A Flow-based Task Scheduling Strategy for Distributed Systems

Sagnik Nandy   Jeanne Ferrante   Larry Carter

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

Abstract

This paper investigates the problem of allocating a large number of independent, equal sized tasks on a distributed grid-like platform. We develop an *efficient, autonomous, scalable, dynamic and generally applicable* protocol for this purpose. The A-FAST protocol embodies the idea of pressure guiding the flow in fluid networks. It uses the number of unprocessed tasks buffered at each node in place of "pressure" to decide whether to move tasks to neighboring nodes. Simulations show that the A-FAST protocol performs well over a wide set of random networks, averaging more than 99% of the optimal performance. Such a protocol has the potential to aid the efficient deployment of large, data intensive applications on heterogeneous peer-to-peer computing platforms.

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

1 Introduction

The advent of high speed networks has given rise to a range of applications consisting of a large set of tasks which can be distributed across a grid-like platform and solved concurrently. Some such problems include collaborative computing efforts such as SETI@home [27], factoring large numbers [7], the Mersenne Prime Search [22] and distributed computing problems organized by companies such as Entropia [10]. This forms the driving motivation of our work, which aims to schedule a large number of *independent, equal-sized* tasks, in an *online* scenario, across a *dynamic* and *heterogeneous* computing platform. Existing scheduling algorithms may be unsuitable for these scenarios owing to the large size and dynamic nature of the underlying platforms. We seek a scheduling strategy with the following properties:

- **Efficient** - The strategy should result in high overall throughput.
- **Generic** - It should be applicable to all kinds of networks, regardless of topology.
- **Scalable** - The strategy should be applicable to networks of very large size.
- **Autonomous** - It should use minimal (or no) global information. In particular it should not require network-wide information. This is extremely important for the scalability of the algorithm.
- **Dynamic** - The algorithm needs to be able to adjust to networks where, due to contention or other reasons, the bandwidths and computation speeds change over time.
- **Practical** - The strategy should be easy to implement in real life scenarios.

For the successful deployment of data intensive applications across large scale networks all the above mentioned features are desired. This is the key problem we address in this paper.

In order to handle dynamic networks scalably, we require an adaptive or incremental algorithm, rather than one that recomputes its allocations from scratch each time there is a change. The autonomic behavior of fluid networks, using pressure as a guiding force, forms the key inspiration for our work. We propose an autonomous scheduling protocol that uses the number of unprocessed tasks in a processor's buffer as an analogue to *pressure*. This pressure is used to decide when to move a task to a neighboring node. Using this idea, the protocol eliminates the need for centralized control over scheduling. Initial simulations show that the protocol achieves more than **99%** of the optimum throughput over a range of dynamic 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, Section 3 describes the protocol in detail and Section 4 discusses various implementation aspects. In Section 5 we present experimental results showing performance under various conditions and in Section 6 we discuss some future research directions in this area.

2 Related Work

Scheduling independent tasks across heterogeneous sets of resources is a well known problem [15, 2, 5, 26, 11, 19, 20, 14, 29]. We differ from previous approaches in that we develop a *distributed*, *autonomous* and *generic* scheduling strategy suitable for large-scale and dynamic computing platforms on which centralized control may not feasible.

Several research efforts have formulated this problem as a max-flow problem [6], [30]. However, the most popular max-flow algorithms, including Ford-Fulkerson [17], Edmonds-Karp [9] and Goldberg [13], use some form of global information to make network-wide decisions. The former two involves finding a path from the source to the sink, and the latter uses a notion of *height* for the source that depends on the total number of nodes in the network. There are also algorithms that solve the max-flow problem in parallel. [16] uses more processors than nodes in the system, which undesirable in practice. [28] avoids the use of additional processors, but 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.

Research has also gone into calculating max flows for changing networks. In [24], it is assumed that one knows ahead of time when and by how much the network will change. In [23] and [1], changes are detected in real time, but the algorithms require global knowledge or decision making. Thus, these algorithms aren't autonomous and dynamic in the sense that we are looking for.

