1-1-1994

Scheduling in a Ring with Unit Capacity Links

Perry Fizzano
Dartmouth College

Clifford Stein
Dartmouth College

Scheduling on a Ring with Unit Capacity Links

Perry Fizzano & Clifford Stein
Department of Computer Science
Dartmouth College
Hanover, New Hampshire 03755

Abstract

We consider the problem of scheduling unit-sized jobs on a ring of processors with the objective of minimizing the completion time of the last job. Unlike much previous work we place restrictions on the capacity of the network links connecting processors. We give a polynomial time centralized algorithm that produces optimal length schedules. We also give a simple distributed 2-approximation algorithm.

1 Preliminaries

We consider the problem of scheduling unit sized jobs on a network of processors arranged in a ring. An instance, $\mathcal{I}$, of network scheduling can be described by $\mathcal{I} = (G, J)$ where $G = (V, E)$ is an undirected graph representing the network and J is the set of jobs to be processed. Using the scheduling nomenclature we say there are m processors (or machines) labeled $p_1, p_2, \ldots p_m$, and n jobs. Each vertex in V corresponds to a processor and each edge corresponds to a network link (notice this means there are m nodes in the graph). Each edge has an associated capacity which restricts the amount of data transmitted across it in a single time step. In this paper we assume that the graph, G, is a ring. A ring is a network such that processor p_i is connected to p_{i+1} and p_{i-1}. (We are assuming throughout the paper that all addition on processor indices is done mod m, i.e. processor p_{m+i} is identical to p_i.) The set of jobs, J, will be indistinguishable and all of unit size, thus an instance can be described by just the number of jobs on a processor at a given time. We denote by j_i the number of jobs currently on processor p_i.

The network model just described allows a machine to process a job on the same step that it passes a job. This model is the same as that in many previous papers [1, 3, 4, 5, 6] and is supported by current technology [2]. Additional restrictions that we place on the model are that it takes unit time to traverse a network link and the capacity of each link is one job per time unit. This implies that a processor can pass one job to each neighbor in one time step but it can not pass two jobs to one neighbor.

Furthermore, we consider two network environments; centralized and distributed. A centralized environment is one in which there is global information available about the number of jobs currently residing on each machine. Conversely, a distributed environment has no source of global knowledge. Each processor only knows its own state and any information about other processors must be gathered explicitly.

The layout of the remainder of the paper is as follows. In Section 2 we give an algorithm for scheduling in a centralized setting and in Section 3 we prove it produces optimal length schedules. In Section 4 we give the timing analysis of the algorithm and develop a polynomial time solution to the problem. We present an algorithm for scheduling in a distributed environment in Section 5 and we make some concluding remarks in Section 6.

2 The Centralized Algorithm

2.1 Outline

We begin by presenting an algorithm for scheduling in a centralized environment. The algorithm we present is actually a decision procedure. Given a deadline, d, the

algorithm answers whether or not there is a schedule of length d or less. We turn this into an optimization procedure by binary searching. The space over which we binary search is bounded below by $\lceil \frac{n}{m} \rceil$ and bounded above by the maximum number of jobs starting on any one machine.

For the algorithm think of each time step as two halves. In the first half each machine processes a job if it has one and in the second half each machine may transmit a single job to each of its neighbors. Decisions about whether to pass a job or not are made after comparing the number of jobs on each machine to the time remaining until the deadline d. Machines that have too many jobs to finish by the deadline are labeled *surplus* machines; those that could have more jobs and still finish by the deadline are labeled *deficit* machines; and those that will finish exactly at the deadline with their current number of jobs are labeled *on-target* machines. Once we determine which machines are surplus, deficit and on-target then we compute a way to send the maximum number of jobs away from surplus machines to deficit machines. This process is repeated until either all of the jobs are processed or the deadline d is reached.

2.2 Details of One Step of the Algorithm

The first thing we need to do is determine which nodes are surplus, deficit and on-target. Given the deadline d and the current time t, the value that is considered on-target is $d - t$. Given the on-target value we can easily determine the labels of all the machines. The procedure ESTABLISH ROUTES then computes the maximum number of routes in a greedy manner between nodes with surplus and nodes with deficit. A *route* is defined as a set of edges that connects a deficit node with a surplus node. A route that starts at a surplus node that is created greedily, stops at the closest deficit node in the clockwise direction. Similarly, if a greedily established route starts at a deficit node it stops at the closest clockwise surplus node. Then jobs are sent according to the results of ESTABLISH ROUTES. The only non-trivial part of the decision procedure described in Figure 1 is the procedure ESTABLISH ROUTES. We proceed to explain it here.

