Competitive Implementation of Parallel Programs

Xiaotie Deng*
Department of Computer Science
York University
North York, Ontario M3J 1P3
deng@cs.yorku.ca

Elias Koutsoupias†
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
elias@cs.ucla.edu

Philip MacKenzie‡
Department of Mathematics and Computer Science
Boise State University
Boise, Idaho 83725
philmac@cs.idbsu.edu

Abstract

We apply the methodology of competitive analysis of algorithms to the implementation of programs on parallel machines. We consider the problem of finding the best on-line distributed scheduling strategy that executes in parallel an unknown directed acyclic graph

*Partially supported by an NSERC grant.
†Partially supported by NSF grant CCR-9521606.
‡Partially supported by TARP grant 003658480.

(dag) which represents the data dependency relation graph of a parallel program and which is revealed as execution proceeds. We study the competitive ratio of some important classes of dags assuming a fixed communication delay ratio τ that captures the average interprocessor communication measured in instruction cycles. We provide competitive algorithms for divide-and-conquer dags, trees, and general dags, when the number of processors depends on the size of the input dag and when the number of processors is fixed. Our major result is a lower bound $\Omega(\tau/\log\tau)$ of the competitive ratio for trees; it shows that it is impossible to design compilers that produce almost optimal execution code for all parallel programs. This fundamental result holds for almost any reasonable distributed memory parallel computation model, including the LogP and BSP model.

Keywords: Parallel computation, competitive analysis, communication delay, scheduling, compiler.

1 Introduction

The execution profile of a program can generally be represented as a directed acyclic graph (dag): Nodes represent instructions, or sets of instructions, and edges represent dependencies between individual nodes. An edge (u, v) denotes that the results of node u are required for the execution of node v. Usually, the dag of a program is not known at the compile time, since it depends on the input data and the runtime conditions. This, however, is no problem at all for a uniprocessor system because nodes can be executed as they become available. Any scheduling which would not intentionally idle achieves the optimal completion time. Even for the system performance metric of mean response time, the Round-Robin scheduling strategy guarantees a mean response time at most twice the optimum, without using any information about the actual executed dag [9].

The situation becomes more complicated when we deal with parallel programs, since one of the intricacies of parallel computation is that the optimum algorithm may depend critically on the profile of the parallel machine. Nevertheless, we may classify parallel models into two major categories: models with shared memory, and models with distributed memory. Our work focuses on a simplified distributed memory model, the communication delay model introduced in [11]. We must stress, however, that our main negative result

applies to most models with distributed memory.

1.1 Distributed Memory Models

In this paper, we assume the Papadimitriou-Yannakakis communication delay model [10, 11] to study the implementation problem of parallel programs on general purpose multicomputer systems. In this model, a universal parameter τ, the communication delay between processors measured in instruction cycles, is used to abstract communication in parallel machines. A processor can execute a node of the dag if every predecessor node has been executed either *by the same processor or by some other processor* $\tau + 1$ *or more time units before*. Putting it differently, there is a delay of τ steps for the result of a node to become available to other processors.

Other models of parallel computation assume a communication delay and usually have more parameters for the communication cost. The BSP model, proposed as a bridge between software and hardware[14], and the LogP model, developed by Culler et al. [2] are two such models. These models are not limited to parallel computing but they try to capture the locality properties of a machine, a factor that is also important in distributed systems, especially for clusters of workstations. Similar trends for unifying parallel and distributed computing are observed in parallel architecture and parallel programming [2, 13]. Even for shared memory machines, there is a factor similar to communication delay: It is the ratio of the cost for accessing the shared memory over the cost for accessing private caches of individual processors (a widely used technology to improve performance of parallel computers.)

1.2 Competitive Analysis

It was shown in [11] that there exists a polynomial time algorithm that approximates the optimum execution time of any dag within a factor of two when the number of available processors is unbounded. The approach of [10] and [11] assumes that the computation dag is known before the algorithm actually runs, essentially at compile time (the algorithm in [11] can be made on-line, but only by using a very high number of processors [5]). This assumption may not be realistic for all the parallel programs, since conditional statements and loops are important parts of any programming language. First, crucial parameters such as the actual shape of the dag and

the size of its nodes are found out by processors executing various parts of the program on-line. Second, runtime parameters such as the number of available processors during execution may not be known beforehand and may change dynamically. Because of communication delays, each processor has to make its own decision on how to proceed based on incomplete information. Thus, the situation naturally falls into a currently very active field: *Competitive analysis of on-line algorithms*. More specifically, competitive analysis requires that the compiler outputs not just code, but a *distributed on-line algorithm* for the execution of the program in hand, hopefully one that minimizes the worst case ratio of the completion time of the strategy on a given source program, to the completion time of the optimal execution of the program (with complete information concerning runtime conditions and execution paths).

The concept of competitive analysis has been introduced recently by Sleator and Tarjan in [12] (see also [6, 8]). For the scheduling problem, however, Graham [4] recognized almost thirty years ago that, without knowing job lengths, a scheduler can achieve twice the optimum. To formally define the competitive ratio for our problem, we first introduce some notation. In general, the on-line problems we consider here depend on three parameters: a family of dags $\mathcal{F}$, the number k of available processors and the communication delay ratio τ. Without loss of generality, we assume that the execution time of each individual node is exactly one time step, since a node with integer execution time m can be represented by a path of m unit jobs. Many values for the three parameters $\mathcal{F}$, k, and τ result in interesting special cases. For example, when the family $\mathcal{F}$ contains only one element, the input dag is essentially known to the algorithm; all problems studied in [11] are of this kind.