Our previous work [4], [18] presented an autonomous algorithm that, when the network is a *tree*, achieves the optimum throughput for a static network, and our experiments show that the protocol reacts quickly to changes in the network. However, it is often difficult to model large generic networks as trees. [3] proved that the problem of finding the best tree from a given network is *NP-Complete* and that there exists networks for which the performance of the optimal tree can be unboundedly worse than the whole network's performance. Thus even though considerable research has gone into different variants of our problem, a definitive solution addressing all the issues is yet to be found.

3 The A-FAST Task Scheduling Protocol

We begin with a formal description of the problem. We are given a labelled, directed graph $G = (N, E, P, C)$ representing the network. Each node $N_i \epsilon N$ represents a computing resource (processor, computer, cluster etc.) of processing speed $p_i \epsilon P$, measured in tasks solved per unit time. Each edge in the graph $(N_i, N_j) \epsilon E$ is labelled with a value $c_{ij} \epsilon C$ which represents the number of tasks that can be sent across the link per unit time. All tasks are of equal size[1] and initially reside in the source node N_0. The graph (N, E, P, C) can change during execution. Nodes and edges can be added to or deleted from N and E (except for N_0, which is always present) and the values in P and C can also change. Our objective is to maximize the overall throughput of the network i.e. maximize the number of tasks completed per unit time.

The A-FAST protocol exploits the fact that incoming tasks can be buffered in a node. Our protocol uses buffer occupancy as an analog of pressure, which autonomously controls the flow of tasks. In Section 5, we will see that the use of buffers in a dynamic network can even give better results than that given by the optimal max-flow solution.

[1]Although we have not yet performed the experiments, we believe 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 ratios are different but known.

We divide task receiving, task sending and task processing as separate procedures. Nodes begin by advertising their buffer occupancy to all their immediate neighbors, requesting tasks. On receiving a request, a node compares the requester's buffer occupancy to its own to decide whether to service the request. This is similar to fluid motion where fluids flow from a region of higher pressure to a region of lower pressure. It allows us to do away with the need for a centralized scheduler, and instead make all scheduling decisions locally. If a node does not service a request, it replies to the requesting node of its decision. On being serviced by a neighbor, a node immediately requests another task. But if its request is denied, it waits for a while before making another request. 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. Since we do not have a centralized scheduler, asynchrony becomes a serious issue, and the buffer manager has to ensure that it doesn't request more tasks than it can handle. To deal with this problem, we introduce the notion of *Intermediate Buffers (IB)*. Any task or denial reply sent from N_i to N_j is sent to IB_{ij}. Tasks in IB_{ij} are then transferred to the main Task Buffer (TB_j) of the node. Figure 1 provides a high level example.

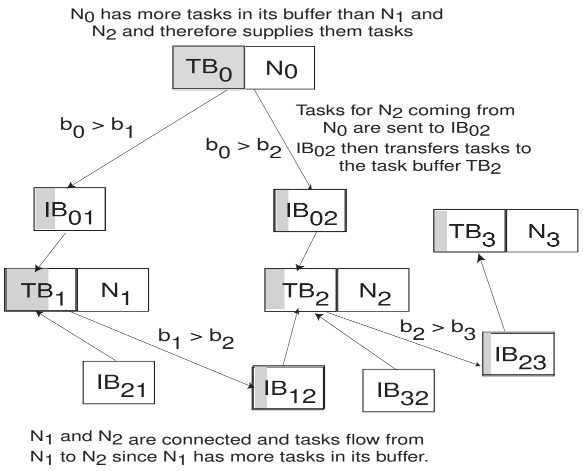
N0 has more tasks in its buffer than N1 and N2 and therefore supplies them tasks
TB0
N0
b0 > b1
b0 > b2
Tasks for N2 coming from N0 are sent to IB02 IB02 then transfers tasks to the task buffer TB2
IB01
IB02
TB3
N3
TB1
N1
TB2
N2
b1 > b2
b2 > b3
IB23
IB21
IB12
IB32
N1 and N2 are connected and tasks flow from N1 to N2 since N1 has more tasks in its buffer.