The first step of the procedure ESTABLISH ROUTES is to form a new graph, G', which differs slightly from the original graph, G. To form G' we *contract* on-target nodes and *duplicate* nodes with surplus or deficit greater than one. Formally, the *contraction* of a node v, originally connected to nodes w and x, amounts to removing v from the graph and connecting w and x directly. The *duplication* of a node v, originally connected to nodes w and x, means that we replace v by two nodes v_1 and v_2. Node v_1 is connected to w, node v_2 is connected to x and finally v_1 is connected to v_2. Notice that contractions and duplications preserve the ring structure of the graph.

The next step is to pick a starting point. If there are two adjacent surplus nodes we pick one of them otherwise any surplus node can serve as the starting point. Once we have a starting point we walk around the ring once in a clockwise direction and make routes in a greedy manner. Figure 2 details the procedure Establish Routes.

One final observation is that the routes we have established in the graph G' correspond directly to routes in G even though we have contracted out on-target nodes. Consider a route (or any piece of a route) in G' that goes from vertex v to vertex w such that v and w are not directly connected in G. Instead they are separated by on-target nodes $u_1, u_2, \ldots u_k$. This route corresponds to a route in G where v passes a job to u_1, u_1 passes to u_2 and so on until u_k passes a job to w. Notice that this chain of passing effectively lets us pass a job further than just one link in a time step.

3 Correctness of the Algorithm

First we need to prove that the subroutine ESTABLISH ROUTES is producing the maximum number of routes between surplus and deficit machines. Let $\mathcal{S}$ be the number of surplus nodes in G' and let $\mathcal{D}$ be the number of deficit nodes in the graph G'. We will use the following simple lemma.

Lemma 1 *The maximum number of routes between surplus and deficit nodes in G' is no more than the minimum of $\mathcal{S}$ and $\mathcal{D}$.*

Proof: Without loss of generality assume that $\mathcal{S}$ is smaller. The best we could do is have every surplus node on a different route since each route must consist of at least one surplus node and one deficit node. Hence the maximum number of routes we could establish is $\mathcal{S}$. □

RING SCHEDULER (d)

d is the length of the schedule to be checked.

for $i = 1$ to d
    -for each processor p_k with a job, $j_k = j_k - 1$
    -label machines as deficit, surplus or on-target
    (the target is $d - i$ on step i)
    -call ESTABLISH ROUTES
    -send job(s) as specified by ESTABLISH ROUTES
if every machine has zero jobs left then
    answer "yes there is a schedule of length d"
else
    answer "no schedule of length d"

Figure 1: The decision procedure

ESTABLISH ROUTES

-create the new graph, G', by contracting on-target nodes and
duplicating nodes two or more in surplus or deficit
-if there are two adjacent surplus nodes then
    $start$ = the clockwise node
-else
    $start$ = any surplus node
-while all the nodes have not been considered
    -establish a clockwise route from $start$ to the first
    node which completes a route, call it v'.
    -$start$ = the first node clockwise of v'
-translate the routes established in G' to routes in G

Figure 2: The procedure ESTABLISH ROUTES

Note that we could get fewer than $\mathcal{S}$ routes if we could not arrange all the surplus nodes to be on different routes (e.g. if there are three surplus nodes in a row).

Next we justify the starting point that the algorithm chooses to establish the maximum number of routes.

Lemma 2 *The first route that* ESTABLISH ROUTES *makes is in some optimal solution.*

Proof: There are two cases to consider. The first case holds is if there are adjacent surplus nodes and the second case holds otherwise.

If there are adjacent surplus nodes, s_1 and s_2, such that s_1 is clockwise of s_2 then we claim that you can start at s_1 and walk clockwise around the ring to produce the maximum number of routes.

Let $OPT = (b_1, b_2, ...b_k)$ be any maximum set of routes (Notice the ordering of the routes in OPT is not relevant. We're just trying to achieve a solution with maximum cardinality.) We show how to convert this solution to a solution of the same cardinality which contains a route whose starting point corresponds to the starting point of the first route our algorithm would establish. There are four cases to consider.

