero. We define the *buffer occupancy*, b_i of a node to be the number of slots in state **S5** at the current time.

[2] We conjecture that if tasks are of different sizes, but have a constant computation-to-communication ratio, that the behavior of algorithms will be similar to the equal-size task problem. An interesting open question is how to make scheduling decisions when the ratio is non-constant but known.

For scheduling tasks in a heterogeneous system we set the pressure, p_i, of each node to its buffer occupancy.

The sub-protocols for responding to a request, processing a response and performing a task are shown in Figures 1 and 2. The highlighted sections of the protocols use the shared variables b_i and/or e_i, and must be synchronized to run correctly. This can be done by acquiring locks (if each protocol is a separate process), or by executing the shaded sections atomically (if a single buffer manager procedure handles all three protocols). The `Wait` primitive in Figure 2 should be implemented using a periodic polling mechanism that prevents livelock.

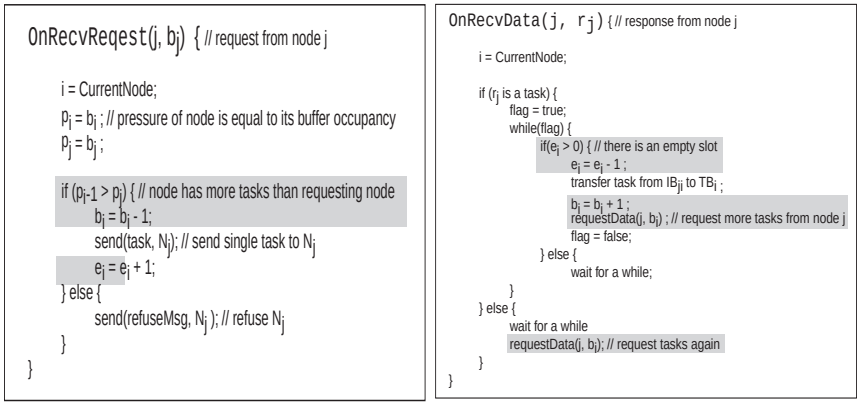

Fig. 1. Protocol nodes follow on (a) receiving a task request (b) on receiving a response from a neighbor

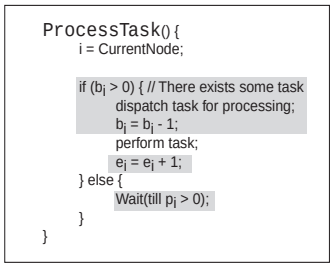

Fig. 2. Protocol nodes follow to perform a task

Intuitively, A-FAST should adapt to both a computation-dominated system as well as a communication-dominated one: faster nodes empty their

buffers faster and their *pressure* decreases, making them likely to receive more tasks. Similarly if a link is fast, tasks will be delivered more quickly across it, making the receiver request more tasks along that link as compared to a slower link. We will verify these claims experimentally in Section 4.

3.2 Adding Reliability to the System

We now show how the idea of pressure can be modified to incorporate a measure of node reliability into the scheduling strategy.

We define an unreliability parameter, τ_i, for each node in the system, which reflects the average time a node remains online. A fair estimate of the value of τ_i can be computed completely independently by each node. This can be done by maintaining a three tuple of $<$*num_of_readings*,τ_i,*last_val*$>$ in the persistent storage of each node. On coming online, node i increments the value of *num_of_readings*, sets τ_i to $\frac{(\tau_i + last\_val)}{num\_of\_readings}$, saves these values, assigns *last_val* to 0 and then continues. *last_val* is periodically updated to the elapsed time and saved. The last recorded value of this variable can then be used as an estimate of how long the node remained online (the accuracy depends on the frequency of updates). τ_i thus gives an estimate of the expected duration node i is likely to remain online.

To incorporate reliability into A-FAST we modify our existing definition of pressure to $p_i = \frac{b_i}{\tau_i^K}$, where K is some real positive constant (we shall term it *Assurance Constant*) denoting the importance of reliability to the system. A node now sends the buffer occupancy and unreliability constant to its neighbors when requesting a task and the neighbor can calculate its value of p.

By doing this we make the pressure of a node inversely proportional to its chances of breaking down. Thus for two nodes with similar buffer occupancies, the node with a smaller value of τ (less reliable) will have a higher pressure, making tasks flow out of it towards a more "reliable" node. It must however be mentioned that giving too much importance to reliability might have adverse effects since slower but more reliable nodes will start getting more jobs assigned to them. This can be controlled by choosing an appropriate value of K and will be studied further in the experimental evaluations.

4 Experimental Results

We now present experimental results from simulations as well as real life systems to show how A-FAST works under different situations.