Figure 1. An example of the A-FAST protocol

3.1 The Protocol in Detail

We assume that for each communication edge $(N_i, N_j) \epsilon E$ used by our application, there is a buffer IB_{ij} residing in node N_j. Each such buffer can hold one task. Additionally, each node N_i, has a task buffer TB_i. TB_i has a capacity of m_i "*slots*", where each slot can hold one task. Each of the slots in TB_i is 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 node N_j i.e. it is being transferred from TB_i into IB_{ij}.

- **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 assume that only one task at a time can be in state **S3**. We define the *buffer occupancy*, b_i of a node to be the number of slots in state **S5** at the current time. We say "TB_i is full" when the number of slots e_i in state **S1** is zero.

Initially all tasks reside in the source node N_0. All other nodes begin by requesting tasks from their neighbors. This is also the protocol that nodes follow when they join the network - once an existing node discovers a new node, it sends it a request for a task. Similarly, when a node joins the network it sends a task request to all its neighbors. Every task request from node N_i is accompanied by its current value of b_i.

Figures 2, 3 and 4 give the protocols for requesting, receiving and performing a task in detail. Some sections of the protocols have been highlighted. These sections use the shared variables b_i and e_i, and they should be synchronized to run correctly, either by acquiring locks if each protocol is a separate process, or by doing the shaded sections atomically if a single buffer manager procedure handles all three protocols. The `Wait` primitive should be implemented using a periodic polling mechanism that prevents livelock.

```
OnRecvReqest(j, $b_j$) { // request from $N_j$
    i = CurrentNode;

    if ($b_i$ > $b_j$) { // node has more tasks than requesting node
    $b_i$ = $b_i$ - 1;
    send(task, $N_j$); // send single task to $N_j$
    $e_i$ = $e_i$ + 1;
    }
    else {
    send(refuseMsg, $N_j$); // refuse $N_j$
    }
}
```

Figure 2. Protocol nodes follow on receiving a request

```
OnRecvData(j, $m_j$) { // message from $N_j$

        i = CurrentNode;

        if ($m_j$ is a task) {
                Wait(until $e_i > 0$);  // wait for an empty slot for the task
                $e_i = e_i - 1$;
                transfer task from $IB_{ji}$ to $TB_i$ ;
                $b_i = b_i + 1$;
                requestData(j, $b_i$); // request more tasks from $N_j$
        }
        else {
                Wait(TimeToWait); // wait for a while
                requestData(j, $b_i$); // request tasks again
        }
}
```

Figure 3. Protocol nodes follow on receiving a response

```
ProcessTask() {
        i = CurrentNode;

        if ($b_i > 0$) { // $TB_i$ not empty
                dispatch task for processing;
                $b_i = b_i - 1$;
                perform task;
                $e_i = e_i + 1$;
        }
        else {
                Wait(till $b_i > 0$);
        }
}
```

Figure 4. Protocol for processing a task

3.2 Discussion of the Protocol

The basic philosophy behind our protocol is simplicity. Each node tries, by sending requests to its neighbors, to keep its Intermediate Buffers full. But a request is only satisfied if the requester has fewer tasks than the provider. Intuitively, the protocol 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 faster across it, decreasing the pressure at the provider node, leading to more tasks being sent to that node.

An interesting aspect of the protocol is that it does not use the values of *(N, E, P, C)*. Since these are the parameters that are most likely to change [12] and are difficult to predict accurately, it seems preferable not to bind the scheduling strategy strongly to them. Although there are techniques for estimating these parameters dynamically ([31], [32], [8]), these estimates may consume considerable effort and not work well for large-scale heterogeneous systems. Some strategies deal with this by using upper bound values for P and C. This can be misleading in shared networks and multi-tasked systems, where these values change rapidly over time.

Our protocol, on the other hand, works on a *supply-on-demand* basis. It uses the notion of Task Buffers and Intermediate Buffers, parameters which are completely under the control of the node. N_i sends a task to N_j only on receiving a request from N_j and similarly N_j sends a request for a new task as soon as it gets a task from N_i. Assuming the request messages are small and the latency of the link is low, the number of tasks flowing from N_i to N_j is nearly equal to the available bandwidth of the link (if the node has enough tasks to send) at that time. This might not hold true in high latency networks but can be dealt with by bundling multiple jobs together into a larger task to amortize the effect of latency (Section 5.2.2). Similarly, when a node finishes a task it starts the next task (if it is available). With sufficient amount of tasks in the TB this number will be nearly equal to the total processing power available to the node at that time.

