Second-Order Methods for Distributed Approximate Single- and Multicommodity Flow

S. Muthukrishnan[1] and **Torsten Suel**[2,3]

[1] Bell Laboratories, 700 Mountain Avenue, Murray Hill, NJ 07974.
muthu@research.bell-labs.com.
[2] Polytechnic University, Six MetroTech Center, Brooklyn, NY 11201.
suel@photon.poly.edu.
[3] This work was done while the author was at Bell Labs.

Abstract. We study local-control algorithms for maximum flow and multicommodity flow problems in distributed networks. We propose a second-order method for accelerating the convergence of the "first-order" distributed algorithms recently proposed by Awerbuch and Leighton. Our experimental study shows that second-order methods are significantly faster than the first-order methods for approximate single- and multicommodity flow problems. Furthermore, our experimental study gives valuable insights into the diffusive processes that underly these local-control algorithms; this leads us to identify many open technical problems for theoretical study.

1 Introduction

The *multicommodity flow problem* is the problem of simultaneously shipping multiple commodities through a capacitated network such that the total amount of flow on each edge is no more than the capacity of the edge. Each commodity i has a source node, a sink node, and an associated *demand* d_i, which is the amount of that commodity that must be shipped from its source to its sink. The objective is to find a flow that meets the individual demands of all the commodities without exceeding any edge capacity (finding a *feasible flow*)[4]. The case when there is only a single commodity and the goal is to maximize the feasible flow is the well known *maximum flow problem*. The importance of the single- and multicommodity flow problems need hardly be stressed – a substantial body of work in Algorithms and Operations Research is devoted to these problems.

In this paper, we focus on local-control (or distributed) algorithms for the single- and multicommodity flow problems. Besides their inherent interest, local-control algorithms for these problems are relevant because of the following reasons:

(1) Many routing, communication, and flow-control problems between multiple senders and receivers, including various uni/broad/multicasts, can be modeled as multicommodity flow problems on networks (e.g., see the references in [BG91, BT89, AL93, AL94, AAB97]). These applications typically require online, local-control (distributed) algorithms, since global communication and control is expensive and cumbersome. Local algorithms for

[4] An alternate objective is to maximize z such that the flow satisfies a percentage z of every demand without exceeding any edge capacity (called the *concurrent flow problem* [SM86]); we do not consider this version here.

multicommodity flow not only provide a generic solution to these problems, but they also give valuable insights for the centralized/global solution of these problems.

(2) The best currently known algorithms for maximum flow and multicommodity flow problems are fairly sophisticated (see, e.g., [GR97, GT88, K97, LM+91, V89]), and typically rely on augmenting paths, blocking flows, min-cost flows, or linear programming. In contrast, local-control algorithms are appealingly simple, as they rely on simple "edge balancing" strategies of appropriately balancing commodities between adjacent nodes (details below). Thus, they are easy to implement, understand, and experiment with.

(3) Local-control algorithms have several other attractive features. For example, they adjust gracefully to dynamic changes in the topology (e.g., link failures) and the traffic demands (e.g., bursty multicasts) in communication networks. They are *iterative*, that is, running them longer gives progressively better approximations to the optimal solution. Hence, one can use them either for rapid coarse solutions or for slow refined solutions. Finally, they may expose alternate structure in the problem, as the convergence of such local-control algorithms is typically related to the eigenstructure of the network (for intuition, see [C89]).

1.1 First-Order Algorithms

Local-control algorithms for the multicommodity flow problem were recently designed by Awerbuch and Leighton [AL93, AL94]. Their algorithms proceed in parallel rounds. At the start of a round, (approximately) d_i units of commodity i are added to the source node of that commodity, where d_i is the demand of commodity i. The commodities accumulated in each node are then distributed equally among the local endpoints of the incident edges, and flow is pushed across each edge of the network so as to "balance" each commodity between the two endpoints of the edge (subject to edge capacity constraints). Finally, any commodity that has arrived at the appropriate sink is removed from the network. How to trade off the flow between different commodities that compete for the capacity of an edge is nontrivial. Awerbuch and Leighton proved in [AL93, AL94] that this simple "edge balancing" algorithm (and some of its variants) converges and, maybe somewhat surprisingly, that it provides a provably approximate solution to the multicommodity flow problem in a small number of rounds.

We refer to such edge-balancing algorithms as *first-order algorithms*. The first-order algorithms in [AL93, AL94] can clearly be implemented on a distributed network in which each node communicates only with neighboring nodes and has no global knowledge of the network.[5] Similar local-control algorithms have been designed for several other problems [LW95], including distributed load balancing [C89, AA+93, MGS98] and end-to-end communication [AMS89].

A particularly simple local-control algorithm can be obtained for the case of the maximum flow problem by specializing the first-order algorithm in [AL93, AL94] for the single-commodity case. There are many other algorithms for the maximum flow problem, but none that is a distributed first-order algorithm. The algorithm most closely related in spirit is the algorithm of Goldberg and Tarjan in [GT88], where a "preflow" is adjusted into a flow by pushing excess local flow towards the sink along estimated shortest paths. However, this algorithm needs to maintain estimated shortest-path information and is thus less amenable to a distributed, local-control implementation in dynamic networks.

[5] In contrast, other approximation algorithms for the multicommodity flow problem rely on global computations [V89, LM+91].

1.2 Second-Order Algorithms

In this paper, we initiate a new direction in distributed flow algorithms aimed at speeding up the first-order algorithms of [AL93, AL94] for the multicommodity flow problem. The basic idea is that in any round, we use the knowledge of the amount of flow that was sent across the edge in the previous round in order to appropriately adjust the flow sent in the current round. Specifically, for a parameter β, the flow sent across an edge is chosen as β times what would be sent by the first-order algorithm, plus $\beta - 1$ times what was actually sent across the edge in the previous round. (A more detailed description of these methods is given in Sections 3 and 4.)

