On Optimal $k$-linear Scheduling of Tree-Like Task Graphs for LogP-Machines

Wolf Zimmermann\textsuperscript{1}, Martin Middendorf\textsuperscript{2}, and Welf Löwe\textsuperscript{1}

\textsuperscript{1} Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, 76128 Karlsruhe, Germany, E-Mail: {zimmer|loewe}@ipd.info.uni-karlsruhe.de
\textsuperscript{2} Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe, 76128 Karlsruhe, Germany, E-Mail mmi@aifb.uni-karlsruhe.de

Abstract. A $k$-linear schedule may map up to $k$ directed paths of a task graph onto one processor. We consider $k$-linear scheduling algorithms for the communication cost model of the LogP-machine, i.e. without assumption on processor bounds. The main result of this paper is that optimal $k$-linear schedules of trees and tree-like task graphs $G$ with $n$ tasks can be computed in time $O(n^{k+1} \log n)$ and $O(n^{k+3} \log n)$, respectively, if $o \geq g$. These schedules satisfy a capacity constraint, i.e., there are at most $\lceil L/g \rceil$ messages in transit from any or to any processor at any time.

1 Introduction

The LogP-machine [3, 4] assumes a cost model reflecting latency of point-to-point-communication in the network, overhead of communication on processors themselves, and the network bandwidth. These communication costs are modeled with parameters Latency, overhead, and gap. The gap is the inverse bandwidth of the network per processor. In addition to $L$, $o$, and $g$, parameter $P$ describes the number of processors. Furthermore, the network has finite capacity, i.e. there are at most $\lceil L/g \rceil$ messages in transit from any or to any processor at any time.

If a processor would attempt to send a message that exceeds this capacity, then it stalls until the message can be sent without violation of the capacity constraint. The LogP-parameters have been determined for several parallel computers [3, 4, 6, 8]. These works confirmed all LogP-based runtime predictions.

Much research has been done in recent years on scheduling task graphs with communication delay [9–11, 14, 21, 23, 25], i.e. $o = g = 0$. In the following, we discuss those works assuming an unrestricted number of processors. Papadimitriou and Yannakakis [21] have shown that given a task graph $G$ with unit computation times for each task, communication latency $L \geq 1$, and integer $T_{\text{max}}$, it is NP-complete to decide whether $G$ can be scheduled in time $\leq T_{\text{max}}$. Their results hold no matter whether redundant computations of tasks are allowed or not. Finding optimal time schedules remains NP-complete for DAGs obtained by the concatenation of a join and a fork (if redundant computations are allowed) and for fine grained trees (no matter if redundant computations are allowed or...
A task graph is fine grained if the granularity $\gamma$ which is a constant closely related to the ratio of computation and communication is $< 1$ [22]. Gerasoulis and Yang [9] find schedules for task-graphs guaranteeing the factor $1 + \frac{1}{\gamma}$ of the optimal time if recomputation is not allowed. For some special classes of task graphs, such as join, fork, coarse grained (inverse) trees, an optimal schedule can be computed in polynomial time [2, 9, 15, 25]. If recomputation is allowed some coarse grained DAGs can also be scheduled optimally in polynomial time [1, 5, 15]. The problem to find optimal schedules having a small amount of redundant computations has been addressed in [23].

Recently, there are some works investigating scheduling task graphs for the LogP-machine without limitations on the number of processors [12, 16–18, 20, 24, 26] and with limitations on the number of processors [19, 7]. Most of these works discuss approximation algorithms. In particular, [18, 26] generalizes the result of Gerasoulis and Yang [9] to LogP-machines using linear schedules. [20] shows that optimal linear schedules for trees and tree-like task graphs can be computed in polynomial time if the cost of each task is at least $g - o$. However, linear schedules for trees often have bad performance. Unfortunately, if we drop the linearity, even scheduling trees becomes NP-complete [20]. [24] shows that the computation of a schedule of length at most $B$ is NP-complete even for fork and join trees and $o = g$. They discuss several approximation algorithms for fork and join trees. [12] discusses an optimal scheduling algorithm for some special cases of fork graphs. In this paper, we generalize the linearity constraint and allow instead to map up to $k$ directed paths on one processor. We show that it is possible to compute optimal $k$-linear schedules on trees and tree-like task graphs in polynomial time if $o = g$. In contrast to most other works on scheduling task graphs for the LogP-machine, we take the capacity constraint into account.

Section 2 gives the basic definitions used in this paper. Section 3 discusses some normalization properties for $k$-linear schedules on trees. Section 4 presents the scheduling algorithm with an example that shows the improvement of $k$-linearity w.r.t. linear schedules.

## 2 Definitions

A task graph is a directed acyclic graph $G = (V, E, \tau)$, where the vertices are sequential tasks, the edges are data dependencies between them, and $\tau_v$ is the computation cost of task $v \in V$. The set of direct predecessors $PRED_v$ of a task $v$ in $G$ is defined by $PRED_v = \{u \in V \mid (u, v) \in E\}$. The set of direct successors $SUCC_v$ is defined analogously. Leaves are tasks without predecessors, roots are tasks without successors. A task $v$ is ancestor (descendant) of a task $u$ iff there is a path from $v$ to $u$ ($u$ to $v$). The set of ancestors (descendants) of $u$ is denoted by $ANC_u$ ($DESC_u$). A set of tasks $U \subseteq V$ is independent iff for all $u, v \in V$, $u \neq v$, neither $u \in ANC_v$ nor $v \in ANC_u$. A set of tasks $U \subseteq V$ is $k$-independent iff there is an independent subset $W \subseteq U$ of size $k$. $G$ is an inverse tree iff $|SUCC_v| \leq 1$ and there is exactly one root. In this case, $SUCC_v$