Partitioning and Scheduling to Counteract Overhead

Roni Khardon and *Shlomit S. Pinter*
Dept. of Electrical Engineering
Technion — Israel Institute of Technology
Haifa 32000, Israel

Abstract

We introduce a scheduling model, inspired by data flow computers, in which the overhead incurred in a system as well as computation time are described explicitly. Using this model, we provide algorithms for partitioning programs so as to minimize their completion time.

In the traditional data flow paradigm, every instruction is considered a "task", and it is scheduled for execution as early as possible. Implementations of this scheme, however, involve overheads which affect the running time of the programs. We propose to partition the program into larger grains, each containing one or more instructions, such that scheduling those grains would result in minimizing the completion time. Our model accounts for both the overhead incurred when executing a program and the actual execution time of its instructions.

Within this framework we derive lower and upper bounds on the execution time of programs represented as trees and DAGs. We provide algorithms for optimally partitioning such programs when there are enough execution units. The algorithms have time complexity $O(|V|^2)$ and $O(|V|^5)$ for trees and DAGs, respectively (where $|V|$ is the number of nodes). In the case of a limited number of execution units, we show that the algorithm for trees approximates the best solution with a ratio of 4. Using the same proof techniques we show sufficient conditions for approximating the problem for DAGs, noting that approximation is the best that can be sought for as the problem is NP-complete.

The scheduling problem solved is general, and its solution can also be used for scheduling problems which have been studied before outside the data flow paradigm. Some of the results are also applicable to existing data flow machines, and to simulations of data flow programs on other machines.

Key words: computation modeling, data-flow computers, grain size, partitioning, scheduling.

1 Introduction

Parallel computation is based on the idea that a task can be divided into many small sub-tasks each to be handled by a different processor. Ideally, the task is partitioned into independent parts, and each of them is run separately. This is, however, only rarely possible. Therefore, programs are partitioned into parts that need to communicate with each other, by sending messages or transferring operands. When such a program is run on a real computer, these communication steps consume time. The communication process is not part of the original program and is therefore referred to as *overhead*. Thus, when partitioning a program to be executed on a parallel machine, one must take into account both computation and communication (or any other overhead). This paper introduces a scheduling model, inspired by data flow computers, in which the overhead incurred in a system as well as computation time are described explicitly. Using this model, we provide algorithms for partitioning programs so as to minimize their completion time.

The basic idea of data flow computers [6] is to achieve the maximum parallelism out of a program by dynamically scheduling every instruction as soon as possible. The implementation of this idea involves overhead for moving instructions and results in the machine, and for executing the match operation i.e. matching actual parameters (arguments) to operations. Data flow programs are represented as directed graphs; a node in the graph represents an operation, and an edge in the graph represents a data dependence between two operations. Data flow computers schedule such programs at run time. Graph nodes (operations) are scheduled as soon as their arguments arrive, and executed as soon as there is a free execution unit to handle them. In this work we consider scheduling *grains* instead of single instructions. A grain is a cluster of instructions, that will be scheduled together, to be executed on a single execution unit, as soon as the arguments for all its nodes arrive. When using grains, that have been carefully chosen to fit the machine parameters and program structure, some of the intermediate results would not consume "communication" time, and therefore the total execution time may be reduced.

We introduce a scheduling model capturing both communication and computation times, and consider programs which can be represented as trees or Directed Acyclic Graphs (DAGs). With this model we look for grains, such that scheduling them yields the shortest execution time on a given machine. This work is related both to data Flow computers, and to scheduling models that have been studied in different contexts.

Scheduling problems that include overhead have been extensively studied before. In Section 4 we relate our model to some previously studied models (that do not necessarily consider single instructions or data flow execution), and show when our results apply. We also introduce some notation that may be useful in comparing scheduling models with overhead.

In Section 5 we describe the relation of our work to existing data flow computers. The implementation of data flow computers is complicated, and no single "right"solution exists. This has yielded a variety of data flow machine architectures, and our model fits some of these implementations. As mentioned above our algorithms find partitions of acyclic programs which correspond to "basic blocks". We discuss how one can use our algorithms in a more general setting, of a compiler for a whole program that does have cycles (e.g. loops and recursive calls). We also describe the relation to previous work that considered combining the benefits of Von Neumann and data flow architectures.

The problem of optimally scheduling a DAG on our data flow machine model is a generalization of the precedence constrained scheduling problem which is NP-complete [7, 29]. Therefore an exact solution for the problem is not expected. We first consider the relaxation of the problem where there is an *unlimited number of processors*. For this case, we provide a lower bound for the generalized problem of scheduling computation trees in our model, and algorithms for optimally dividing a program with n nodes into grains for both trees (time complexity $O(n^2)$) and DAGs (time complexity $O(n^5)$). We then consider the case with a *limited number of processors*. We show that the grains chosen for trees for an unlimited number of processors approximate the best solution with a ratio of 4, for the case with a limited number of processors. In the case when there are no unary operations in the tree we show that the ratio is 3. Using the same proof techniques we show sufficient conditions for approximating the problem for DAGs.

The rest of the paper is organized as follows. In Section 2 we present our model of data flow machines, and formally define the problem of optimally computing their programs. Section 3 includes our results which are stated within the framework of the model. Then, in Sections 4 and 5, we review the relation to scheduling models and the applicability of the results to various data flow architectures. Finally, in Section 6 we discuss the conclusions and some questions for further research.

2 The Model

Our model of a data flow computer consists of one control unit, several execution units, and three data structures (see Figure 1). The model is intended to describe the way in which instructions are dynamically scheduled for execution, thereby allowing us to identify the sources of overhead. Instruction templates, are simply instructions with slots to hold their arguments.

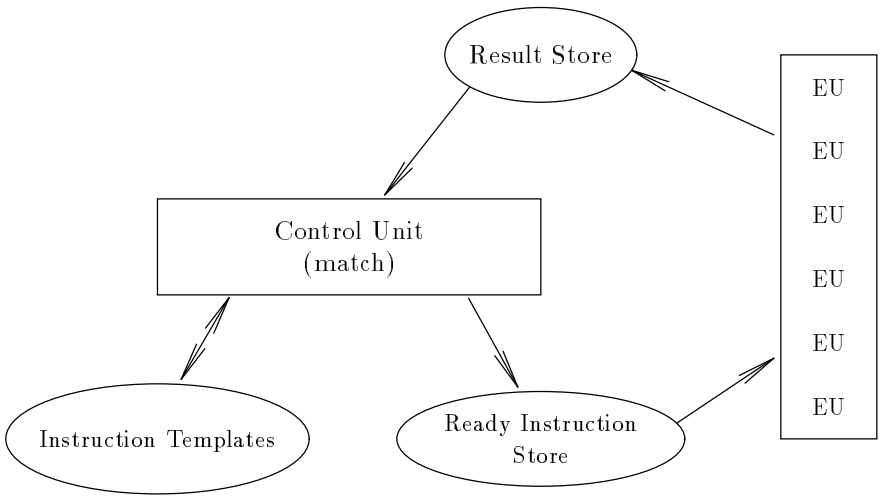


Figure 1: A model for data flow machines.

When all the arguments for an instruction template arrive, the template is ready for execution, and therefore passed to the *ready instructions store*. There, the instructions get fetched and executed. The results are passed to the *result store* and afterwards moved into their destination (matched).

A program for a data flow machine is represented as a DAG. A node in the graph represents an operation; an edge in the graph represents a data dependence between two operations. The leaves of the graph are the input to the program.

The basic idea put forward in this paper is to schedule a set of instructions as one unit. A *grain* is a set of one or more instructions (a subgraph of the graph). The arguments of the grain are the arguments for the contained instructions whose dependence edges are not in the grain. The output of a grain is a single value (usually the output of the most descendent node in the grain). The restriction that grains have only a single output does not cause longer schedules when there are enough execution units, and helps in finding good solutions in the case with limited number of execution units. In the data flow machine, a whole grain is scheduled to a single execution unit and is executed there. We assume the existence of enough registers (or memory) and some control mechanism on the execution units that will enable execution of grains instead of instructions. Scheduling grains saves some of the run-time communication overhead and reduces program execution time.

The assumptions and parameters describing execution time in the model are:

- The execution time of a single instruction v is $ex(v)$.
- The size of each grain is linear in the number of the arguments to the grain. Passing one argument from the instruction ready store to the execution units takes t_f time units. Therefore passing a whole grain from the instruction ready store to the execution units takes $k \times t_f$ time units, where k is the number of arguments to that grain[1].
- Writing the result plus the match operation and moving a grain to the ready instruction store takes t_m time units.
- The program's output (which is the output of some node(s)) is delivered through the network, and takes t_m time units as well.

Let t_0 be the time in which the last argument of a k argument grain, P, arrives at the ready instruction store. Then the execution of P can start at $t_1 = t_0 + k \times t_f$, and end at $t_2 = t_1 + \sum_{v \in P} ex(v)$. The computed result will be in the ready instruction store at time $t_3 = t_2 + t_m$.

Denoting the time t_0, the execution starting time of the grain, by $\mathrm{WS}(grain)$, $t_2 - t_0$ by $t(grain)$, and t_3 by $T(grain)$ we get: $T(P) = \mathrm{WS}(P) + t(P) + t_m$.

Example: Consider the expression tree in Figure 2. We can partition it either into one grain including both operations or into two grains each containing one of the operations.

Assuming $ex(v) = 1$ for all nodes, in the first case we have[2]:

$$t(\text{root grain}) = 2 + 3 \times t_f.$$
$$\mathrm{WS}(\text{root grain}) = 0.$$
$$T(\text{root grain}) = 2 + 3 \times t_f + t_m.$$

[1] The size of a grain is the sum of operands length and the instruction codes length. The above assumption models a case where instruction code length is negligible. To overcome this inaccuracy (if needed) one can add the time taken to fetch the instruction code of node v (this time should be: (code length) $\times t_f$), to $ex(v)$. Since every instruction in a grain is fetched once and executed once, this gives an accurate description for the execution time.

[2] The root grain is the grain containing the root of the expression tree; this grain's output is the tree's result.

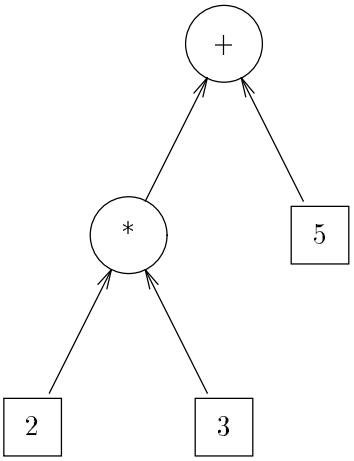


Figure 2: An expression tree.

In the second case we have:

$$t(\text{root grain}) = 1 + 2 \times t_f.$$
$$\text{WS}(\text{root grain}) = 1 + 2 \times t_f + t_m.$$
$$T(\text{root grain}) = 2 + 4 \times t_f + 2 \times t_m.$$

It is clear that for this example the larger grain yields faster execution.

Another approach would consider fetch time to be a constant rather than linear in the number of arguments. This case happens when the overhead in starting communication governs the communication time. Although we continue using the first approach throughout the paper, our results hold for the second approach too. This is discussed in the end of Section 3.3.

2.1 Formalization and Notations

In order to formalize the problem we use the following definitions:

- Let $G = (V, E)$ be an acyclic directed graph representing a program, where V is the set of nodes and $E \subseteq V \times V$ is the set of edges. By $(u, v) \in E$ we denote an edge directed from u to v. Let $u, v \in V$. Node u is a *predecessor* of node v if there is a directed path from u to v in G. Node u is a *direct predecessor* of v if there is an edge (u, v) in G.

- A *grain* $G_i = (V_i, E_i)$ is a connected sub-dag, $G_i \subset G$, that does not contain leaves.

$$V_i \subset V \text{ and } E_i \equiv \{(u, v) \in E \mid u, v \in V_i\}.$$

- A *grain cover* is a set of grains, $\{G_i = (V_i, E_i)\}$, containing all internal (non leaf) nodes of G.

$$V_i \subseteq V \;\; ; \;\; \cup_i \{V_i\} = V \setminus \{v \mid v \text{ is a leaf}\}.$$

- The input degree of a grain is the number of nodes that are sources of edges directed to nodes in the grain. That is the number of input operands to the grain.

$$d_{in}(G_i) \equiv \mid \{u \mid \; (u, v) \in E \;,\; u \notin V_i \;,\; v \in V_i\} \mid .$$

- The *execution time* for the grain is the time to move the ready grain to the execution unit plus the time needed to execute the instructions included in the grain.

$$t(G_i) \equiv t_f \times d_{in}(G_i) + \sum_{v \in V_i} ex(v).$$

- The execution of a grain cannot be started before the computation of all its operands has been completed.

$$\mathrm{WS}(G_i) \geq \max_U \left\{ T(U) \;\middle|\; \begin{array}{l} U \text{ is a grain, or } U = \{y\} \text{ where } y \text{ is a leaf, } U \neq V_i \\ \text{and there is an edge in } E \text{ from } u \in U, u \notin V_i, \text{ to } v \in V_i \end{array} \right\}.$$

- Leaves are ready at time 0. Formally we denote this as:

$$T(\{y\}) \equiv 0, \text{ where } y \text{ is a leaf.}$$

- The completion time of a grain is: $T(G_i) \equiv \mathrm{WS}(G_i) + t(G_i) + t_m$.
- A grain is *active* when an execution unit is handling it. That is, when the time `t` satisfies:

$$\mathrm{WS}(G_i) < \mathtt{t} \leq \mathrm{WS}(G_i) + t(G_i).$$

The Scheduling Problem: Given the input composed of:

- A DAG, $G = (V, E)$, representing the program.
- Parameters, t_f, t_m, representing the overhead in the machine.
- The number of execution units, p.
- A mapping $ex : V \rightarrow N$, representing the execution times of instructions in the program.