4.1 Experimental Setup

We tested A-FAST on two different networks topologies - internet-like graphs (G1), generated using the Network-Emulator package (NEM) [23] and cluster-

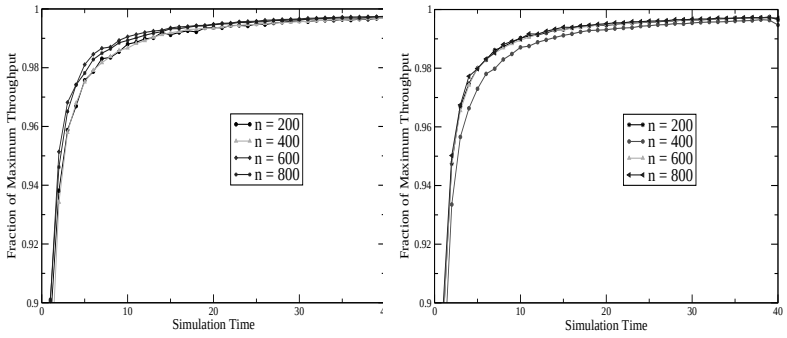

Fig. 3. (a)Performance of A-FAST on *Internet-like* graphs (G1) (b) Performance of A-FAST on *Cluster-like* graphs (G2)

like graphs (G2)[3]. For both topologies, we generated graphs of four different sizes (n = 200, 400, 600 and 800). Each node i in these graphs were assigned a random processing speed, $P(i)$ (ranging between 1 and MAX_P), representing the number of tasks node i can process in a minute of simulated time. The values of $C(i, j)$ were similarly assigned (ranging between 1 and MAX_C), denoting the number of tasks that can be sent along the link in one minute. We set both MAX_P and MAX_C to the same value (40) to allow the throughput of the system to be equally dependent on computation and communication. We assumed zero latency networks for our simulations. This might be unreasonable in certain scenarios where the frequent exchange of request messages and the single task transfer approach of the protocol might affect the performance of the system. We discuss in the concluding section how our ongoing work is addressing this issue for real systems (Note that by assuming zero latency the request/denial message transfer times were reduced to zero but the task transfer times were non-zero, depending on the bandwidth of the connecting links). Experiments were repeated multiple times and the average value over all the runs were reported.

4.2 Throughput in a Heterogeneous System

We compared the performance of the first variant of the protocol as a percentage of the maximum throughput of the system (calculated using the *maxflownet* package [20]). The results for the two types of topologies, are shown in Figures 3(a) and (b).

A-FAST performed very well, averaging over *99.5%* of the maximum throughput for both the topologies and the different sizes. Though our graphs were small, this showed A-FAST to be both generic and scalable. It can also be

[3] For G2, we built k clusters of equal size. Nodes in these clusters were heavily connected (average connectivity of $k/2$). The clusters were then connected to each other in a random tree topology.

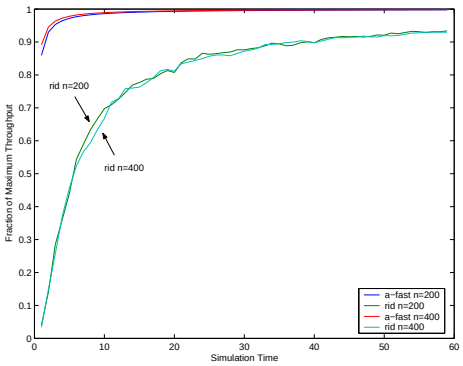

Fig. 4. Relative Performance of A-FAST vs RID on *Communication-dominated Graphs*

observed that the startup time of A-FAST is also small with almost all the simulations reaching 98% efficiency within 5 minutes of simulated time (5 minutes corresponded to approximately 750 completed tasks in our simulation setup).

We also implemented a version of the RID algorithm to compare its performance against A-FAST for communication-dominated systems. We generated these systems by generating the G1 type graphs where link speeds were less than the processing speed of the nodes joining them. The version of RID balanced the load every time the number of tasks in the Task Buffer fell below 5. The experiments were run for 60 minutes to allow the RID algorithm to reach steady state throughput. The results are shown in Figure 4. While A-FAST achieves nearly 99.5% of the optimal throughput, RID only achieves around 95% of the optimal value. RID also takes a larger amount of time to reach the steady state throughput.

