Upper Time Bounds for Executing PRAM-Programs on the LogP-Machine

Welf Löwe and Wolf Zimmermann
Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe
76128 Karlsruhe
Germany.
E-mail: {loewe|zimmer}@ipd.info.uni-karlsruhe.de

Abstract

In sequential computing the step from programming in machine code to programming in machine independent high level languages has been done for decades. Although high level programming languages are available for parallel machines today's parallel programs highly depend on the architectures they are intended to run on. Designing efficient parallel programs is a difficult task that can be performed by specialists only. Porting those programs to other parallel architectures is nearly impossible without a considerable loss of performance. Abstract machine models for parallel computing like the PRAM-model are accepted by theoreticians but have no practical relevance since these models don't take into account properties of existing architectures. However, the PRAM is easy to program.

Recently, Culler et al. defined the LogP machine model which better reflects the behaviour of massively parallel computers. In this work, we show transformations of a subclass of PRAM-programs leading to efficient LogP programs and give upper bounds for executing them on the LogP machine. Therefore, we first briefly summarize the transformations from PRAM to LogP programs. Second, we extend the LogP machine model by a set of machine instructions. Third, we define the classes of coarse and fine grained LogP programs. The former class of programs can be executed within the factor two of the optimum. The latter class of programs has an upper time bound for execution that is a little worse. Finally, we show how to decide statically which strategy is promising for a given program optimization problem.
Keywords: PRAM, LogP machine, Program Transformation, Optimization, Scheduling, Time Bounds.

1 Introduction

The PRAM-model consists of a shared memory and a number of processors with local memory. Processors only communicate via their shared memory. The computation steps are performed in a synchronous lock-step manner. Memory access to different memory locations can be performed at the same time. The PRAMs are distinguished by their ability to access in parallel the same memory location. In our paper we exclude concurrent writes. Most parallel algorithms are designed for flavors of the PRAM-models. For an overview of PRAMs and PRAM-algorithms, see [KR90]. The model has been successfully applied, because it allows to focus on the potential parallelism of the problem at hand. In particular, there is no need to consider a network topology and a memory distribution. For these reasons the model is often chosen to design parallel algorithms and programs.

On the other hand, almost all parallel computers and local area networks are distributed memory architectures. As shown in [ZK93] implementations of the PRAM-model on existing parallel machines are practically expensive, although theoretically optimal results exist [Val90]. The reason is the expensive synchronization, communication latency, communication overhead, and network bandwith. In the LogP machine [CKP+93], these communication costs are taken into account. The number of processors is constant w.r.t. the problem size, and the synchronization must be programmed explicitly. The architecture dependent parameters of the LogP machine are

(1) *the communication latency* L. It is guaranteed that a message (containing only a small number of words) arrives at its destination within this time. Observe that L is an upper bound on all source-destination pairs.

(2) *the communication overhead* o. This is the time required by a processor to send or receive a message. It is assumed that a processor cannot perform operations while sending or receiving a message.

(3) *the gap* g. This is the reciprocal of the communication bandwith per processor. It means that when a processor sends (or receives) a message, the next message can be sent or received after time g, but not necessarily before.

(4) *the number of processors* P.

These parameters have been determined for the CM-5 in [CKP+93] and for the IBM SP1 machine in [DI94]. Predictions of the expected running times of programs on these machines based on the LogP-model were confirmed by these works.

Although a shared memory is not explicitly excluded, we may assume that the memory is distributed over the P processors. Massively parallel computers may contain a shared address space but practically the memory is distributed over the processors when P is large. Designing programs directly

for distributed, asynchronous machines is a difficult task. Usually, it can be performed only by specialists. The programs are often very complicated, not understandable, and hardly portable. It was therefore beneficial to develop a new method to transform programs for the PRAM into distributed programs in a systematic way to ensure correctness, see [ZL94].

However, if we want to apply these techniques in compilers for parallel programs two preconditions have to be satisfied. First, the resulting programs have to be optimized. Second, the quality of the optimized programs has to be predictable. These are the two goals of this work.

Therefore, in section 2 we give some basic definitions and sketch the transformation from PRAM to LogP programs. In section 3 we show the reduction to P processors and the resulting program delay. In section 4 we extend the LogP machine model, we give the upper time bound for executing the unoptimized LogP program, and we define the notion of LogP schedules. In section 5 we define the class of coarse grained LogP programs, show how to optimize them, and prove the upper time bounds for executing the optimized programs. Additionally, we do the same with fine grained LogP programs. In section 6 we apply the transformations to the Fast Fourier Transformation. Finally, we summarize our results and show directions for further work.

2 Classification of Parallel Programs

First, we define the notion of communication structure of a PRAM algorithm. For this the implicit assumption that a PRAM has a global clock is made explicit. Then parallel algorithms are classified according to their communication structure.

Before defining communication structures, some assumptions have to be made. First, we assume that the programs are executed at the statement level, and that the running time is measured in the number of assignments executed. Second, the only composite data structures we use are arrays. This is no restriction as the shared memory may be considered as an array of integers. We allow the introduction of several arrays. Third, inputs are usually measured by their size. We use the overall number of single array elements. Finally, $P_A(n)$ denotes the maximum number of processors used by an algorithm A on inputs of size n and $T_A(n)$ denotes the worst-case running time of algorithm A on inputs of size n.

Definition 2.1 *Processor i* communicates *at time t with processor j iff there is a time t' and a memory cell m which was either written by processor j at time t' or $t' = 0$, no processor writes into m between time t' and time t, and processor i reads at time t from m. We denote this by the predicate $comm(i, t, j, t')$.*

Conditions in conditional statements and loops are viewed as assignments but without writing into the shared memory.

Definition 2.2 *A* communication structure *of a PRAM-algorithm A for an input x of size n is a directed acyclic graph $G_{A,x} = (V_{A,x}, E_{A,x})$, where*

$$V_{A,x} = \{\langle i, t\rangle : 0 \le i < P(n), 0 \le t \le T(n)\},$$

and

$$E_{A,x} = \{(\langle j, t'\rangle, \langle i, t\rangle) : t' < t \wedge comm(i, t, j, t')\}.$$

Let v be a vertex with $v \in V_{A,x}$. We denote the set of all direct predecessors of v in G with $PRED_v$ and the set of all transitive predecessors (ancestors) of v in G with $ANCESTOR_v$.

The set of vertices can be partitioned according to the time when the vertices are executed on the PRAM. We call this partitioning a *layering* of the communication structure:

Definition 2.3 (Layer) *A* layer $\Lambda^t_{A,x}$ *of a communication structure $G_{A,x}$ is the subset of vertices $v \in V_{A,x}$:*

$$\Lambda^t_{A,x} = \{v = \langle i, t'\rangle : v \in V_{a,x} \wedge 0 \le i < P(n) \wedge t' = t\}.$$

There are parallel algorithms where different communication structures for different inputs of the same size may occur. However, the communication structure does not vary for different inputs of the same size for many parallel algorithms e.g. FFT, finite element methods, solution of linear equation systems, matrix multiplication etc.

