Alcom-FT Technical Report Series
ALCOMFT-TR-02-90

On the Problem of Scheduling Flows on Distributed Networks *

T. Lücking, B. Monien, and M. Rode**

Department of Mathematics and Computer Science
University of Paderborn
Fürstenallee 11
33102 Paderborn, Germany
{luck,bm,rode}@uni-paderborn.de

Abstract. We consider the problem of scheduling a given flow on a synchronous distributed network. This problem arises using diffusion-based approaches to compute a balancing flow, which has to be scheduled afterwards. We show that every distributed scheduling strategy requires at least $\frac{3}{2}$ times the minimum number of rounds. Furthermore we give a distributed algorithm for flows on tree networks, which requires at most two times the optimal number of rounds.

1 Introduction

Load balancing is an essential task for the efficient use of parallel computer systems. It can be observed that the work loads of many parallel applications have dynamic behavior and may change dramatically during runtime. To achieve an efficient use of the processor network, the load has to be balanced between the processors during runtime. Obviously, the balancing scheme is supposed to be highly efficient itself in order to ensure an overall benefit.

One major field of application are parallel adaptive finite element simulations where a geometric space, discretized using a mesh, is partitioned into sub-regions. The computation proceeds on the mesh elements in each sub-region independently [5]. As the computation is carried out, the mesh refines and coarsens, depending on the problem characteristics such as turbulence or shocks (in the case of fluid dynamics simulations, for example) and the size of the sub-regions (in terms of the numbers of elements) has to be balanced. The problem of parallel finite element simulation has been extensively studied – see the book [5] for an excellent selection of applications, case studies and references.

Much work has been done on the topic of *load balancing*. The approaches depend on the model used to describe the interprocessor communication. In this

* Partly supported by the DFG-Sonderforschungsbereich 376 Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen, by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT), and by the IST Programme of the EU under contract number IST-2001-33116 (FLAGS).

** Graduate School of Dynamic Intelligent Systems

paper we consider synchronous distributed processor networks. In each round, a processor of the network can send and receive messages to/from all its neighbors simultaneously. Furthermore, we assume that the situation is fixed, i.e., no load is generated or consumed during the balancing process, and the network does not change.

In order to balance the network, it is necessary to migrate parts of the processors' loads during runtime. We assume that the load consists of independent load units, called *tokens*. One possible approach to balance the network is based on a 2-step model.

First, a balancing flow, that is, a flow of tokens putting the network into a balanced state where all processors keep the same load up to one token, is calculated. This can be done efficiently with help of *diffusion* (for more details see [1]), yielding a balancing flow optimal with respect to the l_2-norm [2, 4]. The flow can be represented by a directed acyclic graph $G = (V, E)$, having $|V| = n$ vertices and $|E| = m$ edges, with flow function f.

Second, the load items are migrated according to that flow [2, 3, 8]. The goal is to use the minimum number κ of rounds to reach the balanced state. This *flow scheduling problem* trivially leads to a formulation as a linear program with $\kappa(m + n)$ unknowns and $m + 2\kappa n$ equations. Such a system is solvable in $\mathcal{O}(\kappa^5(m+n)^5)$ rounds using e.g. Karmarkar's algorithm [7]. Diekmann et. al. [2] show that the $(\kappa-1)$-commodity flow problem on bipartite graphs can be reduced to the problem of deciding whether the flow can be scheduled in κ rounds.

Moreover they propose a *distributed* scheduling strategy which needs at most $\mathcal{O}(\sqrt{n})$ times the minimum number of rounds, and they give a worst-case example on a tree network yielding $\Omega(\log(n))$ times the minimum number of rounds. We are not aware of any other results on this problem.

The upper bound of $\mathcal{O}(\sqrt{n})$ is due to missing information on the structure of the flow graph. In this paper we introduce a distributed algorithm for flow graphs on tree networks. It investigates the structure of the flow graph before sending tokens. We show that the algorithm requires at most 2κ rounds.