Thus we hope that the protocol makes maximum use of the resources *if* they are available but does not bind its success to the maximum availability of resources - adjusting automatically to system changes.

4 Extending the Basic Protocol

This section discusses some of the problems the protocol might run into and possible solutions to these problems through extensions made to the basic protocol.

Since tasks get moved around the system autonomously using our protocol, it is possible for a task to keep getting moved across the system unboundedly without getting executed. This can be solved by imposing a simple priority rule based on which tasks are chosen from TB to be processed - nodes perform tasks from their TB based on the issued time of the tasks. Since all tasks originate from the source node N_0, they can be time stamped before being distributed. This does not involve any global notion of time across the network as N_0 is the only node that time-stamps tasks, giving all time-stamps a common frame of reference.

Another possible problem with our protocol is the frequent use of messages to *request* and *refuse* tasks. This is done to autonomously detect changes in flow direction and also to avoid nodes from using "stale" values of b_i. Even though message sizes are expected to be small compared to task sizes, the protocol might consume a considerable fraction of the network bandwidth in low bandwidth networks. This can be dealt with by either increasing the value of $TimeToWait$ (thereby reducing the frequency of redundant messages) or by maintaining at each node a list of neighbors it is currently sending tasks to and not sending task requests to them. Both these schemes will reduce message transfer across the system but might increase the response time of the protocol to system changes. As suggested earlier, it would also help bundle work together into larger tasks. This will however require larger buffers and might also increase the startup time and responsiveness of the protocol.

Our protocol makes use of several memory-to-memory data transfers (IB to TB, TB to the execution process etc.). In a heavily loaded system this can affect the performance and we might want to reduce the amount of copying by using existing techniques such as “container shipping” [25].

5 Experimental Results

To test the effectiveness of A-FAST, we simulated a range of networks and tested the protocol on them. The simulations were run using the basic protocol described in Section 3, with none of the extensions suggested in Section 4. The experiments provided additional insight into various aspects of the protocol, enabling us to improve its overall efficiency.

5.1 Simulation Methodology

We study various metrics that reflect the performance of our protocol. Each test case made use of the parameters - n (number of nodes in the network), c (upper limit on the bandwidth of any edge c_{ij}), l (upper limit on the latency — the time to send a request or refusal message) and p (upper limit on the processing speed p_i of any nodes), i.e. if $(N_i, N_j) \epsilon E$ then $c_{ij} \epsilon$ *[1, c]* and $l_{ij} \epsilon$ *[0, l]*. The actual values were chosen uniformly at random from the indicated range (note: for latency the actual value was the *chosen value* $\times 10^{-3}$). For each set of values, multiple experiments were run,[2] and the average of these runs are plotted.

To generate random graphs we first generated a random tree. This was done by connecting two random nodes first and thereafter connecting a node from the connected set and the unconnected set randomly till we had a connected set of n nodes. We then chose a random number between 0 and $(n-2)(n-1)/2$. This number represented the number of additional edges in the graph. These additional edges were inserted by randomly selecting two unconnected nodes and connecting them. Table 1 shows the average number of edges in the generated graphs for some of the value of n.

Number of Nodes	10	20	30	40	50	60	70	80	90	100
Average Edges	5.11	10.11	15.11	19.97	24.75	30.66	34.50	39.87	41.7	44.92

Table 1. Average edge count of the test cases

5.2 Results

This section presents the results of the simulations where we evaluate our protocol based on a number of criteria.

5.2.1 Performance in Comparison to Optimal Throughput

We ran A-FAST on randomly generated networks with n = *10 to 100.* For all the experiments *TimeToWait* was set to 0.010. To calculate the optimal throughput we used

[2]At least ten, and typically 50 to 70, randomly-generated graphs were used for each problem size.

an existing implementation of the max-flow problem [21]. In the initial experiments, we set the latency $l = 0$, so that the optimum max-flow rate might be achieved. The other parameters for the simulation were *c = 30* and *p = 30*. The results are shown in Figure 5. We see that A-FAST averaged around *99.5%* of the cumulative optimal throughput over all the runs, even including the start-up time of the protocol. The results would be even better if we had run the simulations longer, since the lower rate of execution during "start-up" would be amortized over a longer time — the graph suggests that the actual performance and the optimal differ by only an additive constant. Results are shown for n *= 20, 40, 60, 80, 100*. The results for n = *10, 30, 50 etc.* are also similar but have been omitted for clarity.