Find a grain cover, $\{G_i\}$, and a mapping $\mathrm{WS} : \{G_i\} \rightarrow \mathcal{R}$, such that the number of active grains at any time t is less than or equal to p, and such that $T(\mathrm{DAG}) \equiv \max_i\{T(G_i)\}$ is minimized.

We later use the following notation: Let $D \subset G$ be a connected DAG, then

$$\mathrm{DPRD}(D) \equiv \{u \,|\, u \notin D \text{ is a direct predecessor of a node in } D\}.$$

2.2 One Level Machines

In Section 5 we show the correspondence between our model and the different data flow computers. We use the terminology from [30] which describes data flow machines with respect to communication networks. To enable this discussion we also consider a variant of the model which is denoted as one level machines. In these machines there is a distinct match unit dedicated to each execution unit; this enables overlapping the fetch and execute phases.

For one level machines (assuming execution phase time equals fetch phase time denoted t_f)

$$t(grain) = t_f \times \max\{\text{number of operands} + 1, \text{number of operations} + 2\}.$$

This equation corresponds to either fetching all operands and then executing the last operation (the execution of other operations is overlapped with the fetch phase) when there are more operands than operations, or fetching the two operands for the first operation and then executing all other operations (fetching other operands is overlapped with the execution) when there are more operations than operands. Section 3.2.1 describes an adaption of the algorithm for trees, that is useful for one level machines, and proves the optimality in this model too.

3 Technical Results

We start by considering the case with an unlimited number of execution units. A lower bound for the completion time of trees is derived, and algorithms which produce optimal instruction scheduling for trees and DAGs are presented. For the case with limited number of execution units, we show that the algorithm for trees approximates the best solution with ratio 4, and give sufficient conditions for approximating the problem for DAGs.

We first give the intuition behind the algorithms. The brute force way is to consider all the possible grains rooted at a node v as candidates, and choose the best one. This is of course an exponential search which we want to avoid. The algorithm for trees considers only a small number of candidates, one of which is the optimal grain. The choice of candidates is based on two properties of the model. The first property is that the execution time of a tree shaped grain, $t(g)$, is monotone. That means that if we compare two grains, g_1 and g_2, that are rooted at the same node, and $g_1 \subset g_2$ then $t(g_1) < t(g_2)$ (this is proved in Lemma 1). The second property is the *waiting property* stating that the execution of a grain cannot be started before all its operands are ready. Suppose we know the earliest possible completion time of all predecessors of a node v, and we want to start executing the grain computing v at time t_0. Then the above two properties induce a natural partition as follows: all predecessors of v with completion time greater than t_0 must be included in v's grain, as otherwise it cannot be started at t_0. All other nodes must not be included in the grain, as by the monotonicity property including such a node would only increase execution time, and therefore completion time too. So for each possible starting time t_0 we have a natural grain realizing the best completion time of all grains started at t_0. We find the global minimum by enumerating all possible starting times, and choosing the starting time (and its grain) that minimizes completion time. Since the number of different starting times is bounded by the number of nodes in the graph, this yields a polynomial time algorithm.

The monotonicity property is not true for DAGs, as inclusion of nodes might reduce the number of operands to be fetched. The natural partition is still partly valid; all the nodes that are completed after t_0 must be included in the grain. We call these nodes the "upper part" of the grain. To this set we should add nodes such that (still under the requirement that the grain is started at t_0) $t(g)$ is minimized. The algorithm for DAGs solves this subproblem, of adding nodes to reduce $t(g)$, using an auxiliary max flow subproblem. The high level structure of this procedure is very similar to the one for trees. We enumerate all possible starting times, and choose the starting time and grain that minimize completion time. The only difference is that for each starting time we have to solve a max flow problem, and therefore the algorithm takes more time.

Both algorithms find an optimal grain for each node in the graph. Each such grain has one

output which is the result of the node. When using the set of all optimal grains as the grain cover, some nodes might appear, and therefore also be computed, in more than one grain. Notice that for trees this is not necessary, but for DAGs re-computation of some nodes might be necessary in order to minimize the running time. In both cases, our algorithm makes one more pass on the graph to choose only the *required grains*. Required grains are defined recursively as follows: A grain is required if it computes the root node, or if it computes the operand of a required grain. Therefore, every operand of any required grain has a special grain that computes it. Furthermore, every required grain has exactly one output. Therefore, the scheme of re-computation with required grains guarantees that the dependency structure between the grains is acyclic. This ensures that our partition does not suffer from deadlock.

The results in this section are oblivious to the values of parameters. That means that the guaranteed performance is true for every value of the parameters. In some special cases the value of the parameters might help characterize the structure of grain covers, and therefore help to find the optimal cover (with reduced time complexity). For example in the end of Section 3.3 we show that if $t_f < ex = \min_v ex(v)$ then we can reduce the running time of the algorithm for DAGs.

3.1 Bounds

We first prove a simple lemma on the computation times of subtrees.

Lemma 1 *Consider two binary program trees $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ such that $G_1 \subset G_2$. Then $t(G_1) < t(G_2)$.*

Proof : The execution time of a tree grain $G = (V, E)$ is

$$t(G) = \sum_{v \in V} ex(v) + d_{in}(G) \times t_f .$$

Now, $G_1 \subset G_2$ implies that $d_{in}(G_1) \leq d_{in}(G_2)$ (as every node we add to G_1 in order to create G_2 has at least one new direct predecessor). In addition $|V_1| < |V_2|$ concluding the proof. ∎

To get the lower bound for trees let h be the height of the tree, and consider the "heaviest" chain to be the set of nodes G_* composed of all nodes on a directed path from a leaf to the root for which $t_f \times d_{in}(G_*) + \sum_{v \in G_*} ex(v)$ is maximal. Consider the chain G_* as one grain, and denote the grain cover using G_* by S_1. The completion time of S_1, which is $T(G_*)$ is at least $t(G_*) + t_m$. In the following we show that $t(G_*) + t_m$ is a lower bound for the completion time of the tree.

Consider any other grain cover for the tree, S_2. The cover S_2 separates G_* to j parts ($j \leq h$). Every part of G_* belongs to a grain in S_2. Denote these parts of G_* by $Y_1, \ldots, Y_j$, and the corresponding grains of S_2 by $Z_1, \ldots, Z_j$, and let Z_j be the grain containing the root. For all i, $Y_i \subseteq Z_i$. Using Lemma 1 we have:

$$T_{S_2}(Z_j) \quad \geq \quad \sum_{i=1}^{j} (t(Z_i) + t_m)$$

$$
\begin{aligned}
&\geq \sum_{i=1}^{j} (t(Y_i) + t_m) \\
&= \sum_{i=1}^{j} \left(d_{in}(Y_i) \times t_f + t_m + \sum_{v \in Y_i} ex(v) \right) \\
&\geq d_{in}(G_*) \times t_f + t_m + \sum_{v \in G_*} ex(v).
\end{aligned}
$$

This provides a lower bound in terms of the graph and the model parameters alone. Since Lemma 1 is not true for DAGs, this lower bound is applicable only to trees.

An obvious upper bound for the completion time is the case when every node is a grain. In this case if the depth of the DAG is h, and $ex(v) = 1$ for all nodes, then the DAG can be completed in $h \times (2 \times t_f + t_m + 1)$ time units. However this solution is not optimal. As an example, in the case of scheduling a binary chain (a path in a binary tree), our algorithm finds a solution (which is the lower bound) with a completion time at least half that of the upper bound.

3.2 The Algorithm for Binary Trees

In this section we present an algorithm to partition a program graph whose structure is a binary tree. We later show that a restriction of the algorithm for DAGs (presented in Section 3.3) can be used for general trees. The results for the case with limited number of execution units, that are stated as results for trees, hold for both algorithms (general trees).

Assuming an unlimited number of processors, all we have to do is to calculate the best possible completion time $T(grain)$ for the grain containing the root of the tree. The algorithm traverses the tree from leaves to root (in postorder), calculating for every node v its best completion time $T(v)$ associated with an optimal grain ST(v). For an internal node v this is done by sorting all the nodes in the grains of the direct predecessors according to their best completion times, and computing a grain for v assuming each time a different node v_i is the last one computed before the grain of v can be started. The final grain of v is chosen to be the one which provides the best completion time.

The Algorithm:
Given a tree $G = (V, E)$, numbers t_f, t_m, and a mapping $ex : V \rightarrow N$, traverse the tree from leaves to root (in postorder), computing for each node as follows:

ST(v) — a subtree rooted at v which is the grain calculating v and delivering its result at the best possible time.

WS(v) — the start time of ST(v). (The completion time of the worst direct predecessor of ST(v).)

$t(v)$ — the number of time units needed to execute the grain ST(v).

$T(v)$ — the completion time of ST(v).

1. For all leaves v define $\mathrm{ST}(v) \equiv \phi$, $T(v) \equiv 0$, $\mathrm{WS}(v) \equiv 0$.

2. For a node v having two direct predecessors x, y (x and y have been evaluated due to postorder):

   (a) Define:
   - $\tilde{S}_*(v) = \{v\} \cup \mathrm{ST}(x) \cup \mathrm{ST}(y)$.
   - $S_*(v) = \tilde{S}_*(v) \setminus \{z \,|T(z) \leq \max\{\mathrm{WS}(x), \mathrm{WS}(y)\}\}$.

   (b) Let $|S_*(v)| = k+1$. Sort the $(k+1)$ nodes of $S_*(v)$ in decreasing order of their completion times to get[3]:

   $$T(v_0 = v) > T(v_1) \geq T(v_2) \geq \cdots \geq T(v_k).$$

   (c) For all $i = 1, \ldots, k$ calculate:
   - $\mathrm{ST}_i(v) = \{z \mid z \in S_*(v), T(z) > T(v_i)\}$.
   - $\mathrm{WS}_i(v) = T(v_i)$.

   For $k+1$ calculate:
   - $\mathrm{ST}_{k+1}(v) = S_*(v)$.
   - $\mathrm{WS}_{k+1}(v) = \max\{\mathrm{WS}(x), \mathrm{WS}(y)\}$.

   For all $i = 1, \ldots, (k+1)$ calculate :
   - $t_i(v) = t_f \times d_{in}(\mathrm{ST}_i(v)) + \sum_{s \in \mathrm{ST}_i(v)} ex(s)$.
   - $T_i(v) = \mathrm{WS}_i(v) + t_i(v) + t_m$.

   (d) Choose of the $k+1$ possible grains for v the best one (denoted by index i^*)
   - $T(v) = T_{i^*}(v) = \min_{i=1}^{k+1}\{T_i(v)\}$.
   - $\mathrm{WS}(v) = \mathrm{WS}_{i^*}(v)$.
   - $t(v) = t_{i^*}(v)$.
   - $\mathrm{ST}(v) = \mathrm{ST}_{i^*}(v)$.

3. For a node v having only one direct predecessor, y. If y is not a leaf, let:

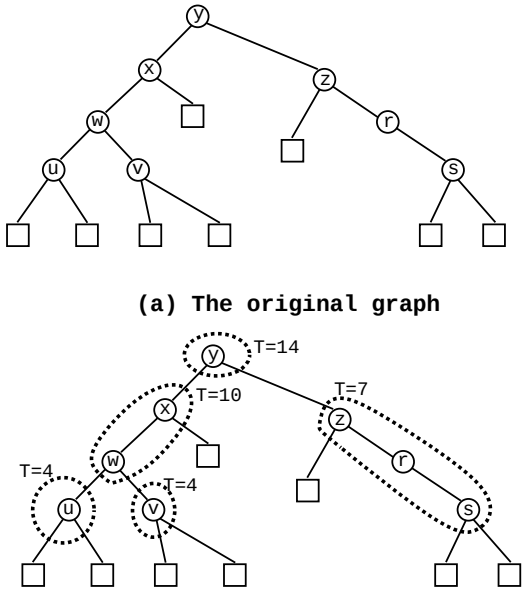
   - $\mathrm{ST}(v) = \{v\} \cup \mathrm{ST}(y)$.
   - $\mathrm{WS}(v) = \mathrm{WS}(y)$.
   - $t(v) = t(y) + ex(v)$.
   - $T(v) = T(y) + ex(v)$.

   If y is a leaf, let:

   - $\mathrm{ST}(v) = \{v\}$.
   - $\mathrm{WS}(v) = 0$.
   - $t(v) = t_f + ex(v)$.
   - $T(v) = t_f + ex(v) + t_m$.