In Section 2 we introduce the notation used in this paper. We then show in Section 3 that every distributed scheduling strategy requires at least $\frac{3}{2}$ times the minimum number of rounds. In Section 4 we propose a distributed algorithm which schedules flows on tree networks. We prove that it requires at most 2 times the minimum number of rounds. Furthermore, we show that this upper bound is tight.

2 Definitions

We consider a *synchronous processor network*. In order to describe this network formally, we use the model defined in [6]. The network $N = (V, C)$ consists of $|V|$ **processors** and $|C|$ undirected **channels**. Execution of the entire system begins with all processors in arbitrary start states, and all channels empty. Then the processors, in lock-step, repeatedly perform the following two steps:

1. Generate the messages to be sent to the neighbors. Put these messages in the appropriate channels.
2. Compute the new state from the current state and the incoming messages. Remove all messages from the channels.

The combination of the two steps is called **round**.

The load situation on N is given by the **load function** $l : V \to \mathbb{N}_0$, representing the number of unit sized tokens on the processors. A **flow** on this processor network N is a directed acyclic graph $G = (V, E)$ with $|E|$ **flow edges**, and a **flow function** $f : E \to \mathbb{N}_0$. We denote by $s : E \to V$ and $t : E \to V$ functions, defining source and target of each edge. For each $v \in V$ we call $\mathrm{in}(v) = \{e \in E \mid t(e) = v\}$ the **set of incoming edges** of v, and $\mathrm{out}(v) = \{e \in E \mid s(e) = v\}$ the **set of outgoing edges** of v. For every $v \in V$ the flow property holds, that is, we have

$$l(v) + \sum_{e \in \mathrm{in}(v)} f(e) \geq \sum_{e \in \mathrm{out}(v)} f(e). \tag{1}$$

We call (G, l, f) a **flow graph**.

Definition 1 (Schedule). *A* **schedule** *S for a flow graph (G, l, f) using k rounds is a decomposition of the flow function f into k flow functions $f^1, \ldots, f^k$, determining the flow in each round, i.e., $S = (f^1, \ldots, f^k)$ with*

$$\forall e \in E : f(e) = \sum_{i=1}^{k} f^i(e) \quad \textit{and} \tag{2}$$

$$\forall v \in V, j \in \{1, \ldots, k\} : l(v) + \sum_{i=1}^{j-1} \sum_{e \in \mathrm{in}(v)} f^i(e) \geq \sum_{i=1}^{j} \sum_{e \in \mathrm{out}(v)} f^i(e). \tag{3}$$

Inequality (3) ensures that in every round each processor has sufficient load.

Definition 2 (Flow Scheduling Problem). *We call the problem of finding a schedule S for a flow graph (G, l, f), which requires the minimum number of rounds $\kappa((G, l, f))$, the* **flow scheduling problem**.

In the following we restrict our discussions to tree networks N. Furthermore, we assume that the network is rooted, that is, exactly one processor is assigned to be the root of the tree, and each other processor v knows its **parent** with respect to this root, denoted by $\mathrm{parent}(v)$. We call the graph G of a flow on N a **directed tree** if each edge of G is directed away from the root of N, i.e., $s(e) = \mathrm{parent}(t(e))$ for each $e \in E$.

3 Worst-case analysis for distributed scheduling strategies

In this section we show that for every distributed scheduling strategy there exists a flow graph on which it requires at least $\frac{3}{2}$ times the minimum number of rounds. The flow graph that will be used to prove this result is illustrated in Figure 1.