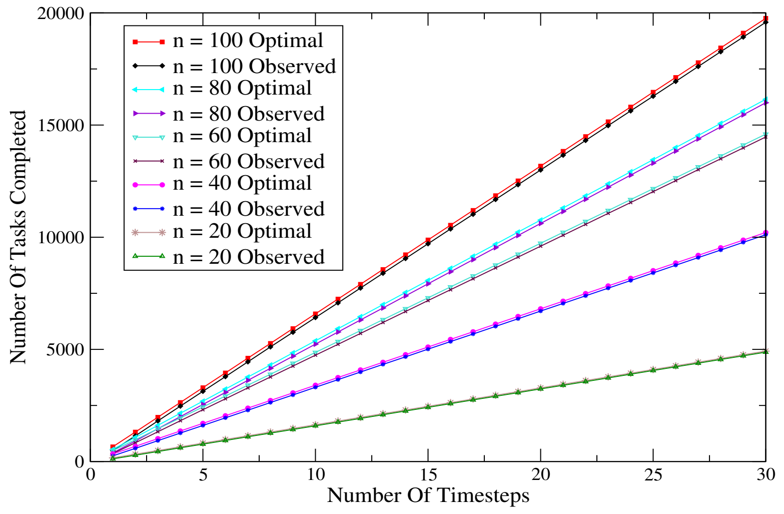

Figure 5. Performance of the algorithm compared to Optimal Max-Flow

5.2.2 Effect of Latency

The initial experiments assumed zero latency. However, in the presence of latency, the total time to first request and then transfer a task increases, thereby affecting performance. Figure 6 shows the results for various values the value of l, using *p=30* and *c=30*. Note that for the largest latency, $l = 30$, the time to send a request is chosen uniformly between 0 and 0.03. Meanwhile, the faster edges — those with c_{ij} near to 30 — require only 0.033 time to communicate a task. Thus, the overhead of requesting a message is comparable to the transfer time on the more important (faster) edges. In most practical scenarios, it should take much longer to transfer a task than a (small) request message. Nevertheless, the graph shows that there isn't a catastrophic drop in performance as latency increases. In practice, task transfer times are likely to be large compared to latency. If not, as mentioned earlier, we can bundle small jobs together to form larger tasks.

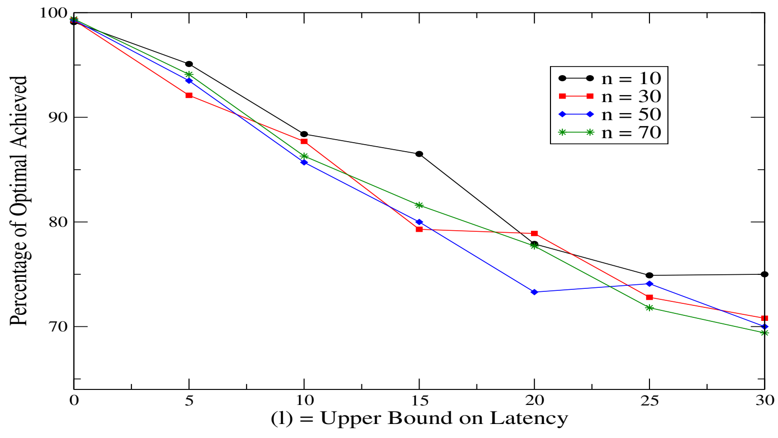

Figure 6. Effect of Latency on Performance

5.2.3 Effect of Buffer Size

Another parameter of the protocol is the maximum capacity of the Task Buffers. In our experiments, all the TB_i's (except for the source node) initially have the same number of empty slots, e_{max}. To study the effect of buffer size on performance, we ran simulations on the same *graph* (*p=30, c=30, l=0*) but varied the value of e_{max}. The results are shown in Figure 7. It is observed that the value of e_{max} needed to achieve high performance is not dependent on the problem size (for $e_{max} \approx 10$ we reach 97% of the optimal throughput for all values of n). This shows that the protocol does a good job of synchronizing communication with computation.