Definition 2.4 *A parallel algorithm is called* oblivious *iff its communication structure is the same for all inputs of the same size. Otherwise, it is called* non-oblivious.

All further transformations are based on the following assumptions:

(1) The PRAM-programs are oblivious.

(2) The input size n of the PRAM-program is known.

(3) Constant folding, loop unwinding, and recursion elimination have been applied completely to the PRAM-program.

(4) The programs representation is its communication structure, where every vertex $\langle i, t\rangle$ additionally contains the PRAM-statement executed at time t on processor i.

Any oblivious PRAM-program can be transformed into a semantically equivalent program that can be executed asynchronously on a distributed memory machine, i.e. on the LogP machine. For this task, three main steps are necessary. First, the number of processes must be reduced to the number of available processors. Second, the synchronous program has to be transformed into an equivalent asynchronous program. Third, the shared memory must be distributed to particular processes. For transforming a PRAM-program we take its communication structure that can be computed at compile time if the program is oblivious. Informally, every vertex of the communication structure corresponds to a separate process that receives data from the processes corresponding to its predecessors and sending data to the processes corresponding to its successors. This transformation leads to an equivalent LogP program. In the following we call this the *naive implementation*. A formal program transformation and the proof of its correctness is shown in [ZL94]. The same is done for non-oblivious programs in [Löw95].

3 Reducing PRAM-Processors

First, we show how to reduce the number of processors to P. We do this at PRAM-level using Brent's Lemma [Bre74]. He showed that an algorithm with input size n requiring work[1] $w(n)$ and time $T(n)$ can be simulated on P processors using time $\frac{w(n)}{P} + T(n)$. Since we know the communication structure of the program this lemma can be applied constructively:

[1] The work is the total number of operations performed by a PRAM algorithm

Algorithm 3.1 Reduce the number of processors of a PRAM-program with communication structure G to P:

Input: *G using $P(n)$ processors*
Output: *G' using P processors*

(1) $V' := V$; $E' := E$;
(2) **forall** $\Lambda^t \in G$ **do in parallel**
(3) **forall** $v = \langle i, t\rangle \in \Lambda^t$ **do in parallel**
(4) $E' := E' \cup \{(\langle i,t\rangle, \langle i+P, t\rangle) : \langle i+P, t\rangle \in \Lambda^t\}$;
(5) $E' := E' \cup \{(\langle i+P, t\rangle, \langle i', t'\rangle) :$
(6) $(\langle i,t\rangle, \langle i', t'\rangle) \in E, \langle i+P, t\rangle \in V\}$;
(7) **end**; $--$ **forall**
(8) **end**; $--$ **forall**
(9) $G' := (V', E')$.

Theorem 3.1 *Let A be a PRAM-program with communication structure G requiring $P(n)$ processors and time $T(n)$. Algorithm 3.1 produces an PRAM-program with communication structure G' requiring P processors and time $\lceil \frac{P(n)}{P} \rceil \times T(n)$.*

Proof: A layer Λ of G contains at most $P(n)$ vertices and can be executed in PRAM-time 1 on $P(n)$ processors. After having applied algorithm 3.1, Λ extends to $\lceil \frac{P(n)}{P} \rceil$ layers in G' with at most P vertices. Since, algorithm 3.1 considers each layer independently G' has $\lceil \frac{P(n)}{P} \rceil \times T(n)$ layers each of them containing at most P vertices. Therefore, A can be executed on P processors requiring PRAM-time $\lceil \frac{P(n)}{P} \rceil \times T(n)$. ◇

Note, that $w(n)$ is bounded by $P(n) \times T(n)$. Therefore, the required time reduces to the time predicted by Brent.

4 Executing LogP-Programs

For all further considerations we assume the communication structures to have P entry nodes and diameter time $T = \lceil \frac{P(n)}{P} \rceil \times T(n)$. With the transformation of the last section this can always be guaranteed. It remains to show that the following transformations do not increase the number of required processors.

We extend the LogP machine model by two features. First, we assume the machine to have a set of machine instructions. We don't specify the instruction. For the following it is just necessary to that each machine instruction requires a certain execution time. Since we know the sequence of instructions in each vertex v of the communication structure, we can statically compute an upper time bound C_v for executing the process corresponding to v on the target machine. Second, the parameter L is valid only for small messages. But, we can measure L for various length' of messages. As the length of the message sent from vertex v is known the required time L_v for this sending this message can be computed statically. These two extensions lead to an extension of the LogP machine that we call *LogPC machine*. Unless stated differently, in the following L, o, g, and P denote the parameters of the LogP-machine.

The naive implementation described at the end of the second section is correct but performs very poorly. This implementation requires at most $P(n) \times T(n)$ processors. Its worst case running time is given in theorem 4.1.

Theorem 4.1 *Let Π be a parallel program whose communication structure G has depth $T(n)$ and degree dg. If C is the maximal computation time for the vertices in G then the execution time of the naive implementation is at most:*

$$TIME_{simple}(G^*) \leq (T(n)-1) \times L + T(n) \times \max[o+C, g] + o + T(n) \times (dg-2) \times \max[o, g]$$

Proof: On the longest path $T(n)$ tasks have to be computed, $T(n)-1$ communications occur sequentially. Each communication requires time L. Sending and receiving a message requires time o. As the degree of the communication structure is dg, each process sends and receives at most dg messages. Between sending/receiving two messages, the time must be g. If $o > g$ then the time g is guaranteed. This explains the factor $\max[o, g]$. Between receiving the last and sending the first message of a vertex g is guaranteed if $o + C > g$. Altogether the time spent in one vertex of the communication structure is therefore at most $(dg-2) \times \max[o,g] + \max[o+C, g] + o + o$. This completes the proof because we chose the longest path. ◇

Observe, that if the number of vertices in the communication structure G is smaller than $TIME_{simple}(G)$ a sequential implementation is faster than the implementation described here.