We call algorithms derived in this manner *second-order algorithms*. Perhaps surprisingly, the main conclusion of this paper is that second-order algorithms appear to substantially outperform their first-order counterparts for maximum flow and multicommodity flow problems, as shown by our experiments.

1.3 Background and Related Work

First-Order methods. The first-order algorithm of Awerbuch and Leighton for the maximum flow problem is conceptually similar to the probabilistic phenomena of diffusion and random walks. The algorithm works based on diffusion since the excess flow always flows down the gradient along each edge. For simpler problems such as distributed load balancing, if one considers the vector of flows accumulated at the nodes as iterations progress, they can be modeled as transitions of a Markov Chain, or a suitable random walk [C89]. However, for the general multicommodity flow problem, these conceptual similarities have not yet been formalized. The analysis of Awerbuch and Leighton is sophisticated even for the case of the maximum flow problem. It does not rely on Markov Chain methods, and is entirely combinatorial.

First-order algorithms for flow problems are also related to matrix-iterative methods for solving linear systems, and in particular, the Gauss-Seidel iterations. This connection is made explicit in [BT89]. Also, there is a way to interpret the first-order algorithms as iteratively solving a dual network optimization problem involving a single variable per node. At each iteration, the dual variables of a single node or its incident edge flows are changed in an attempt to improve the dual cost. This process is also explained in [BT89].

Thus, there are intriguing connections between the first-order methods for flow problems and classical techniques such as matrix-iterative methods, diffusion, random walks and primal-dual relaxations. These techniques have been studied in different areas with somewhat different emphasis, but seem directly relevant to the work in [AL93, AL94].

Second-Order methods. Second-order algorithms, as described above, may seem ad-hoc, and further explanation is needed to motivate them. Our second-order algorithms are motivated by the observation that the first-order flow algorithms in [AL93] are iterative methods reminiscent of the matrix-iterative methods used for solving systems of linear equations. There is already a mature body of knowledge about speeding up these first-order methods (see, e.g., [A94, BB+93, HY81, Var62]). Very recently, these methods were explored for speeding up diffusive load-balancing schemes [MGS98]. Of the many known iterative techniques, the authors in [MGS98] identified a specific second-order scheme best suited for distributed implementations, and our second-order scheme for the multicommodity flow problem is inspired by that method.

There are fundamental similarities between our work here and the work in [MGS98] for distributed load balancing, but there are fundamental differences as well. The basic similarity is that our algorithmic strategy for second-order methods relies on the same stationary acceleration of the first-order method determined by a parameter β (fixed throughout all iterations) as that in [MGS98]. The main difference arises in the fact that the problem of multicommodity flow is much more general than the distributed load-balancing problem considered in [MGS98]. First, the edges in our problem have capacity constraints, while the edge capacities are unbounded in the load-balancing problem. Second, our algorithms are *dynamic* in that they introduce new flow in each round as described in Section 3; in contrast, the total load remains unchanged in [MGS98]. There are other differences (such as the fact that we do not use *IOUs* as in [MGS98]), but we omit these details.

The similarity of iterative flow algorithms to matrix-iterative methods and distributed load balancing is helpful. In particular, known results [Var62] show that $0 < \beta < 2$ is the *only* suitable range for the convergence of that iterative method. Furthermore, from the results in [MGS98], we would expect that the second-order method will be outperformed by the first-order method for $0 < \beta < 1$, and thus the fruitful range for β is $(1, 2)$; as we will see, this also holds for distributed flow problems.[6] However, the above mentioned differences explain the considerable difficulty in analyzing the first-order and second-order method for the multicommodity flow problem [AL93, AL94]. The first-order method for distributed load balancing can be analyzed fairly easily based on stationary Markov Chain methods [C89], and known second-order analyses for matrix-iterative methods can be fairly easily adopted to load balancing [MGS98]. However, standard approaches (e.g., based on Dirichlet boundary conditions [C97] for analyzing dynamic situations) do not seem to apply if edges have capacity constraints.

1.4 Contents of this Paper

In this paper, we propose second-order methods for accelerating the distributed flow algorithms proposed by Awerbuch and Leighton [AL93, AL94]. We perform an experimental study and show that the second-order algorithms are significantly faster than the first-order ones of [AL93, AL94] both for the maximum flow and the multicommodity flow problems. This is of possible applied interest as an online distributed solution for many routing problems arising in communication networks. Surprisingly, our algorithms seem to be of interest in the off-line, centralized context as well. While our algorithms are not as fast as the best known algorithms for the maximum flow problem, they seem to be at least competitive with (and possibly much faster than) the best known algorithms for the approximate multicommodity flow problem. This is a bit surprising since the best known centralized algorithms for the multicommodity flow problem [LM+91] use sophisticated techniques; in contrast, the first-order and second-order algorithms are exceedingly simple.

Our experimental study also leads to a number of observations and conjectures about the behavior of the diffusive processes used in the first- and second-order flow algorithms. We describe some of these as open problems for theoretical study.

[6] See [MGS98, Var62] for results on choosing the “best” value of β, and [DMN97] for choosing the best β for distributed load-balancing as a function of the graph structure. We plan to perform an experimental study of the best choices of β for flow problems on different classes of input graphs in the near future.

The remainder of this paper is organized as follows. The next section provides some definitions and notations used throughout the paper. Section 3 describes the first-order and second-order methods for the maximum flow problem, and presents a variety of experimental results. These results also give intuition to the reader about the behavior of first- and second-order algorithms for flow problems. Section 4 describes the algorithms and experimental results for the case of multicommodity flow, and they are more interesting in terms of comparative performance. A few open questions appear in Section 5.

We have a fully functional implementation with a graphical interface for vizualizing the behavior of our algorithms. Some additional information about our implementation and the input instances used in our experiments is contained in the appendix.