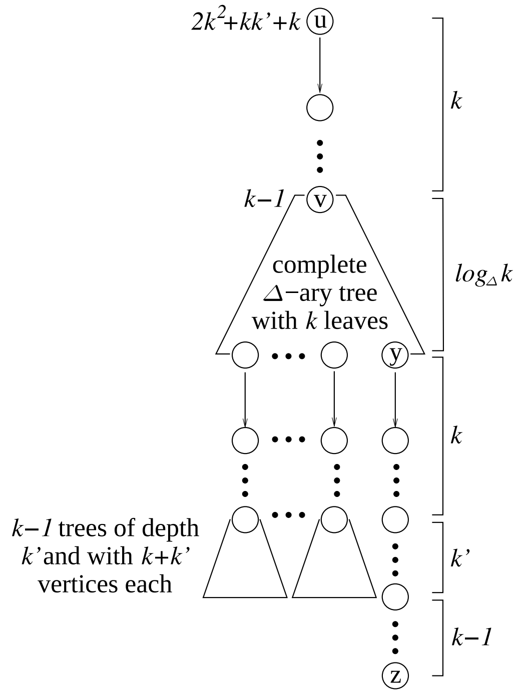

Fig. 1. Flow graph used in the proof of Theorem 1. The terms on the left of vertex u and vertex v give their initial loads. The expressions on the right hand side of the graph denote the lengths of the paths or depths of the trees.

Theorem 1. *Let $\Delta \in \mathbb{N}$ with $\Delta \geq 2$. Then for every distributed scheduling strategy there exists a flow graph (G, l, f) with $\deg(G) = \Delta + 1$ on which it requires at least $\frac{3}{2}$ times the minimum number of rounds.*

Proof. Let k be a power of Δ, and choose $k' \in \mathbb{N}_0$ minimal with $\frac{\Delta^{k'+1}-1}{\Delta-1} \geq k+k'$. Consider the processor network of degree $\Delta+1$ in Figure 1. It consists of a path of length k from vertex u to vertex v, followed by a complete Δ-ary tree of depth $\log_\Delta k$ with root v and k leaves. To each of the first $k-1$ leaves, a path of length k is attached whose last vertex is the root of a Δ-ary tree with $k+k'$ vertices and depth k'. To the remaining leaf y, a path of length $2k+k'-1$ is attached.

Initially, only two vertices have some load. Vertex v has an amount of $k-1$ tokens, and u keeps as much load as necessary to provide exactly one token to each vertex of the network in the balanced state, that is, an amount of $2k^2 + kk' + k$. Since the network is a tree, the balancing flow is uniquely determined and therefore also l_2-optimal.

If the initial load of vertex v is sent towards the path below y, then the minimum number of migration rounds is attained. At the same time u's load is sent downwards. After $2k + k' + \log_\Delta k$ rounds each processor holds one token, and the migration phase is finished.

When using a distributed scheduling strategy, each vertex initially only knows its own load and the flow on its incident edges. Among all vertices with distance at most $k + \log_\Delta k$ to v, the incoming edges of those being at the same distance to v have equal flow. Hence, all subtrees below each inner vertex of the complete tree are equal up to a depth of k. Thus during the first k rounds these vertices cannot gather information that helps to decide in which direction to send the load they might get.

As no helpful information is available during the first k rounds, we can assume that y is the leaf to which the least amount of load has been sent. The only source of load in a distance of at most k from the k leaves is v with load $k-1$. Hence, the least amount of load cannot exceed $\frac{k-1}{k} < 1$. Thus in the worst-case none of the tokens reaches the path at y during the first k rounds of any distributed scheduling algorithm, and the path has to be filled up by tokens stemming from u. One of these tokens has to travel a distance of $3k + k' + \log_\Delta k - 1$ (from u to z, the last vertex of the path). So the ratio between the worst case number of scheduling rounds of any distributed algorithm and the maximum distance a token would have to travel in an optimal schedule is

$$\frac{3k + k' + \log_\Delta k - 1}{2k + k' + \log_\Delta k}.$$

As $\frac{k'}{k}$ becomes 0 for $k \to \infty$, the ratio approaches $\frac{3}{2}$ as k increases. $\square$

Note that the result in Theorem 1 is a lower bound on the worst-case performance of all distributed scheduling strategies. For a certain strategy there might exist a flow graph on which it performs much worse.

4 The distributed algorithm

In this section we give a distributed scheduling algorithm for flow graphs (G, l, f) on tree networks. We start with a distributed algorithm for flow graphs with directed tree G. Using this result, we give a distributed algorithm for any flow graph on tree networks.