[3] Lemma 1 implies that $T(v) > T(v_1)$ and that each $\mathrm{ST}_i(v)$ of Step (2c) is a connected subtree. The Lemma actually implies that if $u \in \mathrm{ST}_i(v)$ and $u \to \cdots \to x \to \cdots \to v$ then $x \in \mathrm{ST}_i(v)$. This is the "convexity" property described in [24], which is stronger than connectivity.

4. Now we have a grain associated with each node. However, grains might not be disjoint. For trees it is sufficient for each node to be calculated only in one grain. This induces a grain cover comprising only *required* grains which are defined as follows: (a) The root's grain is required. (b) For every direct predecessor x of a required grain, the grain rooted at x is required.



(a) The original graph

(b) The grain cover

Figure 3: A tree program graph and its partition.

Example: We illustrate the algorithm using the program graph given in Figure 3(a). We assume in this example that $t_f = t_m = ex = 1$. The algorithm traverses the tree from the leaves to the root. For the nodes u, v and s there is only one possibility for the grain. That is, $\mathrm{ST}(u) = \{u\}$, $\mathrm{WS}(u) = 0$, and $T(u) = 2t_f + ex + t_m = 4$. Similarly, $T(v) = T(s) = 4$.

For node w, the algorithm first computes $\tilde{S}_* = S_* = \{w, u, v\}$. In this case $k = 2$ and the algorithm sorts the nodes in S_* to get $T(w) > T(u) = T(v)$. It then computes $\mathrm{ST}_1(w) = \mathrm{ST}_2(w) = \{w\}$, $\mathrm{WS}_1(w) = 4$, and $T_1(w) = 4 + 2t_f + ex + t_m = 8$. For $k + 1$ it computes $\mathrm{ST}_3(w) = \{w, u, v\}$, $\mathrm{WS}_3(w) = 0$, and $T_3(w) = 0 + 4t_f + 3ex + t_m = 8$. It then (arbitrarily) chooses $i^* = 1$ as its partition.

For node x, we get $\tilde{S}_* = S_* = \{x, w\}$, and $k = 1$. For $i = 1$ the algorithm computes $\mathrm{ST}_1(x) = \{x\}$, $\mathrm{WS}_1(x) = 8$, and $T_1(x) = 8 + 2t_f + ex + t_m = 12$. For $k + 1$ it computes $\mathrm{ST}_2(x) = \{x, w\}$, $\mathrm{WS}_2(x) = 4$, and $T_2(x) = 4 + 3t_f + 2ex + t_m = 10$. It therefore chooses $i^* = 2$ as its partition.

Node r has only one direct predecessor and is therefore handled in Step (3). This yields $\mathrm{ST}(r) = \{r, s\}$, $\mathrm{WS}(r) = 0$, and $T(r) = 0 + 2t_f + 2ex + t_m = 5$.

For node z, we get $\tilde{S}_* = S_* = \{z, r, s\}$, $k = 2$, and $T(z) > T(r) > T(s)$. For $i = 1$ the algorithm computes $\mathrm{ST}_1(z) = \{z\}$, $\mathrm{WS}_1(z) = 5$, and $T_1(z) = 5 + 2t_f + ex + t_m = 9$. For $i = 2$ the algorithm computes $\mathrm{ST}_2(z) = \{z, r\}$, $\mathrm{WS}_2(z) = 4$, and $T_2(z) = 4 + 2t_f + 2ex + t_m = 9$.

For $k+1$ it computes $\mathrm{ST}_3(z) = \{z, r, s\}$, $\mathrm{WS}_3(z) = 0$, and $T_3(z) = 0 + 3t_f + 3ex + t_m = 7$. It therefore chooses $i^* = 3$ as its partition.

The last node to be considered is y. This time we get that $\tilde{S}_* \neq S_*$. In particular, $\tilde{S}_* = \{y, x, w, z, r, s\}$, and $S_* = \{y, x, w, z, r\}$. We have $k = 4$ and the ordering of the nodes is $T(y) > T(x) > T(w) > T(z) > T(r)$. For $i = 1$ the algorithm computes $\mathrm{ST}_1(y) = \{y\}$, $\mathrm{WS}_1(y) = 10$, and $T_1(y) = 10 + 2t_f + ex + t_m = 14$. For $i = 2$ the algorithm computes $\mathrm{ST}_2(y) = \{y, x\}$, $\mathrm{WS}_2(y) = 8$, and $T_2(y) = 8 + 3t_f + 2ex + t_m = 14$. For $i = 3$ the algorithm computes $\mathrm{ST}_3(y) = \{y, x, w\}$, $\mathrm{WS}_3(y) = 7$, and $T_3(y) = 7 + 4t_f + 3ex + t_m = 15$. For $i = 4$ the algorithm computes $\mathrm{ST}_4(y) = \{y, x, w, z\}$, $\mathrm{WS}_4(y) = 5$, and $T_4(y) = 5 + 5t_f + 4ex + t_m = 15$. Lastly, for $i = 5$ the algorithm computes $\mathrm{ST}_5(y) = \{y, x, w, z, r\}$, $\mathrm{WS}_5(y) = 4$, and $T_5(y) = 4 + 5t_f + 5ex + t_m = 15$, and $i^* = 1$ is chosen as its grain. This completes the part of computing the optimal grain for every node.

Next the algorithm identifies the required grains. The grain for y is $\{y\}$. This grain has two operands x, z. Therefore, the grains for x and z, which are respectively $\{x, w\}$ and $\{z, r, s\}$, are required. The grain for z has only leaves for operands so it does not require any other grain for its computation. The grain for x has two non-leaf operands u, and v, and their grains are required. The final grain partition is shown in Figure 3(b). Notice that the grains for w, r, s are not required and therefore do not appear in the final partition.

Lemma 2 *Consider a DAG program graph $G = (V, E)$, and two distinct nodes $u, v \in V$. If there is a path from u to v then $T_{optimal}(u) < T_{optimal}(v)$*

Proof : Consider G_v, the best grain calculating v. If G_v does not include u then it is clear that $T(v) > \mathrm{WS}(v) \geq T(u)$. Otherwise consider the grain G_{uv} containing all the nodes in G_v except for v. The grain G_{uv} finishes its execution at least $ex(v)$ time units before G_v, therefore $T(u) < T(v)$. ∎

Theorem 1 *Let G=(V,E) be a program graph which is a binary tree. For every node v, $T(v)$— the completion time of the grain rooted at v calculated by the algorithm — is the smallest over all possible grain covers having a grain rooted at v.*

Proof : The proof is by contradiction. Assume there is a grain cover in which the completion time of a grain rooted at some node z is less than $T(z)$. Let v be the first node for which there exists such a grain cover. Denote this grain cover by o, the grain of o rooted at v by $\mathrm{ST}_o(v)$, and the completion time of $\mathrm{ST}_o(v)$ by $T_o(v)$. The assumption implies $T_o(v) < T(v)$, and that for all predecessors x of v, $T(x)$— the completion time calculated by the algorithm, is optimal.

As v is a grain's root, $T_o(v)$ can be expressed as

$$T_o(v) = t_o(v) + \mathrm{WS}_o(v) + t_m.$$

The rest of the proof is done by case analysis:

Case 1: The node v has two direct predecessors x, y.

Case 1.1: At least one of x, y is a leaf.

If both x and y are leaves then the only way to calculate v is to put v alone in a grain, getting $T(v) = ex(v) + 2 \times t_f + t_m$. So $T_o(v)$ cannot be smaller than $T(v)$.

For every node u in G define $\tilde{\mathrm{ST}}(u)$ to be the subtree rooted at u containing all internal nodes (not leaves) of G which are below u.

$$\tilde{\mathrm{ST}}(u) = \{z \mid z \text{ is not a leaf and there is a directed path from } z \text{ to } u\}.$$

Define:

- $\mathrm{ST}_{ov}(x) \equiv \mathrm{ST}_o(v) \cap \tilde{\mathrm{ST}}(x)$,
- $\mathrm{ST}_{v,i}(x) \equiv \mathrm{ST}_i(v) \cap \tilde{\mathrm{ST}}(x)$,

(and the same for y).
If x is a leaf and y is not a leaf then $T_o(v)$ can be expressed as either

$$\begin{aligned} T_o(v) &= ex(v) + t_f + t_{ov}(y) + \max\{0, \mathrm{WS}_{ov}(y)\} + t_m \\ &= ex(v) + t_f + T_{ov}(y), \end{aligned}$$

or (if y is not in the grain rooted at v)

$$T_o(v) = ex(v) + 2 \times t_f + T_o(y) + t_m.$$

In both cases knowing that $T(y)$ is optimal we get:

$$\begin{aligned} T_o(v) &\geq ex(v) + t_f + T_o(y) \\ &\geq ex(v) + t_f + T(y) \\ &= ex(v) + t_f + t(y) + \max\{0, \mathrm{WS}(y)\} + t_m \\ &= T_{k+1}(v) \\ &\geq T(v). \end{aligned}$$

The case when x is not a leaf and y is a leaf is symmetric, so this completes the analysis of case 1.1.

Case 1.2: Both x and y are not leaves.

Case 1.2.1: Suppose that

$$S_*(v) \subseteq \mathrm{ST}_o(v). \tag{1}$$

First assume $x, y \in S_*(v)$. In this case (1) implies that: $\mathrm{ST}_{ov}(x) \neq \phi$ and $\mathrm{ST}_{ov}(y) \neq \phi$, and therefore $T_o(v)$ can be expressed as

$$T_o(v) = ex(v) + t_{ov}(x) + t_{ov}(y) + \max\{\mathrm{WS}_{ov}(x), \mathrm{WS}_{ov}(y)\} + t_m. \tag{2}$$

In addition (1) implies that $\mathrm{ST}_{v,k+1}(x) \subseteq \mathrm{ST}_{ov}(x)$ and $\mathrm{ST}_{v,k+1}(y) \subseteq \mathrm{ST}_{ov}(y)$. Thus by Lemma 1 we have:

$$t_{v,k+1}(x) \leq t_{ov}(x). \tag{3}$$

$$t_{v,k+1}(y) \leq t_{ov}(y). \tag{4}$$

Recall that $T(v) \leq T_{k+1}(v)$, and that

$$T_{k+1}(v) \leq ex(v) + t_{v,k+1}(x) + t_{v,k+1}(y) + \max\{\mathrm{WS}(x), \mathrm{WS}(y)\} + t_m. \tag{5}$$

In the case that $\mathrm{WS}(x) \geq \mathrm{WS}(y)$ using (2),(4) we have:

$$\begin{aligned} T(v) \leq T_{k+1}(v) &\leq ex(v) + t_{v,k+1}(x) + t_{v,k+1}(y) + \max\{\mathrm{WS}(x), \mathrm{WS}(y)\} + t_m \\ &= ex(v) + T(x) + t_{v,k+1}(y) \\ &\leq ex(v) + T_{ov}(x) + t_{ov}(y) \\ &\leq ex(v) + t_{ov}(x) + t_{ov}(y) + \max\{\mathrm{WS}_{ov}(x), \mathrm{WS}_{ov}(y)\} + t_m \\ &= T_o(v). \end{aligned}$$

The case $\mathrm{WS}(x) < \mathrm{WS}(y)$ is proved in the same way using (2),(3) deriving a contradiction. This completes the case $x, y \in S_*(v)$. Now observe that at least one of the nodes x, y is in $S_*(v)$. This is true as in Step (2a) both x and y are inserted to $S_*(v)$, and then at most one of them is deleted. Assume $x \in S_*(v)$ and $y \notin S_*(v)$, in this case $\mathrm{WS}(x) \geq T(y)$. This implies:

$$\begin{aligned} T(v) \leq T_{k+1}(v) &= ex(v) + t_{v,k+1}(x) + t_f + \max\{\mathrm{WS}(x), T(y)\} + t_m \\ &= ex(v) + T(x) + t_f. \end{aligned}$$

Consider $\mathrm{ST}_o(v)$, if $x \notin \mathrm{ST}_o(v)$ we get:

$$T_o(v) \geq ex(v) + T_o(x) + t_f \geq T(v),$$

and if $x \in \mathrm{ST}_o(v)$ we get:

$$\begin{aligned} T_o(v) &\geq ex(v) + t_{ov}(x) + t_f + \mathrm{WS}_{ov}(x) + t_m \\ &= ex(v) + T_{ov}(x) + t_f \\ &\geq T(v). \end{aligned}$$

This completes case 1.2.1.

Case 1.2.2: Suppose that

$$S_*(v) \not\subseteq \mathrm{ST}_o(v). \tag{6}$$

So there is a node z such that $z \in S_*(v)$ and $z \notin \mathrm{ST}_o(v)$.
Consider the ordering of $S_*(v)$

$$T(v_0 = v) > T(v_1) \geq T(v_2) \geq \cdots \geq T(v_k),$$

and consider the first node v_n (the node with the smallest index) which is in $S_*(v)$ and not in $\mathrm{ST}_o(v)$. For such v_n:

$$\mathrm{WS}_o(v) \geq T_o(v_n) \geq T(v_n) \tag{7}$$

Claim 1 $ST_n(v) \subseteq ST_o(v)$.