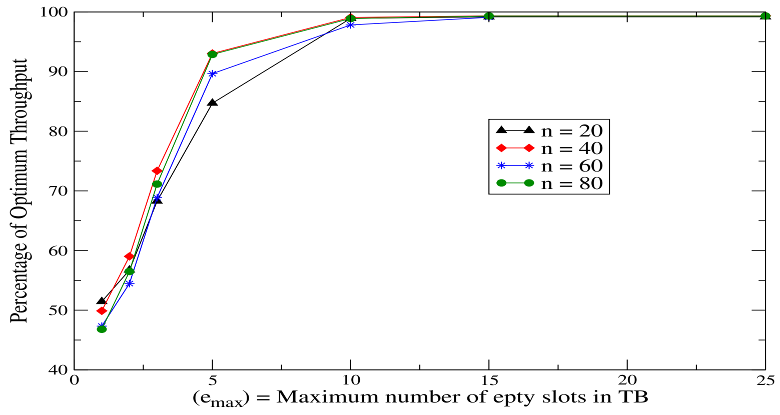

Figure 7. Effect of Buffer Size on Performance

5.2.4 Effect of System Changes

To study how the protocol identifies and adapts to changes in the system, we ran the simulations for 20 timesteps and then changed the system parameters. We increased or decreased the speeds of nodes and edges with a probability of .30. The magnitude of these changes were a random percentage uniformly distributed between 0 and 40% of their original values. Figure 8 shows that the protocol successfully and quickly adapts, without external information and intervention, to these changes.

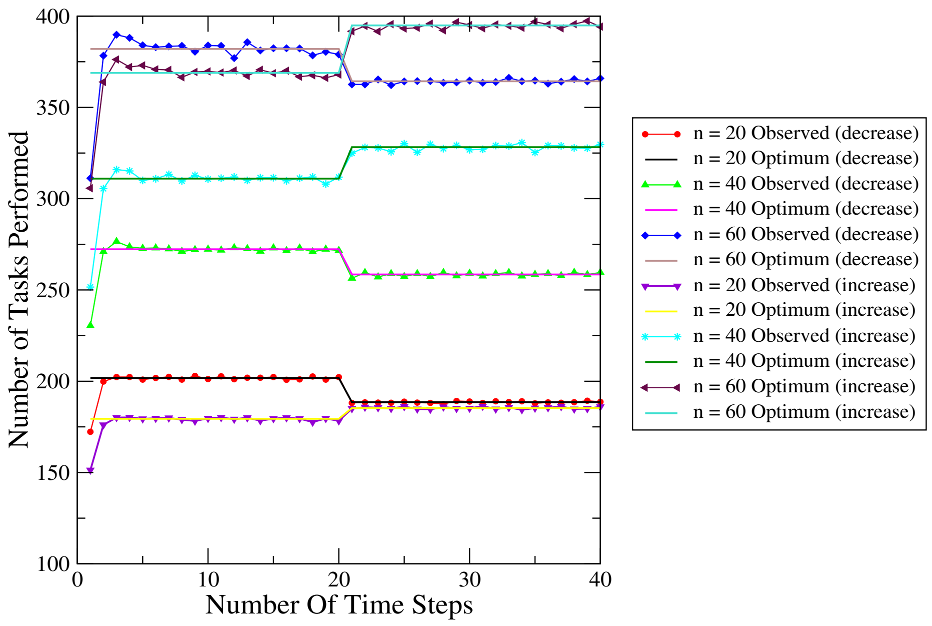

Figure 8. Effect of single set of changes in the system

Real systems properties, however, might change continuously. To study the effect of continuous changes on the protocol we changed the system every 4 timesteps. Nodes and links had a probability of 0.40 of changing. The magnitude of these changes varied from 10% to 99%, with equal probability of increasing or decreasing. The average throughput over the entire run for a given percentage change was calculated. We also recorded the minimum and maximum throughput as a percentage of the optimal (found by summing the optimal max-flow of each time period) over all the runs to give an idea of the best and worst cases of the protocol in dynamic systems. Figure 9 and Table 2 show these results.