Consider an on-line scheduling strategy S that executes a dag from a given family $\mathcal{F}$ with k processors and communication delay ratio τ. For a dag $G \in \mathcal{F}$, let us denote by $T_S(G)$ the execution time of G by the scheduling strategy S and by $\mathrm{opt}(G)$ the execution time of G by an optimal off-line algorithm, which has complete information of the dag G. The on-line strategy S is called c-competitive if there exist constants $c \geq 1$ and d such that for all $G \in \mathcal{F}$:

$$T_S(G) \leq c \cdot \mathrm{opt}(G) + d.$$

The competitive ratio of the strategy S is the infimum of all such c's. The constant d is used to factor out insignificant factors and initial conditions.

We are interested in determining the competitive ratio of the best on-line strategy for a given set of parameters $\mathcal{F}$, k and τ.

In this paper we focus our attention to two important cases of families of graphs:

- *The dag is known qualitatively:* The family $\mathcal{F}$ depends on some fixed dag G and contains exactly the dags that result if we replace each node of G with a path of one or more nodes. In other words, the input dag (G) is known qualitatively, but the execution times of its nodes are unknown. This is a typical high level view of a parallel source program. Some interesting families of dags are of this type: A set of independent tasks with unknown execution time, Fast Fourier Transform (FFT) dags, pyramids, etc.

- Only *the nature of the dag is known:* The family $\mathcal{F}$ consists of dags that result from all possible executions of some fixed program. Some interesting examples are the family of all dags, the family of all trees, the family of binary trees, and the family of divide-and-conquer dags (a tree connected to its mirror image).

1.3 An Impossibility Result on Near Optimal Compiler

Informally, the competitive ratio represents a performance parameter measuring the closeness of a solution to the optimum. A competitive ratio of one represents the best situation for efficient parallel compiler design, because it would imply that there is a way to automatically generate parallel code which fully utilizes the inherent parallelism of a program. Unfortunately, our result(Theorem 7) shows that this is far from possible for the communication delay model [10, 11]: No scheduler can guarantee a competitive ratio for divide-&-conquer trees better than $\Omega(\frac{\tau}{\log \tau})$.

We also extend this result to the LogP model for a lower bound of $\Omega(\frac{L}{\log L})$, where L is the communication latency in the LogP model. Though one may argue that L is a constant in the LogP model, it is significantly larger compared to the access time of private cache memories of individual processors. A similar result applies also to the BSP model. This provides strong evidence for the following thesis: *In any reasonable parallel computation model with distributed memories, it is impossible for a compiler to produce parallel*

code that runs optimally or near optimally (up to a constant factor) for every program. Though our results rule out the existence of near optimal compiler for general parallel programs, they do not inhibit positive results for special important classes of parallel programs. Using Valiant's approach of pipeline routing, one can prove that all PRAM algorithms can be optimally executed, at least in theory, when a slackness in the number of processors is introduced. The same approach may be applied to parallel algorithms specially designed for coarse grained parallel computers. Some recent results [7, 16] are very promising for special classes of parallel algorithms.

Prior to our work, there has already been some work by Wu and Kung in [15] which acknowledges the on-line nature of the parallel implementation of algorithms, in a different model of parallel computation. They consider dynamic parallel implementations of divide-and-conquer algorithms, where the divide-and-conquer tree is discovered on-line, but they consider only the specific class of execution dags that are balanced, in the sense that a linear speed-up is possible for these dags.

1.4 Outline of Presentation

In Section 2, we discuss the case when the dag is known qualitatively. We show that a variant of the algorithm in [11] can be implemented on-line with a competitive ratio 2 when the number of processors is at least equal to the maximum width of the dag. When the number of processors is constant (independent of the size of the input dag), we give a 2-competitive algorithm for the dag consisting of independent paths. We also present a 3-competitive algorithm for merging trees, where the execution starts at the leaves and progresses towards the root.

In Section 3, we consider some problems when only the nature of the dag is known. Of particular interest is the family of full binary trees. For the simple case of two processors we give an optimal algorithm with competitive ratio 3/2. For the general case of fixed number of processors $k > \tau$ we show that the competitive ratio is at least $\Omega(\tau/(\log \tau \log k))$; this makes a trivial algorithm with competitive ratio $O(\tau)$ almost optimal. This lower bound applies also to the case of divide-and-conquer dags. Divide-and-conquer dags combine both cases of the problems we consider here: During the first expanding phase only the nature of the dag is known (a binary tree), but during the second merging phase the dag is known qualitatively (a known tree with unknown execution time of each node). The lower bound for the expanding

phase dominates the lower bound of the merging phase, and the overall competitive ratio is $\Omega(\tau/(\log\tau \log k))$. For general trees we show a lower bound on the competitive ratio of $\Omega(\tau/\log\tau)$. This lower bound is a general result that applies to many models of parallel computation. The Papadimitriou-Yannakakis model [11] assumes that each processor broadcasts the results of its computations to all other processors. The lower bound of $O(\tau/\log\tau)$ holds for much weaker models of communication. All that is needed is that each processor can communicate the result of its computation of one node to some other processor at each step.

2 The dag is known qualitatively.

In this section we assume that the dag is known qualitatively, that is, the dag is known but the execution time of each node is unknown. Equivalently, the family $\mathcal{F}$ of input dags consists of all dags that result if we replace each node of a fixed dag G with a path of one or more nodes.

2.1 Large number of processors.

We first consider the case of a large number of processors. It is an easy observation that with an unlimited number of processors there exists an on-line algorithm with competitive ratio 1. The reason is that we can use the unlimited number of processors to follow any possible execution scenario. One of these scenarios will turn out to be identical to the optimal execution and therefore the on-line execution time is equal to the optimal execution time. However, this is not a satisfactory algorithm since it places a heavy burden on the computational resources. We want to rule out such extreme possibilities and therefore we concentrate on the case when the number of processors is polynomially related to the size of the dag.