The proof is by contradiction. Assume that there is a node z such that $z \in \mathrm{ST}_n(v)$ and $z \notin \mathrm{ST}_o(v)$. $z \in \mathrm{ST}_n(v) \subset S_*(v)$ implies that $T(z) > T(v_n)$. Thus, v_n is not the first node in $S_*(v)$ which is not in $\mathrm{ST}_o(v)$; a contradiction.

From Claim 1 using Lemma 1 we get:

$$t_n(v) \leq t_o(v). \tag{8}$$

From equations (7),(8) we have:

$$\begin{aligned} T_o(v) &= \mathrm{WS}_o(v) + t_o(v) + t_m \\ &\geq T(v_n) + t_n(v) + t_m \\ &= \mathrm{WS}_n(v) + t_n(v) + t_m \\ &= T_n(v) \\ &\geq T(v), \end{aligned}$$

and this completes case 1.2.2.

Case 2: v has only one predecessor, y.

If y is a leaf then the only possibility is to put v alone in the grain.
Otherwise, if y is not in $\mathrm{ST}_o(v)$ then

$$\begin{aligned} T_o(y) &= T_{ov}(y) + t_f + ex(v) + t_m \\ &\geq T(y) + ex(v) \\ &= T(v), \end{aligned}$$

and if y is in $\mathrm{ST}_o(v)$ then

$$\begin{aligned} T_o(y) &= T_{ov}(y) + ex(v) \\ &\geq T(y) + ex(v) \\ &= T(v). \end{aligned}$$

This completes case 2, and the whole proof. ∎

Corollary 1 *The algorithm for trees finds an optimal grain cover, which does not encounter deadlock, in time $O(|V|^2)$.*

Proof : Notice that every grain has a unique single output. Furthermore, the operands of any grain produced by the algorithm, are always predecessors of the node computed by that grain. Therefore, since the original graph is a tree, the order relation between grains is acyclic. This implies that the grain cover cannot encounter deadlock. This proves that the grain cover can be used, and therefore Theorem 1 implies that the grain cover is optimal.

As for the time complexity, Consider a tree $G = (V, E)$ where $|V| = n$. Observe that for every node, the grains of its direct predecessors are already sorted, so merging the sets is enough and can be done in time $O(n)$. Moreover, finding d_{in} can be done (incrementally) in constant time. Lastly, at most n subtrees are considered, so that for every node the time complexity is $O(n)$, getting a total of $O(n^2)$. ∎

3.2.1 One Level Machines

For one level machines, the only parameter that changes in the model is $t(g)$. As already stated in Section 2, $t(g)$ becomes:

$$t(grain) = t_f \times \max\{\text{number of operands} + 1, \text{number of operations} + 2\}.$$

It can be easily verified that Lemma 1 and Lemma 2 hold in this model too (here we can prove $\leq$ instead of $<$ but it is enough for our needs).

The algorithm for trees needs a small modification. In Step (2a) of the algorithm the definition of $S_*(v)$ should be altered to:

$$S_*(v) = \{z \mid z \text{ is a predecessor of } v\}.$$

The case with only one predecessor, y, should be treated like the case of two predecessors, with $S_*(v)$ as in the new definition.

Theorem 2 *Let G=(V,E) be a program graph which is a binary tree to be executed on a one level machine. For every node v, T(v)— the completion time of the grain rooted at v calculated by the algorithm, modified for one level machines, — is the smallest over all possible grain covers having a grain rooted at v.*

Proof: The proof is very similar to the proof of Theorem 1 and only the changes are noted here.

The case when both x and y are leaves remains the same.

Define $a_{ov}(y)$ to be the number of nodes in $\mathrm{ST}_{ov}(y)$ and $e_{ov}(y)$ to be $d_{in}(\mathrm{ST}_{ov}(y))$. Define a_y, e_y to be the respective values for $\mathrm{ST}(y)$. If x is a leaf and y is not a leaf then $T_o(v)$ can be expressed as either

$$\begin{aligned} T_o(v) &= t_f \times \max\{e_{ov}(y) + 1 + 1, a_{ov}(y) + 1 + 2\} + \mathrm{WS}_{ov}(y) + t_m \\ &= t_f + T_{ov}(y), \end{aligned}$$

or (if y is not in the grain rooted at v then the grain has two operands and one node)

$$T_o(v) = t_f \times \max\{2 + 1, 1 + 2\} + T_{ov}(y) + t_m.$$

In both cases knowing that $T(y)$ is optimal we get:

$$\begin{aligned} T_o(v) &\geq t_f + T_{ov}(y) \\ &\geq t_f + T(y) \\ &= t_f + t_f \times \max\{e_y + 1, a_y + 2\} + \mathrm{WS}(y) + t_m \\ &= t_f \times \max\{e_y + 1 + 1, a_y + 1 + 2\} + \mathrm{WS}(y) + t_m \\ &= T_m(v) \qquad (9) \\ &\geq T(v). \end{aligned}$$

In the above equation, $T_m(v)$ is one of the possible grains for v checked in Step (2c). It is checked for $m = i_*(y) + 1$, where $i_*(y)$ is the index of the optimal grain chosen for y. This

index relation is true since $S_*(v) = S_*(y) \cup \{x, v\}$, and since x is one of the last nodes in the sorted list of $S_*(v)$.

In the case when both x and y are not leaves, Case 1 of the proof disappears due to the change in $S_*(v)$. The proof of Case 2 remains the same. The case when there is only one predecessor, y, is now included in Case 2. This completes the proof of the theorem for this model. ∎

3.3 The Algorithm for DAGs

Similar to the algorithm for trees we traverse the DAG, $G = (V, E)$, from the leaves calculating for every node v its best completion time $T(v)$ associated with an optimal grain $\mathrm{ST}(v)$. This is done by sorting all the predecessors of v, according to their best completion time, and considering for every predecessor v_i the best possible grain for v when v_i is the last argument to arrive. The final grain of v is chosen to be the one which provides the best completion time.

Computing the grain of v whenever a node v_i is considered the last argument to arrive, is done by constructing a flow graph and looking for the maximum flow and minimum cut in it. The minimum cut generated defines the grain $\mathrm{ST}_i(v)$. Both the algorithm and the constructions are illustrated with an example that follows the description.

The Algorithm:

1. For all leaves v define $\mathrm{ST}(v) \equiv \phi$, $T(v) \equiv 0$.
2. For a non leaf node v (all predecessors have been evaluated due to postorder):
   (a) Sort all predecessors of v in decreasing order of their completion times to get:
   $$T(v_1) \geq T(v_2) \geq \cdots \geq T(v_k).$$
   (b) For all $i = 1, \ldots, k$ do:
   - Define[4] :
     - $\mathrm{ST}_i^U(v) \equiv \{u \mid u \text{ is a predecessor of } v \text{ and } T(u) > T(v_i)\} \cup \{v\}$.
     - $\mathrm{SL}_i(v) \equiv \left\{ u \;\middle|\; \begin{array}{l} u \text{ is a predecessor of } v_i \text{ and every path from } u \text{ to } v \\ \text{passes through } v_i \text{ (including } v_i) \end{array} \right\}$.
     - $\mathrm{ST}_i^D(v) \equiv \{u \mid u \text{ is a predecessor of } v \text{ and } u \notin \mathrm{ST}_i^U(v) \cup \mathrm{SL}_i(v)\}$.
   - Construct the following max-flow problem : $F_i = (V_{iF}, E_{iF})$.
     The flow graph F_i is constructed from $\mathrm{ST}_i^D(v)$. Every node u of $\mathrm{ST}_i^D(v)$ is split into two nodes u^1, u^2 connected by an edge (u^1, u^2), and all original edges of $\mathrm{ST}_i^D(v)$ change direction. In addition source and target nodes, and some more

[4] Lemma 2 implies that each $\mathrm{ST}_i^U(v)$ of Step (2b) is a connected sub-dag. In this case we have the "convexity" property for $\mathrm{ST}_i^U(v)$.

edges are added as formally described below.

$$
\begin{aligned}
V_{iF} &= V^1 \cup V^2 \cup \{s,t\} \\
V^1 &= \{u^1 \mid u \in \mathrm{ST}_i^D(v)\} \\
V^2 &= \{u^2 \mid u \in \mathrm{ST}_i^D(v)\} \\
E_{iF} &= E^1 \cup E^2 \cup E^3 \cup E^4 \cup E^5 \\
E^1 &= \{(u^1,u^2) \mid u \in \mathrm{ST}_i^D(v)\} \\
E^2 &= \{(u^2,w^1) \mid u,w \in \mathrm{ST}_i^D(v) \text{ and } (w,u) \in E\} \\
E^3 &= \{(s,u^1) \mid u \in \mathrm{ST}_i^D(v) \text{ and } u \in \mathrm{DPRD}(\mathrm{ST}_i^U(v))\} \\
E^4 &= \{(u^2,t) \mid u \in \mathrm{ST}_i^D(v) \text{ is a leaf }\} \\
E^5 &= \{(u^2,t) \mid u \in \mathrm{ST}_i^D(v) \text{ is not a leaf }\}
\end{aligned}
$$

Where the capacity of edges from E^2, E^3, E^4 is unlimited, the capacity of edges from E^1 is t_f, and the capacity of edges from E^5 is $ex(u)$ where u is the original split node.

- Solve the max-flow problem in F_i, let the maximum flow value found be MaxFlow, find a saturated cut, and define :

$$
\begin{aligned}
\mathrm{ST}_i^L(v) &= \left\{ u \;\middle|\; \begin{array}{l} \text{the node } u^2 \text{ is between the source node } s \\ \text{and the saturated cut} \end{array} \right\}. \\
\mathrm{ST}_i(v) &= \mathrm{ST}_i^U(v) \cup \mathrm{ST}_i^L(v).
\end{aligned}
$$

- Calculate

$$
T_i(v) = T(v_i) + d_{in}(\mathrm{ST}_i(v)) \times t_f + t_m + \sum_{u \in \mathrm{ST}_i(v)} ex(u).
$$

(c) Choose

- $T(v) = T_{i^*}(v) = \min_{i=1}^{k}\{T_i(v)\}$.
- $\mathrm{ST}(v) = \mathrm{ST}_{i^*}(v)$.

3. Now we have a grain associated with each node. However some grains might not be needed. To get the required grains, traverse the DAG from the root to the leaves collecting the required grains. The root's grain is required. For every direct predecessor x of a required grain, the grain rooted at x is required.

Note that although we have the convexity property for $\mathrm{ST}_i^U(v)$ we do not have this property for $\mathrm{ST}(v)$, as the solution to the flow problem adds nodes to $\mathrm{ST}_i^U(v)$. The convexity property ensures that scheduling the grains would not encounter deadlock (that is, two grains cannot be scheduled as both are waiting for each others output). Here we avoid deadlock using the scheme of required grains. For each operand we have a grain computing this operand (and only this operand). This implies that the precedence relation between grains is acyclic and therefore deadlock is avoided.

Example: We use the program graph shown in Figure 4 to illustrate the algorithm. The figure gives an example of the original graph (on the left side) and the auxiliary flow graph built from it.

In this example we assume that $ex = t_m = 1$, and $t_f = 3$. Before describing the construction of the flow graph we list here the grains for each of the predecessors of v. These are: $\mathrm{ST}(z) = \{z\}$,

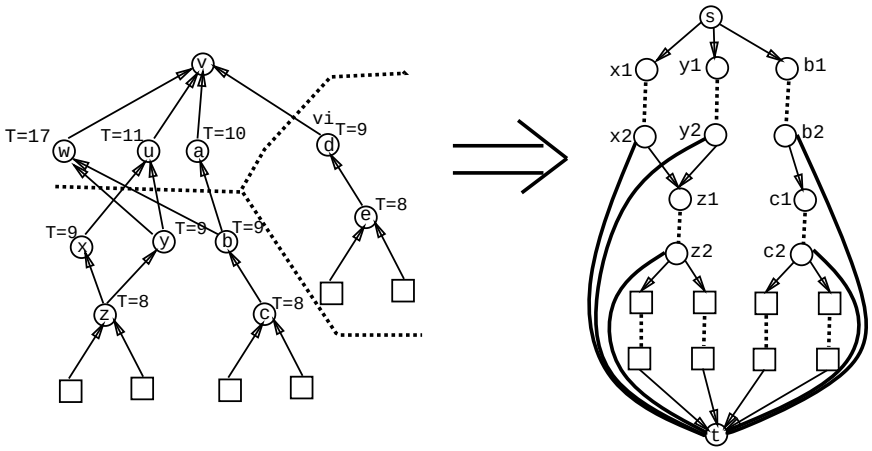


Figure 4: A DAG program graph and the auxiliary flow graph.

$\mathrm{ST}(x) = \{x, z\}$, $\mathrm{ST}(y) = \{y, z\}$, $\mathrm{ST}(u) = \{u, x, y, z\}$, $\mathrm{ST}(w) = \{w\}$, $\mathrm{ST}(c) = \{c\}$, $\mathrm{ST}(b) = \{b, c\}$, $\mathrm{ST}(a) = \{a, b, c\}$, $\mathrm{ST}(e) = \{e\}$, $\mathrm{ST}(d) = \{d, e\}$. The completion times of these grains are given on the graph, each next to its associated node.