2 Preliminaries

Throughout this paper, we assume a network (or graph) $G = (V, E)$ with n nodes and m edges. We assume a model of the graph in which each edge e in the network has one capacity $c_1(e) \geq 0$ in one direction, and another capacity $c_2(e) \geq 0$ in the other direction.[7] Each node v has one queue for each incident edge. This queue can hold an unbounded amount of flow (or commodity), and should be considered as being located at the endpoint v of the edge.

In the case of the maximum flow problem, we are given a source node s and a sink node t, and our goal is to maximize the flow between s and t. In the multicommodity flow problem with k commodities, we are given k source/sink pairs (s_i, t_i) and corresponding demands d_i, and we are interested in finding a flow that satisfies the demands of all commodities, if such a flow exists. In the description of the algorithms, we use $\Delta_i(e)$ (or $\Delta(e)$ in the single-commodity case) to denote the difference between the amounts of commodity i located in the queues at the two endpoints of edge e.

3 Maximum Flow

In this section, we focus on the maximum flow problem. This special case of the multicommodity flow problem leads to particularly simple and efficient versions of the first-order and second-order methods. In the first subsection, we describe the first-order local-control algorithm for maximum flow. In Subsection 3.2 we explain our new second-order method, while Subsection 3.3 presents and discusses our experimental results.

3.1 First-Order Distributed Maximum Flow

We now describe the first-order algorithm for maximum flow. The algorithm proceeds in a number of synchronous parallel rounds (or iterations), where in each round, a small set of elementary operations is performed in each node and each edge of the network. In particular, each round consists of the following steps.

[7] Thus, each edge is equivalent to two directed edges with their own capacities $c_1(e)$ and $c_2(e)$. However, our algorithms and implementations also extend to a graph model where the capacity of each edge is shared between the two directions.

(1) Add d units of flow to the source node, where d is chosen as the sum of the capacities of the outgoing edges (or some other upper bound on the value of the maximum flow).
(2) In each node v, partition the flow that is currently in the node evenly among the $\delta(v)$ local queues of the $\delta(v)$ incident edges.
(3) In each edge e, attempt to balance the amount of commodity between the two queues at the endpoints of the edge, by routing $min\{\frac{\Delta(e)}{2}, c(e)\}$ units of flow across the edge, where $\Delta(e)$ is the difference in the amount of flow between the two queues, and $c(e)$ is the capacity of the edge in the direction from the fuller to the emptier queue.
(4) Remove all flow that has reached the sink from the network.

We point out that this algorithm is a simplified version of the algorithm in [AL93] for the single-commodity case; the simplification results from the fact that we do not have to resolve any contention between different commodities. One consequence is that the algorithm correctly finds the maximum flow even if d is much larger than the value of that flow, that is, the algorithm does not rely on the existence of a feasible flow of value d.

3.2 Second-Order Distributed Maximum Flow

We now describe how to obtain a second-order method for distributed maximum flow. As already mentioned in the introduction, the second-order method computes the flow to be sent across an edge in the current round as a linear combination of the flow that would be sent according to the first-order method and the flow that was sent in the previous iteration. The second-order method has an additional parameter β, with the case $\beta = 1.0$ being identical to the first-order method. More precisely, Step (3) of the above algorithm becomes:

(3a) In each edge e, compute the desired flow across the edge as

$$f = \beta \cdot \frac{\Delta(e)}{2} + (\beta - 1) \cdot f',$$

where $\Delta(e)$ is defined as before, and f' is the (possibly negative) amount of flow that was sent in the direction of the imbalance, in the previous iteration.
(3b) Obtain the amount of flow actually sent across the edge by adjusting f for the capacity of the edge, and for the amount of commodity available at the sending queue.

Note that the value of f computed in Step (3a) can not only exceed the available edge capacity, but may also be larger than the amount of commodity available at the sending queue.

Idealized and Realistic Versions. We distinguish two cases depending on how Step (3b) is handled if the amount of commodity available at the sending queue is smaller than the flow to be sent across that edge as calculated in Step (3a). In the *idealized* algorithm, we treat the flow accumulated at each node as just some (possibly negative) number, and we send out as much flow as the capacity constraint permits even if the amount of commodity stored at a sending queue becomes negative as a result. In the *realistic* algorithm, we treat the flows as physical flows and therefore, flows at nodes may only be non-negative. Thus, we send out the minimum of the flow calculated in Step (3a), the capacity of the edge, and the flow in the sending queue.

We expect the idealistic algorithm to converge faster, and in general, have smoother convergence properties than the realistic algorithm. In order to solve the standard sequential maximum flow problem, it suffices to implement the idealized case. However, if we want to solve the flow problem online in a distributed environment as flow continuously enters the source, the realistic algorithm must be employed. In what follows, our experimental results are for the realistic algorithm unless stated otherwise.

3.3 Experimental Evaluation

In this subsection, we present a number of experimental results on the behavior of the first-order and second-order methods. Due to space constraints, we cannot hope to provide a detailed study of the behavior of the methods on different classes of input graphs. Instead, we present a few selected results that illustrate the most interesting aspects of the behavior of the algorithm, and provide a brief summary of other results at the end. Some information about our implementation, and about the graphs used in the experiments, can be found in the appendix.

Dependence on β We first look at the performance of the second-order method for different values of the parameter β. Figure 1 shows the flow arriving at the sink in each time step, for several choices of β ranging from 1.0 to 1.95, using a 20-level mesh graph with 402 nodes and 1180 edges. The results in Figure 1 show that the rate of convergence increases significantly as we increase β from 1.0 to 1.95. In particular, after 1500 iterations, the first-order method ($\beta = 1.0$) is still more than 10% away from the exact solution. In contrast, the second-order method with $\beta = 1.95$ has already converged to within 0.001%, and with a few thousand more iterations it reaches essentially floating point precision.