4.1 Flow graphs (G, l, f) on tree networks with directed tree G

Let (G, l, f) be a flow graph with directed tree G. For each $e \in E$ we recursively define for all $i \in \mathbb{N}_0$

$$\begin{aligned} d_0(e) &= f(e), \\ d_{i+1}(e) &= \max\left\{ \sum_{\tilde{e} \in \mathrm{out}(t(e))} d_i(\tilde{e}) - l(t(e)), 0 \right\}. \end{aligned}$$

Note that $d_i(e)$ is the number of tokens that have to be sent along edge e to vertices at distance of at least i from vertex $t(e)$.

Theorem 2. *Let (G, l, f) be a flow graph with directed tree G, and let $k \in \mathbb{N}_0$ be minimal such that $d_k(e) = 0$ for all $e \in E$. Then an optimal schedule requires exactly k rounds, that is, $\kappa((G, l, f)) = k$.*

Proof. We prove the theorem in two steps:

1. <u>There exists a valid schedule using k rounds.</u>
   We construct a schedule $S = (f^1, \dots, f^k)$ as follows. In round i, every vertex v with incoming edge e sends $f^i(\tilde{e}) = d_{k-i}(\tilde{e}) - d_{k-i+1}(\tilde{e})$ tokens along each edge $\tilde{e} \in \mathrm{out}(v)$. After i rounds

$$\sum_{j=1}^{i} f^j(\tilde{e}) = \sum_{j=1}^{i} d_{k-j}(\tilde{e}) - d_{k-j+1}(\tilde{e}) = d_{k-i}(\tilde{e})$$

   tokens have been sent along edge $\tilde{e}$ for all $\tilde{e} \in E$. Denote by $l_i(v)$, $0 \le i < k$, the load on vertex v after i rounds. Then we have

$$\begin{aligned} l_i(v) &= l(v) + d_{k-i}(e) - \sum_{\tilde{e} \in \mathrm{out}(v)} d_{k-i}(\tilde{e}) \\ &= l(v) + \max\left\{ \sum_{\tilde{e} \in \mathrm{out}(v)} d_{k-i-1}(\tilde{e}) - l(v), 0 \right\} - \sum_{\tilde{e} \in \mathrm{out}(v)} d_{k-i}(\tilde{e}) \\ &\ge \sum_{\tilde{e} \in \mathrm{out}(v)} (d_{k-i-1}(\tilde{e}) - d_{k-i}(\tilde{e})) = \sum_{\tilde{e} \in \mathrm{out}(v)} f^{i+1}(\tilde{e}). \end{aligned}$$

   Hence the described schedule is feasible, and it requires k rounds.

2. <u>Any valid schedule takes at least k rounds.</u>
   We now show by induction that the schedule described above takes the minimum number of rounds, i.e., that for any flow graph for which an optimal schedule requires k rounds, $d_k(e) = 0$ for all $e \in E$. If the optimal schedule takes $k = 1$ round, then each vertex has sufficient load to fulfill all its outgoing edges at once, and $d_1(e) = 0$ for all $e \in E$. So assume that the described schedule is optimal for all flow graphs (G, l, f) with minimum number of rounds $\kappa((G, l, f)) \le k \in \mathbb{N}_0$. Now let (G, l, f) be a flow graph with minimum number of rounds $\kappa((G, l, f)) = k + 1$.
   Denote by f' the flow computed in the first round of the optimal schedule, and let f'' be the remaining flow. Then we have $f(e) = f'(e) + f''(e)$ for each edge $e \in E$. After the first round the load for each $v \in V$ is

$$l'(v) = l(v) + \sum_{\tilde{e} \in \mathrm{in}(v)} f'(\tilde{e}) - \sum_{\tilde{e} \in \mathrm{out}(v)} f'(\tilde{e}).$$