Merging some of the processes into one processor saves time required for communication. On the other hand, the degree of parallelism is decreased. In the next section we discuss this tradeoff. In fact we *schedule* the computations done in the vertices of a communication structure onto the processors of the LogPC machine such that the execution time is minimal. But, first of all we define the notion of LogPC Schedule:

Definition 4.2 (LogPC Schedule) *Let $G = (V, E)$ be a communication structure. A* trace *tr for the LogPC-machine is a finite sequence of tuples $(v_i, m_i, t_i, p_i) \in V \times \{s, r, c\} \times \mathbb{N} \times (\mathbb{N} \cup \varepsilon)$ satisfying*

(1) $t_0 \geq 0$
(2) $t_{i+1} \geq \begin{cases} t_i + C_{v_i}, & \text{if } m_i = c \\ t_i + o, & \text{if } m_i \in \{r, s\} \end{cases}$
(3) $\forall (v_i, m_i, t_i, p_i), (v_j, m_j, t_j, p_j) \in tr :$
$m_i, m_j \in \{r, s\} \Rightarrow |t_i - t_j| \geq g \vee j = i.$
(4) $p_i \in \mathbb{N} \Leftrightarrow m_i \in \{s, r\}$
(5) $\forall\ (v_i, c, t_i, p_i) \in tr : \forall\ u \in PRED_{v_i} :$
$\exists\ (v_j, m_j, t_j, p_j) \in tr : j < i \wedge v_j = u.$
(6) $\forall\ (v_i, s, t_i, p_i) \in tr :$
$\exists\ (v_j, m_j, t_j, p_j) \in tr : j < i \wedge v_j = v_i$

A clustering *$\mathcal{C}$ of G for the LogPC machine is a finite sequence of traces satisfying:*

(7) $\forall\ (v, r, t, p) \in tr_k \in \mathcal{C} :$
$\exists\ (v, s, t', k) \in tr_p \in \mathcal{C} : t' \leq t - L_v - o.$
(8) $\forall\ tr \in \mathcal{C} : \forall\ (v, s, t, p) \in tr : p < |\mathcal{C}|.$
(9) $\forall\ v \in V : \exists\ tr \in \mathcal{C} : (v, c, t, \varepsilon) \in tr \wedge t \in \mathbb{N}.$

A schedule *$\mathcal{S}$ of G for the LogPC machine is a clustering $\mathcal{C}$ for the LogPC machine with $|\mathcal{C}| \leq P$. A trace $tr = (v_0, m_0, t_0, p_0) \cdots (v_l, m_l, t_l, p_l)$ has* execution time

$$TIME(tr) = \begin{cases} t_l + C_{v_l}, & \text{if } m_l = c \\ t_l + o, & \text{if } m_l \in \{r, s\} \end{cases}$$

The execution time of a schedule *$\mathcal{S}$ is defined as*

$$TIME(\mathcal{S}) = \max_{tr \in S} TIME(tr)$$

The optimal execution time of G for schedules *is defined by*

$$TIME_{opt}(G) = \min\{TIME(\mathcal{S}) : \mathcal{S} \text{ schedules } G\}$$

Remark: A trace corresponds to a program on a processor. A tuple (v, m, t, p) means that at time t, v is computed ($m = c$), received from processor p ($m = r$), or sent to processor p ($m = s$). Condition (1) and (2) ensure that no processor is idle. However it may perform redundant computations. Condition (3) ensures the gap g, condition (4) ensures that a processor number p_i is given whenever a message is sent to or received from p_i. (5) ensures that all operands are available to perform an operation. Condition (6) ensures that data to be sent are available. Condition (7) ensures that messages to be received by a processor are sent early enough by another processor and condition (8) ensures that all messages are sent to valid processor addresses. Property (9) ensures that every vertex of the communication structure is computed. ◇

Definition 4.3 (Linear and Nonlinear) *Two vertices u and v of a communication structure are called* independent *iff neither u is an ancestor of v nor vice versa. A LogPC clustering $\mathcal{C}$ is called* linear *iff it does not contain a trace with two independent vertices. Otherwise it is called* nonlinear.

5 Reducing LogPC-Time

Any oblivious PRAM-program can be implemented as an optimal LogPC program. But, this transformation itself seems to be exponential, see [ZL94]. In [PY90] Papadimitriou and Yannakakis showed that finding an optimal schedule is NP-hard, even if $o = g = 0$ and $P = \infty$. They also showed that approximative solutions with an upper bound smaller than $2 \times TIME_{opt}(G)$ cannot be found in polynomial time when $o = g = 0$ unless P=NP. We can therefore not expect to find an efficient and optimal transformation. However, we demonstrated in [LZ95a] that an optimal solution can be found in polynomial time if the communication structure is coarse grained. Furthermore, Gerasoulis and Yang demonstrated in [GY93] that a solution guaranteeing $2 \times TIME_{opt}(G)$ without vertex duplications can be found for coarse grained communication structures assuming that $o = g = 0$.

A LogPC schedule must additionally take into account that

(1) sending and receiving a message takes time o, that

(2) between two sends or receives on one processor, there must be at least time g, and that

(3) when receiving a message, it must be sent by some other processor at least $L+o$ time units earlier in order to avoid waiting times.

For the LogP machine, Karp et al. presented in [KSSS93] an optimal solution for broadcasting and summation problems.

We are proceeding in the following way: First, we define the property of a coarse grained communication structure and give an algorithm leading to a $2 \times TIME_{opt}(G)$ solution, where P processors are required if G is coarse grained. Second, we adapt the algorithm of Papadimitriou and Yannakakis to the LogPC machine and prove the worst case running time of the solutions on this machine.

Informally speaking, a program is coarse grained if sequential computation always lasts longer than communication between these sequential processes.

Lemma 5.1 (Maximal Communication) *The overall time $L_{max}(u, v)$ for communication from a process u to a direct successing process v is at most*

$$(L_u + 2o + (odg_u + idg_v - 2) \times max[o, g])$$

Proof: Assume that a message is the last to be sent from u. Then there are $odg_u - 1$ messages sent before from u (odg_u is the out-degree of vertex u). Hence, $odg_u - 1$ gaps have to be guaranteed. Then, sending the message takes time o and the communication delay is L_u. The same holds for receiving the message in v where $idg_v - 1$ gaps have to be guaranteed (idg_u is the in-degree of vertex v). ◇

Definition 5.2 (Coarse and Fine Grained) *Let $PRED_v$ be the set of all direct predecessors a u of a vertex v in a communication structure G. Let L_u be the communication delay for sending a message from vertex u to vertex v. Let C_u be the time for executing the statements of u. The* granularity of a vertex *is defined as*

$$g(v) = \frac{\min_{u \in PRED_v} \{C_u\}}{\max_{u \in PRED_v} \{L_{max}(u, v)\}}$$

and the granularity of a communication structure is defined as

$$g(G) = \min_{v \in G} \{g(v)\}.$$

G is coarse grained *if $g(G) \geq 1$, otherwise it is called* fine grained.

Remark: This definition of granularity defines more communication structures as coarse grained than the definition in [GY93]. As we will see, it is easier to optimize communication structures with coarse granularity. ◇

5.1 Scheduling General Coarse Grained Structures

For scheduling coarse grained programs we first consider linear clusterings.

Theorem 5.3 (Maximal Execution Time) *Any linear LogPC clustering of a coarse grained communication structure G leads to a program running in at most $2 \times TIME_{opt}(G)$.*

Proof: Let $\mathcal{P}$ be the set of all paths from vertices v_i with $idg_{v_i} = 0$ to vertices v_o with $odg_{v_o} = 0$. For the optimal clustering of G holds:

$$\forall P \in \mathcal{P} : TIME_{opt}(G) \geq \sum_{v \in P} C_v.$$

Since, G is coarse grained, it additionally holds:

$$\forall P \in \mathcal{P} : \sum_{u,v \in P} (L_{max}(u, v)) \leq \sum_{v \in P} C_v.$$

Therefore, even the naive implementation requires at most:

$$TIME_{naive}(G) \leq 2 \times TIME_{opt}(G)$$

It is easy to see that a linear clustering cannot increase the running time compared to the naive implementation. ◇

Remember, we have assumed that G requires at most P PRAM processors. Any linear clustering algorithm that doesn't increase the number of required processors produces automatically a valid schedule for the LogPC machine running in at most a factor of two of its optimum. Heuristic linear clustering algorithms satisfying this condition can be found for instance in [Sar89, NT93, ZL94].