Figure 2 shows the behavior of the algorithms for very small and very large values of β. In particular, we see that for $\beta = 0.5$ the performance of the algorithm becomes even worse than in the first-order method, while for $\beta = 2.5$, the method becomes unstable, and does not converge to a final value. We point out that we observed a similar overall behavior on all the graphs that we tested, with very rapid convergence for the best values of β (usually, but not always, around 1.9), slower convergence for smaller values of β, and instability as we increase β beyond 2.0.

In general, the "optimal" β, namely, the one that gives the fastest convergence is probably a complex function of the eigenstructure of the underlying graph. This is provably the case in second-order methods for the distributed load balancing problem [MGS98]. Although in many of the examples we show here, the optimal β is large (around 1.95), there are cases when a smaller value of β is preferable; see Section 4.2 for one such example.

Convergence of Edge Flows The results in Figure 1 indicate a very rapid convergence of the amount of flow that arrives at the sink. However, this does not directly imply that all the flows inside the network converge to a steady state. To investigate whether this is the case, we define the *flow change norm* as the sum, over all edges, of the absolute value of the change in flow between the current and the previous iteration. Thus, if this norm converges to zero, then the network converges to a steady flow state.

Figure 3 shows the behavior of this norm for β equal to 1.0, 1.5, and 1.95, for the mesh graph considered before. As can be seen, the flow change norm converges to zero. Convergence is again most rapid for values of β around 1.9. Note that for the first 150 or so iterations, the flow change norm for $\beta = 1.95$ is actually larger than that of the other curves, indicating a

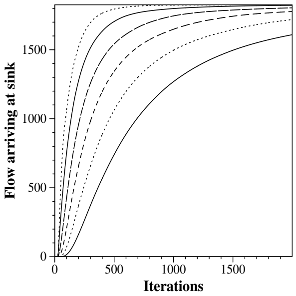

Fig. 1. Convergence of the second-order method with β set to 1.0 (lower curve), 1.2, 1.4, 1.6, 1.8, and 1.95 (upper curve).

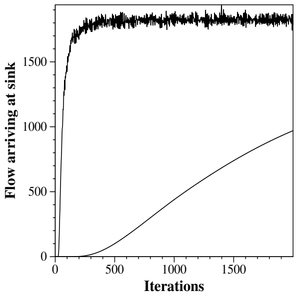

Fig. 2. Behavior for $\beta = 2.5$ (upper curve) and $\beta = 0.5$ (lower curve).

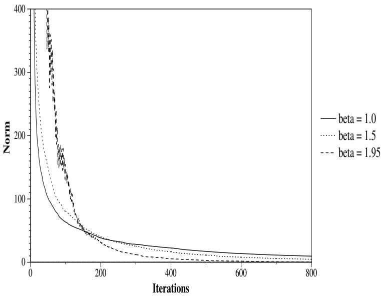

Fig. 3. Convergence of the flow change norm for β equal to 1.0, 1.5, and 1.95.

faster initial response to the injected flow. A similar rapid convergence behavior of the flows was observed in all our experiments.

The convergence of the flows is significant because it allows us to directly use the stabilized flow in the network as an approximate solution for the standard offline maximum flow problem, instead of computing the flow by averaging out the history of the edge flows, as suggested in [AL93]. Averaging the history implies the algorithm must be run for a much longer period to obtain a good approximation since the approximation ratio is then given by the ratio of the area under the curve and the area under the horizontal line at the height of the maximum flow.

Idealized Second-Order Method Recall that in Step (3b) of the second-order method, we may have to adjust the amount of flow sent across an edge in order to avoid getting a negative amount of commodity in the sending queue. In the following, we investigate how the behavior of the algorithm changes if we allow negative amounts of commodity at the nodes, that is, we consider the idealized second-order method described in Subsection 3.2, which does not adjust the flow for the amount of available commodity.

Figure 4 shows the convergence of the idealized and realistic methods for different values of β, for the mesh graph considered before. Note that for $\beta = 1.95$, the flow converges to more than 15 digits of accuracy in less than 1000 iterations. If we increase β further towards 2.0 we notice that the flow starts oscillating more extremely, and for values beyond 2.0 the method does not converge anymore. Figure 5 shows the behavior of the idealized method for the case of $\beta = 2.0$. (For the realistic method, this effect appears to be slightly less abrupt in that the method becomes instable more slowly as we increase β beyond 2.0.)

Note that whether allowing negative amounts of commodity at the nodes is appropriate or not depends on the particular application. If the goal is just to find a solution to the maximum flow problem, and the actual routing of the commodities is done in a separate phase afterwards, then the idealized version is fine. On the other hand, a major advantage of the distributed methods is that they overlap the process of finding the flow paths with that of routing the commodities, in which case the idealized version is not appropriate.

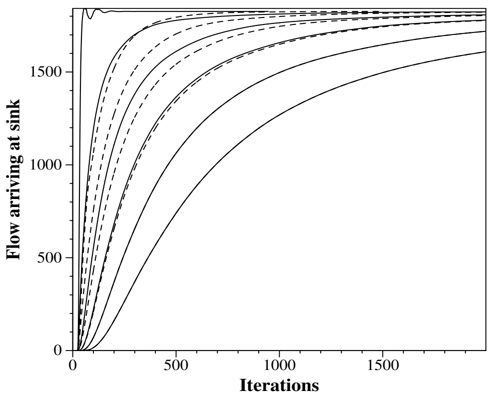

Fig. 4. Convergence of the idealized (solid lines) and realistic (dashed lines) second-order method with β equal to 1.0 (lower curve), 1.2, 1.4, 1.6, 1.8, and 1.95 (upper curve). Note that the two lowest dashed curves are hidden by the corresponding solid curves.