   Applying the schedule construction to the load function l and flow function f', yielding $d'_0(e), d'_1(e), \dots$, and to the load function l' and flow function

f'', yielding $d''_0(e), d''_1(e), \ldots$ for all $e \in E$, we get two schedules using 1 and k rounds, respectively, due to the induction hypothesis. So, $d'_1(e) = 0$ and $d''_k(e) = 0$ for all $e \in E$. By induction we show that applying the construction to the original problem with load l and flow f yields

$$d_i(e) = \begin{cases} d''_i(e) + f'(e) & \text{if } 0 \le i < k, \\ \max\{f'(e) - \xi(e), 0\} & \text{if } i = k, \\ 0 & \text{else,} \end{cases}$$

with $\xi(e) = l'(t(e)) - \sum_{\tilde{e} \in \mathrm{out}(t(e))} d''_{k-1}(\tilde{e}) \ge 0$ for all $e \in E$.
We have $d_0(e) = f(e) = f'(e) + f''(e) = f'(e) + d''_0(e)$ which proves the claim for $i = 0$. So let $i \ge 1$. Using the induction hypothesis we get

$$\begin{aligned} d_i(e) &= \max\left\{ \sum_{\tilde{e} \in \mathrm{out}(t(e))} d_{i-1}(\tilde{e}) - l(t(e)), 0 \right\} \\ &= \max\left\{ \sum_{\tilde{e} \in \mathrm{out}(t(e))} (d''_{i-1}(\tilde{e}) + f'(\tilde{e})) \right. \\ &\qquad\qquad \left. -(l'(t(e)) - f'(e) + \sum_{\tilde{e} \in \mathrm{out}(v)} f'(\tilde{e})), 0 \right\} \\ &= \begin{cases} d''_i(e) + f'(e) & \text{if } 0 < i < k, \\ \max\{f'(e) - \xi(e), 0\} & \text{if } i = k, \end{cases} \\ d_{k+1}(e) &= \max\left\{ \sum_{\tilde{e} \in \mathrm{out}(t(e))} (f'(\tilde{e}) - \xi(\tilde{e})) - l(t(e)), 0 \right\} \\ &= \max\left\{ \sum_{\tilde{e} \in \mathrm{out}(t(e))} d'_0(\tilde{e}) - l(t(e)) - \sum_{\tilde{e} \in \mathrm{out}(t(e))} \xi(\tilde{e}), 0 \right\} = 0. \end{aligned}$$

Hence, as $d_{k+1}(e) = 0$ for all $e \in E$, the schedule takes $k+1$ rounds. □

From Theorem 2 we get a distributed Algorithm A which schedules flow graphs (G, l, f) with directed tree G. The algorithm works as follows. Each vertex v keeps a variable i which is initialized with 0.

Then vertex v successively sends $d_1(e), \ldots, d_{k_e}(e)$ along its incoming edge $e = (\mathrm{parent}(v), v)$ to its parent, where k_e is minimal with $d_{k_e}(e) = 0$. Note that $d_1(e)$ can be computed from the flows on the outgoing edges, and that in the ith round each vertex v can compute $d_i(e)$, $1 < i \le k_e$, from the values received in the previous round. The value $d_i(\tilde{e})$ of an outgoing edge $\tilde{e}$ is assumed to be 0 if no information has been sent along $\tilde{e}$. Due to Theorem 2 we have $k_e \le \kappa((G, l, f))$ for all $e \in E$.

Algorithm A then proceeds as follows. If the number of tokens on v is at least $\sum_{\tilde{e} \in out(v)} (d_{k_e-i-1}(\tilde{e}) - d_{k_e-i}(\tilde{e}))$, then v sends $(d_{k_e-i-1}(\tilde{e}) - d_{k_e-i}(\tilde{e}))$ tokens via each of its outgoing edges $\tilde{e}$ and increments i. Otherwise no tokens are sent. This step is repeated until $i = k_e$. We now prove that the schedule implicitly defined by Algorithm A requires at most $2 \cdot \kappa((G,l,f))$ rounds.

Corollary 1. *Let (G,l,f) be a flow graph with directed tree G. Then Algorithm A schedules the flow graph in $2 \cdot \kappa((G,l,f))$ rounds.*

Proof. Let $\kappa = \kappa((G,l,f))$, let $v \in V$ arbitrary, and denote by i_r the value of i after r rounds of Algorithm A on v. Obviously, after r rounds v has sent an overall amount of $d_{k_e - i_r}(\tilde{e})$ tokens along each outgoing edge $\tilde{e}$, whereas in the schedule constructed in Theorem 2, $d_{k-r}(\tilde{e})$ tokens with $\kappa = \max\{k_e | e \in E\}$ would have been sent. Note that after at most κ rounds every vertex v has received the values $d_1(\tilde{e}), \ldots, d_\kappa(\tilde{e})$ for all $\tilde{e} \in \text{out}(v)$. We show that $k_e - i_{\kappa+r} \leq \kappa - r$ for all $r \geq 0$. Since the sequence $d_0(e), \ldots, d_\kappa(e)$ decreases monotonically, this implies $d_{k_e - i_{\kappa+r}}(\tilde{e}) \geq d_{\kappa-r}(\tilde{e})$.