5.2 Scheduling Coarse Grained Tree Like Structures

Each coarse grained communication structure can be scheduled optimally on a LogPC machine if $o = 0$, $g = 0$, and $P = \infty$, see [LZ95a]. We extend these results to a machine with $o \neq 0$, $g \neq 0$, and $P = const$. Therefore, we define a property *tree like* of communication structures.

Definition 5.4 *A communication structure G is called* tree like *iff all its vertices v with $odg_v = 0$ are roots of inverse trees[2] such that each of these inverse trees contains a vertex of G at most once.*

Of course, it can be decided in polynomial time if a communication structure is an inverse tree. Furthermore, the inverse trees of a tree like communication structure can be computed in polynomial time. The algorithms for doing these computations are omitted. Each inverse tree can be clustered as shown in algorithm 5.1.

Algorithm 5.1 Compute a Clustering for Inverse Trees.

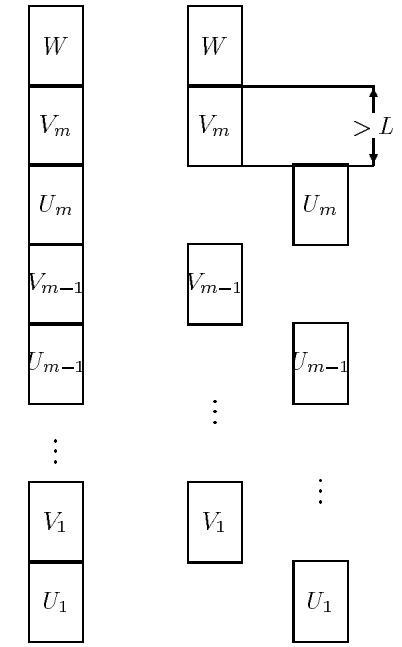
Input: *Inverse Tree G with depth $T(n)$, Coarse Grained*
Output: *Clustering for G*

```
( 1) forall v ∈ Λ^0 do in parallel
( 2)   tr_v := (v, c, 0, 0);
( 3) end; -- forall
( 4) for j := 1 to T(n) do
( 5)  forall v ∈ Λ^j do in parallel
( 6)   choose u* ∈ U where
( 7)    U = {ū : ū ∈ PRED_v ∧ ∀u ∈ PRED_v :
( 8)              TIME(tr_ū) + L_ū ≥ TIME(tr_u) + L_u}
( 9)    tr_v := tr_u*;
(10)   forall u ∈ PRED_v \ {u*} do
(11)    tr_u :=tr_u ∪ {(u, s, TIME(tr_u), p(tr_u*))};
(12)    t_u := TIME(tr_u) + L_u;
(13)    tr_v :=tr_v ∪ {(u, r, max[TIME(tr_u*), t_v], p(tr_u))};
(14)   end; -- for
(15)   tr_v := tr_v ∪ {(v, c, TIME(tr_u), 0)};
(16)  end; -- forall
(17) end; -- for
```

Remark: The algorithm starts with creating a separate trace for every vertex in layer Λ^0 of G. The first tuple in each trace computes a vertex of G with in-degree 0 (lines 1-3). The statements in the lines 6-13 can be executed for each layer in parallel and must be executed for all layers from first to last (lines 4-5). All predecessors $u \in PRED_v$ in the communication structure have already been added to a trace $tr(u)$ and can be computed in $TIME(tr(u))$. Sending data from u to v takes time $L_u + o$. For each considered vertex v the following is computed. First, a trace tr_{u^*} with the longest time for computing plus sending is selected (lines 6-8). Second, the tuples for sending and receiving data from the others predecessors are added to the traces of these predecessors and to the trace tr_{u^*}(lines 11-14). Finally, the tuple for computing v is added to trace tr_{u^*} (line 15). ◇

Lemma 5.5 (Linear versus Nonlinear) *Let be $o = g = 0$. Then there is for every nonlinear clustering $\mathcal{C}_1$ of an inverse tree G a linear clustering $\mathcal{C}_2$ of G with $TIME(\mathcal{C}_2) \leq TIME(\mathcal{C}_1)$.*

Proof: We only have to consider traces tr_v containing two independent vertices u and v (see definition 4.3). We assume w.l.o.g. that u has a smaller starting time in tr than v. Then tr consists of a part W after the computation of v and parts $U_1, \ldots, U_m$ containing only u and ancestors of u and $V_1, \ldots, V_m$ containing only v and ancestors of v (figure 1 (a)). Observe that U_m and V_m end with computing u and v, respectively. We copy all tuples in $U_1, \ldots, U_m$ into a separate trace tr_u and remove them from tr (figure 1 (b)). Because the communication structure is coarse-grained and $o = g = 0$ the result u is available at the beginning of W. Therefore if we replace in a clustering the trace tr as in figure 1 then the result is again a clustering.



(a) trace (b) separated traces

Figure 1: Separation of two independent vertices u and v

By induction, separating traces can be done until the clustering is linear. Since, each separation step doesn't increase the computation time the resulting linear clustering has at most the same computation time as the nonlinear clustering. ◇

Theorem 5.6 *Let G be a coarse grained inverse tree that is the communication structure of an PRAM-algorithm using $P(n)$ processors. Let $T(n)$ be the depth of G and idg the maximal in-degree of the vertices of G. Then algorithm 5.1 computes in time $O(T(n) \times \log idg)$ a clustering of G on a CREW-PRAM with $P(n)$ processors. The resulting clustered program requires at most time $TIME_{opt}(G) + T(n) \times (idg - 1) \cdot \max(o, g) + T(n) \cdot o$ on $P(n)$ LogPC processors.*

Proof: (Correctness) The correctness is proven by induction on the layers of G. First, we show that algorithm 5.1 clusters inverse trees optimally, if $o = g = 0$. Second, we show the maximal impact of o and g on the execution time.