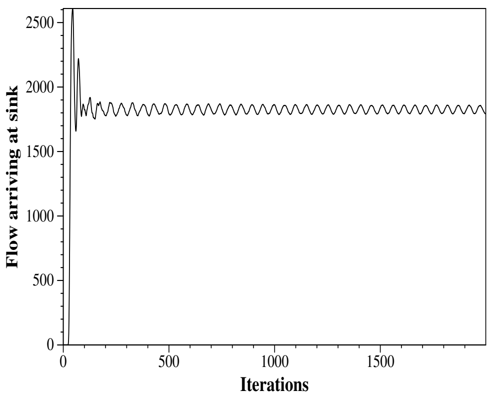

Fig. 5. Behavior of the idealized second-order method with $\beta = 2.0$.

4 Multicommodity Flow

In this section, we consider the case of multiple commodities. We first outline the first-order algorithm, which is a slightly simplified version[8] of the algorithm proposed by Awerbuch and Leighton [AL93], and describe the modifications needed for the second-order method. We then present our experimental results.

4.1 Description of the Algorithms

As in the single-commodity case, the algorithm proceeds in parallel rounds (or iterations). In our first-order implementation, the following operations are performed in each round.

(1) Add d_i units of commodity i to source node s_i, for $0 \leq i < k$.
(2) For each node v and each commodity i, partition the amount of commodity i that is currently in node v evenly among the $\delta(v)$ local queues of the $\delta(v)$ incident edges.
(3) In each edge e, attempt to balance the amount of each commodity between the two queues at the endpoints of the edge, subject to the capacity constraint of the edge. Several commodities may be contending for the capacity of the edge; this contention is resolved in the following way:
Let $\Delta_i(e)$ be the difference in the amount of commodity i between the two queues at the endpoints of edge e. The flow f_i for commodity i is computed from the d_i, $\Delta_i(e)$, and the edge capacity by using the algorithm described in Section 2.4.1 of [AL93], the details of which are omitted here.
(4) Remove from the network any commodity that has reached the appropriate sink.

[8] In particular, we get rid of the ϵ terms needed for the analysis in [AL93].

The second-order method can again be obtained with only a minor change in the algorithm. In particular, we compute

$$\Delta'_i(e) = \beta \cdot \Delta_i(e) + 2.0 \cdot (\beta - 1) \cdot f'_i,$$

where f'_i is the amount of commodity i sent across the edge in the previous iteration. In the non-idealized version of the algorithm, where we do not allow negative amounts of commodity, we also have to adjust $\Delta'_i(e)$ if $\frac{\Delta'_i(e)}{2}$ is larger than the amount of commodity i available in the sending queue; this leads to the idealized and realistic case as with the maximum flow problem. We then apply the same algorithm as in the first-order method to resolve contention between the different commodities, but use the $\Delta'_i(e)$ in place of the $\Delta_i(e)$.

4.2 Experimental Results

We now present experimental results on the performance of the second-order method. Due to space constraints, we can only give a few selected results.

Sample Performance Results. Figure 6 shows the behavior of the idealized second-order method with $\beta = 1.95$ on a $5 \times 5 \times 20$ RMF graph with 5 sources and sinks selected at random from the nodes in the first and last level of the graph, respectively. The demands for the flows were chosen such that the flow is feasible, but within about 2% of the upper bound given by the maximum concurrent flow. Figure 6 shows the 5 flows converging to their respective demands. After about 4500 iterations, all flows have converged to within 16 digits of precision. In contrast, if we use the first-order method on this problem, then we need more than 10000 iterations to converge to within 10% of the demands.

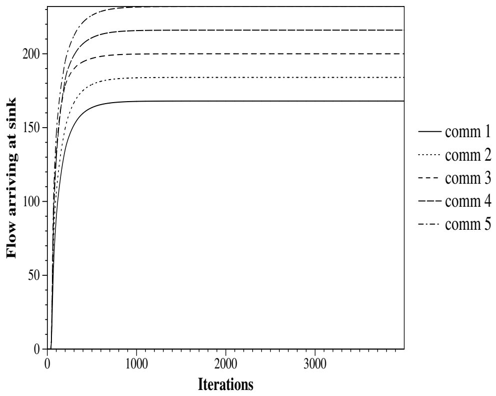

Fig. 6. Convergence of the idealized second-order method with $\beta = 1.95$ on an RMF graph with five commodities.

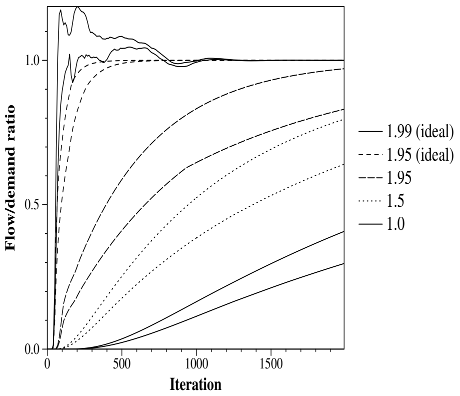

Fig. 7. Convergence of the realistic and idealized second-order methods with different values of β, on a 500 node RMF graph with 25 commodities. For each case, we plot the maximum and minimum flow/demand ratios over all commodities.

Figure 7 shows the behavior of the second-order method for a $5 \times 5 \times 20$ RMF graph with 25 commodities routed between the first and the last layer of the graph, with demands chosen at random and then scaled such that they are within 1% of the maximum concurrent flow. The values measured on the y-axis are the minimum and maximum fractions z, over all commodities, such that z times the demand of a commodity arrives at its sink in a given step. Figure 7 shows the convergence behavior for the realistic second-order method with $\beta = 1.0$, 1.5, and 1.95, and for the idealized second-order method with $\beta = 1.95$ and 1.99. The figure shows a clear advantage of the second-order over the first-order method, and of the idealized over the realistic method.