4.3 Adding Reliability to A-FAST

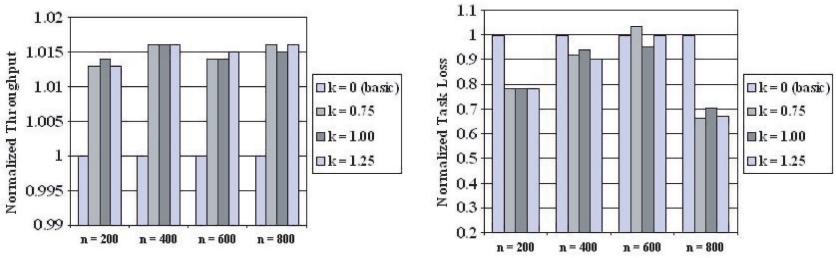

Fig. 5. Effect of adding reliability on (a) system throughput and (b) task loss

To test the reliability-aware A-FAST variant described in Section 3.2, each node was randomly assigned a value τ between the range $(5, 75)$. We conducted our experiments for 40 minutes of simulation time, giving nodes an equal chance of failing or surviving in the lifetime of the experiments. We then tested A-FAST with four different values of the unreliability constant K, denoting the importance of reliability for the experiments (note that for $K = 0$ the protocol reduces to the standard buffer-based pressure approach described in Section 3.1 and is provided as a base case)[4]. We measured the change in throughput and number of lost tasks (tasks that were assigned to nodes when they broke down). The results are shown in Figure 5 (a) and (b).

In all our experiments, the throughput of the reliability-aware version of A-FAST achieves better throughput when compared to the standard version. However, one cannot conclude anything definitive about the impact of K on throughput. This is because a smaller value of K reduces the importance of reliability and increases the chance of a potentially faulty node getting more tasks while a larger value of K might make slower and more reliable nodes get more tasks, thereby affecting performance. However, it is evident that the introduction of reliability as a parameter to pressure, does pay off.

We also see a marked improvement in the reduction of task losses with A-FAST. A task loss might eventually require re-transmitting the task and by reducing the task loss one might eventually improve the system-throughput even further.

4.4 Practical Implementation

As a proof of concept we implemented a prototype version of A-FAST. This section presents some initial results from these experiments. The version of A-FAST described in Section 3.1 was implemented using Java RMI. The system was tested on a 9-node cluster of Pentium III (800 MHz) processors. One of these nodes was designated as a source with a large number of matrix multiplication tasks. We arranged the remaining 8 nodes in three *virtual* topologies - star, ring and a complete binary tree. As a yardstick for A-FAST's performance we also provide an approximate upper bound of the throughput of the system. The maximum throughput of the system is bounded by the sum of the maximum throughput of each participating node. To get an estimate of a node's maximum throughput, we ran the tasks on each target node separately (multiple times). The total of these values is provided in Figure 6 (a). The individual throughput of the nodes are shown in Figure 6 (b)[5].

Though preliminary in nature, these results show A-FAST performs efficiently on all three topologies. The three topologies produced comparable

[4] We also tried the experiments with larger and smaller values of but for K>1.25 the results were similar to that of K=1.25 and therefore have not been shown in the graphs.

[5] We ignored the first 100 tasks transferred by n0 (to avoid the effect of startup time) and tracked the number of results contributed by each node for the *subsequent* 250 tasks.

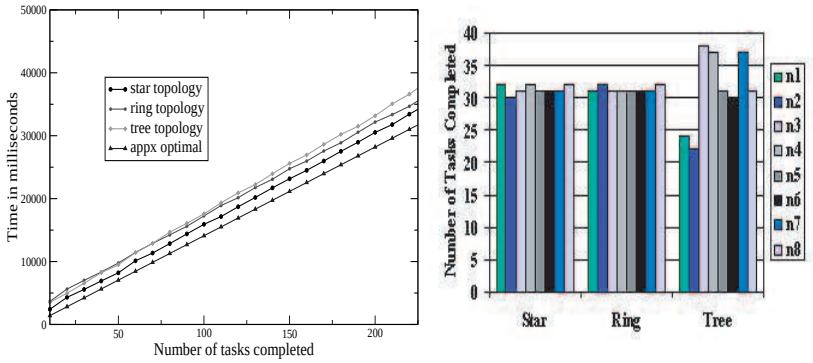

Fig. 6. (a) Execution time of A-FAST on a small cluster. (b) Individual throughput of nodes for the different topologies

results with an additive difference (contributed mostly by their start-up times). Figure 6 (b) shows that for the star and ring topologies A-FAST does a very good job of autonomous load balancing. However, for the tree topology certain nodes outperform the others. This happens mainly because nodes in the tree do not have equal connectivity and the throughput of the communication-intensive nodes was affected negatively.

5 Conclusion and Future Work

This paper presents a new autonomous scheduling protocol based on the idea of pressure in fluid networks. Preliminary experiments show that the protocol is efficient and can scale and autonomously adjust in dynamic heterogeneous networks. We showed how different parameters like throughput and reliability can also be captured. Simulations showed the protocol to be efficient, achieving more than 99.5% of the maximum throughput on average. The need for such protocols is likely to grow as we start using the world wide web not only as an information medium but also as a computing resource.