If $r = 0$, then no load has been sent, and the claim holds. Now assume the claim holds for $r \geq 0$. If $k_e - i_{\kappa+r} < \kappa - r$, then we immediately get $k_e - i_{\kappa+r+1} \leq \kappa - (r+1)$. If $k_e - i_{\kappa+r} = \kappa - r$, then

$$\begin{aligned}
d_{\kappa-r}(e) + l(v) &= \max\left\{ \sum_{\tilde{e} \in out(v)} d_{\kappa-r-1}(\tilde{e}), l(v) \right\} \\
&\geq \sum_{\tilde{e} \in out(v)} d_{k_e - i_{\kappa+r} - 1}(\tilde{e}) \\
&= \sum_{\tilde{e} \in out(v)} d_{k_e - i_{\kappa+r}}(\tilde{e}) + \sum_{\tilde{e} \in out(v)} \left(d_{k_e - i_{\kappa+r} - 1}(\tilde{e}) - d_{k_e - i_{\kappa+r}}(\tilde{e}) \right) .
\end{aligned}$$

Hence the load $l'(v)$ kept by vertex v after r rounds is

$$\begin{aligned}
l'(v) &= l(v) + d_{\kappa-r}(e) - \sum_{\tilde{e} \in out(v)} d_{k_e - i_{\kappa+r}}(\tilde{e}) \\
&\geq \sum_{\tilde{e} \in out(v)} \left(d_{k_e - i_{\kappa+r} - 1}(\tilde{e}) - d_{k_e - i_{\kappa+r}}(\tilde{e}) \right) .
\end{aligned}$$

Thus v has sufficient load to send $\left(d_{k_e - i_{\kappa+r} - 1}(\tilde{e}) - d_{k_e - i_{\kappa+r}}(\tilde{e}) \right)$ tokens via each of its outgoing edges $\tilde{e}$. Therefore $i_{\kappa+r}$ is incremented and the claim remains true. □

4.2 Arbitrary flow graphs (G,l,f) on tree networks

We now consider an arbitrary flow graph (G,l,f) on a tree network.

Theorem 3. *Let (G,l,f) be a flow graph on a tree network. Then there exists a distributed algorithm which schedules the flow graph in $2 \cdot \kappa((G,l,f))$ rounds.*

Proof. We construct an algorithm which we call Algorithm B. We distinguish between **upward edges** and **downward edges**. An edge $e \in E$ is called upward if $\text{parent}(s(e)) = t(e)$ holds, otherwise it is called downward. Since the underlying network is a tree, every vertex has at most one outgoing edge which is upward. Algorithm B works as follows.

In each round every vertex v sends all available load via its outgoing upward edge until it is saturated. Note that a vertex with an outgoing upward edge receives load via all its incoming edges while sending load only via its sole outgoing upward edge. As a consequence all upward edges are saturated after at most $\kappa((G, l, f))$ rounds.

Simultaneously, as in Algorithm A, each vertex v connected with its parent by a downward edge e sends the values $d_1(e), \ldots, d_{k_e}(e)$ along e, where k_e is minimal with $d_{k_e} = 0$. Here, $d_0(e), \ldots, d_{k_e}(e)$ are calculated according to the remaining flow with saturated upward edges. Therefore v has to know the load $l'(v)$ it will have when all upward edges are saturated. This load can easily be computed:

$$l'(v) = l(v) + \sum_{\substack{e \in \text{in}(v) \\ v = \text{parent}(s(e))}} f(e) - \sum_{\substack{e \in \text{out}(v) \\ t(e) = \text{parent}(v)}} f(e)$$

After v has received $d_1, \ldots, d_{k_{\tilde{e}}}$ for all outgoing downward edges $\tilde{e}$, and all incoming upward edges are saturated, which is the case after at most $\kappa((G, l, f))$ rounds, vertex v sends tokens along its outgoing edges in the same way as in Algorithm A.