An important parameter of a dag G is its width $W(G)$ which is the maximum number of pairwise disconnected vertices. By Dilworth's theorem [3], the width of a dag G, is equal to the size of a mimimum chain cover, that is, a minimum size set of chains that covers all nodes of G. Notice that since the graph is known qualitatively, the chain cover number $w(G)$ can be computed at the compile time. The following on-line algorithm is an adaptation of the approximation algorithm in [11]. It uses only $w(G)$ processors by making effective use of a chain cover of a dag G; in contrast,

the algorithm in [11] needs a processor for each node of G.

Algorithm $\mathcal{A}$: Let $w = w(G)$ be the maximum width of G and let C_1, C_2,...,C_w be a chain cover of G. The objective of processor p_i is to execute all nodes of C_i. In order to achieve this, processor p_i may have to either compute some predecessors of the nodes of C_i, or get their results from some other processor after delay τ. Let us call a node u *available* to processor p_i when all immediate predecessors of v have been executed either by p_i or by some other processor τ time units before. The strategy of processor p_i is straightforward: It executes nodes of C_i one by one according to their order in the dag. When a node v of C_i is not available for execution, processor p_i executes available predecessors of v in any *reasonable order* until v becomes eventually available. An order is reasonable when the processor doesn't waste its time executing a node which is already available (calculated by some other processor τ time units ago). Note that this strategy can be implemented on-line. □

This natural algorithm has a small competitive ratio.

Theorem 1 *For every dag G, when the number of processors is at least $w(G)$, the competitive ratio of Algorithm $\mathcal{A}$ is 2.*

Proof. Notice first that the chain cover number does not change when we expand a node into a path of unit nodes with unspecified length, so we can assume without loss of generality that all nodes of the dag have unit execution time. Let us denote by q_v the processor that is assigned to execute the chain that contains node v. We adapt the proof used in [11] for showing that there exists a 2-approximate algorithm:

Let e be an integer function defined inductively on the nodes of the dag G as follows: For a node v with $p \leq \tau + 1$ predecessors, let $e(v) = p$; otherwise, order the predecessors of v in decreasing order, according to their e value: $e(u_1) \geq e(u_2) \geq \cdots \geq e(u_p)$. Then $e(v) = e(u_{\tau+1}) + \tau + 1$.

It was shown in [11] that no algorithm, on-line or off-line one, can execute a node v in less than $e(v) + 1$ steps. So, it suffices to show that Algorithm $\mathcal{A}$ executes every node v by time $2e(v) + 1$.

We use induction on the depth of v. For the basis case, observe that every source node belongs to a different chain in any chain cover of G. Since Algorithm $\mathcal{A}$ assigns initially at least one processor to each of them, the claim holds for nodes at depth 0. Similarly, for a node v with at most

$p \leq \tau + 1$ predecessors, q_v executes all predecessors of v and then v by time $p + 1 = e(v) + 1 \leq 2e(v) + 1$.

For the induction step assume that every predecessor u of v has been executed by time $2e(u)+1$. Order all predecessors of v in decreasing e value: $e(u_1) \geq e(u_2) \geq \cdots \geq e(u_p)$. By the induction hypothesis, all predecessors of v except the τ first ones, u_1, u_2,...,u_τ, have been executed by some processor by time $2e(u_{\tau+1})+1 \leq 2e(v)-2\tau-1$. After τ more steps, i.e., by time $2e(v)-\tau-1$, their results have reached all processors. Hence, from time $2e(v)-\tau$ and onwards q_v will execute either v or some of its τ first predecessors, u_1, u_2, $\cdots$, u_τ. As follows from the description of the algorithm, q_v never executes a node twice and never idles before the execution of v. Since any predecessor of u_i, is also a predecessor of v, by time $2e(v) - \tau$, q_v can start executing u_τ, $u_{\tau-1}$, ..., u_1. Therefore, q_v has enough time to execute the first τ predecessors of v and v itself, between time $2e(v) - \tau$ and $2e(v)$. Since $2e(v)$ is clearly less than $2e(v) + 1$ the induction step follows. Actually, this shows that we can use the same proof to get a slightly tighter result: Any node v is executed in at most $(2 - 1/(\tau + 1))e(v) + 1$ steps and therefore the competitive ratio of Algorithm $\mathcal{A}$ is $2 - 1/(\tau + 1)$. □

There is an immediate application to the important programming paradigm of the Fast Fourier Transform (FFT). Since an FFT dag (butterfly dag) with $n \log n$ nodes has width n we obtain:

Corollary 1 *When the number of processors is at least n, Algorithm $\mathcal{A}$ is 2-competitive for FFT dags of $n \log n$ nodes.*

Similarly, we obtain the following corollary for a diamond dag (a mesh turned 45^o with the convention that edges are ordered from top to bottom) and a pyramid dag (either the upper half or the lower half of a diamond dag):

Corollary 2 *When the number of processors is at least $\sqrt{n}$, Algorithm $\mathcal{A}$ is 2-competitive for diamond or pyramid dags of size n.*

Even though the general result of Theorem 1 assumes the powerful communication delay model, for many specific problems (in particular the problems we discussed above), it can be extended to other models of parallel computation with weaker communication requirements, such as point-to-point communication.

2.2 Constant number of processors.

We now turn our attention to the case where the number of processors k is fixed, that is independent of the input dag. The simplest case to consider is when the dag family consists of n disjoint paths. In the extreme case of no communication delay ($\tau = 0$), the obvious algorithm is that each processor repeatedly chooses a path from the set of unexecuted paths and finishes it. This algorithm is optimal with competitive ratio $2 - 1/k$ [4]. Unfortunately, this algorithm can not be applied when $\tau > 0$, because after a processor finishes a path, it does not instantly know the set of unexecuted paths; this information becomes available with a delay of τ time units.