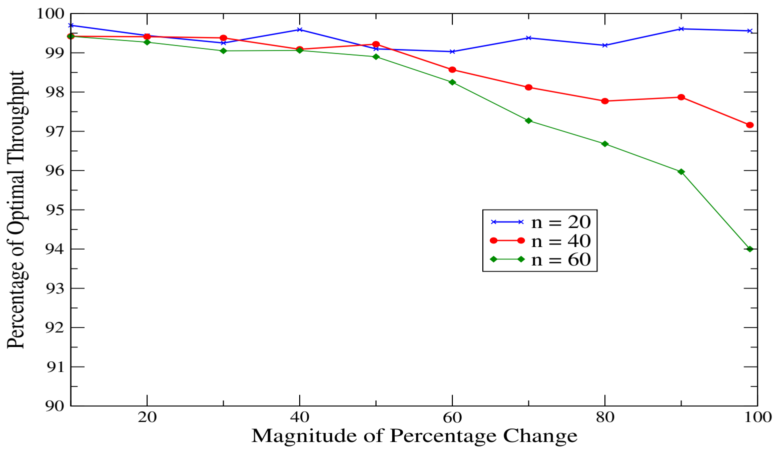

Figure 9. Effect of rapid and continuous set of changes in the system

P	Min(n=20)	Max(n=20)	Min(n=40)	Max(n = 40)	Min(n=60)	Max(n=60)
10	98.71	101.11	98.81	99.70	99.01	99.82
20	98.92	100.97	98.97	99.89	98.35	99.79
30	98.10	101.12	98.39	99.90	98.51	99.72
40	98.89	104.53	97.98	100.62	98.45	100.19
50	97.64	100.64	97.91	104.46	97.74	100.12
60	96.72	101.65	97.05	99.60	97.15	100.03
70	97.31	107.64	96.00	101.51	93.67	99.94
80	91.26	112.42	94.77	105.69	91.54	99.58
90	96.44	110.87	93.56	105.05	92.34	98.86
99	94.84	**113.16**	91.02	106.44	**84.86**	100.50

Table 2. Observed minimum and maximum overall throughput

It is observed that there can be a drop in performance for continuous and sudden large changes. This happens because continuous large changes modify the system behavior almost completely. Our protocol normally takes around 3-4 timesteps to stabilize. Since we change the underlying system drastically every 4 timesteps it is almost like starting anew. However, it is seldom the case that real networks change by such large magnitudes. Even for rapid changes on the order of 99%, our protocol manages to achieve an overall throughput of around 94%. For rapid changes up to magnitudes of 50% the protocol achieves more than 99% of the optimal throughput.

An interesting observation can be made from the table: there are several instances of the protocol significantly outperforming the max-flow solution. This happens because none of the solutions to the max-flow problem use buffers as a task repository. Only the exact amount that can be consumed by a node is transferred, which may not utilize some links completely. With our protocol, links are completely utilized (if there is empty buffer space) and so additional tasks get stored in the TB. If the performance of the link drops then the tasks from the TB can be used to give the impression that the link is still running at a higher speed. Consider the 2-node network in Figure 11. Since N_1 performs 10 tasks/sec, a max-flow based scheduling approach would only send 10 tasks/sec across the link. Our protocol, however would utilize the link completely and send 15 tasks/sec as long as TB_N is not full. Now assume that the link speed drops to 5 tasks/sec after some time. The max-flow solution would now send 5 tasks/sec and **N** will perform 5 tasks/sec. However, using our protocol, **N** will access tasks from TB_N, allowing it to perform 10 tasks/timestep for some additional time (based on capacity of TB_N). Thus the overall throughput using our protocol will be higher than that of a max-flow like approach. This supports our claim that buffers should be incorporated in scheduling strategies.