Since all vertices in layer Λ^0 start at time 0 these vertices have an optimal starting time. We assume now that all

[2]The inverse of a tree is the graph containing all its vertices and all its edges in inverse orientation

vertices of a layer Λ^i are computed optimally and show that vertices in layer Λ^{i+1} have an optimal starting time as well. All vertices in Λ^{i+1} have no predecessor in common since G is an inverse tree. Therefore, we only have to consider one vertex v in Λ^{i+1} to show the correctness of the iteration step. Furthermore, because of lemma 5.5 we only have to consider linear clusterings. To produce a linear cluster containing v we can add v to at most one of the traces of the predecessors of v. If v is computed separately then for the starting time holds $t_v = TIME(tr_{u^*}) + L_{u^*}$. If v is clustered with a predecessor $u \neq u^*$ the starting time t_v does not change. If v is clustered with u^* for the starting time t_v it holds

$$t_v = \max[TIME(tr_{u^*}); \max_{u \in (PRED_v \setminus \{u^*\})}\{TIME(tr_u) + L_u\}].$$

which is at most equal to the time for computing v separately. Since there is no other way to produce a better linear clustering, v is computed optimally if it is clustered with u^*. By induction, the algorithm produces optimal clusterings, if $o = g = 0$.

The overheads for sending always occur at the end of traces. They cannot be saved or hidden. Hence, they do not affect the optimality of the clustering. Receiving operations have not necessarily to be added immediately before the vertex which processes the received data. They can also be inserted in existing gaps before. Therefore, the optimal clustering is not necessarily delayed by overheads for receiving. There are at most $T(n) \times (idg - 1)$ receiving operations in a trace. The resulting clustering is therefore delayed at most with time $T(n) \times (idg - 1) \times \max[o, g] + T(n) \times o$.
(Required LogPC Processors) There are $P(n)$ vertices in Λ^0, each of them is executed in a separate trace. Since all the other vertices are added to existing traces and no trace is copied, $P(n)$ traces (i.e. processors) are required for the clustered program.
(Complexity) Lines (1) to (3) can be computed in parallel in time $O(1)$ if there are $P(n)$ processors available. The overall number of predecessors for any layer is bounded by $P(n)$, because the communication structure is an inverse tree. Computing the maximum of m items takes time $O(\log m)$ on a PRAM with m processors. We can compute the maximum for each v in layer L_j in parallel. As each v has at most idg predecessors, this can be done in time $O(\log idg)$. Hence, step (6) requires altogether at most $P(n)$ processors. Adding the tuple containing v takes time $O(1)$. Observe that $\max_{u \in (PRED_v \setminus \{u^*\})}[TIME(tr_u) + L_{u,v}]]$ is already computed in step (6). There are $T(n)$ layers in the communication structure, thus the complexity for the whole algorithm is $O(T(n) \times \log idg)$. ◇

It is easy to see that applying the described algorithm to tree like communication structures by clustering each inverse tree independent of the others does not increase the time bound. But, there are $|O|$ inverse trees in a tree-like communication structure, where O is the set of all vertices with $odg_v = 0$. Therefore, the described algorithm requires $P(n) \times |\Lambda^{T(n)}|$ processors. Hence, we must not reduce the number of PRAM-processors to P. Since, we require $P(n) \times |O|$ LogPC processors we have to reduce the number of PRAM processors such that $P(n) \times |O| = P$. Unfortunately, $|O|$ is in the worst case equal to $P(n)$. In this case we must reduce the number of PRAM-processors not to P but to $\sqrt{P}$. Hence, it is reasonable to use algorithm 5.1 in two cases. First, if an inverse tree (e.g. for parallel expression evaluation) has be scheduled. Second, if enough LogPC processors are available (i.e. for small sized problems).

5.3 Scheduling Fine Grained Programs

We first show how linear clustering strategies behave in case of fine grained communication structures, follow the ideas of Gerasoulis and Yang in [GY93], and apply them to the LogPC machine. Then we generalize the ideas of [PY90] for the LogPC machine

Theorem 5.7 (Maximal Execution Time) *Any linear LogPC clustering of a communication structure G leads to a program running in at most $(1 + 1/g(G)) \times TIME_{opt}(G)$.*

Proof: Let $\mathcal{P}$ be a path from a vertex v_i with $idg_{v_i} = 0$ to a vertex v_o with $odg_{v_o} = 0$ such that sum of computation times of its vertices and communication delay between these vertices is maximal under all paths from input vertices two output vertices. For the optimal clustering of G holds:

$$TIME_{opt}(G) \geq \sum_{v \in \mathcal{P}} C_v.$$

Let $L_{max}(v) = L_{max}(u, v)$ where u is the predecessor of v in path $\mathcal{P}$. Set $L_{max}(v_i) = 0$. For the naive clustering of G it holds:

$$\begin{aligned}
TIME_{naive}(G) &\leq C_{v_o} + \sum_{v \in \mathcal{P}} C_u + L_{max}(v) \\
&\leq C_{v_o} + \sum_{v \in \mathcal{P}} C_u (1 + \frac{L_{max}(v)}{C_u}) \\
&\leq C_{v_o} + \sum_{v \in \mathcal{P}} C_u (1 + \frac{1}{g(v)}) \\
&\leq C_{v_o} + (1 + \tfrac{1}{g(G)}) \sum_{v \in \mathcal{P}} C_u \\
&\leq (1 + \tfrac{1}{g(G)})(C_{v_o} + \sum_{v \in \mathcal{P}} C_u) \\
&\leq (1 + \tfrac{1}{g(G)}) \sum_{v \in \mathcal{P}} C_v \\
&\leq (1 + \tfrac{1}{g(G)}) \times TIME_{opt}(G)
\end{aligned}$$

Since a linear clustering cannot increase the running time compared to the naive implementation the proof is complete. ◇

Theorem 5.7 is a generalization of theorem 5.3 since in the proof above the communication structure is not assumed to be fine or coarse grained. Furthermore, this solution is a scheduling strategy for fine grained programs because it is easy to find algorithms for linear clustering that do not increase the number of required LogPC processors compared to the number of PRAM-processors. For improving this solution we consider the lower bounds for the optimum more carefully. Following the ideas of Papadimitriou and Yannakakis in [PY90], we compute lower bounds for the minimal starting times $t_{min}(v)$ of each vertex v in the communication structure in terms of L, o, g, and C. This is done by algorithm 5.2.

Algorithm 5.2 Compute the minimal starting times $t_{min}(v)$ for vertices v of a communication structure G:

Input: *G with depth $T(n)$*
Output: *$t_{min}(v)$ for all v in G*

(1) **forall** $v \in \Lambda^0$ **do in parallel**
(2) $t_{min}(v) := 0$;

(3) **end; −− forall**
(4) **for** $i := 1$ **to** $T(n)$ **do**
(5) **forall** $v \in \Lambda^i$ **do in parallel**
(6) $ANCESTOR_v = \{u : \exists path\ (u, \ldots, v)\ in\ G\}$;
(7) **order** $u \in ANCESTOR_v$ **that**
(8) $t_{min}(u_1) + C_{u_1} + o + L_{u_1} \geq \cdots$
(9) $\cdots \geq t_{min}(u_n) + C_{u_n} + o + L_{u_n}$;
(10) $ANCESTOR_v^0 := \emptyset$;
(11) **for** $j := 1$ **to** n **do**
(12) $ANCESTOR_v^j := ANCESTOR_v^{j-1} \cup \{u_j\}$;
(13) **order** $u \in ANCESTOR_v^j$ **that**
(14) $t_{min}(u_1) \geq \cdots \geq t_{min}(u_j)$;
(15) $t_j(v) := \max_{k=1}^{j}[t_{min}(u_k) + \sum_{l=1}^{k} C_{u_l}]$;
(16) **end; −− for**
(17) $t_{min}(v) := \min_{j=1}^{n}\{\max[t_j(v), t_{min}(U_{j+1} + C_{u_{j+1}} + o + L_{u_{j+1}}]\}$;
(18) **end; −− forall**
(19) **end; −− for**