Dependence on β. The behavior of the second-order multicommodity flow algorithms for varying values of β turned out to be similar to that of the second-order maximum flow algorithm. While for most of our input graphs the optimal value of β was between 1.95 and 1.99, there are other classes of graphs where the optimal value is significantly smaller; see Figures 8 and 9 for an example.

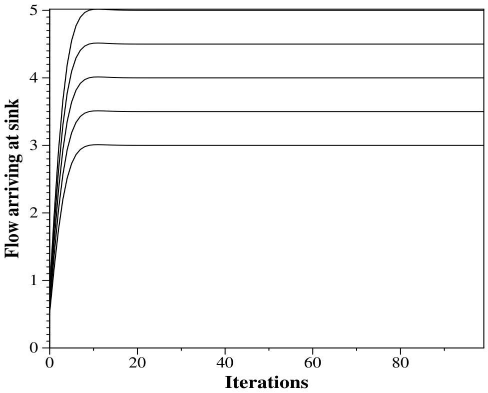

Fig. 8. Behavior of the idealized second-order method on a 5 node clique graph with 5 commodities and $\beta = 1.4$.

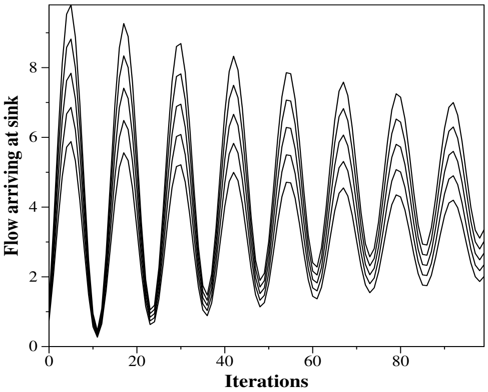

Fig. 9. Behavior of the idealized second-order method on a 5 node clique graph with 5 commodities and $\beta = 1.98$.

Running Times. In Table 1, we provide some very preliminary timing results. All timings were performed on a Sun Ultra 30 workstation with 300 Mhz UltraSPARCII processor and 256 MB of RAM, and the codes were compiled with the `-O` option using the vendor-supplied C compiler.

As input graph, we used a $5 \times 5 \times 20$ RMF graph, with 25, 50, and 100 commodities. All demands had the same value, while the capacities of the forward edges in the RMF graph were chosen at random. The sources and sinks were chosen from the first and last panels, respectively, of the graph.[9]

We give running times for four different methods: (1) the basic first-order method, as described by Awerbuch and Leighton [AL93], (2) the realistic second-order method with $\beta =$

[9] Thus, since the number of nodes in the first panel is 25, the number of "commodity groups" (see [LSS93]) in the implementation of Leong, Shor, and Stein [LSS93] is at most 25, independent of the number of commodities.

1.99, (3) the idealistic second-order method with $\beta = 1.97$, and (4) the Maximum Concurrent Flow code of Leong, Shor, and Stein [LSS93], referred to as LSS.

Algorithm	25 commodities	50 commodities	100 commodities
Leong-Shor-Stein (LSS)	519.77	456.10	501.72
First-order (Awerbuch-Leighton)	642.99	1233.32	2836.62
Realistic second-order, $\beta = 1.99$	149.01	304.64	645.16
Idealistic second-order, $\beta = 1.97$	9.54	27.70	70.41

Table 1. Running times (in seconds) of the different algorithms on a 500 node RMF graph. For LSS, we chose $\epsilon = 0.05$, while for the other codes, we terminated the runs after every commodity was within a 0.01 factor (first-order) or 0.001 factor (second-order) of its demand.

When looking at these numbers, the reader should keep the following points in mind:

(1) The code of Leong, Shor, and Stein [LSS93] solves the more general problem of maximizing the ratio of feasible flow, while our code only finds a feasible flow. However, we are not aware of any code for feasible flow that outperforms LSS. Following the suggestion in [LSS93], all our runs are performed with demands very close to the maximum feasible, by scaling the demands using the maximum edge congestion returned by LSS.
(2) The results for LSS are most likely not optimal, as we were unsure about the best setting of parameters for the code. Given the results reported in [LSS93] and the increases in CPU speed over the last few years, we would have expected slightly better numbers.
(3) We have not yet implemented a good termination condition for our code. Instead, we chose to measure the time until all flows at the sinks have converged to within a factor of at most 0.001 (second-order method) or 0.01 (first-order method) of the demands.
(4) We limit the reported numbers to RMF graphs due to differences in the graph formats used in LSS and in our code, which did not allow a direct comparison on other types of graphs.

We point out that the behavior of the LSS algorithm is fairly complex, while the performance of our second-order methods is dependent on the precise choice of β. Thus, one should be careful when trying to infer general performance trends from the few numbers provided above. However, our experiments with other graphs also showed a similar behavior. Thus, we believe that our implementation is at least competitive with the best previous codes, and may in fact significantly outperform them. We plan to perform a more thorough study in the future. We also see significant room for further improvements in the running times of our codes.

Sensitivity Analysis. An attractive feature of local algorithms is that they are, in general, robust. That is, they are expected to scale gracefully when edges appear or disappear, or traffic patterns change [AL93]. We will not try to formalize this intuition here. In Figure 10, we present an illustrative example of the behavior of local flow algorithms under dynamic situations, which shows how the resulting flows adapt quickly as we change the demands of commodities.