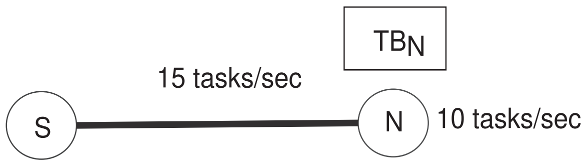

Figure 11. Sample network

6 Conclusion and Future Work

This paper suggests a new scheduling protocol that is efficient and can scale and autonomously adjust in dynamic heterogeneous networks. We have discussed how the protocol can be implemented efficiently in practice and how various parameters can be adjusted to suit specific needs. Experimental results were provided to show the effect of various parameters on the functioning of the protocol. Simulations showed the protocol to be efficient, achieving more than 99% of the optimal 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 looking into 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, the issue of fault tolerance etc. Finally, we would like to implement the protocol in practice and evaluate it on real life networks and systems.

References

[1] Yehuda Afek, Baruch Awerbuch, and Eli Gafni. Applying Static Network Protocols to Dynamic Networks. In *In Proc. 28th IEEE Symp. on Foundations of Computer Science*, pages 358–370, October 1987.

[2] 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)*, pages 301–308, July 1998.

[3] C. Banino, O. Beaumont, A. Legrand, and Y. Robert. Scheduling strategies for master-slave tasking on heterogeneous processor grids. Technical Report RR2002-12, ENS-Lyon, LIP, 2002.

[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] 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)*, pages 349–363, May 2000.

[6] T. Cormen, C. Leiserson, and R. Rivest. Introduction to Algorithms, MIT Press. pages 579–615, 1990.

[7] J. Cowie, B. Dodson, R. Elkenbrach-Huizing, , A. K. Lenstra, P.L. Montgomery, and J. Zayer. A World Wide Number Field Sieve Factoring Record: On to 512 Bits. *Advances in Cryptology*, pages 382–394, 1996. Volume 1163 of LNCS.

[8] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid Information Services for Distributed Resource Sharing. In *10 th IEEE Symp. On High Performance Distributed Computing*, 2001.

[9] Jack Edmonds and Richard M. Karp. Theoretical improvements in the algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, 19:248–264, 1972.

[10] Entropia Inc. http://www.entropia.com, 2001.

[11] 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)*, pages 318–328, Jun 1996.

[12] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[13] Andrew V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers.* PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1987.

[14] T. Hagerup. Allocating Independent Tasks to Parallel Processors: An Experimental Study. *Journal of Parallel and Distributed Computing*, 47:185–197, 1997.

[15] O. H. Ibarra and C. E. Kim. Heuristic algorithms for scheduling independent tasks on non-identical processors. *Journal of the ACM (JACM)*, 24(2):280–289, 1997.

[16] Donald B. Johnson. Parallel algorithms for minimum cuts and maximum flows in planar networks. *Journal of the ACM (JACM)*, 34(4):950–967, 1987.

[17] L. R. Ford Jr. and D. R. Fulkerson. Flow in Networks, Princeton University Press. 1962.

[18] 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. To appear.

[19] C.P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, 11:1001–1016, 1984.

[20] 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.

[21] Max-Flow Solution. http://elib.zib.de/pub/Packages/mathprog/maxflow/index.html.

[22] Mercenne Prime Search. http://www.mercenne.com.

[23] N. Nagy. The maximum flow problem: A real-time approach. Master's thesis, Queen's University, 2001.

[24] A. Orda and R. Rom. On Continuous Network Flows. In *Operations Research Letters*, pages 27–36, February 1995.

[25] J. Pasquale, E. Anderson, and P. K. Muller. Container shipping: Operating system support for I/O-intensive applications. In *IEEE Computer*, volume 27, pages 84–93, March 1994.

[26] 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*, pages 124–131, October 2001.

[27] SETI@home. http://setiathome.ssl.berkeley.edu, 2001.

[28] Y. Shiloach and U. Vishkin. An $O(n^2 log\ n)$ parallel max-flow algorithm. *Journal of Algorithms*, (3):128–146, 1982.

[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):7–17, January 2003.

[30] Kevin Daniel Wayne. *Generalized Maximum Flow Algorithms.* PhD thesis, Cornell University, 1999.

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

[32] Richard Wolski, Neil T. Spring, and Jim Hayes. Predicting the CPU availability of time-shared unix systems on the computational grid. *Cluster Computing*, 3(4):293–301, 2000.