Remark: The algorithm starts with setting the minimal starting times of the vertices in layer Λ^0 to zero (lines 1-3). The statements in the lines 6-17 can be executed for the vertices of each layer in parallel and must be executed for all layers from first to last (lines 4-5). For each considered vertex v the following is computed. First, the set of all its ancestors $ANCESTOR_v$ is computed (line 6). Second, these ancestors are ordered by the earliest possible receiving times for data from these predecessors at v if they are computed on different processors (lines 7-9). Third, the times for computing the first j ancestors on one processor are computed for $j = 1, 2, \ldots, n$ where n is the number of ancestors of v (lines 10-16). Finally, the minimal starting time of v is set to the minimum of the values computed before where it is guaranteed that the results from ancestors not computed on the same processor are available in time (line 17). ◇

Theorem 5.8 (Minimal Starting Time) *It is not possible to find a valid LogPC schedule that schedules v earlier than $t_{min}(v)$ even if we assume $P = \infty$.*

Proof: The proof is by induction on the layers of G. Obviously, no vertex in layer Λ^0 can be scheduled before its minimal starting time since it is zero. Suppose that vertices u in Λ^{i-1} are scheduled at time $t_{min}(u)$ or later and v is scheduled before time $t_{min}(v)$. We prove that then properties of a valid LogPC schedule are violated. Let x be such that $t_{min}(v) = \min[t_x(v), t_{min}(u_{x+1}) + C_{u_{x+1}} + o + L_{u_{x+1}}]$.
CASE 1: v and u_x are computed on the same processor. Then all vertices in $ANCESTOR_v^x$ must also be computed on this processor. But then condition (2) of a LogPC schedule is violated if $t_{min}(v) = t_x(v)$ and condition (7) of a LogPC schedule is violated if $t_{min}(v) = t_{min}(u_{x+1}) + C_{u_{x+1}} + o + L_{u_{x+1}}$. Contradiction.
CASE 2: v and u_x are computed on different processors. Observe that $t_i(v)$ is non-increasing and $t_{min}(u_i) + C_{u_i} + o + L_{u_i}$ is non-decreasing. Therefore $t_{min}(v) \leq C_{u_x} + o + L_{u_x}$ no matter whether $t_{min}(v) = t_x(v)$ or $t_{min}(v) = t_{min}(u_{x+1}) + C_{u_{x+1}} + o + L_{u_{x+1}}$. But then condition (7) of a LogPC schedule is violated, because u_x and v are computed on different processors. ◇

Algorithm 5.3 computes the sequence of vertices in the traces of a schedule of a communication structure. It uses the minimal starting times of every vertex and the set of predecessors leading to this minimum computed in algorithm 5.2. The very algorithm for scheduling the vertices in G additionally has to compute the exact times for starting each vertex and has to insert sending and receiving operations. It is omitted.

Algorithm 5.3 Compute the sequence of vertices v of a communication structure G in the traces of a schedule of G:

Input: *G with depth $T(n)$, $\{t_{min}(v)\}$, $\{ANCESTOR_v^x\}$*
Output: *schedule of G*

(1) **for** $i := T(n)$ **downto** 0 **do**
(2) **forall** $v \in \Lambda^i$ **do in parallel**
(3) **if** *v is not scheduled* **then**
(4) *schedule v on a separate processor;*
(5) *schedule all $\bar{v} \in ANCESTOR_v^x$ on processor of v*
(6) **end; −− if**
(7) **end; −− forall**
(8) **end; −− for**
(9) *compute a linear clustering*
(10) *for the schedule obtained so far;*

Theorem 5.9 (Maximal Execution Time) *Let G be a communication structure of depth T with vertices v that have a maximal in-degree of idg and a maximal out-degree of odg. Let layer Λ^1 contain $|\Lambda^1|$ vertices. Algorithm 5.3 defines schedules of a communication structure G leading to programs that can be executed on the LogPC machine in at most:*

$$TIME(S) \leq 2 \times TIME_{opt}(G) + \max[|\Lambda^1|, T] \times (idg + odg - 2) \times \max[o, g].$$

Let P be the number of PRAM-processors required for computing G, and O be the set of vertices with out-degree of zero. Algorithm 5.3 produces schedules requiring at most $|O| \times P$ LogPC processors.

We prove theorem 5.9 in three steps. First, we show that the resulting programs can be executed in time less or equal $2 \times TIME_{opt}(G)$, if $o = g = 0$ and $P = \infty$. Second, we discuss the influence of o and g to the execution time. Finally, we determine the number of processors required by the resulting schedules.

Proof: (*Correctness)* If $o = g = 0$ the schedule defined by algorithm 5.3 has a maximal execution time of $2 \times TIME_{opt}(G)$. For proving this we refer to [PY90].

Easy computations show that the maximal number of sequential sending and receiving operations occurs either if each vertex is clustered with only one of its predecessors ($T \times (idg + odg - 2)$) or if each vertex of layer Λ^1 is clustered with only one of its predecessors and all vertices of layers $\Lambda^{>1}$ are clustered with all its predecessors ($|\Lambda^1| \times (idg + odg - 2)$). Assuming the worst case we have an delay by sending and receiving operations of at most:

$$\max[|\Lambda^1|, T] \times (idg + odg - 2) \times \max[o, g].$$

(Required Processors) Worst case is each of the vertices is clustered with only one of its predecessors. We consider the number of processors required for computing one vertex in O. Since, a vertex is duplicated only for computing different descendants computing a vertex in O requires at most P processors. Therefore, computing all vertices in O requires at most $|O| \times P$ LogPC processors.
(Complexity) First, we consider the time required for algorithm 5.2 which is a precondition of algorithm 5.3. We can

compute lines (1) to (3) parallel time $O(1)$ if there are $P(n)$ processors available. The statements in the lines 6-17 can be executed for the vertices of each layer in parallel and must be executed for all layers from first to last (line 4). There are $T(n)$ layers, each of them containing at most $P(n)$ vertices. At each layer, the overall number of ancestors is bounded by $P(n) \times T(n)$. Sorting n items takes time $O(\log n)$ on a PRAM with n processors. We order all ancestors twice: first by descending completion time (lines 7-9) and second by descending starting time (lines 13-14). For the latter we precompute the prefix sums in $O(\log n)$ (needed in line 15). Hence, we can compute the innermost loop in time $O(n)$. For computing the minimum of n items we need additionally time $O(\log n)$. Therefore, algorithm 5.2 runs in time $O(T(n)^2 \times P(n))$ on a PRAM with $P(n)^2 \times T(n)$ processors. We can run algorithm 5.3 in time $O(T(n)^2 \times P(n))$ since the number of layers is $T(n)$ and the cardinality of largest set of ancestors of a vertex is at most $T(n) \times P(n))$. Since algorithm 5.3 requires only $P(n)$ processors, the algorithm requires time $O(T(n)^2 \times P(n))$ on a PRAM with $P(n)^2 \times T(n)$ processors. ◇

As proven, algorithm 5.3 produces clusterings with at most $|O| \times P(n)$ traces. Hence, we have to reduce the number of PRAM processors such that $P(n) \times |O| = P$.