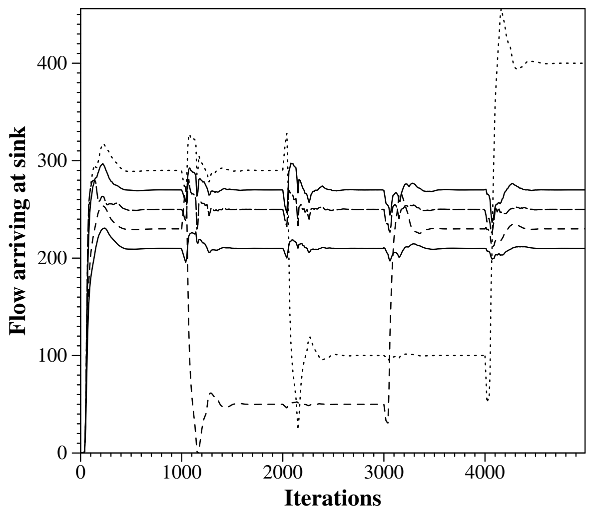

Fig. 10. Sensitivity of the algorithm to changes in demands, for the idealized method with $\beta = 1.98$ on a 500 node RMF graph with 5 commodities. We show the amounts of flow ariving at the sinks as we repeatedly change the demands, and thus the amounts of commodity injected into the network in each step.

5 Concluding Remarks

In this paper, we have proposed second-order methods for distributed, approximate maximum flow and multicommodity flow based on the first-order algorithms recently proposed by Awerbuch and Leighton [AL93, AL94]. We have presented experimental results that illustrate several interesting aspects of the behavior of these algorithms, and that provide strong evidence that the second-order methods significantly outperform their first-order counterparts.

The main open problem raised by our results is to give a formal analysis of the performance of the second-order methods for multicommodity flow, or to at least show a separation between first-order and second-order methods. We believe that this is a very challenging technical problem. Our experimental results also raise, and leave open, a number of other intriguing questions concerning the behavior of such distributed flow algorithms, and the diffusive processes underlying them. We list a few below.

Question 1. It would be very interesting to show that not only the amount of flow reaching the sinks, but in fact the entire "flow pattern" in the network converges to a stable state.[10] This was the case in all our experiments. If true, this will simplify the process of stopping the iteration in a distributed manner when the flows have converged; furthermore, it may improve the analytical bounds on the performance of the algorithm, since we do not have to average the flows over several steps as suggested in [AL93].

Question 2. For the case of the maximum flow problem, it would be interesting to show bounds that are tighter than those implied by the analysis for multicommodity flow in [AL93]. In

[10] As far as we know, this question is still open even in the first-order maximum flow case.

particular, it appears from our experiments that the convergence behavior of the maximum flow algorithms may be significantly better than $1/\epsilon$.

Question 3. Suppose the flow injected into the sources at each iteration consists of a collection of packets. Can we analyze or bound the delays of the packets, given an appropriate scheduling principle for packets at each node (such as first-in-first-out), if only for the first-order methods? This would correspond to providing certain quality-of-service guarantees to the sessions in communication networks. Such analysis was recently done for load balancing [MR98] and packet routing [AK+98] under adversarial models of traffic injection, but assuming unit edge capacities.

Question 4. As mentioned earlier, random walks can be modeled as a matrix iteration which is identical to the behavior of first-order algorithms for distributed load balancing [MGS98]. Can we design random walks that correspond to second-order algorithms? This may lead to improved bounds for mixing times of random walks. Some progress has been made recently for special graphs [S98]. Another question that arises is whether random walks can be set up to yield the first/second-order behavior in the presence of edge capacities. □

We are working on several extensions of our experimental results. In particular, we are working on an implementation of the improved first-order algorithm presented in [AL94], and on dynamic acceleration schemes for the second-order method such as those using Chebyshev polynomials with a β that varies from iteration to iteration. We are also in the process of carrying out a thorough comparison of our distributed implementations to that of the existing sequential multicommodity codes (see [LSS93] and the references therein).

6 Acknowledgments

Sincere thanks to Stephen Rutherford for experimenting with a preliminary version of the second-order method for the maximum flow problem. We also thank Fan Chung for valuable discussions, and Cliff Stein for graciously providing us with his multicommodity flow code.

References

[A94] O. Axelsson. *Iterative Solution Methods.* Cambridge University Press, 1994.

[AAB97] B. Awerbuch, Y. Azar and Y. Bartal. Local multicast rate control with globally optimum throughput. *Manuscript*, 1997.

[AA+93] W. Aiello, B. Awerbuch, B. Maggs, and S. Rao. Approximate load balancing on dynamic and asynchronous networks. *Proc. of 25th ACM Symp. on Theory of Computing*, 632-641, 1993.

[AK+98] W. Aiello, E. Kushilevitz, R. Ostrovsky, and A. Rosen. Adaptive packet routing for bursty adversarial traffic. *Proc. ACM STOC*, 1998.

[AL93] B. Awerbuch and T. Leighton. A simple local-control approximation algorithm for multicommodity flow. *Proc. 34th IEEE FOCS*, 459-468, 1993.

[AL94] B. Awerbuch and T. Leighton. Improved approximation algorithms for the multicommodity flow problem and local competitive routing in dynamic networks. *Proc. 26th ACM Symp. on Theory of Computing*, 487-496, 1994.

[AMS89] B. Awerbuch, Y. Mansour and N. Shavit. End-to-end communication with polynomial overhead. *Proc. 30th IEEE FOCS*, 358-363, 1989.

[BB+93] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H. van der Vorst. Templates for the solution of linear systems: Building blocks for iterative methods, SIAM, Philadelphia, Penn, 1993. `http://netlib2.cs.utk.edu/linalg/html_templates/Templates.html`.

[BG91] D. Bertsekas and R. Gallagher. *Data Networks*. Prentice-Hall Publ. 1991.

[BT89] D. Bertsekas and J. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall, 1989.

[C97] F. Chung. *Spectral Graph Theory*, Chapter 8, CBMS Regional conference series in Mathematics, 1997.

[C89] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 279–301, 1989.