We are currently pursuing a number of different aspects of this problem. Some of these include: a theoretical bound on the performance of the protocol; the effect of unequal-sized tasks; the effect of dependency between tasks; and capturing other aspects of scheduling with pressure. We are also working on a version of A-FAST supporting *lazy* updates of pressure to reduce the effect of latency and periodic message transfers.

References

1. D. Andresen and T. McCune. Towards a Hierarchical Scheduling System for Distributed WWW Server Clusters. In *Proceedings of the Seventh International Symposium on High Performance Distributed Computing (HPDC-7)*, July 1998.

2. J. Baldeschwieler, R. Blumofe, and E. Brewer. ATLAS: An Infrastructure for Global Computing. *In Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.
3. C. Baninio, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Scheduling Strategies for Master-Slave Tasking on Heterogeneous Processor Platforms. In *IEEE Transactions on Parallel and Distributed Systems*, April 2004.
4. O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Bandwidth-centric Allocation of Independent Task on Heterogeneous Platforms. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, Florida*, April 2002.
5. R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, 1995.
6. H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW'00)*, May 2000.
7. T. Cormen, C. Leiserson, and R. Rivest. Introduction to Algorithms, MIT Press. 1990.
8. Jack Edmonds and Richard M. Karp. Theoretical improvements in the algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, 19, 1972.
9. Entropia Inc. http://www.entropia.com, 2001.
10. S. Flynn Hummel, J. Schmidt, R. Uma, and J. Wein. Load-Sharing in Heterogeneous Systems via Weighted Factoring. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'96)*, Jun 1996.
11. Andrew V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers.* PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1987.
12. T. Hagerup. Allocating Independent Tasks to Parallel Processors: An Experimental Study. *Journal of Parallel and Distributed Computing*, 47, 1997.
13. H.-U. Heil and M. Schmitz. Decentralized Dynamic Load Balancing: The Particles Approach. *Proc. 8th Int. Symp. on Computer and Information Sciences*, 1993.
14. O. H. Ibarra and C. E. Kim. Heuristic algorithms for scheduling independent tasks on non-identical processors. *Journal of the ACM (JACM)*, 24(2), 1997.
15. L. R. Ford Jr. and D. R. Fulkerson. Flow in Networks, Princeton University Press. 1962.
16. B. Kreaseck, H. Casanova L. Carter, and J. Ferrante. Autonomous Protocols for Bandwidth-Centric Scheduling of Independent-task Applications. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03), Nice, France*, April 2003.
17. C.P. Kruskal and A. Weiss. Allocating Independent Subtasks on Parallel Processors. *IEEE Transactions on Software Engineering*, 11, 1984.
18. Frank C. H. Lin and Robert M. Keller. The gradient model load balancing method. *IEEE Trans. Softw. Eng.*, 13(1):32–38, 1987.
19. M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. Freund. Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems. In *8th Heterogeneous Computing Workshop (HCW'99)*, pages 30–44, Apr. 1999.
20. Max-Flow-Solution. http://elib.zib.de/pub/Packages/mathprog/maxflow/index.html.
21. Mercenne Prime Search. http://www.mercenne.com.

22. Willebeek-LeMair M.H. and A.P Reeves. Strategies for Dynamic Load Balancing on Highly Parallel Computers. In *Parallel and Distributed Systems, IEEE Transactions*, 1993.
23. Network Emulator. http://clarinet.u-strasbg.fr/nem/.
24. A. Rosenberg. Sharing Partitionable Workloads in Heterogeneous NOWs: Greedier Is Not Better. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'01), Newport Beach, California*, October 2001.
25. SETI@home. http://setiathome.ssl.berkeley.edu, 2001.
26. Y. Shiloach and U. Vishkin. An $O(n^2 log\ n)$ parallel max-flow algorithm. *Journal of Algorithms*, (3), 1982.
27. W. Shu and L. V. Kale. A dynamic scheduling strategy for the chare-kernel system. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 389–398. ACM Press, 1989.
28. Rob van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Satin: Efficient parallel divide-and-conquer in java. In *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 690–699. Springer-Verlag, 2000.
29. B. Veeravalli, D. Ghose, and T. G. Robertazzi. Divisible load theory: A new paradigm for load scheduling in distributed systems. *Cluster Computing*, 6(1), January 2003.
30. Kevin Daniel Wayne. *Generalized Maximum Flow Algorithms.* PhD thesis, Cornell University, 1999.

This article was processed using the LaTeX macro package with LLNCS style