Theorem 5.10 (Linear versus Nonlinear) *Let G be a communication structure with granularity $g(G)$. Let $|O|$ be the number of vertices with out-degree in G. Linear clustering of G can guarantee a smaller upper time bound for execution than a clustering of G with algorithm 5.3 if*

$$g(G) > \begin{cases} 1 & \text{if } P(n) \times |O| \leq P \\ \frac{1}{2 \times |O| - 1} & \text{else} \end{cases}$$

Proof: Theorem 5.10 holds if the time bound for the naive clustered G is smaller than the time bound for the clustering of G with algorithm 5.3. If there are enough (i.e. $P(n) \times |O|$) processors available this condition is satisfied if:

$$(1 + \frac{1}{g(G)}) TIME_{opt}(G) \leq 2 \times TIME_{opt}(G).$$

If the number of processors has to be reduced to $\frac{P}{|O|}$ this condition holds if:

$$(1 + \frac{1}{g(G)}) TIME_{opt}(G) \leq 2|O| \times TIME_{opt}(G).$$

The factor of $|O|$ in the latter can be explained by the reduction of processors to $\frac{1}{|O|}$ of the number of processors required for linear clustering if algorithm 5.3 is applied. Easy transformations of the equations above complete the proof. ◇

Theorem 5.10 allows us to decide statically which clustering strategy is promising for a concrete program that has to be optimized for a concrete machine.

For determining the time bounds of the scheduled LogPC-programs we implicitly assumed a global clock and a synchronous execution w.r.t. this clock. We can drop these assumptions. The asynchronous execution model has a virtual global clock, and at each step each processor and each communication is executed with probability q. Observe that $q = 1$ means synchronous execution. We showed in [LZ95b] that under these assumptions the expected delay is just a constant.

6 Example - FFT

Let ω_n be the n-root of unity. We assume in algorithm 6.1 computing a Fast Fourier Transformation that n is an integral power of 2 and $r(i)$ denotes the number arising from the reversed bit representation of i. The communication structure of this algorithm for $n = 8$ (without steps (1-3) which is just a permutation) is shown in figure 2.

Algorithm 6.1 Compute FFT:

```
( 1) forall i ∈ {0,...,n−1} do in parallel
( 2)   x[i] := x[r(i)];
( 3) end; −− forall;
( 4) for i := 1 to log n do
( 5)   forall j ∈ {0,...,n−1} do in parallel
( 6)     if j mod 2^i < 2^(i−1)
( 7)     then x[j] := x[j] + ω_{2^i}^j · x[j + 2^(i−1)];
( 8)     else x[j] := x[j] + ω_{2^i}^j · x[j + 2^(i−1)];
( 9)     end; −− if
(10)   end; −− forall
(11) end; −− for
```

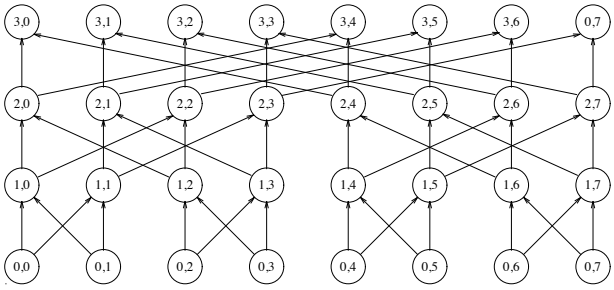


Figure 2: Communication Structure of FFT ($n = 8$).

If we execute algorithm 3.1 on the communication structure of the FFT on $n = 8$ (figure 2) to reduce the number of processors to $P = 4$ we obtain the communication structure shown in figure 3. The dashed edges are the edges inserted by algorithm 3.1.

Figure 4 shows a LogPC-schedule of the communication structure shown in figure 2. We assumed there that the computation time $C = 8$ for each vertex, latency $L = 2$, overhead $o = 1$, and gap $g = 2$, which makes our example coarse grained. Figure 5 shows a LogPC-schedule of the communication structure shown in figure 3. Observe that this last schedule requires $P = 4$ processors. Both schedules are linear.

Table 1 shows the minimal starting times for each vertex in the communication structure shown in figure 3. We assumed that for each vertex the computation time is $C = 1$, the latency $L = 1$, the overhead $o = 1$, and the gap $g = 2$.

We schedule now the communication structure in figure 3 according to scheduling algorithm 5.3. The minimal starting times of the ancestors of vertex (3,4) are shown in table 1. The corresponding schedule is shown in figure 6. The schedules for vertices (3,5), (3,6), and (3,7) are up to the processor numbers the same. Observe that this is not anymore a linear clustering. The schedule computes some vertices more than once. Its execution time is 15. This could

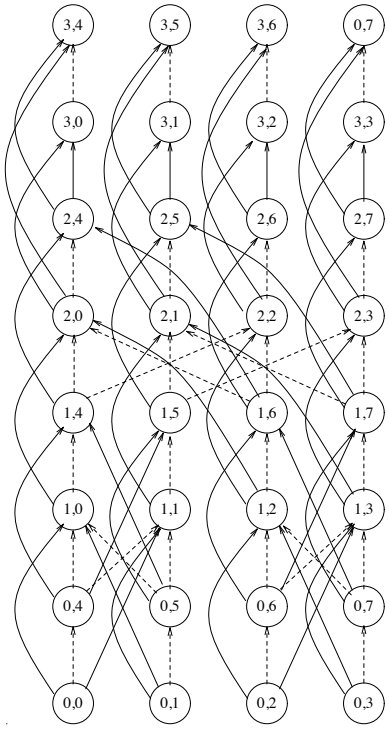


Figure 3: Applying Algorithm 3.1 on FFT ($n = 8, P = 4$).

v	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
$t_{min}(v)$	0	0	0	0	1	1	1	1
v	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
$t_{min}(v)$	3	3	3	3	4	4	4	4
v	(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
$t_{min}(v)$	6	6	6	6	7	7	7	7
v	(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
$t_{min}(v)$	6	6	6	6	7	7	7	7

Table 1: Minimal Starting Times of Processes in FFT ($n = 8, P = 4$).

almost be achieved by clustering all predecessors of the output vertices onto one processor. But this can only be done because the initial numbering of the vertices yields a good result of algorithm 3.1. The general clustering algorithm may use more processors but it also more robust against the initial numbering of vertices.