A simple algorithm that avoids this problem is the following:

Algorithm $\mathcal{B}$: The algorithm operates in rounds: In each round, the unexecuted paths and the partially finished paths are partitioned (almost) evenly among the k processors. If there are more than k remaining paths, each processor executes its own set of paths for $c\tau$ time units, for some constant c, and stops. After τ more time units, processors know all outcomes and they can start a new round. In total, each round takes $c\tau + \tau$ time units. Finally, when the number of paths drops to k or less, each remaining path is assigned to some processor and is executed till its completion. □

The parameter c of the algorithm allows us to balance the communication time τ and the actual execution time $c\tau$ of each phase: When c is large, some processors will remain idle for a long time during a round if they finish all their paths early in the round; on the other hand, the disadvantage of a small c is that processors spend a large fraction of the time communicating instead of executing.

As was mentioned above, the competitive ratio of any on-line algorithm is at least $2 - 1/k$. We now show that we can choose c to achieve almost the same competitive ratio.

Theorem 2 *There is a choice of the parameter c such that the competitive ratio of Algorithm $\mathcal{B}$ is at most 2.*

Proof. Assume that the initial number of paths is greater than k (otherwise the competitive ratio is 1). Let r be the number of rounds before the number of paths drops to k or below during the execution of algorithm $\mathcal{B}$, and let t be the duration of the last round (if there is no such round when the number of paths drops below k, we set $t = 0$). Clearly, the time of the on-line algorithm is $r(c+1)\tau + t$.

We need to bound from below the optimal (off-line) time. There are two commonly used lower bounds of the optimum makespan for scheduling problems: the length of the longest path, and the average work (the total length of all paths divided by k.) Clearly, the length of the longest path is at least t. In order to estimate the average work, we consider two cases: a) when the number of paths at the beginning of a round is at least $kc\tau$, each processor gets at least $c\tau$ paths and therefore remains busy during the first $c\tau$ time units of the round, and b) when the number of paths drops below $kc\tau$, the total number of rounds with some idle processor is at most $kc\tau$, because every time a processor becomes idle the number of paths decreases. It follows that for at least $(r - kc\tau)$ rounds all processors are busy for at least $c\tau$ steps. Summing up the last t steps when at least one processor is busy, we get that the total work is at least $(r - kc\tau)kc\tau + t$. In fact, this is an overly conservative estimation, but it simplifies the analysis without affecting the competitive ratio.

Therefore, the optimum execution time is at least

$$\max\{(r - kc\tau)c\tau + t/k, t\} \geq \lambda((r - kc\tau)c\tau + t/k) + (1 - \lambda)t$$

for any real $\lambda \in [0, 1]$. Let $\lambda = (c/(c + 1) + (k - 1)/k)^{-1}$ which is in the interval $[0, 1]$ for $c > 1/(k - 1)$. For this λ, the optimum execution time is at least

$$\frac{c/(c + 1)}{c/(c + 1) + (k - 1)/k}(r(c + 1)\tau + t) - \lambda kc^2\tau^2.$$

Since the on-line execution time is $r(c + 1)\tau + t$ and $\lambda kc^2\tau^2$ is a constant (independent of the dag), the competitive ratio is at most $1 + \frac{k-1}{k}\frac{c+1}{c}$. For sufficiently large c, $c > k - 1$, this is less than 2. □

2.3 Merging trees.

The algorithm for independent paths, algorithm $\mathcal{B}$, can be adapted to work for merging trees. In a merging tree the execution starts at the leaves and proceeds towards the root. The execution of many algorithms is a merging tree. For example, the standard greedy computation of Huffman code has the structure of an unknown merging tree. Also, the second phase of a divide-and-conquer algorithm is a (qualitatively known) merging tree.

In the same manner with the algorithm for independent paths the leaves of a merging tree are partitioned evenly among the k processors. Every

processor executes the leaves assigned to it and, if possible, their successors for $c\tau$ time units. Processors then stop for τ time units to communicate and after pruning away the executed nodes they start a new round. When the number of leaves drops to k or below, processors switch to algorithm $\mathcal{A}$: The remaining tree is partitioned into chains and each processor executes a chain.

Theorem 3 *There is a 3-competitive algorithm for merging trees for any fixed number of processors.*

Proof. We remark that the strategy for assigning non-leaf nodes to processors in each round (except the last one) does not affect the competitive ratio. In fact, the argument below is valid even when only leaves are executed in each round with the exception of the last round.

As in the proof of Theorem 2, let r denote the number of rounds during the execution of the above algorithm while the number of leaves is more than k and let t be the duration of the last round (when there are at most k leaves). Clearly, the total on-line execution time is $r(c+1)\tau + t$.

We again try to bound from below the optimal execution time with the average work. Using an identical argument with that of the proof of Theorem 2 and using leaves instead of paths, we see that the average work is again at least $(r - kc\tau)c\tau + t/k$. However, we cannot now claim that the optimum execution time is at least t, as in the proof of Theorem 2. Instead, the optimal execution time is now at least $t/2$. This follows from the fact that in the last round we apply the algorithm $\mathcal{A}$ and because $\mathcal{A}$ is 2-competitive, the optimum execution time of the last round is at least half the execution time of $\mathcal{A}$.