[DMN97] R. Diekmann, S. Muthukrishnan and M. Nayakkankuppam. Engineering diffusive load balancing algorithms using experiments. *Proc. IRREGULAR 97*, LNCS Vol 1253, 111-122, 1997.

[GR97] A. Goldberg and S. Rao. Beyond the Flow Decomposition Barrier. *Proc. 38th IEEE FOCS*, 1997.

[GT88] A. Goldberg and R. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 1988, 921-940.

[HY81] L. A. Hageman and D. M. Young. *Applied Iterative Methods*. Academic Press, New York, 1981.

[K97] D. Karger. Using random sampling to find maximum flows in uncapacitated undirected graphs. *Proc. 30th ACM STOC*, 1997.

[LM+91] T. Leighton, F. Makedon, S. Plotkin, C. Stein, E. Tardos and S. Tragoudas. Fast approximation algorithms for multicommodity flow problems. *Proc. 24th ACM STOC*, 101-111, 1991.

[LSS93] T. Leong, P. Shor and C. Stein. Implementation of a Combinatorial Multicommodity Flow Algorithm. *First DIMACS Implementation Challenge: Network Flows and Matchings*, 1993.

[LW95] L. Lovasz and P. Winkler. Mixing of random walks and other diffusions on a graph. Technical Report, Yale University, 1995.

[MR98] S. Muthukrishnan and R. Rajaraman. An adversarial model for distributed dynamic load balancing. *Proc. 10th ACM SPAA*, 1998.

[MGS98] S. Muthukrishnan, B. Ghosh, and M. Schultz. First- and second-order diffusive methods for rapid, coarse, distributed load balancing. To appear in *Theory of Computing Systems*, 1998. Special issue on ACM SPAA 96.

[SM86] F. Shahrokhi and D. Matula. The maximum concurrent flow problem. *Journal of the ACM*, 37:318-334, 1990.

[S98] A. Sinclair. Personal communication, 1998.

[V89] P. Vaidya. Speeding up linear programming using fast matrix multiplication. *Proc. 30th IEEE FOCS*, 332-337, 1989.

[Var62] R. Varga. *Matrix Iterative Analysis*. Prentice–Hall, Englewood Cliffs, NJ, 1962.

7 Appendix: Experimental Setup

Implementation Details. All algorithms were implemented in C. A graphical frontend based on Tcl/Tk was used to run experiments and display the results. All input graphs were supplied in the DIMACS graph format, with some extensions to specify multiple commodities and changes in the demands over time.

Most of the execution time is spent in Steps (2) and (3) of the algorithm, which were implemented together in one single loop over the edges. Thus, the partitioning of the commodities between the queues was done during the edge balancing process, by applying an appropriate scaling factor to the flow stored in a node. This resulted in a very efficient implementation for the maximum flow case.

For the multicommodity flow case, the running time of Step (3) is dominated by the algorithm for resolving contention between different commodities in Section 2.4.1 of [AL93], which requires sorting the commodities in each edge by the values of $\Delta_i(e)/d_i^2$. While these values vary between iterations, the changes become increasingly smaller as the method converges. We exploited this property by using insertion sort and inserting the commodities in the sorted order of the previous iteration.

Input Graphs.

In our experiments described in this paper, we used three different classes of input graphs: mesh graphs, random leveled graphs, and RMF graphs. The first two types of graphs were generated using the GENGRAPH program of Anderson et al. from the University of Washington. The RMF graphs were generated with the GENRMF program of Tamas Badics. Both programs are available from the DIMACS website. Examples of these graphs are shown in Figures 11, 12, and 13.

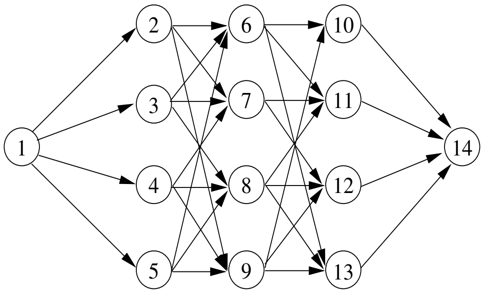

Fig. 11. Mesh graph with 3 levels and 14 nodes. All edges have randomly chosen capacity, except for edges connecting to the source or sink, which have capacity large enough such that they never constitute a bottleneck.

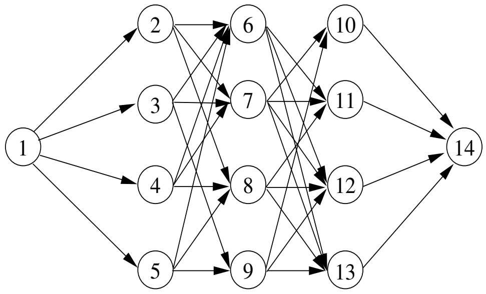

Fig. 12. Random leveled graph with 3 levels and 14 nodes. All edges have randomly chosen capacity, except for edges connecting to the source or sink, which have capacity large enough such that they never constitute a bottleneck.

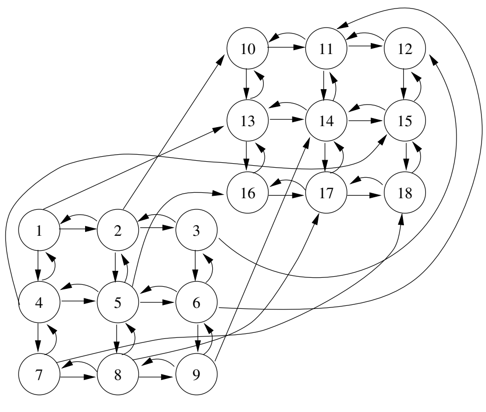

Fig. 13. A $3 \times 3 \times 2$ RMF graph. All edges between different layers have randomly chosen capacity, while edges inside a layer have capacity large enough such that they never constitute a bottleneck.

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