(*i*) s_1 is on a route that does not contain s_2 in OPT.

This is what our algorithm does.

(*ii*) s_2 starts a route which goes through s_1 in OPT.

We can drop s_2 from the route. Now we have a set of routes so that the first node of one route starts at s_1 and proceeds clockwise.

(*iii*) Neither of s_1 and s_2 are on a route in OPT.

Look at the first route in the optimal set which is cw from s_1. If the node on this route which is closest to s_1 is a surplus node then attach s_1 to the front of this route. If the node closest to s_1 is a deficit node, v, then replace the original route to v with a route from s_1 to v. Now we have a solution of the same cardinality with the first route being one that our algorithm produces.

(*iv*) s_1 starts a route which goes through s_2 in OPT.

First, take s_1 off of this route. Then perform the same trick as case *iii* with s_1.

The second part of the proof handles the case when there are not adjacent surplus nodes in G'. If this is the case then we know that there are at least as many deficit nodes as surplus nodes. By Lemma 1 we know that the maximum number of routes we can get is bounded by $\mathcal{S}$. Since there are deficit nodes between every pair of surplus nodes we can produce $\mathcal{S}$ paths by starting at any surplus node and create a clockwise path to the first deficit nodes. □

The previous lemma says that the first route that our algorithm forms is compatible with an optimal solution. This will be used as the basis for an inductive proof that shows that ESTABLISH ROUTES forms a maximum set of routes while using this initial route.

Lemma 3 ESTABLISH ROUTES *produces the maximum number of routes between surplus and deficit nodes in the graph G'.*

Proof: Let $A = (a_1, a_2, \ldots, a_j)$ be the set of routes, in cw order, produced by our greedy method. By Lemma 2 we know that the first route, a_1 is compatible with some optimal solution. Let $OPT = (b_1, b_2, \ldots, b_k)$ be such a solution. Now we claim that after selecting a_1 as our initial route we have reduced the problem to a smaller instance of the same problem.

Say the route a_1 consists of vertices $v_1, v_2, \ldots v_x$. The new problem includes the vertices $(v_{x+1}, ...v_m)$ and the solution $OPT - b_1$ must be an optimal solution to this smaller problem. For if it wasn't, then we could obtain a larger set of routes to the original problem by concatenating a_1 to the beginning of the optimal schedule for this smaller instance. By induction, we can argue that the greedy choice for every route will produce an optimal set of routes. □

Up to this point we have shown that ESTABLISH ROUTES produces the maximum number of routes between surplus and deficit nodes. This implies that we are removing as many jobs as possible from surplus machines on each time step. However, there is one other aspect of the algorithm for which we have not accounted. On each step every machine that has a job processes it. Call this the *Greedy Processing Rule*. The next lemma shows that this rule does not inhibit the production of optimal length schedules.

Lemma 4 *There exist optimal length schedules that use the Greedy Processing Rule.*

Proof: Assume we are given an optimal schedule, S, of length d. Assume that step i is the first step where the schedule S does not use the Greedy Processing Rule.

The claim is that we can replace the i^{th} step of S with a step of our schedule, S^*, which does use the Greedy Processing Rule. The only problem that could develop is that with a different i^{th} step S^* might not be able to do the exact same routing as in some subsequent step of S. But notice that this is only a problem when some processor, p, processes its last job sooner in our schedule than it did in S. This may affect a future routing step because some other processor, q, may be on the receiving end of a route which goes through p. Now q will not be able to receive a job because p will not be able to pass on any jobs towards q. However, processing this job on p instead of q does not lengthen the schedule because after this step every processor has no more jobs than it did in S. Hence, the length of the schedule from here on can be no greater in S^* than in S. □

Theorem 1 RING SCHEDULER *will correctly determine if a schedule of length d exists.*

Proof: If there is no schedule of length d then there will be no routing and processing scheme which could achieve it. RING SCHEDULER will not erroneously find a schedule of length d because it does not process more than one job per time unit nor does is send more than one job across a link in any time step.