As in the proof of Theorem 2 we get the following lower bound on the optimal execution time:

$$\max\{(r - kc\tau)c\tau + t/k, t/2\} \geq \lambda((r - kc\tau)c\tau + t/k) + (1 - \lambda)t/2$$

for any real $\lambda \in [0, 1]$. Choosing $\lambda = (2c/(c+1) + (k-2)/k)^{-1}$, we get that the competitive ratio is at most $2 + \frac{k-2}{k}\frac{c+1}{c}$. For sufficiently large c, $c > k/2 - 1$, this is less than 3. □

3 The nature of the dag is known.

3.1 Executing a tree by 2 processors.

For two processors and general dags a lower bound 3/2 on the competitive ratio holds even with no communication delay ($\tau = 0$) [4]. Even for binary trees (input at the root and outputs at leaves) when we allow nodes with one child, the lower bound is still 3/2. For example, a full binary tree with very long paths hung from the leaves behaves almost as a set of independent tasks with unknown execution times. But this is precisely the class of graphs used in proving a lower bound of 3/2 for general dags in [4].

However, this lower bound does not apply to full binary trees; a binary tree is full if and only if every internal node has exactly two children. For this case the communication delays become important. Here, we prove a lower bound of ratio 3/2 for computing a full binary tree, when $\tau \geq 1/2$. In particular, we have the following lemma:

Lemma 1 *The competitive ratio of two processors executing an unknown full binary tree is at least* 3/2*, when* $\tau \geq 1/2$*. For* τ *less than* 1/2*, the competitive ratio is at least* $1 + \tau$*.*

Proof. For any given on-line algorithm, let us consider an adversary which builds the tree as the execution proceeds. The tree that the adversary plans to build is a very "thin" one, in that it has depth equal to the number of leaves and the internal nodes form a path of length m starting from the root (see Figure 1).

Each level of the tree, except the first and the last one, contains two nodes, one of them internal and the other a leaf. The adversary decides which one is the leaf during the execution. The adversary's strategy is simple: *The node to be executed by the on-line algorithm first is the leaf* (when they are executed simultaneously, either can be the leaf). Notice that initially only the root of the tree is available to processors and therefore the optimal strategy is that both processors execute it. At the second level, the two nodes can be computed by any processor. The first node to be computed (in fact both of them will be computed by an optimal on-line algorithm) will turn out to be a leaf, and the processor that executed it cannot proceed to the next level. For $\tau \geq 1$, it is better for this processor to compute the other leaf rather than wait for its result to be communicated by the other processor. This is repeated in the next level, but with the role of processors reversed. Thus,

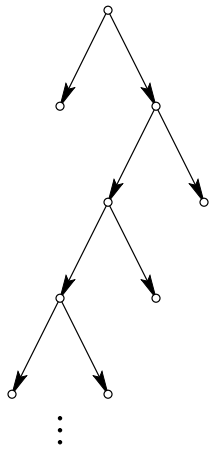

Figure 1: A "thin" tree.

in the third step we will be back where we started, with both processors executing an internal node. It follows that all internal nodes are executed by both processors. The total time for the processors to execute a tree of height m is $3m/2$.

However, the optimal time is bounded from above by $m+\tau$: One processor moves down along the longest path, and the other finishes the leaves while obtaining the value of the internal nodes via communication (after an initial delay of τ time units.) The lower bound follows.

For $\tau < 1$ the situation is more complicated. In order to analyze it, consider the nodes at some level $l \geq 1$ and assume that their execution by one processor starts at time t_l and by the other processor at time $t_l + d_l$, $d_l \geq 0$. The nodes of the next level become available to the first processor either at time $t_l + 2$, after computing both nodes at level l, or at time $t_l + d_l + 1 + \tau$, after getting the results of the internal node from the second processor. Therefore, the first processor starts the execution of level $l+1$ at time $\min\{t_l + 2, t_l + d_l + 1 + \tau\}$ or later. Similarly, the second processor cannot start the execution of level $l+1$ before time $t_l + d_l + 1$. So, we have that $t_{l+1} = t_l + d_l + 1$ and $d_{l+1} = \min\{\tau, 1 - d_l\}$. Initially, $t_1 = 1$ and $d_1 = 0$. Notice also that the sum of the times that the two processors spend in level l is $\min\{2, d_l + 1 + \tau\} + 1$.

When $1/2 \leq \tau \leq 1$, the values of d_l alternate between τ and $1 - \tau$, i.e., $d_2 = \tau$, $d_3 = 1 - \tau$, $d_4 = \tau$, $d_5 = 1 - \tau$, etc. Consequently, the time that both

processors spend on each level is $\min\{2, d_l+1+\tau\}+1 = 3$. Since the average optimum time for each level is 2, the competitive ratio is at least 3/2.

Finally, when $\tau \leq 1/2$, it follows that $d_l = \tau$, $l \geq 2$, and the sum of time that both processors spend on each level is $2+2\tau$. The competitive ratio for this case is at least $1+\tau$. □

Now we show that this lower bound is tight. In fact, we show that there exists a simple on-line algorithm for two processors that achieves a competitive ratio 3/2 on *any* tree.

Algorithm $\mathcal{C}$: The first processor executes the nodes of the unknown tree in a depth-first-search manner from *left to right* while the second processor executes the nodes in a depth-first-search manner from *right to left*. Notice that the algorithm is based on the assumption that a fixed left-to-right order is known to both processors; or equivalently, when a node is executed its children are revealed to both processors with the same order. □

Lemma 2 *Algorithm $\mathcal{C}$ achieves a competitive ratio 3/2 on any tree.*

Proof. Let p be the length of the longest path in the tree, and let b be the number of nodes that are computed by both processors. The crucial observation is that the nodes that are computed by both processors, except for the last τ nodes, belong to a path P (see Figure 2), and thus $p \geq b - \tau$. Since neither processor becomes idle, the time needed for the algorithm $\mathcal{C}$ to complete the execution of the tree is $(n+b)/2$, where n is the total number of nodes. The optimal execution time, however, is bounded from below by both the average work, $n/2$, and the length of the longest path p: $\max\{n/2, p\}$. Therefore, for any $\lambda \in [0,1]$, the optimum execution time is at least $\lambda(n/2) + (1-\lambda)p$. For $\lambda = 2/3$, we have that the optimal execution time is at least $(n+p)/3 \geq (n+b)/3 - \tau/3$. Since the on-line execution time is $(n+b)/2$, the competitive ratio of algorithm $\mathcal{C}$ is at most 3/2. □

Combining Lemmata 1 and 2 we have:

Theorem 4 *For $\tau \geq 1/2$, the competitive ratio for two processors executing an unknown tree is asymptotically equal to* 3/2.