We now describe the step of the algorithm which computes the grain for node v. The ordering of the predecessors is $T(w) > T(u) > T(a) > T(x) = T(y) = T(b) = T(d) > T(z) = T(c) = T(e)$. In this ordering, the node d is v_7. We first illustrate the computation in Step (2) for $i = 7$.

The dotted line in Figure 4 describes the partition of the graph; the part above the line is $\mathrm{ST}_i^U(v)$, the part on the right is $\mathrm{SL}_i(v)$, and the part on the bottom is $\mathrm{ST}_i^D(v)$.

The right side of Figure 4 describes the auxiliary flow graph built for $i = 7$ ($v_i = d$). The nodes in the graph were generated by splitting each of the nine nodes of $\mathrm{ST}_i^D(v)$, and by adding nodes s and t; each such node u is drawn such that u^1 is above u^2. All edges in this graph are directed from top to bottom. The edges of E^1 are dotted, and each has a capacity $t_f = 3$. The fourteen arrowed edges have capacity ∞, and they comprise the sets E^2, E^3 and E^4. Lastly, the five bold edges representing E^5 with capacity $ex(\cdot) = 1$ each.

Figure 5 shows one possibility of maximum flow on the graph. The value of the maximum flow is 8, and the horizontal line describes a minimal cut. This cut induces the optimal grain for v when $v_i = d$ was selected. The set $\mathrm{ST}_i^L(v) = \{x, y\}$, and the induced grain is $\mathrm{ST}_i(v) = \mathrm{ST}_i^U(v) \cup \{x, y\}$. With six nodes in the grain and $d_{in}(\mathrm{ST}_i(v)) = 3$ the optimal completion time is $T_i(v) = 9 + 3 \times t_f + 6 \times ex + t_m = 25$. By trying out all other possibilities one can check that this is the best possible grain for v, and that the algorithm will choose it.

The last step of the algorithm computes the required grains. The grain for v is $\{v, w, u, a, x, y\}$. Therefore the grains for z, b, d are required. Since all the operands to these grains are leaves, no other grain is required. The resulting grain cover is shown in Figure 6. Notice that the grains for w, u, a, x, y, c, e are not required.

The next Lemma proves the relation between the flow problem and the selection of the grain $\mathrm{ST}_i(v)$. The lemma also provides an alternative way to compute $T_i(v)$ directly with the value of the maximum flow.

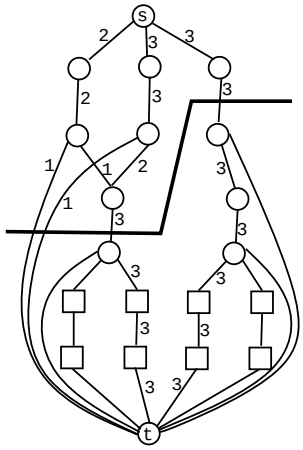


Figure 5: Maximum flow in the auxiliary flow graph.

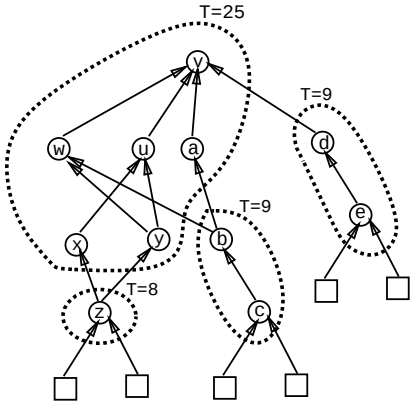


Figure 6: The final grain cover for the DAG program graph.

Lemma 3

$$\begin{aligned} T_i(v) &= T(v_i) + d_{in}(ST_i(v)) \times t_f + t_m + \sum_{u \in \mathrm{ST}_i(v)} ex(u) \\ &= T(v_i) + t_f + \text{ MaxFlow } + t_m + \sum_{u \in \mathrm{ST}_i^U(v)} ex(u). \end{aligned}$$

Proof : Consider the saturated cut. Since the capacity of edges in E^2, E^3, E^4 is infinite, the edges on a saturated cut are all in $E^1 \cup E^5$. In addition knowing the edges from set E^1 defines the other edges in the following way: All edges of the form (u^2, t) such that u^2 is between s and the cut are in the cut. All edges of the form (u^2, t) such that u^2 is between t and the cut are not in the cut. Therefore for every node $u \in \mathrm{ST}_i^L(v)$ there is exactly one edge with capacity $ex(u)$ in the cut. Now the value of the saturated minimum cut is:

$$\text{MinCut} = t_f \times \Big|\{e \mid e \text{ is an edge of type } E^1 \text{ in the minimum cut }\}\Big| + \sum_{u \in \mathrm{ST}_i^L(v)} ex(u).$$

Looking at $\mathrm{ST}_i(v)$ we get:

$$\begin{aligned} d_{in}(\mathrm{ST}_i(v)) &= 1 + d_{in}(\mathrm{ST}_i^L(v)) + \Big|\mathrm{DPRD}(\mathrm{ST}_i^U(v)) \setminus \mathrm{ST}_i^L(v) \setminus \{v_i\}\Big| \\ &= 1 + \Big|\{e \mid e \text{ is an edge of type } E^1 \text{ in the minimum cut }\}\Big|. \end{aligned}$$

Now using the max-flow min-cut theorem the rest follows. ∎

Theorem 3 *Let G=(V,E) be a DAG program graph. For every node v, $T(v)$ — the completion time of the grain rooted at v calculated by the algorithm — is the smallest over all possible grain covers having a grain rooted at v.*

Proof: The proof is by contradiction. Assume there is a grain cover in which the completion time of a grain rooted at some node y is less than $T(y)$. Let v be the first node for which there exists such a grain cover. Denote this grain cover by o, the grain of o rooted at v by $\mathrm{ST}_o(v)$, and the completion time of $\mathrm{ST}_o(v)$ by $T_o(v)$. The assumption implies $T_o(v) < T(v)$, and that for all predecessors x of v, $T(x)$ — the completion time calculated by the algorithm, is optimal.

Consider the ordering of the predecessors of v

$$T(v_1) \geq T(v_2) \geq \cdots \geq T(v_k),$$

and let z be the node which sets the value of $\mathrm{WS}(\mathrm{ST}_o(v))$. (If there is more than one such node choose any of them.) Define:

- $\mathrm{ST}_o^U(v) \equiv \{u \mid u \in \mathrm{ST}_o(v) \text{ and } T(u) > T(z)\}$.
- $\mathrm{ST}_o^L(v) \equiv \mathrm{ST}_o(v) \setminus \mathrm{ST}_o^U(v)$.

Now consider $\mathrm{ST}_i(v)$ and $T_i(v)$ when i is the index of z.

Claim 2 $\mathrm{ST}_i^U(v) = \mathrm{ST}_o^U(v)$.

Proof: It is clear that $\mathrm{ST}_o^U(v) \subseteq \mathrm{ST}_i^U(v)$. The other direction is proved by contradiction. Assume that there is a node w such that $w \in \mathrm{ST}_i^U(v)$ and $w \notin \mathrm{ST}_o^U(v)$. $w \in \mathrm{ST}_i^U(v)$ implies that $T(w) > T(v_i) = T(z)$, and this implies $w \notin \mathrm{ST}_o(v)$. Thus, there is a node w which is a predecessor of $\mathrm{ST}_o(v)$ with $T(w) > \mathrm{WS}(\mathrm{ST}_o(v))$, a contradiction.

Let $\tilde{G}$ be the DAG induced by any other cut in F_i, in the same way that $\mathrm{ST}_i^L(v)$ is induced by the minimal cut. Namely, $\tilde{G}$ includes all the nodes between s and the cut. One possible value for $\tilde{G}$, substituted in the next equation, is $\mathrm{ST}_o^L(v) \setminus \mathrm{SL}_i(v)$. With Lemma 3 and Claim 2 we get:

$$
\begin{aligned}
T(v) &\leq \\
T_i(v) &= T(z) + t_f + t_m + \mathrm{MinCut} + \sum_{u \in \mathrm{ST}_i^U(v)} ex(u) \\
&= T(z) + t_f + t_m + \sum_{u \in \mathrm{ST}_o^U(v)} ex(u) + \\
&\quad \min_{\tilde{G}} \left\{ \left|\mathrm{DPRD}(\tilde{G})\right| \times t_f + \left|\mathrm{DPRD}(\mathrm{ST}_i^U(v)) \setminus \{v_i\} \setminus \{u \mid u \in \tilde{G}\}\right| \times t_f + \sum_{u \in \tilde{G}} ex(u) \right\} \\
&\leq T(z) + t_f + t_m + \sum_{u \in \mathrm{ST}_o^U(v)} ex(u) + \sum_{u \in \mathrm{ST}_o^L(v) \setminus \mathrm{SL}_i(v)} ex(u) + \\
&\quad \left|\mathrm{DPRD}(\mathrm{ST}_o^L(v) \setminus \mathrm{SL}_i(v))\right| \times t_f + \\
&\quad \left|\mathrm{DPRD}(\mathrm{ST}_o^U(v)) \setminus \{v_i\} \setminus \{u \mid u \in \mathrm{ST}_o^L(v) \setminus \mathrm{SL}_i(v)\}\right| \times t_f \\
&\leq T(z) + t_f + t_m + \sum_{u \in \mathrm{ST}_o^U(v)} ex(u) + \sum_{u \in \mathrm{ST}_o^L(v)} ex(u) + \\
&\quad \left|\mathrm{DPRD}(\mathrm{ST}_o^L(v))\right| \times t_f + \left|\mathrm{DPRD}(\mathrm{ST}_o^U(v)) \setminus \{v_i\} \setminus \{u \mid u \in \mathrm{ST}_o^L(v)\}\right| \times t_f \\
&= T_o(v),
\end{aligned}
$$

a contradiction. ∎

Corollary 2 *The algorithm for DAGs finds an optimal grain cover, which does not encounter deadlock, in time* $O(|V|^5)$.

Proof : As in the algorithm for trees, every grain has a unique single output (that may be needed by more than one grain). Furthermore, the operands of any grain produced by the algorithm, are always predecessors of the node computed by that grain. This implies that the order relation between grains is acyclic, and therefore the grain cover cannot encounter deadlock. This proves that the grain cover can be used, and therefore Theorem 3 implies that the grain cover is optimal.

To get the time complexity, observe that for every node v we consider k max-flow problems, and that $k \leq |V|$. In addition, in Step (2b) the max-flow problem can be constructed in time $O(|V| + |E|)$ since $|V_{iF}| \leq 2 \times |V| + 2$ and $|E_{iF}| \leq 3 \times |V| + |E|$. All other parts of Step (2) of the algorithm can be completed in $O(|V| \log(|V|))$ time units. Thus the complexity of the algorithm is

$$O(|V| \times \max\{|V|(|V| + |E|), (|V| \log(|V|), |V|(\text{max-flow algorithm's complexity})\}).$$

The max-flow problem can be solved in time $O(|V_{iF}|^3) = O(|V^3|)$ (for an overview of flow problems and their solution techniques see [9, 18]) therefore the total running time of the algorithm is bounded by $O(|V|^5)$. ∎

Remarks and Generalizations:

- We can easily see that if $t_f < ex = \min_v ex(v)$ then the solution of the flow problem will not add nodes to $\mathrm{ST}_i^U(v)$ (i.e. we will always get $\mathrm{ST}_i(v) = \mathrm{ST}_i^U(v)$). So for this case we don't have to create and solve the flow problem, and the complexity of the algorithm is $O(|V|^2 \log |V|)$.
- When the program graph is a tree (and we run the DAGs algorithm), we would also get $\mathrm{ST}_i(v) = \mathrm{ST}_i^U(v)$. This is implied by Lemma 1. So we have an algorithm for general (not necessarily binary) trees with time complexity $O(|V|^2 \log |V|)$.
- Considering a model with constant fetch time, we can make the following observations: First, Lemma 1 holds in this case too. Second, if we use the algorithm for DAGs (without the flow phase) we get an optimal algorithm. This is true since the claim in the proof that $\mathrm{ST}_i^U(v) = \mathrm{ST}_o^U(v)$ still holds, and since Lemma 1 implies $\mathrm{ST}_i(v) = \mathrm{ST}_i^U(v)$. Therefore we have an optimal algorithm with time complexity $O(|V|^2 \log |V|)$ for this case too.
- The model we presented assumes fixed overhead parameters (t_f, t_m). We can generalize the model allowing each of the parameters to be edge dependent (i.e. for every edge e have overhead $c(e)$ instead of t_f or t_m). If we generalize t_f, Lemma 1 does not hold. So the results derived for trees are not valid. The results for DAGs do not use Lemma 1, and it can be verified that they hold for this generalization as well. If we generalize t_m, Lemma 2 does not hold. As a result the grains created are not continuous, and we need further work to check the implications of this issue.