If there is a schedule of length d then we claim that RING SCHEDULER will find it. By Lemma 4 we know that the Greedy Processing Rule allows us to produce an optimal length schedule. By Lemma 3 we are sending as many jobs as possible away from surplus machines on each time step. We claim that this greedy approach is optimal. The reason is that it doesn't matter what order the surplus machines get rid of work because they all must get rid of all their surplus by the deadline in order for the schedule to complete by time d. Imagine some surplus processor holding onto a job in order to pass more jobs on a subsequent time step. Since that processor is holding one extra job the most number of extra routes that could be established at a later time is one. Thus, more routes are not created overall by holding onto jobs. □

3.1 Different Processor Speeds

Until now, all processors could process one job per time unit. We can modify the problem so that some processors can process more jobs than one per time unit. Let s_i denote the speed of processor p_i. The speed of a processor is defined as the number of jobs it can process in one unit of time. If we know the speed of each processor then we can determine if a processor is a surplus, deficit or on-target node by calculating whether it can process its remaining jobs by the deadline d.

This modification does not change the basic structure of the problem. No node will receive a job that it can't process by the deadline and the most number of jobs are being sent away from surplus nodes on each time step.

We can also modify the space over which we perform binary search. Let S denote the sum of the speeds of all the processors. A lower bound on the schedule length can be expressed as $\lceil \frac{m}{S} \rceil$. An upper bound on the schedule length is the maximum time any processor would take to finish with no jobs getting passed.

4 Time Complexity

4.1 A Simple Analysis

The running time of ESTABLISH ROUTES is linear in the number of nodes of the graph G'. (recall this is within a constant factor of the number of machines which is m). To check a schedule of length d we need to run ESTABLISH ROUTES at most d times. However our choice as to what d to check is the result of a binary search. The interval over which we search is bounded from below by $\lceil \frac{n}{m} \rceil$ and from above by n which is an upper bound on the maximum number of jobs that starts on any one machine. Thus there can be $O(\log n)$ invocations of ESTABLISH ROUTES. This gives RING SCHEDULER a running time of $O(dm \log n)$. In the worst case d is $O(n)$ which results in a pseudo-polynomial time algorithm because the input for this problem can be specified in $O(m \log m + m \log n)$ space since the jobs are indistinguishable and only the number of jobs on each machine is necessary to describe the instance.

Previous results of Deng *et al.* [3] give results for general network structures and general capacities of the network links. However, their solutions are not polynomial either but pseudo-polynomial, because one term of the running time is the number of jobs, n, which as we just said is not bounded by a polynomial in the input size.

Now we analyze a slight modification of our algorithm and show that it runs in polynomial time.

4.2 A Better Analysis

To obtain a faster running time we can take advantage of the fact that no machine ever changes from an on-target machine to a surplus or a deficit machine. Machines that are surplus or deficit will approach a value that is on-target but once they are on-target they never change. Our algorithm enforces this by contracting out on-target nodes from the graph on which we run ESTABLISH ROUTES.

So we can classify each machine in one of the following five states: two or more in surplus, one in surplus, on-target, one in deficit, two or more in deficit. However, since a machine can not go from any surplus state to any deficit state and it monotonically approaches the on-target value it can only take on at most three states; either the first three or the last three. Hence, the network as a whole will have at most $3m$ different configurations over the course of the algorithm. We can speed up the algorithm by not running ESTABLISH ROUTES as often. If after an iteration of the algorithm no machines have changed state then the same set of routes will suffice for the next iteration. Therefore, we only need to run ESTABLISH ROUTES $O(m)$ times instead of $O(d)$ times.

To make this procedure realizable we need to be able to compute the time that the current set of routes must change so that a machine does not go from a surplus state to a deficit state or vice versa. This entails recognizing one of two situations. The first is if a machine is two or more in surplus (or deficit) and it is on the end of two routes then as soon as it becomes only one in surplus (or deficit) or on-target we must run ESTABLISH ROUTES again. The second is if a machine is on the end of only one route it will change state when it reaches the on-target value and we must re-run ESTABLISH ROUTES at this point.

Let t be the value that is considered on-target, let T be the current time and let j_i denote the number of jobs on processor p_i. Let p_i^* be the processors that are on the end of exactly one route in the current set of routes and $p_i^{'}$ represents the processors that are on the end of two routes in the current set of routes. Now we can compute the next time that the routes will change as:

$$T + \min(\min_{p_i^*}(\mid j_i - t \mid), \min_{p_i^{'}}(\lfloor \tfrac{\mid j_i - t \mid}{2} \rfloor)).$$

Theorem 2 RING SCHEDULER *runs in* $O(m^2 \log n)$ *time.*

Proof: The above discussion shows that ESTABLISH ROUTES only needs to be called $O(m)$ times. Given this bound on the number of calls to ESTABLISH ROUTES we can bound the total running time of the algorithm by $O(m^2 \log n)$, which gives us a polynomial time algorithm. □

5 A Distributed Scheduler

Thus far all of our results have depended on some sort of global knowledge. We needed to know exactly how many jobs each machine had on each time step in order to determine if the machine was a surplus machine or a deficit machine. In a distributed setting this information must be obtained by passing messages around the network. We are assuming that a message can be sent, as well as a job, along the network links each time step . The message is just an integer representing the number of jobs on a given machine so we are not really abusing the limited capacity of the network links.