Similar to the proof of Corollary 1, it can be proven that this requires $2 \cdot \kappa((G, l, f))$ rounds. □

Note that the proof of Theorem 3 does not depend on the choice of the root of the tree network. If the processor network is not rooted, then parents can be determined successively, starting at the leaves. The number of required rounds is bounded by half the diameter of N. The following example shows that the bound proved in Theorem 3 is tight.

Example 1. Consider the flow graph (G, l, f) with root r in Figure 2. Using Algorithm B described in the proof of Theorem 3, vertex v sends its k tokens to its parent r. As a result, one of the tokens from u has to be sent along the path from u to w of length $2k$. However, sending the k tokens of v toward w leads to $k + 2$ rounds. So the ratio between the number of rounds used by Algorithm A and the minimum number of rounds is

$$\frac{2k}{k+2} \to 2 \text{ for } k \to \infty.$$

5 Conclusion

In this paper we showed that no distributed scheduling strategy can have a worst-case performance lower than $\frac{3}{2}$, even on tree networks. Furthermore, we

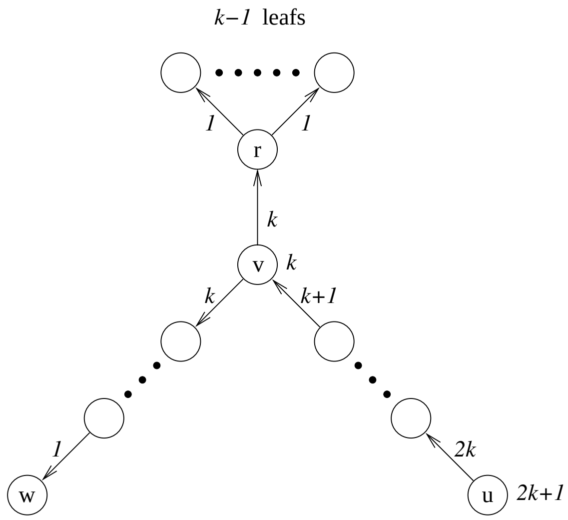

Fig. 2. Flow graph for which Algorithm B needs twice the minimum number of rounds.

presented a distributed algorithm for flows on tree networks yielding a worst-case performance of 2, whereas the local scheduling strategy in [2] takes at least $\Omega(\log(n))$ times the minimum number of rounds at the worst. We showed that the factor of 2 is tight. A natural question is whether the gap between the lower and the upper bound can be closed. Moreover, an open question is what factor can be achieved for arbitrary flow graphs.

6 Acknowledgment

The authors would like to thank Robert Elsässer, Torsten Fahle, Ulf-Peter Schroeder and Meinolf Sellmann for fruitful discussions.

References

1. G. Cybenko. Load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7:279–301, 1989.
2. R. Diekmann, A. Frommer, and B. Monien. Efficient schemes for nearest neighbor load balancing. *Parallel Computing*, pages 789–812, 1999.
3. R. Diekmann, F. Schlimbach, and C. Walshaw. Quality balancing for parallel adaptive fem. In *Solving Irregulary Structured Problems in Parallel (IRREGULAR'98)*, pages 170–181, 1998.
4. R. Elsässer, B. Monien, and R. Preis. Diffusive load balancing schemes on heterogeneous networks. In *Proceedings of the 12th ACM Symposium on Parallel Algorithms and Architectures (SPAA'00)*, pages 30–38, 2000.

5. G. Fox, R. Williams, and P. Messina. *Parallel Computing Works!* Morgan Kaufmann Publishers, Inc., 1994.
6. N.A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., 1996.
7. A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1994.
8. C. Walshaw, M. Cross, and M. Everett. Dynamic load-balancing for parallel adaptive unstructured meshes. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.