3.4 Finite Number of Execution Units

When the number of execution units is limited, the problem is harder. With an unlimited number of execution units, there was no difficulty with scheduling the grains. Once a grain was ready it was immediately executed. With a limited number of execution units, a grain may become ready when there is no idle execution unit. Furthermore, when an execution unit becomes idle, there may be more than one grain waiting to be executed. Therefore, the scheduler has to decide which of the ready grains would be executed first. A wise choice might help in reducing the total running time. This is the inherent difficulty of the scheduling problem, which is NP-Complete even if one ignores overhead [7, 29]. A known heuristic is to

assign some priorities to the grains, and then choose according to these priorities. One way of assigning priorities, uses information on the critical path [5, 12]. In this section we do not provide a new decision procedure, but prove some stronger property of the grains chosen by our algorithm for trees. Namely, every legal schedule for these grains is not longer than 4 times the length of an optimal schedule for an optimal grain cover (the ratio is 3 when there are no unary operations in the tree). The result in this section generalizes the work of Graham in [11]. Using the same proof techniques we show sufficient conditions for approximating the problem for DAGs. The scheduling strategy we consider is:

1. Choose the grains assuming the number of execution units is unlimited (using the algorithm for binary trees or the algorithm for general trees according to the input graph).
2. Schedule the grains in any order that preserves the precedence constraints (and such that the execution of a grain is delayed only when all execution units are busy).

This scheduling strategy is in line with the motivation behind data flow computers. Namely, grains are scheduled dynamically at run time. The partition of the grains guarantees that any schedule is not too bad.

The execution of every grain is composed of three phases. The first is fetching the grain, the second executing the operations, and the third is matching the result. We consider scheduling a directed graph of grains on a machine with p execution units. The execution units perform the first and second phases. The third phase is carried out by another control unit (the match unit). Therefore in a diagram for a schedule (see Figure 8), we have p lines for the execution units, and infinite number of lines for the match operations.

Figure 7 shows a computation tree. The dotted lines indicate the chosen grain cover. For example node 1 will be alone in a grain, nodes 5 and 7 would be together in the same grain. Figure 8 shows a possible schedule for this grain cover on 4 execution units, assuming $t_f = t_m = ex = 1$. The numbers in the boxes (time slots) of the execution units indicate the executed grains. The number of time slots marked by a grain g is $t(g)$.
For example $t(\text{node1}) = 2 \times t_f + ex(\text{node1}) = 3$. The numbers in the boxes of the match unit lines correspond to the time spent matching the results of the grains. It is necessary for a grain to wait until its operands are matched before it can be started. For example node 6 waits for the result of node 4 to be matched, and cannot be started although there is a free execution unit before this time. Note that node 4 starts only at time point 3 as there is no free execution unit to handle it. This fact delays the completion time of this schedule. It is desired to find a schedule with minimum possible delay. We draw only 4 lines for the match unit, as only 4 match operations were required to be performed in parallel.

Let G_∞ be the grain cover calculated by the algorithm for trees, and let S_∞ be a schedule for G_∞, on p execution units. Let the length of S_∞ be w_∞. Define $W(schedule)$ to be the wasted time on the schedule which is the sum of all empty intervals on the p lines of execution units in the diagram for the schedule.

Consider S_∞, and divide the interval $(0, w_\infty]$ into two disjoint parts. The first part BUSY consists of all the time points in which all the execution units are busy. The second part NBUSY includes all other time points (at least one execution unit is idle). BUSY and NBUSY are sets of half open intervals of the form $(t_1, t_2]$; they are disjoint and their union covers $(0, w_\infty]$.

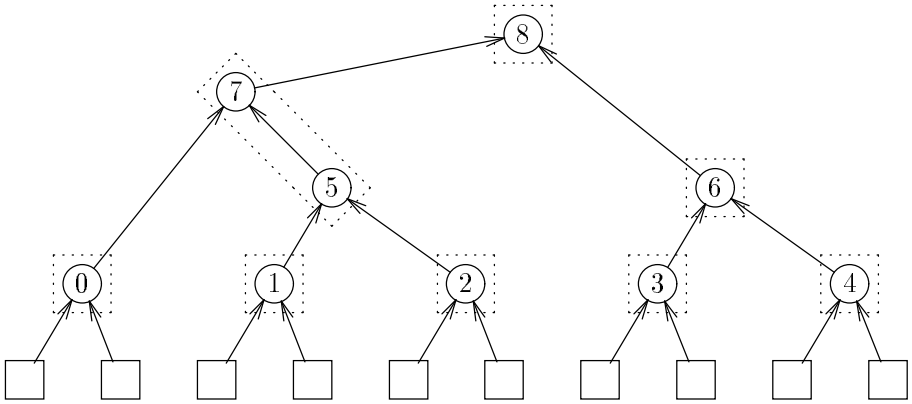


Figure 7: A computation tree and its grain cover.

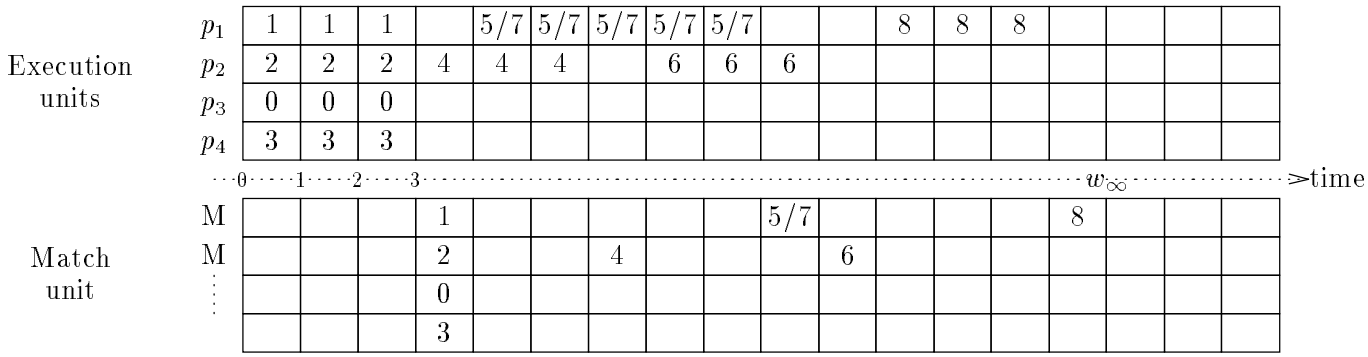


Figure 8: A schedule diagram for the computation tree.

Lemma 4 *There is a chain of grains in S_∞,*

$$g_j \rightarrow g_{(j-1)} \rightarrow g_{(j-2)} \rightarrow \ldots \rightarrow g_1$$

that covers NBUSY.

Proof: We give a procedure to find the required chain.
Consider the tree's root grain, g_1, and its start time $s(g_1)$. Find a grain g_2 which must precede g_1 in the following way:

- If $s(g_1) \in$ NBUSY and $s(g_1) \neq 0$ then there is a time point just before $s(g_1)$ in which there is an idle execution unit. The fact that g_1 was not started before $s(g_1)$ implies it waited for one of its arguments to be matched. This argument was a result of another grain g_2 which must precede g_1 in every execution order (there is a directed edge from g_2 to g_1).
- Otherwise ($s(g_1) \in$ BUSY) consider t_*, the latest time point in NBUSY which precedes $s(g_1)$. Once again (if $t_* \neq 0$) the fact that g_1 did not start at t_* implies that it waited for another grain, g_2, to be finished.

Repeat this process (using g_2 instead of g_1) until no predecessor is found as time 0 is reached. When the procedure ends we have j grains

$$g_j \rightarrow g_{(j-1)} \rightarrow g_{(j-2)} \rightarrow \ldots \rightarrow g_1$$

which precede one another even if we have unlimited number of processors. From the way the chain was built, it obviously covers NBUSY. ∎

Consider G_o, any other grain cover, and a schedule S_o minimizing its completion time w_o. The following corollary gives a lower bound on its length.

Corollary 3 $w_o \geq \frac{W(S_\infty)}{p}$.

Proof: From Lemma 4 we know that all the grains in the chain must precede one another. This is true when there are enough execution units too. Let t_{ch} denote the minimal time for execution of the chain. Then,

$$t_{ch} = \sum_{i=1}^{j} (t(g_i) + t_m) \leq T(G). \tag{10}$$

Where in the above equation $T(G) = T(g_1)$ is the minimal completion time of the graph when the number of execution units is unlimited.

Once again from Lemma 4 we know that the chain covers NBUSY, and as in every time point at most p execution units are idle, we get:

$$W(S_\infty) \leq p \times t_{ch}. \tag{11}$$

Now consider the grain cover G_o. Theorem 1 (Theorem 3 for general trees) implies that the schedule S_o cannot be finished before $T(g_1)$ even when the number of execution units is not limited. Therefore using equations (10),(11) we get:

$$w_o \geq T(g_1) \geq t_{ch} \geq \frac{W(S_\infty)}{p}. \tag{12}$$

∎

Let $\tilde{G}$ be a grain cover. We define its volume as the the time spent by the execution units to execute this grain cover. Note that for any schedule for this grain cover, the volume is the complement of the waste W(schedule) for this schedule.

$$\mathrm{VOL}(\tilde{G}) \equiv \sum_{g \in \tilde{G}} t(g).$$

In the following we will bound the volume of the grain cover calculated by our algorithm, and then use this bound to bound the schedule length.

Lemma 5 *Let G_∞, G_o be as stated above.*

1. *For trees without unary operations:* $\mathrm{VOL}(G_\infty) < 2\mathrm{VOL}(G_o)$.
2. *For trees with unary operations:* $\mathrm{VOL}(G_\infty) < 3\mathrm{VOL}(G_o)$.

Proof: Consider the grain covers G_∞, G_o, and define:

$$G_c \equiv \{g = g_1 \cap g_2 \mid g_1 \in G_\infty \text{ and } g_2 \in G_o\}.$$

The fact that G_c is a finer partition than G_∞ implies that

$$\text{VOL}(G_\infty) \leq \text{VOL}(G_c). \tag{13}$$

This is due to the fact that both covers have the same operation nodes, and the finer cover has more edges contributing to the values of $t(\text{grains})$.

First consider trees with no unary operations. Let g^* be a grain in G_o, and consider $\{g_i^*\}$, the grains in G_c which cover g^*. Let K be the number of internal edges of g^* that contribute to $\text{VOL}(G_c)$. Let t_c^* be the time spent in G_c for executing all the parts of the grain g^*. Then,

$$\begin{aligned} t_c^* &= \sum_i t(g_i^*) \\ &= d_{in}(g^*) \times t_f + K \times t_f + \sum_{v \in g^*} ex(v). \end{aligned}$$

The fact the the tree has no unary operations implies that

$$K < d_{in}(g^*),$$

and as

$$t(g^*) = d_{in}(g^*) \times t_f + \sum_{v \in g^*} ex(v),$$

we get:

$$\begin{aligned} t_c^* &< 2t_f \times d_{in}(g^*) + \sum_{v \in g^*} ex(v) \\ &< 2t(g^*). \end{aligned}$$

Now we can bound the volume:

$$\begin{aligned} \text{VOL}(G_c) &= \sum_{g \in G_c} t(g) \\ &= \sum_{g^* \in G_o} \sum_i t(g_i^*) \\ &< \sum_{g^* \in G_o} 2t(g^*) \\ &= 2\text{VOL}(G_o). \end{aligned} \tag{14}$$

Now consider trees with unary operations. Let K_1 be the number of internal edges in g^* that do not correspond to unary operations and that contribute to $\text{VOL}(G_c)$, and let K_2 be the number of internal edges in g^* that do correspond to unary operations and contribute to $\text{VOL}(G_c)$. Let t_c^* be as before. Then,

$$\begin{aligned} t_c^* &= \sum_i t(g_i^*) \\ &= d_{in}(g^*) \times t_f + K_1 \times t_f + K_2 \times t_f + \sum_{v \in g^*} ex(v). \end{aligned} \tag{15}$$

Consider the grain g^*. Every operation in the grain consumes some operands. Every operation in the grain but the grain's root creates a result which is used as an operand in some other operation in the grain, and might be one of the edges (retrieving the operand) we are counting in K_1. K_1 is maximized when all the input operands of the grain belong to unary operations and all non unary operations are binary (This is true since the number of edges in the tree is maximum in the binary case, when the number of leaves is fixed). In this case we have:

$$K_1 \leq 2d_{in}(g^*) - 2.$$

Considering unary operations, observing the algorithm for trees, we can see that the grain for an unary node contains the node and the whole grain of its predecessor. Therefore an unary chain (a path of unary operations) is not split by the algorithm, that is unless one of its nodes is used as an argument to another node's grain. In this case using the procedure for choosing only the required grains assures that every unary chain is split at most once.

Next we bound $K_1 + K_2$ using the bound on K_1. Consider an unary chain, C, whose root is an operand to a non unary operation, v_C, connected by the edge, e_C. If the chain is not split by G_∞, it does not contribute to $K_1 + K_2$. If the chain is split then v_C, e_C, and the root of the chain belongs to the same grain in G_∞. This means that e_C should not be counted in K_1, and since the chain contributes 1 to $K_1 + K_2$ the bound is not changed.