3.2 The case of k processors.

Even for general dags, it is not difficult to obtain an algorithm achieving a competitive ratio $O(\tau)$. We can start by assigning processors to available

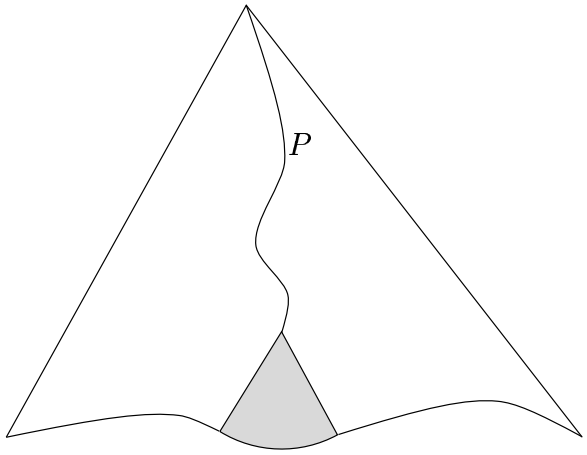

Figure 2: The execution of a tree.

nodes. After executing these nodes the processors wait for τ time units so that each processor knows all the outcomes. The competitive ratio of this algorithm is $(2 - 1/k)(\tau + 1)$. The reason is that if we ignore the communication delays, the competitive ratio is $2 - 1/k$ [4], and the communication multiplies the on-line execution time by $\tau + 1$. On the other hand, a competitive ratio of k is also trivially achievable when only one processor is used by the on-line algorithm. In fact, by using two processors as in Algorithm $\mathcal{C}$ a competitive ratio $(k + 1)/2$ is achievable. The proof is very similar to that of Lemma 2 (the average work of the optimum in this case is n/k instead of $n/2$).

The positive results for two processors in Section 3.1 may suggest that the competitive ratio $O(\tau)$ is pessimistic. Unfortunately this is not the case, and we next give a lower bound which is almost linear in τ. The reason is that with $k \geq 3$ processors the simple structure in 2-processor case breaks down, for the same reason that implementing $k > 2$ stacks is hard: We can no longer divide the nodes into k almost equal portions, because there is no way for k processors to "start from opposite ends".

3.3 Lower bound for trees.

We start with the simplest case of binary trees. Our construction is a chain of complete binary trees of size $2\alpha - 1$. The trees are chained in tandem, so

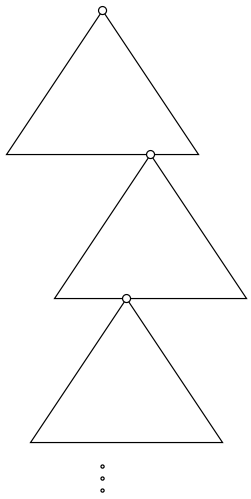

Figure 3: A chain of trees.

that one leaf of the first tree is the root of the second one, one leaf of the second tree is the root of the third one, and so on (see Figure 3).

For simplicity, we will charge the on-line algorithm only for the execution of leaves of each tree. Furthermore, in order to simplify the way the on-line processors communicate, we assume that the on-line execution operates in rounds: In each round the processors execute a sequence of $\beta = \tau/\alpha$ trees. Since there are only τ leaves in total for the first β trees, the processors get no information by communication during their execution. The processors may coordinate their efforts beforehand, but they have to find the leaf of the i-th tree which is the root of the $(i+1)$-st tree in isolation. When the first processor discovers the root of the $(\beta+1)$-st tree, the adversary reveals the roots of all β trees to all processors, and a new round begins with all processors executing the root of the $(\beta+1)$-st tree. It is clear that this can only help the on-line algorithm.

The adversary has the power to decide which leaf of the i-th tree to choose as root of the $(i+1)$-st tree. We will show that no matter how the processors coordinate their strategies the adversary can choose the roots of the trees in such a way that the processors of an on-line algorithm execute a lot of nodes in each tree.

Theorem 5 *The competitive ratio for parallel execution of binary trees with*

k processors is $\Omega(\min\{\frac{\tau}{\log\tau\log k}, k\})$ and $O(\min\{\tau, k\})$.

Proof. As it has already been mentioned, the upper bound holds for all dags. In order to obtain the lower bound we will use the probabilistic method [1]. Consider a random adversary for a chain of binary trees: Each leaf of a complete tree is chosen as the root of the next tree with equal probability.