7 Conclusions

We showed for a subclass of parallel programs for PRAMs that the gap between theory and practice can be bridged by mapping this class onto an asynchronous machine with distributed memory - the LogPC machine. This class of *oblivious* parallel programs is characterized by their communication behavior which is the same for any input of size n. From a practical point of view this class is large. It contains for example parallel programs like FFT, finite element methods, solution of linear equation systems, matrix multiplication etc. For oblivious PRAM-programs the communication behavior can be derived by standard techniques

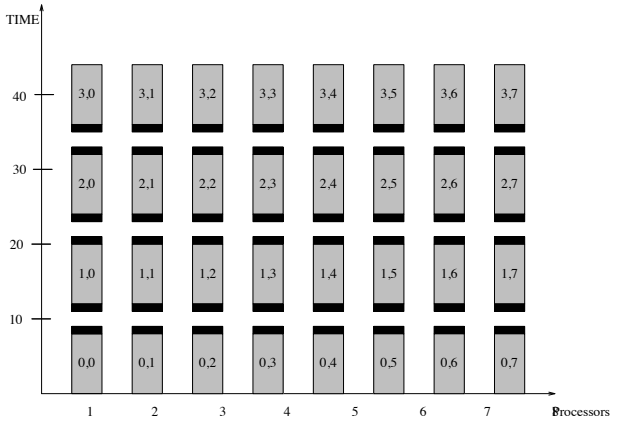


Figure 4: Linear Schedule for FFT ($n = 8$).

[ZL94]. Hence, we are able to apply clustering and scheduling algorithms for mapping oblivious PRAM programs onto the LogPC machine.

We generalized some well-known heuristics for scheduling and clustering by considering not only the communication latency, but also the overheads for sending/receiving messages, the network bandwidth, and a constant number of processors using the LogPC machine model. We discussed these heuristics in terms of this more realistic machine model. It turned out that the algorithms which increase the number of required LogPC processors compared to the number of PRAM-processors are not practicable in general although they give better results if the number of processors available is assumed to be unlimited.

We further showed the upper bounds for executing PRAM-programs on the LogPC machine using different scheduling algorithms. It is therefore possible to predict statically the maximal running time of a PRAM program on existing architectures and its the maximal delay compared to the optimal solution. Furthermore, we can statically decide for a concrete program which scheduling strategy promises to gain the best solution.

Additionally, we demonstrated that the algorithms for clustering and scheduling themselves can run in parallel. Unfortunately, these algorithms are non-oblivious. However, using the same techniques as described in [LZ95a] they can be transformed into oblivious algorithms. Then they can be applied on themselves to get parallel clustering and scheduling programs running efficiently on existing architectures. This is one direction for further research.

The clustering algorithms for fine-grained PRAM-programs may increase the number of required processors. It is therefore necessary to find clustering algorithms for fine grained programs which are better than linear clustering and do not increase the number of required processors. It is an open question whether such algorithms exist at all. Further research has to answer these questions.

Acknowledgments: We thank Arne Frick, Martin Middendorf, and the anonymous referees for their helpful comments.

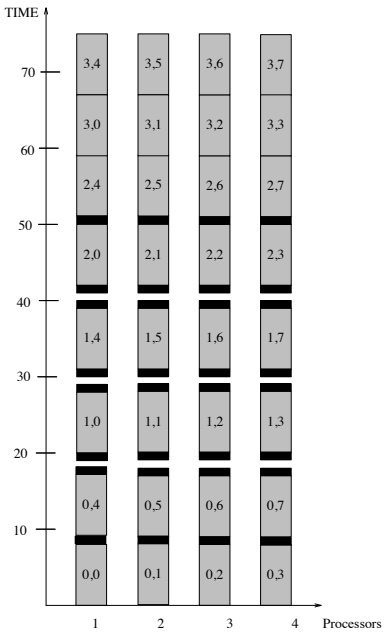


Figure 5: Linear Schedule for FFT ($n = 8, P = 4$).

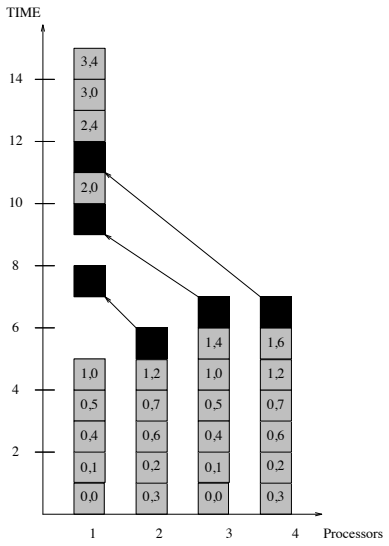


Figure 6: Part of a Nonlinear Schedule for FFT ($n = 8$).

References

[Bre74] R.P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, pages 201 – 206, 1974.

[CKP+93] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP 93)*, pages 1–12, 1993. Published in SIGPLAN Notices (28)7

[DI94] B. Di Martino and G. Ianello. Parallelization of non-simultaneous iterative methods for systems of linear equations. In *LNCS 854, Parallel Processing: CONPAR'94–VAPP VI*, pages 254–264. Springer, 1994.

[GY93] A. Gerasoulis and T. Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4:686–701, June 1993.

[KR90] R. M. Karp and V. Ramachandran. Parallel algorithms for shared memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science Vol. A*, pages 871–941. MIT-Press, 1990.

[KSSS93] R.M. Karp, A. Sahay, E.E. Santos, and K.E. Schauser. Optimal broadcast and summation in the LogP model. *ACM-Symposium on Parallel Algorithms and Architectures*, 1993.

[Löw95] W. Löwe. Optimization of PRAM-programs with input-dependent memory access. To appear in *LNCS, Proceedings of the EUROPAR'95*. Springer, 1995.

[LZ95a] W. Löwe and W. Zimmermann. On finding optimal clusterings of task graphs. In *Aizu International Symposium on Parallel Algorithm/Architecture Synthesis*, pages 241–247. IEEE Computer Society Press, 1995.

[LZ95b] W. Löwe and W. Zimmermann. Programming data-parallel – executing process-parallel. To appear in Proceedings of the *Zeus'95 Workshop*. 1995.

[NT93] M.G. Norman and P. Thanisch. Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys*, 25(3):263–302, 1993.

[PY90] C.H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal on Computing*, 19(2):322 – 328, 1990.

[Sar89] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, Massachusetts, 1989.

[Val90] L. G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science Vol. A*, pages 945–971. MIT-Press, 1990.

[ZK93] W. Zimmermann and H. Kumm. On the implementation of virtual shared memory. In *Programming Models for Massively Parallel Computers*, pages 172–178, 1993.

[ZL94] W. Zimmermann and W. Löwe. An approach to machine-independent parallel programming. In *LNCS 854, Parallel Processing: CONPAR'94–VAPP VI*, pages 277–288. Springer, 1994.