There can be only one chain whose root is not an operand to a non unary operation in g^*. This chain's root is the root of g^*. This chain contributes at most 1 to $K_1 + K_2$ (in the case that the chain is split). Therefore, all unary chains increase the bound by at most 1, implying:

$$K_1 + K_2 \leq (2d_{in}(g^*) - 2) + 1 < 2d_{in}(g^*). \tag{16}$$

Combining equations (15),(16) we get:

$$\begin{aligned} t_c^* &< 3t_f \times d_{in}(g^*) + \sum_{v \in g^*} ex(v) \\ &< 3t(g^*), \end{aligned} \tag{17}$$

and in the same way as in equation (14) we get

$$\mathrm{VOL}(G_c) < 3\mathrm{VOL}(G_o). \tag{18}$$

Now combining equations (13) and (14) we get the first part of the lemma, and from (13) and (18) we get the second part. ∎

Theorem 4 *Let $G_\infty, G_o, S_\infty, S_o, w_\infty, w_o$ be as stated above.*

1. *For trees without unary operations: $w_\infty < 3w_o$.*
2. *For trees with unary operations: $w_\infty < 4w_o$.*

Proof: Observing the definitions for VOL(grain cover) and W(schedule) we can get the following relations:

$$w_\infty = \frac{W(S_\infty) + \mathrm{VOL}(G_\infty)}{p}, \tag{19}$$

and

$$w_o \geq \frac{\mathrm{VOL}(G_o)}{p}. \tag{20}$$

Now using Corollary 3 and equations (19),(20) we get that if:

$$\mathrm{VOL}(G_\infty) < k \times \mathrm{VOL}(G_o) \tag{21}$$

then

$$\begin{aligned} w_\infty &= \frac{W(S_\infty) + \mathrm{VOL}(G_\infty)}{p} \\ &< \frac{W(S_\infty) + k \times \mathrm{VOL}(G_o)}{p} \\ &\leq \frac{p \times w_o + k \times p \times w_o}{p} \\ &= w_o \times (k+1). \end{aligned} \tag{22}$$

Combining equation (22) with Lemma 5 completes the proof. ∎

A better bound can be derived when comparing against single instruction grains. Using the finer partition argument, as in Lemma 5, we can get $k = 1$. Therefore the algorithm's grains schedule length is not longer than twice the length of an optimal schedule for single instruction grains. On the other hand there are tree programs for which the single instruction grains perform poorly (with the right choice for parameters). For DAGs, single instruction grains might perform poorly even for fixed parameters ($ex = t_f = t_m = 1$). For example consider a graph with $n+1$ nodes named $0, \ldots, n$. Node 0 is a leaf, and for all i, j such that $i < j$ the edge (i, j) is in the graph (this graph is an unary chain enhanced with all possible forward edges). The schedule length for single instruction grains is: $T_M = \sum_{i=1}^{n}(ex + i \cdot t_f + t_m) = O(n^2)$. On the other hand our algorithm would find that the optimal grain cover consists of a single grain containing the whole graph. The schedule length of this grain cover is: $T_\infty = t_f + n \cdot ex + t_m = O(n)$, and clearly the single instruction grains perform poorly in this case. The following theorem generalizes the result of Theorem 4, and provides sufficient condition for approximating the problem for DAGs.

Theorem 5 *Let A be an approximation algorithm with ratio* `a` *for the scheduling problem stated in Section 2, when the number of execution units is not limited, and the volume of the grains generated by A is bounded with ratio* `b` *relative to every other grain cover. Then A is an approximation algorithm with ratio* `a+b` *for the scheduling problem, when the number of execution units is limited.*

Proof: Denote the grain cover A generates, a schedule for this grain cover, and its completion time by G_A, S_A, w_A respectively. Observe that similar to Lemma 4, there is a necessary chain

that covers NBUSY for S_A. Using this observation, equation (12) in Corollary 3 becomes:

$$w_o \geq \frac{1}{a} T(g_1) \geq \frac{1}{a} t_{ch} \geq \frac{W(S_A)}{a \cdot p}. \tag{23}$$

The conditions in the theorem imply:

$$\mathrm{VOL}(G_A) \leq b \cdot \mathrm{VOL}(G_o), \tag{24}$$

and combining these equations we get:

$$\begin{aligned} w_A &= \frac{W(S_A) + \mathrm{VOL}(G_A)}{p} \\ &\leq \frac{W(S_A) + b \cdot \mathrm{VOL}(G_o)}{p} \\ &\leq \frac{a \cdot p \cdot w_o + b \cdot p \cdot w_o}{p} \\ &= (a+b) \cdot w_o. \end{aligned} \tag{25}$$

∎

Theorem 5 provides sufficient conditions for approximating the general problem. A natural question is whether the algorithm for DAGs from Section 3.3 complies with these conditions. First observe that as the algorithm finds an optimal grain cover when the number of execution units is not limited, we have $a = 1$ for the first condition. However this algorithm does not have a constant volume bound. In particular, we have an example for which the volume of the grains generated by the algorithm is $\log(n)$ times the volume of a grain cover with single instruction grains. The large volume in this example occurs because of re-computation and does not rely on bad parameters. We also show that scheduling these grains yields completion time that is $\log(n)$ times the completion time of single instruction grains.

Example: Schedules for the DAG from Figure 9.

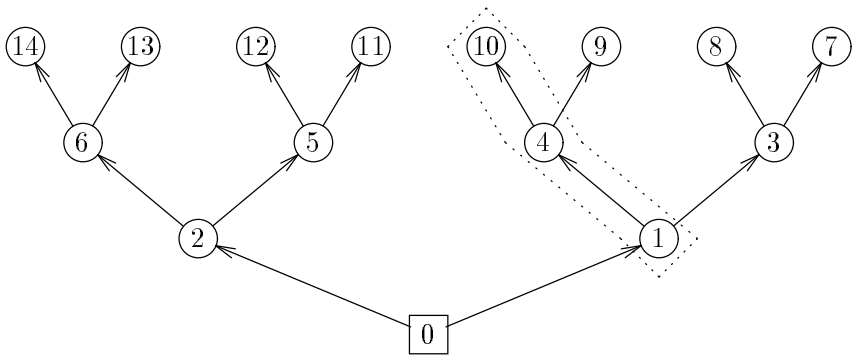


Figure 9: A DAG whose grain cover has a large volume.

For each output node v, the grain that the algorithm generates for v contains all the nodes on the path from the single leaf (the node numbered 0) in the DAG to v. For example the grain for node 10 would include the nodes $1, 4, 10$. If we generalize this graph to $\log(n)$ levels with n nodes, there would be $n/2$ grains each containing $\log(n)$ nodes. The volume of this grain cover

would be $O(n\log(n))$. On the other hand, the volume of a grain cover with single instruction grains is $O(n)$. So for this example the ratio between the volumes of the grain covers is $\log(n)$, and this implies that if there is a volume bound for our DAG algorithm, it is at least $\log(n)$. It is still open whether $\log(n)$ is also an upper bound (which would yield an approximation with ratio $\log(n)$) for the general problem.

Now consider scheduling these grains on p processors, and assume that $t_f = 1, ex = 1, t_m = 0$. The completion time for the grains from our algorithm would be:

$$T_\infty^p = O((1+\log(n)) \times \frac{n}{2p}) = O(\frac{n\log(n)}{p}).$$

The completion time for a grain cover with single instruction grains would be:

$$T_{\min}^p = 2\log(p) + 2\frac{n-2p+1}{p} = O(\frac{n}{p} + \log(p)),$$

and we have a $\log n$ ratio on the completion time. This implies that the algorithm does yield bad schedules sometimes. An alternative approach would be to find another algorithm that does comply with the conditions of Theorem 5. Namely, it would only approximate the best possible time, but on the other hand it would have a small bound on the volume.

4 Scheduling Models

The problem discussed in this paper is a scheduling problem that involves some parameters of overhead. In this section we describe its relation to other scheduling problems that have been studied before.

Our formalized problem is a generalization of the *precedence constrained scheduling* (PCS) problem [7] which was proved to be NP-complete in [29] (the PCS problem does not consider overhead). When the precedence graph is a tree, there is a simple algorithm to solve the PCS problem [13]. For general precedence constraints, when the number of processors is 2, there is a polynomial algorithm [5]. A crucial observation by Graham [11] is that any greedy schedule (i.e. if there is a free execution unit and a ready instruction — schedule it) is an approximation algorithm for the PCS problem with ratio 2. We generalized this result to get approximation algorithms for our problem.

We classify scheduling problems that consider overhead using the following terminology. We say that a computer has *the waiting property* if a processor cannot start working on a grain (fetch operands and execute instructions whose operands are already available) before all its operands are available. This property is relevant only for models that pack instructions into grains in order to save some overhead. We say that the computer has *parallel communication* if a value that has been computed is available to all processors after some delay (this hides the assumptions that a broadcast can be made within that delay, and that a processor can get values from different sources in parallel). We say that communication is *overlapped* with computation, if an execution unit can do other things while making the communication. Our model for data flow machines has the waiting property and two kinds of communication, one parallel which is overlapped, and the other not parallel (serial). The serial communication is not overlapped in the main model, and is overlapped in the model for one level machines. We

also note that (using the remark about constant fetch time in the end of Section 3.3) the serial communication can be replaced with parallel non-overlapped communication.

Aggarwal, Chandra and Snir [1] considered a model of PRAM in which, in addition to the shared memory, each execution unit has a local memory. Access to the local memory does not cost time. In this model communication is not parallel (data have to be serially read from the shared memory), and the waiting property does not hold. They gave an algorithm to find a schedule within a factor of two from the optimal schedule for tree-like programs. The waiting property is very important in selecting the grains, and therefore an algorithm for one model cannot be used for the other. As an example consider a binary chain program graph. The optimal solution of [1] is composed of $\sqrt{n}$ grains in sizes increasing from 1 to $\sqrt{n}$. This schedule requires $\sqrt{n}$ computation steps. For our model the optimal solution is composed of one grain.

Papadimitriou and Yannakakis [19], Anger, Hwang and Chow [3], and Zhu [31] considered a model with parallel communication and where the waiting property does not hold. For this model the problem remains NP-complete even when the number of processors is not limited [19] and the input graph is a tree [31]. When communication time is short (shorter then the execution time of every program node) the problem is solvable [3]. Approximation algorithms (with ratio 3 and 2) for partitioning DAG programs were found for the case with unlimited number of processors [3, 19]. As before optimal solutions for this model are not optimal in our model. However, our algorithm for trees seems somewhat similar in concept to the "generalization" algorithm of [19].

Kruatrachue and Lewis [17] consider a scheduling model in which communication is parallel (non overlapped) and the waiting property holds. They use heuristics to solve the problem. Anderson, Beame and Ruzzo [2] consider a similar model, in which the computer has no communication overhead but scheduling a cluster costs some time. This scheduling latency can be replaced with parallel communication latency (non overlapped), and the waiting property is valid. They derive bounds on the overhead for a specific program graph (the Diamond DAG). This model is a special case of our model (when using the constant fetch time variant) so our results can be used.

Gottlieb and Schwartz [10] suggested simulating a data flow language on a shared memory machine. In this simulation every processor repeatedly executes the following steps: fetch a ready grain, execute it, write and match the result. This simulation fits a scheduling model where the waiting property holds, and both serial (input) and parallel (output) communication exist. The communication in this model is not overlapped. For the case with unlimited number of processors, this model is identical to our model, and our results can be used to get the right grains for the simulation. The same kind of simulation can be done on the Linda machine [16]. In this case the matching is handled by the memory, and therefore the parallel communication can be overlapped. This simulation exactly fits our model.

Sarkar [24] considered mapping general programs onto uniform computers. His model (named macro data flow) has the waiting property, and a large number of parameters. He used heuristics to solve this problem. Our model is a special case of this model, and for this special case we give optimal (and approximation) results.

5 Data Flow Computers

The implementation of data flow computers is quite complicated. Although the basic idea is simple, questions like implementation of the match units, execution of loops and recursions, the communication of tokens in the machine, and the strength of the execution units do not have a distinct "right" solution. Moreover, some data flow machines include special execution units that handle data structures more efficiently. Various machines that fall in the category of data flow computers have been built and simulated. Comparative reviews of data flow machines are given in [25, 26, 30]. In the following we discuss how our results can be related to existing data flow machines.

Machine Architecture

The first issue that affects the applicability of our results is the communication structure of the machine. Our models fits some of the possible architectures as described below.

We use terminology from [30] to describe the communication structure of data flow machines. *Two stage* machines have a set of execution units (EUs) and a set of match units (MUs), and use a communication networks to send ready grains from a MU to an EU, and another network to send results from EUs to MUs. This is exactly the kind of machine described by our model. Dennis and Misunas describe a machine of this type [6].

In *One level* machines each EU has a dedicated MU, and its operation can be overlapped (in a pipe fashion). There is one communication network that is used for sending the results between the EU/MU pairs. The model for one level machines we described captures this structure by changing the "fetch and execute" parameter $t(g)$ to fit a pipeline structure. Examples of machines with this structure are the EM-4 [23] and the MIT TTDA architecture (see description in [20]).