Denote by X_{ij} the number of *leaves* of the j-th binary tree executed by the processor i before reaching the root of the next tree. Thus, X_{ij} is an integer random variable uniformly distributed in $\{1, 2, \ldots, \alpha\}$. By the definition of X_{ij}, processor i discovers the root of the $(\beta+1)$-st tree after executing $\sum_{j=1}^{\beta} X_{ij}$ leaves. A crucial observation is that for a fixed i the random variables X_{ij}, $j = 1, 2, \ldots$, are independent. Notice that the first processor discovers the root of the $(\beta+1)$-st tree after executing $\min_i \sum_{j=1}^{\beta} X_{ij}$ leaves. Therefore, when

$$P\left[\min_{i\in\{1,2,\ldots,k\}} \sum_{j=1}^{\beta} X_{ij} < l\right] < 1$$

the adversary has a strategy to force the on-line algorithm to execute at least l leaves of the first β trees. Since we have that

$$P\left[\min_{i\in\{1,2,\ldots,k\}} \sum_{j=1}^{\beta} X_{ij} < l\right] \leq kP\left[\sum_{j=1}^{\beta} Y_j < l\right]$$

we may concentrate on the probability $P\left[\sum_{j=1}^{\beta} Y_j < l\right]$, where Y_i are independent random variables uniformly distributed in $\{1, 2, \ldots, \alpha\}$. We will need the following Chernoff bound (see [1] for a proof of the case $\alpha = 2$):

$$P\left[\sum_{j=1}^{\beta} Y_j < \frac{\alpha+1}{2}\beta - \lambda\frac{\alpha+1}{2}\sqrt{\beta}\right] < e^{-3\lambda^2/2}$$

Choosing $\lambda = \sqrt{2\ln k/3}$ and $\beta = 16\lambda^2 = 32\ln k/3$, we have

$$P\left[\min_{i\in\{1,2,\ldots,k\}} \sum_{j=1}^{\beta} X_{ij} < \frac{(\alpha+1)\beta}{4}\right] < 1.$$

So, for any on-line algorithm the adversary has a strategy to force every processor to execute at least $(\alpha+1)\beta/4$ leaves of the first β trees. More

generally, the adversary can force the on-line algorithm to execute at least $m(\alpha+1)\beta/4$ leaves of a sequence of $m\beta$ trees.

On the other hand, the optimal execution time of a chain of $m\beta$ trees is at most $m\beta \max\{\log\alpha, 2\alpha/k\} + \tau$. This is achieved by assigning one processor to execute the longest path and to communicate the results to other processors that finish up the remaining nodes; thus, the optimal execution time is dominated either by the length of the longest path, $m\log\alpha$, or by the average work, $m(2\alpha-1)/k$ (there is also an additive term of τ steps for transmitting the first result).

Combining the bounds on the on-line and the optimal execution time, we get that the competitive ratio of any on-line algorithm is at least

$$\min\{\frac{\alpha}{4\log\alpha}, \frac{k}{8}\}.$$

Since $\beta = 32\ln k/3 = \Theta(\log k)$, and $\alpha = \tau/\beta = \Theta(\tau/\log k)$ we get that the competitive ratio is

$$\Omega(\min\{\frac{\tau}{\log\tau\log k}, k\}).$$

We note here that it is assumed that α and β are integers. This may not be always the case for $\beta = 32\ln k/3$. However, for $\tau = \Omega(\log k)$ we can approximate α and β and carry out the above computations without affecting the order of the competitive ratio. For $\tau = O(\log k)$ the theorem trivially holds, since $\frac{\tau}{\log\tau\log k} = O(1)$. □

By replacing the $m\beta$ full binary trees with trees of height one and α leaves in the proof of the above theorem, the lower bound of the on-line execution time remains unaffected, but the optimal execution time drops from $m\beta\max\{\log\alpha, 2\alpha/k\} + \tau$ to $m\beta\max\{1, 2\alpha/k\} + \tau$. So, we have:

Corollary 3 *The competitive ratio for executing general trees by k processors is $O(\min\{\tau, k\})$ and $\Omega(\min\{\frac{\tau}{\log k}, k\})$.*

The above corollary provides a very weak lower bound for small τ's. However, we can strengthen the above result, when $\tau = O(k)$:

Theorem 6 *The competitive ratio for executing general trees by k processors is $\Omega(\min\{\frac{\tau}{\log\tau}, k\})$.*

Proof. When k is at most τ^2, the theorem follows directly from Corollary 3. So, without loss of generality, we can assume that k is at least τ^2.

As in the proof of Theorem 5 the adversary constructs a tree. The building block, a β-chain, is a chain of $\beta = \log \tau$ trees, each of height one and $\alpha = \tau / \log \tau$ leaves. It was argued in the proof of Theorem 5 that when $\tau = \Omega(\log k)$, for any on-line algorithm that uses k processors, there is a chain of $\Theta(\log k)$ trees each of $\alpha = \Theta(\tau / \log k)$ leaves that requires $\Theta(\tau)$ steps. In particular, for any on-line algorithm that uses τ^2 processors, there is β-chain of $\log \tau$ trees that requires $\Theta(\tau)$ steps. On the other hand, this β-chain can be executed optimally by an off-line algorithm that uses only α processors. This means that if there are k/α β-chains in parallel, the off-line execution time for all chains is equal to the execution time of just one chain.

This observation leads to the following construction: The adversary builds a tree in stages, each stage consisting of k/τ β-chains with a common root. The common root of the β-chains of the $(i+1)$-st stage is a leaf of the tree of the i-th stage. The choices of the roots depend on the on-line algorithm. In particular, consider the on-line execution of some stage. In the first τ steps of the execution, the total number of nodes (including re-computation) that can be executed by the on-line algorithm with k processors is $k\tau$. Among the k/τ β-chains, there exists at least one β-chain with at most τ^2 executed nodes (again including re-computation). Therefore, there exists a β-chain whose nodes are executed by at most τ^2 processors. The adversary then makes a leaf of this β-chain the root of the tree of the next stage.