The basic idea of the algorithm is for each processor to know the state of its neighbors at the previous time step, and then pass a job to either or both neighbors if that neighbor is in danger of being idle on the next time step. The details appear in Figure 3, where we use j_i to denote the number of unprocessed jobs on processor p_i. Note that in this description of the algorithm two messages can be sent over a link in one step; it is not hard to reduce this to one.

We wish to prove that this algorithm produces schedules of length close to optimal. We will let $\mathcal{I}$ denote an instance of the scheduling problem, and $OPT(\mathcal{I})$ the length of the shortest possible schedule for $\mathcal{I}$. If algorithm A always yields a schedule of length no more than $\rho OPT(\mathcal{I}) + O(1)$ we call A a *ρ-approximation algorithm.* We first show a lower bound on any scheduling algorithm, even one with global knowledge.

Lemma 5 *If the optimal schedule is of length d then no consecutive group of k processors can start with more than (k+2)d total jobs.*

receive messages from neighbors p_{i-1} and p_{i+1}
set *left* and *right* equal the values of the messages from p_{i-1} and p_{i+1} respectively
if $j_i \neq 0$
    process a job, set j_i to $j_i - 1$
    if $j_i > 3$ and $right \leq 1$ then pass a job to neighbor p_{i+1} and set j_i to $j_i - 1$
    if $j_i > 3$ and $left \leq 1$ then pass a job to neighbor p_{i-1} and set j_i to $j_i - 1$
tell neighbors that p_i has i jobs.

Figure 3: One step of the distributed ring scheduling algorithm for processor p_i.

Proof: The best that could be done with $(k+2)d$ jobs and k consecutive processors is to have the work distributed evenly among the k processors and send two jobs out of the region on every time step. This leads to $(k+2)d-2$ jobs being processed in d units of time. This is a contradiction of the optimal schedule length being d. □

This lemma exposes a significant restriction on the way work can be distributed among the processors. For example, no pair of adjacent processors, at time 0, can contain more than $4d$ jobs. There are also significant restrictions on the conditions under which jobs can be passed.

Lemma 6 *Given a processor p_i, let t be the earliest time that $j_i \leq 1$. Then*

a) p_i receives no jobs before time t.

b) After time t, $j_i \leq 3$.

c) Let $t' > t$ be the first time that $p_j \leq 3$, where p_j is a neighbor of p_i. For any k, $1 \leq k \leq t' - t$, p_j passes a job to p_i in at least half of the time steps between time t and $t + k$, inclusive.

Proof: Part a) is clear from the description of the algorithm. For part b), we observe that due to the time delay between the actual state of a processor and the state that its neighbor is aware of, processor p_i can receive jobs for two consecutive time steps. When it first receives jobs it must have zero jobs, thus after receiving jobs it has at most two jobs. During the next step, it will process one job, and receive up to two jobs, thus having at most three jobs at the end of the step. However, if it had two at the beginning of the previous step it will receive no more jobs on the next step, and will receive no more jobs until it has processed all of its jobs and has zero remaining. Hence there is no way for the number of jobs to rise above three.

A slightly more careful look at the proof of part b) will suffice to establish part c). At time t, if p_i has one job, then at times $t+1$ and $t+2$ it will receive a job from each neighbor that has more than three unprocessed jobs. This is due to the one unit time delay between the actual state of a processor and the state that the processor's neighbor is aware of. It then takes p_i at most two steps to process received jobs until it returns to having one job; the cycle continues until p_i's neighbors run out of work to pass. So in at least two out of every four steps, passing occurs; furthermore, the passing occurs in the first two steps after p_i becomes idle. This establishes the claim. □

We now show that the algorithm in Figure 3 is a 2-approximation algorithm.