In *Two level* machines the EUs are partitioned to sets. Each set of EUs has a dedicated MU. There is a communication network to send results from EUs to MUs, and a communication network to send a ready grain from an MU to one of its EUs. These machines might have a bypass from an EU to its MU which means some of the results do not have to go through the communication network and need only be matched. The Manchester data flow machine has this structure. Our model fits these machines if the bypass does not exist, and fits a subset of the machine (one MU and its EUs) if the bypass exists. It does not model the whole machine with the bypass as this structure has two different possible values for the writing time (t_m in the model).

In some machines the time taken to implement the match operation is not fixed. This may result from techniques like hashing, or simply from the fact that the capacity of the match unit is too low and therefore results wait in a queue before being matched. In such cases a typical or average time for matching should be used with our algorithms.

The last architectural issue is the requirement that execution units should be able to handle grains instead of single instructions. While we do not supply a thorough analysis of this requirement, we expect that a simple extension should be sufficient. That is, the normal data path of an execution unit should be enhanced with a register file and appropriate multiplexers

to choose the input and output operands. This has been implemented in the EM-4 [23].

Data Flow Programs

Our algorithms can handle partitioning of programs which have no cycles. The main implementation question is how to apply the algorithms to whole programs that might have cycles. We next discuss various possible applications.

There are two approaches to handling cycles in data flow machines. One uses a different "tag" for every invocation of an instruction, and the other simply creates another copy of the instruction. In either case, the grains existing during a run of a program must have an acyclic relation in order to prevent deadlock. If we could predict the dynamic behavior of the program we could use our algorithm for the partitioning of the whole program. As this is not possible we can use standard compilation techniques to handle the problem. One possibility is to partition basic blocks or the loop body to grains and use it for all iterations. Another possibility, used in VLIW an other machines, is to use loop unfolding, and find the optimal grain structure in the repeated loop context. The last technique is similar to software pipelining which is used for parallelizing a loop and involves the examination of a few iterations that reveal repeating behavior (in the data dependencies). Similarly, when compiling non-strict languages, one can use our algorithm to re-partition large basic blocks extracted by the compiler (see discussion below).

Other applications can handle only part of a program or part of the machine. Some data flow machine implementations, (when there is more than one MU and a bypass for local results) make a pre-partition of the program and schedule a whole basic block to the same EU. If this is done on a two level machine, then our algorithms can be used for the partition of basic blocks and improve the running time, locally, on each EU.

Another application of our algorithms can be done in creating specialized libraries of routines, which are tailored to the architecture. Several important algorithms are now well understood, and the data dependencies involved in them are all explicit [19]. One such example is the FFT butterfly graph. While it can be written recursively (complicating the compilation process) it can also be written with explicit data dependencies. Our algorithms can be applied to such graphs, and the grain partition can be stored in a library which is specialized to the architecture.

Related Work

The basic idea behind the problem discussed in this paper is to compensate for the delays incurred in a real data flow computer. This means taking advantage of locality when possible but allowing maximum parallelism when possible. Some works in this direction have been done before within the framework of dynamically scheduled instructions. One direction was to use "larger" instructions and in that way make the overhead smaller. Böhm and Gurd [4] considered using iterative instructions as a tool for reducing communication overhead for a data flow machine. They show, through simulations, that the performance is increased in this case. Gaudiot and Ercegovac [8] studied the optimal resolution of actors (similar to grain sizes) for data flow programs. They analyze the problem, and an optimal mean resolution size

is found. However an arbitrary algorithm for dividing the program, into grains of this size, is used.

A similar approach has been taken by several compilers for non-strict languages [14, 27, 28, 21]. In these languages, in general, data dependencies cannot be determined during compilation time. The main difficulty is therefore to avoid deadlock which may result if the grains are not carefully chosen. In early works, the assumption that "larger grains are always better" has been taken. (Although caution not to hide parallelism has balanced this[5].) This has been contested in a recent experimental evaluation [21]. In this work two different partitioning algorithms are compared against the single instruction execution. While the grains produced by these algorithms do better than single instructions execution, it is shown that machine parameters do have an effect on the quality of these partitions. It is concluded that machine parameters should be consulted during the compilation. In contrast with these works, our approach does take machine parameters into consideration. Using these parameters, an optimal partition for the actual machine is found. In fact, our algorithms fit very well as post-processors for the partitioning strategies devised in these works. The first partitioning pass should divide the program into as large grains as possible, using any possible technique, and without worrying about hiding parallelism. Each such grain can then be considered a basic block, and our algorithms can decide how it should be decomposed into smaller grains, therefore exposing just the right amount of parallelism that fits the actual machine.

Several other works that try to incorporate benefits of the Von Neumann architecture and data flow computers have been reported. These architectures are sometimes called "hybrid" architectures. The approach our model describes is that of enhancing the EU with some registers that enable serial execution of a block of code (a grain). Of particular interest is the EM-4 [23], a data flow machine allowing the scheduling and execution of grains. The EM-4 has a one level structure, and some of our results are directly applicable to it.

On the other hand, various other techniques which are not captured by our model have also been implemented; we briefly describe some of them here. The technique of "pipeline optimization" in the EM-4 [22] tries to save time by trying to schedule non-ready grains if the execution unit is idle. If the last missing operand arrives in the next cycle, this pre-fetching technique saves time. This does not comply with our strict model of scheduling only ready grains. Iannucci [14] describes another hybrid architecture where Von Neumann processors are enhanced with a fast context switch mechanism for switching between grains. The processors are connected through some communication network. Unlike our model, the running of a grain can be suspended and rescheduled with almost zero overhead due to the fast context switching. Papadopoulos [20] describes a one level data flow machine in which the EU is enhanced with temporary registers. These registers retain their values beyond grain boundaries, and careful static scheduling of instructions is performed by the compiler in order to use the temporary registers and save time. Again, our model cannot be used with this machine since we only consider retaining information within grain boundaries.

In conclusion of this section, we note again that there is a variety of data flow architectures. Our model does capture some of them, and our algorithms can be incorporated with existing compiling techniques in order to achieve better performance.

[5] In [21] an average grain size of 4, with a large variance, is reported. However, it is mentioned that no loop unrolling or other similar technique has been applied. On the other hand, grain size of 31 has been used in some of the experiments in [8].

6 Conclusions

In this paper we introduced a scheduling model, inspired by data flow computers, that includes some overhead parameters. We considered the problem of partitioning programs into grains such that some of the overhead can be saved, and the completion time of a program would be minimized. For acyclic programs, we presented optimal algorithms for the case where there are enough execution units to get all the parallelism in the program. For the case with a limited number of execution units, we gave an approximation algorithm with an approximation ratio of 4 when the program graph is a tree. We also gave sufficient conditions for approximating the problem for DAGs.

The overhead structure captured in our model is not constrained only to the domain of data flow computers, and our results are valid for some models that have been studied before. We further introduced some terminology, for scheduling models with overhead, that is useful for relating models and results to one another.

There is a variety of implementations for data flow machines, and although our model falls short of modeling all of them, the results of this paper can be applied to some. In particular, some of the results hold for the EM-4, which is a data flow machine with execution units that can handle grains. For the Manchester data flow computer the results can be used to enhance performance locally on each processing element. Furthermore, the model can be used when simulating a data flow program on a shared memory machine or on a Linda machine. The algorithms can be used as a second stage in compilation, which partitions large acyclic program fragments. These large basic blocks can be found using standard compiling techniques like loop unrolling and in-line expansion of procedures, as well as specialized technique for data flow languages. The algorithms can also be used to compile special versions of several important algorithms. This can be applied for example to create a specialized version of the FFT algorithm for a particular architecture.

A few problems remain to be investigated. The first issue concerns approximating the problem with limited number of execution units for DAGs. This can be done if we find a volume bound for the algorithm for DAGs, or find another algorithm that satisfies the conditions of Theorem 5. Another problem is to resolve the complexity issue of finding the best grain cover and schedule for trees for limited number of execution units (we gave an approximation algorithm but did not show that it is NP-complete). For practical reasons we would like to bound the size of the grains. Such a generalization is simple for trees. The case for DAGs is more complicated.

Acknowledgments. We would like to thank Efim Dinic for a fruitful discussion resolving the flow problem and Larry Rudolph for insightful comments. We would also like to thank the anonymous referees for their comments. Roni Khardon would like to thank the generous support from the Miriam and Aaron Gutwirth graduate fellowship. An earlier version of this paper appeared in the International Conference on Parallel Processing (ICPP) 1991 [15].

References

[1] A. Aggarwal, A.K.Chandra, and M.Snir. Communication complexity of PRAMs. *Theo-*

retical Computer Science, 71:3–28, 1991.

[2] R. Anderson, P. Beame, and W. Ruzzo. Low overhead parallel schedules for task graphs. In 2^{nd} *Annual ACM Symp. on Parallel Algorithms and Architectures*, pages 66–75, July 1990.

[3] F. Anger, J. Hwang, and Y. Chow. Scheduling with sufficient loosely coupled processors. *Journal of Parallel and Distributed Computing*, (9), 1990.

[4] A.P. Böhm and J.R.Gurd. Iterative instructions in the Manchester dataflow computer. *IEEE Trans. on Parallel and Distributed Systems*, 1(2), April 1990.

[5] E.G. Coffman. *Computer and Job Shop Scheduling Theory*. john wiley and sons, 1976.

[6] J.B. Dennis and D.P.Misunas. A preliminary architecture for a basic data flow processor. In 2^{nd} *annual Symposium on Computer Architecture*, pages 126–132, 1974.

[7] M.R. Garey and D.S.Johnson. *Computers and Intractability: a Guide to the Theory of NP-completeness*. w.h.freeman and company, 1979.

[8] J.L. Gaudiot and M.D.Ercegovac. Performance evaluation of a simulated data flow computer with low resolution actors. *Journal of Parallel and Distributed Computing*, 2(4):321–351, 1985.

[9] A.V. Goldberg and D.Gusfield. Book review: Flow algorithms by E.A.Dinic and A.V.Karzanov. *Tech.Rep.STAN-CS-90-1313*, June 1990.

[10] A. Gottlieb and J.T. Schwartz. Networks and algorithms for very-large-scale parallel computation. *Computer*, (1):27–36, January 1982.

[11] R.L. Graham. Bounds on multiprocessing timing anomalies. *Siam J. of Applied Math*, 17(2):416–429, March 1969.

[12] M. Granski, I. Koren, and G.M. Silberman. The effect of operation scheduling on the performance of a data flow computer. *IEEE Transactions on Computers*, 36(9):1019–1029, 1987.

[13] T.C. Hu. Parallel sequencing and assembly line problems. *Operations Res.*, 9:841–848, 1961.

[14] R.A. Iannucci. Towards a data flow Von Neumann hybrid architecture. In *International Symposium on Computer Architecture*, pages 131–140, 1988.

[15] R. Khardon and S. S. Pinter. Choosing the right grains for data flow machines. In *International Conference on Parallel Processing*, pages I672–I673, 1991.

[16] V. Krishnaswamy, S. Ahuja, N. Carriero, and D. Gelernter. The architecture of a linda coprocessor. In *International Symposium on Computer Architecture*, pages 240–249, 1988.

[17] B. Kruatrachue and T. Lewis. Grain size determination for parallel processing. *IEEE Software*, pages 23–32, Jan 1988.

[18] C.H. Papadimitriou and K.Steiglitz. *Combinatorial Optimization*. Prentice Hall, 1982.

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

[20] G.M. Papadopoulos. *Implementation of a general purpose dataflow multiprocessor*. The MIT Press, 1991.

[21] L. Roh, W.A. Najjar, and A.P. Böhm. Generation and quantitative evaluation of dataflow clusters. In *Conference on functional programming languages and computer architecture*, pages 159–167, 1993.

[22] S. Sakai, K. Hiraki, Y. Yamaguchi, Y. Kodama, and T. Yuba. Pipeline optimization of a data flow machine. In *Advanced topics in data flow computing*, pages 225–246. Prentice Hall, 1991.

[23] S. Sakai, Y. Yamaguchi, k. Hiraki, Y. Kodama, and T. Yuba. An architecture of a dataflow single chip processor. In *International Symposium on Computer Architecture*, pages 46–53, 1989.

[24] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*. The MIT Press, 1989.

[25] J.A. Sharp. *Data Flow Computing*. Ellis Horwood Limited, 1985.

[26] V.P. Srini. An architectural comparison of dataflow systems. *Computer*, 19(9):68–88, March 1986.

[27] K.R. Traub. Multi thread code generation for dataflow architectures from non-strict programs. In *Conference on functional programming languages and computer architecture*, pages 73–101, 1991.

[28] K.R. Traub, D.E. Culler, and K.E. Schauser. Global analysis for partitioning non-strict programs into sequential threads. In *Conference on Lisp and functional programming*, pages 324–334, 1992.

[29] J.D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, 1975.

[30] A.H. Veen. Dataflow machine architecture. *ACM Computing Surveys*, 18(4):365–396, December 1986.

[31] Y. Zhu. Workload scheduling: A new technique for scheduling task graphs with communication costs in parallel systems. Technical Report CS-TR-90-59, Dept. of Computer Science and Operations Research, North Dakota State University, 1990. A short version of this paper appeared in ICPP-91.