Consider now a tree with γ stages. At each stage the on-line algorithm executes a β-chain using at most τ^2 processors. It was argued above that this takes time $\Theta(\tau)$ per stage and consequently the on-line execution time is $\Theta(\gamma\tau)$. The optimal algorithm instead will assign one processor to execute the longest path in $\gamma\beta = \gamma \log \tau$ steps. The results of these nodes are transmitted to other processors after a delay of τ time units. Observe that the number of nodes in each stage is $(k/\tau)\beta\alpha = k$, so that the optimal execution of the whole tree needs $\tau + \Theta(\gamma \log \tau)$ steps. Thus, the competitive ratio is $\Omega(\tau / \log \tau)$. □

3.4 Lower bound for other models.

Notice that Theorems 5 and 6 hold for much weaker models of communication. The Papadimitriou-Yannakakis model assumes that all processors broadcast their results to everybody else in τ time units. However, in the proof of Theorems 5 and 6, only one processor of the optimal algorithm, the one that moves down the longest path, communicates its results to other

processors; no other processor communicates its results. We use these weak communication requirements to extend Theorem 6 to other models of parallel computation.

The requirement that a single processor broadcasts its results puts a heavy load on this processor, and the lower bound proof does not apply to other distributed memory parallel models that measure the communication cost more accurately. To derive a general lower bound, we assume a very weak protocol of point to point communication, which is adopted in many parallel computation models [2, 14]: At each step, exactly one processor can send the result of a single node to only one processor; the result arrives at that processor after a delay of τ steps.

Consider the proof of Theorem 6, when $k = \Omega(\tau^2)$, and let the number of stages be $\gamma = m(k-1)$. The proof of Theorem 6 shows that the on-line algorithm needs at least $\Omega(m(k-1)\tau)$ steps to complete the execution of the whole tree.

To find an upper bound on the optimal execution time, we analyze the following off-line algorithm: Again one processor, called the principal processor, executes the nodes of the longest path. It communicates its results to other processors, but this time it does not broadcast the results but it sends them to a single processor. As in the proof of Theorem 6, the remaining nodes are assigned to other processors, but this time a processor is assigned a complete stage. More precisely, after the principal processor executes the root of the i-th stage, it communicates the results to the $(i \bmod k-1)$-st processor which executes all nodes of the i-th stage. It takes $O(m(k-1)\log\tau)$ steps for the principal processor to complete the execution of the longest path and to pass the results to the remaining processors. Furthermore, each of the remaining processors can execute a stage in k steps, because there are k nodes per stage, and it is ready to start the execution of the next stage assigned to it. This is repeated every $(k-1)\log\tau$ steps, which is the time that the principal processor will execute the root of the next stage assigned to the same processor. It is easy to see that the total execution time is at most $m(k-1)\log\tau + k + \tau$.

Combining the lower bound $\Omega(m(k-1)\tau)$ for the on-line execution time and the upper bound $m(k-1)\log\tau + k + \tau$ for the optimal execution time, we can conclude that the competitive ratio is $\Omega(\tau/\log\tau)$.

Thus, we have established this general theorem:

Theorem 7 *For general trees, the competitive ratio of on-line execution by k processors is $\Omega(\min\{\frac{\tau}{\log\tau}, k\})$ for any parallel model with point-to-point communication which requires a delay of τ time units in transmitting a message from one processor to another.*

4 Remarks.

Theorem 7 shows that no compiler can produce an almost optimal (up to a factor of $\tau/\log\tau$) scheduling strategy for executing a parallel program. Since this is based on very weak premises, it applies to many distributed memory parallel computation models. In particular, when it applies to the LogP model, it holds for any value of L when $o = g = 1$. Notice however that for the off-line optimal algorithm, each processor other than the principal processor is busy only for a fraction $1/\log\tau$ of the time. Therefore the optimal off-line algorithm can assign more processors to execute the longest path when o and g are larger than one. Thus, it is possible that similar results hold for a wider range of L, o, g values. The theorem, however, leaves open the possibility of positive results for special classes of programs (dags). Further research is needed in this direction. It would also be very interesting to study the competitive ratio for the LogP model for the values of the parameters that our theorem does not apply.

Acknowledgments. We would like to thank Christos Papadimitriou for his help.

References

[1] N. Alon, J. H. Spencer, and P. Erdös. *The probabilistic method.* Wiley & Sons, Inc., New York, 1992.

[2] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, November 1996.

[3] R. P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51(1):161–166, 1950.

[4] R. L. Graham. Bounds for certain multiprocessor timing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[5] H. Jung, L. M. Kirousis, and P. Spirakis. Lower bounds and efficient algorithms for multiprocessor scheduling of directed acyclic graphs with communication delays. *Information and Computation*, 105(1):94–104, July 1993.

[6] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):79–119, 1988.

[7] W. Loewe and W. Zimmermann. Upper time bounds for executing pram-programs on the logp-machine. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 41–50, 1995.

[8] M. Manasse, L. A. McGeoch, and D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11:208–230, 1990.

[9] R. Motwani, S. Phillips, and E. Torng. Nonclairvoyant scheduling. *Theoretical Computer Science*, 130(1):17–47, 1994.

[10] C. H. Papadimitriou and J. D. Ullman. A communication-time tradeoff. *SIAM Journal on Computing*, 16(4):639–646, August 1987.

[11] C. H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal on Computing*, 19(2):322–328, 1990.

[12] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, February 1985.

[13] A. S. Tanenbaum, M. F. Kaashoek, and H. E. Bal. Parallel programming using shared objects and broadcasting. *Computers*, 25(8):10–20, August 1990.

[14] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[15] I-Chen Wu and H. T. Kung. Communication complexity for parallel divide-and-conquer. In *32nd Annual Symposium on Foundations of Computer Science*, pages 151–162, 1991.

[16] W. Zimmermann and W. Loewe. An approach to machine-independent parallel programming. In *Parallel Programming: CONPAR 94-VAPP VI*, volume 854 of *Lecture Notes in Computer Science*, pages 277–288. Springer-Verlag, 1994.