Lemma 7 *Let S' be the schedule in which no processor ever passes a job and let S be the schedule produced by the algorithm in Figure 1. Then S is no longer than S'.*

Proof: Let $m(t)$ be the maximum number of jobs on any processor at time t. In schedule S' it is always the case that $m(t+1) = m(t) - 1$. We will show that in schedule S, $m(t+1) \leq m(t) - 1$, thus proving the lemma.

We observe that in S the only processors that pass jobs have more than three jobs, and that the processor that had the maximum number of jobs $m(t)$ at time t has at most $m(t) - 1$ jobs at time $t+1$. Thus, when $m(t) > 3$, the processors with $m(t)$ jobs decrease by at least one, and by part b) of Lemma 6, no processor's load increases above 3, so $m(t)$ decreases. When $m(t) \leq 3$, no passing occurs. Therefore at each step of S $m(t)$ decreases by

at least 1, which implies that the length of S is at most $m(0)$, which is the length of S'. □

Now using the previous three lemmas we show that the capacitated ring scheduling algorithm gives schedules of length within a factor of two of optimal.

Theorem 3 *Let d be the length of the optimal schedule. Then the capacitated ring scheduling algorithm produces a schedule of length no more than $2d+2$.*

Proof: There are two cases to consider.
Case 1: No processor starts with more than $2d$ work.

By Lemma 7 we know that the schedule length does not increase by passing jobs, thus the maximum schedule length for this case is $2d$.
Case 2: Some processor starts with more than $2d$ work.

Let processor p_i be a processor that starts with more than $2d$ work, i.e. processor p_i has $2d+x$ work (for $0 < x \leq d$). Its neighbors, p_{i-1} and p_{i+1}, start with at most $2d-x'$ (for $x' > x$ by Lemma 5). Assume p_{i+1} starts with $2d-x'$ work, and that this is no less than what p_{i-1} starts with. At time no later than $2d-x'$ p_{i+1} becomes idle. At this point it may receive work from both its neighbors. By part c) of Lemma 6 we know that during at least half the time steps in the interval from time $2d-x'$ through the time when j_i first goes below 3, p_i will pass jobs to p_{i+1}.

At time $2d-x'$ p_i has at most $x+x'$ work. Assume for simplicity that p_i has passed no jobs to its neighbors at any time up to $2d-x'$. It will now begin to pass jobs to its neighbors until it has only three jobs left. In $\lceil \frac{x+x'-3}{2} \rceil$ time it can pass $\lceil \frac{x+x'-3}{4} \rceil$ work to each neighbor and process $\lceil \frac{x+x'-3}{2} \rceil$ jobs. It will then spend three units of time processing the final three jobs. So p_i completes all its jobs in time no more than $2d-x'+\lceil \frac{x+x'-3}{2} \rceil + 3$ which is less than or equal to $2d+2$ since $x' > x$. □

We note that a more careful analysis (on a slightly modified algorithm) which goes through a number of cases for the last three steps can be used to show a bound of exactly $2d$.

6 Conclusions

We have given a simple and efficient centralized scheduling algorithm to produce optimal length schedules on a ring of processors when the bandwidth of the network links is limited to one job per time unit. This is much faster than the best known algorithm for this instance [3] and in addition, is the first polynomial time solution to the problem. This approach has led us to designing a simple distributed algorithm for the same network structure that produces schedules within a factor of two of optimal.

References

[1] B. Awerbuch, S. Kutten, and D.Peleg. Competetive distributed job scheduling. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 571–580, 1992.

[2] H. Choi and A. Esfahanian. A message routing strategy for multicomputer systems. *Networks*, 22:627–646, 1992.

[3] X. Deng, H. Liu, J. Long, and B. Xiao. Deterministic load balancing in computer networks. In *Proceedings of 2nd IEEE Symposium on Parallel and Distributed Processing*, 1992.

[4] P. Fizzano, D. Karger, C. Stein, and J. Wein. Job scheduling in rings. In *Proceedings of the 6th ACM Symposium on Parallel Algorithms and Architectures*, 1994.

[5] B. Hoppe and E. Tardos. The quickest transhipment problem. unpublished manuscript submitted to SODA 95, 1994.

[6] C. A. Phillips, C. Stein, and J. Wein. Task scheduling in networks. In *Proceedings of the 4th Scandinavian Workshop on Algorithm Theory*, July 1994. To appear.